Tutorial: Creating an LLVM Backend for the Cpu0 Architecture

Release 12.0.15

Chen Chung-Shu

Mar 16, 2024

CONTENTS

1 About 3

2 Cpu0 architecture and LLVM structure 17

3 Backend structure 79

4 Arithmetic and logic instructions 203

5 Generating object files 245

6 Global variables 279

7 Other data type 313

8 Control flow statements 339

9 Function call 379

10 ELF Support 491

11 Assembler 511

12 C++ support 569

13 Verify backend on Verilog simulator 607

14 The concept of GPU compiler 633

15 Appendix A: Getting Started: Installing LLVM and the Cpu0 example code 675

16 Appendix B: Cpu0 document and test 681

17 Appendix C: The concept of NPU (Neural Processor Unit) compiler 697

18 Todo List 709

19 Resources 711

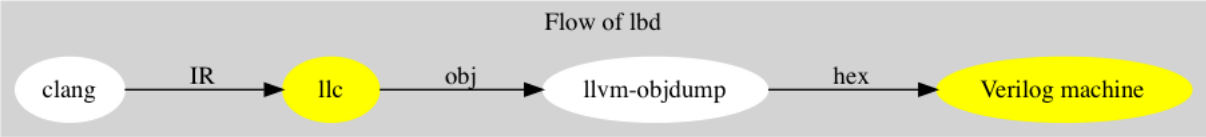


Fig. 1: This book's flow

CHAPTER

ONE

ABOUT

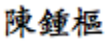
- *Authors*
- *Contributors*
- *Acknowledgments*
- *Support*
- *Build steps*
- *Revision history*
- *Licensing*
- *Motivation*
- *Preface*
- *Prerequisites*
- *Outline of Chapters*

1.1 Authors

Chen Chung-Shu

gamma_chen@yahoo.com.tw

http://jonathan2251.github.io/ws/en/index.html



1.2 Contributors

Anoushe Jamshidi, ajamshidi@gmail.com, Chapters 1, 2, 3 English re-writing and Sphinx tool and format setting.

Chen Wei-Ren, chenwj@iis.sinica.edu.tw, assisted with text and code formatting.

Chen Zhong-Cheng, who is the author of original cpu0 verilog code.

1.3 Acknowledgments

We would like to thank Sean Silva, chisophugis@gmail.com, for his help, encouragement, and assistance with the Sphinx document generator. Without his help, this book would not have been finished and published online. We also thank those corrections from readers who make the book more accurate.

1.4 Support

We get the kind help from LLVM development mail list, llvmdev@cs.uiuc.edu, even we don't know them. So, our experience is you are not alone and can get help from the development list members in working with the LLVM project. Some of them are:

Akira Hatanaka <ahatanak@gmail.com> in va_arg question answer.

Ulrich Weigand <Ulrich.Weigand@de.ibm.com> in AsmParser question answer.

1.5 Build steps

https://github.com/Jonathan2251/lbd/blob/master/README.md

1.6 Revision history

Version 12.0.16, not released yet.

Version 12.0.15, Released March 16, 2024.

> llvmstructure.rst: table: C, llvm-ir and Cpu0. gpu.rst: glsl-spirv.gv, AMD FreeSync, Fragment definition, comment for References/01-triangles.cpp, Refine sections and sub-sections.

Version 12.0.14, Released December 30, 2023.

> gpu.rst: Table More Desciptive Name and Cuda term in GPU and Desciption, Table Mapping saxpy code. gpu.rst: Table More Desciptive Name for Cuda term in Fermi GPU and Desciption. llvmstructure.rst: table: C, llvm-ir and Cpu0. lvmstructure.rst: note What and how the ISA of Cpu0 be selected

Version 12.0.13.9, Released November 30, 2023.

> gpu.rst: Table More Desciptive Name and Cuda term in GPU and Desciption, Table Mapping saxpy code.

Version 12.0.13.8, Released November 30, 2023.

> gpu.rst: graphic_cpu_gpu.png, additive-colors.

Version 12.0.13.7, Released November 16, 2023.

> gpu.rst: 2d-vector-inward, ogl-pointing-outwards and short-rendering-pipeline, Double Buffering and VSync.

Version 12.0.13.6, Released November 13, 2023.

> gpu.rst: VAO.

Version 12.0.13.5, Released September 10, 2023.

> gpu.rst: Tessellation Shading and clearpage for pdf. llvmstructure.rst: section CFG and Register Allocation Passes.

Version 12.0.13.4, Released August 26, 2023.

llvm.rst: Sections of Options of llc for debug and Options of opt.

Version 12.0.13.3, Released August 13, 2023.

gpu.rst: animation, graphic-sw-stack.gv and opengl-flow.gv.

Version 12.0.13.2, Released August 7, 2023.

gpu.rst: Subsection of buffers, vao binding.

Version 12.0.13.1, Released July 24, 2023.

gpu.rst: Section of Basic geometry in computer graphics, a x b = -b x a in 2D, The role of GPU driver. npu.rst: The role of GPU driver.

Version 12.0.13, Released July 15, 2023.

gpu.rst: Section of Basic geometry in computer graphics

Version 12.0.12, Released April 4, 2023.

gpu.rst: refine.

Version 12.0.11, Released Feburary 27, 2023.

README.md. docs.rst: Note of Sphinx. c++.rst: Atomic.

Version 12.0.10, Released December 15, 2022.

gpu.rst: Refine Table 43. ctrl.rst: Refine section "Pipeline architecture". Change test_memcpy.ll. Refine install.rst.

Version 12.0.9, Released November 19, 2022.

gpu.rst: Table 42 Map (Core,Thread) to saxpy and refine section of General purpose GPU. Move null_pointer.cpp from git/note to lbd/References.

Version 12.0.8, Released November 12, 2022.

install.rst: section Toolchain and Brew install in China. Section Work flow of genobj.rst. set-llvm-lit.

Version 12.0.7, Released September 24, 2022.

Atomic, section of Accelerate ML/DL on OpenCL/SYC and refine Makefile and install.rst

Version 12.0.6, Released August 16, 2022.

Fig/backendstructure/class_access_link.puml. Lock-free of chapter c++ and Vulkan link of gpu. Install & doc. Update spirvtoolchain link and grid.png in gpu chapter.

Version 12.0.5, Released February 1, 2022.

Fix regression test.

Version 12.0.4, Released January 22, 2022.

Fix bug: add CMPu, store uses GPROut register to exclude SW registe and Relocation Record: R_CPU0_HI16/fixup_Cpu0_HI16.

Version 12.0.3, Released January 9, 2022.

Expand memory size of cpu0.v to 0x1000000, 24-bit. Section LLVM vs GCC in structure. Add NOR instruction. Fix bug of SLTu SLTiu, SRA and SRAV in verilog code.

Version 12.0.2, Released December 18, 2021.

Remove regression test cases for large frame of not supporting.

Version 12.0.1, Released December 12, 2021.

Section: More about llvm. Table: The differences for speedup in architecture of CPU and GPU. Pipeline diagram and exception handling link. Update chapter Appendix A.

Version 12.0.0, Released August 11, 2021.

Writing and comment.

Version 3.9.4, Released August 5, 2021.

Writing and comment.

Version 3.9.3, Released March 1, 2020.

Add Appendix C: GPU compiler

Version 3.9.2, Released Feburary 17, 2020.

Add section "Add specific backend intrinsic function". Add reasons for regression test. More phi node explanation.

Version 3.9.1, Released May 11, 2018

Fix tailcall bug. Fix return-vector.ll run slowly problem, bug from Cpu0ISelLowering.cpp. Add figure "Tblgen generate files for Cpu0 backend". Modify section float and double of Chapter Other data type. Move storeRegToStack() and loadRegFromStack() from Chapter9_1 to Chapter3_5. Section DSA of chapter Cpu0 architecture and LLVM structure.

Version 3.9.0, Released November 22, 2016

Porting to llvm 3.9. Correct writing.

Version 3.7.4, Released December 7, 2016

Change bal instruction from with delay slot to without delay slot.

Version 3.7.3, Released July 20, 2016

Refine code-block according sphinx lexers. Add search this book.

Version 3.7.2, Released June 29, 2016

Add Verilog delay slot simulation. Explain "tablegen(" in CMakeLists.txt. Correct typing. Add lbdex/install_llvm/*.sh for installation. Upgrade sphinx to 1.4.4.

Version 3.7.1, Released November 7, 2015

Remove EM_CPU0_EL. Add subsection Caller and callee saved registers. Add IR blockaddress and indirectbr support. Correct tglobaladdr, tblockaddress, tjumptable and tglobaltlsaddr of Cpu0InstrInfo.td. Add stacksave and stackrestore support. Add sub-section frameaddress, returnaddress and eh.return support of chapter Function call. Match Mips 3.7 style. Add bswap in Chapter Function call. Add section "Vector type (SIMD) support" of Chapter "Other data type". Add section "Long branch support" of Chapter "Control flow statements". Add sub-section "eh.dwarf intrinsic" of Chapter Function call. Change display "ret $rx" to "jr $rx" where $rx is not $lr. Move sub-section Caller and callee saved registers. Add sub-sections Live in and live out register. Add Phi node. Replace ch3-proepilog.ll with ch3_largeframe.cpp. Remove DecodeCMPInstruction(). Re-organize testing ch4_2_1.cpp, ch4_2_2.cpp and ch9_4.cpp. Fix dynamic alloca bug. Move Cpu0AnalyzeImmediate.cpp and related functions from Chapter3_4 to Chapter3_5. Rename input files.

Version 3.7.0, Released September 24, 2015

Porting to lld 3.7. Change tricore_llvm.pdf web link. Add C++ atomic to regression test.

Version 3.6.4, Released July 15, 2015

Add C++ atomic support.

Version 3.6.3, Released May 25, 2015

Correct typing.

Version 3.6.2, Released May 3, 2015

Write Appendix B. Split chapter Appendix B from Appendix A. Move some test from lbt to lbd. Remove warning in build Cpu0 code.

Version 3.6.1, Released March 22, 2015

Add Cpu0 instructions ROLV and RORV.

Version 3.6.0, Released March 9, 2015

Update Appendix A for llvm 3.6. Replace cpp with ll for appearing in document. Move chapter lld, optimization, library to https://github.com/Jonathan2251/lbt.git.

Version 3.5.9, Released February 2, 2015

Fix bug of 64 bits shift. Fix global address error by replacing addiu with ori. Change encode of "cmp $sw, $3, $2" from 0x10320000 to 0x10f32000.

Version 3.5.8, Released December 27, 2014

Correct typing. Fix typing error for update lbdex/src/modify/src/ of install.rst. Add libsoftfloat/compiler-rt and libc/avr-libc-1.8.1. Add LLVM-VPO in chapter Optimization.

Version 3.5.7, Released December 1, 2014

Fix over 16-bits frame prologue/epilogue error from 3.5.3. Call convention ABI S32 is enabled by option. Change from ADD to ADDu in copyPhysReg() of Cpu0SEInstrInfo.cpp. Add asm directive .weak back which exists in 3.5.3.

Version 3.5.6, Released November 18, 2014

Remove SWI and IRET instructions. Add Cpu0SetChapter.h for ex-build-test.sh. Correct typing. Fix thread variable error come from version 3.5.3 in static mode. Add sub-section "Cpu0 backend machine ID and relocation records" of Chapter 2.

Version 3.5.5, Released November 11, 2014

Rename SPR to C0R. Add ISR simulation.

Version 3.5.4, Released November 6, 2014

Adjust chapter 9 sections. Fix .cprestore bug. Re-organize sections. Add sub-section "Why not using ADD instead of SUB?" in chapter 2. Add overflow control option to use ADD and SUB instructions.

Version 3.5.3, Released October 29, 2014

Merge Cpu0 example code into one copy and it can be config by Cpu0Config.h.

Version 3.5.2, Released October 3, 2014

Move R_CPU0_32 from type of non-relocation record to type ofrelocation record. Correct logic error for setgt of BrcondPatsSlt of Cpu0InstrInfo.td.

Version 3.5.1, Released October 1, 2014

Add move alias instruction for addu $reg, $zero. Add cpu cycles count in verilog. Fix ISD::SIGN_EXTEND_INREG error in other types beside i1. Support DAG op br_jt and DAG node JumpTable.

Version 3.5.0, Released September 05, 2014

Issue NOP in delay slot.

Version 3.4.8, Released August 29, 2014

Add reason that set endian swap in memory module. Add presentation files.

Version 3.4.7, Released August 22, 2014

Fix wrapper_pic for cmov.ll. Add shift operations 64 bits support. Fix wrapper_pic for ch8_5.cpp. Add section thread of chapter 14. Add section Motivation of chapter about. Support little endian for cpu0 verilog. Move ch8_5.cpp test from Chapter Run backend to Chapter lld since it need lld linker. Support both big endian and little endian in cpu0 Verilog, elf2hex and lld. Make branch release_34_7.

Version 3.4.6, Released July 26, 2014

Add Chapter 15, optimization. Correct typing. Add Chapter 14, C++. Fix bug of generating cpu032II instruction in dynamic_linker.cpp.

Version 3.4.5, Released June 30, 2014

Correct typing.

Version 3.4.4, Released June 24, 2014

Correct typing. Add the reason of use SSA form. Move sections LLVM Code Generation Sequence, DAG and Instruction Selection from Chapter 3 to Chapter 2.

Version 3.4.3, Released March 31, 2014

Fix Disassembly bug for GPROut register class. Adjust Chapters. Remove hand copy Table of tblgen in AsmParser.

Version 3.4.2, Released February 9, 2014

Add ch12_2.cpp for slt instruction explanation and fix bug in Cpu0InstrInfo.cpp. Correct typing. Move Cpu0 Status Register from Number 20 to Number 10. Fix llc -mcpu option problem. Update example code build shell script. Add condition move instruction. Fix bug of branch pattern match in Cpu0InstrInfo.td.

Version 3.4.1, Released January 18, 2014

Add ch9_4.cpp to lld test. Fix the wrong reference in lbd/lib/Target/Cpu0 code. inlineasm. First instruction jmp X, where X changed from _Z5startv to start. Correct typing.

Version 3.4.0, Released January 9, 2014

Porting to llvm 3.4 release.

Version 3.3.14, Released January 4, 2014

lld support on iMac. Correct typing.

Version 3.3.13, Released December 27, 2013

Update section Install sphinx on install.rst. Add Fig/llvmstructure/cpu0_arch.odp.

Version 3.3.12, Released December 25, 2013

Correct typing error. Adjust Example Code. Add section Data operands DAGs of backendstructure.rst. Fix bug in instructions lb and lh of cpu0.v. Fix bug in itoa.cpp. Add ch7_2_2.cpp for othertype.rst. Add AsmParser reference web.

Version 3.3.11, Released December 11, 2013

Add Figure Code generation and execution flow in about.rst. Update backendstructure.rst. Correct otherinst.rst. Decoration. Correct typing error.

Version 3.3.10, Released December 5, 2013

Correct typing error. Dynamic linker in lld.rst. Correct errors came from old version of example code. lld.rst.

Version 3.3.9, Released November 22, 2013

Add LLD introduction and Cpu0 static linker document in lld.rst. Fix the plt bug in elf2hex.h for dynamic linker.

Version 3.3.8, Released November 19, 2013

Fix the reference file missing for make gh-page.

Version 3.3.7, Released November 17, 2013

lld.rst documentation. Add cpu032I and cpu032II in *llc -mcpu*. Reference only for Chapter12_2.

Version 3.3.6, Released November 8, 2013

Move example code from github to dropbox since the name is not work for download example code.

Version 3.3.5, Released November 7, 2013

Split the elf2hex code from modiified llvm-objdump.cpp to elf2hex.h. Fix bug for tail call setting in LowerCall(). Fix bug for LowerCPLOAD(). Update elf.rst. Fix typing error. Add dynamic linker support. Merge cpu0 Chapter12_1 and Chapter12_2 code into one, and identify each of them by -mcpu=cpu0I and -mcpu=cpu0II. cpu0II. Update lld.rst for static linker. Change the name of example code from LLVMBackendTutorialExampleCode to lbdex.

Version 3.3.4, Released September 21, 2013

Fix Chapter Global variables error for LUi instructions and the material move to Chapter Other data type. Update regression test items.

Version 3.3.3, Released September 20, 2013

Add Chapter othertype

Version 3.3.2, Released September 17, 2013

Update example code. Fix bug sext_inreg. Fix llvm-objdump.cpp bug to support global variable of .data. Update install.rst to run on llvm 3.3.

Version 3.3.1, Released September 14, 2013

Add load bool type in chapter 6. Fix chapter 4 error. Add interrupt function in cpu0i.v. Fix bug in alloc() support of Chapter 8 by adding code of spill $fp register. Add JSUB texternalsym for memcpy function call of llvm auto reference. Rename cpu0i.v to cpu0s.v. Modify itoa.cpp. Cpu0 of lld.

Version 3.3.0, Released July 13, 2013

Add Table: C operator ! corresponding IR of .bc and IR of DAG and Table: C operator ! corresponding IR of Type-legalized selection DAG and Cpu0 instructions. Add explanation in section Full support %. Add Table: Chapter 4 operators. Add Table: Chapter 3 .bc IR instructions. Rewrite Chapter 5 Global variables. Rewrite section Handle $gp register in PIC addressing mode. Add Large Frame Stack Pointer support. Add dynamic link section in elf.rst. Re-oganize Chapter 3. Re-oganize Chapter 8. Re-oganize Chapter 10. Re-oganize Chapter 11. Re-oganize Chapter 12. Fix bug that ret not $lr register. Porting to LLVM 3.3.

Version 3.2.15, Released June 12, 2013

Porting to llvm 3.3. Rewrite section Support arithmetic instructions of chapter Adding arithmetic and local pointer support with the table adding. Add two sentences in Preface. Add *llc -debug-pass* in section LLVM Code Generation Sequence. Remove section Adjust cpu0 instructions. Remove section Use cpu0 official LDI instead of ADDiu of Appendix-C.

Version 3.2.14, Released May 24, 2013

Fix example code disappeared error.

Version 3.2.13, Released May 23, 2013

Add sub-section "Setup llvm-lit on iMac" of Appendix A. Replace some code-block with literalinclude in *.rst. Add Fig 9 of chapter Backend structure. Add section Dynamic stack allocation support of chapter Function call. Fix bug of Cpu0DelUselessJMP.cpp. Fix cpu0 instruction table errors.

Version 3.2.12, Released March 9, 2013

Add section "Type of char and short int" of chapter "Global variables, structs and arrays, other type".

Version 3.2.11, Released March 8, 2013

Fix bug in generate elf of chapter "Backend Optimization".

Version 3.2.10, Released February 23, 2013

Add chapter "Backend Optimization".

Version 3.2.9, Released February 20, 2013

Correct the "Variable number of arguments" such as sum_i(int amount, ...) errors.

Version 3.2.8, Released February 20, 2013

Add section llvm-objdump -t -r.

Version 3.2.7, Released February 14, 2013

Add chapter Run backend. Add Icarus Verilog tool installation in Appendix A.

Version 3.2.6, Released February 4, 2013

Update CMP instruction implementation. Add llvm-objdump section.

Version 3.2.5, Released January 27, 2013

Add "LLVMBackendTutorialExampleCode/llvm3.1". Add section "Structure type support". Change reference from Figure title to Figure number.

Version 3.2.4, Released January 17, 2013

Update for LLVM 3.2. Change title (book name) from "Write An LLVM Backend Tutorial For Cpu0" to "Tutorial: Creating an LLVM Backend for the Cpu0 Architecture".

Version 3.2.3, Released January 12, 2013

Add chapter "Porting to LLVM 3.2".

Version 3.2.2, Released January 10, 2013

Add section "Full support %" and section "Verify DIV for operator %".

Version 3.2.1, Released January 7, 2013

Add Footnote for references. Reorganize chapters (Move bottom part of chapter "Global variable" to chapter "Other instruction"; Move section "Translate into obj file" to new chapter "Generate obj file". Fix errors in Fig/otherinst/2.png and Fig/otherinst/3.png.

Version 3.2.0, Released January 1, 2013

Add chapter Function. Move Chapter "Installing LLVM and the Cpu0 example code" from beginning to Appendix A. Add subsection "Install other tools on Linux". Add chapter ELF.

Version 3.1.2, Released December 15, 2012

Fix section 6.1 error by add "def : Pat<(brcond RC:$cond, bb:$dst), (JNEOp (CMPOp RC:$cond, ZEROReg), bb:$dst)>;" in last pattern. Modify section 5.5 Fix bug Cpu0InstrInfo.cpp SW to ST. Correct LW to LD; LB to LDB; SB to STB.

Version 3.1.1, Released November 28, 2012

Add Revision history. Correct ldi instruction error (replace ldi instruction with addiu from the beginning and in the all example code). Move ldi instruction change from section of "Adjust cpu0 instruction and support type of local variable pointer" to Section "CPU0 processor architecture". Correct some English & typing errors.

1.7 Licensing

http://llvm.org/docs/DeveloperPolicy.html#license

1.8 Motivation

My intention for writing this book is that I am curious about what a simple and robotic CPU ISA and SW toolchains of llvm based can be.

Table 1.1: Number of lines around in source code (include space-line and comments) for Cpu0

Components	Number of lines
llvm	15,000
llvm-objdump	8
elf2hex	765
verilog	600
lld	140
clang	500
compiler-rt's builtin	5 (abort.c)
total	17,018

- Though llvm backend's source code can be ported from other backend, it still includes a lot of thinking for doing it and not quite easily.

We all learned computer knowledge from school through the concept of book. The concept is an effective way to know the big view. But once getting into develop a real complicated system, we often feel the concept from school or book is not much or not details enough. Compiler is a very complicated system, so traditionally the students in school learn this knowledge in concept and do the home work via yacc/lex tools to translate part of C or other high level language into immediate representation (IR) or assembly to feel the parsing knowledge and tools application.

On the other hand, the compiler engineers who graduated from school often facing the real market complicated CPUs and specification. Since for market reason, there are a serial of CPUs and ABI (Application Binary Interface) to deal with. Moreover, for speed performance reason, the real compiler backend program is too complicated to be a learning material in compiler backend designing even the market CPU include only one CPU and ABI.

This book develop the compiler backend along with a simple school designed CPU which called Cpu0. It include the implementation of a compiler backend, linker, llvm-objdump, elf2hex as well as Verilog language source code of Cpu0 instruction set. We provide readers full source code to compile C/C++ program and see how the programs run on the Cpu0 machine created by verilog language. Through this school learning purpose CPU, you get the chance to know the whole thing in compiler backend, linker, system tools and CPU design. Usually it is not easy from working in real CPU and compiler since the real job is too complicated to be finished by one single person only.

As my observation, LLVM advocated by some software engineers against gcc with two reasons. One is political with BSD license[1][2]. The other is technical with following the 3 tiers of compiler software structure along with C++ object oriented technology. GCC started with C and adopted C++ after near 20 years later[3]. Maybe gcc adopted C++ just because llvm do that. I learned C++ object oriented programming during studing in school. After "Design Pattern", "C++/STL" and "object oriented design" books study, I understand the C is easy to trace while C++ is easy to creating reusable software units known as object. If a programmer has well knowledge in "Design Pattern", then the C++ can supply more reuse ability and rewrite ability. A book of "system language" about software quality that I have ever read , listing these items: read ability, rewrite ability, reuse ability and performance to define the software quality. Object oriented programming exists for solving the big and complex software development. Of course, compiler and OS are complex software without question, why do gcc and linux not using c++[4]? This is the reason I try to create a backend under llvm rather than gcc.

1.9 Preface

The LLVM Compiler Infrastructure provides a versatile structure for creating new backends. Creating a new backend should not be too difficult once you familiarize yourself with this structure. However, the available backend documentation is fairly high level and leaves out many details. This tutorial will provide step-by-step instructions to write a new backend for a new target architecture from scratch.

We will use the Cpu0 architecture as an example to build our new backend. Cpu0 is a simple RISC architecture that has been designed for educational purposes. More information about Cpu0, including its instruction set, is available here[5]. The Cpu0 example code referenced in this book can be found http://jonathan2251.github.io/lbd/lbdex.tar.gz. As you progress from one chapter to the next, you will incrementally build the backend's functionality.

Since Cpu0 is a simple RISC CPU for educational purpose, it makes this llvm backend code simple too and easy to learning. In addition, Cpu0 supply the Verilog source code that you can run on your PC or FPGA platform when you go to chapter "Verify backend on Verilog simulator". To explain the backend design, we carefully design C/C++ program for each chapter new added function. Through these example code, readers can understand what IRs (llvm immediate form) the backend transfer from and the C/C++ code corresponding to these IRs.

This tutorial started using the LLVM 3.1 Mips backend as a reference and sync to llvm 3.5 Mips at version 3.5.3. As our experience, reference and sync with a released backend code will help upgrading your backend features and fixing bugs. You can take advantage by compare difference from version to version, and hire llvm development team effort. Since Cpu0 is an educational architecture, and it has missed some key pieces of documentation needed when developing a compiler, such as an Application Binary Interface (ABI). We implement our backend by borrowing information from the Mips ABI as a guide. You may want to familiarize yourself with the relevant parts of the Mips ABI as you progress through this tutorial.

This document can be a tutorial of toolchain development for a new CPU architecture. Many programmer gradutated from school with the knowledges of Compiler as well as Computer architecture but is not an professional engineer in compiler or CPU design. This document is a material to introduce these engineers how to programming a toolchain as well as designing a CPU based on the LLVM infrastructure without pay any money to buy software or hardware. Computer is the only device needed.

Finally, this book is not a compiler book in concept. It is for those readers who are interested in extending compiler toolchain to support a new CPU based on llvm structure. To program on Linux OS, you program a driver without knowing every details in OS. For example in a specific USB device driver program on Linux plateform, he or she will try to understand the USB specification, linux USB subsystem and common device driver working model and API. In the same way, to extend functions from a large software like this llvm umbrella project, you should find a way to reach the goal and ignore the details not on your way. Try to understand in details of every line of source code is not realistic

[1] http://llvm.org/docs/DeveloperPolicy.html#license
[2] http://www.phoronix.com/scan.php?page=news_item&px=MTU4MjA
[3] http://en.wikipedia.org/wiki/GNU_Compiler_Collection
[4] http://en.wikipedia.org/wiki/C%2B%2B
[5] http://ccckmit.wikidot.com/ocs:cpu0

if your project is an extended function from a well defined software structure. It only makes sense in rewriting the whole software structure. Of course, if there are more llvm backend book or documents, then readers have the chance to know more about llvm by reading book or documents.

1.10 Prerequisites

Readers should be comfortable with the C++ language and Object-Oriented Programming concepts. LLVM has been developed and implemented in C++, and it is written in a modular way so that various classes can be adapted and reused as often as possible.

Already having conceptual knowledge of how compilers work is a plus, and if you already have implemented compilers in the past you will likely have no trouble following this tutorial. As this tutorial will build up an LLVM backend step-by-step, we will introduce important concepts as necessary.

This tutorial references the following materials. We highly recommend you read these documents to get a deeper understanding of what the tutorial is teaching:

The Architecture of Open Source Applications Chapter on LLVM

LLVM's Target-Independent Code Generation documentation

LLVM's TableGen Fundamentals documentation

LLVM's Writing an LLVM Compiler Backend documentation

Description of the Tricore LLVM Backend

Mips ABI document

1.11 Outline of Chapters

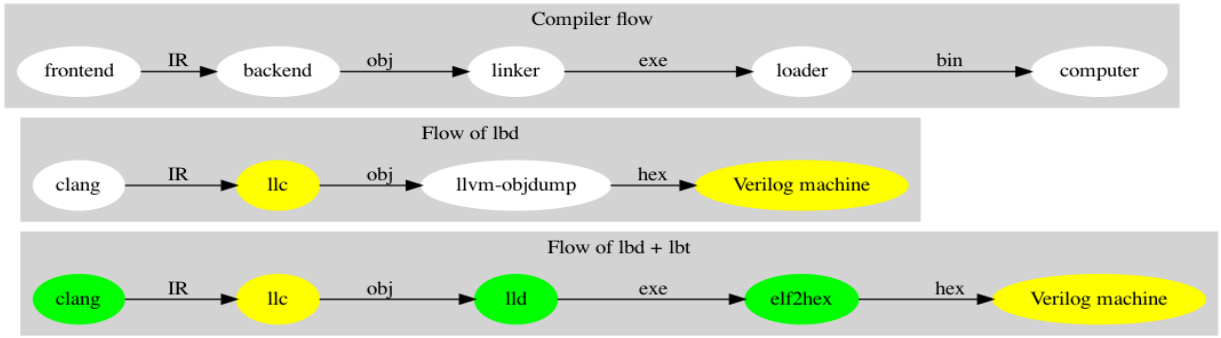


Fig. 1.1: Code generation and execution flow

The top part of Fig. 1.1 is the work flow and software package of a computer program be generated and executed. IR stands for Intermediate Representation. The middle part is this book's work flow. Except clang, the other blocks need to be extended for a new backend development (Cpu0 backend extending clang too, however Cpu0 backend uses Mips ABI and can use Mips-clang). This book implement the yellow boxes part. The green parts of this figure, lld and elf2hex for Cpu0 backend, can be found on http://jonathan2251.github.io/lbt/index.html. The hex is the ascii file format using '0' to '9' and 'a' to 'f' for hexadecimal value representation since the Verilog language machine uses it as input file.

This book include 10,000 lines of source code for

1. Step-by-step, creating an llvm backend for the Cpu0. Chapter 2 to 11.
2. Cpu0 verilog source code. Chapter 12.

With these code, reader can generate Cpu0 machine code through Cpu0 llvm backend compiler, then see how it runs on your computer if the code without global variable or relocation record for handling by linker. The pdf and epub are also available in the web. This is a tutorial for llvm backend developer but not for an expert. It also can be a material for those who have compiler and computer architecture book's knowledges and like to know how to extend the llvm toolchain to support a new CPU.

Cpu0 architecture and LLVM structure:

This chapter introduces the Cpu0 architecture, a high-level view of LLVM, and how Cpu0 will be targeted in in an LLVM backend. This chapter will run you through the initial steps of building the backend, including initial work on the target description (td), setting up cmake file, and target registration. Around 750 lines of source code are added by the end of this chapter.

Backend structure:

This chapter highlights the structure of an LLVM backend using by UML graphs, and we continue to build the Cpu0 backend. Thousands of lines of source code are added, most of which are common from one LLVM backends to another, regardless of the target architecture. By the end of this chapter, the Cpu0 LLVM backend will support less than ten instructions to generate some initial assembly output.

Arithmetic and logic instructions:

Over ten C operators and their corresponding LLVM IR instructions are introduced in this chapter. Few houndred lines of source code, mostly in .td Target Description files, are added. With these houndred lines of source code, the backend can now translate the **+, -, *, /, &, |, ^, <<, >>, !** and **%** C operators into the appropriate Cpu0 assembly code. Usage of the `llc` debug option and of **Graphviz** as a debug tool are introduced in this chapter.

Generating object files:

Object file generation support for the Cpu0 backend is added in this chapter, as the Target Registration structure is introduced. Based on llvm structure, the Cpu0 backend can generate big and little endian ELF object files without much effort.

Global variables:

Global variable handling is added in this chapter. Cpu0 supports PIC and static addressing mode, both addressing mode explained as their functionality are implemented.

Other data type:

In addition to type int, other data type such as pointer, char, bool, long long, structure and array are added in this chapter.

Control flow statements:

Support for flow control statements, such as, **if, else, while, for, goto, switch, case** as well as both a simple optimization software pass and hardware instructions for control statement optimization discussed in this chapter.

Function call:

This chapter details the implementation of function calls in the Cpu0 backend. The stack frame, handling incoming & outgoing arguments, and their corresponding standard LLVM functions are introduced.

ELF Support:

This chapter details Cpu0 support for the well-known ELF object file format. The ELF format and binutils tools are not a part of LLVM, but are introduced. This chapter details how to use the ELF tools to verify and analyze the object files created by the Cpu0 backend. The disassemble command `llvm-objdump -d` support for Cpu0 is added in the last section of this chapter.

Assembler:

Support the translation of hand code assembly language into obj under the llvm insfrastructure.

C++ support:

Support C++ language features. It's under working.

Verify backend on Verilog simulator:

Create the CPU0 virtual machine with Verilog language of Icarus tool first. With this tool, feeding the hex file which generated by llvm-objdump to the Cpu0 virtual machine and seeing the Cpu0 running result on PC computer.

Appendix A: Getting Started: Installing LLVM and the Cpu0 example code:

Details how to set up the LLVM source code, development tools, and environment setting for Mac OS X and Linux platforms.

Appendix B: Cpu0 document and test:

This book uses Sphinx to generate pdf and epub format of document further. Details about how to install tools to and generate these docuemnts and regression test for Cpu0 backend are included.

CHAPTER

TWO

CPU0 ARCHITECTURE AND LLVM STRUCTURE

- *Cpu0 Processor Architecture Details*
  - *Brief introduction*
  - *The Cpu0 Instruction Set*
    - *Why not using ADD instead of SUB?*
  - *The Status Register*
  - *Cpu0's Stages of Instruction Execution*
  - *Cpu0's Interrupt Vector*
- *LLVM Structure*
  - *Three-phase design*
  - *LLVM's Target Description Files: .td*
  - *LLVM Code Generation Sequence*
  - *SSA form*
  - *DSA form*
  - *LLVM vs GCC in structure*
  - *LLVM blog*
  - *CFG (Control Flow Graph)*
  - *DAG (Directed Acyclic Graph)*
  - *Instruction Selection*
  - *Caller and callee saved registers*
  - *Live in and live out register*
- *Create Cpu0 backend*
  - *Cpu0 backend machine ID and relocation records*
  - *Creating the Initial Cpu0 .td Files*
  - *Write cmake file*
  - *Target Registration*
  - *Build libraries and td*

- *Options of llc for debug*
- *Options of opt*

Before you begin this tutorial, you should know that you can always try to develop your own backend by porting code from existing backends. The majority of the code you will want to investigate can be found in the /lib/Target directory of your root LLVM installation. As most major RISC instruction sets have some similarities, this may be the avenue you might try if you are an experienced programmer and knowledgable of compiler backends.

On the other hand, there is a steep learning curve and you may easily get stuck debugging your new backend. You can easily spend a lot of time tracing which methods are callbacks of some function, or which are calling some overridden method deep in the LLVM codebase - and with a codebase as large as LLVM, all of this can easily become difficult to keep track of. This tutorial will help you work through this process while learning the fundamentals of LLVM backend design. It will show you what is necessary to get your first backend functional and complete, and it should help you understand how to debug your backend when it produces incorrect machine code using output provided by the compiler.

This chapter details the Cpu0 instruction set and the structure of LLVM. The LLVM structure information is adapted from Chris Lattner's LLVM chapter of the Architecture of Open Source Applications book[10]. You can read the original article from the AOSA website if you prefer.

At the end of this Chapter, you will begin to create a new LLVM backend by writing register and instruction definitions in the Target Description files which will be used in next chapter.

Finally, there are compiler knowledge like DAG (Directed-Acyclic-Graph) and instruction selection needed in llvm backend design, and they are explained here.

2.1 Cpu0 Processor Architecture Details

This section is based on materials available here[1] (Chinese) and here[2] (English). However, I changed some ISA from original Cpu0 for designing a simple integer operational CPU and llvm backend. This is my intention for writing this book that I want to know what a simple and robotic CPU ISA and llvm backend can be.

2.1.1 Brief introduction

Cpu0 is a 32-bit architecture. It has 16 general purpose registers (R0, . . . , R15), co-processor registers (like Mips), and other special registers. Its structure is illustrated in Fig. 2.1 below.

The registers are used for the following purposes:

Table 2.1: Cpu0 general purpose registers (GPR)

Register	Description
R0	Constant register, value is 0
R1-R10	General-purpose registers
R11	Global Pointer register (GP)
R12	Frame Pointer register (FP)
R13	Stack Pointer register (SP)
R14	Link Register (LR)
R15	Status Word Register (SW)

[10] Chris Lattner, **LLVM**. Published in The Architecture of Open Source Applications. http://www.aosabook.org/en/llvm.html

[1] Original Cpu0 architecture and ISA details (Chinese). http://ccckmit.wikidot.com/ocs:cpu0

[2] English translation of Cpu0 description. http://translate.google.com.tw/translate?js=n&prev=_t&hl=zh-TW&ie=UTF-8&layout=2&eotf=1&sl=zh-CN&tl=en&u=http://ccckmit.wikidot.com/ocs:cpu0

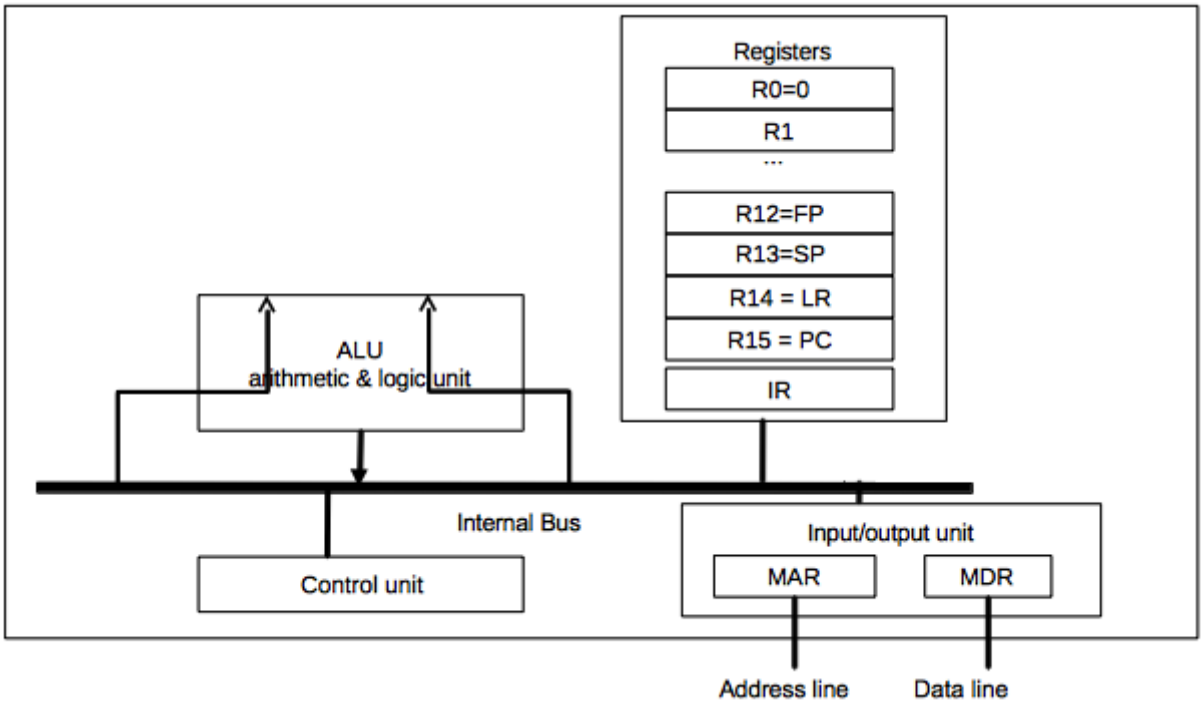


Fig. 2.1: Architectural block diagram of the Cpu0 processor

Table 2.2: Cpu0 co-processor 0 registers (C0R)

Register	Description
0	Program Counter (PC)
1	Error Program Counter (EPC)

Table 2.3: Cpu0 other registers

Register	Description
IR	Instruction register
MAR	Memory Address Register (MAR)
MDR	Memory Data Register (MDR)
HI	High part of MULT result
LO	Low part of MULT result

2.1.2 The Cpu0 Instruction Set

The Cpu0 instruction set can be divided into three types: L-type instructions, which are generally associated with memory operations, A-type instructions for arithmetic operations, and J-type instructions that are typically used when altering control flow (i.e. jumps). Fig. 2.2 illustrates how the bitfields are broken down for each type of instruction.

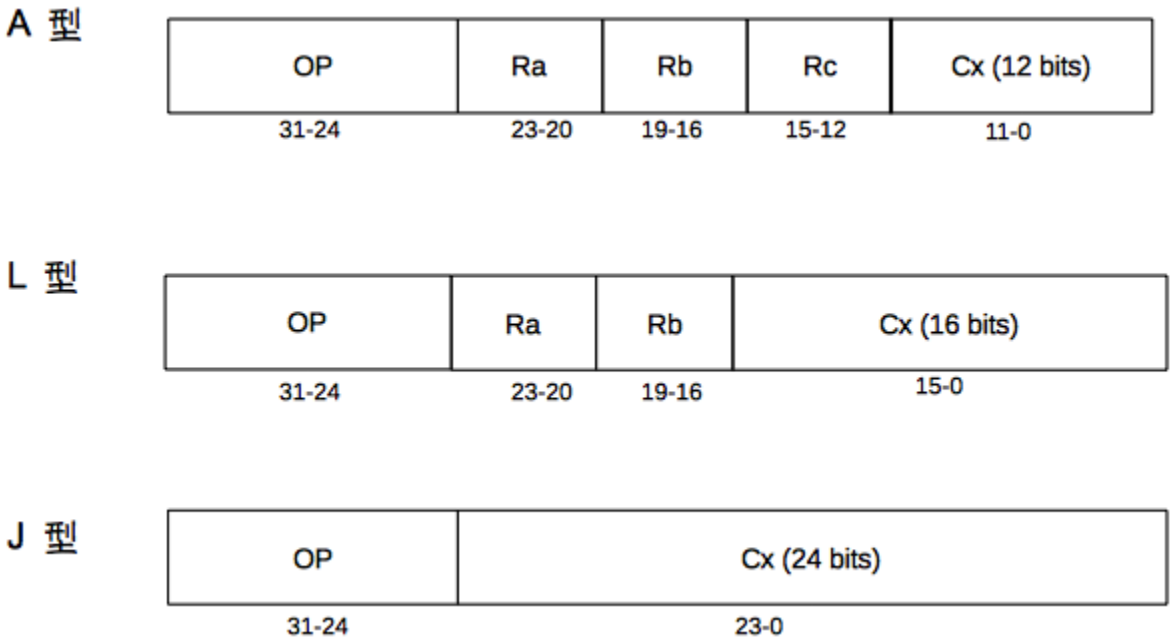


Fig. 2.2: Cpu0's three instruction formats

Table 2.4: C, llvm-ir[Page 23, 13] and Cpu0

C	llvm-ir	Cpu0	I or II	Comment
=	load/store	ld/lb/lbu/lh/lhu	I	
&, &&	and	and	I	
|, ||	or	or	I	
^	xor	xor/nor	I	! can be got from two ir
!	• %tobool = icmp ne i32 %6, 0 • %lnot = xor i1 %tobool, true	• cmp • xor		
==, !=, <, <=, >, >=	icmp/fcmp <cond> cond:eq/ne,...	cmp/ucmp ... + floating-lib	I	
"	"	slt/sltu/slti/sltiu	II	slti/sltiu: ex. a == 3 reduce instructions
if (a <= b)	icmp/fcmp <cond> + br i1 <cond>, ...	cmp/uccmp + jeq/jne/jlt/jgt/jle/jge	I	Conditional branch
if (bool)	br i1 <cond>, ...	jeq/jne	I	
"	"	beq/bne	II	
goto	br <dest>	jmp	I	Uncondictional branch
call sub-function	call	jsub	I	Provide 24-bit address range of calling sub-function (the address from caller to callee is within 24-bit)
"	"	jalr	I	Add for 32-bit address range of calling sub-function
return	ret	ret	I	
+, -, *	add/fadd, sub/fsub, mul/fmul	add/addu/addiu, sub/subu, mul	I	
/, %	udiv/sdiv/fdiv, urem/srem/frem	div, mfhi/mflo/mthi/mtlo	I	
<<, >>	shl, lshr/ashr	shl/rol/rolv, srl/sra/ror/rorv	II	
float <-> int	fptoui, sitofp, ...			Cpu0 uses SW for floating value, and these two IR are for HW floating instruction
__builtin_clz/clo	llvm.clz/llvm_clo	floating-lib + clz, clo	I	For SW floating-lib, uses __builtin_clz / __builtin_clo in clang and clang generates llvm.clz/llvm.clo intrinsic function
__builtin_eh_xxx	llvm.eh.xxx	st/ld	I	pass information to exception handler through $4, $5

Table 2.5: C++, llvm-ir[Page 23, 13] and Cpu0

C++	llvm-ir	Cpu0	I or II	Comment
try { }	invoke void @_Z15throw_exception	jsub _Z15throw_exception	I	
catch { }	landingpad...catch	st and ld	I	st/ld $4 & $5 to/from stack, $4:exception address, $5: exception typeid

Note: What and how the llvm-ir and the ISA of a RISC CPU be selected

- The llvm-ir and the ISA of a RISC CPU emerged after C language. As table above, they can be selected based on C language.
- Not listed in above table, LLVM-IR includes terminator instructions "switch, invoke, ...", atomic and a lot of llvm-intrinsics to provide better performance to backend for their specific instructions such as llvm.vector.reduce.*.
- For vector processing of CPU/GPU, they can use vector-type of math llvm-ir or llvm-intrinsic for implementation.

Note: What and how the ISA of Cpu0 be selected

- The intention of orignal author of Cpu0: Design the ISA for teaching materials without considering performance.
- My intention of goals: Adding a goal that what ISA is good to be selected or designed considering for both to be an llvm simple tutorial material and basic performance to be an ISA. I am not interested in a bad ISA.
  - As you can see from table above, "if (a <= b)" can be replaced with "t = (a <= b)" and "if (t)", so I designed the ISA II of Cpu0 "slt+beq" to replace "cmp+jeq" to reduce jeq/jne/jlt/jgt/jle/jge six intructions to two, beq/bne for the balance of the complexity in Cpu0 ISA and performance.
  - For the same reason, I hired **slt**,... from **Mips** instead of **cmp** from **ARM** as result that destination register can be in any GPR for avoiding the bottle neck on the same "status register".
  - Floating value can be implemented by software, so Cpu0 has integer instructions only. I add clz and clo to Cpu0 since the floating-lib such as compiler-rt/builtin is implemented on top of these two builtin-function. Normalization for Floating precsion can use clz and clo to speedup. Though Cpu0 can use a couple of instructions for handling the corresponding llvm.clz/llvm.clo, adding clz/clo can execute it in one single instruction.
  - I extend II of Cpu0 as reasons above for an better ISA in performace from Mips.

The following table details the cpu032I instruction set:

- First column F.: meaning Format.

Table 2.6: cpu032I Instruction Set

F.	Mnemonic	Op-code	Meaning	Syntax	Operation
L	NOP	00	No Operation		
L	LD	01	Load word	LD Ra, [Rb+Cx]	Ra <= [Rb+Cx]
L	ST	02	Store word	ST Ra, [Rb+Cx]	[Rb+Cx] <= Ra

continues on next page

[13] http://llvm.org/docs/LangRef.html

Table 2.6 – continued from previous page

F.	Mnemonic	Op-code	Meaning	Syntax	Operation
L	LB	03	Load byte	LB Ra, [Rb+Cx]	Ra <= (byte)[Rb+Cx][3]
L	LBu	04	Load byte unsigned	LBu Ra, [Rb+Cx]	Ra <= (byte)[Rb+Cx][Page 25, 3]
L	SB	05	Store byte	SB Ra, [Rb+Cx]	[Rb+Cx] <= (byte)Ra
L	LH	06	Load half word	LH Ra, [Rb+Cx]	Ra <= (2bytes)[Rb+Cx][Page 25, 3]
L	LHu	07	Load half word unsigned	LHu Ra, [Rb+Cx]	Ra <= (2bytes)[Rb+Cx][Page 25, 3]
L	SH	08	Store half word	SH Ra, [Rb+Cx]	[Rb+Cx] <= Ra
L	ADDiu	09	Add immediate	ADDiu Ra, Rb, Cx	Ra <= (Rb + Cx)
L	ANDi	0C	AND imm	ANDi Ra, Rb, Cx	Ra <= (Rb & Cx)
L	ORi	0D	OR	ORi Ra, Rb, Cx	Ra <= (Rb | Cx)
L	XORi	0E	XOR	XORi Ra, Rb, Cx	Ra <= (Rb ^ Cx)
L	LUi	0F	Load upper	LUi Ra, Cx	Ra <= (Cx << 16)
A	ADDu	11	Add unsigned	ADD Ra, Rb, Rc	Ra <= Rb + Rc[4]
A	SUBu	12	Sub unsigned	SUB Ra, Rb, Rc	Ra <= Rb - Rc[Page 25, 4]
A	ADD	13	Add	ADD Ra, Rb, Rc	Ra <= Rb + Rc[Page 25, 4]
A	SUB	14	Subtract	SUB Ra, Rb, Rc	Ra <= Rb - Rc[Page 25, 4]
A	CLZ	15	Count Leading Zero	CLZ Ra, Rb	Ra <= bits of leading zero on Rb
A	CLO	16	Count Leading One	CLO Ra, Rb	Ra <= bits of leading one on Rb
A	MUL	17	Multiply	MUL Ra, Rb, Rc	Ra <= Rb * Rc
A	AND	18	Bitwise and	AND Ra, Rb, Rc	Ra <= Rb & Rc
A	OR	19	Bitwise or	OR Ra, Rb, Rc	Ra <= Rb | Rc
A	XOR	1A	Bitwise exclusive or	XOR Ra, Rb, Rc	Ra <= Rb ^ Rc
A	NOR	1B	Bitwise boolean nor	NOR Ra, Rb, Rc	Ra <= Rb nor Rc
A	ROL	1C	Rotate left	ROL Ra, Rb, Cx	Ra <= Rb rol Cx
A	ROR	1D	Rotate right	ROR Ra, Rb, Cx	Ra <= Rb ror Cx
A	SHL	1E	Shift left	SHL Ra, Rb, Cx	Ra <= Rb << Cx
A	SHR	1F	Shift right	SHR Ra, Rb, Cx	Ra <= Rb >> Cx
A	SRA	20	Shift right	SRA Ra, Rb, Cx	Ra <= Rb ‘>> Cx[6]
A	SRAV	21	Shift right	SRAV Ra, Rb, Rc	Ra <= Rb ‘>> Rc[Page 25, 6]
A	SHLV	22	Shift left	SHLV Ra, Rb, Rc	Ra <= Rb << Rc
A	SHRV	23	Shift right	SHRV Ra, Rb, Rc	Ra <= Rb >> Rc
A	ROL	24	Rotate left	ROL Ra, Rb, Rc	Ra <= Rb rol Rc
A	ROR	25	Rotate right	ROR Ra, Rb, Rc	Ra <= Rb ror Rc
A	CMP	2A	Compare	CMP Ra, Rb	SW <= (Ra cond Rb)[5]
A	CMPu	2B	Compare	CMPu Ra, Rb	SW <= (Ra cond Rb)[Page 25, 5]
J	JEQ	30	Jump if equal (==)	JEQ Cx	if SW(==), PC <= PC + Cx
J	JNE	31	Jump if not equal (!=)	JNE Cx	if SW(!=), PC <= PC + Cx
J	JLT	32	Jump if less than (<)	JLT Cx	if SW(<), PC <= PC + Cx
J	JGT	33	Jump if greater than (>)	JGT Cx	if SW(>), PC <= PC + Cx
J	JLE	34	Jump if less than or equals (<=)	JLE Cx	if SW(<=), PC <= PC + Cx
J	JGE	35	Jump if greater than or equals (>=)	JGE Cx	if SW(>=), PC <= PC + Cx
J	JMP	36	Jump (unconditional)	JMP Cx	PC <= PC + Cx
J	JALR	39	Indirect jump	JALR Rb	LR <= PC; PC <= Rb[7]
J	BAL	3A	Branch and link	BAL Cx	LR <= PC; PC <= PC + Cx

continues on next page

Table 2.6 – continued from previous page

F.	Mnemonic	Op-code	Meaning	Syntax	Operation
J	JSUB	3B	Jump to subroutine	JSUB Cx	LR <= PC; PC <= PC + Cx
J	JR/RET	3C	Return from subroutine	JR $1 or RET LR	PC <= LR[8]
A	MULT	41	Multiply for 64 bits result	MULT Ra, Rb	(HI,LO) <= MULT(Ra,Rb)
A	MULTU	42	MULT for unsigned 64 bits	MULTU Ra, Rb	(HI,LO) <= MULTU(Ra,Rb)
A	DIV	43	Divide	DIV Ra, Rb	HI<=Ra%Rb, LO<=Ra/Rb
A	DIVU	44	Divide unsigned	DIVU Ra, Rb	HI<=Ra%Rb, LO<=Ra/Rb
A	MFHI	46	Move HI to GPR	MFHI Ra	Ra <= HI
A	MFLO	47	Move LO to GPR	MFLO Ra	Ra <= LO
A	MTHI	48	Move GPR to HI	MTHI Ra	HI <= Ra
A	MTLO	49	Move GPR to LO	MTLO Ra	LO <= Ra
A	MFC0	50	Move C0R to GPR	MFC0 Ra, Rb	Ra <= Rb
A	MTC0	51	Move GPR to C0R	MTC0 Ra, Rb	Ra <= Rb
A	C0MOV	52	Move C0R to C0R	C0MOV Ra, Rb	Ra <= Rb

The following table details the cpu032II instruction set added:

Table 2.7: cpu032II Instruction Set

F.	Mnemonic	Op-code	Meaning	Syntax	Operation
L	SLTi	26	Set less Then	SLTi Ra, Rb, Cx	Ra <= (Rb < Cx)
L	SLTiu	27	SLTi unsigned	SLTiu Ra, Rb, Cx	Ra <= (Rb < Cx)
A	SLT	28	Set less Then	SLT Ra, Rb, Rc	Ra <= (Rb < Rc)
A	SLTu	29	SLT unsigned	SLTu Ra, Rb, Rc	Ra <= (Rb < Rc)
L	BEQ	37	Branch if equal	BEQ Ra, Rb, Cx	if (Ra==Rb), PC <= PC + Cx
L	BNE	38	Branch if not equal	BNE Ra, Rb, Cx	if (Ra!=Rb), PC <= PC + Cx

Note: Cpu0 unsigned instructions

Like Mips, except DIVU, the mathematic unsigned instructions such as ADDu and SUBu, are instructions of no overflow exception. The ADDu and SUBu handle both signed and unsigned integers well. For example, (ADDu 1, -2) is -1; (ADDu 0x01, 0xfffffffe) is 0xffffffff = (4G - 1). If you treat the result is negative then it is -1. On the other hand, it's (+4G - 1) if you treat the result is positive.

[3] The difference between LB and LBu is signed and unsigned byte value expand to a word size. For example, After LB Ra, [Rb+Cx], Ra is 0xffffff80(= -128) if byte [Rb+Cx] is 0x80; Ra is 0x0000007f(= 127) if byte [Rb+Cx] is 0x7f. After LBu Ra, [Rb+Cx], Ra is 0x00000080(= 128) if byte [Rb+Cx] is 0x80; Ra is 0x0000007f(= 127) if byte [Rb+Cx] is 0x7f. Difference between LH and LHu is similar.

[4] The only difference between ADDu instruction and the ADD instruction is that the ADDU instruction never causes an Integer Overflow exception. SUBu and SUB is similar.

[6] Rb '>> Cx, Rb '>> Rc: Shift with signed bit remain.

[5] CMP is signed-compare while CMPu is unsigned. Conditions include the following comparisons: >, >=, ==, !=, <=, <. SW is actually set by the subtraction of the two register operands, and the flags indicate which conditions are present.

[7] jsub cx is direct call for 24 bits value of cx while jalr $rb is indirect call for 32 bits value of register $rb.

[8] Both JR and RET has same opcode (actually they are the same instruction for Cpu0 hardware). When user writes "jr $t9" meaning it jumps to address of register $t9; when user writes "jr $lr" meaning it jump back to the caller function (since $lr is the return address). For user read ability, Cpu0 prints "ret $lr" instead of "jr $lr".

Why not using ADD instead of SUB?

From text book of computer introduction, we know SUB can be replaced by ADD as follows,

- (A - B) = (A + (-B))

Since Mips uses 32 bits to represent int type of C language, if B is the value of -2G, then

- (A - (-2G)) = (A + (2G))

But the problem is value -2G can be represented in 32 bits machine while 2G cannot, since the range of 2's complement representation for 32 bits is (-2G .. 2G-1). The 2's complement reprentation has the merit of fast computation in circuits design, it is widely used in real CPU implementation. That's why almost every CPU create SUB instruction, rather than using ADD instead of.

2.1.3 The Status Register

The Cpu0 status word register (SW) contains the state of the Negative (N), Zero (Z), Carry (C), Overflow (V), Debug (D), Mode (M), and Interrupt (I) flags. The bit layout of the SW register is shown in Fig. 2.3 below.

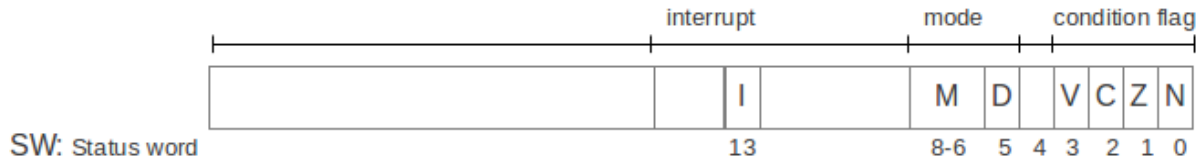


Fig. 2.3: Cpu0 status word (SW) register

When a CMP Ra, Rb instruction executes, the condition flags will change. For example:

- If Ra > Rb, then N = 0, Z = 0
- If Ra < Rb, then N = 1, Z = 0
- If Ra = Rb, then N = 0, Z = 1

The direction (i.e. taken/not taken) of the conditional jump instructions JGT, JLT, JGE, JLE, JEQ, JNE is determined by the N and Z flags in the SW register.

2.1.4 Cpu0's Stages of Instruction Execution

The Cpu0 architecture has a five-stage pipeline. The stages are instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM) and write backe (WB). Here is a description of what happens in the processor for each stage:

1) Instruction fetch (IF)

- The Cpu0 fetches the instruction pointed to by the Program Counter (PC) into the Instruction Register (IR): IR = [PC].
- The PC is then updated to point to the next instruction: PC = PC + 4.

2) Instruction decode (ID)

- The control unit decodes the instruction stored in IR, which routes necessary data stored in registers to the ALU, and sets the ALU's operation mode based on the current instruction's opcode.

3) Execute (EX)

- The ALU executes the operation designated by the control unit upon data in registers. Except load and store instructions, the result is stored in the destination register after the ALU is done.

4) Memory access (MEM)

- Read data from data cache to pipeline register MEM/WB if it is load instruction; write data from register to data cache if it is strore instruction.

5) Write-back (WB)

- Move data from pipeline register MEM/WB to Register if it is load instruction.

2.1.5 Cpu0's Interrupt Vector

Table 2.8: Cpu0's Interrupt Vector

Address	type
0x00	Reset
0x04	Error Handle
0x08	Interrupt

2.2 LLVM Structure

This section introduces the compiler data structure, algorithm and mechanism that llvm uses.

2.2.1 Three-phase design

This content and the following sub-section comes from the AOSA chapter on LLVM written by Chris Lattner[Page 18, 10].

The most popular design for a traditional static compiler (like most C compilers) is the three phase design whose major components are the front end, the optimizer and the back end, as seen in Fig. 2.4. The front end parses source code, checking it for errors, and builds a language-specific Abstract Syntax Tree (AST) to represent the input code. The AST is optionally converted to a new representation for optimization, and the optimizer and back end are run on the code.

Fig. 2.4: Three Major Components of a Three Phase Compiler

The optimizer is responsible for doing a broad variety of transformations to try to improve the code's running time, such as eliminating redundant computations, and is usually more or less independent of language and target. The back end (also known as the code generator) then maps the code onto the target instruction set. In addition to making correct code, it is responsible for generating good code that takes advantage of unusual features of the supported architecture. Common parts of a compiler back end include instruction selection, register allocation, and instruction scheduling.

This model applies equally well to interpreters and JIT compilers. The Java Virtual Machine (JVM) is also an implementation of this model, which uses Java bytecode as the interface between the front end and optimizer.

The most important win of this classical design comes when a compiler decides to support multiple source languages or target architectures. If the compiler uses a common code representation in its optimizer, then a front end can be

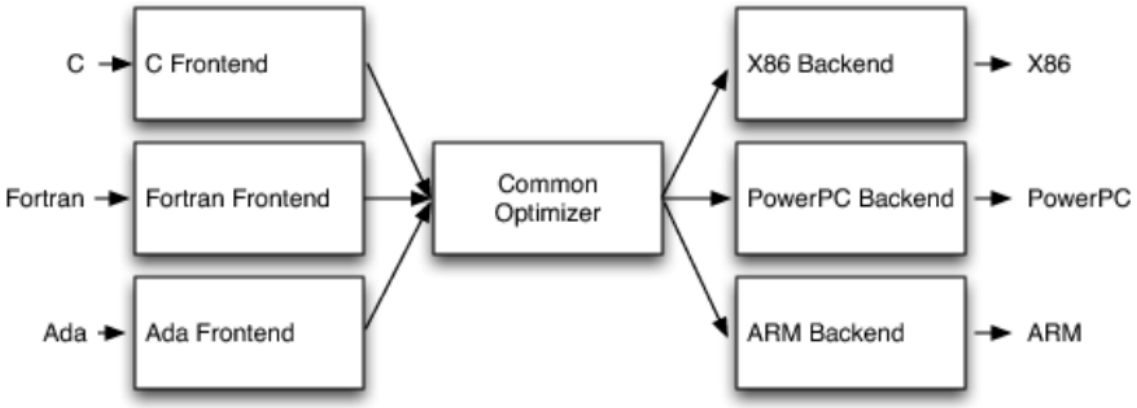


Fig. 2.5: Retargetablity

written for any language that can compile to it, and a back end can be written for any target that can compile from it, as shown in Fig. 2.5.

With this design, porting the compiler to support a new source language (e.g., Algol or BASIC) requires implementing a new front end, but the existing optimizer and back end can be reused. If these parts weren't separated, implementing a new source language would require starting over from scratch, so supporting N targets and M source languages would need N*M compilers.

Another advantage of the three-phase design (which follows directly from retargetability) is that the compiler serves a broader set of programmers than it would if it only supported one source language and one target. For an open source project, this means that there is a larger community of potential contributors to draw from, which naturally leads to more enhancements and improvements to the compiler. This is the reason why open source compilers that serve many communities (like GCC) tend to generate better optimized machine code than narrower compilers like FreePASCAL. This isn't the case for proprietary compilers, whose quality is directly related to the project's budget. For example, the Intel ICC Compiler is widely known for the quality of code it generates, even though it serves a narrow audience.

A final major win of the three-phase design is that the skills required to implement a front end are different than those required for the optimizer and back end. Separating these makes it easier for a "front-end person" to enhance and maintain their part of the compiler. While this is a social issue, not a technical one, it matters a lot in practice, particularly for open source projects that want to reduce the barrier to contributing as much as possible.

The most important aspect of its design is the LLVM Intermediate Representation (IR), which is the form it uses to represent code in the compiler. LLVM IR is designed to host mid-level analyses and transformations that you find in the optimizer chapter of a compiler. It was designed with many specific goals in mind, including supporting lightweight runtime optimizations, cross-function/interprocedural optimizations, whole program analysis, and aggressive restructuring transformations, etc. The most important aspect of it, though, is that it is itself defined as a first class language with well-defined semantics. To make this concrete, here is a simple example of a .ll file:

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}
define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse
recurse:
  %tmp2 = sub i32 %a, 1
```

(continues on next page)

(continued from previous page)

```
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4
done:
  ret i32 %b
}
```

```
// Above LLVM IR corresponds to this C code, which provides two different ways to
//  add integers:
unsigned add1(unsigned a, unsigned b) {
  return a+b;
}
// Perhaps not the most efficient way to add two numbers.
unsigned add2(unsigned a, unsigned b) {
  if (a == 0) return b;
  return add2(a-1, b+1);
}
```

As you can see from this example, LLVM IR is a low-level RISC-like virtual instruction set. Like a real RISC instruction set, it supports linear sequences of simple instructions like add, subtract, compare, and branch. These instructions are in three address form, which means that they take some number of inputs and produce a result in a different register. LLVM IR supports labels and generally looks like a weird form of assembly language.

Unlike most RISC instruction sets, LLVM is strongly typed with a simple type system (e.g., i32 is a 32-bit integer, i32** is a pointer to pointer to 32-bit integer) and some details of the machine are abstracted away. For example, the calling convention is abstracted through call and ret instructions and explicit arguments. Another significant difference from machine code is that the LLVM IR doesn't use a fixed set of named registers, it uses an infinite set of temporaries named with a % character.

Beyond being implemented as a language, LLVM IR is actually defined in three isomorphic forms: the textual format above, an in-memory data structure inspected and modified by optimizations themselves, and an efficient and dense on-disk binary "bitcode" format. The LLVM Project also provides tools to convert the on-disk format from text to binary: llvm-as assembles the textual .ll file into a .bc file containing the bitcode goop and llvm-dis turns a .bc file into a .ll file.

The intermediate representation of a compiler is interesting because it can be a "perfect world" for the compiler optimizer: unlike the front end and back end of the compiler, the optimizer isn't constrained by either a specific source language or a specific target machine. On the other hand, it has to serve both well: it has to be designed to be easy for a front end to generate and be expressive enough to allow important optimizations to be performed for real targets.

2.2.2 LLVM's Target Description Files: .td

The "mix and match" approach allows target authors to choose what makes sense for their architecture and permits a large amount of code reuse across different targets. This brings up another challenge: each shared component needs to be able to reason about target specific properties in a generic way. For example, a shared register allocator needs to know the register file of each target and the constraints that exist between instructions and their register operands. LLVM's solution to this is for each target to provide a target description in a declarative domain-specific language (a set of .td files) processed by the tblgen tool. The (simplified) build process for the x86 target is shown in Fig. 2.6.

The different subsystems supported by the .td files allow target authors to build up the different pieces of their target. For example, the x86 back end defines a register class that holds all of its 32-bit registers named "GR32" (in the .td files, target specific definitions are all caps) like this:

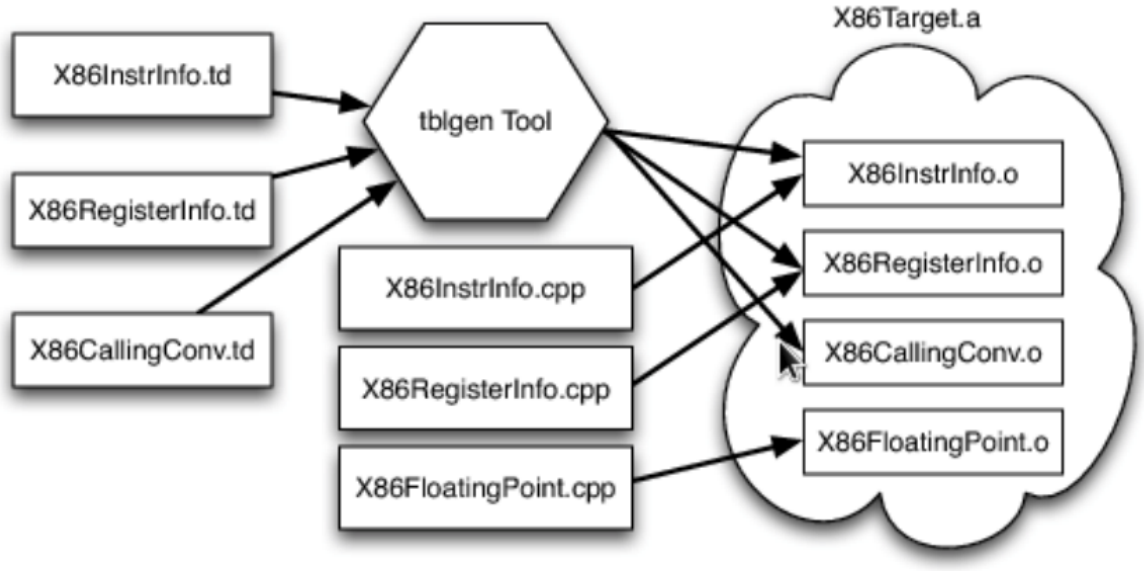


Fig. 2.6: Simplified x86 Target Definition

```
def GR32 : RegisterClass<[i32], 32,
  [EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP,
   R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D]> { ... }
```

The language used in .td files are Target(Hardware) Description Language that let llvm backend compiler engineers to define the transformation for llvm IR and the machine instructions of their CPUs. In frontend, compiler development tools provide the "Parser Generator" for compiler development; in backend, they provide the "Machine Code Generator" for development, as Fig. 2.7 and Fig. 2.8.

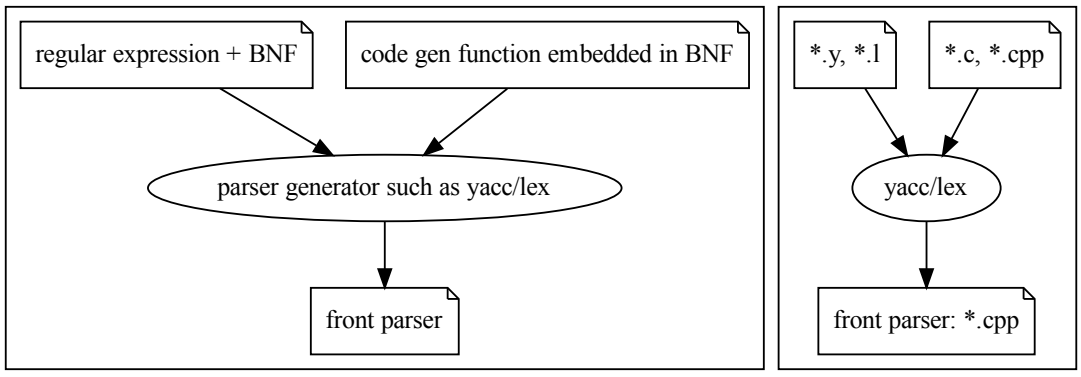


Fig. 2.7: Front TableGen Flow

Since the c++'s grammar is more context-sensitive than context-free, llvm frontend project clang uses handcode parser without BNF generator tools. In backend development, the IR to machine instructions transformation can get great

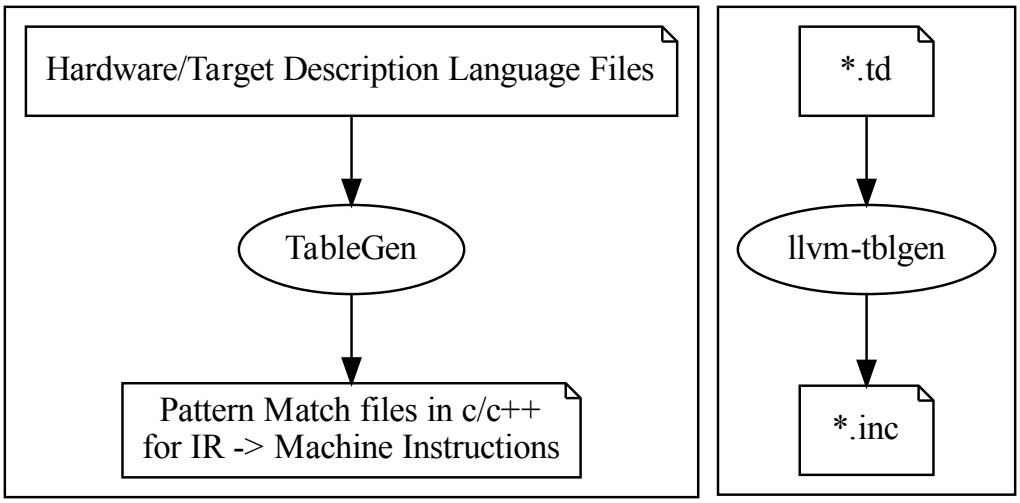


Fig. 2.8: llvm TableGen Flow

benefits from TableGen tools. Though c++ compiler cannot get benefit from BNF generator tools, many computer languages and script languages are more context-free and can get benefit from the tools.

The following come from wiki:

Java syntax has a context-free grammar that can be parsed by a simple LALR parser. Parsing C++ is more complicated[9].

The gnu g++ compiler abandoned BNF tools since version 3.x. I think another reason beyond that c++ has more context-sensitive grammar is handcode parser can provide better error diagnosis than BNF tool since BNF tool always select the rules from BNF grammar if match.

2.2.3 LLVM Code Generation Sequence

Following diagram come from tricore_llvm.pdf.

LLVM is a Static Single Assignment (SSA) based representation. LLVM provides an infinite virtual registers which can hold values of primitive type (integral, floating point, or pointer values). So, every operand can be saved in different virtual register in llvm SSA representation. Comment is ";" in llvm representation. Following is the llvm SSA instructions.

```
store i32 0, i32* %a  ; store i32 type of 0 to virtual register %a, %a is
                      ;  pointer type which point to i32 value
store i32 %b, i32* %c ; store %b contents to %c point to, %b isi32 type virtual
                      ;  register, %c is pointer type which point to i32 value.
%a1 = load i32* %a    ; load the memory value where %a point to and assign the
                      ;  memory value to %a1
%a3 = add i32 %a2, 1  ; add %a2 and 1 and save to %a3
```

[9] https://en.wikipedia.org/wiki/Comparison_of_Java_and_C%2B%2B

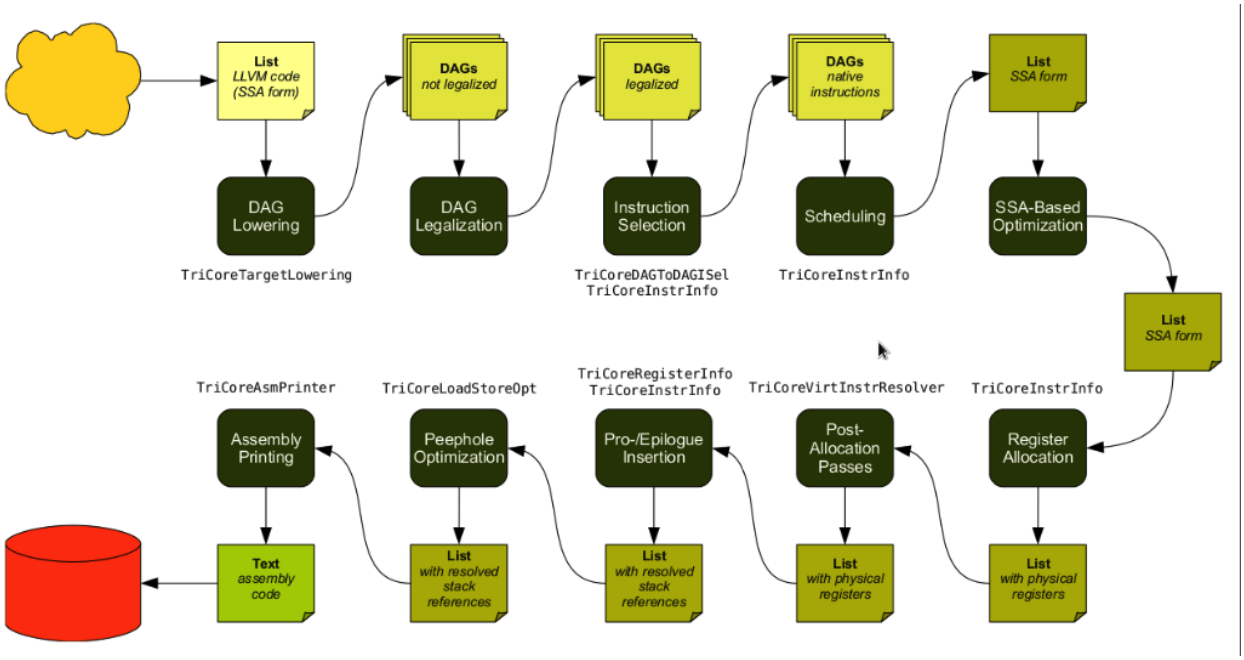


Fig. 2.9: tricore_llvm.pdf: Code generation sequence. On the path from LLVM code to assembly code, numerous passes are run through and several data structures are used to represent the intermediate results.

We explain the code generation process as below. If you don't feel comfortable, please check tricore_llvm.pdf section 4.2 first. You can read "The LLVM Target-Independent Code Generator" from here[12] and "LLVM Language Reference Manual" from here[Page 23, 13] before go ahead, but we think the section 4.2 of tricore_llvm.pdf is enough and suggesting you read the web site documents as above only when you are still not quite understand, even if you have read the articles of this section and next two sections for DAG and Instruction Selection.

1. Instruction Selection

```
// In this stage, transfer the llvm opcode into machine opcode, but the operand
//  still is llvm virtual operand.
    store i16 0, i16* %a // store 0 of i16 type to where virtual register %a
                         //  point to.
=>  st i16 0, i32* %a    // Use Cpu0 backend instruction st instead of IR store.
```

2. Scheduling and Formation

```
// In this stage, reorder the instructions sequence for optimization in
//  instructions cycle or in register pressure.
    st i32 %a, i16* %b,  i16 5 // st %a to *(%b+5)
    st %b, i32* %c, i16 0
    %d = ld i32* %c

// Transfer above instructions order as follows. In RISC CPU of Mips, the ld
//  %c uses the result of the previous instruction st %c. So it must waits 1
//  cycle. Meaning the ld cannot follow st immediately.
=>  st %b, i32* %c, i16 0
    st i32 %a, i16* %b,  i16 5
    %d = ld i32* %c, i16 0
```

(continues on next page)

[12] http://llvm.org/docs/CodeGenerator.html

(continued from previous page)

```
// If without reorder instructions, a instruction nop which do nothing must be
//  filled, contribute one instruction cycle more than optimization. (Actually,
//  Mips is scheduled with hardware dynamically and will insert nop between st
//  and ld instructions if compiler didn't insert nop.)
    st i32 %a, i16* %b,  i16 5
    st %b, i32* %c, i16 0
    nop
    %d = ld i32* %c, i16 0

// Minimum register pressure
//  Suppose %c is alive after the instructions basic block (meaning %c will be
//  used after the basic block), %a and %b are not alive after that.
// The following no-reorder-version need 3 registers at least
    %a = add i32 1, i32 0
    %b = add i32 2, i32 0
    st %a,  i32* %c, 1
    st %b,  i32* %c, 2

// The reorder version needs 2 registers only (by allocate %a and %b in the same
//  register)
=> %a = add i32 1, i32 0
    st %a,  i32* %c, 1
    %b = add i32 2, i32 0
    st %b,  i32* %c, 2
```

3. SSA-based Machine Code Optimization

   For example, common expression remove, shown in next section DAG.

4. Register Allocation

   Allocate real register for virtual register.

5. Prologue/Epilogue Code Insertion

   Explain in section Add Prologue/Epilogue functions

6. Late Machine Code Optimizations

   Any "last-minute" peephole optimizations of the final machine code can be applied during this phase. For example, replace x = x * 2 by x = x < 1 for integer operand.

7. Code Emission

   Finally, the completed machine code is emitted. For static compilation, the end result is an assembly code file; for JIT compilation, the opcodes of the machine instructions are written into memory.

The llvm code generation sequence also can be obtained by `llc -debug-pass=Structure` as the following. The first 4 code generation sequences from Fig. 2.9 are in the **'DAG->DAG Pattern Instruction Selection'** of the `llc -debug-pass=Structure` displayed. The order of Peephole Optimizations and Prologue/Epilogue Insertion is inconsistent between Fig. 2.9 and `llc -debug-pass=Structure` (please check the * in the following). No need to be bothered with this since the the LLVM is under development and changed from time to time.

```
118-165-79-200:input Jonathan$ llc --help-hidden
OVERVIEW: llvm system compiler

USAGE: llc [options] <input bitcode>
```

(continues on next page)

(continued from previous page)

```
OPTIONS:
...
  -debug-pass                             - Print PassManager debugging information
    =None                                 -   disable debug output
    =Arguments                            -   print pass arguments to pass to 'opt'
    =Structure                            -   print pass structure before run()
    =Executions                           -   print pass name before it is executed
    =Details                              -   print pass details when it is executed

118-165-79-200:input Jonathan$ llc -march=mips -debug-pass=Structure ch3.bc
...
Target Library Information
Target Transform Info
Data Layout
Target Pass Configuration
No Alias Analysis (always returns 'may' alias)
Type-Based Alias Analysis
Basic Alias Analysis (stateless AA impl)
Create Garbage Collector Module Metadata
Machine Module Information
Machine Branch Probability Analysis
  ModulePass Manager
    FunctionPass Manager
      Preliminary module verification
      Dominator Tree Construction
      Module Verifier
      Natural Loop Information
      Loop Pass Manager
        Canonicalize natural loops
      Scalar Evolution Analysis
      Loop Pass Manager
        Canonicalize natural loops
        Induction Variable Users
        Loop Strength Reduction
      Lower Garbage Collection Instructions
      Remove unreachable blocks from the CFG
      Exception handling preparation
      Optimize for code generation
      Insert stack protectors
      Preliminary module verification
      Dominator Tree Construction
      Module Verifier
      Machine Function Analysis
      Natural Loop Information
      Branch Probability Analysis
    * MIPS DAG->DAG Pattern Instruction Selection
      Expand ISel Pseudo-instructions
      Tail Duplication
      Optimize machine instruction PHIs
      MachineDominator Tree Construction
      Slot index numbering
```

(continues on next page)

(continued from previous page)

```
    Merge disjoint stack slots
    Local Stack Slot Allocation
    Remove dead machine instructions
    MachineDominator Tree Construction
    Machine Natural Loop Construction
    Machine Loop Invariant Code Motion
    Machine Common Subexpression Elimination
    Machine code sinking
  * Peephole Optimizations
    Process Implicit Definitions
    Remove unreachable machine basic blocks
    Live Variable Analysis
    Eliminate PHI nodes for register allocation
    Two-Address instruction pass
    Slot index numbering
    Live Interval Analysis
    Debug Variable Analysis
    Simple Register Coalescing
    Live Stack Slot Analysis
    Calculate spill weights
    Virtual Register Map
    Live Register Matrix
    Bundle Machine CFG Edges
    Spill Code Placement Analysis
  * Greedy Register Allocator
    Virtual Register Rewriter
    Stack Slot Coloring
    Machine Loop Invariant Code Motion
  * Prologue/Epilogue Insertion & Frame Finalization
    Control Flow Optimizer
    Tail Duplication
    Machine Copy Propagation Pass
  * Post-RA pseudo instruction expansion pass
    MachineDominator Tree Construction
    Machine Natural Loop Construction
    Post RA top-down list latency scheduler
    Analyze Machine Code For Garbage Collection
    Machine Block Frequency Analysis
    Branch Probability Basic Block Placement
    Mips Delay Slot Filler
    Mips Long Branch
    MachineDominator Tree Construction
    Machine Natural Loop Construction
  * Mips Assembly Printer
    Delete Garbage Collector Information
```

- Since Instructions Scheduling and Dead Code Removing will affect Register Allocation. However llvm does not go from later pass onto earlier pass again. The Register Allocation is after Instruction Scheduling. The passes from Live Variable Analysis to Greedy Register Allocator are passes for Register Allocation. About Register Allocation Passes are here[14][15].

[14] https://www.cs.cmu.edu/afs/cs/academic/class/15745-s16/www/lectures/L23-Register-Coalescing.pdf
[15] https://en.wikipedia.org/wiki/Register_allocation

2.2.4 SSA form

SSA form says that each variable is assigned exactly once. LLVM IR is SSA form which has unbounded virtual registers (each variable is assigned exactly once and is keeped in different virtual register). As the result, the optimization steps used in code generation sequence which include stages of **Instruction Selection**, **Scheduling and Formation** and **Register Allocation**, won't loss any optimization opportunity. For example, if using limited virtual registers instead of unlimited as the following code,

```
%a = add nsw i32 1, i32 0
store i32 %a, i32* %c, align 4
%a = add nsw i32 2, i32 0
store i32 %a, i32* %c, align 4
```

Above using limited virtual registers, so virtual register %a used twice. Compiler have to generate the following code since it assigns virtual register %a as output at two different statement.

=> %a = add i32 1, i32 0
st %a, i32* %c, 1 %a = add i32 2, i32 0 st %a, i32* %c, 2

Above code have to run in sequence. On the other hand, the SSA form as the following can be reodered and run in parallel with the following different version[16].

```
  %a = add nsw i32 1, i32 0
  store i32 %a, i32* %c, align 4
  %b = add nsw i32 2, i32 0
  store i32 %b, i32* %d, align 4

// version 1
=> %a = add i32 1, i32 0
    st %a,  i32* %c, 0
    %b = add i32 2, i32 0
    st %b,  i32* %d, 0

// version 2
=> %a = add i32 1, i32 0
    %b = add i32 2, i32 0
    st %a,  i32* %c, 0
    st %b,  i32* %d, 0

// version 3
=> %b = add i32 2, i32 0
    st %b,  i32* %d, 0
    %a = add i32 1, i32 0
    st %a,  i32* %c, 0
```

[16] Refer section 10.2.3 of book Compilers: Principles, Techniques, and Tools (2nd Edition)

2.2.5 DSA form

```
for (int i = 0; i < 1000; i++) {
  b[i] = f(g(a[i]));
}
```

For the source program as above, the following are the SSA form in source code level and llvm IR level respectively.

```
for (int i = 0; i < 1000; i++) {
  t = g(a[i]);
  b[i] = f(t);
}
```

```
  %pi = alloca i32
  store i32 0, i32* %pi
  %i = load i32, i32* %pi
  %cmp = icmp slt i32 %i, 1000
  br i1 %cmp, label %true, label %end
true:
  %a_idx = add i32 %i, i32 %a_addr
  %val0 = load i32, i32* %a_idx
  %t = call i64 %g(i32 %val0)
  %val1 = call i64 %f(i32 %t)
  %b_idx = add i32 %i, i32 %b_addr
  store i32 %val1, i32* %b_idx
end:
```

The following is the DSA (Dynamic Single Assignment) form.

```
for (int i = 0; i < 1000; i++) {
  t[i] = g(a[i]);
  b[i] = f(t[i]);
}
```

```
  %pi = alloca i32
  store i32 0, i32* %pi
  %i = load i32, i32* %pi
  %cmp = icmp slt i32 %i, 1000
  br i1 %cmp, label %true, label %end
true:
  %a_idx = add i32 %i, i32 %a_addr
  %val0 = load i32, i32* %a_idx
  %t_idx = add i32 %i, i32 %t_addr
  %temp = call i64 %g(i32 %val0)
  store i32 %temp, i32* %t_idx
  %val1 = call i64 %f(i32 %temp)
  %b_idx = add i32 %i, i32 %b_addr
  store i32 %val1, i32* %b_idx
end:
```

In some internet video applications and muti-core (SMP) platforms, splitting g() and f() to two different loop have better perfomance. DSA can split as the following while SSA cannot. Of course, it's possible to do extra analysis on %temp of SSA and reverse it into %t_idx and %t_addr as the following DSA. But in compiler discussion, the translation is from high to low level of machine code. Besides, as you see, the llvm ir lose the for loop information already though it can

be reconstructed by extra analysis. So, in this book and almost every paper in compiler discuss with this high-to-low premise, otherwise it's talking about reverse engineering in assembler or compiler.

```
for (int i = 0; i < 1000; i++) {
  t[i] = g(a[i]);
}

for (int i = 0; i < 1000; i++) {
  b[i] = f(t[i]);
}
```

```
  %pi = alloca i32
  store i32 0, i32* %pi
  %i = load i32, i32* %pi
  %cmp = icmp slt i32 %i, 1000
  br i1 %cmp, label %true, label %end
true:
  %a_idx = add i32 %i, i32 %a_addr
  %val0 = load i32, i32* %a_idx
  %t_idx = add i32 %i, i32 %t_addr
  %temp = call i32 %g(i32 %val0)
  store i32 %temp, i32* %t_idx
end:

  %pi = alloca i32
  store i32 0, i32* %pi
  %i = load i32, i32* %pi
  %cmp = icmp slt i32 %i, 1000
  br i1 %cmp, label %true, label %end
true:
  %t_idx = add i32 %i, i32 %t_addr
  %temp = load i32, i32* %t_idx
  %val1 = call i32 %f(i32 %temp)
  %b_idx = add i32 %i, i32 %b_addr
  store i32 %val1, i32* %b_idx
end:
```

Now, the data dependences only exist on t[i] between "t[i] = g(a[i])" and "b[i] = f(t[i])" for each i = (0..999). The program can be run on many different order, and it provides many parallel processing opportunities for multi-core (SMP) and heterogeneous processors. For instance, g(x) is run on GPU and f(x) is run on CPU.

2.2.6 LLVM vs GCC in structure

GCC document is here[17] .

Table 2.9: clang vs gcc-frontend

frontend	clang	gcc-frontend[Page 39, 18]
LANGUAGE	C/C++	C/C++
parsing	parsing	parsing
AST	clang-AST	GENERIC[19]
optimization & codgen	clang-backend	gimplifier
IR	LLVM IR	GIMPLE[20]

Table 2.10: llvm vs gcc (kernal and target/backend)

backend	llvm	gcc
IR	LLVM IR	GIMPLE
transfer	optimziation & pass	optimization & plugins
DAG	DAG	RTL[21]
codgen	tblgen for td	codgen for md[22]

Both LLVM IR and GIMPLE are SSA form. LLVM IR originally designed to be fully reusable across arbitrary tools besides compiler itself. GCC community never had desire to enable any tools besides compiler (Richard Stallman resisted attempts to make IR more reusable to prevent third-party commercial tools from reusing GCC's frontends). Thus GIMPLE (GCC's IR) was never considered to be more than an implementation detail, in particular it doesn't provide a full description of compiled program (e.g. it lacks program's call graph, type definitions, stack offsets and alias information)[23].

2.2.7 LLVM blog

User uses null pointer to guard code is correct. Undef is only happened in compiler optimization[24]. However when user forget to bind null pointer in guarding code directly or indirectly, compiler such as llvm and gcc may treat null pointer as undef and optimzation out[25].

[17] https://en.wikipedia.org/wiki/GNU_Compiler_Collection
[18] https://en.wikipedia.org/wiki/GNU_Compiler_Collection#Front_ends
[19] https://gcc.gnu.org/onlinedocs/gccint/GENERIC.html
[20] https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html
[21] https://gcc.gnu.org/onlinedocs/gccint/RTL.html
[22] https://gcc.gnu.org/onlinedocs/gccint/Machine-Desc.html#Machine-Desc
[23] https://stackoverflow.com/questions/40799696/how-is-gcc-ir-different-from-llvm-ir/40802063
[24] https://github.com/Jonathan2251/lbd/tree/master/References/null_pointer.cpp is an example.
[25] Dereferencing a NULL Pointer: contrary to popular belief, dereferencing a null pointer in C is undefined. It is not defined to trap, and if you mmap a page at 0, it is not defined to access that page. This falls out of the rules that forbid dereferencing wild pointers and the use of NULL as a sentinel, from http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html. As link, https://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html. In this case, the developer forgot to call "set", did not crash with a null pointer dereference, and their code broke when someone else did a debug build.

2.2.8 CFG (Control Flow Graph)

The SSA form can be depicted in CFG and do optimization through the analysis on CFG. Each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block[26].

The following is an example of CFG. **The jump/branch always in the last statement of BBs (Basic Blocks)** in Fig. 2.10.

Fig/llvmstructure/cfg-ex.cpp

```
int cfg_ex(int a, int b, int n)
{
  for (int i = 0; i <= n; i++) {
    if (a < b) {
      a = a + i;
      b = b - 1;
    }
    if (b == 0) {
      goto label_1;
    }
  }

label_1:
  switch (a) {
  case 10:
    a = a*a-b+2;
    a++;
    break;
  }

  return (a+b);
}
```

Fig/llvmstructure/cfg-ex.ll

```
define dso_local i32 @_Z6cfg_exiii(i32 signext %a, i32 signext %b, i32 signext %n) local_
↪unnamed_addr nounwind {
entry:
  %cmp.not23 = icmp slt i32 %n, 0
  br i1 %cmp.not23, label %cleanup, label %for.body

for.cond:                                         ; preds = %for.body
  %inc = add nuw i32 %i.026, 1
  %exitcond.not = icmp eq i32 %i.026, %n
  br i1 %exitcond.not, label %cleanup, label %for.body, !llvm.loop !2

for.body:                                         ; preds = %entry, %for.cond
  %i.026 = phi i32 [ %inc, %for.cond ], [ 0, %entry ]
  %a.addr.025 = phi i32 [ %a.addr.1, %for.cond ], [ %a, %entry ]
```

(continues on next page)

[26] https://en.wikipedia.org/wiki/Control-flow_graph

(continued from previous page)

```
  %b.addr.024 = phi i32 [ %b.addr.1, %for.cond ], [ %b, %entry ]
  %cmp1 = icmp slt i32 %a.addr.025, %b.addr.024
  %sub = sext i1 %cmp1 to i32
  %b.addr.1 = add nsw i32 %b.addr.024, %sub
  %add = select i1 %cmp1, i32 %i.026, i32 0
  %a.addr.1 = add nsw i32 %add, %a.addr.025
  %cmp2 = icmp eq i32 %b.addr.1, 0
  br i1 %cmp2, label %cleanup, label %for.cond

cleanup:                                          ; preds = %for.cond, %for.body, %entry
  %b.addr.2 = phi i32 [ %b, %entry ], [ 0, %for.body ], [ %b.addr.1, %for.cond ]
  %a.addr.2 = phi i32 [ %a, %entry ], [ %a.addr.1, %for.body ], [ %a.addr.1, %for.cond ]
  %cond = icmp eq i32 %a.addr.2, 10
  %inc7 = sub i32 103, %b.addr.2
  %spec.select = select i1 %cond, i32 %inc7, i32 %a.addr.2
  %add8 = add nsw i32 %spec.select, %b.addr.2
  ret i32 %add8
}


!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 12.0.1"}
!2 = distinct !{!2, !3}
!3 = !{!"llvm.loop.mustprogress"}
```

2.2.9 DAG (Directed Acyclic Graph)

The SSA in each BB (Basic Block) from CFG as the previous section can be represented in DAG.

Many important techniques for local optimization begin by transforming a basic block into DAG[27]. For example, the basic block code and it's corresponding DAG as Fig. 2.11.

If b is not live on exit from the block, then we can do "common expression remove" as the following table.

Table 2.11: common expression remove process

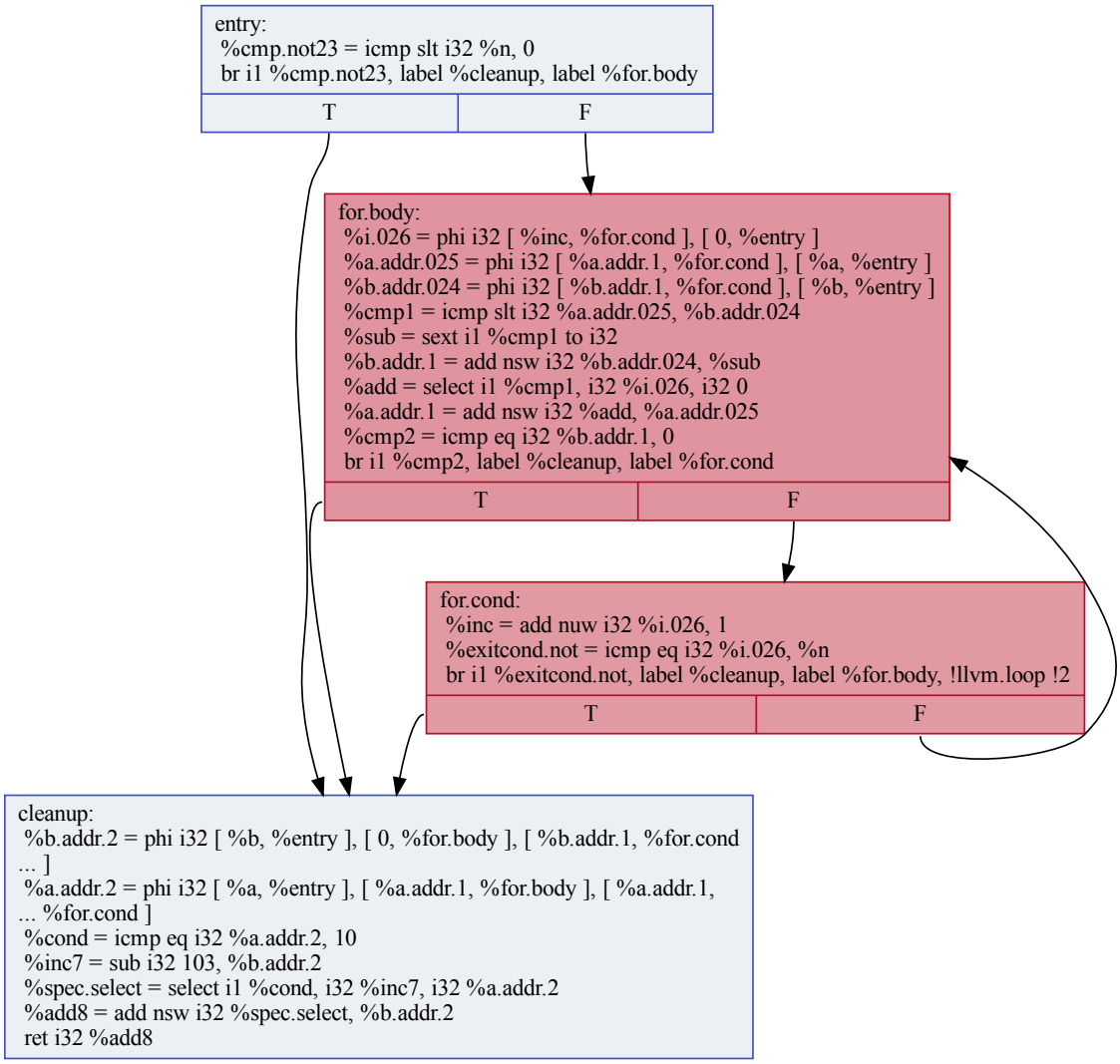
Replace node b with node d	Replace b_0, c_0, d_0 with b, c, d
$a = b_0 + c_0$	a = b + c
$d = a - d_0$	d = a – d
c = d + c	c = d + c

After removing b and traversing the DAGs from bottom to top (traverse binary tree by Depth-first In-order search) , the first column of above table will be gotten.

As you can imagine, the "common expression remove" can apply both in IR or machine code.

DAG is like a tree which opcode is the node and operand (register and const/immediate/offset) is leaf. It can also be represented by list as prefix order in tree. For example, (+ b, c), (+ b, 1) is IR DAG representation.

[27] Refer section 8.5 of book Compilers: Principles, Techniques, and Tools (2nd Edition)



CFG for '_Z6cfg_exiii' function

Fig. 2.10: CFG for cfg-ex.ll

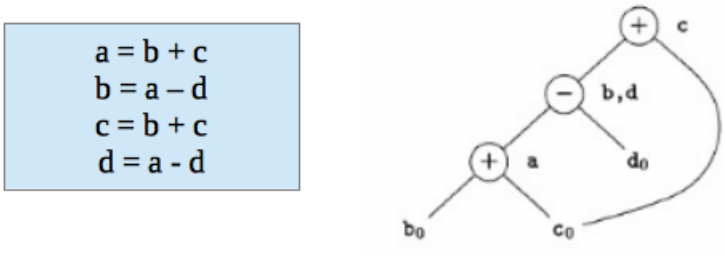


Fig. 2.11: DAG example

In addition to DAG optimization, the "kill" register has also mentioned in section 8.5.5 of the compiler book[Page 41, 27]. This optimization method also applied in llvm implementation.

2.2.10 Instruction Selection

The major function of backend is that translate IR code into machine code at stage of Instruction Selection as Fig. 2.12.

MOV	$r_d = r_s$	ADDI	$r_d = r_s + 0$
MOV	$r_d = r_s$	ADD	$r_d = r_{s1} + r_0$
MOVI	$r_d = c$	ADDI	$r_d = r_0 + c$

Fig. 2.12: IR and it's corresponding machine instruction

For machine instruction selection, the best solution is representing IR and machine instruction by DAG. To simplify in view, the register leaf is skipped in Fig. 2.13. The $r_j + r_k$ is IR DAG representation (for symbol notation, not llvm SSA form). ADD is machine instruction.

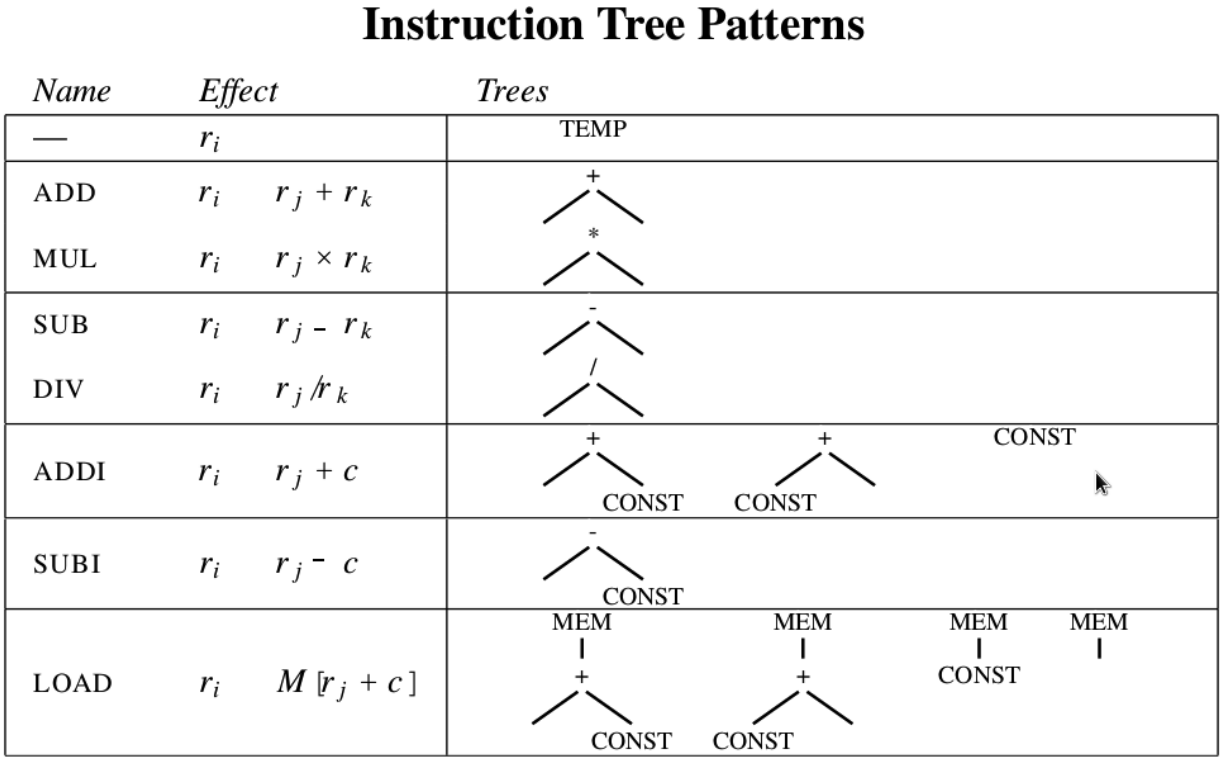


Fig. 2.13: Instruction DAG representation

The IR DAG and machine instruction DAG can also represented as list. For example, (+ r_i, r_jj) and (- r_i, 1) are lists for IR DAG; (ADD r_i, r_j) and (SUBI r_i, 1) are lists for machine instruction DAG.

Now, let's check the ADDiu instruction defined in Cpu0InstrInfo.td as follows,

lbdex/chapters/Chapter2/Cpu0InstrFormats.td

```
//===----------------------------------------------------------------------===//
// Format L instruction class in Cpu0 : <|opcode|ra|rb|cx|>
//===----------------------------------------------------------------------===//

class FL<bits<8> op, dag outs, dag ins, string asmstr, list<dag> pattern,
         InstrItinClass itin>: Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmL>
{
  bits<4>  ra;
  bits<4>  rb;
  bits<16> imm16;

  let Opcode = op;

  let Inst{23-20} = ra;
  let Inst{19-16} = rb;
  let Inst{15-0}  = imm16;
}
```

lbdex/chapters/Chapter2/Cpu0InstrInfo.td

```
// Node immediate fits as 16-bit sign extended on target immediate.
// e.g. addi, andi
def immSExt16  : PatLeaf<(imm), [{ return isInt<16>(N->getSExtValue()); }]>;
```

```
// Arithmetic and logical instructions with 2 register operands.
class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,
                  Operand Od, PatLeaf imm_type, RegisterClass RC> :
  FL<op, (outs GPROut:$ra), (ins RC:$rb, Od:$imm16),
     !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
     [(set GPROut:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {
  let isReMaterializable = 1;
}
```

```
// IR "add" defined in include/llvm/Target/TargetSelectionDAG.td, line 315 (def add).
def ADDiu   : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;
```

Fig. 2.14 shows how the pattern match work in the IR node, **add**, and instruction node, **ADDiu**, which both defined in Cpu0InstrInfo.td. In this example, IR node "add %a, 5" will be translated to "addiu $r1, 5" after %a is allcated to register $r1 in regiter allocation stage since the IR pattern[(set RC:$ra, (OpNode RC:$rb, imm_type:$imm16))] is set in ADDiu and the 2nd operand is "signed immediate" which matched "%a, 5". In addition to pattern match, the .td also set assembly string "addiu" and op code 0x09. With this information, the LLVM TableGen will generate instruction both in assembly and binary automatically (the binary instruction can be issued in obj file of ELF format which will be explained at later chapter). Similarly, the machine instruction DAG nodes LD and ST can be translated from IR DAG nodes **load** and **store**. Notice that the $rb in Fig. 2.14 is virtual register name (not machine register). The detail for Fig. 2.14 is depicted as Fig. 2.15.

From DAG instruction selection we mentioned, the leaf node must be a Data Node. ADDiu is format L type which the last operand must fits in 16 bits range. So, Cpu0InstrInfo.td define a PatLeaf type of immSExt16 to let llvm system know the PatLeaf range. If the imm16 value is out of this range, **"isInt<16>(N->getSExtValue())"** will return false and this pattern won't use ADDiu in instruction selection stage.

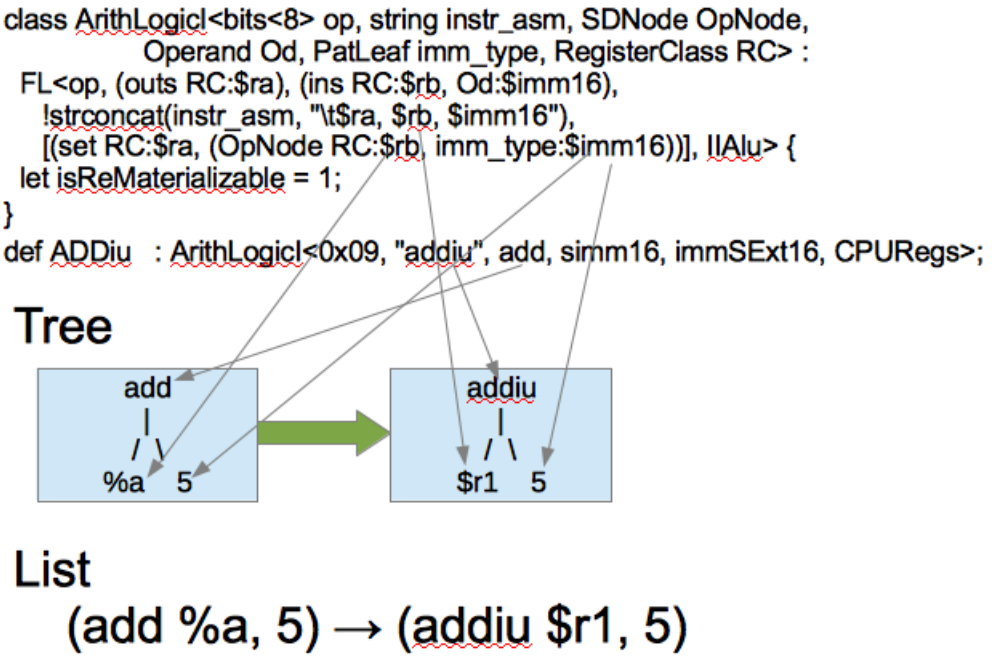


Fig. 2.14: Pattern match for ADDiu instruction and IR node add

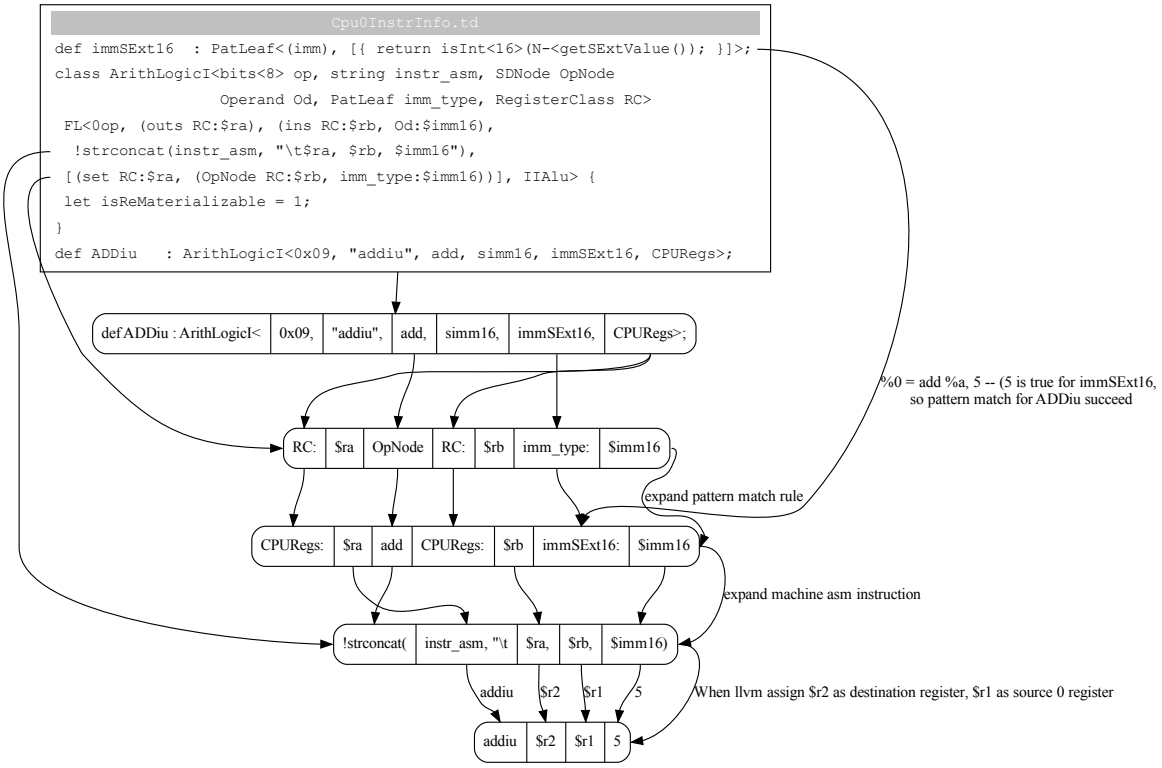


Fig. 2.15: Pattern match for ADDiu instruction and IR node add in detail

Some cpu/fpu (floating point processor) has multiply-and-add floating point instruction, fmadd. It can be represented by DAG list (fadd (fmul ra, rc), rb). For this implementation, we can assign fmadd DAG pattern to instruction td as follows,

```
def FMADDS : AForm_1<59, 29,
          (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRC, F4RC:$FRB),
          "fmadds $FRT, $FRA, $FRC, $FRB",
          [(set F4RC:$FRT, (fadd (fmul F4RC:$FRA, F4RC:$FRC),
                       F4RC:$FRB))]>;
```

Similar with ADDiu, [(set F4RC:$FRT, (fadd (fmul F4RC:$FRA, F4RC:$FRC), F4RC:$FRB))] is the pattern which include nodes **fmul** and **fadd**.

Now, for the following basic block notation IR and llvm SSA IR code,

```
d = a * c
e = d + b
...
```

```
%d = fmul %a, %c
%e = fadd %d, %b
...
```

the Instruction Selection Process will translate this two IR DAG node (fmul %a, %c) (fadd %d, %b) into one machine instruction DAG node (**fmadd** %a, %c, %b), rather than translate them into two machine instruction nodes **fmul** and **fadd** if the FMADDS is appear before FMUL and FADD in your td file.

```
%e = fmadd %a, %c, %b
...
```

As you can see, the IR notation representation is easier to read than llvm SSA IR form. So, this notation form is used in this book sometimes.

For the following basic block code,

```
a = b + c   // in notation IR form
d = a - d
%e = fmadd %a, %c, %b // in llvm SSA IR form
```

We can apply Fig. 2.6 Instruction Tree Patterns to get the following machine code,

```
load  rb, M(sp+8); // assume b allocate in sp+8, sp is stack point register
load  rc, M(sp+16);
add ra, rb, rc;
load  rd, M(sp+24);
sub rd, ra, rd;
fmadd re, ra, rc, rb;
```

2.2.11 Caller and callee saved registers

lbdex/input/ch9_caller_callee_save_registers.cpp

```
extern int add1(int x);

int callee()
{
  int t1 = 3;
  int result = add1(t1);
  result = result - t1;

  return result;
}
```

Run Mips backend with above input will get the following result.

```
JonathantekiiMac:input Jonathan$ ~/llvm/debug/build/bin/llc
-O0 -march=mips -relocation-model=static -filetype=asm
ch9_caller_callee_save_registers.bc -o -
      .text
      .abicalls
      .option pic0
      .section        .mdebug.abi32,"",@progbits
      .nan    legacy
      .file   "ch9_caller_callee_save_registers.bc"
      .text
      .globl  _Z6calleev
      .align  2
      .type   _Z6calleev,@function
      .set    nomicromips
      .set    nomips16
      .ent    _Z6calleev
_Z6callerv:                             # @_Z6callerv
      .cfi_startproc
      .frame  $fp,32,$ra
      .mask   0xc0000000,-4
      .fmask  0x00000000,0
      .set    noreorder
      .set    nomacro
      .set    noat
# BB#0:
      addiu   $sp, $sp, -32
$tmp0:
      .cfi_def_cfa_offset 32
      sw      $ra, 28($sp)            # 4-byte Folded Spill
      sw      $fp, 24($sp)            # 4-byte Folded Spill
$tmp1:
      .cfi_offset 31, -4
$tmp2:
      .cfi_offset 30, -8
      move     $fp, $sp
$tmp3:
```

(continues on next page)

(continued from previous page)

```
      .cfi_def_cfa_register 30
      addiu   $1, $zero, 3
      sw      $1, 20($fp)   # store t1 to 20($fp)
      move     $4, $1
      jal     _Z4add1i
      nop
      sw      $2, 16($fp)   # $2 : the return vaule for fuction add1()
      lw      $1, 20($fp)   # load t1 from 20($fp)
      subu    $1, $2, $1
      sw      $1, 16($fp)
      move     $2, $1     # move result to return register $2
      move     $sp, $fp
      lw      $fp, 24($sp)            # 4-byte Folded Reload
      lw      $ra, 28($sp)            # 4-byte Folded Reload
      addiu   $sp, $sp, 32
      jr      $ra
      nop
      .set    at
      .set    macro
      .set    reorder
      .end    _Z6calleev
$func_end0:
      .size   _Z6calleev, ($func_end0)-_Z6calleev
      .cfi_endproc
```

Caller and callee saved registers definition as follows,

- If the caller wants to use caller-saved registers after callee function, it must save caller-saved registers' content to memory for using and restore these registers from memory after function call.
- If the callee wants to use callee-saved registers, it must save its content to memory before using them and restore these registers from memory before return.

As above definition, if a register is not a callee-saved-registers, then it must be caller-saved-registers because the callee doesn't retore it and the value is changed after callee function. So, Mips only define the callee-saved registers in MipsCallingConv.td, and can be found in CSR_O32_SaveList of MipsGenRgisterInfo.inc for the default ABI.

As above assembly output, Mips allocates t1 variable to register $1 and no need to spill $1 since $1 is caller saved register. On the other hand, $ra is callee saved register, so it spills at beginning of the assembly output since jal uses $ra register. Cpu0 $lr is the same register as Mips $ra, so it calls setAliasRegs(MF, SavedRegs, Cpu0::LR) in determineCalleeSaves() of Cpu0SEFrameLowering.cpp when the function has called another function.

2.2.12 Live in and live out register

As the example of last sub-section. The $ra is "live in" register since the return address is decided by caller. The $2 is "live out" register since the return value of the function is saved in this register, and caller can get the result by read it directly as the comment in above example. Through mark "live in" and "live out" registers, backend provides llvm middle layer information to remove useless instructions in variables access. Of course, llvm applies the DAG analysis mentioned in the previous sub-section to finish it. Since C supports seperate compilation for different functions, the "live in" and "out" information from backend provides the optimization opportunity to llvm. LLVM provides function addLiveIn() to mark "live in" register but no function addLiveOut() provided. For the "live out" register, Mips backend marks it by DAG=DAG.getCopyToReg(..., $2, ...) and return DAG instead, since all local varaiables are not exist after function exit.

2.3 Create Cpu0 backend

From now on, the Cpu0 backend will be created from scratch step by step. To make readers easily understanding the backend structure, Cpu0 example code can be generated chapter by chapter through command here[11]. Cpu0 example code, lbdex, can be found at near left bottom of this web site. Or here http://jonathan2251.github.io/lbd/lbdex.tar.gz.

2.3.1 Cpu0 backend machine ID and relocation records

To create a new backend, there are some files in <<llvm root dir>> need to be modified. The added information include both the ID and name of machine, and relocation records. Chapter "ELF Support" include the relocation records introduction. The following files are modified to add Cpu0 backend as follows,

lbdex/llvm/modify/llvm/config-ix.cmake

```
...
elseif (LLVM_NATIVE_ARCH MATCHES "cpu0")
  set(LLVM_NATIVE_ARCH Cpu0)
...
```

lbdex/llvm/modify/llvm/CMakeLists.txt

```
set(LLVM_ALL_TARGETS
  ...
  Cpu0
  ...
  )
```

lbdex/llvm/modify/llvm/include/llvm/ADT/Triple.h

```
...
#undef mips
#undef cpu0
...
class Triple {
public:
  enum ArchType {
    ...
    cpu0,       // For Tutorial Backend Cpu0
    cpu0el,
    ...
  };
  ...
}
```

[11] http://jonathan2251.github.io/lbd/doc.html#generate-cpu0-document

lbdex/llvm/modify/llvm/include/llvm/Object/ELFObjectFile.h

```
...
template <class ELFT>
StringRef ELFObjectFile<ELFT>::getFileFormatName() const {
  switch (EF.getHeader()->e_ident[ELF::EI_CLASS]) {
  case ELF::ELFCLASS32:
    switch (EF.getHeader()->e_machine) {
    ...
    case ELF::EM_CPU0:        // llvm-objdump -t -r
      return "ELF32-cpu0";
    ...
  }
  ...
}
...
template <class ELFT>
unsigned ELFObjectFile<ELFT>::getArch() const {
  bool IsLittleEndian = ELFT::TargetEndianness == support::little;
  switch (EF.getHeader()->e_machine) {
  ...
  case ELF::EM_CPU0:  // llvm-objdump -t -r
    switch (EF.getHeader()->e_ident[ELF::EI_CLASS]) {
    case ELF::ELFCLASS32:
    return IsLittleEndian ? Triple::cpu0el : Triple::cpu0;
    default:
      report_fatal_error("Invalid ELFCLASS!");
    }
  ...
  }
}
```

lbdex/llvm/modify/llvm/include/llvm/Support/ELF.h

```
enum {
  ...
  EM_CPU0           = 999  // Document LLVM Backend Tutorial Cpu0
};
...
// Cpu0 Specific e_flags
enum {
  EF_CPU0_NOREORDER = 0x00000001, // Don't reorder instructions
  EF_CPU0_PIC       = 0x00000002, // Position independent code
  EF_CPU0_ARCH_32   = 0x50000000, // CPU032 instruction set per linux not elf.h
  EF_CPU0_ARCH      = 0xf0000000  // Mask for applying EF_CPU0_ARCH_ variant
};

// ELF Relocation types for Mips
enum {
#include "ELFRelocs/Cpu0.def"
};
...
```

lbdex/llvm/modify/llvm/lib/MC/MCSubtargetInfo.cpp

```
bool Cpu0DisableUnreconginizedMessage = false;

void MCSubtargetInfo::InitMCProcessorInfo(StringRef CPU, StringRef FS) {
  #if 1 // Disable reconginized processor message. For Cpu0
  if (TargetTriple.getArch() == llvm::Triple::cpu0 ||
      TargetTriple.getArch() == llvm::Triple::cpu0el)
    Cpu0DisableUnreconginizedMessage = true;
  #endif
  ...
}
...
const MCSchedModel &MCSubtargetInfo::getSchedModelForCPU(StringRef CPU) const {
  ...
    #if 1 // Disable reconginized processor message. For Cpu0
    if (TargetTriple.getArch() != llvm::Triple::cpu0 &&
        TargetTriple.getArch() != llvm::Triple::cpu0el)
    #endif
  ...
}
```

lbdex/llvm/modify/llvm/lib/MC/SubtargetFeature.cpp

```
extern bool Cpu0DisableUnreconginizedMessage; // For Cpu0
...
FeatureBitset
SubtargetFeatures::ToggleFeature(FeatureBitset Bits, StringRef Feature,
                                 ArrayRef<SubtargetFeatureKV> FeatureTable) {
  ...
    if (!Cpu0DisableUnreconginizedMessage) // For Cpu0
  ...
}

FeatureBitset
SubtargetFeatures::ApplyFeatureFlag(FeatureBitset Bits, StringRef Feature,
                                    ArrayRef<SubtargetFeatureKV> FeatureTable) {
  ...
    if (!Cpu0DisableUnreconginizedMessage) // For Cpu0
  ...
}

FeatureBitset
SubtargetFeatures::getFeatureBits(StringRef CPU,
                                  ArrayRef<SubtargetFeatureKV> CPUTable,
                                  ArrayRef<SubtargetFeatureKV> FeatureTable) {
  ...
    if (!Cpu0DisableUnreconginizedMessage) // For Cpu0
  ...
}
```

lib/object/ELF.cpp

```
...

StringRef getELFRelocationTypeName(uint32_t Machine, uint32_t Type) {
  switch (Machine) {
  ...
  case ELF::EM_CPU0:
    switch (Type) {
#include "llvm/Support/ELFRelocs/Cpu0.def"
    default:
      break;
    }
    break;
  ...
  }
```

include/llvm/Support/ELFRelocs/Cpu0.def

```
#ifndef ELF_RELOC
#error "ELF_RELOC must be defined"
#endif

ELF_RELOC(R_CPU0_NONE,                0)
ELF_RELOC(R_CPU0_32,                  2)
ELF_RELOC(R_CPU0_HI16,                5)
ELF_RELOC(R_CPU0_LO16,                6)
ELF_RELOC(R_CPU0_GPREL16,             7)
ELF_RELOC(R_CPU0_LITERAL,             8)
ELF_RELOC(R_CPU0_GOT16,               9)
ELF_RELOC(R_CPU0_PC16,               10)
ELF_RELOC(R_CPU0_CALL16,             11)
ELF_RELOC(R_CPU0_GPREL32,            12)
ELF_RELOC(R_CPU0_PC24,               13)
ELF_RELOC(R_CPU0_GOT_HI16,           22)
ELF_RELOC(R_CPU0_GOT_LO16,           23)
ELF_RELOC(R_CPU0_RELGOT,             36)
ELF_RELOC(R_CPU0_TLS_GD,             42)
ELF_RELOC(R_CPU0_TLS_LDM,            43)
ELF_RELOC(R_CPU0_TLS_DTP_HI16,       44)
ELF_RELOC(R_CPU0_TLS_DTP_LO16,       45)
ELF_RELOC(R_CPU0_TLS_GOTTPREL,       46)
ELF_RELOC(R_CPU0_TLS_TPREL32,        47)
ELF_RELOC(R_CPU0_TLS_TP_HI16,        49)
ELF_RELOC(R_CPU0_TLS_TP_LO16,        50)
ELF_RELOC(R_CPU0_GLOB_DAT,           51)
ELF_RELOC(R_CPU0_JUMP_SLOT,         127)
```

lbdex/llvm/modify/llvm/lib/Support/Triple.cpp

```
const char *Triple::getArchTypeName(ArchType Kind) {
  switch (Kind) {
  ...
  case cpu0:        return "cpu0";
  case cpu0el:      return "cpu0el";
  ...
  }
}
...
const char *Triple::getArchTypePrefix(ArchType Kind) {
  switch (Kind) {
  ...
  case cpu0:
  case cpu0el:      return "cpu0";
  ...
}
...
Triple::ArchType Triple::getArchTypeForLLVMName(StringRef Name) {
  return StringSwitch<Triple::ArchType>(Name)
    ...
    .Case("cpu0", cpu0)
    .Case("cpu0el", cpu0el)
    ...
}
...
static Triple::ArchType parseArch(StringRef ArchName) {
  return StringSwitch<Triple::ArchType>(ArchName)
    ...
    .Cases("cpu0", "cpu0eb", "cpu0allegrex", Triple::cpu0)
    .Cases("cpu0el", "cpu0allegrexel", Triple::cpu0el)
    ...
}
...
static Triple::ObjectFormatType getDefaultFormat(const Triple &T) {
  ...
  case Triple::cpu0:
  case Triple::cpu0el:
  ...
}
...
static unsigned getArchPointerBitWidth(llvm::Triple::ArchType Arch) {
  switch (Arch) {
  ...
  case llvm::Triple::cpu0:
  case llvm::Triple::cpu0el:
  ...
    return 32;
  }
}
...
Triple Triple::get32BitArchVariant() const {
```

(continues on next page)

(continued from previous page)

```
  Triple T(*this);
  switch (getArch()) {
  ...
  case Triple::cpu0:
  case Triple::cpu0el:
  ...
    // Already 32-bit.
    break;
  }
  return T;
}
```

2.3.2 Creating the Initial Cpu0 .td Files

As it has been discussed in the previous section, LLVM uses target description files (which uses the .td file extension) to describe various components of a target's backend. For example, these .td files may describe a target's register set, instruction set, scheduling information for instructions, and calling conventions. When your backend is being compiled, the tablegen tool that ships with LLVM will translate these .td files into C++ source code written to files that have a .inc extension. Please refer to[32] for more information regarding how to use tablegen.

Every backend has its own .td to define some target information. These files have a similar syntax to C++. For Cpu0, the target description file is called Cpu0Other.td, which is shown below:

lbdex/chapters/Chapter2/Cpu0Other.td

```
//===-- Cpu0Other.td - Describe the Cpu0 Target Machine ----*- tablegen -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
// This is the top level entry point for the Cpu0 target.
//===----------------------------------------------------------------------===//

//===----------------------------------------------------------------------===//
// Target-independent interfaces
//===----------------------------------------------------------------------===//

include "llvm/Target/Target.td"

//===----------------------------------------------------------------------===//
// Target-dependent interfaces
//===----------------------------------------------------------------------===//

include "Cpu0RegisterInfo.td"
include "Cpu0RegisterInfoGPROutForOther.td" // except AsmParser
```

(continues on next page)

[32] http://llvm.org/docs/TableGen/index.html

(continued from previous page)

```
include "Cpu0.td"
```

Cpu0Other.td and Cpu0.td includes a few other .td files. Cpu0RegisterInfo.td (shown below) describes the Cpu0's set of registers. In this file, we see that each register has been given a name. For example, **"def PC"** indicates that there is a register name as PC. Beside of register information, it also define register class information. You may have multiple register classes such as CPURegs, SR, C0Regs and GPROut. GPROut defined in Cpu0RegisterInfoGPROutForOther.td which include CPURegs except SW, so SW won't be allocated as the output registers in register allocation stage.

lbdex/chapters/Chapter2/Cpu0RegisterInfo.td

```
//===-- Cpu0RegisterInfo.td - Cpu0 Register defs ----------*- tablegen -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

//===----------------------------------------------------------------------===//
//  Declarations that describe the CPU0 register file
//===----------------------------------------------------------------------===//

// We have banks of 16 registers each.
class Cpu0Reg<bits<16> Enc, string n> : Register<n> {
  // For tablegen(... -gen-emitter)  in CMakeLists.txt
  let HWEncoding = Enc;

  let Namespace = "Cpu0";
}

// Cpu0 CPU Registers
class Cpu0GPRReg<bits<16> Enc, string n> : Cpu0Reg<Enc, n>;

// Co-processor 0 Registers
class Cpu0C0Reg<bits<16> Enc, string n> : Cpu0Reg<Enc, n>;

//===----------------------------------------------------------------------===//
//@Registers
//===----------------------------------------------------------------------===//
// The register string, such as "9" or "gp" will show on "llvm-objdump -d"
//@ All registers definition
let Namespace = "Cpu0" in {
  //@ General Purpose Registers
  def ZERO : Cpu0GPRReg<0,  "zero">, DwarfRegNum<[0]>;
  def AT   : Cpu0GPRReg<1,  "1">,    DwarfRegNum<[1]>;
  def V0   : Cpu0GPRReg<2,  "2">,    DwarfRegNum<[2]>;
  def V1   : Cpu0GPRReg<3,  "3">,    DwarfRegNum<[3]>;
  def A0   : Cpu0GPRReg<4,  "4">,    DwarfRegNum<[4]>;
  def A1   : Cpu0GPRReg<5,  "5">,    DwarfRegNum<[5]>;
```

(continues on next page)

(continued from previous page)

```
  def T9   : Cpu0GPRReg<6,  "t9">,   DwarfRegNum<[6]>;
  def T0   : Cpu0GPRReg<7,  "7">,    DwarfRegNum<[7]>;
  def T1   : Cpu0GPRReg<8,  "8">,    DwarfRegNum<[8]>;
  def S0   : Cpu0GPRReg<9,  "9">,    DwarfRegNum<[9]>;
  def S1   : Cpu0GPRReg<10, "10">,   DwarfRegNum<[10]>;
  def GP   : Cpu0GPRReg<11, "gp">,   DwarfRegNum<[11]>;
  def FP   : Cpu0GPRReg<12, "fp">,   DwarfRegNum<[12]>;
  def SP   : Cpu0GPRReg<13, "sp">,   DwarfRegNum<[13]>;
  def LR   : Cpu0GPRReg<14, "lr">,   DwarfRegNum<[14]>;
  def SW   : Cpu0GPRReg<15, "sw">,   DwarfRegNum<[15]>;
//  def MAR  : Register< 16, "mar">,  DwarfRegNum<[16]>;
//  def MDR  : Register< 17, "mdr">,  DwarfRegNum<[17]>;

  def PC   : Cpu0C0Reg<0, "pc">,  DwarfRegNum<[20]>;
  def EPC  : Cpu0C0Reg<1, "epc">, DwarfRegNum<[21]>;
}

//===----------------------------------------------------------------------===//
//@Register Classes
//===----------------------------------------------------------------------===//

def CPURegs : RegisterClass<"Cpu0", [i32], 32, (add
  // Reserved
  ZERO, AT,
  // Return Values and Arguments
  V0, V1, A0, A1,
  // Not preserved across procedure calls
  T9, T0, T1,
  // Callee save
  S0, S1,
  // Reserved
  GP, FP,
  SP, LR, SW)>;

//@Status Registers class
def SR     : RegisterClass<"Cpu0", [i32], 32, (add SW)>;

//@Co-processor 0 Registers class
def C0Regs : RegisterClass<"Cpu0", [i32], 32, (add PC, EPC)>;
```

lbdex/chapters/Chapter2/Cpu0RegisterInfoGPROutForOther.td

```
//===----------------------------------------------------------------------===//
// Register Classes
//===----------------------------------------------------------------------===//

def GPROut : RegisterClass<"Cpu0", [i32], 32, (add (sub CPURegs, SW))>;
```

In C++, class typically provides a structure to lay out some data and functions, while definitions are used to allocate memory for specific instances of a class. For example:

```
class Date {  // declare Date
  int year, month, day;
};
Date birthday;  // define birthday, an instance of Date
```

The class **Date** has the members **year**, **month**, and **day**, but these do not yet belong to an actual object. By defining an instance of **Date** called **birthday**, you have allocated memory for a specific object, and can set the **year**, **month**, and **day** of this instance of the class.

In .td files, class describes the structure of how data is laid out, while definitions act as the specific instances of the class. If you look back at the Cpu0RegisterInfo.td file, you will see a class called **Cpu0Reg** which is derived from the **Register** class provided by LLVM. **Cpu0Reg** inherits all the fields that exist in the **Register** class. The "let HWEncoding = Enc" which means assign field HWEncoding from parameter Enc. Since Cpu0 reserve 4 bits for 16 registers in instruction format, the assigned value range is from 0 to 15. Once assigning the 0 to 15 to HWEncoding, the backend register number will be got from the function of llvm register class since TableGen will set this number automatically.

The **def** keyword is used to create instances of class. In the following line, the ZERO register is defined as a member of the **Cpu0GPRReg** class:

```
def ZERO : Cpu0GPRReg< 0, "ZERO">, DwarfRegNum<[0]>;
```

The **def ZERO** indicates the name of this register. **<0, "ZERO">** are the parameters used when creating this specific instance of the **Cpu0GPRReg** class, thus the field **Enc** is set to 0, and the string **n** is set to **ZERO**.

As the register lives in the **Cpu0** namespace, you can refer to the ZERO register in backend C++ code by using **Cpu0::ZERO**.

Notice the use of the **let** expressions: these allow you to override values that are initially defined in a superclass. For example, **let Namespace = "Cpu0"** in the **Cpu0Reg** class will override the default namespace declared in **Register** class. The Cpu0RegisterInfo.td also defines that **CPURegs** is an instance of the class **RegisterClass**, which is an built-in LLVM class. A **RegisterClass** is a set of **Register** instances, thus **CPURegs** can be described as a set of registers.

The Cpu0 instructions td is named to Cpu0InstrInfo.td which contents as follows,

lbdex/chapters/Chapter2/Cpu0InstrInfo.td

```
//===- Cpu0InstrInfo.td - Target Description for Cpu0 Target -*- tablegen -*-=//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the Cpu0 implementation of the TargetInstrInfo class.
//
//===----------------------------------------------------------------------===//

//===----------------------------------------------------------------------===//
// Cpu0 profiles and nodes
//===----------------------------------------------------------------------===//
```

(continues on next page)

(continued from previous page)

```
def SDT_Cpu0Ret          : SDTypeProfile<0, 1, [SDTCisInt<0>]>;

// Return
def Cpu0Ret : SDNode<"Cpu0ISD::Ret", SDTNone,
                     [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;

//===----------------------------------------------------------------------===//
// Instruction format superclass
//===----------------------------------------------------------------------===//

include "Cpu0InstrFormats.td"

//===----------------------------------------------------------------------===//
// Cpu0 Operand, Complex Patterns and Transformations Definitions.
//===----------------------------------------------------------------------===//
// Instruction operand types

// Signed Operand
def simm16      : Operand<i32> {
  let DecoderMethod= "DecodeSimm16";
}

// Address operand
def mem : Operand<iPTR> {
  let PrintMethod = "printMemOperand";
  let MIOperandInfo = (ops GPROut, simm16);
  let EncoderMethod = "getMemEncoding";
}

// Node immediate fits as 16-bit sign extended on target immediate.
// e.g. addi, andi
def immSExt16  : PatLeaf<(imm), [{ return isInt<16>(N->getSExtValue()); }]>;

// Cpu0 Address Mode! SDNode frameindex could possibily be a match
// since load and store instructions from stack used it.
def addr :
  ComplexPattern<iPTR, 2, "SelectAddr", [frameindex], [SDNPWantParent]>;

//===----------------------------------------------------------------------===//
// Pattern fragment for load/store
//===----------------------------------------------------------------------===//

class AlignedLoad<PatFrag Node> :
  PatFrag<(ops node:$ptr), (Node node:$ptr), [{
  LoadSDNode *LD = cast<LoadSDNode>(N);
  return LD->getMemoryVT().getSizeInBits()/8 <= LD->getAlignment();
}]>;

class AlignedStore<PatFrag Node> :
  PatFrag<(ops node:$val, node:$ptr), (Node node:$val, node:$ptr), [{
  StoreSDNode *SD = cast<StoreSDNode>(N);
```

(continues on next page)

(continued from previous page)

```
  return SD->getMemoryVT().getSizeInBits()/8 <= SD->getAlignment();
}]>;

// Load/Store PatFrags.
def load_a          : AlignedLoad<load>;
def store_a         : AlignedStore<store>;

//===----------------------------------------------------------------------===//
// Instructions specific format
//===----------------------------------------------------------------------===//

// Arithmetic and logical instructions with 2 register operands.
class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,
                  Operand Od, PatLeaf imm_type, RegisterClass RC> :
  FL<op, (outs GPROut:$ra), (ins RC:$rb, Od:$imm16),
     !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
     [(set GPROut:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {
  let isReMaterializable = 1;
}

class FMem<bits<8> op, dag outs, dag ins, string asmstr, list<dag> pattern,
           InstrItinClass itin>: FL<op, outs, ins, asmstr, pattern, itin> {
  bits<20> addr;
  let Inst{19-16} = addr{19-16};
  let Inst{15-0}  = addr{15-0};
  let DecoderMethod = "DecodeMem";
}

// Memory Load/Store
let canFoldAsLoad = 1 in
class LoadM<bits<8> op, string instr_asm, PatFrag OpNode, RegisterClass RC,
            Operand MemOpnd, bit Pseudo>:
  FMem<op, (outs RC:$ra), (ins MemOpnd:$addr),
     !strconcat(instr_asm, "\t$ra, $addr"),
     [(set RC:$ra, (OpNode addr:$addr))], IILoad> {
  let isPseudo = Pseudo;
}

class StoreM<bits<8> op, string instr_asm, PatFrag OpNode, RegisterClass RC,
             Operand MemOpnd, bit Pseudo>:
  FMem<op, (outs), (ins RC:$ra, MemOpnd:$addr),
     !strconcat(instr_asm, "\t$ra, $addr"),
     [(OpNode RC:$ra, addr:$addr)], IIStore> {
  let isPseudo = Pseudo;
}

//@ 32-bit load.
class LoadM32<bits<8> op, string instr_asm, PatFrag OpNode,
                   bit Pseudo = 0>
  : LoadM<op, instr_asm, OpNode, GPROut, mem, Pseudo> {
}
```

(continues on next page)

(continued from previous page)

```
// 32-bit store.
class StoreM32<bits<8> op, string instr_asm, PatFrag OpNode,
                    bit Pseudo = 0>
  : StoreM<op, instr_asm, OpNode, GPROut, mem, Pseudo> {
}

//@JumpFR {
let isBranch=1, isTerminator=1, isBarrier=1, imm16=0, hasDelaySlot = 1,
    isIndirectBranch = 1 in
class JumpFR<bits<8> op, string instr_asm, RegisterClass RC>:
  FL<op, (outs), (ins RC:$ra),
     !strconcat(instr_asm, "\t$ra"), [(brind RC:$ra)], IIBranch> {
  let rb = 0;
  let imm16 = 0;
}
//@JumpFR }

// Return instruction
class RetBase<RegisterClass RC>: JumpFR<0x3c, "ret", RC> {
  let isReturn = 1;
  let isCodeGenOnly = 1;
  let hasCtrlDep = 1;
  let hasExtraSrcRegAllocReq = 1;
}


//===----------------------------------------------------------------------===//
// Instruction definition
//===----------------------------------------------------------------------===//

//===----------------------------------------------------------------------===//
// Cpu0 Instructions
//===----------------------------------------------------------------------===//

/// Load and Store Instructions
///  aligned
def LD     : LoadM32<0x01,  "ld",  load_a>;
def ST     : StoreM32<0x02, "st",  store_a>;

/// Arithmetic Instructions (ALU Immediate)
// IR "add" defined in include/llvm/Target/TargetSelectionDAG.td, line 315 (def add).
def ADDiu   : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;

/// Arithmetic Instructions (3-Operand, R-Type)

/// Shift Instructions

def JR      : JumpFR<0x3c, "jr", GPROut>;

def RET     : RetBase<GPROut>;

/// No operation
```

(continues on next page)

(continued from previous page)

```
let addr=0 in
  def NOP   : FJ<0, (outs), (ins), "nop", [], IIAlu>;

//===----------------------------------------------------------------------===//
//  Arbitrary patterns that map to one or more instructions
//===----------------------------------------------------------------------===//

// Small immediates
def : Pat<(i32 immSExt16:$in),
          (ADDiu ZERO, imm:$in)>;
```

The Cpu0InstrFormats.td is included by Cpu0InstInfo.td as follows,

lbdex/chapters/Chapter2/Cpu0InstrFormats.td

```
//===-- Cpu0InstrFormats.td - Cpu0 Instruction Formats -----*- tablegen -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

//===----------------------------------------------------------------------===//
//  Describe CPU0 instructions format
//
//  CPU INSTRUCTION FORMATS
//
//  opcode  - operation code.
//  ra      - dst reg, only used on 3 regs instr.
//  rb      - src reg.
//  rc      - src reg (on a 3 reg instr).
//  cx      - immediate
//
//===----------------------------------------------------------------------===//

// Format specifies the encoding used by the instruction.  This is part of the
// ad-hoc solution used to emit machine instruction encodings by our machine
// code emitter.
class Format<bits<4> val> {
  bits<4> Value = val;
}

def Pseudo    : Format<0>;
def FrmA      : Format<1>;
def FrmL      : Format<2>;
def FrmJ      : Format<3>;
def FrmOther  : Format<4>; // Instruction w/ a custom format
```

(continues on next page)

(continued from previous page)

```
// Generic Cpu0 Format
class Cpu0Inst<dag outs, dag ins, string asmstr, list<dag> pattern,
               InstrItinClass itin, Format f>: Instruction
{
  // Inst and Size: for tablegen(... -gen-emitter) and
  // tablegen(... -gen-disassembler) in CMakeLists.txt
  field bits<32> Inst;
  Format Form = f;

  let Namespace = "Cpu0";

  let Size = 4;

  bits<8> Opcode = 0;

  // Top 8 bits are the 'opcode' field
  let Inst{31-24} = Opcode;

  let OutOperandList = outs;
  let InOperandList  = ins;

  let AsmString   = asmstr;
  let Pattern     = pattern;
  let Itinerary   = itin;

  //
  // Attributes specific to Cpu0 instructions...
  //
  bits<4> FormBits = Form.Value;

  // TSFlags layout should be kept in sync with Cpu0InstrInfo.h.
  let TSFlags{3-0}   = FormBits;

  let DecoderNamespace = "Cpu0";

  field bits<32> SoftFail = 0;
}

//===----------------------------------------------------------------------===//
// Format A instruction class in Cpu0 : <|opcode|ra|rb|rc|cx|>
//===----------------------------------------------------------------------===//

class FA<bits<8> op, dag outs, dag ins, string asmstr,
         list<dag> pattern, InstrItinClass itin>:
      Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmA>
{
  bits<4>  ra;
  bits<4>  rb;
  bits<4>  rc;
  bits<12> shamt;

  let Opcode = op;
```

(continues on next page)

(continued from previous page)

```
  let Inst{23-20} = ra;
  let Inst{19-16} = rb;
  let Inst{15-12} = rc;
  let Inst{11-0}  = shamt;
}

//@class FL {
//===----------------------------------------------------------------------===//
// Format L instruction class in Cpu0 : <|opcode|ra|rb|cx|>
//===----------------------------------------------------------------------===//

class FL<bits<8> op, dag outs, dag ins, string asmstr, list<dag> pattern,
         InstrItinClass itin>: Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmL>
{
  bits<4>  ra;
  bits<4>  rb;
  bits<16> imm16;

  let Opcode = op;

  let Inst{23-20} = ra;
  let Inst{19-16} = rb;
  let Inst{15-0}  = imm16;
}
//@class FL }

//===----------------------------------------------------------------------===//
// Format J instruction class in Cpu0 : <|opcode|address|>
//===----------------------------------------------------------------------===//

class FJ<bits<8> op, dag outs, dag ins, string asmstr, list<dag> pattern,
         InstrItinClass itin>: Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmJ>
{
  bits<24> addr;

  let Opcode = op;

  let Inst{23-0} = addr;
}
```

ADDiu is a instance of class ArithLogicI inherited from FL, it can be expanded and get member value further as follows,

```
def ADDiu   : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;

/// Arithmetic and logical instructions with 2 register operands.
class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,
          Operand Od, PatLeaf imm_type, RegisterClass RC> :
  FL<op, (outs GPROut:$ra), (ins RC:$rb, Od:$imm16),
   !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
   [(set GPROut:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {
```

(continues on next page)

(continued from previous page)

```
  let isReMaterializable = 1;
}
```

So,

```
op = 0x09
instr_asm = "addiu"
OpNode = add
Od = simm16
imm_type = immSExt16
RC = CPURegs
```

To expand the td, some principles are:

- let: meaning override the existed field from parent class.

  For instance: let isReMaterializable = 1; override the isReMaterializable from class instruction of Target.td.

- declaration: meaning declare a new field for this class.

  For instance: bits<4> ra; declare ra field for class FL.

The details of expanding as the following table:

Table 2.12: ADDiu expand part I

ADDiu	ArithLogicI	FL
0x09	op = 0x09	Opcode = 0x09;
addiu	instr_asm = "addiu"	(outs GPROut:$ra); !strconcat("addiu", "t$ra, $rb, $imm16");
add	OpNode = add	[(set GPROut:$ra, (add CPURegs:$rb, immSExt16:$imm16))]
simm16	Od = simm16	(ins CPURegs:$rb, simm16:$imm16);
immSExt16	imm_type = immSExt16	Inst{15-0} = imm16;
CPURegs	RC = CPURegs isReMaterializable=1;	Inst{23-20} = ra; Inst{19-16} = rb;

Table 2.13: ADDiu expand part II

Cpu0Inst	instruction
Namespace = "Cpu0"	Uses = []; ...
Inst{31-24} = 0x09;	Size = 0; ...
OutOperandList = GPROut:$ra;	
InOperandList = CPURegs:$rb,simm16:$imm16;	
AsmString = "addiut$ra, $rb, $imm16"	
pattern = [(set GPROut:$ra, (add RC:$rb, immSExt16:$imm16))]	
Itinerary = IIAlu	
TSFlags{3-0} = FrmL.value	
DecoderNamespace = "Cpu0"	

The td expanding is a lousy process. Similarly, LD and ST instruction definition can be expanded in this way. Please notice the Pattern = [(set GPROut:$ra, (add RC:$rb, immSExt16:$imm16))] which include keyword **"add"**. The ADDiu with **"add"** is used in sub-section Instruction Selection of last section.

File Cpu0Schedule.td include the function units and pipeline stages information as follows,

lbdex/chapters/Chapter2/Cpu0Schedule.td

```
//===-- Cpu0Schedule.td - Cpu0 Scheduling Definitions ------*- tablegen -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

//===----------------------------------------------------------------------===//
// Functional units across Cpu0 chips sets. Based on GCC/Cpu0 backend files.
//===----------------------------------------------------------------------===//
def ALU     : FuncUnit;
def IMULDIV : FuncUnit;

//===----------------------------------------------------------------------===//
// Instruction Itinerary classes used for Cpu0
//===----------------------------------------------------------------------===//
def IIAlu              : InstrItinClass;
def II_CLO             : InstrItinClass;
def II_CLZ             : InstrItinClass;
def IILoad             : InstrItinClass;
def IIStore            : InstrItinClass;
def IIBranch           : InstrItinClass;

def IIPseudo           : InstrItinClass;

//===----------------------------------------------------------------------===//
// Cpu0 Generic instruction itineraries.
//===----------------------------------------------------------------------===//
//@ http://llvm.org/docs/doxygen/html/structllvm_1_1InstrStage.html
def Cpu0GenericItineraries : ProcessorItineraries<[ALU, IMULDIV], [], [
//@2
  InstrItinData<IIAlu              , [InstrStage<1,  [ALU]>]>,
  InstrItinData<II_CLO             , [InstrStage<1,  [ALU]>]>,
  InstrItinData<II_CLZ             , [InstrStage<1,  [ALU]>]>,
  InstrItinData<IILoad             , [InstrStage<3,  [ALU]>]>,
  InstrItinData<IIStore            , [InstrStage<1,  [ALU]>]>,
  InstrItinData<IIBranch           , [InstrStage<1,  [ALU]>]>
]>;
```

2.3.3 Write cmake file

Target/Cpu0 directory has two files CMakeLists.txt, contents as follows,

lbdex/chapters/Chapter2/CMakeLists.txt

```
add_llvm_component_group(Cpu0)

set(LLVM_TARGET_DEFINITIONS Cpu0Other.td)

# Generate Cpu0GenRegisterInfo.inc and Cpu0GenInstrInfo.inc which included by
#  your hand code C++ files.
# Cpu0GenRegisterInfo.inc came from Cpu0RegisterInfo.td, Cpu0GenInstrInfo.inc
#  came from Cpu0InstrInfo.td.
tablegen(LLVM Cpu0GenRegisterInfo.inc -gen-register-info)
tablegen(LLVM Cpu0GenInstrInfo.inc -gen-instr-info)
tablegen(LLVM Cpu0GenSubtargetInfo.inc -gen-subtarget)
tablegen(LLVM Cpu0GenMCPseudoLowering.inc -gen-pseudo-lowering)

# Cpu0CommonTableGen must be defined
add_public_tablegen_target(Cpu0CommonTableGen)

# Cpu0CodeGen should match with LLVMBuild.txt Cpu0CodeGen
add_llvm_target(Cpu0CodeGen
  Cpu0TargetMachine.cpp

  LINK_COMPONENTS
  Analysis
  AsmPrinter
  CodeGen
  Core
  MC
  Cpu0Desc
  Cpu0Info
  SelectionDAG
  Support
  Target
  GlobalISel

  ADD_TO_COMPONENT
  Cpu0
  )

# Should match with "subdirectories =  MCTargetDesc TargetInfo" in LLVMBuild.txt
add_subdirectory(TargetInfo)
add_subdirectory(MCTargetDesc)
```

CMakeLists.txt is the make information for cmake and # is comment. Comments are prefixed by # in both files. The "tablegen(" in above CMakeLists.txt is defined in cmake/modules/TableGen.cmake as below,

llvm/cmake/modules/TableGen.cmake

```
function(tablegen project ofn)
  ...
  add_custom_command(OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/${ofn}.tmp
    # Generate tablegen output in a temporary file.
    COMMAND ${${project}_TABLEGEN_EXE} ${ARGN} -I ${CMAKE_CURRENT_SOURCE_DIR}
  ...
endfunction()
...
macro(add_tablegen target project)
  ...
  if(LLVM_USE_HOST_TOOLS)
    if( ${${project}_TABLEGEN} STREQUAL "${target}" )
      if (NOT CMAKE_CONFIGURATION_TYPES)
        set(${project}_TABLEGEN_EXE "${LLVM_NATIVE_BUILD}/bin/${target}")
      else()
        set(${project}_TABLEGEN_EXE "${LLVM_NATIVE_BUILD}/Release/bin/${target}")
      endif()
  ...
endmacro()
```

llvm/utils/TableGen/CMakeLists.txt

```
add_tablegen(llvm-tblgen LLVM
  ...
)
```

Above "add_tablegen" in llvm/utils/TableGen/CMakeLists.txt makes the "tablegen(" written in Cpu0 CMakeLists.txt an alias of llvm-tblgen (${project} = LLVM and ${project}_TABLEGEN_EXE = llvm-tblgen). The "tablegen(", "add_public_tablegen_target(Cpu0CommonTableGen)" in lbdex/chapters/Chapter2/CMakeLists.txt and the following code define a target "Cpu0CommonTableGen" with it's output files "Cpu0Gen*.inc" as follows,

llvm/cmake/modules/TableGen.cmake

```
function(tablegen project ofn)
  ...
  set(TABLEGEN_OUTPUT ${TABLEGEN_OUTPUT} ${CMAKE_CURRENT_BINARY_DIR}/${ofn} PARENT_SCOPE)
  ...
endfunction()

# Creates a target for publicly exporting tablegen dependencies.
function(add_public_tablegen_target target)
  ...
  add_custom_target(${target}
    DEPENDS ${TABLEGEN_OUTPUT})
  ...
endfunction()
```

Since execution file llvm-tblgen is built before compiling any llvm backend source code during building llvm, the llvm-tblgen is always ready for backend's TableGen reguest.

This book breaks the whole backend source code by function, add code chapter by chapter. Don't try to understand everything in the text of book, the code added in each chapter is a reading material too. To understand the computer related knowledge in concept, you can ignore source code, but implementing based on an existed open software cannot. In programming, documentation cannot replace the source code totally. Reading source code is a big opportunity in the open source development.

CMakeLists.txt exists in sub-directories **MCTargetDesc** and **TargetInfo**. The contents of MakeLists.txt in these two directories instruct llvm generating Cpu0Desc and Cpu0Info libraries, repectively. After building, you will find three libraries: **libLLVMCpu0CodeGen.a**, **libLLVMCpu0Desc.a** and **libLLVMCpu0Info.a** in lib/ of your build directory. For more details please see "Building LLVM with CMake"[28].

2.3.4 Target Registration

You must also register your target with the TargetRegistry. After registration, llvm tools are able to lookup and use your target at runtime. The TargetRegistry can be used directly, but for most targets there are helper templates which should take care of the work for you.

All targets should declare a global Target object which is used to represent the target during registration. Then, in the target's TargetInfo library, the target should define that object and use the RegisterTarget template to register the target. For example, the file TargetInfo/Cpu0TargetInfo.cpp register TheCpu0Target for big endian and TheCpu0elTarget for little endian, as follows.

lbdex/chapters/Chapter2/Cpu0.h

```
//===-- Cpu0.h - Top-level interface for Cpu0 representation ----*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the entry points for global functions defined in
// the LLVM Cpu0 back-end.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0_H
#define LLVM_LIB_TARGET_CPU0_CPU0_H

#include "Cpu0Config.h"
#include "MCTargetDesc/Cpu0MCTargetDesc.h"
#include "llvm/Target/TargetMachine.h"

namespace llvm {
  class Cpu0TargetMachine;
  class FunctionPass;

} // end namespace llvm;
```

(continues on next page)

[28] http://llvm.org/docs/CMake.html

(continued from previous page)

```
#endif
```

lbdex/chapters/Chapter2/TargetInfo/Cpu0TargetInfo.cpp

```
//===-- Cpu0TargetInfo.cpp - Cpu0 Target Implementation ---------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

#include "Cpu0.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/TargetRegistry.h"
using namespace llvm;

Target llvm::TheCpu0Target, llvm::TheCpu0elTarget;

extern "C" void LLVMInitializeCpu0TargetInfo() {
  RegisterTarget<Triple::cpu0,
        /*HasJIT=*/true> X(TheCpu0Target, "cpu0", "CPU0 (32-bit big endian)", "Cpu0");

  RegisterTarget<Triple::cpu0el,
        /*HasJIT=*/true> Y(TheCpu0elTarget, "cpu0el", "CPU0 (32-bit little endian)",
↪"Cpu0");
}
```

lbdex/chapters/Chapter2/TargetInfo/CMakeLists.txt

```
# llvm 10.0.0 change from add_llvm_library to add_llvm_component_library
add_llvm_component_library(LLVMCpu0Info
  Cpu0TargetInfo.cpp

  LINK_COMPONENTS
  Support

  ADD_TO_COMPONENT
  Cpu0
  )
```

Files Cpu0TargetMachine.cpp and MCTargetDesc/Cpu0MCTargetDesc.cpp just define the empty initialize function since we register nothing for this moment.

lbdex/chapters/Chapter2/Cpu0TargetMachine.cpp

```
//===-- Cpu0TargetMachine.cpp - Define TargetMachine for Cpu0 -------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// Implements the info about Cpu0 target spec.
//
//===----------------------------------------------------------------------===//

#include "Cpu0TargetMachine.h"
#include "Cpu0.h"

#include "llvm/IR/Attributes.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/CodeGen.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/CodeGen/TargetPassConfig.h"
#include "llvm/Support/TargetRegistry.h"
#include "llvm/Target/TargetOptions.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0"

extern "C" void LLVMInitializeCpu0Target() {
}
```

lbdex/chapters/Chapter2/MCTargetDesc/Cpu0MCTargetDesc.h

```
//===-- Cpu0MCTargetDesc.h - Cpu0 Target Descriptions -----------*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file provides Cpu0 specific target descriptions.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCTARGETDESC_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCTARGETDESC_H
```

(continues on next page)

(continued from previous page)

```
#include "Cpu0Config.h"
#include "llvm/Support/DataTypes.h"

#include <memory>

namespace llvm {
class Target;
class Triple;

extern Target TheCpu0Target;
extern Target TheCpu0elTarget;

} // End llvm namespace

// Defines symbolic names for Cpu0 registers.  This defines a mapping from
// register name to register number.
#define GET_REGINFO_ENUM
#include "Cpu0GenRegisterInfo.inc"

// Defines symbolic names for the Cpu0 instructions.
#define GET_INSTRINFO_ENUM
#include "Cpu0GenInstrInfo.inc"

#define GET_SUBTARGETINFO_ENUM
#include "Cpu0GenSubtargetInfo.inc"

#endif
```

lbdex/chapters/Chapter2/MCTargetDesc/Cpu0MCTargetDesc.cpp

```
//===-- Cpu0MCTargetDesc.cpp - Cpu0 Target Descriptions -------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file provides Cpu0 specific target descriptions.
//
//===----------------------------------------------------------------------===//

#include "Cpu0MCTargetDesc.h"
#include "llvm/MC/MachineLocation.h"
#include "llvm/MC/MCELFStreamer.h"
#include "llvm/MC/MCInstrAnalysis.h"
#include "llvm/MC/MCInstPrinter.h"
```

(continues on next page)

(continued from previous page)

```
#include "llvm/MC/MCInstrInfo.h"
#include "llvm/MC/MCRegisterInfo.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/MC/MCSymbol.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/FormattedStream.h"
#include "llvm/Support/TargetRegistry.h"

using namespace llvm;

#define GET_INSTRINFO_MC_DESC
#include "Cpu0GenInstrInfo.inc"

#define GET_SUBTARGETINFO_MC_DESC
#include "Cpu0GenSubtargetInfo.inc"

#define GET_REGINFO_MC_DESC
#include "Cpu0GenRegisterInfo.inc"

//@2 {
extern "C" void LLVMInitializeCpu0TargetMC() {

}
//@2 }
```

lbdex/chapters/Chapter2/MCTargetDesc/CMakeLists.txt

```
# MCTargetDesc/CMakeLists.txt
add_llvm_component_library(LLVMCpu0Desc
  Cpu0MCTargetDesc.cpp

  LINK_COMPONENTS
  MC
  Cpu0Info
  Support

  ADD_TO_COMPONENT
  Cpu0
  )
```

Please see "Target Registration"[29] for reference.

[29] http://llvm.org/docs/WritingAnLLVMBackend.html#target-registration

2.3.5 Build libraries and td

Build steps https://github.com/Jonathan2251/lbd/blob/master/README.md. We set llvm source code in /Users/Jonathan/llvm/debug/llvm and have llvm debug-build in /Users/Jonathan/llvm/debug/build. About how to build llvm, please refer here[30]. In appendix A, we made a copy from /Users/Jonathan/llvm/debug/llvm to /Users/Jonathan/llvm/test/llvm for working with my Cpu0 target backend. Sub-directories llvm is for source code and build is for debug build directory.

Beside directory llvm/lib/Target/Cpu0, there are a couple of files modified to support cpu0 new Target, which includes both the ID and name of machine and relocation records listed in the early sub-section. You can update your llvm working copy and find the modified files by commands, cp -rf lbdex/llvm/modify/llvm/* <yourllvm/workingcopy/sourcedir>/.

```
118-165-78-230:lbd Jonathan$ pwd
/Users/Jonathan/git/lbd
118-165-78-230:lbd Jonathan$ cp -rf lbdex/llvm/modify/llvm/* ~/llvm/test/llvm/.
118-165-78-230:lbd Jonathan$ grep -R "cpu0" ~/llvm/test/llvm/include
llvm/cmake/config-ix.cmake:elseif (LLVM_NATIVE_ARCH MATCHES "cpu0")
llvm/include/llvm/ADT/Triple.h:#undef cpu0
llvm/include/llvm/ADT/Triple.h:    cpu0,       // For Tutorial Backend Cpu0
llvm/include/llvm/ADT/Triple.h:    cpu0el,
llvm/include/llvm/Support/ELF.h:  EF_CPU0_ARCH_32R2 = 0x70000000, // cpu032r2
llvm/include/llvm/Support/ELF.h:  EF_CPU0_ARCH_64R2 = 0x80000000, // cpu064r2
...
```

Next configure the Cpu0 example code to chapter2 as follows,

~/llvm/test/llvm/lib/Target/Cpu0/Cpu0SetChapter.h

```
#define CH       CH2
```

Beside configure chapter as above, I provide gen-chapters.sh that you can get each chapter code as follows,

```
118-165-78-230:lbdex Jonathan$ pwd
/Users/Jonathan/git/lbd/lbdex
118-165-78-230:lbdex Jonathan$ bash gen-chapters.sh
118-165-78-230:lbdex Jonathan$ ls chapters
Chapter10_1   Chapter11_2     Chapter2        Chapter3_2...
Chapter11_1   Chapter12_1     Chapter3_1      Chapter3_3...
```

Now, run the `cmake` and `make` command to build td (the following cmake command is for my setting),

```
118-165-78-230:build Jonathan$ cmake -DCMAKE_CXX_COMPILER=clang++
-DCMAKE_C_COMPILER=clang -DCMAKE_BUILD_TYPE=Debug -G "Unix Makefiles" ../llvm/

-- Targeting Cpu0
...
-- Targeting XCore
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/Jonathan/llvm/test/build
```

(continues on next page)

[30] http://clang.llvm.org/get_started.html

(continued from previous page)

```
118-165-78-230:build Jonathan$ make -j4

118-165-78-230:build Jonathan$
```

After build, you can type command `llc -version` to find the cpu0 backend,

```
118-165-78-230:build Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc --version
LLVM (http://llvm.org/):
...
  Registered Targets:
  arm      - ARM
  ...
  cpp      - C++ backend
  cpu0     - Cpu0
  cpu0el   - Cpu0el
...
```

The `llc -version` can display Registered Targets **"cpu0"** and **"cpu0el"**, because the code in file TargetInfo/Cpu0TargetInfo.cpp we made in last sub-section "Target Registration"[31].

Let's build lbdex/chapters/Chapter2 code as follows,

```
118-165-75-57:test Jonathan$ pwd
/Users/Jonathan/test
118-165-75-57:test Jonathan$ cp -rf lbdex/Cpu0 ~/llvm/test/llvm/lib/Target/.

118-165-75-57:test Jonathan$ cd ~/llvm/test/build
118-165-75-57:build Jonathan$ pwd
/Users/Jonathan/llvm/test/build
118-165-75-57:build Jonathan$ rm -rf *
118-165-75-57:build Jonathan$ cmake -DCMAKE_CXX_COMPILER=clang++
-DCMAKE_C_COMPILER=clang -DCMAKE_BUILD_TYPE=Debug -DLLVM_TARGETS_TO_BUILD=Cpu0
-G "Unix Makefiles" ../llvm/
...
-- Targeting Cpu0
...
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/Jonathan/llvm/test/build
```

In order to save time, we build Cpu0 target only by option -DLLVM_TARGETS_TO_BUILD=Cpu0. After that, you can find the *.inc files in directory /Users/Jonathan/llvm/test/build/lib/Target/Cpu0 as follows,

[31] http://jonathan2251.github.io/lbd/llvmstructure.html#target-registration

build/lib/Target/Cpu0/Cpu0GenRegisterInfo.inc

```
namespace Cpu0 {
enum {
  NoRegister,
  AT = 1,
  EPC = 2,
  FP = 3,
  GP = 4,
  HI = 5,
  LO = 6,
  LR = 7,
  PC = 8,
  SP = 9,
  SW = 10,
  ZERO = 11,
  A0 = 12,
  A1 = 13,
  S0 = 14,
  S1 = 15,
  T0 = 16,
  T1 = 17,
  T9 = 18,
  V0 = 19,
  V1 = 20,
  NUM_TARGET_REGS     // 21
};
}
...
```

These *.inc are generated by llvm-tblgen at directory build/lib/Target/Cpu0 where their input files are the Cpu0 backend *.td files. The llvm-tblgen is invoked by **tablegen** of /Users/Jonathan/llvm/test/llvm/lib/Target/Cpu0/CMakeLists.txt. These *.inc files will be included by Cpu0 backend *.cpp or *.h files and compile into *.o further. TableGen is the important tool illustrated in the early sub-section ".td: LLVM's Target Description Files" of this chapter. List it again as follows,

"The "mix and match" approach allows target authors to choose what makes sense for their architecture and permits a large amount of code reuse across different targets".

Details about TableGen are here[Page 54, 323334].

Now try to run command `llc` to compile input file ch3.cpp as follows,

[33] http://llvm.org/docs/TableGen/LangIntro.html
[34] http://llvm.org/docs/TableGen/LangRef.html

lbdex/input/ch3.cpp

```
int main()
{
  return 0;
}
```

First step, compile it with clang and get output ch3.bc as follows,

```
118-165-78-230:input Jonathan$ pwd
/Users/Jonathan/git/lbd/lbdex/input
118-165-78-230:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch3.cpp -emit-llvm -o ch3.bc
```

As above, compile C to .bc by `clang -target mips-unknown-linux-gnu` because Cpu0 borrows the ABI from Mips. Next step, transfer bitcode .bc to human readable text format as follows,

```
118-165-78-230:test Jonathan$ llvm-dis ch3.bc -o -

// ch3.ll
; ModuleID = 'ch3.bc'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f3
2:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:6
4-S128"
target triple = "mips-unknown-linux-gnu"

define i32 @main() nounwind uwtable {
  %1 = alloca i32, align 4
  store i32 0, i32* %1
  ret i32 0
}
```

Now, when compiling ch3.bc will get the error message as follows,

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
...
... Assertion `target.get() && "Could not allocate target machine!"' failed
...
```

At this point, we finish the Target Registration for Cpu0 backend. The backend compiler command `llc` can recognize Cpu0 backend now. Currently we just define target td files (Cpu0.td, Cpu0Other.td, Cpu0RegisterInfo.td, ...). According to LLVM structure, we need to define our target machine and include those td related files. The error message says we didn't define our target machine. This book is a step-by-step backend delvelopment. You can review the houndreds lines of Chapter2 example code to see how to do the Target Registration.

2.4 Options of llc for debug

llc –help-hidden

The following options for llc need to give a input .bc or .ll file.

- -debug:
- -debug-pass=Structure
- -print-after-all, -print-before-all
- -print-before="pass" and -print-after="pass", eg. -print-before="postra-machine-sink" and -print-after="postra-machine-sink". The pass name can be got as follows,

```
CodeGen % pwd
~/llvm/debug/llvm/lib/CodeGen
CodeGen % grep -R "INITIALIZE_PASS" |grep sink
./MachineSink.cpp:INITIALIZE_PASS(PostRAMachineSinking, "postra-machine-sink",
```

- -view-dag-combine1-dags displays the DAG after being built, before the first optimization pass.
- -view-legalize-dags displays the DAG before Legalization.
- -view-dag-combine2-dags displays the DAG before the second optimization pass.
- -view-isel-dags displays the DAG before the Select phase.
- -view-sched-dags displays the DAG before Scheduling.
- -march=<string>, eg. march=mips;
- -relocation-model=static/pic
- -filetype=asm/obj

Use F.dump() in code where F is class Function for passes in llvm/lib/Transformation.

2.5 Options of opt

Check from *opt –help-hidden* and LLVM passes[35]. Eg.

- *opt -dot-cfg input.ll*: Print CFG of function to 'dot' file
- -dot-cfg-only : Print CFG of function to 'dot' file (with no function bodies)

[35] https://llvm.org/docs/Passes.html

CHAPTER

THREE

BACKEND STRUCTURE

- *TargetMachine structure*
- *Add AsmPrinter*
- *Add Cpu0DAGToDAGISel class*
- *Handle return register $lr*
- *Add Prologue/Epilogue functions*
  - *Concept*
  - *Prologue and Epilogue functions*
  - *Handle stack slot for local variables*
  - *Large stack*
- *Data operands DAGs*
- *Summary of this Chapter*

This chapter introduces the backend class inheritance tree and class members first. Next, following the backend structure, adding individual classes implementation in each section. At the end of this chapter, we will have a backend to compile llvm intermediate code into Cpu0 assembly code.

Many lines of code are added in this chapter. They almost are common in every backend except the backend name (Cpu0 or Mips ...). Actually, we copy almost all the code from Mips and replace the name with Cpu0. In addition to knowing the DAGs pattern match in theoretic compiler and realistic llvm code generation phase, please focus on the classes relationship in this backend structure. Once knowing the structure, you can create your backend structure as quickly as we did, even though there are 5000 lines of code around added in this chapter.

