
Tutorial: Creating an LLVM Backend for the Cpu0 Architecture

Release 3.7.0

Chen Chung-Shu gamma_chen@yahoo.com.tw

September 25, 2015

1 About	1
1.1 Authors	1
1.2 Contributors	1
1.3 Acknowledgments	1
1.4 Support	2
1.5 Revision history	2
1.6 Licensing	5
1.7 Motivation	5
1.8 Preface	6
1.9 Prerequisites	6
1.10 Outline of Chapters	7
2 Cpu0 architecture and LLVM structure	9
2.1 Cpu0 Processor Architecture Details	10
2.2 LLVM Structure	15
2.3 Create Cpu0 backend	26
3 Backend structure	55
3.1 TargetMachine structure	55
3.2 Add AsmPrinter	96
3.3 Add Cpu0DAGToDAGISel class	120
3.4 Handle return register lr	129
3.5 Add Prologue/Epilogue functions	148
3.6 Data operands DAGs	158
3.7 Summary of this Chapter	160
4 Arithmetic and logic instructions	163
4.1 Arithmetic	163
4.2 Logic	192
4.3 Summary	202
5 Generating object files	203
5.1 Translate into obj file	203
5.2 ELF obj related code	204
5.3 Backend Target Registration Structure	219
6 Global variables	231
6.1 Cpu0 global variable options	232
6.2 Static mode	245
6.3 pic mode	250
6.4 Global variable print support	258

6.5	Summary	260
7	Other data type	261
7.1	Local variable pointer	261
7.2	char, short int and bool	263
7.3	long long	269
7.4	float and double	273
7.5	Array and struct support	274
8	Control flow statements	281
8.1	Control flow statement	281
8.2	Cpu0 backend Optimization: Remove useless JMP	287
8.3	Fill Branch Delay Slot	291
8.4	Conditional instruction	294
8.5	RISC CPU knowledge	300
9	Function call	303
9.1	Mips stack frame	303
9.2	Load incoming arguments from stack frame	307
9.3	Store outgoing arguments to stack frame	323
9.4	Structure type support	340
9.5	Function call optiomization	350
9.6	Other features supporting	357
9.7	Summary	383
10	ELF Support	385
10.1	ELF format	386
10.2	llvm-objdump	391
11	Assembler	405
11.1	AsmParser support	405
11.2	Inline assembly	432
12	C++ support	453
12.1	Exception handle	453
12.2	Thread variable	460
12.3	Atomic	473
13	Verify backend on verilog simulator	489
13.1	Create verilog simulator of Cpu0	489
13.2	Verify backend	500
13.3	Other llvm based tools for Cpu0 processor	510
14	Appendix A: Getting Started: Installing LLVM and the Cpu0 example code	511
14.1	Setting Up Your Mac	511
14.2	Setting Up Your Linux Machine	521
15	Appendix B: Cpu0 document and test	525
15.1	Cpu0 document	525
15.2	Cpu0 Regression Test	527
16	Todo List	533
17	Book example code	535
18	Alternate formats	537

ABOUT

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

1.1 Authors

陳鍾樞

Chen Chung-Shu gamma_chen@yahoo.com.tw

<http://jonathan2251.github.io/web/index.html>

Anoushe Jamshidi ajamshidi@gmail.com

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, llvmd@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 Revision history

Version 3.7.1, Not release yet

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 of relocation 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 modified 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-organize Chapter 3. Re-organize Chapter 8. Re-organize Chapter 10. Re-organize Chapter 11. Re-organize 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.

1.6 Licensing

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

1.7 Motivation

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 complicate system, we often feel the concept from school or book is not much or not details enough. Compiler is a very complicate 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) to feel the parsing knowledge and tools application.

On the other hand, the compiler engineers who graduated from school often facing the real market complicate CPUs and spec. Since for market reason, there are a serial of CPUs and ABI (Application Binary Interface) to deal with. Moreover, for speed reason, the real compiler backend program is too complicate to be a learning material in compiler backend design 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 you created by verilog language. Through this school learning purpose CPU, you have 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 complicate to be finished by one single person alone.

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 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 studied 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. And if a programmer has well knowledge in “Design Pattern”, then the C++ can supply more readability and rewratability (I have ever read a book of “system language” about software quality, it lists these items: read ablity, rewrite ablity, reuse ability and performance to define the software quality). Object oriented programming exists for solving the big and complex software development. Since compiler and OS are complex software definitely, why gcc and linux not use c++ ⁴. This is the reason I try to create a backend under llvm rather than gcc.