3.1 TargetMachine structure

lbdex/chapters/Chapter3_1/Cpu0TargetObjectFile.h

```
//===-- llvm/Target/Cpu0TargetObjectFile.h - Cpu0 Object Info ---*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
```

(continues on next page)

(continued from previous page)

```
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0TARGETOBJECTFILE_H
#define LLVM_LIB_TARGET_CPU0_CPU0TARGETOBJECTFILE_H

#include "Cpu0Config.h"

#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/TargetLoweringObjectFileImpl.h"

namespace llvm {
class Cpu0TargetMachine;
  class Cpu0TargetObjectFile : public TargetLoweringObjectFileELF {
    MCSection *SmallDataSection;
    MCSection *SmallBSSSection;
    const Cpu0TargetMachine *TM;

  public:

    void Initialize(MCContext &Ctx, const TargetMachine &TM) override;

  };
} // end namespace llvm

#endif
```

lbdex/chapters/Chapter3_1/Cpu0TargetObjectFile.cpp

```
//===-- Cpu0TargetObjectFile.cpp - Cpu0 Object Files ----------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

#include "Cpu0TargetObjectFile.h"

#include "Cpu0Subtarget.h"
#include "Cpu0TargetMachine.h"
#include "llvm/BinaryFormat/ELF.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/GlobalVariable.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCSectionELF.h"
#include "llvm/Support/CommandLine.h"
```

(continues on next page)

(continued from previous page)

```
#include "llvm/Target/TargetMachine.h"
using namespace llvm;

static cl::opt<unsigned>
SSThreshold("cpu0-ssection-threshold", cl::Hidden,
            cl::desc("Small data and bss section threshold size (default=8)"),
            cl::init(8));

void Cpu0TargetObjectFile::Initialize(MCContext &Ctx, const TargetMachine &TM){
  TargetLoweringObjectFileELF::Initialize(Ctx, TM);
  InitializeELF(TM.Options.UseInitArray);

  SmallDataSection = getContext().getELFSection(
      ".sdata", ELF::SHT_PROGBITS, ELF::SHF_WRITE | ELF::SHF_ALLOC);

  SmallBSSSection = getContext().getELFSection(".sbss", ELF::SHT_NOBITS,
                                               ELF::SHF_WRITE | ELF::SHF_ALLOC);
  this->TM = &static_cast<const Cpu0TargetMachine &>(TM);
}
```

lbdex/chapters/Chapter3_1/Cpu0TargetMachine.h

```
//===-- Cpu0TargetMachine.h - Define TargetMachine for Cpu0 -----*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file declares the Cpu0 specific subclass of TargetMachine.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0TARGETMACHINE_H
#define LLVM_LIB_TARGET_CPU0_CPU0TARGETMACHINE_H

#include "Cpu0Config.h"

#include "MCTargetDesc/Cpu0ABIInfo.h"
#include "Cpu0Subtarget.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/CodeGen/SelectionDAGISel.h"
#include "llvm/CodeGen/TargetFrameLowering.h"
#include "llvm/Support/CodeGen.h"
#include "llvm/Target/TargetMachine.h"

namespace llvm {
```

(continues on next page)

(continued from previous page)

```
class formatted_raw_ostream;
class Cpu0RegisterInfo;

class Cpu0TargetMachine : public LLVMTargetMachine {
  bool isLittle;
  std::unique_ptr<TargetLoweringObjectFile> TLOF;
  // Selected ABI
  Cpu0ABIInfo ABI;
  Cpu0Subtarget DefaultSubtarget;

  mutable StringMap<std::unique_ptr<Cpu0Subtarget>> SubtargetMap;
public:
  Cpu0TargetMachine(const Target &T, const Triple &TT, StringRef CPU,
                    StringRef FS, const TargetOptions &Options,
                    Optional<Reloc::Model> RM, Optional<CodeModel::Model> CM,
                    CodeGenOpt::Level OL, bool JIT, bool isLittle);
  ~Cpu0TargetMachine() override;

  const Cpu0Subtarget *getSubtargetImpl() const {
    return &DefaultSubtarget;
  }

  const Cpu0Subtarget *getSubtargetImpl(const Function &F) const override;

  // Pass Pipeline Configuration
  TargetPassConfig *createPassConfig(PassManagerBase &PM) override;

  TargetLoweringObjectFile *getObjFileLowering() const override {
    return TLOF.get();
  }
  bool isLittleEndian() const { return isLittle; }
  const Cpu0ABIInfo &getABI() const { return ABI; }
};

/// Cpu0ebTargetMachine - Cpu032 big endian target machine.
///
class Cpu0ebTargetMachine : public Cpu0TargetMachine {
  virtual void anchor();
public:
  Cpu0ebTargetMachine(const Target &T, const Triple &TT, StringRef CPU,
                      StringRef FS, const TargetOptions &Options,
                      Optional<Reloc::Model> RM, Optional<CodeModel::Model> CM,
                      CodeGenOpt::Level OL, bool JIT);
};

/// Cpu0elTargetMachine - Cpu032 little endian target machine.
///
class Cpu0elTargetMachine : public Cpu0TargetMachine {
  virtual void anchor();
public:
  Cpu0elTargetMachine(const Target &T, const Triple &TT, StringRef CPU,
                      StringRef FS, const TargetOptions &Options,
```

(continues on next page)

(continued from previous page)

```
                      Optional<Reloc::Model> RM, Optional<CodeModel::Model> CM,
                      CodeGenOpt::Level OL, bool JIT);
};
} // End llvm namespace

#endif
```

lbdex/chapters/Chapter3_1/Cpu0TargetMachine.cpp

```
//===-- Cpu0TargetMachine.cpp - Define TargetMachine for Cpu0 -------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// Implements the info about Cpu0 target spec.
//
//===----------------------------------------------------------------------===//

#include "Cpu0TargetMachine.h"
#include "Cpu0.h"

#include "Cpu0Subtarget.h"
#include "Cpu0TargetObjectFile.h"
#include "llvm/IR/Attributes.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/CodeGen.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/CodeGen/TargetPassConfig.h"
#include "llvm/Support/TargetRegistry.h"
#include "llvm/Target/TargetOptions.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0"

extern "C" void LLVMInitializeCpu0Target() {
  // Register the target.
  //- Big endian Target Machine
  RegisterTargetMachine<Cpu0ebTargetMachine> X(TheCpu0Target);
  //- Little endian Target Machine
  RegisterTargetMachine<Cpu0elTargetMachine> Y(TheCpu0elTarget);
}

static std::string computeDataLayout(const Triple &TT, StringRef CPU,
                                      const TargetOptions &Options,
```

(continues on next page)

(continued from previous page)

```
                                      bool isLittle) {
  std::string Ret = "";
  // There are both little and big endian cpu0.
  if (isLittle)
    Ret += "e";
  else
    Ret += "E";

  Ret += "-m:m";

  // Pointers are 32 bit on some ABIs.
  Ret += "-p:32:32";

  // 8 and 16 bit integers only need to have natural alignment, but try to
  // align them to 32 bits. 64 bit integers have natural alignment.
  Ret += "-i8:8:32-i16:16:32-i64:64";

  // 32 bit registers are always available and the stack is at least 64 bit
  // aligned.
  Ret += "-n32-S64";

  return Ret;
}

static Reloc::Model getEffectiveRelocModel(bool JIT,
                                           Optional<Reloc::Model> RM) {
  if (!RM.hasValue() || JIT)
    return Reloc::Static;
  return *RM;
}

// DataLayout --> Big-endian, 32-bit pointer/ABI/alignment
// The stack is always 8 byte aligned
// On function prologue, the stack is created by decrementing
// its pointer. Once decremented, all references are done with positive
// offset from the stack/frame pointer, using StackGrowsUp enables
// an easier handling.
// Using CodeModel::Large enables different CALL behavior.
Cpu0TargetMachine::Cpu0TargetMachine(const Target &T, const Triple &TT,
                                     StringRef CPU, StringRef FS,
                                     const TargetOptions &Options,
                                     Optional<Reloc::Model> RM,
                                     Optional<CodeModel::Model> CM,
                                     CodeGenOpt::Level OL, bool JIT,
                                     bool isLittle)
  //- Default is big endian
    : LLVMTargetMachine(T, computeDataLayout(TT, CPU, Options, isLittle), TT,
                        CPU, FS, Options, getEffectiveRelocModel(JIT, RM),
                        getEffectiveCodeModel(CM, CodeModel::Small), OL),
      isLittle(isLittle), TLOF(std::make_unique<Cpu0TargetObjectFile>()),
      ABI(Cpu0ABIInfo::computeTargetABI()),
      DefaultSubtarget(TT, CPU, FS, isLittle, *this) {
```

(continues on next page)

(continued from previous page)

```
  // initAsmInfo will display features by llc -march=cpu0 -mcpu=help on 3.7 but
  // not on 3.6
  initAsmInfo();
}

Cpu0TargetMachine::~Cpu0TargetMachine() {}

void Cpu0ebTargetMachine::anchor() { }

Cpu0ebTargetMachine::Cpu0ebTargetMachine(const Target &T, const Triple &TT,
                                         StringRef CPU, StringRef FS,
                                         const TargetOptions &Options,
                                         Optional<Reloc::Model> RM,
                                         Optional<CodeModel::Model> CM,
                                         CodeGenOpt::Level OL, bool JIT)
    : Cpu0TargetMachine(T, TT, CPU, FS, Options, RM, CM, OL, JIT, false) {}

void Cpu0elTargetMachine::anchor() { }

Cpu0elTargetMachine::Cpu0elTargetMachine(const Target &T, const Triple &TT,
                                         StringRef CPU, StringRef FS,
                                         const TargetOptions &Options,
                                         Optional<Reloc::Model> RM,
                                         Optional<CodeModel::Model> CM,
                                         CodeGenOpt::Level OL, bool JIT)
    : Cpu0TargetMachine(T, TT, CPU, FS, Options, RM, CM, OL, JIT, true) {}

const Cpu0Subtarget *
Cpu0TargetMachine::getSubtargetImpl(const Function &F) const {
  std::string CPU = TargetCPU;
  std::string FS = TargetFS;

  auto &I = SubtargetMap[CPU + FS];
  if (!I) {
    // This needs to be done before we create a new subtarget since any
    // creation will depend on the TM and the code generation flags on the
    // function that reside in TargetOptions.
    resetTargetOptions(F);
    I = std::make_unique<Cpu0Subtarget>(TargetTriple, CPU, FS, isLittle,
                                        *this);
  }
  return I.get();
}

namespace {
//@Cpu0PassConfig {
/// Cpu0 Code Generator Pass Configuration Options.
class Cpu0PassConfig : public TargetPassConfig {
public:
  Cpu0PassConfig(Cpu0TargetMachine &TM, PassManagerBase &PM)
    : TargetPassConfig(TM, PM) {}
```

(continues on next page)

(continued from previous page)

```
  Cpu0TargetMachine &getCpu0TargetMachine() const {
    return getTM<Cpu0TargetMachine>();
  }

  const Cpu0Subtarget &getCpu0Subtarget() const {
    return *getCpu0TargetMachine().getSubtargetImpl();
  }
};
} // namespace

TargetPassConfig *Cpu0TargetMachine::createPassConfig(PassManagerBase &PM) {
  return new Cpu0PassConfig(*this, PM);
}
```

include/llvm/Target/TargetInstInfo.h

```
class TargetInstrInfo : public MCInstrInfo {
  TargetInstrInfo(const TargetInstrInfo &) = delete;
  void operator=(const TargetInstrInfo &) = delete;
public:
  ...
}
...
class TargetInstrInfoImpl : public TargetInstrInfo {
protected:
  TargetInstrInfoImpl(int CallFrameSetupOpcode = -1,
            int CallFrameDestroyOpcode = -1)
  : TargetInstrInfo(CallFrameSetupOpcode, CallFrameDestroyOpcode) {}
public:
  ...
}
```

lbdex/chapters/Chapter3_1/Cpu0.td

```
// Without this will have error: 'cpu032I' is not a recognized processor for
//  this target (ignoring processor)
//===----------------------------------------------------------------------===//
// Cpu0 Subtarget features                                                    //
//===----------------------------------------------------------------------===//


def FeatureChapter3_1  : SubtargetFeature<"ch3_1", "HasChapterDummy", "true",
                                 "Enable Chapter instructions.">;
def FeatureChapter3_2  : SubtargetFeature<"ch3_2", "HasChapterDummy", "true",
                                 "Enable Chapter instructions.">;
def FeatureChapter3_3  : SubtargetFeature<"ch3_3", "HasChapterDummy", "true",
                                 "Enable Chapter instructions.">;
def FeatureChapter3_4  : SubtargetFeature<"ch3_4", "HasChapterDummy", "true",
```

(continues on next page)

(continued from previous page)

```
                                "Enable Chapter instructions.">;
def FeatureChapter3_5  : SubtargetFeature<"ch3_5", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter4_1  : SubtargetFeature<"ch4_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter4_2  : SubtargetFeature<"ch4_2", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter5_1  : SubtargetFeature<"ch5_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter6_1  : SubtargetFeature<"ch6_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter7_1  : SubtargetFeature<"ch7_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter8_1  : SubtargetFeature<"ch8_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter8_2  : SubtargetFeature<"ch8_2", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter9_1  : SubtargetFeature<"ch9_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter9_2  : SubtargetFeature<"ch9_2", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter9_3  : SubtargetFeature<"ch9_3", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter10_1 : SubtargetFeature<"ch10_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter11_1 : SubtargetFeature<"ch11_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter11_2 : SubtargetFeature<"ch11_2", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter12_1 : SubtargetFeature<"ch12_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapterAll  : SubtargetFeature<"chall", "HasChapterDummy", "true",
                                "Enable Chapter instructions.",
                                [FeatureChapter3_1, FeatureChapter3_2,
                                 FeatureChapter3_3, FeatureChapter3_4,
                                 FeatureChapter3_5,
                                 FeatureChapter4_1, FeatureChapter4_2,
                                 FeatureChapter5_1, FeatureChapter6_1,
                                 FeatureChapter7_1, FeatureChapter8_1,
                                 FeatureChapter8_2, FeatureChapter9_1,
                                 FeatureChapter9_2, FeatureChapter9_3,
                                 FeatureChapter10_1,
                                 FeatureChapter11_1, FeatureChapter11_2,
                                 FeatureChapter12_1]>;


def FeatureCmp         : SubtargetFeature<"cmp", "HasCmp", "true",
                                "Enable 'cmp' instructions.">;
def FeatureSlt         : SubtargetFeature<"slt", "HasSlt", "true",
                                "Enable 'slt' instructions.">;
def FeatureCpu032I     : SubtargetFeature<"cpu032I", "Cpu0ArchVersion",
                                "Cpu032I", "Cpu032I ISA Support",
```

(continues on next page)

(continued from previous page)

```
                                  [FeatureCmp, FeatureChapterAll]>;
def FeatureCpu032II    : SubtargetFeature<"cpu032II", "Cpu0ArchVersion",
                                  "Cpu032II", "Cpu032II ISA Support (slt)",
                                  [FeatureCmp, FeatureSlt, FeatureChapterAll]>;
```

```
include "Cpu0CallingConv.td"
```

```
class Proc<string Name, list<SubtargetFeature> Features>
 : Processor<Name, Cpu0GenericItineraries, Features>;

def : Proc<"cpu032I",  [FeatureCpu032I]>;
def : Proc<"cpu032II", [FeatureCpu032II]>;
// Above make Cpu0GenSubtargetInfo.inc set feature bit as the following order
// enum {
//   FeatureCmp =  1ULL << 0,
//   FeatureCpu032I =  1ULL << 1,
//   FeatureCpu032II =  1ULL << 2,
//   FeatureSlt =  1ULL << 3
// };
```

lbdex/chapters/Chapter3_1/Cpu0CallingConv.td

```
//===-- Cpu0CallingConv.td - Calling Conventions for Cpu0 --*- tablegen -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
// This describes the calling conventions for Cpu0 architecture.
//===----------------------------------------------------------------------===//

/// CCIfSubtarget - Match if the current subtarget has a feature F.
class CCIfSubtarget<string F, CCAction A>:
  CCIf<!strconcat("State.getTarget().getSubtarget<Cpu0Subtarget>().", F), A>;

def CSR_O32 : CalleeSavedRegs<(add LR, FP,
                                   (sequence "S%u", 1, 0))>;
```

lbdex/chapters/Chapter3_1/Cpu0FrameLowering.h

```
//===-- Cpu0FrameLowering.h - Define frame lowering for Cpu0 ----*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
//
//
//===----------------------------------------------------------------------===//
#ifndef LLVM_LIB_TARGET_CPU0_CPU0FRAMELOWERING_H
#define LLVM_LIB_TARGET_CPU0_CPU0FRAMELOWERING_H

#include "Cpu0Config.h"

#include "Cpu0.h"
#include "llvm/CodeGen/TargetFrameLowering.h"

namespace llvm {
  class Cpu0Subtarget;

class Cpu0FrameLowering : public TargetFrameLowering {
protected:
  const Cpu0Subtarget &STI;

public:
  explicit Cpu0FrameLowering(const Cpu0Subtarget &sti, unsigned Alignment)
    : TargetFrameLowering(StackGrowsDown, Align(Alignment), 0, Align(Alignment)),
      STI(sti) {
  }

  static const Cpu0FrameLowering *create(const Cpu0Subtarget &ST);

  bool hasFP(const MachineFunction &MF) const override;

};

/// Create Cpu0FrameLowering objects.
const Cpu0FrameLowering *createCpu0SEFrameLowering(const Cpu0Subtarget &ST);

} // End llvm namespace

#endif
```

lbdex/chapters/Chapter3_1/Cpu0FrameLowering.cpp

```
//===-- Cpu0FrameLowering.cpp - Cpu0 Frame Information ---------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the Cpu0 implementation of TargetFrameLowering class.
//
//===----------------------------------------------------------------------===//

#include "Cpu0FrameLowering.h"

#include "Cpu0InstrInfo.h"
#include "Cpu0MachineFunction.h"
#include "Cpu0Subtarget.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineModuleInfo.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Target/TargetOptions.h"

using namespace llvm;

//- emitProlog() and emitEpilogue must exist for main().

//===----------------------------------------------------------------------===//
//
// Stack Frame Processing methods
// +----------------------------+
//
// The stack is allocated decrementing the stack pointer on
// the first instruction of a function prologue. Once decremented,
// all stack references are done thought a positive offset
// from the stack/frame pointer, so the stack is considering
// to grow up! Otherwise terrible hacks would have to be made
// to get this stack ABI compliant :)
//
//  The stack frame required by the ABI (after call):
//  Offset
//
//  0                 ----------
//  4                 Args to pass
//  .                 saved $GP  (used in PIC)
//  .                 Alloca allocations
```

(continues on next page)

(continued from previous page)

```
//  .                Local Area
//  .                CPU "Callee Saved" Registers
//  .                saved FP
//  .                saved RA
//  .                FPU "Callee Saved" Registers
//  StackSize        -----------
//
// Offset - offset from sp after stack allocation on function prologue
//
// The sp is the stack pointer subtracted/added from the stack size
// at the Prologue/Epilogue
//
// References to the previous stack (to obtain arguments) are done
// with offsets that exceeds the stack size: (stacksize+(4*(num_arg-1))
//
// Examples:
// - reference to the actual stack frame
//   for any local area var there is smt like : FI >= 0, StackOffset: 4
//     st REGX, 4(SP)
//
// - reference to previous stack frame
//   suppose there's a load to the 5th arguments : FI < 0, StackOffset: 16.
//   The emitted instruction will be something like:
//     ld REGX, 16+StackSize(SP)
//
// Since the total stack size is unknown on LowerFormalArguments, all
// stack references (ObjectOffset) created to reference the function
// arguments, are negative numbers. This way, on eliminateFrameIndex it's
// possible to detect those references and the offsets are adjusted to
// their real location.
//
//===----------------------------------------------------------------------===//

const Cpu0FrameLowering *Cpu0FrameLowering::create(const Cpu0Subtarget &ST) {
  return llvm::createCpu0SEFrameLowering(ST);
}

// hasFP - Return true if the specified function should have a dedicated frame
// pointer register.  This is true if the function has variable sized allocas,
// if it needs dynamic stack realignment, if frame pointer elimination is
// disabled, or if the frame address is taken.
bool Cpu0FrameLowering::hasFP(const MachineFunction &MF) const {
  const MachineFrameInfo &MFI = MF.getFrameInfo();
  const TargetRegisterInfo *TRI = STI.getRegisterInfo();

  return MF.getTarget().Options.DisableFramePointerElim(MF) ||
      MFI.hasVarSizedObjects() || MFI.isFrameAddressTaken() ||
      TRI->needsStackRealignment(MF);
}
```

lbdex/chapters/Chapter3_1/Cpu0SEFrameLowering.h

```
//===-- Cpu0SEFrameLowering.h - Cpu032/64 frame lowering --------*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
//
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0SEFRAMELOWERING_H
#define LLVM_LIB_TARGET_CPU0_CPU0SEFRAMELOWERING_H

#include "Cpu0Config.h"

#include "Cpu0FrameLowering.h"

namespace llvm {

class Cpu0SEFrameLowering : public Cpu0FrameLowering {
public:
  explicit Cpu0SEFrameLowering(const Cpu0Subtarget &STI);

  /// emitProlog/emitEpilog - These methods insert prolog and epilog code into
  /// the function.
  void emitPrologue(MachineFunction &MF, MachineBasicBlock &MBB) const override;
  void emitEpilogue(MachineFunction &MF, MachineBasicBlock &MBB) const override;

};

} // End llvm namespace

#endif
```

lbdex/chapters/Chapter3_1/Cpu0SEFrameLowering.cpp

```
//===-- Cpu0SEFrameLowering.cpp - Cpu0 Frame Information -------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
```

(continues on next page)

(continued from previous page)

```
// This file contains the Cpu0 implementation of TargetFrameLowering class.
//
//===----------------------------------------------------------------------===//

#include "Cpu0SEFrameLowering.h"

#include "Cpu0MachineFunction.h"
#include "Cpu0SEInstrInfo.h"
#include "Cpu0Subtarget.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineModuleInfo.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/RegisterScavenging.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Target/TargetOptions.h"

using namespace llvm;

Cpu0SEFrameLowering::Cpu0SEFrameLowering(const Cpu0Subtarget &STI)
    : Cpu0FrameLowering(STI, STI.stackAlignment()) {}

//@emitPrologue {
void Cpu0SEFrameLowering::emitPrologue(MachineFunction &MF,
                                       MachineBasicBlock &MBB) const {
}
//}

//@emitEpilogue {
void Cpu0SEFrameLowering::emitEpilogue(MachineFunction &MF,
                                   MachineBasicBlock &MBB) const {
}
//}

const Cpu0FrameLowering *
llvm::createCpu0SEFrameLowering(const Cpu0Subtarget &ST) {
  return new Cpu0SEFrameLowering(ST);
}
```

lbdex/chapters/Chapter3_1/Cpu0InstrInfo.h

```
//===-- Cpu0InstrInfo.h - Cpu0 Instruction Information ----------*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the Cpu0 implementation of the TargetInstrInfo class.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0INSTRINFO_H
#define LLVM_LIB_TARGET_CPU0_CPU0INSTRINFO_H

#include "Cpu0Config.h"

#include "Cpu0.h"
#include "Cpu0RegisterInfo.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/TargetInstrInfo.h"

#define GET_INSTRINFO_HEADER
#include "Cpu0GenInstrInfo.inc"

namespace llvm {

class Cpu0InstrInfo : public Cpu0GenInstrInfo {
  virtual void anchor();
protected:
  const Cpu0Subtarget &Subtarget;
public:
  explicit Cpu0InstrInfo(const Cpu0Subtarget &STI);

  static const Cpu0InstrInfo *create(Cpu0Subtarget &STI);

  /// getRegisterInfo - TargetInstrInfo is a superset of MRegister info.  As
  /// such, whenever a client has an instance of instruction info, it should
  /// always be able to get register info as well (through this method).
  ///
  virtual const Cpu0RegisterInfo &getRegisterInfo() const = 0;

  /// Return the number of bytes of code the specified instruction may be.
  unsigned GetInstSizeInBytes(const MachineInstr &MI) const;

protected:
};
const Cpu0InstrInfo *createCpu0SEInstrInfo(const Cpu0Subtarget &STI);
}
```

(continues on next page)

(continued from previous page)

```
#endif
```

lbdex/chapters/Chapter3_1/Cpu0InstrInfo.cpp

```
//===-- Cpu0InstrInfo.cpp - Cpu0 Instruction Information ------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the Cpu0 implementation of the TargetInstrInfo class.
//
//===----------------------------------------------------------------------===//

#include "Cpu0InstrInfo.h"

#include "Cpu0TargetMachine.h"
#include "Cpu0MachineFunction.h"
#include "llvm/ADT/STLExtras.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/TargetRegistry.h"

using namespace llvm;

#define GET_INSTRINFO_CTOR_DTOR
#include "Cpu0GenInstrInfo.inc"

// Pin the vtable to this file.
void Cpu0InstrInfo::anchor() {}

//@Cpu0InstrInfo {
Cpu0InstrInfo::Cpu0InstrInfo(const Cpu0Subtarget &STI)
    :
      Subtarget(STI) {}

const Cpu0InstrInfo *Cpu0InstrInfo::create(Cpu0Subtarget &STI) {
  return llvm::createCpu0SEInstrInfo(STI);
}

//@GetInstSizeInBytes {
/// Return the number of bytes of code the specified instruction may be.
unsigned Cpu0InstrInfo::GetInstSizeInBytes(const MachineInstr &MI) const {
//@GetInstSizeInBytes - body
  switch (MI.getOpcode()) {
  default:
```

(continues on next page)

(continued from previous page)

```
    return MI.getDesc().getSize();
  }
}
```

lbdex/chapters/Chapter3_1/Cpu0InstrInfo.td

```
//===----------------------------------------------------------------------===//
// Cpu0 Instruction Predicate Definitions.
//===----------------------------------------------------------------------===//

def Ch3_1       :     Predicate<"Subtarget->hasChapter3_1()">,
                      AssemblerPredicate<(all_of FeatureChapter3_1)>;
def Ch3_2       :     Predicate<"Subtarget->hasChapter3_2()">,
                      AssemblerPredicate<(all_of FeatureChapter3_2)>;
def Ch3_3       :     Predicate<"Subtarget->hasChapter3_3()">,
                      AssemblerPredicate<(all_of FeatureChapter3_3)>;
def Ch3_4       :     Predicate<"Subtarget->hasChapter3_4()">,
                      AssemblerPredicate<(all_of FeatureChapter3_4)>;
def Ch3_5       :     Predicate<"Subtarget->hasChapter3_5()">,
                      AssemblerPredicate<(all_of FeatureChapter3_5)>;
def Ch4_1       :     Predicate<"Subtarget->hasChapter4_1()">,
                      AssemblerPredicate<(all_of FeatureChapter4_1)>;
def Ch4_2       :     Predicate<"Subtarget->hasChapter4_2()">,
                      AssemblerPredicate<(all_of FeatureChapter4_2)>;
def Ch5_1       :     Predicate<"Subtarget->hasChapter5_1()">,
                      AssemblerPredicate<(all_of FeatureChapter5_1)>;
def Ch6_1       :     Predicate<"Subtarget->hasChapter6_1()">,
                      AssemblerPredicate<(all_of FeatureChapter6_1)>;
def Ch7_1       :     Predicate<"Subtarget->hasChapter7_1()">,
                      AssemblerPredicate<(all_of FeatureChapter7_1)>;
def Ch8_1       :     Predicate<"Subtarget->hasChapter8_1()">,
                      AssemblerPredicate<(all_of FeatureChapter8_1)>;
def Ch8_2       :     Predicate<"Subtarget->hasChapter8_2()">,
                      AssemblerPredicate<(all_of FeatureChapter8_2)>;
def Ch9_1       :     Predicate<"Subtarget->hasChapter9_1()">,
                      AssemblerPredicate<(all_of FeatureChapter9_1)>;
def Ch9_2       :     Predicate<"Subtarget->hasChapter9_2()">,
                      AssemblerPredicate<(all_of FeatureChapter9_2)>;
def Ch9_3       :     Predicate<"Subtarget->hasChapter9_3()">,
                      AssemblerPredicate<(all_of FeatureChapter9_3)>;
def Ch10_1      :     Predicate<"Subtarget->hasChapter10_1()">,
                      AssemblerPredicate<(all_of FeatureChapter10_1)>;
def Ch11_1      :     Predicate<"Subtarget->hasChapter11_1()">,
                      AssemblerPredicate<(all_of FeatureChapter11_1)>;
def Ch11_2      :     Predicate<"Subtarget->hasChapter11_2()">,
                      AssemblerPredicate<(all_of FeatureChapter11_2)>;
def Ch12_1      :     Predicate<"Subtarget->hasChapter12_1()">,
                      AssemblerPredicate<(all_of FeatureChapter12_1)>;
def Ch_all      :     Predicate<"Subtarget->hasChapterAll()">,
```

(continues on next page)

(continued from previous page)

```
                    AssemblerPredicate<(all_of FeatureChapterAll)>;

def EnableOverflow  : Predicate<"Subtarget->enableOverflow()">;
def DisableOverflow : Predicate<"Subtarget->disableOverflow()">;

def HasCmp      :     Predicate<"Subtarget->hasCmp()">;
def HasSlt      :     Predicate<"Subtarget->hasSlt()">;
```

lbdex/chapters/Chapter3_1/Cpu0ISelLowering.h

```
//===-- Cpu0ISelLowering.h - Cpu0 DAG Lowering Interface --------*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file defines the interfaces that Cpu0 uses to lower LLVM code into a
// selection DAG.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0ISELLOWERING_H
#define LLVM_LIB_TARGET_CPU0_CPU0ISELLOWERING_H

#include "Cpu0Config.h"

#include "MCTargetDesc/Cpu0ABIInfo.h"
#include "Cpu0.h"
#include "llvm/CodeGen/CallingConvLower.h"
#include "llvm/CodeGen/SelectionDAG.h"
#include "llvm/IR/Function.h"
#include "llvm/CodeGen/TargetLowering.h"
#include <deque>

namespace llvm {
  namespace Cpu0ISD {
    enum NodeType {
      // Start the numbering from where ISD NodeType finishes.
      FIRST_NUMBER = ISD::BUILTIN_OP_END,

      // Jump and link (call)
      JmpLink,

      // Tail call
      TailCall,

      // Get the Higher 16 bits from a 32-bit immediate
```

(continues on next page)

(continued from previous page)

```
    // No relation with Cpu0 Hi register
    Hi,
    // Get the Lower 16 bits from a 32-bit immediate
    // No relation with Cpu0 Lo register
    Lo,

    // Handle gp_rel (small data/bss sections) relocation.
    GPRel,

    // Thread Pointer
    ThreadPointer,

    // Return
    Ret,

    EH_RETURN,

    // DivRem(u)
    DivRem,
    DivRemU,

    Wrapper,
    DynAlloc,

    Sync
  };
}

//===----------------------------------------------------------------------===//
// TargetLowering Implementation
//===----------------------------------------------------------------------===//
class Cpu0FunctionInfo;
class Cpu0Subtarget;

//@class Cpu0TargetLowering
class Cpu0TargetLowering : public TargetLowering  {
public:
  explicit Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                              const Cpu0Subtarget &STI);

  static const Cpu0TargetLowering *create(const Cpu0TargetMachine &TM,
                                          const Cpu0Subtarget &STI);

  /// getTargetNodeName - This method returns the name of a target specific
  //  DAG node.
  const char *getTargetNodeName(unsigned Opcode) const override;

protected:

  /// ByValArgInfo - Byval argument information.
  struct ByValArgInfo {
    unsigned FirstIdx; // Index of the first register used.
```

(continues on next page)

(continued from previous page)

```
      unsigned NumRegs;  // Number of registers used for this argument.
      unsigned Address;  // Offset of the stack area used to pass this argument.

      ByValArgInfo() : FirstIdx(0), NumRegs(0), Address(0) {}
    };

  protected:
    // Subtarget Info
    const Cpu0Subtarget &Subtarget;
    // Cache the ABI from the TargetMachine, we use it everywhere.
    const Cpu0ABIInfo &ABI;

  private:

    // Lower Operand specifics
    SDValue lowerGlobalAddress(SDValue Op, SelectionDAG &DAG) const;

        //- must be exist even without function all
    SDValue
      LowerFormalArguments(SDValue Chain,
                           CallingConv::ID CallConv, bool IsVarArg,
                           const SmallVectorImpl<ISD::InputArg> &Ins,
                           const SDLoc &dl, SelectionDAG &DAG,
                           SmallVectorImpl<SDValue> &InVals) const override;

    SDValue LowerReturn(SDValue Chain,
                        CallingConv::ID CallConv, bool IsVarArg,
                        const SmallVectorImpl<ISD::OutputArg> &Outs,
                        const SmallVectorImpl<SDValue> &OutVals,
                        const SDLoc &dl, SelectionDAG &DAG) const override;

  };
  const Cpu0TargetLowering *
  createCpu0SETargetLowering(const Cpu0TargetMachine &TM, const Cpu0Subtarget &STI);
}

#endif // Cpu0ISELLOWERING_H
```

lbdex/chapters/Chapter3_1/Cpu0ISelLowering.cpp

```
//===-- Cpu0ISelLowering.cpp - Cpu0 DAG Lowering Implementation -----------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
```

(continues on next page)

(continued from previous page)

```
// This file defines the interfaces that Cpu0 uses to lower LLVM code into a
// selection DAG.
//
//===----------------------------------------------------------------------===//
#include "Cpu0ISelLowering.h"

#include "Cpu0MachineFunction.h"
#include "Cpu0TargetMachine.h"
#include "Cpu0TargetObjectFile.h"
#include "Cpu0Subtarget.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/CodeGen/CallingConvLower.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/SelectionDAG.h"
#include "llvm/CodeGen/ValueTypes.h"
#include "llvm/IR/CallingConv.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/GlobalVariable.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-lower"

//@3_1 1 {
const char *Cpu0TargetLowering::getTargetNodeName(unsigned Opcode) const {
  switch (Opcode) {
  case Cpu0ISD::JmpLink:           return "Cpu0ISD::JmpLink";
  case Cpu0ISD::TailCall:          return "Cpu0ISD::TailCall";
  case Cpu0ISD::Hi:                return "Cpu0ISD::Hi";
  case Cpu0ISD::Lo:                return "Cpu0ISD::Lo";
  case Cpu0ISD::GPRel:             return "Cpu0ISD::GPRel";
  case Cpu0ISD::Ret:               return "Cpu0ISD::Ret";
  case Cpu0ISD::EH_RETURN:         return "Cpu0ISD::EH_RETURN";
  case Cpu0ISD::DivRem:            return "Cpu0ISD::DivRem";
  case Cpu0ISD::DivRemU:           return "Cpu0ISD::DivRemU";
  case Cpu0ISD::Wrapper:           return "Cpu0ISD::Wrapper";
  default:                         return NULL;
  }
}
//@3_1 1 }

//@Cpu0TargetLowering {
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

(continues on next page)

(continued from previous page)

```
}

const Cpu0TargetLowering *Cpu0TargetLowering::create(const Cpu0TargetMachine &TM,
                                                     const Cpu0Subtarget &STI) {
  return llvm::createCpu0SETargetLowering(TM, STI);
}

//===----------------------------------------------------------------------===//
//  Lower helper functions
//===----------------------------------------------------------------------===//

//===----------------------------------------------------------------------===//
//  Misc Lower Operation implementation
//===----------------------------------------------------------------------===//

#include "Cpu0GenCallingConv.inc"

//===----------------------------------------------------------------------===//
//@            Formal Arguments Calling Convention Implementation
//===----------------------------------------------------------------------===//

//@LowerFormalArguments {
/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool IsVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         const SDLoc &DL, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                          const {

  return Chain;
}
// @LowerFormalArguments }

//===----------------------------------------------------------------------===//
//@              Return Value Calling Convention Implementation
//===----------------------------------------------------------------------===//

SDValue
Cpu0TargetLowering::LowerReturn(SDValue Chain,
                                CallingConv::ID CallConv, bool IsVarArg,
                                const SmallVectorImpl<ISD::OutputArg> &Outs,
                                const SmallVectorImpl<SDValue> &OutVals,
                                const SDLoc &DL, SelectionDAG &DAG) const {
  return DAG.getNode(Cpu0ISD::Ret, DL, MVT::Other,
                     Chain, DAG.getRegister(Cpu0::LR, MVT::i32));
}
```

lbdex/chapters/Chapter3_1/Cpu0SEISelLowering.h

```
//===-- Cpu0ISEISelLowering.h - Cpu0ISE DAG Lowering Interface ----*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// Subclass of Cpu0ITargetLowering specialized for cpu032/64.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0SEISELLOWERING_H
#define LLVM_LIB_TARGET_CPU0_CPU0SEISELLOWERING_H

#include "Cpu0Config.h"

#include "Cpu0ISelLowering.h"
#include "Cpu0RegisterInfo.h"

namespace llvm {
  class Cpu0SETargetLowering : public Cpu0TargetLowering  {
  public:
    explicit Cpu0SETargetLowering(const Cpu0TargetMachine &TM,
                                  const Cpu0Subtarget &STI);

    SDValue LowerOperation(SDValue Op, SelectionDAG &DAG) const override;
  private:
  };
}

#endif // Cpu0ISEISELLOWERING_H
```

lbdex/chapters/Chapter3_1/Cpu0SEISelLowering.cpp

```
//===-- Cpu0SEISelLowering.cpp - Cpu0SE DAG Lowering Interface --*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// Subclass of Cpu0TargetLowering specialized for cpu032.
//
//===----------------------------------------------------------------------===//
```

(continues on next page)

(continued from previous page)

```
#include "Cpu0MachineFunction.h"
#include "Cpu0SEISelLowering.h"

#include "Cpu0RegisterInfo.h"
#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/TargetInstrInfo.h"
#include "llvm/IR/Intrinsics.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-isel"

static cl::opt<bool>
EnableCpu0TailCalls("enable-cpu0-tail-calls", cl::Hidden,
                    cl::desc("CPU0: Enable tail calls."), cl::init(false));

//@Cpu0SETargetLowering {
Cpu0SETargetLowering::Cpu0SETargetLowering(const Cpu0TargetMachine &TM,
                                           const Cpu0Subtarget &STI)
    : Cpu0TargetLowering(TM, STI) {
//@Cpu0SETargetLowering body {
  // Set up the register classes
  addRegisterClass(MVT::i32, &Cpu0::CPURegsRegClass);

// must, computeRegisterProperties - Once all of the register classes are
//  added, this allows us to compute derived properties we expose.
  computeRegisterProperties(Subtarget.getRegisterInfo());
}

SDValue Cpu0SETargetLowering::LowerOperation(SDValue Op,
                                             SelectionDAG &DAG) const {

  return Cpu0TargetLowering::LowerOperation(Op, DAG);
}

const Cpu0TargetLowering *
llvm::createCpu0SETargetLowering(const Cpu0TargetMachine &TM,
                                 const Cpu0Subtarget &STI) {
  return new Cpu0SETargetLowering(TM, STI);
}
```

lbdex/chapters/Chapter3_1/Cpu0MachineFunction.h

```
//===-- Cpu0MachineFunctionInfo.h - Private data used for Cpu0 ----*- C++ -*-=//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file declares the Cpu0 specific subclass of MachineFunctionInfo.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0MACHINEFUNCTION_H
#define LLVM_LIB_TARGET_CPU0_CPU0MACHINEFUNCTION_H

#include "Cpu0Config.h"

#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineMemOperand.h"
#include "llvm/CodeGen/PseudoSourceValue.h"
#include "llvm/Target/TargetMachine.h"
#include <map>

namespace llvm {

//@1 {
/// Cpu0FunctionInfo - This class is derived from MachineFunction private
/// Cpu0 target-specific information for each MachineFunction.
class Cpu0FunctionInfo : public MachineFunctionInfo {
public:
  Cpu0FunctionInfo(MachineFunction& MF)
  : MF(MF),
    VarArgsFrameIndex(0),
    MaxCallFrameSize(0)
    {}

  ~Cpu0FunctionInfo();

  int getVarArgsFrameIndex() const { return VarArgsFrameIndex; }
  void setVarArgsFrameIndex(int Index) { VarArgsFrameIndex = Index; }

private:
  virtual void anchor();

  MachineFunction& MF;

    /// VarArgsFrameIndex - FrameIndex for start of varargs area.
  int VarArgsFrameIndex;
```

(continues on next page)

(continued from previous page)

```
  unsigned MaxCallFrameSize;
};
//@1 }

} // end of namespace llvm

#endif // CPU0_MACHINE_FUNCTION_INFO_H
```

lbdex/chapters/Chapter3_1/Cpu0MachineFunction.cpp

```
//===-- Cpu0MachineFunctionInfo.cpp - Private data used for Cpu0 ----------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

#include "Cpu0MachineFunction.h"

#include "Cpu0InstrInfo.h"
#include "Cpu0Subtarget.h"
#include "llvm/IR/Function.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"

using namespace llvm;

bool FixGlobalBaseReg;

Cpu0FunctionInfo::~Cpu0FunctionInfo() {}

void Cpu0FunctionInfo::anchor() { }
```

lbdex/chapters/Chapter3_1/MCTargetDesc/Cpu0ABIInfo.h

```
//===---- Cpu0ABIInfo.h - Information about CPU0 ABI's --------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0ABIINFO_H
```

(continues on next page)

(continued from previous page)

```
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0ABIINFO_H

#include "Cpu0Config.h"

#include "llvm/ADT/ArrayRef.h"
#include "llvm/ADT/Triple.h"
#include "llvm/IR/CallingConv.h"
#include "llvm/MC/MCRegisterInfo.h"

namespace llvm {

class MCTargetOptions;
class StringRef;
class TargetRegisterClass;

class Cpu0ABIInfo {
public:
  enum class ABI { Unknown, O32, S32 };

protected:
  ABI ThisABI;

public:
  Cpu0ABIInfo(ABI ThisABI) : ThisABI(ThisABI) {}

  static Cpu0ABIInfo Unknown() { return Cpu0ABIInfo(ABI::Unknown); }
  static Cpu0ABIInfo O32() { return Cpu0ABIInfo(ABI::O32); }
  static Cpu0ABIInfo S32() { return Cpu0ABIInfo(ABI::S32); }
  static Cpu0ABIInfo computeTargetABI();

  bool IsKnown() const { return ThisABI != ABI::Unknown; }
  bool IsO32() const { return ThisABI == ABI::O32; }
  bool IsS32() const { return ThisABI == ABI::S32; }
  ABI GetEnumValue() const { return ThisABI; }

  /// The registers to use for byval arguments.
  const ArrayRef<MCPhysReg> GetByValArgRegs() const;

  /// The registers to use for the variable argument list.
  const ArrayRef<MCPhysReg> GetVarArgRegs() const;

  /// Obtain the size of the area allocated by the callee for arguments.
  /// CallingConv::FastCall affects the value for O32.
  unsigned GetCalleeAllocdArgSizeInBytes(CallingConv::ID CC) const;

  /// Ordering of ABI's
  /// Cpu0GenSubtargetInfo.inc will use this to resolve conflicts when given
  /// multiple ABI options.
  bool operator<(const Cpu0ABIInfo Other) const {
    return ThisABI < Other.GetEnumValue();
  }
```

(continues on next page)

(continued from previous page)

```
  unsigned GetStackPtr() const;
  unsigned GetFramePtr() const;
  unsigned GetNullPtr() const;

  unsigned GetEhDataReg(unsigned I) const;
  int EhDataRegSize() const;
};
}

#endif
```

lbdex/chapters/Chapter3_1/MCTargetDesc/Cpu0ABIInfo.cpp

```
//===---- Cpu0ABIInfo.cpp - Information about CPU0 ABI's ------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

#include "Cpu0Config.h"

#include "Cpu0ABIInfo.h"
#include "Cpu0RegisterInfo.h"
#include "llvm/ADT/StringRef.h"
#include "llvm/ADT/StringSwitch.h"
#include "llvm/MC/MCTargetOptions.h"
#include "llvm/Support/CommandLine.h"

using namespace llvm;

static cl::opt<bool>
EnableCpu0S32Calls("cpu0-s32-calls", cl::Hidden,
                   cl::desc("CPU0 S32 call: use stack only to pass arguments.\
                   "), cl::init(false));

namespace {
static const MCPhysReg O32IntRegs[4] = {Cpu0::A0, Cpu0::A1};
static const MCPhysReg S32IntRegs = {};
}

const ArrayRef<MCPhysReg> Cpu0ABIInfo::GetByValArgRegs() const {
  if (IsO32())
    return makeArrayRef(O32IntRegs);
  if (IsS32())
    return makeArrayRef(S32IntRegs);
  llvm_unreachable("Unhandled ABI");
```

(continues on next page)

(continued from previous page)

```
}

const ArrayRef<MCPhysReg> Cpu0ABIInfo::GetVarArgRegs() const {
  if (IsO32())
    return makeArrayRef(O32IntRegs);
  if (IsS32())
    return makeArrayRef(S32IntRegs);
  llvm_unreachable("Unhandled ABI");
}

unsigned Cpu0ABIInfo::GetCalleeAllocdArgSizeInBytes(CallingConv::ID CC) const {
  if (IsO32())
    return CC != 0;
  if (IsS32())
    return 0;
  llvm_unreachable("Unhandled ABI");
}

Cpu0ABIInfo Cpu0ABIInfo::computeTargetABI() {
  Cpu0ABIInfo abi(ABI::Unknown);

  if (EnableCpu0S32Calls)
    abi = ABI::S32;
  else
    abi = ABI::O32;
  // Assert exactly one ABI was chosen.
  assert(abi.ThisABI != ABI::Unknown);

  return abi;
}

unsigned Cpu0ABIInfo::GetStackPtr() const {
  return Cpu0::SP;
}

unsigned Cpu0ABIInfo::GetFramePtr() const {
  return Cpu0::FP;
}

unsigned Cpu0ABIInfo::GetNullPtr() const {
  return Cpu0::ZERO;
}

unsigned Cpu0ABIInfo::GetEhDataReg(unsigned I) const {
  static const unsigned EhDataReg[] = {
    Cpu0::A0, Cpu0::A1
  };

  return EhDataReg[I];
}

int Cpu0ABIInfo::EhDataRegSize() const {
```

(continues on next page)

(continued from previous page)

```
  if (ThisABI == ABI::S32)
    return 0;
  else
    return 2;
}
```

lbdex/chapters/Chapter3_1/Cpu0Subtarget.h

```
#include "Cpu0FrameLowering.h"
#include "Cpu0ISelLowering.h"
#include "Cpu0InstrInfo.h"
#include "llvm/CodeGen/SelectionDAGTargetInfo.h"
#include "llvm/CodeGen/TargetSubtargetInfo.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/MC/MCInstrItineraries.h"
#include <string>

#define GET_SUBTARGETINFO_HEADER
#include "Cpu0GenSubtargetInfo.inc"
```

```
namespace llvm {
class StringRef;

class Cpu0TargetMachine;

class Cpu0Subtarget : public Cpu0GenSubtargetInfo {
  virtual void anchor();

public:

  bool HasChapterDummy;
  bool HasChapterAll;

  bool hasChapter3_1() const {
#if CH >= CH3_1
    return true;
#else
    return false;
#endif
  }

  bool hasChapter3_2() const {
#if CH >= CH3_2
    return true;
#else
    return false;
#endif
```

(continues on next page)

(continued from previous page)

```
  }

  bool hasChapter3_3() const {
#if CH >= CH3_3
    return true;
#else
    return false;
#endif
  }

  bool hasChapter3_4() const {
#if CH >= CH3_4
    return true;
#else
    return false;
#endif
  }

  bool hasChapter3_5() const {
#if CH >= CH3_5
    return true;
#else
    return false;
#endif
  }

  bool hasChapter4_1() const {
#if CH >= CH4_1
    return true;
#else
    return false;
#endif
  }

  bool hasChapter4_2() const {
#if CH >= CH4_2
    return true;
#else
    return false;
#endif
  }

  bool hasChapter5_1() const {
#if CH >= CH5_1
    return true;
#else
    return false;
#endif
  }

  bool hasChapter6_1() const {
#if CH >= CH6_1
```

(continues on next page)

(continued from previous page)

```
    return true;
#else
    return false;
#endif
  }

  bool hasChapter7_1() const {
#if CH >= CH7_1
    return true;
#else
    return false;
#endif
  }

  bool hasChapter8_1() const {
#if CH >= CH8_1
    return true;
#else
    return false;
#endif
  }

  bool hasChapter8_2() const {
#if CH >= CH8_2
    return true;
#else
    return false;
#endif
  }

  bool hasChapter9_1() const {
#if CH >= CH9_1
    return true;
#else
    return false;
#endif
  }

  bool hasChapter9_2() const {
#if CH >= CH9_2
    return true;
#else
    return false;
#endif
  }

  bool hasChapter9_3() const {
#if CH >= CH9_3
    return true;
#else
    return false;
#endif
```

(continues on next page)

(continued from previous page)

```
  }

  bool hasChapter10_1() const {
#if CH >= CH10_1
    return true;
#else
    return false;
#endif
  }

  bool hasChapter11_1() const {
#if CH >= CH11_1
    return true;
#else
    return false;
#endif
  }

  bool hasChapter11_2() const {
#if CH >= CH11_2
    return true;
#else
    return false;
#endif
  }

  bool hasChapter12_1() const {
#if CH >= CH12_1
    return true;
#else
    return false;
#endif
  }

protected:
  enum Cpu0ArchEnum {
    Cpu032I,
    Cpu032II
  };

  // Cpu0 architecture version
  Cpu0ArchEnum Cpu0ArchVersion;

  // IsLittle - The target is Little Endian
  bool IsLittle;

  bool EnableOverflow;

  // HasCmp - cmp instructions.
  bool HasCmp;

  // HasSlt - slt instructions.
```

(continues on next page)

(continued from previous page)

```
  bool HasSlt;

  InstrItineraryData InstrItins;
```

```
  const Cpu0TargetMachine &TM;

  Triple TargetTriple;

  const SelectionDAGTargetInfo TSInfo;

  std::unique_ptr<const Cpu0InstrInfo> InstrInfo;
  std::unique_ptr<const Cpu0FrameLowering> FrameLowering;
  std::unique_ptr<const Cpu0TargetLowering> TLInfo;

public:
  bool isPositionIndependent() const;
  const Cpu0ABIInfo &getABI() const;

  /// This constructor initializes the data members to match that
  /// of the specified triple.
  Cpu0Subtarget(const Triple &TT, StringRef CPU, StringRef FS,
                bool little, const Cpu0TargetMachine &_TM);

//- Vitual function, must have
  /// ParseSubtargetFeatures - Parses features string setting specified
  /// subtarget options.  Definition of function is auto generated by tblgen.
  void ParseSubtargetFeatures(StringRef CPU, StringRef TuneCPU, StringRef FS);

  bool isLittle() const { return IsLittle; }
  bool hasCpu032I() const { return Cpu0ArchVersion >= Cpu032I; }
  bool isCpu032I() const { return Cpu0ArchVersion == Cpu032I; }
  bool hasCpu032II() const { return Cpu0ArchVersion >= Cpu032II; }
  bool isCpu032II() const { return Cpu0ArchVersion == Cpu032II; }

  /// Features related to the presence of specific instructions.
  bool enableOverflow() const { return EnableOverflow; }
  bool disableOverflow() const { return !EnableOverflow; }
  bool hasCmp()   const { return HasCmp; }
  bool hasSlt()   const { return HasSlt; }
```

```
  bool abiUsesSoftFloat() const;

  bool enableLongBranchPass() const {
    return hasCpu032II();
  }

  unsigned stackAlignment() const { return 8; }
```

(continues on next page)

(continued from previous page)

```
  Cpu0Subtarget &initializeSubtargetDependencies(StringRef CPU, StringRef FS,
                                                 const TargetMachine &TM);

  const SelectionDAGTargetInfo *getSelectionDAGInfo() const override {
    return &TSInfo;
  }
  const Cpu0InstrInfo *getInstrInfo() const override { return InstrInfo.get(); }
  const TargetFrameLowering *getFrameLowering() const override {
    return FrameLowering.get();
  }
  const Cpu0RegisterInfo *getRegisterInfo() const override {
    return &InstrInfo->getRegisterInfo();
  }
  const Cpu0TargetLowering *getTargetLowering() const override {
    return TLInfo.get();
  }
  const InstrItineraryData *getInstrItineraryData() const override {
    return &InstrItins;
  }
};
} // End llvm namespace


#endif
```

lbdex/chapters/Chapter3_1/Cpu0Subtarget.cpp

```
//===-- Cpu0Subtarget.cpp - Cpu0 Subtarget Information --------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file implements the Cpu0 specific subclass of TargetSubtargetInfo.
//
//===----------------------------------------------------------------------===//

#include "Cpu0Subtarget.h"

#include "Cpu0MachineFunction.h"
#include "Cpu0.h"
#include "Cpu0RegisterInfo.h"

#include "Cpu0TargetMachine.h"
#include "llvm/IR/Attributes.h"
#include "llvm/IR/Function.h"
```

(continues on next page)

(continued from previous page)

```
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/TargetRegistry.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-subtarget"

#define GET_SUBTARGETINFO_TARGET_DESC
#define GET_SUBTARGETINFO_CTOR
#include "Cpu0GenSubtargetInfo.inc"

extern bool FixGlobalBaseReg;

void Cpu0Subtarget::anchor() { }

//@1 {
Cpu0Subtarget::Cpu0Subtarget(const Triple &TT, StringRef CPU,
                             StringRef FS, bool little,
                             const Cpu0TargetMachine &_TM) :
//@1 }
  // Cpu0GenSubtargetInfo will display features by llc -march=cpu0 -mcpu=help
  Cpu0GenSubtargetInfo(TT, CPU, /*TuneCPU*/ CPU, FS),
  IsLittle(little), TM(_TM), TargetTriple(TT), TSInfo(),
      InstrInfo(
          Cpu0InstrInfo::create(initializeSubtargetDependencies(CPU, FS, TM))),
      FrameLowering(Cpu0FrameLowering::create(*this)),
      TLInfo(Cpu0TargetLowering::create(TM, *this)) {

}

bool Cpu0Subtarget::isPositionIndependent() const {
  return TM.isPositionIndependent();
}

Cpu0Subtarget &
Cpu0Subtarget::initializeSubtargetDependencies(StringRef CPU, StringRef FS,
                                               const TargetMachine &TM) {
  if (TargetTriple.getArch() == Triple::cpu0 || TargetTriple.getArch() ==␣
↪Triple::cpu0el) {
    if (CPU.empty() || CPU == "generic") {
      CPU = "cpu032II";
    }
    else if (CPU == "help") {
      CPU = "";
      return *this;
    }
    else if (CPU != "cpu032I" && CPU != "cpu032II") {
      CPU = "cpu032II";
    }
  }
  else {
```

(continues on next page)

(continued from previous page)

```
    errs() << "!!!Error, TargetTriple.getArch() = " << TargetTriple.getArch()
           <<  "CPU = " << CPU << "\n";
    exit(0);
  }

  if (CPU == "cpu032I")
    Cpu0ArchVersion = Cpu032I;
  else if (CPU == "cpu032II")
    Cpu0ArchVersion = Cpu032II;

  if (isCpu032I()) {
    HasCmp = true;
    HasSlt = false;
  }
  else if (isCpu032II()) {
    HasCmp = false;
    HasSlt = true;
  }
  else {
    errs() << "-mcpu must be empty(default:cpu032II), cpu032I or cpu032II" << "\n";
  }

  // Parse features string.
  ParseSubtargetFeatures(CPU, /*TuneCPU*/ CPU, FS);
  // Initialize scheduling itinerary for the specified CPU.
  InstrItins = getInstrItineraryForCPU(CPU);

  return *this;
}

bool Cpu0Subtarget::abiUsesSoftFloat() const {
//  return TM->Options.UseSoftFloat;
  return true;
}

const Cpu0ABIInfo &Cpu0Subtarget::getABI() const { return TM.getABI(); }
```

lbdex/chapters/Chapter3_1/Cpu0RegisterInfo.h

```
//===-- Cpu0RegisterInfo.h - Cpu0 Register Information Impl -----*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the Cpu0 implementation of the TargetRegisterInfo class.
```

(continues on next page)

(continued from previous page)

```
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0REGISTERINFO_H
#define LLVM_LIB_TARGET_CPU0_CPU0REGISTERINFO_H

#include "Cpu0Config.h"

#include "Cpu0.h"
#include "llvm/CodeGen/TargetRegisterInfo.h"

#define GET_REGINFO_HEADER
#include "Cpu0GenRegisterInfo.inc"

namespace llvm {
class Cpu0Subtarget;
class TargetInstrInfo;
class Type;

class Cpu0RegisterInfo : public Cpu0GenRegisterInfo {
protected:
  const Cpu0Subtarget &Subtarget;

public:
  Cpu0RegisterInfo(const Cpu0Subtarget &Subtarget);

  const MCPhysReg *getCalleeSavedRegs(const MachineFunction *MF) const override;

  const uint32_t *getCallPreservedMask(const MachineFunction &MF,
                                       CallingConv::ID) const override;

  BitVector getReservedRegs(const MachineFunction &MF) const override;

  bool requiresRegisterScavenging(const MachineFunction &MF) const override;

  bool trackLivenessAfterRegAlloc(const MachineFunction &MF) const override;

  /// Stack Frame Processing Methods
  void eliminateFrameIndex(MachineBasicBlock::iterator II,
                           int SPAdj, unsigned FIOperandNum,
                           RegScavenger *RS = nullptr) const override;

  /// Debug information queries.
  Register getFrameRegister(const MachineFunction &MF) const override;

  /// \brief Return GPR register class.
  virtual const TargetRegisterClass *intRegClass(unsigned Size) const = 0;
};

} // end namespace llvm

#endif
```

(continues on next page)

(continued from previous page)

lbdex/chapters/Chapter3_1/Cpu0RegisterInfo.cpp

```
//===-- Cpu0RegisterInfo.cpp - CPU0 Register Information -== ---------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the CPU0 implementation of the TargetRegisterInfo class.
//
//===----------------------------------------------------------------------===//

#define DEBUG_TYPE "cpu0-reg-info"

#include "Cpu0RegisterInfo.h"

#include "Cpu0.h"
#include "Cpu0Subtarget.h"
#include "Cpu0MachineFunction.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Type.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"

#define GET_REGINFO_TARGET_DESC
#include "Cpu0GenRegisterInfo.inc"

using namespace llvm;

Cpu0RegisterInfo::Cpu0RegisterInfo(const Cpu0Subtarget &ST)
  : Cpu0GenRegisterInfo(Cpu0::LR), Subtarget(ST) {}

//===----------------------------------------------------------------------===//
// Callee Saved Registers methods
//===----------------------------------------------------------------------===//
/// Cpu0 Callee Saved Registers
// In Cpu0CallConv.td,
// def CSR_O32 : CalleeSavedRegs<(add LR, FP,
//                                    (sequence "S%u", 2, 0))>;
// llc create CSR_O32_SaveList and CSR_O32_RegMask from above defined.
const MCPhysReg *
Cpu0RegisterInfo::getCalleeSavedRegs(const MachineFunction *MF) const {
  return CSR_O32_SaveList;
```

(continues on next page)

(continued from previous page)

```
}

const uint32_t *
Cpu0RegisterInfo::getCallPreservedMask(const MachineFunction &MF,
                                       CallingConv::ID) const {
  return CSR_O32_RegMask;
}

// pure virtual method
//@getReservedRegs {
BitVector Cpu0RegisterInfo::
getReservedRegs(const MachineFunction &MF) const {
//@getReservedRegs body {
  static const uint16_t ReservedCPURegs[] = {
    Cpu0::ZERO, Cpu0::AT, Cpu0::SP, Cpu0::LR, /*Cpu0::SW, */Cpu0::PC
  };
  BitVector Reserved(getNumRegs());

  for (unsigned I = 0; I < array_lengthof(ReservedCPURegs); ++I)
    Reserved.set(ReservedCPURegs[I]);

  return Reserved;
}

//@eliminateFrameIndex {
//- If no eliminateFrameIndex(), it will hang on run.
// pure virtual method
// FrameIndex represent objects inside a abstract stack.
// We must replace FrameIndex with an stack/frame pointer
// direct reference.
void Cpu0RegisterInfo::
eliminateFrameIndex(MachineBasicBlock::iterator II, int SPAdj,
                    unsigned FIOperandNum, RegScavenger *RS) const {
}
//}

bool
Cpu0RegisterInfo::requiresRegisterScavenging(const MachineFunction &MF) const {
  return true;
}

bool
Cpu0RegisterInfo::trackLivenessAfterRegAlloc(const MachineFunction &MF) const {
  return true;
}

// pure virtual method
Register Cpu0RegisterInfo::
getFrameRegister(const MachineFunction &MF) const {
  const TargetFrameLowering *TFI = MF.getSubtarget().getFrameLowering();
  return TFI->hasFP(MF) ? (Cpu0::FP) :
                          (Cpu0::SP);
```

(continues on next page)

(continued from previous page)

```
}
```

lbdex/chapters/Chapter3_1/Cpu0SERegisterInfo.h

```
//===-- Cpu0SERegisterInfo.h - Cpu032 Register Information ------*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the Cpu032/64 implementation of the TargetRegisterInfo
// class.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0SEREGISTERINFO_H
#define LLVM_LIB_TARGET_CPU0_CPU0SEREGISTERINFO_H

#include "Cpu0Config.h"

#include "Cpu0RegisterInfo.h"

namespace llvm {
class Cpu0SEInstrInfo;

class Cpu0SERegisterInfo : public Cpu0RegisterInfo {
public:
  Cpu0SERegisterInfo(const Cpu0Subtarget &Subtarget);

  const TargetRegisterClass *intRegClass(unsigned Size) const override;
};

} // end namespace llvm

#endif
```

lbdex/chapters/Chapter3_1/Cpu0SERegisterInfo.cpp

```
//===-- Cpu0SERegisterInfo.cpp - CPU0 Register Information -------== -------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the CPU0 implementation of the TargetRegisterInfo
// class.
//
//===----------------------------------------------------------------------===//

#include "Cpu0SERegisterInfo.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-reg-info"

Cpu0SERegisterInfo::Cpu0SERegisterInfo(const Cpu0Subtarget &ST)
  : Cpu0RegisterInfo(ST) {}

const TargetRegisterClass *
Cpu0SERegisterInfo::intRegClass(unsigned Size) const {
  return &Cpu0::CPURegsRegClass;
}
```

build/lib/Target/Cpu0/Cpu0GenInstInfo.inc

```
//- Cpu0GenInstInfo.inc which generate from Cpu0InstrInfo.td
#ifdef GET_INSTRINFO_HEADER
#undef GET_INSTRINFO_HEADER
namespace llvm {
struct Cpu0GenInstrInfo : public TargetInstrInfoImpl {
  explicit Cpu0GenInstrInfo(int SO = -1, int DO = -1);
};
} // End llvm namespace
#endif // GET_INSTRINFO_HEADER

#define GET_INSTRINFO_HEADER
#include "Cpu0GenInstrInfo.inc"
//- Cpu0InstInfo.h
class Cpu0InstrInfo : public Cpu0GenInstrInfo {
  Cpu0TargetMachine &TM;
public:
  explicit Cpu0InstrInfo(Cpu0TargetMachine &TM);
};
```

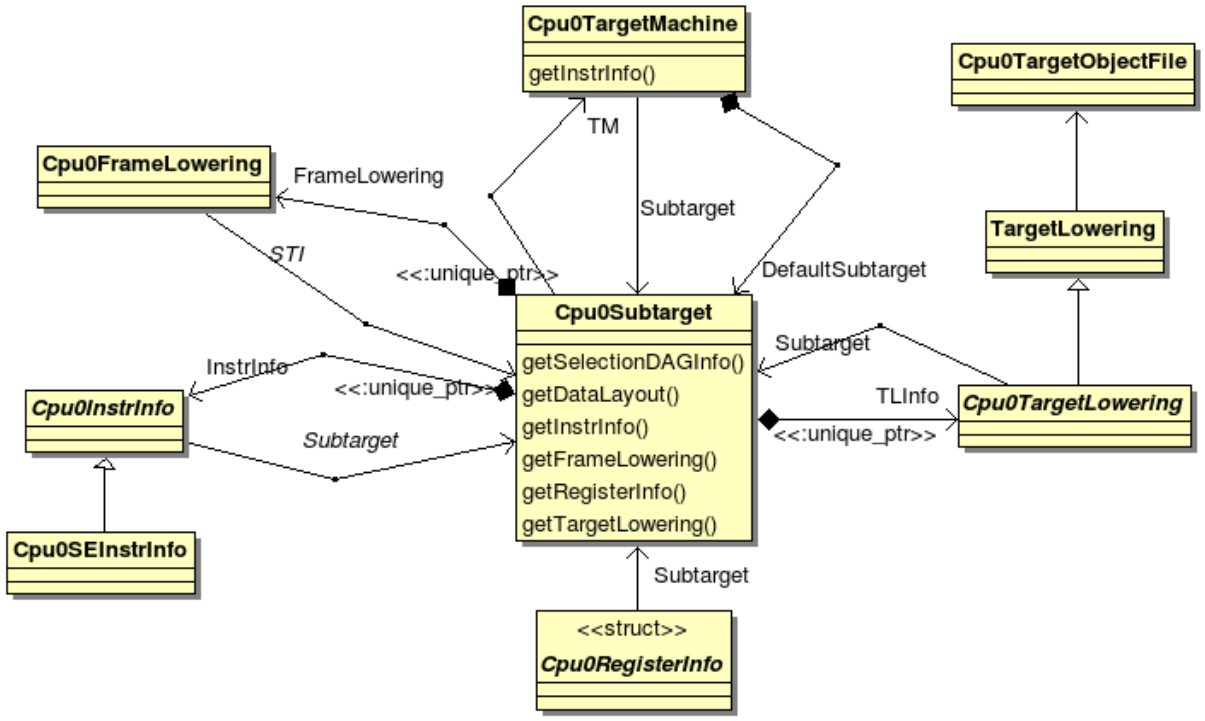


Fig. 3.1: Cpu0 backend class access link

Chapter3_1 add most Cpu0 backend classes. The code of Chapter3_1 can be summaried as Fig. 3.1. Class Cpu0Subtarget provides the interfaces getInstrInfo(), getFrameLowering(), ..., to get other Cpu0 classes. Most classes (like Cpu0InstrInfo, Cpu0RegisterInfo, ...) have Subtarget reference member to allowing them access other classes through the Cpu0Subtarget interface. If the backend module hasn't the Subtarget reference, these classes still can access Subtarget class through Cpu0TargetMachine (usually use TM as symbol) by static_cast<Cpu0TargetMachine &>(TM).getSubtargetImpl(). Once getting Subtarget class, the backend code can access other classes through it. For those name of Cpu0SExx classes, they mean the standard 32 bits class. This arrangement follows llvm 3.5 Mips backend style. Mips backend uses Mips16, MipsSE and Mips64 files/classes name to define classes for 16, 32 and 64 bits architecture, respectively. Since Cpu0Subtarget creates Cpu0InstrInfo, Cpu0RegisterInfo, ..., at constuctor function, it can provide the class reference through the interfaces shown in Fig. 3.1.

Below Fig. 3.2 shows Cpu0 TableGen inheritance relationship. Last chapter mentioned that backend class can include the TableGen generated classes and inherited from it. All the TableGen generated classes of Cpu0 backend are in build/lib/Target/Cpu0/*.inc. Through C++ inheritance mechanism, TableGen provides backend programmers a fexible way to use its generated code. Programmers have chance to override this function if they need to.

Since llvm has deep inheritance tree, they are not digged here. Benefit from the inheritance tree structure, not much code needed to be implemented in classes of instruction, frame/stack and select DAG, since much code are implemented by their parent classes. The llvm-tblgen generate Cpu0GenInstrInfo.inc based on information from Cpu0InstrInfo.td. Cpu0InstrInfo.h extract those code it needs from Cpu0GenInstrInfo.inc by define "#define GET_INSTRINFO_HEADER". With TabelGen, the code size in backend is reduced again through the pattern match theory of compiler developemnt. This is explained in both sections of "DAG" and "Instruction Selection" in last chapter. Following is the code fragment from Cpu0GenInstrInfo.inc. Code between "#if def GET_INSTRINFO_HEADER" and "#endif // GET_INSTRINFO_HEADER"" will be extracted to Cpu0InstrInfo.h.

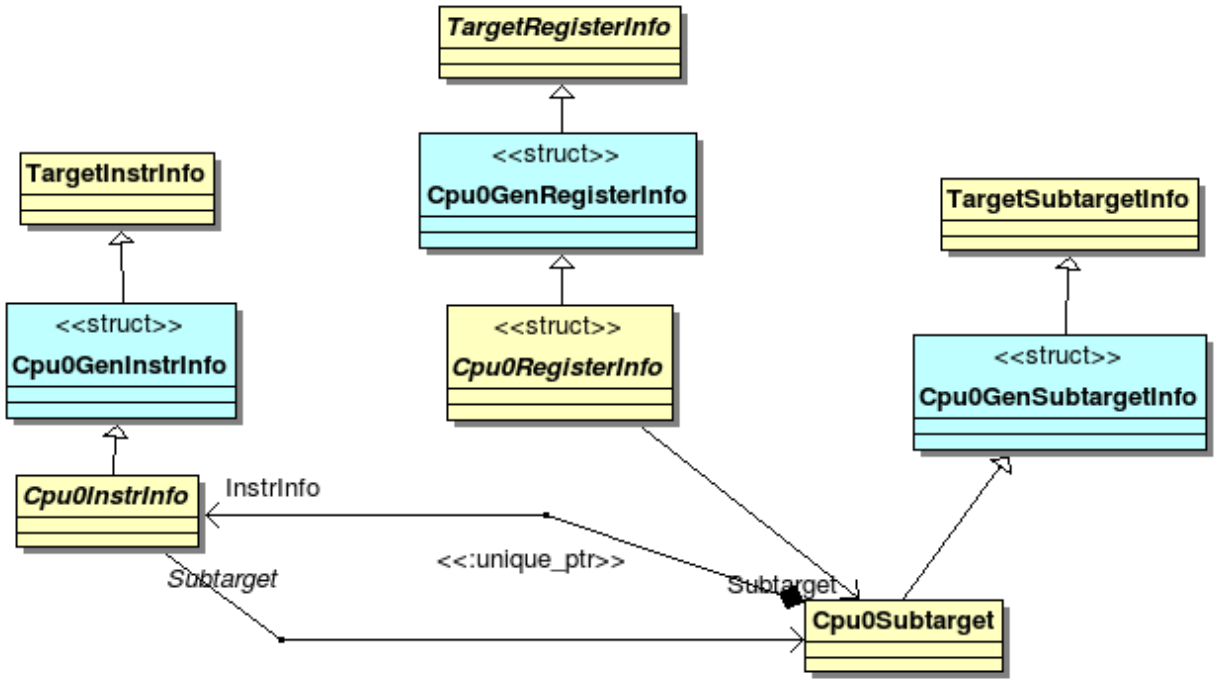


Fig. 3.2: Cpu0 classes inherited from TableGen generated files

build/lib/Target/Cpu0/Cpu0GenInstInfo.inc

```
//- Cpu0GenInstInfo.inc which generate from Cpu0InstrInfo.td
#ifdef GET_INSTRINFO_HEADER
#undef GET_INSTRINFO_HEADER
namespace llvm {
struct Cpu0GenInstrInfo : public TargetInstrInfoImpl {
  explicit Cpu0GenInstrInfo(int SO = -1, int DO = -1);
};
} // End llvm namespace
#endif // GET_INSTRINFO_HEADER
```

Reference web sites are here[1][2].

Chapter3_1/CMakeLists.txt is modified with these new added *.cpp as follows,

[1] http://llvm.org/docs/WritingAnLLVMBackend.html#target-machine
[2] http://llvm.org/docs/LangRef.html#data-layout

lbdex/chapters/Chapter3_1/CMakeLists.txt

```
tablegen(LLVM Cpu0GenDAGISel.inc -gen-dag-isel)
tablegen(LLVM Cpu0GenCallingConv.inc -gen-callingconv)
```

```
  Cpu0FrameLowering.cpp
```

Please take a look for Chapter3_1 code. After that, building Chapter3_1 by **"#define CH CH3_1"** in Cpu0Config.h as follows, and do building with cmake and make again.

~/llvm/test/llvm/lib/Target/Cpu0SetChapter.h

```
#define CH       CH3_1
```

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
... Assertion failed: (MRI && "Unable to create reg info"), function initAsmInfo
...
```

With Chapter3_1 implementation, the Chapter2 error message "Could not allocate target machine!" has gone. The new error say that we have not Target AsmPrinter and asm register info. We will add it in next section.

Chapter3_1 create FeatureCpu032I and FeatureCpu032II for CPU cpu032I and cpu032II, repectively. Beyond that, it defines two more features, FeatureCmp and FeatureSlt. In order to demostrate the "instruction set designing choice" to readers, this book creates two CPU. Readers will realize why Mips CPU uses instruction SLT instead of CMP after they have read later Chapter "Control flow statement". With the added code of supporting cpu032I and cpu32II in Cpu0.td and Cpu0InstrInfo.td of Chapter3_1, the command `llc -march=cpu0 -mcpu=help` can display messages as follows,

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -mcpu=help
Available CPUs for this target:

  cpu032I  - Select the cpu032I processor.
  cpu032II - Select the cpu032II processor.

Available features for this target:

  ch10_1   - Enable Chapter instructions..
  ch11_1   - Enable Chapter instructions..
  ch11_2   - Enable Chapter instructions..
  ch14_1   - Enable Chapter instructions..
  ch3_1    - Enable Chapter instructions..
  ch3_2    - Enable Chapter instructions..
  ch3_3    - Enable Chapter instructions..
  ch3_4    - Enable Chapter instructions..
  ch3_5    - Enable Chapter instructions..
  ch4_1    - Enable Chapter instructions..
  ch4_2    - Enable Chapter instructions..
  ch5_1    - Enable Chapter instructions..
  ch6_1    - Enable Chapter instructions..
  ch7_1    - Enable Chapter instructions..
```

(continues on next page)

(continued from previous page)

```
  ch8_1    - Enable Chapter instructions..
  ch8_2    - Enable Chapter instructions..
  ch9_1    - Enable Chapter instructions..
  ch9_2    - Enable Chapter instructions..
  ch9_3    - Enable Chapter instructions..
  chall    - Enable Chapter instructions..
  cmp      - Enable 'cmp' instructions..
  cpu032I  - Cpu032I ISA Support.
  cpu032II - Cpu032II ISA Support (slt).
  o32      - Enable o32 ABI.
  s32      - Enable s32 ABI.
  slt      - Enable 'slt' instructions..

Use +feature to enable a feature, or -feature to disable it.
For example, llc -mcpu=mycpu -mattr=+feature1,-feature2
...
```

When user input `-mcpu=cpu032I`, the variable IsCpu032I from Cpu0InstrInfo.td will be true since the function isCpu032I() defined in Cpu0Subtarget.h is true (set Cpu0ArchVersion to cpu032I in initializeSubtargetDependencies() called in constructor function, the variable CPU in constructor function is "cpu032I" when user input -mcpu=cpu032I). Please notice variable Cpu0ArchVersion must be initialized in Cpu0Subtarget.cpp, otherwise variable Cpu0ArchVersion can be any value and functions isCpu032I() and isCpu032II() which support `llc -mcpu=cpu032I` and `llc -mcpu=cpu032II`, repectively, will have trouble. The value of variables HasCmp and HasSlt are set depend on Cpu0ArchVersion. Instructions slt and beq, ... are supported only in case of HasSlt is true, and furthermore, HasSlt is true only when Cpu0ArchVersion is Cpu032II. Similiarly, Ch4_1, Ch4_2, ..., are used in controlling the enable or disable of instruction definition. Through **Subtarget->hasChapter4_1()** which exists both in Cpu0.td and Cpu0Subtarget.h, the Predicate, such as Ch4_1, defined in Cpu0InstrInfo.td can be enabled or disabled. For example, the shift-rotate instructions can be enabled by define CH to greater than or equal to CH4_1 as follows,

lbdex/Cpu0/Cpu0InstrInfo.td

```
let Predicates = [Ch4_1] in {
class shift_rotate_reg<bits<8> op, bits<4> isRotate, string instr_asm,
                       SDNode OpNode, RegisterClass RC>:
  FA<op, (outs GPROut:$ra), (ins RC:$rb, RC:$rc),
     !strconcat(instr_asm, "\t$ra, $rb, $rc"),
     [(set GPROut:$ra, (OpNode RC:$rb, RC:$rc))], IIAlu> {
  let shamt = 0;
}
}
```

~/llvm/test/llvm/lib/Target/Cpu0SetChapter.h

```
#define CH       CH4_1
```

On the contrary, it can be disabled by define it to less than CH4_1, for instance CH3_5, as follows,

~/llvm/test/llvm/lib/Target/Cpu0SetChapter.h

```
#define CH       CH3_5
```

3.2 Add AsmPrinter

Chapter3_2/ contains the Cpu0AsmPrinter definition.

lbdex/chapters/Chapter2/Cpu0.td

```
def Cpu0InstrInfo : InstrInfo;

// Will generate Cpu0GenAsmWrite.inc included by Cpu0InstPrinter.cpp, contents
//  as follows,
// void Cpu0InstPrinter::printInstruction(const MCInst *MI, raw_ostream &O) {...}
// const char *Cpu0InstPrinter::getRegisterName(unsigned RegNo) {...}
def Cpu0 : Target {
// def Cpu0InstrInfo : InstrInfo as before.
  let InstructionSet = Cpu0InstrInfo;
}
```

As above comments of Chapter2/Cpu0.td indicate, it will generate Cpu0GenAsmWrite.inc which is included by Cpu0InstPrinter.cpp as follows,

lbdex/chapters/Chapter3_2/InstPrinter/Cpu0InstPrinter.h

```
//=== Cpu0InstPrinter.h - Convert Cpu0 MCInst to assembly syntax -*- C++ -*-==//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This class prints a Cpu0 MCInst to a .s file.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_INSTPRINTER_CPU0INSTPRINTER_H
#define LLVM_LIB_TARGET_CPU0_INSTPRINTER_CPU0INSTPRINTER_H

```

(continues on next page)

(continued from previous page)

```
#include "Cpu0Config.h"

#include "llvm/MC/MCInstPrinter.h"

namespace llvm {
// These enumeration declarations were orignally in Cpu0InstrInfo.h but
// had to be moved here to avoid circular dependencies between
// LLVMCpu0CodeGen and LLVMCpu0AsmPrinter.

class TargetMachine;

class Cpu0InstPrinter : public MCInstPrinter {
public:
  Cpu0InstPrinter(const MCAsmInfo &MAI, const MCInstrInfo &MII,
                  const MCRegisterInfo &MRI)
    : MCInstPrinter(MAI, MII, MRI) {}

  // Autogenerated by tblgen.
  std::pair<const char *, uint64_t> getMnemonic(const MCInst *MI) override;
  void printInstruction(const MCInst *MI, uint64_t Address, raw_ostream &O);
  static const char *getRegisterName(unsigned RegNo);

  void printRegName(raw_ostream &OS, unsigned RegNo) const override;
  void printInst(const MCInst *MI, uint64_t Address, StringRef Annot,
                 const MCSubtargetInfo &STI, raw_ostream &O) override;

  bool printAliasInstr(const MCInst *MI, uint64_t Address, raw_ostream &OS);
  void printCustomAliasOperand(const MCInst *MI, uint64_t Address,
                               unsigned OpIdx, unsigned PrintMethodIdx,
                               raw_ostream &O);

private:
  void printOperand(const MCInst *MI, unsigned OpNo, raw_ostream &O);
  void printOperand(const MCInst *MI, uint64_t /*Address*/, unsigned OpNum,
                    raw_ostream &O) {
    printOperand(MI, OpNum, O);
  }
  void printUnsignedImm(const MCInst *MI, int opNum, raw_ostream &O);
  void printMemOperand(const MCInst *MI, int opNum, raw_ostream &O);
//#if CH >= CH7_1
  void printMemOperandEA(const MCInst *MI, int opNum, raw_ostream &O);
//#endif
};
} // end namespace llvm

#endif
```

lbdex/chapters/Chapter3_2/InstPrinter/Cpu0InstPrinter.cpp

```
//===-- Cpu0InstPrinter.cpp - Convert Cpu0 MCInst to assembly syntax ------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This class prints an Cpu0 MCInst to a .s file.
//
//===----------------------------------------------------------------------===//

#include "Cpu0InstPrinter.h"

#include "Cpu0InstrInfo.h"
#include "llvm/ADT/StringExtras.h"
#include "llvm/MC/MCExpr.h"
#include "llvm/MC/MCInst.h"
#include "llvm/MC/MCInstrInfo.h"
#include "llvm/MC/MCSymbol.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

#define DEBUG_TYPE "asm-printer"

#define PRINT_ALIAS_INSTR
#include "Cpu0GenAsmWriter.inc"

void Cpu0InstPrinter::printRegName(raw_ostream &OS, unsigned RegNo) const {
//- getRegisterName(RegNo) defined in Cpu0GenAsmWriter.inc which indicate in
//   Cpu0.td.
  OS << '$' << StringRef(getRegisterName(RegNo)).lower();
}

//@1 {
void Cpu0InstPrinter::printInst(const MCInst *MI, uint64_t Address,
                                StringRef Annot, const MCSubtargetInfo &STI,
                                raw_ostream &O) {
  // Try to print any aliases first.
  if (!printAliasInstr(MI, Address, O))
//@1 }
    //- printInstruction(MI, O) defined in Cpu0GenAsmWriter.inc which came from
    //   Cpu0.td indicate.
    printInstruction(MI, Address, O);
  printAnnotation(O, Annot);
}

void Cpu0InstPrinter::printOperand(const MCInst *MI, unsigned OpNo,
                                   raw_ostream &O) {
```

(continues on next page)

(continued from previous page)

```
  const MCOperand &Op = MI->getOperand(OpNo);
  if (Op.isReg()) {
    printRegName(O, Op.getReg());
    return;
  }

  if (Op.isImm()) {
    O << Op.getImm();
    return;
  }

  assert(Op.isExpr() && "unknown operand kind in printOperand");
  Op.getExpr()->print(O, &MAI, true);
}

void Cpu0InstPrinter::printUnsignedImm(const MCInst *MI, int opNum,
                                       raw_ostream &O) {
  const MCOperand &MO = MI->getOperand(opNum);
  if (MO.isImm())
    O << (unsigned short int)MO.getImm();
  else
    printOperand(MI, opNum, O);
}

void Cpu0InstPrinter::
printMemOperand(const MCInst *MI, int opNum, raw_ostream &O) {
  // Load/Store memory operands -- imm($reg)
  // If PIC target the target is loaded as the
  // pattern ld $t9,%call16($gp)
  printOperand(MI, opNum+1, O);
  O << "(";
  printOperand(MI, opNum, O);
  O << ")";
}

//#if CH >= CH7_1
// The DAG data node, mem_ea of Cpu0InstrInfo.td, cannot be disabled by
// ch7_1, only opcode node can be disabled.
void Cpu0InstPrinter::
printMemOperandEA(const MCInst *MI, int opNum, raw_ostream &O) {
  // when using stack locations for not load/store instructions
  // print the same way as all normal 3 operand instructions.
  printOperand(MI, opNum, O);
  O << ", ";
  printOperand(MI, opNum+1, O);
  return;
}
//#endif
```

lbdex/chapters/Chapter3_2/InstPrinter/CMakeLists.txt

```
add_llvm_component_library(LLVMCpu0AsmPrinter
  Cpu0InstPrinter.cpp

  LINK_COMPONENTS
  Support

  ADD_TO_COMPONENT
  Cpu0
  )
```

Cpu0GenAsmWrite.inc has the implementations of Cpu0InstPrinter::printInstruction() and Cpu0InstPrinter::getRegisterName(). Both of these functions can be auto-generated from the information we defined in Cpu0InstrInfo.td and Cpu0RegisterInfo.td. To let these two functions work in our code, the only thing needed is adding a class Cpu0InstPrinter and include them as did in Chapter3_2.

File Chapter3_2/Cpu0/InstPrinter/Cpu0InstPrinter.cpp include Cpu0GenAsmWrite.inc and call the auto-generated functions from TableGen.

Function Cpu0InstPrinter::printMemOperand() defined in Chapter3_2/InstPrinter/Cpu0InstPrinter.cpp as above. It will be triggered since Cpu0InstrInfo.td defined **'let PrintMethod = "printMemOperand";'** as follows,

lbdex/chapters/Chapter2/Cpu0InstrInfo.td

```
// Address operand
def mem : Operand<i32> {
  let PrintMethod = "printMemOperand";
  let MIOperandInfo = (ops CPURegs, simm16);
  let EncoderMethod = "getMemEncoding";
}
...
// 32-bit load.
multiclass LoadM32<bits<8> op, string instr_asm, PatFrag OpNode,
                   bit Pseudo = 0> {
  def #NAME# : LoadM<op, instr_asm, OpNode, GPROut, mem, Pseudo>;
}

// 32-bit store.
multiclass StoreM32<bits<8> op, string instr_asm, PatFrag OpNode,
                    bit Pseudo = 0> {
  def #NAME# : StoreM<op, instr_asm, OpNode, CPURegs, mem, Pseudo>;
}

defm LD     : LoadM32<0x01,  "ld",  load_a>;
defm ST     : StoreM32<0x02, "st",  store_a>;
```

Cpu0InstPrinter::printMemOperand() will print backend operands for "local variable access", which is like the following,

```
ld    $2, 16($fp)
st    $2, 8($fp)
```

Next, add Cpu0MCInstLower (Cpu0MCInstLower.h, Cpu0MCInstLower.cpp) as well as Cpu0BaseInfo.h, Cpu0FixupKinds.h and Cpu0MCAsmInfo (Cpu0MCAsmInfo.h, Cpu0MCAsmInfo.cpp) in sub-directory MCTargetDesc as follows,

lbdex/chapters/Chapter3_2/Cpu0MCInstLower.h

```
//===-- Cpu0MCInstLower.h - Lower MachineInstr to MCInst -------*- C++ -*--===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0MCINSTLOWER_H
#define LLVM_LIB_TARGET_CPU0_CPU0MCINSTLOWER_H

#include "Cpu0Config.h"

#include "llvm/ADT/SmallVector.h"
#include "llvm/CodeGen/MachineOperand.h"
#include "llvm/Support/Compiler.h"

namespace llvm {
  class MCContext;
  class MCInst;
  class MCOperand;
  class MachineInstr;
  class MachineFunction;
  class Cpu0AsmPrinter;

//@1 {
/// This class is used to lower an MachineInstr into an MCInst.
class LLVM_LIBRARY_VISIBILITY Cpu0MCInstLower {
//@2
  typedef MachineOperand::MachineOperandType MachineOperandType;
  MCContext *Ctx;
  Cpu0AsmPrinter &AsmPrinter;
public:
  Cpu0MCInstLower(Cpu0AsmPrinter &asmprinter);
  void Initialize(MCContext* C);
  void Lower(const MachineInstr *MI, MCInst &OutMI) const;
  MCOperand LowerOperand(const MachineOperand& MO, unsigned offset = 0) const;
};
}

#endif
```

lbdex/chapters/Chapter3_2/Cpu0MCInstLower.cpp

```
//===-- Cpu0MCInstLower.cpp - Convert Cpu0 MachineInstr to MCInst ---------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains code to lower Cpu0 MachineInstrs to their corresponding
// MCInst records.
//
//===----------------------------------------------------------------------===//

#include "Cpu0MCInstLower.h"

#include "Cpu0AsmPrinter.h"
#include "Cpu0InstrInfo.h"
#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstr.h"
#include "llvm/CodeGen/MachineOperand.h"
#include "llvm/IR/Mangler.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCExpr.h"
#include "llvm/MC/MCInst.h"

using namespace llvm;

Cpu0MCInstLower::Cpu0MCInstLower(Cpu0AsmPrinter &asmprinter)
  : AsmPrinter(asmprinter) {}

void Cpu0MCInstLower::Initialize(MCContext* C) {
  Ctx = C;
}

static void CreateMCInst(MCInst& Inst, unsigned Opc, const MCOperand& Opnd0,
                         const MCOperand& Opnd1,
                         const MCOperand& Opnd2 = MCOperand()) {
  Inst.setOpcode(Opc);
  Inst.addOperand(Opnd0);
  Inst.addOperand(Opnd1);
  if (Opnd2.isValid())
    Inst.addOperand(Opnd2);
}

//@LowerOperand {
MCOperand Cpu0MCInstLower::LowerOperand(const MachineOperand& MO,
                                        unsigned offset) const {
  MachineOperandType MOTy = MO.getType();
```

(continues on next page)

(continued from previous page)

```
  switch (MOTy) {
  //@2
  default: llvm_unreachable("unknown operand type");
  case MachineOperand::MO_Register:
    // Ignore all implicit register operands.
    if (MO.isImplicit()) break;
    return MCOperand::createReg(MO.getReg());
  case MachineOperand::MO_Immediate:
    return MCOperand::createImm(MO.getImm() + offset);
  case MachineOperand::MO_RegisterMask:
    break;
 }

  return MCOperand();
}

void Cpu0MCInstLower::Lower(const MachineInstr *MI, MCInst &OutMI) const {
  OutMI.setOpcode(MI->getOpcode());

  for (unsigned i = 0, e = MI->getNumOperands(); i != e; ++i) {
    const MachineOperand &MO = MI->getOperand(i);
    MCOperand MCOp = LowerOperand(MO);

    if (MCOp.isValid())
      OutMI.addOperand(MCOp);
  }
}
```

lbdex/chapters/Chapter3_2/MCTargetDesc/Cpu0BaseInfo.h

```
//===-- Cpu0BaseInfo.h - Top level definitions for CPU0 MC ------*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains small standalone helper functions and enum definitions for
// the Cpu0 target useful for the compiler back-end and the MC libraries.
//
//===----------------------------------------------------------------------===//
#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0BASEINFO_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0BASEINFO_H

#include "Cpu0Config.h"

#include "Cpu0MCTargetDesc.h"
```

(continues on next page)

(continued from previous page)

```
#include "llvm/MC/MCExpr.h"
#include "llvm/Support/DataTypes.h"
#include "llvm/Support/ErrorHandling.h"

namespace llvm {

/// Cpu0II - This namespace holds all of the target specific flags that
/// instruction info tracks.
//@Cpu0II
namespace Cpu0II {
  /// Target Operand Flag enum.
  enum TOF {
    //===----------------------------------------------------------------------===//
    // Cpu0 Specific MachineOperand flags.

    MO_NO_FLAG,

    /// MO_GOT_CALL - Represents the offset into the global offset table at
    /// which the address of a call site relocation entry symbol resides
    /// during execution. This is different from the above since this flag
    /// can only be present in call instructions.
    MO_GOT_CALL,

    /// MO_GPREL - Represents the offset from the current gp value to be used
    /// for the relocatable object file being produced.
    MO_GPREL,

    /// MO_ABS_HI/LO - Represents the hi or low part of an absolute symbol
    /// address.
    MO_ABS_HI,
    MO_ABS_LO,

    /// MO_GOT_HI16/LO16 - Relocations used for large GOTs.
    MO_GOT_HI16,
    MO_GOT_LO16
  }; // enum TOF {

  enum {
    //===----------------------------------------------------------------------===//
    // Instruction encodings.  These are the standard/most common forms for
    // Cpu0 instructions.
    //

    // Pseudo - This represents an instruction that is a pseudo instruction
    // or one that has not been implemented yet.  It is illegal to code generate
    // it, but tolerated for intermediate implementation stages.
    Pseudo   = 0,

    /// FrmR - This form is for instructions of the format R.
    FrmR  = 1,
    /// FrmI - This form is for instructions of the format I.
    FrmI  = 2,
```

(continues on next page)

(continued from previous page)

```
    /// FrmJ - This form is for instructions of the format J.
    FrmJ  = 3,
    /// FrmOther - This form is for instructions that have no specific format.
    FrmOther = 4,

    FormMask = 15
  };
}

}

#endif
```

lbdex/chapters/Chapter3_2/Cpu0MCAsmInfo.h

```
//===-- Cpu0MCAsmInfo.h - Cpu0 Asm Info ------------------------*- C++ -*--===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the declaration of the Cpu0MCAsmInfo class.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCASMINFO_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCASMINFO_H

#include "Cpu0Config.h"
#if CH >= CH3_2

#include "llvm/MC/MCAsmInfoELF.h"

namespace llvm {
  class Triple;

  class Cpu0MCAsmInfo : public MCAsmInfoELF {
    void anchor() override;
  public:
    explicit Cpu0MCAsmInfo(const Triple &TheTriple);
  };

} // namespace llvm

#endif // #if CH >= CH3_2

#endif
```

lbdex/chapters/Chapter3_2/Cpu0MCAsmInfo.cpp

```
//===-- Cpu0MCAsmInfo.cpp - Cpu0 Asm Properties ----------------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the declarations of the Cpu0MCAsmInfo properties.
//
//===----------------------------------------------------------------------===//

#include "Cpu0MCAsmInfo.h"
#if CH >= CH3_2

#include "llvm/ADT/Triple.h"

using namespace llvm;

void Cpu0MCAsmInfo::anchor() { }

Cpu0MCAsmInfo::Cpu0MCAsmInfo(const Triple &TheTriple) {
  if ((TheTriple.getArch() == Triple::cpu0))
    IsLittleEndian = false; // the default of IsLittleEndian is true

  AlignmentIsInBytes          = false;
  Data16bitsDirective         = "\t.2byte\t";
  Data32bitsDirective         = "\t.4byte\t";
  Data64bitsDirective         = "\t.8byte\t";
  PrivateGlobalPrefix         = "$";
// PrivateLabelPrefix: display $BB for the labels of basic block
  PrivateLabelPrefix          = "$";
  CommentString               = "#";
  ZeroDirective               = "\t.space\t";
  GPRel32Directive            = "\t.gpword\t";
  GPRel64Directive            = "\t.gpdword\t";
  WeakRefDirective            = "\t.weak\t";
  UseAssignmentForEHBegin = true;

  SupportsDebugInformation = true;
  ExceptionsType = ExceptionHandling::DwarfCFI;
  DwarfRegNumForCFI = true;
}

#endif // #if CH >= CH3_2
```

Finally, add code in Cpu0MCTargetDesc.cpp to register Cpu0InstPrinter as below. It also registers other classes (register, instruction and subtarget) which defined in Chapter3_1 at this point.

lbdex/chapters/Chapter3_2/MCTargetDesc/Cpu0MCTargetDesc.h

```
namespace llvm {
class MCAsmBackend;
class MCCodeEmitter;
class MCContext;
class MCInstrInfo;
class MCObjectWriter;
class MCRegisterInfo;
class MCSubtargetInfo;
class StringRef;
...
class raw_ostream;
...
}
```

lbdex/chapters/Chapter3_2/MCTargetDesc/Cpu0MCTargetDesc.cpp

```
#include "InstPrinter/Cpu0InstPrinter.h"
#include "Cpu0MCAsmInfo.h"
```

```
/// Select the Cpu0 Architecture Feature for the given triple and cpu name.
/// The function will be called at command 'llvm-objdump -d' for Cpu0 elf input.
static std::string selectCpu0ArchFeature(const Triple &TT, StringRef CPU) {
  std::string Cpu0ArchFeature;
  if (CPU.empty() || CPU == "generic") {
    if (TT.getArch() == Triple::cpu0 || TT.getArch() == Triple::cpu0el) {
      if (CPU.empty() || CPU == "cpu032II") {
        Cpu0ArchFeature = "+cpu032II";
      }
      else {
        if (CPU == "cpu032I") {
          Cpu0ArchFeature = "+cpu032I";
        }
      }
    }
  }
  return Cpu0ArchFeature;
}
//@1 }

static MCInstrInfo *createCpu0MCInstrInfo() {
  MCInstrInfo *X = new MCInstrInfo();
  InitCpu0MCInstrInfo(X); // defined in Cpu0GenInstrInfo.inc
  return X;
}

static MCRegisterInfo *createCpu0MCRegisterInfo(const Triple &TT) {
  MCRegisterInfo *X = new MCRegisterInfo();
  InitCpu0MCRegisterInfo(X, Cpu0::SW); // defined in Cpu0GenRegisterInfo.inc
  return X;
```

(continues on next page)

(continued from previous page)

```
}

static MCSubtargetInfo *createCpu0MCSubtargetInfo(const Triple &TT,
                                                  StringRef CPU, StringRef FS) {
  std::string ArchFS = selectCpu0ArchFeature(TT,CPU);
  if (!FS.empty()) {
    if (!ArchFS.empty())
      ArchFS = ArchFS + "," + FS.str();
    else
      ArchFS = FS.str();
  }
  return createCpu0MCSubtargetInfoImpl(TT, CPU, /*TuneCPU*/ CPU, ArchFS);
// createCpu0MCSubtargetInfoImpl defined in Cpu0GenSubtargetInfo.inc
}

static MCAsmInfo *createCpu0MCAsmInfo(const MCRegisterInfo &MRI,
                                      const Triple &TT,
                                      const MCTargetOptions &Options) {
  MCAsmInfo *MAI = new Cpu0MCAsmInfo(TT);

  unsigned SP = MRI.getDwarfRegNum(Cpu0::SP, true);
  MCCFIInstruction Inst = MCCFIInstruction::createDefCfaRegister(nullptr, SP);
  MAI->addInitialFrameState(Inst);

  return MAI;
}

static MCInstPrinter *createCpu0MCInstPrinter(const Triple &T,
                                              unsigned SyntaxVariant,
                                              const MCAsmInfo &MAI,
                                              const MCInstrInfo &MII,
                                              const MCRegisterInfo &MRI) {
  return new Cpu0InstPrinter(MAI, MII, MRI);
}

namespace {

class Cpu0MCInstrAnalysis : public MCInstrAnalysis {
public:
  Cpu0MCInstrAnalysis(const MCInstrInfo *Info) : MCInstrAnalysis(Info) {}
};
}

static MCInstrAnalysis *createCpu0MCInstrAnalysis(const MCInstrInfo *Info) {
  return new Cpu0MCInstrAnalysis(Info);
}

//@2 {
extern "C" void LLVMInitializeCpu0TargetMC() {
  for (Target *T : {&TheCpu0Target, &TheCpu0elTarget}) {
    // Register the MC asm info.
    RegisterMCAsmInfoFn X(*T, createCpu0MCAsmInfo);
```

(continues on next page)

(continued from previous page)

```
    // Register the MC instruction info.
    TargetRegistry::RegisterMCInstrInfo(*T, createCpu0MCInstrInfo);

    // Register the MC register info.
    TargetRegistry::RegisterMCRegInfo(*T, createCpu0MCRegisterInfo);

    // Register the MC subtarget info.
    TargetRegistry::RegisterMCSubtargetInfo(*T,
                                            createCpu0MCSubtargetInfo);
    // Register the MC instruction analyzer.
    TargetRegistry::RegisterMCInstrAnalysis(*T, createCpu0MCInstrAnalysis);
    // Register the MCInstPrinter.
    TargetRegistry::RegisterMCInstPrinter(*T,
                                          createCpu0MCInstPrinter);
  }

}
//@2 }
```

lbdex/chapters/Chapter3_2/MCTargetDesc/CMakeLists.txt

```
  Cpu0MCAsmInfo.cpp
```

To make the registration clearly, summary as the following diagram, Fig. 3.3.

Above createCpu0MCAsmInfo() registering the object of class Cpu0MCAsmInfo for target TheCpu0Target and TheCpu0elTarget. TheCpu0Target is for big endian and TheCpu0elTarget is for little endian. Cpu0MCAsmInfo is derived from MCAsmInfo which is an llvm built-in class. Most code is implemented in it's parent, backend reuses those code by inheritance.

Above createCpu0MCInstrInfo() instancing MCInstrInfo object X, and initialize it by InitCpu0MCInstrInfo(X). Since InitCpu0MCInstrInfo(X) is defined in Cpu0GenInstrInfo.inc, this function will add the information from Cpu0InstrInfo.td we specified.

Above createCpu0MCInstPrinter() instancing Cpu0InstPrinter to take care printing function for instructions.

Above createCpu0MCRegisterInfo() is similar to "Register function of MC instruction info", but it initializes the register information specified in Cpu0RegisterInfo.td. They share some values from instruction/register td description, so no need to specify them again in Initialize routine if they are consistant with td description files.

Above createCpu0MCSubtargetInfo() instancing MCSubtargetInfo object and initialize with Cpu0.td information.

According "section Target Registration"[3], we can register Cpu0 backend classes at LLVMInitializeCpu0TargetMC() on demand by the dynamic register mechanism as the above function, LLVMInitializeCpu0TargetMC().

Now, it's time to work with AsmPrinter as follows,

[3] http://jonathan2251.github.io/lbd/llvmstructure.html#target-registration

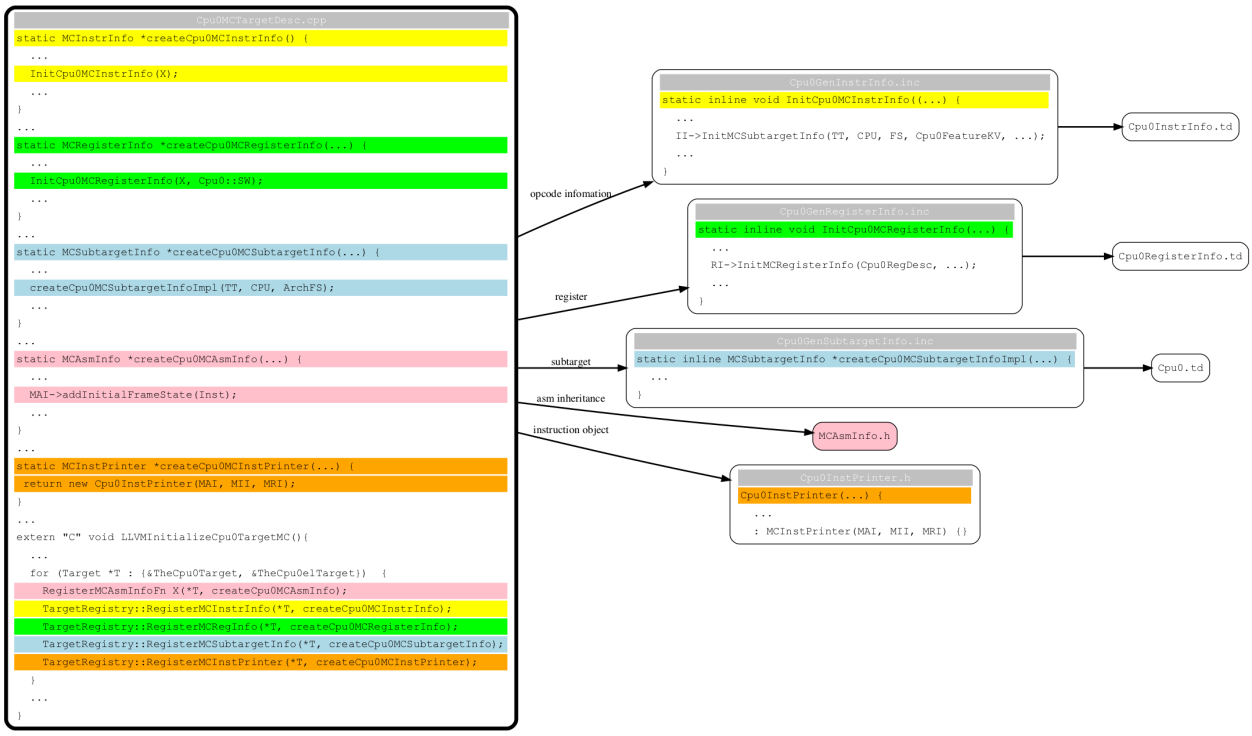


Fig. 3.3: Tblgen generate files for Cpu0 backend

lbdex/chapters/Chapter3_2/Cpu0AsmPrinter.h

```
//===-- Cpu0AsmPrinter.h - Cpu0 LLVM Assembly Printer ----------*- C++ -*--===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// Cpu0 Assembly printer class.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0ASMPRINTER_H
#define LLVM_LIB_TARGET_CPU0_CPU0ASMPRINTER_H

#include "Cpu0Config.h"

#include "Cpu0MachineFunction.h"
#include "Cpu0MCInstLower.h"
#include "Cpu0Subtarget.h"
#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/AsmPrinter.h"
#include "llvm/MC/MCStreamer.h"
```

(continues on next page)

(continued from previous page)

```
#include "llvm/Support/Compiler.h"
#include "llvm/Target/TargetMachine.h"

namespace llvm {
class MCStreamer;
class MachineInstr;
class MachineBasicBlock;
class Module;
class raw_ostream;

class LLVM_LIBRARY_VISIBILITY Cpu0AsmPrinter : public AsmPrinter {

  void EmitInstrWithMacroNoAT(const MachineInstr *MI);

private:

  // lowerOperand - Convert a MachineOperand into the equivalent MCOperand.
  bool lowerOperand(const MachineOperand &MO, MCOperand &MCOp);

public:

  const Cpu0Subtarget *Subtarget;
  const Cpu0FunctionInfo *Cpu0FI;
  Cpu0MCInstLower MCInstLowering;

  explicit Cpu0AsmPrinter(TargetMachine &TM,
                          std::unique_ptr<MCStreamer> Streamer)
    : AsmPrinter(TM, std::move(Streamer)),
      MCInstLowering(*this) {
    Subtarget = static_cast<Cpu0TargetMachine &>(TM).getSubtargetImpl();
  }

  StringRef getPassName() const override {
    return "Cpu0 Assembly Printer";
  }

  virtual bool runOnMachineFunction(MachineFunction &MF) override;

//- emitInstruction() must exists or will have run time error.
  void emitInstruction(const MachineInstr *MI) override;
  void printSavedRegsBitmask(raw_ostream &O);
  void printHex32(unsigned int Value, raw_ostream &O);
  void emitFrameDirective();
  const char *getCurrentABIString() const;
  void emitFunctionEntryLabel() override;
  void emitFunctionBodyStart() override;
  void emitFunctionBodyEnd() override;
  void emitStartOfAsmFile(Module &M) override;
  void PrintDebugValueComment(const MachineInstr *MI, raw_ostream &OS);
};
}
```

(continues on next page)

(continued from previous page)

```
#endif
```

lbdex/chapters/Chapter3_2/Cpu0AsmPrinter.cpp

```
//===-- Cpu0AsmPrinter.cpp - Cpu0 LLVM Assembly Printer -------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains a printer that converts from our internal representation
// of machine-dependent LLVM code to GAS-format CPU0 assembly language.
//
//===----------------------------------------------------------------------===//

#include "Cpu0AsmPrinter.h"

#include "InstPrinter/Cpu0InstPrinter.h"
#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "Cpu0.h"
#include "Cpu0InstrInfo.h"
#include "llvm/ADT/SmallString.h"
#include "llvm/ADT/StringExtras.h"
#include "llvm/ADT/Twine.h"
#include "llvm/CodeGen/MachineConstantPool.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineInstr.h"
#include "llvm/CodeGen/MachineMemOperand.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/Mangler.h"
#include "llvm/MC/MCAsmInfo.h"
#include "llvm/MC/MCInst.h"
#include "llvm/MC/MCStreamer.h"
#include "llvm/MC/MCSymbol.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/TargetRegistry.h"
#include "llvm/Target/TargetLoweringObjectFile.h"
#include "llvm/Target/TargetOptions.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-asm-printer"

bool Cpu0AsmPrinter::runOnMachineFunction(MachineFunction &MF) {
```

(continues on next page)

(continued from previous page)

```
  Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
  AsmPrinter::runOnMachineFunction(MF);
  return true;
}

//@EmitInstruction {
//- emitInstruction() must exists or will have run time error.
void Cpu0AsmPrinter::emitInstruction(const MachineInstr *MI) {
//@EmitInstruction body {
  if (MI->isDebugValue()) {
    SmallString<128> Str;
    raw_svector_ostream OS(Str);

    PrintDebugValueComment(MI, OS);
    return;
  }

  //@print out instruction:
  //  Print out both ordinary instruction and boudle instruction
  MachineBasicBlock::const_instr_iterator I = MI->getIterator();
  MachineBasicBlock::const_instr_iterator E = MI->getParent()->instr_end();

  do {

    if (I->isPseudo())
      llvm_unreachable("Pseudo opcode found in emitInstruction()");

    MCInst TmpInst0;
    MCInstLowering.Lower(&*I, TmpInst0);
    OutStreamer->emitInstruction(TmpInst0, getSubtargetInfo());
  } while ((++I != E) && I->isInsideBundle()); // Delay slot check
}
//@EmitInstruction }

//===----------------------------------------------------------------------===//
//
//  Cpu0 Asm Directives
//
//  -- Frame directive "frame Stackpointer, Stacksize, RARegister"
//  Describe the stack frame.
//
//  -- Mask directives "(f)mask  bitmask, offset"
//  Tells the assembler which registers are saved and where.
//  bitmask - contain a little endian bitset indicating which registers are
//            saved on function prologue (e.g. with a 0x80000000 mask, the
//            assembler knows the register 31 (RA) is saved at prologue.
//  offset  - the position before stack pointer subtraction indicating where
//            the first saved register on prologue is located. (e.g. with a
//
//  Consider the following function prologue:
//
//    .frame  $fp,48,$ra
```

(continues on next page)

(continued from previous page)

```
//    .mask   0xc0000000,-8
//       addiu $sp, $sp, -48
//       st $ra, 40($sp)
//       st $fp, 36($sp)
//
//    With a 0xc0000000 mask, the assembler knows the register 31 (RA) and
//    30 (FP) are saved at prologue. As the save order on prologue is from
//    left to right, RA is saved first. A -8 offset means that after the
//    stack pointer subtration, the first register in the mask (RA) will be
//    saved at address 48-8=40.
//
//===----------------------------------------------------------------------===//

//===----------------------------------------------------------------------===//
// Mask directives
//===----------------------------------------------------------------------===//
//        .frame        $sp,8,$lr
//->        .mask         0x00000000,0
//        .set        noreorder
//        .set        nomacro

// Create a bitmask with all callee saved registers for CPU or Floating Point
// registers. For CPU registers consider LR, GP and FP for saving if necessary.
void Cpu0AsmPrinter::printSavedRegsBitmask(raw_ostream &O) {
  // CPU and FPU Saved Registers Bitmasks
  unsigned CPUBitmask = 0;
  int CPUTopSavedRegOff;

  // Set the CPU and FPU Bitmasks
  const MachineFrameInfo &MFI = MF->getFrameInfo();
  const TargetRegisterInfo *TRI = MF->getSubtarget().getRegisterInfo();
  const std::vector<CalleeSavedInfo> &CSI = MFI.getCalleeSavedInfo();
  // size of stack area to which FP callee-saved regs are saved.
  unsigned CPURegSize = TRI->getRegSizeInBits(Cpu0::CPURegsRegClass) / 8;
  unsigned i = 0, e = CSI.size();

  // Set CPU Bitmask.
  for (; i != e; ++i) {
    unsigned Reg = CSI[i].getReg();
    unsigned RegNum = TRI->getEncodingValue(Reg);
    CPUBitmask |= (1 << RegNum);
  }

  CPUTopSavedRegOff = CPUBitmask ? -CPURegSize : 0;

  // Print CPUBitmask
  O << "\t.mask \t"; printHex32(CPUBitmask, O);
  O << ',' << CPUTopSavedRegOff << '\n';
}

// Print a 32 bit hex number with all numbers.
void Cpu0AsmPrinter::printHex32(unsigned Value, raw_ostream &O) {
```

(continues on next page)

(continued from previous page)

```
  O << "0x";
  for (int i = 7; i >= 0; i--)
    O.write_hex((Value & (0xF << (i*4))) >> (i*4));
}

//===----------------------------------------------------------------------===//
// Frame and Set directives
//===----------------------------------------------------------------------===//
//->        .frame        $sp,8,$lr
//        .mask         0x00000000,0
//        .set        noreorder
//        .set        nomacro
/// Frame Directive
void Cpu0AsmPrinter::emitFrameDirective() {
  const TargetRegisterInfo &RI = *MF->getSubtarget().getRegisterInfo();

  unsigned stackReg  = RI.getFrameRegister(*MF);
  unsigned returnReg = RI.getRARegister();
  unsigned stackSize = MF->getFrameInfo().getStackSize();

  if (OutStreamer->hasRawTextSupport())
    OutStreamer->emitRawText("\t.frame\t$" +
           StringRef(Cpu0InstPrinter::getRegisterName(stackReg)).lower() +
           "," + Twine(stackSize) + ",$" +
           StringRef(Cpu0InstPrinter::getRegisterName(returnReg)).lower());
}

/// Emit Set directives.
const char *Cpu0AsmPrinter::getCurrentABIString() const {
  switch (static_cast<Cpu0TargetMachine &>(TM).getABI().GetEnumValue()) {
  case Cpu0ABIInfo::ABI::O32:  return "abiO32";
  case Cpu0ABIInfo::ABI::S32:  return "abiS32";
  default: llvm_unreachable("Unknown Cpu0 ABI");
  }
}

//                .type        main,@function
//->               .ent        main                    # @main
//        main:
void Cpu0AsmPrinter::emitFunctionEntryLabel() {
  if (OutStreamer->hasRawTextSupport())
    OutStreamer->emitRawText("\t.ent\t" + Twine(CurrentFnSym->getName()));
  OutStreamer->emitLabel(CurrentFnSym);
}

//  .frame  $sp,8,$pc
//  .mask   0x00000000,0
//->  .set  noreorder
//@-> .set  nomacro
/// EmitFunctionBodyStart - Targets can override this to emit stuff before
/// the first basic block in the function.
void Cpu0AsmPrinter::emitFunctionBodyStart() {
```

(continues on next page)

(continued from previous page)

```
  MCInstLowering.Initialize(&MF->getContext());

  emitFrameDirective();

  if (OutStreamer->hasRawTextSupport()) {
    SmallString<128> Str;
    raw_svector_ostream OS(Str);
    printSavedRegsBitmask(OS);
    OutStreamer->emitRawText(OS.str());
    OutStreamer->emitRawText(StringRef("\t.set\tnoreorder"));
    OutStreamer->emitRawText(StringRef("\t.set\tnomacro"));
    if (Cpu0FI->getEmitNOAT())
      OutStreamer->emitRawText(StringRef("\t.set\tnoat"));
  }
}

//->        .set        macro
//->        .set        reorder
//->        .end        main
/// EmitFunctionBodyEnd - Targets can override this to emit stuff after
/// the last basic block in the function.
void Cpu0AsmPrinter::emitFunctionBodyEnd() {
  // There are instruction for this macros, but they must
  // always be at the function end, and we can't emit and
  // break with BB logic.
  if (OutStreamer->hasRawTextSupport()) {
    if (Cpu0FI->getEmitNOAT())
      OutStreamer->emitRawText(StringRef("\t.set\tat"));
    OutStreamer->emitRawText(StringRef("\t.set\tmacro"));
    OutStreamer->emitRawText(StringRef("\t.set\treorder"));
    OutStreamer->emitRawText("\t.end\t" + Twine(CurrentFnSym->getName()));
  }
}

//        .section .mdebug.abi32
//        .previous
void Cpu0AsmPrinter::emitStartOfAsmFile(Module &M) {
  // FIXME: Use SwitchSection.

  // Tell the assembler which ABI we are using
  if (OutStreamer->hasRawTextSupport())
    OutStreamer->emitRawText("\t.section .mdebug." +
                             Twine(getCurrentABIString()));

  // return to previous section
  if (OutStreamer->hasRawTextSupport())
    OutStreamer->emitRawText(StringRef("\t.previous"));
}

void Cpu0AsmPrinter::PrintDebugValueComment(const MachineInstr *MI,
                                            raw_ostream &OS) {
  // TODO: implement
```

(continues on next page)

(continued from previous page)

```
  OS << "PrintDebugValueComment()";
}

// Force static initialization.
extern "C" void LLVMInitializeCpu0AsmPrinter() {
  RegisterAsmPrinter<Cpu0AsmPrinter> X(TheCpu0Target);
  RegisterAsmPrinter<Cpu0AsmPrinter> Y(TheCpu0elTarget);
}
```

When instruction is ready to print, function Cpu0AsmPrinter::EmitInstruction() will be triggered first. And then it will call OutStreamer.EmitInstruction() to print OP code and register name according the information from Cpu0GenInstrInfo.inc and Cpu0GenRegisterInfo.inc both registered at dynamic register function, LLVMInitializeCpu0TargetMC(). Notice, file Cpu0InstPrinter.cpp only print operand while the OP code information come from Cpu0InstrInfo.td.

Add the following code to Cpu0ISelLowering.cpp.

lbdex/chapters/Chapter3_2/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::
Cpu0TargetLowering(Cpu0TargetMachine &TM)
  : TargetLowering(TM, new Cpu0TargetObjectFile()),
    Subtarget(&TM.getSubtarget<Cpu0Subtarget>()) {

//- Set .align 2
// It will emit .align 2 later
  setMinFunctionAlignment(2);

// must, computeRegisterProperties - Once all of the register classes are
//  added, this allows us to compute derived properties we expose.
  computeRegisterProperties();
}
```

Add the following code to Cpu0MachineFunction.h since the Cpu0AsmPrinter.cpp will call getEmitNOAT().

lbdex/chapters/Chapter3_2/Cpu0MachineFunction.h

```
class Cpu0FunctionInfo : public MachineFunctionInfo {
public:
  Cpu0FunctionInfo(MachineFunction& MF)
  : ...
    , EmitNOAT(false)
    {}

  ...
  bool getEmitNOAT() const { return EmitNOAT; }
  void setEmitNOAT() { EmitNOAT = true; }
private:
  ...
```

(continues on next page)

(continued from previous page)

```
  bool EmitNOAT;
};
```

lbdex/chapters/Chapter3_2/CMakeLists.txt

```
tablegen(LLVM Cpu0GenCodeEmitter.inc -gen-emitter)
tablegen(LLVM Cpu0GenMCCodeEmitter.inc -gen-emitter)

tablegen(LLVM Cpu0GenAsmWriter.inc -gen-asm-writer)
```

```
...
add_llvm_target(Cpu0CodeGen
```

```
  Cpu0AsmPrinter.cpp
  Cpu0MCInstLower.cpp
```

```
LINK_COMPONENTS

...
```

```
  Cpu0AsmPrinter
```

```
  ...
  )
...
```

```
add_subdirectory(InstPrinter)
```

Now, run Chapter3_2/Cpu0 for AsmPrinter support, will get new error message as follows,

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
/Users/Jonathan/llvm/test/build/bin/llc: target does not
support generation of this file type!
```

The `llc` fails to compile IR code into machine code since we don't implement class Cpu0DAGToDAGISel.

3.3 Add Cpu0DAGToDAGISel class

The IR DAG to machine instruction DAG transformation is introduced in the previous chapter. Now, let's check what IR DAG nodes the file ch3.bc has. List ch3.ll as follows,

```
// ch3.ll
define i32 @main() nounwind uwtable {
%1 = alloca i32, align 4
store i32 0, i32* %1
```

(continues on next page)

(continued from previous page)

```
ret i32 0
}
```

As above, ch3.ll uses the IR DAG node **store**, **ret**. So, the definitions in Cpu0InstrInfo.td as below is enough. The ADDiu used for stack adjustment which will be needed in later section "Add Prologue/Epilogue functions" of this chapter. IR DAG is defined in file include/llvm/Target/TargetSelectionDAG.td.

lbdex/chapters/Chapter2/Cpu0InstrInfo.td

```
//===----------------------------------------------------------------------===//

/// Load and Store Instructions
///  aligned
defm LD     : LoadM32<0x01,  "ld",  load_a>;
defm ST     : StoreM32<0x02, "st",  store_a>;

/// Arithmetic Instructions (ALU Immediate)
// IR "add" defined in include/llvm/Target/TargetSelectionDAG.td, line 315 (def add).
def ADDiu   : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;

let isReturn=1, isTerminator=1, hasDelaySlot=1, isBarrier=1, hasCtrlDep=1 in
def RetLR : Cpu0Pseudo<(outs), (ins), "", [(Cpu0Ret)]>;

def RET     : RetBase<GPROut>;
```

Add class Cpu0DAGToDAGISel (Cpu0ISelDAGToDAG.cpp) to CMakeLists.txt, and add the following fragment to Cpu0TargetMachine.cpp,

lbdex/chapters/Chapter3_3/CMakeLists.txt

```
add_llvm_target(
  ...
```

```
  Cpu0ISelDAGToDAG.cpp
  Cpu0SEISelDAGToDAG.cpp
```

```
  ...
)
```

The following code in Cpu0TargetMachine.cpp will create a pass in instruction selection stage.

lbdex/chapters/Chapter3_3/Cpu0TargetMachine.cpp

```
#include "Cpu0SEISelDAGToDAG.h"
```

```
...
class Cpu0PassConfig : public TargetPassConfig {
public:
  ...
```

```
  bool addInstSelector() override;
```

```
};
...
```

```
// Install an instruction selector pass using
// the ISelDag to gen Cpu0 code.
bool Cpu0PassConfig::addInstSelector() {
  addPass(createCpu0SEISelDag(getCpu0TargetMachine(), getOptLevel()));
  return false;
}
```

lbdex/chapters/Chapter3_3/Cpu0ISelDAGToDAG.h

```
//===---- Cpu0ISelDAGToDAG.h - A Dag to Dag Inst Selector for Cpu0 --------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file defines an instruction selector for the CPU0 target.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0ISELDAGTODAG_H
#define LLVM_LIB_TARGET_CPU0_CPU0ISELDAGTODAG_H

#include "Cpu0Config.h"

#include "Cpu0.h"
#include "Cpu0Subtarget.h"
#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/SelectionDAGISel.h"
#include "llvm/IR/Type.h"
#include "llvm/Support/Debug.h"

//===----------------------------------------------------------------------===//
// Instruction Selector Implementation
```

(continues on next page)

(continued from previous page)

```
//===----------------------------------------------------------------------===//

//===----------------------------------------------------------------------===//
// Cpu0DAGToDAGISel - CPU0 specific code to select CPU0 machine
// instructions for SelectionDAG operations.
//===----------------------------------------------------------------------===//
namespace llvm {

class Cpu0DAGToDAGISel : public SelectionDAGISel {
public:
  explicit Cpu0DAGToDAGISel(Cpu0TargetMachine &TM, CodeGenOpt::Level OL)
      : SelectionDAGISel(TM, OL), Subtarget(nullptr) {}

  // Pass Name
  StringRef getPassName() const override {
    return "CPU0 DAG->DAG Pattern Instruction Selection";
  }

  bool runOnMachineFunction(MachineFunction &MF) override;

protected:

  /// Keep a pointer to the Cpu0Subtarget around so that we can make the right
  /// decision when generating code for different targets.
  const Cpu0Subtarget *Subtarget;

private:
  // Include the pieces autogenerated from the target description.
  #include "Cpu0GenDAGISel.inc"

  /// getTargetMachine - Return a reference to the TargetMachine, casted
  /// to the target-specific type.
  const Cpu0TargetMachine &getTargetMachine() {
    return static_cast<const Cpu0TargetMachine &>(TM);
  }

  void Select(SDNode *N) override;

  virtual bool trySelect(SDNode *Node) = 0;

  // Complex Pattern.
  bool SelectAddr(SDNode *Parent, SDValue N, SDValue &Base, SDValue &Offset);

  // getImm - Return a target constant with the specified value.
  inline SDValue getImm(const SDNode *Node, unsigned Imm) {
    return CurDAG->getTargetConstant(Imm, SDLoc(Node), Node->getValueType(0));
  }

  virtual void processFunctionAfterISel(MachineFunction &MF) = 0;

};
```

(continues on next page)

(continued from previous page)

```
}

#endif
```

lbdex/chapters/Chapter3_3/Cpu0ISelDAGToDAG.cpp

```
//===-- Cpu0ISelDAGToDAG.cpp - A Dag to Dag Inst Selector for Cpu0 --------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file defines an instruction selector for the CPU0 target.
//
//===----------------------------------------------------------------------===//

#include "Cpu0ISelDAGToDAG.h"
#include "Cpu0.h"

#include "Cpu0MachineFunction.h"
#include "Cpu0RegisterInfo.h"
#include "Cpu0SEISelDAGToDAG.h"
#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/MachineConstantPool.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/SelectionDAGISel.h"
#include "llvm/CodeGen/SelectionDAGNodes.h"
#include "llvm/IR/CFG.h"
#include "llvm/IR/GlobalValue.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/Intrinsics.h"
#include "llvm/IR/Type.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Target/TargetMachine.h"
using namespace llvm;

#define DEBUG_TYPE "cpu0-isel"

//===----------------------------------------------------------------------===//
// Instruction Selector Implementation
//===----------------------------------------------------------------------===//
```

(continues on next page)

(continued from previous page)

```
//===----------------------------------------------------------------------===//
// Cpu0DAGToDAGISel - CPU0 specific code to select CPU0 machine
// instructions for SelectionDAG operations.
//===----------------------------------------------------------------------===//

bool Cpu0DAGToDAGISel::runOnMachineFunction(MachineFunction &MF) {
  bool Ret = SelectionDAGISel::runOnMachineFunction(MF);

  return Ret;
}

//@SelectAddr {
/// ComplexPattern used on Cpu0InstrInfo
/// Used on Cpu0 Load/Store instructions
bool Cpu0DAGToDAGISel::
SelectAddr(SDNode *Parent, SDValue Addr, SDValue &Base, SDValue &Offset) {
//@SelectAddr }
  EVT ValTy = Addr.getValueType();
  SDLoc DL(Addr);

  // If Parent is an unaligned f32 load or store, select a (base + index)
  // floating point load/store instruction (luxc1 or suxc1).
  const LSBaseSDNode* LS = 0;

  if (Parent && (LS = dyn_cast<LSBaseSDNode>(Parent))) {
    EVT VT = LS->getMemoryVT();

    if (VT.getSizeInBits() / 8 > LS->getAlignment()) {
      assert(0 && "Unaligned loads/stores not supported for this type.");
      if (VT == MVT::f32)
        return false;
    }
  }

  // if Address is FI, get the TargetFrameIndex.
  if (FrameIndexSDNode *FIN = dyn_cast<FrameIndexSDNode>(Addr)) {
    Base   = CurDAG->getTargetFrameIndex(FIN->getIndex(), ValTy);
    Offset = CurDAG->getTargetConstant(0, DL, ValTy);
    return true;
  }

  Base   = Addr;
  Offset = CurDAG->getTargetConstant(0, DL, ValTy);
  return true;
}

//@Select {
/// Select instructions not customized! Used for
/// expanded, promoted and normal instructions
void Cpu0DAGToDAGISel::Select(SDNode *Node) {
//@Select }
```

(continues on next page)

(continued from previous page)

```
  unsigned Opcode = Node->getOpcode();

  // If we have a custom node, we already have selected!
  if (Node->isMachineOpcode()) {
    LLVM_DEBUG(errs() << "== "; Node->dump(CurDAG); errs() << "\n");
    Node->setNodeId(-1);
    return;
  }

  // See if subclasses can handle this node.
  if (trySelect(Node))
    return;

  switch(Opcode) {
  default: break;

  }

  // Select the default instruction
  SelectCode(Node);
}
```

lbdex/chapters/Chapter3_3/Cpu0SEISelDAGToDAG.h

```
//===-- Cpu0SEISelDAGToDAG.h - A Dag to Dag Inst Selector for Cpu0SE -----===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// Subclass of Cpu0DAGToDAGISel specialized for cpu032.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0SEISELDAGTODAG_H
#define LLVM_LIB_TARGET_CPU0_CPU0SEISELDAGTODAG_H

#include "Cpu0Config.h"

#include "Cpu0ISelDAGToDAG.h"

namespace llvm {

class Cpu0SEDAGToDAGISel : public Cpu0DAGToDAGISel {

public:
```

(continues on next page)

(continued from previous page)

```
  explicit Cpu0SEDAGToDAGISel(Cpu0TargetMachine &TM, CodeGenOpt::Level OL)
      : Cpu0DAGToDAGISel(TM, OL) {}

private:

  bool runOnMachineFunction(MachineFunction &MF) override;

  bool trySelect(SDNode *Node) override;

  void processFunctionAfterISel(MachineFunction &MF) override;

  // Insert instructions to initialize the global base register in the
  // first MBB of the function.
//  void initGlobalBaseReg(MachineFunction &MF);

};

FunctionPass *createCpu0SEISelDag(Cpu0TargetMachine &TM,
                                  CodeGenOpt::Level OptLevel);

}

#endif
```

lbdex/chapters/Chapter3_3/Cpu0ISelDAGToDAG.cpp

```
//===-- Cpu0SEISelDAGToDAG.cpp - A Dag to Dag Inst Selector for Cpu0SE ----===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// Subclass of Cpu0DAGToDAGISel specialized for cpu032.
//
//===----------------------------------------------------------------------===//

#include "Cpu0SEISelDAGToDAG.h"

#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "Cpu0.h"
#include "Cpu0MachineFunction.h"
#include "Cpu0RegisterInfo.h"
#include "llvm/CodeGen/MachineConstantPool.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
```

(continues on next page)

(continued from previous page)

```
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/SelectionDAGNodes.h"
#include "llvm/IR/CFG.h"
#include "llvm/IR/GlobalValue.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/Intrinsics.h"
#include "llvm/IR/Type.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Target/TargetMachine.h"
using namespace llvm;

#define DEBUG_TYPE "cpu0-isel"

bool Cpu0SEDAGToDAGISel::runOnMachineFunction(MachineFunction &MF) {
  Subtarget = &static_cast<const Cpu0Subtarget &>(MF.getSubtarget());
  return Cpu0DAGToDAGISel::runOnMachineFunction(MF);
}

void Cpu0SEDAGToDAGISel::processFunctionAfterISel(MachineFunction &MF) {
}

//@selectNode
bool Cpu0SEDAGToDAGISel::trySelect(SDNode *Node) {
  unsigned Opcode = Node->getOpcode();
  SDLoc DL(Node);

  ///
  // Instruction Selection not handled by the auto-generated
  // tablegen selection should be handled here.
  ///

  ///
  // Instruction Selection not handled by the auto-generated
  // tablegen selection should be handled here.
  ///
  EVT NodeTy = Node->getValueType(0);
  unsigned MultOpc;

  switch(Opcode) {
  default: break;

  }

  return false;
}

FunctionPass *llvm::createCpu0SEISelDag(Cpu0TargetMachine &TM,
                                        CodeGenOpt::Level OptLevel) {
  return new Cpu0SEDAGToDAGISel(TM, OptLevel);
}
```

(continues on next page)

(continued from previous page)

```
```

Function Cpu0DAGToDAGISel::Select() of Cpu0ISelDAGToDAG.cpp is for the selection of "OP code DAG node" while Cpu0DAGToDAGISel::SelectAddr() is for the selection of "DATA DAG node with **addr** type" which defined in Chapter2/Cpu0InstrInfo.td. This method's name corresponding to Chapter2/Cpu0InstrInfo.td as follows,

lbdex/chapters/Chapter2/Cpu0InstrInfo.td

```
def addr : ComplexPattern<iPTR, 2, "SelectAddr", [frameindex], [SDNPWantParent]>;
```

The iPTR, ComplexPattern, frameindex and SDNPWantParent defined as follows,

llvm/include/llvm/Target/TargetSelection.td

```
def SDNPWantParent  : SDNodeProperty;   // ComplexPattern gets the parent
...
def frameindex  : SDNode<"ISD::FrameIndex",           SDTPtrLeaf, [],
                         "FrameIndexSDNode">;
...
// Complex patterns, e.g. X86 addressing mode, requires pattern matching code
// in C++. NumOperands is the number of operands returned by the select function;
// SelectFunc is the name of the function used to pattern match the max. pattern;
// RootNodes are the list of possible root nodes of the sub-dags to match.
// e.g. X86 addressing mode - def addr : ComplexPattern<4, "SelectAddr", [add]>;
//
class ComplexPattern<ValueType ty, int numops, string fn,
                     list<SDNode> roots = [], list<SDNodeProperty> props = []> {
  ValueType Ty = ty;
  int NumOperands = numops;
  string SelectFunc = fn;
  list<SDNode> RootNodes = roots;
  list<SDNodeProperty> Properties = props;
}
```

llvm/include/llvm/CodeGen/ValueTypes.td

```
// Pseudo valuetype mapped to the current pointer size.
def iPTR    : ValueType<0   , 255>;
```

Build Chapter3_3 and run with it, finding the error message of Chapter3_2 is gone. The new error message for Chapter3_3 as follows,

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
...
LLVM ERROR: Cannot select: t6: ch = Cpu0ISD::Ret t4, Register:i32 $lr
  t5: i32 = Register $lr
...
```

Above can display the error message DAG node "Cpu0ISD::Ret" because the following code added in Chapter3_1/Cpu0ISelLowering.cpp.

lbdex/chapters/Chapter3_1/Cpu0ISelLowering.cpp

```
const char *Cpu0TargetLowering::getTargetNodeName(unsigned Opcode) const {
  switch (Opcode) {
  case Cpu0ISD::JmpLink:           return "Cpu0ISD::JmpLink";
  case Cpu0ISD::TailCall:          return "Cpu0ISD::TailCall";
  case Cpu0ISD::Hi:                return "Cpu0ISD::Hi";
  case Cpu0ISD::Lo:                return "Cpu0ISD::Lo";
  case Cpu0ISD::GPRel:             return "Cpu0ISD::GPRel";
  case Cpu0ISD::Ret:               return "Cpu0ISD::Ret";
  case Cpu0ISD::EH_RETURN:         return "Cpu0ISD::EH_RETURN";
  case Cpu0ISD::DivRem:            return "Cpu0ISD::DivRem";
  case Cpu0ISD::DivRemU:           return "Cpu0ISD::DivRemU";
  case Cpu0ISD::Wrapper:           return "Cpu0ISD::Wrapper";
```

```
  default:                         return NULL;
  }
}
```

3.4 Handle return register $lr

The following code is the result of running Mips backend with ch3.cpp.

```
JonathantekiiMac:input Jonathan$ ~/llvm/debug/build/bin/llc
-march=mips -relocation-model=pic -filetype=asm ch3.bc -o -
  .text
  .abicalls
  .section  .mdebug.abi32,"",@progbits
  .nan  legacy
  .file "ch3.bc"
  .text
  .globl  main
  .align  2
  .type main,@function
  .set  nomicromips
  .set  nomips16
  .ent  main
main:                                   # @main
  .frame  $fp,8,$ra
  .mask   0x40000000,-4
  .fmask  0x00000000,0
  .set  noreorder
  .set  nomacro
  .set  noat
# BB#0:
  addiu $sp, $sp, -8
  sw  $fp, 4($sp)             # 4-byte Folded Spill
```

(continues on next page)

(continued from previous page)

```
  move   $fp, $sp
  sw  $zero, 0($fp)
  addiu $2, $zero, 0
  move   $sp, $fp
  lw  $fp, 4($sp)             # 4-byte Folded Reload
  jr  $ra
  addiu $sp, $sp, 8
  .set  at
  .set  macro
  .set  reorder
  .end  main
$func_end0:
  .size main, ($func_end0)-main
```

As you can see, Mips return to the caller by using "jr $ra" where $ra is a specific register which keeps the caller's next instruction address. And it saves the return value in register $2. If we only create DAGs directly, then will having the following two problems.

1. LLVM can allocate any register for return value, for instance $3, rather than keeps it in $2.

2. LLVM will allocate a register randomly to "jr" since jr needs one operand. For instance, it generate "jr $8" rather than "jr $ra".

If Backend uses the "jal sub-routine" and "jr", and put the return address in the specific register $ra, then no the second problem. But in Mips, it allows programmer uses "jal $rx, sub-routine" and "jr $rx" whereas $rx is not $ra. Allowing programmer uses other register but $ra providing more flexibility in programming of high level language such as C with assembly. File ch8_2_longbranch.cpp in the following is an example, it uses jr $1 without spill $ra register. This will save a lot of time if it is in a hot function.

lbdex/input/ch8_2_longbranch.cpp

```
int test_longbranch()
{
  volatile int a = 2;
  volatile int b = 1;
  int result = 0;

  if (a < b)
    result = 1;
  return result;
}
```

```
JonathantekiiMac:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch8_2_longbranch.cpp -emit-llvm -o ch8_2_longbranch.bc
JonathantekiiMac:input Jonathan$ ~/llvm/debug/build/bin/llc
-march=mips -relocation-model=pic -filetype=asm -force-mips-long-branch
ch8_2_longbranch.bc -o -
  ...
  .ent  _Z15test_longbranchv
_Z15test_longbranchv:                   # @_Z15test_longbranchv
  .frame  $fp,16,$ra
```

(continues on next page)

(continued from previous page)

```
  .mask   0x40000000,-4
  .fmask  0x00000000,0
  .set  noreorder
  .set  nomacro
  .set  noat
# BB#0:
  addiu $sp, $sp, -16
  sw  $fp, 12($sp)            # 4-byte Folded Spill
  move   $fp, $sp
  addiu $1, $zero, 2
  sw  $1, 8($fp)
  addiu $2, $zero, 1
  sw  $2, 4($fp)
  sw  $zero, 0($fp)
  lw  $1, 8($fp)
  lw  $3, 4($fp)
  slt $1, $1, $3
  bnez  $1, $BB0_3
  nop
# BB#1:
  addiu $sp, $sp, -8
  sw  $ra, 0($sp)
  lui $1, %hi(($BB0_4)-($BB0_2))
  bal $BB0_2
  addiu $1, $1, %lo(($BB0_4)-($BB0_2))
$BB0_2:
  addu  $1, $ra, $1
  lw  $ra, 0($sp)
  jr  $1
  addiu $sp, $sp, 8
$BB0_3:
  sw  $2, 0($fp)
$BB0_4:
  lw  $2, 0($fp)
  move   $sp, $fp
  lw  $fp, 12($sp)            # 4-byte Folded Reload
  jr  $ra
  addiu $sp, $sp, 16
  .set  at
  .set  macro
  .set  reorder
  .end  _Z15test_longbranchv
$func_end0:
  .size _Z15test_longbranchv, ($func_end0)-_Z15test_longbranchv
```

The following code handle the return register $lr.

lbdex/chapters/Chapter3_4/Cpu0CallingConv.td

```
def RetCC_Cpu0EABI : CallingConv<[
  // i32 are returned in registers V0, V1, A0, A1
  CCIfType<[i32], CCAssignToReg<[V0, V1, A0, A1]>>
]>;
```

```
def RetCC_Cpu0 : CallingConv<[
  CCDelegateTo<RetCC_Cpu0EABI>
]>;
```

lbdex/chapters/Chapter3_4/Cpu0InstrFormats.td

```
// Cpu0 Pseudo Instructions Format
class Cpu0Pseudo<dag outs, dag ins, string asmstr, list<dag> pattern>:
      Cpu0Inst<outs, ins, asmstr, pattern, IIPseudo, Pseudo> {
  let isCodeGenOnly = 1;
  let isPseudo = 1;
}
```

lbdex/chapters/Chapter3_4/Cpu0InstrInfo.td

```
let Predicates = [Ch3_4] in {
let isReturn=1, isTerminator=1, hasDelaySlot=1, isBarrier=1, hasCtrlDep=1 in
  def RetLR : Cpu0Pseudo<(outs), (ins), "", [(Cpu0Ret)]>;
}
```

lbdex/chapters/Chapter3_4/Cpu0ISelLowering.h

```
    /// Cpu0CC - This class provides methods used to analyze formal and call
    /// arguments and inquire about calling convention information.
    class Cpu0CC {
    public:
      enum SpecialCallingConvType {
        NoSpecialCallingConv
      };

      Cpu0CC(CallingConv::ID CallConv, bool IsO32, CCState &Info,
             SpecialCallingConvType SpecialCallingConv = NoSpecialCallingConv);

      void analyzeCallResult(const SmallVectorImpl<ISD::InputArg> &Ins,
                             bool IsSoftFloat, const SDNode *CallNode,
                             const Type *RetTy) const;

      void analyzeReturn(const SmallVectorImpl<ISD::OutputArg> &Outs,
                         bool IsSoftFloat, const Type *RetTy) const;

      const CCState &getCCInfo() const { return CCInfo; }
```

(continues on next page)

(continued from previous page)

```
    /// hasByValArg - Returns true if function has byval arguments.
    bool hasByValArg() const { return !ByValArgs.empty(); }

    /// reservedArgArea - The size of the area the caller reserves for
    /// register arguments. This is 16-byte if ABI is O32.
    unsigned reservedArgArea() const;

    typedef SmallVectorImpl<ByValArgInfo>::const_iterator byval_iterator;
    byval_iterator byval_begin() const { return ByValArgs.begin(); }
    byval_iterator byval_end() const { return ByValArgs.end(); }

  private:

    /// Return the type of the register which is used to pass an argument or
    /// return a value. This function returns f64 if the argument is an i64
    /// value which has been generated as a result of softening an f128 value.
    /// Otherwise, it just returns VT.
    MVT getRegVT(MVT VT, bool IsSoftFloat) const;

    template<typename Ty>
    void analyzeReturn(const SmallVectorImpl<Ty> &RetVals, bool IsSoftFloat,
                       const SDNode *CallNode, const Type *RetTy) const;

    CCState &CCInfo;
    CallingConv::ID CallConv;
    bool IsO32;
    SmallVector<ByValArgInfo, 2> ByValArgs;
  };
```

lbdex/chapters/Chapter3_4/Cpu0ISelLowering.cpp

```
SDValue
Cpu0TargetLowering::LowerReturn(SDValue Chain,
                                CallingConv::ID CallConv, bool IsVarArg,
                                const SmallVectorImpl<ISD::OutputArg> &Outs,
                                const SmallVectorImpl<SDValue> &OutVals,
                                const SDLoc &DL, SelectionDAG &DAG) const {
```

```
  // CCValAssign - represent the assignment of
  // the return value to a location
  SmallVector<CCValAssign, 16> RVLocs;
  MachineFunction &MF = DAG.getMachineFunction();

  // CCState - Info about the registers and stack slot.
  CCState CCInfo(CallConv, IsVarArg, MF, RVLocs,
                 *DAG.getContext());
  Cpu0CC Cpu0CCInfo(CallConv, ABI.IsO32(),
                    CCInfo);
```

(continues on next page)

(continued from previous page)

```
  // Analyze return values.
  Cpu0CCInfo.analyzeReturn(Outs, Subtarget.abiUsesSoftFloat(),
                           MF.getFunction().getReturnType());

  SDValue Flag;
  SmallVector<SDValue, 4> RetOps(1, Chain);

  // Copy the result values into the output registers.
  for (unsigned i = 0; i != RVLocs.size(); ++i) {
    SDValue Val = OutVals[i];
    CCValAssign &VA = RVLocs[i];
    assert(VA.isRegLoc() && "Can only return in registers!");

    if (RVLocs[i].getValVT() != RVLocs[i].getLocVT())
      Val = DAG.getNode(ISD::BITCAST, DL, RVLocs[i].getLocVT(), Val);

    Chain = DAG.getCopyToReg(Chain, DL, VA.getLocReg(), Val, Flag);

    // Guarantee that all emitted copies are stuck together with flags.
    Flag = Chain.getValue(1);
    RetOps.push_back(DAG.getRegister(VA.getLocReg(), VA.getLocVT()));
  }

//@Ordinary struct type: 2 {
  // The cpu0 ABIs for returning structs by value requires that we copy
  // the sret argument into $v0 for the return. We saved the argument into
  // a virtual register in the entry block, so now we copy the value out
  // and into $v0.
  if (MF.getFunction().hasStructRetAttr()) {
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
    unsigned Reg = Cpu0FI->getSRetReturnReg();

    if (!Reg)
      llvm_unreachable("sret virtual register not created in the entry block");
    SDValue Val =
        DAG.getCopyFromReg(Chain, DL, Reg, getPointerTy(DAG.getDataLayout()));
    unsigned V0 = Cpu0::V0;

    Chain = DAG.getCopyToReg(Chain, DL, V0, Val, Flag);
    Flag = Chain.getValue(1);
    RetOps.push_back(DAG.getRegister(V0, getPointerTy(DAG.getDataLayout())));
  }
//@Ordinary struct type: 2 }

  RetOps[0] = Chain;  // Update chain.

  // Add the flag if we have it.
  if (Flag.getNode())
    RetOps.push_back(Flag);

  // Return on Cpu0 is always a "ret $lr"
  return DAG.getNode(Cpu0ISD::Ret, DL, MVT::Other, RetOps);
```

```
}
```

```
template<typename Ty>
void Cpu0TargetLowering::Cpu0CC::
analyzeReturn(const SmallVectorImpl<Ty> &RetVals, bool IsSoftFloat,
              const SDNode *CallNode, const Type *RetTy) const {
  CCAssignFn *Fn;

  Fn = RetCC_Cpu0;

  for (unsigned I = 0, E = RetVals.size(); I < E; ++I) {
    MVT VT = RetVals[I].VT;
    ISD::ArgFlagsTy Flags = RetVals[I].Flags;
    MVT RegVT = this->getRegVT(VT, IsSoftFloat);

    if (Fn(I, VT, RegVT, CCValAssign::Full, Flags, this->CCInfo)) {
#ifndef NDEBUG
      dbgs() << "Call result #" << I << " has unhandled type "
             << EVT(VT).getEVTString() << '\n';
#endif
      llvm_unreachable(nullptr);
    }
  }
}

void Cpu0TargetLowering::Cpu0CC::
analyzeCallResult(const SmallVectorImpl<ISD::InputArg> &Ins, bool IsSoftFloat,
                  const SDNode *CallNode, const Type *RetTy) const {
  analyzeReturn(Ins, IsSoftFloat, CallNode, RetTy);
}

void Cpu0TargetLowering::Cpu0CC::
analyzeReturn(const SmallVectorImpl<ISD::OutputArg> &Outs, bool IsSoftFloat,
              const Type *RetTy) const {
  analyzeReturn(Outs, IsSoftFloat, nullptr, RetTy);
}
```

```
unsigned Cpu0TargetLowering::Cpu0CC::reservedArgArea() const {
  return (IsO32 && (CallConv != CallingConv::Fast)) ? 8 : 0;
}
```

```
MVT Cpu0TargetLowering::Cpu0CC::getRegVT(MVT VT,
                                         bool IsSoftFloat) const {
  if (IsSoftFloat || IsO32)
    return VT;

  return VT;
}
```

lbdex/chapters/Chapter3_4/Cpu0MachineFunction.h

```
/// Cpu0FunctionInfo - This class is derived from MachineFunction private
/// Cpu0 target-specific information for each MachineFunction.
class Cpu0FunctionInfo : public MachineFunctionInfo {
```

```
    SRetReturnReg(0), CallsEhReturn(false), CallsEhDwarf(false),
```

```
  unsigned getSRetReturnReg() const { return SRetReturnReg; }
  void setSRetReturnReg(unsigned Reg) { SRetReturnReg = Reg; }
```

```
  bool hasByvalArg() const { return HasByvalArg; }
  void setFormalArgInfo(unsigned Size, bool HasByval) {
    IncomingArgSize = Size;
    HasByvalArg = HasByval;
  }
```

```
  /// SRetReturnReg - Some subtargets require that sret lowering includes
  /// returning the value of the returned struct in a register. This field
  /// holds the virtual register into which the sret argument is passed.
  unsigned SRetReturnReg;
```

```
  /// True if function has a byval argument.
  bool HasByvalArg;

  /// Size of incoming argument area.
  unsigned IncomingArgSize;

  /// CallsEhReturn - Whether the function calls llvm.eh.return.
  bool CallsEhReturn;

  /// CallsEhDwarf - Whether the function calls llvm.eh.dwarf.
  bool CallsEhDwarf;

  /// Frame objects for spilling eh data registers.
  int EhDataRegFI[2];
```

```
}
```

lbdex/chapters/Chapter3_4/Cpu0SEInstrInfo.h

```
//@expandPostRAPseudo
  bool expandPostRAPseudo(MachineInstr &MI) const override;
```

```
private:
  void expandRetLR(MachineBasicBlock &MBB, MachineBasicBlock::iterator I) const;
```

lbdex/chapters/Chapter3_4/Cpu0SEInstrInfo.cpp

```
//@expandPostRAPseudo
/// Expand Pseudo instructions into real backend instructions
bool Cpu0SEInstrInfo::expandPostRAPseudo(MachineInstr &MI) const {
//@expandPostRAPseudo-body
  MachineBasicBlock &MBB = *MI.getParent();

  switch (MI.getDesc().getOpcode()) {
  default:
    return false;
  case Cpu0::RetLR:
    expandRetLR(MBB, MI);
    break;
```

```
  }

  MBB.erase(MI);
  return true;
}
```

```
void Cpu0SEInstrInfo::expandRetLR(MachineBasicBlock &MBB,
                                  MachineBasicBlock::iterator I) const {
  BuildMI(MBB, I, I->getDebugLoc(), get(Cpu0::RET)).addReg(Cpu0::LR);
}
```

Build Chapter3_4 and run with it, finding the error message in Chapter3_3 is gone. The compile result will hang on and please press "ctrl+C" to abort as follows,

```
118-165-78-230:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch3.cpp -emit-llvm -o ch3.bc
118-165-78-230:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch3.bc -o -
...
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1
  ret i32 0
}

118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o -
  ...
  .text
  .section .mdebug.abi32
  .previous
  .file "ch3.bc"
^C
```

It hangs on because Cpu0 backend has not handled stack slot for local variables. Instruction "store i32 0, i32* %1" in above IR need Cpu0 allocate a stack slot and save to the stack slot. However, the ch3.cpp can be run with option `clang -O2` as follows,

```
118-165-78-230:input Jonathan$ clang -O2 -target mips-unknown-linux-gnu -c
ch3.cpp -emit-llvm -o ch3.bc
118-165-78-230:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch3.bc -o -
...
define i32 @main() #0 {
  ret i32 0
}

118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o -
  .text
  .section .mdebug.abi32
  .previous
  .file "ch3.bc"
  .globl  main
  .align  2
  .type main,@function
  .ent  main                    # @main
main:
  .frame  $sp,0,$lr
  .mask   0x00000000,0
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $2, $zero, 0
  ret $lr
  .set  macro
  .set  reorder
  .end  main
$func_end0:
  .size main, ($func_end0)-main
```

To see how the **'DAG->DAG Pattern Instruction Selection'** work in llc, let's compile with `llc -print-before-all -print-after-all` option and get the following result. The DAGs which before and after instruction selection stage are shown as follows,

```
118-165-78-230:input Jonathan$ clang -O2 -target mips-unknown-linux-gnu -c
ch3.cpp -emit-llvm -o ch3.bc
118-165-78-12:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
-print-before-all -print-after-all ch3.bc -o -
...
*** IR Dump After Module Verifier ***
; Function Attrs: nounwind readnone
define i32 @main() #0 {
  ret i32 0
}
...
Initial selection DAG: BB#0 'main:'
SelectionDAG has 5 nodes:
    t0: ch = EntryToken
  t3: ch,glue = CopyToReg t0, Register:i32 %V0, Constant:i32<0>
```

(continues on next page)

(continued from previous page)

```
  t4: ch = Cpu0ISD::Ret t3, Register:i32 %V0, t3:1
...
===== Instruction selection begins: BB#0 ''
Selecting: t4: ch = Cpu0ISD::Ret t3, Register:i32 %V0, t3:1

ISEL: Starting pattern match on root node: t4: ch = Cpu0ISD::Ret t3, Register:i32 %V0,␣
↪t3:1

  Morphed node: t4: ch = RetLR Register:i32 %V0, t3, t3:1

ISEL: Match complete!
Selecting: t3: ch,glue = CopyToReg t0, Register:i32 %V0, Constant:i32<0>

Selecting: t2: i32 = Register %V0

Selecting: t1: i32 = Constant<0>

ISEL: Starting pattern match on root node: t1: i32 = Constant<0>

  Initial Opcode index to 3158
  Morphed node: t1: i32 = ADDiu Register:i32 %ZERO, TargetConstant:i32<0>

ISEL: Match complete!
Selecting: t0: ch = EntryToken

===== Instruction selection ends:
Selected selection DAG: BB#0 'main:'
SelectionDAG has 7 nodes:
    t0: ch = EntryToken
    t1: i32 = ADDiu Register:i32 %ZERO, TargetConstant:i32<0>
  t3: ch,glue = CopyToReg t0, Register:i32 %V0, t1
  t4: ch = RetLR Register:i32 %V0, t3, t3:1
...
********** REWRITE VIRTUAL REGISTERS **********
********** Function: main
********** REGISTER MAP **********
[%vreg0 -> %V0] GPROut

0B  BB#0: derived from LLVM BB %0
16B   %vreg0<def> = ADDiu %ZERO, 0; GPROut:%vreg0
32B   %V0<def> = COPY %vreg0<kill>; GPROut:%vreg0
48B   RetLR %V0<imp-use>
> %V0<def> = ADDiu %ZERO, 0
> %V0<def> = COPY %V0<kill>
Identity copy: %V0<def> = COPY %V0<kill>
  deleted.
> RetLR %V0<imp-use>
# *** IR Dump After Virtual Register Rewriter ***:
# Machine code for function main: Properties: <Post SSA, tracking liveness,␣
↪AllVRegsAllocated>

0B  BB#0: derived from LLVM BB %0
```

(continues on next page)

(continued from previous page)

```
16B   %V0<def> = ADDiu %ZERO, 0
48B   RetLR %V0<imp-use>
...
********** EXPANDING POST-RA PSEUDO INSTRS **********
********** Function: main
# *** IR Dump After Post-RA pseudo instruction expansion pass ***:
# Machine code for function main: Properties: <Post SSA, tracking liveness,␣
↪AllVRegsAllocated>

BB#0: derived from LLVM BB %0
  %V0<def> = ADDiu %ZERO, 0
  RET %LR
...

  .globl  main
  .p2align  2
  .type main,@function
  .ent  main                    # @main
main:
  .frame  $sp,0,$lr
  .mask   0x00000000,0
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $2, $zero, 0
  ret $lr
  .set  macro
  .set  reorder
  .end  main
$func_end0:
  .size main, ($func_end0)-main


  .ident  "Apple LLVM version 7.0.0 (clang-700.1.76)"
  .section  ".note.GNU-stack","",@progbits
```

Summary above translation into Table: Chapter 3 .bc IR instructions.

Table 3.1: Chapter 3 .bc IR instructions

.bc	Lower	ISel	RVR	Post-RA	AsmP
constant 0	constant 0	ADDiu	ADDiu	ADDiu	addiu
ret	Cpu0ISD::Ret	CopyToReg,RetLR	RetLR	RET	ret

- Lower: Initial selection DAG (Cpu0ISelLowering.cpp, LowerReturn(...))
- ISel: Instruction selection
- RVR: REWRITE VIRTUAL REGISTERS, remove CopyToReg
- AsmP: Cpu0 Asm Printer
- Post-RA: Post-RA pseudo instruction expansion pass

From above `llc -print-before-all -print-after-all` display, we see **ret** is translated into **Cpu0ISD::Ret** in stage Optimized legalized selection DAG, and then translated into Cpu0 instructions **ret** finally. Since ret use **constant**

0 (**ret i32 0** in this example), the constant 0 will be translated into **"addiu $2, $zero, 0"** via the following pattern defined in Cpu0InstrInfo.td.

lbdex/chapters/Chapter2/Cpu0InstrInfo.td

```
// Small immediates
def : Pat<(i32 immSExt16:$in),
          (ADDiu ZERO, imm:$in)>;
```

In order to handle IR ret, these codes in Cpu0InstrInfo.td do things as below.

1. Declare a pseudo node Cpu0::RetLR to takes care the IR Cpu0ISD::Ret by the following code,

lbdex/chapters/Chapter3_4/Cpu0InstrInfo.td

```
// Return
def Cpu0Ret : SDNode<"Cpu0ISD::Ret", SDTNone,
                     [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;
```

```
let Predicates = [Ch3_4] in {
let isReturn=1, isTerminator=1, hasDelaySlot=1, isBarrier=1, hasCtrlDep=1 in
  def RetLR : Cpu0Pseudo<(outs), (ins), "", [(Cpu0Ret)]>;
}
```

2. Create Cpu0ISD::Ret node in LowerReturn() of Cpu0ISelLowering.cpp, which is called when meets keyword of return in C. Remind, In LowerReturn() put return value in register $2 ($v0).
3. After instruction selection, the Cpu0ISD::Ret is replaced by Cpu0::RetLR as below. This effect come from "def RetLR" as step 1.

```
===== Instruction selection begins: BB#0 'entry'
Selecting: 0x1ea4050: ch = Cpu0ISD::Ret 0x1ea3f50, 0x1ea3e50,
0x1ea3f50:1 [ID=27]

ISEL: Starting pattern match on root node: 0x1ea4050: ch = Cpu0ISD::Ret
0x1ea3f50, 0x1ea3e50, 0x1ea3f50:1 [ID=27]

  Morphed node: 0x1ea4050: ch = RetLR 0x1ea3e50, 0x1ea3f50, 0x1ea3f50:1
...
ISEL: Match complete!
=> 0x1ea4050: ch = RetLR 0x1ea3e50, 0x1ea3f50, 0x1ea3f50:1
...
===== Instruction selection ends:
Selected selection DAG: BB#0 'main:entry'
SelectionDAG has 28 nodes:
...
    0x1ea3e50: <multiple use>
    0x1ea3f50: <multiple use>
    0x1ea3f50: <multiple use>
  0x1ea4050: ch = RetLR 0x1ea3e50, 0x1ea3f50, 0x1ea3f50:1
```

4. Expand the Cpu0ISD::RetLR into instruction Cpu0::RET $lr in "Post-RA pseudo instruction expansion pass" stage by the code in Chapter3_4/Cpu0SEInstrInfo.cpp as above. This stage come after the register allocation, so we can replace the V0 ($r2) by LR ($lr) without any side effect.
5. Print assembly or obj according the information of *.inc generated by TableGen from *.td as the following at "Cpu0 Assembly Print" stage.

lbdex/chapters/Chapter2/Cpu0InstrInfo.td

```
//@JumpFR {
let isBranch=1, isTerminator=1, isBarrier=1, imm16=0, hasDelaySlot = 1,
    isIndirectBranch = 1 in
class JumpFR<bits<8> op, string instr_asm, RegisterClass RC>:
  FL<op, (outs), (ins RC:$ra),
     !strconcat(instr_asm, "\t$ra"), [(brind RC:$ra)], IIBranch> {
  let rb = 0;
  let imm16 = 0;
```

```
}
```

```
def RET     : RetBase<GPROut>;
```

Table 3.2: Handle return register lr

Stage	Function
Write Code	Declare a pseudo node Cpu0::RetLR
•	for IR Cpu0::Ret;
Before CPU0 DAG->DAG Pattern Instruction Selection	Create Cpu0ISD::Ret DAG
Instruction selection	Cpu0::Ret is replaced by Cpu0::RetLR
Post-RA pseudo instruction expansion pass	Cpu0::RetLR -> Cpu0::RET $lr
Cpu0 Assembly Printer	Print according "def RET"

Function LowerReturn() of Cpu0ISelLowering.cpp handle return variable correctly. Chapter3_4/Cpu0ISelLowering.cpp create Cpu0ISD::Ret node in LowerReturn() which is called by llvm system when it meets C's keyword of return. More specificly, it creates DAGs (Cpu0ISD::Ret (CopyToReg %X, %V0, %Y), %V0, Flag). Since the the V0 register is assigned in CopyToReg and Cpu0ISD::Ret use V0, the CopyToReg with V0 register will live out and won't be removed at any later optimization steps. Remember, if use "return DAG.getNode(Cpu0ISD::Ret, DL, MVT::Other, Chain, DAG.getRegister(Cpu0::LR, MVT::i32));" instead of "return DAG.getNode (Cpu0ISD::Ret, DL, MVT::Other, &RetOps[0], RetOps.size());" then the V0 register won't be live out, and the previous DAG (CopyToReg %X, %V0, %Y) will be removed at later optimization steps. It ending with the return value is error.

3.5 Add Prologue/Epilogue functions

3.5.1 Concept

Following come from tricore_llvm.pdf section "4.4.2 Non-static Register Information ".

For some target architectures, some aspects of the target architecture's register set are dependent upon variable factors and have to be determined at runtime. As a consequence, they cannot be generated statically from a TableGen description – although that would be possible for the bulk of them in the case of the TriCore backend. Among them are the following points:

- Callee-saved registers. Normally, the ABI specifies a set of registers that a function must save on entry and restore on return if their contents are possibly modified during execution.
- Reserved registers. Although the set of unavailable registers is already defined in the TableGen file, TriCoreRegisterInfo contains a method that marks all non-allocatable register numbers in a bit vector.

The following methods are implemented:

- emitPrologue() inserts prologue code at the beginning of a function. Thanks to TriCore's context model, this is a trivial task as it is not required to save any registers manually. The only thing that has to be done is reserving space for the function's stack frame by decrementing the stack pointer. In addition, if the function needs a frame pointer, the frame register %a14 is set to the old value of the stack pointer beforehand.
- emitEpilogue() is intended to emit instructions to destroy the stack frame and restore all previously saved registers before returning from a function. However, as %a10 (stack pointer), %a11 (return address), and %a14 (frame pointer, if any) are all part of the upper context, no epilogue code is needed at all. All cleanup operations are performed implicitly by the ret instruction.
- eliminateFrameIndex() is called for each instruction that references a word of data in a stack slot. All previous passes of the code generator have been addressing stack slots through an abstract frame index and an immediate offset. The purpose of this function is to translate such a reference into a register–offset pair. Depending on whether the machine function that contains the instruction has a fixed or a variable stack frame, either the stack pointer %a10 or the frame pointer %a14 is used as the base register. The offset is computed accordingly. Fig. 3.4 demonstrates for both cases how a stack slot is addressed.

If the addressing mode of the affected instruction cannot handle the address because the offset is too large (the offset field has 10 bits for the BO addressing mode and 16 bits for the BOL mode), a sequence of instructions is emitted that explicitly computes the effective address. Interim results are put into an unused address register. If none is available, an already occupied address register is scavenged. For this purpose, LLVM's framework offers a class named RegScavenger that takes care of all the details.

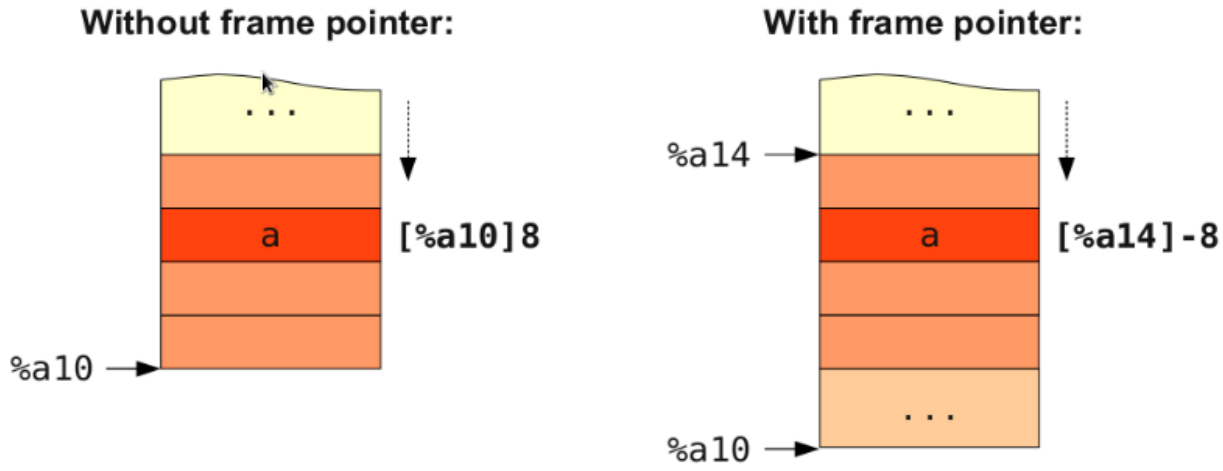


Fig. 3.4: Addressing of a variable a located on the stack. If the stack frame has a variable size, slot must be addressed relative to the frame pointer

3.5.2 Prologue and Epilogue functions

The Prologue and Epilogue functions as follows,

lbdex/chapters/Chapter3_5/Cpu0SEFrameLowering.cpp

```
void Cpu0SEFrameLowering::emitPrologue(MachineFunction &MF,
                                       MachineBasicBlock &MBB) const {
  MachineFrameInfo &MFI    = MF.getFrameInfo();
  Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

  const Cpu0SEInstrInfo &TII =
    *static_cast<const Cpu0SEInstrInfo*>(STI.getInstrInfo());
  const Cpu0RegisterInfo &RegInfo =
      *static_cast<const Cpu0RegisterInfo *>(STI.getRegisterInfo());

  MachineBasicBlock::iterator MBBI = MBB.begin();
  DebugLoc dl = MBBI != MBB.end() ? MBBI->getDebugLoc() : DebugLoc();
  Cpu0ABIInfo ABI = STI.getABI();
  unsigned SP = Cpu0::SP;
  const TargetRegisterClass *RC = &Cpu0::GPROutRegClass;

  // First, compute final stack size.
  uint64_t StackSize = MFI.getStackSize();

  // No need to allocate space on the stack.
  if (StackSize == 0 && !MFI.adjustsStack()) return;

  MachineModuleInfo &MMI = MF.getMMI();
  const MCRegisterInfo *MRI = MMI.getContext().getRegisterInfo();

  // Adjust stack.
  TII.adjustStackPtr(SP, -StackSize, MBB, MBBI);

  // emit ".cfi_def_cfa_offset StackSize"
  unsigned CFIIndex =
      MF.addFrameInst(
      MCCFIInstruction::cfiDefCfaOffset(nullptr, StackSize));
  BuildMI(MBB, MBBI, dl, TII.get(TargetOpcode::CFI_INSTRUCTION))
```

(continued from previous page)

```
          E = CSI.end(); I != E; ++I) {
      int64_t Offset = MFI.getObjectOffset(I->getFrameIdx());
      unsigned Reg = I->getReg();
      {
        // Reg is in CPURegs.
        unsigned CFIIndex = MF.addFrameInst(MCCFIInstruction::createOffset(
            nullptr, MRI->getDwarfRegNum(Reg, true), Offset));
        BuildMI(MBB, MBBI, dl, TII.get(TargetOpcode::CFI_INSTRUCTION))
            .addCFIIndex(CFIIndex);
      }
    }
  }

}
```

```
void Cpu0SEFrameLowering::emitEpilogue(MachineFunction &MF,
                                       MachineBasicBlock &MBB) const {
  MachineBasicBlock::iterator MBBI = MBB.getFirstTerminator();
  MachineFrameInfo &MFI            = MF.getFrameInfo();
  Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

  const Cpu0SEInstrInfo &TII =
      *static_cast<const Cpu0SEInstrInfo *>(STI.getInstrInfo());
  const Cpu0RegisterInfo &RegInfo =
      *static_cast<const Cpu0RegisterInfo *>(STI.getRegisterInfo());

  DebugLoc DL = MBBI != MBB.end() ? MBBI->getDebugLoc() : DebugLoc();
  Cpu0ABIInfo ABI = STI.getABI();
  unsigned SP = Cpu0::SP;

  // Get the number of bytes from FrameInfo
  uint64_t StackSize = MFI.getStackSize();

  if (!StackSize)
    return;

  // Adjust stack.
  TII.adjustStackPtr(SP, StackSize, MBB, MBBI);
}
```

```
bool
Cpu0SEFrameLowering::hasReservedCallFrame(const MachineFunction &MF) const {
  const MachineFrameInfo &MFI = MF.getFrameInfo();

  // Reserve call frame if the size of the maximum call frame fits into 16-bit
  // immediate field and there are no variable sized objects on the stack.
  // Make sure the second register scavenger spill slot can be accessed with one
  // instruction.
  return isInt<16>(MFI.getMaxCallFrameSize() + getStackAlignment()) &&
    !MFI.hasVarSizedObjects();
}
```

lbdex/chapters/Chapter3_5/Cpu0MachineFunction.h

```
  unsigned getIncomingArgSize() const { return IncomingArgSize; }

  bool callsEhReturn() const { return CallsEhReturn; }
  void setCallsEhReturn() { CallsEhReturn = true; }

  bool callsEhDwarf() const { return CallsEhDwarf; }
  void setCallsEhDwarf() { CallsEhDwarf = true; }

  void createEhDataRegsFI();
  int getEhDataRegFI(unsigned Reg) const { return EhDataRegFI[Reg]; }

  unsigned getMaxCallFrameSize() const { return MaxCallFrameSize; }
  void setMaxCallFrameSize(unsigned S) { MaxCallFrameSize = S; }
```

Now we explain the Prologue and Epilogue further by example code. For the following llvm IR code of ch3.cpp, Chapter3_5 of Cpu0 backend will emit the corresponding machine instructions as follows,

```
118-165-78-230:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch3.cpp -emit-llvm -o ch3.bc
118-165-78-230:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch3.bc -o -
...
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1
  ret i32 0
}

118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o -
  ...
  .section .mdebug.abi32
  .previous
  .file "ch3.bc"
  .text
  .globl  main//static void expandLargeImm\\n
  .align  2
  .type main,@function
  .ent  main                    # @main
main:
  .cfi_startproc
  .frame  $sp,8,$lr
  .mask   0x00000000,0
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -8
$tmp1:
  .cfi_def_cfa_offset 8
  addiu $2, $zero, 0
  st  $2, 4($sp)
```

(continues on next page)

(continued from previous page)

```
  addiu $sp, $sp, 8
  ret $lr
  .set  macro
  .set  reorder
  .end  main
$tmp2:
  .size main, ($tmp2)-main
  .cfi_endproc
```

LLVM get the stack size by counting how many virtual registers is assigned to local variables. After that, it calls emitPrologue().

```
  virtual void adjustStackPtr(unsigned SP, int64_t Amount,
                              MachineBasicBlock &MBB,
                              MachineBasicBlock::iterator I) const = 0;
```

lbdex/chapters/Chapter3_5/Cpu0SEInstrInfo.h

```
  /// Adjust SP by Amount bytes.
  void adjustStackPtr(unsigned SP, int64_t Amount, MachineBasicBlock &MBB,
                      MachineBasicBlock::iterator I) const override;

  /// Emit a series of instructions to load an immediate. If NewImm is a
  /// non-NULL parameter, the last instruction is not emitted, but instead
  /// its immediate operand is returned in NewImm.
  unsigned loadImmediate(int64_t Imm, MachineBasicBlock &MBB,
                         MachineBasicBlock::iterator II, const DebugLoc &DL,
                         unsigned *NewImm) const;
```

lbdex/chapters/Chapter3_5/Cpu0SEInstrInfo.cpp

```
/// Adjust SP by Amount bytes.
void Cpu0SEInstrInfo::adjustStackPtr(unsigned SP, int64_t Amount,
                                     MachineBasicBlock &MBB,
                                     MachineBasicBlock::iterator I) const {
  DebugLoc DL = I != MBB.end() ? I->getDebugLoc() : DebugLoc();
  unsigned ADDu = Cpu0::ADDu;
  unsigned ADDiu = Cpu0::ADDiu;

  if (isInt<16>(Amount)) {
    // addiu sp, sp, amount
    BuildMI(MBB, I, DL, get(ADDiu), SP).addReg(SP).addImm(Amount);
  }
  else { // Expand immediate that doesn't fit in 16-bit.
    unsigned Reg = loadImmediate(Amount, MBB, I, DL, nullptr);
    BuildMI(MBB, I, DL, get(ADDu), SP).addReg(SP).addReg(Reg, RegState::Kill);
  }
}
```

(continues on next page)

(continued from previous page)

```
/// This function generates the sequence of instructions needed to get the
/// result of adding register REG and immediate IMM.
unsigned
Cpu0SEInstrInfo::loadImmediate(int64_t Imm, MachineBasicBlock &MBB,
                               MachineBasicBlock::iterator II,
                               const DebugLoc &DL,
                               unsigned *NewImm) const {
  Cpu0AnalyzeImmediate AnalyzeImm;
  unsigned Size = 32;
  unsigned LUi = Cpu0::LUi;
  unsigned ZEROReg = Cpu0::ZERO;
  unsigned ATReg = Cpu0::AT;
  bool LastInstrIsADDiu = NewImm;

  const Cpu0AnalyzeImmediate::InstSeq &Seq =
    AnalyzeImm.Analyze(Imm, Size, LastInstrIsADDiu);
  Cpu0AnalyzeImmediate::InstSeq::const_iterator Inst = Seq.begin();

  assert(Seq.size() && (!LastInstrIsADDiu || (Seq.size() > 1)));

  // The first instruction can be a LUi, which is different from other
  // instructions (ADDiu, ORI and SLL) in that it does not have a register
  // operand.
  if (Inst->Opc == LUi)
    BuildMI(MBB, II, DL, get(LUi), ATReg).addImm(SignExtend64<16>(Inst->ImmOpnd));
  else
    BuildMI(MBB, II, DL, get(Inst->Opc), ATReg).addReg(ZEROReg)
      .addImm(SignExtend64<16>(Inst->ImmOpnd));

  // Build the remaining instructions in Seq.
  for (++Inst; Inst != Seq.end() - LastInstrIsADDiu; ++Inst)
    BuildMI(MBB, II, DL, get(Inst->Opc), ATReg).addReg(ATReg)
      .addImm(SignExtend64<16>(Inst->ImmOpnd));

  if (LastInstrIsADDiu)
    *NewImm = Inst->ImmOpnd;

  return ATReg;
}
```

In emitPrologue(), it emits machine instructions to adjust sp (stack pointer register) for local variables. For our example, it will emit the instruction,

```
addiu $sp, $sp, -8
```

The emitEpilogue() will emit "addiu $sp, $sp, 8", where 8 is the stack size.

In above ch3.cpp assembly output, it generates "addiu $2, $zero, 0" rather than "ori $2, $zero, 0" because ADDiu defined before ORi as following, so it takes the priority. Of course, if the ORi is defined first, the it will translate into "ori" instruction.

lbdex/chapters/Chapter2/Cpu0InstrInfo.td

```
// Small immediates
def : Pat<(i32 immSExt16:$in),
          (ADDiu ZERO, imm:$in)>;
```

lbdex/chapters/Chapter3_5/Cpu0InstrInfo.td

```
let Predicates = [Ch3_5] in {
def : Pat<(i32 immZExt16:$in),
          (ORi ZERO, imm:$in)>;
def : Pat<(i32 immLow16Zero:$in),
          (LUi (HI16 imm:$in))>;

// Arbitrary immediates
def : Pat<(i32 imm:$imm),
          (ORi (LUi (HI16 imm:$imm)), (LO16 imm:$imm))>;
} // let Predicates = [Ch3_4]
```

3.5.3 Handle stack slot for local variables

The following code handle the stack slot for local variables.

lbdex/chapters/Chapter3_5/Cpu0RegisterInfo.cpp

```
//- If no eliminateFrameIndex(), it will hang on run.
// pure virtual method
// FrameIndex represent objects inside a abstract stack.
// We must replace FrameIndex with an stack/frame pointer
// direct reference.
void Cpu0RegisterInfo::
eliminateFrameIndex(MachineBasicBlock::iterator II, int SPAdj,
                    unsigned FIOperandNum, RegScavenger *RS) const {
  MachineInstr &MI = *II;
  MachineFunction &MF = *MI.getParent()->getParent();
  MachineFrameInfo &MFI = MF.getFrameInfo();
  Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

  unsigned i = 0;
  while (!MI.getOperand(i).isFI()) {
    ++i;
    assert(i < MI.getNumOperands() &&
           "Instr doesn't have FrameIndex operand!");
  }

  LLVM_DEBUG(errs() << "\nFunction : " << MF.getFunction().getName() << "\n";
             errs() << "<--------->\n" << MI);

  int FrameIndex = MI.getOperand(i).getIndex();
```

(continues on next page)

(continued from previous page)

```
uint64_t stackSize = MF.getFrameInfo().getStackSize();
int64_t spOffset = MF.getFrameInfo().getObjectOffset(FrameIndex);

LLVM_DEBUG(errs() << "FrameIndex : " << FrameIndex << "\n"
                  << "spOffset   : " << spOffset << "\n"
                  << "stackSize  : " << stackSize << "\n");

const std::vector<CalleeSavedInfo> &CSI = MFI.getCalleeSavedInfo();
int MinCSFI = 0;
int MaxCSFI = -1;

if (CSI.size()) {
  MinCSFI = CSI[0].getFrameIdx();
  MaxCSFI = CSI[CSI.size() - 1].getFrameIdx();
}

// The following stack frame objects are always referenced relative to $sp:
//  1. Outgoing arguments.
//  2. Pointer to dynamically allocated stack space.
//  3. Locations for callee-saved registers.
// Everything else is referenced relative to whatever register
// getFrameRegister() returns.
unsigned FrameReg;

FrameReg = Cpu0::SP;

// Calculate final offset.
// - There is no need to change the offset if the frame object is one of the
//   following: an outgoing argument, pointer to a dynamically allocated
//   stack space or a $gp restore location,
// - If the frame object is any of the following, its offset must be adjusted
//   by adding the size of the stack:
//   incoming argument, callee-saved register location or local variable.
int64_t Offset;
  Offset = spOffset + (int64_t)stackSize;

Offset    += MI.getOperand(i+1).getImm();

LLVM_DEBUG(errs() << "Offset     : " << Offset << "\n" << "<---------->\n");

// If MI is not a debug value, make sure Offset fits in the 16-bit immediate
// field.
if (!MI.isDebugValue() && !isInt<16>(Offset)) {
  errs() << "!!!ERROR!!! Not support large frame over 16-bit at this point.\n"
         << "Though CH3_5 support it."
         << "Reference: "
             "http://jonathan2251.github.io/lbd/backendstructure.html#large-stack\n"
         << "However the CH9_3, dynamic-stack-allocation-support bring instruction "
            "move $fp, $sp that make it complicated in coding against the tutoral "
            "purpose of Cpu0.\n"
         << "Reference: "
             "http://jonathan2251.github.io/lbd/funccall.html#dynamic-stack-allocation-
↪support\n";
```

(continues on next page)

(continued from previous page)

```
    assert(0 && "(!MI.isDebugValue() && !isInt<16>(Offset))");
  }

  MI.getOperand(i).ChangeToRegister(FrameReg, false);
  MI.getOperand(i+1).ChangeToImmediate(Offset);
}
```

The eliminateFrameIndex() of Cpu0RegisterInfo.cpp is called after stages "instruction selection" and "registers allocated". It translates frame index to correct offset of stack pointer by "spOffset = MF.getFrameInfo()->getObjectOffset(FrameIndex);". For instance of ch3.cpp, displaying the offset calculating as follows,

```
Spilling live registers at end of block.
BB#0: derived from LLVM BB %0
      %V0<def> = ADDiu %ZERO, 0
      ST %V0, <fi#0>, 0; mem:ST4[%1]
      RetLR %V0<imp-use,kill>
alloc FI(0) at SP[-4]

Function : main
<--------->
ST %V0, <fi#0>, 0; mem:ST4[%1]
FrameIndex : 0
spOffset   : -4
stackSize  : 8
Offset     : 4
<--------->
  ...
  .file "ch3.bc"
  ...
  .frame  $sp,8,$lr
  ...
# BB#0:
  addiu $sp, $sp, -8
$tmp1:
  .cfi_def_cfa_offset 8
  addiu $2, $zero, 0
  st  $2, 4($sp)
  ...
```

lbdex/chapters/Chapter3_5/Cpu0SEFrameLowering.cpp

```
// This method is called immediately before PrologEpilogInserter scans the
//  physical registers used to determine what callee saved registers should be
//  spilled. This method is optional.
void Cpu0SEFrameLowering::determineCalleeSaves(MachineFunction &MF,
                                               BitVector &SavedRegs,
                                               RegScavenger *RS) const {
//@determineCalleeSaves-body
  TargetFrameLowering::determineCalleeSaves(MF, SavedRegs, RS);
  Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
```

(continues on next page)

(continued from previous page)

```
  if (MF.getFrameInfo().hasCalls())
    setAliasRegs(MF, SavedRegs, Cpu0::LR);

  return;
}
```

The determineCalleeSaves() of Cpu0SEFrameLowering.cpp as above determine the spill registers. Once the spill registers are determined, the function SpillCalleeSavedRegisters() will save/restore registers to/from stack slots via the following code.

lbdex/chapters/Chapter3_5/Cpu0InstrInfo.h

```
  void storeRegToStackSlot(MachineBasicBlock &MBB,
                           MachineBasicBlock::iterator MBBI,
                           Register SrcReg, bool isKill, int FrameIndex,
                           const TargetRegisterClass *RC,
                           const TargetRegisterInfo *TRI) const override {
    storeRegToStack(MBB, MBBI, SrcReg, isKill, FrameIndex, RC, TRI, 0);
  }

  void loadRegFromStackSlot(MachineBasicBlock &MBB,
                            MachineBasicBlock::iterator MBBI,
                            Register DestReg, int FrameIndex,
                            const TargetRegisterClass *RC,
                            const TargetRegisterInfo *TRI) const override {
    loadRegFromStack(MBB, MBBI, DestReg, FrameIndex, RC, TRI, 0);
  }

  virtual void storeRegToStack(MachineBasicBlock &MBB,
                               MachineBasicBlock::iterator MI,
                               Register SrcReg, bool isKill, int FrameIndex,
                               const TargetRegisterClass *RC,
                               const TargetRegisterInfo *TRI,
                               int64_t Offset) const = 0;

  virtual void loadRegFromStack(MachineBasicBlock &MBB,
                                MachineBasicBlock::iterator MI,
                                Register DestReg, int FrameIndex,
                                const TargetRegisterClass *RC,
                                const TargetRegisterInfo *TRI,
                                int64_t Offset) const = 0;
```

```
  MachineMemOperand *GetMemOperand(MachineBasicBlock &MBB, int FI,
                                   MachineMemOperand::Flags Flags) const;
```

lbdex/chapters/Chapter3_5/Cpu0InstrInfo.cpp

```
MachineMemOperand *
Cpu0InstrInfo::GetMemOperand(MachineBasicBlock &MBB, int FI,
                             MachineMemOperand::Flags Flags) const {

  MachineFunction &MF = *MBB.getParent();
  MachineFrameInfo &MFI = MF.getFrameInfo();

  return MF.getMachineMemOperand(MachinePointerInfo::getFixedStack(MF, FI),
                                 Flags, MFI.getObjectSize(FI),
                                 MFI.getObjectAlign(FI));
}
```

lbdex/chapters/Chapter3_5/Cpu0SEInstrInfo.h

```
  void storeRegToStack(MachineBasicBlock &MBB,
                       MachineBasicBlock::iterator MI,
                       Register SrcReg, bool isKill, int FrameIndex,
                       const TargetRegisterClass *RC,
                       const TargetRegisterInfo *TRI,
                       int64_t Offset) const override;

  void loadRegFromStack(MachineBasicBlock &MBB,
                        MachineBasicBlock::iterator MI,
                        Register DestReg, int FrameIndex,
                        const TargetRegisterClass *RC,
                        const TargetRegisterInfo *TRI,
                        int64_t Offset) const override;
```

lbdex/chapters/Chapter3_5/Cpu0SEInstrInfo.cpp

```
void Cpu0SEInstrInfo::
storeRegToStack(MachineBasicBlock &MBB, MachineBasicBlock::iterator I,
                Register SrcReg, bool isKill, int FI,
                const TargetRegisterClass *RC, const TargetRegisterInfo *TRI,
                int64_t Offset) const {
  DebugLoc DL;
  MachineMemOperand *MMO = GetMemOperand(MBB, FI, MachineMemOperand::MOStore);

  unsigned Opc = 0;

  Opc = Cpu0::ST;
  assert(Opc && "Register class not handled!");
  BuildMI(MBB, I, DL, get(Opc)).addReg(SrcReg, getKillRegState(isKill))
    .addFrameIndex(FI).addImm(Offset).addMemOperand(MMO);
}

void Cpu0SEInstrInfo::
loadRegFromStack(MachineBasicBlock &MBB, MachineBasicBlock::iterator I,
```

(continues on next page)

(continued from previous page)

```
                Register DestReg, int FI, const TargetRegisterClass *RC,
                const TargetRegisterInfo *TRI, int64_t Offset) const {
  DebugLoc DL;
  if (I != MBB.end()) DL = I->getDebugLoc();
  MachineMemOperand *MMO = GetMemOperand(MBB, FI, MachineMemOperand::MOLoad);
  unsigned Opc = 0;

  Opc = Cpu0::LD;
  assert(Opc && "Register class not handled!");
  BuildMI(MBB, I, DL, get(Opc), DestReg).addFrameIndex(FI).addImm(Offset)
    .addMemOperand(MMO);
}
```

Functions storeRegToStack() Cpu0SEInstrInfo.cpp and storeRegToStackSlot() of Cpu0InstrInfo.cpp handle the registers spilling during register allocation process. Since each local variable connecting to a frame index, code ".addFrameIndex(FI). addImm(Offset).addMemOperand(MMO);" in storeRegToStack() where Offset is 0 is added for each virtual register. The loadRegFromStackSlot() and loadRegFromStack() will be used when it needs reload from stack slot.

If adding V0 to Cpu0CallingConv.td as the following and without both storeRegToStack() and storeRegToStackSlot() in Cpu0SEInstrInfo.cpp, Cpu0SEInstrInfo.h and Cpu0InstrInfo.h it will get the belowing error.

lbdex/Cpu0/Cpu0CallingConv.td

```
def CSR_O32 : CalleeSavedRegs<(add LR, FP, V0,
                                   (sequence "S%u", 1, 0))>;
```

```
114-43-191-19:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o -
  .text
  .section .mdebug.abiO32
  .previous
  .file "ch3.bc"
Target didn't implement TargetInstrInfo::storeRegToStackSlot!
...
Stack dump:
...
Abort trap: 6
```

Table 3.3: Backend functions called in PrologEpilogInserter.cpp

Stage	Function
Prologue/Epilogue Insertion & Frame Finalization	
• Determine spill callee saved registers	• Cpu0SEFrameLowering::determineCalleeSaves
• Spill callee saved registers	• Cpu0SEFrameLowering::spillCalleeSavedRegisters
• Prolog	• Cpu0SEFrameLowering::emitPrologue
• Epilog	• Cpu0SEFrameLowering::emitEpilogue
• Handle stack slot for local variables	• Cpu0RegisterInfo::eliminateFrameIndex

File PrologEpilogInserter.cpp includes the calling of backend functions spillCalleeSavedRegisters(), emitProlog(), emitEpilog() and eliminateFrameIndex() as follows,

lib/CodeGen/PrologEpilogInserter.cpp

```
class PEI : public MachineFunctionPass {
public:
  static char ID;
  explicit PEI(const TargetMachine *TM = nullptr) : MachineFunctionPass(ID) {
    initializePEIPass(*PassRegistry::getPassRegistry());

    if (TM && (!TM->usesPhysRegsForPEI())) {
      ...
    } else {
      SpillCalleeSavedRegisters = doSpillCalleeSavedRegs;
      ...
    }
  }
  ...
}

/// insertCSRSpillsAndRestores - Insert spill and restore code for
/// callee saved registers used in the function.
///
static void insertCSRSpillsAndRestores(MachineFunction &Fn,
                                       const MBBVector &SaveBlocks,
                                       const MBBVector &RestoreBlocks) {
  ...
  // Spill using target interface.
  for (MachineBasicBlock *SaveBlock : SaveBlocks) {
    ...
    if (!TFI->spillCalleeSavedRegisters(*SaveBlock, I, CSI, TRI)) {
      for (unsigned i = 0, e = CSI.size(); i != e; ++i) {
        // Insert the spill to the stack frame.
```

(continues on next page)

(continued from previous page)

```
        ...
        TII.storeRegToStackSlot(*SaveBlock, I, Reg, true, CSI[i].getFrameIdx(),
                                RC, TRI);
      }
    }
    ...
  }

  // Restore using target interface.
  for (MachineBasicBlock *MBB : RestoreBlocks) {
    ...
    // Restore all registers immediately before the return and any
    // terminators that precede it.
    if (!TFI->restoreCalleeSavedRegisters(*MBB, I, CSI, TRI)) {
      for (unsigned i = 0, e = CSI.size(); i != e; ++i) {
        ...
        TII.loadRegFromStackSlot(*MBB, I, Reg, CSI[i].getFrameIdx(), RC, TRI);
        ...
      }
    ...
  }
  ...
}

static void doSpillCalleeSavedRegs(MachineFunction &Fn, RegScavenger *RS,
                                   unsigned &MinCSFrameIndex,
                                   unsigned &MaxCSFrameIndex,
                                   const MBBVector &SaveBlocks,
                                   const MBBVector &RestoreBlocks) {
  const Function *F = Fn.getFunction();
  const TargetFrameLowering *TFI = Fn.getSubtarget().getFrameLowering();
  MinCSFrameIndex = std::numeric_limits<unsigned>::max();
  MaxCSFrameIndex = 0;

  // Determine which of the registers in the callee save list should be saved.
  BitVector SavedRegs;
  TFI->determineCalleeSaves(Fn, SavedRegs, RS);

  // Assign stack slots for any callee-saved registers that must be spilled.
  assignCalleeSavedSpillSlots(Fn, SavedRegs, MinCSFrameIndex, MaxCSFrameIndex);

  // Add the code to save and restore the callee saved registers.
  if (!F->hasFnAttribute(Attribute::Naked))
    insertCSRSpillsAndRestores(Fn, SaveBlocks, RestoreBlocks);
}

void PEI::insertPrologEpilogCode(MachineFunction &Fn) {
  const TargetFrameLowering &TFI = *Fn.getSubtarget().getFrameLowering();

  // Add prologue to the function...
  for (MachineBasicBlock *SaveBlock : SaveBlocks)
    TFI.emitPrologue(Fn, *SaveBlock);
```

(continues on next page)

(continued from previous page)

```
  // Add epilogue to restore the callee-save registers in each exiting block.
  for (MachineBasicBlock *RestoreBlock : RestoreBlocks)
    TFI.emitEpilogue(Fn, *RestoreBlock);
  ...
}

void PEI::replaceFrameIndices(MachineBasicBlock *BB, MachineFunction &Fn,
                              int &SPAdj) {
  ...
    for (unsigned i = 0, e = MI.getNumOperands(); i != e; ++i) {
      ...
      // If this instruction has a FrameIndex operand, we need to
      // use that target machine register info object to eliminate
      // it.
      TRI.eliminateFrameIndex(MI, SPAdj, i,
                              FrameIndexVirtualScavenging ?  nullptr : RS);
      ...
    }
  ...
}

/// replaceFrameIndices - Replace all MO_FrameIndex operands with physical
/// register references and actual offsets.
///
void PEI::replaceFrameIndices(MachineFunction &Fn) {
  ...
  // Iterate over the reachable blocks in DFS order.
  for (auto DFI = df_ext_begin(&Fn, Reachable), DFE = df_ext_end(&Fn, Reachable);
       DFI != DFE; ++DFI) {
    ...
    replaceFrameIndices(BB, Fn, SPAdj);
    ...
  }

  // Handle the unreachable blocks.
  for (auto &BB : Fn) {
    ...
    replaceFrameIndices(&BB, Fn, SPAdj);
  }
}

bool PEI::runOnMachineFunction(MachineFunction &Fn) {
  const TargetFrameLowering &TFI = *Fn.getSubtarget().getFrameLowering();
  ...
  FrameIndexVirtualScavenging = TRI->requiresFrameIndexScavenging(Fn);
  ...
  // Handle CSR spilling and restoring, for targets that need it.
  SpillCalleeSavedRegisters(Fn, RS, MinCSFrameIndex, MaxCSFrameIndex,
                            SaveBlocks, RestoreBlocks);
  ...
  // Calculate actual frame offsets for all abstract stack objects...
```

(continues on next page)

(continued from previous page)

```
  calculateFrameObjectOffsets(Fn);

  // Add prolog and epilog code to the function.  This function is required
  // to align the stack frame as necessary for any stack variables or
  // called functions.  Because of this, calculateCalleeSavedRegisters()
  // must be called before this function in order to set the AdjustsStack
  // and MaxCallFrameSize variables.
  if (!F->hasFnAttribute(Attribute::Naked))
    insertPrologEpilogCode(Fn);

  // Replace all MO_FrameIndex operands with physical register references
  // and actual offsets.
  //
  replaceFrameIndices(Fn);

  // If register scavenging is needed, as we've enabled doing it as a
  // post-pass, scavenge the virtual registers that frame index elimination
  // inserted.
  if (TRI->requiresRegisterScavenging(Fn) && FrameIndexVirtualScavenging) {
    ScavengeFrameVirtualRegs(Fn, RS);

    // Clear any vregs created by virtual scavenging.
    Fn.getRegInfo().clearVirtRegs();
  }
  ...
}
```

3.5.4 Large stack

At this point, we have translated the very simple main() function with "return 0;" single instruction. Though Cpu0AnalyzeImmediate.cpp and the Cpu0InstrInfo.td instructions defined in Chapter3_5 as the following, which take care the 32 bits stack size adjustments. However the CH9_3, dynamic-stack-allocation-support bring instruction "move $fp, $sp" that make it complicated in coding against the tutoral purpose of Cpu0.[4]

lbdex/chapters/Chapter3_5/CMakeLists.txt

```
add_llvm_target(
  ...
```

```
  Cpu0AnalyzeImmediate.cpp
```

```
  ...
)
```

[4] http://jonathan2251.github.io/lbd/funccall.html#dynamic-stack-allocation-support

lbdex/chapters/Chapter3_5/Cpu0AnalyzeImmediate.h

```
//===-- Cpu0AnalyzeImmediate.h - Analyze Immediates ------------*- C++ -*--===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
#ifndef CPU0_ANALYZE_IMMEDIATE_H
#define CPU0_ANALYZE_IMMEDIATE_H

#include "Cpu0Config.h"
#if CH >= CH3_5

#include "llvm/ADT/SmallVector.h"
#include "llvm/Support/DataTypes.h"

namespace llvm {

  class Cpu0AnalyzeImmediate {
  public:
    struct Inst {
      unsigned Opc, ImmOpnd;
      Inst(unsigned Opc, unsigned ImmOpnd);
    };
    typedef SmallVector<Inst, 7 > InstSeq;

    /// Analyze - Get an instruction sequence to load immediate Imm. The last
    /// instruction in the sequence must be an ADDiu if LastInstrIsADDiu is
    /// true;
    const InstSeq &Analyze(uint64_t Imm, unsigned Size, bool LastInstrIsADDiu);
  private:
    typedef SmallVector<InstSeq, 5> InstSeqLs;

    /// AddInstr - Add I to all instruction sequences in SeqLs.
    void AddInstr(InstSeqLs &SeqLs, const Inst &I);

    /// GetInstSeqLsADDiu - Get instruction sequences which end with an ADDiu to
    /// load immediate Imm
    void GetInstSeqLsADDiu(uint64_t Imm, unsigned RemSize, InstSeqLs &SeqLs);

    /// GetInstSeqLsORi - Get instruction sequences which end with an ORi to
    /// load immediate Imm
    void GetInstSeqLsORi(uint64_t Imm, unsigned RemSize, InstSeqLs &SeqLs);

    /// GetInstSeqLsSHL - Get instruction sequences which end with a SHL to
    /// load immediate Imm
    void GetInstSeqLsSHL(uint64_t Imm, unsigned RemSize, InstSeqLs &SeqLs);

    /// GetInstSeqLs - Get instruction sequences to load immediate Imm.
    void GetInstSeqLs(uint64_t Imm, unsigned RemSize, InstSeqLs &SeqLs);
```

(continues on next page)

(continued from previous page)

```
    /// ReplaceADDiuSHLWithLUi - Replace an ADDiu & SHL pair with a LUi.
    void ReplaceADDiuSHLWithLUi(InstSeq &Seq);

    /// GetShortestSeq - Find the shortest instruction sequence in SeqLs and
    /// return it in Insts.
    void GetShortestSeq(InstSeqLs &SeqLs, InstSeq &Insts);

    unsigned Size;
    unsigned ADDiu, ORi, SHL, LUi;
    InstSeq Insts;
  };
}

#endif // #if CH >= CH3_5

#endif
```

lbdex/chapters/Chapter3_5/Cpu0AnalyzeImmediate.cpp

```
//===-- Cpu0AnalyzeImmediate.cpp - Analyze Immediates ----------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
#include "Cpu0AnalyzeImmediate.h"
#include "Cpu0.h"
#if CH >= CH3_5

#include "llvm/Support/MathExtras.h"

using namespace llvm;

Cpu0AnalyzeImmediate::Inst::Inst(unsigned O, unsigned I) : Opc(O), ImmOpnd(I) {}

// Add I to the instruction sequences.
void Cpu0AnalyzeImmediate::AddInstr(InstSeqLs &SeqLs, const Inst &I) {
  // Add an instruction seqeunce consisting of just I.
  if (SeqLs.empty()) {
    SeqLs.push_back(InstSeq(1, I));
    return;
  }

  for (InstSeqLs::iterator Iter = SeqLs.begin(); Iter != SeqLs.end(); ++Iter)
    Iter->push_back(I);
}
```

(continues on next page)

(continued from previous page)

```
void Cpu0AnalyzeImmediate::GetInstSeqLsADDiu(uint64_t Imm, unsigned RemSize,
                                             InstSeqLs &SeqLs) {
  GetInstSeqLs((Imm + 0x8000ULL) & 0xffffffffffff0000ULL, RemSize, SeqLs);
  AddInstr(SeqLs, Inst(ADDiu, Imm & 0xffffULL));
}

void Cpu0AnalyzeImmediate::GetInstSeqLsORi(uint64_t Imm, unsigned RemSize,
                                           InstSeqLs &SeqLs) {
  GetInstSeqLs(Imm & 0xffffffffffff0000ULL, RemSize, SeqLs);
  AddInstr(SeqLs, Inst(ORi, Imm & 0xffffULL));
}

void Cpu0AnalyzeImmediate::GetInstSeqLsSHL(uint64_t Imm, unsigned RemSize,
                                           InstSeqLs &SeqLs) {
  unsigned Shamt = countTrailingZeros(Imm);
  GetInstSeqLs(Imm >> Shamt, RemSize - Shamt, SeqLs);
  AddInstr(SeqLs, Inst(SHL, Shamt));
}

void Cpu0AnalyzeImmediate::GetInstSeqLs(uint64_t Imm, unsigned RemSize,
                                        InstSeqLs &SeqLs) {
  uint64_t MaskedImm = Imm & (0xffffffffffffffffULL >> (64 - Size));

  // Do nothing if Imm is 0.
  if (!MaskedImm)
    return;

  // A single ADDiu will do if RemSize <= 16.
  if (RemSize <= 16) {
    AddInstr(SeqLs, Inst(ADDiu, MaskedImm));
    return;
  }

  // Shift if the lower 16-bit is cleared.
  if (!(Imm & 0xffff)) {
    GetInstSeqLsSHL(Imm, RemSize, SeqLs);
    return;
  }

  GetInstSeqLsADDiu(Imm, RemSize, SeqLs);

  // If bit 15 is cleared, it doesn't make a difference whether the last
  // instruction is an ADDiu or ORi. In that case, do not call GetInstSeqLsORi.
  if (Imm & 0x8000) {
    InstSeqLs SeqLsORi;
    GetInstSeqLsORi(Imm, RemSize, SeqLsORi);
    SeqLs.insert(SeqLs.end(), SeqLsORi.begin(), SeqLsORi.end());
  }
}

// Replace a ADDiu & SHL pair with a LUi.
// e.g. the following two instructions
```

(continues on next page)

(continued from previous page)

```
//  ADDiu 0x0111
//  SHL 18
// are replaced with
//  LUi 0x444
void Cpu0AnalyzeImmediate::ReplaceADDiuSHLWithLUi(InstSeq &Seq) {
  // Check if the first two instructions are ADDiu and SHL and the shift amount
  // is at least 16.
  if ((Seq.size() < 2) || (Seq[0].Opc != ADDiu) ||
      (Seq[1].Opc != SHL) || (Seq[1].ImmOpnd < 16))
    return;

  // Sign-extend and shift operand of ADDiu and see if it still fits in 16-bit.
  int64_t Imm = SignExtend64<16>(Seq[0].ImmOpnd);
  int64_t ShiftedImm = (uint64_t)Imm << (Seq[1].ImmOpnd - 16);

  if (!isInt<16>(ShiftedImm))
    return;

  // Replace the first instruction and erase the second.
  Seq[0].Opc = LUi;
  Seq[0].ImmOpnd = (unsigned)(ShiftedImm & 0xffff);
  Seq.erase(Seq.begin() + 1);
}

void Cpu0AnalyzeImmediate::GetShortestSeq(InstSeqLs &SeqLs, InstSeq &Insts) {
  InstSeqLs::iterator ShortestSeq = SeqLs.end();
  // The length of an instruction sequence is at most 7.
  unsigned ShortestLength = 8;

  for (InstSeqLs::iterator S = SeqLs.begin(); S != SeqLs.end(); ++S) {
    ReplaceADDiuSHLWithLUi(*S);
    assert(S->size() <= 7);

    if (S->size() < ShortestLength) {
      ShortestSeq = S;
      ShortestLength = S->size();
    }
  }

  Insts.clear();
  Insts.append(ShortestSeq->begin(), ShortestSeq->end());
}

const Cpu0AnalyzeImmediate::InstSeq
&Cpu0AnalyzeImmediate::Analyze(uint64_t Imm, unsigned Size,
                               bool LastInstrIsADDiu) {
  this->Size = Size;

  ADDiu = Cpu0::ADDiu;
  ORi = Cpu0::ORi;
  SHL = Cpu0::SHL;
  LUi = Cpu0::LUi;
```

(continues on next page)

(continued from previous page)

```
  InstSeqLs SeqLs;

  // Get the list of instruction sequences.
  if (LastInstrIsADDiu | !Imm)
    GetInstSeqLsADDiu(Imm, Size, SeqLs);
  else
    GetInstSeqLs(Imm, Size, SeqLs);

  // Set Insts to the shortest instruction sequence.
  GetShortestSeq(SeqLs, Insts);

  return Insts;
}

#endif
```

lbdex/chapters/Chapter3_5/Cpu0InstrInfo.h

```
#include "Cpu0AnalyzeImmediate.h"
```

lbdex/chapters/Chapter3_5/Cpu0InstrInfo.td

```
class Cpu0InstAlias<string Asm, dag Result, bit Emit = 0b1> :
  InstAlias<Asm, Result, Emit>;
```

```
def shamt       : Operand<i32>;

// Unsigned Operand
def uimm16      : Operand<i32> {
  let PrintMethod = "printUnsignedImm";
}
```

```
// Transformation Function - get the lower 16 bits.
def LO16 : SDNodeXForm<imm, [{
  return getImm(N, N->getZExtValue() & 0xffff);
}]>;

// Transformation Function - get the higher 16 bits.
def HI16 : SDNodeXForm<imm, [{
  return getImm(N, (N->getZExtValue() >> 16) & 0xffff);
}]>;
```

```
// Node immediate fits as 16-bit zero extended on target immediate.
// The LO16 param means that only the lower 16 bits of the node
// immediate are caught.
// e.g. addiu, sltiu
def immZExt16  : PatLeaf<(imm), [{
```

(continues on next page)

(continued from previous page)

```
  if (N->getValueType(0) == MVT::i32)
    return (uint32_t)N->getZExtValue() == (unsigned short)N->getZExtValue();
  else
    return (uint64_t)N->getZExtValue() == (unsigned short)N->getZExtValue();
}], LO16>;

// Immediate can be loaded with LUi (32-bit int with lower 16-bit cleared).
def immLow16Zero : PatLeaf<(imm), [{
  int64_t Val = N->getSExtValue();
  return isInt<32>(Val) && !(Val & 0xffff);
}]>;

// shamt field must fit in 5 bits.
def immZExt5 : ImmLeaf<i32, [{return Imm == (Imm & 0x1f);}]>;
```

```
let Predicates = [Ch3_5] in {
// Arithmetic and logical instructions with 3 register operands.
class ArithLogicR<bits<8> op, string instr_asm, SDNode OpNode,
                  InstrItinClass itin, RegisterClass RC, bit isComm = 0>:
  FA<op, (outs GPROut:$ra), (ins RC:$rb, RC:$rc),
     !strconcat(instr_asm, "\t$ra, $rb, $rc"),
     [(set GPROut:$ra, (OpNode RC:$rb, RC:$rc))], itin> {
  let shamt = 0;
  let isCommutable = isComm;        // e.g. add rb rc =  add rc rb
  let isReMaterializable = 1;
}
}
```

```
let Predicates = [Ch3_5] in {
// Shifts
class shift_rotate_imm<bits<8> op, bits<4> isRotate, string instr_asm,
                       SDNode OpNode, PatFrag PF, Operand ImmOpnd,
                       RegisterClass RC>:
  FA<op, (outs GPROut:$ra), (ins RC:$rb, ImmOpnd:$shamt),
     !strconcat(instr_asm, "\t$ra, $rb, $shamt"),
     [(set GPROut:$ra, (OpNode RC:$rb, PF:$shamt))], IIAlu> {
  let rc = 0;
}

// 32-bit shift instructions.
class shift_rotate_imm32<bits<8> op, bits<4> isRotate, string instr_asm,
                         SDNode OpNode>:
  shift_rotate_imm<op, isRotate, instr_asm, OpNode, immZExt5, shamt, CPURegs>;
}
```

```
let Predicates = [Ch3_5] in {
// Load Upper Imediate
class LoadUpper<bits<8> op, string instr_asm, RegisterClass RC, Operand Imm>:
  FL<op, (outs RC:$ra), (ins Imm:$imm16),
     !strconcat(instr_asm, "\t$ra, $imm16"), [], IIAlu> {
  let rb = 0;
```

(continues on next page)

(continued from previous page)

```
  let isReMaterializable = 1;
}
}
```

```
let Predicates = [Ch3_5] in {
def ORi     : ArithLogicI<0x0d, "ori", or, uimm16, immZExt16, CPURegs>;
}
```

```
let Predicates = [Ch3_5] in {
def LUi     : LoadUpper<0x0F, "lui", GPROut, uimm16>;
}
```

```
let Predicates = [Ch3_5] in {
let Predicates = [DisableOverflow] in {
def ADDu    : ArithLogicR<0x11, "addu", add, IIAlu, CPURegs, 1>;
}
}
```

```
let Predicates = [Ch3_5] in {
def SHL     : shift_rotate_imm32<0x1e, 0x00, "shl", shl>;
}
```

```
let Predicates = [Ch3_5] in {
//===----------------------------------------------------------------------===//
// Instruction aliases
//===----------------------------------------------------------------------===//
def : Cpu0InstAlias<"move $dst, $src",
                    (ADDu GPROut:$dst, GPROut:$src,ZERO), 1>;
}
```

```
let Predicates = [Ch3_5] in {
def : Pat<(i32 immZExt16:$in),
          (ORi ZERO, imm:$in)>;
def : Pat<(i32 immLow16Zero:$in),
          (LUi (HI16 imm:$in))>;

// Arbitrary immediates
def : Pat<(i32 imm:$imm),
          (ORi (LUi (HI16 imm:$imm)), (LO16 imm:$imm))>;
} // let Predicates = [Ch3_4]
```

The Cpu0AnalyzeImmediate.cpp written in recursive with a little complicate in logic. However, the recursive skills is used in the front end compile book, you should fimiliar with it. Instead of tracking the code, listing the stack size and the instructions generated in "Table: Cpu0 stack adjustment instructions before replace addiu and shl with lui instruction" as follows and "Table: Cpu0 stack adjustment instructions after replace addiu and shl with lui instruction" at next,

Table 3.4: Cpu0 stack adjustment instructions before replace addiu and shl with lui instruction

stack size range	ex. stack size	Cpu0 Prologue instructions	Cpu0 Epilogue instructions
0 ~ 0x7ff8	• 0x7ff8	• addiu $sp, $sp, -32760;	• addiu $sp, $sp, 32760;
0x8000 ~ 0xfff8	• 0x8000	• addiu $sp, $sp, -32768;	• addiu $1, $zero, 1; • shl $1, $1, 16; • addiu $1, $1, -32768; • addu $sp, $sp, $1;
x10000 ~ 0xfffffff8	• 0x7ffffff8	• addiu $1, $zero, 8; • shl $1, $1, 28; • addiu $1, $1, 8; • addu $sp, $sp, $1;	• addiu $1, $zero, 8; • shl $1, $1, 28; • addiu $1, $1, -8; • addu $sp, $sp, $1;
x10000 ~ 0xffffffff8	• 0x90008000	• addiu $1, $zero, -9; • shl $1, $1, 28; • addiu $1, $1, -32768; • addu $sp, $sp, $1;	• addiu $1, $zero, -28671; • shl $1, $1, 16 • addiu $1, $1, -32768; • addu $sp, $sp, $1;

Since the Cpu0 stack is 8 bytes alignment, addresses from 0x7ff9 to 0x7fff are impossible existing.

Assume sp = 0xa0008000 and stack size = 0x90008000, then (0xa0008000 - 0x90008000) => 0x10000000. Verify with the Cpu0 Prologue instructions as follows,

1. "addiu $1, $zero, -9" => ($1 = 0 + 0xfffffff7) => $1 = 0xfffffff7.
2. "shl $1, $1, 28;" => $1 = 0x70000000.
3. "addiu $1, $1, -32768" => $1 = (0x70000000 + 0xffff8000) => $1 = 0x6fff8000.
4. "addu $sp, $sp, $1" => $sp = (0xa0008000 + 0x6fff8000) => $sp = 0x10000000.

Verify with the Cpu0 Epilogue instructions with sp = 0x10000000 and stack size = 0x90008000 as follows,

1. "addiu $1, $zero, -28671" => ($1 = 0 + 0xffff9001) => $1 = 0xffff9001.
2. "shl $1, $1, 16;" => $1 = 0x90010000.
3. "addiu $1, $1, -32768" => $1 = (0x90010000 + 0xffff8000) => $1 = 0x90008000.
4. "addu $sp, $sp, $1" => $sp = (0x10000000 + 0x90008000) => $sp = 0xa0008000.

The Cpu0AnalyzeImmediate::GetShortestSeq() will call Cpu0AnalyzeImmediate:: ReplaceADDiuSHLWithLUi() to replace addiu and shl with single instruction lui only. The effect as the following table.

Table 3.5: Cpu0 stack adjustment instructions after replace addiu and shl with lui instruction

stack size range	ex. stack size	Cpu0 Prologue instructions	Cpu0 Epilogue instructions
0x8000 ~ 0xfff8	• 0x8000	• addiu $sp, $sp, -32768;	• ori $1, $zero, 32768; • addu $sp, $sp, $1;
x10000 ~ 0xffffff8	• 0x7ffffff8	• lui $1, 32768; • addiu $1, $1, 8; • addu $sp, $sp, $1;	• lui $1, 32767; • ori $1, $1, 65528 • addu $sp, $sp, $1;
x10000 ~ 0xffffffff8	• 0x90008000	• lui $1, 28671; • ori $1, $1, 32768; • addu $sp, $sp, $1;	• lui $1, 36865; • addiu $1, $1, -32768; • addu $sp, $sp, $1;

Assume sp = 0xa0008000 and stack size = 0x90008000, then (0xa0008000 - 0x90008000) => 0x10000000. Verify with the Cpu0 Prologue instructions as follows,

1. "lui $1, 28671" => $1 = 0x6fff0000.
2. "ori $1, $1, 32768" => $1 = (0x6fff0000 + 0x00008000) => $1 = 0x6fff8000.
3. "addu $sp, $sp, $1" => $sp = (0xa0008000 + 0x6fff8000) => $sp = 0x10000000.

Verify with the Cpu0 Epilogue instructions with sp = 0x10000000 and stack size = 0x90008000 as follows,

1. "lui $1, 36865" => $1 = 0x90010000.
2. "addiu $1, $1, -32768" => $1 = (0x90010000 + 0xffff8000) => $1 = 0x90008000.
3. "addu $sp, $sp, $1" => $sp = (0x10000000 + 0x90008000) => $sp = 0xa0008000.

File ch3_largeframe.cpp include the large frame test.

Run Chapter3_5 with ch3_largeframe.cpp will get the following result.

lbdex/input/ch3_largeframe.cpp

```
int test_largegframe() {
  int a[469753856];

  return 0;
}
```

```
118-165-78-12:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch3_largeframe.cpp -emit-llvm -o ch3_largeframe.bc
118-165-78-12:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch3_largeframe.bc.bc -o -
  ...
```

(continues on next page)

(continued from previous page)

```
  .section .mdebug.abi32
  .previous
  .file "ch3_largeframe.bc"
  .globl  _Z16test_largegframev
  .align  2
  .type _Z16test_largegframev,@function
  .ent  _Z16test_largegframev   # @_Z16test_largegframev
_Z16test_largegframev:
  .frame  $fp,1879015424,$lr
  .mask   0x00000000,0
  .set  noreorder
  .set  nomacro
  .set  noat
# BB#0:
  lui $1, 36865
  addiu $1, $1, -32768
  addu  $sp, $sp, $1
  addiu $2, $zero, 0
  lui $1, 28672
  addiu $1, $1, -32768
  addu  $sp, $sp, $1
  ret $lr
  .set  at
  .set  macro
  .set  reorder
  .end  _Z16test_largegframev
$func_end0:
  .size _Z16test_largegframev, ($func_end0)-_Z16test_largegframev
```

3.6 Data operands DAGs

From above or compiler book, you can see all the OP code are the internal nodes in DAGs graph, and operands are the leaves of DAGs. To develop your backend, you can copy the related data operands DAGs node from other backend since the IR data nodes are take cared by all the backend. About the data DAGs nodes, you can understand some of them through the Cpu0InstrInfo.td and find them by command, grep -R "<datadag>" `find llvm/include/llvm`, with spending a little more time to think or guess about it. Some data DAGs we know more, some we know a little and some remains unknown but it's OK for us. List some of data DAGs we understand and occured until now as follows,

include/llvm/Target/TargetSelectionDAG.td

```
// PatLeaf's are pattern fragments that have no operands.  This is just a helper
// to define immediates and other common things concisely.
class PatLeaf<dag frag, code pred = [{}], SDNodeXForm xform = NOOP_SDNodeXForm>
 : PatFrag<(ops), frag, pred, xform>;

// ImmLeaf is a pattern fragment with a constraint on the immediate.  The
// constraint is a function that is run on the immediate (always with the value
// sign extended out to an int64_t) as Imm.  For example:
//
```

(continues on next page)

(continued from previous page)

```
//  def immSExt8 : ImmLeaf<i16, [{ return (char)Imm == Imm; }]>;
//
// this is a more convenient form to match 'imm' nodes in than PatLeaf and also
// is preferred over using PatLeaf because it allows the code generator to
// reason more about the constraint.
//
// If FastIsel should ignore all instructions that have an operand of this type,
// the FastIselShouldIgnore flag can be set.  This is an optimization to reduce
// the code size of the generated fast instruction selector.
class ImmLeaf<ValueType vt, code pred, SDNodeXForm xform = NOOP_SDNodeXForm>
  : PatFrag<(ops), (vt imm), [{}], xform> {
  let ImmediateCode = pred;
  bit FastIselShouldIgnore = 0;
}
```

lbdex/chapters/Chapter3_5/Cpu0InstrInfo.td

```
// Signed Operand
def simm16      : Operand<i32> {
  let DecoderMethod= "DecodeSimm16";
}
```

```
def shamt       : Operand<i32>;

// Unsigned Operand
def uimm16      : Operand<i32> {
  let PrintMethod = "printUnsignedImm";
}
```

```
// Address operand
def mem : Operand<iPTR> {
  let PrintMethod = "printMemOperand";
  let MIOperandInfo = (ops GPROut, simm16);
  let EncoderMethod = "getMemEncoding";
```

```
}
```

```
// Transformation Function - get the lower 16 bits.
def LO16 : SDNodeXForm<imm, [{
  return getImm(N, N->getZExtValue() & 0xffff);
}]>;

// Transformation Function - get the higher 16 bits.
def HI16 : SDNodeXForm<imm, [{
  return getImm(N, (N->getZExtValue() >> 16) & 0xffff);
}]>;
```

```
// Node immediate fits as 16-bit sign extended on target immediate.
// e.g. addi, andi
```

(continues on next page)

(continued from previous page)

```
def immSExt16  : PatLeaf<(imm), [{ return isInt<16>(N->getSExtValue()); }]>;
```

```
// Node immediate fits as 16-bit zero extended on target immediate.
// The LO16 param means that only the lower 16 bits of the node
// immediate are caught.
// e.g. addiu, sltiu
def immZExt16  : PatLeaf<(imm), [{
  if (N->getValueType(0) == MVT::i32)
    return (uint32_t)N->getZExtValue() == (unsigned short)N->getZExtValue();
  else
    return (uint64_t)N->getZExtValue() == (unsigned short)N->getZExtValue();
}], LO16>;

// Immediate can be loaded with LUi (32-bit int with lower 16-bit cleared).
def immLow16Zero : PatLeaf<(imm), [{
  int64_t Val = N->getSExtValue();
  return isInt<32>(Val) && !(Val & 0xffff);
}]>;

// shamt field must fit in 5 bits.
def immZExt5 : ImmLeaf<i32, [{return Imm == (Imm & 0x1f);}]>;
```

```
// Cpu0 Address Mode! SDNode frameindex could possibily be a match
// since load and store instructions from stack used it.
def addr :
  ComplexPattern<iPTR, 2, "SelectAddr", [frameindex], [SDNPWantParent]>;

//===----------------------------------------------------------------------===//
// Pattern fragment for load/store
//===----------------------------------------------------------------------===//

class AlignedLoad<PatFrag Node> :
  PatFrag<(ops node:$ptr), (Node node:$ptr), [{
  LoadSDNode *LD = cast<LoadSDNode>(N);
  return LD->getMemoryVT().getSizeInBits()/8 <= LD->getAlignment();
}]>;

class AlignedStore<PatFrag Node> :
  PatFrag<(ops node:$val, node:$ptr), (Node node:$val, node:$ptr), [{
  StoreSDNode *SD = cast<StoreSDNode>(N);
  return SD->getMemoryVT().getSizeInBits()/8 <= SD->getAlignment();
}]>;
```

```
def load_a          : AlignedLoad<load>;
```

```
def store_a         : AlignedStore<store>;
```

As mentioned in sub-section "instruction selection" of last chapter, immSExt16 is a data leaf DAG node and it will return true if its value is in the range of signed 16 bits integer. The load_a, store_a and others are similar but they check with alignment.

The mem is explained in chapter3_2 for print operand; addr is explained in chapter3_3 for data DAG selection. The

simm16, ..., inherited from Operand<i32> because Cpu0 is 32 bits. It may over 16 bits, so immSExt16 pattern leaf is used to control it as example ADDiu mention in last chapter. PatLeaf immZExt16, immLow16Zero and ImmLeaf immZExt5 are similar to immSExt16.

3.7 Summary of this Chapter

Summary the functions for llvm backend stages as the following table.

```
118-165-79-200:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc
-debug-pass=Structure -o -
...
Machine Branch Probability Analysis
  ModulePass Manager
    FunctionPass Manager
      ...
      CPU0 DAG->DAG Pattern Instruction Selection
        Initial selection DAG
        Optimized lowered selection DAG
        Type-legalized selection DAG
        Optimized type-legalized selection DAG
        Legalized selection DAG
        Optimized legalized selection DAG
        Instruction selection
        Selected selection DAG
        Scheduling
      ...
      Greedy Register Allocator
      ...
      Prologue/Epilogue Insertion & Frame Finalization
      ...
      Post-RA pseudo instruction expansion pass
      ...
      Cpu0 Assembly Printer
```

Table 3.6: Functions for llvm backend stages

Stage	Function
Before CPU0 DAG->DAG Pattern Instruction Selection	• Cpu0TargetLowering::LowerFormalArguments • Cpu0TargetLowering::LowerReturn
Instruction selection	• Cpu0DAGToDAGISel::Select
Prologue/Epilogue Insertion & Frame Finalization	
• Determine spill callee saved registers	• Cpu0SEFrameLowering::determineCalleeSaves
• Spill callee saved registers	• Cpu0SEFrameLowering::spillCalleeSavedRegisters
• Prolog	• Cpu0SEFrameLowering::emitPrologue
• Epilog	• Cpu0SEFrameLowering::emitEpilogue
• Handle stack slot for local variables	• Cpu0RegisterInfo::eliminateFrameIndex
Post-RA pseudo instruction expansion pass	• Cpu0SEInstrInfo::expandPostRAPseudo
Cpu0 Assembly Printer	• Cpu0AsmPrinter.cpp, Cpu0MCInstLower.cpp • Cpu0InstPrinter.cpp

We add a pass in Instruction Section stage in section "Add Cpu0DAGToDAGISel class". You can embed your code into other passes like that. Please check CodeGen/Passes.h for the information. Remember the pass is called according the function unit as the `llc -debug-pass=Structure` indicated.

We have finished a simple compiler for cpu0 which only supports **ld**, **st**, **addiu**, **ori**, **lui**, **addu**, **shl** and **ret** 8 instructions.

We are satisfied with this result. But you may think "After so much code we program, and just get these 8 instructions!". The point is we have created a frame work for Cpu0 target machine (please look back the llvm backend structure class inheritance tree early in this chapter). Until now, we have over 3000 lines of source code with comments which include files *.cpp, *.h, *.td and CMakeLists.txt. It can be counted by command `wc `find dir -name *.cpp`` for files *.cpp, *.h, *.td, *.txt. LLVM front end tutorial have 700 lines of source code without comments in total. Don't feel down with this result. In reality, writing a backend is warm up slowly but run fastly. Clang has over 500,000 lines of source code with comments in clang/lib directory which include C++ and Obj C support. Mips backend of llvm 3.1 has only 15,000 lines with comments. Even the complicate X86 CPU which CISC outside and RISC inside (micro instruction), has only 45,000 lines in llvm 3.1 with comments. In next chapter, we will show you that add a new instruction support is as easy as 123.

CHAPTER

FOUR

ARITHMETIC AND LOGIC INSTRUCTIONS

- *Arithmetic*
  - **+, -, *, <<,** *and* **>>**
  - *Display llvm IR nodes with Graphviz*
  - *Operator % and /*
    - *The DAG of %*
    - *Arm solution*
    - *Mips solution*
    - *Full support %, and /*
  - *Rotate instructions*
- *Logic*
- *Summary*

This chapter adds more Cpu0 arithmetic instructions support first. The section Display llvm IR nodes with Graphviz will show you the steps of DAG optimization and their corresponding `llc` display options. These DAGs translation existed in some steps of optimization can be displayed by the graphic tool of Graphviz which supply useful information in graphic view. Logic instructions support will come after arithmetic section. In spite of that llvm backend handle the IR only, we get the IR from the corresponding C operators with designed C example code. Instead of focusing on classes relationship in this backend structure of last chapter, readers should focus on the mapping of C operators to llvm IR and how to define the mapping relationship of IR and instructions in td. HILO and C0 register class are defined in this chapter. Readers will know how to handle other register classes beside general purpose register class, and why they are needed, from this chapter.

4.1 Arithmetic

The code added in Chapter4_1/ to support arithmetic instructions as follows,

lbdex/chapters/Chapter4_1/Cpu0Subtarget.cpp

```
static cl::opt<bool> EnableOverflowOpt
                ("cpu0-enable-overflow", cl::Hidden, cl::init(false),
                 cl::desc("Use trigger overflow instructions add and sub \
                 instead of non-overflow instructions addu and subu"));
```

```
Cpu0Subtarget::Cpu0Subtarget(const Triple &TT, StringRef CPU,
                             StringRef FS, bool little,
                             const Cpu0TargetMachine &_TM) :
```

```
...
```

```
  EnableOverflow = EnableOverflowOpt;
```

```
  ...
}
```

lbdex/chapters/Chapter4_1/Cpu0InstrInfo.td

```
// Only op DAG can be disabled by ch4_1, data DAG cannot.
def SDT_Cpu0DivRem       : SDTypeProfile<0, 2,
                                         [SDTCisInt<0>,
                                          SDTCisSameAs<0, 1>]>;
```

```
// DivRem(u) nodes
def Cpu0DivRem    : SDNode<"Cpu0ISD::DivRem", SDT_Cpu0DivRem,
                           [SDNPOutGlue]>;
def Cpu0DivRemU   : SDNode<"Cpu0ISD::DivRemU", SDT_Cpu0DivRem,
                           [SDNPOutGlue]>;
```

```
let Predicates = [Ch4_1] in {
class shift_rotate_reg<bits<8> op, bits<4> isRotate, string instr_asm,
                       SDNode OpNode, RegisterClass RC>:
  FA<op, (outs GPROut:$ra), (ins RC:$rb, RC:$rc),
     !strconcat(instr_asm, "\t$ra, $rb, $rc"),
     [(set GPROut:$ra, (OpNode RC:$rb, RC:$rc))], IIAlu> {
  let shamt = 0;
}
}
```

```
let Predicates = [Ch4_1] in {
// Mul, Div
class Mult<bits<8> op, string instr_asm, InstrItinClass itin,
```

(continues on next page)

(continued from previous page)

```
           RegisterClass RC, list<Register> DefRegs>:
  FA<op, (outs), (ins RC:$ra, RC:$rb),
     !strconcat(instr_asm, "\t$ra, $rb"), [], itin> {
  let rc = 0;
  let shamt = 0;
  let isCommutable = 1;
  let Defs = DefRegs;
  let hasSideEffects = 0;
}

class Mult32<bits<8> op, string instr_asm, InstrItinClass itin>:
  Mult<op, instr_asm, itin, CPURegs, [HI, LO]>;

class Div<SDNode opNode, bits<8> op, string instr_asm, InstrItinClass itin,
          RegisterClass RC, list<Register> DefRegs>:
  FA<op, (outs), (ins RC:$ra, RC:$rb),
     !strconcat(instr_asm, "\t$ra, $rb"),
     [(opNode RC:$ra, RC:$rb)], itin> {
  let rc = 0;
  let shamt = 0;
  let Defs = DefRegs;
}

class Div32<SDNode opNode, bits<8> op, string instr_asm, InstrItinClass itin>:
  Div<opNode, op, instr_asm, itin, CPURegs, [HI, LO]>;

// Move from Lo/Hi
class MoveFromLOHI<bits<8> op, string instr_asm, RegisterClass RC,
                   list<Register> UseRegs>:
  FA<op, (outs RC:$ra), (ins),
     !strconcat(instr_asm, "\t$ra"), [], IIHiLo> {
  let rb = 0;
  let rc = 0;
  let shamt = 0;
  let Uses = UseRegs;
  let hasSideEffects = 0;
}

// Move to Lo/Hi
class MoveToLOHI<bits<8> op, string instr_asm, RegisterClass RC,
                 list<Register> DefRegs>:
  FA<op, (outs), (ins RC:$ra),
     !strconcat(instr_asm, "\t$ra"), [], IIHiLo> {
  let rb = 0;
  let rc = 0;
  let shamt = 0;
  let Defs = DefRegs;
  let hasSideEffects = 0;
}

// Move from C0 (co-processor 0) Register
class MoveFromC0<bits<8> op, string instr_asm, RegisterClass RC>:
```

(continues on next page)

(continued from previous page)

```
  FA<op, (outs), (ins RC:$ra, C0Regs:$rb),
     !strconcat(instr_asm, "\t$ra, $rb"), [], IIAlu> {
  let rc = 0;
  let shamt = 0;
  let hasSideEffects = 0;
}

// Move to C0 Register
class MoveToC0<bits<8> op, string instr_asm, RegisterClass RC>:
  FA<op, (outs C0Regs:$ra), (ins RC:$rb),
     !strconcat(instr_asm, "\t$ra, $rb"), [], IIAlu> {
  let rc = 0;
  let shamt = 0;
  let hasSideEffects = 0;
}

// Move from C0 register to C0 register
class C0Move<bits<8> op, string instr_asm>:
  FA<op, (outs C0Regs:$ra), (ins C0Regs:$rb),
     !strconcat(instr_asm, "\t$ra, $rb"), [], IIAlu> {
  let rc = 0;
  let shamt = 0;
  let hasSideEffects = 0;
}
} // let Predicates = [Ch4_1]
```

```
let Predicates = [Ch4_1] in {
let Predicates = [DisableOverflow] in {
def SUBu    : ArithLogicR<0x12, "subu", sub, IIAlu, CPURegs>;
}
let Predicates = [EnableOverflow] in {
def ADD     : ArithLogicR<0x13, "add", add, IIAlu, CPURegs, 1>;
def SUB     : ArithLogicR<0x14, "sub", sub, IIAlu, CPURegs>;
}
def MUL     : ArithLogicR<0x17, "mul", mul, IIImul, CPURegs, 1>;
}
```

```
let Predicates = [Ch4_1] in {
// sra is IR node for ashr llvm IR instruction of .bc
def ROL     : shift_rotate_imm32<0x1c, 0x01, "rol", rotl>;
def ROR     : shift_rotate_imm32<0x1d, 0x01, "ror", rotr>;
}
```

```
let Predicates = [Ch4_1] in {
// srl is IR node for lshr llvm IR instruction of .bc
def SHR     : shift_rotate_imm32<0x1f, 0x00, "shr", srl>;
def SRA     : shift_rotate_imm32<0x20, 0x00, "sra", sra>;
def SRAV    : shift_rotate_reg<0x21, 0x00, "srav", sra, CPURegs>;
def SHLV    : shift_rotate_reg<0x22, 0x00, "shlv", shl, CPURegs>;
def SHRV    : shift_rotate_reg<0x23, 0x00, "shrv", srl, CPURegs>;
def ROLV    : shift_rotate_reg<0x24, 0x01, "rolv", rotl, CPURegs>;
```

(continues on next page)

(continued from previous page)

```
def RORV    : shift_rotate_reg<0x25, 0x01, "rorv", rotr, CPURegs>;
}
```

```
let Predicates = [Ch4_1] in {
/// Multiply and Divide Instructions.
def MULT    : Mult32<0x41, "mult", IIImul>;
def MULTu   : Mult32<0x42, "multu", IIImul>;
def SDIV    : Div32<Cpu0DivRem, 0x43, "div", IIIdiv>;
def UDIV    : Div32<Cpu0DivRemU, 0x44, "divu", IIIdiv>;

def MFHI    : MoveFromLOHI<0x46, "mfhi", CPURegs, [HI]>;
def MFLO    : MoveFromLOHI<0x47, "mflo", CPURegs, [LO]>;
def MTHI    : MoveToLOHI<0x48, "mthi", CPURegs, [HI]>;
def MTLO    : MoveToLOHI<0x49, "mtlo", CPURegs, [LO]>;

def MFC0    : MoveFromC0<0x50, "mfc0", CPURegs>;
def MTC0    : MoveToC0<0x51, "mtc0", CPURegs>;

def C0MOVE  : C0Move<0x52, "c0mov">;
}
```

lbdex/chapters/Chapter4_1/Cpu0ISelLowering.h

```
    SDValue PerformDAGCombine(SDNode *N, DAGCombinerInfo &DCI) const override;
```

lbdex/chapters/Chapter4_1/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
  setOperationAction(ISD::SDIV, MVT::i32, Expand);
  setOperationAction(ISD::SREM, MVT::i32, Expand);
  setOperationAction(ISD::UDIV, MVT::i32, Expand);
  setOperationAction(ISD::UREM, MVT::i32, Expand);
```

```
  setTargetDAGCombine(ISD::SDIVREM);
  setTargetDAGCombine(ISD::UDIVREM);
```

```
  ...
}
...
```

```
static SDValue performDivRemCombine(SDNode *N, SelectionDAG& DAG,
                                    TargetLowering::DAGCombinerInfo &DCI,
                                    const Cpu0Subtarget &Subtarget) {
```

(continues on next page)

(continued from previous page)

```
  if (DCI.isBeforeLegalizeOps())
    return SDValue();

  EVT Ty = N->getValueType(0);
  unsigned LO = Cpu0::LO;
  unsigned HI = Cpu0::HI;
  unsigned Opc = N->getOpcode() == ISD::SDIVREM ? Cpu0ISD::DivRem :
                                                  Cpu0ISD::DivRemU;
  SDLoc DL(N);

  SDValue DivRem = DAG.getNode(Opc, DL, MVT::Glue,
                               N->getOperand(0), N->getOperand(1));
  SDValue InChain = DAG.getEntryNode();
  SDValue InGlue = DivRem;

  // insert MFLO
  if (N->hasAnyUseOfValue(0)) {
    SDValue CopyFromLo = DAG.getCopyFromReg(InChain, DL, LO, Ty,
                                            InGlue);
    DAG.ReplaceAllUsesOfValueWith(SDValue(N, 0), CopyFromLo);
    InChain = CopyFromLo.getValue(1);
    InGlue = CopyFromLo.getValue(2);
  }

  // insert MFHI
  if (N->hasAnyUseOfValue(1)) {
    SDValue CopyFromHi = DAG.getCopyFromReg(InChain, DL,
                                            HI, Ty, InGlue);
    DAG.ReplaceAllUsesOfValueWith(SDValue(N, 1), CopyFromHi);
  }

  return SDValue();
}

SDValue Cpu0TargetLowering::PerformDAGCombine(SDNode *N, DAGCombinerInfo &DCI)
  const {
  SelectionDAG &DAG = DCI.DAG;
  unsigned Opc = N->getOpcode();

  switch (Opc) {
  default: break;
  case ISD::SDIVREM:
  case ISD::UDIVREM:
    return performDivRemCombine(N, DAG, DCI, Subtarget);
  }

  return SDValue();
}
```

lbdex/chapters/Chapter4_1/Cpu0RegisterInfo.td

```
let Namespace = "Cpu0" in {
```

```
  // Hi/Lo registers number and name
  def HI    : Cpu0Reg<0, "ac0">, DwarfRegNum<[18]>;
  def LO    : Cpu0Reg<0, "ac0">, DwarfRegNum<[19]>;
```

```
}
...
```

```
// Hi/Lo Registers class
def HILO    : RegisterClass<"Cpu0", [i32], 32, (add HI, LO)>;
```

lbdex/chapters/Chapter4_1/Cpu0Schedule.td

```
def IIHiLo             : InstrItinClass;
def IIImul             : InstrItinClass;
def IIIdiv             : InstrItinClass;
```

```
def Cpu0GenericItineraries : ProcessorItineraries<[ALU, IMULDIV], [], [
```

```
  InstrItinData<IIHiLo             , [InstrStage<1,  [IMULDIV]>]>,
  InstrItinData<IIImul             , [InstrStage<17, [IMULDIV]>]>,
  InstrItinData<IIIdiv             , [InstrStage<38, [IMULDIV]>]>,
```

```
]>;
```

lbdex/chapters/Chapter4_1/Cpu0SEISelDAGToDAG.h

```
  std::pair<SDNode *, SDNode *> selectMULT(SDNode *N, unsigned Opc,
                                           const SDLoc &DL, EVT Ty, bool HasLo,
                                           bool HasHi);
```

lbdex/chapters/Chapter4_1/Cpu0SEISelDAGToDAG.cpp

```
/// Select multiply instructions.
std::pair<SDNode *, SDNode *>
Cpu0SEDAGToDAGISel::selectMULT(SDNode *N, unsigned Opc, const SDLoc &DL, EVT Ty,
                               bool HasLo, bool HasHi) {
  SDNode *Lo = 0, *Hi = 0;
  SDNode *Mul = CurDAG->getMachineNode(Opc, DL, MVT::Glue, N->getOperand(0),
                                       N->getOperand(1));
  SDValue InFlag = SDValue(Mul, 0);

  if (HasLo) {
    Lo = CurDAG->getMachineNode(Cpu0::MFLO, DL,
```

(continues on next page)

(continued from previous page)

```
                                Ty, MVT::Glue, InFlag);
    InFlag = SDValue(Lo, 1);
  }
  if (HasHi)
    Hi = CurDAG->getMachineNode(Cpu0::MFHI, DL,
                                Ty, InFlag);

  return std::make_pair(Lo, Hi);
}
```

```
bool Cpu0SEDAGToDAGISel::trySelect(SDNode *Node) {
  unsigned Opcode = Node->getOpcode();
  SDLoc DL(Node);

  ///
  // Instruction Selection not handled by the auto-generated
  // tablegen selection should be handled here.
  ///

  ///
  // Instruction Selection not handled by the auto-generated
  // tablegen selection should be handled here.
  ///
  EVT NodeTy = Node->getValueType(0);
  unsigned MultOpc;

  switch(Opcode) {
  default: break;
```

```
  case ISD::MULHS:
  case ISD::MULHU: {
    MultOpc = (Opcode == ISD::MULHU ? Cpu0::MULTu : Cpu0::MULT);
    auto LoHi = selectMULT(Node, MultOpc, DL, NodeTy, false, true);
    ReplaceNode(Node, LoHi.second);
    return true;
  }

  case ISD::Constant: {
    const ConstantSDNode *CN = dyn_cast<ConstantSDNode>(Node);
    unsigned Size = CN->getValueSizeInBits(0);

    if (Size == 32)
      break;

    return true;
  }
```

```
  }
  ...
}
```

lbdex/chapters/Chapter4_1/Cpu0SEInstrInfo.h

```
  void copyPhysReg(MachineBasicBlock &MBB, MachineBasicBlock::iterator MI,
                   const DebugLoc &DL, MCRegister DestReg, MCRegister SrcReg,
                   bool KillSrc) const override;
```

lbdex/chapters/Chapter4_1/Cpu0SEInstrInfo.cpp

```
void Cpu0SEInstrInfo::copyPhysReg(MachineBasicBlock &MBB,
                                  MachineBasicBlock::iterator I,
                                  const DebugLoc &DL, MCRegister DestReg,
                                  MCRegister SrcReg, bool KillSrc) const {
  unsigned Opc = 0, ZeroReg = 0;

  if (Cpu0::CPURegsRegClass.contains(DestReg)) { // Copy to CPU Reg.
    if (Cpu0::CPURegsRegClass.contains(SrcReg))
      Opc = Cpu0::ADDu, ZeroReg = Cpu0::ZERO;
    else if (SrcReg == Cpu0::HI)
      Opc = Cpu0::MFHI, SrcReg = 0;
    else if (SrcReg == Cpu0::LO)
      Opc = Cpu0::MFLO, SrcReg = 0;
  }
  else if (Cpu0::CPURegsRegClass.contains(SrcReg)) { // Copy from CPU Reg.
    if (DestReg == Cpu0::HI)
      Opc = Cpu0::MTHI, DestReg = 0;
    else if (DestReg == Cpu0::LO)
      Opc = Cpu0::MTLO, DestReg = 0;
  }

  assert(Opc && "Cannot copy registers");

  MachineInstrBuilder MIB = BuildMI(MBB, I, DL, get(Opc));

  if (DestReg)
    MIB.addReg(DestReg, RegState::Define);

  if (ZeroReg)
    MIB.addReg(ZeroReg);

  if (SrcReg)
    MIB.addReg(SrcReg, getKillRegState(KillSrc));
}
```

4.1.1 +, -, *, <<, and >>

The ADDu, ADD, SUBu, SUB and MUL defined in Chapter4_1/Cpu0InstrInfo.td are for operators **+, -, *.** SHL (defined before) and SHLV are for **<<**. SRA, SRAV, SHR and SHRV are for **>>**.

In RISC CPU, such as Mips, the multiply/divide function unit and add/sub/logic unit are designed from two different hardware circuits, and more, their data path are separate. Cpu0 is same, so these two function units can be executed at same time (instruction level parallelism). Reference[1] for instruction itineraries.

Chapter4_1/ can handle **+, -, *, <<,** and **>>** operators in C language. The corresponding llvm IR instructions are **add, sub, mul, shl, ashr**. The **'ashr'** instruction (arithmetic shift right) returns the first operand shifted to the right a specified number of bits with sign extension. In brief, we call **ashr** is "shift with sign extension fill".

Note: ashr

Example:

<result> = ashr i32 4, 1 ; yields {i32}:result = 2

<result> = ashr i8 -2, 1 ; yields {i8}:result = -1

<result> = ashr i32 1, 32 ; undefined

The semantic of C operator **>>** for negative operand is dependent on implementation. Most compilers translate it into "shift with sign extension fill", and Mips **sra** is this instruction. Following is the Micosoft web site's explanation,

---

Note: >>, Microsoft Specific

The result of a right shift of a signed negative quantity is implementation dependent. Although Microsoft C++ propagates the most-significant bit to fill vacated bit positions, there is no guarantee that other implementations will do likewise.

In addition to **ashr**, the other instruction "shift with zero filled" **lshr** in llvm (Mips implement lshr with instruction **srl**) has the following meaning.

Note: lshr

Example: <result> = lshr i8 -2, 1 ; yields {i8}:result = 0x7FFFFFFF

In llvm, IR node **sra** is defined for ashr IR instruction, and node **srl** is defined for lshr instruction (We don't know why it doesn't use ashr and lshr as the IR node name directly). Summary as the Table: C operator >> implementation.

Table 4.1: C operator >> implementation

Description	Shift with zero filled	Shift with signed extension filled
symbol in .bc	lshr	ashr
symbol in IR node	srl	sra
Mips instruction	srl	sra
Cpu0 instruction	shr	sra
signed example before x >> 1	0xfffffffe i.e. -2	0xfffffffe i.e. -2
signed example after x >> 1	0x7fffffff i.e 2G-1	0xffffffff i.e. -1
unsigned example before x >> 1	0xfffffffe i.e. 4G-2	0xfffffffe i.e. 4G-2
unsigned example after x >> 1	0x7fffffff i.e 2G-1	0xffffffff i.e. 4G-1

[1] http://llvm.org/docs/doxygen/html/structllvm_1_1InstrStage.html

lshr: Logical SHift Right

ashr: Arithmetic SHift right

srl: Shift Right Logically

sra: Shift Right Arithmetically

shr: SHift Right

If we consider the x >> 1 definition is x = x/2 for compiler implementation. Then as you can see from Table: C operator >> implementation, **lshr** will fail on some signed value (such as -2). In the same way, **ashr** will fail on some unsigned value (such as 4G-2). So, in order to satisfy this definition in both signed and unsigned integers of x, we need these two instructions, **lshr** and **ashr**.

Table 4.2: C operator << implementation

Description	Shift with zero filled
symbol in .bc	shl
symbol in IR node	shl
Mips instruction	sll
Cpu0 instruction	shl
signed example before x << 1	0x40000000 i.e. 1G
signed example after x << 1	0x80000000 i.e -2G
unsigned example before x << 1	0x40000000 i.e. 1G
unsigned example after x << 1	0x80000000 i.e 2G

Again, consider the x << 1 definition is x = x*2. From Table: C operator << implementation, we see **lshr** satisfy "unsigned x=1G" but fails on signed x=1G. It's fine since 2G is out of 32 bits signed integer range (-2G ~ 2G-1). For the overflow case, no way to keep the correct result in register. So, any value in register is OK. You can check that **lshr** satisfy x = x*2, for all x << 1 and the x result is not out of range, no matter operand x is signed or unsigned integer[2].

The 'ashr' Instruction" reference here[3], 'lshr' reference here[4].

The srav, shlv and shrv are for two virtual input registers instructions while the sra, ... are for 1 virtual input registers and 1 constant input operands.

Now, let's build Chapter4_1/ and run with input file ch4_math.ll as follows,

lbdex/input/ch4_math.ll

```
; Function Attrs: nounwind
define i32 @_Z9test_mathv() #0 {
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %1 = load i32, i32* %a, align 4
  %2 = load i32, i32* %b, align 4

  %3 = add nsw i32 %1, %2
  %4 = sub nsw i32 %1, %2
  %5 = mul nsw i32 %1, %2
  %6 = shl i32 %1, 2
```

(continues on next page)

[2] https://open4tech.com/logical-vs-arithmetic-shift
[3] http://llvm.org/docs/LangRef.html#ashr-instruction
[4] http://llvm.org/docs/LangRef.html#lshr-instruction

(continued from previous page)

```
  %7 = ashr i32 %1, 2
  %8 = lshr i32 %1, 30
  %9 = shl i32 1, %2
  %10 = ashr i32 128, %2
  %11 = ashr i32 %1, %2

  %12 = add nsw i32 %3, %4
  %13 = add nsw i32 %12, %5
  %14 = add nsw i32 %13, %6
  %15 = add nsw i32 %14, %7
  %16 = add nsw i32 %15, %8
  %17 = add nsw i32 %16, %9
  %18 = add nsw i32 %17, %10
  %19 = add nsw i32 %18, %11

  ret i32 %19
}
```

```
118-165-78-12:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch4_math.ll -o -
  ...
        ld    $2, 0($sp)
        ld    $3, 4($sp)
        subu  $4, $3, $2
        addu  $5, $3, $2
        addu  $4, $5, $4
        mul   $5, $3, $2
        addu  $4, $4, $5
        shl   $5, $3, 2
        addu  $4, $4, $5
        sra   $5, $3, 2
        addu  $4, $4, $5
        addiu $5, $zero, 128
        shrv  $5, $5, $2
        addiu $t9, $zero, 1
        shlv  $t9, $t9, $2
        srav  $2, $3, $2
        shr   $3, $3, 30
        addu  $3, $4, $3
        addu  $3, $3, $t9
        addu  $3, $3, $5
        addu  $2, $3, $2
        addiu $sp, $sp, 8
        ret   $lr
```

Example input ch4_1_math.cpp as the following is the C file which include **+, -, *, <<,** and **>>** operators. It will generate corresponding llvm IR instructions, **add, sub, mul, shl, ashr** by clang as Chapter 3 indicated.

lbdex/input/ch4_1_math.cpp

```
int test_math()
{
  int a = 5;
  int b = 2;
  unsigned int a1 = -5;
  int c, d, e, f, g, h, i;
  int j;
  unsigned int f1, g1, h1, i1;

  c = a + b;       // c = 7
  d = a - b;       // d = 3
  e = a * b;       // e = 10
  f = (a << 2);    // f = 20
  f1 = (a1 << 1); // f1 = 0xfffffff6 = -10
  g = (a >> 2);    // g = 1
  g1 = (a1 >> 30); // g1 = 0x03 = 3
  h = (1 << a);    // h = 0x20 = 32
  h1 = (1 << b);  // h1 = 0x04
  i = (0x80 >> a); // i = 0x04
  i1 = (b >> a);  // i1 = 0x0
  j = (-24 >> 2);  // j = -6

  return (c+d+e+f+int(f1)+g+(int)g1+h+(int)h1+i+(int)i1+j);
// 7+3+10+20-10+1+3+32+4+4+0-6 = 68
}
```

Cpu0 instructions add and sub will trigger overflow exception while addu and subu truncate overflow value directly. Compile ch4_1_addsuboverflow.cpp with `llc -cpu0-enable-overflow=true` will generate add and sub instructions as follows,

lbdex/input/ch4_1_addsuboverflow.cpp

```
#include "debug.h"

int test_add_overflow()
{
  int a = 0x70000000;
  int b = 0x20000000;
  int c = 0;

  c = a + b;

  return 0;
}

int test_sub_overflow()
{
  int a = -0x70000000;
  int b = 0x20000000;
```

(continues on next page)

(continued from previous page)

```
  int c = 0;

  c = a - b;

  return 0;
}
```

```
118-165-78-12:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch4_1_addsuboverflow.cpp -emit-llvm -o ch4_1_addsuboverflow.bc
118-165-78-12:input Jonathan$ llvm-dis ch4_1_addsuboverflow.bc -o -
...
; Function Attrs: nounwind
define i32 @_Z13test_overflowv() #0 {
  ...
  %3 = add nsw i32 %1, %2
  ...
  %6 = sub nsw i32 %4, %5
  ...
}

118-165-78-12:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
-cpu0-enable-overflow=true ch4_1_addsuboverflow.bc -o -
      ...
      add     $3, $4, $3
      ...
      sub     $3, $4, $3
      ...
```

In modern CPU, programmers are used to using truncate overflow instructions for C operators + and -. Anyway, through option -cpu0-enable-overflow=true, programmer get the chance to compile program with overflow exception program. Usually, this option used in debug purpose. Compile with this option can help to identify the bug and fix it early.

4.1.2 Display llvm IR nodes with Graphviz

The previous section, display the DAG translation process in text on terminal by option `llc -debug`. The `llc` also supports the graphic displaying. The section Install other tools on iMac include the download and installation of tool Graphivz. The `llc` graphic displaying with tool Graphviz is introduced in this section. The graphic displaying is more readable by eyes than displaying text in terminal. It's not a must-have, but helps a lot especially when you are tired in tracking the DAG translation process. List the `llc` graphic support options from the sub-section "SelectionDAG Instruction Selection Process" of web "The LLVM Target-Independent Code Generator"[5] as follows,

---

Note: The `llc` Graphviz DAG display options

-view-dag-combine1-dags displays the DAG after being built, before the first optimization pass.

-view-legalize-dags displays the DAG before Legalization.

-view-dag-combine2-dags displays the DAG before the second optimization pass.

-view-isel-dags displays the DAG before the Select phase.

[5] http://llvm.org/docs/CodeGenerator.html#selectiondag-instruction-selection-process

-view-sched-dags displays the DAG before Scheduling.

By tracking `llc -debug`, you can see the steps of DAG translation as follows,

```
Initial selection DAG
Optimized lowered selection DAG
Type-legalized selection DAG
Optimized type-legalized selection DAG
Legalized selection DAG
Optimized legalized selection DAG
Instruction selection
Selected selection DAG
Scheduling
...
```

Let's run `llc` with option -view-dag-combine1-dags, and open the output result with Graphviz as follows,

```
118-165-12-177:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -view-dag-combine1-dags -march=cpu0
-relocation-model=pic -filetype=asm ch4_1_mult.bc -o ch4_1_mult.cpu0.s
Writing '/tmp/llvm_84ibpm/dag.main.dot'...  done.
118-165-12-177:input Jonathan$ Graphviz /tmp/llvm_84ibpm/dag.main.dot
```

It will show the /tmp/llvm_84ibpm/dag.main.dot as Fig. 4.1.

Fig. 4.1 is the stage of "Initial selection DAG". List the other view options and their corresponding stages of DAG translation as follows,

Note: `llc` Graphviz options and the corresponding stages of DAG translation

-view-dag-combine1-dags: Initial selection DAG

-view-legalize-dags: Optimized type-legalized selection DAG

-view-dag-combine2-dags: Legalized selection DAG

-view-isel-dags: Optimized legalized selection DAG

-view-sched-dags: Selected selection DAG

The -view-isel-dags is important and often used by an llvm backend writer because it is the DAGs before instruction selection. In order to writing the pattern match instruction in target description file .td, backend programmer needs knowing what the DAG nodes are for a given C operator.

4.1.3 Operator % and /

The DAG of %

Example input code ch4_1_mult.cpp which contains the C operator **"%"** and it's corresponding llvm IR, as follows,

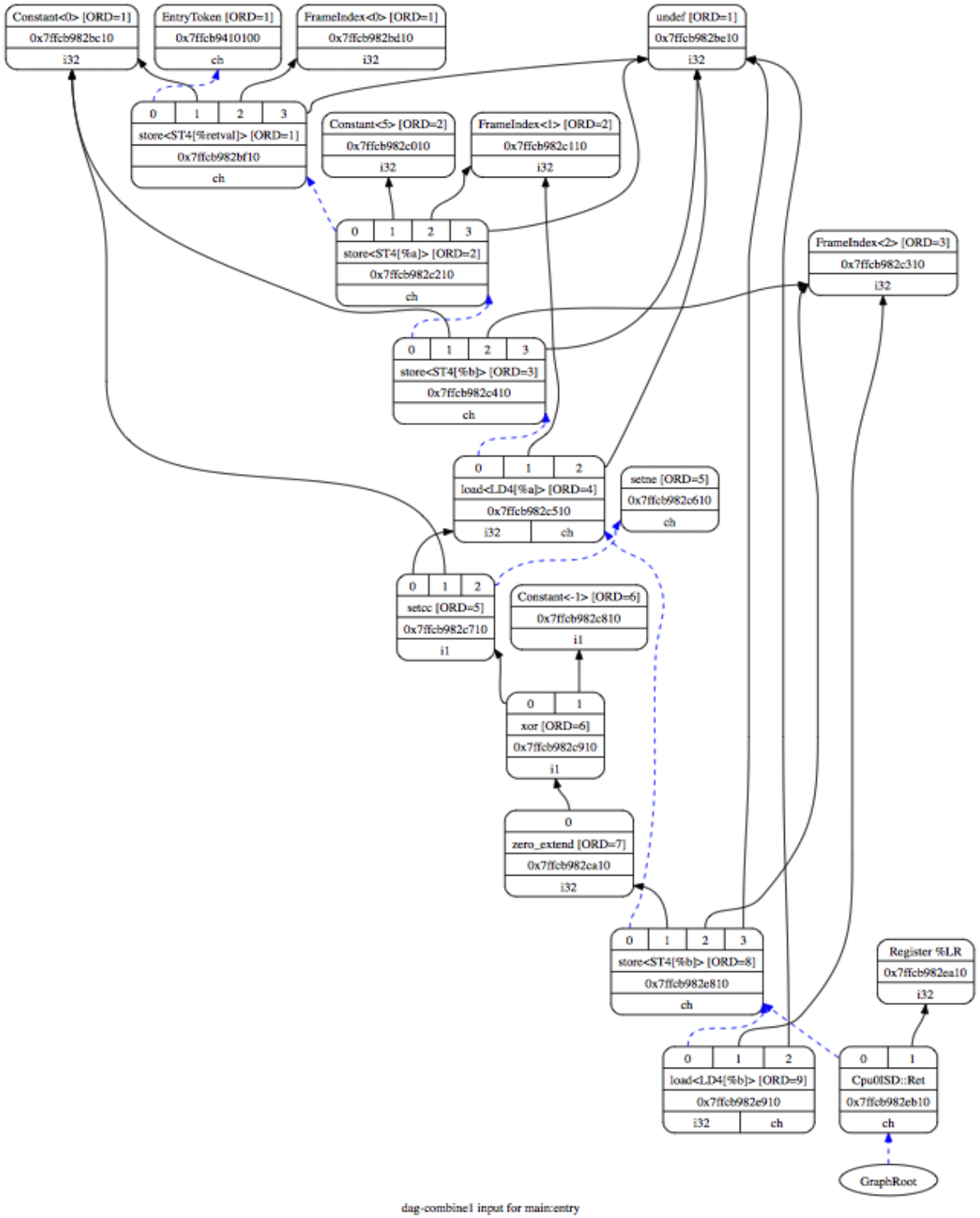


Fig. 4.1: llc option -view-dag-combine1-dags graphic view

lbdex/input/ch4_1_mult.cpp

```
int test_mult()
{
  int b = 11;
//  unsigned int b = 11;

  b = (b+1)%12;

  return b;
}
```

```
...
define i32 @_Z8test_multv() #0 {
  %b = alloca i32, align 4
  store i32 11, i32* %b, align 4
  %1 = load i32* %b, align 4
  %2 = add nsw i32 %1, 1
  %3 = srem i32 %2, 12
  store i32 %3, i32* %b, align 4
  %4 = load i32* %b, align 4
  ret i32 %4
}
```

LLVM **srem** is the IR of corresponding **"%"**, reference here[6]. Copy the reference as follows,

---

Note: 'srem' Instruction

Syntax: **<result> = srem <ty> <op1>, <op2> ; yields {ty}:result**

Overview: The **'srem'** instruction returns the remainder from the signed division of its two operands. This instruction can also take vector versions of the values in which case the elements must be integers.

Arguments: The two arguments to the **'srem'** instruction must be integer or vector of integer values. Both arguments must have identical types.

Semantics: This instruction returns the remainder of a division (where the result is either zero or has the same sign as the dividend, op1), not the modulo operator (where the result is either zero or has the same sign as the divisor, op2) of a value. For more information about the difference, see The Math Forum. For a table of how this is implemented in various languages, please see Wikipedia: modulo operation.

Note that signed integer remainder and unsigned integer remainder are distinct operations; for unsigned integer remainder, use **'urem'**.

Taking the remainder of a division by zero leads to undefined behavior. Overflow also leads to undefined behavior; this is a rare case, but can occur, for example, by taking the remainder of a 32-bit division of -2147483648 by -1. (The remainder doesn't actually overflow, but this rule lets srem be implemented using instructions that return both the result of the division and the remainder.)

Example: <result> = **srem i32 4, %var** ; yields {i32}:result = 4 % %var

---

Run Chapter3_5/ with input file ch4_1_mult.bc via option `llc -view-isel-dags`, will get the following error message and the llvm DAGs of Fig. 4.2 below.

[6] http://llvm.org/docs/LangRef.html#srem-instruction

```
118-165-79-37:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -view-isel-dags -relocation-model=
pic -filetype=asm ch4_1_mult.bc -o -
...
LLVM ERROR: Cannot select: 0x7fa73a02ea10: i32 = mulhs 0x7fa73a02c610,
0x7fa73a02e910 [ID=12]
0x7fa73a02c610: i32 = Constant<12> [ORD=5] [ID=7]
0x7fa73a02e910: i32 = Constant<715827883> [ID=9]
```

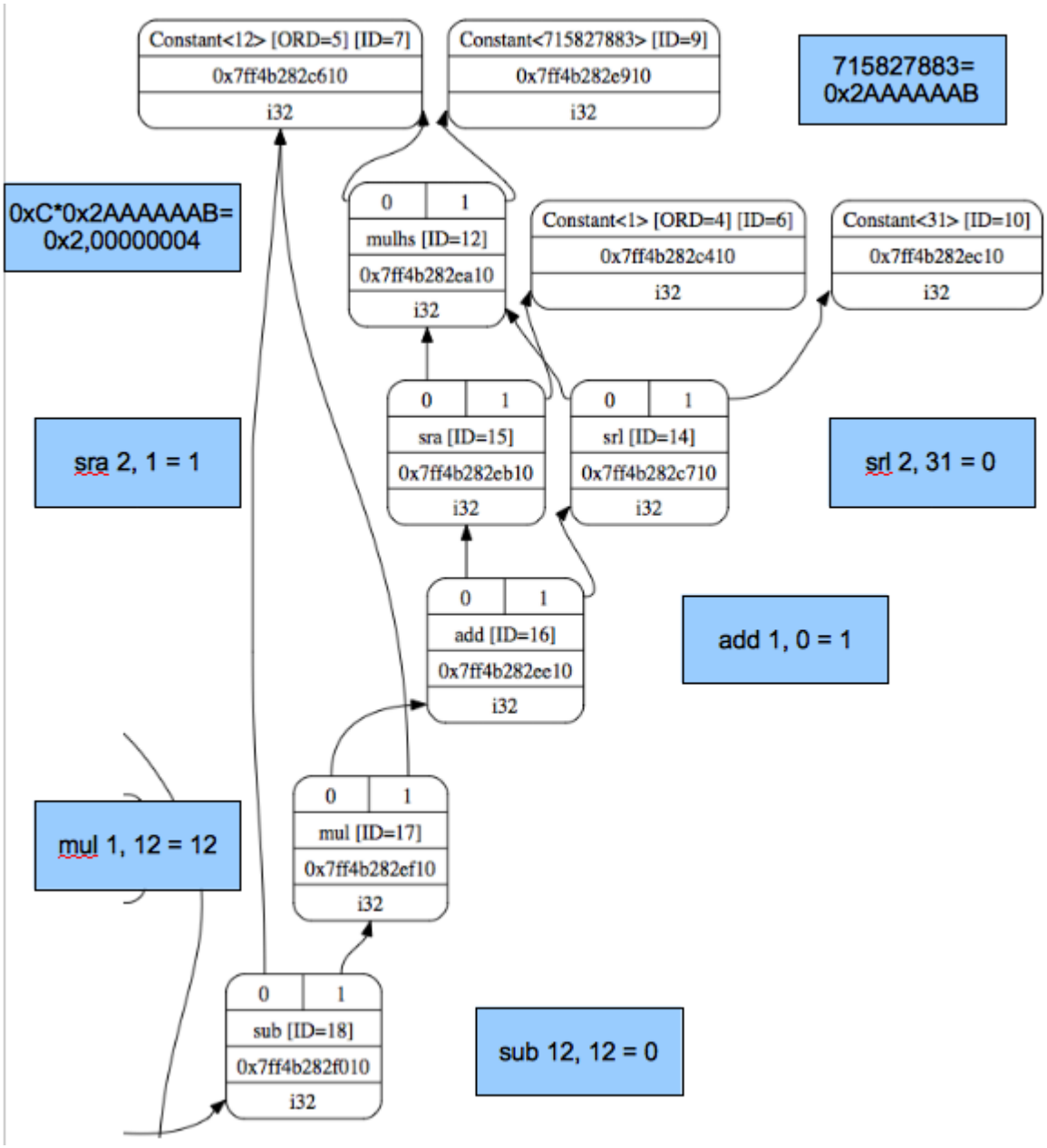


Fig. 4.2: ch4_1_mult.bc DAG

LLVM replaces srem divide operation with multiply operation in DAG optimization because DIV operation costs more in time than MUL. Example code **"int b = 11; b=(b+1)%12;"** is translated into DAGs as Fig. 4.2. The DAGs of gener-

ated result is verified and explained by calculating the value in each node. The 0xC*0x2AAAAAAB=0x2,00000004, (mulhs 0xC, 0x2AAAAAAB) meaning get the Signed mul high word (32bits). Multiply with 2 operands of 1 word size probably generate the 2 word size of result (0x2, 0xAAAAAAAB). The result of high word, in this case is 0x2. The final result (sub 12, 12) is 0 which match the statement (11+1)%12.

Arm solution

To run with ARM solution, change Cpu0InstrInfo.td and Cpu0ISelDAGToDAG.cpp from Chapter4_1/ as follows,

lbdex/chapters/Chapter4_1/Cpu0InstrInfo.td

```
/// Multiply and Divide Instructions.
def SMMUL   : ArithLogicR<0x41, "smmul", mulhs, IIImul, CPURegs, 1>;
def UMMUL   : ArithLogicR<0x42, "ummul", mulhu, IIImul, CPURegs, 1>;
//def MULT    : Mult32<0x41, "mult", IIImul>;
//def MULTu   : Mult32<0x42, "multu", IIImul>;
```

lbdex/chapters/Chapter4_1/Cpu0ISelDAGToDAG.cpp

```
#if 0
/// Select multiply instructions.
std::pair<SDNode*, SDNode*>
Cpu0DAGToDAGISel::SelectMULT(SDNode *N, unsigned Opc, SDLoc DL, EVT Ty,
                             bool HasLo, bool HasHi) {
  SDNode *Lo = 0, *Hi = 0;
  SDNode *Mul = CurDAG->getMachineNode(Opc, DL, MVT::Glue, N->getOperand(0),
                                       N->getOperand(1));
  SDValue InFlag = SDValue(Mul, 0);

  if (HasLo) {
    Lo = CurDAG->getMachineNode(Cpu0::MFLO, DL,
                                Ty, MVT::Glue, InFlag);
    InFlag = SDValue(Lo, 1);
  }
  if (HasHi)
    Hi = CurDAG->getMachineNode(Cpu0::MFHI, DL,
                                Ty, InFlag);

  return std::make_pair(Lo, Hi);
}
#endif

/// Select instructions not customized! Used for
/// expanded, promoted and normal instructions
SDNode* Cpu0DAGToDAGISel::Select(SDNode *Node) {
...
  switch(Opcode) {
  default: break;
#if 0
  case ISD::MULHS:
```

(continues on next page)

(continued from previous page)

```
  case ISD::MULHU: {
    MultOpc = (Opcode == ISD::MULHU ? Cpu0::MULTu : Cpu0::MULT);
    return SelectMULT(Node, MultOpc, DL, NodeTy, false, true).second;
  }
#endif
 ...
}
```

Let's run above changes with ch4_1_mult.cpp as well as `llc -view-sched-dags` option to get Fig. 4.3. Instruction SMMUL will get the high word of multiply result.

The following is the result of run above changes with ch4_1_mult.bc.

```
118-165-66-82:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=pic -filetype=asm
ch4_1_mult.bc -o -
  ...
# BB#0:                                 # %entry
  addiu $sp, $sp, -8
$tmp1:
  .cfi_def_cfa_offset 8
  addiu $2, $zero, 0
  st  $2, 4($fp)
  addiu $2, $zero, 11
  st  $2, 0($fp)
  lui $2, 10922
  ori $3, $2, 43691
  addiu $2, $zero, 12
  smmul $3, $2, $3
  shr $4, $3, 31
  sra $3, $3, 1
  addu  $3, $3, $4
  mul $3, $3, $2
  subu  $2, $2, $3
  st  $2, 0($fp)
  addiu $sp, $sp, 8
  ret $lr
```

The other instruction UMMUL and llvm IR mulhu are unsigned int type for operator %. You can check it by unmark the **"unsigned int b = 11;"** in ch4_1_mult.cpp.

Using SMMUL instruction to get the high word of multiplication result is adopted in ARM.

Mips solution

Mips uses MULT instruction and save the high & low part to registers HI and LO, respectively. After that, uses mfhi/mflo to move register HI/LO to your general purpose registers. ARM SMMUL is fast if you only need the HI part of result (it ignores the LO part of operation). ARM also provides SMULL (signed multiply long) to get the whole 64 bits result. If you need the LO part of result, you can use Cpu0 MUL instruction to get the LO part of result only. Chapter4_1/ is implemented with Mips MULT style. We choose it as the implementation of this book for adding instructions as less as possible. This approach make Cpu0 better both as a tutorial architecture for school teaching purpose material, and an engineer learning materials in compiler design. The MULT, MULTu, MFHI, MFLO, MTHI, MTLO added in Chapter4_1/Cpu0InstrInfo.td; HI, LO registers in Chapter4_1/Cpu0RegisterInfo.td

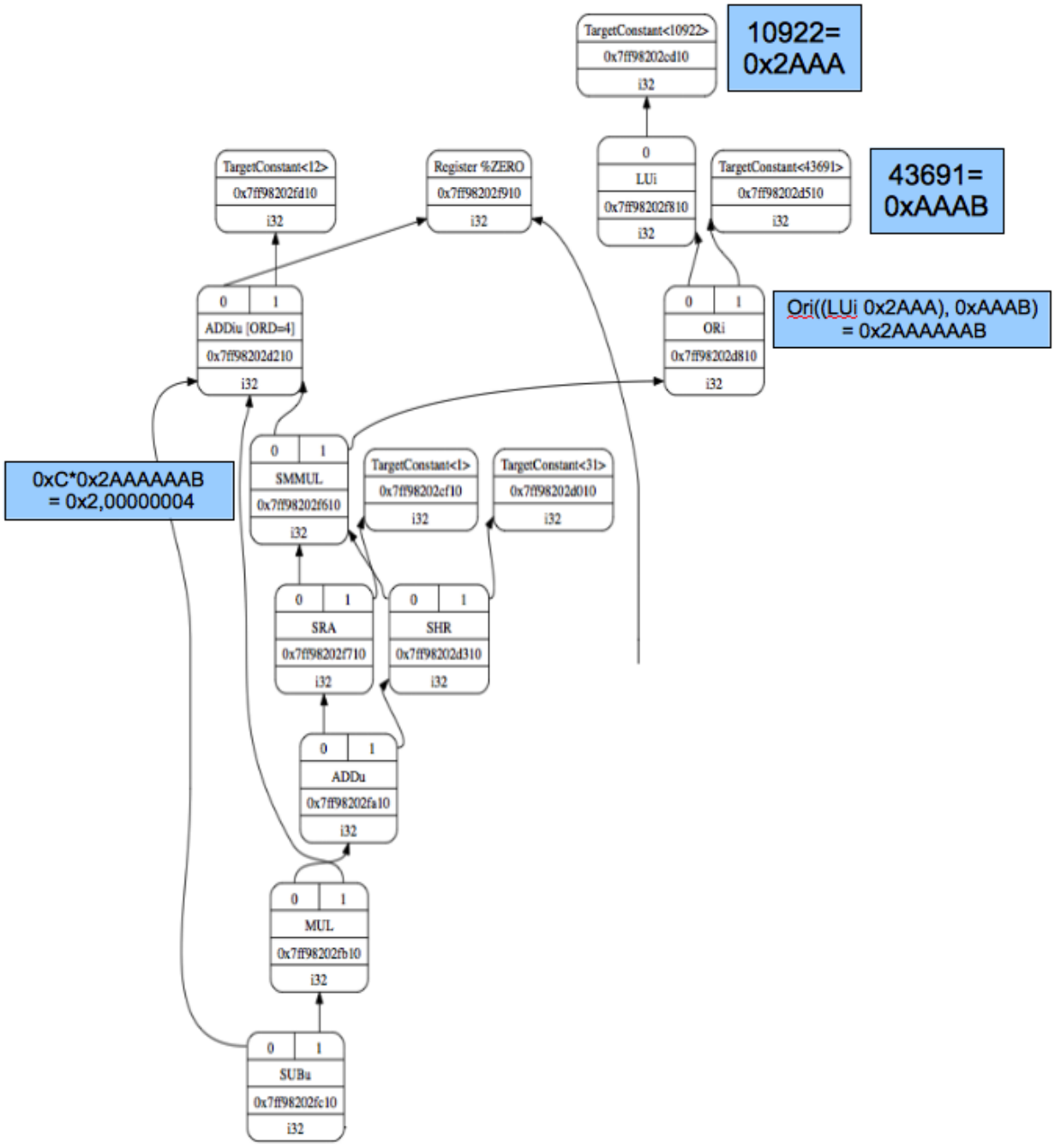


Fig. 4.3: DAG for ch4_1_mult.bc with ARM style SMMUL

and Chapter4_1/MCTargetDesc/ Cpu0BaseInfo.h; IIHiLo, IIImul in Chapter4_1/Cpu0Schedule.td; SelectMULT() in Chapter4_1/Cpu0ISelDAGToDAG.cpp are for Mips style implementation.

The related DAG nodes, mulhs and mulhu, both are used in Chapter4_1/, which come from TargetSelectionDAG.td as follows,

include/llvm/Target/TargetSelectionDAG.td

```
def mulhs    : SDNode<"ISD::MULHS"     , SDTIntBinOp, [SDNPCommutative]>;
def mulhu    : SDNode<"ISD::MULHU"     , SDTIntBinOp, [SDNPCommutative]>;
```

Except the custom type, llvm IR operations of type expand and promote will call Cpu0DAGToDAGISel::Select() during instruction selection of DAG translation. The SelectMULT() which called by Select() return the HI part of multiplication result to HI register for IR operations of mulhs or mulhu. After that, MFHI instruction moves the HI register to Cpu0 field "a" register, $ra. MFHI instruction is FL format and only use Cpu0 field "a" register, we set the $rb and imm16 to 0. Fig. 4.4 and ch4_1_mult.cpu0.s are the results of compile ch4_1_mult.bc.

```
118-165-66-82:input Jonathan$ cat ch4_1_mult.cpu0.s
  ...
# BB#0:
  addiu $sp, $sp, -8
  addiu $2, $zero, 11
  st  $2, 4($sp)
  lui $2, 10922
  ori $3, $2, 43691
  addiu $2, $zero, 12
  mult  $2, $3
  mfhi  $3
  shr $4, $3, 31
  sra $3, $3, 1
  addu  $3, $3, $4
  mul $3, $3, $2
  subu  $2, $2, $3
  st  $2, 4($sp)
  addiu $sp, $sp, 8
  ret $lr
```

Full support %, and /

The sensitive readers may find llvm using **"multiplication"** instead of **"div"** to get the **"%"** result just because our example uses constant as divider, **"(b+1)%12"** in our example. If programmer uses variable as the divider like **"(b+1)%a"**, then: what will happen next? The answer is our code will has error in handling this.

Cpu0 just like Mips uses LO and HI registers to hold the **"quotient"** and **"remainder"**. And uses instructions **"mflo"** and **"mfhi"** to get the result from LO or HI registers furthermore. With this solution, the **"c = a / b"** can be finished by **"div a, b"** and **"mflo c"**; the **"c = a % b"** can be finished by **"div a, b"** and **"mfhi c"**.

To supports operators **"%"** and **"/"**, the following code added in Chapter4_1.

1. SDIV, UDIV and it's reference class, nodes in Cpu0InstrInfo.td.
2. The copyPhysReg() declared and defined in Cpu0InstrInfo.h and Cpu0InstrInfo.cpp.

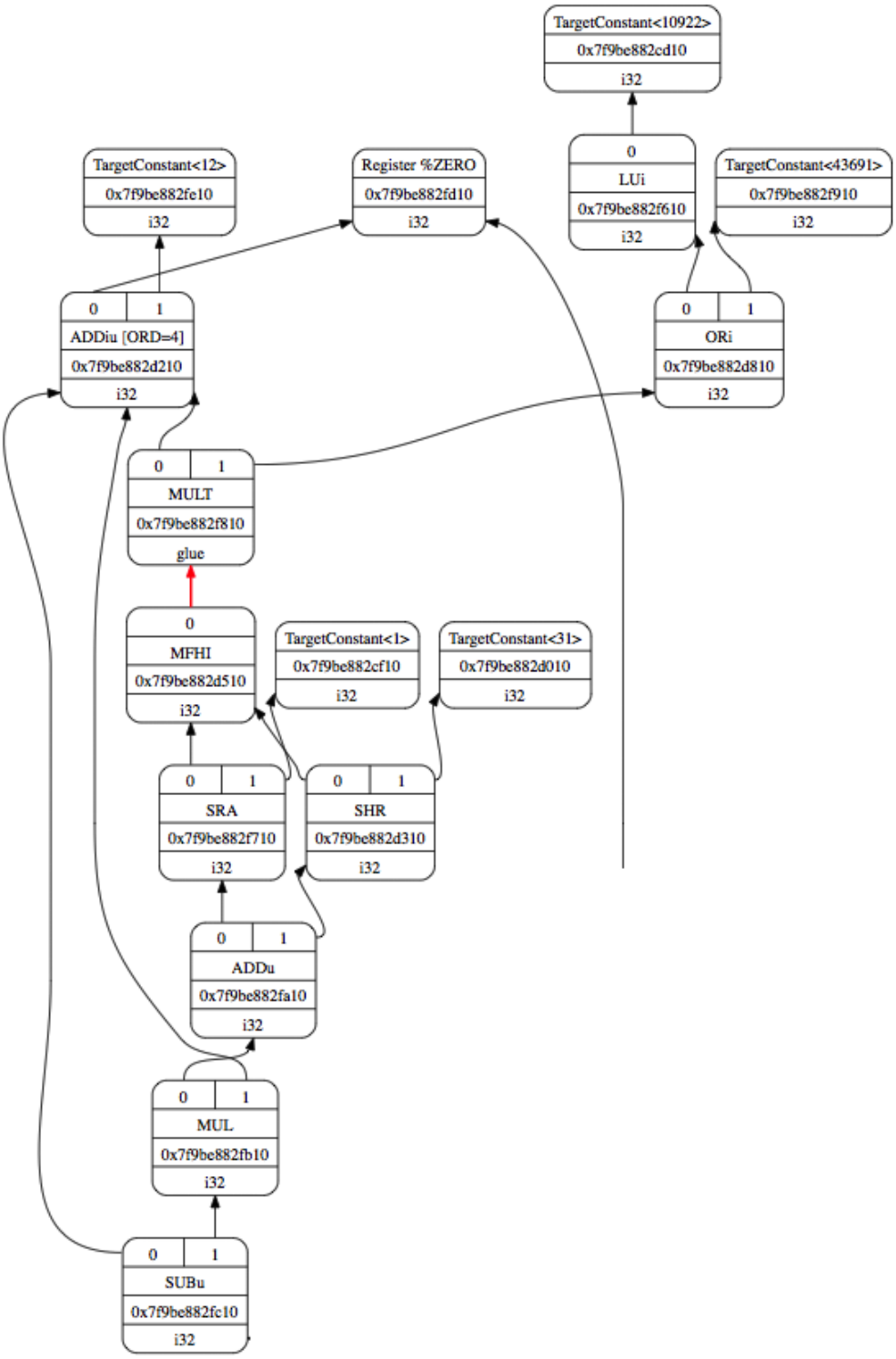


Fig. 4.4: DAG for ch4_1_mult.bc with Mips style MULT

3. The setOperationAction(ISD::SDIV, MVT::i32, Expand), ..., setTargetDAGCombine(ISD::SDIVREM) in constructore of Cpu0ISelLowering.cpp; PerformDivRemCombine() and PerformDAGCombine() in Cpu0ISelLowering.cpp.

The IR instruction **sdiv** stands for signed div while **udiv** stands for unsigned div.

lbdex/input/ch4_1_mult2.cpp

```
int test_mult()
{
  int b = 11;
  int a = 12;

  b = (b+1)%a;

  return b;
}
```

If we run with ch4_1_mult2.cpp, the **"div"** cannot be gotten for operator **"%"**. It still uses **"multiplication"** instead of **"div"** in ch4_1_mult2.cpp because llvm do **"Constant Propagation Optimization"** in this. The ch4_1_mod.cpp can get the **"div"** for **"%"** result since it makes llvm **"Constant Propagation Optimization"** useless in it.

lbdex/input/ch4_1_mod.cpp

```
int test_mod()
{
  int b = 11;
  volatile int a = 12;

  b = (b+1)%a;

  return b;
}
```

```
118-165-77-79:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch4_1_mod.cpp -emit-llvm -o ch4_1_mod.bc
118-165-77-79:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=pic -filetype=asm
ch4_1_mod.bc -o -
...
div $zero, $3, $2
mflo  $2
...
```

To explains how to work with **"div"**, let's run ch4_1_mod.cpp with debug option as follows,

```
118-165-83-58:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch4_1_mod.cpp -I/Applications/Xcode.app/Contents/Developer/Platforms/
MacOSX.platform/Developer/SDKs/MacOSX10.8.sdk/usr/include/ -emit-llvm -o
ch4_1_mod.bc
118-165-83-58:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm -debug
```

(continues on next page)

(continued from previous page)

```
ch4_1_mod.bc -o -
...
=== _Z8test_modi
Initial selection DAG: BB#0 '_Z8test_mod2i:'
SelectionDAG has 21 nodes:
  ...
    0x2447448: <multiple use>
        0x24470d0: <multiple use>
        0x24471f8: i32 = Constant<1>

      0x2447320: i32 = add 0x24470d0, 0x24471f8 [ORD=7]

      0x2447448: <multiple use>
    0x2447570: i32 = srem 0x2447320, 0x2447448 [ORD=9]

    0x24468b8: <multiple use>
    0x2446b08: <multiple use>
  0x2448fc0: ch = store 0x2447448:1, 0x2447570, 0x24468b8, ...

  0x2449210: i32 = Register %V0

    0x2448fc0: <multiple use>
    0x2449210: <multiple use>
      0x2448fc0: <multiple use>
      0x24468b8: <multiple use>
      0x2446b08: <multiple use>
    0x24490e8: i32,ch = load 0x2448fc0, 0x24468b8, 0x2446b08<LD4[%b]> [ORD=11]

  0x2449338: ch,glue = CopyToReg 0x2448fc0, 0x2449210, 0x24490e8 [ORD=12]

    0x2449338: <multiple use>
    0x2449210: <multiple use>
    0x2449338: <multiple use>
  0x2449460: ch = Cpu0ISD::Ret 0x2449338, 0x2449210, 0x2449338:1 [ORD=12]

Replacing.1 0x24490e8: i32,ch = load 0x2448fc0, 0x24468b8, ...

With: 0x2447570: i32 = srem 0x2447320, 0x2447448 [ORD=9]
 and 1 other values
...

Optimized lowered selection DAG: BB#0 '_Z8test_mod2i:'
...
  0x2447570: i32 = srem 0x2447320, 0x2447448 [ORD=9]
...

Type-legalized selection DAG: BB#0 '_Z8test_mod2i:'
SelectionDAG has 16 nodes:
  ...
  0x7fed6882d610: i32,ch = load 0x7fed6882d210, 0x7fed6882cd10,
  0x7fed6882cb10<LD4[%1]> [ORD=5] [ID=-3]
```

(continues on next page)

(continued from previous page)

```
    0x7fed6882d810: i32 = Constant<12> [ID=-3]

    0x7fed6882d610: <multiple use>
  0x7fed6882d710: i32 = srem 0x7fed6882d810, 0x7fed6882d610 [ORD=6] [ID=-3]
  ...

Legalized selection DAG: BB#0 '_Z8test_mod2i:'
  ...
    ... i32 = srem 0x2447320, 0x2447448 [ORD=9] [ID=-3]
  ...
 ... replacing: ...: i32 = srem 0x2447320, 0x2447448 [ORD=9] [ID=13]
     with:      ...: i32,i32 = sdivrem 0x2447320, 0x2447448 [ORD=9]

Optimized legalized selection DAG: BB#0 '_Z8test_mod2i:'
SelectionDAG has 18 nodes:
  ...
    0x2449588: i32 = Register %HI

        0x24470d0: <multiple use>
        0x24471f8: i32 = Constant<1> [ID=6]

      0x2447320: i32 = add 0x24470d0, 0x24471f8 [ORD=7] [ID=12]

      0x2447448: <multiple use>
    0x24490e8: glue = Cpu0ISD::DivRem 0x2447320, 0x2447448 [ORD=9]

  0x24496b0: i32,ch,glue = CopyFromReg 0x240d480, 0x2449588, 0x24490e8 [ORD=9]

    0x2449338: <multiple use>
    0x2449210: <multiple use>
    0x2449338: <multiple use>
  0x2449460: ch = Cpu0ISD::Ret 0x2449338, 0x2449210, ...
  ...

===== Instruction selection begins: BB#0 ''
...
Selecting: 0x24490e8: glue = Cpu0ISD::DivRem 0x2447320, 0x2447448 [ORD=9] [ID=14]

ISEL: Starting pattern match on root node: 0x24490e8: glue = Cpu0ISD::DivRem
0x2447320, 0x2447448 [ORD=9] [ID=14]

  Initial Opcode index to 4044
  Morphed node: 0x24490e8: i32,glue = SDIV 0x2447320, 0x2447448 [ORD=9]

ISEL: Match complete!
=> 0x24490e8: i32,glue = SDIV 0x2447320, 0x2447448 [ORD=9]
...
```

Summary above DAGs translation messages into 4 steps:

1. Reduce DAG nodes in stage "Optimized lowered selection DAG" (Replacing … displayed before "Optimized lowered selection DAG:"). Since SSA form has some redundant nodes for store and load, they can be removed.
2. Change DAG srem to sdivrem in stage "Legalized selection DAG".

3. Change DAG sdivrem to Cpu0ISD::DivRem and in stage "Optimized legalized selection DAG".
4. Add DAG "i32 = Register %HI" and "CopyFromReg ..." in stage "Optimized legalized selection DAG".

Summary as Table: Stages for C operator % and Table: Functions handle the DAG translation and pattern match for C operator %.

Table 4.3: Stages for C operator %

Stage	IR/DAG/instruction
.bc	srem
Legalized selection DAG	sdivrem
Optimized legalized selection DAG	Cpu0ISD::DivRem, CopyFromReg xx, Hi, Cpu0ISD::DivRem
pattern match	div, mfhi

Table 4.4: Functions handle the DAG translation and pattern match for C operator %

Translation	Do by
srem => sdivrem	setOperationAction(ISD::SREM, MVT::i32, Expand);
sdivrem => Cpu0ISD::DivRem	setTargetDAGCombine(ISD::SDIVREM);
sdivrem => CopyFromReg xx, Hi, xx	PerformDivRemCombine();
Cpu0ISD::DivRem => div	SDIV (Cpu0InstrInfo.td)
CopyFromReg xx, Hi, xx => mfhi	MFLO (Cpu0InstrInfo.td)

Step 2 as above, is triggered by code "setOperationAction(ISD::SREM, MVT::i32, Expand);" in Cpu0ISelLowering.cpp. About **Expand** please ref.[7] and[8]. Step 3 is triggered by code "setTargetDAGCombine(ISD::SDIVREM);" in Cpu0ISelLowering.cpp. Step 4 is did by PerformDivRemCombine() which called by performDAGCombine(). Since the **%** corresponding **srem** makes the "N->hasAnyUseOfValue(1)" to true in PerformDivRemCombine(), it creates DAG of "CopyFromReg". When using **"/"** in C, it will make "N->hasAnyUseOfValue(0)" to ture. For sdivrem, **sdiv** makes "N->hasAnyUseOfValue(0)" true while **srem** makes "N->hasAnyUseOfValue(1)" ture.

Above steps will change the DAGs when `llc` is running. After that, the pattern match defined in Chapter4_1/Cpu0InstrInfo.td will translate **Cpu0ISD::DivRem** into **div**; and **"CopyFromReg xxDAG, Register %H, Cpu0ISD::DivRem"** to **mfhi**.

The ch4_1_div.cpp is for **/** div operator test.

4.1.4 Rotate instructions

Chapter4_1 include the rotate operations translation. The instructions "rol", "ror", "rolv" and "rorv" defined in Cpu0InstrInfo.td handle the translation. Compile ch4_1_rotate.cpp will get Cpu0 "rol" instruction.

[7] http://llvm.org/docs/WritingAnLLVMBackend.html#expand
[8] http://llvm.org/docs/CodeGenerator.html#selectiondag-legalizetypes-phase

lbdex/input/ch4_1_rotate.cpp

```
//#define TEST_ROXV

int test_rotate_left()
{
  unsigned int a = 8;
  int result = ((a << 30) | (a >> 2));

  return result;
}

#ifdef TEST_ROXV

int test_rotate_left1()
{
  volatile unsigned int a = 4;
  volatile int n = 30;
  int result = ((a << n) | (a >> (32 - n)));

  return result;
}

int test_rotate_right()
{
  volatile unsigned int a = 1;
  volatile int n = 30;
  int result = ((a >> n) | (a << (32 - n)));

  return result;
}

#endif
```

```
114-43-200-122:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch4_1_rotate.cpp -emit-llvm -o ch4_1_rotate.bc
114-43-200-122:input Jonathan$ llvm-dis ch4_1_rotate.bc -o -
```

```
define i32 @_Z16test_rotate_leftv() #0 {
  %a = alloca i32, align 4
  %result = alloca i32, align 4
  store i32 8, i32* %a, align 4
  %1 = load i32* %a, align 4
  %2 = shl i32 %1, 30
  %3 = load i32* %a, align 4
  %4 = ashr i32 %3, 2
  %5 = or i32 %2, %4
  store i32 %5, i32* %result, align 4
  %6 = load i32* %result, align 4
  ret i32 %6
}
```

```
114-43-200-122:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch4_1_rotate.bc -o -
  ...
  rol $2, $2, 30
  ...
```

Instructions "rolv" and "rorv" cannot be tested at this moment, they need logic "or" implementation which supported at next section. Like the previous subsection mentioned at this chapter, some IRs in function @_Z16test_rotate_leftv() will be combined into one one IR **rotl** during DAGs translation.

4.2 Logic

Chapter4_2 supports logic operators **&, |, ^, !, ==, !=, <, <=, > and >=**. They are trivial and easy. Listing the added code with comments and table for these operators IR, DAG and instructions as below. Please check them with the run result of bc and asm instructions for ch4_2_logic.cpp as below.

lbdex/chapters/Chapter4_2/Cpu0InstrInfo.td

```
let Predicates = [Ch4_2] in {
class CmpInstr<bits<8> op, string instr_asm,
               InstrItinClass itin, RegisterClass RC, RegisterClass RD,
               bit isComm = 0>:
  FA<op, (outs RD:$ra), (ins RC:$rb, RC:$rc),
     !strconcat(instr_asm, "\t$ra, $rb, $rc"), [], itin> {
  let shamt = 0;
  let isCommutable = isComm;
  let Predicates = [HasCmp];
}
}
```

```
//  Logical
class LogicNOR<bits<8> op, string instr_asm, RegisterClass RC>:
  FA<op, (outs RC:$ra), (ins RC:$rb, RC:$rc),
     !strconcat(instr_asm, "\t$ra, $rb, $rc"),
     [(set RC:$ra, (not (or RC:$rb, RC:$rc)))], IIAlu> {
  let shamt = 0;
  let isCommutable = 1;
}
```

```
// SetCC
let Predicates = [Ch4_2] in {
class SetCC_R<bits<8> op, string instr_asm, PatFrag cond_op,
              RegisterClass RC>:
  FA<op, (outs GPROut:$ra), (ins RC:$rb, RC:$rc),
     !strconcat(instr_asm, "\t$ra, $rb, $rc"),
     [(set GPROut:$ra, (cond_op RC:$rb, RC:$rc))],
     IIAlu>, Requires<[HasSlt]> {
  let shamt = 0;
}
```

(continues on next page)

(continued from previous page)

```
class SetCC_I<bits<8> op, string instr_asm, PatFrag cond_op, Operand Od,
              PatLeaf imm_type, RegisterClass RC>:
  FL<op, (outs GPROut:$ra), (ins RC:$rb, Od:$imm16),
     !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
     [(set GPROut:$ra, (cond_op RC:$rb, imm_type:$imm16))],
     IIAlu>, Requires<[HasSlt]> {
}
}
```

```
let Predicates = [Ch4_2] in {
def ANDi     : ArithLogicI<0x0c, "andi", and, uimm16, immZExt16, CPURegs>;
}
```

```
let Predicates = [Ch4_2] in {
def XORi     : ArithLogicI<0x0e, "xori", xor, uimm16, immZExt16, CPURegs>;
}
```

```
let Predicates = [Ch4_2] in {
let Predicates = [HasCmp] in {
def CMP      : CmpInstr<0x2A, "cmp", IIAlu, CPURegs, SR, 0>;
def CMPu     : CmpInstr<0x2B, "cmpu", IIAlu, CPURegs, SR, 0>;
}
}
```

```
let Predicates = [Ch4_2] in {
def AND      : ArithLogicR<0x18, "and", and, IIAlu, CPURegs, 1>;
def OR       : ArithLogicR<0x19, "or", or, IIAlu, CPURegs, 1>;
def XOR      : ArithLogicR<0x1a, "xor", xor, IIAlu, CPURegs, 1>;
def NOR      : LogicNOR<0x1b, "nor", CPURegs>;
}
```

```
let Predicates = [Ch4_2] in {
let Predicates = [HasSlt] in {
def SLTi     : SetCC_I<0x26, "slti", setlt, simm16, immSExt16, CPURegs>;
def SLTiu    : SetCC_I<0x27, "sltiu", setult, simm16, immSExt16, CPURegs>;
def SLT      : SetCC_R<0x28, "slt", setlt, CPURegs>;
def SLTu     : SetCC_R<0x29, "sltu", setult, CPURegs>;
}
}
```

```
let Predicates = [Ch4_2] in {
def : Pat<(not CPURegs:$in),
// 1's complement of in, ex. not(0xf000000f) == 0x0ffffff0
          (NOR CPURegs:$in, ZERO)>;
}
```

```
// setcc patterns

let Predicates = [Ch4_2] in {
```

(continues on next page)

(continued from previous page)

```
// setcc for cmp instruction
multiclass SeteqPatsCmp<RegisterClass RC> {
// a == b
  def : Pat<(seteq RC:$lhs, RC:$rhs),
            (SHR (ANDi (CMP RC:$lhs, RC:$rhs), 2), 1)>;
// a != b
  def : Pat<(setne RC:$lhs, RC:$rhs),
            (XORi (SHR (ANDi (CMP RC:$lhs, RC:$rhs), 2), 1), 1)>;
}

// a < b
multiclass SetltPatsCmp<RegisterClass RC> {
  def : Pat<(setlt RC:$lhs, RC:$rhs),
            (ANDi (CMP RC:$lhs, RC:$rhs), 1)>;
// if cpu0  `define N    `SW[31]  instead of `SW[0] // Negative flag, then need
// 2 more instructions as follows,
//          (XORi (ANDi (SHR (CMP RC:$lhs, RC:$rhs), (LUi 0x8000), 31), 1), 1)>;
  def : Pat<(setult RC:$lhs, RC:$rhs),
            (ANDi (CMPu RC:$lhs, RC:$rhs), 1)>;
}

// a <= b
multiclass SetlePatsCmp<RegisterClass RC> {
  def : Pat<(setle RC:$lhs, RC:$rhs),
// a <= b is equal to (XORi (b < a), 1)
            (XORi (ANDi (CMP RC:$rhs, RC:$lhs), 1), 1)>;
  def : Pat<(setule RC:$lhs, RC:$rhs),
            (XORi (ANDi (CMP RC:$rhs, RC:$lhs), 1), 1)>;
}

// a > b
multiclass SetgtPatsCmp<RegisterClass RC> {
  def : Pat<(setgt RC:$lhs, RC:$rhs),
// a > b is equal to b < a is equal to setlt(b, a)
            (ANDi (CMP RC:$rhs, RC:$lhs), 1)>;
  def : Pat<(setugt RC:$lhs, RC:$rhs),
            (ANDi (CMPu RC:$rhs, RC:$lhs), 1)>;
}

// a >= b
multiclass SetgePatsCmp<RegisterClass RC> {
  def : Pat<(setge RC:$lhs, RC:$rhs),
// a >= b is equal to b <= a
            (XORi (ANDi (CMP RC:$lhs, RC:$rhs), 1), 1)>;
  def : Pat<(setuge RC:$lhs, RC:$rhs),
            (XORi (ANDi (CMPu RC:$lhs, RC:$rhs), 1), 1)>;
}

// setcc for slt instruction
multiclass SeteqPatsSlt<RegisterClass RC, Instruction SLTiuOp, Instruction XOROp,
                        Instruction SLTuOp, Register ZEROReg> {
// a == b
```

(continues on next page)

(continued from previous page)

```
  def : Pat<(seteq RC:$lhs, RC:$rhs),
                (SLTiuOp (XOROp RC:$lhs, RC:$rhs), 1)>;
// a != b
  def : Pat<(setne RC:$lhs, RC:$rhs),
                (SLTuOp ZEROReg, (XOROp RC:$lhs, RC:$rhs))>;
}

// a <= b
multiclass SetlePatsSlt<RegisterClass RC, Instruction SLTOp, Instruction SLTuOp> {
  def : Pat<(setle RC:$lhs, RC:$rhs),
// a <= b is equal to (XORi (b < a), 1)
                (XORi (SLTOp RC:$rhs, RC:$lhs), 1)>;
  def : Pat<(setule RC:$lhs, RC:$rhs),
                (XORi (SLTuOp RC:$rhs, RC:$lhs), 1)>;
}

// a > b
multiclass SetgtPatsSlt<RegisterClass RC, Instruction SLTOp, Instruction SLTuOp> {
  def : Pat<(setgt RC:$lhs, RC:$rhs),
// a > b is equal to b < a is equal to setlt(b, a)
                (SLTOp RC:$rhs, RC:$lhs)>;
  def : Pat<(setugt RC:$lhs, RC:$rhs),
                (SLTuOp RC:$rhs, RC:$lhs)>;
}

// a >= b
multiclass SetgePatsSlt<RegisterClass RC, Instruction SLTOp, Instruction SLTuOp> {
  def : Pat<(setge RC:$lhs, RC:$rhs),
// a >= b is equal to b <= a
                (XORi (SLTOp RC:$lhs, RC:$rhs), 1)>;
  def : Pat<(setuge RC:$lhs, RC:$rhs),
                (XORi (SLTuOp RC:$lhs, RC:$rhs), 1)>;
}

multiclass SetgeImmPatsSlt<RegisterClass RC, Instruction SLTiOp,
                           Instruction SLTiuOp> {
  def : Pat<(setge RC:$lhs, immSExt16:$rhs),
                (XORi (SLTiOp RC:$lhs, immSExt16:$rhs), 1)>;
  def : Pat<(setuge RC:$lhs, immSExt16:$rhs),
                (XORi (SLTiuOp RC:$lhs, immSExt16:$rhs), 1)>;
}

let Predicates = [HasSlt] in {
defm : SeteqPatsSlt<CPURegs, SLTiu, XOR, SLTu, ZERO>;
defm : SetlePatsSlt<CPURegs, SLT, SLTu>;
defm : SetgtPatsSlt<CPURegs, SLT, SLTu>;
defm : SetgePatsSlt<CPURegs, SLT, SLTu>;
defm : SetgeImmPatsSlt<CPURegs, SLTi, SLTiu>;
}

let Predicates = [HasCmp] in {
defm : SeteqPatsCmp<CPURegs>;
```

(continues on next page)

(continued from previous page)

```
defm : SetltPatsCmp<CPURegs>;
defm : SetlePatsCmp<CPURegs>;
defm : SetgtPatsCmp<CPURegs>;
defm : SetgePatsCmp<CPURegs>;
}
} // let Predicates = [Ch4_2]
```

lbdex/chapters/Chapter4_2/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
  // Cpu0 doesn't have sext_inreg, replace them with shl/sra.
  setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::i1 , Expand);
  setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::i8 , Expand);
  setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::i16 , Expand);
  setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::i32 , Expand);
  setOperationAction(ISD::SIGN_EXTEND_INREG, MVT::Other , Expand);
```

```
  ...
}
```

lbdex/input/ch4_2_logic.cpp

```
int test_andorxornotcomplement()
{
  int a = 5;
  int b = 3;
  int c = 0, d = 0, e = 0, f = 0, g = 0;

  c = (a & b);  // c = 1
  d = (a | b);  // d = 7
  e = (a ^ b);  // e = 6
  b = !a;       // b = 0
  g = ~f;       // 1's complement, ~0=(-1)=0xffffffff

  return (c+d+e+b+g); // 13
}

int test_setxx()
{
  int a = 5;
  int b = 3;
  int c, d, e, f, g, h;

  c = (a == b); // seq, c = 0
```

(continues on next page)

(continued from previous page)

```
  d = (a != b); // sne, d = 1
  e = (a < b);  // slt, e = 0
  f = (a <= b); // sle, f = 0
  g = (a > b);  // sgt, g = 1
  h = (a >= b); // sge, g = 1

  return (c+d+e+f+g+h); // 3
}
```

```
114-43-204-152:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch4_2_logic.cpp -emit-llvm -o ch4_2_logic.bc
114-43-204-152:input Jonathan$ llvm-dis ch4_2_logic.bc -o -
...
; Function Attrs: nounwind uwtable
define i32 @_Z16test_andorxornotv() #0 {
entry:
  ...
  %and = and i32 %0, %1
  ...
  %or = or i32 %2, %3
  ...
  %xor = xor i32 %4, %5
  ...
  %tobool = icmp ne i32 %6, 0
  %lnot = xor i1 %tobool, true
  %conv = zext i1 %lnot to i32
  ...
}

; Function Attrs: nounwind uwtable
define i32 @_Z10test_setxxv() #0 {
entry:
  ...
  %cmp = icmp eq i32 %0, %1
  %conv = zext i1 %cmp to i32
  store i32 %conv, i32* %c, align 4
  ...
  %cmp1 = icmp ne i32 %2, %3
  %conv2 = zext i1 %cmp1 to i32
  store i32 %conv2, i32* %d, align 4
  ...
  %cmp3 = icmp slt i32 %4, %5
  %conv4 = zext i1 %cmp3 to i32
  store i32 %conv4, i32* %e, align 4
  ...
  %cmp5 = icmp sle i32 %6, %7
  %conv6 = zext i1 %cmp5 to i32
  store i32 %conv6, i32* %f, align 4
  ...
  %cmp7 = icmp sgt i32 %8, %9
  %conv8 = zext i1 %cmp7 to i32
```

(continues on next page)

(continued from previous page)

```
  store i32 %conv8, i32* %g, align 4
  ...
  %cmp9 = icmp sge i32 %10, %11
  %conv10 = zext i1 %cmp9 to i32
  store i32 %conv10, i32* %h, align 4
  ...
}

114-43-204-152:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -filetype=asm
ch4_2_logic.bc -o -

  .globl  _Z16test_andorxornotv
  ...
  and $3, $4, $3
  ...
  or  $3, $4, $3
  ...
  xor $3, $4, $3
  ...
  cmp $sw, $3, $2
  andi  $2, $sw, 2
  shr $2, $2, 1
  ...

  .globl  _Z10test_setxxv
  ...
  cmp $sw, $3, $2
  andi  $2, $sw, 2
  shr $2, $2, 1
  ...
  cmp $sw, $3, $2
  andi  $2, $sw, 2
  shr $2, $2, 1
  xori  $2, $2, 1
  ...
  cmp $sw, $3, $2
  andi  $2, $sw, 1
  ...
  cmp $sw, $3, $2
  andi  $2, $sw, 1
  xori  $2, $2, 1
  ...
  cmp $sw, $3, $2
  andi  $2, $sw, 1
  ...
  cmp $sw, $3, $2
  andi  $2, $sw, 1
  xori  $2, $2, 1
  ...

114-43-204-152:input Jonathan$ /Users/Jonathan/llvm/test/build/
```

(continues on next page)

(continued from previous page)

```
bin/llc -march=cpu0 -mcpu=cpu032II -relocation-model=pic -filetype=asm
ch4_2_logic.bc -o -
  ...
      sltiu   $2, $2, 1
      andi    $2, $2, 1
      ...
```

Table 4.5: Logic operators for cpu032I

C	.bc	Optimized legalized selection DAG	cpu032I
&, &&	and	and	and
|, ||	or	or	or
^	xor	xor	xor
!	• %tobool = icmp ne i32 %6, 0 • %lnot = xor i1 %tobool, true • %conv = zext i1 %lnot to i32	• %lnot = (setcc %tobool, 0, seteq) • %conv = (and %lnot, 1) •	• xor $3, $4, $3
==	• %cmp = icmp eq i32 %0, %1 • %conv = zext i1 %cmp to i32	• %cmp = (setcc %0, %1, seteq) • and %cmp, 1	• cmp $sw, $3, $2 • andi $2, $sw, 2 • shr $2, $2, 1 • andi $2, $2, 1
!=	• %cmp = icmp ne i32 %0, %1 • %conv = zext i1 %cmp to i32	• %cmp = (setcc %0, %1, setne) • and %cmp, 1	• cmp $sw, $3, $2 • andi $2, $sw, 2 • shr $2, $2, 1 • andi $2, $2, 1
<	• %cmp = icmp lt i32 %0, %1 • %conv = zext i1 %cmp to i32	• (setcc %0, %1, setlt) • and %cmp, 1	• cmp $sw, $3, $2 • andi $2, $sw, 2 • andi $2, $2, 1 • andi $2, $2, 1
<=	• %cmp = icmp le i32 %0, %1 • %conv = zext i1 %cmp to i32	• (setcc %0, %1, setle) • and %cmp, 1	• cmp $sw, $2, $3 • andi $2, $sw, 1 • xori $2, $2, 1 • andi $2, $2, 1
>	• %cmp = icmp gt i32 %0, %1 • %conv = zext i1 %cmp to i32	• (setcc %0, %1, setgt) • and %cmp, 1	• cmp $sw, $2, $3 • andi $2, $sw, 2 • andi $2, $2, 1
>=	• %cmp = icmp le i32 %0, %1 • %conv = zext i1 %cmp to i32	• (setcc %0, %1, setle) • and %cmp, 1	• cmp $sw, $3, $2 • andi $2, $sw, 1 • xori $2, $2, 1 • andi $2, $2, 1

Table 4.6: Logic operators for cpu032II

C	.bc	Optimized legalized selection DAG	cpu032II
&, &&	and	and	and
|, ||	or	or	or
^	xor	xor	xor
!	• %tobool = icmp ne i32 %6, 0 • %lnot = xor i1 %tobool, true • %conv = zext i1 %lnot to i32	• %lnot = (setcc %tobool, 0, seteq) • %conv = (and %lnot, 1) •	• xor $3, $4, $3
==	• %cmp = icmp eq i32 %0, %1 • %conv = zext i1 %cmp to i32	• %cmp = (setcc %0, %1, seteq) • and %cmp, 1	• xor $2, $3, $2 • sltiu $2, $2, 1 • andi $2, $2, 1
!=	• %cmp = icmp ne i32 %0, %1 • %conv = zext i1 %cmp to i32	• %cmp = (setcc %0, %1, setne) • and %cmp, 1	• xor $2, $3, $2 • sltu $2, $zero, 2 • shr $2, $2, 1 • andi $2, $2, 1
<	• %cmp = icmp lt i32 %0, %1 • %conv = zext i1 %cmp to i32	• (setcc %0, %1, setlt) • and %cmp, 1	• slt $2, $3, $2 • andi $2, $2, 1
<=	• %cmp = icmp le i32 %0, %1 • %conv = zext i1 %cmp to i32	• (setcc %0, %1, setle) • and %cmp, 1	• slt $2, $3, $2 • xori $2, $2, 1 • andi $2, $2, 1
>	• %cmp = icmp gt i32 %0, %1 • %conv = zext i1 %cmp to i32	• (setcc %0, %1, setgt) • and %cmp, 1	• slt $2, $3, $2 • andi $2, $2, 1
>=	• %cmp = icmp le i32 %0, %1 • %conv = zext i1 %cmp to i32	• (setcc %0, %1, setle) • and %cmp, 1	• slt $2, $3, $2 • xori $2, $2, 1 • andi $2, $2, 1

In relation operators ==, !=, . . . , %0 = $3 = 5, %1 = $2 = 3 for ch4_2_logic.cpp.

The "Optimized legalized selection DAG" is the last DAG stage just before the "instruction selection" as the previous section mentioned in this chapter. You can see the whole DAG stages by `llc -debug` option.

From above result, slt spend less instructions than cmp for relation operators translation. Beyond that, slt uses general purpose register while cmp uses $sw dedicated register.

lbdex/input/ch4_2_slt_explain.cpp

```
int test_OptSlt()
{
  int a = 3, b = 1;
  int d = 0, e = 0, f = 0;

  d = (a < 1);
  e = (b < 2);
  f = d + e;

  return (f);
}
```

```
118-165-78-10:input Jonathan$ clang -target mips-unknown-linux-gnu -O2
-c ch4_2_slt_explain.cpp -emit-llvm -o ch4_2_slt_explain.bc
118-165-78-10:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch4_2_slt_explain.bc -o -
  ...
  ld  $3, 20($sp)
  cmp $sw, $3, $2
  andi  $2, $sw, 1
  andi  $2, $2, 1
  st  $2, 12($sp)
  addiu $2, $zero, 2
  ld  $3, 16($sp)
  cmp $sw, $3, $2
  andi  $2, $sw, 1
  andi  $2, $2, 1
  ...
118-165-78-10:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032II -relocation-model=static -filetype=asm
ch4_2_slt_explain.bc -o -
  ...
  ld  $2, 20($sp)
  slti  $2, $2, 1
  andi  $2, $2, 1
  st  $2, 12($sp)
  ld  $2, 16($sp)
  slti  $2, $2, 2
  andi  $2, $2, 1
  st  $2, 8($sp)
  ...
```

Run these two *llc -mcpu* option for Chapter4_2 with ch4_2_slt_explain.cpp to get the above result. Regardless of the move between $sw and general purpose register in *llc -mcpu=cpu032I*, the two cmp instructions in it will has hazard in instruction reorder since both of them use $sw register. The *llc -mcpu=cpu032II* has not this problem because it uses

slti[9]. The slti version can reorder as follows,

```
...
ld  $2, 16($sp)
slti  $2, $2, 2
andi  $2, $2, 1
st  $2, 8($sp)
ld  $2, 20($sp)
slti  $2, $2, 1
andi  $2, $2, 1
st  $2, 12($sp)
...
```

Chapter4_2 include instructions cmp and slt. Though cpu032II include both of these two instructions, the slt takes the priority since "let Predicates = [HasSlt]" appeared before "let Predicates = [HasCmp]" in Cpu0InstrInfo.td.

4.3 Summary

List C operators, IR of .bc, Optimized legalized selection DAG and Cpu0 instructions implemented in this chapter in Table: Chapter 4 mathmetic operators. There are over 20 operators totally in mathmetic and logic support in this chapter and spend 4xx lines of source code.

[9] See book Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)

Table 4.7: Chapter 4 mathmetic operators

C	.bc	Optimized legalized selection DAG	Cpu0
+	add	add	addu
-	sub	sub	subu
*	mul	mul	mul
/	sdiv	Cpu0ISD::DivRem	div
•	udiv	Cpu0ISD::DivRemU	divu
<<	shl	shl	shl
>>	• ashr • lshr	• sra • srl	• sra • shr
!	• %tobool = icmp ne i32 %0, 0 • %lnot = xor i1 %tobool, true	• %lnot = (setcc %tobool, 0, seteq) • %conv = (and %lnot, 1)	• %1 = (xor %tobool, 0) • %true = (addiu $r0, 1) • %lnot = (xor %1, %true)
•	• %conv = zext i1 %lnot to i32	• %conv = (and %lnot, 1)	• %conv = (and %lnot, 1)
%	• srem • sremu	• Cpu0ISD::DivRem • Cpu0ISD::DivRemU	• div • divu
(x<<n)|(x>>32-n)	shl + lshr	rotl, rotr	rol, rolv, ror, rorv

CHAPTER

FIVE

GENERATING OBJECT FILES

- *Translate into obj file*
- *ELF obj related code*
- *Work flow*
- *Backend Target Registration Structure*

The previous chapters introducing the assembly code generation only. This chapter adding the elf obj support and verify the generated obj by objdump utility. With LLVM support, the Cpu0 backend can generate both big endian and little endian obj files with only a few code added. The Target Registration mechanism and their structure are introduced in this chapter.

5.1 Translate into obj file

Currently, we only support translation of llvm IR code into assembly code. If you try running Chapter4_2/ to translate it into obj code will get the error message as follows,

```
bin$ pwd
$HOME/llvm/test/build/bin/
bin$ llc -march=cpu0 -relocation-model=pic -filetype=obj ch4_1_math_math.bc -o
ch4_1_math.cpu0.o
~/llvm/test/build/bin/llc: target does not
support generation of this file type!
```

Chapter5_1/ support obj file generation. It produces obj files both for big endian and little endian with command `llc -march=cpu0` and `llc -march=cpu0el`, respectively. Run with them will get the obj files as follows,

```
input$ cat ch4_1_math.cpu0.s
...
  .set  nomacro
# BB#0:                                 # %entry
  addiu $sp, $sp, -40
$tmp1:
  .cfi_def_cfa_offset 40
  addiu $2, $zero, 5
  st  $2, 36($fp)
  addiu $2, $zero, 2
  st  $2, 32($fp)
```

(continues on next page)

(continued from previous page)

```
  addiu $2, $zero, 0
  st  $2, 28($fp)
...

bin$ pwd
$HOME/llvm/test/build/bin/
bin$ llc -march=cpu0 -relocation-model=pic -filetype=obj ch4_1_math.bc -o
ch4_1_math.cpu0.o
input$ objdump -s ch4_1_math.cpu0.o

ch4_1_math.cpu0.o:     file format elf32-big

Contents of section .text:
 0000 09ddffc8 09200005 022d0034 09200002  ..... ...-.4. ..
 0010 022d0030 0920fffb 022d002c 012d0030  .-.0. ...-.,.-.0
 0020 013d0034 11232000 022d0028 012d0030  .=.4.# ..-.(.-.0
 0030 013d0034 12232000 022d0024 012d0030  .=.4.# ..-.$.-.0
 0040 013d0034 17232000 022d0020 012d0034  .=.4.# ..-. .-.4
 0050 1e220002 022d001c 012d002c 1e220001  ."...-...-.,."..
 0060 022d000c 012d0034 1d220002 022d0018  .-...-.4."...-..
 0070 012d002c 1f22001e 022d0008 09200001  .-.,."...-... ..
 0080 013d0034 21323000 023d0014 013d0030  .=.4!20..=...=.0
 0090 21223000 022d0004 09200080 013d0034  !"0..-... ...=.4
 00a0 22223000 022d0010 012d0034 013d0030  ""0..-...-.4.=.0
 00b0 20232000 022d0000 09dd0038 3ce00000   # ..-.....8<...

input$ ~/llvm/test/
build/bin/llc -march=cpu0el -relocation-model=pic -filetype=obj
ch4_1_math.bc -o ch4_1_math.cpu0el.o
input$ objdump -s ch4_1_math.cpu0el.o

ch4_1_math.cpu0el.o:     file format elf32-little

Contents of section .text:
 0000 c8ffdd09 05002009 34002d02 02002009  ...... .4.-... .
 0010 30002d02 fbff2009 2c002d02 30002d01  0.-... .,.-.0.-.
 0020 34003d01 00202311 28002d02 30002d01  4.=.. #.(.-.0.-.
 0030 34003d01 00202312 24002d02 30002d01  4.=.. #.$.-.0.-.
 0040 34003d01 00202317 20002d02 34002d01  4.=.. #. .-.4.-.
 0050 0200221e 1c002d02 2c002d01 0100221e  .."...-.,.-..."."
 0060 0c002d02 34002d01 0200221d 18002d02  ..-.4.-..."...-.
 0070 2c002d01 1e00221f 08002d02 01002009  ,.-..."...-... .
 0080 34003d01 00303221 14003d02 30003d01  4.=..02!..=.0.=.
 0090 00302221 04002d02 80002009 34003d01  .0"!..-... .4.=.
 00a0 00302222 10002d02 34002d01 30003d01  .0""..-.4.-.0.=.
 00b0 00202320 00002d02 3800dd09 0000e03c  . # ..-.8......<
```

The first instruction is **"addiu $sp, -56"** and its corresponding obj is 0x09ddffc8. The opcode of addiu is 0x09, 8 bits; $sp register number is 13(0xd), 4bits; and the immediate is 16 bits -56(=0xffc8), so it is correct. The third instruction **"st $2, 52($fp)"** and it's corresponding obj is 0x022b0034. The **st** opcode is **0x02**, $2 is 0x2, $fp is 0xb and immediate is 52(0x0034). Thanks to Cpu0 instruction format which opcode, register operand and offset(imediate value) size are multiple of 4 bits. Base on the 4 bits multiple, the obj format is easy to check by eyes. The big endian (B0, B1, B2, B3) = (09, dd, ff, c8), objdump from B0 to B3 is 0x09ddffc8 and the little endian is (B3, B2, B1, B0) = (09, dd, ff, c8),

objdump from B0 to B3 is 0xc8ffdd09.

5.2 ELF obj related code

To support elf obj generation, the following code changed and added to Chapter5_1.

lbdex/chapters/Chapter5_1/InstPrinter/Cpu0InstPrinter.cpp

```
#include "MCTargetDesc/Cpu0MCExpr.h"
```

lbdex/chapters/Chapter5_1/MCTargetDesc/CMakeLists.txt

```
  Cpu0AsmBackend.cpp
  Cpu0MCCodeEmitter.cpp
  Cpu0MCExpr.cpp
  Cpu0ELFObjectWriter.cpp
  Cpu0TargetStreamer.cpp
```

lbdex/chapters/Chapter5_1/MCTargetDesc/Cpu0AsmBackend.h

```
//===-- Cpu0AsmBackend.h - Cpu0 Asm Backend  ------------------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file defines the Cpu0AsmBackend class.
//
//===----------------------------------------------------------------------===//
//

#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0ASMBACKEND_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0ASMBACKEND_H

#include "Cpu0Config.h"

#include "MCTargetDesc/Cpu0FixupKinds.h"
#include "llvm/ADT/Triple.h"
#include "llvm/MC/MCAsmBackend.h"

namespace llvm {

class MCAssembler;
struct MCFixupKindInfo;
```

(continues on next page)

(continued from previous page)

```
class Target;
class MCObjectWriter;

class Cpu0AsmBackend : public MCAsmBackend {
  Triple TheTriple;

public:
  Cpu0AsmBackend(const Target &T, const Triple &TT)
      : MCAsmBackend(TT.isLittleEndian() ? support::little : support::big),
        TheTriple(TT) {}

  std::unique_ptr<MCObjectTargetWriter>
  createObjectTargetWriter() const override;

  void applyFixup(const MCAssembler &Asm, const MCFixup &Fixup,
                  const MCValue &Target, MutableArrayRef<char> Data,
                  uint64_t Value, bool IsResolved,
                  const MCSubtargetInfo *STI) const override;

  const MCFixupKindInfo &getFixupKindInfo(MCFixupKind Kind) const override;

  unsigned getNumFixupKinds() const override {
    return Cpu0::NumTargetFixupKinds;
  }

  /// @name Target Relaxation Interfaces
  /// @{

  /// MayNeedRelaxation - Check whether the given instruction may need
  /// relaxation.
  ///
  /// \param Inst - The instruction to test.
  bool mayNeedRelaxation(const MCInst &Inst,
                         const MCSubtargetInfo &STI) const override {
    return false;
  }

  /// fixupNeedsRelaxation - Target specific predicate for whether a given
  /// fixup requires the associated instruction to be relaxed.
   bool fixupNeedsRelaxation(const MCFixup &Fixup, uint64_t Value,
                             const MCRelaxableFragment *DF,
                             const MCAsmLayout &Layout) const override {
    // FIXME.
    llvm_unreachable("RelaxInstruction() unimplemented");
    return false;
  }

  /// @}

  bool writeNopData(raw_ostream &OS, uint64_t Count) const override;
}; // class Cpu0AsmBackend
```

(continues on next page)

(continued from previous page)

```
} // namespace

#endif
```

lbdex/chapters/Chapter5_1/MCTargetDesc/Cpu0AsmBackend.cpp

```
//===-- Cpu0AsmBackend.cpp - Cpu0 Asm Backend  ----------------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file implements the Cpu0AsmBackend class.
//
//===----------------------------------------------------------------------===//
//

#include "MCTargetDesc/Cpu0FixupKinds.h"
#include "MCTargetDesc/Cpu0AsmBackend.h"

#include "MCTargetDesc/Cpu0MCTargetDesc.h"
#include "llvm/MC/MCAsmBackend.h"
#include "llvm/MC/MCAssembler.h"
#include "llvm/MC/MCDirectives.h"
#include "llvm/MC/MCELFObjectWriter.h"
#include "llvm/MC/MCFixupKindInfo.h"
#include "llvm/MC/MCObjectWriter.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

static cl::opt<bool> HasLLD(
  "has-lld",
  cl::init(false),
  cl::desc("CPU0: Has lld linker for Cpu0."),
  cl::Hidden);

//@adjustFixupValue {
// Prepare value for the target space for it
static unsigned adjustFixupValue(const MCFixup &Fixup, uint64_t Value,
                                 MCContext &Ctx) {

  unsigned Kind = Fixup.getKind();
```

(continues on next page)

(continued from previous page)

```
  // Add/subtract and shift
  switch (Kind) {
  default:
    return 0;
  case FK_GPRel_4:
  case FK_Data_4:
  case Cpu0::fixup_Cpu0_LO16:
    break;
  case Cpu0::fixup_Cpu0_HI16:
  case Cpu0::fixup_Cpu0_GOT:
    // Get the higher 16-bits. Also add 1 if bit 15 is 1.
    Value = (Value >> 16) & 0xffff;
    break;
  }

  return Value;
}
//@adjustFixupValue }

std::unique_ptr<MCObjectTargetWriter>
Cpu0AsmBackend::createObjectTargetWriter() const {
  return createCpu0ELFObjectWriter(TheTriple);
}

/// ApplyFixup - Apply the \p Value for given \p Fixup into the provided
/// data fragment, at the offset specified by the fixup and following the
/// fixup kind as appropriate.
void Cpu0AsmBackend::applyFixup(const MCAssembler &Asm, const MCFixup &Fixup,
                                const MCValue &Target,
                                MutableArrayRef<char> Data, uint64_t Value,
                                bool IsResolved,
                                const MCSubtargetInfo *STI) const {
  MCFixupKind Kind = Fixup.getKind();
  MCContext &Ctx = Asm.getContext();
  Value = adjustFixupValue(Fixup, Value, Ctx);

  if (!Value)
    return; // Doesn't change encoding.

  // Where do we start in the object
  unsigned Offset = Fixup.getOffset();
  // Number of bytes we need to fixup
  unsigned NumBytes = (getFixupKindInfo(Kind).TargetSize + 7) / 8;
  // Used to point to big endian bytes
  unsigned FullSize;

  switch ((unsigned)Kind) {
  default:
    FullSize = 4;
    break;
  }
```

(continues on next page)

(continued from previous page)

```
  // Grab current value, if any, from bits.
  uint64_t CurVal = 0;

  for (unsigned i = 0; i != NumBytes; ++i) {
    unsigned Idx = TheTriple.isLittleEndian() ? i : (FullSize - 1 - i);
    CurVal |= (uint64_t)((uint8_t)Data[Offset + Idx]) << (i*8);
  }

  uint64_t Mask = ((uint64_t)(-1) >>
                   (64 - getFixupKindInfo(Kind).TargetSize));
  CurVal |= Value & Mask;

  // Write out the fixed up bytes back to the code/data bits.
  for (unsigned i = 0; i != NumBytes; ++i) {
    unsigned Idx = TheTriple.isLittleEndian() ? i : (FullSize - 1 - i);
    Data[Offset + Idx] = (uint8_t)((CurVal >> (i*8)) & 0xff);
  }
}

//@getFixupKindInfo {
const MCFixupKindInfo &Cpu0AsmBackend::
getFixupKindInfo(MCFixupKind Kind) const {
  unsigned JSUBReloRec = 0;
  if (HasLLD) {
    JSUBReloRec = MCFixupKindInfo::FKF_IsPCRel;
  }
  else {
    JSUBReloRec = MCFixupKindInfo::FKF_IsPCRel | MCFixupKindInfo::FKF_Constant;
  }
  const static MCFixupKindInfo Infos[Cpu0::NumTargetFixupKinds] = {
    // This table *must* be in same the order of fixup_* kinds in
    // Cpu0FixupKinds.h.
    //
    // name                         offset  bits  flags
    { "fixup_Cpu0_32",              0,     32,   0 },
    { "fixup_Cpu0_HI16",            0,     16,   0 },
    { "fixup_Cpu0_LO16",            0,     16,   0 },
    { "fixup_Cpu0_GPREL16",         0,     16,   0 },
    { "fixup_Cpu0_GOT",             0,     16,   0 },
    { "fixup_Cpu0_GOT_HI16",        0,     16,   0 },
    { "fixup_Cpu0_GOT_LO16",        0,     16,   0 }
  };

  if (Kind < FirstTargetFixupKind)
    return MCAsmBackend::getFixupKindInfo(Kind);

  assert(unsigned(Kind - FirstTargetFixupKind) < getNumFixupKinds() &&
         "Invalid kind!");
  return Infos[Kind - FirstTargetFixupKind];
}
//@getFixupKindInfo }
```

(continues on next page)

(continued from previous page)

```
/// WriteNopData - Write an (optimal) nop sequence of Count bytes
/// to the given output. If the target cannot generate such a sequence,
/// it should return an error.
///
/// \return - True on success.
bool Cpu0AsmBackend::writeNopData(raw_ostream &OS, uint64_t Count) const {
  return true;
}

// MCAsmBackend
MCAsmBackend *llvm::createCpu0AsmBackend(const Target &T,
                                         const MCSubtargetInfo &STI,
                                         const MCRegisterInfo &MRI,
                                         const MCTargetOptions &Options) {
  return new Cpu0AsmBackend(T, STI.getTargetTriple());
}
```

lbdex/chapters/Chapter5_1/MCTargetDesc/Cpu0BaseInfo.h

```
#include "Cpu0FixupKinds.h"
```

lbdex/chapters/Chapter5_1/MCTargetDesc/Cpu0ELFObjectWriter.cpp

```
//===-- Cpu0ELFObjectWriter.cpp - Cpu0 ELF Writer -------------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

#include "Cpu0Config.h"

#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "MCTargetDesc/Cpu0FixupKinds.h"
#include "MCTargetDesc/Cpu0MCTargetDesc.h"
#include "llvm/MC/MCAssembler.h"
#include "llvm/MC/MCELFObjectWriter.h"
#include "llvm/MC/MCExpr.h"
#include "llvm/MC/MCSection.h"
#include "llvm/MC/MCValue.h"
#include "llvm/Support/ErrorHandling.h"
#include <list>

using namespace llvm;
```

(continues on next page)

(continued from previous page)

```
namespace {
  class Cpu0ELFObjectWriter : public MCELFObjectTargetWriter {
  public:
    Cpu0ELFObjectWriter(uint8_t OSABI, bool HasRelocationAddend, bool Is64);

        ~Cpu0ELFObjectWriter() = default;

    unsigned getRelocType(MCContext &Ctx, const MCValue &Target,
                          const MCFixup &Fixup, bool IsPCRel) const override;
    bool needsRelocateWithSymbol(const MCSymbol &Sym,
                                 unsigned Type) const override;
  };
}

Cpu0ELFObjectWriter::Cpu0ELFObjectWriter(uint8_t OSABI,
                                         bool HasRelocationAddend, bool Is64)
    : MCELFObjectTargetWriter(/*Is64Bit_=false*/ Is64, OSABI, ELF::EM_CPU0,
          /*HasRelocationAddend_ = false*/ HasRelocationAddend) {}

//@GetRelocType {
unsigned Cpu0ELFObjectWriter::getRelocType(MCContext &Ctx,
                                           const MCValue &Target,
                                           const MCFixup &Fixup,
                                           bool IsPCRel) const {
  // determine the type of the relocation
  unsigned Type = (unsigned)ELF::R_CPU0_NONE;
  unsigned Kind = (unsigned)Fixup.getKind();

  switch (Kind) {
  default:
    llvm_unreachable("invalid fixup kind!");
  case FK_Data_4:
    Type = ELF::R_CPU0_32;
    break;
  case Cpu0::fixup_Cpu0_32:
    Type = ELF::R_CPU0_32;
    break;
  case Cpu0::fixup_Cpu0_GPREL16:
    Type = ELF::R_CPU0_GPREL16;
    break;
  case Cpu0::fixup_Cpu0_GOT:
    Type = ELF::R_CPU0_GOT16;
    break;
  case Cpu0::fixup_Cpu0_HI16:
    Type = ELF::R_CPU0_HI16;
    break;
  case Cpu0::fixup_Cpu0_LO16:
    Type = ELF::R_CPU0_LO16;
    break;
  case Cpu0::fixup_Cpu0_GOT_HI16:
    Type = ELF::R_CPU0_GOT_HI16;
    break;
```

(continues on next page)

(continued from previous page)

```
  case Cpu0::fixup_Cpu0_GOT_LO16:
    Type = ELF::R_CPU0_GOT_LO16;
    break;
  }

  return Type;
}
//@GetRelocType }

bool
Cpu0ELFObjectWriter::needsRelocateWithSymbol(const MCSymbol &Sym,
                                             unsigned Type) const {
  // FIXME: This is extremelly conservative. This really needs to use a
  // whitelist with a clear explanation for why each realocation needs to
  // point to the symbol, not to the section.
  switch (Type) {
  default:
    return true;

  case ELF::R_CPU0_GOT16:
  // For Cpu0 pic mode, I think it's OK to return true but I didn't confirm.
  //  llvm_unreachable("Should have been handled already");
    return true;

  // These relocations might be paired with another relocation. The pairing is
  // done by the static linker by matching the symbol. Since we only see one
  // relocation at a time, we have to force them to relocate with a symbol to
  // avoid ending up with a pair where one points to a section and another
  // points to a symbol.
  case ELF::R_CPU0_HI16:
  case ELF::R_CPU0_LO16:
  // R_CPU0_32 should be a relocation record, I don't know why Mips set it to
  // false.
  case ELF::R_CPU0_32:
    return true;

  case ELF::R_CPU0_GPREL16:
    return false;
  }
}

std::unique_ptr<MCObjectTargetWriter>
llvm::createCpu0ELFObjectWriter(const Triple &TT) {
  uint8_t OSABI = MCELFObjectTargetWriter::getOSABI(TT.getOS());
  bool IsN64 = false;
  bool HasRelocationAddend = TT.isArch64Bit();
  return std::make_unique<Cpu0ELFObjectWriter>(OSABI, HasRelocationAddend,
                                               IsN64);
}
```

lbdex/chapters/Chapter5_1/MCTargetDesc/Cpu0FixupKinds.h

```
//===-- Cpu0FixupKinds.h - Cpu0 Specific Fixup Entries ----------*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0FIXUPKINDS_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0FIXUPKINDS_H

#include "Cpu0Config.h"

#include "llvm/MC/MCFixup.h"

namespace llvm {
namespace Cpu0 {
  // Although most of the current fixup types reflect a unique relocation
  // one can have multiple fixup types for a given relocation and thus need
  // to be uniquely named.
  //
  // This table *must* be in the save order of
  // MCFixupKindInfo Infos[Cpu0::NumTargetFixupKinds]
  // in Cpu0AsmBackend.cpp.
  //@Fixups {
  enum Fixups {
    //@ Pure upper 32 bit fixup resulting in - R_CPU0_32.
    fixup_Cpu0_32 = FirstTargetFixupKind,

    // Pure upper 16 bit fixup resulting in - R_CPU0_HI16.
    fixup_Cpu0_HI16,

    // Pure lower 16 bit fixup resulting in - R_CPU0_LO16.
    fixup_Cpu0_LO16,

    // 16 bit fixup for GP offest resulting in - R_CPU0_GPREL16.
    fixup_Cpu0_GPREL16,

    // GOT (Global Offset Table)
    // Symbol fixup resulting in - R_CPU0_GOT16.
    fixup_Cpu0_GOT,



    // resulting in - R_CPU0_GOT_HI16
    fixup_Cpu0_GOT_HI16,

    // resulting in - R_CPU0_GOT_LO16
    fixup_Cpu0_GOT_LO16,
```

(continues on next page)

(continued from previous page)

```
    // Marker
    LastTargetFixupKind,
    NumTargetFixupKinds = LastTargetFixupKind - FirstTargetFixupKind
  };
  //@Fixups }
} // namespace Cpu0
} // namespace llvm

#endif // LLVM_CPU0_CPU0FIXUPKINDS_H
```

lbdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCCodeEmitter.h

```
//===-- Cpu0MCCodeEmitter.h - Convert Cpu0 Code to Machine Code -----------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file defines the Cpu0MCCodeEmitter class.
//
//===----------------------------------------------------------------------===//
//

#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCCODEEMITTER_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCCODEEMITTER_H

#include "Cpu0Config.h"

#include "llvm/MC/MCCodeEmitter.h"
#include "llvm/Support/DataTypes.h"

using namespace llvm;

namespace llvm {
class MCContext;
class MCExpr;
class MCInst;
class MCInstrInfo;
class MCFixup;
class MCOperand;
class MCSubtargetInfo;
class raw_ostream;

class Cpu0MCCodeEmitter : public MCCodeEmitter {
  Cpu0MCCodeEmitter(const Cpu0MCCodeEmitter &) = delete;
  void operator=(const Cpu0MCCodeEmitter &) = delete;
```

(continues on next page)

(continued from previous page)

```
  const MCInstrInfo &MCII;
  MCContext &Ctx;
  bool IsLittleEndian;

public:
  Cpu0MCCodeEmitter(const MCInstrInfo &mcii, MCContext &Ctx_, bool IsLittle)
      : MCII(mcii), Ctx(Ctx_), IsLittleEndian(IsLittle) {}

  ~Cpu0MCCodeEmitter() override {}

  void EmitByte(unsigned char C, raw_ostream &OS) const;

  void EmitInstruction(uint64_t Val, unsigned Size, raw_ostream &OS) const;

  void encodeInstruction(const MCInst &MI, raw_ostream &OS,
                         SmallVectorImpl<MCFixup> &Fixups,
                         const MCSubtargetInfo &STI) const override;

  // getBinaryCodeForInstr - TableGen'erated function for getting the
  // binary encoding for an instruction.
  uint64_t getBinaryCodeForInstr(const MCInst &MI,
                                 SmallVectorImpl<MCFixup> &Fixups,
                                 const MCSubtargetInfo &STI) const;

  // getBranch16TargetOpValue - Return binary encoding of the branch
  // target operand, such as BEQ, BNE. If the machine operand
  // requires relocation, record the relocation and return zero.
  unsigned getBranch16TargetOpValue(const MCInst &MI, unsigned OpNo,
                                    SmallVectorImpl<MCFixup> &Fixups,
                                    const MCSubtargetInfo &STI) const;

  // getBranch24TargetOpValue - Return binary encoding of the branch
  // target operand, such as JMP #BB01, JEQ, JSUB. If the machine operand
  // requires relocation, record the relocation and return zero.
  unsigned getBranch24TargetOpValue(const MCInst &MI, unsigned OpNo,
                                    SmallVectorImpl<MCFixup> &Fixups,
                                    const MCSubtargetInfo &STI) const;

  // getJumpTargetOpValue - Return binary encoding of the jump
  // target operand, such as JSUB #function_addr.
  // If the machine operand requires relocation,
  // record the relocation and return zero.
   unsigned getJumpTargetOpValue(const MCInst &MI, unsigned OpNo,
                                 SmallVectorImpl<MCFixup> &Fixups,
                                 const MCSubtargetInfo &STI) const;

  // getMachineOpValue - Return binary encoding of operand. If the machin
  // operand requires relocation, record the relocation and return zero.
  unsigned getMachineOpValue(const MCInst &MI, const MCOperand &MO,
                             SmallVectorImpl<MCFixup> &Fixups,
                             const MCSubtargetInfo &STI) const;
```

(continues on next page)

(continued from previous page)

```
  unsigned getMemEncoding(const MCInst &MI, unsigned OpNo,
                          SmallVectorImpl<MCFixup> &Fixups,
                          const MCSubtargetInfo &STI) const;

  unsigned getExprOpValue(const MCExpr *Expr, SmallVectorImpl<MCFixup> &Fixups,
                          const MCSubtargetInfo &STI) const;
}; // class Cpu0MCCodeEmitter
} // namespace llvm.

#endif
```

lbdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
//===-- Cpu0MCCodeEmitter.cpp - Convert Cpu0 Code to Machine Code ---------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file implements the Cpu0MCCodeEmitter class.
//
//===----------------------------------------------------------------------===//
//

#include "Cpu0MCCodeEmitter.h"

#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "MCTargetDesc/Cpu0FixupKinds.h"
#include "MCTargetDesc/Cpu0MCExpr.h"
#include "MCTargetDesc/Cpu0MCTargetDesc.h"
#include "llvm/ADT/APFloat.h"
#include "llvm/MC/MCCodeEmitter.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCExpr.h"
#include "llvm/MC/MCInst.h"
#include "llvm/MC/MCInstrInfo.h"
#include "llvm/MC/MCRegisterInfo.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/Support/raw_ostream.h"

#define DEBUG_TYPE "mccodeemitter"

#define GET_INSTRMAP_INFO
#include "Cpu0GenInstrInfo.inc"
#undef GET_INSTRMAP_INFO
```

(continues on next page)

(continued from previous page)

```
using namespace llvm;

MCCodeEmitter *llvm::createCpu0MCCodeEmitterEB(const MCInstrInfo &MCII,
                                               const MCRegisterInfo &MRI,
                                               MCContext &Ctx) {
  return new Cpu0MCCodeEmitter(MCII, Ctx, false);
}

MCCodeEmitter *llvm::createCpu0MCCodeEmitterEL(const MCInstrInfo &MCII,
                                               const MCRegisterInfo &MRI,
                                               MCContext &Ctx) {
  return new Cpu0MCCodeEmitter(MCII, Ctx, true);
}

void Cpu0MCCodeEmitter::EmitByte(unsigned char C, raw_ostream &OS) const {
  OS << (char)C;
}

void Cpu0MCCodeEmitter::EmitInstruction(uint64_t Val, unsigned Size, raw_ostream &OS)␣
↪const {
  // Output the instruction encoding in little endian byte order.
  for (unsigned i = 0; i < Size; ++i) {
    unsigned Shift = IsLittleEndian ? i * 8 : (Size - 1 - i) * 8;
    EmitByte((Val >> Shift) & 0xff, OS);
  }
}

/// encodeInstruction - Emit the instruction.
/// Size the instruction (currently only 4 bytes)
void Cpu0MCCodeEmitter::
encodeInstruction(const MCInst &MI, raw_ostream &OS,
                  SmallVectorImpl<MCFixup> &Fixups,
                  const MCSubtargetInfo &STI) const
{
  uint32_t Binary = getBinaryCodeForInstr(MI, Fixups, STI);

  // Check for unimplemented opcodes.
  // Unfortunately in CPU0 both NOT and SLL will come in with Binary == 0
  // so we have to special check for them.
  unsigned Opcode = MI.getOpcode();
  if ((Opcode != Cpu0::NOP) && (Opcode != Cpu0::SHL) && !Binary)
    llvm_unreachable("unimplemented opcode in encodeInstruction()");

  const MCInstrDesc &Desc = MCII.get(MI.getOpcode());
  uint64_t TSFlags = Desc.TSFlags;

  // Pseudo instructions don't get encoded and shouldn't be here
  // in the first place!
  if ((TSFlags & Cpu0II::FormMask) == Cpu0II::Pseudo)
    llvm_unreachable("Pseudo opcode found in encodeInstruction()");

  // For now all instructions are 4 bytes
```

(continues on next page)

(continued from previous page)

```
  int Size = 4; // FIXME: Have Desc.getSize() return the correct value!

  EmitInstruction(Binary, Size, OS);
}

//@CH8_1 {
/// getBranch16TargetOpValue - Return binary encoding of the branch
/// target operand. If the machine operand requires relocation,
/// record the relocation and return zero.
unsigned Cpu0MCCodeEmitter::
getBranch16TargetOpValue(const MCInst &MI, unsigned OpNo,
                         SmallVectorImpl<MCFixup> &Fixups,
                         const MCSubtargetInfo &STI) const {
  return 0;
}

/// getBranch24TargetOpValue - Return binary encoding of the branch
/// target operand. If the machine operand requires relocation,
/// record the relocation and return zero.
unsigned Cpu0MCCodeEmitter::
getBranch24TargetOpValue(const MCInst &MI, unsigned OpNo,
                        SmallVectorImpl<MCFixup> &Fixups,
                        const MCSubtargetInfo &STI) const {
  return 0;
}

/// getJumpTargetOpValue - Return binary encoding of the jump
/// target operand, such as JSUB.
/// If the machine operand requires relocation,
/// record the relocation and return zero.
//@getJumpTargetOpValue {
unsigned Cpu0MCCodeEmitter::
getJumpTargetOpValue(const MCInst &MI, unsigned OpNo,
                     SmallVectorImpl<MCFixup> &Fixups,
                     const MCSubtargetInfo &STI) const {
  return 0;
}
//@CH8_1 }

//@getExprOpValue {
unsigned Cpu0MCCodeEmitter::
getExprOpValue(const MCExpr *Expr,SmallVectorImpl<MCFixup> &Fixups,
               const MCSubtargetInfo &STI) const {
//@getExprOpValue body {
  MCExpr::ExprKind Kind = Expr->getKind();
  if (Kind == MCExpr::Constant) {
    return cast<MCConstantExpr>(Expr)->getValue();
  }

  if (Kind == MCExpr::Binary) {
    unsigned Res = getExprOpValue(cast<MCBinaryExpr>(Expr)->getLHS(), Fixups, STI);
    Res += getExprOpValue(cast<MCBinaryExpr>(Expr)->getRHS(), Fixups, STI);
```

(continues on next page)

(continued from previous page)

```
    return Res;
  }

  if (Kind == MCExpr::Target) {
    const Cpu0MCExpr *Cpu0Expr = cast<Cpu0MCExpr>(Expr);

    Cpu0::Fixups FixupKind = Cpu0::Fixups(0);
    switch (Cpu0Expr->getKind()) {
    default: llvm_unreachable("Unsupported fixup kind for target expression!");
    } // switch
    Fixups.push_back(MCFixup::create(0, Expr, MCFixupKind(FixupKind)));
    return 0;
  }

  // All of the information is in the fixup.
  return 0;
}

/// getMachineOpValue - Return binary encoding of operand. If the machine
/// operand requires relocation, record the relocation and return zero.
unsigned Cpu0MCCodeEmitter::
getMachineOpValue(const MCInst &MI, const MCOperand &MO,
                  SmallVectorImpl<MCFixup> &Fixups,
                  const MCSubtargetInfo &STI) const {
  if (MO.isReg()) {
    unsigned Reg = MO.getReg();
    unsigned RegNo = Ctx.getRegisterInfo()->getEncodingValue(Reg);
    return RegNo;
  } else if (MO.isImm()) {
    return static_cast<unsigned>(MO.getImm());
  } else if (MO.isFPImm()) {
    return static_cast<unsigned>(APFloat(MO.getFPImm())
        .bitcastToAPInt().getHiBits(32).getLimitedValue());
  }
  // MO must be an Expr.
  assert(MO.isExpr());
  return getExprOpValue(MO.getExpr(),Fixups, STI);
}

/// getMemEncoding - Return binary encoding of memory related operand.
/// If the offset operand requires relocation, record the relocation.
unsigned
Cpu0MCCodeEmitter::getMemEncoding(const MCInst &MI, unsigned OpNo,
                                  SmallVectorImpl<MCFixup> &Fixups,
                                  const MCSubtargetInfo &STI) const {
  // Base register is encoded in bits 20-16, offset is encoded in bits 15-0.
  assert(MI.getOperand(OpNo).isReg());
  unsigned RegBits = getMachineOpValue(MI, MI.getOperand(OpNo), Fixups, STI) << 16;
  unsigned OffBits = getMachineOpValue(MI, MI.getOperand(OpNo+1), Fixups, STI);

  return (OffBits & 0xFFFF) | RegBits;
}
```

(continues on next page)

(continued from previous page)

```
#include "Cpu0GenMCCodeEmitter.inc"
```

lbdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCExpr.h

```
//===-- Cpu0MCExpr.h - Cpu0 specific MC expression classes ------*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCEXPR_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCEXPR_H

#include "Cpu0Config.h"
#if CH >= CH5_1

#include "llvm/MC/MCAsmLayout.h"
#include "llvm/MC/MCExpr.h"
#include "llvm/MC/MCValue.h"

namespace llvm {

class Cpu0MCExpr : public MCTargetExpr {
public:
  enum Cpu0ExprKind {
    CEK_None,
    CEK_ABS_HI,
    CEK_ABS_LO,
    CEK_CALL_HI16,
    CEK_CALL_LO16,
    CEK_DTP_HI,
    CEK_DTP_LO,
    CEK_GOT,
    CEK_GOTTPREL,
    CEK_GOT_CALL,
    CEK_GOT_DISP,
    CEK_GOT_HI16,
    CEK_GOT_LO16,
    CEK_GPREL,
    CEK_TLSGD,
    CEK_TLSLDM,
    CEK_TP_HI,
    CEK_TP_LO,
    CEK_Special,
  };
```

(continues on next page)

(continued from previous page)

```
private:
  const Cpu0ExprKind Kind;
  const MCExpr *Expr;

  explicit Cpu0MCExpr(Cpu0ExprKind Kind, const MCExpr *Expr)
    : Kind(Kind), Expr(Expr) {}

public:
  static const Cpu0MCExpr *create(Cpu0ExprKind Kind, const MCExpr *Expr,
                                  MCContext &Ctx);
  static const Cpu0MCExpr *create(const MCSymbol *Symbol,
                                  Cpu0MCExpr::Cpu0ExprKind Kind, MCContext &Ctx);
  static const Cpu0MCExpr *createGpOff(Cpu0ExprKind Kind, const MCExpr *Expr,
                                       MCContext &Ctx);

  /// Get the kind of this expression.
  Cpu0ExprKind getKind() const { return Kind; }

  /// Get the child of this expression.
  const MCExpr *getSubExpr() const { return Expr; }

  void printImpl(raw_ostream &OS, const MCAsmInfo *MAI) const override;
  bool evaluateAsRelocatableImpl(MCValue &Res, const MCAsmLayout *Layout,
                                 const MCFixup *Fixup) const override;
  void visitUsedExpr(MCStreamer &Streamer) const override;
  MCFragment *findAssociatedFragment() const override {
    return getSubExpr()->findAssociatedFragment();
  }

  void fixELFSymbolsInTLSFixups(MCAssembler &Asm) const override;

  static bool classof(const MCExpr *E) {
    return E->getKind() == MCExpr::Target;
  }

  bool isGpOff(Cpu0ExprKind &Kind) const;
  bool isGpOff() const {
    Cpu0ExprKind Kind;
    return isGpOff(Kind);
  }
};
} // end namespace llvm

#endif // #if CH >= CH5_1

#endif
```

lbdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCExpr.cpp

```
//===-- Cpu0MCExpr.cpp - Cpu0 specific MC expression classes --------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

#include "Cpu0.h"

#if CH >= CH5_1

#include "Cpu0MCExpr.h"
#include "llvm/BinaryFormat/ELF.h"
#include "llvm/MC/MCAsmInfo.h"
#include "llvm/MC/MCAssembler.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCObjectStreamer.h"
#include "llvm/MC/MCSymbolELF.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0mcexpr"

const Cpu0MCExpr *Cpu0MCExpr::create(Cpu0MCExpr::Cpu0ExprKind Kind,
                                     const MCExpr *Expr, MCContext &Ctx) {
  return new (Ctx) Cpu0MCExpr(Kind, Expr);
}

const Cpu0MCExpr *Cpu0MCExpr::create(const MCSymbol *Symbol, Cpu0MCExpr::Cpu0ExprKind␣
↪Kind,
                           MCContext &Ctx) {
  const MCSymbolRefExpr *MCSym =
      MCSymbolRefExpr::create(Symbol, MCSymbolRefExpr::VK_None, Ctx);
  return new (Ctx) Cpu0MCExpr(Kind, MCSym);
}

const Cpu0MCExpr *Cpu0MCExpr::createGpOff(Cpu0MCExpr::Cpu0ExprKind Kind,
                                          const MCExpr *Expr, MCContext &Ctx) {
  return create(Kind, create(CEK_None, create(CEK_GPREL, Expr, Ctx), Ctx), Ctx);
}

void Cpu0MCExpr::printImpl(raw_ostream &OS, const MCAsmInfo *MAI) const {
  int64_t AbsVal;

  switch (Kind) {
  case CEK_None:
  case CEK_Special:
    llvm_unreachable("CEK_None and CEK_Special are invalid");
    break;
```

(continues on next page)

(continued from previous page)

```
  case CEK_CALL_HI16:
    OS << "%call_hi";
    break;
  case CEK_CALL_LO16:
    OS << "%call_lo";
    break;
  case CEK_DTP_HI:
    OS << "%dtp_hi";
    break;
  case CEK_DTP_LO:
    OS << "%dtp_lo";
    break;
  case CEK_GOT:
    OS << "%got";
    break;
  case CEK_GOTTPREL:
    OS << "%gottprel";
    break;
  case CEK_GOT_CALL:
    OS << "%call16";
    break;
  case CEK_GOT_DISP:
    OS << "%got_disp";
    break;
  case CEK_GOT_HI16:
    OS << "%got_hi";
    break;
  case CEK_GOT_LO16:
    OS << "%got_lo";
    break;
  case CEK_GPREL:
    OS << "%gp_rel";
    break;
  case CEK_ABS_HI:
    OS << "%hi";
    break;
  case CEK_ABS_LO:
    OS << "%lo";
    break;
  case CEK_TLSGD:
    OS << "%tlsgd";
    break;
  case CEK_TLSLDM:
    OS << "%tlsldm";
    break;
  case CEK_TP_HI:
    OS << "%tp_hi";
    break;
  case CEK_TP_LO:
    OS << "%tp_lo";
    break;
  }
```

(continues on next page)

(continued from previous page)

```
  OS << '(';
  if (Expr->evaluateAsAbsolute(AbsVal))
    OS << AbsVal;
  else
    Expr->print(OS, MAI, true);
  OS << ')';
}

bool
Cpu0MCExpr::evaluateAsRelocatableImpl(MCValue &Res,
                                      const MCAsmLayout *Layout,
                                      const MCFixup *Fixup) const {
  return getSubExpr()->evaluateAsRelocatable(Res, Layout, Fixup);
}

void Cpu0MCExpr::visitUsedExpr(MCStreamer &Streamer) const {
  Streamer.visitUsedExpr(*getSubExpr());
}

void Cpu0MCExpr::fixELFSymbolsInTLSFixups(MCAssembler &Asm) const {
  switch ((int)getKind()) {
  case CEK_None:
  case CEK_Special:
    llvm_unreachable("CEK_None and CEK_Special are invalid");
    break;
  case CEK_CALL_HI16:
  case CEK_CALL_LO16:
    break;
  }
}

bool Cpu0MCExpr::isGpOff(Cpu0ExprKind &Kind) const {
  if (const Cpu0MCExpr *S1 = dyn_cast<const Cpu0MCExpr>(getSubExpr())) {
    if (const Cpu0MCExpr *S2 = dyn_cast<const Cpu0MCExpr>(S1->getSubExpr())) {
      if (S1->getKind() == CEK_None && S2->getKind() == CEK_GPREL) {
        Kind = getKind();
        return true;
      }
    }
  }
  return false;
}

#endif
```

lbdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCTargetDesc.h

```
MCCodeEmitter *createCpu0MCCodeEmitterEB(const MCInstrInfo &MCII,
                                         const MCRegisterInfo &MRI,
                                         MCContext &Ctx);
MCCodeEmitter *createCpu0MCCodeEmitterEL(const MCInstrInfo &MCII,
                                         const MCRegisterInfo &MRI,
                                         MCContext &Ctx);

MCAsmBackend *createCpu0AsmBackend(const Target &T,
                                   const MCSubtargetInfo &STI,
                                   const MCRegisterInfo &MRI,
                                   const MCTargetOptions &Options);

std::unique_ptr<MCObjectTargetWriter> createCpu0ELFObjectWriter(const Triple &TT);
```

lbdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCTargetDesc.cpp

```
static MCStreamer *createMCStreamer(const Triple &TT, MCContext &Context,
                                    std::unique_ptr<MCAsmBackend> &&MAB,
                                    std::unique_ptr<MCObjectWriter> &&OW,
                                    std::unique_ptr<MCCodeEmitter> &&Emitter,
                                    bool RelaxAll) {
  return createELFStreamer(Context, std::move(MAB), std::move(OW),
                           std::move(Emitter), RelaxAll);;
}

static MCTargetStreamer *createCpu0AsmTargetStreamer(MCStreamer &S,
                                                     formatted_raw_ostream &OS,
                                                     MCInstPrinter *InstPrint,
                                                     bool isVerboseAsm) {
  return new Cpu0TargetAsmStreamer(S, OS);
}
```

```
extern "C" void LLVMInitializeCpu0TargetMC() {
```

```
    // Register the elf streamer.
    TargetRegistry::RegisterELFStreamer(*T, createMCStreamer);

    // Register the asm target streamer.
    TargetRegistry::RegisterAsmTargetStreamer(*T, createCpu0AsmTargetStreamer);

    // Register the asm backend.
    TargetRegistry::RegisterMCAsmBackend(*T, createCpu0AsmBackend);
```

```
  // Register the MC Code Emitter
  TargetRegistry::RegisterMCCodeEmitter(TheCpu0Target,
                                        createCpu0MCCodeEmitterEB);
  TargetRegistry::RegisterMCCodeEmitter(TheCpu0elTarget,
                                        createCpu0MCCodeEmitterEL);
```

```
}
```

lbdex/chapters/Chapter5_1/Cpu0MCInstLower.h

```
#include "MCTargetDesc/Cpu0MCExpr.h"
```

5.3 Work flow

In Chapter3_2, OutStreamer->emitInstruction print the asm. To support elf obj generation, this chapter create MCELFObjectStreamer inherited from OutStreamer by calling createELFStreamer in Cpu0MCTargetDesc.cpp above. Once MCELFObjectStreamer is created. The OutStreamer->emitInstruction will work with other code added in directory MCTargetDesc of this chapter. The details of expanation as follows,

llvm/include/llvm/CodeGen/AsmPrinter.h

```
class AsmPrinter : public MachineFunctionPass {
public:
  ...
  std::unique_ptr<MCStreamer> OutStreamer;
  ...
}
```

lbdex/chapters/Chapter3_2/Cpu0AsmPrinter.h

```
class LLVM_LIBRARY_VISIBILITY Cpu0AsmPrinter : public AsmPrinter {
```

lbdex/chapters/Chapter3_2/Cpu0AsmPrinter.cpp

```
void Cpu0AsmPrinter::emitInstruction(const MachineInstr *MI) {
  ...
  do {
    ...
    OutStreamer->emitInstruction(TmpInst0, getSubtargetInfo());
    ...
  } while ((++I != E) && I->isInsideBundle()); // Delay slot check
}
```

The ELF encoder calling functions shown as Fig. 5.1 above. AsmPrinter::OutStreamer is set to MCObjectStreamer when by llc driver when user input `llc -filetype=obj`.

The instruction operands information for encoder is got as Fig. 5.2 above. Steps as follows,

1. Function encodeInstruction() pass MI.Opcode to getBinaryCodeForInstr().
2. getBinaryCodeForInstr() pass MI.Operand[n] to getMachineOpValue() and then,
3. get register number by calling getMachineOpValue().

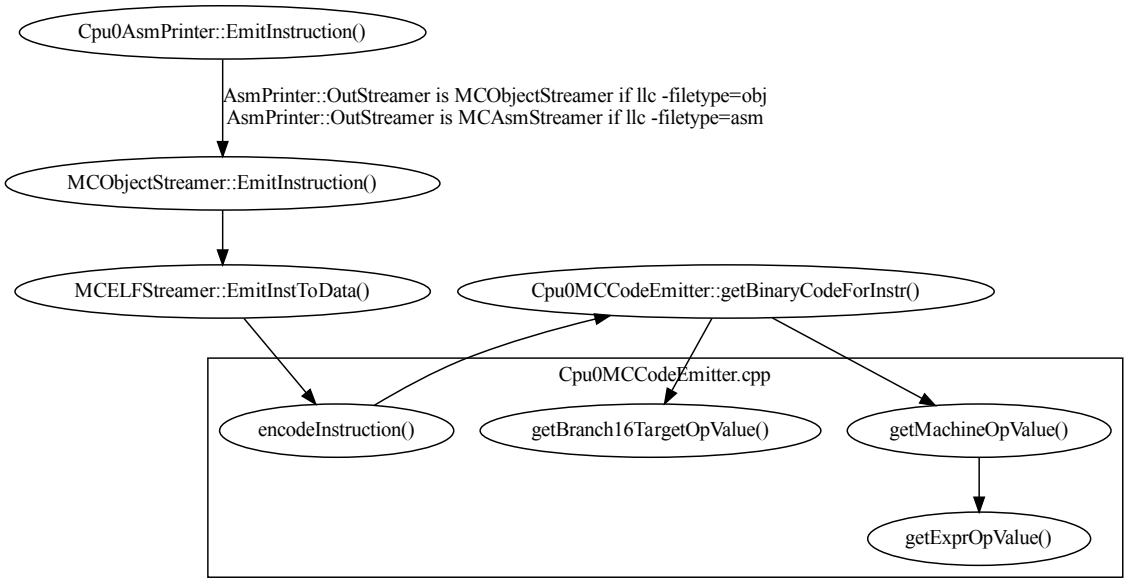


Fig. 5.1: Calling Functions of elf encoder

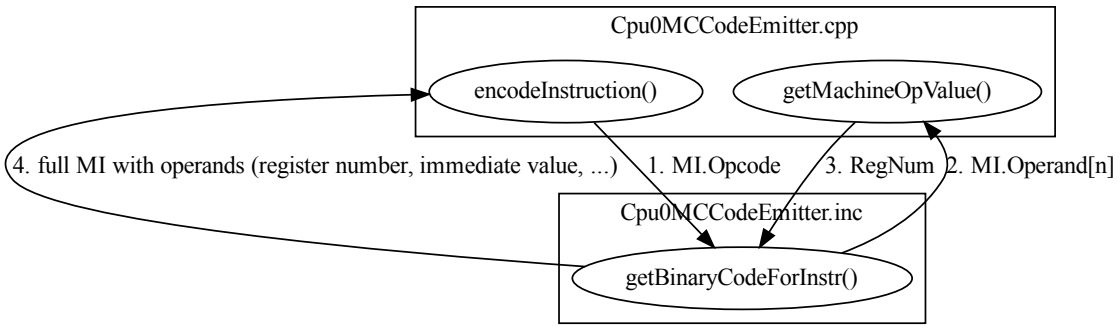


Fig. 5.2: DFD flow for instruction encode

4. getBinaryCodeForInstr() return the MI with all number of registers to encodeInstruction().

The MI.Opcode is set in Instruction Selection Stage. The table gen function getBinaryCodeForInstr() get all the operands information from the td files set by programmer as Fig. 5.3.

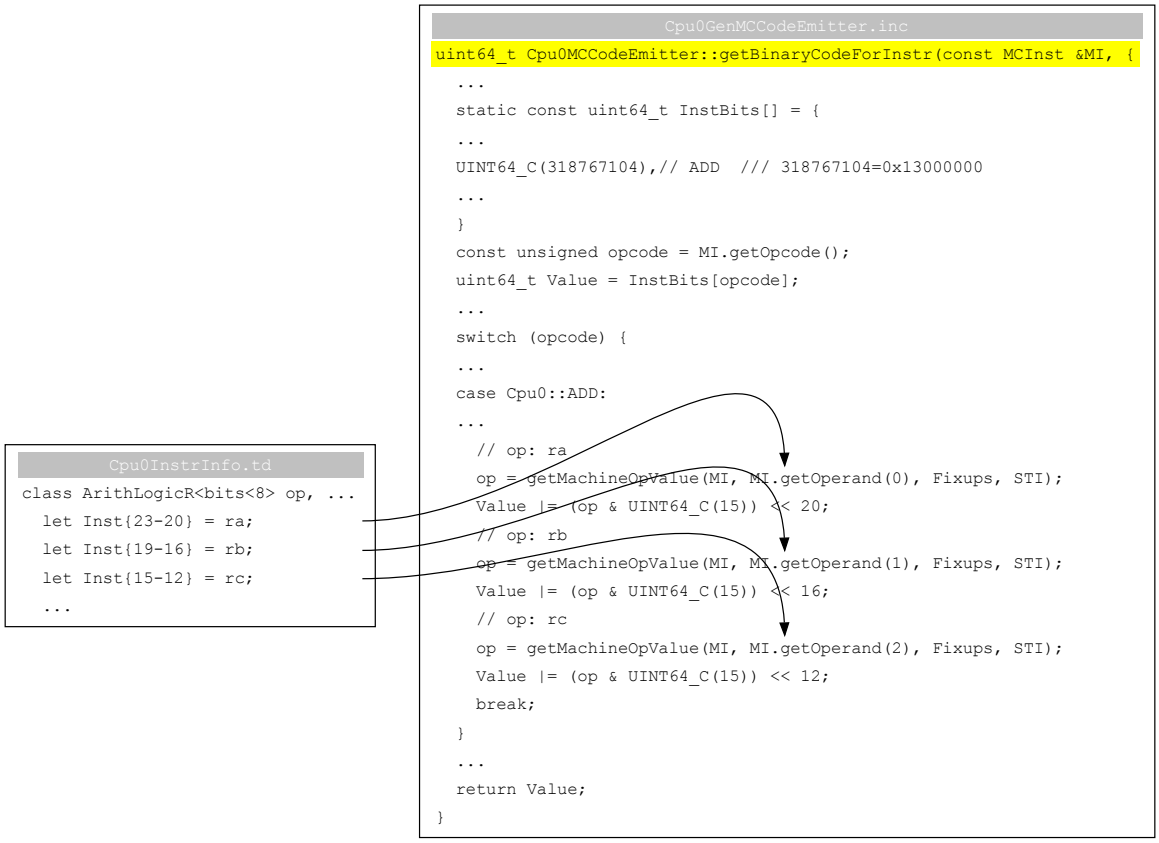


Fig. 5.3: Instruction encode, for instance: addu $v0, $at, $v1n v0:MI.getOperand(0), at:MI.getOperand(1), v1:MI.getOperand(2)

For instance, Cpu0 backend will generate "addu $v0, $at, $v1" for the IR "%0 = add %1, %2" once llvm allocate registers $v0, $at and $v1 for Operands %0, %1 and %2 individually. The MCOperand structure for MI.Operands[] include register number set in the pass of llvm allocate registers which can be got in getMachineOpValue().

The getEncodingValue(Reg) in getMachineOpValue() as the following will get the RegNo of encode from Register name such as AT, V0, or V1, … by using table gen information from Cpu0RegisterInfo.td as the following. My comment is after "///".

include//llvm/MC/MCRegisterInfo.h

```
void InitMCRegisterInfo(...,
                        const uint16_t *RET) {
  ...
  RegEncodingTable = RET;
}

unsigned Cpu0MCCodeEmitter::
getMachineOpValue(const MCInst &MI, const MCOperand &MO,
                  SmallVectorImpl<MCFixup> &Fixups,
                  const MCSubtargetInfo &STI) const {
  if (MO.isReg()) {
    unsigned Reg = MO.getReg();
    unsigned RegNo = Ctx.getRegisterInfo()->getEncodingValue(Reg);
    return RegNo;
  ...
}
```

include/llvm/MC/MCRegisterInfo.h

```
void InitMCRegisterInfo(...,
                        const uint16_t *RET) {
  ...
  RegEncodingTable = RET;
}

 /// \brief Returns the encoding for RegNo
uint16_t getEncodingValue(unsigned RegNo) const {
  assert(RegNo < NumRegs &&
         "Attempting to get encoding for invalid register number!");
  return RegEncodingTable[RegNo];
}
```

lbdex/chapters/Chapter5_1/Cpu0RegisterInfo.td

```
let Namespace = "Cpu0" in {
  ...
  def AT   : Cpu0GPRReg<1,  "1">,    DwarfRegNum<[1]>;
  def V0   : Cpu0GPRReg<2,  "2">,    DwarfRegNum<[2]>;
  def V1   : Cpu0GPRReg<3,  "3">,    DwarfRegNum<[3]>;
  ...
}
```

build/lib/Target/Cpu0/Cpu0GenRegisterInfo.inc

```
namespace Cpu0 {
enum {
  NoRegister,
  AT = 1,
  ...
  V0 = 19,
  V1 = 20,
  NUM_TARGET_REGS       // 21
};
} // end namespace Cpu0

extern const uint16_t Cpu0RegEncodingTable[] = {
  0,
  1,     /// 1, AT
  1,
  12,
  11,
  0,
  0,
  14,
  0,
  13,
  15,
  0,
  4,
  5,
  9,
  10,
  7,
  8,
  6,
  2,    /// 19, V0
  3,    /// 20, V1
};

static inline void InitCpu0MCRegisterInfo(MCRegisterInfo *RI, ...) {
  RI->InitMCRegisterInfo(..., Cpu0RegEncodingTable);
```

The applyFixup() of Cpu0AsmBackend.cpp will fix up the **jeq**, **jub**, ... instructions of "address control flow statements" or "function call statements" used in later chapters. The setting of true or false for each relocation record in needsRelocateWithSymbol() of Cpu0ELFObjectWriter.cpp depends on whethor this relocation record is needed to adjust address value during link or not. If set true, then linker has chance to adjust this address value with correct information. On the other hand, if set false, then linker has no correct information to adjust this relocation record. About relocation record, it will be introduced in later chapter ELF Support.

When emit elf obj format instruction, the EncodeInstruction() of Cpu0MCCodeEmitter.cpp will be called since it override the same name of function in parent class MCCodeEmitter.

lbdex/chapters/Chapter2/Cpu0InstrInfo.td

```
// Address operand
def mem : Operand<iPTR> {
  let PrintMethod = "printMemOperand";
  let MIOperandInfo = (ops GPROut, simm16);
  let EncoderMethod = "getMemEncoding";
```

```
}
```

```
class LoadM32<bits<8> op, string instr_asm, PatFrag OpNode,
              bit Pseudo = 0>
  : LoadM<op, instr_asm, OpNode, GPROut, mem, Pseudo> {
}

// 32-bit store.
class StoreM32<bits<8> op, string instr_asm, PatFrag OpNode,
               bit Pseudo = 0>
  : StoreM<op, instr_asm, OpNode, GPROut, mem, Pseudo> {
}
```

The "let EncoderMethod = "getMemEncoding";" in Cpu0InstrInfo.td as above will making llvm call function getMemEncoding() when either **ld** or **st** instruction is issued in elf obj since these two instructions use **mem** Operand.

The other functions in Cpu0MCCodeEmitter.cpp are called by these two functions.

After encoder, the following code will write the encode instructions to buffer.

src/lib/MC/MCELFStreamer.cpp

```
void MCELFStreamer::EmitInstToData(const MCInst &Inst,
                                   const MCSubtargetInfo &STI) {
  ...
  DF->setHasInstructions(true);
  DF->getContents().append(Code.begin(), Code.end());
  ...
}
```

Then, ELFObjectWriter::writeObject() will write the buffer to elf file.

5.4 Backend Target Registration Structure

Now, let's examine Cpu0MCTargetDesc.cpp. Cpu0MCTargetDesc.cpp do the target registration as mentioned in the previous chapter here[1], and the assembly output has explained here[2]. List the register functions of ELF obj output as follows,

[1] http://jonathan2251.github.io/lbd/llvmstructure.html#target-registration
[2] http://jonathan2251.github.io/lbd/backendstructure.html#add-asmprinter

Register function of elf streamer

```
// Register the elf streamer.
TargetRegistry::RegisterELFStreamer(*T, createMCStreamer);

  static MCStreamer *createMCStreamer(const Triple &TT, MCContext &Context,
                                      MCAsmBackend &MAB, raw_pwrite_stream &OS,
                                      MCCodeEmitter *Emitter, bool RelaxAll) {
    return createELFStreamer(Context, MAB, OS, Emitter, RelaxAll);
  }

  // MCELFStreamer.cpp
  MCStreamer *llvm::createELFStreamer(MCContext &Context, MCAsmBackend &MAB,
                                      raw_pwrite_stream &OS, MCCodeEmitter *CE,
                                      bool RelaxAll) {
    MCELFStreamer *S = new MCELFStreamer(Context, MAB, OS, CE);
    if (RelaxAll)
      S->getAssembler().setRelaxAll(true);
    return S;
  }
```

Above createELFStreamer takes care the elf obj streamer. Fig. 5.4 as follow is MCELFStreamer inheritance tree. You can find a lot of operations in that inheritance tree.

Register function of asm target streamer

```
// Register the asm target streamer.
TargetRegistry::RegisterAsmTargetStreamer(*T, createCpu0AsmTargetStreamer);

  static MCTargetStreamer *createCpu0AsmTargetStreamer(MCStreamer &S,
                                                       formatted_raw_ostream &OS,
                                                       MCInstPrinter *InstPrint,
                                                       bool isVerboseAsm) {
    return new Cpu0TargetAsmStreamer(S, OS);
  }

    // Cpu0TargetStreamer.h
    class Cpu0TargetStreamer : public MCTargetStreamer {
    public:
      Cpu0TargetStreamer(MCStreamer &S);
    };

    // This part is for ascii assembly output
    class Cpu0TargetAsmStreamer : public Cpu0TargetStreamer {
      formatted_raw_ostream &OS;

    public:
      Cpu0TargetAsmStreamer(MCStreamer &S, formatted_raw_ostream &OS);
    };
```

Above instancing MCTargetStreamer instance.

Fig. 5.4: MCELFStreamer inherit tree

Register function of MC Code Emitter

```
// Register the MC Code Emitter
TargetRegistry::RegisterMCCodeEmitter(TheCpu0Target,
                                      createCpu0MCCodeEmitterEB);
TargetRegistry::RegisterMCCodeEmitter(TheCpu0elTarget,
                                      createCpu0MCCodeEmitterEL);

  // Cpu0MCCodeEmitter.cpp
  MCCodeEmitter *llvm::createCpu0MCCodeEmitterEB(const MCInstrInfo &MCII,
                                                 const MCRegisterInfo &MRI,
                                                 MCContext &Ctx) {
    return new Cpu0MCCodeEmitter(MCII, Ctx, false);
  }

  MCCodeEmitter *llvm::createCpu0MCCodeEmitterEL(const MCInstrInfo &MCII,
                                                 const MCRegisterInfo &MRI,
                                                 MCContext &Ctx) {
    return new Cpu0MCCodeEmitter(MCII, Ctx, true);
  }
```

Above instancing two objects Cpu0MCCodeEmitter, one is for big endian and the other is for little endian. They take care the obj format generated while RegisterELFStreamer() reuse the elf streamer class.

Reader maybe has the question: "What are the actual arguments in createCpu0MCCodeEmitterEB(const MCInstrInfo &MCII, const MCSubtargetInfo &STI, MCContext &Ctx)?" and "When they are assigned?" Yes, we didn't assign it at this point, we register the createXXX() function by function pointer only (according C, TargetRegistry::RegisterXXX(TheCpu0Target, createXXX()) where createXXX is function pointer). LLVM keeps a function pointer to createXXX() when we call target registry, and will call these createXXX() function back at proper time with arguments assigned during the target registration process, RegisterXXX().

Register function of asm backend

```
// Register the asm backend.
TargetRegistry::RegisterMCAsmBackend(TheCpu0Target,
                                     createCpu0AsmBackendEB32);
TargetRegistry::RegisterMCAsmBackend(TheCpu0elTarget,
                                     createCpu0AsmBackendEL32);

  // Cpu0AsmBackend.cpp
  MCAsmBackend *llvm::createCpu0AsmBackendEL32(const Target &T,
                                               const MCRegisterInfo &MRI,
                                               const Triple &TT, StringRef CPU) {
    return new Cpu0AsmBackend(T, TT.getOS(), /*IsLittle*/true);
  }

  MCAsmBackend *llvm::createCpu0AsmBackendEB32(const Target &T,
                                               const MCRegisterInfo &MRI,
                                               const Triple &TT, StringRef CPU) {
    return new Cpu0AsmBackend(T, TT.getOS(), /*IsLittle*/false);
  }
```

(continues on next page)

(continued from previous page)

```
// Cpu0AsmBackend.h
class Cpu0AsmBackend : public MCAsmBackend {
...
}
```

Above Cpu0AsmBackend class is the bridge for asm to obj. Two objects take care big endian and little endian, respectively. It derived from MCAsmBackend. Most of code for object file generated is implemented by MCELFStreamer and it's parent, MCAsmBackend.

CHAPTER

SIX

GLOBAL VARIABLES

- *Cpu0 global variable options*
- *Static mode*
  - *data or bss*
  - *sdata or sbss*
- *pic mode*
  - *sdata or sbss*
  - *data or bss*
- *Global variable print support*
- *Summary*

In the last three chapters, we only access the local variables. This chapter deals global variable access translation.

The global variable DAG translation is different from the previous DAG translations until now we have. It creates IR DAG nodes at run time in backend C++ code according the `llc -relocation-model` option while the others of DAG just do IR DAG to Machine DAG translation directly according the input file of IR DAGs (except the Pseudo instruction RetLR used in Chapter3_4). Readers should focus on how to add code for creating DAG nodes at run time and how to define the pattern match in td for the run time created DAG nodes. In addition, the machine instruction printing function for global variable related assembly directive (macro) should be cared if your backend has it.

Chapter6_1/ supports the global variable, let's compile ch6_1.cpp with this version first, then explain the code changes after that.

lbdex/input/ch6_1.cpp

```
int gStart = 3;
int gI = 100;
int test_global()
{
  int c = 0;

  c = gI;

  return c;
}
```

```
118-165-78-166:input Jonathan$ llvm-dis ch6_1.bc -o -
...
@gStart = global i32 2, align 4
@gI = global i32 100, align 4

define i32 @_Z3funv() nounwind uwtable ssp {
  %1 = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 0, i32* %1
  store i32 0, i32* %c, align 4
  %2 = load i32* @gI, align 4
  store i32 %2, i32* %c, align 4
  %3 = load i32* %c, align 4
  ret i32 %3
}
```

6.1 Cpu0 global variable options

Just like Mips, Cpu0 supports both static and pic mode. There are two different layout of global variables for static mode which controlled by option cpu0-use-small-section. Chapter6_1/ supports the global variable translation. Let's run Chapter6_1/ with ch6_1.cpp via four different options `llc -relocation-model=static -cpu0-use-small-section=false`, `llc -relocation-model=static -cpu0-use-small-section=true`, `llc -relocation-model=pic -cpu0-use-small-section=false` and `llc -relocation-model=pic -cpu0-use-small-section=true` to tracing the DAGs and Cpu0 instructions.