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>

1.8 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](#). The Cpu0 example code referenced in this book can be found [here](#). 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 Run backend.

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 fix bugs. You can take advantage by compare difference from version to version, and hire llvm development team effort. Since Cpu0 is an educational architecture, it is missing some key pieces of documentation needed when developing a compiler, such as an Application Binary Interface (ABI). We implement our backend 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 extend compiler toolchain to support a new CPU based on llvm structure. To program on Linux OS, you program or write a driver without knowing every details in OS. For example in a specific USB device driver program on Linux platform, he or she will try to understand the USB spec., 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 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.

1.9 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](#)

1.10 Outline of Chapters

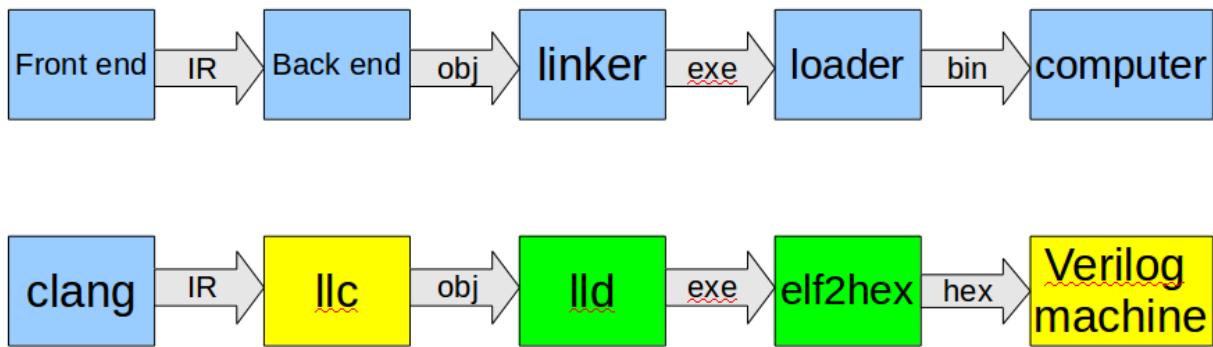


Figure 1.1: Code generation and execution flow

The upper half of Figure 1.1 is the work flow and software package of a computer program be generated and executed. IR stands for Intermediate Representation. The lower half is this book's work flow and software package of the toolchain extended implementation based on llvm. Except clang, the other blocks need to be extended for a new backend development. This book implement the yellow boxes part. The Cpu0 llvm backend can be find 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, create an llvm backend for the Cpu0. Chapter 2 to 11.
2. ELF linker for Cpu0 which extended from lld. Chapter 13.
3. elf2hex extended from llvm-objdump. Chapter 13.
4. Cpu0 verilog source code. Chapter 12.

With these code, reader can generate Cpu0 machine code through Cpu0 llvm backend compiler, linker and elf2hex, then see how it runs on your computer. 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 and LLVMBuild files, 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. Around 3100 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. Around 345 lines of source code, mostly in .td Target Description files, are added. With these 345 lines, the backend can now translate the +, -, *, /, &, |, ^, <<, >>, ! and % C operators into the appropriate Cpu0 assembly code. Use 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. With 700 lines of additional code, the Cpu0 backend can now generate big and little endian ELF object files.

Global variables:

Global variable, struct and array support, char and short int, are added in this chapter. About 300 lines of source code are added to do this. The Cpu0 supports PIC and static addressing mode, both addressing mode explained as their functionality is implemented.

Other data type:

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

Control flow statements:

Support for the **if**, **else**, **while**, **for**, **goto**, **switch**, **case** flow control statements as well as both a simple optimization software pass and hardware instructions for control statement optimization discussed in this chapter. Around 500 lines of source code added.

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. Over 700 lines of source code are added.

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 llvm-objdump -d support for Cpu0 which translate elf into hex file format is added in the last section.

Assembler:

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

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, feed the hex file which generated by llvm-objdump to the CPU0 virtual machine and see 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.

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
 - DAG (Directed Acyclic Graph)
 - Instruction Selection
- 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

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 knowledgeable 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 ¹. 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.

¹ Chris Lattner, **LLVM**. Published in The Architecture of Open Source Applications. <http://www.aosabook.org/en/llvm.html>

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 ² (Chinese) and here ³ (English).

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 [Figure 2.1](#) below.

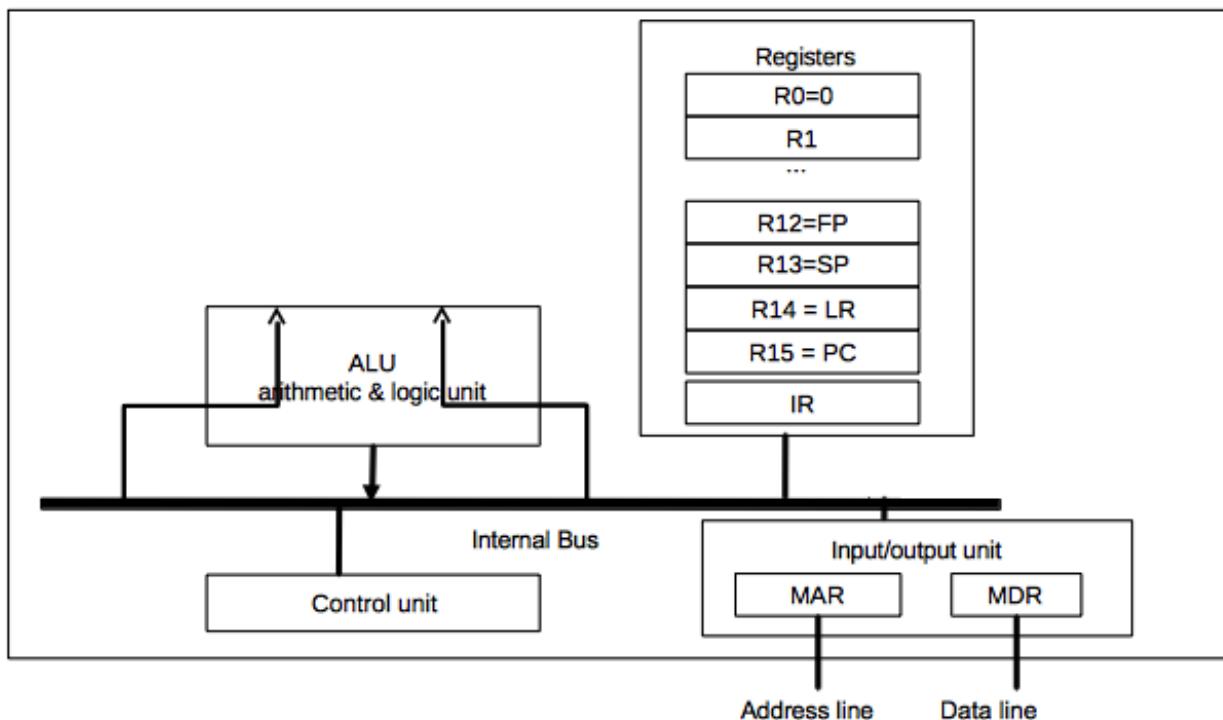


Figure 2.1: Architectural block diagram of the Cpu0 processor

The registers are used for the following purposes:

² Original Cpu0 architecture and ISA details (Chinese). <http://ccckmit.wikidot.com/ocs:cpu0>

³ 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

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)

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). Figure 2.2 illustrates how the bitfields are broken down for each type of instruction.

The Cpu0 has two ISA, the first ISA-I is cpu032I which hired CMP instruction from ARM; the second ISA-II is cpu032II which hired SLT instruction from Mips. The cpu032II include all cpu032I instruction set and add SLT, BEQ, ..., instructions. The main purpose to add cpu032II is for instruction set design explanation. As you will see in later chapter (chapter Control flow statements), the SLT instruction will have better performance than CMP old style instruction. The following table details the cpu032I instruction set:

- First column F.: meaning Format.

Table 2.4: cpu032I Instruction Set

F.	Mnemonic	Opcode	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
L	LB	03	Load byte	LB Ra, [Rb+Cx]	Ra <= (byte)[Rb+Cx] ⁴
L	LBu	04	Load byte unsigned	LBu Ra, [Rb+Cx]	Ra <= (byte)[Rb+Cx] ³
L	SB	05	Store byte	SB Ra, [Rb+Cx]	[Rb+Cx] <= (byte)Ra

Continued on next page

⁴ 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 0xfffffff80(= -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.