```
118-165-78-166:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch6_1.cpp -emit-llvm -o ch6_1.bc
118-165-78-166:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -cpu0-use-small-section=false
-filetype=asm -debug ch6_1.bc -o -

...
Type-legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 12 nodes:
  ...
      0x7ffd5902cc10: <multiple use>
    0x7ffd5902cf10: ch = store 0x7ffd5902cd10, 0x7ffd5902ca10, 0x7ffd5902ce10,
    0x7ffd5902cc10<ST4[%c]> [ORD=2] [ID=-3]

    0x7ffd5902d010: i32 = GlobalAddress<i32* @gI> 0 [ORD=3] [ID=-3]

    0x7ffd5902cc10: <multiple use>
  0x7ffd5902d110: i32,ch = load 0x7ffd5902cf10, 0x7ffd5902d010,
  0x7ffd5902cc10<LD4[@gI]> [ORD=3] [ID=-3]
  ...

Legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 16 nodes:
  ...
      0x7ffd5902cc10: <multiple use>
    0x7ffd5902cf10: ch = store 0x7ffd5902cd10, 0x7ffd5902ca10, 0x7ffd5902ce10,
```

(continues on next page)

(continued from previous page)

```
    0x7ffd5902cc10<ST4[%c]> [ORD=2] [ID=8]

        0x7ffd5902d310: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=5]

      0x7ffd5902d710: i32 = Cpu0ISD::Hi 0x7ffd5902d310

        0x7ffd5902d610: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=6]

      0x7ffd5902d810: i32 = Cpu0ISD::Lo 0x7ffd5902d610

    0x7ffd5902fe10: i32 = add 0x7ffd5902d710, 0x7ffd5902d810

    0x7ffd5902cc10: <multiple use>
  0x7ffd5902d110: i32,ch = load 0x7ffd5902cf10, 0x7ffd5902fe10,
  0x7ffd5902cc10<LD4[@gI]> [ORD=3] [ID=9]
  ...
  lui $2, %hi(gI)
  ori $2, $2, %lo(gI)
      ld      $2, 0($2)
      ...
      .type   gStart,@object          # @gStart
      .data
      .globl  gStart
      .align  2
gStart:
      .4byte  2                       # 0x2
      .size   gStart, 4

      .type   gI,@object              # @gI
      .globl  gI
      .align  2
gI:
      .4byte  100                     # 0x64
      .size   gI, 4
```

```
118-165-78-166:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -cpu0-use-small-section=true
-filetype=asm -debug ch6_1.bc -o -

...
Type-legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 12 nodes:
  ...
      0x7fc5f382cc10: <multiple use>
    0x7fc5f382cf10: ch = store 0x7fc5f382cd10, 0x7fc5f382ca10, 0x7fc5f382ce10,
    0x7fc5f382cc10<ST4[%c]> [ORD=2] [ID=-3]

    0x7fc5f382d010: i32 = GlobalAddress<i32* @gI> 0 [ORD=3] [ID=-3]

    0x7fc5f382cc10: <multiple use>
  0x7fc5f382d110: i32,ch = load 0x7fc5f382cf10, 0x7fc5f382d010,
  0x7fc5f382cc10<LD4[@gI]> [ORD=3] [ID=-3]
```

(continues on next page)

(continued from previous page)

```
  ...
Legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 15 nodes:
  ...
      0x7fc5f382cc10: <multiple use>
    0x7fc5f382cf10: ch = store 0x7fc5f382cd10, 0x7fc5f382ca10, 0x7fc5f382ce10,
    0x7fc5f382cc10<ST4[%c]> [ORD=2] [ID=8]

      0x7fc5f382d710: i32 = register %GP

        0x7fc5f382d310: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=4]

      0x7fc5f382d610: i32 = Cpu0ISD::GPRel 0x7fc5f382d310

    0x7fc5f382d810: i32 = add 0x7fc5f382d710, 0x7fc5f382d610

    0x7fc5f382cc10: <multiple use>
  0x7fc5f382d110: i32,ch = load 0x7fc5f382cf10, 0x7fc5f382d810,
  0x7fc5f382cc10<LD4[@gI]> [ORD=3] [ID=9]
  ...

      ori     $2, $gp, %gp_rel(gI)
      ld      $2, 0($2)
      ...
      .type   gStart,@object          # @gStart
      .section        .sdata,"aw",@progbits
      .globl  gStart
      .align  2
gStart:
      .4byte  2                       # 0x2
      .size   gStart, 4

      .type   gI,@object              # @gI
      .globl  gI
      .align  2
gI:
      .4byte  100                     # 0x64
      .size   gI, 4
```

```
118-165-78-166:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -cpu0-use-small-section=false
-filetype=asm -debug ch6_1.bc -o -

  ...
Type-legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 11 nodes:
  ...
      0x7fe03c02e010: <multiple use>
    0x7fe03c02e118: ch = store 0x7fe03b50dee0, 0x7fe03c02de00, 0x7fe03c02df08,
    0x7fe03c02e010<ST4[%c]> [ORD=3] [ID=-3]

    0x7fe03c02e220: i32 = GlobalAddress<i32* @gI> 0 [ORD=4] [ID=-3]
```

(continues on next page)

(continued from previous page)

```
    0x7fe03c02e010: <multiple use>
  0x7fe03c02e328: i32,ch = load 0x7fe03c02e118, 0x7fe03c02e220,
  0x7fe03c02e010<LD4[@gI]> [ORD=4] [ID=-3]
  ...
Legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 15 nodes:
  ...
      0x7fe03c02e010: <multiple use>
    0x7fe03c02e118: ch = store 0x7fe03b50dee0, 0x7fe03c02de00, 0x7fe03c02df08,
    0x7fe03c02e010<ST4[%c]> [ORD=3] [ID=6]

        0x7fe03c02e538: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=5] [ORD=4]

      0x7fe03c02ea60: i32 = Cpu0ISD::Hi 0x7fe03c02e538 [ORD=4]

        0x7fe03c02e958: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=6] [ORD=4]

      0x7fe03c02eb68: i32 = Cpu0ISD::Lo 0x7fe03c02e958 [ORD=4]

    0x7fe03c02ec70: i32 = add 0x7fe03c02ea60, 0x7fe03c02eb68 [ORD=4]

    0x7fe03c02e010: <multiple use>
  0x7fe03c02e328: i32,ch = load 0x7fe03c02e118, 0x7fe03c02ec70,
  0x7fe03c02e010<LD4[@gI]> [ORD=4] [ID=7]
  ...
        lui   $2, %got_hi(gI)
        addu  $2, $2, $gp
        ld    $2, %got_lo(gI)($2)
  ...
    .type gStart,@object          # @gStart
  .data
  .globl  gStart
  .align  2
gStart:
  .4byte  3                       # 0x3
  .size gStart, 4

  .type gI,@object              # @gI
  .globl  gI
  .align  2
gI:
  .4byte  100                     # 0x64
  .size gI, 4
```

```
118-165-78-166:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -cpu0-use-small-section=true
-filetype=asm -debug ch6_1.bc -o -

...
Type-legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 11 nodes:
```

(continues on next page)

(continued from previous page)

```
  ...
      0x7fad7102cc10: <multiple use>
    0x7fad7102cf10: ch = store 0x7fad7102cd10, 0x7fad7102ca10, 0x7fad7102ce10,
    0x7fad7102cc10<ST4[%c]> [ORD=2] [ID=-3]

    0x7fad7102d010: i32 = GlobalAddress<i32* @gI> 0 [ORD=3] [ID=-3]

    0x7fad7102cc10: <multiple use>
  0x7fad7102d110: i32,ch = load 0x7fad7102cf10, 0x7fad7102d010,
  0x7fad7102cc10<LD4[@gI]> [ORD=3] [ID=-3]
  ...
Legalized selection DAG: BB#0 '_Z11test_globalv:'
SelectionDAG has 14 nodes:
  0x7ff3c9c10b98: ch = EntryToken [ORD=1] [ID=0]
  ...
      0x7fad7102cc10: <multiple use>
    0x7fad7102cf10: ch = store 0x7fad7102cd10, 0x7fad7102ca10, 0x7fad7102ce10,
    0x7fad7102cc10<ST4[%c]> [ORD=2] [ID=8]

      0x7fad70c10b98: <multiple use>
        0x7fad7102d610: i32 = Register %GP

        0x7fad7102d310: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=1]

      0x7fad7102d710: i32 = Cpu0ISD::Wrapper 0x7fad7102d610, 0x7fad7102d310

      0x7fad7102cc10: <multiple use>
    0x7fad7102d810: i32,ch = load 0x7fad70c10b98, 0x7fad7102d710,
    0x7fad7102cc10<LD4[<unknown>]>

    0x7ff3ca02cc10: <multiple use>
  0x7ff3ca02d110: i32,ch = load 0x7ff3ca02cf10, 0x7ff3ca02d810,
  0x7ff3ca02cc10<LD4[@gI]> [ORD=3] [ID=9]
  ...
        .set  noreorder
        .cpload       $6
        .set  nomacro
  ...
      ld      $2, %got(gI)($gp)
      ld      $2, 0($2)
  ...
      .type   gStart,@object          # @gStart
      .data
      .globl  gStart
      .align  2
gStart:
      .4byte  2                       # 0x2
      .size   gStart, 4

      .type   gI,@object              # @gI
      .globl  gI
      .align  2
```

(continues on next page)

(continued from previous page)

```
gI:
      .4byte  100                     # 0x64
      .size   gI, 4
```

Summary above information to Table: Cpu0 global variable options.

Table 6.1: Cpu0 global variable options

option name	default	other option value	discription
-relocation-model	pic	static	• pic: Postion Independent Address • static: Absolute Address
-cpu0-use-small-section	false	true	• false: .data or .bss, 32 bits addressable • true: .sdata or .sbss, 16 bits addressable

Table 6.2: Cpu0 DAGs and instructions for -relocation-model=static

option: cpu0-use-small-section	false	true
addressing mode	absolute	$gp relative
addressing	absolute	$gp+offset
Legalized selection DAG	(add Cpu0ISD::Hi<gI offset Hi16> Cpu0ISD::Lo<gI offset Lo16>)	(add register %GP, Cpu0ISD::GPRel<gI offset>)
Cpu0	lui $2, %hi(gI); ori $2, $2, %lo(gI);	ori $2, $gp, %gp_rel(gI);
relocation records solved	link time	link time

- In static, cpu0-use-small-section=true, offset between gI and .data can be calculated since the $gp is assigned at fixed address of the start of global address table.
- In "static, cpu0-use-small-section=false", the gI high and low address (%hi(gI) and %lo(gI)) are translated into absolute address.

Table 6.3: Cpu0 DAGs and instructions for -relocation-model=pic

option: cpu0-use-small-section	false	true
addressing mode	$gp relative	$gp relative
addressing	$gp+offset	$gp+offset
Legalized selection DAG	(load (Cpu0ISD::Wrapper register %GP, <gI offset>))	(load EntryToken, (Cpu0ISD::Wrapper (add Cpu0ISD::Hi<gI offset Hi16>, Register %GP), Cpu0ISD::Lo<gI offset Lo16>))
Cpu0	ld $2, %got(gI)($gp);	lui $2, %got_hi(gI); add $2, $2, $gp; ld $2, %got_lo(gI)($2);
relocation records solved	link/load time	link/load time

- In pic, offset between gI and .data cannot be calculated if the function is loaded at run time (dynamic link); the offset can be calculated if use static link.
- In C, all variable names binding staticly. In C++, the overload variable or function are binding dynamicly.

According book of system program, there are Absolute Addressing Mode and Position Independent Addressing Mode. The dynamic function must be compiled with Position Independent Addressing Mode. In general, option -relocation-model is used to generate either Absolute Addressing or Position Independent Addressing. The exception is -relocation-model=static and -cpu0-use-small-section=false. In this case, the register $gp is reserved to set at the start address of global variable area. Cpu0 uses $gp relative addressing in this mode.

To support global variable, first add **UseSmallSectionOpt** command variable to Cpu0Subtarget.cpp. After that, user can run llc with option `llc -cpu0-use-small-section=false` to specify **UseSmallSectionOpt** to false. The default of **UseSmallSectionOpt** is false if without specify it further. About the **cl::opt** command line variable, you can refer to here[1] further.

lbdex/chapters/Chapter6_1/Cpu0Subtarget.h

```
extern bool Cpu0ReserveGP;
extern bool Cpu0NoCpload;
```

```
class Cpu0Subtarget : public Cpu0GenSubtargetInfo {
  ...
```

```
  // UseSmallSection - Small section is used.
  bool UseSmallSection;
```

```
  bool useSmallSection() const { return UseSmallSection; }
```

```
  ...
};
```

lbdex/chapters/Chapter6_1/Cpu0Subtarget.cpp

```
static cl::opt<bool> UseSmallSectionOpt
                ("cpu0-use-small-section", cl::Hidden, cl::init(false),
                 cl::desc("Use small section. Only work when -relocation-model="
                 "static. pic always not use small section."));

static cl::opt<bool> ReserveGPOpt
                ("cpu0-reserve-gp", cl::Hidden, cl::init(false),
                 cl::desc("Never allocate $gp to variable"));

static cl::opt<bool> NoCploadOpt
                ("cpu0-no-cpload", cl::Hidden, cl::init(false),
                 cl::desc("No issue .cpload"));

bool Cpu0ReserveGP;
bool Cpu0NoCpload;
```

```
Cpu0Subtarget::Cpu0Subtarget(const Triple &TT, StringRef CPU,
                             StringRef FS, bool little,
                             const Cpu0TargetMachine &_TM) :
```

[1] http://llvm.org/docs/CommandLine.html

```
  // Set UseSmallSection.
  UseSmallSection = UseSmallSectionOpt;
  Cpu0ReserveGP = ReserveGPOpt;
  Cpu0NoCpload = NoCploadOpt;
```

```
  ...
}
```

The options ReserveGPOpt and NoCploadOpt will used in Cpu0 linker at later Chapter. Next add the following code to files Cpu0BaseInfo.h, Cpu0TargetObjectFile.h, Cpu0TargetObjectFile.cpp, Cpu0RegisterInfo.cpp and Cpu0ISelLowering.cpp.

lbdex/chapters/Chapter6_1/Cpu0BaseInfo.h

```
enum TOF {
  ...
  /// MO_GOT16 - Represents the offset into the global offset table at which
  /// the address the relocation entry symbol resides during execution.
  MO_GOT16,
  MO_GOT,
...
}; // enum TOF {
```

lbdex/chapters/Chapter6_1/Cpu0TargetObjectFile.h

```
    bool IsGlobalInSmallSection(const GlobalObject *GO, const TargetMachine &TM,
                                SectionKind Kind) const;
    bool IsGlobalInSmallSectionImpl(const GlobalObject *GO,
                                    const TargetMachine &TM) const;
```

lbdex/chapters/Chapter6_1/Cpu0TargetObjectFile.cpp

```
// A address must be loaded from a small section if its size is less than the
// small section size threshold. Data in this section must be addressed using
// gp_rel operator.
static bool IsInSmallSection(uint64_t Size) {
  return Size > 0 && Size <= SSThreshold;
}

bool Cpu0TargetObjectFile::IsGlobalInSmallSection(
    const GlobalObject *GO, const TargetMachine &TM) const {
  // We first check the case where global is a declaration, because finding
  // section kind using getKindForGlobal() is only allowed for global
  // definitions.
  if (GO->isDeclaration() || GO->hasAvailableExternallyLinkage())
    return IsGlobalInSmallSectionImpl(GO, TM);

  return IsGlobalInSmallSection(GO, TM, getKindForGlobal(GO, TM));
```

(continues on next page)

(continued from previous page)

```
}

/// IsGlobalInSmallSection - Return true if this global address should be
/// placed into small data/bss section.
bool Cpu0TargetObjectFile::
IsGlobalInSmallSection(const GlobalObject *GO, const TargetMachine &TM,
                       SectionKind Kind) const {
  return IsGlobalInSmallSectionImpl(GO, TM) &&
         (Kind.isData() || Kind.isBSS() || Kind.isCommon() ||
          Kind.isReadOnly());
}

/// Return true if this global address should be placed into small data/bss
/// section. This method does all the work, except for checking the section
/// kind.
bool Cpu0TargetObjectFile::
IsGlobalInSmallSectionImpl(const GlobalObject *GV,
                           const TargetMachine &TM) const {
  const Cpu0Subtarget &Subtarget =
      *static_cast<const Cpu0TargetMachine &>(TM).getSubtargetImpl();

  // Return if small section is not available.
  if (!Subtarget.useSmallSection())
    return false;

  // Only global variables, not functions.
  const GlobalVariable *GVA = dyn_cast<GlobalVariable>(GV);
  if (!GVA)
    return false;

  Type *Ty = GV->getValueType();
  return IsInSmallSection(
      GV->getParent()->getDataLayout().getTypeAllocSize(Ty));
}


MCSection *
Cpu0TargetObjectFile::SelectSectionForGlobal(
    const GlobalObject *GO, SectionKind Kind, const TargetMachine &TM) const {
  // TODO: Could also support "weak" symbols as well with ".gnu.linkonce.s.*"
  // sections?

  // Handle Small Section classification here.
  if (Kind.isBSS() && IsGlobalInSmallSection(GO, TM, Kind))
    return SmallBSSSection;
  if (Kind.isData() && IsGlobalInSmallSection(GO, TM, Kind))
    return SmallDataSection;
  if (Kind.isReadOnly() && IsGlobalInSmallSection(GO, TM, Kind))
    return SmallDataSection;

  // Otherwise, we work the same as ELF.
  return TargetLoweringObjectFileELF::SelectSectionForGlobal(GO, Kind, TM);
```

(continues on next page)

(continued from previous page)

```
}
```

lbdex/chapters/Chapter6_1/Cpu0RegisterInfo.cpp

```
BitVector Cpu0RegisterInfo::
getReservedRegs(const MachineFunction &MF) const {
  ...
    Reserved.set(Cpu0::GP);
  ...
}
```

lbdex/chapters/Chapter6_1/Cpu0ISelLowering.h

```
    SDValue getGlobalReg(SelectionDAG &DAG, EVT Ty) const;

    // This method creates the following nodes, which are necessary for
    // computing a local symbol's address:
    //
    // (add (load (wrapper $gp, %got(sym)), %lo(sym))
    template<class NodeTy>
    SDValue getAddrLocal(NodeTy *N, EVT Ty, SelectionDAG &DAG) const {
      SDLoc DL(N);
      unsigned GOTFlag = Cpu0II::MO_GOT;
      SDValue GOT = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, getGlobalReg(DAG, Ty),
                                getTargetNode(N, Ty, DAG, GOTFlag));
      SDValue Load =
          DAG.getLoad(Ty, DL, DAG.getEntryNode(), GOT,
                      MachinePointerInfo::getGOT(DAG.getMachineFunction()));
      unsigned LoFlag = Cpu0II::MO_ABS_LO;
      SDValue Lo = DAG.getNode(Cpu0ISD::Lo, DL, Ty,
                               getTargetNode(N, Ty, DAG, LoFlag));
      return DAG.getNode(ISD::ADD, DL, Ty, Load, Lo);
    }

    //@getAddrGlobal {
    // This method creates the following nodes, which are necessary for
    // computing a global symbol's address:
    //
    // (load (wrapper $gp, %got(sym)))
    template<class NodeTy>
    SDValue getAddrGlobal(NodeTy *N, EVT Ty, SelectionDAG &DAG,
                          unsigned Flag, SDValue Chain,
                          const MachinePointerInfo &PtrInfo) const {
      SDLoc DL(N);
      SDValue Tgt = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, getGlobalReg(DAG, Ty),
                                getTargetNode(N, Ty, DAG, Flag));
      return DAG.getLoad(Ty, DL, Chain, Tgt, PtrInfo);
    }
```

(continues on next page)

(continued from previous page)

```
    //@getAddrGlobal }

    //@getAddrGlobalLargeGOT {
    // This method creates the following nodes, which are necessary for
    // computing a global symbol's address in large-GOT mode:
    //
    // (load (wrapper (add %hi(sym), $gp), %lo(sym)))
    template<class NodeTy>
    SDValue getAddrGlobalLargeGOT(NodeTy *N, EVT Ty, SelectionDAG &DAG,
                                  unsigned HiFlag, unsigned LoFlag,
                                  SDValue Chain,
                                  const MachinePointerInfo &PtrInfo) const {
      SDLoc DL(N);
      SDValue Hi = DAG.getNode(Cpu0ISD::Hi, DL, Ty,
                               getTargetNode(N, Ty, DAG, HiFlag));
      Hi = DAG.getNode(ISD::ADD, DL, Ty, Hi, getGlobalReg(DAG, Ty));
      SDValue Wrapper = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, Hi,
                                    getTargetNode(N, Ty, DAG, LoFlag));
      return DAG.getLoad(Ty, DL, Chain, Wrapper, PtrInfo);
    }
    //@getAddrGlobalLargeGOT }

    //@getAddrNonPIC
    // This method creates the following nodes, which are necessary for
    // computing a symbol's address in non-PIC mode:
    //
    // (add %hi(sym), %lo(sym))
    template<class NodeTy>
    SDValue getAddrNonPIC(NodeTy *N, EVT Ty, SelectionDAG &DAG) const {
      SDLoc DL(N);
      SDValue Hi = getTargetNode(N, Ty, DAG, Cpu0II::MO_ABS_HI);
      SDValue Lo = getTargetNode(N, Ty, DAG, Cpu0II::MO_ABS_LO);
      return DAG.getNode(ISD::ADD, DL, Ty,
                         DAG.getNode(Cpu0ISD::Hi, DL, Ty, Hi),
                         DAG.getNode(Cpu0ISD::Lo, DL, Ty, Lo));
    }
```

lbdex/chapters/Chapter6_1/Cpu0ISelLowering.cpp

```
SDValue Cpu0TargetLowering::getGlobalReg(SelectionDAG &DAG, EVT Ty) const {
  Cpu0FunctionInfo *FI = DAG.getMachineFunction().getInfo<Cpu0FunctionInfo>();
  return DAG.getRegister(FI->getGlobalBaseReg(), Ty);
}

//@getTargetNode(GlobalAddressSDNode
SDValue Cpu0TargetLowering::getTargetNode(GlobalAddressSDNode *N, EVT Ty,
                                          SelectionDAG &DAG,
                                          unsigned Flag) const {
  return DAG.getTargetGlobalAddress(N->getGlobal(), SDLoc(N), Ty, 0, Flag);
}
```

(continues on next page)

(continued from previous page)

```
//@getTargetNode(ExternalSymbolSDNode
SDValue Cpu0TargetLowering::getTargetNode(ExternalSymbolSDNode *N, EVT Ty,
                                          SelectionDAG &DAG,
                                          unsigned Flag) const {
  return DAG.getTargetExternalSymbol(N->getSymbol(), Ty, Flag);
}
```

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
  setOperationAction(ISD::GlobalAddress,      MVT::i32,   Custom);
```

```
}
```

```
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
  switch (Op.getOpcode())
  {
```

```
  case ISD::GlobalAddress:      return lowerGlobalAddress(Op, DAG);
```

```
  }
  return SDValue();
}
```

```
SDValue Cpu0TargetLowering::lowerGlobalAddress(SDValue Op,
                                               SelectionDAG &DAG) const {
  //@lowerGlobalAddress }
  SDLoc DL(Op);
  const Cpu0TargetObjectFile *TLOF =
        static_cast<const Cpu0TargetObjectFile *>(
            getTargetMachine().getObjFileLowering());
  //@lga 1 {
  EVT Ty = Op.getValueType();
  GlobalAddressSDNode *N = cast<GlobalAddressSDNode>(Op);
  const GlobalValue *GV = N->getGlobal();
  //@lga 1 }

  if (!isPositionIndependent()) {
    //@ %gp_rel relocation
    const GlobalObject *GO = GV->getBaseObject();
    if (GO && TLOF->IsGlobalInSmallSection(GO, getTargetMachine())) {
      SDValue GA = DAG.getTargetGlobalAddress(GV, DL, MVT::i32, 0,
                                              Cpu0II::MO_GPREL);
      SDValue GPRelNode = DAG.getNode(Cpu0ISD::GPRel, DL,
                                      DAG.getVTList(MVT::i32), GA);
```

(continues on next page)

(continued from previous page)

```
      SDValue GPReg = DAG.getRegister(Cpu0::GP, MVT::i32);
      return DAG.getNode(ISD::ADD, DL, MVT::i32, GPReg, GPRelNode);
    }

    //@ %hi/%lo relocation
    return getAddrNonPIC(N, Ty, DAG);
  }

  if (GV->hasInternalLinkage() || (GV->hasLocalLinkage() && !isa<Function>(GV)))
    return getAddrLocal(N, Ty, DAG);

  //@large section
  const GlobalObject *GO = GV->getBaseObject();
  if (GO && !TLOF->IsGlobalInSmallSection(GO, getTargetMachine()))
    return getAddrGlobalLargeGOT(
        N, Ty, DAG, Cpu0II::MO_GOT_HI16, Cpu0II::MO_GOT_LO16,
        DAG.getEntryNode(),
        MachinePointerInfo::getGOT(DAG.getMachineFunction()));
  return getAddrGlobal(
      N, Ty, DAG, Cpu0II::MO_GOT, DAG.getEntryNode(),
      MachinePointerInfo::getGOT(DAG.getMachineFunction()));
}
```

The setOperationAction(ISD::GlobalAddress, MVT::i32, Custom) tells `llc` that we implement global address operation in C++ function Cpu0TargetLowering::LowerOperation(). LLVM will call this function only when llvm want to translate IR DAG of loading global variable into machine code. Although all the Custom type of IR operations set by setOperationAction(ISD::XXX, MVT::XXX, Custom) in construction function Cpu0TargetLowering() will invoke llvm to call Cpu0TargetLowering::LowerOperation() in stage "Legalized selection DAG", the global address access operation can be identified by checking whether the opcode of DAG Node is ISD::GlobalAddress or not, furthmore.

Finally, add the following code in Cpu0ISelDAGToDAG.cpp and Cpu0InstrInfo.td.

lbdex/chapters/Chapter6_1/Cpu0ISelDAGToDAG.h

```
  SDNode *getGlobalBaseReg();
```

lbdex/chapters/Chapter6_1/Cpu0ISelDAGToDAG.cpp

```
/// getGlobalBaseReg - Output the instructions required to put the
/// GOT address into a register.
SDNode *Cpu0DAGToDAGISel::getGlobalBaseReg() {
  unsigned GlobalBaseReg = MF->getInfo<Cpu0FunctionInfo>()->getGlobalBaseReg();
  return CurDAG->getRegister(GlobalBaseReg, getTargetLowering()->getPointerTy(
                                              CurDAG->getDataLayout()))
      .getNode();
}
```

```
/// ComplexPattern used on Cpu0InstrInfo
/// Used on Cpu0 Load/Store instructions
```

(continues on next page)

(continued from previous page)

```
bool Cpu0DAGToDAGISel::
SelectAddr(SDNode *Parent, SDValue Addr, SDValue &Base, SDValue &Offset) {
```

```
  // on PIC code Load GA
  if (Addr.getOpcode() == Cpu0ISD::Wrapper) {
    Base   = Addr.getOperand(0);
    Offset = Addr.getOperand(1);
    return true;
  }

  //@static
  if (TM.getRelocationModel() != Reloc::PIC_) {
    if ((Addr.getOpcode() == ISD::TargetExternalSymbol ||
        Addr.getOpcode() == ISD::TargetGlobalAddress))
      return false;
  }
```

```
  ...
}
```

```
/// Select instructions not customized! Used for
/// expanded, promoted and normal instructions
void Cpu0DAGToDAGISel::Select(SDNode *Node) {
```

```
  // Get target GOT address.
  case ISD::GLOBAL_OFFSET_TABLE:
    ReplaceNode(Node, getGlobalBaseReg());
    return;
```

```
  ...
}
```

lbdex/chapters/Chapter6_1/Cpu0InstrInfo.td

```
// Hi and Lo nodes are used to handle global addresses. Used on
// Cpu0ISelLowering to lower stuff like GlobalAddress, ExternalSymbol
// static model. (nothing to do with Cpu0 Registers Hi and Lo)
def Cpu0Hi    : SDNode<"Cpu0ISD::Hi", SDTIntUnaryOp>;
def Cpu0Lo    : SDNode<"Cpu0ISD::Lo", SDTIntUnaryOp>;
def Cpu0GPRel : SDNode<"Cpu0ISD::GPRel", SDTIntUnaryOp>;
```

```
def Cpu0Wrapper    : SDNode<"Cpu0ISD::Wrapper", SDTIntBinOp>;
```

```
def RelocPIC    :     Predicate<"TM.getRelocationModel() == Reloc::PIC_">;
```

```
// hi/lo relocs
let Predicates = [Ch6_1] in {
def : Pat<(Cpu0Hi tglobaladdr:$in), (LUi tglobaladdr:$in)>;
}
```

```
let Predicates = [Ch6_1] in {
def : Pat<(Cpu0Lo tglobaladdr:$in), (ORi ZERO, tglobaladdr:$in)>;
}
```

```
let Predicates = [Ch6_1] in {
def : Pat<(add CPURegs:$hi, (Cpu0Lo tglobaladdr:$lo)),
          (ORi CPURegs:$hi, tglobaladdr:$lo)>;
}
```

```
// gp_rel relocs
let Predicates = [Ch6_1] in {
def : Pat<(add CPURegs:$gp, (Cpu0GPRel tglobaladdr:$in)),
          (ORi CPURegs:$gp, tglobaladdr:$in)>;
}

//@ wrapper_pic
let Predicates = [Ch6_1] in {
class WrapperPat<SDNode node, Instruction ORiOp, RegisterClass RC>:
      Pat<(Cpu0Wrapper RC:$gp, node:$in),
              (ORiOp RC:$gp, node:$in)>;

def : WrapperPat<tglobaladdr, ORi, GPROut>;
}
```

6.2 Static mode

From Table: Cpu0 global variable options, option cpu0-use-small-section=false puts the global varibale in data/bss while cpu0-use-small-section=true puts in sdata/sbss. The sdata stands for small data area. Section data and sdata are areas for global variables with initial value (such as int gI = 100 in this example) while Section bss and sbss are areas for global variables without initial value (for instance, int gI;).

6.2.1 data or bss

The data/bss are 32 bits addressable areas since Cpu0 is a 32 bits architecture. Option cpu0-use-small-section=false will generate the following instructions.

```
  ...
  lui $2, %hi(gI)
  ori $2, $2, %lo(gI)
  ld  $2, 0($2)
  ...
  .type       gStart,@object          # @gStart
  .data
  .globl      gStart
  .align      2
gStart:
  .4byte      2                       # 0x2
  .size       gStart, 4
```

(continues on next page)

(continued from previous page)

```
  .type       gI,@object              # @gI
  .globl      gI
  .align      2
gI:
  .4byte      100                     # 0x64
  .size       gI, 4
```

As above code, it loads the high address part of gI PC relative address (16 bits) to register $2 and shift 16 bits. Now, the register $2 got it's high part of gI absolute address. Next, it adds register $2 and low part of gI absolute address into $2. At this point, it gets the gI memory address. Finally, it gets the gI content by instruction "ld $2, 0($2)". The `llc -relocation-model=static` is for absolute address mode which must be used in static link mode. The dynamic link must be encoded with Position Independent Addressing. As you can see, the PC relative address can be solved in static link (The offset between the address of gI and instruction "lui $2, %hi(gI)" can be caculated). Since Cpu0 uses PC relative address coding, this program can be loaded to any address and run correctly there. If this program uses absolute address and can be loaded at a specific address known at link stage, the relocation record of gI variable access instruction such as "lui $2, %hi(gI)" and "ori $2, $2, %lo(gI)" can be solved at link time. On the other hand, if this program use absolute address and the loading address is known at load time, then this relocation record will be solved by loader at load time.

IsGlobalInSmallSection() returns true or false depends on UseSmallSectionOpt.

The code fragment of lowerGlobalAddress() as the following corresponding option `llc -relocation-model=static -cpu0-use-small-section=false` will translate DAG (GlobalAddress<i32* @gI> 0) into (add Cpu0ISD::Hi<gI offset Hi16> Cpu0ISD::Lo<gI offset Lo16>) in stage "Legalized selection DAG" as below.

lbdex/chapters/Chapter6_1/Cpu0ISelLowering.h

```
    // This method creates the following nodes, which are necessary for
    // computing a symbol's address in non-PIC mode:
    //
    // (add %hi(sym), %lo(sym))
    template<class NodeTy>
    SDValue getAddrNonPIC(NodeTy *N, EVT Ty, SelectionDAG &DAG) const {
      SDLoc DL(N);
      SDValue Hi = getTargetNode(N, Ty, DAG, Cpu0II::MO_ABS_HI);
      SDValue Lo = getTargetNode(N, Ty, DAG, Cpu0II::MO_ABS_LO);
      return DAG.getNode(ISD::ADD, DL, Ty,
                         DAG.getNode(Cpu0ISD::Hi, DL, Ty, Hi),
                         DAG.getNode(Cpu0ISD::Lo, DL, Ty, Lo));
    }
```

lbdex/chapters/Chapter6_1/Cpu0ISelLowering.cpp

```
SDValue Cpu0TargetLowering::getTargetNode(GlobalAddressSDNode *N, EVT Ty,
                                          SelectionDAG &DAG,
                                          unsigned Flag) const {
  return DAG.getTargetGlobalAddress(N->getGlobal(), SDLoc(N), Ty, 0, Flag);
}
```

```
SDValue Cpu0TargetLowering::lowerGlobalAddress(SDValue Op,
                                               SelectionDAG &DAG) const {
  ...
  EVT Ty = Op.getValueType();
  GlobalAddressSDNode *N = cast<GlobalAddressSDNode>(Op);
  ..

  if (getTargetMachine().getRelocationModel() != Reloc::PIC_) {
    ...
    // %hi/%lo relocation
    return getAddrNonPIC(N, Ty, DAG);
  }
  ...
}
```

```
118-165-78-166:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch6_1.cpp -emit-llvm -o ch6_1.bc
118-165-78-166:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=static -cpu0-use-small-section=false
-filetype=asm -debug ch6_1.bc -o -

...
Type-legalized selection DAG: BB#0 '_Z3funv:entry'
SelectionDAG has 12 nodes:
  ...
      0x7ffd5902cc10: <multiple use>
    0x7ffd5902cf10: ch = store 0x7ffd5902cd10, 0x7ffd5902ca10, 0x7ffd5902ce10,
    0x7ffd5902cc10<ST4[%c]> [ORD=2] [ID=-3]

    0x7ffd5902d010: i32 = GlobalAddress<i32* @gI> 0 [ORD=3] [ID=-3]

    0x7ffd5902cc10: <multiple use>
  0x7ffd5902d110: i32,ch = load 0x7ffd5902cf10, 0x7ffd5902d010,
  0x7ffd5902cc10<LD4[@gI]> [ORD=3] [ID=-3]
  ...

Legalized selection DAG: BB#0 '_Z3funv:entry'
SelectionDAG has 16 nodes:
  ...
      0x7ffd5902cc10: <multiple use>
    0x7ffd5902cf10: ch = store 0x7ffd5902cd10, 0x7ffd5902ca10, 0x7ffd5902ce10,
    0x7ffd5902cc10<ST4[%c]> [ORD=2] [ID=8]
```

(continues on next page)

(continued from previous page)

```
        0x7ffd5902d310: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=5]

      0x7ffd5902d710: i32 = Cpu0ISD::Hi 0x7ffd5902d310

        0x7ffd5902d610: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=6]

      0x7ffd5902d810: i32 = Cpu0ISD::Lo 0x7ffd5902d610

    0x7ffd5902fe10: i32 = add 0x7ffd5902d710, 0x7ffd5902d810

    0x7ffd5902cc10: <multiple use>
  0x7ffd5902d110: i32,ch = load 0x7ffd5902cf10, 0x7ffd5902fe10,
  0x7ffd5902cc10<LD4[@gI]> [ORD=3] [ID=9]
```

Finally, the pattern defined in Cpu0InstrInfo.td as the following will translate DAG (add Cpu0ISD::Hi<gI offset Hi16> Cpu0ISD::Lo<gI offset Lo16>) into Cpu0 instructions as below.

lbdex/chapters/Chapter6_1/Cpu0InstrInfo.td

```
// Hi and Lo nodes are used to handle global addresses. Used on
// Cpu0ISelLowering to lower stuff like GlobalAddress, ExternalSymbol
// static model. (nothing to do with Cpu0 Registers Hi and Lo)
def Cpu0Hi    : SDNode<"Cpu0ISD::Hi", SDTIntUnaryOp>;
def Cpu0Lo    : SDNode<"Cpu0ISD::Lo", SDTIntUnaryOp>;
```

```
// hi/lo relocs
let Predicates = [Ch6_1] in {
def : Pat<(Cpu0Hi tglobaladdr:$in), (LUi tglobaladdr:$in)>;
}
```

```
let Predicates = [Ch6_1] in {
def : Pat<(Cpu0Lo tglobaladdr:$in), (ORi ZERO, tglobaladdr:$in)>;
}
```

```
let Predicates = [Ch6_1] in {
def : Pat<(add CPURegs:$hi, (Cpu0Lo tglobaladdr:$lo)),
          (ORi CPURegs:$hi, tglobaladdr:$lo)>;
}
```

```
...
lui $2, %hi(gI)
ori $2, $2, %lo(gI)
...
```

As above, Pat<(...),(...)> include two lists of DAGs. The left is IR DAG and the right is machine instruction DAG. "Pat<(Cpu0Hi tglobaladdr:$in), (LUi, tglobaladdr:$in)>;" will translate DAG (Cpu0ISD::Hi tglobaladdr) into (lui (ori ZERO, tglobaladdr), 16). "Pat<(add CPURegs:$hi, (Cpu0Lo tglobaladdr:$lo)), (ORi CPURegs:$hi, tglobaladdr:$lo)>;" will translate DAG (add Cpu0ISD::Hi, Cpu0ISD::Lo) into Cpu0 instruction (ori Cpu0ISD::Hi, Cpu0ISD::Lo).

6.2.2 sdata or sbss

The sdata/sbss are 16 bits addressable areas which placed in ELF for fast access. Option cpu0-use-small-section=true will generate the following instructions.

```
  ori $2, $gp, %gp_rel(gI)
  ld  $2, 0($2)
  ...
  .type       gStart,@object          # @gStart
  .section    .sdata,"aw",@progbits
  .globl      gStart
  .align      2
gStart:
  .4byte      2                       # 0x2
  .size       gStart, 4

  .type       gI,@object              # @gI
  .globl      gI
  .align      2
gI:
  .4byte      100                     # 0x64
  .size       gI, 4
```

The code fragment of lowerGlobalAddress() as the following corresponding option `llc -relocation-model=static -cpu0-use-small-section=true` will translate DAG (GlobalAddress<i32* @gI> 0) into (add register %GP Cpu0ISD::GPRel<gI offset>) in stage "Legalized selection DAG" as below.

lbdex/chapters/Chapter6_1/Cpu0ISelLowering.cpp

```
SDValue Cpu0TargetLowering::lowerGlobalAddress(SDValue Op,
                                               SelectionDAG &DAG) const {
  //@lowerGlobalAddress }
  SDLoc DL(Op);
  const Cpu0TargetObjectFile *TLOF =
        static_cast<const Cpu0TargetObjectFile *>(
            getTargetMachine().getObjFileLowering());
  //@lga 1 {
  EVT Ty = Op.getValueType();
  GlobalAddressSDNode *N = cast<GlobalAddressSDNode>(Op);
  const GlobalValue *GV = N->getGlobal();
  //@lga 1 }

  if (!isPositionIndependent()) {
    //@ %gp_rel relocation
    const GlobalObject *GO = GV->getBaseObject();
    if (GO && TLOF->IsGlobalInSmallSection(GO, getTargetMachine())) {
      SDValue GA = DAG.getTargetGlobalAddress(GV, DL, MVT::i32, 0,
                                              Cpu0II::MO_GPREL);
      SDValue GPRelNode = DAG.getNode(Cpu0ISD::GPRel, DL,
                                      DAG.getVTList(MVT::i32), GA);
      SDValue GPReg = DAG.getRegister(Cpu0::GP, MVT::i32);
      return DAG.getNode(ISD::ADD, DL, MVT::i32, GPReg, GPRelNode);
```

(continues on next page)

(continued from previous page)

```
    }
```

```
    ...
  }
  ...
}
```

```
...
Type-legalized selection DAG: BB#0 '_Z3funv:entry'
SelectionDAG has 12 nodes:
  ...
      0x7fc5f382cc10: <multiple use>
    0x7fc5f382cf10: ch = store 0x7fc5f382cd10, 0x7fc5f382ca10, 0x7fc5f382ce10,
    0x7fc5f382cc10<ST4[%c]> [ORD=2] [ID=-3]

    0x7fc5f382d010: i32 = GlobalAddress<i32* @gI> 0 [ORD=3] [ID=-3]

    0x7fc5f382cc10: <multiple use>
  0x7fc5f382d110: i32,ch = load 0x7fc5f382cf10, 0x7fc5f382d010,
  0x7fc5f382cc10<LD4[@gI]> [ORD=3] [ID=-3]

Legalized selection DAG: BB#0 '_Z3funv:entry'
SelectionDAG has 15 nodes:
  ...
      0x7fc5f382cc10: <multiple use>
    0x7fc5f382cf10: ch = store 0x7fc5f382cd10, 0x7fc5f382ca10, 0x7fc5f382ce10,
    0x7fc5f382cc10<ST4[%c]> [ORD=2] [ID=8]

      0x7fc5f382d710: i32 = register %GP

        0x7fc5f382d310: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=4]

      0x7fc5f382d610: i32 = Cpu0ISD::GPRel 0x7fc5f382d310

    0x7fc5f382d810: i32 = add 0x7fc5f382d710, 0x7fc5f382d610

    0x7fc5f382cc10: <multiple use>
  0x7fc5f382d110: i32,ch = load 0x7fc5f382cf10, 0x7fc5f382d810,
  0x7fc5f382cc10<LD4[@gI]> [ORD=3] [ID=9]
  ...
```

Finally, the pattern defined in Cpu0InstrInfo.td as the following will translate DAG (add register %GP Cpu0ISD::GPRel<gI offset>) into Cpu0 instruction as below.

lbdex/chapters/Chapter6_1/Cpu0InstrInfo.td

```
def Cpu0GPRel : SDNode<"Cpu0ISD::GPRel", SDTIntUnaryOp>;
```

```
// gp_rel relocs
let Predicates = [Ch6_1] in {
def : Pat<(add CPURegs:$gp, (Cpu0GPRel tglobaladdr:$in)),
          (ORi CPURegs:$gp, tglobaladdr:$in)>;
}
```

```
ori $2, $gp, %gp_rel(gI)
...
```

"Pat<(add CPURegs:$gp, (Cpu0GPRel tglobaladdr:$in)), (ADD CPURegs:$gp, (ORi ZERO, tglobaladdr:$in))>;" will translate (add register %GP Cpu0ISD::GPRel tglobaladdr) into (add $gp, (ori ZERO, tglobaladdr)).

In this mode, the $gp content is assigned at compile/link time, changed only at program be loaded, and is fixed during the program running; on the contrary, when -relocation-model=pic the $gp can be changed during program running. For this example code, if $gp is assigned to the start address of .sdata by loader when program ch6_1.cpu0.s is loaded, then linker can caculate %gp_rel(gI) (= the relative address distance between gI and start of .sdata section). Which meaning this relocation record can be solved at link time, that's why it is static mode.

In this mode, we reserve $gp to a specfic fixed address of the program is loaded. As a result, the $gp cannot be allocated as a general purpose for variables. The following code tells llvm never allocate $gp for variables.

lbdex/chapters/Chapter6_1/Cpu0Subtarget.cpp

```
Cpu0Subtarget::Cpu0Subtarget(const Triple &TT, StringRef CPU,
                             StringRef FS, bool little,
                             const Cpu0TargetMachine &_TM) :
```

```
  // Set UseSmallSection.
  UseSmallSection = UseSmallSectionOpt;
  Cpu0ReserveGP = ReserveGPOpt;
  Cpu0NoCpload = NoCploadOpt;
#ifdef ENABLE_GPRESTORE
  if (!TM.isPositionIndependent() && !UseSmallSection && !Cpu0ReserveGP)
    FixGlobalBaseReg = false;
  else
#endif
    FixGlobalBaseReg = true;
```

```
}
```

lbdex/chapters/Chapter6_1/Cpu0RegisterInfo.cpp

```
BitVector Cpu0RegisterInfo::
getReservedRegs(const MachineFunction &MF) const {
//@getReservedRegs body {
```

```
#ifdef ENABLE_GPRESTORE //1
  const Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
  // Reserve GP if globalBaseRegFixed()
  if (Cpu0FI->globalBaseRegFixed())
#endif
    Reserved.set(Cpu0::GP);
```

```
  ...
}
```

6.3 pic mode

6.3.1 sdata or sbss

Option `llc -relocation-model=pic -cpu0-use-small-section=true` will generate the following instructions.

```
  ...
  .set        noreorder
  .cpload     $6
  .set        nomacro
  ...
  ld  $2, %got(gI)($gp)
  ld  $2, 0($2)
  ...
  .type       gStart,@object          # @gStart
  .data
  .globl      gStart
  .align      2
gStart:
  .4byte      2                       # 0x2
  .size       gStart, 4

  .type       gI,@object              # @gI
  .globl      gI
  .align      2
gI:
  .4byte      100                     # 0x64
  .size       gI, 4
```

The following code fragment of Cpu0AsmPrinter.cpp will emit **.cpload** asm pseudo instruction at function entry point as below.

lbdex/chapters/Chapter6_1/Cpu0MachineFunction.h

```
/// Cpu0FunctionInfo - This class is derived from MachineFunction private
/// Cpu0 target-specific information for each MachineFunction.
class Cpu0FunctionInfo : public MachineFunctionInfo {
public:
  Cpu0FunctionInfo(MachineFunction& MF)
  : MF(MF),
    VarArgsFrameIndex(0),
```

```
    GlobalBaseReg(0),
```

```
  bool globalBaseRegFixed() const;
  bool globalBaseRegSet() const;
  unsigned getGlobalBaseReg();
```

```
  /// GlobalBaseReg - keeps track of the virtual register initialized for
  /// use as the global base register. This is used for PIC in some PIC
  /// relocation models.
  unsigned GlobalBaseReg;
```

```
  int GPFI; // Index of the frame object for restoring $gp
```

```
  ...
};
```

lbdex/chapters/Chapter6_1/Cpu0MachineFunction.cpp

```
bool Cpu0FunctionInfo::globalBaseRegFixed() const {
  return FixGlobalBaseReg;
}

bool Cpu0FunctionInfo::globalBaseRegSet() const {
  return GlobalBaseReg;
}

unsigned Cpu0FunctionInfo::getGlobalBaseReg() {
  return GlobalBaseReg = Cpu0::GP;
}
```

lbdex/chapters/Chapter6_1/Cpu0AsmPrinter.cpp

```
/// EmitFunctionBodyStart - Targets can override this to emit stuff before
/// the first basic block in the function.
void Cpu0AsmPrinter::emitFunctionBodyStart() {
```

```
  bool EmitCPLoad = (MF->getTarget().getRelocationModel() == Reloc::PIC_) &&
    Cpu0FI->globalBaseRegSet() &&
```

(continues on next page)

(continued from previous page)

```
    Cpu0FI->globalBaseRegFixed();
  if (Cpu0NoCpload)
    EmitCPLoad = false;
```

```
    // Emit .cpload directive if needed.
    if (EmitCPLoad)
      OutStreamer->emitRawText(StringRef("\t.cpload\t$t9"));
```

```
  } else if (EmitCPLoad) {
    SmallVector<MCInst, 4> MCInsts;
    MCInstLowering.LowerCPLOAD(MCInsts);
    for (SmallVector<MCInst, 4>::iterator I = MCInsts.begin();
       I != MCInsts.end(); ++I)
      OutStreamer->emitInstruction(*I, getSubtargetInfo());
```

```
}
```

```
...
.set        noreorder
.cpload     $6
.set        nomacro
...
```

The **.cpload** is the assembly directive (macro) which will expand to several instructions. Issue **.cpload** before **.set nomacro** since the **.set nomacro** option causes the assembler to print a warning message whenever an assembler operation generates more than one machine language instruction, reference Mips ABI[2].

Following code will exspand .cpload into machine instructions as below. "0fa00000 09aa0000 13aa6000" is the **.cpload** machine instructions displayed in comments of Cpu0MCInstLower.cpp.

lbdex/chapters/Chapter6_1/Cpu0MCInstLower.h

```
/// This class is used to lower an MachineInstr into an MCInst.
class LLVM_LIBRARY_VISIBILITY Cpu0MCInstLower {
```

```
  void LowerCPLOAD(SmallVector<MCInst, 4>& MCInsts);
```

```
private:
  MCOperand LowerSymbolOperand(const MachineOperand &MO,
                               MachineOperandType MOTy, unsigned Offset) const;
```

```
  ...
}
```

[2] http://www.linux-mips.org/pub/linux/mips/doc/ABI/mipsabi.pdf

lbdex/chapters/Chapter6_1/Cpu0MCInstLower.cpp

```
// Lower ".cpload $reg" to
//  "lui   $gp, %hi(_gp_disp)"
//  "addiu $gp, $gp, %lo(_gp_disp)"
//  "addu  $gp, $gp, $t9"
void Cpu0MCInstLower::LowerCPLOAD(SmallVector<MCInst, 4>& MCInsts) {
  MCOperand GPReg = MCOperand::createReg(Cpu0::GP);
  MCOperand T9Reg = MCOperand::createReg(Cpu0::T9);
  StringRef SymName("_gp_disp");
  const MCSymbol *Sym = Ctx->getOrCreateSymbol(SymName);
  const Cpu0MCExpr *MCSym;

  MCSym = Cpu0MCExpr::create(Sym, Cpu0MCExpr::CEK_ABS_HI, *Ctx);
  MCOperand SymHi = MCOperand::createExpr(MCSym);
  MCSym = Cpu0MCExpr::create(Sym, Cpu0MCExpr::CEK_ABS_LO, *Ctx);
  MCOperand SymLo = MCOperand::createExpr(MCSym);

  MCInsts.resize(3);

  CreateMCInst(MCInsts[0], Cpu0::LUi, GPReg, SymHi);
  CreateMCInst(MCInsts[1], Cpu0::ORi, GPReg, GPReg, SymLo);
  CreateMCInst(MCInsts[2], Cpu0::ADD, GPReg, GPReg, T9Reg);
}
```

```
118-165-76-131:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -relocation-model=pic -filetype=
obj ch6_1.bc -o ch6_1.cpu0.o
118-165-76-131:input Jonathan$ gobjdump -s ch6_1.cpu0.o

ch6_1.cpu0.o:     file format elf32-big

Contents of section .text:
 0000 0fa00000 0daa0000 13aa6000  ...
...

118-165-76-131:input Jonathan$ gobjdump -tr ch6_1.cpu0.o
...
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000000 UNKNOWN           _gp_disp
00000008 UNKNOWN           _gp_disp
00000020 UNKNOWN           gI
```

Note: // **Mips ABI: _gp_disp** After calculating the gp, a function allocates the local stack space and saves the gp on the stack, so it can be restored after subsequent function calls. In other words, the gp is a caller saved register.

...

_gp_disp represents the offset between the beginning of the function and the global offset table. Various optimizations are possible in this code example and the others that follow. For example, the calculation of gp need not be done for a position-independent function that is strictly local to an object module.

The _gp_disp as above is a relocation record, it means both the machine instructions 0da00000 (offset 0) and 0daa0000 (offset 8) which equal to assembly "ori $gp, $zero, %hi(_gp_disp)" and assembly "ori $gp, $gp, %lo(_gp_disp)", respectively, are relocated records depend on _gp_disp. The loader or OS can caculate _gp_disp by (x - start address of .data) when load the dynamic function into memory x, and adjusts these two instructions offet correctly. Since shared function is loaded when this function is called, the relocation record "ld $2, %got(gI)($gp)" cannot be resolved in link time. In spite of the reloation record is solved on load time, the name binding is static, since linker deliver the memory address to loader, and loader can solve this just by caculate the offset directly. The memory reference bind with the offset of _gp_disp at link time. The ELF relocation records will be introduced in Chapter ELF Support. So, don't worry if you don't quite understand it at this point.

The code fragment of lowerGlobalAddress() as the following corresponding option `llc -relocation-model=pic` will translate DAG (GlobalAddress<i32* @gI> 0) into (load EntryToken, (Cpu0ISD::Wrapper Register %GP, TargetGlobalAddress<i32* @gI> 0)) in stage "Legalized selection DAG" as below.

lbdex/chapters/Chapter6_1/Cpu0ISelLowering.h

```
  // This method creates the following nodes, which are necessary for
  // computing a global symbol's address:
  //
  // (load (wrapper $gp, %got(sym)))
  template<class NodeTy>
  SDValue getAddrGlobal(NodeTy *N, EVT Ty, SelectionDAG &DAG,
                        unsigned Flag, SDValue Chain,
                        const MachinePointerInfo &PtrInfo) const {
    SDLoc DL(N);
    SDValue Tgt = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, getGlobalReg(DAG, Ty),
                              getTargetNode(N, Ty, DAG, Flag));
    return DAG.getLoad(Ty, DL, Chain, Tgt, PtrInfo);
  }
```

lbdex/chapters/Chapter6_1/Cpu0ISelLowering.cpp

```
SDValue Cpu0TargetLowering::lowerGlobalAddress(SDValue Op,
                                               SelectionDAG &DAG) const {
```

```
  EVT Ty = Op.getValueType();
  GlobalAddressSDNode *N = cast<GlobalAddressSDNode>(Op);
  const GlobalValue *GV = N->getGlobal();
```

```
  const GlobalObject *GO = GV->getBaseObject();
  if (GO && TLOF->IsGlobalInSmallSection(GO, getTargetMachine())) {
    SDValue GA = DAG.getTargetGlobalAddress(GV, DL, MVT::i32, 0,
                                            Cpu0II::MO_GPREL);
    SDValue GPRelNode = DAG.getNode(Cpu0ISD::GPRel, DL,
                                    DAG.getVTList(MVT::i32), GA);
    SDValue GPReg = DAG.getRegister(Cpu0::GP, MVT::i32);
    return DAG.getNode(ISD::ADD, DL, MVT::i32, GPReg, GPRelNode);
  }
```

```
  ...
}
```

lbdex/chapters/Chapter6_1/Cpu0ISelDAGToDAG.cpp

```
/// ComplexPattern used on Cpu0InstrInfo
/// Used on Cpu0 Load/Store instructions
bool Cpu0DAGToDAGISel::
SelectAddr(SDNode *Parent, SDValue Addr, SDValue &Base, SDValue &Offset) {
```

```
  // on PIC code Load GA
  if (Addr.getOpcode() == Cpu0ISD::Wrapper) {
    Base   = Addr.getOperand(0);
    Offset = Addr.getOperand(1);
    return true;
  }
```

```
  ...
}
```

```
...
Type-legalized selection DAG: BB#0 '_Z3funv:entry'
SelectionDAG has 12 nodes:
  ...
      0x7fad7102cc10: <multiple use>
    0x7fad7102cf10: ch = store 0x7fad7102cd10, 0x7fad7102ca10, 0x7fad7102ce10,
    0x7fad7102cc10<ST4[%c]> [ORD=2] [ID=-3]

    0x7fad7102d010: i32 = GlobalAddress<i32* @gI> 0 [ORD=3] [ID=-3]

    0x7fad7102cc10: <multiple use>
  0x7fad7102d110: i32,ch = load 0x7fad7102cf10, 0x7fad7102d010,
  0x7fad7102cc10<LD4[@gI]> [ORD=3] [ID=-3]
  ...
Legalized selection DAG: BB#0 '_Z3funv:entry'
SelectionDAG has 15 nodes:
  0x7ff3c9c10b98: ch = EntryToken [ORD=1] [ID=0]
  ...
      0x7fad7102cc10: <multiple use>
    0x7fad7102cf10: ch = store 0x7fad7102cd10, 0x7fad7102ca10, 0x7fad7102ce10,
    0x7fad7102cc10<ST4[%c]> [ORD=2] [ID=8]

      0x7fad70c10b98: <multiple use>
        0x7fad7102d610: i32 = Register %GP

        0x7fad7102d310: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=1]

      0x7fad7102d710: i32 = Cpu0ISD::Wrapper 0x7fad7102d610, 0x7fad7102d310
```

(continues on next page)

(continued from previous page)

```
    0x7fad7102cc10: <multiple use>
  0x7fad7102d810: i32,ch = load 0x7fad70c10b98, 0x7fad7102d710,
  0x7fad7102cc10<LD4[<unknown>]>
  0x7ff3ca02cc10: <multiple use>
0x7ff3ca02d110: i32,ch = load 0x7ff3ca02cf10, 0x7ff3ca02d810,
0x7ff3ca02cc10<LD4[@gI]> [ORD=3] [ID=9]
...
```

Finally, the pattern Cpu0 instruction **ld** defined before in Cpu0InstrInfo.td will translate DAG (load EntryToken, (Cpu0ISD::Wrapper Register %GP, TargetGlobalAddress<i32* @gI> 0)) into Cpu0 instruction as follows,

```
...
ld  $2, %got(gI)($gp)
...
```

Remind in pic mode, Cpu0 uses ".cpload" and "ld $2, %got(gI)($gp)" to access global variable as Mips. It takes 4 instructions in both Cpu0 and Mips. The cost came from we didn't assume that register $gp is always assigned to address .sdata and fixed there. Even we reserve $gp in this function, the $gp register can be changed at other functions. In last sub-section, the $gp is assumed to preserved at any function. If $gp is fixed during the run time, then ".cpload" can be removed here and have only one instruction cost in global variable access. The advantage of ".cpload" removing come from losing one general purpose register $gp which can be allocated for variables. In last sub-section, .sdata mode, we use ".cpload" removing since it is static link. In pic mode, the dynamic loading takes too much time. Romove ".cpload" with the cost of losing one general purpose register at all functions is not deserved here. The relocation records of ".cpload" from `llc -relocation-model=pic` can also be solved in link stage if we want to link this function by static link.

6.3.2 data or bss

The code fragment of lowerGlobalAddress() as the following corresponding option `llc -relocation-model=pic` will translate DAG (GlobalAddress<i32* @gI> 0) into (load EntryToken, (Cpu0ISD::Wrapper (add Cpu0ISD::Hi<gI offset Hi16>, Register %GP), TargetGlobalAddress<i32* @gI> 0)) in stage "Legalized selection DAG" as below.

lbdex/chapters/Chapter6_1/Cpu0ISelLowering.h

```
  // This method creates the following nodes, which are necessary for
  // computing a global symbol's address in large-GOT mode:
  //
  // (load (wrapper (add %hi(sym), $gp), %lo(sym)))
  template<class NodeTy>
  SDValue getAddrGlobalLargeGOT(NodeTy *N, EVT Ty, SelectionDAG &DAG,
                                unsigned HiFlag, unsigned LoFlag,
                                SDValue Chain,
                                const MachinePointerInfo &PtrInfo) const {
    SDLoc DL(N);
    SDValue Hi = DAG.getNode(Cpu0ISD::Hi, DL, Ty,
                             getTargetNode(N, Ty, DAG, HiFlag));
    Hi = DAG.getNode(ISD::ADD, DL, Ty, Hi, getGlobalReg(DAG, Ty));
    SDValue Wrapper = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, Hi,
                                  getTargetNode(N, Ty, DAG, LoFlag));
    return DAG.getLoad(Ty, DL, Chain, Wrapper, PtrInfo);
  }
```

lbdex/chapters/Chapter6_1/Cpu0ISelLowering.cpp

```
SDValue Cpu0TargetLowering::lowerGlobalAddress(SDValue Op,
                                               SelectionDAG &DAG) const {
```

```
  EVT Ty = Op.getValueType();
  GlobalAddressSDNode *N = cast<GlobalAddressSDNode>(Op);
  const GlobalValue *GV = N->getGlobal();
```

```
  const GlobalObject *GO = GV->getBaseObject();
  if (GO && !TLOF->IsGlobalInSmallSection(GO, getTargetMachine()))
    return getAddrGlobalLargeGOT(
        N, Ty, DAG, Cpu0II::MO_GOT_HI16, Cpu0II::MO_GOT_LO16,
        DAG.getEntryNode(),
        MachinePointerInfo::getGOT(DAG.getMachineFunction()));
  return getAddrGlobal(
      N, Ty, DAG, Cpu0II::MO_GOT, DAG.getEntryNode(),
      MachinePointerInfo::getGOT(DAG.getMachineFunction()));
}
```

```
...
Type-legalized selection DAG: BB#0 '_Z3funv:'
SelectionDAG has 10 nodes:
  ...
    0x7fb77a02cd10: ch = store 0x7fb779c10a08, 0x7fb77a02ca10, 0x7fb77a02cb10,
    0x7fb77a02cc10<ST4[%c]> [ORD=1] [ID=-3]

    0x7fb77a02ce10: i32 = GlobalAddress<i32* @gI> 0 [ORD=2] [ID=-3]

    0x7fb77a02cc10: <multiple use>
  0x7fb77a02cf10: i32,ch = load 0x7fb77a02cd10, 0x7fb77a02ce10,
  0x7fb77a02cc10<LD4[@gI]> [ORD=2] [ID=-3]
  ...

Legalized selection DAG: BB#0 '_Z3funv:'
SelectionDAG has 16 nodes:
  ...
    0x7fb77a02cd10: ch = store 0x7fb779c10a08, 0x7fb77a02ca10, 0x7fb77a02cb10,
    0x7fb77a02cc10<ST4[%c]> [ORD=1] [ID=6]

      0x7fb779c10a08: <multiple use>
            0x7fb77a02d110: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=19]

          0x7fb77a02d410: i32 = Cpu0ISD::Hi 0x7fb77a02d110

          0x7fb77a02d510: i32 = Register %GP

        0x7fb77a02d610: i32 = add 0x7fb77a02d410, 0x7fb77a02d510

        0x7fb77a02d710: i32 = TargetGlobalAddress<i32* @gI> 0 [TF=20]

      0x7fb77a02d810: i32 = Cpu0ISD::Wrapper 0x7fb77a02d610, 0x7fb77a02d710
```

(continues on next page)

(continued from previous page)

```
    0x7fb77a02cc10: <multiple use>
  0x7fb77a02fe10: i32,ch = load 0x7fb779c10a08, 0x7fb77a02d810,
  0x7fb77a02cc10<LD4[GOT]>

  0x7fb77a02cc10: <multiple use>
0x7fb77a02cf10: i32,ch = load 0x7fb77a02cd10, 0x7fb77a02fe10,
0x7fb77a02cc10<LD4[@gI]> [ORD=2] [ID=7]
...
```

Finally, the pattern Cpu0 instruction **ld** defined before in Cpu0InstrInfo.td will translate DAG (load EntryToken, (Cpu0ISD::Wrapper (add Cpu0ISD::Hi<gI offset Hi16>, Register %GP), Cpu0ISD::Lo<gI offset Lo16>)) into Cpu0 instructions as below.

```
...
ori $2, $zero, %got_hi(gI)
shl $2, $2, 16
add $2, $2, $gp
ld  $2, %got_lo(gI)($2)
...
```

The following code in Cpu0InstrInfo.td is needed for example input ch8_2_select_global_pic.cpp. Since ch8_2_select_global_pic.cpp uses llvm IR **select**, it cannot be run at this point. It will be run in later Chapter Control flow statements.

lbdex/chapters/Chapter6_1/Cpu0InstrInfo.td

```
def Cpu0Wrapper    : SDNode<"Cpu0ISD::Wrapper", SDTIntBinOp>;
```

```
let Predicates = [Ch6_1] in {
class WrapperPat<SDNode node, Instruction ORiOp, RegisterClass RC>:
      Pat<(Cpu0Wrapper RC:$gp, node:$in),
              (ORiOp RC:$gp, node:$in)>;

def : WrapperPat<tglobaladdr, ORi, GPROut>;
}
```

lbdex/input/ch8_2_select_global_pic.cpp

```
volatile int a1 = 1;
volatile int b1 = 2;

int gI1 = 100;
int gJ1 = 50;

int test_select_global_pic()
{
  if (a1 < b1)
    return gI1;
```

(continues on next page)

(continued from previous page)

```
  else
    return gJ1;
}
```

6.4 Global variable print support

Above code is for global address DAG translation. Next, add the following code to Cpu0MCInstLower.cpp and Cpu0ISelLowering.cpp for global variable printing operand function.

lbdex/chapters/Chapter6_1/Cpu0MCInstLower.cpp

```
MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,
                                              MachineOperandType MOTy,
                                              unsigned Offset) const {
  MCSymbolRefExpr::VariantKind Kind = MCSymbolRefExpr::VK_None;
  Cpu0MCExpr::Cpu0ExprKind TargetKind = Cpu0MCExpr::CEK_None;
  const MCSymbol *Symbol;

  switch(MO.getTargetFlags()) {
  default:                   llvm_unreachable("Invalid target flag!");
  case Cpu0II::MO_NO_FLAG:
    break;

// Cpu0_GPREL is for llc -march=cpu0 -relocation-model=static -cpu0-islinux-
//  format=false (global var in .sdata).
  case Cpu0II::MO_GPREL:
    TargetKind = Cpu0MCExpr::CEK_GPREL;
    break;

  case Cpu0II::MO_GOT:
    TargetKind = Cpu0MCExpr::CEK_GOT;
    break;
// ABS_HI and ABS_LO is for llc -march=cpu0 -relocation-model=static (global
//  var in .data).
  case Cpu0II::MO_ABS_HI:
    TargetKind = Cpu0MCExpr::CEK_ABS_HI;
    break;
  case Cpu0II::MO_ABS_LO:
    TargetKind = Cpu0MCExpr::CEK_ABS_LO;
    break;
  case Cpu0II::MO_GOT_HI16:
    TargetKind = Cpu0MCExpr::CEK_GOT_HI16;
    break;
  case Cpu0II::MO_GOT_LO16:
    TargetKind = Cpu0MCExpr::CEK_GOT_LO16;
    break;
  }

  switch (MOTy) {
```

(continues on next page)

(continued from previous page)

```
  case MachineOperand::MO_GlobalAddress:
    Symbol = AsmPrinter.getSymbol(MO.getGlobal());
    Offset += MO.getOffset();
    break;

  default:
    llvm_unreachable("<unknown operand type>");
  }

  const MCExpr *Expr = MCSymbolRefExpr::create(Symbol, Kind, *Ctx);

  if (Offset) {
    // Assume offset is never negative.
    assert(Offset > 0);
    Expr = MCBinaryExpr::createAdd(Expr, MCConstantExpr::create(Offset, *Ctx),
                                   *Ctx);
  }

  if (TargetKind != Cpu0MCExpr::CEK_None)
    Expr = Cpu0MCExpr::create(TargetKind, Expr, *Ctx);

  return MCOperand::createExpr(Expr);

}
```

```
MCOperand Cpu0MCInstLower::LowerOperand(const MachineOperand& MO,
                                        unsigned offset) const {
  MachineOperandType MOTy = MO.getType();

  switch (MOTy) {
```

```
  case MachineOperand::MO_GlobalAddress:
//@1
    return LowerSymbolOperand(MO, MOTy, offset);
```

```
  ...
  }
...
}
```

The Cpu0MCExpr::printImpl() of Cpu0InstPrinter.cpp in last chapter is for global variable printing operand function too.

The following function is for `llc -debug` this chapter DAG node name printing. It is added at Chapter3_1 already.

lbdex/chapters/Chapter3_1/Cpu0ISelLowering.cpp

```
const char *Cpu0TargetLowering::getTargetNodeName(unsigned Opcode) const {
  switch (Opcode) {
  ..
  case Cpu0ISD::GPRel:              return "Cpu0ISD::GPRel";
  ...
  case Cpu0ISD::Wrapper:            return "Cpu0ISD::Wrapper";
  ...
  }
}
```

OS is the output stream which output to the assembly file.

6.5 Summary

The global variable Instruction Selection for DAG translation is not like the ordinary IR node translation, it has static (absolute address) and pic mode. Backend deals this translation by create DAG nodes in function lowerGlobalAddress() which called by LowerOperation(). Function LowerOperation() takes care all Custom type of operation. Backend set global address as Custom operation by **"setOperationAction(ISD::GlobalAddress, MVT::i32, Custom);"** in Cpu0TargetLowering() constructor. Different address mode create their own DAG list at run time. By setting the pattern Pat<> in Cpu0InstrInfo.td, the llvm can apply the compiler mechanism, pattern match, in the Instruction Selection stage.

There are three types for setXXXAction(), Promote, Expand and Custom. Except Custom, the other two maybe no need to coding. Here[3] is the references.

As shown in this chapter, the global variable can be laid in .sdata/.sbss by option -cpu0-use-small-section=true. It is possible that the variables of small data section (16 bits addressable) are full out at link stage. When that happens, linker will highlights that error and forces the toolchain users to fix it. As the result, the toolchain user need to reconsider which global variables should be moved from .sdata/.sbss to .data/.bss by set option -cpu0-use-small-section=false in Makefile as follows,

Makefile

```
# Set the global variables declared in a.cpp to .data/.bss
llc  -march=cpu0 -relocation-model=static -cpu0-use-small-section=false \
-filetype=obj a.bc -o a.cpu0.o
# Set the global variables declared in b.cpp to .sdata/.sbss
llc  -march=cpu0 -relocation-model=static -cpu0-use-small-section=true \
-filetype=obj b.bc -o b.cpu0.o
```

The rule for global variables allocation is "set the small and frequent variables in small 16 addressable area".

[3] http://llvm.org/docs/WritingAnLLVMBackend.html#the-selectiondag-legalize-phase

CHAPTER

SEVEN

OTHER DATA TYPE

- *Local variable pointer*
- *char, short int and bool*
- *long long*
- *float and double*
- *Array and struct support*
- *Vector type (SIMD) support*

Until now, we only handle both int and long type of 32 bits size. This chapter introduce other types, such as pointer and those are not 32-bit size which inlcude bool, char, short int and long long.

7.1 Local variable pointer

To support pointer to local variable, add this code fragment in Cpu0InstrInfo.td and Cpu0InstPrinter.cpp as follows,

lbdex/chapters/Chapter7_1/Cpu0InstrInfo.td

```
def mem_ea : Operand<iPTR> {
  let PrintMethod = "printMemOperandEA";
  let MIOperandInfo = (ops GPROut, simm16);
  let EncoderMethod = "getMemEncoding";
}
```

```
class EffectiveAddress<string instr_asm, RegisterClass RC, Operand Mem> :
  FMem<0x09, (outs RC:$ra), (ins Mem:$addr),
     instr_asm, [(set RC:$ra, addr:$addr)], IIAlu>;
}
```

```
// FrameIndexes are legalized when they are operands from load/store
// instructions. The same not happens for stack address copies, so an
// add op with mem ComplexPattern is used and the stack address copy
// can be matched. It's similar to Sparc LEA_ADDRi
def LEA_ADDiu : EffectiveAddress<"addiu\t$ra, $addr", CPURegs, mem_ea> {
```

(continues on next page)

(continued from previous page)

```
  let isCodeGenOnly = 1;
}
```

lbdex/chapters/Chapter3_2/InstPrinter/Cpu0InstPrinter.h

```
  void printMemOperandEA(const MCInst *MI, int opNum, raw_ostream &O);
```

lbdex/chapters/Chapter3_2/InstPrinter/Cpu0InstPrinter.cpp

```
// The DAG data node, mem_ea of Cpu0InstrInfo.td, cannot be disabled by
// ch7_1, only opcode node can be disabled.
void Cpu0InstPrinter::
printMemOperandEA(const MCInst *MI, int opNum, raw_ostream &O) {
  // when using stack locations for not load/store instructions
  // print the same way as all normal 3 operand instructions.
  printOperand(MI, opNum, O);
  O << ", ";
  printOperand(MI, opNum+1, O);
  return;
}
```

As comment in Cpu0InstPrinter.cpp, the printMemOperandEA is added at early chapter 3_2 since the DAG data node, mem_ea of Cpu0InstrInfo.td, cannot be disabled by ch7_1_localpointer, only opcode node can be disabled. Run ch7_1_localpointer.cpp with code Chapter7_1/ which support pointer to local variable, will get result as follows,

lbdex/input/ch7_1_localpointer.cpp

```
int test_local_pointer()
{
  int b = 3;

  int* p = &b;

  return *p;
}
```

```
118-165-66-82:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_1_localpointer.cpp -emit-llvm -o ch7_1_localpointer.bc
118-165-66-82:input Jonathan$ llvm-dis ch7_1_localpointer.bc -o -
...
; Function Attrs: nounwind
define i32 @_Z18test_local_pointerv() #0 {
  %b = alloca i32, align 4
  %p = alloca i32*, align 4
  store i32 3, i32* %b, align 4
  store i32* %b, i32** %p, align 4
  %1 = load i32** %p, align 4
  %2 = load i32* %1, align 4
```

(continues on next page)

(continued from previous page)

```
  ret i32 %2
}
...

118-165-66-82:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=pic -filetype=asm
ch7_1_localpointer.bc -o -
  ...
        addiu $sp, $sp, -8
        addiu $2, $zero, 3
        st    $2, 4($fp)
        addiu $2, $fp, 4     // b address is 4($sp)
        st    $2, 0($fp)
        ld    $2, 4($fp)
        addiu $sp, $sp, 8
        ret   $lr
  ...
```

7.2 char, short int and bool

To support signed/unsigned type of char and short int, adding the following code to Chapter7_1/.

lbdex/chapters/Chapter7_1/Cpu0InstrInfo.td

```
def sextloadi16_a   : AlignedLoad<sextloadi16>;
def zextloadi16_a   : AlignedLoad<zextloadi16>;
def extloadi16_a    : AlignedLoad<extloadi16>;
```

```
def truncstorei16_a : AlignedStore<truncstorei16>;
```

```
let Predicates = [Ch7_1] in {
def LB     : LoadM32<0x03, "lb",  sextloadi8>;
def LBu    : LoadM32<0x04, "lbu", zextloadi8>;
def SB     : StoreM32<0x05, "sb", truncstorei8>;
def LH     : LoadM32<0x06, "lh",  sextloadi16_a>;
def LHu    : LoadM32<0x07, "lhu", zextloadi16_a>;
def SH     : StoreM32<0x08, "sh", truncstorei16_a>;
}
```

Run Chapter7_1/ with ch7_1_char_in_struct.cpp will get the following result.

lbdex/input/ch7_1_char_in_struct.cpp

```
struct Date
{
  short year;
  char month;
  char day;
  char hour;
  char minute;
  char second;
};

unsigned char b[4] = {'a', 'b', 'c', '\0'};

int test_char()
{
  unsigned char a = b[1];
  char c = (char)b[1];
  Date date1 = {2012, (char)11, (char)25, (char)9, (char)40, (char)15};
  char m = date1.month;
  char s = date1.second;

  return 0;
}
```

```
118-165-64-245:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-dis ch7_1_char_in_struct.bc -o -
define i32 @_Z9test_charv() #0 {
  %a = alloca i8, align 1
  %c = alloca i8, align 1
  %date1 = alloca %struct.Date, align 2
  %m = alloca i8, align 1
  %s = alloca i8, align 1
  %1 = load i8* getelementptr inbounds ([4 x i8]* @b, i32 0, i32 1), align 1
  store i8 %1, i8* %a, align 1
  %2 = load i8* getelementptr inbounds ([4 x i8]* @b, i32 0, i32 1), align 1
  store i8 %2, i8* %c, align 1
  %3 = bitcast %struct.Date* %date1 to i8*
  call void @llvm.memcpy.p0i8.p0i8.i32(i8* %3, i8* bitcast ({ i16, i8, i8, i8,
  i8, i8, i8 }* @_ZZ9test_charvE5date1 to i8*), i32 8, i32 2, i1 false)
  %4 = getelementptr inbounds %struct.Date* %date1, i32 0, i32 1
  %5 = load i8* %4, align 1
  store i8 %5, i8* %m, align 1
  %6 = getelementptr inbounds %struct.Date* %date1, i32 0, i32 5
  %7 = load i8* %6, align 1
  store i8 %7, i8* %s, align 1
  ret i32 0
}

118-165-64-245:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_1_char_in_struct.cpp -emit-llvm -o ch7_1_char_in_struct.bc
118-165-64-245:input Jonathan$ /Users/Jonathan/llvm/test/build/
```

(continues on next page)

(continued from previous page)

```
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch7_1_char_in_struct.bc -o -
  ...
# BB#0:                                 # %entry
  addiu $sp, $sp, -24
  lui $2, %got_hi(b)
  addu  $2, $2, $gp
  ld  $2, %got_lo(b)($2)
  lbu $3, 1($2)
  sb  $3, 20($fp)
  lbu $2, 1($2)
  sb  $2, 16($fp)
  ld  $2, %got($_ZZ9test_charvE5date1)($gp)
  addiu $2, $2, %lo($_ZZ9test_charvE5date1)
  lhu $3, 4($2)
  shl $3, $3, 16
  lhu $4, 6($2)
  or  $3, $3, $4
  st  $3, 12($fp) // store hour, minute and second on 12($sp)
  lhu $3, 2($2)
  lhu $2, 0($2)
  shl $2, $2, 16
  or  $2, $2, $3
  st  $2, 8($fp)    // store year, month and day on 8($sp)
  lbu $2, 10($fp)   // m = date1.month;
  sb  $2, 4($fp)
  lbu $2, 14($fp)   // s = date1.second;
  sb  $2, 0($fp)
  addiu $sp, $sp, 24
  ret $lr
  .set  macro
  .set  reorder
  .end  _Z9test_charv
$tmp1:
  .size _Z9test_charv, ($tmp1)-_Z9test_charv

  .type b,@object               # @b
  .data
  .globl  b
b:
  .asciz   "abc"
  .size b, 4

  .type $_ZZ9test_charvE5date1,@object # @_ZZ9test_charvE5date1
  .section  .rodata.cst8,"aM",@progbits,8
  .align  1
$_ZZ9test_charvE5date1:
  .2byte  2012                    # 0x7dc
  .byte 11                      # 0xb
  .byte 25                      # 0x19
  .byte 9                       # 0x9
  .byte 40                      # 0x28
```

(continues on next page)

(continued from previous page)

```
  .byte 15                        # 0xf
  .space  1
  .size $_ZZ9test_charvE5date1, 8
```

Run Chapter7_1/ with ch7_1_char_short.cpp will get the following result.

lbdex/input/ch7_1_char_short.cpp

```
int test_signed_char()
{
  char a = 0x80;
  int i = (signed int)a;
  i = i + 2; // i = (-128+2) = -126

  return i;
}

int test_unsigned_char()
{
  unsigned char c = 0x80;
  unsigned int ui = (unsigned int)c;
  ui = ui + 2; // i = (128+2) = 130

  return (int)ui;
}

int test_signed_short()
{
  short a = 0x8000;
  int i = (signed int)a;
  i = i + 2; // i = (-32768+2) = -32766

  return i;
}

int test_unsigned_short()
{
  unsigned short c = 0x8000;
  unsigned int ui = (unsigned int)c;
  ui = ui + 2; // i = (32768+2) = 32770
  c = (unsigned short)ui;

  return (int)ui;
}
```

```
1-160-136-236:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-dis ch7_1_char_short.bc -o -
  ...
define i32 @_Z16test_signed_charv() #0 {
  ...
```

(continues on next page)

(continued from previous page)

```
  %1 = load i8* %a, align 1
  %2 = sext i8 %1 to i32
  ...
}

; Function Attrs: nounwind
define i32 @_Z18test_unsigned_charv() #0 {
  ...
  %1 = load i8* %c, align 1
  %2 = zext i8 %1 to i32
  ...
}

; Function Attrs: nounwind
define i32 @_Z17test_signed_shortv() #0 {
  ...
  %1 = load i16* %a, align 2
  %2 = sext i16 %1 to i32
  ...
}

; Function Attrs: nounwind
define i32 @_Z19test_unsigned_shortv() #0 {
  ...
  %1 = load i16* %c, align 2
  %2 = zext i16 %1 to i32
  ...
}

attributes #0 = { nounwind }

1-160-136-236:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -filetype=asm ch7_1_char_short.bc -o -
  ...
  .globl  _Z16test_signed_charv
  ...
  lb  $2, 4($sp)
  ...
  .end  _Z16test_signed_charv

  .globl  _Z18test_unsigned_charv
  ...
  lbu $2, 4($sp)
  ...
  .end  _Z18test_unsigned_charv

  .globl  _Z17test_signed_shortv
  ...
  lh  $2, 4($sp)
  ...
  .end  _Z17test_signed_shortv
```

(continues on next page)

(continued from previous page)

```
  .globl  _Z19test_unsigned_shortv
  ...
  lhu $2, 4($sp)
  ...
  .end  _Z19test_unsigned_shortv
  ...
```

As you can see lb/lh are for signed byte/short type while lbu/lhu are for unsigned byte/short type. To support C type-cast or type-conversion feature efficiently, Cpu0 provide instruction "lb" to converse type char to int with one single instruction. The other instructions lbu, lh, lhu, sb and sh are applied in both signed or unsigned of type byte and short conversion. Their differences have been explained in Chapter 2.

To support load bool type, the following code added.

lbdex/chapters/Chapter7_1/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
  // Cpu0 does not have i1 type, so use i32 for
  // setcc operations results (slt, sgt, ...).
  setBooleanContents(ZeroOrOneBooleanContent);
  setBooleanVectorContents(ZeroOrNegativeOneBooleanContent);

  // Load extented operations for i1 types must be promoted
  for (MVT VT : MVT::integer_valuetypes()) {
    setLoadExtAction(ISD::EXTLOAD,  VT, MVT::i1,  Promote);
    setLoadExtAction(ISD::ZEXTLOAD, VT, MVT::i1,  Promote);
    setLoadExtAction(ISD::SEXTLOAD, VT, MVT::i1,  Promote);
  }
```

```
  ...
}
```

The setBooleanContents() purpose as following, but I don't know it well. Without it, the ch7_1_bool2.ll still works as below. The IR input file ch7_1_bool2.ll is used in testing here since the c++ version need flow control which is not supported at this point. File ch_run_backend.cpp include the test fragment for bool as below.

include/llvm/Target/TargetLowering.h

```
  enum BooleanContent { // How the target represents true/false values.
    UndefinedBooleanContent,    // Only bit 0 counts, the rest can hold garbage.
    ZeroOrOneBooleanContent,        // All bits zero except for bit 0.
    ZeroOrNegativeOneBooleanContent // All bits equal to bit 0.
  };
...
protected:
  /// setBooleanContents - Specify how the target extends the result of a
```

(continues on next page)

(continued from previous page)

```
  /// boolean value from i1 to a wider type.  See getBooleanContents.
  void setBooleanContents(BooleanContent Ty) { BooleanContents = Ty; }
  /// setBooleanVectorContents - Specify how the target extends the result
  /// of a vector boolean value from a vector of i1 to a wider type.  See
  /// getBooleanContents.
  void setBooleanVectorContents(BooleanContent Ty) {
    BooleanVectorContents = Ty;
  }
```

lbdex/input/ch7_1_bool2.ll

```
define zeroext i1 @verify_load_bool() #0 {
entry:
  %retval = alloca i1, align 1
  store i1 1, i1* %retval, align 1
  %0 = load i1, i1* %retval
  ret i1 %0
}
```

```
  118-165-64-245:input Jonathan$ /Users/Jonathan/llvm/test/build/
  bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch7_1_bool2.ll -o -

  .section .mdebug.abi32
  .previous
  .file "ch7_1_bool2.ll"
  .text
  .globl  verify_load_bool
  .align  2
  .type verify_load_bool,@function
  .ent  verify_load_bool        # @verify_load_bool
verify_load_bool:
  .cfi_startproc
  .frame  $sp,8,$lr
  .mask   0x00000000,0
  .set  noreorder
  .set  nomacro
# BB#0:                                 # %entry
  addiu $sp, $sp, -8
$tmp1:
  .cfi_def_cfa_offset 8
  addiu $2, $zero, 1
  sb  $2, 7($sp)
  addiu $sp, $sp, 8
  ret $lr
  .set  macro
  .set  reorder
  .end  verify_load_bool
$tmp2:
  .size verify_load_bool, ($tmp2)-verify_load_bool
  .cfi_endproc
```

The ch7_1_bool.cpp is the bool test version for C language. You can run with it at Chapter8_1 to get the similar result with ch7_1_bool2.ll.

lbdex/input/ch7_1_bool.cpp

```
bool test_load_bool()
{
  int a = 1;

  if (a < 0)
    return false;

  return true;
}
```

Summary as the following table.

Table 7.1: The C, IR, and DAG translation for char, short and bool translation (ch7_1_char_short.cpp and ch7_1_bool2.ll).

C	.bc	Optimized legalized selection DAG
char a =0x80;	%1 = load i8* %a, align 1	•
int i = (signed int)a;	%2 = sext i8 %1 to i32	load ..., <..., sext from i8>
unsigned char c = 0x80;	%1 = load i8* %c, align 1	•
unsigned int ui = (unsigned int)c;	%2 = zext i8 %1 to i32	load ..., <..., zext from i8>
short a =0x8000;	%1 = load i16* %a, align 2	•
int i = (signed int)a;	%2 = sext i16 %1 to i32	load ..., <..., sext from i16>
unsigned short c = 0x8000;	%1 = load i16* %c, align 2	•
unsigned int ui = (unsigned int)c;	%2 = zext i16 %1 to i32	load ..., <..., zext from i16>
c = (unsigned short)ui;	%6 = trunc i32 %5 to i16	•
•	store i16 %6, i16* %c, align 2	store ...,<..., trunc to i16>
return true;	store i1 1, i1* %retval, align 1	store ...,<..., trunc to i8>

Table 7.2: The backend translation for char, short and bool translation (ch7_1_char_short.cpp and ch7_1_bool2.ll).

Optimized legalized selection DAG	Cpu0	pattern in Cpu0InstrInfo.td
load ..., <..., sext from i8>	lb	LB : LoadM32<0x03, "lb", sextloadi8>;
load ..., <..., zext from i8>	lbu	LBu : LoadM32<0x04, "lbu", zextloadi8>;
load ..., <..., sext from i16>	lh	LH : LoadM32<0x06, "lh", sextloadi16_a>;
load ..., <..., zext from i16>	lhu	LHu : LoadM32<0x07, "lhu", zextloadi16_a>;
store ...,<..., trunc to i16>	sh	SH : StoreM32<0x08, "sh", truncstorei16_a>;
store ...,<..., trunc to i8>	sb	SB : StoreM32<0x05, "sb", truncstorei8>;

7.3 long long

Like Mips, the type long of Cpu0 is 32-bit and type long long is 64-bit for C language. To support type long long, we add the following code to Chapter7_1/.

lbdex/chapters/Chapter7_1/Cpu0SEISelDAGToDAG.cpp

```
void Cpu0SEDAGToDAGISel::selectAddESubE(unsigned MOp, SDValue InFlag,
                                        SDValue CmpLHS, const SDLoc &DL,
                                        SDNode *Node) const {
  unsigned Opc = InFlag.getOpcode(); (void)Opc;
  assert(((Opc == ISD::ADDC || Opc == ISD::ADDE) ||
          (Opc == ISD::SUBC || Opc == ISD::SUBE)) &&
         "(ADD|SUB)E flag operand must come from (ADD|SUB)C/E insn");

  SDValue Ops[] = { CmpLHS, InFlag.getOperand(1) };
  SDValue LHS = Node->getOperand(0), RHS = Node->getOperand(1);
  EVT VT = LHS.getValueType();

  SDNode *Carry;
  if (Subtarget->hasCpu032II())
    Carry = CurDAG->getMachineNode(Cpu0::SLTu, DL, VT, Ops);
  else {
    SDNode *StatusWord = CurDAG->getMachineNode(Cpu0::CMP, DL, VT, Ops);
    SDValue Constant1 = CurDAG->getTargetConstant(1, DL, VT);
    Carry = CurDAG->getMachineNode(Cpu0::ANDi, DL, VT,
                                        SDValue(StatusWord,0), Constant1);
  }
  SDNode *AddCarry = CurDAG->getMachineNode(Cpu0::ADDu, DL, VT,
                                         SDValue(Carry,0), RHS);

  CurDAG->SelectNodeTo(Node, MOp, VT, MVT::Glue, LHS, SDValue(AddCarry,0));
}
```

```
bool Cpu0SEDAGToDAGISel::trySelect(SDNode *Node) {
  unsigned Opcode = Node->getOpcode();
  SDLoc DL(Node);

  ///
  // Instruction Selection not handled by the auto-generated
  // tablegen selection should be handled here.
  ///

  ///
  // Instruction Selection not handled by the auto-generated
  // tablegen selection should be handled here.
  ///
  EVT NodeTy = Node->getValueType(0);
  unsigned MultOpc;

  switch(Opcode) {
```

(continues on next page)

(continued from previous page)

```
  default: break;
```

```
  case ISD::SUBE: {
    SDValue InFlag = Node->getOperand(2);
    selectAddESubE(Cpu0::SUBu, InFlag, InFlag.getOperand(0), DL, Node);
    return true;
  }

  case ISD::ADDE: {
    SDValue InFlag = Node->getOperand(2);
    selectAddESubE(Cpu0::ADDu, InFlag, InFlag.getValue(0), DL, Node);
    return true;
  }

  /// Mul with two results
  case ISD::SMUL_LOHI:
  case ISD::UMUL_LOHI: {
    MultOpc = (Opcode == ISD::UMUL_LOHI ? Cpu0::MULTu : Cpu0::MULT);

    std::pair<SDNode*, SDNode*> LoHi =
        selectMULT(Node, MultOpc, DL, NodeTy, true, true);

    if (!SDValue(Node, 0).use_empty())
      ReplaceUses(SDValue(Node, 0), SDValue(LoHi.first, 0));

    if (!SDValue(Node, 1).use_empty())
      ReplaceUses(SDValue(Node, 1), SDValue(LoHi.second, 0));

    CurDAG->RemoveDeadNode(Node);
    return true;
  }
```

```
  ...
}
```

lbdex/chapters/Chapter7_1/Cpu0ISelLowering.h

```
  class Cpu0TargetLowering : public TargetLowering  {
```

```
    bool isOffsetFoldingLegal(const GlobalAddressSDNode *GA) const override;
```

```
  ...
}
```

lbdex/chapters/Chapter7_1/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
  // Handle i64 shl
  setOperationAction(ISD::SHL_PARTS,          MVT::i32,   Expand);
  setOperationAction(ISD::SRA_PARTS,          MVT::i32,   Expand);
  setOperationAction(ISD::SRL_PARTS,          MVT::i32,   Expand);
```

```
  ...
}
```

The added code in Cpu0ISelLowering.cpp are for shift operations which support type long long 64-bit. When applying operators << and >> in 64-bit variables will create DAG SHL_PARTS, SRA_PARTS and SRL_PARTS those which take care the 32 bits operands during llvm DAGs translation. File ch9_7.cpp of 64-bit shift operations cannot be run at this point. It will be verified on later chapter "Function call".

Run Chapter7_1 with ch7_1_longlong.cpp to get the result as follows,

lbdex/input/ch7_1_longlong.cpp

```
long long test_longlong()
{
  long long a = 0x300000002;
  long long b = 0x100000001;
  int a1 = 0x3001000;
  int b1 = 0x2001000;

  long long c = a + b;   // c = 0x00000004,00000003
  long long d = a - b;   // d = 0x00000002,00000001
  long long e = a * b;   // e = 0x00000005,00000002
  long long f = (long long)a1 * (long long)b1; // f = 0x00060050,01000000

  long long g = ((-7 * 8) + 1) >> 4; // g = -55/16=-3.4375=-4

  return (c+d+e+f+g); // (0x0006005b,01000002) = (393307,16777218)
}
```

```
1-160-134-62:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_1_longlong.cpp -emit-llvm -o ch7_1_longlong.bc
1-160-134-62:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -filetype=asm
ch7_1_longlong.bc -o -
  ...
# BB#0:
        addiu $sp, $sp, -72
        st    $8, 68($fp)             # 4-byte Folded Spill
```

(continues on next page)

(continued from previous page)

```
        addiu $2, $zero, 2
        st    $2, 60($fp)
        addiu $2, $zero, 3
        st    $2, 56($fp)
        addiu $2, $zero, 1
        st    $2, 52($fp)
        st    $2, 48($fp)
        lui   $2, 768
        ori   $2, $2, 4096
        st    $2, 44($fp)
        lui   $2, 512
        ori   $2, $2, 4096
        st    $2, 40($fp)
        ld    $2, 52($fp)
        ld    $3, 60($fp)
        addu  $3, $3, $2
        ld    $4, 56($fp)
        ld    $5, 48($fp)
        st    $3, 36($fp)
        cmp   $sw, $3, $2
        andi  $2, $sw, 1
        addu  $2, $2, $5
        addu  $2, $4, $2
        st    $2, 32($fp)
        ld    $2, 52($fp)
        ld    $3, 60($fp)
        subu  $4, $3, $2
        ld    $5, 56($fp)
        ld    $t9, 48($fp)
        st    $4, 28($fp)
        cmp   $sw, $3, $2
        andi  $2, $sw, 1
        addu  $2, $2, $t9
        subu  $2, $5, $2
        st    $2, 24($fp)
        ld    $2, 52($fp)
        ld    $3, 60($fp)
        multu $3, $2
        mflo  $4
        mfhi  $5
        ld    $t9, 56($fp)
        ld    $7, 48($fp)
        st    $4, 20($fp)
        mul   $3, $3, $7
        addu  $3, $5, $3
        mul   $2, $t9, $2
        addu  $2, $3, $2
        st    $2, 16($fp)
        ld    $2, 40($fp)
        ld    $3, 44($fp)
        mult  $3, $2
        mflo  $2
```

(continues on next page)

(continued from previous page)

```
        mfhi  $4
        st    $2, 12($fp)
        st    $4, 8($fp)
        ld    $5, 28($fp)
        ld    $3, 36($fp)
        addu  $t9, $3, $5
        ld    $7, 20($fp)
        addu  $8, $t9, $7
        addu  $3, $8, $2
        cmp   $sw, $3, $2
        andi  $2, $sw, 1
        addu  $2, $2, $4
        cmp   $sw, $t9, $5
        st    $sw, 4($fp)             # 4-byte Folded Spill
        cmp   $sw, $8, $7
        andi  $4, $sw, 1
        ld    $5, 16($fp)
        addu  $4, $4, $5
        ld    $sw, 4($fp)             # 4-byte Folded Reload
        andi  $5, $sw, 1
        ld    $t9, 24($fp)
        addu  $5, $5, $t9
        ld    $t9, 32($fp)
        addu  $5, $t9, $5
        addu  $4, $5, $4
        addu  $2, $4, $2
        ld    $8, 68($fp)             # 4-byte Folded Reload
        addiu $sp, $sp, 72
        ret   $lr
  ...
```

7.4 float and double

Cpu0 only has integer instructions at this point. For float operations, Cpu0 backend will call the library function to translate integer to float as follows,

lbdex/input/ch7_1_fmul.c

```
/*
~/llvm/debug/build/bin/clang -target mips-unknown-linux-gnu -emit-llvm -S ch7_1_fmul.c
        ...
        %mul = fmul float %0, %1

~/llvm/debug/build/bin/llc -march=mips ch7_1_fmul.ll -relocation-model=static -o -
        ...
        v_log_f32_e32 v1, v0
        v_mul_legacy_f32_e32 v0, v0, v1
        v_exp_f32_e32 v0, v0
```

(continues on next page)

(continued from previous page)

```
~/llvm/test/build/bin/llc -march=cpu0 ch7_1_fmul.ll -relocation-model=static -o -
        ...
        jsub __mulsf3
*/

float ch7_1_fmul(float a, float b) {
  float c = a * b;
  return c;
}
```

This float (or double) function call for Cpu0 will be supported after the chapter of function call. For hardware cost reason, many CPU have no hardware float instructions. They call library function to finish float operations. Mips sperarate float operations with a sperarate co-processor for those needing "float intended" application.

In order to support float point library (part of compiler-rt)[2], the following code are added to support instructions clz and clo. Though clz and clo instructions are implemented in compiler-rt. However these two instructions are integer operations and will get better speed up in float point application.

lbdex/chapters/Chapter7_1/Cpu0InstrInfo.td

```
let Predicates = [Ch7_1] in {
// Count Leading Ones/Zeros in Word
class CountLeading0<bits<8> op, string instr_asm, RegisterClass RC>:
  FA<op, (outs GPROut:$ra), (ins RC:$rb),
     !strconcat(instr_asm, "\t$ra, $rb"),
     [(set GPROut:$ra, (ctlz RC:$rb))], II_CLZ> {
  let rc = 0;
  let shamt = 0;
}

class CountLeading1<bits<8> op, string instr_asm, RegisterClass RC>:
  FA<op, (outs GPROut:$ra), (ins RC:$rb),
     !strconcat(instr_asm, "\t$ra, $rb"),
     [(set GPROut:$ra, (ctlz (not RC:$rb)))], II_CLO> {
  let rc = 0;
  let shamt = 0;
}
```

```
let Predicates = [Ch7_1] in {
/// Count Leading
def CLZ : CountLeading0<0x15, "clz", CPURegs>;
def CLO : CountLeading1<0x16, "clo", CPURegs>;
```

[2] http://jonathan2251.github.io/lbt/lib.html#compiler-rt

7.5 Array and struct support

LLVM uses getelementptr to represent the array and struct type in C. Please reference here[1]. For ch7_1_globalstructoffset.cpp, the llvm IR as follows,

lbdex/input/ch7_1_globalstructoffset.cpp

```
struct Date
{
  int year;
  int month;
  int day;
};

Date date = {2012, 10, 12};
int a[3] = {2012, 10, 12};

int test_struct()
{
  int day = date.day;
  int i = a[1];

  return (i+day); // 10+12=22
}
```

```
// ch7_1_globalstructoffset.ll
; ModuleID = 'ch7_1_globalstructoffset.bc'
...
%struct.Date = type { i32, i32, i32 }

@date = global %struct.Date { i32 2012, i32 10, i32 12 }, align 4
@a = global [3 x i32] [i32 2012, i32 10, i32 12], align 4

; Function Attrs: nounwind
define i32 @_Z11test_structv() #0 {
  %day = alloca i32, align 4
  %i = alloca i32, align 4
  %1 = load i32* getelementptr inbounds (%struct.Date* @date, i32 0, i32 2), align 4
  store i32 %1, i32* %day, align 4
  %2 = load i32* getelementptr inbounds ([3 x i32]* @a, i32 0, i32 1), align 4
  store i32 %2, i32* %i, align 4
  %3 = load i32* %i, align 4
  %4 = load i32* %day, align 4
  %5 = add nsw i32 %3, %4
  ret i32 %5
}
```

Run Chapter6_1/ with ch7_1_globalstructoffset.bc on static mode will get the incorrect asm file as follows,

[1] http://llvm.org/docs/LangRef.html#getelementptr-instruction

```
1-160-134-62:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/
llc -march=cpu0 -relocation-model=static -filetype=asm
ch7_1_globalstructoffset.bc -o -
  ...
  lui $2, %hi(date)
  ori $2, $2, %lo(date)
  ld  $2, 0($2)   // the correct one is   ld  $2, 8($2)
  ...
```

For **"day = date.day"**, the correct one is **"ld $2, 8($2)"**, not **"ld $2, 0($2)"**, since date.day is offset 8(date) (Type int is 4 bytes in Cpu0, and the date.day has fields year and month before it). Let's use debug option in llc to see what's wrong,

```
jonathantekiimac:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -debug -relocation-model=static
-filetype=asm ch6_2.bc -o ch6_2.cpu0.static.s
...
=== main
Initial selection DAG: BB#0 'main:entry'
SelectionDAG has 20 nodes:
  0x7f7f5b02d210: i32 = undef [ORD=1]

      0x7f7f5ac10590: ch = EntryToken [ORD=1]

      0x7f7f5b02d010: i32 = Constant<0> [ORD=1]

      0x7f7f5b02d110: i32 = FrameIndex<0> [ORD=1]

      0x7f7f5b02d210: <multiple use>
    0x7f7f5b02d310: ch = store 0x7f7f5ac10590, 0x7f7f5b02d010, 0x7f7f5b02d110,
    0x7f7f5b02d210<ST4[%retval]> [ORD=1]

      0x7f7f5b02d410: i32 = GlobalAddress<%struct.Date* @date> 0 [ORD=2]

      0x7f7f5b02d510: i32 = Constant<8> [ORD=2]

    0x7f7f5b02d610: i32 = add 0x7f7f5b02d410, 0x7f7f5b02d510 [ORD=2]

    0x7f7f5b02d210: <multiple use>
  0x7f7f5b02d710: i32,ch = load 0x7f7f5b02d310, 0x7f7f5b02d610, 0x7f7f5b02d210
  <LD4[getelementptr inbounds (%struct.Date* @date, i32 0, i32 2)]> [ORD=3]

  0x7f7f5b02db10: i64 = Constant<4>

      0x7f7f5b02d710: <multiple use>
      0x7f7f5b02d710: <multiple use>
      0x7f7f5b02d810: i32 = FrameIndex<1> [ORD=4]

      0x7f7f5b02d210: <multiple use>
    0x7f7f5b02d910: ch = store 0x7f7f5b02d710:1, 0x7f7f5b02d710, 0x7f7f5b02d810,
     0x7f7f5b02d210<ST4[%day]> [ORD=4]

      0x7f7f5b02da10: i32 = GlobalAddress<[3 x i32]* @a> 0 [ORD=5]
```

(continues on next page)

(continued from previous page)

```
      0x7f7f5b02dc10: i32 = Constant<4> [ORD=5]

    0x7f7f5b02dd10: i32 = add 0x7f7f5b02da10, 0x7f7f5b02dc10 [ORD=5]

    0x7f7f5b02d210: <multiple use>
  0x7f7f5b02de10: i32,ch = load 0x7f7f5b02d910, 0x7f7f5b02dd10, 0x7f7f5b02d210
  <LD4[getelementptr inbounds ([3 x i32]* @a, i32 0, i32 1)]> [ORD=6]

...


Replacing.3 0x7f7f5b02dd10: i32 = add 0x7f7f5b02da10, 0x7f7f5b02dc10 [ORD=5]

With: 0x7f7f5b030010: i32 = GlobalAddress<[3 x i32]* @a> + 4


Replacing.3 0x7f7f5b02d610: i32 = add 0x7f7f5b02d410, 0x7f7f5b02d510 [ORD=2]

With: 0x7f7f5b02db10: i32 = GlobalAddress<%struct.Date* @date> + 8

Optimized lowered selection DAG: BB#0 'main:entry'
SelectionDAG has 15 nodes:
  0x7f7f5b02d210: i32 = undef [ORD=1]

      0x7f7f5ac10590: ch = EntryToken [ORD=1]

      0x7f7f5b02d010: i32 = Constant<0> [ORD=1]

      0x7f7f5b02d110: i32 = FrameIndex<0> [ORD=1]

      0x7f7f5b02d210: <multiple use>
    0x7f7f5b02d310: ch = store 0x7f7f5ac10590, 0x7f7f5b02d010, 0x7f7f5b02d110,
    0x7f7f5b02d210<ST4[%retval]> [ORD=1]

    0x7f7f5b02db10: i32 = GlobalAddress<%struct.Date* @date> + 8

    0x7f7f5b02d210: <multiple use>
  0x7f7f5b02d710: i32,ch = load 0x7f7f5b02d310, 0x7f7f5b02db10, 0x7f7f5b02d210
  <LD4[getelementptr inbounds (%struct.Date* @date, i32 0, i32 2)]> [ORD=3]

      0x7f7f5b02d710: <multiple use>
      0x7f7f5b02d710: <multiple use>
      0x7f7f5b02d810: i32 = FrameIndex<1> [ORD=4]

      0x7f7f5b02d210: <multiple use>
    0x7f7f5b02d910: ch = store 0x7f7f5b02d710:1, 0x7f7f5b02d710, 0x7f7f5b02d810,
     0x7f7f5b02d210<ST4[%day]> [ORD=4]

    0x7f7f5b030010: i32 = GlobalAddress<[3 x i32]* @a> + 4

    0x7f7f5b02d210: <multiple use>
```

(continues on next page)

(continued from previous page)

```
  0x7f7f5b02de10: i32,ch = load 0x7f7f5b02d910, 0x7f7f5b030010, 0x7f7f5b02d210
  <LD4[getelementptr inbounds ([3 x i32]* @a, i32 0, i32 1)]> [ORD=6]

...
```

Through `llc -debug`, you can see the DAG translation process. As above, the DAG list for date.day (add GlobalAddress<[3 x i32]* @a> 0, Constant<8>) with 3 nodes is replaced by 1 node GlobalAddress<%struct.Date* @date> + 8. The DAG list for a[1] is same. The replacement occurs since TargetLowering.cpp::isOffsetFoldingLegal(...) return true in `llc -static` static addressing mode as below. In Cpu0 the **ld** instruction format is **"ld $r1, offset($r2)"** which meaning load $r2 address+offset to $r1. So, we just replace the isOffsetFoldingLegal(...) function by override mechanism as below.

lib/CodeGen/SelectionDAG/TargetLowering.cpp

```
bool
TargetLowering::isOffsetFoldingLegal(const GlobalAddressSDNode *GA) const {
  // Assume that everything is safe in static mode.
  if (getTargetMachine().getRelocationModel() == Reloc::Static)
    return true;

  // In dynamic-no-pic mode, assume that known defined values are safe.
  if (getTargetMachine().getRelocationModel() == Reloc::DynamicNoPIC &&
     GA &&
     !GA->getGlobal()->isDeclaration() &&
     !GA->getGlobal()->isWeakForLinker())
  return true;

  // Otherwise assume nothing is safe.
  return false;
}
```

lbdex/chapters/Chapter7_1/Cpu0ISelLowering.cpp

```
bool
Cpu0TargetLowering::isOffsetFoldingLegal(const GlobalAddressSDNode *GA) const {
  // The Cpu0 target isn't yet aware of offsets.
  return false;
}
```

Beyond that, we need to add the following code fragment to Cpu0ISelDAGToDAG.cpp,

lbdex/chapters/Chapter7_1/Cpu0ISelDAGToDAG.cpp

```
/// ComplexPattern used on Cpu0InstrInfo
/// Used on Cpu0 Load/Store instructions
bool Cpu0DAGToDAGISel::
SelectAddr(SDNode *Parent, SDValue Addr, SDValue &Base, SDValue &Offset) {
```

```
  // Addresses of the form FI+const or FI|const
  if (CurDAG->isBaseWithConstantOffset(Addr)) {
    ConstantSDNode *CN = dyn_cast<ConstantSDNode>(Addr.getOperand(1));
    if (isInt<16>(CN->getSExtValue())) {

      // If the first operand is a FI, get the TargetFI Node
      if (FrameIndexSDNode *FIN = dyn_cast<FrameIndexSDNode>
                                  (Addr.getOperand(0)))
        Base = CurDAG->getTargetFrameIndex(FIN->getIndex(), ValTy);
      else
        Base = Addr.getOperand(0);

      Offset = CurDAG->getTargetConstant(CN->getZExtValue(), DL, ValTy);
      return true;
    }
  }
```

```
  ...
}
```

Recall we have translated DAG list for date.day (add GlobalAddress<[3 x i32]* @a> 0, Constant<8>) into (add (add Cpu0ISD::Hi (Cpu0II::MO_ABS_HI), Cpu0ISD::Lo(Cpu0II::MO_ABS_LO)), Constant<8>) by the following code in Cpu0ISelLowering.cpp.

lbdex/chapters/Chapter6_1/Cpu0ISelLowering.h

```
    // This method creates the following nodes, which are necessary for
    // computing a symbol's address in non-PIC mode:
    //
    // (add %hi(sym), %lo(sym))
    template<class NodeTy>
    SDValue getAddrNonPIC(NodeTy *N, EVT Ty, SelectionDAG &DAG) const {
      SDLoc DL(N);
      SDValue Hi = getTargetNode(N, Ty, DAG, Cpu0II::MO_ABS_HI);
      SDValue Lo = getTargetNode(N, Ty, DAG, Cpu0II::MO_ABS_LO);
      return DAG.getNode(ISD::ADD, DL, Ty,
                         DAG.getNode(Cpu0ISD::Hi, DL, Ty, Hi),
                         DAG.getNode(Cpu0ISD::Lo, DL, Ty, Lo));
    }
```

So, when the SelectAddr(...) of Cpu0ISelDAGToDAG.cpp is called. The Addr SDValue in SelectAddr(..., Addr, ...) is DAG list for date.day (add (add Cpu0ISD::Hi (Cpu0II::MO_ABS_HI), Cpu0ISD::Lo(Cpu0II::MO_ABS_LO)), Constant<8>). Since Addr.getOpcode() = ISD:ADD, Addr.getOperand(0) = (add Cpu0ISD::Hi (Cpu0II::MO_ABS_HI), Cpu0ISD::Lo(Cpu0II::MO_ABS_LO)) and Addr.getOperand(1).getOpcode() = ISD::Constant, the Base = SDValue (add Cpu0ISD::Hi (Cpu0II::MO_ABS_HI), Cpu0ISD::Lo(Cpu0II::MO_ABS_LO)) and Offset = Constant<8>. After

set Base and Offset, the load DAG will translate the global address date.day into machine instruction **"ld $r1, 8($r2)"** in Instruction Selection stage.

Chapter7_1/ include these changes as above, you can run it with ch7_1_globalstructoffset.cpp to get the correct generated instruction **"ld $r1, 8($r2)"** for date.day access, as follows.

```
...
      lui   $2, %hi(date)
      ori   $2, $2, %lo(date)
      ld    $2, 8($2)   // correct
...
```

The ch7_1_localarrayinit.cpp is for local variable initialization test. The result as follows,

lbdex/input/ch7_1_localarrayinit.cpp

```
int main()
{
  int a[3]={0, 1, 2};

  return 0;
}
```

```
118-165-79-206:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_1_localarrayinit.cpp -emit-llvm -o ch7_1_localarrayinit.bc
118-165-79-206:input Jonathan$ llvm-dis ch7_1_localarrayinit.bc -o -
...
```

```
define i32 @main() nounwind ssp {
entry:
  %retval = alloca i32, align 4
  %a = alloca [3 x i32], align 4
  store i32 0, i32* %retval
  %0 = bitcast [3 x i32]* %a to i8*
  call void @llvm.memcpy.p0i8.p0i8.i32(i8* %0, i8* bitcast ([3 x i32]*
    @_ZZ4mainE1a to i8*), i32 12, i32 4, i1 false)
  ret i32 0
}
; Function Attrs: nounwind
declare void @llvm.memcpy.p0i8.p0i8.i32(i8* nocapture, i8* nocapture, i32, i32, i1) #1
```

```
118-165-79-206:input Jonathan$ ~/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch7_1_localarrayinit.bc -o -
        ...
# BB#0:                                 # %entry
        addiu $sp, $sp, -16
        addiu $2, $zero, 0
        st    $2, 12($fp)
        ld    $2, %got($_ZZ4mainE1a)($gp)
        ori   $2, $2, %lo($_ZZ4mainE1a)
        ld    $3, 8($2)
        st    $3, 8($fp)
```

(continues on next page)

(continued from previous page)

```
        ld    $3, 4($2)
        st    $3, 4($fp)
        ld    $2, 0($2)
        st    $2, 0($fp)
        addiu $sp, $sp, 16
        ret   $lr
        ...
        .type $_ZZ4mainE1a,@object    # @_ZZ4mainE1a
        .section      .rodata,"a",@progbits
        .align        2
$_ZZ4mainE1a:
        .4byte        0                       # 0x0
        .4byte        1                       # 0x1
        .4byte        2                       # 0x2
        .size $_ZZ4mainE1a, 12
```

7.6 Vector type (SIMD) support

Vector types are used when multiple primitive data are operated in parallel using a single instruction (SIMD)[3]. Mips supports the following llvm IRs "icmp slt" and "sext" for vector type, Cpu0 supports them either.

lbdex/input/ch7_1_vector.cpp

```
typedef long   vector8long   __attribute__((__vector_size__(32)));
typedef long   vector8short   __attribute__((__vector_size__(16)));


int test_cmplt_short() {
  volatile vector8short a0 = {0, 1, 2, 3};
  volatile vector8short b0 = {2, 2, 2, 4};
  volatile vector8short c0;
  c0 = a0 < b0; // c0[0] = -1 (since 0 < 2 is true), c0[1] = -1, c0[2] = 0 (since 2 < 2␣
↪is false), c0[3] = -1

  return (int)(c0[0]+c0[1]+c0[2]+c0[3]); // -3
}


int test_cmplt_long() {
  volatile vector8long a0 = {2, 2, 2, 2, 1, 1, 1, 1};
  volatile vector8long b0 = {1, 1, 1, 1, 2, 2, 2, 2};
  volatile vector8long c0;
  c0 = a0 < b0; // c0[0..3] = {0, 0, ...}, c0[4..7] = {-1, ...}

  return (c0[0]+c0[1]+c0[2]+c0[3]+c0[4]+c0[5]+c0[6]+c0[7]); //-4
}
```

[3] http://llvm.org/docs/LangRef.html#vector-type

```
118-165-79-206:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_1_vector.cpp -emit-llvm -o ch7_1_vector.bc
118-165-79-206:input Jonathan$ ~/llvm/test/build/bin/
llvm-dis ch7_1_vector.bc -o -
...
```

```
; Function Attrs: nounwind
define i32 @_Z16test_cmplt_shortv() #0 {
  %a0 = alloca <4 x i32>, align 16
  %b0 = alloca <4 x i32>, align 16
  %c0 = alloca <4 x i32>, align 16
  store volatile <4 x i32> <i32 0, i32 1, i32 2, i32 3>, <4 x i32>* %a0, align 16
  store volatile <4 x i32> <i32 2, i32 2, i32 2, i32 2>, <4 x i32>* %b0, align 16
  %1 = load volatile <4 x i32>, <4 x i32>* %a0, align 16
  %2 = load volatile <4 x i32>, <4 x i32>* %b0, align 16
  %3 = icmp slt <4 x i32> %1, %2
  %4 = sext <4 x i1> %3 to <4 x i32>
  store volatile <4 x i32> %4, <4 x i32>* %c0, align 16
  %5 = load volatile <4 x i32>, <4 x i32>* %c0, align 16
  %6 = extractelement <4 x i32> %5, i32 0
  %7 = load volatile <4 x i32>, <4 x i32>* %c0, align 16
  %8 = extractelement <4 x i32> %7, i32 1
  %9 = add nsw i32 %6, %8
  %10 = load volatile <4 x i32>, <4 x i32>* %c0, align 16
  %11 = extractelement <4 x i32> %10, i32 2
  %12 = add nsw i32 %9, %11
  %13 = load volatile <4 x i32>, <4 x i32>* %c0, align 16
  %14 = extractelement <4 x i32> %13, i32 3
  %15 = add nsw i32 %12, %14
  ret i32 %15
}
```

```
118-165-79-206:input Jonathan$ ~/llvm/test/build/bin/llc
  -march=cpu0 -mcpu=cpu032II -relocation-model=pic -filetype=asm ch7_1_vector.bc
  -o -
  .text
  .section .mdebug.abi32
  .previous
  .file "ch7_1_vector.bc"
  .globl  _Z16test_cmplt_shortv
  .p2align  2
  .type _Z16test_cmplt_shortv,@function
  .ent  _Z16test_cmplt_shortv   # @_Z16test_cmplt_shortv
_Z16test_cmplt_shortv:
  .frame  $fp,48,$lr
  .mask   0x00000000,0
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -48
  addiu $2, $zero, 3
  st  $2, 44($sp)
```

(continues on next page)

(continued from previous page)

```
addiu $2, $zero, 1
st  $2, 36($sp)
addiu $2, $zero, 0
st  $2, 32($sp)
addiu $2, $zero, 2
st  $2, 40($sp)
st  $2, 28($sp)
st  $2, 24($sp)
st  $2, 20($sp)
st  $2, 16($sp)
ld  $2, 32($sp)
ld  $3, 44($sp)
ld  $4, 40($sp)
ld  $5, 36($sp)
ld  $t9, 20($sp)
slt $5, $5, $t9
ld  $t9, 24($sp)
slt $4, $4, $t9
ld  $t9, 28($sp)
slt $3, $3, $t9
shl $3, $3, 31
sra $3, $3, 31
ld  $t9, 16($sp)
st  $3, 12($sp)
shl $3, $4, 31
sra $3, $3, 31
st  $3, 8($sp)
shl $3, $5, 31
sra $3, $3, 31
st  $3, 4($sp)
slt $2, $2, $t9
shl $2, $2, 31
sra $2, $2, 31
st  $2, 0($sp)
ld  $2, 12($sp)
ld  $2, 8($sp)
ld  $2, 4($sp)
ld  $2, 0($sp)
ld  $3, 4($sp)
addu  $2, $2, $3
ld  $3, 12($sp)
ld  $3, 8($sp)
ld  $3, 0($sp)
ld  $3, 8($sp)
addu  $2, $2, $3
ld  $3, 12($sp)
ld  $3, 4($sp)
ld  $3, 0($sp)
ld  $3, 12($sp)
addu  $2, $2, $3
ld  $3, 8($sp)
ld  $3, 4($sp)
```

(continues on next page)

(continued from previous page)

```
  ld  $3, 0($sp)
  addiu $sp, $sp, 48
  ret $lr
  .set  macro
  .set  reorder
  .end  _Z16test_cmplt_shortv
$func_end0:
  .size _Z16test_cmplt_shortv, ($func_end0)-_Z16test_cmplt_shortv


  .ident  "Apple LLVM version 7.0.0 (clang-700.1.76)"
  .section  ".note.GNU-stack","",@progbits
```

Since test_longlong_shift2() of ch7_1_vector.cpp needs implementation storeRegToStack() of Cpu0SEInstInfo.cpp, at this point it cannot be verified.

lbdex/chapters/Chapter7_1/Cpu0ISelLowering.h

```
    /// getSetCCResultType - get the ISD::SETCC result ValueType
    EVT getSetCCResultType(const DataLayout &DL, LLVMContext &Context,
                           EVT VT) const override;
```

lbdex/chapters/Chapter7_1/Cpu0ISelLowering.cpp

```
EVT Cpu0TargetLowering::getSetCCResultType(const DataLayout &, LLVMContext &,
                                           EVT VT) const {
  if (!VT.isVector())
    return MVT::i32;
  return VT.changeVectorElementTypeToInteger();
}
```

CHAPTER

EIGHT

CONTROL FLOW STATEMENTS

- *Pipeline architecture*
- *Control flow statement*
- *Long branch support*
- *Cpu0 backend Optimization: Remove useless JMP*
- *Fill Branch Delay Slot*
- *Conditional instruction*
- *Phi node*
- *RISC CPU knowledge*

This chapter illustrates the corresponding IR for control flow statements, such as **"if else"**, **"while"** and **"for"** loop statements in C, and how to translate these control flow statements of llvm IR into Cpu0 instructions in section I. In section Remove useless JMP, an optimization pass of control flow for backend is introduced. It's a simple tutorial program to let readers know how to add a backend optimization pass and program it. section Conditional instruction, includes the Conditional Instructions Handling since clang will generate specific IRs, select and select_cc, to support the backend optimiation in control flow statement.

8.1 Pipeline architecture

The following figures are from book "Computer Architecture A Quantitative Approach Fourth Edition".

- IF: Instruction fetch cycle; ID: Instruction decode/register fetch cycle; EX: Execution/effective address cycle; MEM: Memory access; WB: Write-back cycle.

Multibanked Caches as Fig. 8.3 to Increase Cache Bandwidth.

Block size in cache L1 is usualy 16-256 bytes. Euipped with with multibanked Caches can provide super-pipeline as Fig. 8.2 and superscalar (multi-issues pipeline) archtectures for fetching (4*block-size/instruction-size) instructions in a cycle.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

Figure A.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its 5-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write back.

Fig. 8.1: 5 stages of pipeline

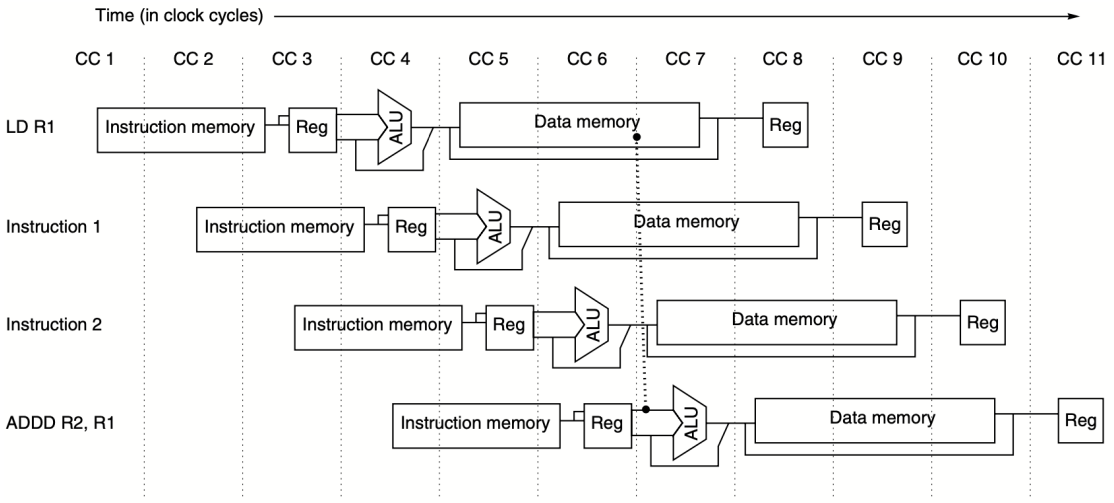


Figure A.38 The structure of the R4000 integer pipeline leads to a 2-cycle load delay. A 2-cycle delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

Fig. 8.2: Super pipeline

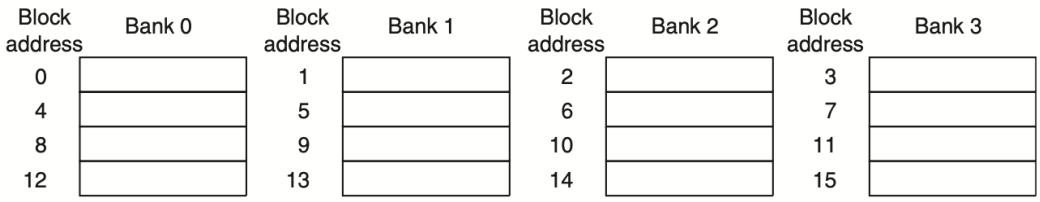


Figure 5.6 Four-way interleaved cache banks using block addressing. Assuming 64 bytes per blocks, each of these addresses would be multiplied by 64 to get byte addressing.

Fig. 8.3: Interleaved cache banks

8.2 Control flow statement

Run ch8_1_1.cpp with clang will get result as follows,

lbdex/input/ch8_1_1.cpp

```
int test_ifctrl()
{
  unsigned int a = 0;

  if (a == 0) {
    a++; // a = 1
  }

  return a;
}
```

```
  ...
  %0 = load i32* %a, align 4
  %cmp = icmp eq i32 %0, 0
  br i1 %cmp, label %if.then, label %if.end

; <label>:3:                                      ; preds = %0
  %1 = load i32* %a, align 4
  %inc = add i32 %1, 1
  store i32 %inc, i32* %a, align 4
  br label %if.end
  ...
```

The **"icmp ne"** stands for integer compare NotEqual, **"slt"** stands for Set Less Than, **"sle"** stands for Set Less or Equal. Run version Chapter8_1/ with `llc -view-isel-dags` or `-debug` option, you can see the **if** statement is translated into (br (brcond (%1, setcc(%2, Constant<c>, setne)), BasicBlock_02), BasicBlock_01). Ignore %1, then we will get the form (br (brcond (setcc(%2, Constant<c>, setne)), BasicBlock_02), BasicBlock_01). For explanation, listing the IR DAG as follows,

```
Optimized legalized selection DAG: BB#0 '_Z11test_ifctrlv:entry'
  SelectionDAG has 12 nodes:
    ...
    t5: i32,ch = load<Volatile LD4[%a]> t4, FrameIndex:i32<0>, undef:i32
        t16: i32 = setcc t5, Constant:i32<0>, setne:ch
      t11: ch = brcond t5:1, t16, BasicBlock:ch<if.end 0x10305a338>
    t13: ch = br t11, BasicBlock:ch<if.then 0x10305a288>
```

We want to translate them into Cpu0 instruction DAGs as follows,

```
addiu %3, ZERO, Constant<c>
cmp %2, %3
jne BasicBlock_02
jmp BasicBlock_01
```

For the last IR br, we translate unconditional branch (br BasicBlock_01) into jmp BasicBlock_01 by the following pattern definition,

lbdex/chapters/Chapter8_1/Cpu0InstrInfo.td

```
// Unconditional branch, such as JMP
let Predicates = [Ch8_1] in {
class UncondBranch<bits<8> op, string instr_asm>:
  FJ<op, (outs), (ins jmptarget:$addr),
             !strconcat(instr_asm, "\t$addr"), [(br bb:$addr)], IIBranch> {
  let isBranch = 1;
  let isTerminator = 1;
  let isBarrier = 1;
  let hasDelaySlot = 0;
}
}
```

```
...
def JMP     : UncondBranch<0x26, "jmp">;
```

The pattern [(br bb:$imm24)] in class UncondBranch is translated into jmp machine instruction. The translation for the pair Cpu0 instructions, **cmp** and **jne**, is not happened before this chapter. To solve this chained IR to machine instructions translation, we define the following pattern,

lbdex/chapters/Chapter8_1/Cpu0InstrInfo.td

```
// brcond patterns
multiclass BrcondPatsCmp<RegisterClass RC, Instruction JEQOp, Instruction JNEOp,
  Instruction JLTOp, Instruction JGTOp, Instruction JLEOp, Instruction JGEOp,
  Instruction CMPOp> {
...
def : Pat<(brcond (i32 (setne RC:$lhs, RC:$rhs)), bb:$dst),
          (JNEOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;
...
def : Pat<(brcond RC:$cond, bb:$dst),
```

(continues on next page)

(continued from previous page)

```
          (JNEOp (CMPOp RC:$cond, ZERORegs), bb:$dst)>;
...
}
```

Since the aboved BrcondPats pattern uses RC (Register Class) as operand, the following ADDiu pattern defined in Chapter2 will generate instruction **addiu** before the instruction **cmp** for the first IR, **setcc(%2, Constant<c>, setne)**, as above.

lbdex/chapters/Chapter2/Cpu0InstrInfo.td

```
// Small immediates
def : Pat<(i32 immSExt16:$in),
          (ADDiu ZERO, imm:$in)>;
```

The definition of BrcondPats supports setne, seteq, setlt, ..., register operand compare and setult, setugt, ..., for unsigned int type. In addition to seteq and setne, we define setueq and setune by refering Mips code, even though we don't find how to generate setune IR from C language. We have tried to define unsigned int type, but clang still generates setne instead of setune. The order of Pattern Search is from the order of their appearing in context. The last pattern (brcond RC:$cond, bb:$dst) means branch to $dst if $cond != 0. So we set the corresponding translation to (JNEOp (CMPOp RC:$cond, ZERORegs), bb:$dst).

The CMP instruction will set the result to register SW, and then JNE check the condition based on SW status as Fig. 8.4. Since SW belongs to a different register class, it will be correct even an instruction is inserted between CMP and JNE as follows,

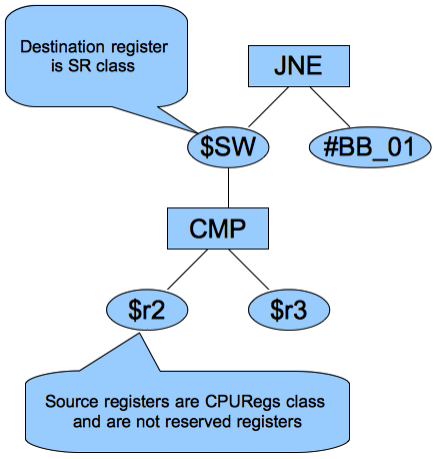


Fig. 8.4: JNE (CMP $r2, $r3),

```
cmp %2, %3
addiu $r1, $r2, 3   // $r1 register never be allocated to $SW because in
                    //  class ArithLogicI, GPROut is the output register
                    //  class and the GPROut is defined without $SW in
                    //  Cpu0RegisterInforGPROutForOther.td
jne BasicBlock_02
```

The reserved registers setting by the following function code we defined before,

lbdex/chapters/Chapter3_1/Cpu0RegisterInfo.cpp

```
BitVector Cpu0RegisterInfo::
getReservedRegs(const MachineFunction &MF) const {
//@getReservedRegs body {
  static const uint16_t ReservedCPURegs[] = {
    Cpu0::ZERO, Cpu0::AT, Cpu0::SP, Cpu0::LR, /*Cpu0::SW, */Cpu0::PC
  };
  BitVector Reserved(getNumRegs());

  for (unsigned I = 0; I < array_lengthof(ReservedCPURegs); ++I)
    Reserved.set(ReservedCPURegs[I]);

  return Reserved;
}
```

Although the following definition in Cpu0RegisterInfo.td has no real effect in Reserved Registers, it's better to comment the Reserved Registers in it for readability. Setting SW in both register classes CPURegs and SR to allow access SW by RISC instructions like `andi`, and allow programmers use traditional assembly instruction `cmp`. The copyPhysReg() is called when both DestReg and SrcReg are belonging to different Register Classes.

lbdex/chapters/Chapter2/Cpu0RegisterInfo.td

```
//===----------------------------------------------------------------------===//

def CPURegs : RegisterClass<"Cpu0", [i32], 32, (add
  // Reserved
  ZERO, AT,
  // Return Values and Arguments
  V0, V1, A0, A1,
  // Not preserved across procedure calls
  T9, T0, T1,
  // Callee save
  S0, S1,
  // Reserved
  GP, FP,
  SP, LR, SW)>;
```

```
def SR     : RegisterClass<"Cpu0", [i32], 32, (add SW)>;
```

lbdex/chapters/Chapter2/Cpu0RegisterInfoGPROutForOther.td

```
//===----------------------------------------------------------------------===//
// Register Classes
//===----------------------------------------------------------------------===//

def GPROut : RegisterClass<"Cpu0", [i32], 32, (add (sub CPURegs, SW))>;
```

Chapter8_1/ include support for control flow statement. Run with it as well as the following `llc` option, you will get the obj file. Dump it's content by gobjdump or hexdump after as follows,

```
  118-165-79-206:input Jonathan$ /Users/Jonathan/llvm/test/
  build/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic
  -filetype=asm ch8_1_1.bc -o -
  ...
  ld  $4, 36($fp)
  cmp $sw, $4, $3
  jne $BB0_2
  jmp $BB0_1
$BB0_1:                                 # %if.then
  ld  $4, 36($fp)
  addiu $4, $4, 1
  st  $4, 36($fp)
$BB0_2:                                 # %if.end
  ld  $4, 32($fp)
  ...
```

```
118-165-79-206:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic
-filetype=obj ch8_1_1.bc -o ch8_1_1.cpu0.o

118-165-79-206:input Jonathan$ hexdump ch8_1_1.cpu0.o
    // jmp offset is 0x10=16 bytes which is correct
0000080 ..................................... 10 43 00 00
0000090 31 00 00 10 36 00 00 00 ........................
```

The immediate value of jne (op 0x31) is 16; The offset between jne and $BB0_2 is 20 (5 words = 5*4 bytes). Suppose the jne address is X, then the label $BB0_2 is X+20. Cpu0's instruction set is designed as a RISC CPU with 5 stages of pipeline just like 5 stages of Mips. Cpu0 do branch instruction execution at decode stage which like mips too. After the jne instruction fetched, the PC (Program Counter) is X+4 since cpu0 update PC at fetch stage. The $BB0_2 address is equal to PC+16 for the jne branch instruction execute at decode stage. List and explain this again as follows,

```
              // Fetch instruction stage for jne instruction. The fetch stage
              // can be divided into 2 cycles. First cycle fetch the
              // instruction. Second cycle adjust PC = PC+4.
  jne $BB0_2  // Do jne compare in decode stage. PC = X+4 at this stage.
              // When jne immediate value is 16, PC = PC+16. It will fetch
              //  X+20 which equal to label $BB0_2 instruction, ld $4, 32($sp).
  nop
$BB0_1:                                 # %if.then
  ld  $4, 36($fp)
```

(continues on next page)

(continued from previous page)

```
  addiu $4, $4, 1
  st  $4, 36($fp)
$BB0_2:                                 # %if.end
  ld  $4, 32($fp)
```

If Cpu0 do **"jne"** in execution stage, then we should set PC=PC+12, offset of ($BB0_2, jne $BB02) – 8, in this example.

In reality, the conditional branch is important in performance of CPU design. According bench mark information, every 7 instructions will meet 1 branch instruction in average. The cpu032I spends 2 instructions in conditional branch, (jne(cmp...)), while cpu032II use one instruction (bne) as follws,

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic
-filetype=asm ch8_1_1.bc -o -
  ...
      cmp     $sw, $4, $3
      jne     $sw, $BB0_2
      jmp     $BB0_1
$BB0_1:

JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -mcpu=cpu032II -relocation-model=pic
-filetype=asm ch8_1_1.bc -o -
  ...
      bne     $4, $zero, $BB0_2
      jmp     $BB0_1
$BB0_1:
```

Beside brcond explained in this section, above code also include DAG opcode **br_jt** and label **JumpTable** which occurs during DAG translation for some kind of program.

The ch8_1_ctrl.cpp include **"nest if"** **"for loop"**, **"while loop"**, **"continue"**, **"break"** and **"goto"**. The ch8_1_br_jt.cpp is for **br_jt** and **JumpTable** test. The ch8_1_blockaddr.cpp is for **blockaddress** and **indirectbr** test. You can run with them if you like to test more.

List the control flow statements of C, IR, DAG and Cpu0 instructions as the following table.

Table 8.1: Control flow statements of C, IR, DAG and Cpu0 instructions

C	if, else, for, while, goto, switch, break
IR	(icmp + (eq, ne, sgt, sge, slt, sle)0 + br
DAG	(seteq, setne, setgt, setge, setlt, setle) + brcond,
•	(setueq, setune, setugt, setuge, setult, setule) + brcond
cpu032I	CMP + (JEQ, JNE, JGT, JGE, JLT, JLE)
cpu032II	(SLT, SLTu, SLTi, SLTiu) + (BEG, BNE)

8.3 Long branch support

As last section, cpu032II uses beq and bne to improve performance but the jump offset reduces from 24 bits to 16 bits. If program exists more than 16 bits, cpu032II will fail to generate code. Mips backend has solution and Cpu0 hire the solution from it.

To support long branch the following code added in Chapter8_1.

lbdex/chapters/Chapter8_2/CMakeLists.txt

```
  Cpu0LongBranch.cpp
```

lbdex/chapters/Chapter8_2/Cpu0.h

```
  FunctionPass *createCpu0LongBranchPass(Cpu0TargetMachine &TM);
```

lbdex/chapters/Chapter8_2/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
unsigned Cpu0MCCodeEmitter::
getJumpTargetOpValue(const MCInst &MI, unsigned OpNo,
                     SmallVectorImpl<MCFixup> &Fixups,
                     const MCSubtargetInfo &STI) const {
```

```
  if (Opcode == Cpu0::JMP || Opcode == Cpu0::BAL)
```

```
  ...
}
```

lbdex/chapters/Chapter8_2/Cpu0AsmPrinter.h

```
  bool isLongBranchPseudo(int Opcode) const;
```

lbdex/chapters/Chapter8_2/Cpu0AsmPrinter.cpp

```
//- emitInstruction() must exists or will have run time error.
void Cpu0AsmPrinter::emitInstruction(const MachineInstr *MI) {
```

```
    if (I->isPseudo() && !isLongBranchPseudo(I->getOpcode()))
```

```
  ...
}
```

```
bool Cpu0AsmPrinter::isLongBranchPseudo(int Opcode) const {
  return (Opcode == Cpu0::LONG_BRANCH_LUi
          || Opcode == Cpu0::LONG_BRANCH_ADDiu);
}
```

lbdex/chapters/Chapter8_2/Cpu0InstrInfo.h

```
  virtual unsigned getOppositeBranchOpc(unsigned Opc) const = 0;
```

lbdex/chapters/Chapter8_2/Cpu0InstrInfo.td

```
let Predicates = [Ch8_2] in {
// We need these two pseudo instructions to avoid offset calculation for long
// branches.  See the comment in file Cpu0LongBranch.cpp for detailed
// explanation.

// Expands to: lui $dst, %hi($tgt - $baltgt)
def LONG_BRANCH_LUi : Cpu0Pseudo<(outs GPROut:$dst),
  (ins jmptarget:$tgt, jmptarget:$baltgt), "", []>;

// Expands to: addiu $dst, $src, %lo($tgt - $baltgt)
def LONG_BRANCH_ADDiu : Cpu0Pseudo<(outs GPROut:$dst),
  (ins GPROut:$src, jmptarget:$tgt, jmptarget:$baltgt), "", []>;
}
```

```
let isBranch = 1, isTerminator = 1, isBarrier = 1,
    hasDelaySlot = 0, Defs = [LR] in
def BAL: FJ<0x3A, (outs), (ins jmptarget:$addr), "bal\t$addr", [], IIBranch>,
             Requires<[HasSlt]>;
```

lbdex/chapters/Chapter8_2/Cpu0LongBranch.cpp

```
//===-- Cpu0LongBranch.cpp - Emit long branches ---------------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This pass expands a branch or jump instruction into a long branch if its
// offset is too large to fit into its immediate field.
//
// FIXME: Fix pc-region jump instructions which cross 256MB segment boundaries.
//===----------------------------------------------------------------------===//

// In 9.0.0 rename MipsLongBranch.cpp to MipsBranchExpansion.cpp
```

(continues on next page)

(continued from previous page)

```
#include "Cpu0.h"

#if CH >= CH8_2

#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "Cpu0MachineFunction.h"
#include "Cpu0TargetMachine.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/TargetInstrInfo.h"
#include "llvm/CodeGen/TargetRegisterInfo.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/MathExtras.h"
#include "llvm/Target/TargetMachine.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-long-branch"

STATISTIC(LongBranches, "Number of long branches.");

static cl::opt<bool> ForceLongBranch(
  "force-cpu0-long-branch",
  cl::init(false),
  cl::desc("CPU0: Expand all branches to long format."),
  cl::Hidden);

namespace {
  typedef MachineBasicBlock::iterator Iter;
  typedef MachineBasicBlock::reverse_iterator ReverseIter;

  struct MBBInfo {
    uint64_t Size, Address;
    bool HasLongBranch;
    MachineInstr *Br;

    MBBInfo() : Size(0), HasLongBranch(false), Br(nullptr) {}
  };

  class Cpu0LongBranch : public MachineFunctionPass {

  public:
    static char ID;
    Cpu0LongBranch(TargetMachine &tm)
        : MachineFunctionPass(ID), TM(tm), IsPIC(TM.isPositionIndependent()),
          ABI(static_cast<const Cpu0TargetMachine &>(TM).getABI()) {}

    StringRef getPassName() const override {
      return "Cpu0 Long Branch";
```

(continues on next page)

(continued from previous page)

```
    }

    bool runOnMachineFunction(MachineFunction &F) override;

  private:
    void splitMBB(MachineBasicBlock *MBB);
    void initMBBInfo();
    int64_t computeOffset(const MachineInstr *Br);
    void replaceBranch(MachineBasicBlock &MBB, Iter Br, const DebugLoc &DL,
                       MachineBasicBlock *MBBOpnd);
    void expandToLongBranch(MBBInfo &Info);

    const TargetMachine &TM;
    MachineFunction *MF;
    SmallVector<MBBInfo, 16> MBBInfos;
    bool IsPIC;
    Cpu0ABIInfo ABI;
    unsigned LongBranchSeqSize;
  };

  char Cpu0LongBranch::ID = 0;
} // end of anonymous namespace

/// createCpu0LongBranchPass - Returns a pass that converts branches to long
/// branches.
FunctionPass *llvm::createCpu0LongBranchPass(Cpu0TargetMachine &tm) {
  return new Cpu0LongBranch(tm);
}

/// Iterate over list of Br's operands and search for a MachineBasicBlock
/// operand.
static MachineBasicBlock *getTargetMBB(const MachineInstr &Br) {
  for (unsigned I = 0, E = Br.getDesc().getNumOperands(); I < E; ++I) {
    const MachineOperand &MO = Br.getOperand(I);

    if (MO.isMBB())
      return MO.getMBB();
  }

  llvm_unreachable("This instruction does not have an MBB operand.");
}

// Traverse the list of instructions backwards until a non-debug instruction is
// found or it reaches E.
static ReverseIter getNonDebugInstr(ReverseIter B, const ReverseIter &E) {
  for (; B != E; ++B)
    if (!B->isDebugValue())
      return B;

  return E;
}
```

(continues on next page)

(continued from previous page)

```
// Split MBB if it has two direct jumps/branches.
void Cpu0LongBranch::splitMBB(MachineBasicBlock *MBB) {
  ReverseIter End = MBB->rend();
  ReverseIter LastBr = getNonDebugInstr(MBB->rbegin(), End);

  // Return if MBB has no branch instructions.
  if ((LastBr == End) ||
      (!LastBr->isConditionalBranch() && !LastBr->isUnconditionalBranch()))
    return;

  ReverseIter FirstBr = getNonDebugInstr(std::next(LastBr), End);

  // MBB has only one branch instruction if FirstBr is not a branch
  // instruction.
  if ((FirstBr == End) ||
      (!FirstBr->isConditionalBranch() && !FirstBr->isUnconditionalBranch()))
    return;

  assert(!FirstBr->isIndirectBranch() && "Unexpected indirect branch found.");

  // Create a new MBB. Move instructions in MBB to the newly created MBB.
  MachineBasicBlock *NewMBB =
    MF->CreateMachineBasicBlock(MBB->getBasicBlock());

  // Insert NewMBB and fix control flow.
  MachineBasicBlock *Tgt = getTargetMBB(*FirstBr);
  NewMBB->transferSuccessors(MBB);
  if (Tgt != getTargetMBB(*LastBr))
    NewMBB->removeSuccessor(Tgt, true);
  MBB->addSuccessor(NewMBB);
  MBB->addSuccessor(Tgt);
  MF->insert(std::next(MachineFunction::iterator(MBB)), NewMBB);

  NewMBB->splice(NewMBB->end(), MBB, LastBr.getReverse(), MBB->end());
}

// Fill MBBInfos.
void Cpu0LongBranch::initMBBInfo() {
  // Split the MBBs if they have two branches. Each basic block should have at
  // most one branch after this loop is executed.
  for (auto &MBB : *MF)
    splitMBB(&MBB);

  MF->RenumberBlocks();
  MBBInfos.clear();
  MBBInfos.resize(MF->size());

  const Cpu0InstrInfo *TII =
      static_cast<const Cpu0InstrInfo *>(MF->getSubtarget().getInstrInfo());
  for (unsigned I = 0, E = MBBInfos.size(); I < E; ++I) {
    MachineBasicBlock *MBB = MF->getBlockNumbered(I);
```

(continues on next page)

(continued from previous page)

```
    // Compute size of MBB.
    for (MachineBasicBlock::instr_iterator MI = MBB->instr_begin();
         MI != MBB->instr_end(); ++MI)
      MBBInfos[I].Size += TII->GetInstSizeInBytes(*MI);

    // Search for MBB's branch instruction.
    ReverseIter End = MBB->rend();
    ReverseIter Br = getNonDebugInstr(MBB->rbegin(), End);

    if ((Br != End) && !Br->isIndirectBranch() &&
        (Br->isConditionalBranch() || (Br->isUnconditionalBranch() && IsPIC)))
      MBBInfos[I].Br = &(*Br.getReverse());
  }
}

// Compute offset of branch in number of bytes.
int64_t Cpu0LongBranch::computeOffset(const MachineInstr *Br) {
  int64_t Offset = 0;
  int ThisMBB = Br->getParent()->getNumber();
  int TargetMBB = getTargetMBB(*Br)->getNumber();

  // Compute offset of a forward branch.
  if (ThisMBB < TargetMBB) {
    for (int N = ThisMBB + 1; N < TargetMBB; ++N)
      Offset += MBBInfos[N].Size;

    return Offset + 4;
  }

  // Compute offset of a backward branch.
  for (int N = ThisMBB; N >= TargetMBB; --N)
    Offset += MBBInfos[N].Size;

  return -Offset + 4;
}

// Replace Br with a branch which has the opposite condition code and a
// MachineBasicBlock operand MBBOpnd.
void Cpu0LongBranch::replaceBranch(MachineBasicBlock &MBB, Iter Br,
                                   const DebugLoc &DL,
                                   MachineBasicBlock *MBBOpnd) {
  const Cpu0InstrInfo *TII = static_cast<const Cpu0InstrInfo *>(
      MBB.getParent()->getSubtarget().getInstrInfo());
  unsigned NewOpc = TII->getOppositeBranchOpc(Br->getOpcode());
  const MCInstrDesc &NewDesc = TII->get(NewOpc);

  MachineInstrBuilder MIB = BuildMI(MBB, Br, DL, NewDesc);

  for (unsigned I = 0, E = Br->getDesc().getNumOperands(); I < E; ++I) {
    MachineOperand &MO = Br->getOperand(I);

    if (!MO.isReg()) {
```

(continues on next page)

(continued from previous page)

```
      assert(MO.isMBB() && "MBB operand expected.");
      break;
    }

    MIB.addReg(MO.getReg());
  }

  MIB.addMBB(MBBOpnd);

  if (Br->hasDelaySlot()) {
    // Bundle the instruction in the delay slot to the newly created branch
    // and erase the original branch.
    assert(Br->isBundledWithSucc());
    MachineBasicBlock::instr_iterator II = Br.getInstrIterator();
    MIBundleBuilder(&*MIB).append((++II)->removeFromBundle());
  }
  Br->eraseFromParent();
}

// Expand branch instructions to long branches.
// TODO: This function has to be fixed for beqz16 and bnez16, because it
// currently assumes that all branches have 16-bit offsets, and will produce
// wrong code if branches whose allowed offsets are [-128, -126, ..., 126]
// are present.
void Cpu0LongBranch::expandToLongBranch(MBBInfo &I) {
  MachineBasicBlock::iterator Pos;
  MachineBasicBlock *MBB = I.Br->getParent(), *TgtMBB = getTargetMBB(*I.Br);
  DebugLoc DL = I.Br->getDebugLoc();
  const BasicBlock *BB = MBB->getBasicBlock();
  MachineFunction::iterator FallThroughMBB = ++MachineFunction::iterator(MBB);
  MachineBasicBlock *LongBrMBB = MF->CreateMachineBasicBlock(BB);
  const Cpu0Subtarget &Subtarget =
      static_cast<const Cpu0Subtarget &>(MF->getSubtarget());
  const Cpu0InstrInfo *TII =
      static_cast<const Cpu0InstrInfo *>(Subtarget.getInstrInfo());

  MF->insert(FallThroughMBB, LongBrMBB);
  MBB->replaceSuccessor(TgtMBB, LongBrMBB);

  if (IsPIC) {
    MachineBasicBlock *BalTgtMBB = MF->CreateMachineBasicBlock(BB);
    MF->insert(FallThroughMBB, BalTgtMBB);
    LongBrMBB->addSuccessor(BalTgtMBB);
    BalTgtMBB->addSuccessor(TgtMBB);

    unsigned BalOp = Cpu0::BAL;

    // $longbr:
    //  addiu $sp, $sp, -8
    //  st $lr, 0($sp)
    //  lui $at, %hi($tgt - $baltgt)
    //  addiu $lr, $lr, %lo($tgt - $baltgt)
```

(continues on next page)

(continued from previous page)

```
  //  bal $baltgt
  //  nop
  // $baltgt:
  //  addu $at, $lr, $at
  //  addiu $sp, $sp, 8
  //  ld $lr, 0($sp)
  //  jr $at
  //  nop
  // $fallthrough:
  //

  Pos = LongBrMBB->begin();

  BuildMI(*LongBrMBB, Pos, DL, TII->get(Cpu0::ADDiu), Cpu0::SP)
    .addReg(Cpu0::SP).addImm(-8);
  BuildMI(*LongBrMBB, Pos, DL, TII->get(Cpu0::ST)).addReg(Cpu0::LR)
    .addReg(Cpu0::SP).addImm(0);

  // LUi and ADDiu instructions create 32-bit offset of the target basic
  // block from the target of BAL instruction.  We cannot use immediate
  // value for this offset because it cannot be determined accurately when
  // the program has inline assembly statements.  We therefore use the
  // relocation expressions %hi($tgt-$baltgt) and %lo($tgt-$baltgt) which
  // are resolved during the fixup, so the values will always be correct.
  //
  // Since we cannot create %hi($tgt-$baltgt) and %lo($tgt-$baltgt)
  // expressions at this point (it is possible only at the MC layer),
  // we replace LUi and ADDiu with pseudo instructions
  // LONG_BRANCH_LUi and LONG_BRANCH_ADDiu, and add both basic
  // blocks as operands to these instructions.  When lowering these pseudo
  // instructions to LUi and ADDiu in the MC layer, we will create
  // %hi($tgt-$baltgt) and %lo($tgt-$baltgt) expressions and add them as
  // operands to lowered instructions.

  BuildMI(*LongBrMBB, Pos, DL, TII->get(Cpu0::LONG_BRANCH_LUi), Cpu0::AT)
    .addMBB(TgtMBB).addMBB(BalTgtMBB);
  BuildMI(*LongBrMBB, Pos, DL, TII->get(Cpu0::LONG_BRANCH_ADDiu), Cpu0::AT)
    .addReg(Cpu0::AT).addMBB(TgtMBB).addMBB(BalTgtMBB);
  MIBundleBuilder(*LongBrMBB, Pos)
      .append(BuildMI(*MF, DL, TII->get(BalOp)).addMBB(BalTgtMBB));

  Pos = BalTgtMBB->begin();

  BuildMI(*BalTgtMBB, Pos, DL, TII->get(Cpu0::ADDu), Cpu0::AT)
    .addReg(Cpu0::LR).addReg(Cpu0::AT);
  BuildMI(*BalTgtMBB, Pos, DL, TII->get(Cpu0::LD), Cpu0::LR)
    .addReg(Cpu0::SP).addImm(0);
  BuildMI(*BalTgtMBB, Pos, DL, TII->get(Cpu0::ADDiu), Cpu0::SP)
    .addReg(Cpu0::SP).addImm(8);

  MIBundleBuilder(*BalTgtMBB, Pos)
    .append(BuildMI(*MF, DL, TII->get(Cpu0::JR)).addReg(Cpu0::AT))
```

(continues on next page)

(continued from previous page)

```
      .append(BuildMI(*MF, DL, TII->get(Cpu0::NOP)));

    assert(LongBrMBB->size() + BalTgtMBB->size() == LongBranchSeqSize);
  } else {
    // $longbr:
    //  jmp $tgt
    //  nop
    // $fallthrough:
    //
    Pos = LongBrMBB->begin();
    LongBrMBB->addSuccessor(TgtMBB);
    MIBundleBuilder(*LongBrMBB, Pos)
      .append(BuildMI(*MF, DL, TII->get(Cpu0::JMP)).addMBB(TgtMBB))
      .append(BuildMI(*MF, DL, TII->get(Cpu0::NOP)));

    assert(LongBrMBB->size() == LongBranchSeqSize);
  }

  if (I.Br->isUnconditionalBranch()) {
    // Change branch destination.
    assert(I.Br->getDesc().getNumOperands() == 1);
    I.Br->RemoveOperand(0);
    I.Br->addOperand(MachineOperand::CreateMBB(LongBrMBB));
  } else
    // Change branch destination and reverse condition.
    replaceBranch(*MBB, I.Br, DL, &*FallThroughMBB);
}

static void emitGPDisp(MachineFunction &F, const Cpu0InstrInfo *TII) {
  MachineBasicBlock &MBB = F.front();
  MachineBasicBlock::iterator I = MBB.begin();
  DebugLoc DL = MBB.findDebugLoc(MBB.begin());
  BuildMI(MBB, I, DL, TII->get(Cpu0::LUi), Cpu0::V0)
    .addExternalSymbol("_gp_disp", Cpu0II::MO_ABS_HI);
  BuildMI(MBB, I, DL, TII->get(Cpu0::ADDiu), Cpu0::V0)
    .addReg(Cpu0::V0).addExternalSymbol("_gp_disp", Cpu0II::MO_ABS_LO);
  MBB.removeLiveIn(Cpu0::V0);
}

bool Cpu0LongBranch::runOnMachineFunction(MachineFunction &F) {
  const Cpu0Subtarget &STI =
      static_cast<const Cpu0Subtarget &>(F.getSubtarget());
  const Cpu0InstrInfo *TII =
      static_cast<const Cpu0InstrInfo *>(STI.getInstrInfo());
  LongBranchSeqSize =
      !IsPIC ? 2 : 10;

  if (!STI.enableLongBranchPass())
    return false;
  if (IsPIC && static_cast<const Cpu0TargetMachine &>(TM).getABI().IsO32() &&
      F.getInfo<Cpu0FunctionInfo>()->globalBaseRegSet())
    emitGPDisp(F, TII);
```

(continues on next page)

(continued from previous page)

```
  MF = &F;
  initMBBInfo();

  SmallVectorImpl<MBBInfo>::iterator I, E = MBBInfos.end();
  bool EverMadeChange = false, MadeChange = true;

  while (MadeChange) {
    MadeChange = false;

    for (I = MBBInfos.begin(); I != E; ++I) {
      // Skip if this MBB doesn't have a branch or the branch has already been
      // converted to a long branch.
      if (!I->Br || I->HasLongBranch)
        continue;

      int ShVal = 4;
      int64_t Offset = computeOffset(I->Br) / ShVal;

      // Check if offset fits into 16-bit immediate field of branches.
      if (!ForceLongBranch && isInt<16>(Offset))
        continue;

      I->HasLongBranch = true;
      I->Size += LongBranchSeqSize * 4;
      ++LongBranches;
      EverMadeChange = MadeChange = true;
    }
  }

  if (!EverMadeChange)
    return true;

  // Compute basic block addresses.
  if (IsPIC) {
    uint64_t Address = 0;

    for (I = MBBInfos.begin(); I != E; Address += I->Size, ++I)
      I->Address = Address;
  }

  // Do the expansion.
  for (I = MBBInfos.begin(); I != E; ++I)
    if (I->HasLongBranch)
      expandToLongBranch(*I);

  MF->RenumberBlocks();

  return true;
}

#endif //#if CH >= CH8_2
```

lbdex/chapters/Chapter8_2/Cpu0MCInstLower.h

```
  MCOperand createSub(MachineBasicBlock *BB1, MachineBasicBlock *BB2,
                      Cpu0MCExpr::Cpu0ExprKind Kind) const;
  void lowerLongBranchLUi(const MachineInstr *MI, MCInst &OutMI) const;
  void lowerLongBranchADDiu(const MachineInstr *MI, MCInst &OutMI,
                            int Opcode,
                            Cpu0MCExpr::Cpu0ExprKind Kind) const;
  bool lowerLongBranch(const MachineInstr *MI, MCInst &OutMI) const;
```

lbdex/chapters/Chapter8_2/Cpu0MCInstLower.cpp

```
MCOperand Cpu0MCInstLower::createSub(MachineBasicBlock *BB1,
                                     MachineBasicBlock *BB2,
                                     Cpu0MCExpr::Cpu0ExprKind Kind) const {
  const MCSymbolRefExpr *Sym1 = MCSymbolRefExpr::create(BB1->getSymbol(), *Ctx);
  const MCSymbolRefExpr *Sym2 = MCSymbolRefExpr::create(BB2->getSymbol(), *Ctx);
  const MCBinaryExpr *Sub = MCBinaryExpr::createSub(Sym1, Sym2, *Ctx);

  return MCOperand::createExpr(Cpu0MCExpr::create(Kind, Sub, *Ctx));
}

void Cpu0MCInstLower::
lowerLongBranchLUi(const MachineInstr *MI, MCInst &OutMI) const {
  OutMI.setOpcode(Cpu0::LUi);

  // Lower register operand.
  OutMI.addOperand(LowerOperand(MI->getOperand(0)));

  // Create %hi($tgt-$baltgt).
  OutMI.addOperand(createSub(MI->getOperand(1).getMBB(),
                             MI->getOperand(2).getMBB(),
                             Cpu0MCExpr::CEK_ABS_HI));
}

void Cpu0MCInstLower::
lowerLongBranchADDiu(const MachineInstr *MI, MCInst &OutMI, int Opcode,
                     Cpu0MCExpr::Cpu0ExprKind Kind) const {
  OutMI.setOpcode(Opcode);

  // Lower two register operands.
  for (unsigned I = 0, E = 2; I != E; ++I) {
    const MachineOperand &MO = MI->getOperand(I);
    OutMI.addOperand(LowerOperand(MO));
  }

  // Create %lo($tgt-$baltgt) or %hi($tgt-$baltgt).
  OutMI.addOperand(createSub(MI->getOperand(2).getMBB(),
                             MI->getOperand(3).getMBB(), Kind));
}
```

(continues on next page)

(continued from previous page)

```
bool Cpu0MCInstLower::lowerLongBranch(const MachineInstr *MI,
                                      MCInst &OutMI) const {
  switch (MI->getOpcode()) {
  default:
    return false;
  case Cpu0::LONG_BRANCH_LUi:
    lowerLongBranchLUi(MI, OutMI);
    return true;
  case Cpu0::LONG_BRANCH_ADDiu:
    lowerLongBranchADDiu(MI, OutMI, Cpu0::ADDiu,
                         Cpu0MCExpr::CEK_ABS_LO);
    return true;
  }
}
```

```
void Cpu0MCInstLower::Lower(const MachineInstr *MI, MCInst &OutMI) const {
```

```
  if (lowerLongBranch(MI, OutMI))
    return;
```

```
  ...
}
```

lbdex/chapters/Chapter8_2/Cpu0SEInstrInfo.h

```
  unsigned getOppositeBranchOpc(unsigned Opc) const override;
```

lbdex/chapters/Chapter8_2/Cpu0SEInstrInfo.cpp

```
/// getOppositeBranchOpc - Return the inverse of the specified
/// opcode, e.g. turning BEQ to BNE.
unsigned Cpu0SEInstrInfo::getOppositeBranchOpc(unsigned Opc) const {
  switch (Opc) {
  default:           llvm_unreachable("Illegal opcode!");
  case Cpu0::BEQ:    return Cpu0::BNE;
  case Cpu0::BNE:    return Cpu0::BEQ;
  }
}
```

lbdex/chapters/Chapter8_2/Cpu0TargetMachine.cpp

```
  void addPreEmitPass() override;
```

```
// Implemented by targets that want to run passes immediately before
// machine code is emitted. return true if -print-machineinstrs should
// print out the code after the passes.
void Cpu0PassConfig::addPreEmitPass() {
  Cpu0TargetMachine &TM = getCpu0TargetMachine();
```

```
  addPass(createCpu0LongBranchPass(TM));
  return;
}
```

The code of Chapter8_2 will compile the following example as follows,

lbdex/input/ch8_2_longbranch.cpp

```
int test_longbranch()
{
  volatile int a = 2;
  volatile int b = 1;
  int result = 0;

  if (a < b)
    result = 1;
  return result;
}
```

```
118-165-78-10:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -mcpu=cpu032II -relocation-model=pic -filetype=asm
-force-cpu0-long-branch ch8_2_longbranch.bc -o -
  ...
        .text
        .section .mdebug.abi32
        .previous
        .file "ch8_2_longbranch.bc"
        .globl        _Z15test_longbranchv
        .align        2
        .type _Z15test_longbranchv,@function
        .ent  _Z15test_longbranchv    # @_Z15test_longbranchv
_Z15test_longbranchv:
        .frame        $fp,16,$lr
        .mask         0x00001000,-4
        .set  noreorder
        .set  nomacro
# BB#0:
        addiu $sp, $sp, -16
        st    $fp, 12($sp)            # 4-byte Folded Spill
        move   $fp, $sp
```

(continues on next page)

(continued from previous page)

```
        addiu $2, $zero, 1
        st    $2, 8($fp)
        addiu $3, $zero, 2
        st    $3, 4($fp)
        addiu $3, $zero, 0
        st    $3, 0($fp)
        ld    $3, 8($fp)
        ld    $4, 4($fp)
        slt   $3, $3, $4
        bne   $3, $zero, .LBB0_3
        nop
# BB#1:
        addiu $sp, $sp, -8
        st    $lr, 0($sp)
        lui   $1, %hi(.LBB0_4-.LBB0_2)
        addiu $1, $1, %lo(.LBB0_4-.LBB0_2)
        bal   .LBB0_2
.LBB0_2:
        addu  $1, $lr, $1
        ld    $lr, 0($sp)
        addiu $sp, $sp, 8
        jr    $1
        nop
.LBB0_3:
        st    $2, 0($fp)
.LBB0_4:
        ld    $2, 0($fp)
        move   $sp, $fp
        ld    $fp, 12($sp)            # 4-byte Folded Reload
        addiu $sp, $sp, 16
        ret   $lr
        nop
        .set  macro
        .set  reorder
        .end  _Z15test_longbranchv
$func_end0:
        .size _Z15test_longbranchv, ($func_end0)-_Z15test_longbranchv
```

8.4 Cpu0 backend Optimization: Remove useless JMP

LLVM uses functional pass both in code generation and optimization. Following the 3 tiers of compiler architecture, LLVM do most optimization in middle tier of LLVM IR, SSA form. Beyond middle tier optimization, there are opportunities in optimization which depend on backend features. The "fill delay slot" in Mips is an example of backend optimization used in pipeline RISC machine. You can port it from Mips if your backend is a pipeline RISC with delay slot. In this section, we apply the "delete useless jmp" in Cpu0 backend optimization. This algorithm is simple and effective to be a perfect tutorial in optimization. Through this example, you can understand how to add an optimization pass and coding your complicated optimization algorithm on your backend in real project.

Chapter8_2/ supports "delete useless jmp" optimization algorithm which add codes as follows,

lbdex/chapters/Chapter8_2/CMakeLists.txt

```
  Cpu0DelUselessJMP.cpp
```

lbdex/chapters/Chapter8_2/Cpu0.h

```
  FunctionPass *createCpu0DelJmpPass(Cpu0TargetMachine &TM);
```

lbdex/chapters/Chapter8_2/Cpu0TargetMachine.cpp

```
// Implemented by targets that want to run passes immediately before
// machine code is emitted. return true if -print-machineinstrs should
// print out the code after the passes.
void Cpu0PassConfig::addPreEmitPass() {
  Cpu0TargetMachine &TM = getCpu0TargetMachine();
```

```
  addPass(createCpu0DelJmpPass(TM));
```

```
}
```

lbdex/chapters/Chapter8_2/Cpu0DelUselessJMP.cpp

```
//===-- Cpu0DelUselessJMP.cpp - Cpu0 DelJmp -------------------------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// Simple pass to fills delay slots with useful instructions.
//
//===----------------------------------------------------------------------===//

#include "Cpu0.h"
#if CH >= CH8_2

#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/TargetInstrInfo.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/ADT/SmallSet.h"
#include "llvm/ADT/Statistic.h"

using namespace llvm;

```

(continues on next page)

(continued from previous page)

```
#define DEBUG_TYPE "del-jmp"

STATISTIC(NumDelJmp, "Number of useless jmp deleted");

static cl::opt<bool> EnableDelJmp(
  "enable-cpu0-del-useless-jmp",
  cl::init(true),
  cl::desc("Delete useless jmp instructions: jmp 0."),
  cl::Hidden);

namespace {
  struct DelJmp : public MachineFunctionPass {
    static char ID;
    DelJmp(TargetMachine &tm)
      : MachineFunctionPass(ID) { }

    StringRef getPassName() const override {
      return "Cpu0 Del Useless jmp";
    }

    bool runOnMachineBasicBlock(MachineBasicBlock &MBB, MachineBasicBlock &MBBN);
    bool runOnMachineFunction(MachineFunction &F) override {
      bool Changed = false;
      if (EnableDelJmp) {
        MachineFunction::iterator FJ = F.begin();
        if (FJ != F.end())
          FJ++;
        if (FJ == F.end())
          return Changed;
        for (MachineFunction::iterator FI = F.begin(), FE = F.end();
             FJ != FE; ++FI, ++FJ)
          // In STL style, F.end() is the dummy BasicBlock() like '\0' in
          //  C string.
          // FJ is the next BasicBlock of FI; When FI range from F.begin() to
          //  the PreviousBasicBlock of F.end() call runOnMachineBasicBlock().
          Changed |= runOnMachineBasicBlock(*FI, *FJ);
      }
      return Changed;
    }

  };
  char DelJmp::ID = 0;
} // end of anonymous namespace

bool DelJmp::
runOnMachineBasicBlock(MachineBasicBlock &MBB, MachineBasicBlock &MBBN) {
  bool Changed = false;

  MachineBasicBlock::iterator I = MBB.end();
  if (I != MBB.begin())
    I--;        // set I to the last instruction
  else
```

(continues on next page)

(continued from previous page)

```
    return Changed;

  if (I->getOpcode() == Cpu0::JMP && I->getOperand(0).getMBB() == &MBBN) {
    // I is the instruction of "jmp #offset=0", as follows,
    //     jmp        $BB0_3
    // $BB0_3:
    //     ld        $4, 28($sp)
    ++NumDelJmp;
    MBB.erase(I);        // delete the "JMP 0" instruction
    Changed = true;        // Notify LLVM kernel Changed
  }
  return Changed;

}

/// createCpu0DelJmpPass - Returns a pass that DelJmp in Cpu0 MachineFunctions
FunctionPass *llvm::createCpu0DelJmpPass(Cpu0TargetMachine &tm) {
  return new DelJmp(tm);
}

#endif
```

As above code, except Cpu0DelUselessJMP.cpp, other files are changed for registering class DelJmp as a functional pass. As the comment of above code, MBB is the current block and MBBN is the next block. For each last instruction of every MBB, we check whether or not it is the JMP instruction and its Operand is the next basic block. By getMBB() in MachineOperand, you can get the MBB address. For the member functions of MachineOperand, please check include/llvm/CodeGen/MachineOperand.h Now, let's run Chapter8_2/ with ch8_2_deluselessjmp.cpp for explanation.

lbdex/input/ch8_2_deluselessjmp.cpp

```
int test_DelUselessJMP()
{
  int a = 1; int b = -2; int c = 3;

  if (a == 0) {
    a++;
  }
  if (b == 0) {
    a = a + 3;
    b++;
  } else if (b < 0) {
    a = a + b;
    b--;
  }
  if (c > 0) {
    a = a + c;
    c++;
  }

  return a;
}
```

(continues on next page)

(continued from previous page)

```
118-165-78-10:input Jonathan$ clang -target mips-unknown-linux-gnu
-c ch8_2_deluselessjmp.cpp -emit-llvm -o ch8_2_deluselessjmp.bc
118-165-78-10:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -filetype=asm -stats
ch8_2_deluselessjmp.bc -o -
  ...
        cmp   $sw, $4, $3
        jne   $sw, $BB0_2
        nop
# BB#1:
  ...
        cmp   $sw, $3, $2
        jlt   $sw, $BB0_8
        nop
# BB#7:
...
===-------------------------------------------------------------------------===
                          ... Statistics Collected ...
===-------------------------------------------------------------------------===
 ...
 2 del-jmp        - Number of useless jmp deleted
 ...
```

The terminal displays "Number of useless jmp deleted" by `llc -stats` option because we set the "STATISTIC(NumDelJmp, "Number of useless jmp deleted")" in code. It deletes 2 jmp instructions from block "# BB#0" and "$BB0_6". You can check it by `llc -enable-cpu0-del-useless-jmp=false` option to see the difference to non-optimization version. If you run with ch8_1_1.cpp, you will find 10 jmp instructions are deleted from 120 lines of assembly code, which meaning 8% improvement in speed and code size[1].

8.5 Fill Branch Delay Slot

Cpu0 instruction set is designed to be a classical RISC pipeline machine. Classical RISC machine has many perfect features[3][4]. I change Cpu0 backend to a 5 stages of classical RISC pipeline machine with one delay slot like some of Mips model (The original Cpu0 from its author, is a 3 stages of RISC machine). With this change, the backend needs filling the NOP instruction in the branch delay slot. In order to make this tutorial simple for learning, Cpu0 backend code not fill the branch delay slot with any useful instruction for optimization. Readers can reference the MipsDelaySlotFiller.cpp to know how to insert useful instructions in backend optimization. Following code added in Chapter8_2 for NOP fill in Branch Delay Slot.

[1] On a platform with cache and DRAM, the cache miss costs serveral tens time of instruction cycle. Usually, the compiler engineers who work in the vendor of platform solution are spending much effort of trying to reduce the cache miss for speed. Reduce code size will decrease the cache miss frequency too.

[3] See book Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)

[4] http://en.wikipedia.org/wiki/Classic_RISC_pipeline

lbdex/chapters/Chapter8_2/CMakeLists.txt

```
  Cpu0DelaySlotFiller.cpp
```

lbdex/chapters/Chapter8_2/Cpu0.h

```
  FunctionPass *createCpu0DelaySlotFillerPass(Cpu0TargetMachine &TM);
```

lbdex/chapters/Chapter8_2/Cpu0TargetMachine.cpp

```
// Implemented by targets that want to run passes immediately before
// machine code is emitted. return true if -print-machineinstrs should
// print out the code after the passes.
void Cpu0PassConfig::addPreEmitPass() {
  Cpu0TargetMachine &TM = getCpu0TargetMachine();
```

```
  addPass(createCpu0DelaySlotFillerPass(TM));
```

```
}
```

lbdex/chapters/Chapter8_2/Cpu0DelaySlotFiller.cpp

```
//===-- Cpu0DelaySlotFiller.cpp - Cpu0 Delay Slot Filler -----------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// Simple pass to fill delay slots with useful instructions.
//
//===----------------------------------------------------------------------===//

#include "Cpu0.h"
#if CH >= CH8_2

#include "Cpu0InstrInfo.h"
#include "Cpu0TargetMachine.h"
#include "llvm/ADT/BitVector.h"
#include "llvm/ADT/SmallPtrSet.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/Analysis/AliasAnalysis.h"
#include "llvm/Analysis/ValueTracking.h"
#include "llvm/CodeGen/MachineBranchProbabilityInfo.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
```

(continues on next page)

(continued from previous page)

```
#include "llvm/CodeGen/PseudoSourceValue.h"
#include "llvm/CodeGen/TargetInstrInfo.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/CodeGen/TargetRegisterInfo.h"

using namespace llvm;

#define DEBUG_TYPE "delay-slot-filler"

STATISTIC(FilledSlots, "Number of delay slots filled");

namespace {
  typedef MachineBasicBlock::iterator Iter;
  typedef MachineBasicBlock::reverse_iterator ReverseIter;

  class Filler : public MachineFunctionPass {
  public:
    Filler(TargetMachine &tm)
      : MachineFunctionPass(ID) { }

    StringRef getPassName() const override {
      return "Cpu0 Delay Slot Filler";
    }

    bool runOnMachineFunction(MachineFunction &F) override {
      bool Changed = false;
      for (MachineFunction::iterator FI = F.begin(), FE = F.end();
           FI != FE; ++FI)
        Changed |= runOnMachineBasicBlock(*FI);
      return Changed;
    }
  private:
    bool runOnMachineBasicBlock(MachineBasicBlock &MBB);

    static char ID;
  };
  char Filler::ID = 0;
} // end of anonymous namespace

static bool hasUnoccupiedSlot(const MachineInstr *MI) {
  return MI->hasDelaySlot() && !MI->isBundledWithSucc();
}

/// runOnMachineBasicBlock - Fill in delay slots for the given basic block.
/// We assume there is only one delay slot per delayed instruction.
bool Filler::runOnMachineBasicBlock(MachineBasicBlock &MBB) {
  bool Changed = false;
  const Cpu0Subtarget &STI = MBB.getParent()->getSubtarget<Cpu0Subtarget>();
  const Cpu0InstrInfo *TII = STI.getInstrInfo();

  for (Iter I = MBB.begin(); I != MBB.end(); ++I) {
```

(continues on next page)

(continued from previous page)

```
    if (!hasUnoccupiedSlot(&*I))
      continue;

    ++FilledSlots;
    Changed = true;

    // Bundle the NOP to the instruction with the delay slot.
    BuildMI(MBB, std::next(I), I->getDebugLoc(), TII->get(Cpu0::NOP));
    MIBundleBuilder(MBB, I, std::next(I, 2));
  }

  return Changed;
}

/// createCpu0DelaySlotFillerPass - Returns a pass that fills in delay
/// slots in Cpu0 MachineFunctions
FunctionPass *llvm::createCpu0DelaySlotFillerPass(Cpu0TargetMachine &tm) {
  return new Filler(tm);
}

#endif
```

To make the basic block label remains same, statement MIBundleBuilder() needs to be inserted after the statement BuildMI(..., NOP) of Cpu0DelaySlotFiller.cpp. MIBundleBuilder() make both the branch instruction and NOP bundled into one instruction (first part is branch instruction and second part is NOP).

lbdex/chapters/Chapter3_2/Cpu0AsmPrinter.cpp

```
//- emitInstruction() must exists or will have run time error.
void Cpu0AsmPrinter::emitInstruction(const MachineInstr *MI) {
```

```
  //  Print out both ordinary instruction and boudle instruction
  MachineBasicBlock::const_instr_iterator I = MI->getIterator();
  MachineBasicBlock::const_instr_iterator E = MI->getParent()->instr_end();

  do {
```

```
    if (I->isPseudo() && !isLongBranchPseudo(I->getOpcode()))
```

```
      llvm_unreachable("Pseudo opcode found in emitInstruction()");

    MCInst TmpInst0;
    MCInstLowering.Lower(&*I, TmpInst0);
    OutStreamer->emitInstruction(TmpInst0, getSubtargetInfo());
  } while ((++I != E) && I->isInsideBundle()); // Delay slot check
}
```

In order to print the NOP, the Cpu0AsmPrinter.cpp of Chapter3_2 prints all bundle instructions in loop. Without the loop, only the first part of the bundle instruction (branch instruction only) is printed. In llvm 3.1 the basice block label remains same even if you didn't do the bundle after it. But for some reasons, it changed in llvm at some later version and you need doing "bundle" in order to keep block label unchanged at later llvm phase.