Table 2.4 – continued from previous page

F.	Mnemonic	Opcode	Meaning	Syntax	Operation
A	LH	06	Load half word	LH Ra, [Rb+Cx]	Ra <= (2bytes)[Rb+Cx] ³
A	LHu	07	Load half word unsigned	LHu Ra, [Rb+Cx]	Ra <= (2bytes)[Rb+Cx] ³
A	SH	08	Store half word	SH Ra, [Rb+Cx]	[Rb+Rc] <= 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	CMP	10	Compare	CMP Ra, Rb	SW <= (Ra cond Rb) ⁵
A	ADDu	11	Add unsigned	ADD Ra, Rb, Rc	Ra <= Rb + Rc ⁶
A	SUBu	12	Sub unsigned	SUB Ra, Rb, Rc	Ra <= Rb - Rc ⁴
A	ADD	13	Add	ADD Ra, Rb, Rc	Ra <= Rb + Rc ⁴
A	SUB	14	Subtract	SUB Ra, Rb, Rc	Ra <= Rb - Rc ⁴
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	ROL	1B	Rotate left	ROL Ra, Rb, Cx	Ra <= Rb rol Cx
A	ROR	1C	Rotate right	ROR Ra, Rb, Cx	Ra <= Rb ror Cx
A	SRA	1D	Shift right	SRA Ra, Rb, Cx	Ra <= Rb ' >> Cx ⁷
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	SRAV	20	Shift right	SRAV Ra, Rb, Rc	Ra <= Rb ' >> Rc ⁶
A	SHLV	21	Shift left	SHLV Ra, Rb, Rc	Ra <= Rb << Rc
A	SHRV	22	Shift right	SHRV Ra, Rb, Rc	Ra <= Rb >> Rc
A	ROL	23	Rotate left	ROL Ra, Rb, Rc	Ra <= Rb rol Rc
A	ROR	24	Rotate right	ROR Ra, Rb, Rc	Ra <= Rb ror Rc
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 ⁸
J	JSUB	3B	Jump to subroutine	JSUB Cx	LR <= PC; PC <= PC + Cx
J	RET	3C	Return from subroutine	RET LR	PC <= LR
L	MULT	41	Multiply for 64 bits result	MULT Ra, Rb	(HI,LO) <= MULT(Ra,Rb)
L	MULTU	42	MULT for unsigned 64 bits	MULTU Ra, Rb	(HI,LO) <= MULTU(Ra,Rb)
L	DIV	43	Divide	DIV Ra, Rb	HI<=Ra%Rb, LO<=Ra/Rb
L	DIVU	44	Divide unsigned	DIVU Ra, Rb	HI<=Ra%Rb, LO<=Ra/Rb
L	MFHI	46	Move HI to GPR	MFHI Ra	Ra <= HI
L	MFLO	47	Move LO to GPR	MFLO Ra	Ra <= LO
L	MTHI	48	Move GPR to HI	MTHI Ra	HI <= Ra

Continued on next page

⁵ 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.

⁶ 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.

⁷ Rb ' >> Cx, Rb ' >> Rc: Shift with signed bit remain. It's equal to ((Rb&'h80000000)|Rb>>Cx) or ((Rb&'h80000000)|Rb>>Rc).

⁸ jsub cx is direct call for 24 bits value of cx while jalr \$rb is indirect call for 32 bits value of register \$rb.

Table 2.4 – continued from previous page

F.	Mnemonic	Opcode	Meaning	Syntax	Operation
L	MTLO	49	Move GPR to LO	MTLO Ra	LO <= Ra
L	MFC0	50	Move C0R to GPR	MFC0 Ra, Rb	Ra <= Rb
L	MTC0	51	Move GPR to C0R	MTC0 Ra, Rb	Ra <= Rb
L	C0MOV	52	Move C0R to C0R	C0MOV Ra, Rb	Ra <= Rb

The following table details the cpu032II instruction set added:

Table 2.5: cpu032II Instruction Set

F.	Mnemonic	Opcode	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	Jump if equal	BEQ Ra, Rb, Cx	if (Ra==Rb), PC <= PC + Cx
L	BNE	38	Jump 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, 0xffffffff) 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.

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 representation 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 Figure 2.3 below.

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.