8.6 Conditional instruction

lbdex/input/ch8_2_select.cpp

```
// The following files will generate IR select even compile with clang -O0.
int test_movx_1()
{
  volatile int a = 1;
  int c = 0;

  c = !a ? 1:3;

  return c;
}

int test_movx_2()
{
  volatile int a = 1;
  int c = 0;

  c = a ? 1:3;

  return c;
}
```

Run Chapter8_1 with ch8_2_select.cpp will get the following result.

```
114-37-150-209:input Jonathan$ clang -O1 -target mips-unknown-linux-gnu
-c ch8_2_select.cpp -emit-llvm -o ch8_2_select.bc
114-37-150-209:input Jonathan$ ~/llvm/test/build/bin/
llvm-dis ch8_2_select.bc -o -
...
; Function Attrs: nounwind uwtable
define i32 @_Z11test_movx_1v() #0 {
  %a = alloca i32, align 4
  %c = alloca i32, align 4
  store volatile i32 1, i32* %a, align 4
  store i32 0, i32* %c, align 4
  %1 = load volatile i32* %a, align 4
  %2 = icmp ne i32 %1, 0
  %3 = xor i1 %2, true
  %4 = select i1 %3, i32 1, i32 3
  store i32 %4, i32* %c, align 4
  %5 = load i32* %c, align 4
  ret i32 %5
}

; Function Attrs: nounwind uwtable
define i32 @_Z11test_movx_2v() #0 {
  %a = alloca i32, align 4
```

(continues on next page)

(continued from previous page)

```
  %c = alloca i32, align 4
  store volatile i32 1, i32* %a, align 4
  store i32 0, i32* %c, align 4
  %1 = load volatile i32* %a, align 4
  %2 = icmp ne i32 %1, 0
  %3 = select i1 %2, i32 1, i32 3
  store i32 %3, i32* %c, align 4
  %4 = load i32* %c, align 4
  ret i32 %4
}
...

114-37-150-209:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch8_2_select.bc -o -
...
LLVM ERROR: Cannot select: 0x39f47c0: i32 = select_cc ...
```

As above llvm IR, ch8_2_select.bc, clang generates **select** IR for small basic control block (if statement only include one assign statement). This **select** IR is the result of optimization for CPUs with conditional instructions support. And from above error message, obviously IR **select** is changed to **select_cc** during DAG optimization stages.

Chapter8_2 supports **select** with the following code added and changed.

lbdex/chapters/Chapter8_2/Cpu0InstrInfo.td

```
let Predicates = [Ch8_2] in {
include "Cpu0CondMov.td"
} // let Predicates = [Ch8_2]
```

lbdex/chapters/Chapter8_2/Cpu0CondMov.td

```
//===-- Cpu0CondMov.td - Describe Cpu0 Conditional Moves --*- tablegen -*--===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This is the Conditional Moves implementation.
//
//===----------------------------------------------------------------------===//

// Conditional moves:
// These instructions are expanded in
// Cpu0ISelLowering::EmitInstrWithCustomInserter if target does not have
// conditional move instructions.
// cond:int, data:int
```

(continues on next page)

(continued from previous page)

```
class CondMovIntInt<RegisterClass CRC, RegisterClass DRC, bits<8> op,
                    string instr_asm> :
  FA<op, (outs DRC:$ra), (ins DRC:$rb, CRC:$rc, DRC:$F),
     !strconcat(instr_asm, "\t$ra, $rb, $rc"), [], IIAlu> {
  let shamt = 0;
  let Constraints = "$F = $ra";
}

// select patterns
multiclass MovzPats0Slt<RegisterClass CRC, RegisterClass DRC,
                     Instruction MOVZInst, Instruction SLTOp,
                     Instruction SLTuOp, Instruction SLTiOp,
                     Instruction SLTiuOp> {
  def : Pat<(select (i32 (setge CRC:$lhs, CRC:$rhs)), DRC:$T, DRC:$F),
            (MOVZInst DRC:$T, (SLTOp CRC:$lhs, CRC:$rhs), DRC:$F)>;
  def : Pat<(select (i32 (setuge CRC:$lhs, CRC:$rhs)), DRC:$T, DRC:$F),
            (MOVZInst DRC:$T, (SLTuOp CRC:$lhs, CRC:$rhs), DRC:$F)>;
  def : Pat<(select (i32 (setge CRC:$lhs, immSExt16:$rhs)), DRC:$T, DRC:$F),
            (MOVZInst DRC:$T, (SLTiOp CRC:$lhs, immSExt16:$rhs), DRC:$F)>;
  def : Pat<(select (i32 (setuge CRC:$lh, immSExt16:$rh)), DRC:$T, DRC:$F),
            (MOVZInst DRC:$T, (SLTiuOp CRC:$lh, immSExt16:$rh), DRC:$F)>;
  def : Pat<(select (i32 (setle CRC:$lhs, CRC:$rhs)), DRC:$T, DRC:$F),
            (MOVZInst DRC:$T, (SLTOp CRC:$rhs, CRC:$lhs), DRC:$F)>;
  def : Pat<(select (i32 (setule CRC:$lhs, CRC:$rhs)), DRC:$T, DRC:$F),
            (MOVZInst DRC:$T, (SLTuOp CRC:$rhs, CRC:$lhs), DRC:$F)>;
}

multiclass MovzPats1<RegisterClass CRC, RegisterClass DRC,
                     Instruction MOVZInst, Instruction XOROp> {
  def : Pat<(select (i32 (seteq CRC:$lhs, CRC:$rhs)), DRC:$T, DRC:$F),
            (MOVZInst DRC:$T, (XOROp CRC:$lhs, CRC:$rhs), DRC:$F)>;
  def : Pat<(select (i32 (seteq CRC:$lhs, 0)), DRC:$T, DRC:$F),
            (MOVZInst DRC:$T, CRC:$lhs, DRC:$F)>;
}

multiclass MovnPats<RegisterClass CRC, RegisterClass DRC, Instruction MOVNInst,
                    Instruction XOROp> {
  def : Pat<(select (i32 (setne CRC:$lhs, CRC:$rhs)), DRC:$T, DRC:$F),
            (MOVNInst DRC:$T, (XOROp CRC:$lhs, CRC:$rhs), DRC:$F)>;
  def : Pat<(select CRC:$cond, DRC:$T, DRC:$F),
            (MOVNInst DRC:$T, CRC:$cond, DRC:$F)>;
  def : Pat<(select (i32 (setne CRC:$lhs, 0)),DRC:$T, DRC:$F),
            (MOVNInst DRC:$T, CRC:$lhs, DRC:$F)>;
}

// Instantiation of instructions.
def MOVZ_I_I     : CondMovIntInt<CPURegs, CPURegs, 0x0a, "movz">;

def MOVN_I_I     : CondMovIntInt<CPURegs, CPURegs, 0x0b, "movn">;

// Instantiation of conditional move patterns.
let Predicates = [HasSlt] in {
```

(continues on next page)

(continued from previous page)

```
defm : MovzPats0Slt<CPURegs, CPURegs, MOVZ_I_I, SLT, SLTu, SLTi, SLTiu>;
}

defm : MovzPats1<CPURegs, CPURegs, MOVZ_I_I, XOR>;

defm : MovnPats<CPURegs, CPURegs, MOVN_I_I, XOR>;
```

lbdex/chapters/Chapter8_2/Cpu0ISelLowering.h

```
    SDValue lowerSELECT(SDValue Op, SelectionDAG &DAG) const;
```

lbdex/chapters/Chapter8_2/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
  setOperationAction(ISD::SELECT,             MVT::i32,   Custom);
```

```
  setOperationAction(ISD::SELECT_CC,         MVT::i32, Expand);
  setOperationAction(ISD::SELECT_CC,         MVT::Other, Expand);
```

```
}
```

```
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
  switch (Op.getOpcode())
  {
```

```
  case ISD::SELECT:             return lowerSELECT(Op, DAG);
```

```
  }
  ...
}
```

```
SDValue Cpu0TargetLowering::
lowerSELECT(SDValue Op, SelectionDAG &DAG) const
{
  return Op;
}
```

Set ISD::SELECT_CC to "Expand" will stop llvm optimization from merging "setcc" and "select" into one IR "select_cc"[2]. Next the LowerOperation() return Op code directly for ISD::SELECT. Finally the pattern defined in Cpu0CondMov.td will translate the **select** IR into conditional instruction, **movz** or **movn**. Let's run Chapter8_2 with

[2] http://llvm.org/docs/WritingAnLLVMBackend.html#expand

ch8_2_select.cpp to get the following result. Again, the cpu032II uses **slt** instead of **cmp** has a little improved in instructions number.

```
114-37-150-209:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm ch8_2_select.bc -o -
...
      .type   _Z11test_movx_1v,@function
      ...
      addiu   $2, $zero, 3
      movz    $2, $3, $4
      ...
      .type   _Z11test_movx_2v,@function
      ...
      addiu   $2, $zero, 3
      movn    $2, $3, $4
      ...
```

The clang uses **select** IR in small basic block to reduce the branch cost in pipeline machine since the branch will make the pipeline "stall". But it needs the conditional instruction support[Page 364, 3]. If your backend has no conditional instruction and needs clang compiler with optimization option **O1** above level, you can change clang to force it generating traditional branch basic block instead of generating IR **select**. RISC CPU came from the advantage of pipeline and add more and more instruction when time passed. Compare Mips and ARM, the Mips has only **movz** and **movn** two instructions while ARM has many. We create Cpu0 instructions as a simple instructions RISC pipeline machine for compiler toolchain tutorial. However the **cmp** instruction is hired because many programmer is used to it in past and now (ARM use it). This instruction matches the thinking in assembly programming, but the **slt** instruction is more efficient in RISC pipleline. If you designed a backend aimed for C/C++ highlevel language, you may consider **slt** instead of **cmp** since assembly code are rare used in programming and beside, the assembly programmer can accept **slt** not difficultly since usually they are professional.

File ch8_2_select2.cpp will generate IR **select** if compile with `clang -O1`.

lbdex/input/ch8_2_select2.cpp

```
// The following files will generate IR select when compile with clang -O1 but
// clang -O0 won't generate IR select.
volatile int a = 1;
volatile int b = 2;

int test_movx_3()
{
  int c = 0;

  if (a < b)
    return 1;
  else
    return 2;
}

int test_movx_4()
{
  int c = 0;
```

(continues on next page)

(continued from previous page)

```
  if (a)
    c = 1;
  else
    c = 3;

  return c;
}
```

List the conditional statements of C, IR, DAG and Cpu0 instructions as the following table.

Table 8.2: Conditional statements of C, IR, DAG and Cpu0 instructions

C	if (a < b) c = 1; else c = 3;
•	c = a ? 1:3;
IR	icmp + (eq, ne, sgt, sge, slt, sle) + br
DAG	((seteq, setne, setgt, setge, setlt, setle) + setcc) + select
Cpu0	movz, movn

File ch8_2_select_global_pic.cpp mentioned in Chapter Global variables can be tested now as follows,

lbdex/input/ch8_2_select_global_pic.cpp

```
volatile int a1 = 1;
volatile int b1 = 2;

int gI1 = 100;
int gJ1 = 50;

int test_select_global_pic()
{
  if (a1 < b1)
    return gI1;
  else
    return gJ1;
}
```

```
JonathantekiiMac:input Jonathan$ clang -O1 -target mips-unknown-linux-gnu
-c ch8_2_select_global_pic.cpp -emit-llvm -o ch8_2_select_global_pic.bc
JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/
llvm-dis ch8_2_select_global_pic.bc -o -
...
@a1 = global i32 1, align 4
@b1 = global i32 2, align 4
@gI1 = global i32 100, align 4
@gJ1 = global i32 50, align 4

; Function Attrs: nounwind
define i32 @_Z18test_select_globalv() #0 {
```

(continues on next page)

(continued from previous page)

```
  %1 = load volatile i32* @a1, align 4, !tbaa !1
  %2 = load volatile i32* @b1, align 4, !tbaa !1
  %3 = icmp slt i32 %1, %2
  %gI1.val = load i32* @gI1, align 4
  %gJ1.val = load i32* @gJ1, align 4
  %.0 = select i1 %3, i32 %gI1.val, i32 %gJ1.val
  ret i32 %.0
}
...
JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/
llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -filetype=asm ch8_2_select_global_
↪pic.bc -o -
  .section .mdebug.abi32
  .previous
  .file "ch8_2_select_global_pic.bc"
  .text
  .globl  _Z18test_select_globalv
  .align  2
  .type _Z18test_select_globalv,@function
  .ent  _Z18test_select_globalv # @_Z18test_select_globalv
_Z18test_select_globalv:
  .frame  $sp,0,$lr
  .mask   0x00000000,0
  .set  noreorder
  .cpload $t9
  .set  nomacro
# BB#0:
  lui $2, %got_hi(a1)
  addu  $2, $2, $gp
  ld  $2, %got_lo(a1)($2)
  ld  $2, 0($2)
  lui $3, %got_hi(b1)
  addu  $3, $3, $gp
  ld  $3, %got_lo(b1)($3)
  ld  $3, 0($3)
  cmp $sw, $2, $3
  andi  $2, $sw, 1
  lui $3, %got_hi(gJ1)
  addu  $3, $3, $gp
  ori $3, $3, %got_lo(gJ1)
  lui $4, %got_hi(gI1)
  addu  $4, $4, $gp
  ori $4, $4, %got_lo(gI1)
  movn  $3, $4, $2
  ld  $2, 0($3)
  ld  $2, 0($2)
  ret $lr
  .set  macro
  .set  reorder
  .end  _Z18test_select_globalv
$tmp0:
  .size _Z18test_select_globalv, ($tmp0)-_Z18test_select_globalv
```

(continues on next page)

(continued from previous page)

```
  .type a1,@object              # @a1
  .data
  .globl  a1
  .align  2
a1:
  .4byte  1                       # 0x1
  .size a1, 4

  .type b1,@object              # @b1
  .globl  b1
  .align  2
b1:
  .4byte  2                       # 0x2
  .size b1, 4

  .type gI1,@object             # @gI1
  .globl  gI1
  .align  2
gI1:
  .4byte  100                     # 0x64
  .size gI1, 4

  .type gJ1,@object             # @gJ1
  .globl  gJ1
  .align  2
gJ1:
  .4byte  50                      # 0x32
  .size gJ1, 4
```

8.7 Phi node

Since phi node is popular used in SSA form[5], llvm applies phi node in IR for optimization work either. Phi node exists for "live variable analysis". An example for C is here[6]. As mentioned in wiki web site of reference above, through finding dominance frontiers, compiler knows where to insert phi functions. The following input let you know the benefits of phi node as follows,

lbdex/input/ch8_2_phinode.cpp

```
int test_phinode(int a , int b, int c)
{
  int d = 2;

  if (a == 0) {
    a = a+1; // a = 1
  }
```

(continues on next page)

[5] https://en.wikipedia.org/wiki/Static_single_assignment_form
[6] http://stackoverflow.com/questions/11485531/what-exactly-phi-instruction-does-and-how-to-use-it-in-llvm

(continued from previous page)

```
  else if (b != 0) {
    a = a-1;
  }
  else if (c == 0) {
    a = a+2;
  }
  d = a + b;

  return d;
}
```

Compile it with debug build clang for O3 as follows,

```
114-43-212-251:input Jonathan$ ~/llvm/debug/build/bin/clang
-O3 -target mips-unknown-linux-gnu -c ch8_2_phinode.cpp -emit-llvm -o
ch8_2_phinode.bc
114-43-212-251:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch8_2_phinode.bc -o -
...
define i32 @_Z12test_phinodeiii(i32 signext %a, i32 signext %b, i32 signext %c) local_
↪unnamed_addr #0 {
entry:
  %cmp = icmp eq i32 %a, 0
  br i1 %cmp, label %if.end7, label %if.else

if.else:                                          ; preds = %entry
  %cmp1 = icmp eq i32 %b, 0
  br i1 %cmp1, label %if.else3, label %if.then2

if.then2:                                         ; preds = %if.else
  %dec = add nsw i32 %a, -1
  br label %if.end7

if.else3:                                         ; preds = %if.else
  %cmp4 = icmp eq i32 %c, 0
  %add = add nsw i32 %a, 2
  %add.a = select i1 %cmp4, i32 %add, i32 %a
  br label %if.end7

if.end7:                                          ; preds = %entry, %if.else3, %if.then2
  %a.addr.0 = phi i32 [ %dec, %if.then2 ], [ %add.a, %if.else3 ], [ 1, %entry ]
  %add8 = add nsw i32 %a.addr.0, %b
  ret i32 %add8
}
```

Because SSA form, the llvm ir for destination variable *a* in different basic block (if then, else) must use different name. But how does the source variable *a* in "d = a + b;" be named? The basic block "a = a-1;" and "a = a+2;" have different names. The basic block "a = a-1;" uses %dec and the basic block "a = a+2;" uses "%add" as destination variable name in SSA llvm ir. In order to solve the source variable name from different basic blocks in SSA form, the phi structure is created as above. The compiler option O0 as the following doesn't apply phi node. Instead, it uses store to solve the source variable name from different basic block.

```
114-43-212-251:input Jonathan$ ~/llvm/debug/build/bin/clang
-O0 -target mips-unknown-linux-gnu -c ch8_2_phinode.cpp -emit-llvm -o
ch8_2_phinode.bc
114-43-212-251:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch8_2_phinode.bc -o -
...
define i32 @_Z12test_phinodeiii(i32 signext %a, i32 signext %b, i32 signext %c) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  %c.addr = alloca i32, align 4
  %d = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  store i32 %c, i32* %c.addr, align 4
  store i32 2, i32* %d, align 4
  %0 = load i32, i32* %a.addr, align 4
  %cmp = icmp eq i32 %0, 0
  br i1 %cmp, label %if.then, label %if.else

if.then:                                          ; preds = %entry
  %1 = load i32, i32* %a.addr, align 4
  %inc = add nsw i32 %1, 1
  store i32 %inc, i32* %a.addr, align 4
  br label %if.end7

if.else:                                          ; preds = %entry
  %2 = load i32, i32* %b.addr, align 4
  %cmp1 = icmp ne i32 %2, 0
  br i1 %cmp1, label %if.then2, label %if.else3

if.then2:                                         ; preds = %if.else
  %3 = load i32, i32* %a.addr, align 4
  %dec = add nsw i32 %3, -1
  store i32 %dec, i32* %a.addr, align 4
  br label %if.end6

if.else3:                                         ; preds = %if.else
  %4 = load i32, i32* %c.addr, align 4
  %cmp4 = icmp eq i32 %4, 0
  br i1 %cmp4, label %if.then5, label %if.end

if.then5:                                         ; preds = %if.else3
  %5 = load i32, i32* %a.addr, align 4
  %add = add nsw i32 %5, 2
  store i32 %add, i32* %a.addr, align 4
  br label %if.end

if.end:                                           ; preds = %if.then5, %if.else3
  br label %if.end6

if.end6:                                          ; preds = %if.end, %if.then2
  br label %if.end7
```

(continues on next page)

(continued from previous page)

```
if.end7:                                          ; preds = %if.end6, %if.then
  %6 = load i32, i32* %a.addr, align 4
  %7 = load i32, i32* %b.addr, align 4
  %add8 = add nsw i32 %6, %7
  store i32 %add8, i32* %d, align 4
  %8 = load i32, i32* %d, align 4
  ret i32 %8
}
```

Compile with `clang -O3` generate phi function. The phi function can assign virtual register value directly from multi basic blocks. Compile with `clang -O0` doesn't generate phi, it assigns virtual register value by loading stack slot where the stack slot is saved in each of multi basic blocks before. In this example the pointer of %a.addr point to the stack slot, and "store i32 %inc, i32* %a.addr, align 4", "store i32 %dec, i32* %a.addr, align 4", "store i32 %add, i32* %a.addr, align 4" in label if.then:, if.then2: and if.then5:, respectively. In other words, it needs 3 store instructions. It's possible that compiler finds that the a == 0 is always true after optimization analysis through phi node. If so, the phi node version will bring better result because `clang -O0` version uses load and store with pointer %a.addr which may cut the optimization opportunity. Compiler books discuss the Control Flow Graph (CFG) analysis through dominance frontiers calculation for setting phi node. Then compiler apply the global optimization on CFG with phi node, and remove phi node by replacing with "load store" at the end.

If you are interested in more details than the wiki web site, please refer book here[7] for phi node, or book here[8] for the dominator tree analysis if you have this book.

8.8 RISC CPU knowledge

As mentioned in the previous section, Cpu0 is a RISC (Reduced Instruction Set Computer) CPU with 5 stages of pipeline. RISC CPU is full in the world, even the X86 of CISC (Complex Instruction Set Computer) is RISC inside (It translates CISC instruction into micro-instructions which do pipeline as RISC). Knowledge with RISC concept may make you satisfied in compiler design. List these two excellent books we have read for reference. Sure, there are many books in Computer Architecture and some of them contain real RISC CPU knowledge needed, but these two are excellent and popular.

Computer Organization and Design: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)

Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)

The book of "Computer Organization and Design: The Hardware/Software Interface" (there are 4 editions at the book is written) is for the introduction, while "Computer Architecture: A Quantitative Approach" is more complicate and deep in CPU architecture (there are 5 editions at the book is written).

Above two books use Mips CPU as an example since Mips is more RISC-like than other market CPUs. ARM serials of CPU dominate the embedded market especially in mobile phone and other portable devices. The following book is good which I am reading now.

ARM System Developer's Guide: Designing and Optimizing System Software (The Morgan Kaufmann Series in Computer Architecture and Design).

[7] Section 8.11 of Muchnick, Steven S. (1997). Advanced Compiler Design and Implementation. Morgan Kaufmann. ISBN 1-55860-320-4.
[8] Refer chapter 9 of book Compilers: Principles, Techniques, and Tools (2nd Edition)

CHAPTER

NINE

FUNCTION CALL

- *Mips stack frame*
- *Load incoming arguments from stack frame*
- *Store outgoing arguments to stack frame*
  - *Pseudo hook instruction ADJCALLSTACKDOWN and ADJCALLSTACKUP*
  - *Read Lowercall() with Graphivz's help*
  - *Long and short string initialization*
- *Structure type support*
  - *Ordinary struct type*
  - *byval struct type*
- *Function call optiomization*
  - *Tail call optimization*
  - *Recursion optimization*
- *Other features supporting*
  - *The $gp register caller saved register in PIC addressing mode*
  - *Variable number of arguments*
  - *Dynamic stack allocation support*
  - *Variable sized array support*
  - *Function related Intrinsics support*
    - *frameaddress and returnaddress intrinsics*
    - *eh.return intrinsic*
    - *eh.dwarf intrinsic*
    - *bswap intrinsic*
  - *Add specific backend intrinsic function*
- *Summary*

The subroutine/function call of backend translation is supported in this chapter. A lot of code are needed to support function call in this chapter. They are added according llvm supplied interface to explain easily. This chapter starts from introducing the Mips stack frame structure since we borrow many parts of ABI from it. Although each CPU has

it's own ABI, most of ABI for RISC CPUs are similar. The section "4.5 DAG Lowering" of tricore_llvm.pdf contains knowledge about Lowering process. Section "4.5.1 Calling Conventions" of tricore_llvm.pdf is the related material you can reference further.

If you have problem in reading the stack frame illustrated in the first three sections of this chapter, you can read the appendix B of "Procedure Call Convention" of book "Computer Organization and Design, 1st Edition"[1], "Run Time Memory" of compiler book, or "Function Call Sequence" and "Stack Frame" of Mips ABI[3].

9.1 Mips stack frame

The first thing for designing the Cpu0 function call is deciding how to pass arguments in function call. There are two options. One is passing arguments all in stack. The other is passing arguments in the registers which are reserved for function arguments, and put the other arguments in stack if it over the number of registers reserved for function call. For example, Mips pass the first 4 arguments in register $a0, $a1, $a2, $a3, and the other arguments in stack if it over 4 arguments. Fig. 9.1 is the Mips stack frame.

Base	Offset	Contents	Frame
		unspecified	*High addresses*
		. . .	
		variable size	
		(if present)	
		incoming arguments	Previous
	+16	passed in stack frame	
		space for incoming	
old *$sp*	+0	arguments 1-4	
		locals and	
		temporaries	
		general register	
		save area	Current
		floating-point	
		register save area	
		argument	
$sp	+0	build area	*Low addresses*

Fig. 9.1: Mips stack frame

Run `llc -march=mips` for ch9_1.bc, you will get the following result. See comments **"//"**.

lbdex/input/ch9_1.cpp

```
int gI = 100;

int sum_i(int x1, int x2, int x3, int x4, int x5, int x6)
{
  int sum = gI + x1 + x2 + x3 + x4 + x5 + x6;

  return sum;
}
```

(continues on next page)

[1] Computer Organization and Design: The Hardware/Software Interface 1st edition (The Morgan Kaufmann Series in Computer Architecture and Design)

[3] http://www.linux-mips.org/pub/linux/mips/doc/ABI/mipsabi.pdf

(continued from previous page)

```
int main()
{
  int a = sum_i(1, 2, 3, 4, 5, 6);

  return a;
}
```

```
118-165-78-230:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_1.cpp -emit-llvm -o ch9_1.bc
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=mips -relocation-model=pic -filetype=asm ch9_1.bc -o
ch9_1.mips.s
118-165-78-230:input Jonathan$ cat ch9_1.mips.s
  .section .mdebug.abi32
  .previous
  .file "ch9_1.bc"
  .text
  .globl  _Z5sum_iiiiiii
  .align  2
  .type _Z5sum_iiiiiii,@function
  .set  nomips16                # @_Z5sum_iiiiiii
  .ent  _Z5sum_iiiiiii
_Z5sum_iiiiiii:
  .cfi_startproc
  .frame  $sp,32,$ra
  .mask   0x00000000,0
  .fmask  0x00000000,0
  .set  noreorder
  .set  nomacro
  .set  noat
# BB#0:
  addiu $sp, $sp, -32
$tmp1:
  .cfi_def_cfa_offset 32
  sw  $4, 28($sp)
  sw  $5, 24($sp)
  sw  $t9, 20($sp)
  sw  $7, 16($sp)
  lw  $1, 48($sp) // load argument 5
  sw  $1, 12($sp)
  lw  $1, 52($sp) // load argument 6
  sw  $1, 8($sp)
  lw  $2, 24($sp)
  lw  $3, 28($sp)
  addu  $2, $3, $2
  lw  $3, 20($sp)
  addu  $2, $2, $3
  lw  $3, 16($sp)
  addu  $2, $2, $3
  lw  $3, 12($sp)
  addu  $2, $2, $3
```

(continues on next page)

(continued from previous page)

```
  addu  $2, $2, $1
  sw  $2, 4($sp)
  jr  $ra
  addiu $sp, $sp, 32
  .set  at
  .set  macro
  .set  reorder
  .end  _Z5sum_iiiiiii
$tmp2:
  .size _Z5sum_iiiiiii, ($tmp2)-_Z5sum_iiiiiii
  .cfi_endproc

  .globl  main
  .align  2
  .type main,@function
  .set  nomips16                # @main
  .ent  main
main:
  .cfi_startproc
  .frame  $sp,40,$ra
  .mask   0x80000000,-4
  .fmask  0x00000000,0
  .set  noreorder
  .set  nomacro
  .set  noat
# BB#0:
  lui $2, %hi(_gp_disp)
  ori $2, $2, %lo(_gp_disp)
  addiu $sp, $sp, -40
$tmp5:
  .cfi_def_cfa_offset 40
  sw  $ra, 36($sp)            # 4-byte Folded Spill
$tmp6:
  .cfi_offset 31, -4
  addu  $gp, $2, $25
  sw  $zero, 32($sp)
  addiu $1, $zero, 6
  sw  $1, 20($sp) // Save argument 6 to 20($sp)
  addiu $1, $zero, 5
  sw  $1, 16($sp) // Save argument 5 to 16($sp)
  lw  $25, %call16(_Z5sum_iiiiiii)($gp)
  addiu $4, $zero, 1    // Pass argument 1 to $4 (=$a0)
  addiu $5, $zero, 2    // Pass argument 2 to $5 (=$a1)
  addiu $t9, $zero, 3
  jalr  $25
  addiu $7, $zero, 4
  sw  $2, 28($sp)
  lw  $ra, 36($sp)            # 4-byte Folded Reload
  jr  $ra
  addiu $sp, $sp, 40
  .set  at
  .set  macro
```

(continues on next page)

(continued from previous page)

```
  .set  reorder
  .end  main
$tmp7:
  .size main, ($tmp7)-main
  .cfi_endproc
```

From the mips assembly code generated as above, we see that it saves the first 4 arguments to $a0..$a3 and last 2 arguments to 16($sp) and 20($sp). Fig. 9.2 is the location of arguments for example code ch9_1.cpp. It loads argument 5 from 48($sp) in sum_i() since the argument 5 is saved to 16($sp) in main(). The stack size of sum_i() is 32, so 16+32($sp) is the location of incoming argument 5.

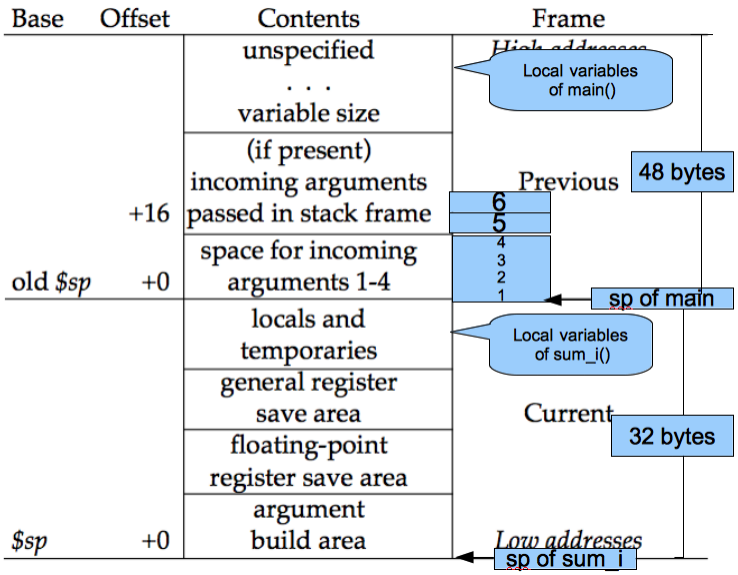


Fig. 9.2: Mips arguments location in stack frame

The 007-2418-003.pdf in here[2] is the Mips assembly language manual. Here[Page 380, 3] is Mips Application Binary Interface which include the Fig. 9.1.

9.2 Load incoming arguments from stack frame

From last section, in order to support function call, we need implementing the arguments passing mechanism with stack frame. Before doing it, let's run the old version of code Chapter8_2/ with ch9_1.cpp and see what happens.

```
118-165-79-31:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch9_1.bc -o ch9_1.cpu0.s
Assertion failed: (InVals.size() == Ins.size() && "LowerFormalArguments didn't
emit the correct number of values!"), function LowerArguments, file /Users/
Jonathan/llvm/test/llvm/lib/CodeGen/SelectionDAG/
SelectionDAGBuilder.cpp, ...
...
0.  Program arguments: /Users/Jonathan/llvm/test/build/
```

(continues on next page)

[2] http://math-atlas.sourceforge.net/devel/assembly/007-2418-003.pdf

(continued from previous page)

```
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_1.bc -o
ch9_1.cpu0.s
1.  Running pass 'Function Pass Manager' on module 'ch9_1.bc'.
2.  Running pass 'CPU0 DAG->DAG Pattern Instruction Selection' on function
'@_Z5sum_iiiiiii'
Illegal instruction: 4
```

Since Chapter8_2/ define the LowerFormalArguments() with empty body, we get the error messages as above. Before defining LowerFormalArguments(), we have to choose how to pass arguments in function call. For demonstration, Cpu0 passes first two arguments in registers as default setting of `llc -cpu0-s32-calls=false`. When `llc -cpu0-s32-calls=true`, Cpu0 passes all it's arguments in stack.

Function LowerFormalArguments() is in charge of incoming arguments creation. We define it as follows,

lbdex/chapters/Chapter9_1/Cpu0ISelLowering.h

```
  class Cpu0TargetLowering : public TargetLowering  {
```

```
    /// Cpu0CC - This class provides methods used to analyze formal and call
    /// arguments and inquire about calling convention information.
    class Cpu0CC {
```

```
      void analyzeFormalArguments(const SmallVectorImpl<ISD::InputArg> &Ins,
                                  bool IsSoftFloat,
                                  Function::const_arg_iterator FuncArg);
```

```
      /// regSize - Size (in number of bits) of integer registers.
      unsigned regSize() const { return IsO32 ? 4 : 4; }
      /// numIntArgRegs - Number of integer registers available for calls.
      unsigned numIntArgRegs() const;
```

```
      /// Return pointer to array of integer argument registers.
      const ArrayRef<MCPhysReg> intArgRegs() const;
```

```
      void handleByValArg(unsigned ValNo, MVT ValVT, MVT LocVT,
                          CCValAssign::LocInfo LocInfo,
                          ISD::ArgFlagsTy ArgFlags);

      /// useRegsForByval - Returns true if the calling convention allows the
      /// use of registers to pass byval arguments.
      bool useRegsForByval() const { return CallConv != CallingConv::Fast; }

      /// Return the function that analyzes fixed argument list functions.
      llvm::CCAssignFn *fixedArgFn() const;
```

```
      void allocateRegs(ByValArgInfo &ByVal, unsigned ByValSize,
                        unsigned Align);
```

```
};
...
```

```
    /// isEligibleForTailCallOptimization - Check whether the call is eligible
    /// for tail call optimization.
    virtual bool
    isEligibleForTailCallOptimization(const Cpu0CC &Cpu0CCInfo,
                                      unsigned NextStackOffset,
                                      const Cpu0FunctionInfo& FI) const = 0;
```

```
    /// copyByValArg - Copy argument registers which were used to pass a byval
    /// argument to the stack. Create a stack frame object for the byval
    /// argument.
    void copyByValRegs(SDValue Chain, const SDLoc &DL,
                       std::vector<SDValue> &OutChains, SelectionDAG &DAG,
                       const ISD::ArgFlagsTy &Flags,
                       SmallVectorImpl<SDValue> &InVals,
                       const Argument *FuncArg,
                       const Cpu0CC &CC, const ByValArgInfo &ByVal) const;
```

```
    SDValue LowerCall(TargetLowering::CallLoweringInfo &CLI,
                      SmallVectorImpl<SDValue> &InVals) const override;
```

```
  ...
}
```

lbdex/chapters/Chapter9_1/Cpu0ISelLowering.cpp

```
// addLiveIn - This helper function adds the specified physical register to the
// MachineFunction as a live in value.  It also creates a corresponding
// virtual register for it.
static unsigned
addLiveIn(MachineFunction &MF, unsigned PReg, const TargetRegisterClass *RC)
{
  unsigned VReg = MF.getRegInfo().createVirtualRegister(RC);
  MF.getRegInfo().addLiveIn(PReg, VReg);
  return VReg;
}
```

```
//===----------------------------------------------------------------------===//
// TODO: Implement a generic logic using tblgen that can support this.
// Cpu0 32 ABI rules:
// ---
//===----------------------------------------------------------------------===//

// Passed in stack only.
static bool CC_Cpu0S32(unsigned ValNo, MVT ValVT, MVT LocVT,
                       CCValAssign::LocInfo LocInfo, ISD::ArgFlagsTy ArgFlags,
                       CCState &State) {
  // Do not process byval args here.
  if (ArgFlags.isByVal())
    return true;
```

(continues on next page)

(continued from previous page)

```
  // Promote i8 and i16
  if (LocVT == MVT::i8 || LocVT == MVT::i16) {
    LocVT = MVT::i32;
    if (ArgFlags.isSExt())
      LocInfo = CCValAssign::SExt;
    else if (ArgFlags.isZExt())
      LocInfo = CCValAssign::ZExt;
    else
      LocInfo = CCValAssign::AExt;
  }

  Align OrigAlign = ArgFlags.getNonZeroOrigAlign();
  unsigned Offset = State.AllocateStack(ValVT.getSizeInBits() >> 3,
                                        OrigAlign);
  State.addLoc(CCValAssign::getMem(ValNo, ValVT, Offset, LocVT, LocInfo));
  return false;
}

// Passed first two i32 arguments in registers and others in stack.
static bool CC_Cpu0O32(unsigned ValNo, MVT ValVT, MVT LocVT,
                       CCValAssign::LocInfo LocInfo, ISD::ArgFlagsTy ArgFlags,
                       CCState &State) {
  static const MCPhysReg IntRegs[] = { Cpu0::A0, Cpu0::A1 };

  // Do not process byval args here.
  if (ArgFlags.isByVal())
    return true;

  // Promote i8 and i16
  if (LocVT == MVT::i8 || LocVT == MVT::i16) {
    LocVT = MVT::i32;
    if (ArgFlags.isSExt())
      LocInfo = CCValAssign::SExt;
    else if (ArgFlags.isZExt())
      LocInfo = CCValAssign::ZExt;
    else
      LocInfo = CCValAssign::AExt;
  }

  unsigned Reg;

  // f32 and f64 are allocated in A0, A1 when either of the following
  // is true: function is vararg, argument is 3rd or higher, there is previous
  // argument which is not f32 or f64.
  bool AllocateFloatsInIntReg = true;
  Align OrigAlign = ArgFlags.getNonZeroOrigAlign();
  bool isI64 = (ValVT == MVT::i32 && OrigAlign == 8);

  if (ValVT == MVT::i32 || (ValVT == MVT::f32 && AllocateFloatsInIntReg)) {
    Reg = State.AllocateReg(IntRegs);
    // If this is the first part of an i64 arg,
    // the allocated register must be A0.
```

(continues on next page)

(continued from previous page)

```
    if (isI64 && (Reg == Cpu0::A1))
      Reg = State.AllocateReg(IntRegs);
    LocVT = MVT::i32;
  } else if (ValVT == MVT::f64 && AllocateFloatsInIntReg) {
    // Allocate int register. If first
    // available register is Cpu0::A1, shadow it too.
    Reg = State.AllocateReg(IntRegs);
    if (Reg == Cpu0::A1)
      Reg = State.AllocateReg(IntRegs);
    State.AllocateReg(IntRegs);
    LocVT = MVT::i32;
  } else
    llvm_unreachable("Cannot handle this ValVT.");

  if (!Reg) {
    unsigned Offset = State.AllocateStack(ValVT.getSizeInBits() >> 3,
                                          Align(OrigAlign));
    State.addLoc(CCValAssign::getMem(ValNo, ValVT, Offset, LocVT, LocInfo));
  } else
    State.addLoc(CCValAssign::getReg(ValNo, ValVT, Reg, LocVT, LocInfo));

  return false;
}
```

```
//===----------------------------------------------------------------------===//
//                  Call Calling Convention Implementation
//===----------------------------------------------------------------------===//

static const MCPhysReg O32IntRegs[] = {
  Cpu0::A0, Cpu0::A1
};
```

```
//@LowerCall {
/// LowerCall - functions arguments are copied from virtual regs to
/// (physical regs)/(stack frame), CALLSEQ_START and CALLSEQ_END are emitted.
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                              SmallVectorImpl<SDValue> &InVals) const {
```

```
//@LowerCall {
/// LowerCall - functions arguments are copied from virtual regs to
/// (physical regs)/(stack frame), CALLSEQ_START and CALLSEQ_END are emitted.
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                              SmallVectorImpl<SDValue> &InVals) const {
```

```
  return CLI.Chain;
```

```
}
```

```
//===----------------------------------------------------------------------===//

//@LowerFormalArguments {
/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool IsVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         const SDLoc &DL, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                          const {
  MachineFunction &MF = DAG.getMachineFunction();
  MachineFrameInfo &MFI = MF.getFrameInfo();
  Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

  Cpu0FI->setVarArgsFrameIndex(0);

  // Assign locations to all of the incoming arguments.
  SmallVector<CCValAssign, 16> ArgLocs;
  CCState CCInfo(CallConv, IsVarArg, DAG.getMachineFunction(),
                 ArgLocs, *DAG.getContext());
  Cpu0CC Cpu0CCInfo(CallConv, ABI.IsO32(),
                    CCInfo);

  const Function &Func = DAG.getMachineFunction().getFunction();
  Function::const_arg_iterator FuncArg = Func.arg_begin();

  bool UseSoftFloat = Subtarget.abiUsesSoftFloat();

  Cpu0CCInfo.analyzeFormalArguments(Ins, UseSoftFloat, FuncArg);
  Cpu0FI->setFormalArgInfo(CCInfo.getNextStackOffset(),
                           Cpu0CCInfo.hasByValArg());

  // Used with vargs to acumulate store chains.
  std::vector<SDValue> OutChains;

  unsigned CurArgIdx = 0;
  Cpu0CC::byval_iterator ByValArg = Cpu0CCInfo.byval_begin();

  //@2 {
  for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
  //@2 }
    CCValAssign &VA = ArgLocs[i];
    if (Ins[i].isOrigArg()) {
      std::advance(FuncArg, Ins[i].getOrigArgIndex() - CurArgIdx);
      CurArgIdx = Ins[i].getOrigArgIndex();
    }
    EVT ValVT = VA.getValVT();
    ISD::ArgFlagsTy Flags = Ins[i].Flags;
    bool IsRegLoc = VA.isRegLoc();
```

(continues on next page)

(continued from previous page)

```
  //@byval pass {
  if (Flags.isByVal()) {
    assert(Flags.getByValSize() &&
           "ByVal args of size 0 should have been ignored by front-end.");
    assert(ByValArg != Cpu0CCInfo.byval_end());
    copyByValRegs(Chain, DL, OutChains, DAG, Flags, InVals, &*FuncArg,
                  Cpu0CCInfo, *ByValArg);
    ++ByValArg;
    continue;
  }
  //@byval pass }
  // Arguments stored on registers
  if (ABI.IsO32() && IsRegLoc) {
    MVT RegVT = VA.getLocVT();
    unsigned ArgReg = VA.getLocReg();
    const TargetRegisterClass *RC = getRegClassFor(RegVT);

    // Transform the arguments stored on
    // physical registers into virtual ones
    unsigned Reg = addLiveIn(DAG.getMachineFunction(), ArgReg, RC);
    SDValue ArgValue = DAG.getCopyFromReg(Chain, DL, Reg, RegVT);

    // If this is an 8 or 16-bit value, it has been passed promoted
    // to 32 bits.  Insert an assert[sz]ext to capture this, then
    // truncate to the right size.
    if (VA.getLocInfo() != CCValAssign::Full) {
      unsigned Opcode = 0;
      if (VA.getLocInfo() == CCValAssign::SExt)
        Opcode = ISD::AssertSext;
      else if (VA.getLocInfo() == CCValAssign::ZExt)
        Opcode = ISD::AssertZext;
      if (Opcode)
        ArgValue = DAG.getNode(Opcode, DL, RegVT, ArgValue,
                               DAG.getValueType(ValVT));
      ArgValue = DAG.getNode(ISD::TRUNCATE, DL, ValVT, ArgValue);
    }

    // Handle floating point arguments passed in integer registers.
    if ((RegVT == MVT::i32 && ValVT == MVT::f32) ||
        (RegVT == MVT::i64 && ValVT == MVT::f64))
      ArgValue = DAG.getNode(ISD::BITCAST, DL, ValVT, ArgValue);
    InVals.push_back(ArgValue);
  } else { // VA.isRegLoc()
    MVT LocVT = VA.getLocVT();

    // sanity check
    assert(VA.isMemLoc());

    // The stack pointer offset is relative to the caller stack frame.
    int FI = MFI.CreateFixedObject(ValVT.getSizeInBits()/8,
                                   VA.getLocMemOffset(), true);
```

(continues on next page)

(continued from previous page)

```
      // Create load nodes to retrieve arguments from the stack
      SDValue FIN = DAG.getFrameIndex(FI, getPointerTy(DAG.getDataLayout()));
      SDValue Load = DAG.getLoad(
          LocVT, DL, Chain, FIN,
          MachinePointerInfo::getFixedStack(DAG.getMachineFunction(), FI));
      InVals.push_back(Load);
      OutChains.push_back(Load.getValue(1));
    }
  }

//@Ordinary struct type: 1 {
  for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
    // The cpu0 ABIs for returning structs by value requires that we copy
    // the sret argument into $v0 for the return. Save the argument into
    // a virtual register so that we can access it from the return points.
    if (Ins[i].Flags.isSRet()) {
      unsigned Reg = Cpu0FI->getSRetReturnReg();
      if (!Reg) {
        Reg = MF.getRegInfo().createVirtualRegister(
            getRegClassFor(MVT::i32));
        Cpu0FI->setSRetReturnReg(Reg);
      }
      SDValue Copy = DAG.getCopyToReg(DAG.getEntryNode(), DL, Reg, InVals[i]);
      Chain = DAG.getNode(ISD::TokenFactor, DL, MVT::Other, Copy, Chain);
      break;
    }
  }
//@Ordinary struct type: 1 }

  // All stores are grouped in one node to allow the matching between
  // the size of Ins and InVals. This only happens when on varg functions
  if (!OutChains.empty()) {
    OutChains.push_back(Chain);
    Chain = DAG.getNode(ISD::TokenFactor, DL, MVT::Other, OutChains);
  }

  return Chain;
}
// @LowerFormalArguments }

//===----------------------------------------------------------------------===//
```

```
void Cpu0TargetLowering::Cpu0CC::
analyzeFormalArguments(const SmallVectorImpl<ISD::InputArg> &Args,
                       bool IsSoftFloat, Function::const_arg_iterator FuncArg) {
  unsigned NumArgs = Args.size();
  llvm::CCAssignFn *FixedFn = fixedArgFn();
  unsigned CurArgIdx = 0;

  for (unsigned I = 0; I != NumArgs; ++I) {
    MVT ArgVT = Args[I].VT;
    ISD::ArgFlagsTy ArgFlags = Args[I].Flags;
```

(continues on next page)

(continued from previous page)

```
    if (Args[I].isOrigArg()) {
      std::advance(FuncArg, Args[I].getOrigArgIndex() - CurArgIdx);
      CurArgIdx = Args[I].getOrigArgIndex();
    }
    CurArgIdx = Args[I].OrigArgIndex;

    if (ArgFlags.isByVal()) {
      handleByValArg(I, ArgVT, ArgVT, CCValAssign::Full, ArgFlags);
      continue;
    }

    MVT RegVT = getRegVT(ArgVT, IsSoftFloat);

    if (!FixedFn(I, ArgVT, RegVT, CCValAssign::Full, ArgFlags, CCInfo))
      continue;

#ifndef NDEBUG
    dbgs() << "Formal Arg #" << I << " has unhandled type "
           << EVT(ArgVT).getEVTString();
#endif
    llvm_unreachable(nullptr);
  }
}
```

```
void Cpu0TargetLowering::Cpu0CC::handleByValArg(unsigned ValNo, MVT ValVT,
                                                MVT LocVT,
                                                CCValAssign::LocInfo LocInfo,
                                                ISD::ArgFlagsTy ArgFlags) {
  assert(ArgFlags.getByValSize() && "Byval argument's size shouldn't be 0.");

  struct ByValArgInfo ByVal;
  unsigned RegSize = regSize();
  unsigned ByValSize = alignTo(ArgFlags.getByValSize(), RegSize);
  Align Alignment = std::min(std::max(ArgFlags.getNonZeroByValAlign(), Align(RegSize)),
                             Align(RegSize * 2));

  if (useRegsForByval())
    allocateRegs(ByVal, ByValSize, Alignment.value());

  // Allocate space on caller's stack.
  ByVal.Address = CCInfo.AllocateStack(ByValSize - RegSize * ByVal.NumRegs,
                                       Alignment);
  CCInfo.addLoc(CCValAssign::getMem(ValNo, ValVT, ByVal.Address, LocVT,
                                    LocInfo));
  ByValArgs.push_back(ByVal);
}

unsigned Cpu0TargetLowering::Cpu0CC::numIntArgRegs() const {
  return IsO32 ? array_lengthof(O32IntRegs) : 0;
}
```

```
const ArrayRef<MCPhysReg> Cpu0TargetLowering::Cpu0CC::intArgRegs() const {
  return makeArrayRef(O32IntRegs);
}

llvm::CCAssignFn *Cpu0TargetLowering::Cpu0CC::fixedArgFn() const {
  if (IsO32)
    return CC_Cpu0O32;
  else // IsS32
    return CC_Cpu0S32;
}
```

```
void Cpu0TargetLowering::Cpu0CC::allocateRegs(ByValArgInfo &ByVal,
                                              unsigned ByValSize,
                                              unsigned Align) {
  unsigned RegSize = regSize(), NumIntArgRegs = numIntArgRegs();
  const ArrayRef<MCPhysReg> IntArgRegs = intArgRegs();
  assert(!(ByValSize % RegSize) && !(Align % RegSize) &&
         "Byval argument's size and alignment should be a multiple of"
         "RegSize.");

  ByVal.FirstIdx = CCInfo.getFirstUnallocated(IntArgRegs);

  // If Align > RegSize, the first arg register must be even.
  if ((Align > RegSize) && (ByVal.FirstIdx % 2)) {
    CCInfo.AllocateReg(IntArgRegs[ByVal.FirstIdx]);
    ++ByVal.FirstIdx;
  }

  // Mark the registers allocated.
  for (unsigned I = ByVal.FirstIdx; ByValSize && (I < NumIntArgRegs);
       ByValSize -= RegSize, ++I, ++ByVal.NumRegs)
    CCInfo.AllocateReg(IntArgRegs[I]);
}
```

Refresh "section Global variable"[4], we handled global variable translation by creating the IR DAG in LowerGlobalAddress() first, and then finish the Instruction Selection according their corresponding machine instruction DAGs in Cpu0InstrInfo.td. LowerGlobalAddress() is called when `llc` meets the global variable access. LowerFormalArguments() work in the same way. It is called when function is entered. It gets incoming arguments information by CCInfo(CallConv,..., ArgLocs, ...) before entering **"for loop"**. In ch9_1.cpp, there are 6 arguments in sum_i(...) function call. So ArgLocs.size() is 6, each argument information is in ArgLocs[i]. When VA.isRegLoc() is true, meaning the arguement passes in register. On the contrary, when VA.isMemLoc() is true, meaning the arguement pass in memory stack. When passing in register, it marks the register "live in" and copy directly from the register. When passing in memory stack, it creates stack offset for this frame index object and load node with the created stack offset, and then puts the load node into vector InVals.

When `llc -cpu0-s32-calls=false` it passes first two arguments registers and the other arguments in stack frame. When `llc -cpu0-s32-calls=true` it passes all arguments in stack frame.

Before taking care the arguments as above, it calls analyzeFormalArguments(). In analyzeFormalArguments() it calls fixedArgFn() which return the function pointer of CC_Cpu0O32() or CC_Cpu0S32(). ArgFlags.isByVal() will be true when it meets "struct pointer byval" keyword, such as "%struct.S* byval" in tailcall.ll. When `llc -cpu0-s32-calls=false` the stack offset begin from 8 (in case the arguement registers need spill out) while `llc -cpu0-s32-calls=true` stack offset begin from 0.

[4] http://jonathan2251.github.io/lbd/globalvar.html#global-variable

For instance of example code ch9_1.cpp with `llc -cpu0-s32-calls=true` (using memory stack only to pass arguments), LowerFormalArguments() will be called twice. First time is for sum_i() which will create 6 "load DAGs" for 6 incoming arguments passing into this function. Second time is for main() which won't create any "load DAG" since no incoming argument passing into main(). In addition to LowerFormalArguments() which creates the "load DAG", we need loadRegFromStackSlot() (defined in the early chapter) to issue the machine instruction **"ld $r, offset($sp)"** to load incoming arguments from stack frame offset. GetMemOperand(..., FI, ...) return the Memory location of the frame index variable, which is the offset.

For input ch9_incoming.cpp as below, LowerFormalArguments() will generate the red box parts of DAG nodes shown as the next Fig. 9.3 and Fig. 9.4 for `llc -cpu0-s32-calls=true` and `llc -cpu0-s32-calls=false`, respectively. The root node at bottom is created by

lbdex/input/ch9_incoming.cpp

```
int sum_i(int x1, int x2, int x3)
{
  int sum = x1 + x2 + x3;

  return sum;
}
```

```
JonathantekiiMac:input Jonathan$ clang -O3 -target mips-unknown-linux-gnu -c
ch9_incoming.cpp -emit-llvm -o ch9_incoming.bc
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-dis ch9_incoming.bc -o -
...
define i32 @_Z5sum_iiii(i32 %x1, i32 %x2, i32 %x3) #0 {
  %1 = add nsw i32 %x2, %x1
  %2 = add nsw i32 %1, %x3
  ret i32 %2
}
```

In addition to Calling Convention and LowerFormalArguments(), Chapter9_1/ adds the following code for the instruction selection and printing of Cpu0 instructions **swi** (Software Interrupt), **jsub** and **jalr** (function call).

lbdex/chapters/Chapter9_1/Cpu0InstrInfo.td

```
def SDT_Cpu0JmpLink      : SDTypeProfile<0, 1, [SDTCisVT<0, iPTR>]>;
```

```
// Call
def Cpu0JmpLink : SDNode<"Cpu0ISD::JmpLink",SDT_Cpu0JmpLink,
                         [SDNPHasChain, SDNPOutGlue, SDNPOptInGlue,
                          SDNPVariadic]>;
```

```
class IsTailCall {
  bit isCall = 1;
  bit isTerminator = 1;
  bit isReturn = 1;
  bit isBarrier = 1;
  bit hasExtraSrcRegAllocReq = 1;
```

(continues on next page)

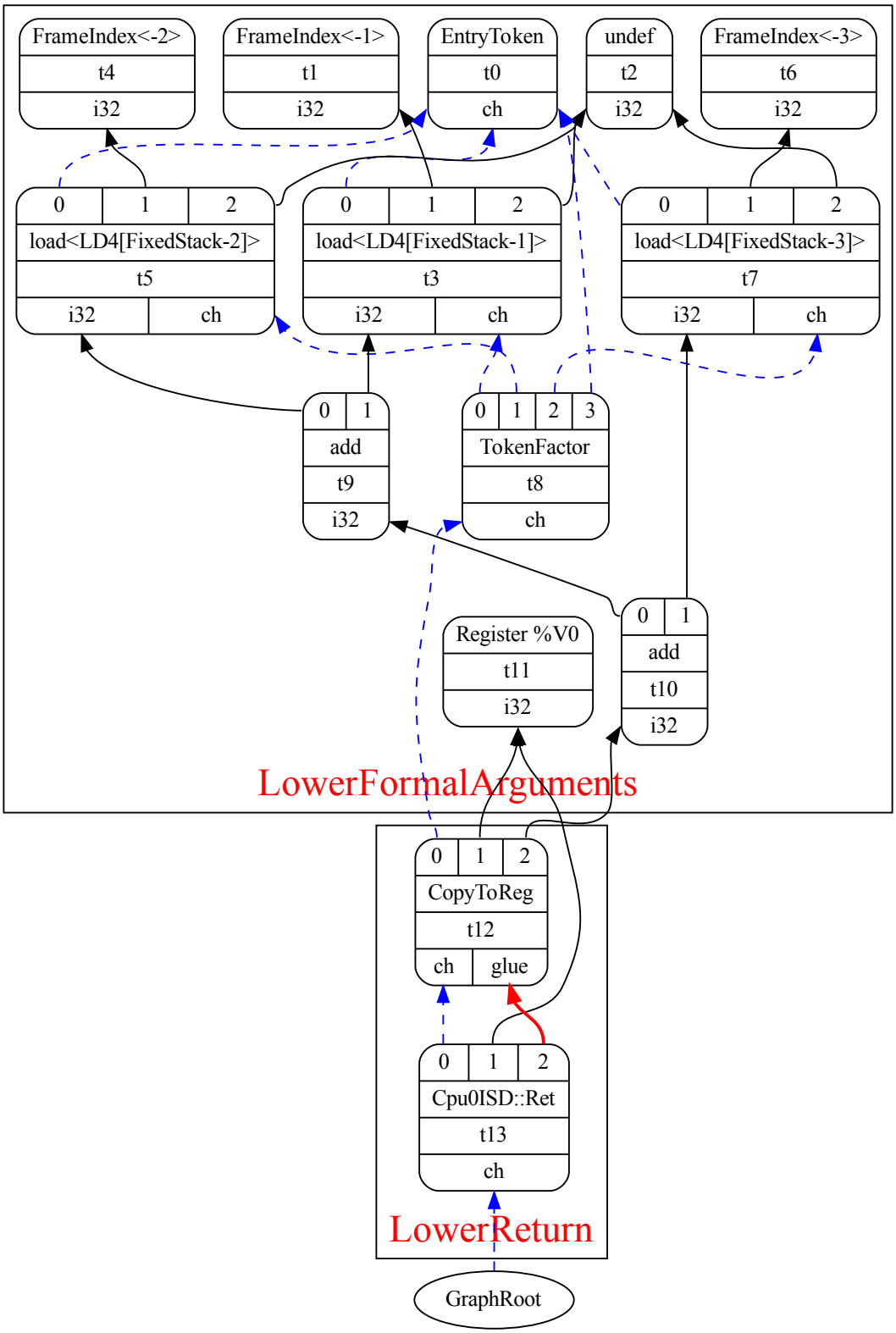


Fig. 9.3: Incoming arguments DAG created for ch9_incoming.cpp with -cpu0-s32-calls=true

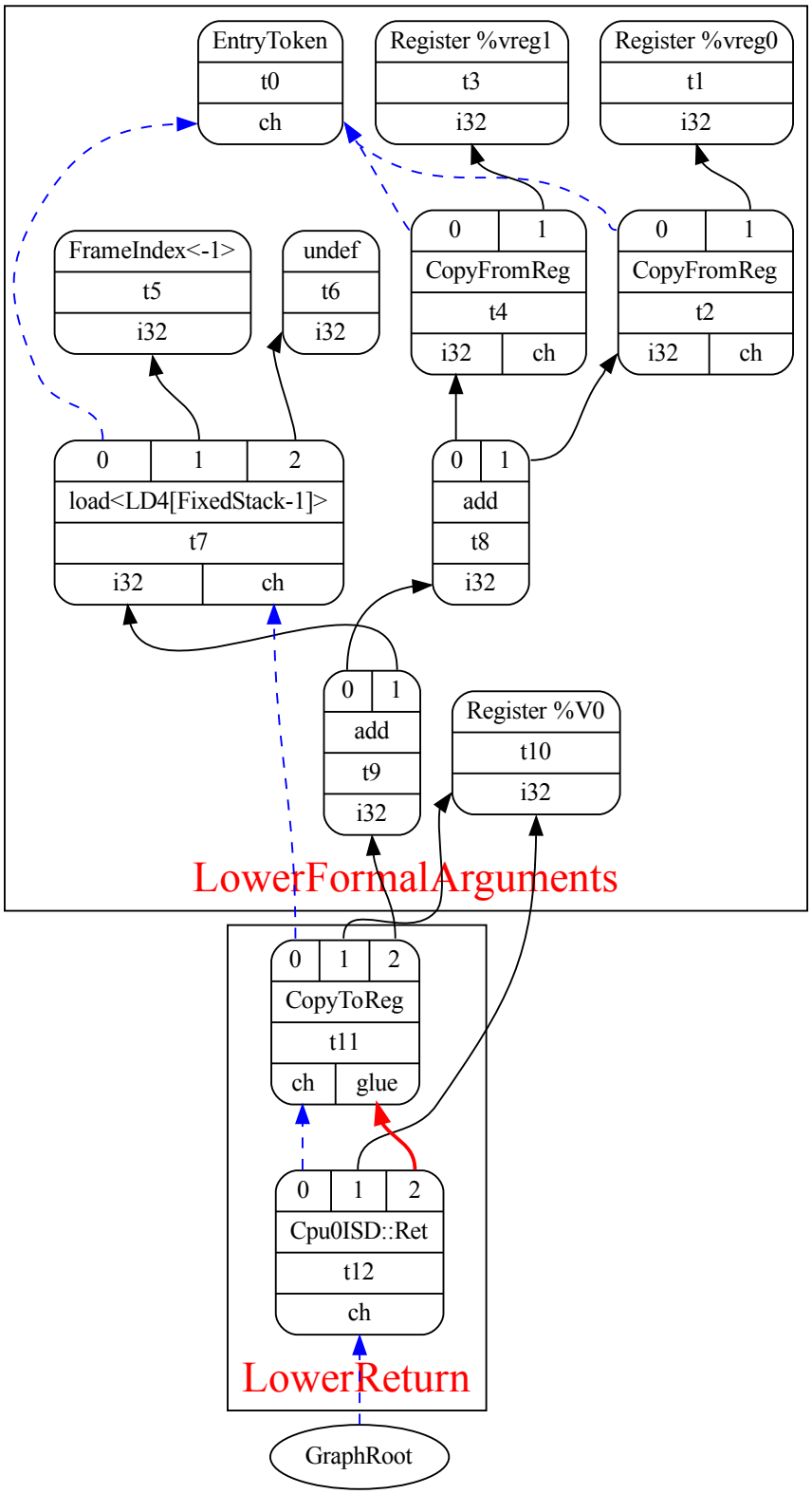


Fig. 9.4: Incoming arguments DAG created for ch9_incoming.cpp with -cpu0-s32-calls=false

(continued from previous page)

```
  bit isCodeGenOnly = 1;
}
```

```
def calltarget  : Operand<iPTR> {
  let EncoderMethod = "getJumpTargetOpValue";
  let OperandType = "OPERAND_PCREL";
}
```

```
let Predicates = [Ch9_1] in {
// Jump and Link (Call)
let isCall=1, hasDelaySlot=1 in {
  //@JumpLink {
  class JumpLink<bits<8> op, string instr_asm>:
    FJ<op, (outs), (ins calltarget:$target, variable_ops),
       !strconcat(instr_asm, "\t$target"), [(Cpu0JmpLink imm:$target)],
       IIBranch> {
//#if CH >= CH10_1 2
       let DecoderMethod = "DecodeJumpTarget";
//#endif
       }
  //@JumpLink }

  class JumpLinkReg<bits<8> op, string instr_asm,
                    RegisterClass RC>:
    FA<op, (outs), (ins RC:$rb, variable_ops),
       !strconcat(instr_asm, "\t$rb"), [(Cpu0JmpLink RC:$rb)], IIBranch> {
    let rc = 0;
    let ra = 14;
    let shamt = 0;
  }
}
```

```
/// Jump & link and Return Instructions
let Predicates = [Ch9_1] in {
def JSUB    : JumpLink<0x3b, "jsub">;
}
```

```
let Predicates = [Ch9_1] in {
def JALR    : JumpLinkReg<0x39, "jalr", GPROut>;
}
```

```
let Predicates = [Ch9_1] in {
def : Pat<(Cpu0JmpLink (i32 tglobaladdr:$dst)),
          (JSUB tglobaladdr:$dst)>;
def : Pat<(Cpu0JmpLink (i32 texternalsym:$dst)),
          (JSUB texternalsym:$dst)>;
```

```
}
```

lbdex/chapters/Chapter9_1/Cpu0MCInstLower.cpp

```
MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,
                                              MachineOperandType MOTy,
                                              unsigned Offset) const {
  MCSymbolRefExpr::VariantKind Kind = MCSymbolRefExpr::VK_None;
  Cpu0MCExpr::Cpu0ExprKind TargetKind = Cpu0MCExpr::CEK_None;
  const MCSymbol *Symbol;

  switch(MO.getTargetFlags()) {
```

```
  case Cpu0II::MO_GOT_CALL:
    TargetKind = Cpu0MCExpr::CEK_GOT_CALL;
    break;
```

```
  ...
  }
  switch (MOTy) {
. ...
```

```
  case MachineOperand::MO_ExternalSymbol:
    Symbol = AsmPrinter.GetExternalSymbolSymbol(MO.getSymbolName());
    Offset += MO.getOffset();
    break;
```

```
  ...
  }
  ...
}
```

```
MCOperand Cpu0MCInstLower::LowerOperand(const MachineOperand& MO,
                                        unsigned offset) const {
  MachineOperandType MOTy = MO.getType();

  switch (MOTy) {
  //@2
```

```
  case MachineOperand::MO_ExternalSymbol:
```

```
    return LowerSymbolOperand(MO, MOTy, offset);
```

```
  ...
  }
  ...
}
```

lbdex/chapters/Chapter9_1/MCTargetDesc/Cpu0AsmBackend.cpp

```
// Prepare value for the target space for it
static unsigned adjustFixupValue(const MCFixup &Fixup, uint64_t Value,
                                 MCContext &Ctx) {

  unsigned Kind = Fixup.getKind();

  // Add/subtract and shift
  switch (Kind) {
```

```
  case Cpu0::fixup_Cpu0_CALL16:
```

```
  ...
  }
  ...
}
```

lbdex/chapters/Chapter9_1/MCTargetDesc/Cpu0ELFObjectWriter.cpp

```
unsigned Cpu0ELFObjectWriter::getRelocType(MCContext &Ctx,
                                           const MCValue &Target,
                                           const MCFixup &Fixup,
                                           bool IsPCRel) const {
  // determine the type of the relocation
  unsigned Type = (unsigned)ELF::R_CPU0_NONE;
  unsigned Kind = (unsigned)Fixup.getKind();

  switch (Kind) {
```

```
  case Cpu0::fixup_Cpu0_CALL16:
    Type = ELF::R_CPU0_CALL16;
    break;
```

```
  ...
  }
  ...
}
```

lbdex/chapters/Chapter9_1/MCTargetDesc/Cpu0FixupKinds.h

```
  enum Fixups {
```

```
    // resulting in - R_CPU0_CALL16.
    fixup_Cpu0_CALL16,
```

```
    ...
. }
```

lbdex/chapters/Chapter9_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
unsigned Cpu0MCCodeEmitter::
getJumpTargetOpValue(const MCInst &MI, unsigned OpNo,
                     SmallVectorImpl<MCFixup> &Fixups,
                     const MCSubtargetInfo &STI) const {
```

```
  if (Opcode == Cpu0::JSUB || Opcode == Cpu0::JMP || Opcode == Cpu0::BAL)
#elif CH >= CH8_2 //1
  if (Opcode == Cpu0::JMP || Opcode == Cpu0::BAL)
```

```
    Fixups.push_back(MCFixup::create(0, Expr,
                                     MCFixupKind(Cpu0::fixup_Cpu0_PC24)));
```

```
  ...
}
```

```
unsigned Cpu0MCCodeEmitter::
getExprOpValue(const MCExpr *Expr,SmallVectorImpl<MCFixup> &Fixups,
               const MCSubtargetInfo &STI) const {
```

```
//    switch(cast<MCSymbolRefExpr>(Expr)->getKind()) {
```

```
    case Cpu0MCExpr::CEK_GOT_CALL:
      FixupKind = Cpu0::fixup_Cpu0_CALL16;
      break;
```

```
  ...
  }
...
}
```

lbdex/chapters/Chapter9_1/Cpu0MachineFunction.h

```
/// Cpu0FunctionInfo - This class is derived from MachineFunction private
/// Cpu0 target-specific information for each MachineFunction.
class Cpu0FunctionInfo : public MachineFunctionInfo {
public:
  Cpu0FunctionInfo(MachineFunction& MF)
  : MF(MF),
    VarArgsFrameIndex(0),
```

```
    InArgFIRange(std::make_pair(-1, 0)),
    OutArgFIRange(std::make_pair(-1, 0)), GPFI(0), DynAllocFI(0),
```

```
  bool isInArgFI(int FI) const {
    return FI <= InArgFIRange.first && FI >= InArgFIRange.second;
  }
  void setLastInArgFI(int FI) { InArgFIRange.second = FI; }
```

(continues on next page)

(continued from previous page)

```
  bool isOutArgFI(int FI) const {
    return FI <= OutArgFIRange.first && FI >= OutArgFIRange.second;
  }
```

```
  int getGPFI() const { return GPFI; }
  void setGPFI(int FI) { GPFI = FI; }
  bool isGPFI(int FI) const { return GPFI && GPFI == FI; }
```

```

  bool isDynAllocFI(int FI) const { return DynAllocFI && DynAllocFI == FI; }
```

```
  // Range of frame object indices.
  // InArgFIRange: Range of indices of all frame objects created during call to
  //               LowerFormalArguments.
  // OutArgFIRange: Range of indices of all frame objects created during call to
  //                LowerCall except for the frame object for restoring $gp.
  std::pair<int, int> InArgFIRange, OutArgFIRange;
```

```
  mutable int DynAllocFI; // Frame index of dynamically allocated stack area.
```

```
  ...
};
```

lbdex/chapters/Chapter9_1/Cpu0SEFrameLowering.h

```
  bool spillCalleeSavedRegisters(MachineBasicBlock &MBB,
                                 MachineBasicBlock::iterator MI,
                                 ArrayRef<CalleeSavedInfo> CSI,
                                 const TargetRegisterInfo *TRI) const override;
```

lbdex/chapters/Chapter9_1/Cpu0SEFrameLowering.cpp

```
bool Cpu0SEFrameLowering::
spillCalleeSavedRegisters(MachineBasicBlock &MBB,
                          MachineBasicBlock::iterator MI,
                          ArrayRef<CalleeSavedInfo> CSI,
                          const TargetRegisterInfo *TRI) const {
  MachineFunction *MF = MBB.getParent();
  MachineBasicBlock *EntryBlock = &MF->front();
  const TargetInstrInfo &TII = *MF->getSubtarget().getInstrInfo();

  for (unsigned i = 0, e = CSI.size(); i != e; ++i) {
    // Add the callee-saved register as live-in. Do not add if the register is
    // LR and return address is taken, because it has already been added in
    // method Cpu0TargetLowering::LowerRETURNADDR.
    // It's killed at the spill, unless the register is LR and return address
    // is taken.
```

(continues on next page)

(continued from previous page)

```
    unsigned Reg = CSI[i].getReg();
    bool IsRAAndRetAddrIsTaken = (Reg == Cpu0::LR)
        && MF->getFrameInfo().isReturnAddressTaken();
    if (!IsRAAndRetAddrIsTaken)
      EntryBlock->addLiveIn(Reg);

    // Insert the spill to the stack frame.
    bool IsKill = !IsRAAndRetAddrIsTaken;
    const TargetRegisterClass *RC = TRI->getMinimalPhysRegClass(Reg);
    TII.storeRegToStackSlot(*EntryBlock, MI, Reg, IsKill,
                            CSI[i].getFrameIdx(), RC, TRI);
  }

  return true;
}
```

Both JSUB and JALR defined in Cpu0InstrInfo.td as above use Cpu0JmpLink node. They are distinguishable since JSUB use "imm" operand while JALR uses register operand.

lbdex/chapters/Chapter9_1/Cpu0InstrInfo.td

```
let Predicates = [Ch9_1] in {
def : Pat<(Cpu0JmpLink (i32 tglobaladdr:$dst)),
          (JSUB tglobaladdr:$dst)>;
def : Pat<(Cpu0JmpLink (i32 texternalsym:$dst)),
          (JSUB texternalsym:$dst)>;
```

The code tells TableGen generating pattern match code that matching the "imm" for "tglobaladdr" pattern first. If it fails then trying to match "texternalsym" next. The function you declared belongs to "tglobaladdr", (for instance the function sum_i(...) defined in ch9_1.cpp belongs to "tglobaladdr"); the function which implicitly used by llvm belongs to "texternalsym" (for instance the function "memcpy" belongs to "texternalsym"). The "memcpy" will be generated when defining a long string. The ch9_1_2.cpp is an example for generating "memcpy" function call. It will be shown in next section with Chapter9_2 example code. Cpu0GenDAGISel.inc contains pattern matched information of JSUB and JALR which generated from TablGen as follows,

```
          /*SwitchOpcode*/ 74,  TARGET_VAL(Cpu0ISD::JmpLink),// ->734
/*660*/     OPC_RecordNode,   // #0 = 'Cpu0JmpLink' chained node
/*661*/     OPC_CaptureGlueInput,
/*662*/     OPC_RecordChild1, // #1 = $target
/*663*/     OPC_Scope, 57, /*->722*/ // 2 children in Scope
/*665*/       OPC_MoveChild, 1,
/*667*/       OPC_SwitchOpcode /*3 cases */, 22,  TARGET_VAL(ISD::Constant),
// ->693
/*671*/         OPC_MoveParent,
/*672*/         OPC_EmitMergeInputChains1_0,
/*673*/         OPC_EmitConvertToTarget, 1,
/*675*/         OPC_Scope, 7, /*->684*/ // 2 children in Scope
/*684*/         /*Scope*/ 7, /*->692*/
/*685*/           OPC_MorphNodeTo, TARGET_VAL(Cpu0::JSUB), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
```

(continues on next page)

(continued from previous page)

```
                    0/*#VTs*/, 1/*#Ops*/, 2,
                  // Src: (Cpu0JmpLink (imm:iPTR):$target) - Complexity = 6
                  // Dst: (JSUB (imm:iPTR):$target)
/*692*/          0, /*End of Scope*/
               /*SwitchOpcode*/ 11,  TARGET_VAL(ISD::TargetGlobalAddress),// ->707
/*696*/          OPC_CheckType, MVT::i32,
/*698*/          OPC_MoveParent,
/*699*/          OPC_EmitMergeInputChains1_0,
/*700*/          OPC_MorphNodeTo, TARGET_VAL(Cpu0::JSUB), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
                    0/*#VTs*/, 1/*#Ops*/, 1,
                  // Src: (Cpu0JmpLink (tglobaladdr:i32):$dst) - Complexity = 6
                  // Dst: (JSUB (tglobaladdr:i32):$dst)
               /*SwitchOpcode*/ 11,  TARGET_VAL(ISD::TargetExternalSymbol),// ->721
/*710*/          OPC_CheckType, MVT::i32,
/*712*/          OPC_MoveParent,
/*713*/          OPC_EmitMergeInputChains1_0,
/*714*/          OPC_MorphNodeTo, TARGET_VAL(Cpu0::JSUB), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
                    0/*#VTs*/, 1/*#Ops*/, 1,
                  // Src: (Cpu0JmpLink (texternalsym:i32):$dst) - Complexity = 6
                  // Dst: (JSUB (texternalsym:i32):$dst)
               0, // EndSwitchOpcode
/*722*/      /*Scope*/ 10, /*->733*/
/*723*/        OPC_CheckChild1Type, MVT::i32,
/*725*/        OPC_EmitMergeInputChains1_0,
/*726*/        OPC_MorphNodeTo, TARGET_VAL(Cpu0::JALR), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
                    0/*#VTs*/, 1/*#Ops*/, 1,
                  // Src: (Cpu0JmpLink CPURegs:i32:$rb) - Complexity = 3
                  // Dst: (JALR CPURegs:i32:$rb)
/*733*/      0, /*End of Scope*/
```

After above changes, you can run Chapter9_1/ with ch9_1.cpp and see what happens in the following,

```
118-165-79-83:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch9_1.bc -o ch9_1.cpu0.s
Assertion failed: ((CLI.IsTailCall || InVals.size() == CLI.Ins.size()) &&
"LowerCall didn't emit the correct number of values!"), function LowerCallTo,
file /Users/Jonathan/llvm/test/llvm/lib/CodeGen/SelectionDAG/SelectionDAGBuilder.
cpp, ...
...
0.  Program arguments: /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_1.bc -o
ch9_1.cpu0.s
1.  Running pass 'Function Pass Manager' on module 'ch9_1.bc'.
2.  Running pass 'CPU0 DAG->DAG Pattern Instruction Selection' on function
'@main'
Illegal instruction: 4
```

Now, the LowerFormalArguments() has the correct number, but LowerCall() has not the correct number of values!

9.3 Store outgoing arguments to stack frame

Fig. 9.2 depicts two steps to take care arguments passing. One is store outgoing arguments into caller function, the other is load incoming arguments into callee function. We defined LowerFormalArguments() for **"load incoming arguments"** in callee function last section. Now, we will finish **"store outgoing arguments"** in caller function. LowerCall() is responsible in doing this. The implementation as follows,

lbdex/chapters/Chapter9_2/Cpu0MachineFunction.h

```
  /// Create a MachinePointerInfo that has an ExternalSymbolPseudoSourceValue
  /// object representing a GOT entry for an external function.
  MachinePointerInfo callPtrInfo(const char *ES);

  /// Create a MachinePointerInfo that has a GlobalValuePseudoSourceValue object
  /// representing a GOT entry for a global function.
  MachinePointerInfo callPtrInfo(const GlobalValue *GV);
```

lbdex/chapters/Chapter9_2/Cpu0MachineFunction.cpp

```
MachinePointerInfo Cpu0FunctionInfo::callPtrInfo(const char *ES) {
  return MachinePointerInfo(MF.getPSVManager().getExternalSymbolCallEntry(ES));
}

MachinePointerInfo Cpu0FunctionInfo::callPtrInfo(const GlobalValue *GV) {
  return MachinePointerInfo(MF.getPSVManager().getGlobalValueCallEntry(GV));
}
```

lbdex/chapters/Chapter9_2/Cpu0ISelLowering.h

```
    /// This function fills Ops, which is the list of operands that will later
    /// be used when a function call node is created. It also generates
    /// copyToReg nodes to set up argument registers.
    virtual void
    getOpndList(SmallVectorImpl<SDValue> &Ops,
                std::deque< std::pair<unsigned, SDValue> > &RegsToPass,
                bool IsPICCall, bool GlobalOrExternal, bool InternalLinkage,
                CallLoweringInfo &CLI, SDValue Callee, SDValue Chain) const;
```

```
    /// Cpu0CC - This class provides methods used to analyze formal and call
    /// arguments and inquire about calling convention information.
    class Cpu0CC {
```

```
      void analyzeCallOperands(const SmallVectorImpl<ISD::OutputArg> &Outs,
                               bool IsVarArg, bool IsSoftFloat,
                               const SDNode *CallNode,
                               std::vector<ArgListEntry> &FuncArgs);
```

```
.  };
```

```
    Cpu0CC::SpecialCallingConvType getSpecialCallingConv(SDValue Callee) const;
```

```
    // Lower Operand helpers
    SDValue LowerCallResult(SDValue Chain, SDValue InFlag,
                            CallingConv::ID CallConv, bool isVarArg,
                            const SmallVectorImpl<ISD::InputArg> &Ins,
                            const SDLoc &dl, SelectionDAG &DAG,
                            SmallVectorImpl<SDValue> &InVals,
                            const SDNode *CallNode, const Type *RetTy) const;
```

```
    /// passByValArg - Pass a byval argument in registers or on stack.
    void passByValArg(SDValue Chain, const SDLoc &DL,
                      std::deque< std::pair<unsigned, SDValue> > &RegsToPass,
                      SmallVectorImpl<SDValue> &MemOpChains, SDValue StackPtr,
                      MachineFrameInfo &MFI, SelectionDAG &DAG, SDValue Arg,
                      const Cpu0CC &CC, const ByValArgInfo &ByVal,
                      const ISD::ArgFlagsTy &Flags, bool isLittle) const;
```

```
    SDValue passArgOnStack(SDValue StackPtr, unsigned Offset, SDValue Chain,
                           SDValue Arg, const SDLoc &DL, bool IsTailCall,
                           SelectionDAG &DAG) const;
```

```
    bool CanLowerReturn(CallingConv::ID CallConv, MachineFunction &MF,
                        bool isVarArg,
                        const SmallVectorImpl<ISD::OutputArg> &Outs,
                        LLVMContext &Context) const override;
```

lbdex/chapters/Chapter9_2/Cpu0ISelLowering.cpp

```
SDValue
Cpu0TargetLowering::passArgOnStack(SDValue StackPtr, unsigned Offset,
                                   SDValue Chain, SDValue Arg, const SDLoc &DL,
                                   bool IsTailCall, SelectionDAG &DAG) const {
  if (!IsTailCall) {
    SDValue PtrOff =
        DAG.getNode(ISD::ADD, DL, getPointerTy(DAG.getDataLayout()), StackPtr,
                    DAG.getIntPtrConstant(Offset, DL));
    return DAG.getStore(Chain, DL, Arg, PtrOff, MachinePointerInfo());
  }

  MachineFrameInfo &MFI = DAG.getMachineFunction().getFrameInfo();
  int FI = MFI.CreateFixedObject(Arg.getValueSizeInBits() / 8, Offset, false);
  SDValue FIN = DAG.getFrameIndex(FI, getPointerTy(DAG.getDataLayout()));
  return DAG.getStore(Chain, DL, Arg, FIN, MachinePointerInfo(),
                      /* Alignment = */ 0, MachineMemOperand::MOVolatile);
}

void Cpu0TargetLowering::
```

(continues on next page)

(continued from previous page)

```
getOpndList(SmallVectorImpl<SDValue> &Ops,
            std::deque< std::pair<unsigned, SDValue> > &RegsToPass,
            bool IsPICCall, bool GlobalOrExternal, bool InternalLinkage,
            CallLoweringInfo &CLI, SDValue Callee, SDValue Chain) const {
  // T9 should contain the address of the callee function if
  // -reloction-model=pic or it is an indirect call.
  if (IsPICCall || !GlobalOrExternal) {
    unsigned T9Reg = Cpu0::T9;
    RegsToPass.push_front(std::make_pair(T9Reg, Callee));
  } else
    Ops.push_back(Callee);

  // Insert node "GP copy globalreg" before call to function.
  //
  // R_CPU0_CALL* operators (emitted when non-internal functions are called
  // in PIC mode) allow symbols to be resolved via lazy binding.
  // The lazy binding stub requires GP to point to the GOT.
  if (IsPICCall && !InternalLinkage) {
    unsigned GPReg = Cpu0::GP;
    EVT Ty = MVT::i32;
    RegsToPass.push_back(std::make_pair(GPReg, getGlobalReg(CLI.DAG, Ty)));
  }

  // Build a sequence of copy-to-reg nodes chained together with token
  // chain and flag operands which copy the outgoing args into registers.
  // The InFlag in necessary since all emitted instructions must be
  // stuck together.
  SDValue InFlag;

  for (unsigned i = 0, e = RegsToPass.size(); i != e; ++i) {
    Chain = CLI.DAG.getCopyToReg(Chain, CLI.DL, RegsToPass[i].first,
                                 RegsToPass[i].second, InFlag);
    InFlag = Chain.getValue(1);
  }

  // Add argument registers to the end of the list so that they are
  // known live into the call.
  for (unsigned i = 0, e = RegsToPass.size(); i != e; ++i)
    Ops.push_back(CLI.DAG.getRegister(RegsToPass[i].first,
                                      RegsToPass[i].second.getValueType()));

  // Add a register mask operand representing the call-preserved registers.
  const TargetRegisterInfo *TRI = Subtarget.getRegisterInfo();
  const uint32_t *Mask =
      TRI->getCallPreservedMask(CLI.DAG.getMachineFunction(), CLI.CallConv);
  assert(Mask && "Missing call preserved mask for calling convention");
  Ops.push_back(CLI.DAG.getRegisterMask(Mask));

  if (InFlag.getNode())
    Ops.push_back(InFlag);
}
```

```
/// LowerCall - functions arguments are copied from virtual regs to
/// (physical regs)/(stack frame), CALLSEQ_START and CALLSEQ_END are emitted.
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                              SmallVectorImpl<SDValue> &InVals) const {
  SelectionDAG &DAG                     = CLI.DAG;
  SDLoc DL                              = CLI.DL;
  SmallVectorImpl<ISD::OutputArg> &Outs = CLI.Outs;
  SmallVectorImpl<SDValue> &OutVals     = CLI.OutVals;
  SmallVectorImpl<ISD::InputArg> &Ins   = CLI.Ins;
  SDValue Chain                         = CLI.Chain;
  SDValue Callee                        = CLI.Callee;
  bool &IsTailCall                      = CLI.IsTailCall;
  CallingConv::ID CallConv              = CLI.CallConv;
  bool IsVarArg                         = CLI.IsVarArg;

  MachineFunction &MF = DAG.getMachineFunction();
  MachineFrameInfo &MFI = MF.getFrameInfo();
  const TargetFrameLowering *TFL = MF.getSubtarget().getFrameLowering();
  Cpu0FunctionInfo *FuncInfo = MF.getInfo<Cpu0FunctionInfo>();
  bool IsPIC = isPositionIndependent();
  Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

  // Analyze operands of the call, assigning locations to each operand.
  SmallVector<CCValAssign, 16> ArgLocs;
  CCState CCInfo(CallConv, IsVarArg, DAG.getMachineFunction(),
                 ArgLocs, *DAG.getContext());
  Cpu0CC::SpecialCallingConvType SpecialCallingConv =
    getSpecialCallingConv(Callee);
  Cpu0CC Cpu0CCInfo(CallConv, ABI.IsO32(),
                    CCInfo, SpecialCallingConv);

  Cpu0CCInfo.analyzeCallOperands(Outs, IsVarArg,
                                 Subtarget.abiUsesSoftFloat(),
                                 Callee.getNode(), CLI.getArgs());

  // Get a count of how many bytes are to be pushed on the stack.
  unsigned NextStackOffset = CCInfo.getNextStackOffset();

  //@TailCall 1 {
  // Check if it's really possible to do a tail call.
  if (IsTailCall)
    IsTailCall =
      isEligibleForTailCallOptimization(Cpu0CCInfo, NextStackOffset,
                                        *MF.getInfo<Cpu0FunctionInfo>());

  if (!IsTailCall && CLI.CB && CLI.CB->isMustTailCall())
    report_fatal_error("failed to perform tail call elimination on a call "
                       "site marked musttail");

  if (IsTailCall)
    ++NumTailCalls;
  //@TailCall 1 }
```

(continues on next page)

(continued from previous page)

```
// Chain is the output chain of the last Load/Store or CopyToReg node.
// ByValChain is the output chain of the last Memcpy node created for copying
// byval arguments to the stack.
unsigned StackAlignment = TFL->getStackAlignment();
NextStackOffset = alignTo(NextStackOffset, StackAlignment);
SDValue NextStackOffsetVal = DAG.getIntPtrConstant(NextStackOffset, DL, true);

//@TailCall 2 {
if (!IsTailCall)
  Chain = DAG.getCALLSEQ_START(Chain, NextStackOffset, 0, DL);
//@TailCall 2 }

SDValue StackPtr =
    DAG.getCopyFromReg(Chain, DL, Cpu0::SP,
                       getPointerTy(DAG.getDataLayout()));

// With EABI is it possible to have 16 args on registers.
std::deque< std::pair<unsigned, SDValue> > RegsToPass;
SmallVector<SDValue, 8> MemOpChains;
Cpu0CC::byval_iterator ByValArg = Cpu0CCInfo.byval_begin();

//@1 {
// Walk the register/memloc assignments, inserting copies/loads.
for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
//@1 }
  SDValue Arg = OutVals[i];
  CCValAssign &VA = ArgLocs[i];
  MVT LocVT = VA.getLocVT();
  ISD::ArgFlagsTy Flags = Outs[i].Flags;

  //@ByVal Arg {
  if (Flags.isByVal()) {
    assert(Flags.getByValSize() &&
           "ByVal args of size 0 should have been ignored by front-end.");
    assert(ByValArg != Cpu0CCInfo.byval_end());
    assert(!IsTailCall &&
           "Do not tail-call optimize if there is a byval argument.");
    passByValArg(Chain, DL, RegsToPass, MemOpChains, StackPtr, MFI, DAG, Arg,
                 Cpu0CCInfo, *ByValArg, Flags, Subtarget.isLittle());
    ++ByValArg;
    continue;
  }
  //@ByVal Arg }

  // Promote the value if needed.
  switch (VA.getLocInfo()) {
  default: llvm_unreachable("Unknown loc info!");
  case CCValAssign::Full:
    break;
  case CCValAssign::SExt:
    Arg = DAG.getNode(ISD::SIGN_EXTEND, DL, LocVT, Arg);
```

(continues on next page)

(continued from previous page)

```
    break;
  case CCValAssign::ZExt:
    Arg = DAG.getNode(ISD::ZERO_EXTEND, DL, LocVT, Arg);
    break;
  case CCValAssign::AExt:
    Arg = DAG.getNode(ISD::ANY_EXTEND, DL, LocVT, Arg);
    break;
  }

  // Arguments that can be passed on register must be kept at
  // RegsToPass vector
  if (VA.isRegLoc()) {
    RegsToPass.push_back(std::make_pair(VA.getLocReg(), Arg));
    continue;
  }

  // Register can't get to this point...
  assert(VA.isMemLoc());

  // emit ISD::STORE whichs stores the
  // parameter value to a stack Location
  MemOpChains.push_back(passArgOnStack(StackPtr, VA.getLocMemOffset(),
                                       Chain, Arg, DL, IsTailCall, DAG));
}

// Transform all store nodes into one single node because all store
// nodes are independent of each other.
if (!MemOpChains.empty())
  Chain = DAG.getNode(ISD::TokenFactor, DL, MVT::Other, MemOpChains);

// If the callee is a GlobalAddress/ExternalSymbol node (quite common, every
// direct call is) turn it into a TargetGlobalAddress/TargetExternalSymbol
// node so that legalize doesn't hack it.
bool IsPICCall = IsPIC; // true if calls are translated to
                                       // jalr $t9
bool GlobalOrExternal = false, InternalLinkage = false;
EVT Ty = Callee.getValueType();

if (GlobalAddressSDNode *G = dyn_cast<GlobalAddressSDNode>(Callee)) {
  if (IsPICCall) {
    const GlobalValue *Val = G->getGlobal();
    InternalLinkage = Val->hasInternalLinkage();

    if (InternalLinkage)
      Callee = getAddrLocal(G, Ty, DAG);
    else
      Callee = getAddrGlobal(G, Ty, DAG, Cpu0II::MO_GOT_CALL, Chain,
                             FuncInfo->callPtrInfo(Val));
  } else
    Callee = DAG.getTargetGlobalAddress(G->getGlobal(), DL,
                                        getPointerTy(DAG.getDataLayout()), 0,
                                        Cpu0II::MO_NO_FLAG);
```

(continues on next page)

(continued from previous page)

```
    GlobalOrExternal = true;
  }
  else if (ExternalSymbolSDNode *S = dyn_cast<ExternalSymbolSDNode>(Callee)) {
    const char *Sym = S->getSymbol();

    if (!IsPIC) // static
      Callee = DAG.getTargetExternalSymbol(Sym,
                                           getPointerTy(DAG.getDataLayout()),
                                           Cpu0II::MO_NO_FLAG);
    else // PIC
      Callee = getAddrGlobal(S, Ty, DAG, Cpu0II::MO_GOT_CALL, Chain,
                             FuncInfo->callPtrInfo(Sym));

    GlobalOrExternal = true;
  }

  SmallVector<SDValue, 8> Ops(1, Chain);
  SDVTList NodeTys = DAG.getVTList(MVT::Other, MVT::Glue);

  getOpndList(Ops, RegsToPass, IsPICCall, GlobalOrExternal, InternalLinkage,
              CLI, Callee, Chain);

  //@TailCall 3 {
  if (IsTailCall)
    return DAG.getNode(Cpu0ISD::TailCall, DL, MVT::Other, Ops);
  //@TailCall 3 }

  Chain = DAG.getNode(Cpu0ISD::JmpLink, DL, NodeTys, Ops);
  SDValue InFlag = Chain.getValue(1);

  // Create the CALLSEQ_END node.
  Chain = DAG.getCALLSEQ_END(Chain, NextStackOffsetVal,
                             DAG.getIntPtrConstant(0, DL, true), InFlag, DL);
  InFlag = Chain.getValue(1);

  // Handle result values, copying them out of physregs into vregs that we
  // return.
  return LowerCallResult(Chain, InFlag, CallConv, IsVarArg,
                         Ins, DL, DAG, InVals, CLI.Callee.getNode(), CLI.RetTy);
}
```

```
/// LowerCallResult - Lower the result values of a call into the
/// appropriate copies out of appropriate physical registers.
SDValue
Cpu0TargetLowering::LowerCallResult(SDValue Chain, SDValue InFlag,
                                    CallingConv::ID CallConv, bool IsVarArg,
                                    const SmallVectorImpl<ISD::InputArg> &Ins,
                                    const SDLoc &DL, SelectionDAG &DAG,
                                    SmallVectorImpl<SDValue> &InVals,
                                    const SDNode *CallNode,
                                    const Type *RetTy) const {
  // Assign locations to each value returned by this call.
```

(continues on next page)

(continued from previous page)

```
  SmallVector<CCValAssign, 16> RVLocs;
  CCState CCInfo(CallConv, IsVarArg, DAG.getMachineFunction(),
                 RVLocs, *DAG.getContext());

  Cpu0CC Cpu0CCInfo(CallConv, ABI.IsO32(), CCInfo);

  Cpu0CCInfo.analyzeCallResult(Ins, Subtarget.abiUsesSoftFloat(),
                               CallNode, RetTy);

  // Copy all of the result registers out of their specified physreg.
  for (unsigned i = 0; i != RVLocs.size(); ++i) {
    SDValue Val = DAG.getCopyFromReg(Chain, DL, RVLocs[i].getLocReg(),
                                     RVLocs[i].getLocVT(), InFlag);
    Chain = Val.getValue(1);
    InFlag = Val.getValue(2);

    if (RVLocs[i].getValVT() != RVLocs[i].getLocVT())
      Val = DAG.getNode(ISD::BITCAST, DL, RVLocs[i].getValVT(), Val);

    InVals.push_back(Val);
  }

  return Chain;
}
```

```
bool
Cpu0TargetLowering::CanLowerReturn(CallingConv::ID CallConv,
                                   MachineFunction &MF, bool IsVarArg,
                                   const SmallVectorImpl<ISD::OutputArg> &Outs,
                                   LLVMContext &Context) const {
  SmallVector<CCValAssign, 16> RVLocs;
  CCState CCInfo(CallConv, IsVarArg, MF,
                 RVLocs, Context);
  return CCInfo.CheckReturn(Outs, RetCC_Cpu0);
}
```

```
Cpu0TargetLowering::Cpu0CC::SpecialCallingConvType
  Cpu0TargetLowering::getSpecialCallingConv(SDValue Callee) const {
  Cpu0CC::SpecialCallingConvType SpecialCallingConv =
    Cpu0CC::NoSpecialCallingConv;
  return SpecialCallingConv;
}
```

```
void Cpu0TargetLowering::Cpu0CC::
analyzeCallOperands(const SmallVectorImpl<ISD::OutputArg> &Args,
                    bool IsVarArg, bool IsSoftFloat, const SDNode *CallNode,
                    std::vector<ArgListEntry> &FuncArgs) {
//@analyzeCallOperands body {
  assert((CallConv != CallingConv::Fast || !IsVarArg) &&
         "CallingConv::Fast shouldn't be used for vararg functions.");
```

(continues on next page)

(continued from previous page)

```
  unsigned NumOpnds = Args.size();
  llvm::CCAssignFn *FixedFn = fixedArgFn();

  //@3 {
  for (unsigned I = 0; I != NumOpnds; ++I) {
  //@3 }
    MVT ArgVT = Args[I].VT;
    ISD::ArgFlagsTy ArgFlags = Args[I].Flags;
    bool R;

    if (ArgFlags.isByVal()) {
      handleByValArg(I, ArgVT, ArgVT, CCValAssign::Full, ArgFlags);
      continue;
    }

    {
      MVT RegVT = getRegVT(ArgVT, IsSoftFloat);
      R = FixedFn(I, ArgVT, RegVT, CCValAssign::Full, ArgFlags, CCInfo);
    }

    if (R) {
#ifndef NDEBUG
      dbgs() << "Call operand #" << I << " has unhandled type "
             << EVT(ArgVT).getEVTString();
#endif
      llvm_unreachable(nullptr);
    }
  }
}
```

Just like load incoming arguments from stack frame, we call CCInfo(CallConv,. . . , ArgLocs, . . .) to get outgoing arguments information before entering **"for loop"*. They're almost same in **"for loop"** with LowerFormalArguments(), except LowerCall() creates "store DAG vector" instead of "load DAG vector". After the **"for loop"**, it create **"ld $t9, %call16(_Z5sum_iiiiii)($gp)"** and jalr $t9 for calling subroutine (the $6 is $t9) in PIC mode.

Like loading incoming arguments, we need to implement storeRegToStackSlot() at early chapter.

9.3.1 Pseudo hook instruction ADJCALLSTACKDOWN and ADJCALLSTACKUP

DAG.getCALLSEQ_START() and DAG.getCALLSEQ_END() are set before and after the **"for loop"**, respectively, they insert CALLSEQ_START, CALLSEQ_END, and translate them into pseudo machine instructions !ADJCALLSTACKDOWN, !ADJCALLSTACKUP later according Cpu0InstrInfo.td definition as follows.

lbdex/chapters/Chapter9_2/Cpu0InstrInfo.td

```
def SDT_Cpu0CallSeqStart : SDCallSeqStart<[SDTCisVT<0, i32>]>;
def SDT_Cpu0CallSeqEnd   : SDCallSeqEnd<[SDTCisVT<0, i32>, SDTCisVT<1, i32>]>;
```

```
// These are target-independent nodes, but have target-specific formats.
def callseq_start : SDNode<"ISD::CALLSEQ_START", SDT_Cpu0CallSeqStart,
                           [SDNPHasChain, SDNPOutGlue]>;
def callseq_end   : SDNode<"ISD::CALLSEQ_END", SDT_Cpu0CallSeqEnd,
                           [SDNPHasChain, SDNPOptInGlue, SDNPOutGlue]>;
```

```
//===----------------------------------------------------------------------===//
// Pseudo instructions
//===----------------------------------------------------------------------===//

let Predicates = [Ch9_2] in {
// As stack alignment is always done with addiu, we need a 16-bit immediate
let Defs = [SP], Uses = [SP] in {
def ADJCALLSTACKDOWN : Cpu0Pseudo<(outs), (ins uimm16:$amt1, uimm16:$amt2),
                                  "!ADJCALLSTACKDOWN $amt1",
                                  [(callseq_start timm:$amt1, timm:$amt2)]>;
def ADJCALLSTACKUP   : Cpu0Pseudo<(outs), (ins uimm16:$amt1, uimm16:$amt2),
                                  "!ADJCALLSTACKUP $amt1",
                                  [(callseq_end timm:$amt1, timm:$amt2)]>;
}

//@def CPRESTORE {
// When handling PIC code the assembler needs .cpload and .cprestore
// directives. If the real instructions corresponding these directives
// are used, we have the same behavior, but get also a bunch of warnings
// from the assembler.
let hasSideEffects = 0 in
def CPRESTORE : Cpu0Pseudo<(outs), (ins i32imm:$loc, CPURegs:$gp),
                           ".cprestore\t$loc", []>;
} // let Predicates = [Ch9_2]
```

With below definition, eliminateCallFramePseudoInstr() will be called when llvm meets pseudo instructions ADJCALLSTACKDOWN and ADJCALLSTACKUP. It justs discard these 2 pseudo instructions, and llvm will add offset to stack.

lbdex/chapters/Chapter9_2/Cpu0InstrInfo.cpp

```
Cpu0InstrInfo::Cpu0InstrInfo(const Cpu0Subtarget &STI)
    :
```

```
    Cpu0GenInstrInfo(Cpu0::ADJCALLSTACKDOWN, Cpu0::ADJCALLSTACKUP),
```

lbdex/chapters/Chapter9_2/Cpu0FrameLowering.h

```
  MachineBasicBlock::iterator
  eliminateCallFramePseudoInstr(MachineFunction &MF,
                                MachineBasicBlock &MBB,
                                MachineBasicBlock::iterator I) const override;
```

lbdex/chapters/Chapter9_2/Cpu0FrameLowering.cpp

```
// Eliminate ADJCALLSTACKDOWN, ADJCALLSTACKUP pseudo instructions
MachineBasicBlock::iterator Cpu0FrameLowering::
eliminateCallFramePseudoInstr(MachineFunction &MF, MachineBasicBlock &MBB,
                              MachineBasicBlock::iterator I) const {
```

```
  return MBB.erase(I);
}
```

9.3.2 Read Lowercall() with Graphivz's help

The whole DAGs created for outgoing arguments as Fig. 9.5 below for ch9_outgoing.cpp with cpu032I. LowerCall() (excluding calling LowerCallResult()) will generate the DAG nodes as Fig. 9.6 below for ch9_outgoing.cpp with cpu032I. The corresponding code of DAGs Store and TargetGlobalAddress are listed in the figures , user can match the other DAGs to function LowerCall() easily. Through Graphivz tool with llc option -view-dag-combine1-dags, you can design a small input C or llvm IR source code and then check the DAGs to understand the code in LowerCall() and LowerFormalArguments(). At the sub-sections "variable arguments" and "dynamic stack allocation support" in the later section of this chapter, you can design the input example with this features and check the DAGs with these two functions again to make sure you know the code in these two function. About Graphivz, please refer to section "Display llvm IR nodes with Graphviz" of chapter 4, Arithmetic and logic instructions. The DAGs diagram can be got by llc option as follows,

lbdex/input/ch9_outgoing.cpp

```
extern int sum_i(int x1);

int call_sum_i() {
  return sum_i(1);
}
```

```
JonathantekiiMac:input Jonathan$ clang -O3 -target mips-unknown-linux-gnu -c
ch9_outgoing.cpp -emit-llvm -o ch9_outgoing.bc
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-dis ch9_outgoing.bc -o -
...
define i32 @_Z10call_sum_iv() #0 {
  %1 = tail call i32 @_Z5sum_ii(i32 1)
```

(continues on next page)

(continued from previous page)

```
  ret i32 %1
}
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032I -view-dag-combine1-dags -relocation-
model=static -filetype=asm ch9_outgoing.bc -o -
      .text
      .section .mdebug.abiS32
      .previous
      .file   "ch9_outgoing.bc"
Writing '/var/folders/rf/8bgdgt9d6vgf5sn8h8_zycd00000gn/T/dag._Z10call_sum_iv-
0dfaf1.dot'...  done.
Running 'Graphviz' program...
```

Mentioned in last section, option `llc -cpu0-s32-calls=true` uses S32 calling convention which passes all arguements at registers while option `llc -cpu0-s32-calls=false` uses O32 pass first two arguments at registers and other arguments at stack. The result as follows,

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032I -cpu0-s32-calls=true
-relocation-model=pic -filetype=asm ch9_1.bc -o -
        .text
        .section .mdebug.abiS32
        .previous
        .file "ch9_1.bc"
        .globl        _Z5sum_iiiiiii
        .align        2
        .type _Z5sum_iiiiiii,@function
        .ent  _Z5sum_iiiiiii          # @_Z5sum_iiiiiii
_Z5sum_iiiiiii:
        .frame        $fp,32,$lr
        .mask         0x00000000,0
        .set  noreorder
        .cpload       $t9
        .set  nomacro
# BB#0:
        addiu $sp, $sp, -32
        ld    $2, 52($sp)
        ld    $3, 48($sp)
        ld    $4, 44($sp)
        ld    $5, 40($sp)
        ld    $t9, 36($sp)
        ld    $7, 32($sp)
        st    $7, 28($sp)
        st    $t9, 24($sp)
        st    $5, 20($sp)
        st    $4, 16($sp)
        st    $3, 12($sp)
        lui   $3, %got_hi(gI)
        addu  $3, $3, $gp
        st    $2, 8($sp)
        ld    $3, %got_lo(gI)($3)
        ld    $3, 0($3)
```

(continues on next page)

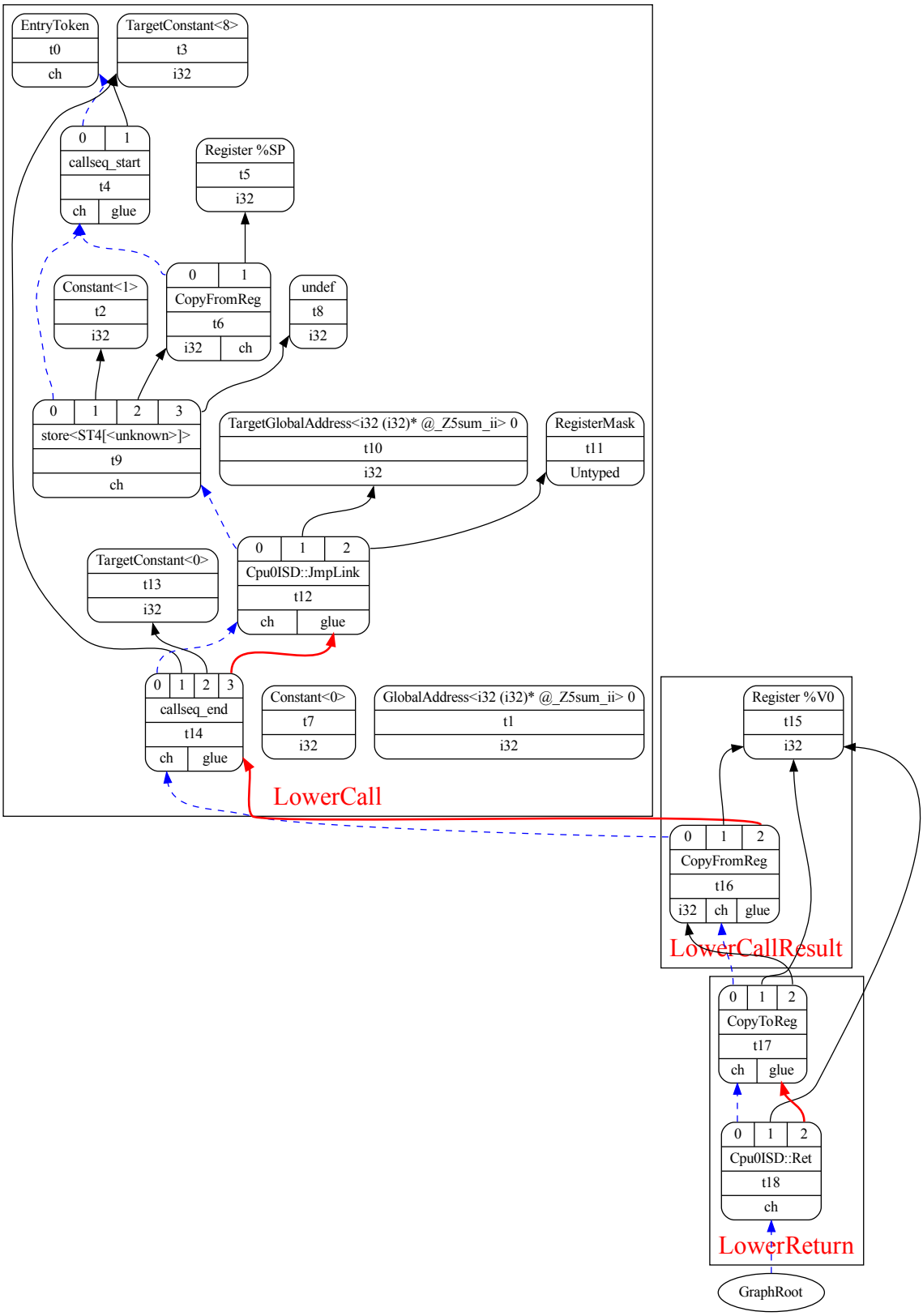


Fig. 9.5: Outgoing arguments DAG (A) created for ch9_outgoing.cpp with -cpu0-s32-calls=true

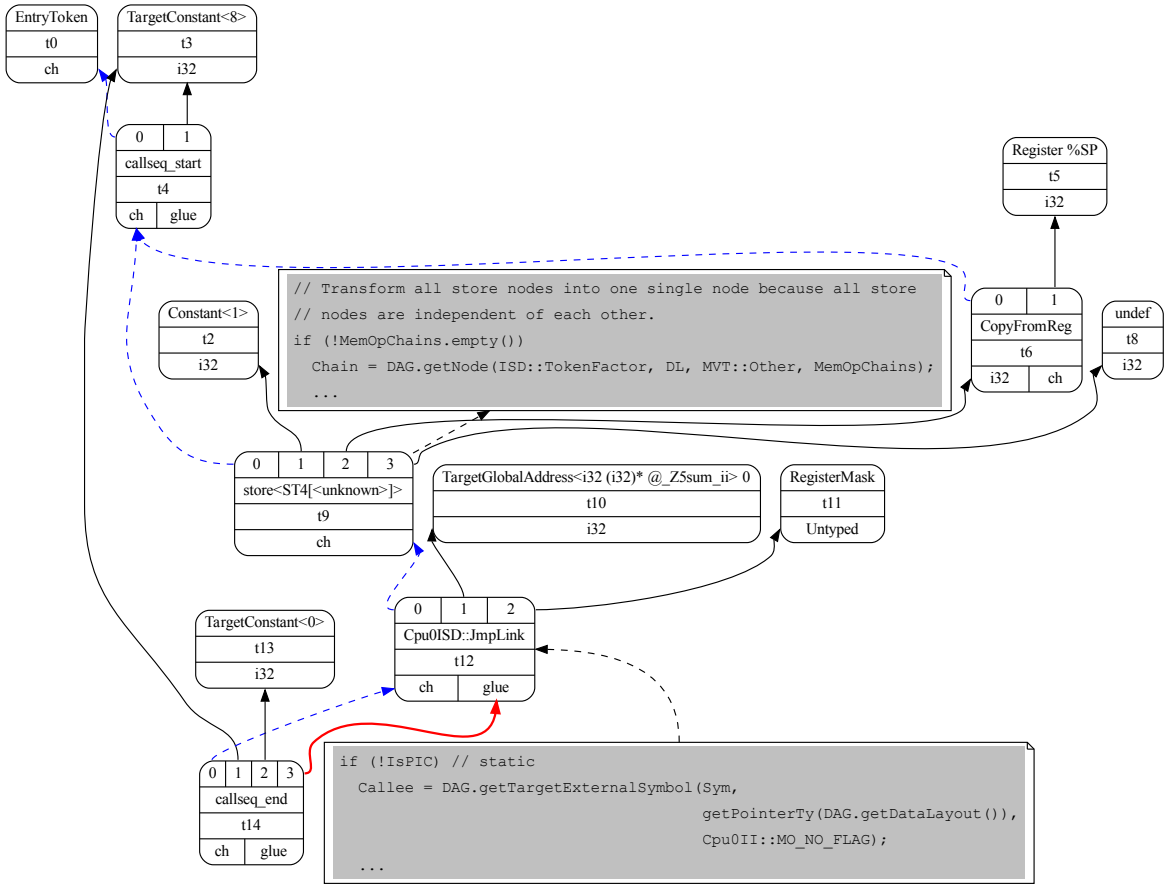


Fig. 9.6: Outgoing arguments DAG (B) created by LowerCall() for ch9_outgoing.cpp with -cpu0-s32-calls=true

(continued from previous page)

```
        ld    $4, 28($sp)
        addu  $3, $3, $4
        ld    $4, 24($sp)
        addu  $3, $3, $4
        ld    $4, 20($sp)
        addu  $3, $3, $4
        ld    $4, 16($sp)
        addu  $3, $3, $4
        ld    $4, 12($sp)
        addu  $3, $3, $4
        addu  $2, $3, $2
        st    $2, 4($sp)
        addiu $sp, $sp, 32
        ret   $lr
        nop
        .set  macro
        .set  reorder
        .end  _Z5sum_iiiiiii
$tmp0:
        .size _Z5sum_iiiiiii, ($tmp0)-_Z5sum_iiiiiii

        .globl        main
        .align        2
        .type main,@function
        .ent  main                    # @main
main:
        .frame        $fp,40,$lr
        .mask         0x00004000,-4
        .set  noreorder
        .cpload       $t9
        .set  nomacro
# BB#0:
        addiu $sp, $sp, -40
        st    $lr, 36($sp)            # 4-byte Folded Spill
        addiu $2, $zero, 0
        st    $2, 32($sp)
        addiu $2, $zero, 6
        st    $2, 20($sp)
        addiu $2, $zero, 5
        st    $2, 16($sp)
        addiu $2, $zero, 4
        st    $2, 12($sp)
        addiu $2, $zero, 3
        st    $2, 8($sp)
        addiu $2, $zero, 2
        st    $2, 4($sp)
        addiu $2, $zero, 1
        st    $2, 0($sp)
        ld    $t9, %call16(_Z5sum_iiiiiii)($gp)
        jalr  $t9
        nop
        st    $2, 28($sp)
```

(continues on next page)

(continued from previous page)

```
        ld    $lr, 36($sp)            # 4-byte Folded Reload
        addiu $sp, $sp, 40
        ret   $lr
        nop
        .set  macro
        .set  reorder
        .end  main
$tmp1:
        .size main, ($tmp1)-main

        .type gI,@object              # @gI
        .data
        .globl        gI
        .align        2
gI:
        .4byte        100                     # 0x64
        .size gI, 4

118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032II -cpu0-s32-calls=false
-relocation-model=pic -filetype=asm ch9_1.bc -o -
  ...
        .globl        main
        .align        2
        .type main,@function
        .ent  main                    # @main
main:
        .frame        $fp,40,$lr
        .mask         0x00004000,-4
        .set  noreorder
        .cpload       $t9
        .set  nomacro
# BB#0:
        addiu $sp, $sp, -40
        st    $lr, 36($sp)            # 4-byte Folded Spill
        addiu $2, $zero, 0
        st    $2, 32($sp)
        addiu $2, $zero, 6
        st    $2, 20($sp)
        addiu $2, $zero, 5
        st    $2, 16($sp)
        addiu $2, $zero, 4
        st    $2, 12($sp)
        addiu $2, $zero, 3
        st    $2, 8($sp)
        ld    $t9, %call16(_Z5sum_iiiiiii)($gp)
        addiu $4, $zero, 1
        addiu $5, $zero, 2
        jalr  $t9
        nop
        st    $2, 28($sp)
        ld    $lr, 36($sp)            # 4-byte Folded Reload
```

(continues on next page)

(continued from previous page)

```
        addiu $sp, $sp, 40
        ret   $lr
        nop
        .set  macro
        .set  reorder
        .end  main
```

9.3.3 Long and short string initialization

The last section mentioned the "JSUB texternalsym" pattern. Run Chapter9_2 with ch9_1_2.cpp to get the result as below. For long string, llvm call memcpy() to initialize string (char str[81] = "Hello world" in this case). For short string, the "call memcpy" is translated into "store with contant" in stages of optimization.

lbdex/input/ch9_1_2.cpp

```
int main()
{
  char str[81] = "Hello world";
  char s[6] = "Hello";

  return 0;
}
```

```
JonathantekiiMac:input Jonathan$ llvm-dis ch9_1_2.bc -o -
; ModuleID = 'ch9_1_2.bc'
...
@_ZZ4mainE3str = private unnamed_addr constant [81 x i8] c"Hello world\00\00\00\
00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00
\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\00\0
0\00\00\00\00\00\00\00\00\00\00\00\00", align 1
@_ZZ4mainE1s = private unnamed_addr constant [6 x i8] c"Hello\00", align 1

; Function Attrs: nounwind
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %str = alloca [81 x i8], align 1
  store i32 0, i32* %retval
  %0 = bitcast [81 x i8]* %str to i8*
  call void @llvm.memcpy.p0i8.p0i8.i32(i8* %0, i8* getelementptr inbounds
  ([81 x i8]* @_ZZ4mainE3str, i32 0, i32 0), i32 81, i32 1, i1 false)
  %1 = bitcast [6 x i8]* %s to i8*
  call void @llvm.memcpy.p0i8.p0i8.i32(i8* %1, i8* getelementptr inbounds
  ([6 x i8]* @_ZZ4mainE1s, i32 0, i32 0), i32 6, i32 1, i1 false)

  ret i32 0
}

JonathantekiiMac:input Jonathan$ clang -target mips-unknown-linux-gnu -c
```

(continues on next page)

(continued from previous page)

```
ch9_1_2.cpp -emit-llvm -o ch9_1_2.bc
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build
/bin/llc -march=cpu0 -mcpu=cpu032II -cpu0-s32-calls=true
-relocation-model=static -filetype=asm ch9_1_2.bc -o -
  .section .mdebug.abi32
  ...
        lui   $2, %hi($_ZZ4mainE3str)
        ori   $2, $2, %lo($_ZZ4mainE3str)
        st    $2, 4($sp)
        addiu $2, $sp, 24
        st    $2, 0($sp)
        jsub  memcpy
        nop
        lui   $2, %hi($_ZZ4mainE1s)
        ori   $2, $2, %lo($_ZZ4mainE1s)
        lbu   $3, 4($2)
        shl   $3, $3, 8
        lbu   $4, 5($2)
        or    $3, $3, $4
        sh    $3, 20($sp)
        lbu   $3, 2($2)
        shl   $3, $3, 8
        lbu   $4, 3($2)
        or    $3, $3, $4
        lbu   $4, 1($2)
        lbu   $2, 0($2)
        shl   $2, $2, 8
        or    $2, $2, $4
        shl   $2, $2, 16
        or    $2, $2, $3
        st    $2, 16($sp)
  ...
      .type   $_ZZ4mainE3str,@object  # @_ZZ4mainE3str
      .section        .rodata,"a",@progbits
$_ZZ4mainE3str:
        .asciz        "Hello world\000\000\000\000\000\000\000\000\000\000\000\000\000\
↪000
  \000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000
  \000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000
  \000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000"
        .size $_ZZ4mainE3str, 81

        .type $_ZZ4mainE1s,@object    # @_ZZ4mainE1s
        .section      .rodata.str1.1,"aMS",@progbits,1
$_ZZ4mainE1s:
        .asciz        "Hello"
        .size $_ZZ4mainE1s, 6
```

The “call memcpy” for short string is optimized by llvm before “DAG->DAG Pattern Instruction Selection” stage and translates it into “store with contant” as follows,

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build
```

(continues on next page)

(continued from previous page)

```
/bin/llc -march=cpu0 -mcpu=cpu032II -cpu0-s32-calls=true
-relocation-model=static -filetype=asm ch9_1_2.bc -debug -o -

Initial selection DAG: BB#0 'main:entry'
SelectionDAG has 35 nodes:
  ...
        0x7fd909030810: <multiple use>
        0x7fd909030c10: i32 = Constant<1214606444>  // 1214606444=0x48656c6c="Hell"

        0x7fd909030910: <multiple use>
        0x7fd90902d810: <multiple use>
      0x7fd909030d10: ch = store 0x7fd909030810, 0x7fd909030c10, 0x7fd909030910,
      0x7fd90902d810<ST4[%1]>

        0x7fd909030810: <multiple use>
        0x7fd909030e10: i16 = Constant<28416>      // 28416=0x6f00="o\0"

        ...

        0x7fd90902d810: <multiple use>
      0x7fd909031210: ch = store 0x7fd909030810, 0x7fd909030e10, 0x7fd909031010,
      0x7fd90902d810<ST2[%1+4](align=4)>
  ...
```

The incoming arguments is the formal arguments defined in compiler and program language books. The outgoing arguments is the actual arguments. Summary as Table: Callee incoming arguments and caller outgoing arguments.

Table 9.1: Callee incoming arguments and caller outgoing arguments

Description	Callee	Caller
Charged Function	LowerFormalArguments()	LowerCall()
Charged Function Created	Create load vectors for incoming arguments	Create store vectors for outgoing arguments

9.4 Structure type support

9.4.1 Ordinary struct type

The following code in Chapter9_1/ and Chapter3_4/ support the ordinary structure type in function call.

lbdex/chapters/Chapter9_1/Cpu0ISelLowering.cpp

```
/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool IsVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
```

(continues on next page)

(continued from previous page)

```
                                          const SDLoc &DL, SelectionDAG &DAG,
                                          SmallVectorImpl<SDValue> &InVals)
                                           const {
```

```
  for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
    // The cpu0 ABIs for returning structs by value requires that we copy
    // the sret argument into $v0 for the return. Save the argument into
    // a virtual register so that we can access it from the return points.
    if (Ins[i].Flags.isSRet()) {
      unsigned Reg = Cpu0FI->getSRetReturnReg();
      if (!Reg) {
        Reg = MF.getRegInfo().createVirtualRegister(
            getRegClassFor(MVT::i32));
        Cpu0FI->setSRetReturnReg(Reg);
      }
      SDValue Copy = DAG.getCopyToReg(DAG.getEntryNode(), DL, Reg, InVals[i]);
      Chain = DAG.getNode(ISD::TokenFactor, DL, MVT::Other, Copy, Chain);
      break;
    }
  }
```

```
}
```

```
SDValue
Cpu0TargetLowering::LowerReturn(SDValue Chain,
                                CallingConv::ID CallConv, bool IsVarArg,
                                const SmallVectorImpl<ISD::OutputArg> &Outs,
                                const SmallVectorImpl<SDValue> &OutVals,
                                const SDLoc &DL, SelectionDAG &DAG) const {
```

```
  // The cpu0 ABIs for returning structs by value requires that we copy
  // the sret argument into $v0 for the return. We saved the argument into
  // a virtual register in the entry block, so now we copy the value out
  // and into $v0.
  if (MF.getFunction().hasStructRetAttr()) {
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
    unsigned Reg = Cpu0FI->getSRetReturnReg();

    if (!Reg)
      llvm_unreachable("sret virtual register not created in the entry block");
    SDValue Val =
        DAG.getCopyFromReg(Chain, DL, Reg, getPointerTy(DAG.getDataLayout()));
    unsigned V0 = Cpu0::V0;

    Chain = DAG.getCopyToReg(Chain, DL, V0, Val, Flag);
    Flag = Chain.getValue(1);
    RetOps.push_back(DAG.getRegister(V0, getPointerTy(DAG.getDataLayout())));
  }
```

```
}
```

In addition to above code, we defined the calling convention in early chapter as follows,

lbdex/chapters/Chapter3_4/Cpu0CallingConv.td

```
def RetCC_Cpu0EABI : CallingConv<[
  // i32 are returned in registers V0, V1, A0, A1
  CCIfType<[i32], CCAssignToReg<[V0, V1, A0, A1]>>
]>;
```

It meaning for the return value, we keep it in registers V0, V1, A0, A1 if the size of return value doesn't over 4 registers; If it overs 4 registers, cpu0 will save them in memory with a pointer of memory in register. For explanation, let's run Chapter9_2/ with ch9_1_struct.cpp and explain with this example.

lbdex/input/ch9_1_struct.cpp

```
extern "C" int printf(const char *format, ...);

struct Date
{
  int year;
  int month;
  int day;
  int hour;
  int minute;
  int second;
};
static Date gDate = {2012, 10, 12, 1, 2, 3};

struct Time
{
  int hour;
  int minute;
  int second;
};
static Time gTime = {2, 20, 30};

static Date getDate()
{
  return gDate;
}

static Date copyDate(Date date)
{
  return date;
}

static Date copyDate(Date* date)
{
  return *date;
}
```

(continues on next page)

(continued from previous page)

```
static Time copyTime(Time time)
{
  return time;
}

static Time copyTime(Time* time)
{
  return *time;
}

int test_func_arg_struct()
{
  Time time1 = {1, 10, 12};
  Date date1 = getDate();
  Date date2 = copyDate(date1);
  Date date3 = copyDate(&date1);
  Time time2 = copyTime(time1);
  Time time3 = copyTime(&time1);
  if (!(date1.year == 2012 && date1.month == 10 && date1.day == 12 && date1.hour
      == 1 && date1.minute == 2 && date1.second == 3))
    return 1;
  if (!(date2.year == 2012 && date2.month == 10 && date2.day == 12 && date2.hour
      == 1 && date2.minute == 2 && date2.second == 3))
    return 1;
  if (!(time2.hour == 1 && time2.minute == 10 && time2.second == 12))
    return 1;
  if (!(time3.hour == 1 && time3.minute == 10 && time3.second == 12))
    return 1;

#ifdef PRINT_TEST
  printf("date1 = %d %d %d %d %d %d", date1.year, date1.month, date1.day,
    date1.hour, date1.minute, date1.second); // date1 = 2012 10 12 1 2 3
  if (date1.year == 2012 && date1.month == 10 && date1.day == 12 && date1.hour
      == 1 && date1.minute == 2 && date1.second == 3)
    printf(", PASS\n");
  else
    printf(", FAIL\n");
  printf("date2 = %d %d %d %d %d %d", date2.year, date2.month, date2.day,
    date2.hour, date2.minute, date2.second); // date2 = 2012 10 12 1 2 3
  if (date2.year == 2012 && date2.month == 10 && date2.day == 12 && date2.hour
      == 1 && date2.minute == 2 && date2.second == 3)
    printf(", PASS\n");
  else
    printf(", FAIL\n");
  // time2 = 1 10 12
  printf("time2 = %d %d %d", time2.hour, time2.minute, time2.second);
  if (time2.hour == 1 && time2.minute == 10 && time2.second == 12)
    printf(", PASS\n");
  else
    printf(", FAIL\n");
  // time3 = 1 10 12
  printf("time3 = %d %d %d", time3.hour, time3.minute, time3.second);
```

(continues on next page)

(continued from previous page)

```
  if (time3.hour == 1 && time3.minute == 10 && time3.second == 12)
    printf(", PASS\n");
  else
    printf(", FAIL\n");
#endif

  return 0;
}
```

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -filetype=asm
ch9_1_struct.bc -o -
  .section .mdebug.abi32
  .previous
  .file "ch9_1_struct.bc"
  .text
  .globl  _Z7getDatev
  .align  2
  .type _Z7getDatev,@function
  .ent  _Z7getDatev             # @_Z7getDatev
_Z7getDatev:
  .cfi_startproc
  .frame  $sp,0,$lr
  .mask   0x00000000,0
  .set  noreorder
  .cpload $t9
  .set  nomacro
# BB#0:
        lui   $2, %got_hi(gDate)
        addu  $2, $2, $gp
        ld    $3, %got_lo(gDate)($2)
        ld    $2, 0($sp)
  ld  $4, 20($3)        // save gDate contents to 212..192($sp)
  st  $4, 20($2)
  ld  $4, 16($3)
  st  $4, 16($2)
  ld  $4, 12($3)
  st  $4, 12($2)
  ld  $4, 8($3)
  st  $4, 8($2)
  ld  $4, 4($3)
  st  $4, 4($2)
  ld  $3, 0($3)
  st  $3, 0($2)
  ret $lr
  nop
  .set  macro
  .set  reorder
  .end  _Z7getDatev
$tmp0:
  .size _Z7getDatev, ($tmp0)-_Z7getDatev
  .cfi_endproc
```

(continues on next page)

(continued from previous page)

```
  ...
  .globl  _Z20test_func_arg_structv
  .align  2
  .type _Z20test_func_arg_structv,@function
  .ent  _Z20test_func_arg_structv                    # @main
_Z20test_func_arg_structv:
  .cfi_startproc
  .frame  $sp,248,$lr
  .mask   0x00004180,-4
  .set  noreorder
  .cpload $t9
  .set  nomacro
  # BB#0:
        addiu $sp, $sp, -200
        st    $lr, 196($sp)           # 4-byte Folded Spill
        st    $8, 192($sp)            # 4-byte Folded Spill
        ld    $2, %got($_ZZ20test_func_arg_structvE5time1)($gp)
        ori   $2, $2, %lo($_ZZ20test_func_arg_structvE5time1)
        ld    $3, 8($2)
        st    $3, 184($sp)
        ld    $3, 4($2)
        st    $3, 180($sp)
        ld    $2, 0($2)
        st    $2, 176($sp)
        addiu $8, $sp, 152
        st    $8, 0($sp)
        ld    $t9, %call16(_Z7getDatev)($gp) // copy gDate contents to date1, 176..152(
↪$sp)
        jalr  $t9
        nop
        ld    $gp, 176($sp)
        ld    $2, 172($sp)
        st    $2, 124($sp)
        ld    $2, 168($sp)
        st    $2, 120($sp)
        ld    $2, 164($sp)
        st    $2, 116($sp)
        ld    $2, 160($sp)
        st    $2, 112($sp)
        ld    $2, 156($sp)
        st    $2, 108($sp)
        ld    $2, 152($sp)
        st    $2, 104($sp)
  ...
```

The ch9_1_constructor.cpp includes C++ class "Date" implementation. It can be translated into cpu0 backend too since the frontend (clang in this example) translate them into C language form. If you mark the "if hasStructRetAttr()" part from both of above functions, the output of cpu0 code for ch9_1_struct.cpp will use $3 instead of $2 as return register as follows,

```
        .text
        .section .mdebug.abiS32
```

(continues on next page)

(continued from previous page)

```
        .previous
        .file "ch9_1_struct.bc"
        .globl        _Z7getDatev
        .align        2
        .type _Z7getDatev,@function
        .ent  _Z7getDatev             # @_Z7getDatev
_Z7getDatev:
        .frame        $fp,0,$lr
        .mask         0x00000000,0
        .set  noreorder
        .cpload       $t9
        .set  nomacro
# BB#0:
        lui   $2, %got_hi(gDate)
        addu  $2, $2, $gp
        ld    $2, %got_lo(gDate)($2)
        ld    $3, 0($sp)
        ld    $4, 20($2)
        st    $4, 20($3)
        ld    $4, 16($2)
        st    $4, 16($3)
        ld    $4, 12($2)
        st    $4, 12($3)
        ld    $4, 8($2)
        st    $4, 8($3)
        ld    $4, 4($2)
        st    $4, 4($3)
        ld    $2, 0($2)
        st    $2, 0($3)
        ret   $lr
        nop
  ...
```

Mips ABI asks "return struct varaible address" to be set at $2.

9.4.2 byval struct type

The following code in Chapter9_1/ and Chapter9_2/ support the byval structure type in function call.

lbdex/chapters/Chapter9_1/Cpu0ISelLowering.cpp

```
void Cpu0TargetLowering::
copyByValRegs(SDValue Chain, const SDLoc &DL, std::vector<SDValue> &OutChains,
              SelectionDAG &DAG, const ISD::ArgFlagsTy &Flags,
              SmallVectorImpl<SDValue> &InVals, const Argument *FuncArg,
              const Cpu0CC &CC, const ByValArgInfo &ByVal) const {
  MachineFunction &MF = DAG.getMachineFunction();
  MachineFrameInfo &MFI = MF.getFrameInfo();
  unsigned RegAreaSize = ByVal.NumRegs * CC.regSize();
  unsigned FrameObjSize = std::max(Flags.getByValSize(), RegAreaSize);
```

(continues on next page)

(continued from previous page)

```
  int FrameObjOffset;

  const ArrayRef<MCPhysReg> ByValArgRegs = CC.intArgRegs();

  if (RegAreaSize)
    FrameObjOffset = (int)CC.reservedArgArea() -
      (int)((CC.numIntArgRegs() - ByVal.FirstIdx) * CC.regSize());
  else
    FrameObjOffset = ByVal.Address;

  // Create frame object.
  EVT PtrTy = getPointerTy(DAG.getDataLayout());
  int FI = MFI.CreateFixedObject(FrameObjSize, FrameObjOffset, true);
  SDValue FIN = DAG.getFrameIndex(FI, PtrTy);
  InVals.push_back(FIN);

  if (!ByVal.NumRegs)
    return;

  // Copy arg registers.
  MVT RegTy = MVT::getIntegerVT(CC.regSize() * 8);
  const TargetRegisterClass *RC = getRegClassFor(RegTy);

  for (unsigned I = 0; I < ByVal.NumRegs; ++I) {
    unsigned ArgReg = ByValArgRegs[ByVal.FirstIdx + I];
    unsigned VReg = addLiveIn(MF, ArgReg, RC);
    unsigned Offset = I * CC.regSize();
    SDValue StorePtr = DAG.getNode(ISD::ADD, DL, PtrTy, FIN,
                                   DAG.getConstant(Offset, DL, PtrTy));
    SDValue Store = DAG.getStore(Chain, DL, DAG.getRegister(VReg, RegTy),
                                 StorePtr, MachinePointerInfo(FuncArg, Offset));
    OutChains.push_back(Store);
  }
}
```

```
/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool IsVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         const SDLoc &DL, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                          const {
```

```
  for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
```

```
    if (Flags.isByVal()) {
      assert(Flags.getByValSize() &&
             "ByVal args of size 0 should have been ignored by front-end.");
```

(continues on next page)

(continued from previous page)

```
      assert(ByValArg != Cpu0CCInfo.byval_end());
      copyByValRegs(Chain, DL, OutChains, DAG, Flags, InVals, &*FuncArg,
                    Cpu0CCInfo, *ByValArg);
      ++ByValArg;
      continue;
    }
```

```
    ...
. }
```

```
  for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
    // The cpu0 ABIs for returning structs by value requires that we copy
    // the sret argument into $v0 for the return. Save the argument into
    // a virtual register so that we can access it from the return points.
    if (Ins[i].Flags.isSRet()) {
      unsigned Reg = Cpu0FI->getSRetReturnReg();
      if (!Reg) {
        Reg = MF.getRegInfo().createVirtualRegister(
            getRegClassFor(MVT::i32));
        Cpu0FI->setSRetReturnReg(Reg);
      }
      SDValue Copy = DAG.getCopyToReg(DAG.getEntryNode(), DL, Reg, InVals[i]);
      Chain = DAG.getNode(ISD::TokenFactor, DL, MVT::Other, Copy, Chain);
      break;
    }
  }
```

```
  ...
}
```

lbdex/chapters/Chapter9_2/Cpu0ISelLowering.cpp

```
// Copy byVal arg to registers and stack.
void Cpu0TargetLowering::
passByValArg(SDValue Chain, const SDLoc &DL,
             std::deque< std::pair<unsigned, SDValue> > &RegsToPass,
             SmallVectorImpl<SDValue> &MemOpChains, SDValue StackPtr,
             MachineFrameInfo &MFI, SelectionDAG &DAG, SDValue Arg,
             const Cpu0CC &CC, const ByValArgInfo &ByVal,
             const ISD::ArgFlagsTy &Flags, bool isLittle) const {
  unsigned ByValSizeInBytes = Flags.getByValSize();
  unsigned OffsetInBytes = 0; // From beginning of struct
  unsigned RegSizeInBytes = CC.regSize();
  unsigned Alignment = std::min((unsigned)Flags.getNonZeroByValAlign().value(),␣
↪RegSizeInBytes);
  EVT PtrTy = getPointerTy(DAG.getDataLayout()),
      RegTy = MVT::getIntegerVT(RegSizeInBytes * 8);

  if (ByVal.NumRegs) {
    const ArrayRef<MCPhysReg> ArgRegs = CC.intArgRegs();
```

(continues on next page)

(continued from previous page)

```
  bool LeftoverBytes = (ByVal.NumRegs * RegSizeInBytes > ByValSizeInBytes);
  unsigned I = 0;

  // Copy words to registers.
  for (; I < ByVal.NumRegs - LeftoverBytes;
       ++I, OffsetInBytes += RegSizeInBytes) {
    SDValue LoadPtr = DAG.getNode(ISD::ADD, DL, PtrTy, Arg,
                                  DAG.getConstant(OffsetInBytes, DL, PtrTy));
    SDValue LoadVal = DAG.getLoad(RegTy, DL, Chain, LoadPtr,
                                  MachinePointerInfo());
    MemOpChains.push_back(LoadVal.getValue(1));
    unsigned ArgReg = ArgRegs[ByVal.FirstIdx + I];
    RegsToPass.push_back(std::make_pair(ArgReg, LoadVal));
  }

  // Return if the struct has been fully copied.
  if (ByValSizeInBytes == OffsetInBytes)
    return;

  // Copy the remainder of the byval argument with sub-word loads and shifts.
  if (LeftoverBytes) {
    assert((ByValSizeInBytes > OffsetInBytes) &&
           (ByValSizeInBytes < OffsetInBytes + RegSizeInBytes) &&
           "Size of the remainder should be smaller than RegSizeInBytes.");
    SDValue Val;

    for (unsigned LoadSizeInBytes = RegSizeInBytes / 2, TotalBytesLoaded = 0;
         OffsetInBytes < ByValSizeInBytes; LoadSizeInBytes /= 2) {
      unsigned RemainingSizeInBytes = ByValSizeInBytes - OffsetInBytes;

      if (RemainingSizeInBytes < LoadSizeInBytes)
        continue;

      // Load subword.
      SDValue LoadPtr = DAG.getNode(ISD::ADD, DL, PtrTy, Arg,
                                    DAG.getConstant(OffsetInBytes, DL, PtrTy));
      SDValue LoadVal = DAG.getExtLoad(
          ISD::ZEXTLOAD, DL, RegTy, Chain, LoadPtr, MachinePointerInfo(),
          MVT::getIntegerVT(LoadSizeInBytes * 8), Alignment);
      MemOpChains.push_back(LoadVal.getValue(1));

      // Shift the loaded value.
      unsigned Shamt;

      if (isLittle)
        Shamt = TotalBytesLoaded * 8;
      else
        Shamt = (RegSizeInBytes - (TotalBytesLoaded + LoadSizeInBytes)) * 8;

      SDValue Shift = DAG.getNode(ISD::SHL, DL, RegTy, LoadVal,
                                  DAG.getConstant(Shamt, DL, MVT::i32));
```

(continues on next page)

(continued from previous page)

```
        if (Val.getNode())
          Val = DAG.getNode(ISD::OR, DL, RegTy, Val, Shift);
        else
          Val = Shift;

        OffsetInBytes += LoadSizeInBytes;
        TotalBytesLoaded += LoadSizeInBytes;
        Alignment = std::min(Alignment, LoadSizeInBytes);
      }

      unsigned ArgReg = ArgRegs[ByVal.FirstIdx + I];
      RegsToPass.push_back(std::make_pair(ArgReg, Val));
      return;
    }
  }

  // Copy remainder of byval arg to it with memcpy.
  unsigned MemCpySize = ByValSizeInBytes - OffsetInBytes;
  SDValue Src = DAG.getNode(ISD::ADD, DL, PtrTy, Arg,
                            DAG.getConstant(OffsetInBytes, DL, PtrTy));
  SDValue Dst = DAG.getNode(ISD::ADD, DL, PtrTy, StackPtr,
                            DAG.getIntPtrConstant(ByVal.Address, DL));
  Chain = DAG.getMemcpy(Chain, DL, Dst, Src,
                        DAG.getConstant(MemCpySize, DL, PtrTy),
                        Align(Alignment), /*isVolatile=*/false, /*AlwaysInline=*/false,
                        /*isTailCall=*/false,
                        MachinePointerInfo(), MachinePointerInfo());
  MemOpChains.push_back(Chain);
}
```

```
/// LowerCall - functions arguments are copied from virtual regs to
/// (physical regs)/(stack frame), CALLSEQ_START and CALLSEQ_END are emitted.
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                              SmallVectorImpl<SDValue> &InVals) const {
```

```
  // Walk the register/memloc assignments, inserting copies/loads.
  for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
```

```
    if (Flags.isByVal()) {
      assert(Flags.getByValSize() &&
             "ByVal args of size 0 should have been ignored by front-end.");
      assert(ByValArg != Cpu0CCInfo.byval_end());
      assert(!IsTailCall &&
             "Do not tail-call optimize if there is a byval argument.");
      passByValArg(Chain, DL, RegsToPass, MemOpChains, StackPtr, MFI, DAG, Arg,
                   Cpu0CCInfo, *ByValArg, Flags, Subtarget.isLittle());
      ++ByValArg;
      continue;
    }
```

```
    ...
  }
  ...
}
```

In LowerCall(), Flags.isByVal() will be true if it meets **byval** for struct type in caller function as follows,

lbdex/input/tailcall.ll

```
define internal fastcc i32 @caller9_1() nounwind noinline {
entry:
  ...
  %call = tail call i32 @callee9(%struct.S* byval @gs1) nounwind
  ret i32 %call
}
```

In LowerFormalArguments(), Flags.isByVal() will be true when it meets **byval** in callee function as follows,

lbdex/input/tailcall.ll

```
define i32 @caller12(%struct.S* nocapture byval %a0) nounwind {
entry:
  ...
}
```

At this point, I don't know how to create a make clang to generate byval IR with C language.

9.5 Function call optiomization

9.5.1 Tail call optimization

Tail call optimization is used in some situation of function call. For some situation, the caller and callee stack can share the same memory stack. When this situation applied in recursive function call, it often asymptotically reduces stack space requirements from linear, or O(n), to constant, or O(1)[5]. LLVM IR supports tailcall here[6].

The **tailcall** appeared in Cpu0ISelLowering.cpp and Cpu0InstrInfo.td are used to make tail call optimization.

lbdex/input/ch9_2_tailcall.cpp

```
int factorial(int x)
{
  if (x > 0)
    return x*factorial(x-1);
  else
    return 1;
```

(continues on next page)

[5] http://en.wikipedia.org/wiki/Tail_call
[6] http://llvm.org/docs/CodeGenerator.html#tail-call-optimization

(continued from previous page)

```
}

int test_tailcall(int a)
{
  return factorial(a);
}
```

Run Chapter9_2/ with ch9_2_tailcall.cpp will get the following result.

```
JonathantekiiMac:input Jonathan$ clang -O1 -target mips-unknown-linux-gnu -c
ch9_2_tailcall.cpp -emit-llvm -o ch9_2_tailcall.bc
JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/
llvm-dis ch9_2_tailcall.bc -o -
...
; Function Attrs: nounwind readnone
define i32 @_Z9factoriali(i32 %x) #0 {
  %1 = icmp sgt i32 %x, 0
  br i1 %1, label %tailrecurse, label %tailrecurse._crit_edge

tailrecurse:                                      ; preds = %tailrecurse, %0
  %x.tr2 = phi i32 [ %2, %tailrecurse ], [ %x, %0 ]
  %accumulator.tr1 = phi i32 [ %3, %tailrecurse ], [ 1, %0 ]
  %2 = add nsw i32 %x.tr2, -1
  %3 = mul nsw i32 %x.tr2, %accumulator.tr1
  %4 = icmp sgt i32 %2, 0
  br i1 %4, label %tailrecurse, label %tailrecurse._crit_edge

tailrecurse._crit_edge:                           ; preds = %tailrecurse, %0
  %accumulator.tr.lcssa = phi i32 [ 1, %0 ], [ %3, %tailrecurse ]
  ret i32 %accumulator.tr.lcssa
}

; Function Attrs: nounwind readnone
define i32 @_Z13test_tailcalli(i32 %a) #0 {
  %1 = tail call i32 @_Z9factoriali(i32 %a)
  ret i32 %1
}
...
JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/
llc -march=cpu0 -mcpu=cpu032II -relocation-model=static -filetype=asm
-enable-cpu0-tail-calls ch9_2_tailcall.bc -stats -o -
        .text
        .section .mdebug.abi32
        .previous
        .file "ch9_2_tailcall.bc"
        .globl        _Z9factoriali
        .align        2
        .type _Z9factoriali,@function
        .ent  _Z9factoriali           # @_Z9factoriali
_Z9factoriali:
        .frame        $sp,0,$lr
```

(continues on next page)

(continued from previous page)

```
        .mask         0x00000000,0
        .set  noreorder
        .set  nomacro
# BB#0:
        addiu $2, $zero, 1
        slt   $3, $4, $2
        bne   $3, $zero, $BB0_2
        nop
$BB0_1:                                 # %tailrecurse
                                        # =>This Inner Loop Header: Depth=1
        mul   $2, $4, $2
        addiu $4, $4, -1
        addiu $3, $zero, 0
        slt   $3, $3, $4
        bne   $3, $zero, $BB0_1
        nop
$BB0_2:                                 # %tailrecurse._crit_edge
        ret   $lr
        nop
        .set  macro
        .set  reorder
        .end  _Z9factoriali
$tmp0:
        .size _Z9factoriali, ($tmp0)-_Z9factoriali

        .globl        _Z13test_tailcalli
        .align        2
        .type _Z13test_tailcalli,@function
        .ent  _Z13test_tailcalli      # @_Z13test_tailcalli
_Z13test_tailcalli:
        .frame        $sp,0,$lr
        .mask         0x00000000,0
        .set  noreorder
        .set  nomacro
# BB#0:
        jmp   _Z9factoriali
        nop
        .set  macro
        .set  reorder
        .end  _Z13test_tailcalli
$tmp1:
        .size _Z13test_tailcalli, ($tmp1)-_Z13test_tailcalli


===-------------------------------------------------------------------------===
                          ... Statistics Collected ...
===-------------------------------------------------------------------------===

 ...
 1 cpu0-lower        - Number of tail calls
 ...
```

The tail call optimization shares caller's and callee's stack and it is applied in cpu032II only for this example (it uses

"jmp _Z9factoriali" instead of "jsub _Z9factoriali"). Then cpu032I (pass all arguments in stack) doesn't satisfy the statement, NextStackOffset <= FI.getIncomingArgSize() in isEligibleForTailCallOptimization(), and return false for the function as follows,

lbdex/chapters/Chapter9_2/Cpu0SEISelLowering.cpp

```
bool Cpu0SETargetLowering::
isEligibleForTailCallOptimization(const Cpu0CC &Cpu0CCInfo,
                                  unsigned NextStackOffset,
                                  const Cpu0FunctionInfo& FI) const {
  if (!EnableCpu0TailCalls)
    return false;

  // Return false if either the callee or caller has a byval argument.
  if (Cpu0CCInfo.hasByValArg() || FI.hasByvalArg())
    return false;

  // Return true if the callee's argument area is no larger than the
  // caller's.
  return NextStackOffset <= FI.getIncomingArgSize();
}
```

lbdex/chapters/Chapter9_2/Cpu0ISelLowering.cpp

```
/// LowerCall - functions arguments are copied from virtual regs to
/// (physical regs)/(stack frame), CALLSEQ_START and CALLSEQ_END are emitted.
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                              SmallVectorImpl<SDValue> &InVals) const {
```

```
  // Check if it's really possible to do a tail call.
  if (IsTailCall)
    IsTailCall =
      isEligibleForTailCallOptimization(Cpu0CCInfo, NextStackOffset,
                                        *MF.getInfo<Cpu0FunctionInfo>());

  if (!IsTailCall && CLI.CB && CLI.CB->isMustTailCall())
    report_fatal_error("failed to perform tail call elimination on a call "
                       "site marked musttail");

  if (IsTailCall)
    ++NumTailCalls;
```

```
  if (!IsTailCall)
    Chain = DAG.getCALLSEQ_START(Chain, NextStackOffset, 0, DL);
```

```
  if (IsTailCall)
    return DAG.getNode(Cpu0ISD::TailCall, DL, MVT::Other, Ops);
```

```
  ...
}
```

Since tailcall optimization will translate jmp instruction directly instead of jsub. The callseq_start, callseq_end, and the DAG nodes created in LowerCallResult() and LowerReturn() are needless. It creates DAGs for ch9_2_tailcall.cpp as the following Fig. 9.7,

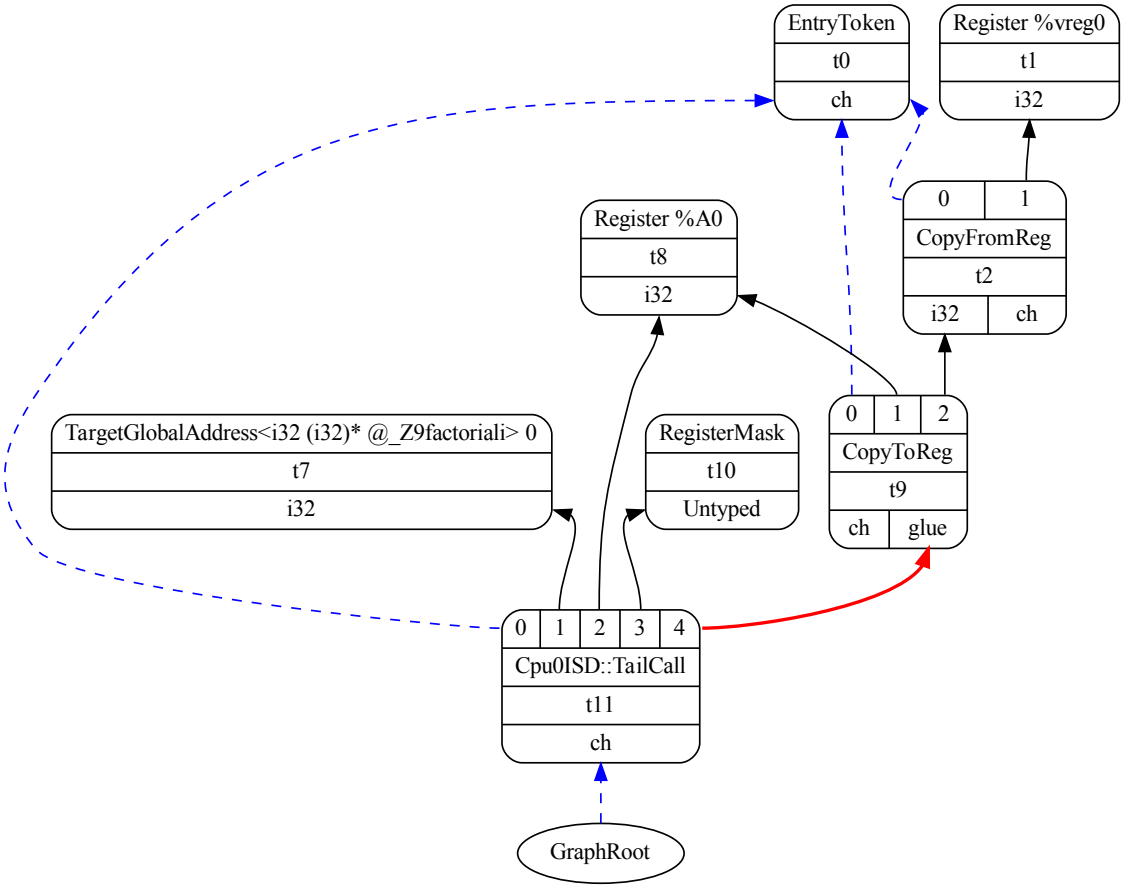


Fig. 9.7: Outgoing arguments DAGs created for ch9_2_tailcall.cpp

Finally, listing the DAGs translation of tail call as the following table.

Table 9.2: the DAGs translation of tail call

Stage	DAG	Function
Backend lowering	Cpu0ISD::TailCall	LowerCall()
Instruction selection	TAILCALL	note 1
Instruction Print	JMP	note 2

note 1: by Cpu0InstrInfo.td as follows,

lbdex/chapters/Chapter9_1/Cpu0InstrInfo.td

```
// Tail call
def Cpu0TailCall : SDNode<"Cpu0ISD::TailCall", SDT_Cpu0JmpLink,
                          [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;
```

```
def : Pat<(Cpu0TailCall (iPTR tglobaladdr:$dst)),
              (TAILCALL tglobaladdr:$dst)>;
def : Pat<(Cpu0TailCall (iPTR texternalsym:$dst)),
              (TAILCALL texternalsym:$dst)>;
```

note 2: by Cpu0InstrInfo.td and emitPseudoExpansionLowering() of Cpu0AsmPrinter.cpp as follows,

lbdex/chapters/Chapter9_1/Cpu0InstrInfo.td

```
let isCall = 1, isTerminator = 1, isReturn = 1, isBarrier = 1, hasDelaySlot = 1,
    hasExtraSrcRegAllocReq = 1, Defs = [AT] in {
  class TailCall<Instruction JumpInst> :
    PseudoSE<(outs), (ins calltarget:$target), [], IIBranch>,
    PseudoInstExpansion<(JumpInst jmptarget:$target)>;

  class TailCallReg<RegisterClass RO, Instruction JRInst,
                    RegisterClass ResRO = RO> :
    PseudoSE<(outs), (ins RO:$rs), [(Cpu0TailCall RO:$rs)], IIBranch>,
    PseudoInstExpansion<(JRInst ResRO:$rs)>;
}
```

```
let Predicates = [Ch9_1] in {
def TAILCALL : TailCall<JMP>;
def TAILCALL_R : TailCallReg<GPROut, JR>;
}
```

lbdex/chapters/Chapter9_1/Cpu0AsmPrinter.h

```
  // tblgen'erated function.
  bool emitPseudoExpansionLowering(MCStreamer &OutStreamer,
                                   const MachineInstr *MI);
```

lbdex/chapters/Chapter9_1/Cpu0AsmPrinter.cpp

```
//- emitInstruction() must exists or will have run time error.
void Cpu0AsmPrinter::emitInstruction(const MachineInstr *MI) {
//@EmitInstruction body {
  if (MI->isDebugValue()) {
    SmallString<128> Str;
    raw_svector_ostream OS(Str);

    PrintDebugValueComment(MI, OS);
```

(continues on next page)

(continued from previous page)

```
    return;
  }

  //@print out instruction:
  //  Print out both ordinary instruction and boudle instruction
  MachineBasicBlock::const_instr_iterator I = MI->getIterator();
  MachineBasicBlock::const_instr_iterator E = MI->getParent()->instr_end();

  do {
    // Do any auto-generated pseudo lowerings.
    if (emitPseudoExpansionLowering(*OutStreamer, &*I))
      continue;

    if (I->isPseudo() && !isLongBranchPseudo(I->getOpcode()))
      llvm_unreachable("Pseudo opcode found in emitInstruction()");

    MCInst TmpInst0;
    MCInstLowering.Lower(&*I, TmpInst0);
    OutStreamer->emitInstruction(TmpInst0, getSubtargetInfo());
  } while ((++I != E) && I->isInsideBundle()); // Delay slot check
}
```

Function emitPseudoExpansionLowering() is generated by TableGen and exists in Cpu0GenMCPseudoLowering.inc.

9.5.2 Recursion optimization

As last section, cpu032I cannot does tail call optimization in ch9_2_tailcall.cpp since the limitation of arguments size is not satisfied. If runnig with `clang -O3` option, it can get the same or better performance than tail call as follows,

```
JonathantekiiMac:input Jonathan$ clang -O1 -target mips-unknown-linux-gnu -c
ch9_2_tailcall.cpp -emit-llvm -o ch9_2_tailcall.bc
JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/
llvm-dis ch9_2_tailcall.bc -o -
...
; Function Attrs: nounwind readnone
define i32 @_Z9factoriali(i32 %x) #0 {
  %1 = icmp sgt i32 %x, 0
  br i1 %1, label %tailrecurse.preheader, label %tailrecurse._crit_edge

tailrecurse.preheader:                            ; preds = %0
  br label %tailrecurse

tailrecurse:                                      ; preds = %tailrecurse,
%tailrecurse.preheader
  %x.tr2 = phi i32 [ %2, %tailrecurse ], [ %x, %tailrecurse.preheader ]
  %accumulator.tr1 = phi i32 [ %3, %tailrecurse ], [ 1, %tailrecurse.preheader ]
  %2 = add nsw i32 %x.tr2, -1
  %3 = mul nsw i32 %x.tr2, %accumulator.tr1
  %4 = icmp sgt i32 %2, 0
  br i1 %4, label %tailrecurse, label %tailrecurse._crit_edge.loopexit
```

(continues on next page)

(continued from previous page)

```
tailrecurse._crit_edge.loopexit:                  ; preds = %tailrecurse
  %.lcssa = phi i32 [ %3, %tailrecurse ]
  br label %tailrecurse._crit_edge

tailrecurse._crit_edge:                           ; preds = %tailrecurse._crit
  _edge.loopexit, %0
  %accumulator.tr.lcssa = phi i32 [ 1, %0 ], [ %.lcssa, %tailrecurse._crit_edge
  .loopexit ]
  ret i32 %accumulator.tr.lcssa
}

; Function Attrs: nounwind readnone
define i32 @_Z13test_tailcalli(i32 %a) #0 {
  %1 = icmp sgt i32 %a, 0
  br i1 %1, label %tailrecurse.i.preheader, label %_Z9factoriali.exit

tailrecurse.i.preheader:                          ; preds = %0
  br label %tailrecurse.i

tailrecurse.i:                                    ; preds = %tailrecurse.i,
  %tailrecurse.i.preheader
  %x.tr2.i = phi i32 [ %2, %tailrecurse.i ], [ %a, %tailrecurse.i.preheader ]
  %accumulator.tr1.i = phi i32 [ %3, %tailrecurse.i ], [ 1, %tailrecurse.i.
  preheader ]
  %2 = add nsw i32 %x.tr2.i, -1
  %3 = mul nsw i32 %accumulator.tr1.i, %x.tr2.i
  %4 = icmp sgt i32 %2, 0
  br i1 %4, label %tailrecurse.i, label %_Z9factoriali.exit.loopexit

_Z9factoriali.exit.loopexit:                      ; preds = %tailrecurse.i
  %.lcssa = phi i32 [ %3, %tailrecurse.i ]
  br label %_Z9factoriali.exit

_Z9factoriali.exit:                               ; preds = %_Z9factoriali.
  exit.loopexit, %0
  %accumulator.tr.lcssa.i = phi i32 [ 1, %0 ], [ %.lcssa, %_Z9factoriali.
  exit.loopexit ]
  ret i32 %accumulator.tr.lcssa.i
}
...
JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/
llc -march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch9_2_tailcall.bc -o -
        .text
        .section .mdebug.abiS32
        .previous
        .file "ch9_2_tailcall.bc"
        .globl        _Z9factoriali
        .align        2
        .type _Z9factoriali,@function
        .ent  _Z9factoriali           # @_Z9factoriali
_Z9factoriali:
```

(continues on next page)

(continued from previous page)

```
        .frame        $sp,0,$lr
        .mask         0x00000000,0
        .set  noreorder
        .set  nomacro
# BB#0:
        addiu $2, $zero, 1
        ld    $3, 0($sp)
        cmp   $sw, $3, $2
        jlt   $sw, $BB0_2
        nop
$BB0_1:                                 # %tailrecurse
                                        # =>This Inner Loop Header: Depth=1
        mul   $2, $3, $2
        addiu $3, $3, -1
        addiu $4, $zero, 0
        cmp   $sw, $3, $4
        jgt   $sw, $BB0_1
        nop
$BB0_2:                                 # %tailrecurse._crit_edge
        ret   $lr
        nop
        .set  macro
        .set  reorder
        .end  _Z9factoriali
$tmp0:
        .size _Z9factoriali, ($tmp0)-_Z9factoriali

        .globl        _Z13test_tailcalli
        .align        2
        .type _Z13test_tailcalli,@function
        .ent  _Z13test_tailcalli      # @_Z13test_tailcalli
_Z13test_tailcalli:
        .frame        $sp,0,$lr
        .mask         0x00000000,0
        .set  noreorder
        .set  nomacro
# BB#0:
        addiu $2, $zero, 1
        ld    $3, 0($sp)
        cmp   $sw, $3, $2
        jlt   $sw, $BB1_2
        nop
$BB1_1:                                 # %tailrecurse.i
                                        # =>This Inner Loop Header: Depth=1
        mul   $2, $2, $3
        addiu $3, $3, -1
        addiu $4, $zero, 0
        cmp   $sw, $3, $4
        jgt   $sw, $BB1_1
        nop
$BB1_2:                                 # %_Z9factoriali.exit
        ret   $lr
```

(continues on next page)

(continued from previous page)

```
        nop
        .set  macro
        .set  reorder
        .end  _Z13test_tailcalli
$tmp1:
        .size _Z13test_tailcalli, ($tmp1)-_Z13test_tailcalli
```

According above llvm IR, `clang -O3` option replace recursion with loop by inline the callee recursion function. This is a frontend optimization through cross over function analysis.

Cpu0 doesn't support fastcc[7] but it can pass the fastcc keyword of IR. Mips supports fastcc by using as more registers as possible without following ABI specification.

9.6 Other features supporting

This section supports features for "$gp register caller saved register in PIC addressing mode", "variable number of arguments" and "dynamic stack allocation".

Run Chapter9_2/ with ch9_3_vararg.cpp to get the following error,

lbdex/input/ch9_3_vararg.cpp

```
#include <stdarg.h>

int sum_i(int amount, ...)
{
  int i = 0;
  int val = 0;
  int sum = 0;

  va_list vl;
  va_start(vl, amount);
  for (i = 0; i < amount; i++)
  {
    val = va_arg(vl, int);
    sum += val;
  }
  va_end(vl);

  return sum;
}

long long sum_ll(long long amount, ...)
{
  long long i = 0;
  long long val = 0;
  long long sum = 0;

  va_list vl;
```

(continues on next page)

[7] http://llvm.org/docs/LangRef.html#calling-conventions

(continued from previous page)

```
  va_start(vl, amount);
  for (i = 0; i < amount; i++)
  {
    val = va_arg(vl, long long);
    sum += val;
  }
  va_end(vl);

  return sum;
}

int test_va_arg()
{
  int a = sum_i(6, 0, 1, 2, 3, 4, 5);
  long long b = sum_ll(6LL, 0LL, 1LL, 2LL, 3LL, -4LL, -5LL);

  return a+(int)b; // 12
}
```

```
118-165-78-230:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_vararg.cpp -emit-llvm -o ch9_3_vararg.bc
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_3_vararg.bc -o -
...
LLVM ERROR: Cannot select: 0x7f8b6902fd10: ch = vastart 0x7f8b6902fa10,
0x7f8b6902fb10, 0x7f8b6902fc10 [ORD=9] [ID=22]
  0x7f8b6902fb10: i32 = FrameIndex<5> [ORD=7] [ID=9]
In function: _Z5sum_iiz
```

lbdex/input/ch9_3_alloc.cpp

```
// This file needed compile without option, -target mips-unknown-linux-gnu, so
// it is verified by build-run_backend2.sh or verified in lld linker support
// (build-slinker.sh).

//#include <alloca.h>
//#include <stdlib.h>

int sum(int x1, int x2, int x3, int x4, int x5, int x6)
{
  int sum = x1 + x2 + x3 + x4 + x5 + x6;

  return sum;
}

int weight_sum(int x1, int x2, int x3, int x4, int x5, int x6)
{
//  int *b = (int*)alloca(sizeof(int) * 1 * x1);
  int* b = (int*)__builtin_alloca(sizeof(int) * 1 * x1);
```

(continues on next page)

(continued from previous page)

```
  int *a = b;
  *b = x3;

  int weight = sum(3*x1, x2, x3, x4, 2*x5, x6);

  return (weight + (*a));
}

int test_alloc()
{
  int a = weight_sum(1, 2, 3, 4, 5, 6); // 31

  return a;
}
```

Run Chapter9_2 with ch9_3_alloc.cpp will get the following error.

```
118-165-72-242:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_alloc.cpp -emit-llvm -o ch9_3_alloc.bc
118-165-72-242:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032I -cpu0-s32-calls=false
-relocation-model=pic -filetype=asm ch9_3_alloc.bc -o -
...
LLVM ERROR: Cannot select: 0x7ffd8b02ff10: i32,ch = dynamic_stackalloc
0x7ffd8b02f910:1, 0x7ffd8b02fe10, 0x7ffd8b02c010 [ORD=12] [ID=48]
  0x7ffd8b02fe10: i32 = and 0x7ffd8b02fc10, 0x7ffd8b02fd10 [ORD=12] [ID=47]
    0x7ffd8b02fc10: i32 = add 0x7ffd8b02fa10, 0x7ffd8b02fb10 [ORD=12] [ID=46]
      0x7ffd8b02fa10: i32 = shl 0x7ffd8b02f910, 0x7ffd8b02f510 [ID=45]
        0x7ffd8b02f910: i32,ch = load 0x7ffd8b02ee10, 0x7ffd8b02e310,
        0x7ffd8b02b310<LD4[%1]> [ID=44]
          0x7ffd8b02e310: i32 = FrameIndex<1> [ORD=3] [ID=10]
          0x7ffd8b02b310: i32 = undef [ORD=1] [ID=2]
        0x7ffd8b02f510: i32 = Constant<2> [ID=25]
      0x7ffd8b02fb10: i32 = Constant<7> [ORD=12] [ID=16]
    0x7ffd8b02fd10: i32 = Constant<-8> [ORD=12] [ID=17]
  0x7ffd8b02c010: i32 = Constant<0> [ORD=12] [ID=8]
In function: _Z5sum_iiiiiii
```

9.6.1 The $gp register caller saved register in PIC addressing mode

According the original cpu0 web site information, it only supports **"jsub"** of 24-bit address range access. We add **"jalr"** to cpu0 and expand it to 32 bit address. We do this change for two reasons. One is that cpu0 can be expanded to 32 bit address space by only adding this instruction, and the other is cpu0 and this book are designed for tutorial. We reserve **"jalr"** as PIC mode for dynamic linking function to demonstrates:

1. How caller handles the caller saved register $gp in calling the function.
2. How the code in the shared libray function uses $gp to access global variable address.
3. The jalr for dynamic linking function is easier in implementation and faster. As we have depicted in section "pic mode" of chapter "Global variables, structs and arrays, other type". This solution is popular in reality and deserve changing cpu0 official design as a compiler book.

In chapter "Global variable", we mentioned two link type, the static link and dynamic link. The option -relocation-model=static is for static link function while option -relocation-model=pic is for dynamic link function. One instance of dynamic link function is used is for calling functions of share library. Share library includes a lots of dynamic link functions usually can be loaded at run time. Since share library can be loaded in different memory address, the global variable address be accessed cannot be decided at link time. Whatever, he distance between the global variable address and the start address of shared library function can be calculated when it has been loaded.

Let's run Chapter9_3/ with ch9_gprestore.cpp to get the following result. We putting the comments in the result for explanation.

lbdex/input/ch9_gprestore.cpp

```
extern int sum_i(int x1);

int call_sum_i() {
  int a = sum_i(1);
  a += sum_i(2);
  return a;
}
```

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032II-cpu0-s32-calls=true
-relocation-model=pic -filetype=asm ch9_gprestore.bc -o -
...
  .cpload $t9
  .set  nomacro
# BB#0:                                 # %entry
  addiu $sp, $sp, -24
$tmp0:
  .cfi_def_cfa_offset 24
  st  $lr, 12($sp)            # 4-byte Folded Spill
  st  $fp, 16($sp)              # 4-byte Folded Spill
$tmp1:
  .cfi_offset 14, -4
$tmp2:
  .cfi_offset 12, -8
  .cprestore  8    // save $gp to 8($sp)
  ld  $t9, %call16(_Z5sum_ii)($gp)
  addiu $4, $zero, 1
  jalr  $t9
  nop
  ld  $gp, 8($sp)  // restore $gp from 8($sp)
  add $8, $zero, $2
  ld  $t9, %call16(_Z5sum_ii)($gp)
  addiu $4, $zero, 2
  jalr  $t9
  nop
  ld  $gp, 8($sp)  // restore $gp from 8($sp)
  addu  $2, $2, $8
  ld  $8, 8($sp)              # 4-byte Folded Reload
  ld  $lr, 12($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 16
```

(continues on next page)

(continued from previous page)

```
  ret $lr
  nop
```

As above code comment, **".cprestore 8"** is a pseudo instruction for saving **$gp** to **8($sp)** while Instruction **"ld $gp, 8($sp)"** restore the $gp, refer to Table 8-1 of "MIPSpro TM Assembly Language Programmer's Guide"[2]. In other words, $gp is a caller saved register, so main() need to save/restore $gp before/after call the shared library _Z5sum_ii() function. In llvm Mips 3.5, it removed the .cprestore in mode PIC which meaning $gp is not a caller saved register in PIC anymore. However, it is still existed in Cpu0 and this feature can be removed by not defining it in Cpu0Config.h. The #ifdef ENABLE_GPRESTORE part of code in Cpu0 can be removed but it comes with the cost of reserving $gp register as a specific register and cannot be allocated for the program variable in PIC mode. As explained in early chapter Gloabal variable, the PIC is not critial function and the performance advantage can be ignored in dynamic link, so we keep this feature in Cpu0. Reserving $gp as a specific register in PIC will save a lot of code in programming. When reserving $gp, .cprestore can be disabled by option "-cpu0-reserve-gp". The .cpload is needed even reserving $gp (considering that programmers implement a boot code function with C and assembly mixed, programmer can set $gp value through .cpload be issued.

If enabling "-cpu0-no-cpload", and undefining ENABLE_GPRESTORE or enable "-cpu0-reserve-gp", .cpload and $gp save/restore won't be issued as follow,

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032II-cpu0-s32-calls=true
-relocation-model=pic -filetype=asm ch9_gprestore.bc -cpu0-no-cpload
-cpu0-reserve-gp -o -
...
# BB#0:
  addiu $sp, $sp, -24
$tmp0:
  .cfi_def_cfa_offset 24
  st  $lr, 20($sp)            # 4-byte Folded Spill
  st  $fp, 16($sp)            # 4-byte Folded Spill
$tmp1:
  .cfi_offset 14, -4
$tmp2:
  .cfi_offset 12, -8
  move   $fp, $sp
$tmp3:
  .cfi_def_cfa_register 12
  ld  $t9, %call16(_Z5sum_ii)($gp)
  addiu $4, $zero, 1
  jalr  $t9
  nop
  st  $2, 12($fp)
  addiu $4, $zero, 2
  ld  $t9, %call16(_Z5sum_ii)($gp)
  jalr  $t9
  nop
  ld  $3, 12($fp)
  addu  $2, $3, $2
  st  $2, 12($fp)
  move   $sp, $fp
  ld  $fp, 16($sp)            # 4-byte Folded Reload
  ld  $lr, 20($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 24
```

(continues on next page)

(continued from previous page)

```
  ret $lr
  nop
```

LLVM Mips 3.1 issues the .cpload and .cprestore and Cpu0 borrows it from that version. But now, llvm Mips replace .cpload with real instructions and remove .cprestore. It treats $gp as reserved register in PIC mode. Since the Mips assembly document which I reference say $gp is "caller save register", Cpu0 follows this document at this point and provides reserving $gp register as option.

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=mips -relocation-model=pic -filetype=asm ch9_gprestore.bc
-o -
...
# BB#0:                                 # %entry
  lui $2, %hi(_gp_disp)
  ori $2, $2, %lo(_gp_disp)
  addiu $sp, $sp, -32
$tmp0:
  .cfi_def_cfa_offset 32
  sw  $ra, 28($sp)            # 4-byte Folded Spill
  sw  $fp, 24($sp)            # 4-byte Folded Spill
  sw  $16, 20($sp)            # 4-byte Folded Spill
$tmp1:
  .cfi_offset 31, -4
$tmp2:
  .cfi_offset 30, -8
$tmp3:
  .cfi_offset 16, -12
  move   $fp, $sp
$tmp4:
  .cfi_def_cfa_register 30
  addu  $16, $2, $25
  lw  $25, %call16(_Z5sum_ii)($16)
  addiu $4, $zero, 1
  jalr  $25
  move   $gp, $16
  sw  $2, 16($fp)
  lw  $25, %call16(_Z5sum_ii)($16)
  jalr  $25
  addiu $4, $zero, 2
  lw  $1, 16($fp)
  addu  $2, $1, $2
  sw  $2, 16($fp)
  move   $sp, $fp
  lw  $16, 20($sp)            # 4-byte Folded Reload
  lw  $fp, 24($sp)            # 4-byte Folded Reload
  lw  $ra, 28($sp)            # 4-byte Folded Reload
  jr  $ra
  addiu $sp, $sp, 32
```

The following code added in Chapter9_3/ issues **".cprestore"** or the corresponding machine code before the first time of PIC function call.

lbdex/chapters/Chapter9_3/Cpu0ISelLowering.cpp

```
/// LowerCall - functions arguments are copied from virtual regs to
/// (physical regs)/(stack frame), CALLSEQ_START and CALLSEQ_END are emitted.
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                              SmallVectorImpl<SDValue> &InVals) const {
```

```
#ifdef ENABLE_GPRESTORE
  if (!Cpu0ReserveGP) {
    // If this is the first call, create a stack frame object that points to
    // a location to which .cprestore saves $gp.
    if (IsPIC && Cpu0FI->globalBaseRegFixed() && !Cpu0FI->getGPFI())
      Cpu0FI->setGPFI(MFI.CreateFixedObject(4, 0, true));
    if (Cpu0FI->needGPSaveRestore())
      MFI.setObjectOffset(Cpu0FI->getGPFI(), NextStackOffset);
  }
#endif
```

```
...
}
```

lbdex/chapters/Chapter9_3/Cpu0MachineFunction.h

```
#ifdef ENABLE_GPRESTORE
  bool needGPSaveRestore() const { return getGPFI(); }
#endif
```

lbdex/chapters/Chapter9_3/Cpu0SEFrameLowering.cpp

```
void Cpu0SEFrameLowering::emitPrologue(MachineFunction &MF,
                                       MachineBasicBlock &MBB) const {
```

```
#ifdef ENABLE_GPRESTORE
  // Restore GP from the saved stack location
  if (Cpu0FI->needGPSaveRestore()) {
    unsigned Offset = MFI.getObjectOffset(Cpu0FI->getGPFI());
    BuildMI(MBB, MBBI, dl, TII.get(Cpu0::CPRESTORE)).addImm(Offset)
      .addReg(Cpu0::GP);
  }
#endif
```

```
}
```

lbdex/chapters/Chapter9_3/Cpu0RegisterInfo.cpp

```
//- If no eliminateFrameIndex(), it will hang on run.
// pure virtual method
// FrameIndex represent objects inside a abstract stack.
// We must replace FrameIndex with an stack/frame pointer
// direct reference.
void Cpu0RegisterInfo::
eliminateFrameIndex(MachineBasicBlock::iterator II, int SPAdj,
                    unsigned FIOperandNum, RegScavenger *RS) const {
```

```
#ifdef ENABLE_GPRESTORE //2
  if (Cpu0FI->isOutArgFI(FrameIndex) || Cpu0FI->isGPFI(FrameIndex) ||
      Cpu0FI->isDynAllocFI(FrameIndex))
    Offset = spOffset;
  else
#endif
```

```
  ...
}
```

lbdex/chapters/Chapter9_3/Cpu0InstrInfo.td

```
// When handling PIC code the assembler needs .cpload and .cprestore
// directives. If the real instructions corresponding these directives
// are used, we have the same behavior, but get also a bunch of warnings
// from the assembler.
let hasSideEffects = 0 in
def CPRESTORE : Cpu0Pseudo<(outs), (ins i32imm:$loc, CPURegs:$gp),
                           ".cprestore\t$loc", []>;
```

lbdex/chapters/Chapter9_3/Cpu0AsmPrinter.cpp

```
#ifdef ENABLE_GPRESTORE
void Cpu0AsmPrinter::EmitInstrWithMacroNoAT(const MachineInstr *MI) {
  MCInst TmpInst;

  MCInstLowering.Lower(MI, TmpInst);
  OutStreamer->emitRawText(StringRef("\t.set\tmacro"));
  if (Cpu0FI->getEmitNOAT())
    OutStreamer->emitRawText(StringRef("\t.set\tat"));
  OutStreamer->emitInstruction(TmpInst, getSubtargetInfo());
  if (Cpu0FI->getEmitNOAT())
    OutStreamer->emitRawText(StringRef("\t.set\tnoat"));
  OutStreamer->emitRawText(StringRef("\t.set\tnomacro"));
}
#endif
```

```
#ifdef ENABLE_GPRESTORE
void Cpu0AsmPrinter::emitPseudoCPRestore(MCStreamer &OutStreamer,
                                         const MachineInstr *MI) {
  SmallVector<MCInst, 4> MCInsts;
  const MachineOperand &MO = MI->getOperand(0);
  assert(MO.isImm() && "CPRESTORE's operand must be an immediate.");
  int64_t Offset = MO.getImm();

  if (OutStreamer.hasRawTextSupport()) {
    // output assembly
    if (!isInt<16>(Offset)) {
      EmitInstrWithMacroNoAT(MI);
      return;
    }
    MCInst TmpInst0;
    MCInstLowering.Lower(MI, TmpInst0);
    OutStreamer.emitInstruction(TmpInst0, getSubtargetInfo());
  } else {
    // output elf
    MCInstLowering.LowerCPRESTORE(Offset, MCInsts);

    for (SmallVector<MCInst, 4>::iterator I = MCInsts.begin();
         I != MCInsts.end(); ++I)
      OutStreamer.emitInstruction(*I, getSubtargetInfo());

    return;
  }
}
#endif
```

```
//- emitInstruction() must exists or will have run time error.
void Cpu0AsmPrinter::emitInstruction(const MachineInstr *MI) {
```

```
#ifdef ENABLE_GPRESTORE
    if (I->getOpcode() == Cpu0::CPRESTORE) {
      emitPseudoCPRestore(*OutStreamer, &*I);
      continue;
    }
#endif
```

```
  ...
}
```

lbdex/chapters/Chapter9_3/Cpu0MCInstLower.h

```
#ifdef ENABLE_GPRESTORE
  void LowerCPRESTORE(int64_t Offset, SmallVector<MCInst, 4>& MCInsts);
#endif
```

lbdex/chapters/Chapter9_3/Cpu0MCInstLower.cpp

```
#ifdef ENABLE_GPRESTORE
// Lower ".cprestore offset" to "st $gp, offset($sp)".
void Cpu0MCInstLower::LowerCPRESTORE(int64_t Offset,
                                     SmallVector<MCInst, 4>& MCInsts) {
  assert(isInt<32>(Offset) && (Offset >= 0) &&
         "Imm operand of .cprestore must be a non-negative 32-bit value.");

  MCOperand SPReg = MCOperand::createReg(Cpu0::SP), BaseReg = SPReg;
  MCOperand GPReg = MCOperand::createReg(Cpu0::GP);
  MCOperand ZEROReg = MCOperand::createReg(Cpu0::ZERO);

  if (!isInt<16>(Offset)) {
    unsigned Hi = ((Offset + 0x8000) >> 16) & 0xffff;
    Offset &= 0xffff;
    MCOperand ATReg = MCOperand::createReg(Cpu0::AT);
    BaseReg = ATReg;

    // lui   at,hi
    // add   at,at,sp
    MCInsts.resize(2);
    CreateMCInst(MCInsts[0], Cpu0::LUi, ATReg, ZEROReg, MCOperand::createImm(Hi));
    CreateMCInst(MCInsts[1], Cpu0::ADD, ATReg, ATReg, SPReg);
  }

  MCInst St;
  CreateMCInst(St, Cpu0::ST, GPReg, BaseReg, MCOperand::createImm(Offset));
  MCInsts.push_back(St);
}
#endif
```

The added code of Cpu0AsmPrinter.cpp as above will call the LowerCPRESTORE() when user run program with `llc -filetype=obj`. The added code of Cpu0MCInstLower.cpp as above takes care the .cprestore machine instructions.

```
118-165-76-131:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -relocation-model=pic -filetype=
obj ch9_1.bc -o ch9_1.cpu0.o
118-165-76-131:input Jonathan$ hexdump  ch9_1.cpu0.o
...
// .cprestore machine instruction “ 01 ad 00 18”
00000d0 01 ad 00 18 09 20 00 00 01 2d 00 40 09 20 00 06
...

118-165-67-25:input Jonathan$ cat ch9_1.cpu0.s
...
```

(continues on next page)

(continued from previous page)

```
  .ent  _Z5sum_iiiiiii          # @_Z5sum_iiiiiii
_Z5sum_iiiiiii:
...
  .cpload $t9 // assign $gp = $t9 by loader when loader load re-entry function
              // (shared library) of _Z5sum_iiiiiii
  .set  nomacro
# BB#0:
...
  .ent  main                    # @main
...
  .cprestore  24  // save $gp to 24($sp)
...
```

Run `llc -static` will call jsub instruction instead of jalr as follows,

```
118-165-76-131:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -relocation-model=static -filetype=
asm ch9_1.bc -o ch9_1.cpu0.s
118-165-76-131:input Jonathan$ cat ch9_1.cpu0.s
...
  jsub  _Z5sum_iiiiiii
...
```

Run ch9_1.bc with `llc -filetype=obj`, you will find the Cx of **"jsub Cx"** is 0 since the Cx is calculated by linker as below. Mips has the same 0 in it's jal instruction.

```
// jsub _Z5sum_iiiiiii translate into 2B 00 00 00
00F0: 2B 00 00 00 01 2D 00 34 00 ED 00 3C 09 DD 00 40
```

The following code will emit "ld $gp, ($gp save slot on stack)" after jalr by creating file Cpu0EmitGPRestore.cpp which run as a function pass.

lbdex/chapters/Chapter9_3/CMakeLists.txt

```
  Cpu0EmitGPRestore.cpp
```

lbdex/chapters/Chapter9_3/Cpu0TargetMachine.cpp

```
/// Cpu0 Code Generator Pass Configuration Options.
class Cpu0PassConfig : public TargetPassConfig {
```

```
#ifdef ENABLE_GPRESTORE
  void addPreRegAlloc() override;
#endif
```

```
#ifdef ENABLE_GPRESTORE
void Cpu0PassConfig::addPreRegAlloc() {
  if (!Cpu0ReserveGP) {
    // $gp is a caller-saved register.
```

(continues on next page)

(continued from previous page)

```
    addPass(createCpu0EmitGPRestorePass(getCpu0TargetMachine()));
  }
  return;
}
#endif
```

lbdex/chapters/Chapter9_3/Cpu0.h

```
#ifdef ENABLE_GPRESTORE
  FunctionPass *createCpu0EmitGPRestorePass(Cpu0TargetMachine &TM);
#endif
```

lbdex/chapters/Chapter9_3/Cpu0EmitGPRestore.cpp

```
//===-- Cpu0EmitGPRestore.cpp - Emit GP Restore Instruction ---------------===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This pass emits instructions that restore $gp right
// after jalr instructions.
//
//===----------------------------------------------------------------------===//

#include "Cpu0.h"
#if CH >= CH9_3
#ifdef ENABLE_GPRESTORE

#include "Cpu0TargetMachine.h"
#include "Cpu0MachineFunction.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/TargetInstrInfo.h"

using namespace llvm;

#define DEBUG_TYPE "emit-gp-restore"

namespace {
  struct Inserter : public MachineFunctionPass {

    TargetMachine &TM;

    static char ID;
```

(continues on next page)

(continued from previous page)

```
    Inserter(TargetMachine &tm)
      : MachineFunctionPass(ID), TM(tm) { }

    StringRef getPassName() const override {
      return "Cpu0 Emit GP Restore";
    }

    bool runOnMachineFunction(MachineFunction &F) override;
  };
  char Inserter::ID = 0;
} // end of anonymous namespace

bool Inserter::runOnMachineFunction(MachineFunction &F) {
  Cpu0FunctionInfo *Cpu0FI = F.getInfo<Cpu0FunctionInfo>();
  const TargetSubtargetInfo *STI =  TM.getSubtargetImpl(F.getFunction());
  const TargetInstrInfo *TII = STI->getInstrInfo();

  if ((TM.getRelocationModel() != Reloc::PIC_) ||
      (!Cpu0FI->globalBaseRegFixed()))
    return false;

  bool Changed = false;
  int FI = Cpu0FI->getGPFI();

  for (MachineFunction::iterator MFI = F.begin(), MFE = F.end();
       MFI != MFE; ++MFI) {
    MachineBasicBlock& MBB = *MFI;
    MachineBasicBlock::iterator I = MFI->begin();

    /// isEHPad - Indicate that this basic block is entered via an
    /// exception handler.
    // If MBB is a landing pad, insert instruction that restores $gp after
    // EH_LABEL.
    if (MBB.isEHPad()) {
      // Find EH_LABEL first.
      for (; I->getOpcode() != TargetOpcode::EH_LABEL; ++I) ;

      // Insert ld.
      ++I;
      DebugLoc dl = I != MBB.end() ? I->getDebugLoc() : DebugLoc();
      BuildMI(MBB, I, dl, TII->get(Cpu0::LD), Cpu0::GP).addFrameIndex(FI)
                                                       .addImm(0);
      Changed = true;
    }

    while (I != MFI->end()) {
      if (I->getOpcode() != Cpu0::JALR) {
        ++I;
        continue;
      }

      DebugLoc dl = I->getDebugLoc();
```

(continues on next page)

(continued from previous page)

```
      // emit ld $gp, ($gp save slot on stack) after jalr
      BuildMI(MBB, ++I, dl, TII->get(Cpu0::LD), Cpu0::GP).addFrameIndex(FI)
                                                         .addImm(0);
      Changed = true;
    }
  }

  return Changed;
}

/// createCpu0EmitGPRestorePass - Returns a pass that emits instructions that
/// restores $gp clobbered by jalr instructions.
FunctionPass *llvm::createCpu0EmitGPRestorePass(Cpu0TargetMachine &tm) {
  return new Inserter(tm);
}

#endif

#endif
```

9.6.2 Variable number of arguments

Until now, we support fixed number of arguments in formal function definition (Incoming Arguments). This subsection supports variable number of arguments since C language supports this feature.

Run Chapter9_3/ with ch9_3_vararg.cpp as well as clang option, **clang -target mips-unknown-linux-gnu**, to get the following result,

```
118-165-76-131:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_vararg.cpp -emit-llvm -o ch9_3_vararg.bc
118-165-76-131:input Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc -march=cpu0 -mcpu=cpu032I -cpu0-s32-calls=false
-relocation-model=pic -filetype=asm ch9_3_vararg.bc -o ch9_3_vararg.cpu0.s
118-165-76-131:input Jonathan$ cat ch9_3_vararg.cpu0.s
  .section .mdebug.abi32
  .previous
  .file "ch9_3_vararg.bc"
  .text
  .globl  _Z5sum_iiz
  .align  2
  .type _Z5sum_iiz,@function
  .ent  _Z5sum_iiz              # @_Z5sum_iiz
_Z5sum_iiz:
  .frame  $fp,24,$lr
  .mask   0x00001000,-4
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -24
  st  $fp, 20($sp)            # 4-byte Folded Spill
  move    $fp, $sp
```

(continues on next page)

(continued from previous page)

```
  ld  $2, 24($fp)     // amount
  st  $2, 16($fp)     // amount
  addiu $2, $zero, 0
  st  $2, 12($fp)     // i
  st  $2, 8($fp)     // val
  st  $2, 4($fp)      // sum
  addiu $3, $fp, 28
  st  $3, 0($fp)      // arg_ptr = 2nd argument = &arg[1],
              // since &arg[0] = 24($sp)
  st  $2, 12($fp)
$BB0_1:                                 # =>This Inner Loop Header: Depth=1
  ld  $2, 16($fp)
  ld  $3, 12($fp)
  cmp $sw, $3, $2        // compare(i, amount)
  jge $BB0_4
  nop
  jmp $BB0_2
  nop
$BB0_2:                                 #   in Loop: Header=BB0_1 Depth=1
              // i < amount
  ld  $2, 0($fp)
  addiu $3, $2, 4   // arg_ptr  + 4
  st  $3, 0($fp)
  ld  $2, 0($2)     // *arg_ptr
  st  $2, 8($fp)
  ld  $3, 4($fp)      // sum
  add $2, $3, $2      // sum += *arg_ptr
  st  $2, 4($fp)
# BB#3:                                 #   in Loop: Header=BB0_1 Depth=1
              // i >= amount
  ld  $2, 12($fp)
  addiu $2, $2, 1   // i++
  st  $2, 12($fp)
  jmp $BB0_1
  nop
$BB0_4:
  ld  $2, 4($fp)
  move    $sp, $fp
  ld  $fp, 20($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 24
  ret $lr
  .set  macro
  .set  reorder
  .end  _Z5sum_iiz
$tmp1:
  .size _Z5sum_iiz, ($tmp1)-_Z5sum_iiz

  .globl  _Z11test_varargv
  .align  2
  .type _Z11test_varargv,@function
  .ent  _Z11test_varargv                    # @_Z11test_varargv
_Z11test_varargv:
```

(continues on next page)

(continued from previous page)

```
  .frame  $sp,88,$lr
  .mask   0x00004000,-4
  .set  noreorder
  .cpload $t9
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -48
  st  $lr, 44($sp)            # 4-byte Folded Spill
  st  $fp, 40($sp)            # 4-byte Folded Spill
  move    $fp, $sp
  .cprestore  32
  addiu $2, $zero, 5
  st  $2, 24($sp)
  addiu $2, $zero, 4
  st  $2, 20($sp)
  addiu $2, $zero, 3
  st  $2, 16($sp)
  addiu $2, $zero, 2
  st  $2, 12($sp)
  addiu $2, $zero, 1
  st  $2, 8($sp)
  addiu $2, $zero, 0
  st  $2, 4($sp)
  addiu $2, $zero, 6
  st  $2, 0($sp)
  ld  $t9, %call16(_Z5sum_iiz)($gp)
  jalr  $t9
  nop
  ld  $gp, 28($fp)
  st  $2, 36($fp)
  move    $sp, $fp
  ld  $fp, 40($sp)            # 4-byte Folded Reload
  ld  $lr, 44($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 48
  ret $lr
  nop
  .set  macro
  .set  reorder
  .end  _Z11test_varargv
$tmp1:
  .size _Z11test_varargv, ($tmp1)-_Z11test_varargv
```

The analysis of output ch9_3_vararg.cpu0.s as above in comment. As above code in # BB#0, we get the first argument **"amount"** from **"ld $2, 24($fp)"** since the stack size of the callee function **"_Z5sum_iiz()"** is 24. And then setting argument pointer, arg_ptr, to 0($fp), &arg[1]. Next, checking i < amount in block $BB0_1. If i < amount, than entering into $BB0_2. In $BB0_2, it does sum += *arg_ptr and arg_ptr+=4. In # BB#3, it does i+=1.

To support variable number of arguments, the following code needed to add in Chapter9_3/. The ch9_3_template.cpp is C++ template example code, it can be translated into cpu0 backend code too.

lbdex/chapters/Chapter9_3/Cpu0ISelLowering.h

```
  class Cpu0TargetLowering : public TargetLowering  {
```

```
    /// Cpu0CC - This class provides methods used to analyze formal and call
    /// arguments and inquire about calling convention information.
    class Cpu0CC {
```

```
      /// Return the function that analyzes variable argument list functions.
      llvm::CCAssignFn *varArgFn() const;
```

```
      ...
.   };
```

```
    SDValue lowerVASTART(SDValue Op, SelectionDAG &DAG) const;
    SDValue lowerFRAMEADDR(SDValue Op, SelectionDAG &DAG) const;
    SDValue lowerRETURNADDR(SDValue Op, SelectionDAG &DAG) const;
    SDValue lowerEH_RETURN(SDValue Op, SelectionDAG &DAG) const;
    SDValue lowerADD(SDValue Op, SelectionDAG &DAG) const;
```

```
    /// writeVarArgRegs - Write variable function arguments passed in registers
    /// to the stack. Also create a stack frame object for the first variable
    /// argument.
    void writeVarArgRegs(std::vector<SDValue> &OutChains, const Cpu0CC &CC,
                         SDValue Chain, const SDLoc &DL, SelectionDAG &DAG) const;
```

```
    ...
. };
```

lbdex/chapters/Chapter9_3/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
  setOperationAction(ISD::VASTART,            MVT::Other, Custom);
```

```
  // Support va_arg(): variable numbers (not fixed numbers) of arguments
  //  (parameters) for function all
  setOperationAction(ISD::VAARG,              MVT::Other, Expand);
  setOperationAction(ISD::VACOPY,             MVT::Other, Expand);
  setOperationAction(ISD::VAEND,              MVT::Other, Expand);

  //@llvm.stacksave
  // Use the default for now
  setOperationAction(ISD::STACKSAVE,          MVT::Other, Expand);
  setOperationAction(ISD::STACKRESTORE,       MVT::Other, Expand);
```

```
  ...
}
```

```
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
  switch (Op.getOpcode())
  {
```

```
  case ISD::VASTART:            return lowerVASTART(Op, DAG);
```

```
  }
  return SDValue();
}
```

```
SDValue Cpu0TargetLowering::lowerVASTART(SDValue Op, SelectionDAG &DAG) const {
  MachineFunction &MF = DAG.getMachineFunction();
  Cpu0FunctionInfo *FuncInfo = MF.getInfo<Cpu0FunctionInfo>();

  SDLoc DL = SDLoc(Op);
  SDValue FI = DAG.getFrameIndex(FuncInfo->getVarArgsFrameIndex(),
                                 getPointerTy(MF.getDataLayout()));

  // vastart just stores the address of the VarArgsFrameIndex slot into the
  // memory location argument.
  const Value *SV = cast<SrcValueSDNode>(Op.getOperand(2))->getValue();
  return DAG.getStore(Op.getOperand(0), DL, FI, Op.getOperand(1),
                      MachinePointerInfo(SV));
}
```

```
/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool IsVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         const SDLoc &DL, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                          const {
```

```
  if (IsVarArg)
    writeVarArgRegs(OutChains, Cpu0CCInfo, Chain, DL, DAG);
```

```
  ...
}
```

```
void Cpu0TargetLowering::Cpu0CC::
analyzeCallOperands(const SmallVectorImpl<ISD::OutputArg> &Args,
                    bool IsVarArg, bool IsSoftFloat, const SDNode *CallNode,
                    std::vector<ArgListEntry> &FuncArgs) {
```

```
  llvm::CCAssignFn *VarFn = varArgFn();
```

```
  for (unsigned I = 0; I != NumOpnds; ++I) {
```

```
    if (IsVarArg && !Args[I].IsFixed)
      R = VarFn(I, ArgVT, ArgVT, CCValAssign::Full, ArgFlags, CCInfo);
    else
```

```
    ...
  }
  ...
}
```

```
llvm::CCAssignFn *Cpu0TargetLowering::Cpu0CC::varArgFn() const {
  if (IsO32)
    return CC_Cpu0O32;
  else // IsS32
    return CC_Cpu0S32;
}
```

```
void Cpu0TargetLowering::writeVarArgRegs(std::vector<SDValue> &OutChains,
                                         const Cpu0CC &CC, SDValue Chain,
                                         const SDLoc &DL, SelectionDAG &DAG) const {
  unsigned NumRegs = CC.numIntArgRegs();
  const ArrayRef<MCPhysReg> ArgRegs = CC.intArgRegs();
  const CCState &CCInfo = CC.getCCInfo();
  unsigned Idx = CCInfo.getFirstUnallocated(ArgRegs);
  unsigned RegSize = CC.regSize();
  MVT RegTy = MVT::getIntegerVT(RegSize * 8);
  const TargetRegisterClass *RC = getRegClassFor(RegTy);
  MachineFunction &MF = DAG.getMachineFunction();
  MachineFrameInfo &MFI = MF.getFrameInfo();
  Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

  // Offset of the first variable argument from stack pointer.
  int VaArgOffset;

  if (NumRegs == Idx)
    VaArgOffset = alignTo(CCInfo.getNextStackOffset(), RegSize);
  else
    VaArgOffset = (int)CC.reservedArgArea() - (int)(RegSize * (NumRegs - Idx));

  // Record the frame index of the first variable argument
  // which is a value necessary to VASTART.
  int FI = MFI.CreateFixedObject(RegSize, VaArgOffset, true);
  Cpu0FI->setVarArgsFrameIndex(FI);

  // Copy the integer registers that have not been used for argument passing
  // to the argument register save area. For O32, the save area is allocated
  // in the caller's stack frame, while for N32/64, it is allocated in the
  // callee's stack frame.
```

(continues on next page)

(continued from previous page)

```
  for (unsigned I = Idx; I < NumRegs; ++I, VaArgOffset += RegSize) {
    unsigned Reg = addLiveIn(MF, ArgRegs[I], RC);
    SDValue ArgValue = DAG.getCopyFromReg(Chain, DL, Reg, RegTy);
    FI = MFI.CreateFixedObject(RegSize, VaArgOffset, true);
    SDValue PtrOff = DAG.getFrameIndex(FI, getPointerTy(DAG.getDataLayout()));
    SDValue Store = DAG.getStore(Chain, DL, ArgValue, PtrOff,
                                 MachinePointerInfo());
    cast<StoreSDNode>(Store.getNode())->getMemOperand()->setValue(
        (Value *)nullptr);
    OutChains.push_back(Store);
  }
}
```

lbdex/input/ch9_3_template.cpp

```
#include <stdarg.h>

template<class T>
T sum(T amount, ...)
{
  T i = 0;
  T val = 0;
  T sum = 0;

  va_list vl;
  va_start(vl, amount);
  for (i = 0; i < amount; i++)
  {
    val = va_arg(vl, T);
    sum += val;
  }
  va_end(vl);

  return sum;
}

int test_template()
{
  int a = (int)(sum<int>(6, 0, 1, 2, 3, 4, 5));

  return a; // 15
}

long long test_template_ll()
{
  long long a = (long long)(sum<long long>(6LL, 0LL, 1LL, 2LL, -3LL, 4LL, -5LL));

  return a; // -1
}
```

Mips qemu reference[8], you can download and run it with gcc to verify the result with printf() function at this point. We will verify the correction of the code in chapter "Verify backend on Verilog simulator" through the CPU0 Verilog language machine.

9.6.3 Dynamic stack allocation support

Even though C language is very rare using dynamic stack allocation, there are languages use it frequently. The following C example code uses it.

Chapter9_3 supports dynamic stack allocation with the following code added.

lbdex/chapters/Chapter9_2/Cpu0FrameLowering.cpp

```
// Eliminate ADJCALLSTACKDOWN, ADJCALLSTACKUP pseudo instructions
MachineBasicBlock::iterator Cpu0FrameLowering::
eliminateCallFramePseudoInstr(MachineFunction &MF, MachineBasicBlock &MBB,
                              MachineBasicBlock::iterator I) const {
#if CH >= CH9_3 // dynamic alloc
  unsigned SP = Cpu0::SP;

  if (!hasReservedCallFrame(MF)) {
    int64_t Amount = I->getOperand(0).getImm();
    if (I->getOpcode() == Cpu0::ADJCALLSTACKDOWN)
      Amount = -Amount;

    STI.getInstrInfo()->adjustStackPtr(SP, Amount, MBB, I);
  }
#endif // dynamic alloc

  return MBB.erase(I);
}
```

lbdex/chapters/Chapter9_3/Cpu0SEFrameLowering.cpp

```
void Cpu0SEFrameLowering::emitPrologue(MachineFunction &MF,
                                       MachineBasicBlock &MBB) const {
```

```
  unsigned FP = Cpu0::FP;
  unsigned ZERO = Cpu0::ZERO;
  unsigned ADDu = Cpu0::ADDu;
```

```
  // if framepointer enabled, set it to point to the stack pointer.
  if (hasFP(MF)) {
    if (Cpu0FI->callsEhDwarf()) {
      BuildMI(MBB, MBBI, dl, TII.get(ADDu), Cpu0::V0).addReg(FP).addReg(ZERO)
        .setMIFlag(MachineInstr::FrameSetup);
    }
    //@ Insert instruction "move $fp, $sp" at this location.
```

(continues on next page)

[8] http://developer.mips.com/clang-llvm/

(continued from previous page)

```
    BuildMI(MBB, MBBI, dl, TII.get(ADDu), FP).addReg(SP).addReg(ZERO)
      .setMIFlag(MachineInstr::FrameSetup);

    // emit ".cfi_def_cfa_register $fp"
    unsigned CFIIndex = MF.addFrameInst(MCCFIInstruction::createDefCfaRegister(
        nullptr, MRI->getDwarfRegNum(FP, true)));
    BuildMI(MBB, MBBI, dl, TII.get(TargetOpcode::CFI_INSTRUCTION))
        .addCFIIndex(CFIIndex);
  }
```

```
}
```

```
void Cpu0SEFrameLowering::emitEpilogue(MachineFunction &MF,
                                       MachineBasicBlock &MBB) const {
```

```
  unsigned FP = Cpu0::FP;
  unsigned ZERO = Cpu0::ZERO;
  unsigned ADDu = Cpu0::ADDu;

  // if framepointer enabled, restore the stack pointer.
  if (hasFP(MF)) {
    // Find the first instruction that restores a callee-saved register.
    MachineBasicBlock::iterator I = MBBI;

    for (unsigned i = 0; i < MFI.getCalleeSavedInfo().size(); ++i)
      --I;

    // Insert instruction "move $sp, $fp" at this location.
    BuildMI(MBB, I, DL, TII.get(ADDu), SP).addReg(FP).addReg(ZERO);
  }
```

```
}
```

```
  unsigned FP = Cpu0::FP;

  // Mark $fp as used if function has dedicated frame pointer.
  if (hasFP(MF))
    setAliasRegs(MF, SavedRegs, FP);
```

lbdex/chapters/Chapter9_3/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
  setOperationAction(ISD::DYNAMIC_STACKALLOC, MVT::i32,  Expand);
```

```
  setStackPointerRegisterToSaveRestore(Cpu0::SP);
```

```
}
```

lbdex/chapters/Chapter9_3/Cpu0RegisterInfo.cpp

```
BitVector Cpu0RegisterInfo::
getReservedRegs(const MachineFunction &MF) const {
```

```
  // Reserve FP if this function should have a dedicated frame pointer register.
  if (MF.getSubtarget().getFrameLowering()->hasFP(MF)) {
    Reserved.set(Cpu0::FP);
  }
```

```
}
```

```
//- If no eliminateFrameIndex(), it will hang on run.
// pure virtual method
// FrameIndex represent objects inside a abstract stack.
// We must replace FrameIndex with an stack/frame pointer
// direct reference.
void Cpu0RegisterInfo::
eliminateFrameIndex(MachineBasicBlock::iterator II, int SPAdj,
                    unsigned FIOperandNum, RegScavenger *RS) const {
```

```
  if (Cpu0FI->isOutArgFI(FrameIndex) || Cpu0FI->isGPFI(FrameIndex) ||
      Cpu0FI->isDynAllocFI(FrameIndex))
    Offset = spOffset;
```

```
}
```

Run Chapter9_3 with ch9_3_alloc.cpp will get the following correct result.

```
118-165-72-242:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_alloc.cpp -emit-llvm -o ch9_3_alloc.bc
118-165-72-242:input Jonathan$ llvm-dis ch9_3_alloc.bc -o ch9_3_alloc.ll
118-165-72-242:input Jonathan$ cat ch9_3_alloc.ll
; ModuleID = 'ch9_3_alloc.bc'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:
32:64-S128"
target triple = "x86_64-apple-macosx10.8.0"

define i32 @_Z5sum_iiiiiii(i32 %x1, i32 %x2, i32 %x3, i32 %x4, i32 %x5, i32 %x6)
 nounwind uwtable ssp {
  ...
  %9 = alloca i8, i32 %8      // int* b = (int*)__builtin_alloca(sizeof(int) * 1 * x1);
  %10 = bitcast i8* %9 to i32*
  store i32* %10, i32** %b, align 4
  ...
```

(continues on next page)

(continued from previous page)

```
}
...

118-165-72-242:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -mcpu=cpu032I -cpu0-s32-calls=false
-relocation-model=pic -filetype=asm ch9_3_alloc.bc -o ch9_3_alloc.cpu0.s
118-165-72-242:input Jonathan$ cat ch9_3_alloc.cpu0.s
...
    .globl  _Z10weight_sumiiiiii
  .align  2
  .type _Z10weight_sumiiiiii,@function
  .ent  _Z10weight_sumiiiiii    # @_Z10weight_sumiiiiii
_Z10weight_sumiiiiii:
  .frame  $fp,48,$lr
  .mask   0x00005000,-4
  .set  noreorder
  .cpload $t9
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -48
  st  $lr, 44($sp)            # 4-byte Folded Spill
  st  $fp, 40($sp)            # 4-byte Folded Spill
  move   $fp, $sp
  .cprestore  24
  ld  $2, 68($fp)
  ld  $3, 64($fp)
  ld  $t9, 60($fp)
  ld  $7, 56($fp)
  st  $4, 36($fp)
  st  $5, 32($fp)
  st  $7, 28($fp)
  st  $t9, 24($fp)
  st  $3, 20($fp)
  st  $2, 16($fp)
  shl $2, $2, 2    // $2 = sizeof(int) * 1 * x2;
  addiu $2, $2, 7
  addiu $3, $zero, -8
  and $2, $2, $3
  addiu $sp, $sp, 0
  subu  $2, $sp, $2
  addu  $sp, $zero, $2  // set sp to the bottom of alloca area
  addiu $sp, $sp, 0
  st  $2, 12($fp)
  st  $2, 8($fp)
  ld  $2, 12($fp)
  ld  $3, 28($fp)
  st  $3, 0($2)    // *b = x3
  ld  $5, 32($fp)
  ld  $2, 36($fp)
  ld  $3, 20($fp)
  ld  $4, 28($fp)
  ld  $t9, 24($fp)
```

(continues on next page)

(continued from previous page)

```
  ld  $7, 16($fp)
  addiu $sp, $sp, -24
  st  $7, 20($sp)
  st  $t9, 12($sp)
  st  $4, 8($sp)
  shl $3, $3, 1
  st  $3, 16($sp)
  addiu $3, $zero, 3
  mul $4, $2, $3
  ld  $t9, %call16(_Z3sumiiiiii)($gp)
  jalr  $t9
  nop
  ld  $gp, 24($fp)
  addiu $sp, $sp, 24
  st  $2, 4($fp)
  ld  $3, 8($fp)
  ld  $3, 0($3)
  addu  $2, $2, $3
  move   $sp, $fp
  ld  $fp, 40($sp)            # 4-byte Folded Reload
  ld  $lr, 44($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 48
  ret $lr
  nop
  .set  macro
  .set  reorder
  .end  _Z10weight_sumiiiiii
$func_end1:
  .size _Z10weight_sumiiiiii, ($func_end1)-_Z10weight_sumiiiiii
...
```

As you can see, the dynamic stack allocation needs frame pointer register **fp** support. As above assembly, the sp is adjusted to (sp - 48) when it enter the function as usual by instruction **addiu $sp, $sp, -48**. Next, the fp is set to sp where the position is just above alloca() spaces area as Fig. 9.8 when meets instruction **move $fp, $sp**. After that, the sp is changed to the area just below of alloca(). Remind, the alloca() area where the b point to, **"*b = (int*)__builtin_alloca(sizeof(int) * 2 * x6)"**, is allocated at run time since the size of the space which depends on x1 variable and cannot be calculated at link time.

Fig. 9.9 depict how the stack pointer changes back to the caller stack bottom. As above, the **fp** is set to the address just above of alloca(). The first step is changing the sp to fp by instruction **move $sp, $fp**. Next, sp is changed back to caller stack bottom by instruction **addiu $sp, $sp, 40**.

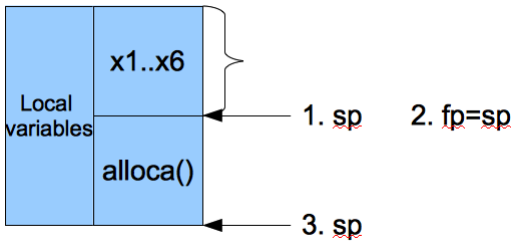


Fig. 9.8: Frame pointer changes when enter function

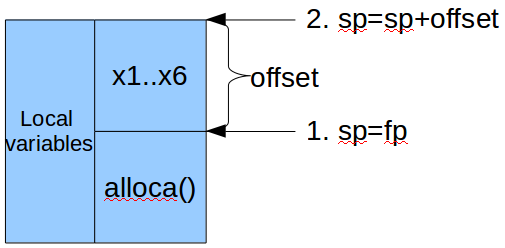


Fig. 9.9: Stack pointer changes when exit function

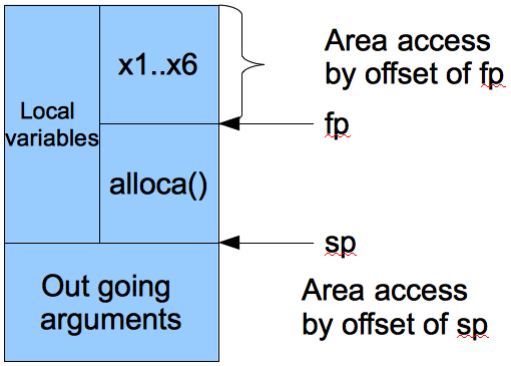


Fig. 9.10: fp and sp access areas

Using fp to keep the old stack pointer value is not the only solution. Actually, we can keep the size of alloca() spaces on a specific memory address and the sp can be set back to the the old sp by adding the size of alloca() spaces. Most ABI like Mips and ARM access the above area of alloca() by fp and the below area of alloca() by sp, as Fig. 9.10 depicted. The reason for this definition is the speed for local variable access. Since the RISC CPU use immediate offset for load and store as below, using fp and sp for access both areas of local variables have better performance comparing to use the sp only.

```
ld      $2, 64($fp)
st      $3, 4($sp)
```

Cpu0 uses fp and sp to access the above and below areas of alloca() too. As ch9_3_alloc.cpu0.s, it accesses local variables (above of alloca()) by fp offset and outgoing arguments (below of alloca()) by sp offset.

And more, the "move $sp, $fp" is the alias instruction of "addu $fp, $sp, $zero". The machine code is the latter one, and the former is only for easy understanding by user. This alias comes from code added in Chapter3_2 and Chapter3_5 as follows,

lbdex/chapters/Chapter3_2/InstPrinter/Cpu0InstPrinter.cpp

```
void Cpu0InstPrinter::printInst(const MCInst *MI, uint64_t Address,
                                StringRef Annot, const MCSubtargetInfo &STI,
                                raw_ostream &O) {
  // Try to print any aliases first.
  if (!printAliasInstr(MI, Address, O))
```

lbdex/chapters/Chapter3_5/Cpu0InstrInfo.td

```
class Cpu0InstAlias<string Asm, dag Result, bit Emit = 0b1> :
  InstAlias<Asm, Result, Emit>;
```

```
let Predicates = [Ch3_5] in {
//===----------------------------------------------------------------------===//
// Instruction aliases
//===----------------------------------------------------------------------===//
def : Cpu0InstAlias<"move $dst, $src",
                    (ADDu GPROut:$dst, GPROut:$src,ZERO), 1>;
}
```

Finally the MFI->hasVarSizedObjects() defined in hasReservedCallFrame() of Cpu0SEFrameLowering.cpp is true when it meets "%9 = alloca i8, i32 %8" of IR which corresponding "(int*)__builtin_alloca(sizeof(int) * 1 * x1);" of C. It will generate asm "addiu $sp, $sp, -24" for ch9_3_alloc.cpp by calling "adjustStackPtr()" in eliminateCallFramePseudoInstr() of Cpu0FrameLowering.cpp.

File ch9_3_longlongshift.cpp is for type "long long shift operations" which can be tested now as follows.

lbdex/input/ch9_3_longlongshift.cpp

```
#include "debug.h"

long long test_longlong_shift1()
{
  long long a = 4;
  long long b = 0x12;
  long long c;
  long long d;

  c = (b >> a);  // cc = 0x1
  d = (b << a);  // cc = 0x120

  long long e = 0x7FFFFFFFFFFFFFFFLL >> 63;
  return (c+d+e); // 0x121 = 289
}

long long test_longlong_shift2()
{
  long long a = 48;
  long long b = 0x0016666660000000a;
  long long c;
```

(continues on next page)

(continued from previous page)

```
  c = (b >> a);

  return c; // 22
}
```

```
114-37-150-209:input Jonathan$ clang -O0 -target mips-unknown-linux-gnu
-c ch9_3_longlongshift.cpp -emit-llvm -o ch9_3_longlongshift.bc

114-37-150-209:input Jonathan$ ~/llvm/test/build/bin/
llvm-dis ch9_3_longlongshift.bc -o -
...
; Function Attrs: nounwind
define i64 @_Z19test_longlong_shiftv() #0 {
  %a = alloca i64, align 8
  %b = alloca i64, align 8
  %c = alloca i64, align 8
  %d = alloca i64, align 8
  store i64 4, i64* %a, align 8
  store i64 18, i64* %b, align 8
  %1 = load i64* %b, align 8
  %2 = load i64* %a, align 8
  %3 = ashr i64 %1, %2
  store i64 %3, i64* %c, align 8
  %4 = load i64* %b, align 8
  %5 = load i64* %a, align 8
  %6 = shl i64 %4, %5
  store i64 %6, i64* %d, align 8
  %7 = load i64* %c, align 8
  %8 = load i64* %d, align 8
  %9 = add nsw i64 %7, %8
  ret i64 %9
}
...
114-37-150-209:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch9_3_longlongshift.bc -o -
  .text
  .section .mdebug.abi32
  .previous
  .file "ch9_3_longlongshift.bc"
  .globl  _Z20test_longlong_shift1v
  .align  2
  .type _Z20test_longlong_shift1v,@function
  .ent  _Z20test_longlong_shift1v # @_Z20test_longlong_shift1v
_Z20test_longlong_shift1v:
  .frame  $fp,56,$lr
  .mask   0x00005000,-4
  .set  noreorder
  .set  nomacro
# BB#0:
```

(continues on next page)

(continued from previous page)

```
  addiu $sp, $sp, -56
  st  $lr, 52($sp)            # 4-byte Folded Spill
  st  $fp, 48($sp)            # 4-byte Folded Spill
  move   $fp, $sp
  addiu $2, $zero, 4
  st  $2, 44($fp)
  addiu $4, $zero, 0
  st  $4, 40($fp)
  addiu $5, $zero, 18
  st  $5, 36($fp)
  st  $4, 32($fp)
  ld  $2, 44($fp)
  st  $2, 8($sp)
  jsub  __lshrdi3
  nop
  st  $3, 28($fp)
  st  $2, 24($fp)
  ld  $2, 44($fp)
  st  $2, 8($sp)
  ld  $4, 32($fp)
  ld  $5, 36($fp)
  jsub  __ashldi3
  nop
  st  $3, 20($fp)
  st  $2, 16($fp)
  ld  $4, 28($fp)
  addu  $4, $4, $3
  cmp $sw, $4, $3
  andi  $3, $sw, 1
  addu  $2, $3, $2
  ld  $3, 24($fp)
  addu  $2, $3, $2
  addu  $3, $zero, $4
  move   $sp, $fp
  ld  $fp, 48($sp)            # 4-byte Folded Reload
  ld  $lr, 52($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 56
  ret $lr
  nop
  .set  macro
  .set  reorder
  .end  _Z20test_longlong_shift1v
$tmp0:
  .size _Z20test_longlong_shift1v, ($tmp0)-_Z20test_longlong_shift1v

  .globl  _Z20test_longlong_shift2v
  .align  2
  .type _Z20test_longlong_shift2v,@function
  .ent  _Z20test_longlong_shift2v # @_Z20test_longlong_shift2v
_Z20test_longlong_shift2v:
  .frame  $fp,48,$lr
  .mask   0x00005000,-4
```

(continues on next page)

(continued from previous page)

```
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -48
  st  $lr, 44($sp)            # 4-byte Folded Spill
  st  $fp, 40($sp)            # 4-byte Folded Spill
  move   $fp, $sp
  addiu $2, $zero, 48
  st  $2, 36($fp)
  addiu $2, $zero, 0
  st  $2, 32($fp)
  addiu $5, $zero, 10
  st  $5, 28($fp)
  lui $2, 22
  ori $4, $2, 26214
  st  $4, 24($fp)
  ld  $2, 36($fp)
  st  $2, 8($sp)
  jsub  __lshrdi3
  nop
  st  $3, 20($fp)
  st  $2, 16($fp)
  move   $sp, $fp
  ld  $fp, 40($sp)            # 4-byte Folded Reload
  ld  $lr, 44($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 48
  ret $lr
  nop
  .set  macro
  .set  reorder
  .end  _Z20test_longlong_shift2v
$tmp1:
  .size _Z20test_longlong_shift2v, ($tmp1)-_Z20test_longlong_shift2v
```

9.6.4 Variable sized array support

LLVM supports variable sized arrays in C99[9]. The following code added for this support. Set them to expand, meaning llvm uses other DAGs replace them.

lbdex/chapters/Chapter9_3/Cpu0ISelLowering.cpp

```
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
  switch (Op.getOpcode())
  {
```

[9] http://www.llvm.org/docs/LangRef.html#llvm-stacksave-intrinsic

```
  // Use the default for now
  setOperationAction(ISD::STACKSAVE,          MVT::Other, Expand);
  setOperationAction(ISD::STACKRESTORE,       MVT::Other, Expand);
```

```
    ...
  }
  ...
}
```

lbdex/input/ch9_3_stacksave.cpp

```
int test_stacksaverestore(unsigned x) {
  // CHECK: call i8* @llvm.stacksave()
  char s1[x];
  s1[x] = 5;

  return s1[x];
  // CHECK: call void @llvm.stackrestore(i8*
}
```

```
JonathantekiiMac:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_stacksave.cpp -emit-llvm -o ch9_3_stacksave.bc
JonathantekiiMac:input Jonathan$ llvm-dis ch9_3_stacksave.bc -o -

define i32 @_Z21test_stacksaverestorej(i32 zeroext %x) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i8*
  %3 = alloca i32
  store i32 %x, i32* %1, align 4
  %4 = load i32, i32* %1, align 4
  %5 = call i8* @llvm.stacksave()
  store i8* %5, i8** %2
  %6 = alloca i8, i32 %4, align 1
  %7 = load i32, i32* %1, align 4
  %8 = getelementptr inbounds i8, i8* %6, i32 %7
  store i8 5, i8* %8, align 1
  %9 = load i32, i32* %1, align 4
  %10 = getelementptr inbounds i8, i8* %6, i32 %9
  %11 = load i8, i8* %10, align 1
  %12 = sext i8 %11 to i32
  store i32 1, i32* %3
  %13 = load i8*, i8** %2
  call void @llvm.stackrestore(i8* %13)
  ret i32 %12
}

JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch9_3_stacksave.bc -o -
...
```

9.6.5 Function related Intrinsics support

I think these llvm instinsic IRs are for the implementation of exception handle[1011]. With these IRs, programmer can record the frame address and return address to be used in implementing program of exception handler by C++ as the example below. In order to support these llvm intrinsic IRs, the following code added to Cpu0 backend.

lbdex/chapters/Chapter9_3/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
  setOperationAction(ISD::EH_RETURN, MVT::Other, Custom);
```

```
  setOperationAction(ISD::ADD,                MVT::i32,   Custom);
```

```
  ...
}
```

```
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
  switch (Op.getOpcode())
  {
```

```
  case ISD::FRAMEADDR:          return lowerFRAMEADDR(Op, DAG);
  case ISD::RETURNADDR:         return lowerRETURNADDR(Op, DAG);
  case ISD::EH_RETURN:          return lowerEH_RETURN(Op, DAG);
  case ISD::ADD:                return lowerADD(Op, DAG);
```

```
    ...
  }
  ...
}
```

```
SDValue Cpu0TargetLowering::
lowerFRAMEADDR(SDValue Op, SelectionDAG &DAG) const {
  // check the depth
  assert((cast<ConstantSDNode>(Op.getOperand(0))->getZExtValue() == 0) &&
         "Frame address can only be determined for current frame.");

  MachineFrameInfo &MFI = DAG.getMachineFunction().getFrameInfo();
  MFI.setFrameAddressIsTaken(true);
  EVT VT = Op.getValueType();
  SDLoc DL(Op);
  SDValue FrameAddr = DAG.getCopyFromReg(
      DAG.getEntryNode(), DL, Cpu0::FP, VT);
```

(continues on next page)

[10] http://llvm.org/docs/ExceptionHandling.html#overview
[11] http://llvm.org/docs/LangRef.html#llvm-returnaddress-intrinsic

(continued from previous page)

```
  return FrameAddr;
}

SDValue Cpu0TargetLowering::lowerRETURNADDR(SDValue Op,
                                            SelectionDAG &DAG) const {
  if (verifyReturnAddressArgumentIsConstant(Op, DAG))
    return SDValue();

  // check the depth
  assert((cast<ConstantSDNode>(Op.getOperand(0))->getZExtValue() == 0) &&
         "Return address can be determined only for current frame.");

  MachineFunction &MF = DAG.getMachineFunction();
  MachineFrameInfo &MFI = MF.getFrameInfo();
  MVT VT = Op.getSimpleValueType();
  unsigned LR = Cpu0::LR;
  MFI.setReturnAddressIsTaken(true);

  // Return LR, which contains the return address. Mark it an implicit live-in.
  unsigned Reg = MF.addLiveIn(LR, getRegClassFor(VT));
  return DAG.getCopyFromReg(DAG.getEntryNode(), SDLoc(Op), Reg, VT);
}

// An EH_RETURN is the result of lowering llvm.eh.return which in turn is
// generated from __builtin_eh_return (offset, handler)
// The effect of this is to adjust the stack pointer by "offset"
// and then branch to "handler".
SDValue Cpu0TargetLowering::lowerEH_RETURN(SDValue Op, SelectionDAG &DAG)
                                                                     const {
  MachineFunction &MF = DAG.getMachineFunction();
  Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

  Cpu0FI->setCallsEhReturn();
  SDValue Chain     = Op.getOperand(0);
  SDValue Offset    = Op.getOperand(1);
  SDValue Handler   = Op.getOperand(2);
  SDLoc DL(Op);
  EVT Ty = MVT::i32;

  // Store stack offset in V1, store jump target in V0. Glue CopyToReg and
  // EH_RETURN nodes, so that instructions are emitted back-to-back.
  unsigned OffsetReg = Cpu0::V1;
  unsigned AddrReg = Cpu0::V0;
  Chain = DAG.getCopyToReg(Chain, DL, OffsetReg, Offset, SDValue());
  Chain = DAG.getCopyToReg(Chain, DL, AddrReg, Handler, Chain.getValue(1));
  return DAG.getNode(Cpu0ISD::EH_RETURN, DL, MVT::Other, Chain,
                     DAG.getRegister(OffsetReg, Ty),
                     DAG.getRegister(AddrReg, getPointerTy(MF.getDataLayout())),
                     Chain.getValue(1));
}

SDValue Cpu0TargetLowering::lowerADD(SDValue Op, SelectionDAG &DAG) const {
```

(continues on next page)

(continued from previous page)

```
  if (Op->getOperand(0).getOpcode() != ISD::FRAMEADDR
      || cast<ConstantSDNode>
        (Op->getOperand(0).getOperand(0))->getZExtValue() != 0
      || Op->getOperand(1).getOpcode() != ISD::FRAME_TO_ARGS_OFFSET)
    return SDValue();

  MachineFunction &MF = DAG.getMachineFunction();
  Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

  Cpu0FI->setCallsEhDwarf();
  return Op;
}
```

frameaddress and returnaddress intrinsics

Run with the following input to get the following result.

lbdex/input/ch9_3_frame_return_addr.cpp

```
int display_frameaddress() {
  return (int)__builtin_frame_address(0);
}

extern int fn();

int display_returnaddress() {
  int a = (int)__builtin_return_address(0);
  fn();
  return a;
}
```

```
JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/
llvm-dis ch9_3_frame_return_addr.bc -o -
...
; Function Attrs: nounwind
define i32 @_Z20display_frameaddressv() #0 {
  %1 = call i8* @llvm.frameaddress(i32 0)
  %2 = ptrtoint i8* %1 to i32
  ret i32 %2
}

; Function Attrs: nounwind readnone
declare i8* @llvm.frameaddress(i32) #1

define i32 @_Z22display_returnaddressv() #2 {
  %a = alloca i32, align 4
  %1 = call i8* @llvm.returnaddress(i32 0)
  %2 = ptrtoint i8* %1 to i32
  store i32 %2, i32* %a, align 4
```

(continues on next page)

(continued from previous page)

```
  %3 = call i32 @_Z2fnv()
  %4 = load i32, i32* %a, align 4
  ret i32 %4
}

JonathantekiiMac:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=static -filetype=asm ch9_3_frame_return_addr.bc
-o -
      .text
      .section .mdebug.abi32
      .previous
      .file "ch9_3_frame_return_addr.bc"
      .globl  _Z20display_frameaddressv
      .align  2
      .type _Z20display_frameaddressv,@function
      .ent    _Z20display_frameaddressv # @_Z20display_frameaddressv
_Z20display_frameaddressv:
      .frame  $fp,8,$lr
      .mask           0x00001000,-4
      .set    noreorder
      .set    nomacro
# BB#0:
      addiu $sp, $sp, -8
      st      $fp, 4($sp)                                             # 4-byte␣
↪Folded Spill
      move     $fp, $sp
      addu    $2, $zero, $fp
      move     $sp, $fp
      ld      $fp, 4($sp)                                             # 4-byte␣
↪Folded Reload
      addiu $sp, $sp, 8
      ret $lr
      nop
      .set    macro
      .set    reorder
      .end    _Z20display_frameaddressv
$func_end0:
      .size _Z20display_frameaddressv, ($func_end0)-_Z20display_frameaddressv

      .globl  _Z22display_returnaddress1v
      .align  2
      .type _Z22display_returnaddress1v,@function
      .ent    _Z22display_returnaddress1v # @_Z22display_returnaddress1v
_Z22display_returnaddress1v:
      .cfi_startproc
      .frame  $fp,24,$lr
      .mask           0x00005000,-4
      .set    noreorder
      .set    nomacro
# BB#0:
      addiu $sp, $sp, -24
$tmp0:
```

(continues on next page)

(continued from previous page)

```
      .cfi_def_cfa_offset 24
      st      $lr, 20($sp)                                    # 4-byte Folded␣
↪Spill
      st      $fp, 16($sp)                                    # 4-byte Folded␣
↪Spill
$tmp1:
      .cfi_offset 14, -4
$tmp2:
      .cfi_offset 12, -8
      move     $fp, $sp
$tmp3:
      .cfi_def_cfa_register 12
      st      $lr, 12($fp)
      jsub    _Z2fnv
      nop
      ld      $2, 12($fp)
      move     $sp, $fp
      ld      $fp, 16($sp)                                    # 4-byte Folded␣
↪Reload
      ld      $lr, 20($sp)                                    # 4-byte Folded␣
↪Reload
      addiu $sp, $sp, 24
      ret $lr
      nop
      .set    macro
      .set    reorder
      .end    _Z22display_returnaddress1v
$func_end1:
      .size _Z22display_returnaddress1v, ($func_end1)-_Z22display_returnaddress1v
      .cfi_endproc
```

The asm "ld $2, 12($fp)" in function _Z22display_returnaddress1v reloads $lr to $2 after "jsub _Z3fnv". The reason that Cpu0 doesn't produce "addiu $2, $zero, $lr" is if a bug program in _Z3fnv changes $lr value without following ABI then it will get the wrong $lr to $2. The following code kills $lr register and make the reference to $lr by loading from stack slot rather than uses register directly.

lbdex/chapters/Chapter9_1/Cpu0SEFrameLowering.cpp

```
bool Cpu0SEFrameLowering::
spillCalleeSavedRegisters(MachineBasicBlock &MBB,
                          MachineBasicBlock::iterator MI,
                          const std::vector<CalleeSavedInfo> &CSI,
                          const TargetRegisterInfo *TRI) const {
  ...
  for (unsigned i = 0, e = CSI.size(); i != e; ++i) {
    // Add the callee-saved register as live-in. Do not add if the register is
    // LR and return address is taken, because it has already been added in
    // method Cpu0TargetLowering::LowerRETURNADDR.
    // It's killed at the spill, unless the register is LR and return address
    // is taken.
    unsigned Reg = CSI[i].getReg();
```

(continues on next page)

(continued from previous page)

```
    bool IsRAAndRetAddrIsTaken = (Reg == Cpu0::LR)
        && MF->getFrameInfo()->isReturnAddressTaken();
    if (!IsRAAndRetAddrIsTaken)
      EntryBlock->addLiveIn(Reg);

    // Insert the spill to the stack frame.
    bool IsKill = !IsRAAndRetAddrIsTaken;
    const TargetRegisterClass *RC = TRI->getMinimalPhysRegClass(Reg);
    TII.storeRegToStackSlot(*EntryBlock, MI, Reg, IsKill,
                            CSI[i].getFrameIdx(), RC, TRI);
  }
  ...
}
```

eh.return intrinsic

Considering the following code,

unwind example

```
int func() {
  if (...) {
    throw std::bad_alloc();
  }
}

int A() {
  try {
    func();
  }
  catch(...) {
    ...
  }
}

int B() {
  try {
    func();
    A();
  }
  catch(...) {
    ...
  }
}
```

When B() -> call func() -> exception, unwind frame to B and handle over to B's execption handler; when B() -> call A() -> call func() -> exception, unwind frame to A and handle over to A's exception handler.

__builtin_eh_return (offset, handler), which adjusts the stack by offset and then jumps to the handler. __builtin_eh_return is used in GCC unwinder (libgcc), but not in LLVM unwinder (libunwind)[12].

[12] https://llvm.org/docs/ExceptionHandling.html#exception-handling-support-on-the-target

Beside lowerRETURNADDR() in Cpu0ISelLowering, the following code is for eh.return supporting only, and it can run with input ch9_3_detect_exception.cpp as below.

lbdex/chapters/Chapter9_3/Cpu0SEFrameLowering.cpp

```
void Cpu0SEFrameLowering::emitPrologue(MachineFunction &MF,
                                       MachineBasicBlock &MBB) const {
```

```
  if (Cpu0FI->callsEhReturn()) {
    // Insert instructions that spill eh data registers.
    for (int I = 0; I < ABI.EhDataRegSize(); ++I) {
      if (!MBB.isLiveIn(ABI.GetEhDataReg(I)))
        MBB.addLiveIn(ABI.GetEhDataReg(I));
      TII.storeRegToStackSlot(MBB, MBBI, ABI.GetEhDataReg(I), false,
                              Cpu0FI->getEhDataRegFI(I), RC, &RegInfo);
    }

    // Emit .cfi_offset directives for eh data registers.
    for (int I = 0; I < ABI.EhDataRegSize(); ++I) {
      int64_t Offset = MFI.getObjectOffset(Cpu0FI->getEhDataRegFI(I));
      unsigned Reg = MRI->getDwarfRegNum(ABI.GetEhDataReg(I), true);
      unsigned CFIIndex = MF.addFrameInst(
          MCCFIInstruction::createOffset(nullptr, Reg, Offset));
      BuildMI(MBB, MBBI, dl, TII.get(TargetOpcode::CFI_INSTRUCTION))
          .addCFIIndex(CFIIndex);
    }
  }
```

```
  ...
}
```

```
void Cpu0SEFrameLowering::emitEpilogue(MachineFunction &MF,
                                       MachineBasicBlock &MBB) const {
```

```
  if (Cpu0FI->callsEhReturn()) {
    const TargetRegisterClass *RC = &Cpu0::GPROutRegClass;

    // Find first instruction that restores a callee-saved register.
    MachineBasicBlock::iterator I = MBBI;
    for (unsigned i = 0; i < MFI.getCalleeSavedInfo().size(); ++i)
      --I;

    // Insert instructions that restore eh data registers.
    for (int J = 0; J < ABI.EhDataRegSize(); ++J) {
      TII.loadRegFromStackSlot(MBB, I, ABI.GetEhDataReg(J),
                               Cpu0FI->getEhDataRegFI(J), RC, &RegInfo);
    }
  }
```

```
  ...
}
```

```
// This method is called immediately before PrologEpilogInserter scans the
//  physical registers used to determine what callee saved registers should be
//  spilled. This method is optional.
void Cpu0SEFrameLowering::determineCalleeSaves(MachineFunction &MF,
                                               BitVector &SavedRegs,
                                               RegScavenger *RS) const {
```

```
  // Create spill slots for eh data registers if function calls eh_return.
  if (Cpu0FI->callsEhReturn())
    Cpu0FI->createEhDataRegsFI();
```

```
  ...
}
```

lbdex/chapters/Chapter9_3/Cpu0InstrInfo.td

```
// Exception handling related node and instructions.
// The conversion sequence is:
// ISD::EH_RETURN -> Cpu0ISD::EH_RETURN ->
// CPU0eh_return -> (stack change + indirect branch)
//
// CPU0eh_return takes the place of regular return instruction
// but takes two arguments (V1, V0) which are used for storing
// the offset and return address respectively.
def SDT_Cpu0EHRET : SDTypeProfile<0, 2, [SDTCisInt<0>, SDTCisPtrTy<1>]>;

def CPU0ehret : SDNode<"Cpu0ISD::EH_RETURN", SDT_Cpu0EHRET,
                       [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;

let Uses = [V0, V1], isTerminator = 1, isReturn = 1, isBarrier = 1 in {
  def CPU0eh_return32 : Cpu0Pseudo<(outs), (ins GPROut:$spoff, GPROut:$dst), "",
                                   [(CPU0ehret GPROut:$spoff, GPROut:$dst)]>;
}
```

lbdex/chapters/Chapter9_3/Cpu0SEInstrInfo.h

```
  void expandEhReturn(MachineBasicBlock &MBB,
                      MachineBasicBlock::iterator I) const;
```

lbdex/chapters/Chapter9_3/Cpu0SEInstrInfo.cpp

```
/// Expand Pseudo instructions into real backend instructions
bool Cpu0SEInstrInfo::expandPostRAPseudo(MachineInstr &MI) const {
```

```
  case Cpu0::CPU0eh_return32:
    expandEhReturn(MBB, MI);
    break;
```

```
  ...
}
```

```
void Cpu0SEInstrInfo::expandEhReturn(MachineBasicBlock &MBB,
                                     MachineBasicBlock::iterator I) const {
  // This pseudo instruction is generated as part of the lowering of
  // ISD::EH_RETURN. We convert it to a stack increment by OffsetReg, and
  // indirect jump to TargetReg
  unsigned ADDU = Cpu0::ADDu;
  unsigned SP = Cpu0::SP;
  unsigned LR = Cpu0::LR;
  unsigned T9 = Cpu0::T9;
  unsigned ZERO = Cpu0::ZERO;
  unsigned OffsetReg = I->getOperand(0).getReg();
  unsigned TargetReg = I->getOperand(1).getReg();

  // addu $lr, $v0, $zero
  // addu $sp, $sp, $v1
  // jr   $lr (via RetLR)
  const TargetMachine &TM = MBB.getParent()->getTarget();
  if (TM.isPositionIndependent())
    BuildMI(MBB, I, I->getDebugLoc(), get(ADDU), T9)
        .addReg(TargetReg)
        .addReg(ZERO);
  BuildMI(MBB, I, I->getDebugLoc(), get(ADDU), LR)
      .addReg(TargetReg)
      .addReg(ZERO);
  BuildMI(MBB, I, I->getDebugLoc(), get(ADDU), SP).addReg(SP).addReg(OffsetReg);
  expandRetLR(MBB, I);
}
```

lbdex/input/ch9_3_detect_exception.cpp

```
bool exceptionOccur = false;
void* returnAddr;

// Even though __builtin_frame_address is useless in this example, I believe
// it will be used in real exception handler implementation. Because in real
// implementation, the exception handler keeps a table and decide which function
// should be triggered for a specific exception and hand over to it.
// The hand over process needs unwinding the stack frame. The stack frame address
// can be gotten by calling __builtin_frame_address in the charged function.
void exception_handler() {
  exceptionOccur = true;
  int frameaddr = (int)__builtin_frame_address(0);
  __builtin_eh_return(0, returnAddr); // no warning, eh_return never returns.
}

__attribute__ ((weak))
int test_detect_exception(bool exception) {
  exceptionOccur = false;
```

(continues on next page)

(continued from previous page)

```
  void* handler = (void*)(&exception_handler);
  if (exception) {
    returnAddr = __builtin_return_address(0);
    __builtin_eh_return(0, handler); // no warning, eh_return never returns.
  }
  else {
    return 0;
  }
}
```

```
114-37-150-48:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_detect_exception.cpp -emit-llvm -o ch9_3_detect_exception.bc
114-37-150-48:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch9_3_detect_exception.bc -o -
; ModuleID = 'ch9_3_detect_exception.bc'
target datalayout = "E-m:m-p:32:32-i8:8:32-i16:16:32-i64:64-n32-S64"
target triple = "mips-unknown-linux-gnu"

@exceptionOccur = global i8 0, align 1
@returnAddr = global i8* null, align 4

; Function Attrs: nounwind
define void @_Z17exception_handlerv() #0 {
  %frameaddr = alloca i32, align 4
  store i8 1, i8* @exceptionOccur, align 1
  %1 = call i8* @llvm.frameaddress(i32 0)
  %2 = ptrtoint i8* %1 to i32
  store i32 %2, i32* %frameaddr, align 4
  %3 = load i8*, i8** @returnAddr, align 4
  call void @llvm.eh.return.i32(i32 0, i8* %3)
  unreachable
                                                  ; No predecessors!
  ret void
}

; Function Attrs: nounwind readnone
declare i8* @llvm.frameaddress(i32) #1

; Function Attrs: nounwind
declare void @llvm.eh.return.i32(i32, i8*) #2

define weak i32 @_Z21test_detect_exceptionb(i1 zeroext %exception) #3 {
  %1 = alloca i8, align 1
  %handler = alloca i8*, align 4
  %2 = zext i1 %exception to i8
  store i8 %2, i8* %1, align 1
  store i8 0, i8* @exceptionOccur, align 1
  store i8* bitcast (void ()* @_Z17exception_handlerv to i8*), i8** %handler, align 4
  %3 = load i8, i8* %1, align 1
  %4 = trunc i8 %3 to i1
  br i1 %4, label %5, label %8
```

(continues on next page)

(continued from previous page)

```
; <label>:5                                       ; preds = %0
  %6 = call i8* @llvm.returnaddress(i32 0)
  store i8* %6, i8** @returnAddr, align 4
  %7 = load i8*, i8** %handler, align 4
  call void @llvm.eh.return.i32(i32 0, i8* %7)
  unreachable

; <label>:8                                       ; preds = %0
  ret i32 0
}

; Function Attrs: nounwind readnone
declare i8* @llvm.returnaddress(i32) #1

attributes #0 = { nounwind ... }
attributes #1 = { nounwind readnone }
attributes #2 = { nounwind }
attributes #3 = { "less-precise-fpmad"="false" ... }
...

114-37-150-48:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -mcpu=cpu032II -relocation-model=pic -filetype=asm
ch9_3_detect_exception.bc -o -
  .text
  .section .mdebug.abi32
  .previous
  .file "ch9_3_detect_exception.bc"
  .globl  _Z17exception_handlerv
  .align  2
  .type _Z17exception_handlerv,@function
  .ent  _Z17exception_handlerv  # @_Z17exception_handlerv
_Z17exception_handlerv:
  .frame  $fp,16,$lr
  .mask   0x00001000,-4
  .set  noreorder
  .cpload $t9
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -16
  st  $fp, 12($sp)            # 4-byte Folded Spill
  st  $4, 4($fp)
  st  $5, 0($fp)
  move   $fp, $sp
  lui $2, %got_hi(exceptionOccur)
  addu  $2, $2, $gp
  ld  $2, %got_lo(exceptionOccur)($2)
  addiu $3, $zero, 1
  sb  $3, 0($2)
  st  $fp, 8($fp)
  lui $2, %got_hi(returnAddr)
  addu  $2, $2, $gp
```

(continues on next page)

(continued from previous page)

```
  ld  $2, %got_lo(returnAddr)($2)
  ld  $2, 0($2)
  addiu $3, $zero, 0
  move   $sp, $fp
  ld  $4, 4($fp)
  ld  $5, 0($fp)
  ld  $fp, 12($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 16
  move   $t9, $2
  move   $lr, $2
  addu  $sp, $sp, $3
  ret $lr
  nop
  .set  macro
  .set  reorder
  .end  _Z17exception_handlerv
$func_end0:
  .size _Z17exception_handlerv, ($func_end0)-_Z17exception_handlerv

  .weak _Z21test_detect_exceptionb
  .align  2
  .type _Z21test_detect_exceptionb,@function
  .ent  _Z21test_detect_exceptionb # @_Z21test_detect_exceptionb
_Z21test_detect_exceptionb:
  .cfi_startproc
  .frame  $fp,24,$lr
  .mask   0x00001000,-4
  .set  noreorder
  .cpload $t9
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -24
$tmp0:
  .cfi_def_cfa_offset 24
  st  $fp, 20($sp)            # 4-byte Folded Spill
$tmp1:
  .cfi_offset 12, -4
  st  $4, 8($fp)
  st  $5, 4($fp)
$tmp2:
  .cfi_offset 4, -16
$tmp3:
  .cfi_offset 5, -20
  move   $fp, $sp
$tmp4:
  .cfi_def_cfa_register 12
  sb  $4, 16($fp)
  lui $2, %got_hi(exceptionOccur)
  addu  $2, $2, $gp
  ld  $2, %got_lo(exceptionOccur)($2)
  addiu $3, $zero, 0
  sb  $3, 0($2)
```

(continues on next page)

(continued from previous page)

```
  lui $2, %got_hi(_Z17exception_handlerv)
  addu  $2, $2, $gp
  ld  $2, %got_lo(_Z17exception_handlerv)($2)
  st  $2, 12($fp)
  lbu $2, 16($fp)
  andi  $2, $2, 1
  beq $2, $zero, .LBB1_2
  nop
  jmp .LBB1_1
  nop
.LBB1_2:
  addiu $2, $zero, 0
  move   $sp, $fp
  ld  $4, 8($fp)
  ld  $5, 4($fp)
  ld  $fp, 20($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 24
  ret $lr
  nop
.LBB1_1:
  lui $2, %got_hi(returnAddr)
  addu  $2, $2, $gp
  ld  $2, %got_lo(returnAddr)($2)
  st  $lr, 0($2)
  ld  $2, 12($fp)
  addiu $3, $zero, 0
  move   $sp, $fp
  ld  $4, 8($fp)
  ld  $5, 4($fp)
  ld  $fp, 20($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 24
  move   $t9, $2
  move   $lr, $2
  addu  $sp, $sp, $3
  ret $lr
  nop
  .set  macro
  .set  reorder
  .end  _Z21test_detect_exceptionb
$func_end1:
  .size _Z21test_detect_exceptionb, ($func_end1)-_Z21test_detect_exceptionb
  .cfi_endproc

  .type exceptionOccur,@object  # @exceptionOccur
  .bss
  .globl  exceptionOccur
exceptionOccur:
  .byte 0                       # 0x0
  .size exceptionOccur, 1

  .type returnAddr,@object      # @returnAddr
  .globl  returnAddr
```

(continues on next page)

(continued from previous page)

```
  .align  2
returnAddr:
  .4byte  0
  .size returnAddr, 4
  ...
```

If you disable "__attribute__ ((weak))" in the C file, then the IR will has "nounwind" in attributes #3. The side effect in asm output is "No .cfi_offset issued" like function exception_handler().

This example code of exception handler implementation can get frame, return and call exception handler by call __builtin_xxx in clang in C language, without introducing any assembly instruction. And this example can be verified in the Chapter "Cpu0 ELF linker" of the other book "llvm tool chain for Cpu0"[13]. Through examining global variable, exceptionOccur, is true or false, program will set the control flow to exception_handler() or not to correctly.

eh.dwarf intrinsic

Beside lowerADD() in Cpu0ISelLowering, the following code is for the eh.dwarf supporting only, and it can run with input eh-dwarf-cfa.ll as below.

lbdex/chapters/Chapter9_3/Cpu0SEFrameLowering.cpp

```
  // if framepointer enabled, set it to point to the stack pointer.
  if (hasFP(MF)) {
    if (Cpu0FI->callsEhDwarf()) {
      BuildMI(MBB, MBBI, dl, TII.get(ADDu), Cpu0::V0).addReg(FP).addReg(ZERO)
        .setMIFlag(MachineInstr::FrameSetup);
    }
```

```
  ...
}
```

lbdex/input/eh-dwarf-cfa.ll

```
; RUN: llc -march=cpu0el -mcpu=cpu032II < %s | FileCheck %s

declare i8* @llvm.eh.dwarf.cfa(i32) nounwind
declare i8* @llvm.frameaddress(i32) nounwind readnone

define i8* @f1() nounwind {
entry:
  %x = alloca [32 x i8], align 1
  %0 = call i8* @llvm.eh.dwarf.cfa(i32 0)
  ret i8* %0

; CHECK:        addiu   $sp, $sp, -40
; CHECK:        addu    $2,  $zero, $fp
}
```

(continues on next page)

[13] http://jonathan2251.github.io/lbt/lld.html

(continued from previous page)

```
define i8* @f2() nounwind {
entry:
  %x = alloca [65536 x i8], align 1
  %0 = call i8* @llvm.eh.dwarf.cfa(i32 0)
  ret i8* %0

; check stack size (65536 + 8)
; CHECK:        lui     $[[R0:[a-z0-9]+]], 65535
; CHECK:        addiu   $[[R0]], $[[R0]], -8
; CHECK:        addu    $sp, $sp, $[[R0]]

; check return value ($sp + stack size)
; CHECK:        addu    $2,  $zero, $fp
}


define i32 @f3() nounwind {
entry:
  %x = alloca [32 x i8], align 1
  %0 = call i8* @llvm.eh.dwarf.cfa(i32 0)
  %1 = ptrtoint i8* %0 to i32
  %2 = call i8* @llvm.frameaddress(i32 0)
  %3 = ptrtoint i8* %2 to i32
  %add = add i32 %1, %3
  ret i32 %add

; CHECK:        addiu   $sp, $sp, -40

; check return value ($fp + stack size + $fp)
; CHECK:        move     $fp, $sp
; CHECK:        addu    $2, $fp, $fp
}
```

bswap intrinsic

Cpu0 supports llvm instrinsics bswap intrinsic[14].

[14] http://llvm.org/docs/LangRef.html#llvm-bswap-intrinsics

lbdex/chapters/Chapter12_1/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
  setOperationAction(ISD::BSWAP, MVT::i32, Expand);
  setOperationAction(ISD::BSWAP, MVT::i64, Expand);
```

```
  ...
}
```

lbdex/input/ch9_3_bswap.cpp

```
int test_bswap16() {
  volatile int a = 0x1234;
  int result = (__builtin_bswap16(a) ^ 0x3412);

  return result;
}

int test_bswap32() {
  volatile int a = 0x1234;
  int result = (__builtin_bswap32(a) ^ 0x34120000);

  return result;
}

int test_bswap64() {
  volatile int a = 0x1234;
  int result = (__builtin_bswap64(a) ^ 0x3412000000000000);

  return result;
}

int test_bswap() {
  int result = test_bswap16() + test_bswap32() + test_bswap64();

  return result;
}
```

```
114-37-150-48:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3_bswap.cpp -emit-llvm -o ch9_3_bswap.bc
114-37-150-48:input Jonathan$ ~/llvm/test/build/bin/llvm-dis
ch9_3_bswap.bc -o -
...
define i32 @_Z12test_bswap16v() #0 {
  %a = alloca i32, align 4
  %result = alloca i32, align 4
  store volatile i32 4660, i32* %a, align 4
```

(continues on next page)

(continued from previous page)

```
  %1 = load volatile i32, i32* %a, align 4
  %2 = trunc i32 %1 to i16
  %3 = call i16 @llvm.bswap.i16(i16 %2)
  %4 = zext i16 %3 to i32
  %5 = xor i32 %4, 13330
  store i32 %5, i32* %result, align 4
  %6 = load i32, i32* %result, align 4
  ret i32 %6
}
...
```

9.6.6 Add specific backend intrinsic function

LLVM intrinsic functions is designed to extend llvm IRs for hardware acceleration in compiler design[15]. Many cpu implement their intrinsic functions for their speedup hardware instructions. Some gpu apply llvm infrastructure as their OpenGL/CL backend compiler using many llvm extended intrinsic functions. To demonstrate how to use backend proprietary intrinsic functions to support their specific instructions to getting better performance in some domain language, Cpu0 add a intrinsic function @llvm.cpu0.gcd for its gcd(greatest common divider) instruction. This instruction explaining how to do it in llvm only, it is not added in Verilog Cpu0 implementation. The code as follows,

lbdex/llvm/modify/llvm/include/llvm/IR/Intrinsics.td

```
...
include "llvm/IR/IntrinsicsCpu0.td"
...
```

lbdex/llvm/modify/llvm/include/llvm/IR/IntrinsicsCpu0.td

```
//===- IntrinsicsCpu0.td - Defines Mips intrinsics ---------*- tablegen -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file defines all of the CPU0-specific intrinsics.
//
//===----------------------------------------------------------------------===//

// __builtin_cpu0_gcd defined in
// https://github.com/Jonathan2251/lbt/blob/master/exlbt/clang/include/clang/Basic/
↪BuiltinsCpu0.def
def int_cpu0_gcd : GCCBuiltin<"__builtin_cpu0_gcd">,
  Intrinsic<[llvm_i32_ty], [llvm_i32_ty, llvm_i32_ty],
  [Commutative, IntrNoMem]>;
```

[15] https://llvm.org/docs/ExtendingLLVM.html

lbdex/chapters/Chapter9_3/Cpu0InstrInfo.td

```
class IntrinArithLogicR<bits<8> op, string instr_asm, SDPatternOperator OpNode,
                  InstrItinClass itin, RegisterClass RC, bit isComm = 0>:
  FA<op, (outs GPROut:$ra), (ins RC:$rb, RC:$rc),
     !strconcat(instr_asm, "\t$ra, $rb, $rc"),
     [(set GPROut:$ra, (OpNode RC:$rb, RC:$rc))], itin> {
  let shamt = 0;
  let isCommutable = isComm;        // e.g. add rb rc =  add rc rb
  let isReMaterializable = 1;
}
```

```
def GCD : IntrinArithLogicR<0x60, "gcd", int_cpu0_gcd, IIAlu, CPURegs, 1>;
```

When running llc with cpu0_gcd.ll, it gets the gcd machine instruction, meanwhile, when running cpu0_gcd_soft.ll, it gets the "call cpu0_gcd_soft" function. In other words, "@llvm.cpu0.gcd" is intrinsic function for "gcd" machine instruction; "@cpu0_gcd_soft" is ordinary function for hand-written function code.

For those undefined intrinsic functions for Cpu0, such as "fmul float %0, %1". LLVM will compile into function call "jsub fmul" for Cpu0[16].

The test_memcpy.ll is an example for IntrWriteMem which prevent to be optimized out.

9.7 Summary

Now, Cpu0 backend code can take care both the integer function call and control statement just like the example code of llvm frontend tutorial does. It can translate some of the C++ OOP language into Cpu0 instructions also without much effort in backend, because the most complex things in language, such as C++ syntax, is handled by frontend. LLVM is a real structure following the compiler theory, any backend of LLVM can get benefit from this structure. The best part of 3 tiers compiler structure is that backend will grow up automatically in languages support as the frontend supporting languages more and more when the frontend doesn't add any new IR for a new language.

[16] file:///Users/cschen/git/lbd/build/html/othertype.html#float-and-double

CHAPTER

TEN

ELF SUPPORT

- *ELF format*
  - *ELF header and Section header table*
  - *Relocation Record*
  - *Cpu0 ELF related files*
- *llvm-objdump*
  - *llvm-objdump -t -r*
  - *llvm-objdump -d*

Cpu0 backend generated the ELF format of obj. The ELF (Executable and Linkable Format) is a common standard file format for executables, object code, shared libraries and core dumps. First published in the System V Application Binary Interface specification, and later in the Tool Interface Standard, it was quickly accepted among different vendors of Unixsystems. In 1999 it was chosen as the standard binary file format for Unix and Unix-like systems on x86 by the x86open project. Please reference[1].

The binary encode of Cpu0 instruction set in obj has been checked in the previous chapters. But we didn't dig into the ELF file format like elf header and relocation record at that time. You will learn the llvm-objdump, llvm-readelf, ..., tools and understand the ELF file format itself through using these tools to analyze the cpu0 generated obj in this chapter.

This chapter introduces the tool to readers since we think it is a valuable knowledge in this popular ELF format and the ELF binutils analysis tool. An LLVM compiler engineer has the responsibility to make sure that his backend generate a correct obj. With this tool, you can verify your generated ELF format.

The cpu0 author has published a "System Software" book which introduces the topics of assembler, linker, loader, compiler and OS in concept, and at same time demonstrates how to use binutils and gcc to analysis ELF through the example code in his book. It's a Chinese book of "System Software" in concept and practice. This book does the real analysis through binutils. The "System Software"[2] written by Beck is a famous book in concept telling readers what the compiler output about, what the linker output about, what the loader output about, and how they work together in concept. You can reference it to understand how the **"Relocation Record"** works if you need to refresh or learning this knowledge for this chapter.

[3],[4],[5] are the Chinese documents available from the cpu0 author on web site.

[1] http://en.wikipedia.org/wiki/Executable_and_Linkable_Format
[2] Leland Beck, System Software: An Introduction to Systems Programming.
[3] http://ccckmit.wikidot.com/lk:aout
[4] http://ccckmit.wikidot.com/lk:objfile
[5] http://ccckmit.wikidot.com/lk:elffile

10.1 ELF format

ELF is a format used in both obj and executable file. So, there are two views in it as Fig. 10.1.

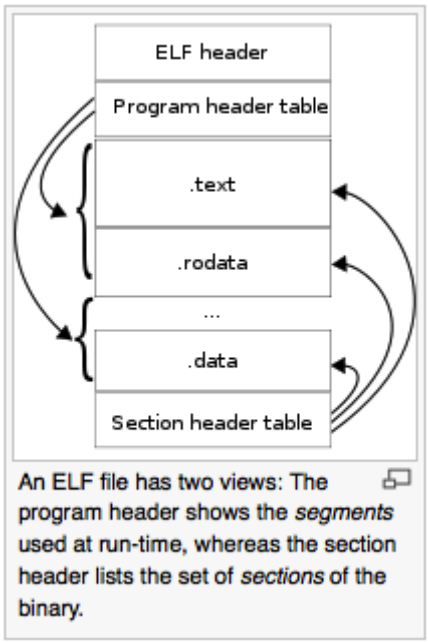


Fig. 10.1: ELF file format overview

As Fig. 10.1, the "Section header table" include sections .text, .rodata, …, .data which are sections layout for code, read only data, …, and read/write data, respectively. "Program header table" include segments for run time code and data. The definition of segments is the run time layout for code and data while sections is the link time layout for code and data.

10.1.1 ELF header and Section header table

Let's run Chapter9_3/ with ch6_1.cpp, and dump ELF header information by `llvm-readelf -h` to see what information the ELF header contains.

```
input$ ~/llvm/test/build/bin/llc -march=cpu0
-relocation-model=pic -filetype=obj ch6_1.bc -o ch6_1.cpu0.o

input$ llvm-readelf -h ch6_1.cpu0.o
  Magic:   7f 45 4c 46 01 02 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           <unknown>: 0xc9
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
```

(continues on next page)

(continued from previous page)

```
  Start of section headers:          176 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         8
  Section header string table index: 5
input$

input$ ~/llvm/test/build/bin/llc
-march=mips -relocation-model=pic -filetype=obj ch6_1.bc -o ch6_1.mips.o

input$ llvm-readelf -h ch6_1.mips.o
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           MIPS R3000
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          200 (bytes into file)
  Flags:                             0x50001007, noreorder, pic, cpic, o32, mips32
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         9
  Section header string table index: 6
input$
```

As above ELF header display, it contains information of magic number, version, ABI, ..., . The Machine field of cpu0 is unknown while mips is known as MIPSR3000. It is unknown because cpu0 is not a popular CPU recognized by utility llvm-readelf. Let's check ELF segments information as follows,

```
input$ llvm-readelf -l ch6_1.cpu0.o

There are no program headers in this file.
input$
```

The result is in expectation because cpu0 obj is for link only, not for execution. So, the segments is empty. Check ELF sections information as follows. Every section contains offset and size information.

```
input$ llvm-readelf -S ch6_1.cpu0.o
There are 10 section headers, starting at offset 0xd4:

Section Headers:
```

(continues on next page)

(continued from previous page)

```
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000034 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000310 000018 08      8   1  4
  [ 3] .data             PROGBITS        00000000 000068 000004 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 00006c 000000 00  WA  0   0  4
  [ 5] .eh_frame         PROGBITS        00000000 00006c 000028 00   A  0   0  4
  [ 6] .rel.eh_frame     REL             00000000 000328 000008 08      8   5  4
  [ 7] .shstrtab         STRTAB          00000000 000094 00003e 00      0   0  1
  [ 8] .symtab           SYMTAB          00000000 000264 000090 10      9   6  4
  [ 9] .strtab           STRTAB          00000000 0002f4 00001b 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
input$
```

10.1.2 Relocation Record

Cpu0 backend translate global variable as follows,

```
input$ clang -target mips-unknown-linux-gnu -c ch6_1.cpp
-emit-llvm -o ch6_1.bc
input$ ~/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch6_1.bc -o ch6_1.cpu0.s
input$ cat ch6_1.cpu0.s
  .section .mdebug.abi32
  .previous
  .file "ch6_1.bc"
  .text
  ...
  .cfi_startproc
  .frame  $sp,8,$lr
  .mask   0x00000000,0
  .set  noreorder
  .cpload $t9
  ...
  lui $2, %got_hi(gI)
  addu $2, $2, $gp
  ld $2, %got_lo(gI)($2)
  ...
  .type gI,@object              # @gI
  .data
  .globl  gI
  .align  2
gI:
  .4byte  100                     # 0x64
  .size gI, 4


input$ ~/llvm/test/build/
```

(continues on next page)

(continued from previous page)

```
bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch6_1.bc -o ch6_1.cpu0.o
input$ llvm-objdump -s ch6_1.cpu0.o

ch6_1.cpu0.o:     file format elf32-big

Contents of section .text:
// .cpload machine instruction
 0000 0fa00000 0daa0000 13aa6000 ........  ..............`.
 ...
 0020 002a0000 00220000 012d0000 0ddd0008  .*..."...-......
 ...
input$

input$ llvm-readelf -tr ch6_1.cpu0.o
There are 8 section headers, starting at offset 0xb0:

Section Headers:
  [Nr] Name
       Type            Addr     Off    Size   ES   Lk Inf Al
       Flags
  [ 0]
       NULL            00000000 000000 000000 00   0   0  0
       [00000000]:
  [ 1] .text
       PROGBITS        00000000 000034 000044 00   0   0  4
       [00000006]: ALLOC, EXEC
  [ 2] .rel.text
       REL             00000000 0002a8 000020 08   6   1  4
       [00000000]:
  [ 3] .data
       PROGBITS        00000000 000078 000008 00   0   0  4
       [00000003]: WRITE, ALLOC
  [ 4] .bss
       NOBITS          00000000 000080 000000 00   0   0  4
       [00000003]: WRITE, ALLOC
  [ 5] .shstrtab
       STRTAB          00000000 000080 000030 00   0   0  1
       [00000000]:
  [ 6] .symtab
       SYMTAB          00000000 0001f0 000090 10   7   5  4
       [00000000]:
  [ 7] .strtab
       STRTAB          00000000 000280 000025 00   0   0  1
       [00000000]:

Relocation section '.rel.text' at offset 0x2a8 contains 4 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000000  00000805 unrecognized: 5       00000000   _gp_disp
00000004  00000806 unrecognized: 6       00000000   _gp_disp
00000020  00000616 unrecognized: 16      00000004   gI
00000028  00000617 unrecognized: 17      00000004   gI
```

(continues on next page)

(continued from previous page)

```
input$ llvm-readelf -tr ch6_1.mips.o
There are 9 section headers, starting at offset 0xc8:

Section Headers:
  [Nr] Name
       Type            Addr     Off    Size   ES   Lk Inf Al
       Flags
  [ 0]
       NULL            00000000 000000 000000 00   0   0  0
       [00000000]:
  [ 1] .text
       PROGBITS        00000000 000034 000038 00   0   0  4
       [00000006]: ALLOC, EXEC
  [ 2] .rel.text
       REL             00000000 0002f8 000018 08   7   1  4
       [00000000]:
  [ 3] .data
       PROGBITS        00000000 00006c 000008 00   0   0  4
       [00000003]: WRITE, ALLOC
  [ 4] .bss
       NOBITS          00000000 000074 000000 00   0   0  4
       [00000003]: WRITE, ALLOC
  [ 5] .reginfo
       MIPS_REGINFO    00000000 000074 000018 00   0   0  1
       [00000002]: ALLOC
  [ 6] .shstrtab
       STRTAB          00000000 00008c 000039 00   0   0  1
       [00000000]:
  [ 7] .symtab
       SYMTAB          00000000 000230 0000a0 10   8   6  4
       [00000000]:
  [ 8] .strtab
       STRTAB          00000000 0002d0 000025 00   0   0  1
       [00000000]:

Relocation section '.rel.text' at offset 0x2f8 contains 3 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000000  00000905 R_MIPS_HI16       00000000   _gp_disp
00000004  00000906 R_MIPS_LO16       00000000   _gp_disp
0000001c  00000709 R_MIPS_GOT16      00000004   gI
```

As depicted in section Handle $gp register in PIC addressing mode, it translates **".cpload %reg"** into the following.

```
// Lower ".cpload $reg" to
//  "lui   $gp, %hi(_gp_disp)"
//  "ori $gp, $gp, %lo(_gp_disp)"
//  "addu  $gp, $gp, $t9"
```

The _gp_disp value is determined by loader. So, it's undefined in obj. You can find both the Relocation Records for offset 0 and 4 of .text section refer to _gp_disp value. The offset 0 and 4 of .text section are instructions "lui $gp, %hi(_gp_disp)" and "ori $gp, $gp, %lo(_gp_disp)" which their corresponding obj encode are 0fa00000 and 0daa0000, respectively. The obj translates the %hi(_gp_disp) and %lo(_gp_disp) into 0 since when loader loads this obj into

memory, loader will know the _gp_disp value at run time and will update these two offset relocation records to the correct offset value. You can check if the cpu0 of %hi(_gp_disp) and %lo(_gp_disp) are correct by above mips Relocation Records of R_MIPS_HI(_gp_disp) and R_MIPS_LO(_gp_disp) even though the cpu0 is not a CPU recognized by llvm-readelf utilitly. The instruction **"ld $2, %got(gI)($gp)"** is same since we don't know what the address of .data section variable will load to. So, Cpu0 translate the address to 0 and made a relocation record on 0x00000020 of .text section. Linker or Loader will change this address when this program is linked or loaded depends on the program is static link or dynamic link.

10.1.3 Cpu0 ELF related files

Files Cpu0ELFObjectWrite.cpp and Cpu0MC*.cpp are the files take care the obj format. Most obj code translation about specific instructions are defined by Cpu0InstrInfo.td and Cpu0RegisterInfo.td. With these td description, LLVM translate Cpu0 instructions into obj format automatically.

10.2 llvm-objdump

10.2.1 llvm-objdump -t -r

`llvm-objdump -tr` can display the information of relocation records like `llvm-readelf -tr`. Let's run llvm-objdump with and without Cpu0 backend commands as follows to see the differences.

```
118-165-83-12:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3.cpp -emit-llvm -o ch9_3.bc
118-165-83-10:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch9_3.bc -o
ch9_3.cpu0.o

118-165-78-12:input Jonathan$ objdump -t -r ch9_3.cpu0.o

ch9_3.cpu0.o:     file format elf32-big

SYMBOL TABLE:
00000000 l    df *ABS*        00000000 ch9_3.bc
00000000 l    d  .text        00000000 .text
00000000 l    d  .data        00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 g     F .text        00000084 _Z5sum_iiz
00000084 g     F .text        00000080 main
00000000         *UND*        00000000 _gp_disp


RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000084 UNKNOWN           _gp_disp
00000088 UNKNOWN           _gp_disp
000000e0 UNKNOWN           _Z5sum_iiz


118-165-83-10:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-objdump -t -r ch9_3.cpu0.o
```

(continues on next page)

(continued from previous page)

```
ch9_3.cpu0.o: file format ELF32-CPU0

RELOCATION RECORDS FOR [.text]:
132 R_CPU0_HI16 _gp_disp
136 R_CPU0_LO16 _gp_disp
224 R_CPU0_CALL16 _Z5sum_iiz

SYMBOL TABLE:
00000000 l    df *ABS*        00000000 ch9_3.bc
00000000 l    d  .text        00000000 .text
00000000 l    d  .data        00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 g     F .text        00000084 _Z5sum_iiz
00000084 g     F .text        00000080 main
00000000         *UND*        00000000 _gp_disp
```

The llvm-objdump can display the file format and relocation records information well while the objdump cannot since we add the relocation records information in ELF.h as follows,

include/llvm/support/ELF.h

```
// Machine architectures
enum {
  ...
  EM_CPU0           = 998, // Document LLVM Backend Tutorial Cpu0
  EM_CPU0_LE        = 999  // EM_CPU0_LE: little endian; EM_CPU0: big endian
}
```

lib/object/ELF.cpp

```
...

StringRef getELFRelocationTypeName(uint32_t Machine, uint32_t Type) {
  switch (Machine) {
  ...
  case ELF::EM_CPU0:
    switch (Type) {
#include "llvm/Support/ELFRelocs/Cpu0.def"
    default:
      break;
    }
    break;
  ...
  }
```

include/llvm/Support/ELFRelocs/Cpu0.def

```
#ifndef ELF_RELOC
#error "ELF_RELOC must be defined"
#endif

ELF_RELOC(R_CPU0_NONE,                0)
ELF_RELOC(R_CPU0_32,                  2)
ELF_RELOC(R_CPU0_HI16,                5)
ELF_RELOC(R_CPU0_LO16,                6)
ELF_RELOC(R_CPU0_GPREL16,             7)
ELF_RELOC(R_CPU0_LITERAL,             8)
ELF_RELOC(R_CPU0_GOT16,               9)
ELF_RELOC(R_CPU0_PC16,               10)
ELF_RELOC(R_CPU0_CALL16,             11)
ELF_RELOC(R_CPU0_GPREL32,            12)
ELF_RELOC(R_CPU0_PC24,               13)
ELF_RELOC(R_CPU0_GOT_HI16,           22)
ELF_RELOC(R_CPU0_GOT_LO16,           23)
ELF_RELOC(R_CPU0_RELGOT,             36)
ELF_RELOC(R_CPU0_TLS_GD,             42)
ELF_RELOC(R_CPU0_TLS_LDM,            43)
ELF_RELOC(R_CPU0_TLS_DTP_HI16,       44)
ELF_RELOC(R_CPU0_TLS_DTP_LO16,       45)
ELF_RELOC(R_CPU0_TLS_GOTTPREL,       46)
ELF_RELOC(R_CPU0_TLS_TPREL32,        47)
ELF_RELOC(R_CPU0_TLS_TP_HI16,        49)
ELF_RELOC(R_CPU0_TLS_TP_LO16,        50)
ELF_RELOC(R_CPU0_GLOB_DAT,           51)
ELF_RELOC(R_CPU0_JUMP_SLOT,          127)
```

include/llvm/Object/ELFObjectFile.h

```
template<support::endianness target_endianness, bool is64Bits>
error_code ELFObjectFile<target_endianness, is64Bits>
            ::getRelocationValueString(DataRefImpl Rel,
                       SmallVectorImpl<char> &Result) const {
  ...
  case ELF::EM_CPU0:  // llvm-objdump -t -r
  res = symname;
  break;
  ...
}

template<support::endianness target_endianness, bool is64Bits>
StringRef ELFObjectFile<target_endianness, is64Bits>
             ::getFileFormatName() const {
  switch(Header->e_ident[ELF::EI_CLASS]) {
  case ELF::ELFCLASS32:
  switch(Header->e_machine) {
```

(continues on next page)

(continued from previous page)

```
  ...
  case ELF::EM_CPU0:  // llvm-objdump -t -r
    return "ELF32-CPU0";
  ...
}

template<support::endianness target_endianness, bool is64Bits>
unsigned ELFObjectFile<target_endianness, is64Bits>::getArch() const {
  switch(Header->e_machine) {
  ...
  case ELF::EM_CPU0:  // llvm-objdump -t -r
  return (target_endianness == support::little) ?
       Triple::cpu0el : Triple::cpu0;
  ...
}
```

In addition to `llvm-objdump -t -r`, the `llvm-readobj -h` can display the Cpu0 elf header information with EM_CPU0 defined above.

10.2.2 llvm-objdump -d

Run the last Chapter example code with command `llvm-objdump -d` for dumping file from elf to hex as follows,

```
JonathantekiiMac:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch8_1_1.cpp -emit-llvm -o ch8_1_1.bc
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch8_1_1.bc
-o ch8_1_1.cpu0.o
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-objdump -d ch8_1_1.cpu0.o

ch8_1_1.cpu0.o: file format ELF32-unknown

Disassembly of section .text:error: no disassembler for target cpu0-unknown-
unknown
```

To support llvm-objdump, the following code added to Chapter10_1/ (the DecoderMethod for brtarget24 has been added in previous chapter).

lbdex/chapters/Chapter10_1/CMakeLists.txt

```
tablegen(LLVM Cpu0GenDisassemblerTables.inc -gen-disassembler)
```

```
  Cpu0Disassembler
```

```
add_subdirectory(Disassembler)
```

lbdex/chapters/Chapter10_1/Cpu0InstrInfo.td

```
let isBranch=1, isTerminator=1, isBarrier=1, imm16=0, hasDelaySlot = 1,
    isIndirectBranch = 1 in
class JumpFR<bits<8> op, string instr_asm, RegisterClass RC>:
  FL<op, (outs), (ins RC:$ra),
     !strconcat(instr_asm, "\t$ra"), [(brind RC:$ra)], IIBranch> {
  let rb = 0;
  let imm16 = 0;
//#if CH >= CH10_1 1.5
  let DecoderMethod = "DecodeJumpFR";
//#endif
}
```

```
  class JumpLink<bits<8> op, string instr_asm>:
    FJ<op, (outs), (ins calltarget:$target, variable_ops),
       !strconcat(instr_asm, "\t$target"), [(Cpu0JmpLink imm:$target)],
       IIBranch> {
//#if CH >= CH10_1 2
       let DecoderMethod = "DecodeJumpTarget";
//#endif
       }
```

lbdex/chapters/Chapter10_1/Disassembler/CMakeLists.txt

```
add_llvm_component_library(LLVMCpu0Disassembler
  Cpu0Disassembler.cpp

  LINK_COMPONENTS
  MCDisassembler
  Cpu0Info
  Support

  ADD_TO_COMPONENT
  Cpu0
  )
```

lbdex/chapters/Chapter10_1/Disassembler/Cpu0Disassembler.cpp

```
//===- Cpu0Disassembler.cpp - Disassembler for Cpu0 -------------*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file is part of the Cpu0 Disassembler.
//
```

(continues on next page)

(continued from previous page)

```
//===----------------------------------------------------------------------===//

#include "Cpu0.h"

#include "Cpu0RegisterInfo.h"
#include "Cpu0Subtarget.h"
#include "llvm/MC/MCDisassembler/MCDisassembler.h"
#include "llvm/MC/MCFixedLenDisassembler.h"
#include "llvm/MC/MCInst.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/Support/MathExtras.h"
#include "llvm/Support/TargetRegistry.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-disassembler"

typedef MCDisassembler::DecodeStatus DecodeStatus;

namespace {

/// Cpu0DisassemblerBase - a disasembler class for Cpu0.
class Cpu0DisassemblerBase : public MCDisassembler {
public:
  /// Constructor     - Initializes the disassembler.
  ///
  Cpu0DisassemblerBase(const MCSubtargetInfo &STI, MCContext &Ctx,
                       bool bigEndian) :
    MCDisassembler(STI, Ctx),
    IsBigEndian(bigEndian) {}

  virtual ~Cpu0DisassemblerBase() {}

protected:
  bool IsBigEndian;
};

/// Cpu0Disassembler - a disasembler class for Cpu032.
class Cpu0Disassembler : public Cpu0DisassemblerBase {
public:
  /// Constructor     - Initializes the disassembler.
  ///
  Cpu0Disassembler(const MCSubtargetInfo &STI, MCContext &Ctx, bool bigEndian)
      : Cpu0DisassemblerBase(STI, Ctx, bigEndian) {
  }

  /// getInstruction - See MCDisassembler.
  DecodeStatus getInstruction(MCInst &Instr, uint64_t &Size,
                              ArrayRef<uint8_t> Bytes, uint64_t Address,
                              raw_ostream &CStream) const override;
};
```

(continues on next page)

(continued from previous page)

```
} // end anonymous namespace

// Decoder tables for GPR register
static const unsigned CPURegsTable[] = {
  Cpu0::ZERO, Cpu0::AT, Cpu0::V0, Cpu0::V1,
  Cpu0::A0, Cpu0::A1, Cpu0::T9, Cpu0::T0,
  Cpu0::T1, Cpu0::S0, Cpu0::S1, Cpu0::GP,
  Cpu0::FP, Cpu0::SP, Cpu0::LR, Cpu0::SW
};

// Decoder tables for co-processor 0 register
static const unsigned C0RegsTable[] = {
  Cpu0::PC, Cpu0::EPC
};

static DecodeStatus DecodeCPURegsRegisterClass(MCInst &Inst,
                                               unsigned RegNo,
                                               uint64_t Address,
                                               const void *Decoder);
static DecodeStatus DecodeGPROutRegisterClass(MCInst &Inst,
                                              unsigned RegNo,
                                              uint64_t Address,
                                              const void *Decoder);
static DecodeStatus DecodeSRRegisterClass(MCInst &Inst,
                                          unsigned RegNo,
                                          uint64_t Address,
                                          const void *Decoder);
static DecodeStatus DecodeC0RegsRegisterClass(MCInst &Inst,
                                              unsigned RegNo,
                                              uint64_t Address,
                                              const void *Decoder);
static DecodeStatus DecodeBranch16Target(MCInst &Inst,
                                         unsigned Insn,
                                         uint64_t Address,
                                         const void *Decoder);
static DecodeStatus DecodeBranch24Target(MCInst &Inst,
                                         unsigned Insn,
                                         uint64_t Address,
                                         const void *Decoder);
static DecodeStatus DecodeJumpTarget(MCInst &Inst,
                                     unsigned Insn,
                                     uint64_t Address,
                                     const void *Decoder);
static DecodeStatus DecodeJumpFR(MCInst &Inst,
                                 unsigned Insn,
                                 uint64_t Address,
                                 const void *Decoder);

static DecodeStatus DecodeMem(MCInst &Inst,
                              unsigned Insn,
                              uint64_t Address,
                              const void *Decoder);
```

(continues on next page)

(continued from previous page)

```
static DecodeStatus DecodeSimm16(MCInst &Inst,
                                 unsigned Insn,
                                 uint64_t Address,
                                 const void *Decoder);

namespace llvm {
extern Target TheCpu0elTarget, TheCpu0Target, TheCpu064Target,
              TheCpu064elTarget;
}

static MCDisassembler *createCpu0Disassembler(
                       const Target &T,
                       const MCSubtargetInfo &STI,
                       MCContext &Ctx) {
  return new Cpu0Disassembler(STI, Ctx, true);
}

static MCDisassembler *createCpu0elDisassembler(
                       const Target &T,
                       const MCSubtargetInfo &STI,
                       MCContext &Ctx) {
  return new Cpu0Disassembler(STI, Ctx, false);
}

extern "C" void LLVMInitializeCpu0Disassembler() {
  // Register the disassembler.
  TargetRegistry::RegisterMCDisassembler(TheCpu0Target,
                                         createCpu0Disassembler);
  TargetRegistry::RegisterMCDisassembler(TheCpu0elTarget,
                                         createCpu0elDisassembler);
}

#include "Cpu0GenDisassemblerTables.inc"

/// Read four bytes from the ArrayRef and return 32 bit word sorted
/// according to the given endianess
static DecodeStatus readInstruction32(ArrayRef<uint8_t> Bytes, uint64_t Address,
                                      uint64_t &Size, uint32_t &Insn,
                                      bool IsBigEndian) {
  // We want to read exactly 4 Bytes of data.
  if (Bytes.size() < 4) {
    Size = 0;
    return MCDisassembler::Fail;
  }

  if (IsBigEndian) {
    // Encoded as a big-endian 32-bit word in the stream.
    Insn = (Bytes[3] <<  0) |
           (Bytes[2] <<  8) |
           (Bytes[1] << 16) |
           (Bytes[0] << 24);
  }
```

(continues on next page)

(continued from previous page)

```
  else {
    // Encoded as a small-endian 32-bit word in the stream.
    Insn = (Bytes[0] <<  0) |
           (Bytes[1] <<  8) |
           (Bytes[2] << 16) |
           (Bytes[3] << 24);
  }

  return MCDisassembler::Success;
}

DecodeStatus
Cpu0Disassembler::getInstruction(MCInst &Instr, uint64_t &Size,
                                                ArrayRef<uint8_t> Bytes,
                                                uint64_t Address,
                                                raw_ostream &CStream) const {
  uint32_t Insn;

  DecodeStatus Result;

  Result = readInstruction32(Bytes, Address, Size, Insn, IsBigEndian);

  if (Result == MCDisassembler::Fail)
    return MCDisassembler::Fail;

  // Calling the auto-generated decoder function.
  Result = decodeInstruction(DecoderTableCpu032, Instr, Insn, Address,
                             this, STI);
  if (Result != MCDisassembler::Fail) {
    Size = 4;
    return Result;
  }

  return MCDisassembler::Fail;
}

static DecodeStatus DecodeCPURegsRegisterClass(MCInst &Inst,
                                               unsigned RegNo,
                                               uint64_t Address,
                                               const void *Decoder) {
  if (RegNo > 15)
    return MCDisassembler::Fail;

  Inst.addOperand(MCOperand::createReg(CPURegsTable[RegNo]));
  return MCDisassembler::Success;
}

static DecodeStatus DecodeGPROutRegisterClass(MCInst &Inst,
                                              unsigned RegNo,
                                              uint64_t Address,
                                              const void *Decoder) {
  return DecodeCPURegsRegisterClass(Inst, RegNo, Address, Decoder);
```

(continues on next page)

(continued from previous page)

```
}

static DecodeStatus DecodeSRRegisterClass(MCInst &Inst,
                                          unsigned RegNo,
                                          uint64_t Address,
                                          const void *Decoder) {
  return DecodeCPURegsRegisterClass(Inst, RegNo, Address, Decoder);
}

static DecodeStatus DecodeC0RegsRegisterClass(MCInst &Inst,
                                              unsigned RegNo,
                                              uint64_t Address,
                                              const void *Decoder) {
  if (RegNo > 1)
    return MCDisassembler::Fail;

  Inst.addOperand(MCOperand::createReg(C0RegsTable[RegNo]));
  return MCDisassembler::Success;
}

//@DecodeMem {
static DecodeStatus DecodeMem(MCInst &Inst,
                              unsigned Insn,
                              uint64_t Address,
                              const void *Decoder) {
//@DecodeMem body {
  int Offset = SignExtend32<16>(Insn & 0xffff);
  int Reg = (int)fieldFromInstruction(Insn, 20, 4);
  int Base = (int)fieldFromInstruction(Insn, 16, 4);

  Inst.addOperand(MCOperand::createReg(CPURegsTable[Reg]));
  Inst.addOperand(MCOperand::createReg(CPURegsTable[Base]));
  Inst.addOperand(MCOperand::createImm(Offset));

  return MCDisassembler::Success;
}

static DecodeStatus DecodeBranch16Target(MCInst &Inst,
                                         unsigned Insn,
                                         uint64_t Address,
                                         const void *Decoder) {
  int BranchOffset = fieldFromInstruction(Insn, 0, 16);
  if (BranchOffset > 0x8fff)
          BranchOffset = -1*(0x10000 - BranchOffset);
  Inst.addOperand(MCOperand::createImm(BranchOffset));
  return MCDisassembler::Success;
}

/* CBranch instruction define $ra and then imm24; The printOperand() print
operand 1 (operand 0 is $ra and operand 1 is imm24), so we Create register
operand first and create imm24 next, as follows,
```

(continues on next page)

(continued from previous page)

```
// Cpu0InstrInfo.td
class CBranch<bits<8> op, string instr_asm, RegisterClass RC,
                  list<Register> UseRegs>:
  FJ<op, (outs), (ins RC:$ra, brtarget:$addr),
             !strconcat(instr_asm, "\t$addr"),
             [(brcond RC:$ra, bb:$addr)], IIBranch> {

// Cpu0AsmWriter.inc
void Cpu0InstPrinter::printInstruction(const MCInst *MI, raw_ostream &O) {
...
  case 3:
    // CMP, JEQ, JGE, JGT, JLE, JLT, JNE
    printOperand(MI, 1, O);
    break;
*/
static DecodeStatus DecodeBranch24Target(MCInst &Inst,
                                         unsigned Insn,
                                         uint64_t Address,
                                         const void *Decoder) {
  int BranchOffset = fieldFromInstruction(Insn, 0, 24);
  if (BranchOffset > 0x8fffff)
          BranchOffset = -1*(0x1000000 - BranchOffset);
  Inst.addOperand(MCOperand::createReg(Cpu0::SW));
  Inst.addOperand(MCOperand::createImm(BranchOffset));
  return MCDisassembler::Success;
}

static DecodeStatus DecodeJumpTarget(MCInst &Inst,
                                     unsigned Insn,
                                     uint64_t Address,
                                     const void *Decoder) {

  unsigned JumpOffset = fieldFromInstruction(Insn, 0, 24);
  Inst.addOperand(MCOperand::createImm(JumpOffset));
  return MCDisassembler::Success;
}

static DecodeStatus DecodeJumpFR(MCInst &Inst,
                                     unsigned Insn,
                                     uint64_t Address,
                                     const void *Decoder) {
  int Reg_a = (int)fieldFromInstruction(Insn, 20, 4);
  Inst.addOperand(MCOperand::createReg(CPURegsTable[Reg_a]));
// exapin in http://jonathan2251.github.io/lbd/llvmstructure.html#jr-note
  if (CPURegsTable[Reg_a] == Cpu0::LR)
    Inst.setOpcode(Cpu0::RET);
  else
    Inst.setOpcode(Cpu0::JR);
  return MCDisassembler::Success;
}

static DecodeStatus DecodeSimm16(MCInst &Inst,
```

(continues on next page)

(continued from previous page)

```
                                unsigned Insn,
                                uint64_t Address,
                                const void *Decoder) {
  Inst.addOperand(MCOperand::createImm(SignExtend32<16>(Insn)));
  return MCDisassembler::Success;
}
```

As above code, it adds directory Disassembler to handle the reverse translation from obj to assembly. So, add Disassembler/Cpu0Disassembler.cpp and modify the CMakeList.txt to build directory Disassembler, and enable the disassembler table generated by "has_disassembler = 1". Most of code is handled by the table defined in *.td files. Not every instruction in *.td can be disassembled without trouble even though they can be translated into assembly and obj successfully. For those cannot be disassembled, LLVM supply the **"let DecoderMethod"** keyword to allow programmers implement their decode function. For example in Cpu0, we define functions DecodeBranch24Target(), DecodeJumpTarget() and DecodeJumpFR() in Cpu0Disassembler.cpp and tell the llvm-tblgen by writing **"let DecoderMethod = ..."** in the corresponding instruction definitions or ISD node of Cpu0InstrInfo.td. LLVM will call these DecodeMethod when user uses Disassembler tools, such as `llvm-objdump -d`.

Finally cpu032II includes all cpu032I instruction set and adds some instrucitons. When `llvm-objdump -d` is invoked, function selectCpu0ArchFeature() as the following will be called through createCpu0MCSubtargetInfo(). The llvm-objdump cannot set cpu option like llc as `llc -mcpu=cpu032I`, so the varaible CPU in selectCpu0ArchFeature() is empty when invoked by `llvm-objdump -d`. Set Cpu0ArchFeature to "+cpu032II" than it can disassemble all instructions (cpu032II include all cpu032I instructions and add some new instructions).

lbdex/chapters/Chapter10_1/MCTargetDesc/Cpu0MCTargetDesc.cpp

```
/// Select the Cpu0 Architecture Feature for the given triple and cpu name.
/// The function will be called at command 'llvm-objdump -d' for Cpu0 elf input.
static std::string selectCpu0ArchFeature(const Triple &TT, StringRef CPU) {
  std::string Cpu0ArchFeature;
  if (CPU.empty() || CPU == "generic") {
    if (TT.getArch() == Triple::cpu0 || TT.getArch() == Triple::cpu0el) {
      if (CPU.empty() || CPU == "cpu032II") {
        Cpu0ArchFeature = "+cpu032II";
      }
      else {
        if (CPU == "cpu032I") {
          Cpu0ArchFeature = "+cpu032I";
        }
      }
    }
  }
  return Cpu0ArchFeature;
}
```

Now, run Chapter10_1/ with command `llvm-objdump -d ch8_1_1.cpu0.o` will get the following result.

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=obj
ch8_1_1.bc -o ch8_1_1.cpu0.o
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-objdump -d ch8_1_1.cpu0.o
```

(continues on next page)

(continued from previous page)

```
ch8_1_1.cpu0.o:       file format ELF32-CPU0

Disassembly of section .text:
_Z13test_control1v:
       0: 09 dd ff d8                                   addiu $sp, $sp, -40
       4: 09 30 00 00                                   addiu $3, $zero, 0
       8: 02 3d 00 24                                   st  $3, 36($sp)
       c: 09 20 00 01                                   addiu $2, $zero, 1
      10: 02 2d 00 20                                   st  $2, 32($sp)
      14: 09 40 00 02                                   addiu $4, $zero, 2
      18: 02 4d 00 1c                                   st  $4, 28($sp)
      ...
```

CHAPTER

ELEVEN

ASSEMBLER

- *AsmParser support*
  - *Code structure explanation*
  - *Code list and some detail functions explanation*
- *Inline assembly*

This chapter include the assembly programming support in Cpu0 backend.

When it comes to assembly language programming, there are two type of writting in C/C++ as follows,

ordinary assembly

```
asm("ld       $2, 8($sp)");
```

inline assembly

```
int foo = 10;
const int bar = 15;

__asm__ __volatile__("addu %0,%1,%2"
                     :"=r"(foo) // 5
                     :"r"(foo), "r"(bar)
                     );
```

In llvm, the first is supported by LLVM AsmParser, and the second is inline assembly handler. With AsmParser and inline assembly support in Cpu0 backend, we can hand-code the assembly language in C/C++ file and translate it into obj (elf format).

11.1 AsmParser support

This section lists all the AsmParser code for Cpu0 backend with only a few explanation. Please refer here[1] for more AsmParser explanation.

Run Chapter10_1/ with ch11_1.cpp will get the following error message.

lbdex/input/ch11_1.cpp

```
asm("ld        $2, 8($sp)");
asm("st        $0, 4($sp)");
asm("addiu $3,        $ZERO, 0");
asm("add $v1, $at, $v0");
asm("sub $3, $2, $3");
asm("mul $2, $1, $3");
asm("div $3, $2");
asm("divu $2, $3");
asm("and $2, $1, $3");
asm("or $3, $1, $2");
asm("xor $1, $2, $3");
asm("mult $4, $3");
asm("multu $3, $2");
asm("mfhi $3");
asm("mflo $2");
asm("mthi $2");
asm("mtlo $2");
asm("sra $2, $2, 2");
asm("rol $2, $1, 3");
asm("ror $3, $3, 4");
asm("shl $2, $2, 2");
asm("shr $2, $3, 5");
asm("cmp $sw, $2, $3");
asm("jeq $sw, 20");
asm("jne $sw, 16");
asm("jlt $sw, -20");
asm("jle $sw, -16");
asm("jgt $sw, -4");
asm("jge $sw, -12");
asm("jsub 0x000010000");
asm("jr $4");
asm("ret $lr");
asm("jalr $t9");
asm("li $3, 0x00700000");
asm("la $3, 0x00800000($6)");
asm("la $3, 0x00900000");
```

```
JonathantekiiMac:input Jonathan$ clang -c ch11_1.cpp -emit-llvm -o
ch11_1.bc
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=pic -filetype=obj ch11_1.bc
```

(continues on next page)

[1] http://www.embecosm.com/appnotes/ean10/ean10-howto-llvmas-1.0.html

(continued from previous page)

```
-o ch11_1.cpu0.o
LLVM ERROR: Inline asm not supported by this streamer because we don't have
an asm parser for this target
```

Since we havn't implemented Cpu0 assembler, it has the error message as above. The Cpu0 can translate LLVM IR into assembly and obj directly, but it cannot translate hand-code assembly instructions into obj.

11.1.1 Code structure explanation

Directory AsmParser handle the assembly to obj translation. The assembling Data Flow Diagram (DFD) as Fig. 11.1 and Fig. 11.2.

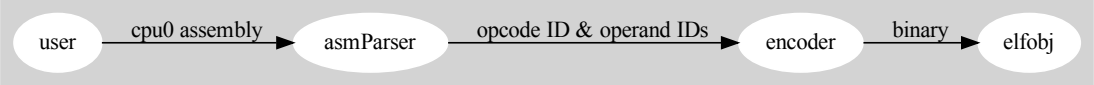


Fig. 11.1: Assemble flow

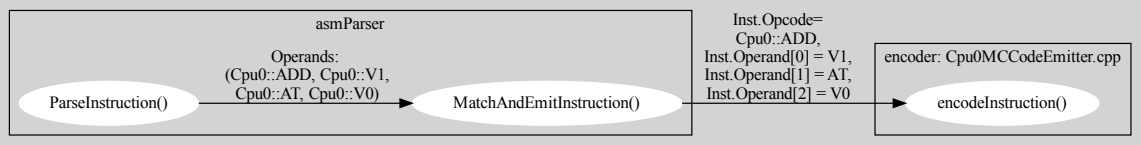


Fig. 11.2: Assemble flow, for instance: add $v1, $v0, $at

Given an example of assembly instruction "add $v1, $v0, $at", llvm AsmParser kernel call backend ParseInstruction() of Cpu0AsmParser.cpp when it parses and recognises that the first token at the beginning of line is identifier. ParseInstruction() parses one assembly instruction, creates Operands and return to llvm AsmParser. Then AsmParser calls backend MatchAndEmitInstruction() to set Opcode and Operands to MCInst, then encoder can encode binary instruction from MCInst with the information come from Cpu0InstrInfo.td which includes binary value for Opcode ID and Operand IDs of the instruction.

List the key functions and data structure of MatchAndEmitInstruction() and encodeInstruction(), explaining in comments which begin with ///.

llvm/build/lib/Target/Cpu0/Cpu0GenAsmMatcher.inc

```
enum InstructionConversionKind {
  Convert__Reg1_0__Reg1_1__Reg1_2,
  Convert__Reg1_0__Reg1_1__Imm1_2,
  ...
  CVT_NUM_SIGNATURES
};

} // end anonymous namespace

  struct MatchEntry {
    uint16_t Mnemonic;
    uint16_t Opcode;
    uint8_t ConvertFn;
    uint32_t RequiredFeatures;
    uint8_t Classes[3];
    StringRef getMnemonic() const {
      return StringRef(MnemonicTable + Mnemonic + 1,
                       MnemonicTable[Mnemonic]);
    }
  };

static const MatchEntry MatchTable0[] = {
  { 0 /* add */, Cpu0::ADD, Convert__Reg1_0__Reg1_1__Reg1_2, 0, { MCK_CPURegs, MCK_
↪CPURegs, MCK_CPURegs }, },
  { 4 /* addiu */, Cpu0::ADDiu, Convert__Reg1_0__Reg1_1__Imm1_2, 0, { MCK_CPURegs, MCK_
↪CPURegs, MCK_Imm }, },
  ...
};

unsigned Cpu0AsmParser::
MatchInstructionImpl(const OperandVector &Operands,
                     MCInst &Inst, uint64_t &ErrorInfo,
                     bool matchingInlineAsm, unsigned VariantID) {
  ...
  // Find the appropriate table for this asm variant.
  const MatchEntry *Start, *End;
  switch (VariantID) {
  default: llvm_unreachable("invalid variant!");
  case 0: Start = std::begin(MatchTable0); End = std::end(MatchTable0); break;
  }
  // Search the table.
  auto MnemonicRange = std::equal_range(Start, End, Mnemonic, LessOpcode());
  ...
  for (const MatchEntry *it = MnemonicRange.first, *ie = MnemonicRange.second;
       it != ie; ++it) {
    ...
    // We have selected a definite instruction, convert the parsed
    // operands into the appropriate MCInst.

    /// For instance ADD , V1, AT, V0
    /// MnemonicRange.first = &MatchTable0[0]
```

(continues on next page)

(continued from previous page)

```
    /// MnemonicRange.second = &MatchTable0[1]
    /// it.ConvertFn = Convert__Reg1_0__Reg1_1__Reg1_2

    convertToMCInst(it->ConvertFn, Inst, it->Opcode, Operands);
    ...
  }
  ...
}

static const uint8_t ConversionTable[CVT_NUM_SIGNATURES][9] = {
  // Convert__Reg1_0__Reg1_1__Reg1_2
  { CVT_95_Reg, 1, CVT_95_Reg, 2, CVT_95_Reg, 3, CVT_Done },
  // Convert__Reg1_0__Reg1_1__Imm1_2
  { CVT_95_Reg, 1, CVT_95_Reg, 2, CVT_95_addImmOperands, 3, CVT_Done },
  ...
};

/// When kind = Convert__Reg1_0__Reg1_1__Reg1_2, ConversionTable[Kind] is equal to CVT_
↪95_Reg
/// For Operands[1], Operands[2], Operands[3] do the following:
///   static_cast<Cpu0Operand&>(*Operands[OpIdx]).addRegOperands(Inst, 1);
/// Since p = 0, 2, 4, then OpIdx = 1, 2, 3 when OpIdx=*(p+1).
/// Since, Operands[1] = V1, Operands[2] = AT, Operands[3] = V0,
///   for "ADD , V1, AT, V0" which created by ParseInstruction().
/// Inst.Opcode = ADD, Inst.Operand[0] = V1, Inst.Operand[1] = AT,
///   Inst.Operand[2] = V0.
void Cpu0AsmParser::
convertToMCInst(unsigned Kind, MCInst &Inst, unsigned Opcode,
                const OperandVector &Operands) {
  assert(Kind < CVT_NUM_SIGNATURES && "Invalid signature!");
  const uint8_t *Converter = ConversionTable[Kind];
  unsigned OpIdx;
  Inst.setOpcode(Opcode);
  for (const uint8_t *p = Converter; *p; p+= 2) {
    OpIdx = *(p + 1);
    switch (*p) {
    default: llvm_unreachable("invalid conversion entry!");
    case CVT_Reg:
      static_cast<Cpu0Operand&>(*Operands[OpIdx]).addRegOperands(Inst, 1);
      break;
    ...
    }
  }
}
```

lbdex/chapters/Chapter11_1/AsmParser/Cpu0AsmParser.cpp

```
/// For "ADD , V1, AT, V0", ParseInstruction() set Operands[1].Reg.RegNum = V1,
///   Operands[2].Reg.RegNum = AT, ..., by Cpu0Operand::CreateReg(RegNo, S,
///   Parser.getTok().getLoc()) in calling ParseOperand().
/// So, after (*Operands[1..3]).addRegOperands(Inst, 1),
///   Inst.Opcode = ADD, Inst.Operand[0] = V1, Inst.Operand[1] = AT,
///   Inst.Operand[2] = V0.
class Cpu0Operand : public MCParsedAsmOperand {
  ...
  void addRegOperands(MCInst &Inst, unsigned N) const {
    assert(N == 1 && "Invalid number of operands!");
    Inst.addOperand(MCOperand::createReg(getReg()));
  }
  ...
  unsigned getReg() const override {
    assert((Kind == k_Register) && "Invalid access!");
    return Reg.RegNum;
  }
  ...
}
```

lbdex/chapters/Chapter11_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
void Cpu0MCCodeEmitter::
encodeInstruction(const MCInst &MI, raw_ostream &OS,
                  SmallVectorImpl<MCFixup> &Fixups,
                  const MCSubtargetInfo &STI) const
{
  uint32_t Binary = getBinaryCodeForInstr(MI, Fixups, STI);
  ...
  EmitInstruction(Binary, Size, OS);
}
```

llvm/build/lib/Target/Cpu0/Cpu0GenMCCodeEmitter.inc

```
uint64_t Cpu0MCCodeEmitter::getBinaryCodeForInstr(const MCInst &MI,
    SmallVectorImpl<MCFixup> &Fixups,
    const MCSubtargetInfo &STI) const {
  static const uint64_t InstBits[] = {
    ...
    UINT64_C(318767104),      // ADD  /// 318767104=0x13000000
    ...
  };
  ...

  const unsigned opcode = MI.getOpcode();
  ...
  switch (opcode) {
    case Cpu0::ADD:
```

(continues on next page)

(continued from previous page)

```
    ...
      // op: ra
      op = getMachineOpValue(MI, MI.getOperand(0), Fixups, STI);
      Value |= (op & UINT64_C(15)) << 20;
      // op: rb
      op = getMachineOpValue(MI, MI.getOperand(1), Fixups, STI);
      Value |= (op & UINT64_C(15)) << 16;
      // op: rc
      op = getMachineOpValue(MI, MI.getOperand(2), Fixups, STI);
      Value |= (op & UINT64_C(15)) << 12;
      break;
    }
    ...
  }
  return Value;
}
```

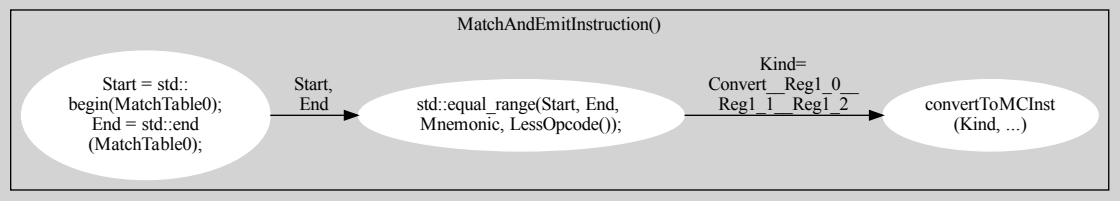


Fig. 11.3: Data flow in MatchAndEmitInstruction(), for instance: add $v1, $v0, $at"

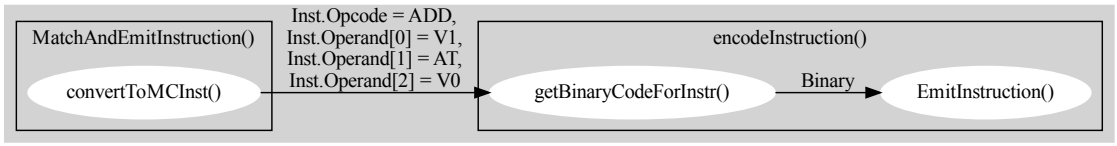


Fig. 11.4: Data flow in and between MatchAndEmitInstruction() and encodeInstruction(), for instance: add $v1, $v0, $at

MatchTable0 include all the possibile combinations of opcode and operands type. Even the assembly instruction of user input may pass Cpu0AsmParser in syntax check, the MatchAndEmitInstruction() still can be fail. For example, instruction "asm("move $3, $2);" can pass but "asm("move $3, $2, $1");" will fail.

11.1.2 Code list and some detail functions explanation

The Chapter11_1/ include AsmParser implementation as follows,

lbdex/chapters/Chapter11_1/AsmParser/Cpu0AsmParser.cpp

```
//===-- Cpu0AsmParser.cpp - Parse Cpu0 assembly to MCInst instructions ----===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

#include "Cpu0.h"
#if CH >= CH11_1

#include "MCTargetDesc/Cpu0MCExpr.h"
#include "MCTargetDesc/Cpu0MCTargetDesc.h"
#include "Cpu0RegisterInfo.h"
#include "llvm/ADT/APInt.h"
#include "llvm/ADT/StringSwitch.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCExpr.h"
#include "llvm/MC/MCInst.h"
#include "llvm/MC/MCInstBuilder.h"
#include "llvm/MC/MCParser/MCAsmLexer.h"
#include "llvm/MC/MCParser/MCParsedAsmOperand.h"
#include "llvm/MC/MCParser/MCTargetAsmParser.h"
#include "llvm/MC/MCStreamer.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/MC/MCSymbol.h"
#include "llvm/MC/MCParser/MCAsmLexer.h"
#include "llvm/MC/MCParser/MCParsedAsmOperand.h"
#include "llvm/MC/MCValue.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/MathExtras.h"
#include "llvm/Support/TargetRegistry.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-asm-parser"

namespace {
class Cpu0AssemblerOptions {
public:
  Cpu0AssemblerOptions():
    reorder(true), macro(true) {
  }

  bool isReorder() {return reorder;}
```

(continues on next page)

(continued from previous page)

```
  void setReorder() {reorder = true;}
  void setNoreorder() {reorder = false;}

  bool isMacro() {return macro;}
  void setMacro() {macro = true;}
  void setNomacro() {macro = false;}

private:
  bool reorder;
  bool macro;
};
}

namespace {
class Cpu0AsmParser : public MCTargetAsmParser {
  MCAsmParser &Parser;
  Cpu0AssemblerOptions Options;


#define GET_ASSEMBLER_HEADER
#include "Cpu0GenAsmMatcher.inc"

  bool MatchAndEmitInstruction(SMLoc IDLoc, unsigned &Opcode,
                               OperandVector &Operands, MCStreamer &Out,
                               uint64_t &ErrorInfo,
                               bool MatchingInlineAsm) override;

  bool ParseRegister(unsigned &RegNo, SMLoc &StartLoc, SMLoc &EndLoc) override;

  OperandMatchResultTy tryParseRegister(unsigned &RegNo, SMLoc &StartLoc,
                                        SMLoc &EndLoc) override;

  bool ParseInstruction(ParseInstructionInfo &Info, StringRef Name,
                        SMLoc NameLoc, OperandVector &Operands) override;

  bool ParseDirective(AsmToken DirectiveID) override;

  OperandMatchResultTy parseMemOperand(OperandVector &);

  bool ParseOperand(OperandVector &Operands, StringRef Mnemonic);

  int tryParseRegister(StringRef Mnemonic);

  bool tryParseRegisterOperand(OperandVector &Operands,
                               StringRef Mnemonic);

  bool needsExpansion(MCInst &Inst);

  void expandInstruction(MCInst &Inst, SMLoc IDLoc,
                         SmallVectorImpl<MCInst> &Instructions);
  void expandLoadImm(MCInst &Inst, SMLoc IDLoc,
                     SmallVectorImpl<MCInst> &Instructions);
```

(continues on next page)

(continued from previous page)

```
  void expandLoadAddressImm(MCInst &Inst, SMLoc IDLoc,
                            SmallVectorImpl<MCInst> &Instructions);
  void expandLoadAddressReg(MCInst &Inst, SMLoc IDLoc,
                            SmallVectorImpl<MCInst> &Instructions);
  bool reportParseError(StringRef ErrorMsg);

  bool parseMemOffset(const MCExpr *&Res);
  bool parseRelocOperand(const MCExpr *&Res);

  const MCExpr *evaluateRelocExpr(const MCExpr *Expr, StringRef RelocStr);

  bool parseDirectiveSet();

  bool parseSetAtDirective();
  bool parseSetNoAtDirective();
  bool parseSetMacroDirective();
  bool parseSetNoMacroDirective();
  bool parseSetReorderDirective();
  bool parseSetNoReorderDirective();

  int matchRegisterName(StringRef Symbol);

  int matchRegisterByNumber(unsigned RegNum, StringRef Mnemonic);

  unsigned getReg(int RC,int RegNo);

public:
  Cpu0AsmParser(const MCSubtargetInfo &sti, MCAsmParser &parser,
                const MCInstrInfo &MII, const MCTargetOptions &Options)
    : MCTargetAsmParser(Options, sti, MII), Parser(parser) {
    // Initialize the set of available features.
    setAvailableFeatures(ComputeAvailableFeatures(getSTI().getFeatureBits()));
  }

  MCAsmParser &getParser() const { return Parser; }
  MCAsmLexer &getLexer() const { return Parser.getLexer(); }

};
}

namespace {

/// Cpu0Operand - Instances of this class represent a parsed Cpu0 machine
/// instruction.
class Cpu0Operand : public MCParsedAsmOperand {

  enum KindTy {
    k_Immediate,
    k_Memory,
    k_Register,
    k_Token
  } Kind;
```

(continues on next page)

(continued from previous page)

```
public:
  Cpu0Operand(KindTy K) : MCParsedAsmOperand(), Kind(K) {}

  struct Token {
    const char *Data;
    unsigned Length;
  };
  struct PhysRegOp {
    unsigned RegNum; /// Register Number
  };
  struct ImmOp {
    const MCExpr *Val;
  };
  struct MemOp {
    unsigned Base;
    const MCExpr *Off;
  };

  union {
    struct Token Tok;
    struct PhysRegOp Reg;
    struct ImmOp Imm;
    struct MemOp Mem;
  };

  SMLoc StartLoc, EndLoc;

public:
  void addRegOperands(MCInst &Inst, unsigned N) const {
    assert(N == 1 && "Invalid number of operands!");
    Inst.addOperand(MCOperand::createReg(getReg()));
  }

  void addExpr(MCInst &Inst, const MCExpr *Expr) const{
    // Add as immediate when possible.  Null MCExpr = 0.
    if (Expr == 0)
      Inst.addOperand(MCOperand::createImm(0));
    else if (const MCConstantExpr *CE = dyn_cast<MCConstantExpr>(Expr))
      Inst.addOperand(MCOperand::createImm(CE->getValue()));
    else
      Inst.addOperand(MCOperand::createExpr(Expr));
  }

  void addImmOperands(MCInst &Inst, unsigned N) const {
    assert(N == 1 && "Invalid number of operands!");
    const MCExpr *Expr = getImm();
    addExpr(Inst,Expr);
  }

  void addMemOperands(MCInst &Inst, unsigned N) const {
    assert(N == 2 && "Invalid number of operands!");
```

(continues on next page)

(continued from previous page)

```
  Inst.addOperand(MCOperand::createReg(getMemBase()));

  const MCExpr *Expr = getMemOff();
  addExpr(Inst,Expr);
}

bool isReg() const override { return Kind == k_Register; }
bool isImm() const override { return Kind == k_Immediate; }
bool isToken() const override { return Kind == k_Token; }
bool isMem() const override { return Kind == k_Memory; }

StringRef getToken() const {
  assert(Kind == k_Token && "Invalid access!");
  return StringRef(Tok.Data, Tok.Length);
}

unsigned getReg() const override {
  assert((Kind == k_Register) && "Invalid access!");
  return Reg.RegNum;
}

const MCExpr *getImm() const {
  assert((Kind == k_Immediate) && "Invalid access!");
  return Imm.Val;
}

unsigned getMemBase() const {
  assert((Kind == k_Memory) && "Invalid access!");
  return Mem.Base;
}

const MCExpr *getMemOff() const {
  assert((Kind == k_Memory) && "Invalid access!");
  return Mem.Off;
}

static std::unique_ptr<Cpu0Operand> CreateToken(StringRef Str, SMLoc S) {
  auto Op = std::make_unique<Cpu0Operand>(k_Token);
  Op->Tok.Data = Str.data();
  Op->Tok.Length = Str.size();
  Op->StartLoc = S;
  Op->EndLoc = S;
  return Op;
}

/// Internal constructor for register kinds
static std::unique_ptr<Cpu0Operand> CreateReg(unsigned RegNum, SMLoc S,
                                              SMLoc E) {
  auto Op = std::make_unique<Cpu0Operand>(k_Register);
  Op->Reg.RegNum = RegNum;
  Op->StartLoc = S;
```

(continues on next page)

(continued from previous page)

```
    Op->EndLoc = E;
    return Op;
  }

  static std::unique_ptr<Cpu0Operand> CreateImm(const MCExpr *Val, SMLoc S, SMLoc E) {
    auto Op = std::make_unique<Cpu0Operand>(k_Immediate);
    Op->Imm.Val = Val;
    Op->StartLoc = S;
    Op->EndLoc = E;
    return Op;
  }

  static std::unique_ptr<Cpu0Operand> CreateMem(unsigned Base, const MCExpr *Off,
                                SMLoc S, SMLoc E) {
    auto Op = std::make_unique<Cpu0Operand>(k_Memory);
    Op->Mem.Base = Base;
    Op->Mem.Off = Off;
    Op->StartLoc = S;
    Op->EndLoc = E;
    return Op;
  }

  /// getStartLoc - Get the location of the first token of this operand.
  SMLoc getStartLoc() const override { return StartLoc; }
  /// getEndLoc - Get the location of the last token of this operand.
  SMLoc getEndLoc() const override { return EndLoc; }

  void print(raw_ostream &OS) const override {
    switch (Kind) {
    case k_Immediate:
      OS << "Imm<";
      OS << *Imm.Val;
      OS << ">";
      break;
    case k_Memory:
      OS << "Mem<";
      OS << Mem.Base;
      OS << ", ";
      OS << *Mem.Off;
      OS << ">";
      break;
    case k_Register:
      OS << "Register<" << Reg.RegNum << ">";
      break;
    case k_Token:
      OS << Tok.Data;
      break;
    }
  }
};
}
```

(continues on next page)

(continued from previous page)

```
void printCpu0Operands(OperandVector &Operands) {
  for (size_t i = 0; i < Operands.size(); i++) {
    Cpu0Operand* op = static_cast<Cpu0Operand*>(&*Operands[i]);
    assert(op != nullptr);
    LLVM_DEBUG(dbgs() << "<" << *op << ">");
  }
  LLVM_DEBUG(dbgs() << "\n");
}

//@1 {
bool Cpu0AsmParser::needsExpansion(MCInst &Inst) {

  switch(Inst.getOpcode()) {
    case Cpu0::LoadImm32Reg:
    case Cpu0::LoadAddr32Imm:
    case Cpu0::LoadAddr32Reg:
      return true;
    default:
      return false;
  }
}

void Cpu0AsmParser::expandInstruction(MCInst &Inst, SMLoc IDLoc,
                        SmallVectorImpl<MCInst> &Instructions){
  switch(Inst.getOpcode()) {
    case Cpu0::LoadImm32Reg:
      return expandLoadImm(Inst, IDLoc, Instructions);
    case Cpu0::LoadAddr32Imm:
      return expandLoadAddressImm(Inst,IDLoc,Instructions);
    case Cpu0::LoadAddr32Reg:
      return expandLoadAddressReg(Inst,IDLoc,Instructions);
    }
}
//@1 }

void Cpu0AsmParser::expandLoadImm(MCInst &Inst, SMLoc IDLoc,
                                  SmallVectorImpl<MCInst> &Instructions){
  MCInst tmpInst;
  const MCOperand &ImmOp = Inst.getOperand(1);
  assert(ImmOp.isImm() && "expected immediate operand kind");
  const MCOperand &RegOp = Inst.getOperand(0);
  assert(RegOp.isReg() && "expected register operand kind");

  int ImmValue = ImmOp.getImm();
  tmpInst.setLoc(IDLoc);
  if ( 0 <= ImmValue && ImmValue <= 65535) {
    // for 0 <= j <= 65535.
    // li d,j => ori d,$zero,j
    tmpInst.setOpcode(Cpu0::ORi);
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(
              MCOperand::createReg(Cpu0::ZERO));
```

(continues on next page)

(continued from previous page)

```
    tmpInst.addOperand(MCOperand::createImm(ImmValue));
    Instructions.push_back(tmpInst);
  } else if ( ImmValue < 0 && ImmValue >= -32768) {
    // for -32768 <= j < 0.
    // li d,j => addiu d,$zero,j
    tmpInst.setOpcode(Cpu0::ADDiu); //TODO:no ADDiu64 in td files?
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(
              MCOperand::createReg(Cpu0::ZERO));
    tmpInst.addOperand(MCOperand::createImm(ImmValue));
    Instructions.push_back(tmpInst);
  } else {
    // for any other value of j that is representable as a 32-bit integer.
    // li d,j => lui d,hi16(j)
    //           ori d,d,lo16(j)
    tmpInst.setOpcode(Cpu0::LUi);
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm((ImmValue & 0xffff0000) >> 16));
    Instructions.push_back(tmpInst);
    tmpInst.clear();
    tmpInst.setOpcode(Cpu0::ORi);
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm(ImmValue & 0xffff));
    tmpInst.setLoc(IDLoc);
    Instructions.push_back(tmpInst);
  }
}

void Cpu0AsmParser::expandLoadAddressReg(MCInst &Inst, SMLoc IDLoc,
                                         SmallVectorImpl<MCInst> &Instructions){
  MCInst tmpInst;
  const MCOperand &ImmOp = Inst.getOperand(2);
  assert(ImmOp.isImm() && "expected immediate operand kind");
  const MCOperand &SrcRegOp = Inst.getOperand(1);
  assert(SrcRegOp.isReg() && "expected register operand kind");
  const MCOperand &DstRegOp = Inst.getOperand(0);
  assert(DstRegOp.isReg() && "expected register operand kind");
  int ImmValue = ImmOp.getImm();
  if ( -32768 <= ImmValue && ImmValue <= 32767) {
    // for -32768 <= j < 32767.
    //la d,j(s) => addiu d,s,j
    tmpInst.setOpcode(Cpu0::ADDiu); //TODO:no ADDiu64 in td files?
    tmpInst.addOperand(MCOperand::createReg(DstRegOp.getReg()));
    tmpInst.addOperand(MCOperand::createReg(SrcRegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm(ImmValue));
    Instructions.push_back(tmpInst);
  } else {
    // for any other value of j that is representable as a 32-bit integer.
    // la d,j(s) => lui d,hi16(j)
    //              ori d,d,lo16(j)
    //              add d,d,s
```

(continues on next page)

(continued from previous page)

```
    tmpInst.setOpcode(Cpu0::LUi);
    tmpInst.addOperand(MCOperand::createReg(DstRegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm((ImmValue & 0xffff0000) >> 16));
    Instructions.push_back(tmpInst);
    tmpInst.clear();
    tmpInst.setOpcode(Cpu0::ORi);
    tmpInst.addOperand(MCOperand::createReg(DstRegOp.getReg()));
    tmpInst.addOperand(MCOperand::createReg(DstRegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm(ImmValue & 0xffff));
    Instructions.push_back(tmpInst);
    tmpInst.clear();
    tmpInst.setOpcode(Cpu0::ADD);
    tmpInst.addOperand(MCOperand::createReg(DstRegOp.getReg()));
    tmpInst.addOperand(MCOperand::createReg(DstRegOp.getReg()));
    tmpInst.addOperand(MCOperand::createReg(SrcRegOp.getReg()));
    Instructions.push_back(tmpInst);
  }
}

void Cpu0AsmParser::expandLoadAddressImm(MCInst &Inst, SMLoc IDLoc,
                                         SmallVectorImpl<MCInst> &Instructions){
  MCInst tmpInst;
  const MCOperand &ImmOp = Inst.getOperand(1);
  assert(ImmOp.isImm() && "expected immediate operand kind");
  const MCOperand &RegOp = Inst.getOperand(0);
  assert(RegOp.isReg() && "expected register operand kind");
  int ImmValue = ImmOp.getImm();
  if ( -32768 <= ImmValue && ImmValue <= 32767) {
    // for -32768 <= j < 32767.
    //la d,j => addiu d,$zero,j
    tmpInst.setOpcode(Cpu0::ADDiu);
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(
              MCOperand::createReg(Cpu0::ZERO));
    tmpInst.addOperand(MCOperand::createImm(ImmValue));
    Instructions.push_back(tmpInst);
  } else {
    // for any other value of j that is representable as a 32-bit integer.
    // la d,j => lui d,hi16(j)
    //           ori d,d,lo16(j)
    tmpInst.setOpcode(Cpu0::LUi);
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm((ImmValue & 0xffff0000) >> 16));
    Instructions.push_back(tmpInst);
    tmpInst.clear();
    tmpInst.setOpcode(Cpu0::ORi);
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(MCOperand::createImm(ImmValue & 0xffff));
    Instructions.push_back(tmpInst);
  }
}
```

(continues on next page)

(continued from previous page)

```
//@2 {
bool Cpu0AsmParser::MatchAndEmitInstruction(SMLoc IDLoc, unsigned &Opcode,
                                            OperandVector &Operands,
                                            MCStreamer &Out,
                                            uint64_t &ErrorInfo,
                                            bool MatchingInlineAsm) {
  printCpu0Operands(Operands);
  MCInst Inst;
  unsigned MatchResult = MatchInstructionImpl(Operands, Inst, ErrorInfo,
                                              MatchingInlineAsm);
  switch (MatchResult) {
  default: break;
  case Match_Success: {
    if (needsExpansion(Inst)) {
      SmallVector<MCInst, 4> Instructions;
      expandInstruction(Inst, IDLoc, Instructions);
      for(unsigned i =0; i < Instructions.size(); i++){
        Out.emitInstruction(Instructions[i], getSTI());
      }
    } else {
        Inst.setLoc(IDLoc);
        Out.emitInstruction(Inst, getSTI());
      }
    return false;
  }
//@2 }
  case Match_MissingFeature:
    Error(IDLoc, "instruction requires a CPU feature not currently enabled");
    return true;
  case Match_InvalidOperand: {
    SMLoc ErrorLoc = IDLoc;
    if (ErrorInfo != ~0U) {
      if (ErrorInfo >= Operands.size())
        return Error(IDLoc, "too few operands for instruction");

      ErrorLoc = ((Cpu0Operand &)*Operands[ErrorInfo]).getStartLoc();
      if (ErrorLoc == SMLoc()) ErrorLoc = IDLoc;
    }

    return Error(ErrorLoc, "invalid operand for instruction");
  }
  case Match_MnemonicFail:
    return Error(IDLoc, "invalid instruction");
  }
  return true;
}

int Cpu0AsmParser::matchRegisterName(StringRef Name) {

   int CC;
    CC = StringSwitch<unsigned>(Name)
```

(continues on next page)

(continued from previous page)

```
      .Case("zero",  Cpu0::ZERO)
      .Case("at",  Cpu0::AT)
      .Case("v0",  Cpu0::V0)
      .Case("v1",  Cpu0::V1)
      .Case("a0",  Cpu0::A0)
      .Case("a1",  Cpu0::A1)
      .Case("t9",  Cpu0::T9)
      .Case("t0",  Cpu0::T0)
      .Case("t1",  Cpu0::T1)
      .Case("s0",  Cpu0::S0)
      .Case("s1",  Cpu0::S1)
      .Case("sw",  Cpu0::SW)
      .Case("gp",  Cpu0::GP)
      .Case("fp",  Cpu0::FP)
      .Case("sp",  Cpu0::SP)
      .Case("lr",  Cpu0::LR)
      .Case("pc",  Cpu0::PC)
      .Case("hi",  Cpu0::HI)
      .Case("lo",  Cpu0::LO)
      .Case("epc", Cpu0::EPC)
      .Default(-1);

  if (CC != -1)
    return CC;

  return -1;
}

unsigned Cpu0AsmParser::getReg(int RC,int RegNo) {
  return *(getContext().getRegisterInfo()->getRegClass(RC).begin() + RegNo);
}

int Cpu0AsmParser::matchRegisterByNumber(unsigned RegNum, StringRef Mnemonic) {
  if (RegNum > 15)
    return -1;

  return getReg(Cpu0::CPURegsRegClassID, RegNum);
}

int Cpu0AsmParser::tryParseRegister(StringRef Mnemonic) {
  const AsmToken &Tok = Parser.getTok();
  int RegNum = -1;

  if (Tok.is(AsmToken::Identifier)) {
    std::string lowerCase = Tok.getString().lower();
    RegNum = matchRegisterName(lowerCase);
  } else if (Tok.is(AsmToken::Integer))
    RegNum = matchRegisterByNumber(static_cast<unsigned>(Tok.getIntVal()),
                                   Mnemonic.lower());
    else
      return RegNum;  //error
  return RegNum;
```

(continues on next page)

(continued from previous page)

```
}

bool Cpu0AsmParser::
  tryParseRegisterOperand(OperandVector &Operands,
                          StringRef Mnemonic){

  SMLoc S = Parser.getTok().getLoc();
  int RegNo = -1;

    RegNo = tryParseRegister(Mnemonic);
  if (RegNo == -1)
    return true;

  Operands.push_back(Cpu0Operand::CreateReg(RegNo, S,
      Parser.getTok().getLoc()));
  Parser.Lex(); // Eat register token.
  return false;
}

bool Cpu0AsmParser::ParseOperand(OperandVector &Operands,
                                 StringRef Mnemonic) {
  LLVM_DEBUG(dbgs() << "ParseOperand\n");
  // Check if the current operand has a custom associated parser, if so, try to
  // custom parse the operand, or fallback to the general approach.
  OperandMatchResultTy ResTy = MatchOperandParserImpl(Operands, Mnemonic);
  if (ResTy == MatchOperand_Success)
    return false;
  // If there wasn't a custom match, try the generic matcher below. Otherwise,
  // there was a match, but an error occurred, in which case, just return that
  // the operand parsing failed.
  if (ResTy == MatchOperand_ParseFail)
    return true;

  LLVM_DEBUG(dbgs() << ".. Generic Parser\n");

  switch (getLexer().getKind()) {
  default:
    Error(Parser.getTok().getLoc(), "unexpected token in operand");
    return true;
  case AsmToken::Dollar: {
    // parse register
    SMLoc S = Parser.getTok().getLoc();
    Parser.Lex(); // Eat dollar token.
    // parse register operand
    if (!tryParseRegisterOperand(Operands, Mnemonic)) {
      if (getLexer().is(AsmToken::LParen)) {
        // check if it is indexed addressing operand
        Operands.push_back(Cpu0Operand::CreateToken("(", S));
        Parser.Lex(); // eat parenthesis
        if (getLexer().isNot(AsmToken::Dollar))
          return true;
```

(continues on next page)

(continued from previous page)

```
      Parser.Lex(); // eat dollar
      if (tryParseRegisterOperand(Operands, Mnemonic))
        return true;

      if (!getLexer().is(AsmToken::RParen))
        return true;

      S = Parser.getTok().getLoc();
      Operands.push_back(Cpu0Operand::CreateToken(")", S));
      Parser.Lex();
    }
    return false;
  }
  // maybe it is a symbol reference
  StringRef Identifier;
  if (Parser.parseIdentifier(Identifier))
    return true;

  SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);

  MCSymbol *Sym = getContext().getOrCreateSymbol("$" + Identifier);

  // Otherwise create a symbol ref.
  const MCExpr *Res = MCSymbolRefExpr::create(Sym, MCSymbolRefExpr::VK_None,
                                              getContext());

  Operands.push_back(Cpu0Operand::CreateImm(Res, S, E));
  return false;
}
case AsmToken::Identifier:
case AsmToken::LParen:
case AsmToken::Minus:
case AsmToken::Plus:
case AsmToken::Integer:
case AsmToken::String: {
   // quoted label names
  const MCExpr *IdVal;
  SMLoc S = Parser.getTok().getLoc();
  if (getParser().parseExpression(IdVal))
    return true;
  SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);
  Operands.push_back(Cpu0Operand::CreateImm(IdVal, S, E));
  return false;
}
case AsmToken::Percent: {
  // it is a symbol reference or constant expression
  const MCExpr *IdVal;
  SMLoc S = Parser.getTok().getLoc(); // start location of the operand
  if (parseRelocOperand(IdVal))
    return true;

  SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);
```

(continues on next page)

(continued from previous page)

```
    Operands.push_back(Cpu0Operand::CreateImm(IdVal, S, E));
    return false;
  } // case AsmToken::Percent
  } // switch(getLexer().getKind())
  return true;
}

const MCExpr *Cpu0AsmParser::evaluateRelocExpr(const MCExpr *Expr,
                                               StringRef RelocStr) {
  Cpu0MCExpr::Cpu0ExprKind Kind =
      StringSwitch<Cpu0MCExpr::Cpu0ExprKind>(RelocStr)
          .Case("call16", Cpu0MCExpr::CEK_GOT_CALL)
          .Case("call_hi", Cpu0MCExpr::CEK_CALL_HI16)
          .Case("call_lo", Cpu0MCExpr::CEK_CALL_LO16)
          .Case("dtp_hi", Cpu0MCExpr::CEK_DTP_HI)
          .Case("dtp_lo", Cpu0MCExpr::CEK_DTP_LO)
          .Case("got", Cpu0MCExpr::CEK_GOT)
          .Case("got_hi", Cpu0MCExpr::CEK_GOT_HI16)
          .Case("got_lo", Cpu0MCExpr::CEK_GOT_LO16)
          .Case("gottprel", Cpu0MCExpr::CEK_GOTTPREL)
          .Case("gp_rel", Cpu0MCExpr::CEK_GPREL)
          .Case("hi", Cpu0MCExpr::CEK_ABS_HI)
          .Case("lo", Cpu0MCExpr::CEK_ABS_LO)
          .Case("tlsgd", Cpu0MCExpr::CEK_TLSGD)
          .Case("tlsldm", Cpu0MCExpr::CEK_TLSLDM)
          .Case("tp_hi", Cpu0MCExpr::CEK_TP_HI)
          .Case("tp_lo", Cpu0MCExpr::CEK_TP_LO)
          .Default(Cpu0MCExpr::CEK_None);

  assert(Kind != Cpu0MCExpr::CEK_None);
  return Cpu0MCExpr::create(Kind, Expr, getContext());
}

bool Cpu0AsmParser::parseRelocOperand(const MCExpr *&Res) {

  Parser.Lex(); // eat % token
  const AsmToken &Tok = Parser.getTok(); // get next token, operation
  if (Tok.isNot(AsmToken::Identifier))
    return true;

  std::string Str = Tok.getIdentifier().str();

  Parser.Lex(); // eat identifier
  // now make expression from the rest of the operand
  const MCExpr *IdVal;
  SMLoc EndLoc;

  if (getLexer().getKind() == AsmToken::LParen) {
    while (1) {
      Parser.Lex(); // eat '(' token
      if (getLexer().getKind() == AsmToken::Percent) {
```

(continues on next page)

(continued from previous page)

```
        Parser.Lex(); // eat % token
        const AsmToken &nextTok = Parser.getTok();
        if (nextTok.isNot(AsmToken::Identifier))
          return true;
        Str += "(%";
        Str += nextTok.getIdentifier();
        Parser.Lex(); // eat identifier
        if (getLexer().getKind() != AsmToken::LParen)
          return true;
      } else
        break;
    }
    if (getParser().parseParenExpression(IdVal,EndLoc))
      return true;

    while (getLexer().getKind() == AsmToken::RParen)
      Parser.Lex(); // eat ')' token

  } else
    return true; // parenthesis must follow reloc operand

  Res = evaluateRelocExpr(IdVal, Str);
  return false;
}

bool Cpu0AsmParser::ParseRegister(unsigned &RegNo, SMLoc &StartLoc,
                                  SMLoc &EndLoc) {

  StartLoc = Parser.getTok().getLoc();
  RegNo = tryParseRegister("");
  EndLoc = Parser.getTok().getLoc();
  return (RegNo == (unsigned)-1);
}

OperandMatchResultTy Cpu0AsmParser::tryParseRegister(unsigned &RegNo,
                                                     SMLoc &StartLoc,
                                                     SMLoc &EndLoc) {
  StartLoc = Parser.getTok().getLoc();
  RegNo = tryParseRegister("");
  EndLoc = Parser.getTok().getLoc();
  return (RegNo == (unsigned)-1) ? MatchOperand_NoMatch
                                 : MatchOperand_Success;
}

bool Cpu0AsmParser::parseMemOffset(const MCExpr *&Res) {
  switch(getLexer().getKind()) {
  default:
    return true;
  case AsmToken::Integer:
  case AsmToken::Minus:
  case AsmToken::Plus:
    return (getParser().parseExpression(Res));
```

(continues on next page)

(continued from previous page)

```
  case AsmToken::Percent:
    return parseRelocOperand(Res);
  case AsmToken::LParen:
    return false;  // it's probably assuming 0
  }
  return true;
}

// eg, 12($sp) or 12(la)
OperandMatchResultTy Cpu0AsmParser::parseMemOperand(
                OperandVector &Operands) {

  const MCExpr *IdVal = 0;
  SMLoc S;
  // first operand is the offset
  S = Parser.getTok().getLoc();

  if (parseMemOffset(IdVal))
    return MatchOperand_ParseFail;

  const AsmToken &Tok = Parser.getTok(); // get next token
  if (Tok.isNot(AsmToken::LParen)) {
    Cpu0Operand &Mnemonic = static_cast<Cpu0Operand &>(*Operands[0]);
    if (Mnemonic.getToken() == "la") {
      SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer()-1);
      Operands.push_back(Cpu0Operand::CreateImm(IdVal, S, E));
      return MatchOperand_Success;
    }
    Error(Parser.getTok().getLoc(), "'(' expected");
    return MatchOperand_ParseFail;
  }

  Parser.Lex(); // Eat '(' token.

  const AsmToken &Tok1 = Parser.getTok(); // get next token
  if (Tok1.is(AsmToken::Dollar)) {
    Parser.Lex(); // Eat '$' token.
    if (tryParseRegisterOperand(Operands,"")) {
      Error(Parser.getTok().getLoc(), "unexpected token in operand");
      return MatchOperand_ParseFail;
    }

  } else {
    Error(Parser.getTok().getLoc(), "unexpected token in operand");
    return MatchOperand_ParseFail;
  }

  const AsmToken &Tok2 = Parser.getTok(); // get next token
  if (Tok2.isNot(AsmToken::RParen)) {
    Error(Parser.getTok().getLoc(), "')' expected");
    return MatchOperand_ParseFail;
  }
```

(continues on next page)

(continued from previous page)

```
  SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);

  Parser.Lex(); // Eat ')' token.

  if (!IdVal)
    IdVal = MCConstantExpr::create(0, getContext());

  // Replace the register operand with the memory operand.
  std::unique_ptr<Cpu0Operand> op(
      static_cast<Cpu0Operand *>(Operands.back().release()));
  int RegNo = op->getReg();
  // remove register from operands
  Operands.pop_back();
  // and add memory operand
  Operands.push_back(Cpu0Operand::CreateMem(RegNo, IdVal, S, E));
  return MatchOperand_Success;
}

bool Cpu0AsmParser::
ParseInstruction(ParseInstructionInfo &Info, StringRef Name, SMLoc NameLoc,
                 OperandVector &Operands) {

  // Create the leading tokens for the mnemonic, split by '.' characters.
  size_t Start = 0, Next = Name.find('.');
  StringRef Mnemonic = Name.slice(Start, Next);
  // Refer to the explanation in source code of function DecodeJumpFR(...) in
  // Cpu0Disassembler.cpp
  if (Mnemonic == "ret")
    Mnemonic = "jr";

  Operands.push_back(Cpu0Operand::CreateToken(Mnemonic, NameLoc));

  // Read the remaining operands.
  if (getLexer().isNot(AsmToken::EndOfStatement)) {
    // Read the first operand.
    if (ParseOperand(Operands, Name)) {
      SMLoc Loc = getLexer().getLoc();
      Parser.eatToEndOfStatement();
      return Error(Loc, "unexpected token in argument list");
    }

    while (getLexer().is(AsmToken::Comma) ) {
      Parser.Lex();  // Eat the comma.

      // Parse and remember the operand.
      if (ParseOperand(Operands, Name)) {
        SMLoc Loc = getLexer().getLoc();
        Parser.eatToEndOfStatement();
        return Error(Loc, "unexpected token in argument list");
      }
    }
```

(continues on next page)

(continued from previous page)

```
  }

  if (getLexer().isNot(AsmToken::EndOfStatement)) {
    SMLoc Loc = getLexer().getLoc();
    Parser.eatToEndOfStatement();
    return Error(Loc, "unexpected token in argument list");
  }

  Parser.Lex(); // Consume the EndOfStatement
  return false;
}

bool Cpu0AsmParser::reportParseError(StringRef ErrorMsg) {
   SMLoc Loc = getLexer().getLoc();
   Parser.eatToEndOfStatement();
   return Error(Loc, ErrorMsg);
}

bool Cpu0AsmParser::parseSetReorderDirective() {
  Parser.Lex();
  // if this is not the end of the statement, report error
  if (getLexer().isNot(AsmToken::EndOfStatement)) {
    reportParseError("unexpected token in statement");
    return false;
  }
  Options.setReorder();
  Parser.Lex(); // Consume the EndOfStatement
  return false;
}

bool Cpu0AsmParser::parseSetNoReorderDirective() {
    Parser.Lex();
    // if this is not the end of the statement, report error
    if (getLexer().isNot(AsmToken::EndOfStatement)) {
      reportParseError("unexpected token in statement");
      return false;
    }
    Options.setNoreorder();
    Parser.Lex(); // Consume the EndOfStatement
    return false;
}

bool Cpu0AsmParser::parseSetMacroDirective() {
  Parser.Lex();
  // if this is not the end of the statement, report error
  if (getLexer().isNot(AsmToken::EndOfStatement)) {
    reportParseError("unexpected token in statement");
    return false;
  }
  Options.setMacro();
  Parser.Lex(); // Consume the EndOfStatement
  return false;
```

(continues on next page)

(continued from previous page)

```
}

bool Cpu0AsmParser::parseSetNoMacroDirective() {
  Parser.Lex();
  // if this is not the end of the statement, report error
  if (getLexer().isNot(AsmToken::EndOfStatement)) {
    reportParseError("`noreorder' must be set before `nomacro'");
    return false;
  }
  if (Options.isReorder()) {
    reportParseError("`noreorder' must be set before `nomacro'");
    return false;
  }
  Options.setNomacro();
  Parser.Lex(); // Consume the EndOfStatement
  return false;
}
bool Cpu0AsmParser::parseDirectiveSet() {

  // get next token
  const AsmToken &Tok = Parser.getTok();

  if (Tok.getString() == "reorder") {
    return parseSetReorderDirective();
  } else if (Tok.getString() == "noreorder") {
    return parseSetNoReorderDirective();
  } else if (Tok.getString() == "macro") {
    return parseSetMacroDirective();
  } else if (Tok.getString() == "nomacro") {
    return parseSetNoMacroDirective();
  }
  return true;
}

bool Cpu0AsmParser::ParseDirective(AsmToken DirectiveID) {

  if (DirectiveID.getString() == ".ent") {
    // ignore this directive for now
    Parser.Lex();
    return false;
  }

  if (DirectiveID.getString() == ".end") {
    // ignore this directive for now
    Parser.Lex();
    return false;
  }

  if (DirectiveID.getString() == ".frame") {
    // ignore this directive for now
    Parser.eatToEndOfStatement();
    return false;
```

(continues on next page)

(continued from previous page)

```
  }

  if (DirectiveID.getString() == ".set") {
    return parseDirectiveSet();
  }

  if (DirectiveID.getString() == ".fmask") {
    // ignore this directive for now
    Parser.eatToEndOfStatement();
    return false;
  }

  if (DirectiveID.getString() == ".mask") {
    // ignore this directive for now
    Parser.eatToEndOfStatement();
    return false;
  }

  if (DirectiveID.getString() == ".gpword") {
    // ignore this directive for now
    Parser.eatToEndOfStatement();
    return false;
  }

  return true;
}

extern "C" void LLVMInitializeCpu0AsmParser() {
  RegisterMCAsmParser<Cpu0AsmParser> X(TheCpu0Target);
  RegisterMCAsmParser<Cpu0AsmParser> Y(TheCpu0elTarget);
}

#define GET_REGISTER_MATCHER
#define GET_MATCHER_IMPLEMENTATION
#include "Cpu0GenAsmMatcher.inc"

#else // #if CH >= CH11_1
extern "C" void LLVMInitializeCpu0AsmParser() {}
#endif
```

lbdex/chapters/Chapter11_1/AsmParser/CMakeLists.txt

```
add_llvm_component_library(LLVMCpu0AsmParser
  Cpu0AsmParser.cpp

  LINK_COMPONENTS
  MC
  MCParser
  Cpu0Desc
  Cpu0Info
```

(continues on next page)

(continued from previous page)

```
  Support

  ADD_TO_COMPONENT
  Cpu0
  )
```

The Cpu0AsmParser.cpp contains one thousand lines of code which do the assembly language parsing. You can understand it with a little patience only. To let files in directory of AsmParser be built, modify CMakeLists.txt as follows,

lbdex/chapters/Chapter11_1/CMakeLists.txt

```
set(LLVM_TARGET_DEFINITIONS Cpu0Asm.td)
tablegen(LLVM Cpu0GenAsmMatcher.inc -gen-asm-matcher)
```

```
  Cpu0AsmParser
```

```
add_subdirectory(AsmParser)
```

lbdex/chapters/Chapter11_1/Cpu0Asm.td

```
//===-- Cpu0Asm.td - Describe the Cpu0 Target Machine ------*- tablegen -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
// This is the top level entry point for the Cpu0 target.
//===----------------------------------------------------------------------===//

//===----------------------------------------------------------------------===//
// Target-independent interfaces
//===----------------------------------------------------------------------===//

include "llvm/Target/Target.td"

//===----------------------------------------------------------------------===//
// Target-dependent interfaces
//===----------------------------------------------------------------------===//

include "Cpu0RegisterInfo.td"
include "Cpu0RegisterInfoGPROutForAsm.td"
include "Cpu0.td"
```

lbdex/chapters/Chapter11_1/Cpu0RegisterInfoGPROutForAsm.td

```
//===----------------------------------------------------------------------===//
// Register Classes
//===----------------------------------------------------------------------===//

def GPROut : RegisterClass<"Cpu0", [i32], 32, (add CPURegs)>;
```

The CMakeLists.txt adds code as above to generate Cpu0GenAsmMatcher.inc used by Cpu0AsmParser.cpp. Cpu0Asm.td include Cpu0RegisterInfoGPROutForAsm.td which define GPROut to CPURegs while Cpu0Other.td include Cpu0RegisterInfoGPROutForOther.td which define GPROut to CPURegs exclude SW. Cpu0Other.td is used when translating llvm IR to Cpu0 instruction. In latter case, the register SW is reserved for keeping the CPU status and not allowed to be allocated as a general purpose register. For example, if setting GPROut to include SW, when compile with C statement "a = (b & c);", it may generate instruction "and $sw, $1, $2", as a result that interrupt status in $sw will be destroyed. When programming in assembly, instruction "andi $sw, $sw, 0xffdf" is allowed. This assembly program is accepted and Cpu0 backend treats it safe since assembler programmer can disable trace debug message by "andi $sw, $sw, 0xffdf" and enable debug message by "ori $sw, $sw, 0x0020". In addition, the interrupt bits can also be enabled or disabled by "ori" and "andi" instructions.

The EPC must set to CPURegs as follows, otherwise, MatchInstructionImpl() of MatchAndEmitInstruction() will return fail for "asm("mfc0 $pc, $epc");".

lbdex/chapters/Chapter2/Cpu0RegisterInfo.td

```
def CPURegs : RegisterClass<"Cpu0", [i32], 32, (add
  ...
  , PC, EPC)>;
```

lbdex/chapters/Chapter11_1/Cpu0.td

```
def Cpu0AsmParser : AsmParser {
  let ShouldEmitMatchRegisterName = 0;
}

def Cpu0AsmParserVariant : AsmParserVariant {
  int Variant = 0;

  // Recognize hard coded registers.
  string RegisterPrefix = "$";
}
```

```
def Cpu0 : Target {
  ...
```

```
  let AssemblyParsers = [Cpu0AsmParser];
  let AssemblyParserVariants = [Cpu0AsmParserVariant];
```

```
}
```

lbdex/chapters/Chapter11_1/Cpu0InstrFormats.td

```
// Pseudo-instructions for alternate assembly syntax (never used by codegen).
// These are aliases that require C++ handling to convert to the target
// instruction, while InstAliases can be handled directly by tblgen.
class Cpu0AsmPseudoInst<dag outs, dag ins, string asmstr>:
  Cpu0Inst<outs, ins, asmstr, [], IIPseudo, Pseudo> {
  let isPseudo = 1;
  let Pattern = [];
}
```

lbdex/chapters/Chapter11_1/Cpu0InstrInfo.td

```
def Cpu0MemAsmOperand : AsmOperandClass {
  let Name = "Mem";
  let ParserMethod = "parseMemOperand";
}

// Address operand
def mem : Operand<i32> {
  ...
  let ParserMatchClass = Cpu0MemAsmOperand;
}
...
```

```
//===----------------------------------------------------------------------===//
// Pseudo Instruction definition
//===----------------------------------------------------------------------===//

let Predicates = [Ch11_1] in {
class LoadImm32< string instr_asm, Operand Od, RegisterClass RC> :
  Cpu0AsmPseudoInst<(outs RC:$ra), (ins Od:$imm32),
                    !strconcat(instr_asm, "\t$ra, $imm32")> ;
def LoadImm32Reg : LoadImm32<"li", shamt, GPROut>;

class LoadAddress<string instr_asm, Operand MemOpnd, RegisterClass RC> :
  Cpu0AsmPseudoInst<(outs RC:$ra), (ins MemOpnd:$addr),
                    !strconcat(instr_asm, "\t$ra, $addr")> ;
def LoadAddr32Reg : LoadAddress<"la", mem, GPROut>;

class LoadAddressImm<string instr_asm, Operand Od, RegisterClass RC> :
  Cpu0AsmPseudoInst<(outs RC:$ra), (ins Od:$imm32),
                    !strconcat(instr_asm, "\t$ra, $imm32")> ;
def LoadAddr32Imm : LoadAddressImm<"la", shamt, GPROut>;
}
```

Above Cpu0InstrInfo.td declare the **let ParserMethod = "parseMemOperand"** and implement the parseMemOperand() in Cpu0AsmParser.cpp to handle the **"mem"** operand which used in Cpu0 instructions ld and st. For example, ld $2, 4($sp), the **mem** operand is 4($sp). Accompany with **"let ParserMatchClass = Cpu0MemAsmOperand;"**, LLVM will call parseMemOperand() of Cpu0AsmParser.cpp when it meets the assembly **mem** operand 4($sp). With above **"let"** assignment, TableGen will generate the following structure and functions in Cpu0GenAsmMatcher.inc.

build/lib/Target/Cpu0/Cpu0GenAsmMatcher.inc

```
  enum OperandMatchResultTy {
    MatchOperand_Success,    // operand matched successfully
    MatchOperand_NoMatch,    // operand did not match
    MatchOperand_ParseFail   // operand matched but had errors
  };
  OperandMatchResultTy MatchOperandParserImpl(
    OperandVector &Operands,
    StringRef Mnemonic);
  OperandMatchResultTy tryCustomParseOperand(
    OperandVector &Operands,
    unsigned MCK);
...
Cpu0AsmParser::OperandMatchResultTy Cpu0AsmParser::
tryCustomParseOperand(OperandVector &Operands,
            unsigned MCK) {

  switch(MCK) {
  case MCK_Mem:
    return parseMemOperand(Operands);
  default:
    return MatchOperand_NoMatch;
  }
  return MatchOperand_NoMatch;
}

Cpu0AsmParser::OperandMatchResultTy Cpu0AsmParser::
MatchOperandParserImpl(OperandVector &Operands,
             StringRef Mnemonic) {
  ...
}

/// MatchClassKind - The kinds of classes which participate in
/// instruction matching.
enum MatchClassKind {
  ...
  MCK_Mem, // user defined class 'Cpu0MemAsmOperand'
  ...
};
```

Above three Pseudo Instruction definitions in Cpu0InstrInfo.td, such as LoadImm32Reg, are handled by Cpu0AsmParser.cpp as follows,

lbdex/chapters/Chapter11_1/AsmParser/Cpu0AsmParser.cpp

```
bool Cpu0AsmParser::needsExpansion(MCInst &Inst) {

  switch(Inst.getOpcode()) {
    case Cpu0::LoadImm32Reg:
    case Cpu0::LoadAddr32Imm:
    case Cpu0::LoadAddr32Reg:
      return true;
    default:
      return false;
  }
}

void Cpu0AsmParser::expandInstruction(MCInst &Inst, SMLoc IDLoc,
                        SmallVectorImpl<MCInst> &Instructions){
  switch(Inst.getOpcode()) {
    case Cpu0::LoadImm32Reg:
      return expandLoadImm(Inst, IDLoc, Instructions);
    case Cpu0::LoadAddr32Imm:
      return expandLoadAddressImm(Inst,IDLoc,Instructions);
    case Cpu0::LoadAddr32Reg:
      return expandLoadAddressReg(Inst,IDLoc,Instructions);
    }
}
```

```
bool Cpu0AsmParser::MatchAndEmitInstruction(SMLoc IDLoc, unsigned &Opcode,
                                            OperandVector &Operands,
                                            MCStreamer &Out,
                                            uint64_t &ErrorInfo,
                                            bool MatchingInlineAsm) {
  printCpu0Operands(Operands);
  MCInst Inst;
  unsigned MatchResult = MatchInstructionImpl(Operands, Inst, ErrorInfo,
                                              MatchingInlineAsm);
  switch (MatchResult) {
  default: break;
  case Match_Success: {
    if (needsExpansion(Inst)) {
      SmallVector<MCInst, 4> Instructions;
      expandInstruction(Inst, IDLoc, Instructions);
      for(unsigned i =0; i < Instructions.size(); i++){
        Out.emitInstruction(Instructions[i], getSTI());
      }
    } else {
        Inst.setLoc(IDLoc);
        Out.emitInstruction(Inst, getSTI());
      }
    return false;
  }
```

```
  ...
}
```

Finally, remind that the CPURegs as below must follow the order of register number because AsmParser uses them when do register number encoding.

lbdex/chapters/Chapter11_1/Cpu0RegisterInfo.td

```
//===----------------------------------------------------------------------===//
// The register string, such as "9" or "gp" will show on "llvm-objdump -d"
//@ All registers definition
let Namespace = "Cpu0" in {
  //@ General Purpose Registers
  def ZERO : Cpu0GPRReg<0,  "zero">, DwarfRegNum<[0]>;
  def AT   : Cpu0GPRReg<1,  "1">,    DwarfRegNum<[1]>;
  def V0   : Cpu0GPRReg<2,  "2">,    DwarfRegNum<[2]>;
  def V1   : Cpu0GPRReg<3,  "3">,    DwarfRegNum<[3]>;
  def A0   : Cpu0GPRReg<4,  "4">,    DwarfRegNum<[4]>;
  def A1   : Cpu0GPRReg<5,  "5">,    DwarfRegNum<[5]>;
  def T9   : Cpu0GPRReg<6,  "t9">,   DwarfRegNum<[6]>;
  def T0   : Cpu0GPRReg<7,  "7">,    DwarfRegNum<[7]>;
  def T1   : Cpu0GPRReg<8,  "8">,    DwarfRegNum<[8]>;
  def S0   : Cpu0GPRReg<9,  "9">,    DwarfRegNum<[9]>;
  def S1   : Cpu0GPRReg<10, "10">,   DwarfRegNum<[10]>;
  def GP   : Cpu0GPRReg<11, "gp">,   DwarfRegNum<[11]>;
  def FP   : Cpu0GPRReg<12, "fp">,   DwarfRegNum<[12]>;
  def SP   : Cpu0GPRReg<13, "sp">,   DwarfRegNum<[13]>;
  def LR   : Cpu0GPRReg<14, "lr">,   DwarfRegNum<[14]>;
  def SW   : Cpu0GPRReg<15, "sw">,   DwarfRegNum<[15]>;
//  def MAR  : Register< 16, "mar">,  DwarfRegNum<[16]>;
//  def MDR  : Register< 17, "mdr">,  DwarfRegNum<[17]>;

//#if CH >= CH4_1 1
  // Hi/Lo registers number and name
  def HI   : Cpu0Reg<0, "ac0">, DwarfRegNum<[18]>;
  def LO   : Cpu0Reg<0, "ac0">, DwarfRegNum<[19]>;
//#endif
  def PC   : Cpu0C0Reg<0, "pc">,  DwarfRegNum<[20]>;
  def EPC  : Cpu0C0Reg<1, "epc">, DwarfRegNum<[21]>;
}

//===----------------------------------------------------------------------===//
//@Register Classes
//===----------------------------------------------------------------------===//

def CPURegs : RegisterClass<"Cpu0", [i32], 32, (add
  // Reserved
  ZERO, AT,
  // Return Values and Arguments
  V0, V1, A0, A1,
  // Not preserved across procedure calls
  T9, T0, T1,
  // Callee save
  S0, S1,
  // Reserved
```

(continues on next page)

(continued from previous page)

```
  GP, FP,
  SP, LR, SW)>;
```

Run Chapter11_1/ with ch11_1.cpp to get the correct result as follows,

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=pic -filetype=obj ch11_1.bc -o
ch11_1.cpu0.o
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/bin/
llvm-objdump -d ch11_1.cpu0.o

ch11_1.cpu0.o:  file format ELF32-unknown

Disassembly of section .text:
.text:
       0:     01 2d 00 08                                     ld      $2, 8($sp)
       4:     02 0d 00 04                                     st      $zero, 4($sp)
       8:     09 30 00 00                                     addiu   $3, $zero, 0
       c:     13 31 20 00                                     add     $3, $1, $2
      10:     14 32 30 00                                     sub     $3, $2, $3
      ...
```

The instructions cmp and jeg printed with explicit $sw displayed in assembly and disassembly. You can change the code in AsmParser and Dissassembly (the last chapter) to hide the $sw printed in these instructions (such as "jeq 20" rather than "jeq $sw, 20").

Both AsmParser and Cpu0AsmParser inherited from MCAsmParser as follows,

lIvm/lib/MC/MCParser/AsmParser.cpp

```
class AsmParser : public MCAsmParser {
  ...
}
...
```

AsmParser will call functions ParseInstruction() and MatchAndEmitInstruction() of Cpu0AsmParser as follows,

lIvm/lib/MC/MCParser/AsmParser.cpp

```
bool AsmParser::parseStatement(ParseStatementInfo &Info) {
  ...
  // Directives start with "."
  if (IDVal[0] == '.' && IDVal != ".") {
    // First query the target-specific parser. It will return 'true' if it
    // isn't interested in this directive.
    if (!getTargetParser().ParseDirective(ID))
      return false;
    ...
  }
  ...
```

(continues on next page)

(continued from previous page)

```
  bool HadError = getTargetParser().ParseInstruction(IInfo, OpcodeStr, IDLoc,
                                                     Info.ParsedOperands);
  ...
  // If parsing succeeded, match the instruction.
  if (!HadError) {
    unsigned ErrorInfo;
    getTargetParser().MatchAndEmitInstruction(IDLoc, Info.Opcode,
                                              Info.ParsedOperands, Out,
                                              ErrorInfo, ParsingInlineAsm);
  }
  ...
}
```

The other functions in Cpu0AsmParser called as follows,

- ParseDirective() -> parseDirectiveSet() -> parseSetReorderDirective(), parseSetNoReorderDirective(), parseSetMacroDirective(), parseSetNoMacroDirective() -> reportParseError()
- ParseInstruction() -> ParseOperand() -> MatchOperandParserImpl() of Cpu0GenAsmMatcher.inc -> tryCustomParseOperand() of Cpu0GenAsmMatcher.inc -> parseMemOperand() -> parseMemOffset(), tryParseRegisterOperand()
- MatchAndEmitInstruction() -> MatchInstructionImpl() of Cpu0GenAsmMatcher.inc, needsExpansion(), expandInstruction()
- parseMemOffset() -> parseRelocOperand() -> getVariantKind()
- tryParseRegisterOperand() -> tryParseRegister() -> matchRegisterName() -> getReg()), matchRegisterByNumber()
- expandInstruction() -> expandLoadImm(), expandLoadAddressImm(), expandLoadAddressReg() -> EmitInstruction() of Cpu0AsmPrint.cpp

11.2 Inline assembly

Run Chapter11_1 with ch11_2 will get the following error.

lbdex/input/ch11_2.cpp

```
extern "C" int printf(const char *format, ...);
int inlineasm_addu(void)
{
  int foo = 10;
  const int bar = 15;

//  call i32 asm sideeffect "addu $0,$1,$2", "=r,r,r"(i32 %1, i32 %2) #1, !srcloc !1
  __asm__ __volatile__("addu %0,%1,%2"
                       :"=r"(foo) // 5
                       :"r"(foo), "r"(bar)
                       );

  return foo;
```

(continues on next page)

(continued from previous page)

```
}

int inlineasm_longlong(void)
{
  int a, b;
  const long long bar = 0x0000000500000006;
  int* p = (int*)&bar;
//  int* q = (p+1); // Do not set q here.