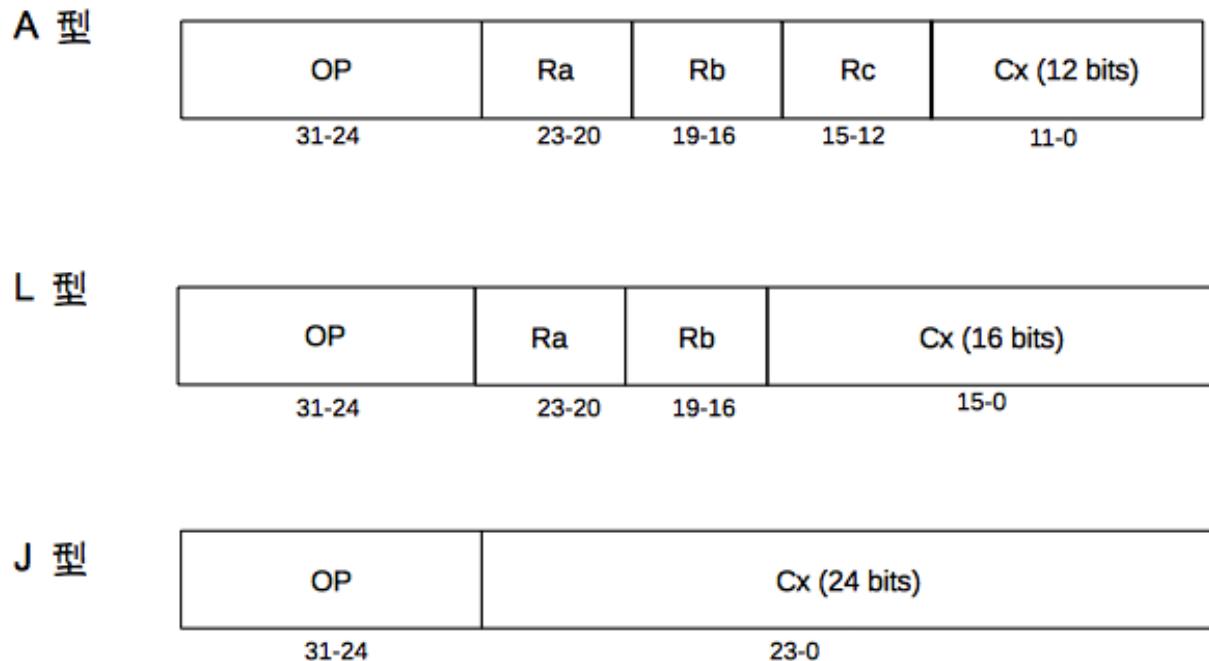


Figure 2.2: Cpu0's three instruction formats

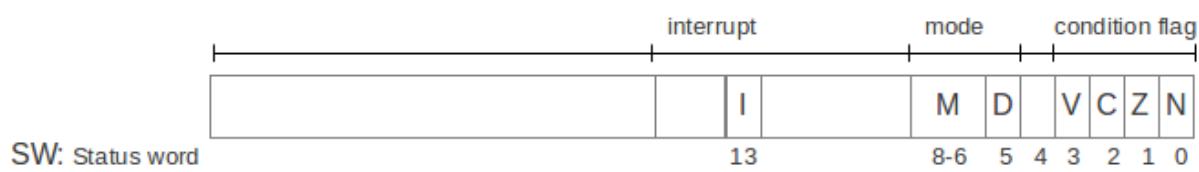


Figure 2.3: Cpu0 status word (SW) register

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 2.6: Cpu0's Interrupt Vector

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

2.2 LLVM Structure

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

2.2.1 Three-phase design

The text in this and the following sub-section comes from the AOSA chapter on LLVM written by Chris Lattner ⁸.

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 [Figure 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.

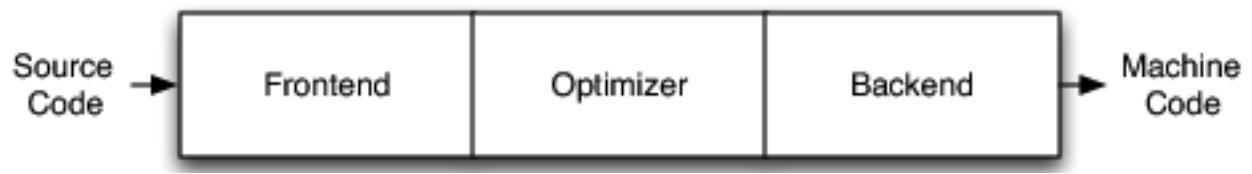


Figure 2.4: Three Major Components of a Three Phase Compiler

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 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 Figure 2.5.