//  call i32 asm sideeffect "ld $0,$1", "=r,*m"(i32* %2) #2, !srcloc !2
  __asm__ __volatile__("ld %0,%1"
                       :"=r"(a) // 0x700070007000700b
                       :"m"(*p)
                       );
  int* q = (p+1); // Set q just before inline asm refer to avoid register clobbered.
//  call i32 asm sideeffect "ld $0,$1", "=r,*m"(i32* %6) #2, !srcloc !3
  __asm__ __volatile__("ld %0,%1"
                       :"=r"(b) // 11
                       :"m"(*q)
//              Or use :"m"(*(p+1)) to avoid register clobbered.
                       );

  return (a+b);
}

int inlineasm_constraint(void)
{
  int foo = 10;
  const int n_5 = -5;
  const int n5 = 5;
  const int n0 = 0;
  const unsigned int un5 = 5;
  const int n65536 = 0x10000;
  const int n_65531 = -65531;

//   call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,I"(i32 %1, i32 15) #1, !srcloc !2
  __asm__ __volatile__("addiu %0,%1,%2"
                       :"=r"(foo) // 15
                       :"r"(foo), "I"(n_5)
                       );

  __asm__ __volatile__("addiu %0,%1,%2"
                       :"=r"(foo) // 15
                       :"r"(foo), "J"(n0)
                       );

  __asm__ __volatile__("addiu %0,%1,%2"
                       :"=r"(foo) // 10
                       :"r"(foo), "K"(n5)
                       );

  __asm__ __volatile__("ori %0,%1,%2"
```

(continues on next page)

(continued from previous page)

```
                        :"=r"(foo) // 10
                        :"r"(foo), "L"(n65536) // 0x10000 = 65536
                        );

  __asm__ __volatile__("addiu %0,%1,%2"
                        :"=r"(foo) // 15
                        :"r"(foo), "N"(n_65531)
                        );

  __asm__ __volatile__("addiu %0,%1,%2"
                        :"=r"(foo) // 10
                        :"r"(foo), "O"(n_5)
                        );

  __asm__ __volatile__("addiu %0,%1,%2"
                        :"=r"(foo) // 15
                        :"r"(foo), "P"(un5)
                        );

  return foo;
}

int inlineasm_arg(int u, int v)
{
  int w;

  __asm__ __volatile__("subu %0,%1,%2"
                        :"=r"(w)
                        :"r"(u), "r"(v)
                        );

  return w;
}

int g[3] = {1,2,3};

int inlineasm_global()
{
  int c, d;
  __asm__ __volatile__("ld %0,%1"
                        :"=r"(c) // c=3
                        :"m"(g[2])
                        );

  __asm__ __volatile__("addiu %0,%1,1"
                        :"=r"(d) // d=4
                        :"r"(c)
                        );

  return d;
}
```

(continues on next page)

(continued from previous page)

```
#ifdef TESTSOFTFLOATLIB
// test_float() will call soft float library
int inlineasm_float()
{
  float a = 2.2;
  float b = 3.3;

  int c = (int)(a + b);

  int d;
  __asm__ __volatile__("addiu %0,%1,1"
                       :"=r"(d)
                       :"r"(c)
                       );

  return d;
}
#endif

int test_inlineasm()
{
  int a, b, c, d, e, f;

  a = inlineasm_addu(); // 25
  b = inlineasm_longlong(); // 11
  c = inlineasm_constraint(); // 15
  d = inlineasm_arg(1, 10); // -9
  e = inlineasm_arg(6, 3); // 3
  __asm__ __volatile__("addiu %0,%1,1"
                       :"=r"(f) // e=4
                       :"r"(e)
                       );

  return (a+b+c+d+e+f); // 25+11+15-9+3+4=49
}
```

```
1-160-129-73:input Jonathan$ ~/llvm/test/build/bin/llc
-march=cpu0 -relocation-model=static -filetype=asm ch11_2.bc -o -
  .section .mdebug.abi32
  .previous
  .file "ch11_2.bc"
error: couldn't allocate output register for constraint 'r'
```

The ch11_2.cpp is a inline assembly example. The clang supports inline assembly like gcc. The inline assembly used in C/C++ when program need to access the specific allocated register or memory for the C/C++ variable. For example, the variable foo of ch11_2.cpp may be allocated by compiler to register $2, $3 or any other register. The inline assembly fills the gap between high level language and assembly language. Reference here[2]. Chapter11_2 supports inline assembly as follows,

[2] http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

lbdex/chapters/Chapter11_2/Cpu0AsmPrinter.h

```
  bool PrintAsmOperand(const MachineInstr *MI, unsigned OpNo,
                       const char *ExtraCode, raw_ostream &O) override;
  bool PrintAsmMemoryOperand(const MachineInstr *MI, unsigned OpNum,
                             const char *ExtraCode, raw_ostream &O) override;
  void printOperand(const MachineInstr *MI, int opNum, raw_ostream &O);
```

lbdex/chapters/Chapter11_2/Cpu0AsmPrinter.cpp

```
// Print out an operand for an inline asm expression.
bool Cpu0AsmPrinter::PrintAsmOperand(const MachineInstr *MI, unsigned OpNum,
                                     const char *ExtraCode, raw_ostream &O) {
  // Does this asm operand have a single letter operand modifier?
  if (ExtraCode && ExtraCode[0]) {
    if (ExtraCode[1] != 0) return true; // Unknown modifier.

    const MachineOperand &MO = MI->getOperand(OpNum);
    switch (ExtraCode[0]) {
    default:
      // See if this is a generic print operand
      return AsmPrinter::PrintAsmOperand(MI,OpNum, ExtraCode,O);
    case 'X': // hex const int
      if ((MO.getType()) != MachineOperand::MO_Immediate)
        return true;
      O << "0x" << StringRef(utohexstr(MO.getImm())).lower();
      return false;
    case 'x': // hex const int (low 16 bits)
      if ((MO.getType()) != MachineOperand::MO_Immediate)
        return true;
      O << "0x" << StringRef(utohexstr(MO.getImm() & 0xffff)).lower();
      return false;
    case 'd': // decimal const int
      if ((MO.getType()) != MachineOperand::MO_Immediate)
        return true;
      O << MO.getImm();
      return false;
    case 'm': // decimal const int minus 1
      if ((MO.getType()) != MachineOperand::MO_Immediate)
        return true;
      O << MO.getImm() - 1;
      return false;
    case 'z': {
      // $0 if zero, regular printing otherwise
      if (MO.getType() != MachineOperand::MO_Immediate)
        return true;
      int64_t Val = MO.getImm();
      if (Val)
        O << Val;
      else
        O << "$0";
```

(continues on next page)

(continued from previous page)

```
      return false;
    }
    }
  }

  printOperand(MI, OpNum, O);
  return false;
}

bool Cpu0AsmPrinter::PrintAsmMemoryOperand(const MachineInstr *MI,
                                           unsigned OpNum,
                                           const char *ExtraCode,
                                           raw_ostream &O) {
  int Offset = 0;
  // Currently we are expecting either no ExtraCode or 'D'
  if (ExtraCode) {
    return true; // Unknown modifier.
  }

  const MachineOperand &MO = MI->getOperand(OpNum);
  assert(MO.isReg() && "unexpected inline asm memory operand");
  O << Offset << "($" << Cpu0InstPrinter::getRegisterName(MO.getReg()) << ")";

  return false;
}

void Cpu0AsmPrinter::printOperand(const MachineInstr *MI, int opNum,
                                  raw_ostream &O) {
  const MachineOperand &MO = MI->getOperand(opNum);
  bool closeP = false;

  if (MO.getTargetFlags())
    closeP = true;

  switch(MO.getTargetFlags()) {
  case Cpu0II::MO_GPREL:    O << "%gp_rel("; break;
  case Cpu0II::MO_GOT_CALL: O << "%call16("; break;
  case Cpu0II::MO_GOT:      O << "%got(";    break;
  case Cpu0II::MO_ABS_HI:   O << "%hi(";     break;
  case Cpu0II::MO_ABS_LO:   O << "%lo(";     break;
  case Cpu0II::MO_GOT_HI16: O << "%got_hi16("; break;
  case Cpu0II::MO_GOT_LO16: O << "%got_lo16("; break;
  }

  switch (MO.getType()) {
    case MachineOperand::MO_Register:
      O << '$'
        << StringRef(Cpu0InstPrinter::getRegisterName(MO.getReg())).lower();
      break;

    case MachineOperand::MO_Immediate:
      O << MO.getImm();
```

(continues on next page)

(continued from previous page)

```
      break;

    case MachineOperand::MO_MachineBasicBlock:
      O << *MO.getMBB()->getSymbol();
      return;

    case MachineOperand::MO_GlobalAddress:
      O << *getSymbol(MO.getGlobal());
      break;

    case MachineOperand::MO_BlockAddress: {
      MCSymbol *BA = GetBlockAddressSymbol(MO.getBlockAddress());
      O << BA->getName();
      break;
    }

    case MachineOperand::MO_ExternalSymbol:
      O << *GetExternalSymbolSymbol(MO.getSymbolName());
      break;

    case MachineOperand::MO_JumpTableIndex:
      O << MAI->getPrivateGlobalPrefix() << "JTI" << getFunctionNumber()
        << '_' << MO.getIndex();
      break;

    case MachineOperand::MO_ConstantPoolIndex:
      O << MAI->getPrivateGlobalPrefix() << "CPI"
        << getFunctionNumber() << "_" << MO.getIndex();
      if (MO.getOffset())
        O << "+" << MO.getOffset();
      break;

    default:
      llvm_unreachable("<unknown operand type>");
  }

  if (closeP) O << ")";
}
```

lbdex/chapters/Chapter11_2/Cpu0InstrInfo.cpp

```
/// Return the number of bytes of code the specified instruction may be.
unsigned Cpu0InstrInfo::GetInstSizeInBytes(const MachineInstr &MI) const {
```

```
  case  TargetOpcode::INLINEASM: {       // Inline Asm: Variable size.
    const MachineFunction *MF = MI.getParent()->getParent();
    const char *AsmStr = MI.getOperand(0).getSymbolName();
    return getInlineAsmLength(AsmStr, *MF->getTarget().getMCAsmInfo());
  }
```

lbdex/chapters/Chapter11_2/Cpu0ISelDAGToDAG.h

```
  bool SelectInlineAsmMemoryOperand(const SDValue &Op,
                                    unsigned ConstraintID,
                                    std::vector<SDValue> &OutOps) override;
```

lbdex/chapters/Chapter11_2/Cpu0ISelDAGToDAG.cpp

```
// inlineasm begin
bool Cpu0DAGToDAGISel::
SelectInlineAsmMemoryOperand(const SDValue &Op, unsigned ConstraintID,
                             std::vector<SDValue> &OutOps) {
  // All memory constraints can at least accept raw pointers.
  switch(ConstraintID) {
  default:
    llvm_unreachable("Unexpected asm memory constraint");
  case InlineAsm::Constraint_m:
    OutOps.push_back(Op);
    return false;
  }
  return true;
}
// inlineasm end
```

lbdex/chapters/Chapter11_2/Cpu0ISelLowering.h

```
    // Inline asm support
    ConstraintType getConstraintType(StringRef Constraint) const override;

    /// Examine constraint string and operand type and determine a weight value.
    /// The operand object must already have been set up with the operand type.
    ConstraintWeight getSingleConstraintMatchWeight(
      AsmOperandInfo &info, const char *constraint) const override;

    /// This function parses registers that appear in inline-asm constraints.
    /// It returns pair (0, 0) on failure.
    std::pair<unsigned, const TargetRegisterClass *>
    parseRegForInlineAsmConstraint(const StringRef &C, MVT VT) const;

    std::pair<unsigned, const TargetRegisterClass *>
    getRegForInlineAsmConstraint(const TargetRegisterInfo *TRI,
                                 StringRef Constraint, MVT VT) const override;

    /// LowerAsmOperandForConstraint - Lower the specified operand into the Ops
    /// vector.  If it is invalid, don't add anything to Ops. If hasMemory is
    /// true it means one of the asm constraint of the inline asm instruction
    /// being processed is 'm'.
    void LowerAsmOperandForConstraint(SDValue Op,
                                      std::string &Constraint,
                                      std::vector<SDValue> &Ops,
```

(continues on next page)

(continued from previous page)

```
                                    SelectionDAG &DAG) const override;

    bool isLegalAddressingMode(const DataLayout &DL, const AddrMode &AM,
                               Type *Ty, unsigned AS,
                               Instruction *I = nullptr) const override;
```

lbdex/chapters/Chapter11_2/Cpu0ISelLowering.cpp

```
//===----------------------------------------------------------------------===//
//                           Cpu0 Inline Assembly Support
//===----------------------------------------------------------------------===//

/// getConstraintType - Given a constraint letter, return the type of
/// constraint it is for this target.
Cpu0TargetLowering::ConstraintType
Cpu0TargetLowering::getConstraintType(StringRef Constraint) const
{
  // Cpu0 specific constraints
  // GCC config/mips/constraints.md
  // 'c' : A register suitable for use in an indirect
  //       jump. This will always be $t9 for -mabicalls.
  if (Constraint.size() == 1) {
    switch (Constraint[0]) {
      default : break;
      case 'c':
        return C_RegisterClass;
      case 'R':
        return C_Memory;
    }
  }
  return TargetLowering::getConstraintType(Constraint);
}

/// Examine constraint type and operand type and determine a weight value.
/// This object must already have been set up with the operand type
/// and the current alternative constraint selected.
TargetLowering::ConstraintWeight
Cpu0TargetLowering::getSingleConstraintMatchWeight(
    AsmOperandInfo &info, const char *constraint) const {
  ConstraintWeight weight = CW_Invalid;
  Value *CallOperandVal = info.CallOperandVal;
    // If we don't have a value, we can't do a match,
    // but allow it at the lowest weight.
  if (!CallOperandVal)
    return CW_Default;
  Type *type = CallOperandVal->getType();
  // Look at the constraint type.
  switch (*constraint) {
  default:
    weight = TargetLowering::getSingleConstraintMatchWeight(info, constraint);
```

(continues on next page)

(continued from previous page)

```
    break;
  case 'c': // $t9 for indirect jumps
    if (type->isIntegerTy())
      weight = CW_SpecificReg;
    break;
  case 'I': // signed 16 bit immediate
  case 'J': // integer zero
  case 'K': // unsigned 16 bit immediate
  case 'L': // signed 32 bit immediate where lower 16 bits are 0
  case 'N': // immediate in the range of -65535 to -1 (inclusive)
  case 'O': // signed 15 bit immediate (+- 16383)
  case 'P': // immediate in the range of 65535 to 1 (inclusive)
    if (isa<ConstantInt>(CallOperandVal))
      weight = CW_Constant;
    break;
  case 'R':
    weight = CW_Memory;
    break;
  }
  return weight;
}

/// This is a helper function to parse a physical register string and split it
/// into non-numeric and numeric parts (Prefix and Reg). The first boolean flag
/// that is returned indicates whether parsing was successful. The second flag
/// is true if the numeric part exists.
static std::pair<bool, bool>
parsePhysicalReg(const StringRef &C, std::string &Prefix,
                 unsigned long long &Reg) {
  if (C.front() != '{' || C.back() != '}')
    return std::make_pair(false, false);

  // Search for the first numeric character.
  StringRef::const_iterator I, B = C.begin() + 1, E = C.end() - 1;
  I = std::find_if(B, E, isdigit);

  Prefix.assign(B, I - B);

  // The second flag is set to false if no numeric characters were found.
  if (I == E)
    return std::make_pair(true, false);

  // Parse the numeric characters.
  return std::make_pair(!getAsUnsignedInteger(StringRef(I, E - I), 10, Reg),
                        true);
}

std::pair<unsigned, const TargetRegisterClass *> Cpu0TargetLowering::
parseRegForInlineAsmConstraint(const StringRef &C, MVT VT) const {
  const TargetRegisterClass *RC;
  std::string Prefix;
  unsigned long long Reg;
```

(continues on next page)

(continued from previous page)

```
  std::pair<bool, bool> R = parsePhysicalReg(C, Prefix, Reg);

  if (!R.first)
    return std::make_pair(0U, nullptr);
  if (!R.second)
    return std::make_pair(0U, nullptr);

 // Parse $0-$15.
  assert(Prefix == "$");
  RC = getRegClassFor((VT == MVT::Other) ? MVT::i32 : VT);

  assert(Reg < RC->getNumRegs());
  return std::make_pair(*(RC->begin() + Reg), RC);
}

/// Given a register class constraint, like 'r', if this corresponds directly
/// to an LLVM register class, return a register of 0 and the register class
/// pointer.
std::pair<unsigned, const TargetRegisterClass *>
Cpu0TargetLowering::getRegForInlineAsmConstraint(const TargetRegisterInfo *TRI,
                                                 StringRef Constraint,
                                                 MVT VT) const
{
  if (Constraint.size() == 1) {
    switch (Constraint[0]) {
    case 'r':
      if (VT == MVT::i32 || VT == MVT::i16 || VT == MVT::i8) {
        return std::make_pair(0U, &Cpu0::CPURegsRegClass);
      }
      if (VT == MVT::i64)
        return std::make_pair(0U, &Cpu0::CPURegsRegClass);
      // This will generate an error message
      return std::make_pair(0u, static_cast<const TargetRegisterClass*>(0));
    case 'c': // register suitable for indirect jump
      if (VT == MVT::i32)
        return std::make_pair((unsigned)Cpu0::T9, &Cpu0::CPURegsRegClass);
      assert(0 && "Unexpected type.");
    }
  }

  std::pair<unsigned, const TargetRegisterClass *> R;
  R = parseRegForInlineAsmConstraint(Constraint, VT);

  if (R.second)
    return R;

  return TargetLowering::getRegForInlineAsmConstraint(TRI, Constraint, VT);
}

/// LowerAsmOperandForConstraint - Lower the specified operand into the Ops
/// vector.  If it is invalid, don't add anything to Ops.
```

(continues on next page)

(continued from previous page)

```
void Cpu0TargetLowering::LowerAsmOperandForConstraint(SDValue Op,
                                                      std::string &Constraint,
                                                      std::vector<SDValue>&Ops,
                                                      SelectionDAG &DAG) const {
  SDLoc DL(Op);
  SDValue Result;

  // Only support length 1 constraints for now.
  if (Constraint.length() > 1) return;

  char ConstraintLetter = Constraint[0];
  switch (ConstraintLetter) {
  default: break; // This will fall through to the generic implementation
  case 'I': // Signed 16 bit constant
    // If this fails, the parent routine will give an error
    if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
      EVT Type = Op.getValueType();
      int64_t Val = C->getSExtValue();
      if (isInt<16>(Val)) {
        Result = DAG.getTargetConstant(Val, DL, Type);
        break;
      }
    }
    return;
  case 'J': // integer zero
    if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
      EVT Type = Op.getValueType();
      int64_t Val = C->getZExtValue();
      if (Val == 0) {
        Result = DAG.getTargetConstant(0, DL, Type);
        break;
      }
    }
    return;
  case 'K': // unsigned 16 bit immediate
    if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
      EVT Type = Op.getValueType();
      uint64_t Val = (uint64_t)C->getZExtValue();
      if (isUInt<16>(Val)) {
        Result = DAG.getTargetConstant(Val, DL, Type);
        break;
      }
    }
    return;
  case 'L': // signed 32 bit immediate where lower 16 bits are 0
    if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
      EVT Type = Op.getValueType();
      int64_t Val = C->getSExtValue();
      if ((isInt<32>(Val)) && ((Val & 0xffff) == 0)){
        Result = DAG.getTargetConstant(Val, DL, Type);
        break;
      }
```

(continues on next page)

(continued from previous page)

```
    }
    return;
  case 'N': // immediate in the range of -65535 to -1 (inclusive)
    if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
      EVT Type = Op.getValueType();
      int64_t Val = C->getSExtValue();
      if ((Val >= -65535) && (Val <= -1)) {
        Result = DAG.getTargetConstant(Val, DL, Type);
        break;
      }
    }
    return;
  case 'O': // signed 15 bit immediate
    if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
      EVT Type = Op.getValueType();
      int64_t Val = C->getSExtValue();
      if ((isInt<15>(Val))) {
        Result = DAG.getTargetConstant(Val, DL, Type);
        break;
      }
    }
    return;
  case 'P': // immediate in the range of 1 to 65535 (inclusive)
    if (ConstantSDNode *C = dyn_cast<ConstantSDNode>(Op)) {
      EVT Type = Op.getValueType();
      int64_t Val = C->getSExtValue();
      if ((Val <= 65535) && (Val >= 1)) {
        Result = DAG.getTargetConstant(Val, DL, Type);
        break;
      }
    }
    return;
  }

  if (Result.getNode()) {
    Ops.push_back(Result);
    return;
  }

  TargetLowering::LowerAsmOperandForConstraint(Op, Constraint, Ops, DAG);
}

bool Cpu0TargetLowering::isLegalAddressingMode(const DataLayout &DL,
                                               const AddrMode &AM, Type *Ty,
                                               unsigned AS, Instruction *I) const {
  // No global is ever allowed as a base.
  if (AM.BaseGV)
    return false;

  switch (AM.Scale) {
  case 0: // "r+i" or just "i", depending on HasBaseReg.
    break;
```

(continues on next page)

(continued from previous page)

```
  case 1:
    if (!AM.HasBaseReg) // allow "r+i".
      break;
    return false; // disallow "r+r" or "r+r+i".
  default:
    return false;
  }

  return true;
}
```

Same with backend structure, the structure of inline assembly can be divided by file name as Table: the structure of inline assembly.

Table 11.1: inline assembly functions

File	Function
Cpu0ISelLowering.cpp	inline asm DAG node create
Cpu0ISelDAGToDAG.cpp	save OP code
Cpu0AsmPrinter.cpp,	inline asm instructions printing
Cpu0InstrInfo.cpp	•

Except Cpu0ISelDAGToDAG.cpp, the other functions are same with backend's compile code. The Cpu0ISelLowering.cpp inline asm is explained after the result of running with ch11_2.cpp. Cpu0ISelDAGToDAG.cpp just save OP code in SelectInlineAsmMemoryOperand(). Since the the OP code is Cpu0 inline assembly instruction, no llvm IR DAG translation needed further, just save OP directly and return false to notify llvm system that Cpu0 backend has finished processing this inline assembly instruction.

Run Chapter11_2 with ch11_2.cpp will get the following result.

```
1-160-129-73:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch11_2.cpp -emit-llvm -o ch11_2.bc

1-160-129-73:input Jonathan$ ~/llvm/test/build/bin/
llvm-dis ch11_2.bc -o -
...
target triple = "mips-unknown-linux-gnu"

@g = global [3 x i32] [i32 1, i32 2, i32 3], align 4

; Function Attrs: nounwind
define i32 @_Z14inlineasm_adduv() #0 {
  %foo = alloca i32, align 4
  %bar = alloca i32, align 4
  store i32 10, i32* %foo, align 4
  store i32 15, i32* %bar, align 4
  %1 = load i32* %foo, align 4
  %2 = call i32 asm sideeffect "addu $0,$1,$2", "=r,r,r"(i32 %1, i32 15) #1,
  !srcloc !1
  store i32 %2, i32* %foo, align 4
  %3 = load i32* %foo, align 4
```

(continues on next page)

(continued from previous page)

```
  ret i32 %3
}

; Function Attrs: nounwind
define i32 @_Z18inlineasm_longlongv() #0 {
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %bar = alloca i64, align 8
  %p = alloca i32*, align 4
  %q = alloca i32*, align 4
  store i64 21474836486, i64* %bar, align 8
  %1 = bitcast i64* %bar to i32*
  store i32* %1, i32** %p, align 4
  %2 = load i32** %p, align 4
  %3 = call i32 asm sideeffect "ld $0,$1", "=r,*m"(i32* %2) #1, !srcloc !2
  store i32 %3, i32* %a, align 4
  %4 = load i32** %p, align 4
  %5 = getelementptr inbounds i32* %4, i32 1
  store i32* %5, i32** %q, align 4
  %6 = load i32** %q, align 4
  %7 = call i32 asm sideeffect "ld $0,$1", "=r,*m"(i32* %6) #1, !srcloc !3
  store i32 %7, i32* %b, align 4
  %8 = load i32* %a, align 4
  %9 = load i32* %b, align 4
  %10 = add nsw i32 %8, %9
  ret i32 %10
}

; Function Attrs: nounwind
define i32 @_Z20inlineasm_constraintv() #0 {
  %foo = alloca i32, align 4
  %n_5 = alloca i32, align 4
  %n5 = alloca i32, align 4
  %n0 = alloca i32, align 4
  %un5 = alloca i32, align 4
  %n65536 = alloca i32, align 4
  %n_65531 = alloca i32, align 4
  store i32 10, i32* %foo, align 4
  store i32 -5, i32* %n_5, align 4
  store i32 5, i32* %n5, align 4
  store i32 0, i32* %n0, align 4
  store i32 5, i32* %un5, align 4
  store i32 65536, i32* %n65536, align 4
  store i32 -65531, i32* %n_65531, align 4
  %1 = load i32* %foo, align 4
  %2 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,I"(i32 %1, i32 -5) #1,
  !srcloc !4
  store i32 %2, i32* %foo, align 4
  %3 = load i32* %foo, align 4
  %4 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,J"(i32 %3, i32 0) #1,
  !srcloc !5
  store i32 %4, i32* %foo, align 4
```

(continues on next page)

(continued from previous page)

```
  %5 = load i32* %foo, align 4
  %6 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,K"(i32 %5, i32 5) #1,
  !srcloc !6
  store i32 %6, i32* %foo, align 4
  %7 = load i32* %foo, align 4
  %8 = call i32 asm sideeffect "ori $0,$1,$2", "=r,r,L"(i32 %7, i32 65536) #1,
  !srcloc !7
  store i32 %8, i32* %foo, align 4
  %9 = load i32* %foo, align 4
  %10 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,N"(i32 %9, i32 -65531)
  #1, !srcloc !8
  store i32 %10, i32* %foo, align 4
  %11 = load i32* %foo, align 4
  %12 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,O"(i32 %11, i32 -5) #1,
  !srcloc !9
  store i32 %12, i32* %foo, align 4
  %13 = load i32* %foo, align 4
  %14 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,P"(i32 %13, i32 5) #1,
  !srcloc !10
  store i32 %14, i32* %foo, align 4
  %15 = load i32* %foo, align 4
  ret i32 %15
}

; Function Attrs: nounwind
define i32 @_Z13inlineasm_argii(i32 %u, i32 %v) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %w = alloca i32, align 4
  store i32 %u, i32* %1, align 4
  store i32 %v, i32* %2, align 4
  %3 = load i32* %1, align 4
  %4 = load i32* %2, align 4
  %5 = call i32 asm sideeffect "subu $0,$1,$2", "=r,r,r"(i32 %3, i32 %4) #1,
  !srcloc !11
  store i32 %5, i32* %w, align 4
  %6 = load i32* %w, align 4
  ret i32 %6
}

; Function Attrs: nounwind
define i32 @_Z16inlineasm_globalv() #0 {
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  %1 = call i32 asm sideeffect "ld $0,$1", "=r,*m"(i32* getelementptr inbounds
  ([3 x i32]* @g, i32 0, i32 2)) #1, !srcloc !12
  store i32 %1, i32* %c, align 4
  %2 = load i32* %c, align 4
  %3 = call i32 asm sideeffect "addiu $0,$1,1", "=r,r"(i32 %2) #1, !srcloc !13
  store i32 %3, i32* %d, align 4
  %4 = load i32* %d, align 4
  ret i32 %4
```

(continues on next page)

(continued from previous page)

```
}

; Function Attrs: nounwind
define i32 @_Z14test_inlineasmv() #0 {
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  %d = alloca i32, align 4
  %e = alloca i32, align 4
  %f = alloca i32, align 4
  %g = alloca i32, align 4
  %1 = call i32 @_Z14inlineasm_adduv()
  store i32 %1, i32* %a, align 4
  %2 = call i32 @_Z18inlineasm_longlongv()
  store i32 %2, i32* %b, align 4
  %3 = call i32 @_Z20inlineasm_constraintv()
  store i32 %3, i32* %c, align 4
  %4 = call i32 @_Z13inlineasm_argii(i32 1, i32 10)
  store i32 %4, i32* %d, align 4
  %5 = call i32 @_Z13inlineasm_argii(i32 6, i32 3)
  store i32 %5, i32* %e, align 4
  %6 = load i32* %e, align 4
  %7 = call i32 asm sideeffect "addiu $0,$1,1", "=r,r"(i32 %6) #1, !srcloc !14
  store i32 %7, i32* %f, align 4
  %8 = call i32 @_Z16inlineasm_globalv()
  store i32 %8, i32* %g, align 4
  %9 = load i32* %a, align 4
  %10 = load i32* %b, align 4
  %11 = add nsw i32 %9, %10
  %12 = load i32* %c, align 4
  %13 = add nsw i32 %11, %12
  %14 = load i32* %d, align 4
  %15 = add nsw i32 %13, %14
  %16 = load i32* %e, align 4
  %17 = add nsw i32 %15, %16
  %18 = load i32* %f, align 4
  %19 = add nsw i32 %17, %18
  %20 = load i32* %g, align 4
  %21 = add nsw i32 %19, %20
  ret i32 %21
}
...
1-160-129-73:input Jonathan$ ~/llvm/test/build/bin/llc
  -march=cpu0 -relocation-model=static -filetype=asm ch11_2.bc -o -
  .section .mdebug.abi32
  .previous
  .file "ch11_2.bc"
  .text
  .globl  _Z14inlineasm_adduv
  .align  2
  .type _Z14inlineasm_adduv,@function
  .ent  _Z14inlineasm_adduv     # @_Z14inlineasm_adduv
```

(continues on next page)

(continued from previous page)

```
_Z14inlineasm_adduv:
  .frame  $fp,16,$lr
  .mask   0x00001000,-4
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -16
  st  $fp, 12($sp)            # 4-byte Folded Spill
  addu  $fp, $sp, $zero
  addiu $2, $zero, 10
  st  $2, 8($fp)
  addiu $2, $zero, 15
  st  $2, 4($fp)
  ld  $3, 8($fp)
  #APP
  addu $2,$3,$2
  #NO_APP
  st  $2, 8($fp)
  addu  $sp, $fp, $zero
  ld  $fp, 12($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 16
  ret $lr
  nop
  .set  macro
  .set  reorder
  .end  _Z14inlineasm_adduv
$tmp3:
  .size _Z14inlineasm_adduv, ($tmp3)-_Z14inlineasm_adduv

  .globl  _Z18inlineasm_longlongv
  .align  2
  .type _Z18inlineasm_longlongv,@function
  .ent  _Z18inlineasm_longlongv # @_Z18inlineasm_longlongv
_Z18inlineasm_longlongv:
  .frame  $fp,32,$lr
  .mask   0x00001000,-4
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -32
  st  $fp, 28($sp)            # 4-byte Folded Spill
  addu  $fp, $sp, $zero
  addiu $2, $zero, 6
  st  $2, 12($fp)
  addiu $2, $zero, 5
  st  $2, 8($fp)
  addiu $2, $fp, 8
  st  $2, 4($fp)
  #APP
  ld $2,0($2)
  #NO_APP
  st  $2, 24($fp)
```

(continues on next page)

(continued from previous page)

```
  ld  $2, 4($fp)
  addiu $2, $2, 4
  st  $2, 0($fp)
  #APP
  ld $2,0($2)
  #NO_APP
  st  $2, 20($fp)
  ld  $3, 24($fp)
  addu  $2, $3, $2
  addu  $sp, $fp, $zero
  ld  $fp, 28($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 32
  ret $lr
  .set  macro
  .set  reorder
  .end  _Z18inlineasm_longlongv
$tmp7:
  .size _Z18inlineasm_longlongv, ($tmp7)-_Z18inlineasm_longlongv

  .globl  _Z20inlineasm_constraintv
  .align  2
  .type _Z20inlineasm_constraintv,@function
  .ent  _Z20inlineasm_constraintv # @_Z20inlineasm_constraintv
_Z20inlineasm_constraintv:
  .frame  $fp,32,$lr
  .mask   0x00001000,-4
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -32
  st  $fp, 28($sp)            # 4-byte Folded Spill
  addu  $fp, $sp, $zero
  addiu $2, $zero, 10
  st  $2, 24($fp)
  addiu $2, $zero, -5
  st  $2, 20($fp)
  addiu $2, $zero, 5
  st  $2, 16($fp)
  addiu $3, $zero, 0
  st  $3, 12($fp)
  st  $2, 8($fp)
  lui $2, 1
  st  $2, 4($fp)
  lui $2, 65535
  ori $2, $2, 5
  st  $2, 0($fp)
  ld  $2, 24($fp)
  #APP
  addiu $2,$2,-5
  #NO_APP
  st  $2, 24($fp)
  #APP
```

(continues on next page)

(continued from previous page)

```
  addiu $2,$2,0
  #NO_APP
  st  $2, 24($fp)
  #APP
  addiu $2,$2,5
  #NO_APP
  st  $2, 24($fp)
  #APP
  ori $2,$2,65536
  #NO_APP
  st  $2, 24($fp)
  #APP
  addiu $2,$2,-65531
  #NO_APP
  st  $2, 24($fp)
  #APP
  addiu $2,$2,-5
  #NO_APP
  st  $2, 24($fp)
  #APP
  addiu $2,$2,5
  #NO_APP
  st  $2, 24($fp)
  addu  $sp, $fp, $zero
  ld  $fp, 28($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 32
  ret $lr
  nop
  .set  macro
  .set  reorder
  .end  _Z20inlineasm_constraintv
$tmp11:
  .size _Z20inlineasm_constraintv, ($tmp11)-_Z20inlineasm_constraintv

  .globl  _Z13inlineasm_argii
  .align  2
  .type _Z13inlineasm_argii,@function
  .ent  _Z13inlineasm_argii     # @_Z13inlineasm_argii
_Z13inlineasm_argii:
  .frame  $fp,16,$lr
  .mask   0x00001000,-4
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -16
  st  $fp, 12($sp)            # 4-byte Folded Spill
  addu  $fp, $sp, $zero
  ld  $2, 16($fp)
  st  $2, 8($fp)
  ld  $2, 20($fp)
  st  $2, 4($fp)
  ld  $3, 8($fp)
```

(continues on next page)

(continued from previous page)

```
  #APP
  subu $2,$3,$2
  #NO_APP
  st  $2, 0($fp)
  addu  $sp, $fp, $zero
  ld  $fp, 12($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 16
  ret $lr
  nop
  .set  macro
  .set  reorder
  .end  _Z13inlineasm_argii
$tmp15:
  .size _Z13inlineasm_argii, ($tmp15)-_Z13inlineasm_argii

  .globl  _Z16inlineasm_globalv
  .align  2
  .type _Z16inlineasm_globalv,@function
  .ent  _Z16inlineasm_globalv   # @_Z16inlineasm_globalv
_Z16inlineasm_globalv:
  .frame  $fp,16,$lr
  .mask   0x00001000,-4
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -16
  st  $fp, 12($sp)            # 4-byte Folded Spill
  addu  $fp, $sp, $zero
  lui $2, %hi(g)
  ori $2, $2, %lo(g)
  addiu $2, $2, 8
  #APP
  ld $2,0($2)
  #NO_APP
  st  $2, 8($fp)
  #APP
  addiu $2,$2,1
  #NO_APP
  st  $2, 4($fp)
  addu  $sp, $fp, $zero
  ld  $fp, 12($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 16
  ret $lr
  nop
  .set  macro
  .set  reorder
  .end  _Z16inlineasm_globalv
$tmp19:
  .size _Z16inlineasm_globalv, ($tmp19)-_Z16inlineasm_globalv

  .globl  _Z14test_inlineasmv
  .align  2
```

(continues on next page)

(continued from previous page)

```
  .type _Z14test_inlineasmv,@function
  .ent  _Z14test_inlineasmv     # @_Z14test_inlineasmv
_Z14test_inlineasmv:
  .frame  $fp,48,$lr
  .mask   0x00005000,-4
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -48
  st  $lr, 44($sp)            # 4-byte Folded Spill
  st  $fp, 40($sp)            # 4-byte Folded Spill
  addu  $fp, $sp, $zero
  jsub  _Z14inlineasm_adduv
  nop
  st  $2, 36($fp)
  jsub  _Z18inlineasm_longlongv
  nop
  st  $2, 32($fp)
  jsub  _Z20inlineasm_constraintv
  nop
  st  $2, 28($fp)
  addiu $2, $zero, 10
  st  $2, 4($sp)
  addiu $2, $zero, 1
  st  $2, 0($sp)
  jsub  _Z13inlineasm_argii
  nop
  st  $2, 24($fp)
  addiu $2, $zero, 3
  st  $2, 4($sp)
  addiu $2, $zero, 6
  st  $2, 0($sp)
  jsub  _Z13inlineasm_argii
  nop
  st  $2, 20($fp)
  #APP
  addiu $2,$2,1
  #NO_APP
  st  $2, 16($fp)
  jsub  _Z16inlineasm_globalv
  nop
  st  $2, 12($fp)
  ld  $3, 32($fp)
  ld  $4, 36($fp)
  addu  $3, $4, $3
  ld  $4, 28($fp)
  addu  $3, $3, $4
  ld  $4, 24($fp)
  addu  $3, $3, $4
  ld  $4, 20($fp)
  addu  $3, $3, $4
  ld  $4, 16($fp)
```

(continues on next page)

(continued from previous page)

```
  addu  $3, $3, $4
  addu  $2, $3, $2
  addu  $sp, $fp, $zero
  ld  $fp, 40($sp)            # 4-byte Folded Reload
  ld  $lr, 44($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 48
  ret $lr
  nop
  .set  macro
  .set  reorder
  .end  _Z14test_inlineasmv
$tmp23:
  .size _Z14test_inlineasmv, ($tmp23)-_Z14test_inlineasmv

  .type g,@object               # @g
  .data
  .globl  g
  .align  2
g:
  .4byte  1                       # 0x1
  .4byte  2                       # 0x2
  .4byte  3                       # 0x3
  .size g, 12
```

Clang translates gcc style inline assembly __asm__ into llvm IR Inline Assembler Expressions first[3], then replace the variable registers of SSA form to physical registers during llc register allocation stage. From above example, functions LowerAsmOperandForConstraint() and getSingleConstraintMatchWeight() of Cpu0ISelLowering.cpp will create different range of const operand by I, J, K, L, N, O, or P, and register operand by r . For instance, the following __asm__ will create the llvm asm immediately after it.

```
__asm__ __volatile__("addiu %0,%1,%2"
                     :"=r"(foo) // 15
                     :"r"(foo), "I"(n_5)
                     );
```

```
%2 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,I"(i32 %1, i32 -5) #0, !srcloc !1
```

```
__asm__ __volatile__("addiu %0,%1,%2"
                     :"=r"(foo) // 15
                     :"r"(foo), "N"(n_65531)
                     );
```

```
%10 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,N"(i32 %9, i32 -65531) #0, !srcloc␣
↪!5
```

```
__asm__ __volatile__("addiu %0,%1,%2"
                     :"=r"(foo) // 15
                     :"r"(foo), "P"(un5)
                     );
```

[3] http://llvm.org/docs/LangRef.html#inline-assembler-expressions

```
%14 = call i32 asm sideeffect "addiu $0,$1,$2", "=r,r,P"(i32 %13, i32 5) #0, !srcloc !7
```

The r in __asm__ will generate register, %1, in llvm IR asm while I in __asm__ will generate const operand, -5, in llvm IR asm. Remind, the LowerAsmOperandForConstraint() limit the range of positive or negative const operand value to 16 bits since FL type immediate operand is 16 bits in Cpu0 instruction. So, the range of N is -65535 to -1 and the range of P is 65535 to 1. For any value out of the range, the code in LowerAsmOperandForConstraint() will treat it as error since FL instruction format has limitation of 16 bits.

CHAPTER

TWELVE

C++ SUPPORT

- *Exception handle*
- *Thread variable*
- *Atomic*

This chapter supports some C++ compiler features.

12.1 Exception handle

The Chapter11_2 can be built and run with the C++ polymorphism example code of ch12_inherit.cpp as follows,

lbdex/input/ch12_inherit.cpp

```
...
class CPolygon { // _ZTVN10__cxxabiv117__class_type_infoE for parent class
  ...
#ifdef COUT_TEST
 // generate IR nvoke, landing, resume and unreachable on iMac
    { cout << this->area() << endl; }
#else
    { printf("%d\n", this->area()); }
#endif
};
...
```

If using cout instead of printf in ch12_inherit.cpp, it won't generate exception handler IRs on Linux, whereas it will generate invoke, landing, resume and unreachable exception handler IRs on iMac. Example code, ch12_eh.cpp, which includes **try** and **catch** exception handler as the following will generate these exception handler IRs both on iMac and Linux.

lbdex/input/ch12_eh.cpp

```
class Ex1 {};
void throw_exception(int a, int b) {
  Ex1 ex1;

  if (a > b) {
    throw ex1;
  }
}

int test_try_catch() {
  try {
    throw_exception(2, 1);
  }
  catch(...) {
    return 1;
  }
  return 0;
}
```

```
JonathantekiiMac:input Jonathan$ clang -c ch12_eh.cpp -emit-llvm
-o ch12_eh.bc
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-dis ch12_eh.bc -o -
```

../lbdex/output/ch12_eh.ll

```
...
define dso_local i32 @_Z14test_try_catchv() #0 personality i8* bitcast (i32 (...
)* @__gxx_personality_v0 to i8*) {
entry:
  ...
  invoke void @_Z15throw_exceptionii(i32 signext 2, i32 signext 1)
        to label %invoke.cont unwind label %lpad

invoke.cont:                                      ; preds = %entry
  br label %try.cont

lpad:                                             ; preds = %entry
  %0 = landingpad { i8*, i32 }
          catch i8* null
  ...
}
...
```

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -filetype=asm ch12_eh.bc -o -
        .section .mdebug.abi32
        .previous
```

(continues on next page)

(continued from previous page)

```
        .file "ch12_eh.bc"
llc: /Users/Jonathan/llvm/test/llvm/lib/CodeGen/LiveVariables.cpp:133: void llvm::
LiveVariables::HandleVirtRegUse(unsigned int, llvm::MachineBasicBlock *, llvm
::MachineInstr *): Assertion `MRI->getVRegDef(reg) && "Register use before
def!"' failed.
```

A description for the C++ exception table formats can be found here[2]. About the IRs of LLVM exception handling, please reference here[1]. Chapter12_1 supports the llvm IRs of corresponding **try** and **catch** exception C++ keywords. It can compile ch12_eh.bc as follows,

lbdex/chapters/Chapter12_1/Cpu0ISelLowering.h

```
    /// If a physical register, this returns the register that receives the
    /// exception address on entry to an EH pad.
    Register
    getExceptionPointerRegister(const Constant *PersonalityFn) const override {
      return Cpu0::A0;
    }

    /// If a physical register, this returns the register that receives the
    /// exception typeid on entry to a landing pad.
    Register
    getExceptionSelectorRegister(const Constant *PersonalityFn) const override {
      return Cpu0::A1;
    }
```

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -filetype=asm ch12_eh.bc -o -
```

../lbdex/output/ch12_eh.cpu0.s

```
  .type  _Z14test_try_catchv,@function
  .ent  _Z14test_try_catchv             # @_Z14test_try_catchv
_Z14test_try_catchv:
  ...
$tmp0:
  addiu  $4, $zero, 2
  addiu  $5, $zero, 1
  jsub  _Z15throw_exceptionii
  nop
$tmp1:
# %bb.1:                                # %invoke.cont
  jmp  $BB1_4
$BB1_2:                                 # %lpad
$tmp2:
  st  $4, 16($fp)
  st  $5, 12($fp)
```

(continues on next page)

[2] http://itanium-cxx-abi.github.io/cxx-abi/exceptions.pdf
[1] http://llvm.org/docs/ExceptionHandling.html

(continued from previous page)

```
# %bb.3:                                # %catch
  ld  $4, 16($fp)
  jsub  __cxa_begin_catch
  nop
  addiu  $2, $zero, 1
  st  $2, 20($fp)
  jsub  __cxa_end_catch
  nop
  jmp  $BB1_5
$BB1_4:                                 # %try.cont
  addiu  $2, $zero, 0
  st  $2, 20($fp)
$BB1_5:                                 # %return
  ld  $2, 20($fp)
  ...
```

12.2 Thread variable

C++ support thread variable as the following file ch12_thread_var.cpp.

lbdex/input/ch12_thread_var.cpp

```
__thread int a = 0;
thread_local int b = 0; // need option -std=c++11
int test_thread_var()
{
    a = 2;
    return a;
}

int test_thread_var_2()
{
    b = 3;
    return b;
}
```

While global variable is a single instance shared by all threads in a process, thread variable has different instances for each different thread in a process. The same thread share the thread variable but different threads have their own thread variable with the same name[3].

To support thread variable, tlsgd, tlsldm, dtp_hi, dtp_lo, gottp, tp_hi and tp_lo in both evaluateRelocExpr() of Cpu0AsmParser.cpp and printImpl() of Cpu0MCExpr.cpp are needed, and the following code are required. Most of them are for relocation record handle and display since the thread variable created by OS or language library which support multi-threads programming.

[3] http://en.wikipedia.org/wiki/Thread-local_storage

lbdex/chapters/Chapter12_1/MCTargetDesc/Cpu0AsmBackend.cpp

```
const MCFixupKindInfo &Cpu0AsmBackend::
getFixupKindInfo(MCFixupKind Kind) const {
  unsigned JSUBReloRec = 0;
  if (HasLLD) {
    JSUBReloRec = MCFixupKindInfo::FKF_IsPCRel;
  }
  else {
    JSUBReloRec = MCFixupKindInfo::FKF_IsPCRel | MCFixupKindInfo::FKF_Constant;
  }
  const static MCFixupKindInfo Infos[Cpu0::NumTargetFixupKinds] = {
    // This table *must* be in same the order of fixup_* kinds in
    // Cpu0FixupKinds.h.
    //
    // name                          offset  bits  flags
```

```
    { "fixup_Cpu0_TLSGD",           0,      16,   0 },
    { "fixup_Cpu0_GOTTP",           0,      16,   0 },
    { "fixup_Cpu0_TP_HI",           0,      16,   0 },
    { "fixup_Cpu0_TP_LO",           0,      16,   0 },
    { "fixup_Cpu0_TLSLDM",          0,      16,   0 },
    { "fixup_Cpu0_DTP_HI",          0,      16,   0 },
    { "fixup_Cpu0_DTP_LO",          0,      16,   0 },
```

```
    ...
  };
  ...
}
```

lbdex/chapters/Chapter12_1/MCTargetDesc/Cpu0BaseInfo.h

```
namespace Cpu0II {
  /// Target Operand Flag enum.
  enum TOF {
    //===----------------------------------------------------------------------===//
    // Cpu0 Specific MachineOperand flags.
```

```
    /// MO_TLSGD - Represents the offset into the global offset table at which
    // the module ID and TSL block offset reside during execution (General
    // Dynamic TLS).
    MO_TLSGD,

    /// MO_TLSLDM - Represents the offset into the global offset table at which
    // the module ID and TSL block offset reside during execution (Local
    // Dynamic TLS).
    MO_TLSLDM,
    MO_DTP_HI,
    MO_DTP_LO,
```

(continues on next page)

(continued from previous page)

```
    /// MO_GOTTPREL - Represents the offset from the thread pointer (Initial
    // Exec TLS).
    MO_GOTTPREL,

    /// MO_TPREL_HI/LO - Represents the hi and low part of the offset from
    // the thread pointer (Local Exec TLS).
    MO_TP_HI,
    MO_TP_LO,
```

```
    ...
  };
  ...
}
```

lbdex/chapters/Chapter12_1/MCTargetDesc/Cpu0ELFObjectWriter.cpp

```
unsigned Cpu0ELFObjectWriter::getRelocType(MCContext &Ctx,
                                           const MCValue &Target,
                                           const MCFixup &Fixup,
                                           bool IsPCRel) const {
  // determine the type of the relocation
  unsigned Type = (unsigned)ELF::R_CPU0_NONE;
  unsigned Kind = (unsigned)Fixup.getKind();

  switch (Kind) {
```

```
  case Cpu0::fixup_Cpu0_TLSGD:
    Type = ELF::R_CPU0_TLS_GD;
    break;
  case Cpu0::fixup_Cpu0_GOTTPREL:
    Type = ELF::R_CPU0_TLS_GOTTPREL;
    break;
```

```
  ...
}
```

lbdex/chapters/Chapter12_1/MCTargetDesc/Cpu0FixupKinds.h

```
  enum Fixups {
```

```
    // resulting in - R_CPU0_TLS_GD.
    fixup_Cpu0_TLSGD,

    // resulting in - R_CPU0_TLS_GOTTPREL.
    fixup_Cpu0_GOTTPREL,

    // resulting in - R_CPU0_TLS_TPREL_HI16.
    fixup_Cpu0_TP_HI,
```

(continues on next page)

(continued from previous page)

```
    // resulting in - R_CPU0_TLS_TPREL_LO16.
    fixup_Cpu0_TP_LO,

    // resulting in - R_CPU0_TLS_LDM.
    fixup_Cpu0_TLSLDM,

    // resulting in - R_CPU0_TLS_DTP_HI16.
    fixup_Cpu0_DTP_HI,

    // resulting in - R_CPU0_TLS_DTP_LO16.
    fixup_Cpu0_DTP_LO,
```

```
  ...
};
```

lbdex/chapters/Chapter12_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
unsigned Cpu0MCCodeEmitter::
getExprOpValue(const MCExpr *Expr,SmallVectorImpl<MCFixup> &Fixups,
               const MCSubtargetInfo &STI) const {
```

```
    case Cpu0MCExpr::CEK_TLSGD:
      FixupKind = Cpu0::fixup_Cpu0_TLSGD;
      break;
    case Cpu0MCExpr::CEK_TLSLDM:
      FixupKind = Cpu0::fixup_Cpu0_TLSLDM;
      break;
    case Cpu0MCExpr::CEK_DTP_HI:
      FixupKind = Cpu0::fixup_Cpu0_DTP_HI;
      break;
    case Cpu0MCExpr::CEK_DTP_LO:
      FixupKind = Cpu0::fixup_Cpu0_DTP_LO;
      break;
    case Cpu0MCExpr::CEK_GOTTPREL:
      FixupKind = Cpu0::fixup_Cpu0_GOTTPREL;
      break;
    case Cpu0MCExpr::CEK_TP_HI:
      FixupKind = Cpu0::fixup_Cpu0_TP_HI;
      break;
    case Cpu0MCExpr::CEK_TP_LO:
      FixupKind = Cpu0::fixup_Cpu0_TP_LO;
      break;
```

```
  ...
}
```

lbdex/chapters/Chapter12_1/Cpu0InstrInfo.td

```
// TlsGd node is used to handle General Dynamic TLS
def Cpu0TlsGd : SDNode<"Cpu0ISD::TlsGd", SDTIntUnaryOp>;

// TpHi and TpLo nodes are used to handle Local Exec TLS
def Cpu0TpHi  : SDNode<"Cpu0ISD::TpHi", SDTIntUnaryOp>;
def Cpu0TpLo  : SDNode<"Cpu0ISD::TpLo", SDTIntUnaryOp>;
```

```
let Predicates = [Ch12_1] in {
def : Pat<(Cpu0Hi tglobaltlsaddr:$in), (LUi tglobaltlsaddr:$in)>;
}
```

```
let Predicates = [Ch12_1] in {
def : Pat<(Cpu0Lo tglobaltlsaddr:$in), (ORi ZERO, tglobaltlsaddr:$in)>;
}
```

```
let Predicates = [Ch12_1] in {
def : Pat<(add CPURegs:$hi, (Cpu0Lo tglobaltlsaddr:$lo)),
              (ORi CPURegs:$hi, tglobaltlsaddr:$lo)>;
}
```

```
let Predicates = [Ch12_1] in {
def : WrapperPat<tglobaltlsaddr, ORi, CPURegs>;
}
```

lbdex/chapters/Chapter12_1/Cpu0SelLowering.cpp

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
  setOperationAction(ISD::GlobalTLSAddress,   MVT::i32,   Custom);
```

```
  ...
}
```

```
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
  switch (Op.getOpcode())
  {
```

```
  case ISD::GlobalTLSAddress:   return lowerGlobalTLSAddress(Op, DAG);
```

```
    ...
  }
```

(continues on next page)

(continued from previous page)

```
  ...
}
```

```
SDValue Cpu0TargetLowering::
lowerGlobalTLSAddress(SDValue Op, SelectionDAG &DAG) const
{
  // If the relocation model is PIC, use the General Dynamic TLS Model or
  // Local Dynamic TLS model, otherwise use the Initial Exec or
  // Local Exec TLS Model.

  GlobalAddressSDNode *GA = cast<GlobalAddressSDNode>(Op);
  if (DAG.getTarget().Options.EmulatedTLS)
    return LowerToTLSEmulatedModel(GA, DAG);

  SDLoc DL(GA);
  const GlobalValue *GV = GA->getGlobal();
  EVT PtrVT = getPointerTy(DAG.getDataLayout());

  TLSModel::Model model = getTargetMachine().getTLSModel(GV);

  if (model == TLSModel::GeneralDynamic || model == TLSModel::LocalDynamic) {
    // General Dynamic and Local Dynamic TLS Model.
    unsigned Flag = (model == TLSModel::LocalDynamic) ? Cpu0II::MO_TLSLDM
                                                      : Cpu0II::MO_TLSGD;

    SDValue TGA = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0, Flag);
    SDValue Argument = DAG.getNode(Cpu0ISD::Wrapper, DL, PtrVT,
                                   getGlobalReg(DAG, PtrVT), TGA);
    unsigned PtrSize = PtrVT.getSizeInBits();
    IntegerType *PtrTy = Type::getIntNTy(*DAG.getContext(), PtrSize);

    SDValue TlsGetAddr = DAG.getExternalSymbol("__tls_get_addr", PtrVT);

    ArgListTy Args;
    ArgListEntry Entry;
    Entry.Node = Argument;
    Entry.Ty = PtrTy;
    Args.push_back(Entry);

    TargetLowering::CallLoweringInfo CLI(DAG);
    CLI.setDebugLoc(DL).setChain(DAG.getEntryNode())
      .setCallee(CallingConv::C, PtrTy, TlsGetAddr, std::move(Args));
    std::pair<SDValue, SDValue> CallResult = LowerCallTo(CLI);

    SDValue Ret = CallResult.first;

    if (model != TLSModel::LocalDynamic)
      return Ret;

    SDValue TGAHi = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0,
                                               Cpu0II::MO_DTP_HI);
    SDValue Hi = DAG.getNode(Cpu0ISD::Hi, DL, PtrVT, TGAHi);
```

(continues on next page)

(continued from previous page)

```
    SDValue TGALo = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0,
                                               Cpu0II::MO_DTP_LO);
    SDValue Lo = DAG.getNode(Cpu0ISD::Lo, DL, PtrVT, TGALo);
    SDValue Add = DAG.getNode(ISD::ADD, DL, PtrVT, Hi, Ret);
    return DAG.getNode(ISD::ADD, DL, PtrVT, Add, Lo);
  }

  SDValue Offset;
  if (model == TLSModel::InitialExec) {
    // Initial Exec TLS Model
    SDValue TGA = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0,
                                             Cpu0II::MO_GOTTPREL);
    TGA = DAG.getNode(Cpu0ISD::Wrapper, DL, PtrVT, getGlobalReg(DAG, PtrVT),
                      TGA);
    Offset =
        DAG.getLoad(PtrVT, DL, DAG.getEntryNode(), TGA, MachinePointerInfo());
  } else {
    // Local Exec TLS Model
    assert(model == TLSModel::LocalExec);
    SDValue TGAHi = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0,
                                               Cpu0II::MO_TP_HI);
    SDValue TGALo = DAG.getTargetGlobalAddress(GV, DL, PtrVT, 0,
                                               Cpu0II::MO_TP_LO);
    SDValue Hi = DAG.getNode(Cpu0ISD::Hi, DL, PtrVT, TGAHi);
    SDValue Lo = DAG.getNode(Cpu0ISD::Lo, DL, PtrVT, TGALo);
    Offset = DAG.getNode(ISD::ADD, DL, PtrVT, Hi, Lo);
  }
  return Offset;
}
```

lbdex/chapters/Chapter12_1/Cpu0ISelLowering.h

```
    SDValue lowerGlobalTLSAddress(SDValue Op, SelectionDAG &DAG) const;
```

lbdex/chapters/Chapter12_1/Cpu0MCInstLower.cpp

```
MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,
                                              MachineOperandType MOTy,
                                              unsigned Offset) const {
  MCSymbolRefExpr::VariantKind Kind = MCSymbolRefExpr::VK_None;
  Cpu0MCExpr::Cpu0ExprKind TargetKind = Cpu0MCExpr::CEK_None;
  const MCSymbol *Symbol;

  switch(MO.getTargetFlags()) {
```

```
  case Cpu0II::MO_TLSGD:
    TargetKind = Cpu0MCExpr::CEK_TLSGD;
    break;
  case Cpu0II::MO_TLSLDM:
```

(continues on next page)

(continued from previous page)

```
    TargetKind = Cpu0MCExpr::CEK_TLSLDM;
    break;
  case Cpu0II::MO_DTP_HI:
    TargetKind = Cpu0MCExpr::CEK_DTP_HI;
    break;
  case Cpu0II::MO_DTP_LO:
    TargetKind = Cpu0MCExpr::CEK_DTP_LO;
    break;
  case Cpu0II::MO_GOTTPREL:
    TargetKind = Cpu0MCExpr::CEK_GOTTPREL;
    break;
  case Cpu0II::MO_TP_HI:
    TargetKind = Cpu0MCExpr::CEK_TP_HI;
    break;
  case Cpu0II::MO_TP_LO:
    TargetKind = Cpu0MCExpr::CEK_TP_LO;
    break;
```

```
  ...
  }
  ...
}
```

```
JonathantekiiMac:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch12_thread_var.cpp -emit-llvm -std=c++11 -o ch12_thread_var.bc
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llvm-dis ch12_thread_var.bc -o -
```

../lbdex/output/ch12_thread_var.ll

```
...
@a = dso_local thread_local global i32 0, align 4
@b = dso_local thread_local global i32 0, align 4

; Function Attrs: noinline nounwind optnone mustprogress
define dso_local i32 @_Z15test_thread_varv() #0 {
entry:
  store i32 2, i32* @a, align 4
  %0 = load i32, i32* @a, align 4
  ret i32 %0
}

; Function Attrs: noinline nounwind optnone mustprogress
define dso_local i32 @_Z17test_thread_var_2v() #0 {
entry:
  store i32 3, i32* @b, align 4
  %0 = load i32, i32* @b, align 4
  ret i32 %0
}
...
```

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch12_thread_var.bc
-o ch12_thread_var.cpu0.pic.s
JonathantekiiMac:input Jonathan$ cat ch12_thread_var.cpu0.pic.s
```

../lbdex/output/ch12_thread_var.cpu0.pic.s

```
...
  .ent  _Z15test_thread_varv            # @_Z15test_thread_varv
_Z15test_thread_varv:
  ...
  ori  $4, $gp, %tlsldm(a)
  ld  $t9, %call16(__tls_get_addr)($gp)
  jalr  $t9
  nop
  ld  $gp, 8($fp)
  lui  $3, %dtp_hi(a)
  addu  $2, $3, $2
  ori  $2, $2, %dtp_lo(a)
  ...
```

In pic mode, the __thread variable access by call function __tls_get_addr with the address of thread variable. The c++11 standard thread_local variable is accessed by calling function _ZTW1b which also call the function __tls_get_addr to get the thread_local variable address. In static mode, the thread variable is accessed by getting address of thread variables "a" and "b" with machine instructions as follows,

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=static -filetype=asm
ch12_thread_var.bc -o ch12_thread_var.cpu0.static.s
JonathantekiiMac:input Jonathan$ cat ch12_thread_var.cpu0.static.s
```

../lbdex/output/ch12_thread_var.cpu0.static.s

```
...
lui  $2, %tp_hi(a)
ori  $2, $2, %tp_lo(a)
...
lui  $2, %tp_hi(b)
ori  $2, $2, %tp_lo(b)
...
```

While Mips uses rdhwr instruction to access thread varaible as below, Cpu0 access thread varaible without inventing any new instruction. The thread variables are keeped in thread varaible memory location which accessed through %tp_hi and %tp_lo, and furthermore, this section of memory is protected through kernel mode program. Thus, the user mode program cannot access this area of memory and no space to breathe for hack program.

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=mips -relocation-model=static -filetype=asm
ch12_thread_var.bc -o -
  ...
```

(continues on next page)

(continued from previous page)

```
lui $1, %tprel_hi(a)
ori $1, $1, %tprel_lo(a)
.set  push
.set  mips32r2
rdhwr $3, $29
.set  pop
addu  $1, $3, $1
addiu $2, $zero, 2
sw  $2, 0($1)
addiu $2, $zero, 2
...
```

In static mode, the thread variable is similar to global variable. In general, they are same in IRs, DAGs and machine code translation. List them in the following tables. You can check them with debug option enabled.

Table 12.1: The DAGs of thread varaible of static mode

stage	DAG
IR	load i32* @a, align 4;
Legalized selection DAG	(add Cpu0ISD::Hi Cpu0ISD::Lo);
Instruction Selection	ori $2, $zero, %tp_lo(a);
•	lui $3, %tp_hi(a);
•	addu $3, $3, $2;

Table 12.2: The DAGs of local_thread varaible of static mode

stage	DAG
IR	ret i32* @b;
Legalized selection DAG	%0=(add Cpu0ISD::Hi Cpu0ISD::Lo);...
Instruction Selection	ori $2, $zero, %tp_lo(a);
•	lui $3, %tp_hi(a);
•	addu $3, $3, $2;

12.3 Atomic

In tradition, C uses different API which provided by OS or library to support multi-thread programming. For example, posix thread API on unix/linux, MS windows API, ..., etc. In order to achieve synchronization to solve race condition between threads, OS provide their own lock or semaphore functions to programmer. But this solution is OS dependent. After c++11, programmer can use atomic to program and run the code on every different platform since the thread and atomic are part of c++ standard. Beside of portability, the other important benifit is the compiler can generate high performance code by the target hardware instructions rather than couting on lock() function only[5][6][7].

[5] https://en.wikipedia.org/wiki/Memory_model_%28programming%29
[6] http://stackoverflow.com/questions/6319146/c11-introduced-a-standardized-memory-model-what-does-it-mean-and-how-is-it-g
[7] http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/

Compare-and-swap operation[9] is used to implement synchronization primitives like semaphores and mutexes, as well as more sophisticated wait-free and lock-free[8] algorithms. For atomic variables, Mips lock instructions, ll and sc, to solve the race condition problem. The semantic as follows,

- Load-link returns the current value of a memory location, while a subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to that location since the load-link. If any updates have occurred, the store-conditional is guaranteed to fail, even if the value read by the load-link has since been restored.[10]

Mips sync and ARM/X86-64 memory-barrier instruction[11] provide synchronization mechanism very efficiently in some scenarios.

Mips sync[12] is explained simply as follows,

A Simple Description: SYNC affects only uncached and cached coherent loads and stores. The loads and stores that occur prior to the SYNC must be completed before the loads and stores after the SYNC are allowed to start. Loads are completed when the destination register is written. Stores are completed when the stored value is visible to every other processor in the system.

In order to support atomic in C++ and java, llvm provides the atomic IRs and memory ordering here[13][14]. C++ memory order is explained and exampled here[15]. The chapter 19 of book DPC++[16] explains the memory ordering better and I add the related code fragment of lbdex/input/atomics.ll to it for explanation as follows,

- memory_order::relaxed

Read and write operations can be re-ordered before or after the operation with no restrictions. There are no ordering guarantees.

```
define i8 @load_i8_unordered(i8* %mem) {
; CHECK-LABEL: load_i8_unordered
; CHECK: ll
; CHECK: sc
; CHECK-NOT: sync
  %val = load atomic i8, i8* %mem unordered, align 1
  ret i8 %val
}
```

No **sync** from CodeGen instructions above.

- memory_order::acquire

Read and write operations appearing after the operation in the program must occur after it (i.e., they cannot be re-ordered before the operation).

[9] https://en.wikipedia.org/wiki/Compare-and-swap

[8] An algorithm is wait-free if every operation has a bound on the number of steps the algorithm will take before the operation completes. In other words, wait-free algorithm has no starvation. Lock-freedom allows individual threads to starve but guarantees system-wide throughput. An algorithm is lock-free if, when the program threads are run for a sufficiently long time, at least one of the threads makes progress (for some sensible definition of progress). All wait-free algorithms are lock-free. In particular, if one thread is suspended, then a lock-free algorithm guarantees that the remaining threads can still make progress. Hence, if two threads can contend for the same mutex lock or spinlock, then the algorithm is not lock-free. (If we suspend one thread that holds the lock, then the second thread will block.) https://en.wikipedia.org/wiki/Non-blocking_algorithm

[10] https://en.wikipedia.org/wiki/Load-link/store-conditional

[11] https://en.wikipedia.org/wiki/Memory_barrier

[12] From page A-158, it is same with ARM's barrier that all instructions before SYNC are completed before issuing the instructions after SYNC. Page 167 (A-155) of https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf

[13] http://llvm.org/docs/Atomics.html

[14] http://llvm.org/docs/LangRef.html#ordering

[15] https://en.cppreference.com/w/cpp/atomic/memory_order

[16] Section "The memory_order Enumeration Class" which include figure 19-10 of book https://link.springer.com/book/10.1007/978-1-4842-5574-2

```
define i32 @load_i32_acquire(i32* %mem) {
; CHECK-LABEL: load_i32_acquire
; CHECK: ll
; CHECK: sc
  %val = load atomic i32, i32* %mem acquire, align 4
; CHECK: sync
  ret i32 %val
}
```

Sync guarantees "load atomic" complete before the next R/W (Read/Write). All writes in other threads that release the same atomic variable are visible in the current thread.

- memory_order::release

Read and write operations appearing before the operation in the program must occur before it (i.e., they cannot be re-ordered after the operation), and preceding write operations are guaranteed to be visible to other program instances which have been synchronized by a corresponding acquire operation (i.e., an atomic operation using the same variable and memory_order::acquire or a barrier function).

```
define void @store_i32_release(i32* %mem) {
; CHECK-LABEL: store_i32_release
; CHECK: sync
; CHECK: ll
; CHECK: sc
  store atomic i32 42, i32* %mem release, align 4
  ret void
}
```

Sync guarantees preceding R/W complete before "store atomic". Mips' ll and sc guarantee that "store atomic release" is visible to other processors.

- memory_order::acq_rel

The operation acts as both an acquire and a release. Read and write operations cannot be re-ordered around the operation, and preceding writes must be made visible as previously described for memory_order::release.

```
define i32 @cas_strong_i32_acqrel_acquire(i32* %mem) {
; CHECK-LABEL: cas_strong_i32_acqrel_acquire
; CHECK: ll
; CHECK: sc
  %val = cmpxchg i32* %mem, i32 0, i32 1 acq_rel acquire
; CHECK: sync
  %loaded = extractvalue { i32, i1} %val, 0
  ret i32 %loaded
}
```

Sync guarantees preceding R/W complete before "cmpxchg". Other processors' preceding write operations are guaranteed to be visible to this "cmpxchg acquire" (Mips's ll and sc quarantee it).

- memory_order::seq_cst

The operation acts as an acquire, release, or both depending on whether it is a read, write, or read-modify-write operation, respectively. All operations with this memory order are observed in a sequentially consistent order.

```
define i8 @cas_strong_i8_sc_sc(i8* %mem) {
; CHECK-LABEL: cas_strong_i8_sc_sc
```

(continues on next page)

(continued from previous page)

```
; CHECK: sync
; CHECK: ll
; CHECK: sc
  %val = cmpxchg i8* %mem, i8 0, i8 1 seq_cst seq_cst
; CHECK: sync
  %loaded = extractvalue { i8, i1} %val, 0
  ret i8 %loaded
}
```

First sync guarantees preceding R/W complete before "cmpxchg seq_cst" and visible to "cmpxchg seq_cst". For seq_cst, a store performs a release operation. Which means "cmpxchg seq_cst" are visible to other threads/processors that acquire the same atomic variable as the memory_order_release definition. Mips' ll and sc quarantees this feature of "cmpxchg seq_cst". Second Sync guarantees "cmpxchg seq_cst" complete before the next R/W.

There are several restrictions on which memory orders are supported by each operation. Fig. 12.1 (from book Figure 19-10) summarizes which combinations are valid.

Functions	Supported memory_order Values				
	relaxed	acquire	release	acq_rel	seq_cst
load	✓	✓	✗	✗	✓
store	✓	✗	✓	✗	✓
exchange compare_exchange_* fetch_*	✓	✓	✓	✓	✓
fence	✓	✓	✓	✓	✓

Figure 19-10. *Supporting atomic operations with* `memory_order`

Fig. 12.1: Supporting atomic operations with memory_order

Load operations do not write values to memory and are therefore incompatible with release semantics. Similarly, store operations do not read values from memory and are therefore incompatible with acquire semantics. The remaining read-modify-write atomic operations and fences are compatible with all memory orderings[Page 582, 16].

Note: C++ memory_order_consume

The C++ memory model additionally includes memory_order::consume, with similar behavior to memory_order::acquire. however, the C++17 standard discourages its use, noting that its definition is being revised. its inclusion in dpC++ has therefore been postponed to a future version.

Feature	Standard C++	SYCL / DPC++
Atomic Objects	`std::atomic`	Not available.
Atomic References	`std::atomic_ref` (C++20 onwards)	`sycl::atomic_ref`

Memory Orders	`relaxed`
	`consume`
	`acquire`
	`release`
	`scq_rel`
	`seq_cst`
		`relaxed`
		
		`acquire`
		`release`
		`scq_rel`
		`seq_cst`

Memory Scopes	Not available.
	
	Behavior of atomics and fences matches DPC++ `system` scope.
		`work_item`
		`sub_group`
		`work_group`
		`device`
		`system`

Fences	`std::atomic_thread_fence`	`sycl::atomic_fence`
Barriers	`std::barrier` (C++20 onwards)	`nd_item::barrier` `sub_group::barrier`
Address Spaces	All memory is in a single (host) address space.	Host Device (Global) Device (Local) Device (Private) Shared (USM)

Figure 19-9. *Comparing standard C++ and SYCL/DPC++ memory models*

Fig. 12.2: Comparing standard C++ and SYCL/DPC++ memory models

For a few years now, compilers have treated consume as a synonym for acquire[17].

The current expectation is that the replacement facility will rely on core memory model and atomics definitions very similar to what's currently there. Since memory_order_consume does have a profound impact on the memory model, removing this text would allow drastic simplification, but conversely would make it very difficult to add anything along the lines of memory_order_consume back in later, especially if the standard evolves in the meantime, as expected. Thus we are not proposing to remove the current wording[18].

The following test files are extracted from memory_checks() of clang/test/Sema/atomic-ops.c. The "__c11_atomic_xxx" builtin-functions from clang defined in clang/include/clang/Basic/Builtins.def. Complie with clang will get the results same with Fig. 12.1. Clang compile memory_order_consume to the same result of memory_order_acquire.

lbdex/input/ch12_sema_atomic-ops.c

```
// clang -S ch12_sema_atomic-ops.c -emit-llvm -o -
// Uses /opt/homebrew/opt/llvm/bin/clang in macOS.

#include <stdatomic.h>

// From memory_checks() of Sema/atomic-ops.c
void memory_checks(_Atomic(int) *Ap, int *p, int val) {
  (void)__c11_atomic_load(Ap, memory_order_relaxed);
  (void)__c11_atomic_load(Ap, memory_order_acquire);
  (void)__c11_atomic_load(Ap, memory_order_consume);
  (void)__c11_atomic_load(Ap, memory_order_release); // expected-warning {{memory order␣
↪argument to atomic operation is invalid}}
  (void)__c11_atomic_load(Ap, memory_order_acq_rel); // expected-warning {{memory order␣
↪argument to atomic operation is invalid}}
  (void)__c11_atomic_load(Ap, memory_order_seq_cst);
  (void)__c11_atomic_load(Ap, val);
  (void)__c11_atomic_load(Ap, -1); // expected-warning {{memory order argument to atomic␣
↪operation is invalid}}
  (void)__c11_atomic_load(Ap, 42); // expected-warning {{memory order argument to atomic␣
↪operation is invalid}}

  (void)__c11_atomic_store(Ap, val, memory_order_relaxed);
  (void)__c11_atomic_store(Ap, val, memory_order_acquire); // expected-warning {{memory␣
↪order argument to atomic operation is invalid}}
  (void)__c11_atomic_store(Ap, val, memory_order_consume); // expected-warning {{memory␣
↪order argument to atomic operation is invalid}}
  (void)__c11_atomic_store(Ap, val, memory_order_release);
  (void)__c11_atomic_store(Ap, val, memory_order_acq_rel); // expected-warning {{memory␣
↪order argument to atomic operation is invalid}}
  (void)__c11_atomic_store(Ap, val, memory_order_seq_cst);

  (void)__c11_atomic_exchange(Ap, val, memory_order_relaxed);
  (void)__c11_atomic_exchange(Ap, val, memory_order_acquire);
  (void)__c11_atomic_exchange(Ap, val, memory_order_consume);
  (void)__c11_atomic_exchange(Ap, val, memory_order_release);
  (void)__c11_atomic_exchange(Ap, val, memory_order_acq_rel);
```

(continues on next page)

[17] https://stackoverflow.com/questions/65336409/what-does-memory-order-consume-really-do
[18] https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0371r1.html

(continued from previous page)

```
  (void)__c11_atomic_exchange(Ap, val, memory_order_seq_cst);

  (void)__c11_atomic_compare_exchange_strong(Ap, p, val, memory_order_relaxed, memory_
↪order_relaxed);
  (void)__c11_atomic_compare_exchange_strong(Ap, p, val, memory_order_acquire, memory_
↪order_relaxed);
  (void)__c11_atomic_compare_exchange_strong(Ap, p, val, memory_order_consume, memory_
↪order_relaxed);
  (void)__c11_atomic_compare_exchange_strong(Ap, p, val, memory_order_release, memory_
↪order_relaxed);
  (void)__c11_atomic_compare_exchange_strong(Ap, p, val, memory_order_acq_rel, memory_
↪order_relaxed);
  (void)__c11_atomic_compare_exchange_strong(Ap, p, val, memory_order_seq_cst, memory_
↪order_relaxed);

  (void)__c11_atomic_compare_exchange_weak(Ap, p, val, memory_order_relaxed, memory_
↪order_relaxed);
  (void)__c11_atomic_compare_exchange_weak(Ap, p, val, memory_order_acquire, memory_
↪order_relaxed);
  (void)__c11_atomic_compare_exchange_weak(Ap, p, val, memory_order_consume, memory_
↪order_relaxed);
  (void)__c11_atomic_compare_exchange_weak(Ap, p, val, memory_order_release, memory_
↪order_relaxed);
  (void)__c11_atomic_compare_exchange_weak(Ap, p, val, memory_order_acq_rel, memory_
↪order_relaxed);
  (void)__c11_atomic_compare_exchange_weak(Ap, p, val, memory_order_seq_cst, memory_
↪order_relaxed);

  atomic_thread_fence(memory_order_relaxed);
  atomic_thread_fence(memory_order_acquire);
  atomic_thread_fence(memory_order_consume); // For a few years now, compilers have␣
↪treated consume as a synonym for acquire.
  atomic_thread_fence(memory_order_release);
  atomic_thread_fence(memory_order_acq_rel);
  atomic_thread_fence(memory_order_seq_cst);
  atomic_signal_fence(memory_order_seq_cst);
}
```

lbdex/input/ch12_sema_atomic-fetch.c

```
// clang -S ch12_sema_atomic-fetch.c -emit-llvm -o -
// Uses /opt/homebrew/opt/llvm/bin/clang in macOS.

#include <stdatomic.h>

//#define WANT_COMPILE_FAIL

// From __c11_atomic_fetch_xxx of memory_checks() of Sema/atomic-ops.c
void memory_checks(_Atomic(int) *Ap, int *p, int val) {
  (void)__c11_atomic_fetch_add(Ap, 1, memory_order_relaxed);
```

(continues on next page)

(continued from previous page)

```
  (void)__c11_atomic_fetch_add(Ap, 1, memory_order_acquire);
  (void)__c11_atomic_fetch_add(Ap, 1, memory_order_consume);
  (void)__c11_atomic_fetch_add(Ap, 1, memory_order_release);
  (void)__c11_atomic_fetch_add(Ap, 1, memory_order_acq_rel);
  (void)__c11_atomic_fetch_add(Ap, 1, memory_order_seq_cst);

#ifdef WANT_COMPILE_FAIL // fail to compile:
  (void)__c11_atomic_fetch_add(
      (struct Incomplete * _Atomic *)0, // expected-error {{incomplete type 'struct␣
↪Incomplete'}}
      1, memory_order_seq_cst);
#endif

  (void)__c11_atomic_init(Ap, val);
  (void)__c11_atomic_init(Ap, val);
  (void)__c11_atomic_init(Ap, val);
  (void)__c11_atomic_init(Ap, val);
  (void)__c11_atomic_init(Ap, val);
  (void)__c11_atomic_init(Ap, val);

  (void)__c11_atomic_fetch_sub(Ap, val, memory_order_relaxed);
  (void)__c11_atomic_fetch_sub(Ap, val, memory_order_acquire);
  (void)__c11_atomic_fetch_sub(Ap, val, memory_order_consume);
  (void)__c11_atomic_fetch_sub(Ap, val, memory_order_release);
  (void)__c11_atomic_fetch_sub(Ap, val, memory_order_acq_rel);
  (void)__c11_atomic_fetch_sub(Ap, val, memory_order_seq_cst);

  (void)__c11_atomic_fetch_and(Ap, val, memory_order_relaxed);
  (void)__c11_atomic_fetch_and(Ap, val, memory_order_acquire);
  (void)__c11_atomic_fetch_and(Ap, val, memory_order_consume);
  (void)__c11_atomic_fetch_and(Ap, val, memory_order_release);
  (void)__c11_atomic_fetch_and(Ap, val, memory_order_acq_rel);
  (void)__c11_atomic_fetch_and(Ap, val, memory_order_seq_cst);

  (void)__c11_atomic_fetch_or(Ap, val, memory_order_relaxed);
  (void)__c11_atomic_fetch_or(Ap, val, memory_order_acquire);
  (void)__c11_atomic_fetch_or(Ap, val, memory_order_consume);
  (void)__c11_atomic_fetch_or(Ap, val, memory_order_release);
  (void)__c11_atomic_fetch_or(Ap, val, memory_order_acq_rel);
  (void)__c11_atomic_fetch_or(Ap, val, memory_order_seq_cst);

  (void)__c11_atomic_fetch_xor(Ap, val, memory_order_relaxed);
  (void)__c11_atomic_fetch_xor(Ap, val, memory_order_acquire);
  (void)__c11_atomic_fetch_xor(Ap, val, memory_order_consume);
  (void)__c11_atomic_fetch_xor(Ap, val, memory_order_release);
  (void)__c11_atomic_fetch_xor(Ap, val, memory_order_acq_rel);
  (void)__c11_atomic_fetch_xor(Ap, val, memory_order_seq_cst);

  (void)__c11_atomic_fetch_nand(Ap, val, memory_order_relaxed);
  (void)__c11_atomic_fetch_nand(Ap, val, memory_order_acquire);
  (void)__c11_atomic_fetch_nand(Ap, val, memory_order_consume);
  (void)__c11_atomic_fetch_nand(Ap, val, memory_order_release);
```

(continues on next page)

(continued from previous page)

```
  (void)__c11_atomic_fetch_nand(Ap, val, memory_order_acq_rel);
  (void)__c11_atomic_fetch_nand(Ap, val, memory_order_seq_cst);

  (void)__c11_atomic_fetch_min(Ap, val, memory_order_relaxed);
  (void)__c11_atomic_fetch_min(Ap, val, memory_order_acquire);
  (void)__c11_atomic_fetch_min(Ap, val, memory_order_consume);
  (void)__c11_atomic_fetch_min(Ap, val, memory_order_release);
  (void)__c11_atomic_fetch_min(Ap, val, memory_order_acq_rel);
  (void)__c11_atomic_fetch_min(Ap, val, memory_order_seq_cst);

  (void)__c11_atomic_fetch_max(Ap, val, memory_order_relaxed);
  (void)__c11_atomic_fetch_max(Ap, val, memory_order_acquire);
  (void)__c11_atomic_fetch_max(Ap, val, memory_order_consume);
  (void)__c11_atomic_fetch_max(Ap, val, memory_order_release);
  (void)__c11_atomic_fetch_max(Ap, val, memory_order_acq_rel);
  (void)__c11_atomic_fetch_max(Ap, val, memory_order_seq_cst);
}
```

Table 12.3: Atomic related between clang's builtin and llvm ir

clang's builtin	llvm ir
__c11_atomic_load	load atomic
__c11_atomic_store	store atomic
__c11_atomic_exchange_xxx	cmpxchg
atomic_thread_fence	fence
__c11_atomic_fetch_xxx	atomicrmw xxx

C++ atomic functions supported by calling implemented functions from C++ libary. These implemented functions evently call "__c11_atomic_xxx" builtin-functions for implementation. So, "__c11_atomic_xxx" listed in above providing lower-level of better performance functions for C++ programmers. An example as follows,

lbdex/input/ch12_c++_atomics.cpp

```
// clang -S ch12_c++_atomics.cpp -emit-llvm -o -
// Uses /opt/homebrew/opt/llvm/bin/clang in macOS.

#include <atomic>

std::atomic<bool> winner (false);

int test_atomics() {
  int count = 0;
  bool res = winner.exchange(true);
  if (res) count++;

  return count;
}
```

For supporting llvm atomic IRs, the following code added to Chapter12_1.

lbdex/chapters/Chapter12_1/Disassembler/Cpu0Disassembler.cpp

```
static DecodeStatus DecodeMem(MCInst &Inst,
                              unsigned Insn,
                              uint64_t Address,
                              const void *Decoder) {
```

```
  if(Inst.getOpcode() == Cpu0::SC){
    Inst.addOperand(MCOperand::createReg(Reg));
  }
```

```
  ...
}
```

lbdex/chapters/Chapter12_1/Cpu0InstrInfo.td

```
def SDT_Sync             : SDTypeProfile<0, 1, [SDTCisVT<0, i32>]>;
```

```
def Cpu0Sync : SDNode<"Cpu0ISD::Sync", SDT_Sync, [SDNPHasChain]>;
```

```
def PtrRC : Operand<iPTR> {
  let MIOperandInfo = (ops ptr_rc);
  let DecoderMethod = "DecodeCPURegsRegisterClass";
}
```

```
// Atomic instructions with 2 source operands (ATOMIC_SWAP & ATOMIC_LOAD_*).
class Atomic2Ops<PatFrag Op, RegisterClass DRC> :
  PseudoSE<(outs DRC:$dst), (ins PtrRC:$ptr, DRC:$incr),
           [(set DRC:$dst, (Op iPTR:$ptr, DRC:$incr))]>;

// Atomic Compare & Swap.
class AtomicCmpSwap<PatFrag Op, RegisterClass DRC> :
  PseudoSE<(outs DRC:$dst), (ins PtrRC:$ptr, DRC:$cmp, DRC:$swap),
           [(set DRC:$dst, (Op iPTR:$ptr, DRC:$cmp, DRC:$swap))]>;
```

```
class LLBase<bits<8> Opc, string opstring, RegisterClass RC, Operand Mem> :
  FMem<Opc, (outs RC:$ra), (ins Mem:$addr),
       !strconcat(opstring, "\t$ra, $addr"), [], IILoad> {
  let mayLoad = 1;
}

class SCBase<bits<8> Opc, string opstring, RegisterOperand RO, Operand Mem> :
  FMem<Opc, (outs RO:$dst), (ins RO:$ra, Mem:$addr),
       !strconcat(opstring, "\t$ra, $addr"), [], IIStore> {
  let mayStore = 1;
  let Constraints = "$ra = $dst";
}
```

```
let Predicates = [Ch12_1] in {
let usesCustomInserter = 1 in {
  def ATOMIC_LOAD_ADD_I8   : Atomic2Ops<atomic_load_add_8, CPURegs>;
  def ATOMIC_LOAD_ADD_I16  : Atomic2Ops<atomic_load_add_16, CPURegs>;
  def ATOMIC_LOAD_ADD_I32  : Atomic2Ops<atomic_load_add_32, CPURegs>;
  def ATOMIC_LOAD_SUB_I8   : Atomic2Ops<atomic_load_sub_8, CPURegs>;
  def ATOMIC_LOAD_SUB_I16  : Atomic2Ops<atomic_load_sub_16, CPURegs>;
  def ATOMIC_LOAD_SUB_I32  : Atomic2Ops<atomic_load_sub_32, CPURegs>;
  def ATOMIC_LOAD_AND_I8   : Atomic2Ops<atomic_load_and_8, CPURegs>;
  def ATOMIC_LOAD_AND_I16  : Atomic2Ops<atomic_load_and_16, CPURegs>;
  def ATOMIC_LOAD_AND_I32  : Atomic2Ops<atomic_load_and_32, CPURegs>;
  def ATOMIC_LOAD_OR_I8    : Atomic2Ops<atomic_load_or_8, CPURegs>;
  def ATOMIC_LOAD_OR_I16   : Atomic2Ops<atomic_load_or_16, CPURegs>;
  def ATOMIC_LOAD_OR_I32   : Atomic2Ops<atomic_load_or_32, CPURegs>;
  def ATOMIC_LOAD_XOR_I8   : Atomic2Ops<atomic_load_xor_8, CPURegs>;
  def ATOMIC_LOAD_XOR_I16  : Atomic2Ops<atomic_load_xor_16, CPURegs>;
  def ATOMIC_LOAD_XOR_I32  : Atomic2Ops<atomic_load_xor_32, CPURegs>;
  def ATOMIC_LOAD_NAND_I8  : Atomic2Ops<atomic_load_nand_8, CPURegs>;
  def ATOMIC_LOAD_NAND_I16 : Atomic2Ops<atomic_load_nand_16, CPURegs>;
  def ATOMIC_LOAD_NAND_I32 : Atomic2Ops<atomic_load_nand_32, CPURegs>;

  def ATOMIC_SWAP_I8       : Atomic2Ops<atomic_swap_8, CPURegs>;
  def ATOMIC_SWAP_I16      : Atomic2Ops<atomic_swap_16, CPURegs>;
  def ATOMIC_SWAP_I32      : Atomic2Ops<atomic_swap_32, CPURegs>;

  def ATOMIC_CMP_SWAP_I8   : AtomicCmpSwap<atomic_cmp_swap_8, CPURegs>;
  def ATOMIC_CMP_SWAP_I16  : AtomicCmpSwap<atomic_cmp_swap_16, CPURegs>;
  def ATOMIC_CMP_SWAP_I32  : AtomicCmpSwap<atomic_cmp_swap_32, CPURegs>;
}
}
```

```
let Predicates = [Ch12_1] in {
let hasSideEffects = 1 in
def SYNC : Cpu0Inst<(outs), (ins i32imm:$stype), "sync $stype",
                    [(Cpu0Sync imm:$stype)], NoItinerary, FrmOther>
{
  bits<5> stype;
  let Opcode = 0x60;
  let Inst{25-11} = 0;
  let Inst{10-6} = stype;
  let Inst{5-0} = 0;
}
}
```

```
/// Load-linked, Store-conditional
def LL      : LLBase<0x61, "ll", CPURegs, mem>;
def SC      : SCBase<0x62, "sc", RegisterOperand<CPURegs>, mem>;
```

```
def : Cpu0InstAlias<"sync",
                    (SYNC 0), 1>;
```

lbdex/chapters/Chapter12_1/Cpu0ISelLowering.h

```
    MachineBasicBlock *
    EmitInstrWithCustomInserter(MachineInstr &MI,
                                MachineBasicBlock *MBB) const override;
```

```
    SDValue lowerATOMIC_FENCE(SDValue Op, SelectionDAG& DAG) const;
```

```
    bool shouldInsertFencesForAtomic(const Instruction *I) const override {
      return true;
    }

    /// Emit a sign-extension using shl/sra appropriately.
    MachineBasicBlock *emitSignExtendToI32InReg(MachineInstr &MI,
                                                MachineBasicBlock *BB,
                                                unsigned Size, unsigned DstReg,
                                                unsigned SrcRec) const;
    MachineBasicBlock *emitAtomicBinary(MachineInstr &MI, MachineBasicBlock *BB,
                    unsigned Size, unsigned BinOpcode, bool Nand = false) const;
    MachineBasicBlock *emitAtomicBinaryPartword(MachineInstr &MI,
                    MachineBasicBlock *BB, unsigned Size, unsigned BinOpcode,
                    bool Nand = false) const;
    MachineBasicBlock *emitAtomicCmpSwap(MachineInstr &MI,
                                 MachineBasicBlock *BB, unsigned Size) const;
    MachineBasicBlock *emitAtomicCmpSwapPartword(MachineInstr &MI,
                                 MachineBasicBlock *BB, unsigned Size) const;
```

lbdex/chapters/Chapter12_1/Cpu0SelLowering.cpp

```
const char *Cpu0TargetLowering::getTargetNodeName(unsigned Opcode) const {
```

```
  case Cpu0ISD::Sync:              return "Cpu0ISD::Sync";
```

```
  ...
}
```

```
Cpu0TargetLowering::Cpu0TargetLowering(const Cpu0TargetMachine &TM,
                                       const Cpu0Subtarget &STI)
    : TargetLowering(TM), Subtarget(STI), ABI(TM.getABI()) {
```

```
  setOperationAction(ISD::ATOMIC_LOAD,       MVT::i32,    Expand);
  setOperationAction(ISD::ATOMIC_LOAD,       MVT::i64,    Expand);
  setOperationAction(ISD::ATOMIC_STORE,      MVT::i32,    Expand);
  setOperationAction(ISD::ATOMIC_STORE,      MVT::i64,    Expand);
```

```
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
```

(continues on next page)

(continued from previous page)

```
  switch (Op.getOpcode())
  {
```

```
  case ISD::ATOMIC_FENCE:       return lowerATOMIC_FENCE(Op, DAG);
```

```
  ...
}
```

```
MachineBasicBlock *
Cpu0TargetLowering::EmitInstrWithCustomInserter(MachineInstr &MI,
                                                MachineBasicBlock *BB) const {
  switch (MI.getOpcode()) {
  default:
    llvm_unreachable("Unexpected instr type to insert");
  case Cpu0::ATOMIC_LOAD_ADD_I8:
    return emitAtomicBinaryPartword(MI, BB, 1, Cpu0::ADDu);
  case Cpu0::ATOMIC_LOAD_ADD_I16:
    return emitAtomicBinaryPartword(MI, BB, 2, Cpu0::ADDu);
  case Cpu0::ATOMIC_LOAD_ADD_I32:
    return emitAtomicBinary(MI, BB, 4, Cpu0::ADDu);

  case Cpu0::ATOMIC_LOAD_AND_I8:
    return emitAtomicBinaryPartword(MI, BB, 1, Cpu0::AND);
  case Cpu0::ATOMIC_LOAD_AND_I16:
    return emitAtomicBinaryPartword(MI, BB, 2, Cpu0::AND);
  case Cpu0::ATOMIC_LOAD_AND_I32:
    return emitAtomicBinary(MI, BB, 4, Cpu0::AND);

  case Cpu0::ATOMIC_LOAD_OR_I8:
    return emitAtomicBinaryPartword(MI, BB, 1, Cpu0::OR);
  case Cpu0::ATOMIC_LOAD_OR_I16:
    return emitAtomicBinaryPartword(MI, BB, 2, Cpu0::OR);
  case Cpu0::ATOMIC_LOAD_OR_I32:
    return emitAtomicBinary(MI, BB, 4, Cpu0::OR);

  case Cpu0::ATOMIC_LOAD_XOR_I8:
    return emitAtomicBinaryPartword(MI, BB, 1, Cpu0::XOR);
  case Cpu0::ATOMIC_LOAD_XOR_I16:
    return emitAtomicBinaryPartword(MI, BB, 2, Cpu0::XOR);
  case Cpu0::ATOMIC_LOAD_XOR_I32:
    return emitAtomicBinary(MI, BB, 4, Cpu0::XOR);

  case Cpu0::ATOMIC_LOAD_NAND_I8:
    return emitAtomicBinaryPartword(MI, BB, 1, 0, true);
  case Cpu0::ATOMIC_LOAD_NAND_I16:
    return emitAtomicBinaryPartword(MI, BB, 2, 0, true);
  case Cpu0::ATOMIC_LOAD_NAND_I32:
    return emitAtomicBinary(MI, BB, 4, 0, true);

  case Cpu0::ATOMIC_LOAD_SUB_I8:
    return emitAtomicBinaryPartword(MI, BB, 1, Cpu0::SUBu);
```

(continues on next page)

(continued from previous page)

```
  case Cpu0::ATOMIC_LOAD_SUB_I16:
    return emitAtomicBinaryPartword(MI, BB, 2, Cpu0::SUBu);
  case Cpu0::ATOMIC_LOAD_SUB_I32:
    return emitAtomicBinary(MI, BB, 4, Cpu0::SUBu);

  case Cpu0::ATOMIC_SWAP_I8:
    return emitAtomicBinaryPartword(MI, BB, 1, 0);
  case Cpu0::ATOMIC_SWAP_I16:
    return emitAtomicBinaryPartword(MI, BB, 2, 0);
  case Cpu0::ATOMIC_SWAP_I32:
    return emitAtomicBinary(MI, BB, 4, 0);

  case Cpu0::ATOMIC_CMP_SWAP_I8:
    return emitAtomicCmpSwapPartword(MI, BB, 1);
  case Cpu0::ATOMIC_CMP_SWAP_I16:
    return emitAtomicCmpSwapPartword(MI, BB, 2);
  case Cpu0::ATOMIC_CMP_SWAP_I32:
    return emitAtomicCmpSwap(MI, BB, 4);
  }
}

// This function also handles Cpu0::ATOMIC_SWAP_I32 (when BinOpcode == 0), and
// Cpu0::ATOMIC_LOAD_NAND_I32 (when Nand == true)
MachineBasicBlock *Cpu0TargetLowering::emitAtomicBinary(
    MachineInstr &MI, MachineBasicBlock *BB, unsigned Size, unsigned BinOpcode,
    bool Nand) const {
  assert((Size == 4) && "Unsupported size for EmitAtomicBinary.");

  MachineFunction *MF = BB->getParent();
  MachineRegisterInfo &RegInfo = MF->getRegInfo();
  const TargetRegisterClass *RC = getRegClassFor(MVT::getIntegerVT(Size * 8));
  const TargetInstrInfo *TII = Subtarget.getInstrInfo();
  DebugLoc DL = MI.getDebugLoc();
  unsigned LL, SC, AND, XOR, ZERO, BEQ;

  LL = Cpu0::LL;
  SC = Cpu0::SC;
  AND = Cpu0::AND;
  XOR = Cpu0::XOR;
  ZERO = Cpu0::ZERO;
  BEQ = Cpu0::BEQ;

  unsigned OldVal = MI.getOperand(0).getReg();
  unsigned Ptr = MI.getOperand(1).getReg();
  unsigned Incr = MI.getOperand(2).getReg();

  unsigned StoreVal = RegInfo.createVirtualRegister(RC);
  unsigned AndRes = RegInfo.createVirtualRegister(RC);
  unsigned AndRes2 = RegInfo.createVirtualRegister(RC);
  unsigned Success = RegInfo.createVirtualRegister(RC);

  // insert new blocks after the current block
```

(continues on next page)

(continued from previous page)

```
  const BasicBlock *LLVM_BB = BB->getBasicBlock();
  MachineBasicBlock *loopMBB = MF->CreateMachineBasicBlock(LLVM_BB);
  MachineBasicBlock *exitMBB = MF->CreateMachineBasicBlock(LLVM_BB);
  MachineFunction::iterator It = ++BB->getIterator();
  MF->insert(It, loopMBB);
  MF->insert(It, exitMBB);

  // Transfer the remainder of BB and its successor edges to exitMBB.
  exitMBB->splice(exitMBB->begin(), BB,
                  std::next(MachineBasicBlock::iterator(MI)), BB->end());
  exitMBB->transferSuccessorsAndUpdatePHIs(BB);

  //  thisMBB:
  //    ...
  //    fallthrough --> loopMBB
  BB->addSuccessor(loopMBB);
  loopMBB->addSuccessor(loopMBB);
  loopMBB->addSuccessor(exitMBB);

  //  loopMBB:
  //    ll oldval, 0(ptr)
  //    <binop> storeval, oldval, incr
  //    sc success, storeval, 0(ptr)
  //    beq success, $0, loopMBB
  BB = loopMBB;
  BuildMI(BB, DL, TII->get(LL), OldVal).addReg(Ptr).addImm(0);
  if (Nand) {
    //  and andres, oldval, incr
    //  xor storeval, $0, andres
    //  xor storeval2, $0, storeval
    BuildMI(BB, DL, TII->get(AND), AndRes).addReg(OldVal).addReg(Incr);
    BuildMI(BB, DL, TII->get(XOR), StoreVal).addReg(ZERO).addReg(AndRes);
    BuildMI(BB, DL, TII->get(XOR), AndRes2).addReg(ZERO).addReg(AndRes);
  } else if (BinOpcode) {
    //  <binop> storeval, oldval, incr
    BuildMI(BB, DL, TII->get(BinOpcode), StoreVal).addReg(OldVal).addReg(Incr);
  } else {
    StoreVal = Incr;
  }
  BuildMI(BB, DL, TII->get(SC), Success).addReg(StoreVal).addReg(Ptr).addImm(0);
  BuildMI(BB, DL, TII->get(BEQ)).addReg(Success).addReg(ZERO).addMBB(loopMBB);

  MI.eraseFromParent(); // The instruction is gone now.

  return exitMBB;
}

MachineBasicBlock *Cpu0TargetLowering::emitSignExtendToI32InReg(
    MachineInstr &MI, MachineBasicBlock *BB, unsigned Size, unsigned DstReg,
    unsigned SrcReg) const {
  const TargetInstrInfo *TII = Subtarget.getInstrInfo();
  DebugLoc DL = MI.getDebugLoc();
```

(continues on next page)

(continued from previous page)

```
  MachineFunction *MF = BB->getParent();
  MachineRegisterInfo &RegInfo = MF->getRegInfo();
  const TargetRegisterClass *RC = getRegClassFor(MVT::i32);
  unsigned ScrReg = RegInfo.createVirtualRegister(RC);

  assert(Size < 32);
  int64_t ShiftImm = 32 - (Size * 8);

  BuildMI(BB, DL, TII->get(Cpu0::SHL), ScrReg).addReg(SrcReg).addImm(ShiftImm);
  BuildMI(BB, DL, TII->get(Cpu0::SRA), DstReg).addReg(ScrReg).addImm(ShiftImm);

  return BB;
}

MachineBasicBlock *Cpu0TargetLowering::emitAtomicBinaryPartword(
    MachineInstr &MI, MachineBasicBlock *BB, unsigned Size, unsigned BinOpcode,
    bool Nand) const {
  assert((Size == 1 || Size == 2) &&
         "Unsupported size for EmitAtomicBinaryPartial.");

  MachineFunction *MF = BB->getParent();
  MachineRegisterInfo &RegInfo = MF->getRegInfo();
  const TargetRegisterClass *RC = getRegClassFor(MVT::i32);
  const TargetInstrInfo *TII = Subtarget.getInstrInfo();
  DebugLoc DL = MI.getDebugLoc();

  unsigned Dest = MI.getOperand(0).getReg();
  unsigned Ptr = MI.getOperand(1).getReg();
  unsigned Incr = MI.getOperand(2).getReg();

  unsigned AlignedAddr = RegInfo.createVirtualRegister(RC);
  unsigned ShiftAmt = RegInfo.createVirtualRegister(RC);
  unsigned Mask = RegInfo.createVirtualRegister(RC);
  unsigned Mask2 = RegInfo.createVirtualRegister(RC);
  unsigned Mask3 = RegInfo.createVirtualRegister(RC);
  unsigned NewVal = RegInfo.createVirtualRegister(RC);
  unsigned OldVal = RegInfo.createVirtualRegister(RC);
  unsigned Incr2 = RegInfo.createVirtualRegister(RC);
  unsigned MaskLSB2 = RegInfo.createVirtualRegister(RC);
  unsigned PtrLSB2 = RegInfo.createVirtualRegister(RC);
  unsigned MaskUpper = RegInfo.createVirtualRegister(RC);
  unsigned AndRes = RegInfo.createVirtualRegister(RC);
  unsigned BinOpRes = RegInfo.createVirtualRegister(RC);
  unsigned BinOpRes2 = RegInfo.createVirtualRegister(RC);
  unsigned MaskedOldVal0 = RegInfo.createVirtualRegister(RC);
  unsigned StoreVal = RegInfo.createVirtualRegister(RC);
  unsigned MaskedOldVal1 = RegInfo.createVirtualRegister(RC);
  unsigned SrlRes = RegInfo.createVirtualRegister(RC);
  unsigned Success = RegInfo.createVirtualRegister(RC);

  // insert new blocks after the current block
```

(continues on next page)

(continued from previous page)

```
const BasicBlock *LLVM_BB = BB->getBasicBlock();
MachineBasicBlock *loopMBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineBasicBlock *sinkMBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineBasicBlock *exitMBB = MF->CreateMachineBasicBlock(LLVM_BB);
MachineFunction::iterator It = ++BB->getIterator();

MF->insert(It, loopMBB);
MF->insert(It, sinkMBB);
MF->insert(It, exitMBB);

// Transfer the remainder of BB and its successor edges to exitMBB.
exitMBB->splice(exitMBB->begin(), BB,
                std::next(MachineBasicBlock::iterator(MI)), BB->end());
exitMBB->transferSuccessorsAndUpdatePHIs(BB);

BB->addSuccessor(loopMBB);
loopMBB->addSuccessor(loopMBB);
loopMBB->addSuccessor(sinkMBB);
sinkMBB->addSuccessor(exitMBB);

//  thisMBB:
//    addiu   masklsb2,$0,-4                # 0xfffffffc
//    and     alignedaddr,ptr,masklsb2
//    andi    ptrlsb2,ptr,3
//    sll     shiftamt,ptrlsb2,3
//    ori     maskupper,$0,255               # 0xff
//    sll     mask,maskupper,shiftamt
//    xor     mask2,$0,mask
//    xor     mask3,$0,mask2
//    sll     incr2,incr,shiftamt

int64_t MaskImm = (Size == 1) ? 255 : 65535;
BuildMI(BB, DL, TII->get(Cpu0::ADDiu), MaskLSB2)
  .addReg(Cpu0::ZERO).addImm(-4);
BuildMI(BB, DL, TII->get(Cpu0::AND), AlignedAddr)
  .addReg(Ptr).addReg(MaskLSB2);
BuildMI(BB, DL, TII->get(Cpu0::ANDi), PtrLSB2).addReg(Ptr).addImm(3);
if (Subtarget.isLittle()) {
  BuildMI(BB, DL, TII->get(Cpu0::SHL), ShiftAmt).addReg(PtrLSB2).addImm(3);
} else {
  unsigned Off = RegInfo.createVirtualRegister(RC);
  BuildMI(BB, DL, TII->get(Cpu0::XORi), Off)
    .addReg(PtrLSB2).addImm((Size == 1) ? 3 : 2);
  BuildMI(BB, DL, TII->get(Cpu0::SHL), ShiftAmt).addReg(Off).addImm(3);
}
BuildMI(BB, DL, TII->get(Cpu0::ORi), MaskUpper)
  .addReg(Cpu0::ZERO).addImm(MaskImm);
BuildMI(BB, DL, TII->get(Cpu0::SHLV), Mask)
  .addReg(MaskUpper).addReg(ShiftAmt);
BuildMI(BB, DL, TII->get(Cpu0::XOR), Mask2).addReg(Cpu0::ZERO).addReg(Mask);
BuildMI(BB, DL, TII->get(Cpu0::XOR), Mask3).addReg(Cpu0::ZERO).addReg(Mask2);
BuildMI(BB, DL, TII->get(Cpu0::SHLV), Incr2).addReg(Incr).addReg(ShiftAmt);
```

(continues on next page)

(continued from previous page)

```
// atomic.load.binop
// loopMBB:
//   ll      oldval,0(alignedaddr)
//   binop   binopres,oldval,incr2
//   and     newval,binopres,mask
//   and     maskedoldval0,oldval,mask3
//   or      storeval,maskedoldval0,newval
//   sc      success,storeval,0(alignedaddr)
//   beq     success,$0,loopMBB

// atomic.swap
// loopMBB:
//   ll      oldval,0(alignedaddr)
//   and     newval,incr2,mask
//   and     maskedoldval0,oldval,mask3
//   or      storeval,maskedoldval0,newval
//   sc      success,storeval,0(alignedaddr)
//   beq     success,$0,loopMBB

BB = loopMBB;
unsigned LL = Cpu0::LL;
BuildMI(BB, DL, TII->get(LL), OldVal).addReg(AlignedAddr).addImm(0);
if (Nand) {
  //  and andres, oldval, incr2
  //  xor binopres,  $0, andres
  //  xor binopres2, $0, binopres
  //  and newval, binopres, mask
  BuildMI(BB, DL, TII->get(Cpu0::AND), AndRes).addReg(OldVal).addReg(Incr2);
  BuildMI(BB, DL, TII->get(Cpu0::XOR), BinOpRes)
    .addReg(Cpu0::ZERO).addReg(AndRes);
  BuildMI(BB, DL, TII->get(Cpu0::XOR), BinOpRes2)
    .addReg(Cpu0::ZERO).addReg(BinOpRes);
  BuildMI(BB, DL, TII->get(Cpu0::AND), NewVal).addReg(BinOpRes).addReg(Mask);
} else if (BinOpcode) {
  //  <binop> binopres, oldval, incr2
  //  and newval, binopres, mask
  BuildMI(BB, DL, TII->get(BinOpcode), BinOpRes).addReg(OldVal).addReg(Incr2);
  BuildMI(BB, DL, TII->get(Cpu0::AND), NewVal).addReg(BinOpRes).addReg(Mask);
} else { // atomic.swap
  //  and newval, incr2, mask
  BuildMI(BB, DL, TII->get(Cpu0::AND), NewVal).addReg(Incr2).addReg(Mask);
}

BuildMI(BB, DL, TII->get(Cpu0::AND), MaskedOldVal0)
  .addReg(OldVal).addReg(Mask2);
BuildMI(BB, DL, TII->get(Cpu0::OR), StoreVal)
  .addReg(MaskedOldVal0).addReg(NewVal);
unsigned SC = Cpu0::SC;
BuildMI(BB, DL, TII->get(SC), Success)
  .addReg(StoreVal).addReg(AlignedAddr).addImm(0);
BuildMI(BB, DL, TII->get(Cpu0::BEQ))
```

(continues on next page)

(continued from previous page)

```
    .addReg(Success).addReg(Cpu0::ZERO).addMBB(loopMBB);

  //  sinkMBB:
  //    and     maskedoldval1,oldval,mask
  //    srl     srlres,maskedoldval1,shiftamt
  //    sign_extend dest,srlres
  BB = sinkMBB;

  BuildMI(BB, DL, TII->get(Cpu0::AND), MaskedOldVal1)
    .addReg(OldVal).addReg(Mask);
  BuildMI(BB, DL, TII->get(Cpu0::SHRV), SrlRes)
      .addReg(MaskedOldVal1).addReg(ShiftAmt);
  BB = emitSignExtendToI32InReg(MI, BB, Size, Dest, SrlRes);

  MI.eraseFromParent(); // The instruction is gone now.

  return exitMBB;
}

MachineBasicBlock * Cpu0TargetLowering::emitAtomicCmpSwap(MachineInstr &MI,
                                                          MachineBasicBlock *BB,
                                                          unsigned Size) const {
  assert((Size == 4) && "Unsupported size for EmitAtomicCmpSwap.");

  MachineFunction *MF = BB->getParent();
  MachineRegisterInfo &RegInfo = MF->getRegInfo();
  const TargetRegisterClass *RC = getRegClassFor(MVT::getIntegerVT(Size * 8));
  const TargetInstrInfo *TII = Subtarget.getInstrInfo();
  DebugLoc DL = MI.getDebugLoc();
  unsigned LL, SC, ZERO, BNE, BEQ;

  LL = Cpu0::LL;
  SC = Cpu0::SC;
  ZERO = Cpu0::ZERO;
  BNE = Cpu0::BNE;
  BEQ = Cpu0::BEQ;

  unsigned Dest    = MI.getOperand(0).getReg();
  unsigned Ptr     = MI.getOperand(1).getReg();
  unsigned OldVal  = MI.getOperand(2).getReg();
  unsigned NewVal  = MI.getOperand(3).getReg();

  unsigned Success = RegInfo.createVirtualRegister(RC);

  // insert new blocks after the current block
  const BasicBlock *LLVM_BB = BB->getBasicBlock();
  MachineBasicBlock *loop1MBB = MF->CreateMachineBasicBlock(LLVM_BB);
  MachineBasicBlock *loop2MBB = MF->CreateMachineBasicBlock(LLVM_BB);
  MachineBasicBlock *exitMBB = MF->CreateMachineBasicBlock(LLVM_BB);
  MachineFunction::iterator It = ++BB->getIterator();

  MF->insert(It, loop1MBB);
```

(continues on next page)

(continued from previous page)

```
  MF->insert(It, loop2MBB);
  MF->insert(It, exitMBB);

  // Transfer the remainder of BB and its successor edges to exitMBB.
  exitMBB->splice(exitMBB->begin(), BB,
                  std::next(MachineBasicBlock::iterator(MI)), BB->end());
  exitMBB->transferSuccessorsAndUpdatePHIs(BB);

  //  thisMBB:
  //    ...
  //    fallthrough --> loop1MBB
  BB->addSuccessor(loop1MBB);
  loop1MBB->addSuccessor(exitMBB);
  loop1MBB->addSuccessor(loop2MBB);
  loop2MBB->addSuccessor(loop1MBB);
  loop2MBB->addSuccessor(exitMBB);

  // loop1MBB:
  //   ll dest, 0(ptr)
  //   bne dest, oldval, exitMBB
  BB = loop1MBB;
  BuildMI(BB, DL, TII->get(LL), Dest).addReg(Ptr).addImm(0);
  BuildMI(BB, DL, TII->get(BNE))
    .addReg(Dest).addReg(OldVal).addMBB(exitMBB);

  // loop2MBB:
  //   sc success, newval, 0(ptr)
  //   beq success, $0, loop1MBB
  BB = loop2MBB;
  BuildMI(BB, DL, TII->get(SC), Success)
    .addReg(NewVal).addReg(Ptr).addImm(0);
  BuildMI(BB, DL, TII->get(BEQ))
    .addReg(Success).addReg(ZERO).addMBB(loop1MBB);

  MI.eraseFromParent(); // The instruction is gone now.

  return exitMBB;
}

MachineBasicBlock *
Cpu0TargetLowering::emitAtomicCmpSwapPartword(MachineInstr &MI,
                                              MachineBasicBlock *BB,
                                              unsigned Size) const {
  assert((Size == 1 || Size == 2) &&
      "Unsupported size for EmitAtomicCmpSwapPartial.");

  MachineFunction *MF = BB->getParent();
  MachineRegisterInfo &RegInfo = MF->getRegInfo();
  const TargetRegisterClass *RC = getRegClassFor(MVT::i32);
  const TargetInstrInfo *TII = Subtarget.getInstrInfo();
  DebugLoc DL = MI.getDebugLoc();
```

(continues on next page)

(continued from previous page)

```
  unsigned Dest    = MI.getOperand(0).getReg();
  unsigned Ptr     = MI.getOperand(1).getReg();
  unsigned CmpVal  = MI.getOperand(2).getReg();
  unsigned NewVal  = MI.getOperand(3).getReg();

  unsigned AlignedAddr = RegInfo.createVirtualRegister(RC);
  unsigned ShiftAmt = RegInfo.createVirtualRegister(RC);
  unsigned Mask = RegInfo.createVirtualRegister(RC);
  unsigned Mask2 = RegInfo.createVirtualRegister(RC);
  unsigned Mask3 = RegInfo.createVirtualRegister(RC);
  unsigned ShiftedCmpVal = RegInfo.createVirtualRegister(RC);
  unsigned OldVal = RegInfo.createVirtualRegister(RC);
  unsigned MaskedOldVal0 = RegInfo.createVirtualRegister(RC);
  unsigned ShiftedNewVal = RegInfo.createVirtualRegister(RC);
  unsigned MaskLSB2 = RegInfo.createVirtualRegister(RC);
  unsigned PtrLSB2 = RegInfo.createVirtualRegister(RC);
  unsigned MaskUpper = RegInfo.createVirtualRegister(RC);
  unsigned MaskedCmpVal = RegInfo.createVirtualRegister(RC);
  unsigned MaskedNewVal = RegInfo.createVirtualRegister(RC);
  unsigned MaskedOldVal1 = RegInfo.createVirtualRegister(RC);
  unsigned StoreVal = RegInfo.createVirtualRegister(RC);
  unsigned SrlRes = RegInfo.createVirtualRegister(RC);
  unsigned Success = RegInfo.createVirtualRegister(RC);

  // insert new blocks after the current block
  const BasicBlock *LLVM_BB = BB->getBasicBlock();
  MachineBasicBlock *loop1MBB = MF->CreateMachineBasicBlock(LLVM_BB);
  MachineBasicBlock *loop2MBB = MF->CreateMachineBasicBlock(LLVM_BB);
  MachineBasicBlock *sinkMBB = MF->CreateMachineBasicBlock(LLVM_BB);
  MachineBasicBlock *exitMBB = MF->CreateMachineBasicBlock(LLVM_BB);
  MachineFunction::iterator It = ++BB->getIterator();

  MF->insert(It, loop1MBB);
  MF->insert(It, loop2MBB);
  MF->insert(It, sinkMBB);
  MF->insert(It, exitMBB);

  // Transfer the remainder of BB and its successor edges to exitMBB.
  exitMBB->splice(exitMBB->begin(), BB,
                  std::next(MachineBasicBlock::iterator(MI)), BB->end());
  exitMBB->transferSuccessorsAndUpdatePHIs(BB);

  BB->addSuccessor(loop1MBB);
  loop1MBB->addSuccessor(sinkMBB);
  loop1MBB->addSuccessor(loop2MBB);
  loop2MBB->addSuccessor(loop1MBB);
  loop2MBB->addSuccessor(sinkMBB);
  sinkMBB->addSuccessor(exitMBB);

  // FIXME: computation of newval2 can be moved to loop2MBB.
  //  thisMBB:
  //    addiu   masklsb2,$0,-4                # 0xfffffffc
```

(continues on next page)

(continued from previous page)

```
//    and     alignedaddr,ptr,masklsb2
//    andi    ptrlsb2,ptr,3
//    shl     shiftamt,ptrlsb2,3
//    ori     maskupper,$0,255               # 0xff
//    shl     mask,maskupper,shiftamt
//    xor     mask2,$0,mask
//    xor     mask3,$0,mask2
//    andi    maskedcmpval,cmpval,255
//    shl     shiftedcmpval,maskedcmpval,shiftamt
//    andi    maskednewval,newval,255
//    shl     shiftednewval,maskednewval,shiftamt
int64_t MaskImm = (Size == 1) ? 255 : 65535;
BuildMI(BB, DL, TII->get(Cpu0::ADDiu), MaskLSB2)
  .addReg(Cpu0::ZERO).addImm(-4);
BuildMI(BB, DL, TII->get(Cpu0::AND), AlignedAddr)
  .addReg(Ptr).addReg(MaskLSB2);
BuildMI(BB, DL, TII->get(Cpu0::ANDi), PtrLSB2).addReg(Ptr).addImm(3);
if (Subtarget.isLittle()) {
  BuildMI(BB, DL, TII->get(Cpu0::SHL), ShiftAmt).addReg(PtrLSB2).addImm(3);
} else {
  unsigned Off = RegInfo.createVirtualRegister(RC);
  BuildMI(BB, DL, TII->get(Cpu0::XORi), Off)
    .addReg(PtrLSB2).addImm((Size == 1) ? 3 : 2);
  BuildMI(BB, DL, TII->get(Cpu0::SHL), ShiftAmt).addReg(Off).addImm(3);
}
BuildMI(BB, DL, TII->get(Cpu0::ORi), MaskUpper)
  .addReg(Cpu0::ZERO).addImm(MaskImm);
BuildMI(BB, DL, TII->get(Cpu0::SHLV), Mask)
  .addReg(MaskUpper).addReg(ShiftAmt);
BuildMI(BB, DL, TII->get(Cpu0::XOR), Mask2).addReg(Cpu0::ZERO).addReg(Mask);
BuildMI(BB, DL, TII->get(Cpu0::XOR), Mask3).addReg(Cpu0::ZERO).addReg(Mask2);
BuildMI(BB, DL, TII->get(Cpu0::ANDi), MaskedCmpVal)
  .addReg(CmpVal).addImm(MaskImm);
BuildMI(BB, DL, TII->get(Cpu0::SHLV), ShiftedCmpVal)
  .addReg(MaskedCmpVal).addReg(ShiftAmt);
BuildMI(BB, DL, TII->get(Cpu0::ANDi), MaskedNewVal)
  .addReg(NewVal).addImm(MaskImm);
BuildMI(BB, DL, TII->get(Cpu0::SHLV), ShiftedNewVal)
  .addReg(MaskedNewVal).addReg(ShiftAmt);

//  loop1MBB:
//    ll      oldval,0(alginedaddr)
//    and     maskedoldval0,oldval,mask
//    bne     maskedoldval0,shiftedcmpval,sinkMBB
BB = loop1MBB;
unsigned LL = Cpu0::LL;
BuildMI(BB, DL, TII->get(LL), OldVal).addReg(AlignedAddr).addImm(0);
BuildMI(BB, DL, TII->get(Cpu0::AND), MaskedOldVal0)
  .addReg(OldVal).addReg(Mask);
BuildMI(BB, DL, TII->get(Cpu0::BNE))
  .addReg(MaskedOldVal0).addReg(ShiftedCmpVal).addMBB(sinkMBB);
```

(continues on next page)

(continued from previous page)

```
  //  loop2MBB:
  //    and     maskedoldval1,oldval,mask3
  //    or      storeval,maskedoldval1,shiftednewval
  //    sc      success,storeval,0(alignedaddr)
  //    beq     success,$0,loop1MBB
  BB = loop2MBB;
  BuildMI(BB, DL, TII->get(Cpu0::AND), MaskedOldVal1)
    .addReg(OldVal).addReg(Mask3);
  BuildMI(BB, DL, TII->get(Cpu0::OR), StoreVal)
    .addReg(MaskedOldVal1).addReg(ShiftedNewVal);
  unsigned SC = Cpu0::SC;
  BuildMI(BB, DL, TII->get(SC), Success)
      .addReg(StoreVal).addReg(AlignedAddr).addImm(0);
  BuildMI(BB, DL, TII->get(Cpu0::BEQ))
      .addReg(Success).addReg(Cpu0::ZERO).addMBB(loop1MBB);

  //  sinkMBB:
  //    srl     srlres,maskedoldval0,shiftamt
  //    sign_extend dest,srlres
  BB = sinkMBB;

  BuildMI(BB, DL, TII->get(Cpu0::SHRV), SrlRes)
      .addReg(MaskedOldVal0).addReg(ShiftAmt);
  BB = emitSignExtendToI32InReg(MI, BB, Size, Dest, SrlRes);

  MI.eraseFromParent();   // The instruction is gone now.

  return exitMBB;
}
```

```
SDValue Cpu0TargetLowering::lowerATOMIC_FENCE(SDValue Op,
                                              SelectionDAG &DAG) const {
  // FIXME: Need pseudo-fence for 'singlethread' fences
  // FIXME: Set SType for weaker fences where supported/appropriate.
  unsigned SType = 0;
  SDLoc DL(Op);
  return DAG.getNode(Cpu0ISD::Sync, DL, MVT::Other, Op.getOperand(0),
                     DAG.getConstant(SType, DL, MVT::i32));
}
```

lbdex/chapters/Chapter12_1/Cpu0RegisterInfo.h

```
  /// Code Generation virtual methods...
  const TargetRegisterClass *getPointerRegClass(const MachineFunction &MF,
                                                unsigned Kind) const override;
```

lbdex/chapters/Chapter12_1/Cpu0RegisterInfo.cpp

```
const TargetRegisterClass *
Cpu0RegisterInfo::getPointerRegClass(const MachineFunction &MF,
                                     unsigned Kind) const {
  return &Cpu0::CPURegsRegClass;
}
```

lbdex/chapters/Chapter12_1/Cpu0SEISelLowering.cpp

```
Cpu0SETargetLowering::Cpu0SETargetLowering(const Cpu0TargetMachine &TM,
                                           const Cpu0Subtarget &STI)
    : Cpu0TargetLowering(TM, STI) {
```

```
  setOperationAction(ISD::ATOMIC_FENCE,       MVT::Other, Custom);
```

```
  ...
}
```

lbdex/chapters/Chapter12_1/Cpu0TargetMachine.cpp

```
/// Cpu0 Code Generator Pass Configuration Options.
class Cpu0PassConfig : public TargetPassConfig {
```

```
  void addIRPasses() override;
```

```
  ...
};
```

```
void Cpu0PassConfig::addIRPasses() {
  TargetPassConfig::addIRPasses();
  addPass(createAtomicExpandPass());
}
```

Since SC instruction uses RegisterOperand type in Cpu0InstrInfo.td and SC uses FMem node which DecoderMethod is "DecodeMem", the DecodeMem() of Cpu0Disassembler.cpp need to be changed as above.

The atomic node defined in "let usesCustomInserter = 1 in" of Cpu0InstrInfo.td tells llvm calling EmitInstrWithCustomInserter() of Cpu0ISelLowering.cpp after Instruction Selection stage at Cpu0TargetLowering::EmitInstrWithCustomInserter() of ExpandISelPseudos::runOnMachineFunction() stage. For example, "def ATOMIC_LOAD_ADD_I8 : Atomic2Ops<atomic_load_add_8, CPURegs>;" will calling EmitInstrWithCustomInserter() with Machine Instruction Opcode "ATOMIC_LOAD_ADD_I8" when it meets IR "load atomic i8*".

The "setInsertFencesForAtomic(true);" in Cpu0ISelLowering.cpp will trigger addIRPasses() of Cpu0TargetMachine.cpp, then the createAtomicExpandPass() of addIRPasses() will create llvm IR ATOMIC_FENCE. Next, the lowerATOMIC_FENCE() of Cpu0ISelLowering.cpp will create Cpu0ISD::Sync when it meets IR ATOMIC_FENCE since "setOperationAction(ISD::ATOMIC_FENCE, MVT::Other, Custom);" of Cpu0SEISelLowering.cpp. Finally the pattern defined in Cpu0InstrInfo.td translate it into instruction "sync" by "def SYNC" and alias "SYNC 0".

This part of Cpu0 backend code is same with Mips except Cpu0 has no instruction "nor".

List the atomic IRs, corresponding DAGs and Opcode as the following table.

Table 12.4: The atomic related IRs, their corresponding DAGs and Opcode of Cpu0ISelLowering.cpp

IR	DAG	Opcode
load atomic	AtomicLoad	ATOMIC_CMP_SWAP_XXX
store atomic	AtomicStore	ATOMIC_SWAP_XXX
atomicrmw add	AtomicLoadAdd	ATOMIC_LOAD_ADD_XXX
atomicrmw sub	AtomicLoadSub	ATOMIC_LOAD_SUB_XXX
atomicrmw xor	AtomicLoadXor	ATOMIC_LOAD_XOR_XXX
atomicrmw and	AtomicLoadAnd	ATOMIC_LOAD_AND_XXX
atomicrmw nand	AtomicLoadNand	ATOMIC_LOAD_NAND_XXX
atomicrmw or	AtomicLoadOr	ATOMIC_LOAD_OR_XXX
cmpxchg	AtomicCmpSwapWithSuccess	ATOMIC_CMP_SWAP_XXX
atomicrmw xchg	AtomicLoadSwap	ATOMIC_SWAP_XXX

Input files atomics.ll and atomics-fences.ll include the llvm atomic IRs test. Input files ch12_atomics.cpp and ch12_atomics-fences.cpp are the C++ files for generating llvm atomic IRs. The C++ files need to run with clang options "clang++ -pthread -std=c++11".

CHAPTER

THIRTEEN

VERIFY BACKEND ON VERILOG SIMULATOR

- *Create verilog simulator of Cpu0*
- *Verify backend*
- *Other llvm based tools for Cpu0 processor*

Until now, we have an llvm backend to compile C or assembly as the white part of Fig. 13.1. If without global variable, the elf obj can be dumped to hex file via `llvm-objdump -d` which finished in Chapter ELF Support.

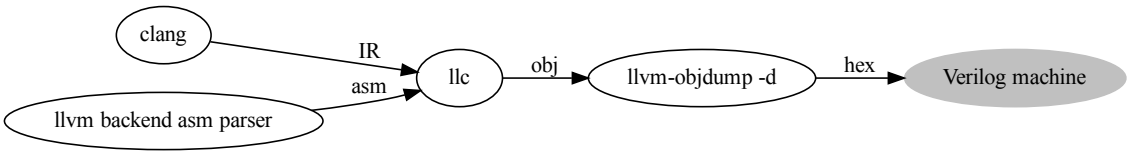


Fig. 13.1: Cpu0 backend without linker

This chapter will implement Cpu0 instructions by Verilog language as the gray part of above figure. With this Verilog machine, we can run this hex program on the Cpu0 Verilog machine on PC and see the Cpu0 instructions execution result.

13.1 Create verilog simulator of Cpu0

Verilog language is an IEEE standard in IC design. There are a lot of books and documents for this language. Free documents exist in Web sites[1][2][3][4][5]. Verilog also called as Verilog HDL but not VHDL. VHDL is the same purpose language which compete against Verilog. About VHDL reference here[6]. Example code, lbdex/verilog/cpu0.v, is the Cpu0 design in Verilog. In Appendix A, we have downloaded and installed Icarus Verilog tool both on iMac and Linux. The cpu0.v is a simple design with only few hundreds lines of code totally. This implementation hasn't the pipeline features, but through implement the delay slot simulation (SIMULATE_DELAY_SLOT part of code), the exact pipeline machine cycles can be calculated.

[1] http://ccckmit.wikidot.com/ve:main
[2] http://www.ece.umd.edu/courses/enee359a/
[3] http://www.ece.umd.edu/courses/enee359a/verilog_tutorial.pdf
[4] http://d1.amobbs.com/bbs_upload782111/files_33/ourdev_585395BQ8J9A.pdf
[5] http://en.wikipedia.org/wiki/Verilog
[6] http://en.wikipedia.org/wiki/VHDL

Verilog is a C like language in syntex and this book is a compiler book, so we list the cpu0.v as well as the building command without explanation as below. We expect readers can understand the Verilog code just with a little patience in reading it. There are two type of I/O according computer architecture. One is memory mapped I/O, the other is instruction I/O. Cpu0 uses memory mapped I/O where memory address 0x80000 as the output port. When meet the instruction **"st $ra, cx($rb)"**, where cx($rb) is 0x80000, Cpu0 displays the content as follows,

```
ST : begin
  ...
  if (R[b]+c16 == `IOADDR) begin
    outw(R[a]);
```

lbdex/verilog/cpu0.v

```
// https://www.francisz.cn/download/IEEE_Standard_1800-2012%20SystemVerilog.pdf

// configuable value below
`define SIMULATE_DELAY_SLOT
// cpu032I memory limit, jsub:24-bit
`define MEMSIZE   'h1000000
`define MEMEMPTY   8'hFF
`define NULL       8'h00
`define IOADDR    'hff000000  // IO mapping address
`define TIMEOUT   #3000000000


// Operand width
`define INT32 2'b11     // 32 bits
`define INT24 2'b10     // 24 bits
`define INT16 2'b01     // 16 bits
`define BYTE  2'b00     // 8  bits

`define EXE 3'b000
`define RESET 3'b001
`define ABORT 3'b010
`define IRQ 3'b011
`define ERROR 3'b100

// Reference web: http://ccckmit.wikidot.com/ocs:cpu0
module cpu0(input clock, reset, input [2:0] itype, output reg [2:0] tick,
            output reg [31:0] ir, pc, mar, mdr, inout [31:0] dbus,
            output reg m_en, m_rw, output reg [1:0] m_size,
            input cfg);
  reg signed [31:0] R [0:15];
  reg signed [31:0] C0R [0:1]; // co-processor 0 register
  // High and Low part of 64 bit result
  reg [7:0] op;
  reg [3:0] a, b, c;
  reg [4:0] c5;
  reg signed [31:0] c12, c16, c24, Ra, Rb, Rc, pc0; // pc0: instruction pc
  reg [31:0] uc16, URa, URb, URc, HI, LO, CF, tmp;
  reg [63:0] cycles;
```

(continues on next page)

(continued from previous page)

```
  // register name
  `define SP   R[13]   // Stack Pointer
  `define LR   R[14]   // Link Register
  `define SW   R[15]   // Status Word

  // C0 register name
  `define PC   C0R[0]   // Program Counter
  `define EPC  C0R[1]  // exception PC value

  // SW Flage
  `define I2   `SW[16] // Hardware Interrupt 1, IO1 interrupt, status,
                       // 1: in interrupt
  `define I1   `SW[15] // Hardware Interrupt 0, timer interrupt, status,
                       // 1: in interrupt
  `define I0   `SW[14] // Software interrupt, status, 1: in interrupt
  `define I    `SW[13] // Interrupt, 1: in interrupt
  `define I2E  `SW[12]  // Hardware Interrupt 1, IO1 interrupt, Enable
  `define I1E  `SW[11]  // Hardware Interrupt 0, timer interrupt, Enable
  `define I0E  `SW[10]  // Software Interrupt Enable
  `define IE   `SW[9]  // Interrupt Enable
  `define M    `SW[8:6]  // Mode bits, itype
  `define D    `SW[5]  // Debug Trace
  `define V    `SW[3]  // Overflow
  `define C    `SW[2]  // Carry
  `define Z    `SW[1]  // Zero
  `define N    `SW[0]  // Negative flag

  `define LE   CF[0]  // Endian bit, Big Endian:0, Little Endian:1
  // Instruction Opcode
  parameter [7:0] NOP=8'h00,LD=8'h01,ST=8'h02,LB=8'h03,LBu=8'h04,SB=8'h05,
  LH=8'h06,LHu=8'h07,SH=8'h08,ADDiu=8'h09,MOVZ=8'h0A,MOVN=8'h0B,ANDi=8'h0C,
  ORi=8'h0D,XORi=8'h0E,LUi=8'h0F,
  ADDu=8'h11,SUBu=8'h12,ADD=8'h13,SUB=8'h14,CLZ=8'h15,CLO=8'h16,MUL=8'h17,
  AND=8'h18,OR=8'h19,XOR=8'h1A,NOR=8'h1B,
  ROL=8'h1C,ROR=8'h1D,SHL=8'h1E,SHR=8'h1F,
  SRA=8'h20,SRAV=8'h21,SHLV=8'h22,SHRV=8'h23,ROLV=8'h24,RORV=8'h25,
`ifdef CPU0II
  SLTi=8'h26,SLTiu=8'h27, SLT=8'h28,SLTu=8'h29,
`endif
  CMP=8'h2A,
  CMPu=8'h2B,
  JEQ=8'h30,JNE=8'h31,JLT=8'h32,JGT=8'h33,JLE=8'h34,JGE=8'h35,
  JMP=8'h36,
`ifdef CPU0II
  BEQ=8'h37,BNE=8'h38,
`endif
  JALR=8'h39,BAL=8'h3A,JSUB=8'h3B,RET=8'h3C,
  MULT=8'h41,MULTu=8'h42,DIV=8'h43,DIVu=8'h44,
  MFHI=8'h46,MFLO=8'h47,MTHI=8'h48,MTLO=8'h49,
  MFC0=8'h50,MTC0=8'h51,C0MOV=8'h52;

  reg [0:0] inExe = 0;
```

(continues on next page)

(continued from previous page)

```
  reg [2:0] state, next_state;
  reg [2:0] st_taskInt, ns_taskInt;
  parameter Reset=3'h0, Fetch=3'h1, Decode=3'h2, Execute=3'h3, MemAccess=3'h4,
            WriteBack=3'h5;
  integer i;
`ifdef SIMULATE_DELAY_SLOT
  reg [0:0] nextInstIsDelaySlot;
  reg [0:0] isDelaySlot;
  reg signed [31:0] delaySlotNextPC;
`endif

  //transform data from the memory to little-endian form
  task changeEndian(input [31:0] value, output [31:0] changeEndian); begin
    changeEndian = {value[7:0], value[15:8], value[23:16], value[31:24]};
  end endtask

  // Read Memory Word
  task memReadStart(input [31:0] addr, input [1:0] size); begin
    mar = addr;     // read(m[addr])
    m_rw = 1;     // Access Mode: read
    m_en = 1;     // Enable read
    m_size = size;
  end endtask

  // Read Memory Finish, get data
  task memReadEnd(output [31:0] data); begin
    mdr = dbus; // get momory, dbus = m[addr]
    data = mdr; // return to data
    m_en = 0; // read complete
  end endtask

  // Write memory -- addr: address to write, data: date to write
  task memWriteStart(input [31:0] addr, input [31:0] data, input [1:0] size);
  begin
    mar = addr;    // write(m[addr], data)
    mdr = data;
    m_rw = 0;    // access mode: write
    m_en = 1;     // Enable write
    m_size  = size;
  end endtask

  task memWriteEnd; begin // Write Memory Finish
    m_en = 0; // write complete
  end endtask

  task regSet(input [3:0] i, input [31:0] data); begin
    if (i != 0) R[i] = data;
  end endtask

  task C0regSet(input [3:0] i, input [31:0] data); begin
    if (i < 2) C0R[i] = data;
  end endtask
```

(continues on next page)

(continued from previous page)

```
task PCSet(input [31:0] data); begin
`ifdef SIMULATE_DELAY_SLOT
  nextInstIsDelaySlot = 1;
  delaySlotNextPC = data;
`else
  `PC = data;
`endif
end endtask

task retValSet(input [3:0] i, input [31:0] data); begin
  if (i != 0)
  `ifdef SIMULATE_DELAY_SLOT
    R[i] = data + 4;
  `else
    R[i] = data;
  `endif
end endtask

task regHILOSet(input [31:0] data1, input [31:0] data2); begin
  HI = data1;
  LO = data2;
end endtask

// output a word to Output port (equal to display the word to terminal)
task outw(input [31:0] data); begin
  if (`LE) begin // Little Endian
    changeEndian(data, data);
  end
  if (data[7:0] != 8'h00) begin
    $write("%c", data[7:0]);
    if (data[15:8] != 8'h00)
      $write("%c", data[15:8]);
    if (data[23:16] != 8'h00)
      $write("%c", data[23:16]);
    if (data[31:24] != 8'h00)
      $write("%c", data[31:24]);
  end
end endtask

// output a character (a byte)
task outc(input [7:0] data); begin
  $write("%c", data);
end endtask

task taskInterrupt(input [2:0] iMode); begin
if (inExe == 0) begin
  case (iMode)
    `RESET: begin
      `PC = 0; tick = 0; R[0] = 0; `SW = 0; `LR = -1;
      `IE = 0; `I0E = 1; `I1E = 1; `I2E = 1;
      `I = 0; `I0 = 0; `I1 = 0; `I2 = 0; inExe = 1;
```

(continues on next page)

(continued from previous page)

```
      `LE = cfg;
      cycles = 0;
    end
    `ABORT: begin `PC = 4; end
    `IRQ:   begin `PC = 8; `IE = 0; inExe = 1; end
    `ERROR: begin `PC = 12; end
  endcase
end
$display("taskInterrupt(%3b)", iMode);
end endtask

task taskExecute; begin
  tick = tick+1;
  case (state)
  Fetch: begin  // Tick 1 : instruction fetch, throw PC to address bus,
                // memory.read(m[PC])
    memReadStart(`PC, `INT32);
    pc0  = `PC;
 `ifdef SIMULATE_DELAY_SLOT
   if (nextInstIsDelaySlot == 1) begin
     isDelaySlot = 1;
     nextInstIsDelaySlot = 0;
     `PC = delaySlotNextPC;
   end
   else begin
     if (isDelaySlot == 1) isDelaySlot = 0;
     `PC = `PC+4;
   end
 `else
   `PC = `PC+4;
 `endif
    next_state = Decode;
  end
  Decode: begin  // Tick 2 : instruction decode, ir = m[PC]
    memReadEnd(ir); // IR = dbus = m[PC]
    {op,a,b,c} = ir[31:12];
    c24 = $signed(ir[23:0]);
    c16 = $signed(ir[15:0]);
    uc16 = ir[15:0];
    c12 = $signed(ir[11:0]);
    c5  = ir[4:0];
    Ra = R[a];
    Rb = R[b];
    Rc = R[c];
    URa = R[a];
    URb = R[b];
    URc = R[c];
    next_state = Execute;
  end
  Execute: begin // Tick 3 : instruction execution
    case (op)
    NOP:   ;
```

(continues on next page)

(continued from previous page)

```
    // load and store instructions
    LD:    memReadStart(Rb+c16, `INT32);      // LD Ra,[Rb+Cx]; Ra<=[Rb+Cx]
    ST:    memWriteStart(Rb+c16, Ra, `INT32); // ST Ra,[Rb+Cx]; Ra=>[Rb+Cx]
    // LB Ra,[Rb+Cx]; Ra<=(byte)[Rb+Cx]
    LB:    memReadStart(Rb+c16, `BYTE);
    // LBu Ra,[Rb+Cx]; Ra<=(byte)[Rb+Cx]
    LBu:   memReadStart(Rb+c16, `BYTE);
    // SB Ra,[Rb+Cx]; Ra=>(byte)[Rb+Cx]
    SB:    memWriteStart(Rb+c16, Ra, `BYTE);
    LH:    memReadStart(Rb+c16, `INT16); // LH Ra,[Rb+Cx]; Ra<=(2bytes)[Rb+Cx]
    LHu:   memReadStart(Rb+c16, `INT16); // LHu Ra,[Rb+Cx]; Ra<=(2bytes)[Rb+Cx]
    // SH Ra,[Rb+Cx]; Ra=>(2bytes)[Rb+Cx]
    SH:    memWriteStart(Rb+c16, Ra, `INT16);
    // Conditional move
    MOVZ:  if (Rc==0) regSet(a, Rb);             // move if Rc equal to 0
    MOVN:  if (Rc!=0) regSet(a, Rb);             // move if Rc not equal to 0
    // Mathematic
    ADDiu: regSet(a, Rb+c16);                   // ADDiu Ra, Rb+Cx; Ra<=Rb+Cx
    CMP:  begin
      if (Rb < Rc) `N=1; else `N=0;
      // `N=(Rb-Rc<0); // why not work for bash make.sh cpu032I el Makefile.builtins?
     `Z=(Rb-Rc==0);
    end // CMP Rb, Rc; SW=(Rb >=< Rc)
    CMPu:  begin
      if (URb < URc) `N=1; else `N=0;
     `Z=(URb-URc==0);
    end // CMPu URb, URc; SW=(URb >=< URc)
    ADDu:  regSet(a, Rb+Rc);               // ADDu Ra,Rb,Rc; Ra<=Rb+Rc
    ADD:   begin regSet(a, Rb+Rc); if (a < Rb) `V = 1; else `V = 0;
      if (`V) begin `I0 = 1; `I = 1; end
    end
                                           // ADD Ra,Rb,Rc; Ra<=Rb+Rc
    SUBu:  regSet(a, Rb-Rc);               // SUBu Ra,Rb,Rc; Ra<=Rb-Rc
    SUB:   begin regSet(a, Rb-Rc); if (Rb < 0 && Rc > 0 && a >= 0)
           `V = 1; else `V =0;
      if (`V) begin `I0 = 1; `I = 1; end
    end         // SUB Ra,Rb,Rc; Ra<=Rb-Rc
    CLZ:   begin
      for (i=0; (i<32)&&((Rb&32'h80000000)==32'h00000000); i=i+1) begin
          Rb=Rb<<1;
      end
      regSet(a, i);
    end
    CLO:   begin
      for (i=0; (i<32)&&((Rb&32'h80000000)==32'h80000000); i=i+1) begin
          Rb=Rb<<1;
      end
      regSet(a, i);
    end
    MUL:   regSet(a, Rb*Rc);               // MUL Ra,Rb,Rc;     Ra<=Rb*Rc
    DIVu:  regHILOSet(URa%URb, URa/URb);   // DIVu URa,URb; HI<=URa%URb;
                                           // LO<=URa/URb
```

(continues on next page)

(continued from previous page)

```
                                        // without exception overflow
    DIV:   begin regHILOSet(Ra%Rb, Ra/Rb);
           if ((Ra < 0 && Rb < 0) || (Ra == 0)) `V = 1;
           else `V =0; end  // DIV Ra,Rb; HI<=Ra%Rb; LO<=Ra/Rb; With overflow
    AND:   regSet(a, Rb&Rc);               // AND Ra,Rb,Rc; Ra<=(Rb and Rc)
    ANDi:  regSet(a, Rb&uc16);             // ANDi Ra,Rb,c16; Ra<=(Rb and c16)
    OR:    regSet(a, Rb|Rc);               // OR Ra,Rb,Rc; Ra<=(Rb or Rc)
    ORi:   regSet(a, Rb|uc16);             // ORi Ra,Rb,c16; Ra<=(Rb or c16)
    XOR:   regSet(a, Rb^Rc);               // XOR Ra,Rb,Rc; Ra<=(Rb xor Rc)
    NOR:   regSet(a, ~(Rb|Rc));            // NOR Ra,Rb,Rc; Ra<=(Rb nor Rc)
    XORi:  regSet(a, Rb^uc16);             // XORi Ra,Rb,c16; Ra<=(Rb xor c16)
    LUi:   regSet(a, uc16<<16);
    SHL:   regSet(a, Rb<<c5);     // Shift Left; SHL Ra,Rb,Cx; Ra<=(Rb << Cx)
    SRA:   regSet(a, (Rb>>>c5));  // Shift Right with signed bit fill;
      // https://stackoverflow.com/questions/39911655/how-to-synthesize-hardware-for-
↪sra-instruction
    SHR:   regSet(a, Rb>>c5);     // Shift Right with 0 fill;
                                  // SHR Ra,Rb,Cx; Ra<=(Rb >> Cx)
    SHLV:  regSet(a, Rb<<Rc);     // Shift Left; SHLV Ra,Rb,Rc; Ra<=(Rb << Rc)
    SRAV:  regSet(a, (Rb>>>Rc));  // Shift Right with signed bit fill;
    SHRV:  regSet(a, Rb>>Rc);     // Shift Right with 0 fill;
                                  // SHRV Ra,Rb,Rc; Ra<=(Rb >> Rc)
    ROL:   regSet(a, (Rb<<c5)|(Rb>>(32-c5)));     // Rotate Left;
    ROR:   regSet(a, (Rb>>c5)|(Rb<<(32-c5)));     // Rotate Right;
    ROLV:  begin // Can set Rc to -32<=Rc<=32 more efficently.
      while (Rc < -32) Rc=Rc+32;
      while (Rc > 32) Rc=Rc-32;
      regSet(a, (Rb<<Rc)|(Rb>>(32-Rc)));     // Rotate Left;
    end
    RORV:  begin
      while (Rc < -32) Rc=Rc+32;
      while (Rc > 32) Rc=Rc-32;
      regSet(a, (Rb>>Rc)|(Rb<<(32-Rc)));     // Rotate Right;
    end
    MFLO:  regSet(a, LO);         // MFLO Ra; Ra<=LO
    MFHI:  regSet(a, HI);         // MFHI Ra; Ra<=HI
    MTLO:  LO = Ra;               // MTLO Ra; LO<=Ra
    MTHI:  HI = Ra;               // MTHI Ra; HI<=Ra
    MULT:  {HI, LO}=Ra*Rb;        // MULT Ra,Rb; HI<=((Ra*Rb)>>32);
                                  // LO<=((Ra*Rb) and 0x00000000ffffffff);
                                  // with exception overflow
    MULTu: {HI, LO}=URa*URb;      // MULT URa,URb; HI<=((URa*URb)>>32);
                                  // LO<=((URa*URb) and 0x00000000ffffffff);
                                  // without exception overflow
    MFC0:  regSet(a, C0R[b]);     // MFC0 a, b; Ra<=C0R[Rb]
    MTC0:  C0regSet(a, Rb);       // MTC0 a, b; C0R[a]<=Rb
    C0MOV: C0regSet(a, C0R[b]);   // C0MOV a, b; C0R[a]<=C0R[b]
  `ifdef CPU0II
    // set
    SLT:   if (Rb < Rc) R[a]=1; else R[a]=0;
    SLTu:  if (URb < URc) R[a]=1; else R[a]=0;
    SLTi:  if (Rb < c16) R[a]=1; else R[a]=0;
```

(continues on next page)

(continued from previous page)

```
    SLTiu: if (URb < uc16) R[a]=1; else R[a]=0;
    // Branch Instructions
    BEQ:   if (Ra==Rb) PCSet(`PC+c16);
    BNE:   if (Ra!=Rb) PCSet(`PC+c16);
  `endif
    // Jump Instructions
    JEQ:   if (`Z) PCSet(`PC+c24);            // JEQ Cx; if SW(=) PC  PC+Cx
    JNE:   if (!`Z) PCSet(`PC+c24);           // JNE Cx; if SW(!=) PC PC+Cx
    JLT:   if (`N) PCSet(`PC+c24);            // JLT Cx; if SW(<) PC  PC+Cx
    JGT:   if (!`N&&!`Z) PCSet(`PC+c24);      // JGT Cx; if SW(>) PC  PC+Cx
    JLE:   if (`N || `Z) PCSet(`PC+c24);      // JLE Cx; if SW(<=) PC PC+Cx
    JGE:   if (!`N || `Z) PCSet(`PC+c24);     // JGE Cx; if SW(>=) PC PC+Cx
    JMP:   `PC = `PC+c24;                     // JMP Cx; PC <= PC+Cx
    JALR:  begin retValSet(a, `PC); PCSet(Rb); end    // JALR Ra,Rb; Ra<=PC; PC<=Rb
    BAL:   begin `LR = `PC; `PC = `PC+c24; end // BAL Cx; LR<=PC; PC<=PC+Cx
    JSUB:  begin retValSet(14, `PC); PCSet(`PC+c24); end // JSUB Cx; LR<=PC; PC<=PC+Cx
    RET:   begin PCSet(Ra); end               // RET; PC <= Ra
    default :
      $display("%4dns %8x : OP code %8x not support", $stime, pc0, op);
    endcase
    if (`IE && `I && (`I0E && `I0 || `I1E && `I1 || `I2E && `I2)) begin
      `EPC = `PC;
      next_state = Fetch;
      inExe = 0;
    end else
      next_state = MemAccess;
  end
  MemAccess: begin
    case (op)
    ST, SB, SH  :
      memWriteEnd();                // write memory complete
    endcase
    next_state = WriteBack;
  end
  WriteBack: begin // Read/Write finish, close memory
    case (op)
    LB, LBu  :
      memReadEnd(R[a]);        //read memory complete
    LH, LHu  :
      memReadEnd(R[a]);
    LD  : begin
      memReadEnd(R[a]);
      if (`D)
        $display("%4dns %8x : %8x m[%-04x+%-04x]=%8x  SW=%8x", $stime, pc0,
                 ir, R[b], c16, R[a], `SW);
    end
    endcase
    case (op)
    LB  : begin
      if (R[a] > 8'h7f) R[a]=R[a]|32'hffffff80;
    end
    LH  : begin
```

(continues on next page)

(continued from previous page)

```
      if (R[a] > 16'h7fff) R[a]=R[a]|32'hffff8000;
    end
    endcase
    case (op)
    MULT, MULTu, DIV, DIVu, MTHI, MTLO :
      if (`D)
        $display("%4dns %8x : %8x HI=%8x LO=%8x SW=%8x", $stime, pc0, ir, HI,
                 LO, `SW);
    ST : begin
      if (`D)
        $display("%4dns %8x : %8x m[%-04x+%-04x]=%8x  SW=%8x", $stime, pc0,
                 ir, R[b], c16, R[a], `SW);
      if (R[b]+c16 == `IOADDR) begin
        outw(R[a]);
      end
    end
    SB : begin
      if (`D)
        $display("%4dns %8x : %8x m[%-04x+%-04x]=%c  SW=%8x, R[a]=%8x",
                 $stime, pc0, ir, R[b], c16, R[a][7:0], `SW, R[a]);
      if (R[b]+c16 == `IOADDR) begin
        if (`LE)
          outc(R[a][7:0]);
        else
          outc(R[a][7:0]);
      end
    end
    MFC0, MTC0 :
      if (`D)
        $display("%4dns %8x : %8x R[%02d]=%-8x  C0R[%02d]=%-8x SW=%8x",
                 $stime, pc0, ir, a, R[a], a, C0R[a], `SW);
    C0MOV :
      if (`D)
        $display("%4dns %8x : %8x C0R[%02d]=%-8x C0R[%02d]=%-8x SW=%8x",
                 $stime, pc0, ir, a, C0R[a], b, C0R[b], `SW);
    default :
      if (`D) // Display the written register content
        $display("%4dns %8x : %8x R[%02d]=%-8x SW=%8x", $stime, pc0, ir,
                 a, R[a], `SW);
    endcase
    if (`PC < 0) begin
      $display("total cpu cycles = %-d", cycles);
      $display("RET to PC < 0, finished!");
      $finish;
    end
    next_state = Fetch;
  end
  endcase
end endtask

always @(posedge clock) begin
  if (inExe == 0 && (state == Fetch) && (`IE && `I) && (`I0E && `I0)) begin
```

(continues on next page)

(continued from previous page)

```
    // software int
      `M = `IRQ;
      taskInterrupt(`IRQ);
      m_en = 0;
      state = Fetch;
    end else if (inExe == 0 && (state == Fetch) && (`IE && `I) &&
                 ((`I1E && `I1) || (`I2E && `I2)) ) begin
      `M = `IRQ;
      taskInterrupt(`IRQ);
      m_en = 0;
      state = Fetch;
    end else if (inExe == 0 && itype == `RESET) begin
    // Condition itype == `RESET must after the other `IE condition
      taskInterrupt(`RESET);
      `M = `RESET;
      state = Fetch;
    end else begin
    `ifdef TRACE
      `D = 1; // Trace register content at beginning
    `endif
      taskExecute();
      state = next_state;
    end
    pc = `PC;
    cycles = cycles + 1;
  end
endmodule

module memory0(input clock, reset, en, rw, input [1:0] m_size,
               input [31:0] abus, dbus_in, output [31:0] dbus_out,
               output cfg);
  reg [31:0] mconfig [0:0];
  reg [7:0] m [0:`MEMSIZE-1];
  reg [31:0] data;

  integer i;

  `define LE  mconfig[0][0:0]   // Endian bit, Big Endian:0, Little Endian:1

  initial begin
  // erase memory
    for (i=0; i < `MEMSIZE; i=i+1) begin
       m[i] = `MEMEMPTY;
    end
  // load config from file to memory
    $readmemh("cpu0.config", mconfig);
  // load program from file to memory
    $readmemh("cpu0.hex", m);
  // display memory contents
    `ifdef TRACE
      for (i=0; i < `MEMSIZE && (m[i] != `MEMEMPTY || m[i+1] != `MEMEMPTY ||
         m[i+2] != `MEMEMPTY || m[i+3] != `MEMEMPTY); i=i+4) begin
```

(continues on next page)

(continued from previous page)

```
        $display("%8x: %8x", i, {m[i], m[i+1], m[i+2], m[i+3]});
      end
    `endif
  end

  always @(clock or abus or en or rw or dbus_in)
  begin
    if (abus >= 0 && abus <= `MEMSIZE-4) begin
      if (en == 1 && rw == 0) begin // r_w==0:write
        data = dbus_in;
        if (`LE) begin // Little Endian
          case (m_size)
          `BYTE:  {m[abus]} = dbus_in[7:0];
          `INT16: {m[abus], m[abus+1] } = {dbus_in[7:0], dbus_in[15:8]};
          `INT24: {m[abus], m[abus+1], m[abus+2]} =
                  {dbus_in[7:0], dbus_in[15:8], dbus_in[23:16]};
          `INT32: {m[abus], m[abus+1], m[abus+2], m[abus+3]} =
                  {dbus_in[7:0], dbus_in[15:8], dbus_in[23:16], dbus_in[31:24]};
          endcase
        end else begin // Big Endian
          case (m_size)
          `BYTE:  {m[abus]} = dbus_in[7:0];
          `INT16: {m[abus], m[abus+1] } = dbus_in[15:0];
          `INT24: {m[abus], m[abus+1], m[abus+2]} = dbus_in[23:0];
          `INT32: {m[abus], m[abus+1], m[abus+2], m[abus+3]} = dbus_in;
          endcase
        end
      end else if (en == 1 && rw == 1) begin // r_w==1:read
        if (`LE) begin // Little Endian
          case (m_size)
          `BYTE:  data = {8'h00,     8'h00,     8'h00,     m[abus]};
          `INT16: data = {8'h00,     8'h00,     m[abus+1], m[abus]};
          `INT24: data = {8'h00,     m[abus+2], m[abus+1], m[abus]};
          `INT32: data = {m[abus+3], m[abus+2], m[abus+1], m[abus]};
          endcase
        end else begin // Big Endian
          case (m_size)
          `BYTE:  data = {8'h00  , 8'h00,     8'h00,     m[abus]  };
          `INT16: data = {8'h00  , 8'h00,     m[abus],   m[abus+1]};
          `INT24: data = {8'h00  , m[abus],   m[abus+1], m[abus+2]};
          `INT32: data = {m[abus], m[abus+1], m[abus+2], m[abus+3]};
          endcase
        end
      end else
        data = 32'hZZZZZZZZ;
    end else
      data = 32'hZZZZZZZZ;
  end
  assign dbus_out = data;
  assign cfg = mconfig[0][0:0];
endmodule
```

(continues on next page)

(continued from previous page)

```
module main;
  reg clock;
  reg [2:0] itype;
  wire [2:0] tick;
  wire [31:0] pc, ir, mar, mdr, dbus;
  wire m_en, m_rw;
  wire [1:0] m_size;
  wire cfg;

  cpu0 cpu(.clock(clock), .itype(itype), .pc(pc), .tick(tick), .ir(ir),
  .mar(mar), .mdr(mdr), .dbus(dbus), .m_en(m_en), .m_rw(m_rw), .m_size(m_size),
  .cfg(cfg));

  memory0 mem(.clock(clock), .reset(reset), .en(m_en), .rw(m_rw),
  .m_size(m_size), .abus(mar), .dbus_in(mdr), .dbus_out(dbus), .cfg(cfg));

  initial
  begin
    clock = 0;
    itype = `RESET;
    `TIMEOUT $finish;
  end

  always #10 clock=clock+1;

endmodule
```

lbdex/verilog/Makefile

```
#TRACE=-D TRACE
all:
        iverilog ${TRACE} -o cpu0Is cpu0.v
        iverilog ${TRACE} -D CPU0II -o cpu0IIs cpu0.v

.PHONY: clean
clean:
        rm -rf cpu0.hex cpu0Is cpu0IIs
        rm -f *~ cpu0.config
```

Since Cpu0 Verilog machine supports both big and little endian, the memory and cpu module both have a wire connecting each other. The endian information stored in ROM of memory module, and memory module send the information when it is up according the following code,

lbdex/verilog/cpu0.v

```
assign cfg = mconfig[0][0:0];
...
wire cfg;

cpu0 cpu(.clock(clock), .itype(itype), .pc(pc), .tick(tick), .ir(ir),
.mar(mar), .mdr(mdr), .dbus(dbus), .m_en(m_en), .m_rw(m_rw), .m_size(m_size),
.cfg(cfg));

memory0 mem(.clock(clock), .reset(reset), .en(m_en), .rw(m_rw),
.m_size(m_size), .abus(mar), .dbus_in(mdr), .dbus_out(dbus), .cfg(cfg));
```

Instead of setting endian tranfer in memory module, the endian transfer can also be set in CPU module, and memory moudle always return with big endian. I am not an professional engineer in FPGA/CPU hardware design. But according book "Computer Architecture: A Quantitative Approach", some operations may have no tolerance in time of execution stage. Any endian swap will make the clock cycle time longer and affect the CPU performance. So, I set the endian transfer in memory module. In system with bus, it will be set in bus system I think.

13.2 Verify backend

Now let's compile ch_run_backend.cpp as below. Since code size grows up from low to high address and stack grows up from high to low address. $sp is set at 0x7fffc because assuming cpu0.v uses 0x80000 bytes of memory.

lbdex/input/start.h

```
#ifndef _START_H_
#define _START_H_

#include "config.h"

#define SET_SW \
asm("andi $sw, $zero, 0"); \
asm("ori  $sw, $sw, 0x1e00"); // enable all interrupts

#define initRegs() \
  asm("addiu $1,  $zero, 0"); \
  asm("addiu $2,  $zero, 0"); \
  asm("addiu $3,  $zero, 0"); \
  asm("addiu $4,  $zero, 0"); \
  asm("addiu $5,  $zero, 0"); \
  asm("addiu $t9, $zero, 0"); \
  asm("addiu $7,  $zero, 0"); \
  asm("addiu $8,  $zero, 0"); \
  asm("addiu $9,  $zero, 0"); \
  asm("addiu $10, $zero, 0"); \
  SET_SW;                     \
  asm("addiu $fp, $zero, 0");

#endif
```

lbdex/input/boot.cpp

```
#include "start.h"

// boot:
  asm("boot:");
//  asm("_start:");
  asm("jmp 12"); // RESET: jmp RESET_START;
  asm("jmp 4");  // ERROR: jmp ERR_HANDLE;
  asm("jmp 4");  // IRQ: jmp IRQ_HANDLE;
  asm("jmp -4"); // ERR_HANDLE: jmp ERR_HANDLE; (loop forever)

// RESET_START:
  initRegs();
  asm("addiu $gp, $ZERO, 0");
  asm("addiu $lr, $ZERO, -1");

  INIT_SP;
  asm("mfc0 $3, $pc");
  asm("addiu $3, $3, 0x8"); // Assume main() entry point is at the next next
                            // instruction.
  asm("jr $3");
  asm("nop");
```

lbdex/input/print.h

```
#ifndef _PRINT_H_
#define _PRINT_H_

#include "start.h"

void print_char(const char c);
void dump_mem(unsigned char *str, int n);
void print_string(const char *str);
void print_integer(int x);
#endif
```

lbdex/input/print.cpp

```
#include "print.h"
#include "itoa.cpp"

// For memory IO
void print_char(const char c)
{
  char *p = (char*)IOADDR;
  *p = c;

  return;
```

(continues on next page)

(continued from previous page)

```
}

void print_string(const char *str)
{
  const char *p;

  for (p = str; *p != '\0'; p++)
    print_char(*p);
  print_char(*p);
  print_char('\n');

  return;
}

// For memory IO
void print_integer(int x)
{
  char str[INT_DIGITS + 2];
  itoa(str, x);
  print_string(str);

  return;
}
```

lbdex/input/ch_nolld.h

```
#include "debug.h"
#include "boot.cpp"

#include "print.h"

int test_nolld();
```

lbdex/input/ch_nolld.cpp

```
#define TEST_ROXV
#define RUN_ON_VERILOG

#include "print.cpp"

#include "ch4_1_math.cpp"
#include "ch4_1_rotate.cpp"
#include "ch4_1_mult2.cpp"
#include "ch4_1_mod.cpp"
#include "ch4_1_div.cpp"
#include "ch4_2_logic.cpp"
#include "ch7_1_localpointer.cpp"
```

(continues on next page)

(continued from previous page)

```
#include "ch7_1_char_short.cpp"
#include "ch7_1_bool.cpp"
#include "ch7_1_longlong.cpp"
#include "ch7_1_vector.cpp"
#include "ch8_1_ctrl.cpp"
#include "ch8_2_deluselessjmp.cpp"
#include "ch8_2_select.cpp"
#include "ch9_1_longlong.cpp"
#include "ch9_3_vararg.cpp"
#include "ch9_3_stacksave.cpp"
#include "ch9_3_bswap.cpp"
#include "ch9_3_alloc.cpp"
#include "ch11_2.cpp"

// Test build only for the following files on build-run_backend.sh since it
// needs lld linker support.
// Test in build-slink.sh
#include "ch6_1.cpp"
#include "ch9_1_struct.cpp"
#include "ch9_1_constructor.cpp"
#include "ch9_3_template.cpp"
#include "ch12_inherit.cpp"

void test_asm_build()
{
  #include "ch11_1.cpp"
#ifdef CPU032II
  #include "ch11_1_2.cpp"
#endif
}

int test_rotate()
{
  int a = test_rotate_left1(); // rolv 4, 30 = 1
  int b = test_rotate_left(); // rol 8, 30  = 2
  int c = test_rotate_right(); // rorv 1, 30 = 4

  return (a+b+c);
}

int test_nolld()
{
  bool pass = true;
  int a = 0;

  a = test_math();
  print_integer(a);  // a = 68
  if (a != 68) pass = false;
  a = test_rotate();
  print_integer(a);  // a = 7
  if (a != 7) pass = false;
  a = test_mult();
```

(continues on next page)

(continued from previous page)

```
  print_integer(a);  // a = 0
  if (a != 0) pass = false;
  a = test_mod();
  print_integer(a);  // a = 0
  if (a != 0) pass = false;
  a = test_div();
  print_integer(a);  // a = 253
  if (a != 253) pass = false;
  a = test_local_pointer();
  print_integer(a);  // a = 3
  if (a != 3) pass = false;
  a = (int)test_load_bool();
  print_integer(a);  // a = 1
  if (a != 1) pass = false;
  a = test_andorxornotcomplement();
  print_integer(a); // a = 13
  if (a != 13) pass = false;
  a = test_setxx();
  print_integer(a); // a = 3
  if (a != 3) pass = false;
  a = test_signed_char();
  print_integer(a); // a = -126
  if (a != -126) pass = false;
  a = test_unsigned_char();
  print_integer(a); // a = 130
  if (a != 130) pass = false;
  a = test_signed_short();
  print_integer(a); // a = -32766
  if (a != -32766) pass = false;
  a = test_unsigned_short();
  print_integer(a); // a = 32770
  if (a != 32770) pass = false;
  long long b = test_longlong();
  print_integer((int)(b >> 32)); // 393307
  if ((int)(b >> 32) != 393307) pass = false;
  print_integer((int)b); // 16777218
  if ((int)(b) != 16777218) pass = false;
  a = test_cmplt_short();
  print_integer(a); // a = -3
  if (a != -3) pass = false;
  a = test_cmplt_long();
  print_integer(a); // a = -4
  if (a != -4) pass = false;
  a = test_control1();
  print_integer(a);        // a = 51
  if (a != 51) pass = false;
  a = test_DelUselessJMP();
  print_integer(a); // a = 2
  if (a != 2) pass = false;
  a = test_movx_1();
  print_integer(a); // a = 3
  if (a != 3) pass = false;
```

(continues on next page)

(continued from previous page)

```
  a = test_movx_2();
  print_integer(a); // a = 1
  if (a != 1) pass = false;
  print_integer(2147483647); // test mod % (mult) from itoa.cpp
  print_integer(-2147483648); // test mod % (multu) from itoa.cpp
  a = test_sum_longlong();
  print_integer(a); // a = 9
  if (a != 9) pass = false;
  a = test_va_arg();
  print_integer(a); // a = 12
  if (a != 12) pass = false;
  a = test_stacksaverestore(100);
  print_integer(a); // a = 5
  if (a != 5) pass = false;
  a = test_bswap();
  print_integer(a); // a = 0
  if (a != 0) pass = false;
  a = test_alloc();
  print_integer(a); // a = 31
  if (a != 31) pass = false;
  a = test_inlineasm();
  print_integer(a); // a = 49
  if (a != 49) pass = false;

  return pass;
}
```

lbdex/input/ch_run_backend.cpp

```
#include "ch_nolld.h"

int main()
{
  bool pass = true;
  pass = test_nolld();

  return pass;
}

#include "ch_nolld.cpp"
```

lbdex/input/functions.sh

```
prologue() {
  if [ $ARG_NUM == 0 ]; then
    echo "useage: bash $sh_name cpu_type endian"
    echo "  cpu_type: cpu032I or cpu032II"
    echo "  endian: eb (big endian, default) or el (little endian)"
    echo "for example:"
    echo "  bash build-slinker.sh cpu032I be"
    exit 1;
  fi
  if [ $CPU != cpu032I ] && [ $CPU != cpu032II ]; then
    echo "1st argument is cpu032I or cpu032II"
    exit 1
  fi

  OS=`uname -s`
  echo "OS =" ${OS}

  TOOLDIR=~/llvm/test/build/bin
  CLANG=~/llvm/test/build/bin/clang

  CPU=$CPU
  echo "CPU =" "${CPU}"

  if [ "$ENDIAN" != "" ] && [ $ENDIAN != el ] && [ $ENDIAN != eb ]; then
    echo "2nd argument is eb (big endian, default) or el (little endian)"
    exit 1
  fi
  if [ $ENDIAN == eb ]; then
    ENDIAN=
  fi
  echo "ENDIAN =" "${ENDIAN}"

  bash clean.sh
}

isLittleEndian() {
  echo "ENDIAN = " "$ENDIAN"
  if [ "$ENDIAN" == "LittleEndian" ] ; then
    LE="true"
  elif [ "$ENDIAN" == "BigEndian" ] ; then
    LE="false"
  else
    echo "!ENDIAN unknown"
    exit 1
  fi
}

elf2hex() {
  ${TOOLDIR}/llvm-objdump -elf2hex -le=$LE a.out > ../verilog/cpu0.hex
  if [ $LE == "true" ] ; then
    echo "1   /* 0: big endian, 1: little endian */" > ../verilog/cpu0.config
```

(continues on next page)

(continued from previous page)

```
  else
    echo "0   /* 0: big endian, 1: little endian */" > ../verilog/cpu0.config
  fi
  cat ../verilog/cpu0.config
}

epilogue() {
  endian=`${TOOLDIR}/llvm-readobj -h a.out|grep "DataEncoding"|awk '{print $2}'`
  isLittleEndian;
  elf2hex;
}
```

lbdex/input/build-run_backend.sh

```
#!/usr/bin/env bash

# for example:
# bash build-run_backend.sh cpu032I el
# bash build-run_backend.sh cpu032II eb

source functions.sh

sh_name=build-run_backend.sh
ARG_NUM=$#
CPU=$1
ENDIAN=$2

DEFFLAGS=""
if [ "$CPU" == cpu032II ] ; then
  DEFFLAGS=${DEFFLAGS}" -DCPU032II"
fi
echo ${DEFFLAGS}

prologue;

# ch8_2_select_global_pic.cpp just for compile build test only, without running
# on verilog.
$CLANG ${DEFFLAGS} -target mips-unknown-linux-gnu -c ch8_2_select_global_pic.cpp \
-emit-llvm -o ch8_2_select_global_pic.bc
${TOOLDIR}/llc -march=cpu0${ENDIAN} -mcpu=${CPU} -relocation-model=pic \
-filetype=obj ch8_2_select_global_pic.bc -o ch8_2_select_global_pic.cpu0.o

$CLANG ${DEFFLAGS} -target mips-unknown-linux-gnu -c ch_run_backend.cpp \
-emit-llvm -o ch_run_backend.bc
echo "${TOOLDIR}/llc -march=cpu0${ENDIAN} -mcpu=${CPU} -relocation-model=static \
-filetype=obj -enable-cpu0-tail-calls ch_run_backend.bc -o ch_run_backend.cpu0.o"
${TOOLDIR}/llc -march=cpu0${ENDIAN} -mcpu=${CPU} -relocation-model=static \
-filetype=obj -enable-cpu0-tail-calls ch_run_backend.bc -o ch_run_backend.cpu0.o
# print must at the same line, otherwise it will spilt into 2 lines
${TOOLDIR}/llvm-objdump --section=.text -d ch_run_backend.cpu0.o | tail -n +8| awk \
```

(continues on next page)

(continued from previous page)

```
'{print "/* " $1 " */\t" $2 " " $3 " " $4 " " $5 "\t/* " $6"\t" $7" " $8" " $9" " $10 "\
↪t*/"}' \
 > ../verilog/cpu0.hex

ENDIAN=`${TOOLDIR}/llvm-readobj -h ch_run_backend.cpu0.o|grep "DataEncoding"|awk '{print
↪$2}'`
isLittleEndian;

if [ $LE == "true" ] ; then
  echo "1   /* 0: big ENDIAN, 1: little ENDIAN */" > ../verilog/cpu0.config
else
  echo "0   /* 0: big ENDIAN, 1: little ENDIAN */" > ../verilog/cpu0.config
fi
cat ../verilog/cpu0.config
```

To run program without linker implementation at this point, the boot.cpp must be set at the beginning of code, and the main() of ch_run_backend.cpp comes immediately after it. Let's run Chapter11_2/ with `llvm-objdump -d` for input file ch_run_backend.cpp to generate the hex file via build-run_bacekend.sh, then feed hex file to cpu0Is Verilog simulator to get the output result as below. Remind ch_run_backend.cpp have to be compiled with option `clang -target mips-unknown-linux-gnu` since the example code ch9_3_vararg.cpp which uses the vararg needs to be compiled with this option. Other example codes have no differences between this option and default option.

```
JonathantekiiMac:input Jonathan$ pwd
/Users/Jonathan/llvm/test/lbdex/input
JonathantekiiMac:input Jonathan$ bash build-run_backend.sh cpu032I eb
JonathantekiiMac:input Jonathan$ cd ../verilog cd ../verilog
JonathantekiiMac:input Jonathan$ pwd
/Users/Jonathan/llvm/test/lbdex/verilog
JonathantekiiMac:verilog Jonathan$ make
JonathantekiiMac:verilog Jonathan$ ./cpu0Is
WARNING: cpu0Is.v:386: $readmemh(cpu0.hex): Not enough words in the file for the
taskInterrupt(001)
68
7
0
0
253
3
1
13
3
-126
130
-32766
32770
393307
16777218
3
4
51
2
3
```

(continues on next page)

(continued from previous page)

```
1
2147483647
-2147483648
15
5
0
31
49
total cpu cycles = 50645
RET to PC < 0, finished!

JonathantekiiMac:input Jonathan$ bash build-run_backend.sh cpu032II eb
JonathantekiiMac:input Jonathan$ cd ../verilog
JonathantekiiMac:verilog Jonathan$ ./cpu0IIs
...
total cpu cycles = 48335
RET to PC < 0, finished!
```

The "total cpu cycles" is calculated in this verilog simualtor so that the backend compiler and CPU performance can be reviewed. Only the CPU cycles are counted in this implemenation since I/O cycles time is unknown. As explained in chapter "Control flow statements", cpu032II which uses instructions slt and beq has better performance than cmp and jeq in cpu032I. Instructions "jmp" has no delay slot so it is better used in dynamic linker implementation.

You can trace the memory binary code and destination register changed at every instruction execution by unmark TRACE in Makefile as below,

lbdex/verilog/Makefile

```
TRACE=-D TRACE
```

```
JonathantekiiMac:raw Jonathan$ ./cpu0Is
WARNING: cpu0.v:386: $readmemh(cpu0.hex): Not enough words in the file for the
requested range [0:28671].
00000000: 2600000c
00000004: 26000004
00000008: 26000004
0000000c: 26fffffc
00000010: 09100000
00000014: 09200000
...
taskInterrupt(001)
1530ns 00000054 : 02ed002c m[28620+44  ]=-1          SW=00000000
1610ns 00000058 : 02bd0028 m[28620+40  ]=0           SW=00000000
...
RET to PC < 0, finished!
```

As above result, cpu0.v dumps the memory first after reading input file cpu0.hex. Next, it runs instructions from address 0 and print each destination register value in the fourth column. The first column is the nano seconds of timing. The second is instruction address. The third is instruction content. Now, most example codes depicted in the previous chapters are verified by print the variable with print_integer().

Since the cpu0.v machine is created by Verilog language, suppose it can run on real FPGA device (but I never do it). The real output hardware interface/port is hardware output device dependent, such as RS232, speaker, LED, ...

You should implement the I/O interface/port when you want to program FPGA and wire I/O device to the I/O port. Through running the compiled code on Verilog simulator, Cpu0 backend compiled result and CPU cycles are verified and calculated. Currently, this Cpu0 Verilog program is not a pipeline architecture, but according the instruction set it can be implemented as a pipeline model. The cycle time of Cpu0 pipeline model is more than 1/5 of "total cpu cycles" displayed as above since there are dependences exist between instructions. Though the Verilog simulator is slow in running the whole system program and not include the cycles counting in cache and I/O, it is a simple and easy way to verify your idea about CPU design at early stage with small program pattern. The overall system simulator is complex to create. Even wiki web site here[7] include tools for creating the simulator, it needs a lot of effort.

To generate cpu032I as well as little endian code, you can run with the following command. File build-run_backend.sh write the endian information to ../verilog/cpu0.config as below.

```
JonathantekiiMac:input Jonathan$ bash build-run_backend.sh cpu032I el
```

../verilog/cpu0.config

```
1   /* 0: big endian, 1: little endian */
```

The following files test more features.

lbdex/input/ch_nolld2.h

```

#include "debug.h"
#include "boot.cpp"

#include "print.h"

int test_nolld2();
```

lbdex/input/ch_nolld2.cpp

```

#include "print.cpp"

#include "ch9_3_alloc.cpp"

int test_nolld2()
{
  bool pass = true;
  int a = 0;

  a = test_alloc();
  print_integer(a);  // a = 31
  if (a != 31) pass = false;
  return pass;
}
```

[7] https://en.wikipedia.org/wiki/Computer_architecture_simulator

lbdex/input/ch_run_backend2.cpp

```
#include "ch_nolld2.h"

int main()
{
  bool pass = true;
  pass = test_nolld2();

  return pass;
}

#include "ch_nolld2.cpp"
```

lbdex/input/build-run_backend2.sh

```
#!/usr/bin/env bash

source functions.sh

sh_name=build-run_backend.sh
ARG_NUM=$#
CPU=$1
ENDIAN=$2

DEFFLAGS=""
if [ "$arg1" == cpu032II ] ; then
  DEFFLAGS=${DEFFLAGS}" -DCPU032II"
fi
echo ${DEFFLAGS}

prologue;

${CLANG} ${DEFFLAGS} -c ch_run_backend2.cpp \
-emit-llvm -o ch_run_backend2.bc
${TOOLDIR}/llc -march=cpu0${ENDIAN} -mcpu=${CPU} -relocation-model=static \
-filetype=obj ch_run_backend2.bc -o ch_run_backend2.cpu0.o
${TOOLDIR}/llvm-objdump -d ch_run_backend2.cpu0.o | tail -n +8| awk \
'{print "/* " $1 " */\t" $2 " " $3 " " $4 " " $5 "\t/* " $6"\t" $7" " $8" \
" $9" " $10 "\t*/"}' > ../verilog/cpu0.hex

ENDIAN=`${TOOLDIR}/llvm-readobj -h ch_run_backend2.cpu0.o|grep "DataEncoding"|awk '
↪{print $2}'`
isLittleEndian;

if [ $LE == "true" ] ; then
  echo "1   /* 0: big endian, 1: little endian */" > ../verilog/cpu0.config
else
  echo "0   /* 0: big endian, 1: little endian */" > ../verilog/cpu0.config
fi
cat ../verilog/cpu0.config
```

```
JonathantekiiMac:input Jonathan$ bash build-run_backend.sh cpu032II el
...
JonathantekiiMac:input Jonathan$ cd ../verilog
JonathantekiiMac:verilog Jonathan$ ./cpu0IIs
...
31
...
```

13.3 Other llvm based tools for Cpu0 processor

You can find the Cpu0 ELF linker implementation based on lld which is the llvm official linker project, as well as elf2hex which modified from llvm-objdump driver at web: http://jonathan2251.github.io/lbt/index.html.

CHAPTER

FOURTEEN

THE CONCEPT OF GPU COMPILER

- *Concept in graphic and system*
  - *3D modeling*
  - *Graphic SW stack*
  - *Basic geometry in computer graphics*
- *OpenGL*
  - *Example of OpenGL program*
  - *3D Rendering*
  - *GLSL (GL Shader Language)*
  - *OpenGL Shader compiler*
- *Architecture*
  - *SIMT*
  - *Buffers*
- *General purpose GPU*
  - *Mapping data in GPU*
  - *Work between CPU and GPU in Cuda*
  - *Volta (Cuda thread/SIMD lane with PC, Program Couner and Call Stack)*
- *Vulkan and spir-v*
- *Accelerate ML/DL on OpenCL/SYCL*
- *Open Sources*

Basicly CPU is SISD (Single Instruction Single Data) Architecture in each core. The multimedia instructions in CPU are smaller scaled of SIMD (Single Instruction Multiple Data) while GPU is a large scaled of SIMD processor, coloring millions of pixels of image in few mini seconds. Since the 2D or 3D graphic processing provides large opportunity in parallel data processing, GPU hardware usually composed of thousands of functional units in each core in N-Vidia processors.

This chapter is giving a overview for how 3D animation to be created and run on CPU+GPU. Give a concept for GPU compiler and HW featrues for graphic application. Furthermore, explaining how GPU has taking more applications from CPU through GPGPU concept and related standards emerged.

Webiste, Basic theory of 3D graphics with OpenGL,[1].

14.1 Concept in graphic and system

14.1.1 3D modeling

Through creating 3D model with Triangles or Quads along on skin, the 3D model is created with polygon mesh[2] formed by all the vertices on the first image as Fig. 14.1.

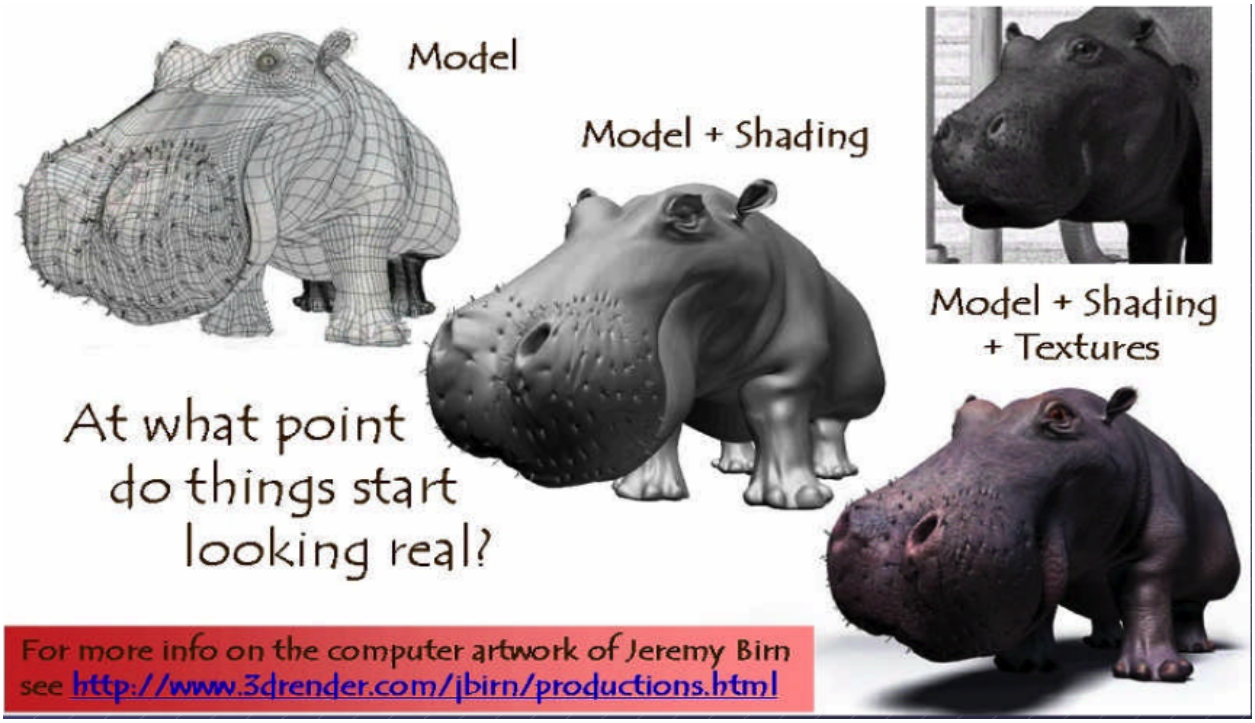


Fig. 14.1: Creating 3D model and texturing

After the next smooth shading[Page 634, 2], the vertices and edge lines are covered with color (or remove edges, actually edge never has black line), and model looks much more smooth[3]. Further, after texturing (texture mapping), the model looks real more[4].

To get to know how animation for a 3D modeling, please look video here[5]. According the video for skeleton animation, setting the joints poistion at different poses and giving time to each pose (keyframe) as Fig. 14.2.

In this series of video, you find the 3D modeling tools creating Java instead of C/C++ code calling OpenGL api and shaders. It's because Java can call OpenGL api through a wrapper library[6].

Every CAD software manufacturer such as AutoDesk and Blender has their own proprietary format. To solve the problem of interoperability, neutral or open source formats were invented as intermediate formats for converting between two proprietary formats. Naturally, these formats have become hugely popular now. Two famous examples of neutral

[1] https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
[2] https://www.quora.com/Which-one-is-better-for-3D-modeling-Quads-or-Tris
[3] https://en.wikipedia.org/wiki/Shading
[4] https://en.wikipedia.org/wiki/Texture_mapping
[5] https://www.youtube.com/watch?v=f3Cr8Yx3GGA
[6] https://en.wikipedia.org/wiki/Java_OpenGL

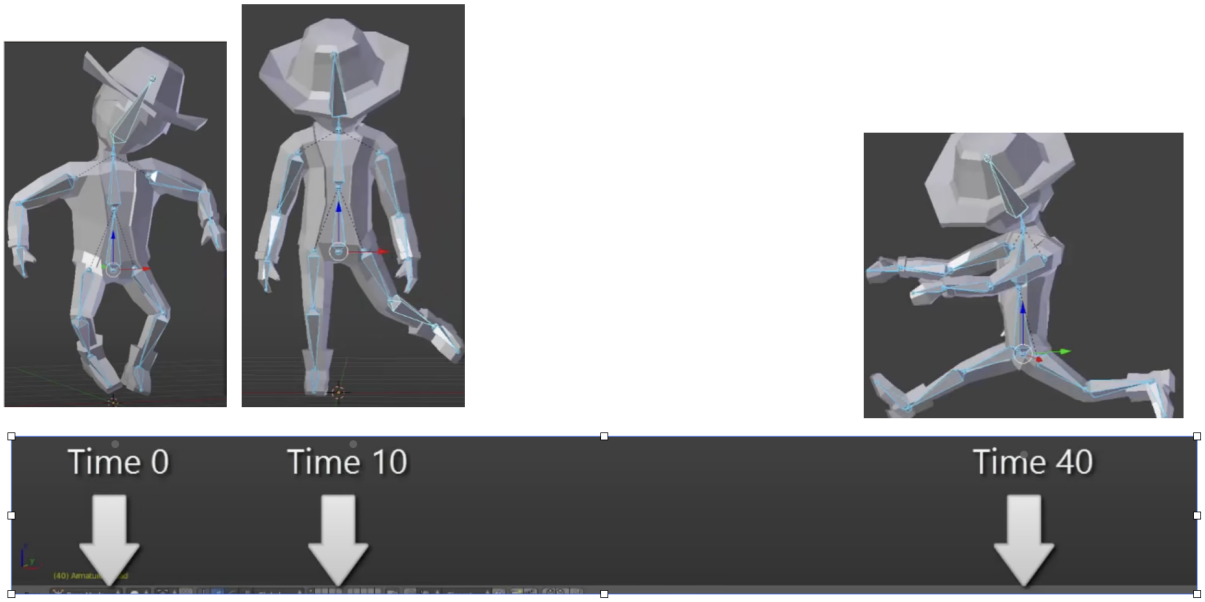


Fig. 14.2: Set time point at keyframes

formats are STL (with a .STL extension) and COLLADA (with a .DAE extension). Here is the list, where the 3D file formats are marked with their type.

Table 14.1: 3D file formats[Page 635, 7]

3D file format	Type
STL	Neutral
OBJ	ASCII variant is neutral, binary variant is proprietary
FBX	Proprietary
COLLADA	Neutral
3DS	Proprietary
IGES	Neutral
STEP	Neutral
VRML/X3D	Neutral

The four key features a 3D file can store include the model's geometry, the model's surface texture, scene details, and animation of the model[7].

Specifically, they can store details about four key features of a 3D model, though it's worth bearing in mind that you may not always take advantage of all four features in all projects, and not all file formats support all four features!

3D printer applications do not to support animation. CAD and CAM such as designing airplane does not need feature of scene details.

DAE (Collada) appeared in the video animation above. Collada files belong to a neutral format used heavily in the video game and film industries. It's managed by the non-profit technology consortium, the Khronos Group.

The file extension for the Collada format is .dae. The Collada format stores data using the XML mark-up language.

The original intention behind the Collada format was to become a standard among 3D file formats. Indeed, in 2013, it was adopted by ISO as a publicly available specification, ISO/PAS 17506. As a result, many 3D modeling programs

[7] https://all3dp.com/3d-file-format-3d-files-3d-printer-3d-cad-vrml-stl-obj/

support the Collada format.

That said, the consensus is that the Collada format hasn't kept up with the times. It was once used heavily as an interchange format for Autodesk Max/Maya in film production, but the industry has now shifted more towards OBJ, FBX, and Alembic[7].

14.1.2 Graphic SW stack

The role of CPU and GPU for graphic animation as Fig. 14.3.

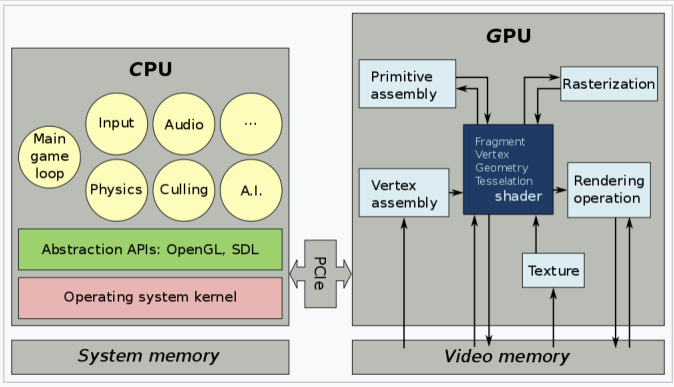


Fig. 14.3: OpenGL and Vulkan are both rendering APIs. In both cases, the GPU executes shaders, while the CPU executes everything else[8].

- GPU can't directly read user input from, say, keyboard, mouse, gamepad, or play audio, or load files from a hard drive, or anything like that. In this situation, cannot let GPU handle the animation work[9].

The driver run on CPU side as Fig. 14.4. The OpenGL Api will call driver's function eventually and driver finish the function's work via issuing GPU-HW's command and/or sending data to GPU. GPU's firmware only manage clock, voltage, power comsumption, ..., etc[10]. Even so, GPU's rendor work from the data of 3D vertex, colors, ... sending from CPU and storing in GPU's memory or shared memory consume more computing power than CPU.

- According the previous section, after user create skeleton and skin for each model and set keyframes time through 3D modeling tool, the 3D modeling tool can either generate Java code which calling JOGL (Java OpenGL)[Page 634, 6], or generate OpenCL API directly. The frame data can be calculated from interplation between keyframes.
- As above, every animation the client CPU program set new position of obect (vertices) and colors, the data of one frame, server (driver and GPU) does the 3D to 2D rendering. Higher-level libraries and frameworks on top of OpenGL provide animation framework and tools to generate OpenGL API and shaders from 3D model.
- Shader may call Builtin-functions which written from Compute Shader, spriv or LLVM-IR. LLVM libclc is a project for builtin-functions in OpenCL which can be used in OpenGL too[11]. Like CPU's builtin-functions, new GPU ISA/architecture has to implement their builtin-functions or porting from open source such as libclc.
- 3D model (CPU) does the rendering animation to generate each frame between keyframes (poses) while GPU does the rendering pipeline from each frame to each pixel's value.
- These frames data existed in the form of VAO (Vertex Array Object) in OpenGL. It will be explaned in later section OpenGL.

[8] https://en.wikipedia.org/wiki/Vulkan
[9] https://stackoverflow.com/questions/47426655/cpu-and-gpu-in-3d-game-whos-doing-what
[10] https://antonelly.com.co/do-gpus-have-firmware/#:~:text=Providing%20access%20to%20new%20features,drivers%20during%20the%20boot%20process
[11] https://libclc.llvm.org

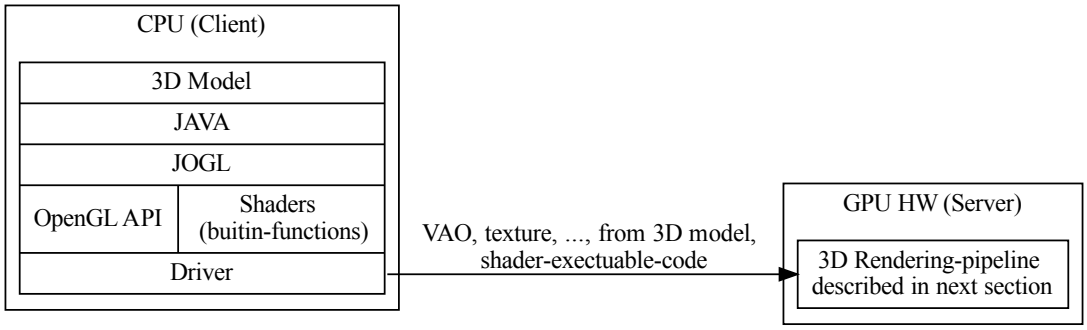


Fig. 14.4: Graphic SW Stack

- In addition, OpenGL provides vertex buffer object (VBO) allowing vertex array data to be stored in high-performance graphics memory on the server side and promotes efficient data transfer[12][13].

The flow for 3D/2D graphic processing as Fig. 14.5.

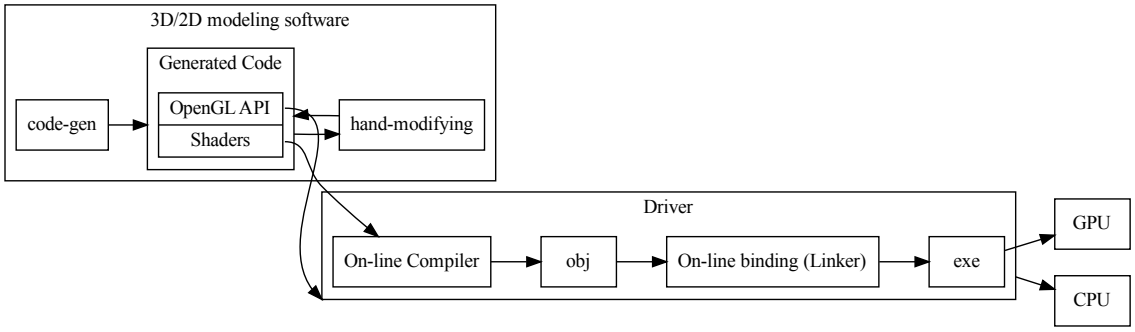


Fig. 14.5: OpenGL Flow

[12] http://www.songho.ca/opengl/gl_vbo.html

[13] If your models will be rigid, meaning you will not change each vertex individually, and you will render many frames with the same model, you will achieve the best performance not by storing the models in your class, but in vertex buffer objects (VBOs) https://gamedev.stackexchange.com/questions/19560/what-is-the-best-way-to-store-meshes-or-3d-models-in-a-class

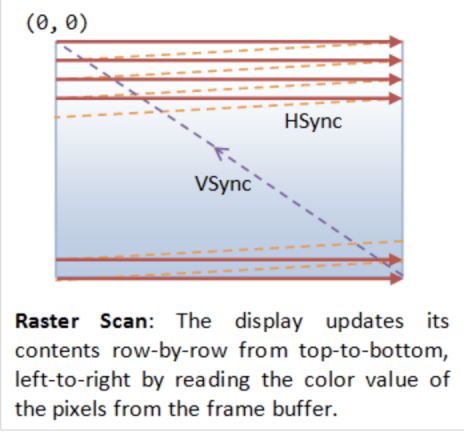


Fig. 14.6: VSync

VSync

```
No tearing, GPU and Display run at same refresh rate since GPU refresh faster
than Display.

              A    B

GPU       | ----| ----|

Display   |-----|-----|

            B      A

Tearing, GPU has exactly refresh cycles but VSync takes one cycle more.
than Display.

              A

GPU       | -----|

Display   |-----|-----|

            B      A

Avoid tearing, GPU has refresh rate 1/2 of Display's refresh rate.
than Display.

              A          B

GPU       | -----|     | -----|

Display   |-----|-----|-----|-----|

            B      B    A     A
```

- Double Buffering

While the display is reading from the frame buffer to display the current frame, we might be updating its contents for the next frame (not necessarily in raster-scan manner). This would result in the so-called tearing, in which the screen shows parts of the old frame and parts of the new frame. This could be resolved by using so-called double buffering. Instead of using a single frame buffer, modern GPU uses two of them: a front buffer and a back buffer. The display reads from the front buffer, while we can write the next frame to the back buffer. When we finish, we signal to GPU to swap the front and back buffer (known as buffer swap or page flip).

- VSync

  Double buffering alone does not solve the entire problem, as the buffer swap might occur at an inappropriate time, for example, while the display is in the middle of displaying the old frame. This is resolved via the so-called vertical synchronization (or VSync) at the end of the raster-scan. When we signal to the GPU to do a buffer swap, the GPU will wait till the next VSync to perform the actual swap, after the entire current frame is displayed.

  As above text digram. The most important point is: When the VSync buffer-swap is enabled, you cannot refresh the display faster than the refresh rate of the display!!! If GPU is capable of producing higher frame rates than the display's refresh rate, then GPU can use fast rate without tearing. If GPU has same or less frame rates then display's and you application refreshes at a fixed rate, the resultant refresh rate is likely to be an integral factor of the display's refresh rate, i.e., 1/2, 1/3, 1/4, etc. Otherwise it will cause tearing[Page 634, 1].

- NVIDIA G-SYNC and AMD FreeSync

  If your monitor and graphics card both in your customer computer support NVIDIA G-SYNC, you're in luck. With this technology, a special chip in the display communicates with the graphics card. This lets the monitor vary the refresh rate to match the frame rate of the NVIDIA GTX graphics card, up to the maximum refresh rate of the display. This means that the frames are displayed as soon as they are rendered by the GPU, eliminating screen tearing and reducing stutter for when the frame rate is both higher and lower than the refresh rate of the display. This makes it perfect for situations where the frame rate varies, which happens a lot when gaming. Today, you can even find G-SYNC technology in gaming laptops!

  AMD has a similar solution called FreeSync. However, this doesn't require a proprietary chip in the monitor. In FreeSync, the AMD Radeon driver, and the display firmware handle the communication. Generally, FreeSync monitors are less expensive than their G-SYNC counterparts, but gamers generally prefer G-SYNC over FreeSync as the latter may cause ghosting, where old images leave behind artifacts[14].

14.1.3 Basic geometry in computer graphics

- Additive colors in light as Fig. 14.7[15][16]. If in paints, it adds shade and become light grey since it add shade (dark color)[17].

This section instroduces the basic geometry math for computer graphics. Every computer graphics book has provided topics of transformation of object and position in space. Chapter 4 of Blue book: OpenGL SuperBible 7th Edition give a short description (40 pages) and useful concept is a good material for knowing the concept. Given the following for Quaternion Product (Hamilton product) from Wiki[18] since the book miss this.

$$\mathbf{i}j = -ji = k, jk = -kj = i, ki = -ik = j.$$

The complete concept can be found in Book: "Computer graphics principles and practice 3rd editon, authors: JOHN F, …". But it is 1 thousand of pages. This book is very complete and may take much time to understand every detail.

Both Triangles or Quads are ploygon. So, objects can be formed with ploygon in both 2D and 3D. About transfermation in 2D or 3D, almost every book of computer graphics has mentioned well already. This section introduces the most

[14] https://www.avadirect.com/blog/frame-rate-fps-vs-hz-refresh-rate/
[15] https://en.wikipedia.org/wiki/RGB_color_model
[16] https://www.youtube.com/watch?v=kEnz_3miiAc
[17] https://www.tiktok.com/@tonesterpaints/video/7059565281227853102
[18] https://en.wikipedia.org/wiki/Quaternion

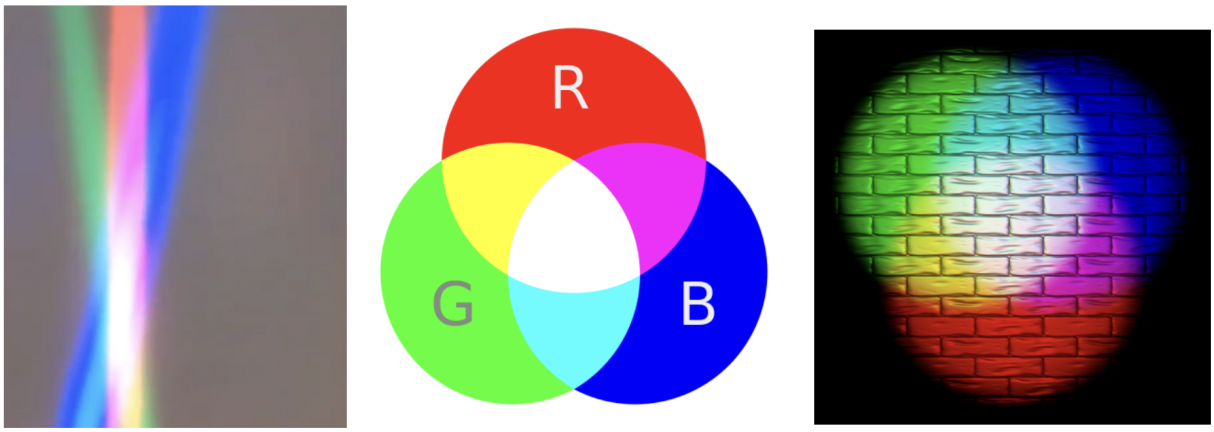


Fig. 14.7: Additive colors in light

important concept and method for deciding Inner and Outer planes, then a point or object can be checked for showing or hidding during 2D or 3D rendering.

Any area of polygon can be calculated by dividing into Triangles or Quads. And any area of Triangle or Quad can be calculated by cross product in 3D. The cross product in **3D** is defined by the formula and can be represented with matrix notation as proved here[19].

$$\mathbf{a}\mathbf{x}\mathbf{b} = \|a\|\|b\|sin(\Theta)n$$

$$\mathbf{a}\mathbf{x}\mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix}$$

The cross product in **2D** is defined by the formula and can be represented with matrix notation as proved here[20][21].

$$\mathbf{a}\mathbf{x}\mathbf{b} = \|a\|\|b\|sin(\Theta)$$

$$\mathbf{a}\mathbf{x}\mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & 0 \\ b_1 & b_2 & 0 \end{vmatrix} = \begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \end{bmatrix}$$

After above matrix form is proved, the Antisymmetric may be proved as follows,

$$axb = \mathrm{x} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \end{bmatrix} = a_1 b_2 - a_2 b_1 =$$

$$-b_1 a_2 - (-b_2 a_1) = \begin{bmatrix} -b_1 & -b_2 \\ a_1 & a_2 \end{bmatrix} = \mathrm{x} \begin{bmatrix} -b \\ a \end{bmatrix} = -b\mathrm{x}a$$

In 2D, any two points from P_i to P_{i+1} can form a vector and decide inner side or outer side. For example, as Fig. 14.8, Θ is the angle from P_iP_{i+1} to $P_iP'_{i+1} = 180°$. So, with right-hand rule, counter clockwise order, any P_iQ between P_iP_{i+1} to $P_iP'_{i+1}$, the angle of P_iP_{i+1} to $P_iQ = \theta, 0° < \theta < 180°$ then the inward direction be decided.

Base on this observation, the rule for inward and outward to any vector as Fig. 14.8. Face the same direction of a specific vector, the left side is inward and right side is outward as Fig. 14.9.

For each edge $P_i - P_{i+1}$, the inward edge normal is the vector $\mathrm{x}\ v_i$; the outward edge normal is $-\mathrm{x}\ v_i$. Where $\mathrm{x}\ v_i$ is coss-product(v_i) as Fig. 14.8.

[19] https://en.wikipedia.org/wiki/Cross_product
[20] https://www.xarg.org/book/linear-algebra/2d-perp-product/
[21] https://www.nagwa.com/en/explainers/175169159270/

Fig. 14.8: Inward edge normals

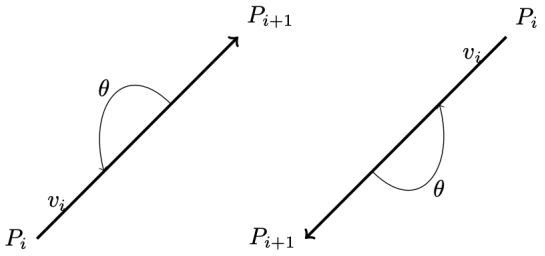


Fig. 14.9: Inward and outward in 2D for a vector.

Polygon can be created from vertices. Suppose that $(P_0, P_1, ..., P_n)$ is a polygon. The line segments P_0P_1, P_1P_2, etc., are the edges of the polygon; the vectors $v_0 = P_1 - P_0, v_1 = P_2 - P_1, ..., v_n = P_0 - P_n$ are the edges of the polygon. Through counter clockwise, the left side is inward, then the inward region of polygon can be decided.

For a convex polygon whose vertices are listed in counter clockwise order, the inward edge normals point toward the interior of the polygon, and the outward edge normals point toward the unbounded exterior of the polygon, corresponding to our ordinary intuition. But if the vertices of a polygon are given in clockwise order, the interior and exterior swap roles.

This cross product has an important property: Going from v to ×v involves a rotation by 90° in the same direction as the rotation that takes the positive x-axis to the positive y-axis.

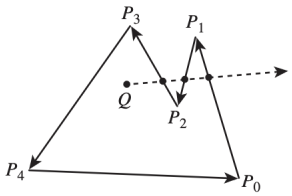


Fig. 14.10: Draw a polygon with vectices counter clockwise

As Fig. 14.10, when drawing polygon with vectors(lines) counter clockwise, the ploygon will be created and the two sides of a vector(line) can be indentified[22]. Further a point in polygon or out of polygon can be identified. One simple way of finding whether the point is inside or outside a simple polygon is to test how many times a ray, starting from the point and going in any fixed direction, intersects the edges of the polygon. If the point is on the outside of the polygon the ray will intersect its edge an even number of times. If the point is on the inside of the polygon then it will intersect

[22] Figure 7.19 of Book: Computer graphics principles and practice 3rd edition

the edge an odd number of times[23].

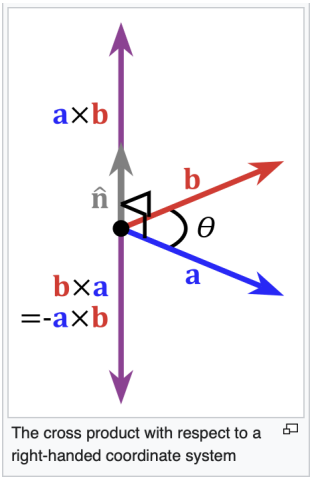


Fig. 14.11: Cross product definition in 3D

In the same way, through following the same direction counter clockwise to create 2D polygon one by one, then the 3D polygon will be created. As Fig. 14.11 from wiki[Page 640, 19], the inward direction can be decided with a x b < 0 and outward is a x b > 0 in OpenGL. Replace a, b with x, y as Fig. 14.12 axis z+ is the outer surface and z- is the inner surface[24].

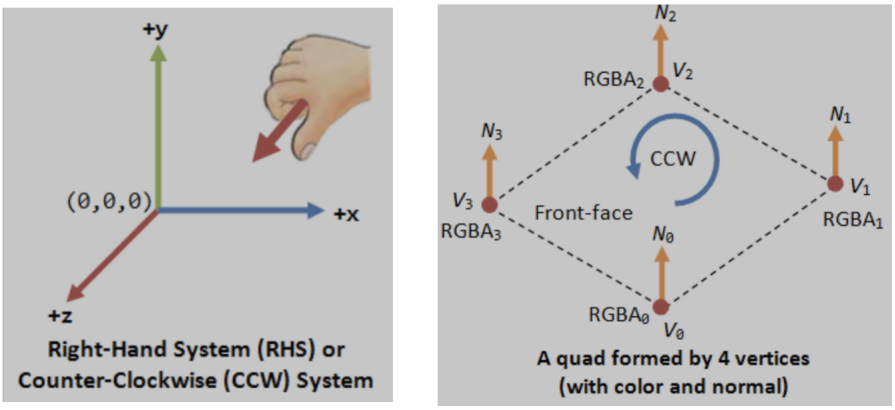


Fig. 14.12: OpenGL pointing outwards, indicating the outer surface (z axis is +)

The Fig. 14.13 is an example of 3D polygon created by 2D triangles. The direction of plane (triangle) as the line perpendicular to the plane.

Cast a ray from the 3D point along X-axis and check how many intersections with outer object you find. Depending on the intersection number on each axis (even or odd) you can understand if your point is inside or outside[25]. Inside is odd and outside is even. As Fig. 14.14, points on the line going through the object satisfy this rule.

[23] https://en.wikipedia.org/wiki/Point_in_polygon

[24] Normals are used to differentiate the front- and back-face, and for other processing such as lighting. Right-hand rule (or counter-clockwise) is used in OpenGL. The normal is pointing outwards, indicating the outer surface (or front-face). https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html

[25] https://stackoverflow.com/questions/63557043/how-to-determine-whether-a-point-is-inside-or-outside-a-3d-model-computationally

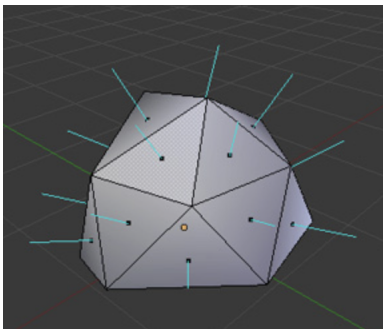


Fig. 14.13: 3D polygon with directions on each plane

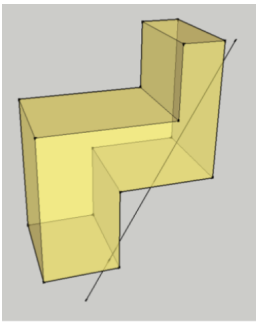


Fig. 14.14: Point in or out 3D object

OpenGL uses counter clockwise and pointing outwards as default[Page 637, 12]**.**

```
// unit cube
// A cube has 6 sides and each side has 4 vertices, therefore, the total number
// of vertices is 24 (6 sides * 4 verts), and 72 floats in the vertex array
// since each vertex has 3 components (x,y,z) (= 24 * 3)
//    v6----- v5
//   /|      /|
//  v1------v0|
//  | |     | |
//  | v7----|-v4
//  |/      |/
//  v2------v3

// vertex position array
GLfloat vertices[]  = {
   .5f, .5f, .5f,  -.5f, .5f, .5f,  -.5f,-.5f, .5f,  .5f,-.5f, .5f, // v0,v1,v2,v3␣
↪(front)
   .5f, .5f, .5f,   .5f,-.5f, .5f,   .5f,-.5f,-.5f,  .5f, .5f,-.5f, // v0,v3,v4,v5␣
↪(right)
   .5f, .5f, .5f,   .5f, .5f,-.5f,  -.5f, .5f,-.5f, -.5f, .5f, .5f, // v0,v5,v6,v1 (top)
  -.5f, .5f, .5f,  -.5f, .5f,-.5f,  -.5f,-.5f,-.5f, -.5f,-.5f, .5f, // v1,v6,v7,v2 (left)
  -.5f,-.5f,-.5f,   .5f,-.5f,-.5f,   .5f,-.5f, .5f, -.5f,-.5f, .5f, // v7,v4,v3,v2␣
↪(bottom)
   .5f,-.5f,-.5f,  -.5f,-.5f,-.5f,  -.5f, .5f,-.5f,  .5f, .5f,-.5f  // v4,v7,v6,v5 (back)
};
```

From code above, we can see that OpenGL uses counter clockwise and pointing outwards as default. However OpenGL provides glFrontFace(GL_CW) for clockwise[26].

For group of objects, scene graph provides better animation and saving memory[27].

14.2 OpenGL

14.2.1 Example of OpenGL program

The following example from openGL redbook and example code[32][33].

References/triangles.vert

```
#version 400 core

layout( location = 0 ) in vec4 vPosition;

void
main()
{
    gl_Position = vPosition;
}
```

References/triangles.frag

```
#version 450 core

out vec4 fColor;

void main()
{
    fColor = vec4(0.5, 0.4, 0.8, 1.0);
}
```

References/01-triangles.cpp

```
1  //////////////////////////////////////////////////////////////////////////////
2  //
3  //  Triangles.cpp
4  //
5  //////////////////////////////////////////////////////////////////////////////
6
7  #include "vgl.h"
```

(continues on next page)

[26] https://registry.khronos.org/OpenGL-Refpages/gl4/html/glFrontFace.xhtml
[27] https://en.wikipedia.org/wiki/Scene_graph
[32] http://www.opengl-redbook.com
[33] https://github.com/openglredbook/examples

(continued from previous page)

```
#include "LoadShaders.h"

enum VAO_IDs { Triangles, NumVAOs };
enum Buffer_IDs { ArrayBuffer, NumBuffers };
enum Attrib_IDs { vPosition = 0 };

GLuint  VAOs[NumVAOs];
GLuint  Buffers[NumBuffers];

const GLuint  NumVertices = 6;

//----------------------------------------------------------------------------
//
// init
//

void
init( void )
{
    glGenVertexArrays( NumVAOs, VAOs ); // Same with glCreateVertexArray( NumVAOs, VAOs );
      // https://stackoverflow.com/questions/24441430/glgen-vs-glcreate-naming-convention
    // Make the new VAO:VAOs[Triangles] active, creating it if necessary.
    glBindVertexArray( VAOs[Triangles] );
    // opengl->current_array_buffer = VAOs[Triangles]

    GLfloat  vertices[NumVertices][2] = {
        { -0.90f, -0.90f }, {  0.85f, -0.90f }, { -0.90f,  0.85f },  // Triangle 1
        {  0.90f, -0.85f }, {  0.90f,  0.90f }, { -0.85f,  0.90f }   // Triangle 2
    };

    glCreateBuffers( NumBuffers, Buffers );

    // Make the buffer the active array buffer.
    glBindBuffer( GL_ARRAY_BUFFER, Buffers[ArrayBuffer] );
    // Attach the active VBO:Buffers[ArrayBuffer] to VAOs[Triangles]
    // as an array of vectors with 4 floats each.
    // Kind of like:
    // opengl->current_vertex_array->attributes[attr] = {
    //     type = GL_FLOAT,
    //     size = 4,
    //     data = opengl->current_array_buffer
    // }
    // Can be replaced with glVertexArrayVertexBuffer(VAOs[Triangles], Triangles,
    // buffer[ArrayBuffer], ArrayBuffer, sizeof(vmath::vec2));,glVertexArrayAttribFormat(), ...
    // in OpenGL 4.5.

    glBufferStorage( GL_ARRAY_BUFFER, sizeof(vertices), vertices, 0);

    ShaderInfo  shaders[] =
    {
```

(continues on next page)

(continued from previous page)

```
        { GL_VERTEX_SHADER, "media/shaders/triangles/triangles.vert" },
        { GL_FRAGMENT_SHADER, "media/shaders/triangles/triangles.frag" },
        { GL_NONE, NULL }
    };

    GLuint program = LoadShaders( shaders );
    glUseProgram( program );

    glVertexAttribPointer( vPosition, 2, GL_FLOAT,
                           GL_FALSE, 0, BUFFER_OFFSET(0) );
    glEnableVertexAttribArray( vPosition );
    // Above two functions specify vPosition to vertex shader at layout (location = 0)
}

//----------------------------------------------------------------------------
//
// display
//

void
display( void )
{
    static const float black[] = { 0.0f, 0.0f, 0.0f, 0.0f };

    glClearBufferfv(GL_COLOR, 0, black);

    glBindVertexArray( VAOs[Triangles] );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
}

//----------------------------------------------------------------------------
//
// main
//

#ifdef _WIN32
int CALLBACK WinMain(
  _In_ HINSTANCE hInstance,
  _In_ HINSTANCE hPrevInstance,
  _In_ LPSTR     lpCmdLine,
  _In_ int       nCmdShow
)
#else
int
main( int argc, char** argv )
#endif
{
    glfwInit();

    GLFWwindow* window = glfwCreateWindow(800, 600, "Triangles", NULL, NULL);

    glfwMakeContextCurrent(window);
```

(continues on next page)

(continued from previous page)

```
    gl3wInit();

    init();

    while (!glfwWindowShouldClose(window))
    {
        display();
        glfwSwapBuffers(window);
        glfwPollEvents();
    }

    glfwDestroyWindow(window);

    glfwTerminate();
}
```

Init():

- Generate Vertex Array VAOs and bind VAOs[0].

  (glGenVertexArrays(NumVAOs, VAOs); glBindVertexArray(VAOs[Triangles]); glCreateBuffers(NumBuffers, Buffers);) A vertex-array object holds various data related to a collection of vertices. Those data are stored in buffer objects and managed by the currently bound vertex-array object.

  - glBindBuffer(GL_ARRAY_BUFFER, Buffers[ArrayBuffer]);

    Because there are many different places where buffer objects can be in OpenGL, when we bind a buffer, we need to specify which what we'd like to use it for. In our example, because we're storing vertex data into the buffer, we use GL_ARRAY_BUFFER. The place where the buffer is bound is known as the binding target.

- According counter clockwise rule in previous section, Triangle Primitives are defined in varaible vertices. After binding OpenGL VBO Buffers[0] to vertices, vertices data will send to memory of server(gpu). Think of the "active" buffer as just a global variable, and there are a bunch of functions which use the active buffer instead of using a parameter. These global state variables are the ugly side of OpenGL[34] and can be replaced with glVertexArrayVertexBuffer(), glVertexArrayAttribFormat(), ..., then call glBindVertexArray(vao) before drawing in OpenGL 4.5[35][36].

- glVertexAttribPointer(vPosition, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0)): During gpu rendering, each vertex position will be held in vPosition and pass to "triangles.vert" shader because LoadShaders(shaders).

glfwSwapBuffers(window):

- You've already used double buffering for animation. Double buffering is done by making the main color buffer have two parts: a front buffer that's displayed in your window; and a back buffer, which is where you render the new image. When you swap the buffers (by calling glfwSwapBuffers(), for example), the front and back buffers are exchanged[71].

display():

- Bind VAOs[0], set render mode to GL_TRIANGLES and send vertex data to Buffer (gpu memory, OpenGL pipeline). Next, GPU will do rendering pipeline descibed in next section.

[34] https://stackoverflow.com/questions/21652546/what-is-the-role-of-glbindvertexarrays-vs-glbindbuffer-and-what-is-their-relatio
[35] https://registry.khronos.org/OpenGL-Refpages/gl4/html/glBindVertexBuffer.xhtml
[36] Page 152 of Blue book: OpenGL SuperBible 7th Edition.
[71] Section of Color Buffer, page 222-223 of book "OpenGL Programming Guide 9th Edition"Page 644, 32.

The triangles.vert has input vPosition and no output variable, so using gl_Position default varaible without declaration. The triangles.frag has not defined input variable and has defined output variable fColor instead of using gl_FragColor.

The "in" and "out" in shaders above are "type qualifier". A type qualifier is used in the OpenGL Shading Language (GLSL) to modify the storage or behavior of global and locally defined variables. These qualifiers change particular aspects of the variable, such as where they get their data from and so forth[40].

Though attribute and varying are removed from later version 1.4 of OpenGL, many materials in website using them[41][42]. It's better to use "in" and "out" to replace them as the following code. OpenGL has a few ways to binding API's variable with shader's variable. glVertexAttrib* as the following code and glBindAttribLocation()[43], ...

replace attribute and varying with in and out

```
uniform float scale;
layout (location = 0) attribute vec2 position;
// layout (location = 0) in vec2 position;
layout (location = 1) attribute vec4 color;
// layout (location = 1) in vec4 color;
varying vec4 v_color;
// out v_color

void main()
{
  gl_Position = vec4(position*scale, 0.0, 1.0);
  v_color = color;
}
```

```
// OpenGL API
GLfloat attrib[] = { x * 0.5f, x * 0.6f, x* 0.4f, 0.0f };
// Update the value of input attribute 1 : layout (location = 1) in vec4 color
glVertexAttrib4fv(1, attrib);
```

```
varying vec4 v_color;
// in vec4 v_color;

void main()
{
  gl_FragColor = v_color;
}
```

An OpenGL program is made of two shaders[38][39]:

- The vertex shader is (commonly) executed once for every vertex we want to draw. It receives some attributes as input, computes the position of this vertex in space and returns it in a variable called gl_Position. It also defines some varyings.
- The fragment shader is executed once for each pixel to be rendered. It receives some varyings as input, computes the color of this pixel and returns it in a variable called fColor.

[40] https://www.khronos.org/opengl/wiki/Type_Qualifier_(GLSL)
[41] https://www.khronos.org/opengl/wiki/Type_Qualifier_(GLSL)#Removed_qualifiers
[42] https://github.com/vispy/vispy/issues/242
[43] https://www.khronos.org/opengl/wiki/Layout_Qualifier_(GLSL)
[38] https://engineering.monstar-lab.com/en/post/2022/03/01/Introduction-To-GPUs-With-OpenGL/
[39] https://glumpy.github.io/modern-gl.html

Since we have 6 vertices in our buffer, this shader will be executed 6 times by the GPU (once per vertex)! We can also expect all 6 instances of the shader to be executed in parallel, since a GPU have so many cores.

14.2.2 3D Rendering

Based on the previous section of 3D modeling, the 3D modeling tool will generate 3D vertex model and OpenGL code, then programmers may hand-change OpenGL code and add or modify shaders. The 3D animation will trigger the 3D rendering for each 2D image drawing.

3D rendering is the process of converting 3D models into 2D images on a computer[28]. The steps as the following Fig. 14.15.

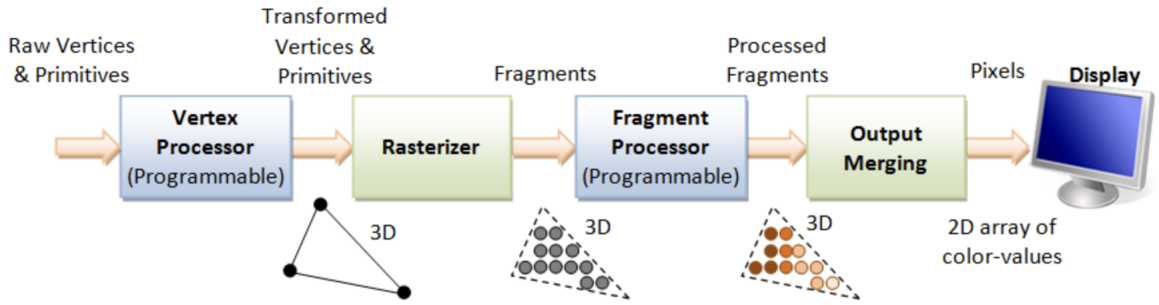


Fig. 14.15: 3D Graphics Rendering Pipeline

- A fragment can be treated as a pixel in 3D spaces, which is aligned with the pixel grid, with attributes such as position, color, normal and texture.

The complete steps as the following Fig. 14.16 from OpenGL website[29] and the website has descripiton for each stage.

In addition, list OpenGL rendering pipeline Figure 1.2 and stage from book "OpenGL Programming Guide 9th Edition"Page 644, 32 as follows,

[28] https://en.wikipedia.org/wiki/3D_rendering
[29] https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

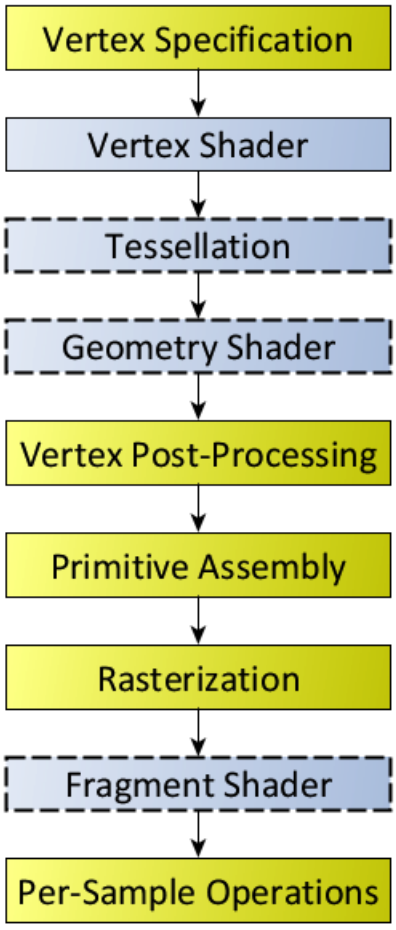


Fig. 14.16: Diagram of the Rendering Pipeline. The blue boxes are programmable shader stages.

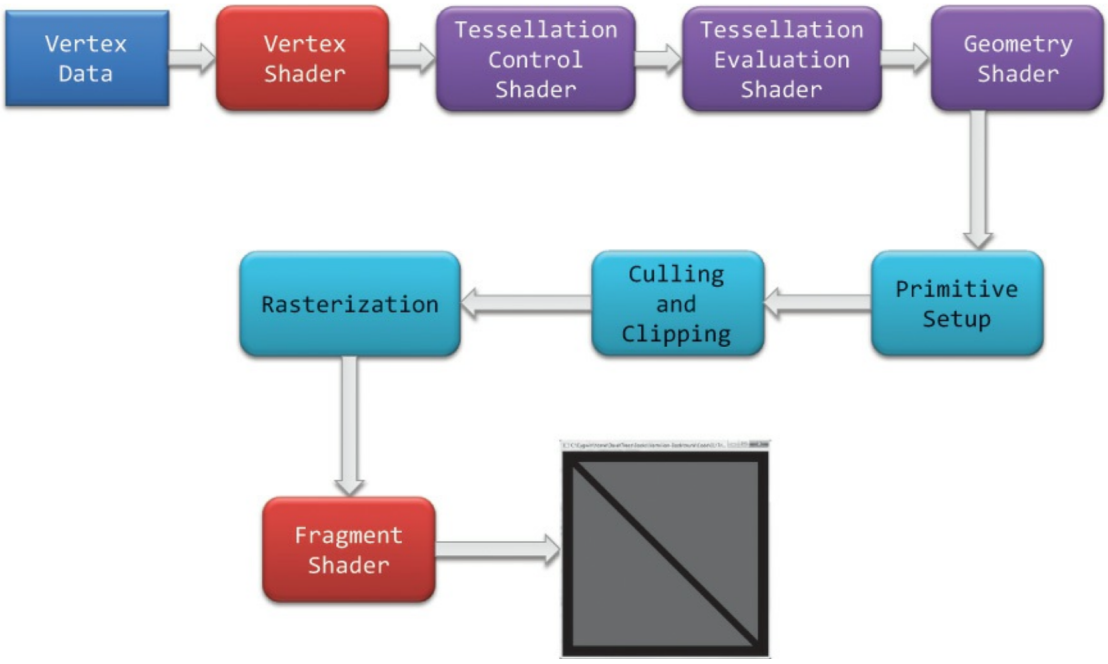


Figure 1.2 OpenGL pipeline

Table 14.2: OpenGL rendering pipeline from page 10 of book "OpenGL Programming Guide 9th Edition"[Page 644, 32] and[Page 649, 29].

Stage.	Description
Vertex Specification	After setting data as the example of previous section, glDrawArrays() will send data to gpu through buffer objects.
Vertex Shading	**Vertex -> Vertex and other data such as color for later passes.** For each vertex that is issued by a drawing command, a vertex shader will be called to process the data associated with that vertex.
Tessellation Shading	**Create more detail on demand when room in.** After the vertex shader has processed each vertex's associated data, the tessellation shader stage will continue processing that data, if it's been activated. Reference below.
Geometry Shading	**The next shader stage, geometry shading, allows additional processing of individual geometric primitives, including creating new ones, before rasterization.** Chapter 10 of Red Book[Page 644, 32] has details.
Primitive Assembly	The previous shading stages all operate on vertices, with the information about how those vertices are organized into geometric primitives being carried along internal to OpenGL. **The primitive assembly stage organizes the vertices into their associated geometric primitives in preparation for clipping and rasterization.**
Clipping	**Clipping hidden parts.** Occasionally, vertices will be outside of the viewport—the region of the window where you're permitted to draw—and cause the primitive associated with that vertex to be modified so none of its pixels are outside of the viewport. This operation is called clipping and is handled automatically by OpenGL.
Rasterization	**Vertex -> Fragment.** The job of the rasterizer is to determine which screen locations are covered by a particular piece of geometry (point, line, or triangle). Knowing those locations, along with the input vertex data, the rasterizer linearly interpolates the data values for each varying variable in the fragment shader and sends those values as inputs into your fragment shader. A fragment can be treated as a pixel in 3D spaces, which is aligned with the pixel grid, with attributes such as position, color, normal and texture.
Fragment Shading	**Determine color for each pixel.** The final stage where you have programmable control over the color of a screen location is fragment shading. In this shader stage, you use a shader to determine the fragment's final color (although the next stage, per-fragment operations, can modify the color one last time) and potentially its depth value. Fragment shaders are very powerful, as they often employ texture mapping to augment the colors provided by the vertex processing stages. A fragment shader may also terminate processing a fragment if it determines the fragment shouldn't be drawn; this process is called fragment discard. A helpful way of thinking about the difference between shaders that deal with vertices and fragment shaders is this: vertex shading (including tessellation and geometry shading) determines where on the screen a primitive is, while fragment shading uses that information to determine what color that fragment will be.
Per-Fragment Operations	During this stage, a fragment's visibility is determined using depth testing (also commonly known as z-buffering) and stencil testing. If a fragment successfully makes it through all of the enabled tests, it may be written directly to the framebuffer, updating the color (and possibly depth value) of its pixel, or **if blending is enabled, the fragment's color will be combined with the pixel's current color to generate a new color that is written into the framebuffer.**

- Tessellation Shading: The core problem that Tessellation deals with is the static nature of 3D models in terms of their detail and polygon count. The thing is that when we look at a complex model such as a human face up close we prefer to use a highly detailed model that will bring out the tiny details (e.g. skin bumps, etc). A highly detailed model automatically translates to more triangles and more compute power required for processing. … One possible way to solve this problem using the existing features of OpenGL is to generate the same model at multiple levels of detail (LOD). For example, highly detailed, average and low. We can then select the version

> to use based on the distance from the camera. This, however, will require more artist resources and often will not be flexible enough. ... Let's take a look at how Tessellation has been implemented in the graphics pipeline. The core components that are responsible for Tessellation are two new shader stages and in between them a fixed function stage that can be configured to some degree but does not run a shader. The first shader stage is called Tessellation Control Shader (TCS), the fixed function stage is called the Primitive Generator (PG), and the second shader stage is called Tessellation Evaluation Shader (TES). Some GPU havn't this fixed function stage implemented in HW and even havn't provide these TCS, TES and Gemoetry Shader. User can write Compute Shaders instead for this on-fly detail display. This surface is usually defined by some polynomial formula and the idea is that moving a CP has an effect on the entire surface. ... The group of CPs is usually called a Patch[30]. Chapter 9 of Red Book[Page 644, 32] has details.

For 2D animation, the model is created by 2D only (1 face only), so it only can be viewed from the same face of model. If you want to display different faces of model, multiple 2D models need to be created and switch these 2D models from face(flame) to face(flame) from time to time[31].

14.2.3 GLSL (GL Shader Language)

OpenGL is a standard for designing 2D/3D animation in computer graphic. To do animation well, OpenGL provides a lots of api(functions) call for graphic processing. The 3D model construction tools such as Maya, Blender, ..., etc, only need to call this api to finish the 3D to 2D projecting function in computer. Any GPU hardware dependent code in these api provided by GPU manufacturer. An OpenGL program looks like the following,

Vertex shader

```
#version 330 core
layout (location = 0) in vec3 aPos; // the position variable has attribute position 0

out vec4 vertexColor; // specify a color output to the fragment shader

void main()
{
    gl_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's␣
↪constructor
    vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // set the output variable to a dark-red␣
↪color
}
```

Fragment shader

```
#version 330 core
out vec4 FragColor;

in vec4 vertexColor; // the input variable from the vertex shader (same name and same␣
↪type)

void main()
{
```

(continues on next page)

[30] https://ogldev.org/www/tutorial30/tutorial30.html

[31] https://tw.video.search.yahoo.com/search/video?fr=yfp-search-sb&p=2d+animation#id=12&vid=46be09edf57b960ae79e9cd077eea1ea&action=view

(continued from previous page)

```
    FragColor = computeColorOfThisPixel(...);
}
```

OpenGl user program

```
int main(int argc, char ** argv)
{
  // init window, detect user input and do corresponding animation by calling opengl api
  ...
}
```

The last main() is programed by user obviously. Let's explain what the first two main() work for. As you know, the OpenGL is a lots of api to let programmer display the 3D object into 2D computer screen explained from book of concept of computer graphic. 3D graphic model can set light and object texture by user firstly, and calculating the postion of each vertex secondly, then color for each pixel automatically by 3D software and GPU thirdly, finally display the color of each pixel in computer screen. But in order to let user/programmer add some special effect or decoration in coordinate for each vertex or in color for each pixel, OpenGL provides these two functions to do it. OpenGL uses fragment shader instead of pixel is : "Fragment shaders are a more accurate name for the same functionality as Pixel shaders. They aren't pixels yet, since the output still has to past several tests (depth, alpha, stencil) as well as the fact that one may be using antialiasing, which renders one-fragment-to-one-pixel non-true[44]. Programmer is allowed to add their converting functions that compiler translate them into GPU instructions running on GPU processor. With these two shaders, new features have been added to allow for increased flexibility in the rendering pipeline at the vertex and fragment level[45]. Unlike the shaders example here[46], some converting functions for coordinate in vertex shader or for color in fragment shade are more complicated according the scenes of animation. Here is an example[47]. In wiki shading page[Page 634, 3], Gourand and Phong shading methods make the surface of object more smooth by glsl. Example glsl code of Gourand and Phong shading on OpenGL api are here[48]. Since the hardware of graphic card and software graphic driver can be replaced, the compiler is run on-line meaning driver will compile the shaders program when it is run at first time and kept in cache after compilation[49].

The shaders program is C-like syntax and can be compiled in few mini-seconds, add up this few mini-seconds of on-line compilation time in running OpenGL program is a good choice for dealing the cases of driver software or gpu hardware replacement[50].

[44] https://community.khronos.org/t/pixel-vs-fragment-shader/52838

[45] https://en.m.wikipedia.org/wiki/OpenGL_Shading_Language

[46] https://learnopengl.com/Getting-started/Shaders

[47] https://www.youtube.com/watch?v=LyoSSoYyfVU at 5:25 from beginning: combine different textures.

[48] https://github.com/ruange/Gouraud-Shading-and-Phong-Shading

[49] Compiler and interpreter: (https://www.guru99.com/difference-compiler-vs-interpreter.html). AOT compiler: compiles before running; JIT compiler: compiles while running; interpreter: runs (reference https://softwareengineering.stackexchange.com/questions/246094/understanding-the-differences-traditional-interpreter-jit-compiler-jit-interp). Both online and offline compiler are AOT compiler. User call OpenGL api to run their program and the driver call call online compiler to compile user's shaders without user compiling their shader before running their program. When user run a CPU program of C language, he must compile C program before running the program. This is offline compiler.

[50] https://community.khronos.org/t/offline-glsl-compilation/61784

14.2.4 OpenGL Shader compiler

OpenGL standard is here[51]. The OpenGL is for desktop computer or server while the OpenGL ES is for embedded system[52]. Though shaders are only a small part of the whole OpenGL software/hardware system. It is still a large effort to finish the compiler implementation since there are lots of api need to be implemented. For example, there are 80 related texture APIs[53]. This implementation can be done by generating llvm extended intrinsic functions from shader parser of frontend compiler as well as llvm backend converting those intrinsic to gpu instructions as follows,

Fragment shader

```
#version 320 es
uniform sampler2D x;
out vec4 FragColor;

void main()
{
    FragColor = texture(x, uv_2d, bias);
}
```

llvm-ir

```
...
!1 = !{!"sampler_2d"}
!2 = !{i32 SAMPLER_2D} ; SAMPLER_2D is integer value for sampler2D, for example: 0x0f02
; A named metadata.
!x_meta = !{!1, !2}

define void @main() #0 {
    ...
    %1 = @llvm.gpu0.texture(metadata !x_meta, %1, %2, %3); ; %1: %sampler_2d, %2: %uv_2d,
↪ %3: %bias
    ...
}
```

asm of gpu

```
...
// gpu machine code
load $1, tex_a;
sample2d_inst $1, $2, $3 // $1: tex_a, $2: %uv_2d, $3: %bias

.tex_a // Driver set the index of gpu descriptor regsters here
```

As the bottom of code above, .tex_a memory address includes the Texture Object which binding by driver in on-line compilation/linking. Through binding Texture Object (SW) and Texture Unit (HW) with OpenGL API, gpu will uses Texture Unit HW resources efficiently. Explaining it the following.

[51] https://www.khronos.org/registry/OpenGL-Refpages/
[52] https://en.wikipedia.org/wiki/OpenGL_ES
[53] All the api listed in section 8.9 of https://www.khronos.org/registry/OpenGL/specs/es/3.2/GLSL_ES_Specification_3.20.html#texture-functions

About llvm intrinsic extended function, please refer this book here[54].

```
gvec4 texture(gsampler2D sampler, vec2 P, [float bias]);
```

GPU provides 'Texture Unit' to speedup fragment shader. However the 'Texture Unit' HW is expensive resouce and only few of them in a GPU. Driver can associate 'Texture Unit' to sampler variable by OpenGL api and switch between shaders as the following statements.

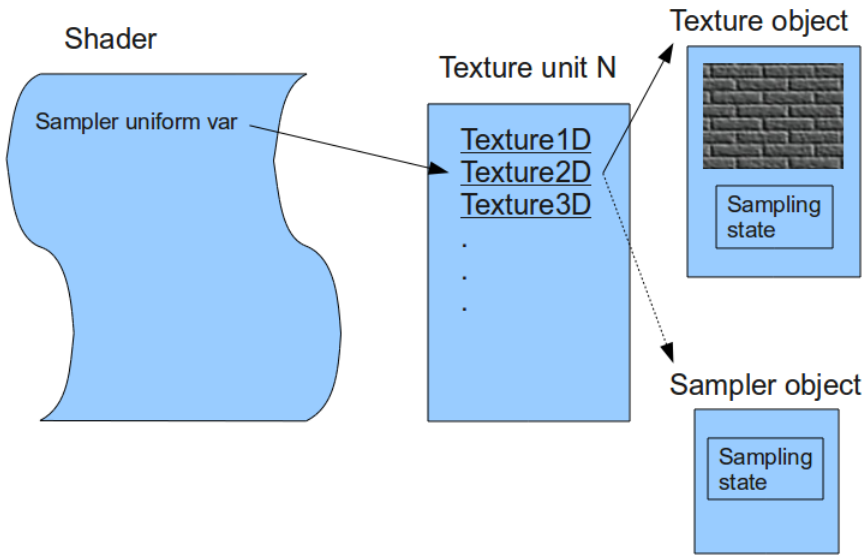


Fig. 14.17: Relationships between the texturing concept[Page 656, 55].

The Fig. 14.17 as above. The texture object is not bound directly into the shader (where the actual sampling takes place). Instead, it is bound to a 'texture unit' whose index is passed to the shader. So the shader reaches the texture object by going through the texture unit. There are usually multiple texture units available and the exact number depends on the capability of your graphic card[55]. A texture unit, also called a texture mapping unit (TMU) or a texture processing unit (TPU), is a hardware component in a GPU that does sampling operation. The argument sampler in texture function as above is sampler_2d index from 'teuxture unit' for texture object[55].

'sampler uniform variable':

There is a group of special uniform variables for that, according to the texture target: 'sampler1D', 'sampler2D', 'sampler3D', 'samplerCube', etc. You can create as many 'sampler uniform variables' as you want and assign the value of a texture unit to each one from the application. Whenever you call a sampling function on a 'sampler uniform variable' the corresponding texture unit (and texture object) will be used[55].

As Fig. 14.18, the Java api gl.bindTexture binding 'Texture Object' to 'Texture Unit'. The gl.getUniformLocation and gl.uniform1i associate 'Texture Unit' to 'sampler uniform variables'.

gl.uniform1i(xLoc, 1): where 1 is 'Texture Unit 1', 2 is 'Texture Unit 2', ..., etc[Page 656, 56].

The following figure depicts how driver read metadata from compiled glsl obj, OpenGL api associate 'Sample Variable' and gpu executing texture instruction.

Explaining the detail steps for figure above as the following.

[54] http://jonathan2251.github.io/lbd/funccall.html#add-specific-backend-intrinsic-function
[55] http://ogldev.atspace.co.uk/www/tutorial16/tutorial16.html
[56] http://math.hws.edu/graphicsbook/c6/s4.html

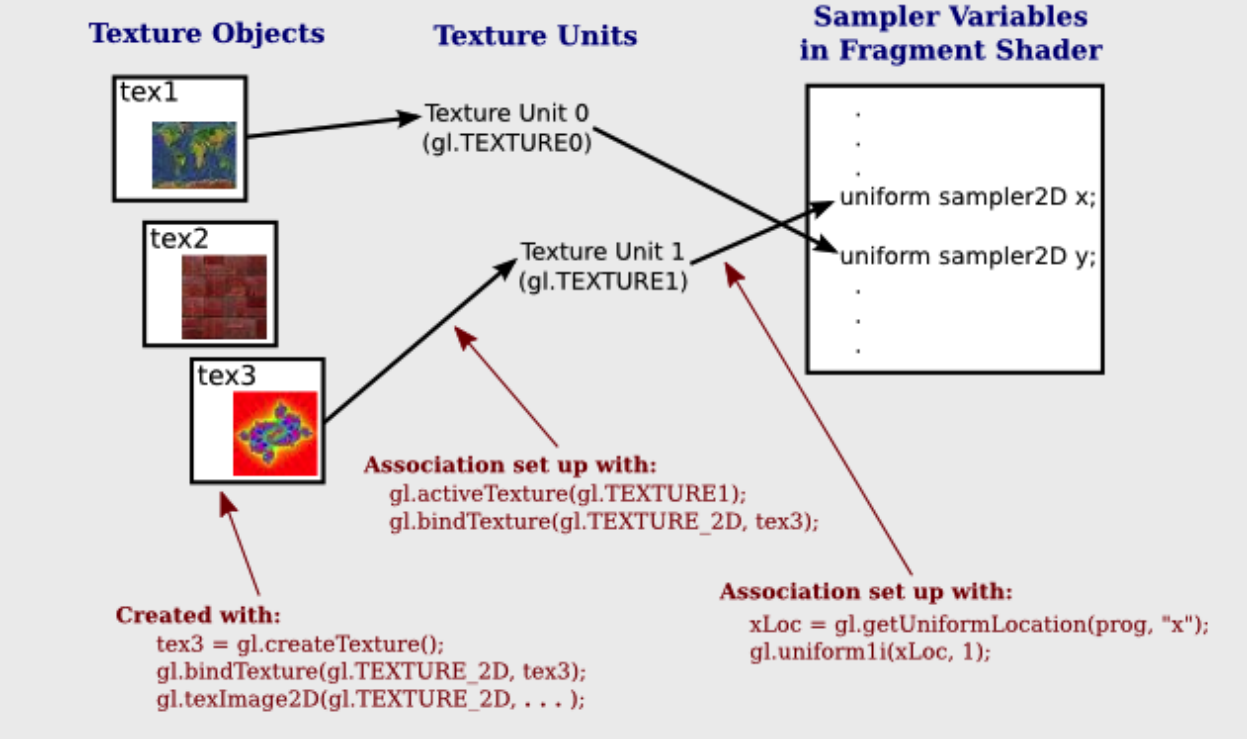


Fig. 14.18: Binding sampler variables[56].

1. In order to let the 'texture unit' binding by driver, frontend compiler must pass the metadata of 'sampler uniform variable' (sampler_2d_var in this example)[59] to backend, and backend must allocate the metadata of 'sampler uniform variable' in the compiled binary file[57].

2. After gpu driver executing glsl on-line compiling, driver read this metadata from compiled binary file and maintain a table of {name, type, location} for each 'sampler uniform variable'. Driver also fill this information to Texture Desciptor in GPU's memory.

3. Api,

```
xLoc = gl.getUniformLocation(prog, "x"); // prog: glsl program, xLoc
```

will get the location from the table for 'sampler uniform variable' x that driver created.

SAMPLER_2D: is integer value for Sampler2D type.

4. Api,

```
gl.uniform1i( xLoc, 1 );
```

will binding xLoc of 'sampler uniform variable' x to 'Texture Unit 1' by writing 1 to the glsl binary metadata location of 'sampler uniform variable' x as follows,

```
{xLoc, 1} : 1 is 'Texture Unit 1', xLoc is the location(memory address) of 'sampler␣
↪uniform variable' x
```

This api will set the texture descriptors in gpu with this {xLoc, 1} information. Next, driver set the index or memory address of gpu texture descriptors to variable .tex_a of memory address. For example as diagram, driver set k to .tex_a.

[59] The type of 'sampler uniform variable' called "sampler variables". http://math.hws.edu/graphicsbook/c6/s4.html
[57] This can be done by llvm metadata. http://llvm.org/docs/LangRef.html#namedmetadatastructure http://llvm.org/docs/LangRef.html#metadata

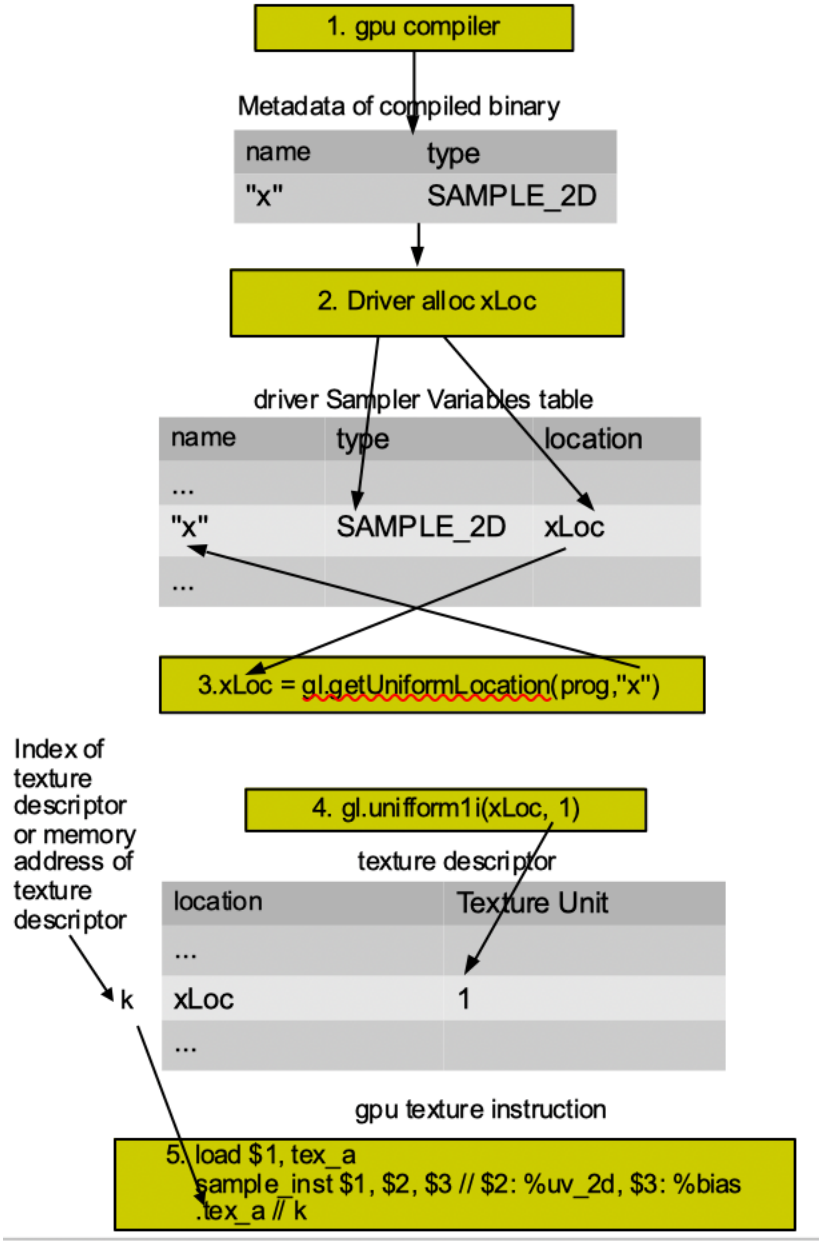


Fig. 14.19: Associating Sampler Variables and gpu executing texture instruction

5.

```
// gpu machine code
load $1, tex_a;
sample2d_inst $1, $2, $3 // $1: tex_a, $2: %uv_2d, $3: %bias

.tex_a // Driver set the index of gpu descriptor regsters here at step 4
```

When executing the texture instructions from glsl binary file on gpu, the corresponding 'Texture Unit 1' on gpu will being executed through texture descriptor in gpu's memory because .tex_a: {xLoc, 1}. Driver may set texture descriptor in gpu's texture desciptors if gpu provides specific texture descriptors in architecture[60].

For instance, Nvidia texture instruction as follow,

```
// the content of tex_a bound to texture unit as step 5 above
tex.3d.v4.s32.s32  {r1,r2,r3,r4}, [tex_a, {f1,f2,f3,f4}];

.tex_a
```

The content of tex_a bound to texture unit set by driver as the end of step 4. The pixel of coordinates (x,y,z) is given by (f1,f2,f3) user input. The f4 is skipped for 3D texture.

Above tex.3d texture instruction load the calculated color of pixel (x,y,z) from texture image into GPRs (r1,r2,r3,r4)=(R,G,B,A). And fragment shader can re-calculate the color of this pixel with the color of this pixel at texture image[58].

If it is 1d texture instruction, the tex.1d as follows,

```
tex.1d.v4.s32.f32  {r1,r2,r3,r4}, [tex_a, {f1}];
```

Since 'Texture Unit' is limited hardware accelerator on gpu, OpenGL providing api to user program for binding 'Texture Unit' to 'Sampler Variables'. As a result, user program is allowed doing load balance in using 'Texture Unit' through OpenGL api without recompiling glsl. Fast texture sampling is one of the key requirements for good GPU performance[56].

In addition to api for binding texture, OpenGL provides glTexParameteri api to do Texture Wrapping[61]. Furthmore the texture instruction for some gpu may including S# T# values in operands. Same with associating 'Sampler Variables' to 'Texture Unit', S# and T# are location of memory associated to Texture Wrapping descriptor registers allowing user program to change Wrapping option without re-compiling glsl.

Even glsl frontend compiler always expanding function call into inline function as well as llvm intrinsic extended function providing an easy way to do code generation through llvm td (Target Description) file written, GPU backend compiler is still a little complex than CPU backend. (But when considering the effort in frontend compier such as clang, or other toolchain such as linker and gdb/lldb, of course, CPU compiler is not easier than GPU compiler.)

Here is the software stack of 3D graphic system for OpenGL in linux[62]. And mesa open source website is here[63].

[60] When performing a texture fetch, the addresses to read pixel data from are computed by reading the GPRs that hold the texture descriptor and the GPRs that hold the texture coordinates. It's mostly just general purpose memory fetching. https://www.gamedev.net/forums/topic/681503-texture-units/

[58] page 84: tex instruction, p24: texture memory https://www.nvidia.com/content/CUDA-ptx_isa_1.4.pdf

[61] https://learnopengl.com/Getting-started/Textures

[62] https://en.wikipedia.org/wiki/Mesa_(computer_graphics)

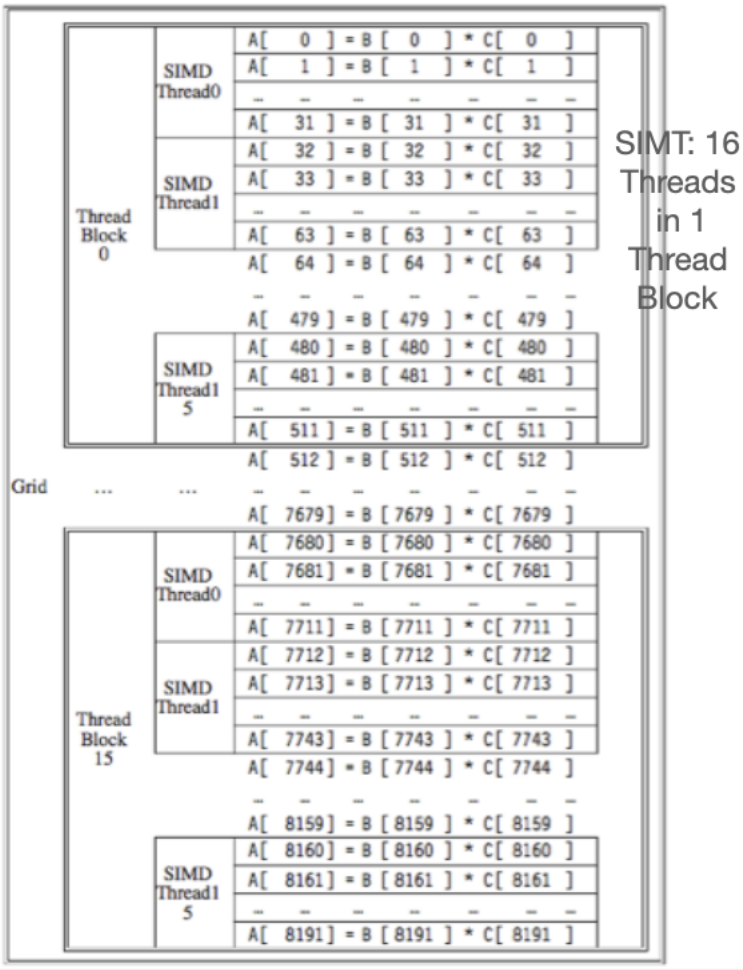
[63] https://www.mesa3d.org/

14.3 Architecture

14.3.1 SIMT

Single instruction, multiple threads (SIMT) is an execution model used in parallel computing where single instruction, multiple data (SIMD) is combined with multithreading[64].

The leading GPU architecture of Nvidia's gpu is as the following figures.



igure 4.13 The mapping of a Grid (vectorizable loop), Thread Blocks (SIMD basic blocks), and threads of SIM nstructions to a vector–vector multiply, with each vector being 8192 elements long. Each thread of SIMD instru ions calculates 32 elements per instruction, and in this example each Thread Block contains 16 threads of SIM nstructions and the Grid contains 16 Thread Blocks. The hardware Thread Block Scheduler assigns Thread Blocks nultithreaded SIMD Processors and the hardware Thread Scheduler picks which thread of SIMD instructions to ru ach clock cycle within a SIMD Processor. Only SIMD Threads in the same Thread Block can communicate via Loc Memory. (The maximum number of SIMD Threads that can execute simultaneously per Thread Block is 16 for Tesl eneration GPUs and 32 for the later Fermi-generation GPUs.)

Fig. 14.20: core(grid) in Nvidia gpu (figure from book[Page 660, 65])

[64] https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads

[65] Book Figure 4.13 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

[66] Book Figure 4.15 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture

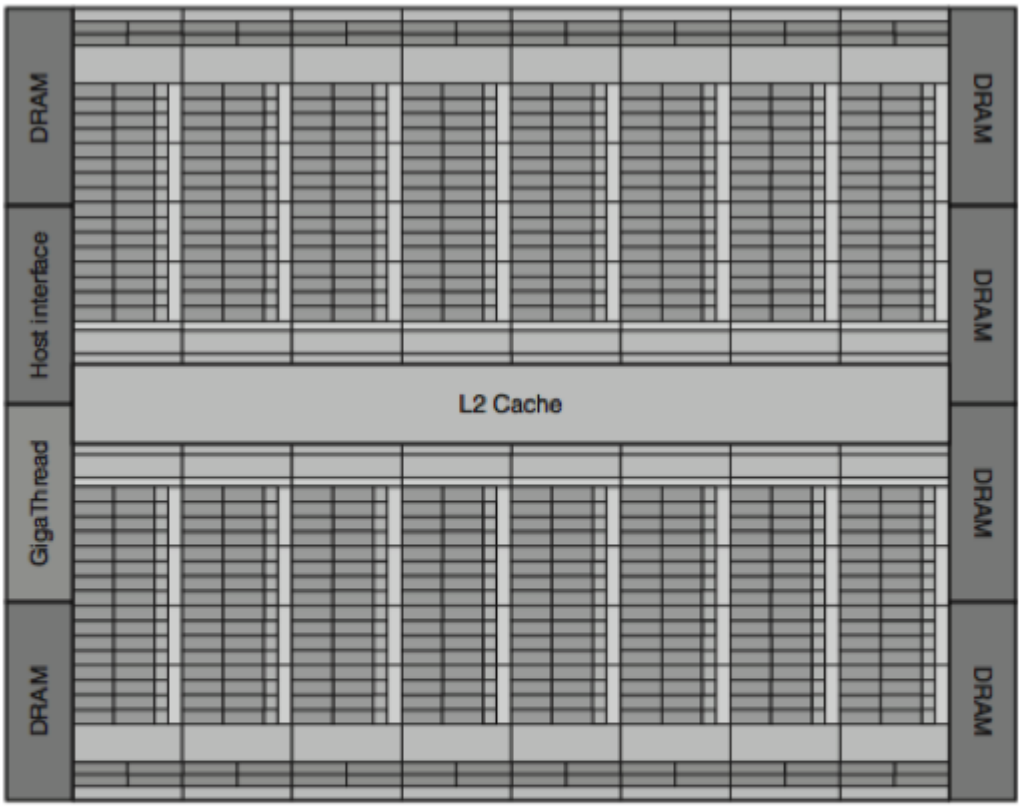


Figure 4.15 Floor plan of the Fermi GTX 480 GPU. This diagram shows 16 multithreaded SIMD Processors. The Thread Block Scheduler is highlighted on the left. The GTX 480 has 6 GDDR5 ports, each 64 bits wide, supporting up to 6 GB of capacity. The Host Interface is PCI Express 2.0 x 16. Giga Thread is the name of the scheduler that distributes thread blocks to Multiprocessors, each of which has its own SIMD Thread Scheduler.

Fig. 14.21: SIMD processors (figure from book[66])

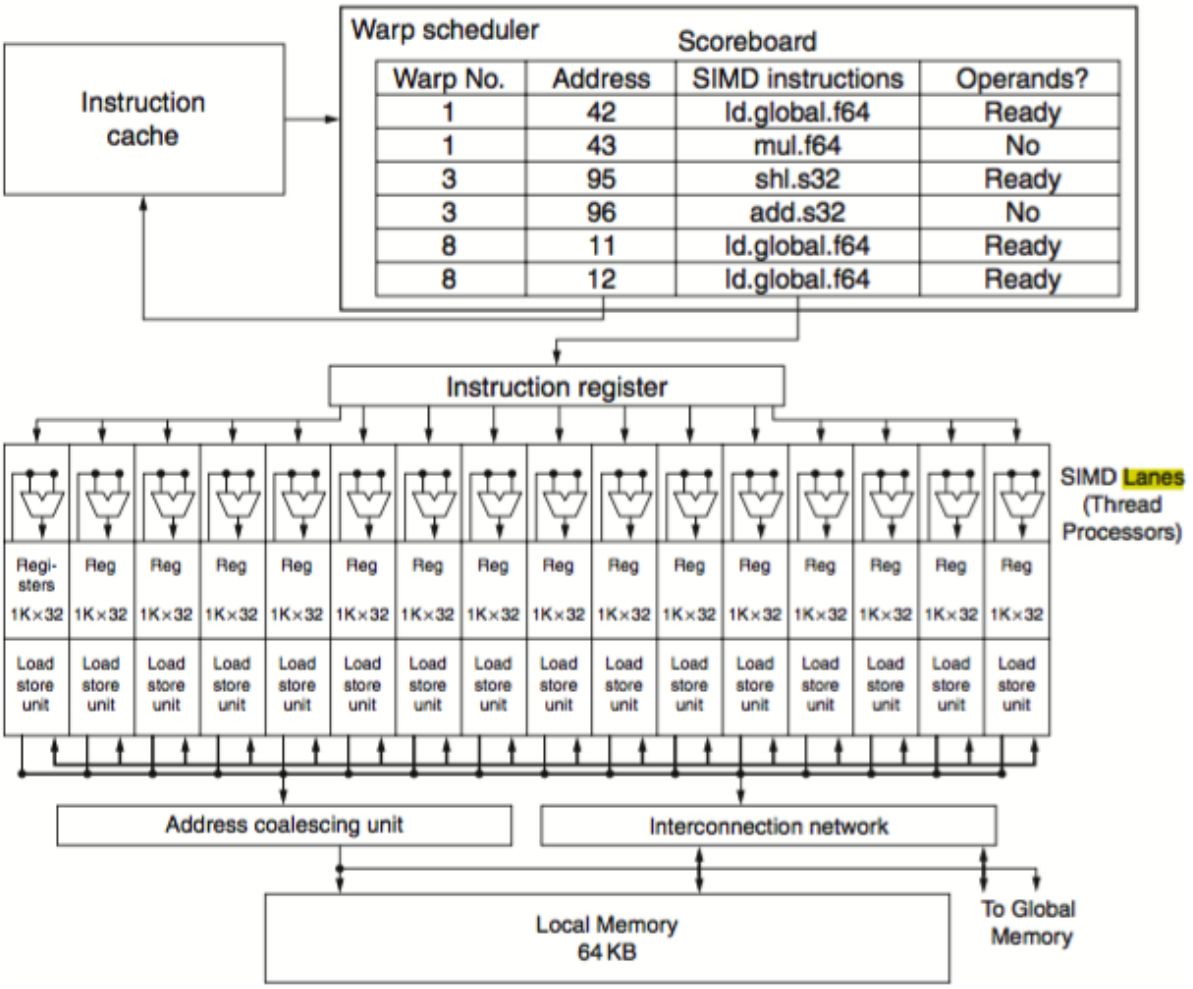


Figure 4.14 Simplified block diagram of a Multithreaded SIMD Processor. It has 16 SIMD lanes. The SIMD Thread Scheduler has, say, 48 independent threads of SIMD instructions that it schedules with a table of 48 PCs.

Fig. 14.22: threads and lanes in gpu (figure from book[Page 663, 67])

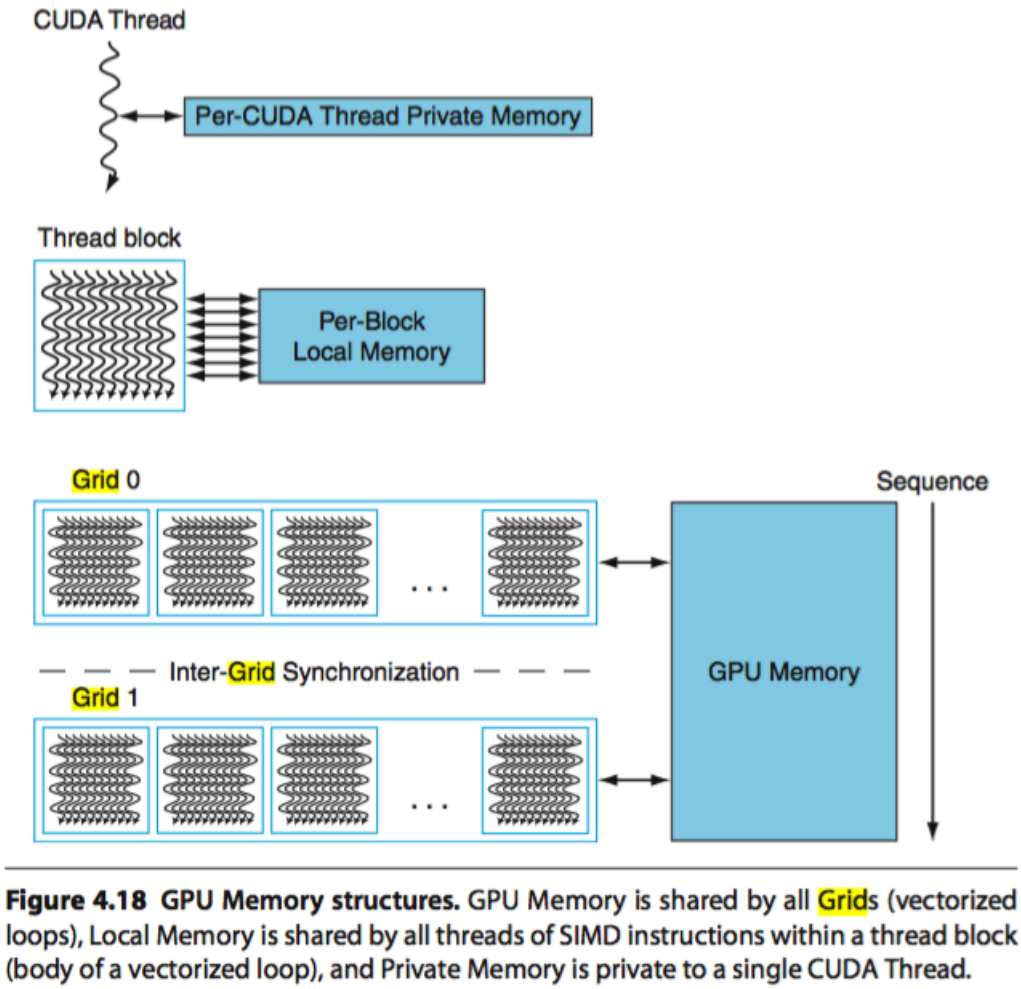


Figure 4.18 GPU Memory structures. GPU Memory is shared by all Grids (vectorized loops), Local Memory is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop), and Private Memory is private to a single CUDA Thread.

Fig. 14.23: core(grid) in Nvidia's gpu (figure from book[Page 663, 68])

and Design)

[67] The SIMD Thread Scheduler includes a scoreboard that lets it know which threads of SIMD instructions are ready to run, and then it sends them off to a dispatch unit to be run on the multithreaded SIMD Processor. It is identical to a hardware thread scheduler in a traditional multithreaded processor (see Chapter 3), just that it is scheduling threads of SIMD instructions. Thus, GPU hardware has two levels of hardware schedulers: (1) the Thread Block Scheduler that assigns Thread Blocks (bodies of vectorized loops) to multi- threaded SIMD Processors, which ensures that thread blocks are assigned to the processors whose local memories have the corresponding data, and (2) the SIMD Thread Scheduler within a SIMD Processor, which schedules when threads of SIMD instructions should run. Book Figure 4.14 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

[68] Book Figure 4.17 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

14.3.2 Buffers

In addition the texture unit and instruction, GPU provides different Buffers to speedup OpenGL pipeline rendering[37].

- Color buffer

  They contain the RGB or sRGB color data and may also contain alpha values for each pixel in the framebuffer. There may be multiple color buffers in a framebuffer. You've already used double buffering for animation. Double buffering is done by making the main color buffer have two parts: a front buffer that's displayed in your window; and a back buffer, which is where you render the new image[69].

- Depth buffer (Z buffer)

  Depth is measured in terms of distance to the eye, so pixels with larger depth-buffer values are overwritten by pixels with smaller values[70][72][73].

- Stencil Buffer

  In the simplest case, the stencil buffer is used to limit the area of rendering (stenciling)[74][73].

- Frame Buffer

  OpenGL offers: the color, depth and stencil buffers. This combination of buffers is known as the default framebuffer and as you've seen, a framebuffer is an area in memory that can be rendered to[75].

14.4 General purpose GPU

Since GLSL shaders provide a general way for writing C code in them, if applying a software frame work instead of OpenGL api, then the system can run some data parallel computation on GPU for speeding up and even get CPU and GPU executing simultaneously. Furthmore, any language that allows the code running on the CPU to poll a GPU shader for return values, can create a GPGPU framework[76].

[37] Page 155 - 185 of book "OpenGL Programming Guide 9th Edition"Page 644, 32.
[69] Page 155 of book "OpenGL Programming Guide 9th Edition"Page 644, 32.
[70] Page 156 of book "OpenGL Programming Guide 9th Edition"Page 644, 32.
[72] https://en.wikipedia.org/wiki/Z-buffering
[73] https://open.gl/depthstencils
[74] https://en.wikipedia.org/wiki/Stencil_buffer
[75] https://open.gl/framebuffers
[76] https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units
[77] Book Figure 4.12 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

Figure 4.12 Quick guide to GPU terms used in this chapter. We use the first column for hardware terms. Four groups cluster these 11 terms. From top to bottom: Program Abstractions, Machine Objects, Processing Hardware, and Memory Hardware. Figure 4.21 on page 309 associates vector terms with the closest terms here, and Figure 4.24 on page 313 and Figure 4.25 on page 314 reveal the official CUDA/NVIDIA and AMD terms and definitions along with the terms used by OpenCL.

Fig. 14.24: Terms in Nvidia's gpu (figure from book[77])

Table 14.3: More Desciptive Name for Cuda term in Fermi GPU and Desciption.

More Desciptive Name	Cuda term	Structure	Description
Grid	Grid		Grid is Vectorizable Loop as Fig. 14.24.
SIMD Processor / SIMD Block / Core	Cuda Thread Engine	Each Grid has 16 SIMD Processors.	Each multithreaded SIMD Processor is assigned 512 elements of the vectors to work on. SIMD Processors are full processors with separate PCs and are programmed using threads[79]. As Fig. 14.21, it assigns 16 Thread Blocks to 16 SIMD Processors. CPU Core is the processor which include multi-threads. A thread of CPU is execution unit with its own PC (Program Counter). As this concept, GPU Core is the SIMD Processor includes several SIMD Thread (Warp). Each Warp has its PC[80].
SIMD Thread	Warp	Each Warp has 16 Cuda Thread.	Warp has it's own PC and TLR (Thread Level Registers). Warp may map to one whole function or part of function. Compiler and run time may assign them to the same Warp or different Warps[81].
SIMD Lane	Cuda Thread	Each SIMD Thread has 16 Lanes.	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. It is a vector instruction with processing 16-elements.
Chime	Chime	Each SIMD Lane has 2 chimes.	If vector length would be 32 (32 elements) and SIMD Lanes is 16, the chime is 2 clock cycles, also known as "ping pong" cycles. As Fig. 14.20 for the later Fermi-generation GPUs. Each SIMD Thread (Warp) has 32 elements run as Fig. 14.22 on 16 SIMD lanes (number of functional units just same as in vector processor). So it takes 2 clock cycles to (chime is 2 clock cycles) complete[82].

14.4.1 Mapping data in GPU

A GPU may has the HW structure and handle the subset of y[]=a*x[]+y[] array-calculation as follows,

```
// Invoke DAXPY with 256 threads per Thread Block
__host__
int nblocks = (n+ 255) / 256;
daxpy<<<nblocks, 256>>>(n, 2.0, x, y);
// DAXPY in CUDA
__device__
void daxpy(int n, double a, double *x, double *y) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
```

(continues on next page)

[79] search these words from section 4.4 of A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

[80] https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)

[81] Book Figure 4.14 and 4.24 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

[82] "With Fermi, each 32-wide thread of SIMD instructions is mapped to 16 physical SIMD Lanes, so each SIMD instruction in a thread of SIMD instructions takes two clock cycles to complete" search these words from Page 296 of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design).

(continued from previous page)

```
  if (i < n) y[i] = a*x[i] + y[i];
}
```

```
shl.u32 R8, blockIdx, 9   ; Thread Block ID * Block size (512 or 29)
add.u32 R8, R8, threadIdx ; R8 = i = my CUDA Thread ID
shl.u32 R8, R8, 3         ; byte offset
ld.global.f64 RD0, [X+R8] ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8] ; RD2 = Y[i]
mul.f64 RD0, RD0, RD4     ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 RD0, RD0, RD2     ; SuminRD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0 ; Y[i] = sum (X[i]*a + Y[i])
```

The following table explains how the elemements of saxpy() maps to lane of SIMD Thread(Warp) of Thread Block(Core) of Grid.

Table 14.4: Mapping saxpy code to Fig. 14.20.

saxpy(()	Instance in Fig. 14.20	Description
blockDim.x	The index of Thread Block	blockDim: in this example configured as Fig. 14.20 is 16(Thread Blocks) * 16(SIDM Threads) = 256
blockIdx.x	The index of SIMD Thread	blockIdx: the index of Thread Block within the Grid
threadIdx.x	The index of elements	threadIdx: the index of the SIMD Thread within its Thread Block

- With Fermi, each 32-wide thread of SIMD instructions is mapped to 16 physical SIMD Lanes, so each SIMD instruction in a thread of SIMD instructions takes two clock cycles to complete.
- You could say that it has 16 lanes, the vector length would be 32, and the chime is 2 clock cycles.
- The mape of y[0..31] = a * x[0..31] * y[0..31] to <Core, Warp, Cuda Thread> of GPU as the following table. x[0..31] map to 32 Cuda Threads; two Cuda Thread map to one SIMD lane.

Table 14.5: Map (Core,Warp) to saxpy

•	Warp-0	Warp-1	...	Warp-15
Core-0	y[0..31] = a * x[0..31] * y[0..31]	y[32..63] = a * x[32..63] + y[32..63]	...	y[480..511] = a * x[480..511] + y[480..511]
...	...	...	...	...
Core-15	y[7680..7711] = a * ...	...	...	y[8160..8191] = a * x[8160..8191] + y[8160..8191]

- If a SIMD Lane (Cuda Thread) handles 2 elements computing, assuming 4 registers for 1 element, then there are 4*32=128 Thread Level Registers, TLR, occupied in a SIMD Thread (Warp) to support the SIMT computing. So, assume a GPU architecture allocating 256 TLR to a SIMD Thread (Warp), then it has sufficient TLR for more complicated statement, such as a*X[i]+b*Y[i]+c*Z[i] without spilling in register allocation. All 16 lanes share the 256 TLR. Each Thread Block (Core/Warp) has 16 SIMD Threads, so there are 16*256 = 4K TLR in a SIMD Processor (Core, Cuda Thread Engine).
- When mapping to the fragments/pixels in graphic GPU, x[0..15] corresponding to a two dimensions of tile of fragments/pixels at pixel[0..3][0..3] since image uses tile base for grouping closest color together.

14.4.2 Work between CPU and GPU in Cuda

Above daxpy() GPU code did not mention the host (CPU) side of code for triggering GPU's function. The following is host (CPU) side of a CUDA example to call saxpy on GPU[78] as follows,

```
for(i=0;i<64; i=i+1)
  if (X[i] != 0)
    X[i] = X[i] - Y[i];
```

```
__global__
void saxpy(int n, float a, float * x, float * y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  ...
  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
  ...
  saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);
  ...
  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
  ...
}
```

The main() run on CPU while the saxpy() run on GPU. CPU copy the data from x and y to the corresponding device arrays d_x and d_y using cudaMemcpy. The saxpy kernel is launched by the statement: saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y); Through cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost, CPU can pass data in x and y arrays to GPU and get result from GPU to y array. Since both of these memory transfers trigger the DMA functions without CPU operation, it may speed up by running both CPU/GPU with their data in their own cache repectively. After DMA memcpy from cpu's memory to gpu's, gpu operates the whole loop of matrix operation for "y[] = a*x[]+y[];" instructions with one Grid. Furthermore like vector processor, gpu provides Vector Mask Registers to Handling IF Statements in Vector Loops as the following code[84],

```
LV V1,Rx          ;load vector X into V1
LV V2,Ry          ;load vector Y
L.D F0,#0         ;load FP zero into F0
SNEVS.D V1,F0     ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D V1,V1,V2  ;subtract under vector mask
SV V1,Rx          ;store the result in X
```

GPU persues throughput from SIMD application. Can hide cache-miss latence from SMT. As result GPU may hasn't L2 and L3 like CPU for each core since GPU is highly latency-tolerant multithreading for data parallel application[85]. DMA memcpy map the data in cpu memory to each l1 cache of core on gpu memory. Many gpu provides operations

[78] https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/

[84] subsection Vector Mask Registers: Handling IF Statements in Vector Loops of Computer Architecture: A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

[85] From section 2.3.2 of book "Heterogeneous Computing with OpenCL 2.0" 3rd edition. https://dahlan.unimal.ac.id/files/ebooks2/2015%203rd%20Heterogeneous%20Computing%20with%20OpenCL%202.0.pdf as follows, "These tasks and the pixels they process are highly parallel, which gives a substan- tial amount of independent work to process for devices with multiple cores and highly latency-tolerant multithreading."

scatter and gather to access DRAM data for stream processing[86Page 664, 7687].

When the GPU function is dense computation in array such as MPEG4 encoder or deep learning for tuning weights, it may get much speed up[88]. However when GPU function is matrix addition and CPU will idle for waiting GPU's result. It may slow down than doing matrix addition by CPU only. Arithmetic intensity is defined as the number of operations performed per word of memory transferred. It is important for GPGPU applications to have high arithmetic intensity else the memory access latency will limit computational speedup[Page 664, 76].

Wiki here[89] includes speedup applications for gpu as follows:

General Purpose Computing on GPU, has found its way into fields as diverse as machine learning, oil exploration, scientific image processing, linear algebra, statistics, 3D reconstruction and even stock options pricing determination. In addition, section "GPU accelerated video decoding and encoding" for video compressing[89] gives the more applications for GPU acceleration.

Table 14.6: The differences for speedup in architecture of CPU and GPU

Item	CPU	GPU
Application	Non-data parallel	Data parallel
Architecture	SISD, small vector (eg.4*32bits)	Large SIMD (eg.16*32bits)
Cache	Smaller and faster	Larger and slower (ref. The following Note)
ILP	Pipeline	Pipeline
•	Superscalar, SMT	SIMT
•	Super-pipeline	
Core	Smaller threads for SMT (2 or 4)	Larger threads (16 or 32)
Branch	Conditional-instructions	Mask & conditional-instructions

Note: GPU-Cache

In theory for data parallel application in GPU's SMT, GPU can schedule more threads and pursues throughput rather speedup for one single thread as SISD in CPU. However in reality, GPU provides small L1 cache like CPU's and fill the cache-miss with scheduline another thread. So, GPU may has no L2 and L3 while CPU has deep level of caches.

14.4.3 Volta (Cuda thread/SIMD lane with PC, Program Couner and Call Stack)

Todo: According the assembly of dapxy(), this is not useful. Why is this existed?

One way the compiler handles this is by keeping executing instructions in order and if some threads don't have to execute certain instructions it switches off those threads and turns them on their relevant instructions and switches off the other threads, this process is called masking.

- After Volta GPU of Nvidia, each thread in Warp has it's own PC as Fig. 14.26. In Cuda Applications, this feature provides more parallel opportunities with __syncwarp() to user programmers as Fig. 14.27.

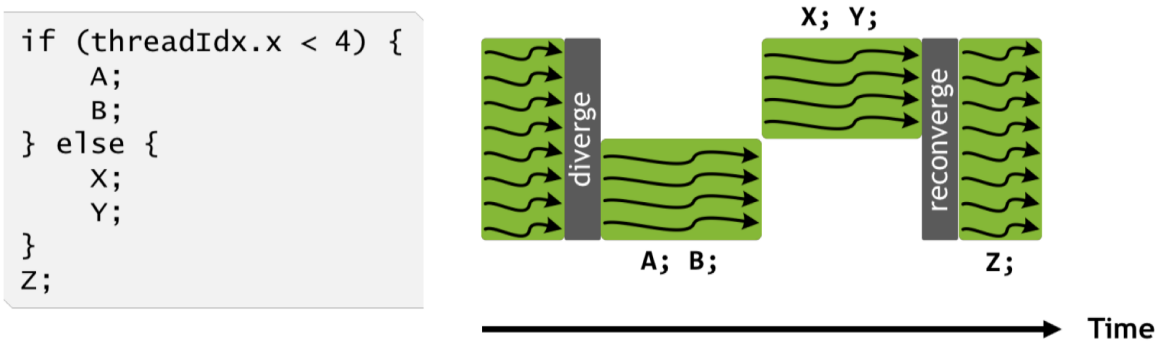
[86] Reference "Gather-Scatter: Handling Sparse Matrices in Vector Architectures": section 4.2 Vector Architecture of A Quantitative Approach 5th edition (The Morgan Kaufmann Series in Computer Architecture and Design)

[87] The whole chip shares a single L2 cache, but the different units will have individual L1 caches. https://computergraphics.stackexchange.com/questions/355/how-does-texture-cache-work-considering-multiple-shader-units

[88] https://www.manchestervideo.com/2016/06/11/speed-h-264-encoding-budget-gpu/

[89] https://en.wikipedia.org/wiki/Graphics_processing_unit

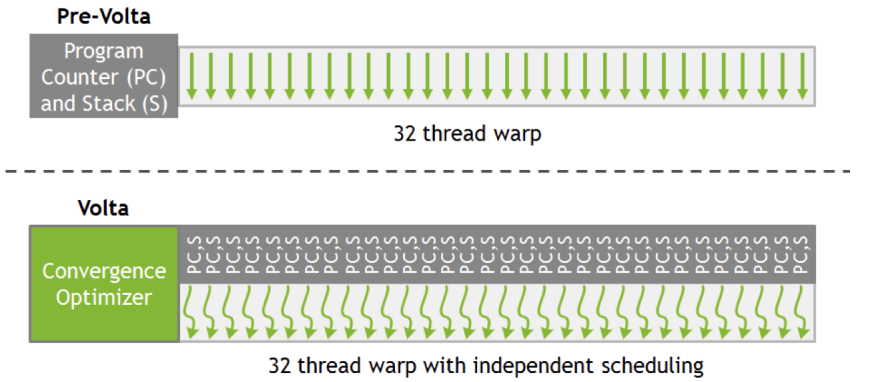
[83] See the same Figures from https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf



Thread scheduling under the SIMT warp execution model of Pascal and earlier NVIDIA GPUs. Capital letters represent statements in the program pseudocode. Divergent branches within a warp are serialized so that all statements in one side of the branch are executed together to completion before any statements in the other side are executed. After the else statement, the threads of the warp will typically reconverge.

Figure 20. SIMT Warp Execution Model of Pascal and Earlier GPUs

Fig. 14.25: SIMT Warp Execution Model of Pascal and Earlier GPUs[83]



Volta (bottom) independent thread scheduling architecture block diagram compared to Pascal and earlier architectures (top). Volta maintains per-thread scheduling resources such as program counter (PC) and call stack (S), while earlier architectures maintained these resources per warp.

Figure 21. Volta Warp with Per-Thread Program Counter and Call Stack

Fig. 14.26: Volta Warp with Per-Thread Program Counter and Call Stack[Page 669, 83]

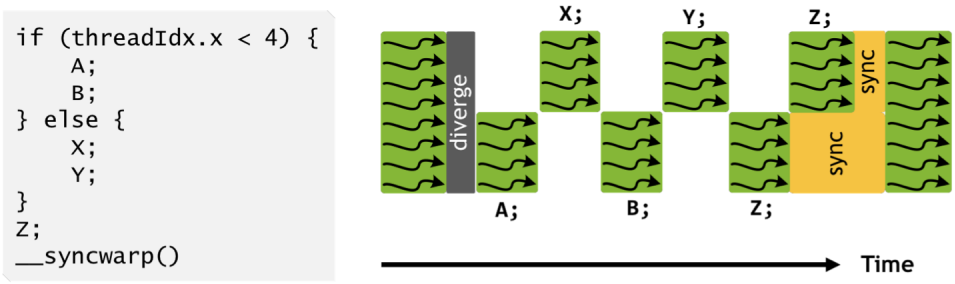


Figure 23. Programs use Explicit Synchronization to Reconverge Threads in a Warp

Fig. 14.27: Programs use Explicit Synchronization to Reconverge Threads in a Warp[Page 669, 83]

14.5 Vulkan and spir-v

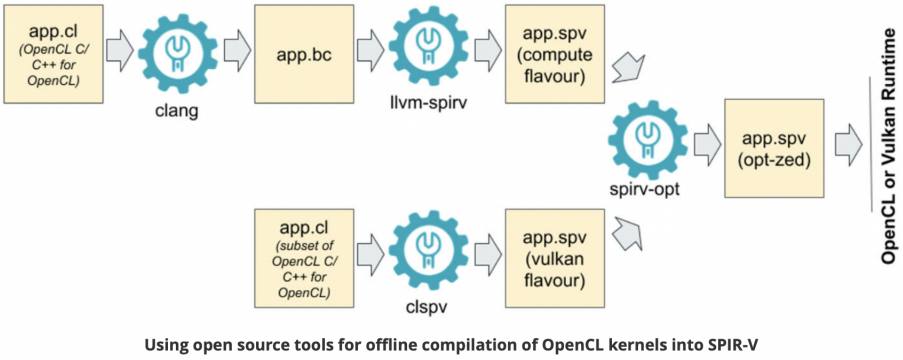


Fig. 14.28: Offline Compilation of OpenCL Kernels into SPIR-V Using Open Source Tooling[91]

- Comparsion for OpenCL and OpenGL's compute shader.
  - Same:

    Both are for General Computing of GPU.
  - Difference:

    OpenCL include GPU and other accelerate device/processor. OpenCL is C language on Device and C++ on Host based on OpenCL runtime. Compute shader is GLSL shader language run on OpenGL graphic enviroment and integrate and access data of OpenGL API easily[90].
- OpenGL/GLSL vs Vulkan/spir-v.
  - High level of API and shader: OpenGL, GLSL.
  - Low level of API and shader: Vulkan, spir-v.

Though OpenGL api existed in higher level with many advantages from sections above, sometimes it cannot compete in efficience with direct3D providing lower levels api for operating memory by user program[92]. Vulkan api is lower level's

[91] https://www.khronos.org/blog/offline-compilation-of-opencl-kernels-into-spir-v-using-open-source-tooling
[90] https://stackoverflow.com/questions/15868498/what-is-the-difference-between-opencl-and-opengls-compute-shader
[92] Vulkan offers lower overhead, more direct control over the GPU, and lower CPU usage… By allowing shader pre-compilation, application

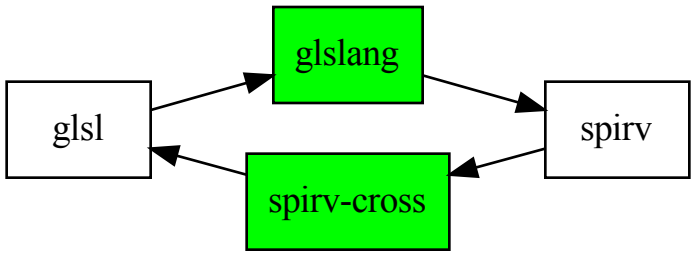


Fig. 14.29: Convertion between glsl and spirv

C/C++ api to fill the gap allowing user program to do these things in OpenGL to compete against Microsoft direct3D. Here is an example[93]. Meanwhile glsl is C-like language. The vulkan infrastructure provides tool, glslangValidator[94], to compile glsl into an Intermediate Representation Form (IR) called spir-v off-line. As a result, it saves part of compilation time from glsl to gpu instructions on-line since spir-v is an IR of level closing to llvm IR[95]. In addition, vulkan api reduces gpu drivers efforts in optimization and code generation[Page 671, 92]. These standards provide user programmer option to using vulkan/spir-v instead of OpenGL/glsl, and allow them pre-compiling glsl into spir-v off-line to saving part of on-line compilation time.

With vulkan and spir-v standard, the gpu can be used in OpenCL for Parallel Programming of Heterogeneous Systems[96][97]. Similar with Cuda, a OpenCL example for fast Fourier transform (FFT) is here[98]. Once OpenCL grows into a popular standard when more computer languages or framework supporting OpenCL language, GPU will take more jobs from CPU[99].

Now, you find llvm IR expanding from cpu to gpu becoming influentially more and more. And actually, llvm IR expanding from version 3.1 util now as I can feel.

initialization speed is improved... A Vulkan driver only needs to do GPU specific optimization and code generation, resulting in easier driver maintenance...[Page 636, 8] https://en.wikipedia.org/wiki/Vulkan#OpenGL_vs._Vulkan

[93] https://github.com/SaschaWillems/Vulkan/blob/master/examples/triangle/triangle.cpp

[94] glslangValidator is the tool used to compile GLSL shaders into SPIR-V, Vulkan's shader format. https://vulkan.lunarg.com/doc/sdk/latest/windows/spirv_toolchain.html

[95] SPIR 2.0: LLVM IR version 3.4. SPIR-V 1.X: 100% Khronos defined Round-trip lossless conversion to llvm. https://en.wikipedia.org/wiki/Standard_Portable_Intermediate_Representation

[96] https://www.khronos.org/opencl/

[97] https://en.wikipedia.org/wiki/Compute_kernel

[98] https://en.wikipedia.org/wiki/OpenCL

[99] The OpenCL standard defines host APIs for C and C++; third-party APIs exist for other programming languages and platforms such as Python,[15] Java, Perl[15] and .NET.[11]:15 https://en.wikipedia.org/wiki/OpenCL

14.6 Accelerate ML/DL on OpenCL/SYCL

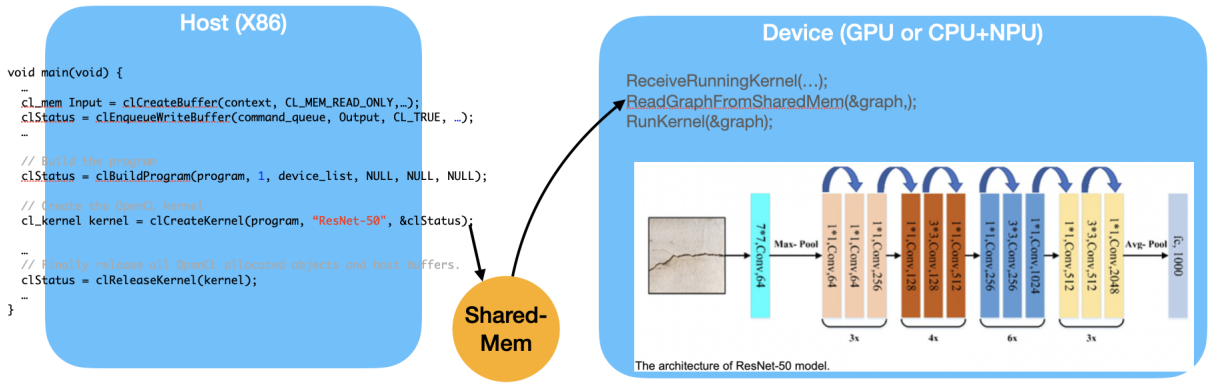


Fig. 14.30: Implement ML graph scheduler both on compiler and runtime

As above figure, the Device of GPU or CPU+NPU is able to run the whole ML graph. However if the Device has NPU only, then the CPU operation such as Avg-Pool has to run on Host side which add communication cost between Host and Device.

Like OpenGL's shader, the "kernel" function may be compiled on-line or off-line and sending to GPU as programmable functions.

In order to run ML (Machine Learning) efficiently, all platforms for ML on GPU/NPU implement scheduling SW both on graph compiler and runtime. **If OpenCL can extend to support ML graph, then graph compiler such as TVM or Runtime from Open Source have chance to leverage the effort of scheduling SW from programmers**[100]. Cuda graph is an idea like this[101] [102] .

14.7 Open Sources

- https://registry.khronos.org/OpenGL-Refpages/
- https://www.mesa3d.org
- https://www.opengl.org/sdk/, https://www.opengl.org/sdk/libs/

[100] https://easychair.org/publications/preprint/GjhX
[101] https://developer.nvidia.com/blog/cuda-graphs/
[102] https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/

CHAPTER

FIFTEEN

APPENDIX A: GETTING STARTED: INSTALLING LLVM AND THE CPU0 EXAMPLE CODE

- *Build steps*
- *Setting Up Your Mac*
  - *Install Icarus Verilog tool on iMac*
  - *Install other tools on iMac*
- *Setting Up Your Linux Machine*
  - *Install Icarus Verilog tool on Linux*
  - *Install other tools on Linux*
- *Toolchain*

Cpu0 example code, lbdex, can be found near left bottom of this web site. Or here http://jonathan2251.github.io/lbd/lbdex.tar.gz.

For information in using `cmake` to build LLVM, please refer to the "Building LLVM with CMake"[1] documentation for further information.

We install two llvm directories in this chapter. One is the directory ~/llvm/debug/ which contains the clang and clang++ compiler we will use to translate the C/C++ input file into llvm IR. The other is the directory ~/llvm/test/ which contains our cpu0 backend program and clang.

15.1 Build steps

On linux, multi-threading -DLLVM_PARALLEL_COMPILE_JOBS=4, need more than 16GB memory. I create 64GB of swapfile to fix the problem of link failure[2][3]. The iMac has no this problem.

```
$ cat /etc/fstab
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
...
/swapfile       swap            swap    default         0       0
```

After setup brew install for iMac or install necessory packages. Build as https://github.com/Jonathan2251/lbd/blob/master/README.md.

[1] http://llvm.org/docs/CMake.html?highlight=cmake
[2] https://bogdancornianu.com/change-swap-size-in-ubuntu/
[3] https://linuxize.com/post/how-to-add-swap-space-on-ubuntu-18-04/

15.2 Setting Up Your Mac

Brew install and setup first[4].

```
% /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
↪install.sh)"
```

Then add brew command to your path as the bottom of installed message of bash above, like the following.

```
% echo 'eval "$(/opt/homebrew/bin/brew shellenv)"' >> /Users/cschen/.zprofile
% eval "$(/opt/homebrew/bin/brew shellenv)"
```

Brew install in China, use the following install.sh instead[5].

```
% /bin/zsh -c "$(curl -fsSL https://gitee.com/cunkai/HomebrewCN/raw/master/Homebrew.sh)"
...
% source /Users/cschen/.zprofile
% brew --version
Homebrew 3.6.7-28-g560f571
fatal: detected dubious ownership in repository at '/usr/local/Homebrew/Library/Taps/
↪homebrew/homebrew-core'
To add an exception for this directory, call:

        git config --global --add safe.directory /usr/local/Homebrew/Library/Taps/
↪homebrew/homebrew-core
Homebrew/homebrew-core (no Git repository)
fatal: detected dubious ownership in repository at '/usr/local/Homebrew/Library/Taps/
↪homebrew/homebrew-cask'
To add an exception for this directory, call:

        git config --global --add safe.directory /usr/local/Homebrew/Library/Taps/
↪homebrew/homebrew-cask

% git config --global --add safe.directory /usr/local/Homebrew/Library/Taps/homebrew/
↪homebrew-core
% git config --global --add safe.directory /usr/local/Homebrew/Library/Taps/homebrew/
↪homebrew-cask
% brew install cmake
...
==> Running `brew cleanup cmake`...
Disable this behaviour by setting HOMEBREW_NO_INSTALL_CLEANUP.
Hide these hints with HOMEBREW_NO_ENV_HINTS (see `man brew`).

% brew install ninja
```

[4] https://brew.sh/
[5] https://blog.csdn.net/weixin_45571585/article/details/126977413

15.2.1 Install Icarus Verilog tool on iMac

Install Icarus Verilog tool by command `brew install icarus-verilog` as follows,

```
% brew install icarus-verilog
==> Downloading ftp://icarus.com/pub/eda/verilog/v0.9/verilog-0.9.5.tar.gz
######################################################################## 100.0%
######################################################################## 100.0%
==> ./configure --prefix=/usr/local/Cellar/icarus-verilog/0.9.5
==> make
==> make installdirs
==> make install
/usr/local/Cellar/icarus-verilog/0.9.5: 39 files, 12M, built in 55 seconds
```

15.2.2 Install other tools on iMac

Install cmake and ninja as follows,

```
$ brew install cmake
$ brew install ninja
```

Install Graphviz for display llvm IR nodes in debugging,[7].

```
% brew install graphviz
```

The Graphviz information for llvm is at section "SelectionDAG Instruction Selection Process" " of "The LLVM Target-Independent Code Generator" here[8] and at section "Viewing graphs while debugging code" of "LLVM Programmer's Manual" here[9].

Install binutils by command `brew install binutils` as follows,

```
// get brew by the following ruby command if you don't have installed brew
118-165-77-214:~ Jonathan$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
↪Homebrew/install/master/install)" < /dev/null 2> /dev/null
118-165-77-214:~ Jonathan$ brew install binutils
==> Downloading http://ftpmirror.gnu.org/binutils/binutils-2.22.tar.gz
######################################################################## 100.0%
==> ./configure --program-prefix=g --prefix=/usr/local/Cellar/binutils/2.22
--infodir=/usr/loca
==> make
==> make install
/usr/local/Cellar/binutils/2.22: 90 files, 19M, built in 4.7 minutes
118-165-77-214:~ Jonathan$ ls /usr/local/Cellar/binutils/2.22
COPYING     README      lib
ChangeLog     bin       share
INSTALL_RECEIPT.json    include       x86_64-apple-darwin12.2.0
118-165-77-214:binutils-2.23 Jonathan$ ls /usr/local/Cellar/binutils/2.22/bin
gaddr2line  gc++filt  gnm   gobjdump  greadelf  gstrings
gar   gelfedit  gobjcopy  granlib gsize   gstrip
```

[7] https://graphviz.org/download/
[8] http://llvm.org/docs/CodeGenerator.html#selectiondag-instruction-selection-process
[9] http://llvm.org/docs/ProgrammersManual.html#viewing-graphs-while-debugging-code

15.3 Setting Up Your Linux Machine

15.3.1 Install Icarus Verilog tool on Linux

Download icarus verilog as follows[10],

```
$ git clone http://iverilog.icarus.com/
```

Follow the README or INSTALL file guide to install it.

Install *sh autoconf.sh* dependences and other dependences as follows,

```
$ pwd
$ ~/git/iverilog
$ sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev \
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool \
patchutils bc zlib1g-dev libexpat-dev
$ sh autoconf.sh
```

Then install icarus as follows,

```
$ ./configure
// or below if you are in shared server
$ ./configure --prefix=$HOME/local
$ make
$ make check
```

15.3.2 Install other tools on Linux

Install cmake and ninja as follows,

```
$ pwd
$ ~/local
$ wget -b https://github.com/Kitware/CMake/releases/download/v3.23.3/cmake-3.23.3-linux-
↪x86_64.sh
$ bash cmake-3.23.3-linux-x86_64.sh
Do you accept the license? [yn]:
y
By default the CMake will be installed in:
  "/u/jonathanchen/local/cmake-3.23.3-linux-x86_64"
Do you want to include the subdirectory cmake-3.23.3-linux-x86_64?
Saying no will install in: "/u/jonathanchen/local" [Yn]:
Y
...
Unpacking finished successfully
$ ls
bin  cmake-3.23.3-linux-x86_64 ...

$ sudo apt install ninja-build
```

[10] http://iverilog.icarus.com/

Download Graphviz from[11] according your Linux distribution. Files compare tools Kdiff3 came from web site[6].

```
$ sudo apt install graphviz
$ dot -V
dot - graphviz version 2.40.1 (20161225.0304)
```

Set "~/.profile" as follows,

~/.profile

```
 ~/.profile: executed by the command interpreter for login shells.
...
# set PATH so it includes user's private bin if it exists
if [ -d "$HOME/local/bin" ] ; then
    PATH="$HOME/local/bin:$PATH"
fi
# set PATH for cmake
if [ -d "$HOME/local/cmake-3.23.3-linux-x86_64/bin" ] ; then
    PATH="$HOME/local/cmake-3.23.3-linux-x86_64/bin:$PATH"
fi
...
```

15.4 Toolchain

List some gnu and llvm tools as follows,

```
// Linux
~/git/lbd/lbdex/input$ ~/llvm/debug/build/bin/clang -fpic hello.c
~/git/lbd/lbdex/input$ man ldd
ldd - print shared object dependencies
~/git/lbd/lbdex/input$ ldd a.out
      linux-vdso.so.1 (0x00007fffd1fe5000)
      libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2c92a82000)
      /lib64/ld-linux-x86-64.so.2 (0x00007f2c92e73000)

// MacOS
% man otool
otool-classic - object file displaying tool
  ...
  -L     Display the names and version numbers of the shared libraries that the object␣
↪file uses, as well as the shared library ID if the file is a shared library.
% otool -L a.out
a.out:
      /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1292.100.
↪5)

// Linux
~/git/lbd/lbdex/input$ man objcopy
```

(continues on next page)

[11] http://www.graphviz.org/Download.php
[6] http://kdiff3.sourceforge.net

(continued from previous page)

```
objcopy - copy and translate object files
  ...
  [-O bfdname|--output-target=bfdname]
~/git/lbd/lbdex/input$ objcopy -O verilog a.out a.hex
~/git/lbd/lbdex/input$ vi a.hex
@00400238
2F 6C 69 62 36 34 2F 6C 64 2D 6C 69 6E 75 78 2D
78 38 36 2D 36 34 2E 73 6F 2E 32 00
@00400254
04 00 00 00 10 00 00 00 01 00 00 00 47 4E 55 00
...
```

CHAPTER

SIXTEEN

APPENDIX B: CPU0 DOCUMENT AND TEST

- *Cpu0 document*
  - *Install sphinx*
  - *Note of Sphinx*
  - *Note of graphviz*
  - *Generate Cpu0 document*
  - *About Cpu0 document*
- *Cpu0 Regression Test*

16.1 Cpu0 document

This section illustrates how to generate Cpu0 backend document.

16.1.1 Install sphinx

LLVM and this book use Sphinx to generate html document. This book uses Sphinx to generate pdf and epub format of document further. The installation of Sphinx reference[1]. Sphinx uses restructured text format here[8][9][10]. About the code-block in this document, please reference[11][12].

On iMac you can install as follows,

```
brew install sphinx-doc
echo 'export PATH="/opt/homebrew/opt/sphinx-doc/bin:$PATH"' >> ~/.zshrc
% source ~/.zshrc
```

On Linux install it as follows,

```
sudo apt-get install python3-sphinx
```

[1] https://www.sphinx-doc.org/en/master/usage/installation.html
[8] http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html
[9] http://docutils.sourceforge.net/docs/ref/rst/directives.html
[10] http://docutils.sourceforge.net/rst.html
[11] http://llvm.org/docs/SphinxQuickstartTemplate.html If you need to show LLVM IR use the llvm code block. https://llvm.org/docs/SphinxQuickstartTemplate.html#code-blocks
[12] http://pygments.org/docs/lexers/

Above installaton for making html document but not for pdf. The following installation for making pdf/latex document.

On iMac, install MacTex.pkg from here[2] and restart computer.

On Linux, install texlive as follows,

```
sudo apt-get install texlive texlive-latex-extra latexmk
```

or

```
sudo yum install texlive texlive-latex-extra latexmk
```

On Fedora 17, the texlive-latex-extra is missing. We install the package which include the pdflatex instead. For instance, we install pdfjam on Fedora 17 as follows,

```
[root@localhost lbd]$ yum list pdfjam
Loaded plugins: langpacks, presto, refresh-packagekit
Installed Packages
pdfjam.noarch                          2.08-3.fc17                          @fedora
[root@localhost lbd]$
```

Do sudo apt-get update -y; sudo apt-get install -y latexmk; if latexmk error in 'make latexpdf'[3].

After upgrade to iMac OS X 10.11.1, pdflatex link is missing, fix it by set in .profile as follows,

```
114-37-153-62:lbd Jonathan$ ls /usr/local/texlive/2012/bin/universal-darwin/pdflatex
/usr/local/texlive/2012/bin/universal-darwin/pdflatex
114-37-153-62:lbd Jonathan$ cat ~/.profile
export PATH=$PATH:...:/usr/local/texlive/2012/bin/universal-darwin
```

16.1.2 Note of Sphinx

- Do not use > 90 characters in a line of code-block:: console or any code-block, otherwise it may show the error in *make latexpdf*.
- Use "clearpage" (reference gpu.rst) to clear page for pdf to avoid split into two pages.

```
$HOME/git/lbd$ make latexpdf

Latexmk: applying rule 'pdflatex'...
Rule 'pdflatex': File changes, etc:
   Changed files, or newly in use since previous run(s):
  LLVMToolchainCpu0.aux
  LLVMToolchainCpu0.toc
Rule 'pdflatex': The following rules & subrules became out-of-date:
  pdflatex
Latexmk: Maximum runs of pdflatex reached without getting stable files
Latexmk: All targets (LLVMToolchainCpu0.pdf) are up-to-date
----------------------
This message may duplicate earlier message.
Latexmk: Failure in processing file 'LLVMToolchainCpu0.tex':
   'pdflatex' needed too many passes
```

(continues on next page)

[2] http://www.tug.org/mactex/
[3] https://zoomadmin.com/HowToInstall/UbuntuPackage/latexmk

(continued from previous page)

```
----------------------
Latexmk: If appropriate, the -f option can be used to get latexmk
  to try to force complete processing.
make[1]: *** [LLVMToolchainCpu0.pdf] Error 12
make: *** [latexpdf] Error 2
make latexpdf  14.57s user 1.11s system 99% cpu 15.790 total
```

- Sphinx support math symbols here[4][5].

16.1.3 Note of graphviz

- Drawing nodes inside a parent node: "compound=true;". Subgraph cluster_name[7]. Edge from/to parent node: "ltail/lhead = parent_node".
- Documents[6].

16.1.4 Generate Cpu0 document

Cpu0 example code is added chapter by chapter. It can be configured to a specific chapter by change CH definition in Cpu0SetChapter.h. For example, the following definition configue it to chapter 2.

lbdex/Cpu0/Cpu0SetChapter.h

```
#define CH       CH2
```

To make readers easily understanding the backend structure step by step, Cpu0 example code can be generated with chapter by chapter through commands as follws,

```
118-165-12-177:lbd Jonathan$ pwd
/home/Jonathan/test/lbd
118-165-12-177:lbd Jonathan$ make genexample
...
118-165-12-177:lbd Jonathan$ ls lbdex/chapters/
Chapter10_1  Chapter2    Chapter3_4  Chapter5_1  Chapter8_2
Chapter11_1  Chapter3_1  Chapter3_5  Chapter6_1  Chapter9_1
Chapter11_2  Chapter3_2  Chapter4_1  Chapter7_1  Chapter9_2
Chapter12_1  Chapter3_3  Chapter4_2  Chapter8_1  Chapter9_3
```

Beside chapters example code, above html and pdf of Cpu0 documents also include files *.ll and *.s in lbd/lbdex/output.

```
JonathantekiiMac:lbd Jonathan$ ls lbdex/output/
ch12_eh.cpu0.s                      ch12_thread_var.cpu0.pic.s      ch12_thread_var.ll
ch12_eh.ll                  ch12_thread_var.cpu0.static.s   ch4_math.s
```

Then, this book html/pdf can be generated by the following commands.

[4] https://sphinx-rtd-trial.readthedocs.io/en/latest/ext/math.html#module-sphinx.ext.mathbase

[5] https://mirrors.mit.edu/CTAN/info/short-math-guide/short-math-guide.pdf

[7] Ex. lbd/Fig/gpu/opengl-flow.gv. If the name of the subgraph begins with cluster, Graphviz notes the subgraph as a special cluster subgraph. If supported, the layout engine will do the layout so that the nodes belonging to the cluster are drawn together, with the entire drawing of the cluster contained within a bounding rectangle. https://graphviz.org/doc/info/lang.html.

[6] https://www.graphviz.org/pdf/dotguide.pdf

```
118-165-12-177:lbd Jonathan$ pwd
/home/Jonathan/test/lbd
118-165-12-177:lbd Jonathan$ make html
...
118-165-12-177:lbd Jonathan$ make latexpdf
...
```

16.1.5 About Cpu0 document

Since llvm have a new release version about every 6 months and every name of file, function, class, variable, ..., etc, may be changed, the Cpu0 document maintains is an effort because it adds the code chapter by chapter. In order to make the document as correct and easily maintain. I use the ":start-after:" and ":end-before:" of restructured text format to keep the document update to date. For every new release, when the Cpu0 backend code is changed, the document will reflect the changes in most of the contents of document.

In lbdex/Cpu0, the text begin from "//@" and "#ifdef CH > CHxx" are refered by document files *.rst.

In lbdex/llvm/modify/llvm, the *.rst refer the code by copy them directly. Most of references exist in llvmstructure.rst and elf.rst.

The example C/C++ code in lbdex/input come from my thinking and refer the directory clang/test/CodeGen of clang source code release.

16.2 Cpu0 Regression Test

The last chapter can verify Cpu0 backend's generated code by Verilog simulator for those code without global variable access. The chapter lld in web https://github.com/Jonathan2251/lbt.git includes llvm ELF linker implementation and is capable of verifying those test items with global variable access. However, LLVM has its test cases (regression test) for each backend to verify your backend compiler[13] without implementing any simulator or real hardware platform. Cpu0 regression test items existed in lbdex.tar.gz example code. Untar it to lbdex/, and:

For both iMac and Linux, copy lbdex/regression-test/Cpu0 to ~/llvm/test/llvm/test/CodeGen/Cpu0.

Then run as follows for both of single and all test cases on iMac. The **llvm-lit -a** can display executing command.

```
1-160-130-77:Cpu0 Jonathan$ pwd
/Users/Jonathan/llvm/test/llvm/test/CodeGen/Cpu0
1-160-130-77:Cpu0 Jonathan$ ~/llvm/test/build/bin/llvm-lit seteq.ll
-- Testing: 1 tests, 1 threads --
PASS: LLVM :: CodeGen/Cpu0/seteq.ll (1 of 1)
Testing Time: 0.08s
  Expected Passes    : 1
1-160-130-77:Cpu0 Jonathan$ ~/llvm/test/build/bin/llvm-lit .
...
PASS: LLVM :: CodeGen/Cpu0/zeroreg.ll
PASS: LLVM :: CodeGen/Cpu0/tailcall.ll
...
```

Run as follows for test on Linux.

[13] http://llvm.org/docs/TestingGuide.html

```
$ pwd
/home/cschen/llvm/test/llvm/test/CodeGen/Cpu0
$ ~/llvm/test/build/bin/llvm-lit seteq.ll
-- Testing: 1 tests, 1 threads --
PASS: LLVM :: CodeGen/Cpu0/seteq.ll (1 of 1)
Testing Time: 0.08s
  Expected Passes    : 1
$ ~/llvm/test/build/bin/llvm-lit .
...
PASS: LLVM :: CodeGen/Cpu0/zeroreg.ll
PASS: LLVM :: CodeGen/Cpu0/tailcall.ll
...
```

Listing the chapters of this book and the related regression test items as follows,

Table 16.1: Chapters

1	about
2	Cpu0 architecture and LLVM structure
3	Backend structure
4	Arithmetic and logic instructions
5	Generating object files
6	Global variables
7	Other data type
8	Control flow statements
9	Function call
10	ELF Support
11	Assembler
12	C++ support
13	Verify backend on verilog simulator

Table 16.2: Regression test items for Cpu0

File	v:pass x:fail	test ir, -> output asm	chapter
2008-06-05-Carry.ll	v		7
2008-07-15-InternalConstant.ll	v		6
2008-07-15-SmallSection.ll	v		6
2008-07-03-SRet.ll	v		9
2008-07-29-icmp.ll	v		8
2008-08-06-Alloca.ll	v		9
2008-08-01-AsmInline.ll	v		11
2008-08-08-ctlz.ll	v		7
2008-08-08-bswap.ll	v	bswap	12
2008-10-13-LegalizerBug.ll	v		8
2010-11-09-Mul.ll	v		4
2010-11-09-CountLeading.ll	v		7
2008-11-10-xint_to_fp.ll	v		7
addc.ll	v	64-bit add	7

continues on next page

Table 16.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
addi.ll	v	32-bit add, sub	4
address-mode.ll	v	br, -> BB0_2:	8
alloca.ll	v	alloca i8, i32 %size, dynamic allocation	9
analyzebranch.ll	v	br, -> bne, beq	8
and1.ll	v	and	4
asm-large-immediate.ll	v	inline asm	11
atomic-1.ll	v	atomic	12
atomic-2.ll	v	atomic	12
atomics.ll	v	atomic	12
atomics-index.ll	v	atomic	12
atomics-fence.ll	v	atomic	12
br-jmp.ll	v	br, -> jmp	8
brockaddress.ll	v	blockaddress, -> lui, ori	8
cmov.ll	v	select, -> movn, movz	8
cprestore.ll	v	-> .cprestore	9
div.ll	v	sdiv, -> div, mflo	4
divrem.ll	v	sdiv, srem, udiv, urem, -> div, divu	4
div_rem.ll	v	sdiv, srem, -> div, mflo, mfhi	4
divu.ll	v	udiv, -> divu, mflo	4
divu_reml.ll	v	udiv, urem -> div, mflo, mfhi	4
double2int.ll	v	double to int, -> %call16(__fixdfsi)	7
eh-dwraf-cfa.ll	v		9
eh-return32.ll	v	Spill and reload all registers used for exception	9
eh.ll	v	c++ exception handling	12
ex2.ll	v	c++ exception handling	12
fastcc.ll	v	No effect in fastcc but can pass	9
fneg.ll	v	verify Cpu0 don't uses hard float instruction	7
fp-spill-reload.ll	v	-> st $fp, ld $fp	9
frame-address.ll	v	addu $2, $zero, $fp	9
global-address.ll	v	global address, global variable	6
global-pointer.ll	v	global register load and retore, -> .cpload, .cprestore	9
gprestore.ll	v	global register retore, -> .cprestore	9
helloworld.ll	v	global register load and retore, -> .cpload, .cprestore	9
hf16_1.ll	v	function call in PIC, -> ld, jalr	9
i32k.ll	v	argument of constant int passing in register	9

continues on next page

Table 16.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
i64arg.ll	v	argument of constant 64-bit passing in register	9
imm.ll	v	return constant 32-bit in register	9
indirectcall.ll	v	indirect function call	9
init-array.ll	v	check .init	6
inlineasm_constraint.ll	v	inline asm	11
inlineasm-cnstrnt-reg.ll	v	•	11
inlineasmmemop.ll	v	•	11
inlineasm-operand-code.ll	v	•	11
internalfunc.ll	v	internal function	9
jstat.ll	v	switch, -> JTI	8
lb1.ll	v	load i8*, sext i8, -> lb	7
lbu1.ll	v	load i8*, zext i8, -> lbu	7
lh1.ll	v	load i16*, sext i16, -> lh	7
lhu1.ll	v	load i16*, zext i16, -> lhu	7
llcarry.ll	v	64-bit add sub	7
longbranch.ll	v		8
machineverifier.ll	v	delay slot, (comment in machineverifier.ll)	8
mipslopat.ll	v	no check output (comment in mipslopat.ll)	6
misha.ll	v	miss alignment half word access	7
module-asm.ll	v	module asm	11
module-asm-cpu032II.ll	v	module asm	11
mul.ll	v	mul	4
mulll.ll	v	64-bit mul	4
mulull.ll	v	64-bit mul	4
not1.ll	v	not 1	4
null.ll	v	ret i32 0, -> ret $lr	3
o32_cc_byval.ll	v	by value	9
o32_cc_vararg.ll	v	variable argument	9
private.ll	v	private function call	9
rem.ll	v	srem, -> div, mfhi	4
remat-immed-load.ll	v	immediate load	3
remul.ll	v	urem, -> div, mfhi	4
return-vector-float4.ll	v	return vector, -> lui lui …	3
return-vector.ll	v	return vector, -> ld ld …, st st …	3
return_address.ll	v	llvm.returnaddress, -> addu $2, $zero, $lr	9
rotate.ll	v	rotl, rotr, -> rolv, rol, rorv	4
sb1.ll	v	store i8, sb	7

continues on next page

Table 16.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
select.ll	v	select, -> movn, movz	8
seleq.ll	v	following for br with different condition	8
seleqk.ll	v	•	8
selgek.ll	v	•	8
selgt.ll	v	•	8
selle.ll	v	•	8
selltk.ll	v	•	8
selne.ll	v	•	8
selnek.ll	v	•	8
seteq.ll	v	•	8
seteqz.ll	v	•	8
setge.ll	v	•	8
setgek.ll	v	•	8
setle.ll	v	•	8
setlt.ll	v	•	8
setltk.ll	v	•	8
setne.ll	v	•	8

continues on next page

Table 16.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
setuge.ll	v	•	8
setugt.ll	v	•	8
setule.ll	v	•	8
setult.ll	v	•	8
setultk.ll	v	•	8
sext_inreg.ll	v	sext i1, -> shl, sra	4
shift-parts.ll	v	64-bit shl, lshr, ashr, -> call function	9
shl1.ll	v	shl, -> shl	4
shl2.ll	v	shl, -> shlv	4
shr1.ll	v	shr, -> shr	4
shr2.ll	v	shr, -> shrv	4
sitofp-selectcc-opt.ll	v	comment in sitofp-selectcc-opt.ll	7
small-section-reserve-gp.ll	v	Cpu0 option -cpu0-use-small-section=true	6
sra1.ll	v	ashr, -> sra	4
sra2.ll	v	ashr, -> srav	4
stacksave-restore.ll	v		9
stacksize.ll	v	comment in stacksize.ll	9
stchar.ll	v	load and store i16, i8	7
stldst.ll	v	register sp spill	9
sub1.ll	v	sub, -> addiu	4
sub2.ll	v	sub, -> sub	4
tailcall.ll	v	tail call	9
tls.ll	v	ir thread_local global is for c++ "__thread int b;"	12
tls-alias.ll	v	thread_local global and thread local alias	12
tls-models.ll	v	ir external/internal thread_local global	12
uitofp.ll	v	integer2float, uitofp, -> jsub __floatunsisf	9
uli.ll	v	unalignment init, -> sb sb …	6
unalignedload.ll	v	unalignment init, -> sb sb …	6
vector-setcc.ll	v		7
weak.ll	v	extern_weak function, -> .weak	9

continues on next page

Table 16.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
xor1.ll	v	xor, -> xor	4
zeroreg.ll	v	check register $zero	4

These supported test cases are in lbdex/regression-test/Cpu0 which can be gotten from `tar -xf lbdex.tar.gz`.

The regression test is useful in two major reasons. First, it provides the llvm input, assembly output and the running command as well as options in the the sample input file, so all the first glimpse of documentation needed for both end user and programmer are well written in the same test input file. Second, once programmers change their backend compiler especially for optimization, the regression testing may fail for some side effect or bugs from this new change. That is the name of "regression test" stands for. The following file includes the assembly output pattern of two subtargets for Cpu0 backend. Beside checking opcode, it is capable to check register number. In this case, the destination register of "andi" must be the first source register in the following instruction "xori" once it is specified to T1 at the corresponding registers of these two assembly output.

lbdex/regression-test/Cpu0/setule.ll

```
; RUN: llc  -march=cpu0 -mcpu=cpu032I  -relocation-model=pic -O3 %s -o - | FileCheck %s -
↪check-prefix=cpu032I
; RUN: llc  -march=cpu0 -mcpu=cpu032II  -relocation-model=pic -O3 %s -o - | FileCheck %s␣
↪-check-prefix=cpu032II

@j = global i32 5, align 4
@k = global i32 10, align 4
@l = global i32 20, align 4
@m = global i32 10, align 4
@r1 = common global i32 0, align 4
@r2 = common global i32 0, align 4
@r3 = common global i32 0, align 4

define void @test() nounwind {
entry:
  %0 = load i32, i32* @j, align 4
  %1 = load i32, i32* @k, align 4
  %cmp = icmp ule i32 %0, %1
  %conv = zext i1 %cmp to i32
  store i32 %conv, i32* @r1, align 4
; cpu032I:  cmp        $sw, ${{[0-9]+|t9}}, ${{[0-9]+|t9}}
; cpu032I:  andi        $[[T1:[0-9]+|t9]], $sw, 1
; cpu032I:  xori        ${{[0-9]+|t9}}, $[[T1]], 1
; cpu032II:  sltu        $[[T0:[0-9]+|t9]], ${{[0-9]+|t9}}, ${{[0-9]+|t9}}
; cpu032II:  xori        ${{[0-9]+|t9}}, $[[T0]], 1
  %2 = load i32, i32* @m, align 4
  %cmp1 = icmp ule i32 %2, %1
  %conv2 = zext i1 %cmp1 to i32
  store i32 %conv2, i32* @r2, align 4
  ret void
}
```

Run regression test must after build llvm. The following READ.rst and changes in related config files and llvm-lit allowing to setup llvm-lit for pre-build/ pre-installed llvm for running llvm-lit without build llvm.

lbdex/set-llvm-lit/README.txt

```
llvm-lit can be changed to support the installed clang/llvm. Then llvm-lit can
be run directly without build the clang/llvm.

## Run llvm-lit without building llvm-project

    $ pwd
    $ $HOME/test/llvm/llvm/test/CodeGen/Cpu0
    // change the following to dir for your llvm-project
    $ export LLVM_DIR=$HOME/test/llvm
    $ export LLVM_INSTALLED_DIR=$HOME/llvm-installed
    $ ~/llvm-installed/bin/llvm-lit addc.ll -a
    $ ~/llvm-installed/bin/llvm-lit .
    ~/test/llvm/clang/test/CodeGen/Cpu0$ ~/riscv/riscv_newlib/bin/llvm-lit . -a
```

set-llvm-lit % *diff -r origin modify &> set-llvm-lit.diff*

lbdex/set-llvm-lit/set-llvm-lit.diff

```
diff -r origin/bin/llvm-lit modify/bin/llvm-lit
8c8
< install_dir = r'/usr/local/riscv/andes/riscv_newlib'
---
> install_dir = os.environ["LLVM_INSTALLED_DIR"]
diff -r origin/test/Unit/lit.site.cfg.py modify/test/Unit/lit.site.cfg.py
3a4,9
> import os
>
> llvm_project_dir = os.environ["PHOENIX_LLVM_DIR"]
> print('llvm_project_dir: ', llvm_project_dir)
> install_dir = os.environ["LLVM_INSTALLED_DIR"]
>
4a11
> from pathlib import Path
7c14
<     return os.path.join(os.path.dirname(os.path.abspath(__file__)), p)
---
>     return str((Path(__file__).parent / p).resolve())
12,14c19,21
< config.llvm_src_root = path(r"/Users/cschen/llvm/test/llvm")
< config.llvm_obj_root = path(r"/Users/cschen/llvm/test/build")
< config.llvm_tools_dir = path(r"/Users/cschen/llvm/test/build/./bin")
---
> config.llvm_src_root = path(llvm_project_dir)+path(r"/llvm")
> config.llvm_obj_root = path(install_dir)
> config.llvm_tools_dir = path(install_dir)+path(r"/./bin")
17c24
< config.shlibdir = path(r"/Users/cschen/llvm/test/build/./lib")
---
```

(continues on next page)

(continued from previous page)

```
> config.shlibdir = path(install_dir)+path(r"/./lib")
23a31
>     config.shlibdir = config.shlibdir % lit_config.params
diff -r origin/test/lit.site.cfg.py modify/test/lit.site.cfg.py
1c1
< # Autogenerated from /Users/cschen/llvm/test/llvm/test/lit.site.cfg.py.in
---
> # Autogenerated from /u/jonathanchen/andes/riscv/Phoenix/llvm-project/llvm/test/lit.
↪site.cfg.py.in
3a4,9
> import os
>
> llvm_project_dir = os.environ["PHOENIX_LLVM_DIR"]
> print('llvm_project_dir: ', llvm_project_dir)
> install_dir = r'/usr/local/riscv/andes/riscv_newlib'
>
4a11
> from pathlib import Path
7c14
<     return os.path.join(os.path.dirname(os.path.abspath(__file__)), p)
---
>     return str((Path(__file__).parent / p).resolve())
12,19c19,27
< config.host_triple = "arm64-apple-darwin20.6.0"
< config.target_triple = "cpu0-unknown-elf"
< config.llvm_src_root = path(r"/Users/cschen/llvm/test/llvm")
< config.llvm_obj_root = path(r"/Users/cschen/llvm/test/build")
< config.llvm_tools_dir = path(r"/Users/cschen/llvm/test/build/./bin")
< config.llvm_lib_dir = path(r"/Users/cschen/llvm/test/build/./lib")
< config.llvm_shlib_dir = path(r"/Users/cschen/llvm/test/build/./lib")
< config.llvm_shlib_ext = ".dylib"
---
> config.host_triple = "x86_64-unknown-linux-gnu"
> config.target_triple = "riscv64-unknown-elf"
> config.llvm_src_root = path(llvm_project_dir)+path(r"/llvm")
> config.llvm_obj_root = path(install_dir)
> config.llvm_tools_dir = path(install_dir)+path(r"/./bin")
> config.llvm_lib_dir = path(install_dir)+path(r"/./lib")
> config.llvm_shlib_dir = path(install_dir)+path(r"/./lib")
> config.llvm_shlib_ext = ".so"
> config.llvm_plugin_ext = ".so"
22,24c30,34
< config.python_executable = "/opt/homebrew/Frameworks/Python.framework/Versions/3.9/bin/
↪python3.9"
< config.gold_executable = "/opt/homebrew/bin/riscv64-unknown-elf-ld"
< config.ld64_executable = "/Applications/Xcode.app/Contents/Developer/Toolchains/
↪XcodeDefault.xctoolchain/usr/bin/ld"
---
> config.errc_messages = "No such file or directory;Is a directory;Invalid argument;
↪Permission denied"
> config.python_executable = "/usr/bin/python3.6"
> config.gold_executable = "/usr/bin/ld.gold"
```

(continues on next page)

(continued from previous page)

```
> config.ld64_executable = ""
> config.osx_sysroot = path(r"")
33,34c43,44
< config.targets_to_build = " Cpu0"
< config.native_target = "AArch64"
---
> config.targets_to_build = " RISCV"
> config.native_target = "X86"
36,38c46,48
< config.host_os = "Darwin"
< config.host_cc = "/opt/homebrew/opt/llvm/bin/clang "
< config.host_cxx = "/opt/homebrew/opt/llvm/bin/clang++ "
---
> config.host_os = "Linux"
> config.host_cc = "/usr/bin/cc "
> config.host_cxx = "/usr/bin/c++ "
44,45c54,55
< config.have_libxar = 1
< config.have_libxml2 = 1
---
> config.have_libxar = 0
> config.have_libxml2 = 0
52,53c62,63
< config.llvm_host_triple = 'arm64-apple-darwin20.6.0'
< config.host_arch = "arm64"
---
> config.llvm_host_triple = 'x86_64-unknown-linux-gnu'
> config.host_arch = "x86_64"
65a76
>     config.llvm_lib_dir = config.llvm_lib_dir % lit_config.params
diff -r origin/tools/clang/test/Unit/lit.site.cfg.py modify/tools/clang/test/Unit/lit.
↪site.cfg.py
3a4,9
> import os
>
> llvm_project_dir = os.environ["LLVM_DIR"]
> print('llvm_project_dir: ', llvm_project_dir)
> install_dir = os.environ["LLVM_INSTALLED_DIR"]
>
4a11
> from pathlib import Path
7c14
<     return os.path.join(os.path.dirname(os.path.abspath(__file__)), p)
---
>     return str((Path(__file__).parent / p).resolve())
12,15c19,22
< config.llvm_src_root = path(r"/Users/cschen/llvm/test/llvm")
< config.llvm_obj_root = path(r"/Users/cschen/llvm/test/build")
< config.llvm_tools_dir = path(r"/Users/cschen/llvm/test/build/./bin")
< config.llvm_libs_dir = path(r"/Users/cschen/llvm/test/build/./lib")
---
> config.llvm_src_root = path(llvm_project_dir)+path(r"/llvm")
```

(continues on next page)

(continued from previous page)

```
> config.llvm_obj_root = path(install_dir)
> config.llvm_tools_dir = path(install_dir)+path(r"/./bin")
> config.llvm_libs_dir = path(install_dir)+path(r"/./lib")
17c24
< config.clang_obj_root = path(r"/Users/cschen/llvm/test/build/tools/clang")
---
> config.clang_obj_root = path(install_dir)+path(r"/tools/clang")
19,20c26,27
< config.shlibdir = path(r"/Users/cschen/llvm/test/build/./lib")
< config.target_triple = "cpu0-unknown-elf"
---
> config.shlibdir = path(install_dir)+path(r"/./lib")
> config.target_triple = "riscv64-unknown-elf"
28a36
>     config.shlibdir = config.shlibdir % lit_config.params
36c44
<     config, os.path.join(path(r"/Users/cschen/llvm/test/clang"), "test/Unit/lit.cfg.py
↪"))
---
>     config, os.path.join(path(llvm_project_dir)+path(r"/clang"), "test/Unit/lit.cfg.py
↪"))
diff -r origin/tools/clang/test/lit.site.cfg.py modify/tools/clang/test/lit.site.cfg.py
3a4,9
> import os
>
> llvm_project_dir = os.environ["LLVM_DIR"]
> print('llvm_project_dir: ', llvm_project_dir)
> install_dir = os.environ["LLVM_INSTALLED_DIR"]
>
4a11
> from pathlib import Path
7c14
<     return os.path.join(os.path.dirname(os.path.abspath(__file__)), p)
---
>     return str((Path(__file__).parent / p).resolve())
12,17c19,24
< config.llvm_src_root = path(r"/Users/cschen/llvm/test/llvm")
< config.llvm_obj_root = path(r"/Users/cschen/llvm/test/build")
< config.llvm_tools_dir = path(r"/Users/cschen/llvm/test/build/./bin")
< config.llvm_libs_dir = path(r"/Users/cschen/llvm/test/build/./lib")
< config.llvm_shlib_dir = path(r"/Users/cschen/llvm/test/build/./lib")
< config.llvm_plugin_ext = ".dylib"
---
> config.llvm_src_root = path(llvm_project_dir)+path(r"/llvm")
> config.llvm_obj_root = path(install_dir)
> config.llvm_tools_dir = path(install_dir)+path(r"/./bin")
> config.llvm_libs_dir = path(install_dir)+path(r"/./lib")
> config.llvm_shlib_dir = path(install_dir)+path(r"/./lib")
> config.llvm_plugin_ext = ".so"
19,24c26,35
< config.clang_obj_root = path(r"/Users/cschen/llvm/test/build/tools/clang")
< config.clang_src_dir = path(r"/Users/cschen/llvm/test/clang")
```

(continues on next page)

(continued from previous page)

```
< config.clang_tools_dir = path(r"/Users/cschen/llvm/test/build/./bin")
< config.host_triple = "arm64-apple-darwin20.6.0"
< config.target_triple = "cpu0-unknown-elf"
< config.host_cxx = "/opt/homebrew/opt/llvm/bin/clang++"
---
> config.errc_messages = "No such file or directory;Is a directory;Invalid argument;
↪Permission denied"
> config.clang_lit_site_cfg = __file__
> config.clang_obj_root = path(install_dir)+path(r"/tools/clang")
> config.clang_src_dir = path(llvm_project_dir)+path(r"/clang")
> config.clang_tools_dir = path(install_dir)+path(r"/./bin")
> config.clang_lib_dir = path(install_dir)+path(r"/lib")
> config.host_triple = "x86_64-unknown-linux-gnu"
> config.target_triple = "riscv64-unknown-elf"
> config.host_cc = "/usr/bin/cc"
> config.host_cxx = "/usr/bin/c++"
34c45
< config.enable_experimental_new_pass_manager = 0
---
> config.enable_experimental_new_pass_manager = 1
36,37c47,48
< config.host_arch = "arm64"
< config.python_executable = "/opt/homebrew/Frameworks/Python.framework/Versions/3.9/bin/
↪python3.9"
---
> config.host_arch = "x86_64"
> config.python_executable = "/usr/bin/python3.6"
40a52
> config.llvm_external_lit = path(r"")
diff -r origin/utils/lit/lit.site.cfg modify/utils/lit/lit.site.cfg
3a4,9
> import os
>
> llvm_project_dir = os.environ["PHOENIX_LLVM_DIR"]
> print('llvm_project_dir: ', llvm_project_dir)
> install_dir = os.environ["LLVM_INSTALLED_DIR"]
>
4a11
> from pathlib import Path
7c14
<     return os.path.join(os.path.dirname(os.path.abspath(__file__)), p)
---
>     return str((Path(__file__).parent / p).resolve())
13,15c20,22
< config.llvm_src_root = "/Users/cschen/llvm/test/llvm"
< config.llvm_obj_root = "/Users/cschen/llvm/test/build"
< config.llvm_tools_dir = "/Users/cschen/llvm/test/build/./bin"
---
> config.llvm_src_root = path(llvm_project_dir)+path(r"/llvm")
> config.llvm_obj_root = path(install_dir)
> config.llvm_tools_dir = path(install_dir)+path(r"/./bin")
30c37
```

(continues on next page)

(continued from previous page)

```
< lit_config.load_config(config, "/Users/cschen/llvm/test/build/utils/lit/tests/lit.cfg")
---
> lit_config.load_config(config, os.path.join(config.llvm_obj_root, "utils/lit/tests/lit.
↪cfg"))
```

- Only tools/clang/test/lit.site.cfg.py and test/lit.site.cfg.py need to be modified. The other tools/clang/test/Unit/lit.site.cfg.py, test/Unit/lit.site.cfg.py and utils/lit/tests/lit.site.cfg.in are empty and useless. However I modify them either.

CHAPTER

SEVENTEEN

APPENDIX C: THE CONCEPT OF NPU (NEURAL PROCESSOR UNIT) COMPILER

- *Deep Learning Theory*
  - *Deep Learning*
  - *CNN*
  - *Book*
- *Deep learning compiler*
  - *Survey*
  - *GPU*
  - *NNEF*
  - *OpenVX*
    - *SYCL*
  - *Tools*
- *NPU compiler*
  - *Abstract*
  - *MLIR and IREE*
  - *Tensorflow*
  - *mlir to onnx*
  - *Support tensorflow*
  - *ONNC*
- *LLVM based NPU compiler*
  - *llvm IR for NPU compiler*
  - *Open source project*
- *ONNX*
  - *viewer*
  - *install*
  - *onnx api*

17.1 Deep Learning Theory

17.1.1 Deep Learning

Hung-Yi Lee's video[1].

17.1.2 CNN

CNN: They have applications in image and video recognition, recommender systems, image classification, medical image analysis, natural language processing, and financial time series[4].

Concept about how to apply Convolution and MaxPool to getting features from image[2]. Conv+MaxPool -> get features map and downsize image, more Conv+MaxPool can filter image and higher level of features and downsize more image. CNN model used in image recognition.

Concept and data applying in Deap Learning for different models of CNN[3].

17.1.3 Book

http://www.deeplearningbook.org/ NTU: Hung-yi Lee MLDS (Machine Learing Deep + Structure approaches), the trend for future implementation.

17.2 Deep learning compiler

17.2.1 Survey

The current open source compilers for deep learning[7] as follows,

TVM compiler open source infrastructure supports most of DL Frameworks such as TensorFlow, PyTorch…, and generating CUDA/OpenCL/OpenGL for gpu and LLVM for cpu.

17.2.2 GPU

The NVIDIA CUDA Toolkit provides a development environment for creating high-performance GPU-accelerated applications. GPU-accelerated CUDA libraries enable acceleration across multiple domains such as linear algebra, image and video processing, deep learning and graph analytics[8].

OpenCL runs on AMD GPUs and provides partial support for TensorFlow and PyTorch. If you want to develop new networks some details might be missing, which could prevent you from implementing the features you need[Page 698, 8]. For instance, if ARM GPU doesn't implement operation "Cosh" on TVM while a DL model created from PyTorch generate "Cosh" operation, then it will fail to run the DL model though TVM compile PyTorch model into OpenCL. Once "Cosh" is implemented with kernel fuction "Cosh" in OpenCL by calling GPU's instructions, it can be fixed.

[1] https://www.youtube.com/watch?v=CXgbekl66jc&list=PLJV_el3uVTsPy9oCRY30oBPNLCo89yu49
[4] https://en.wikipedia.org/wiki/Convolutional_neural_network
[2] http://violin-tao.blogspot.com/2017/07/ml-convolutional-neural-network-cnn.html
[3] https://github.com/onnx/models
[7] https://arxiv.org/pdf/2002.03794.pdf
[8] https://missinglink.ai/guides/tensorflow/tensorflow-support-opencl/

Table 1. The detailed comparison of popular DL compilers.

	TVM	TC	Glow	nGraph+PlaidML	XLA
Core/Programming Language					
Core	C++	C++	C++	C++	C++
Programming	Python/C++	Python/C++	Python/C++	Python/C++	Python/C++
Supported Hardware Targets					
CPU	✓	✓	✓	✓	✓
NVIDIA-GPU	✓	✓	✓	✓	✓
AMD-GPU	✓	×	✓	✓	✓
FPGA	✓	×	×	×	×
TPU	×	×	×	×	✓
NNP	×	×	×	✓	×
Customed	✓	×	✓	✓	✓
Supported DL Frameworks					
TensorFlow	✓	×	×	✓	✓
PyTorch	✓	✓	✓	×	✓
MXNet	✓	×	×	× not active	×
Caffe2	✓	✓	✓	×	×
ONNX	✓	×	✓	✓	×
CoreML	✓	×	×	×	×
Keras	✓	×	×	✓	✓
PaddlePaddle	×	×	×	✓	×
DarkNet	✓	×	×	×	×
Supported Generating Languages					
CUDA	✓	✓	×	×	✓
OpenCL	✓	×	✓	✓	✓
Metal	✓	×	×	✓	×
LLVM	✓	✓	✓	✓	✓
OpenGL	✓	×	×	✓	×
Supported Features/Strategies					
AOT	✓	×	✓	✓ official release	✓
JIT	✓	✓	✓	✓	✓
Training	–	✓	✓	✓	✓
Quantization	–	×	✓	✓	×
Automatic Differentiation	–	✓	✓	✓	×
Dynamic Shape	✓	×	×	✓	×
Auto-tuning	✓	✓	×	✓ only tiling	×

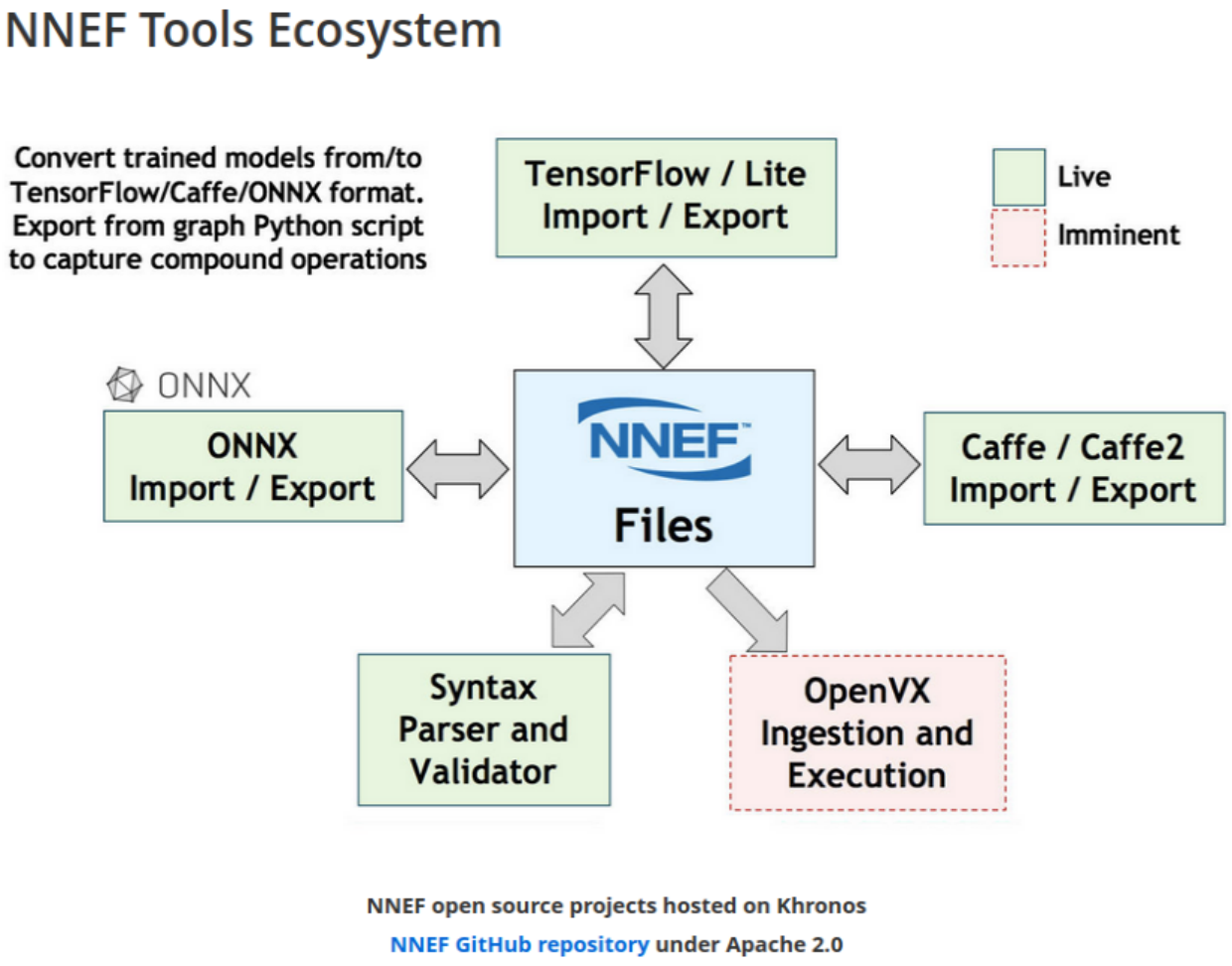
17.2.3 NNEF

Neural Network Exchange Format.

NPU or GPU can apply NNEF to supprt all AI models.

The goal of NNEF is to enable data scientists and engineers to easily transfer trained networks from their chosen training framework into a wide variety of inference engines. A stable, flexible and extensible standard that equipment manufacturers can rely on is critical for the widespread deployment of neural networks onto edge devices, and so NNEF encapsulates a complete description of the structure, operations and parameters of a trained neural network, independent of the training tools used to produce it and the inference engine used to execute it[9].

ONNX and NNEF are Complementary ONNX moves quickly to track authoring framework updates NNEF provides a stable bridge from training into edge inferencing engines[Page 700, 9].



ONNX import/export to NNEF, should edge NPU use NNEF[9]?

[9] https://www.khronos.org/nnef

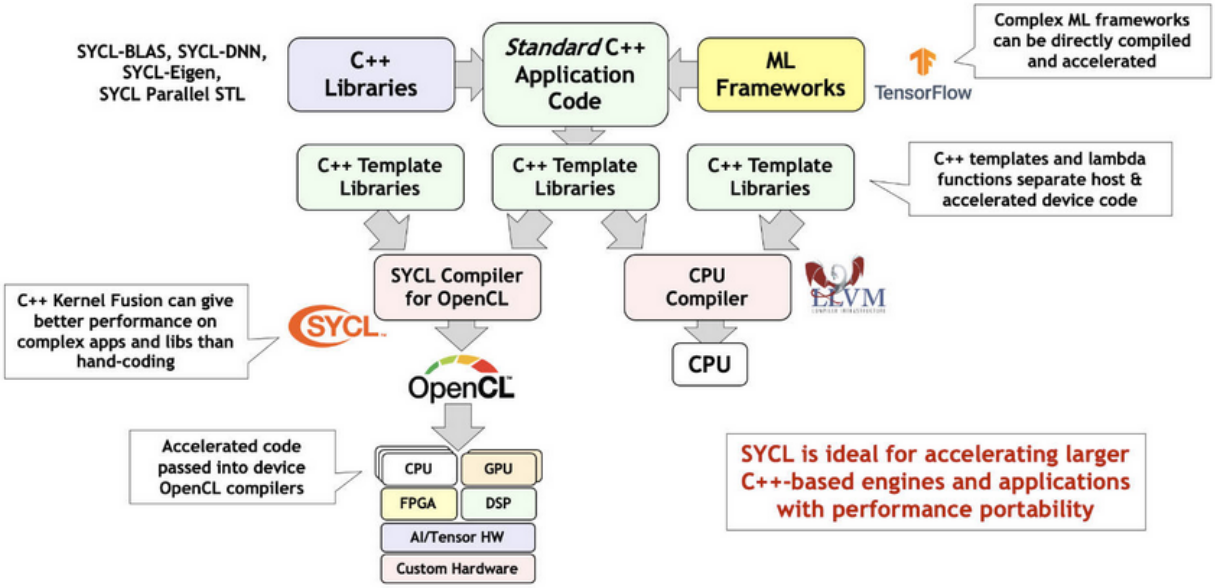
17.2.4 OpenVX

OpenVX enables the graph to be extended to include hardware architectures that don't support programmable APIs[10].

It is designed by the Khronos Group to facilitate portable, optimized and power-efficient processing of methods for vision algorithms[11]. Nothing about NPU I think.

SYCL

User use SYCL or DSL compile domain language into SYCL and run on OpenCL hardwares[12]. An example here[13].



https://github.com/Jonathan2251/nc/OpenCL_SYCL

17.2.5 Tools

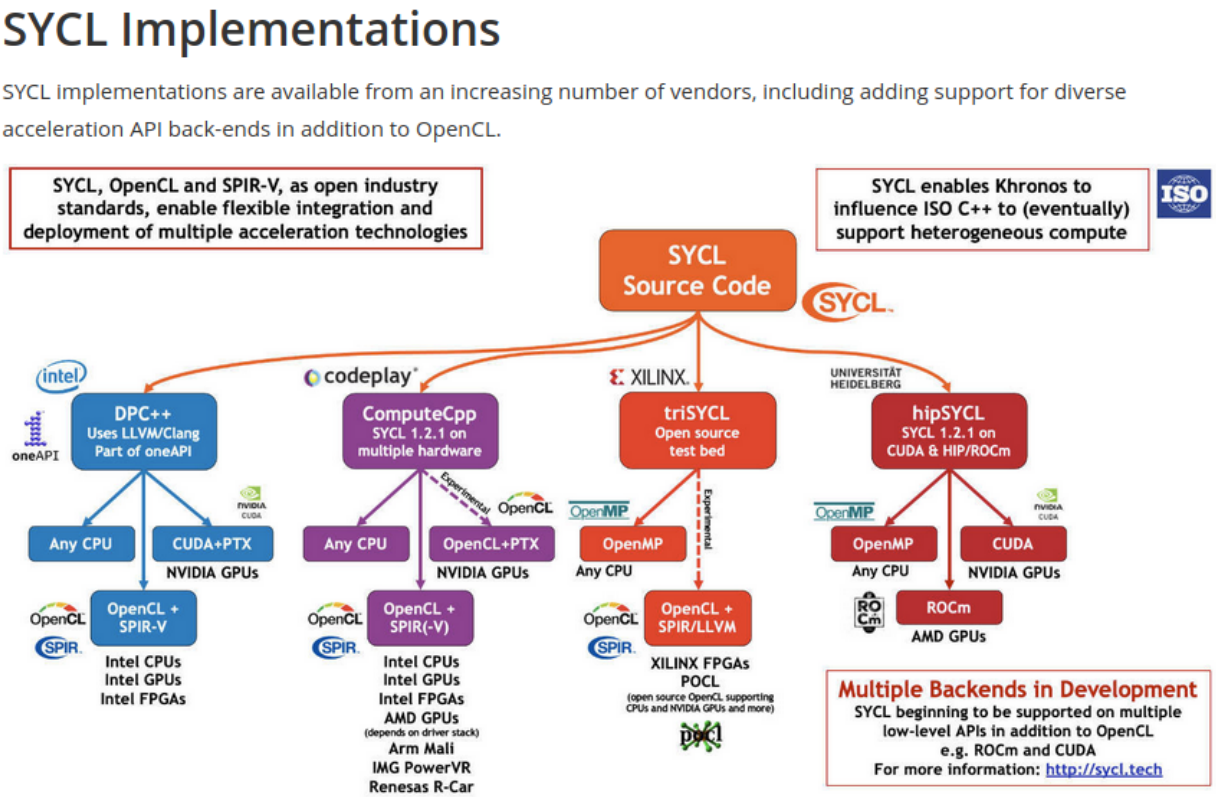
Create onnx test file[14].

17.3 NPU compiler

17.3.1 Abstract

Tensorflow support unknown shape[16]. Though our npu support kernel call where kernel call is a set of commands to npu to deal shape at run time, it is unefficiency. As I remember mlit supports binding shape for unknown at compile-time but not always work. Luckily, we can customilze by redefining model to binding shape staticlly [20200412]

[10] https://www.khronos.org/openvx
[11] https://en.wikipedia.org/wiki/OpenVX
[12] https://www.khronos.org/sycl
[13] https://en.wikipedia.org/wiki/SYCL
[14] https://www.youtube.com/watch?v=QZQwmZTfLmI
[16] https://pgaleone.eu/tensorflow/2018/07/28/understanding-tensorflow-tensors-shape-static-dynamic/



17.3.2 MLIR and IREE

IREE (Intermediate Representation Execution Environment, pronounced as "eerie") is an MLIR-based end-to-end compiler that lowers ML models to a unified IR optimized for real-time mobile/edge inference against heterogeneous hardware accelerators. IREE also provides flexible deployment solutions for the compiled ML models[15] as the following figure.

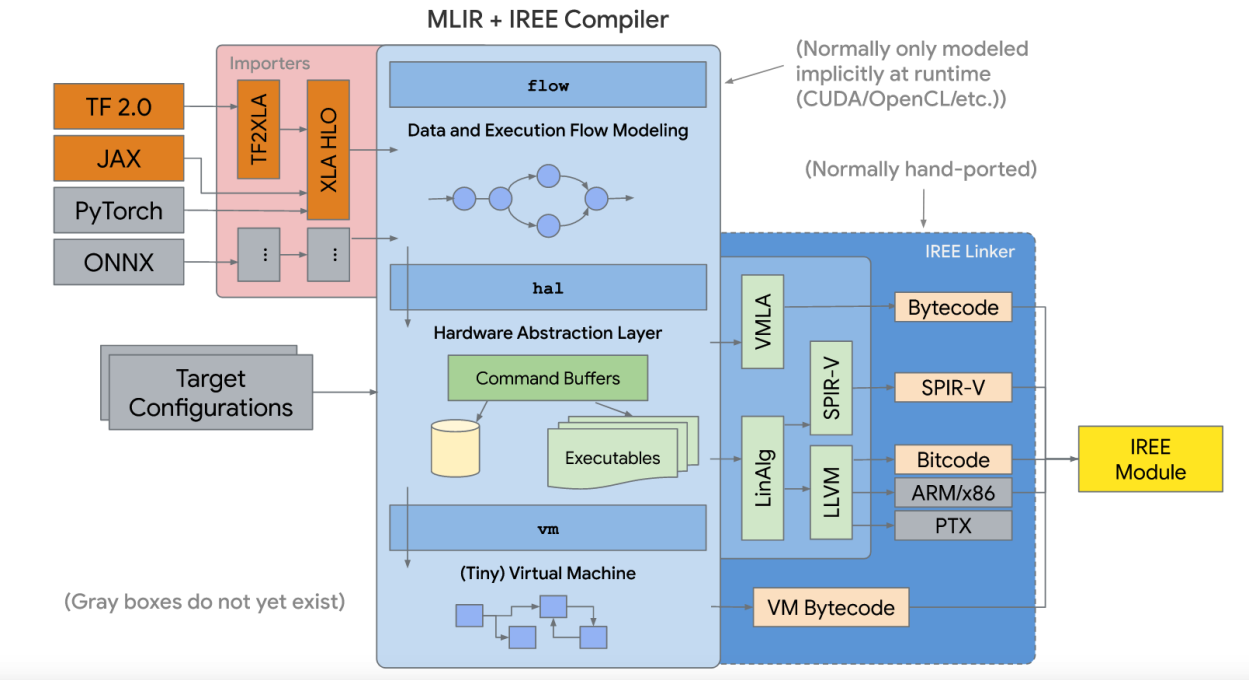
- HAL IR: Vulkan-like allocation and execution model encoding -> on-line first-time compilation and save in cache. Executable compilation via architecture specific backend compiler plugins.
- VM IR: Dynamic module linkage definitions (imports, exports, globals, etc)[17].

The purpose of mlir is:

- Connect cpu with mlir-to-llvm-ir.

[15] https://github.com/google/iree

[17] Page 15 of https://docs.google.com/presentation/d/1RCQ4ZPQFK9cVgu3IH1e5xbrBcqy7d_cEZ578j84OvYI/edit#slide=id.g6e31674683_0_23101



17.3.3 Tensorflow

The mechansim of Mlir and iree applied on tensorflow as the figure above section is not fitted for off-line edge npu that stand alone without server-connection for tunning weight of face detection's purpose. It is designed for on-line server-connected npu. The gpu of supporting spirv is best candidate until this date 2020/5/12.

At beginning, tensorflow rely on api without fixed format such as ONNX[19]. As a result ONNX emerged and adopted for most of npu in their private backend compiler. Google does not like to hire onnx as the format for npu backend compiler onnx-mlir project[20] which convert onnx to mlir dialect is sponsored by Google I guess[21] for encourging new npu compiler development hiring mlir as their compiler input (convert onnx to mlir then handling mlir input).

With mlir and iree appear on tensorflow as a series of fixed formats in tensorflow as section above. The hardware vendors for cloud server AI machine with heterogeneous hardware accelerators will run tensorflow system by supporting mlir/iree input format in their compilers more and more. So, it is unavoidable that tensorflow system's npu vendors have to support mlir/iree input format beyond onnx. Or open source software or vendor software appear to do transfer from mlir/iree to onnx. (python in tensorflow api allow unknown type and shape size, so it cannot transer python api to onnx fully).

If lucky, google may hire onnx. Because onnx format is older than mlir in history. In addition in aspect of format, mlir has mult-level mult-dialect and more complicate while onnx is easy and better to understand (P.S. I don't dig into mlir yet). Many AI models has supported onnx file format. For some AI model's formats that run on tensorflow without supporting onnx, aplly tensorflow-onnx open source project[22] can convert tensorflow to onnx partly.

Onnx alliance may release some programs for transfering mlir to onnx for fighting agiant mlir-iree growing in npu compiler but not at this moment.

For off-line edge npu that stand alone without server-connection for tunning weight of face detection's purpose, supprting mlir-iree compiler may not necessary.

[19] Actually onnx format based on IO api with protobuffer. It has real binary format but may change from version to version. Tensorflow api has no real binary format.

[20] https://github.com/onnx/onnx-mlir

[21] https://groups.google.com/a/tensorflow.org/forum/#!topic/mlir/2FT4sD8kqTY

[22] https://github.com/onnx/tensorflow-onnx

17.3.4 mlir to onnx

https://www.tensorflow.org/mlir

https://mlir.llvm.org/talks/

https://llvm.org/devmtg/2019-04/talks.html#Tutorial_1

- 3 ppt in llvm tutorials

https://llvm.org/devmtg/2019-04/slides/Tutorial-AminiVasilacheZinenko-MLIR.pdf

build mlir: https://mlir.llvm.org/getting_started/

```
~/llvm/1/llvm-project/build$ cmake -G Ninja ../llvm \
>    -DLLVM_ENABLE_PROJECTS=mlir \
>    -DLLVM_BUILD_EXAMPLES=ON \
>    -DLLVM_TARGETS_TO_BUILD="X86;NVPTX;AMDGPU" \
>    -DCMAKE_BUILD_TYPE=Release \
>    -DLLVM_ENABLE_ASSERTIONS=ON

~/llvm/1/llvm-project/build$ cmake --build . --target check-mlir
[200/1919] Generating VCSRevision.h
-- Found Git: /usr/bin/git (found version "2.17.1")
[1604/1919] Building CXX object tools/mlir/tools/mlir-linalg-ods-gen/CMakeFiles/mlir-
↪linalg-ods-gen.dir/mlir-linalg-ods-gen.cpp.o
/home/cschen/llvm/1/llvm-project/mlir/tools/mlir-linalg-ods-gen/mlir-linalg-ods-gen.
↪cpp:935:6: warning: 'bool {anonymous}::Expression::operator==(const {anonymous}
↪::Expression&) const' defined but not used [-Wunused-function]
 bool Expression::operator==(const Expression &e) const {
      ^~~~~~~~~~
[1918/1919] Running the MLIR regression tests

Testing Time: 9.88s
  Unsupported Tests:  16
  Expected Passes  : 465
```

run: https://mlir.llvm.org/docs/Tutorials/Toy/

```
~/llvm/1/llvm-project/mlir/test/Examples/Toy/Ch1$ ~/llvm/1/llvm-project/build/bin/toyc-
↪ch1 ast.toy -emit=ast
...
~/llvm/1/llvm-project/mlir/test/Examples/Toy/Ch1$ ~/llvm/1/llvm-project/build/bin/toyc-
↪ch1 ast.toy -emit=ast 2>&1 | ~/llvm/1/llvm-project/build/bin/FileCheck ast.toy
~/llvm/1/llvm-project/mlir/test/Examples/Toy/Ch1$ ~/llvm/1/llvm-project/build/bin/llvm-
↪lit ast.toy
-- Testing: 1 tests, 1 workers --
PASS: MLIR :: Examples/Toy/Ch1/ast.toy (1 of 1)

Testing Time: 0.11s
  Expected Passes: 1
```

The result I run is based on git commit 455ccde1377b3ec32d321eb7c38808fecdf230a8 Date: Sun May 17 21:00:09 2020 -0400

17.3.5 Support tensorflow

Question:

Sean,

As I said, we can always redefine AI model to remove unknown type or dimension at ahead of time compilation to fit static compilation binding, and my AI input models are CNN without loop (it is DAG form). For this kind of models on tensorflow, can it be translated absolutely to mlir form based on what you know? If it can, then I can write converting program for mlir to my npu internal ir to support tensorflow.

Answer:

For programs with those restrictions, converting to MLIR xla_hlo dialect is always possible.

Note that it is always possible to convert a TensorFlow GraphDef into MLIR tensorflow dialect. MLIR is very flexible. But MLIR tensorflow dialect is too general for NPU and needs to be converted to MLIR xla_hlo dialect.

– Sean Silva

Sean,

Thank you! I am going to pass this information to my boss. We don't study mlir yet. I believe it will take effort and we only have few engineers on compiler taking a lot of works. There other resource such as tensorflow-onnx but only part of supporting tensorflow to onnx converting.

Jonathan

17.3.6 ONNC

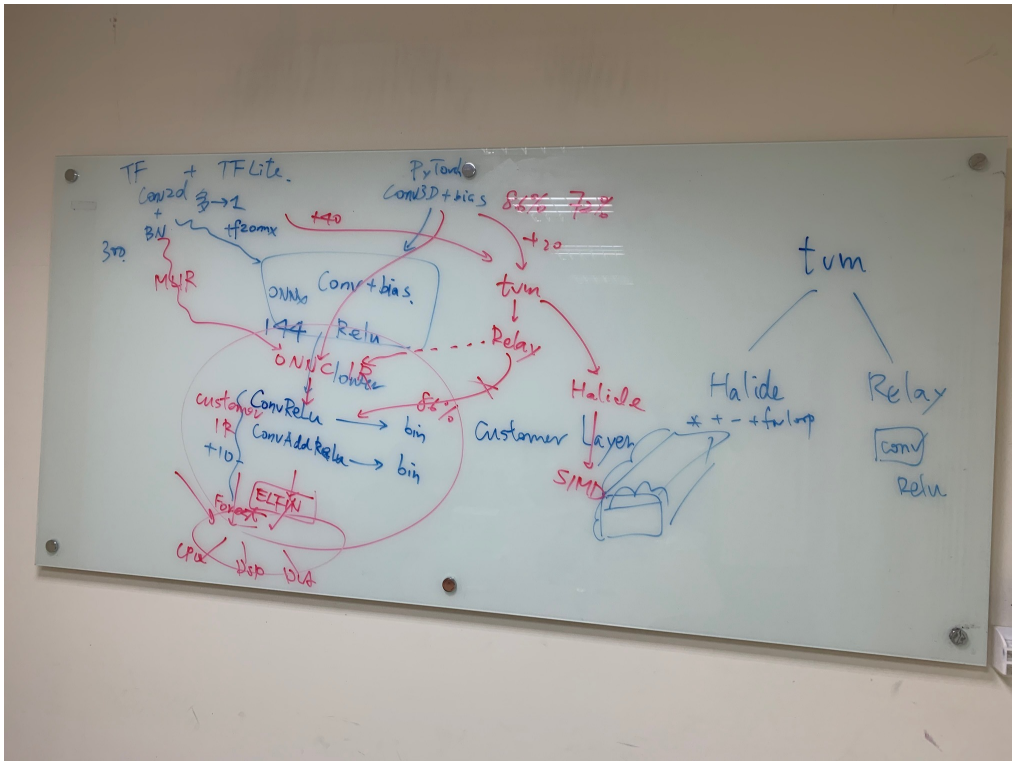


Fig. 17.1: ONNC

- Private IR is better than llvm intrinsic IR for non-VLIW (SIMD or MIMD). Stride, memory dependences, ..., llvm has not much advantages in this. Private IR is better.
- Support MLIR to private IR in Novemember. Open source tensorflow to onnx has limited operations support only, bad and not good.
- TVM support python interfaces but from TVM -> relay is not good according their experience. From MLIR is better.
- Support MLIR, pytorch, caffe are enough. Future has less AI model tools.
- Run time scheduling customer support.
- tf-onnx is not sufficient to support tf's operators and bad. So, translate tf through MLIR to ONNC to customer IR is must.

https://onnc.ai/

17.4 LLVM based NPU compiler

Use LLVM rather than GCC because TVM open source compiler generating llvm-ir for X86 and ARM and may extend to support other CPU in future. Though TVM may uses BYOC to generate C function calling NPU's builtin function, the AutoTVM layer allows doing more optimization code generation such as vectorization, CPU/NPU instructions iterleaving, ..., etc[26][27]. However in cloud of DL scenario, since the time of data transfer from global DRAM to PE's (Processor Entity) local memory SRAM is unknown until run time, applying BYOC for calling NPU's builtin function then using GCC instead of llvm is possilbe. NPU usually implements data parallel instructions such as matrix multiplication, convolution, relu, pool, ..., to speed up the Deep Learning Operations. For other operations not very data parallel such as global pool, concat, sort, ..., may leave to CPU finishing them. TVM output llvm-ir rather than GCC since GCC community never had desire to enable any tools besides compiler (Richard Stallman resisted attempts to make IR more reusable to prevent third-party commercial tools from reusing GCC's frontends)[28].

The way for supporting llvm based NPU compiler is to implement builtin functions in clang and the corresponding specific NPU's llvm-intrinsice functions in llvm backend. For instance, the matrix multiplication operations of clang/llvm support as the following table.

Table 17.1: Matrix Multiplication defined in clang, llvm and ASM

Component	Function/IR/Instruction
clang's builtin	__builtin_tensor_matmul(A, B, C)
llvm's intrinsic	@llvm.lt.matmul %A, %B, %C
NPU's ASM instruction	matmul $A, $B, %C

The detail steps to support clang's builtin and llvm's intrinsic function for backend are in my books[29][30].

[26] https://discuss.tvm.apache.org/t/how-to-see-different-ir-relay-te-tir-from-my-own-pytorch-model/13684
[27] https://discuss.tvm.apache.org/t/which-is-the-best-way-to-port-tvm-to-a-new-ai-accelerator/6905
[28] https://stackoverflow.com/questions/40799696/how-is-gcc-ir-different-from-llvm-ir/40802063
[29] http://jonathan2251.github.io/lbt/clang.html#builtin-functions
[30] http://jonathan2251.github.io/lbd/funccall.html#add-specific-backend-intrinsic-function

17.4.1 llvm IR for NPU compiler

Though npu has no general purpose registers GPR, it is possible to apply llvm ir for npu to do codegen by llvm as follows,

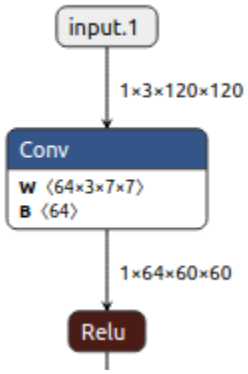


Fig. 17.2: Conv operation in onnx file

```
@x1 = global [1 x [3 x [120 x [120 x float]]]], align 4
@w1 = global [64 x [3 x [7 x [7 x float]]]], align 4
@conv = @llvm.npu1.conv float* @x, float* @weight, ...
```

Conclusion:

1. No GPRs in NPU but can get advantage of code-gen by llvm-tblgen tool.
2. The vector size of llvm is power of 2 (1, 2, 4, 8, . . .). But it can be achieved by modifying llvm kernel source data type.

ref. code/llvm-ex1.c

3. Though NPU has no GPRs, the memory allocation can be done by adjust instructions order and split instructions (if over NPU's memory) in passes of LLVM IR level.

reference:

- section 5.2.2 Code Generation based on Low-Level IR.The low-level IR adopted by most DL compilers can be eventually lowered to LLVM IR, and benefits from LLVM's mature optimizer and code generator[Page 698, 7].

17.4.2 Open source project

- onnx to mlir dialect: https://github.com/onnx/onnx-mlir
- tensorflow to onnx: https://github.com/onnx/tensorflow-onnx
- onnx to tensorflow: https://github.com/onnx/onnx-tensorflow

17.5 ONNX

17.5.1 viewer

- Web for opening onnx https://lutzroeder.github.io/netron/
- Application tool for opening onnx https://github.com/lutzroeder/netron // find "Browser: Start" in this page

Netron app in ubuntu.

17.5.2 install

$ pip install onnx

17.5.3 onnx api

- copy onnx file: ex. nc/code/copy_onnx.py referenece here[23].
- create onnx file: ex. nc/code/create1.py referenece here[24].
- Kneron onnx creating tool[25].

[23] https://github.com/onnx/onnx/issues/2052

[24] https://www.google.com/search?client=ubuntu&hs=bS9&channel=fs&sxsrf=ALeKk00IITG3Dj_IryeytZ_iTJJE3PszMA%3A1597217944046&ei=mJwzX8KZAuiLr7wPr_aq0AU&q=onnx+python+api&oq=onnx+python+api&gs_lcp=CgZwc3ktYWIQAzIECCMQJzIGCAAQCBAeOggIABA sclient=psy-ab&ved=0ahUKEwjCxbvBlJXrAhXoxYsBHS-7CloQ4dUDCAs&uact=5

[25] https://github.com/kneron/ONNX_Convertor/blob/master/optimizer_scripts/onnx2onnx.py

CHAPTER

EIGHTEEN

TODO LIST

CHAPTER

NINETEEN

RESOURCES

19.1 Build steps

https://github.com/Jonathan2251/lbd/blob/master/README.md

19.2 Book example code

The example code lbdex.tar.gz is available in:

http://jonathan2251.github.io/lbd/lbdex.tar.gz

19.3 Alternate formats

The book is also available in the following formats:

19.4 Presentation files

19.5 Search this website

- search