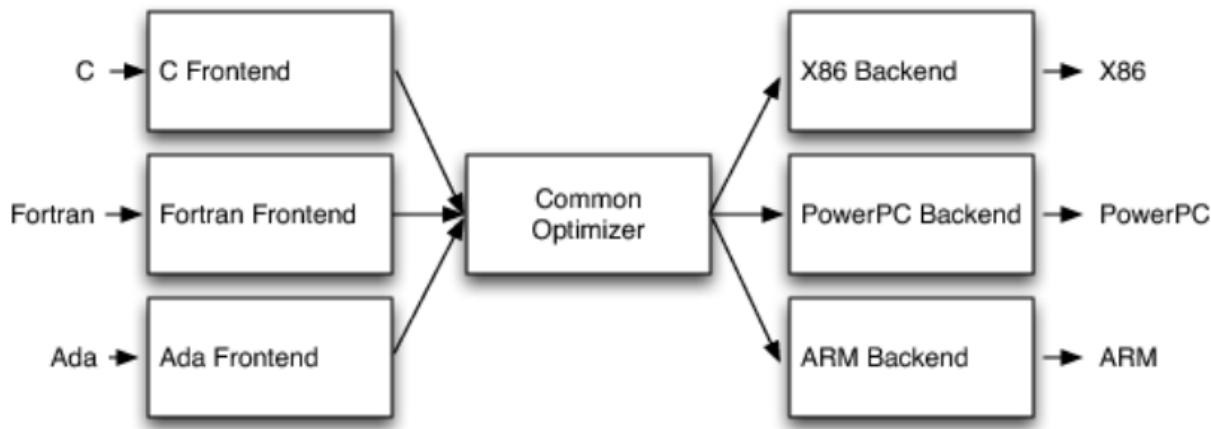


Figure 2.5: Retargetability

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 $N \times M$ 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
    %tmp3 = add i32 %b, 1
    %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
    ret i32 %tmp4
done:
    ret i32 %b
}
// This 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 Figure 2.6.

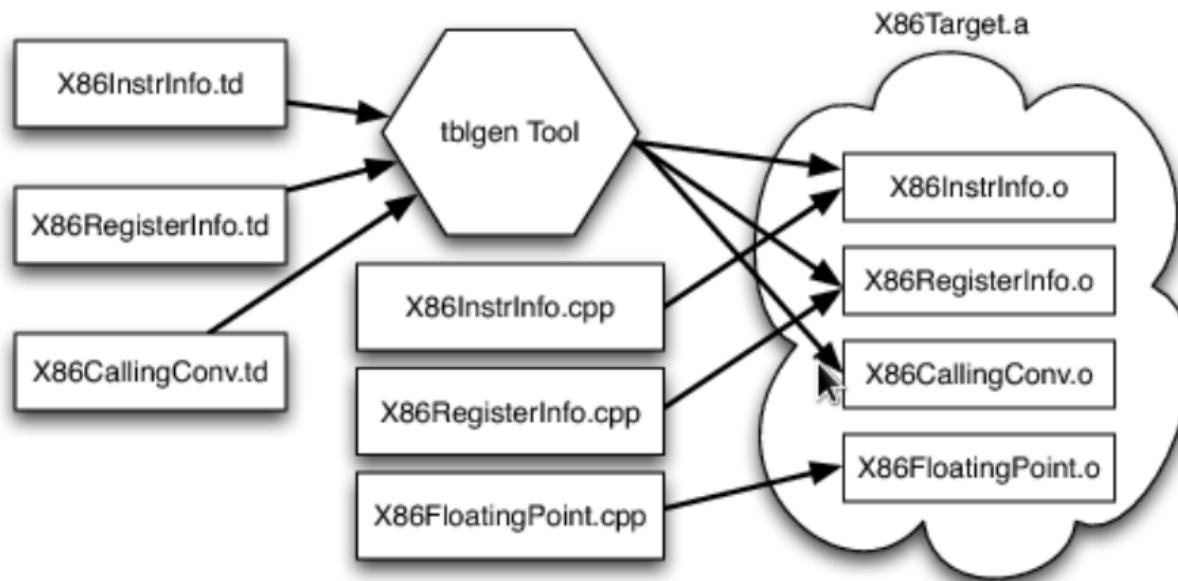


Figure 2.6: Simplified x86 Target Definition

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:

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

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 save 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 is i32 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
```

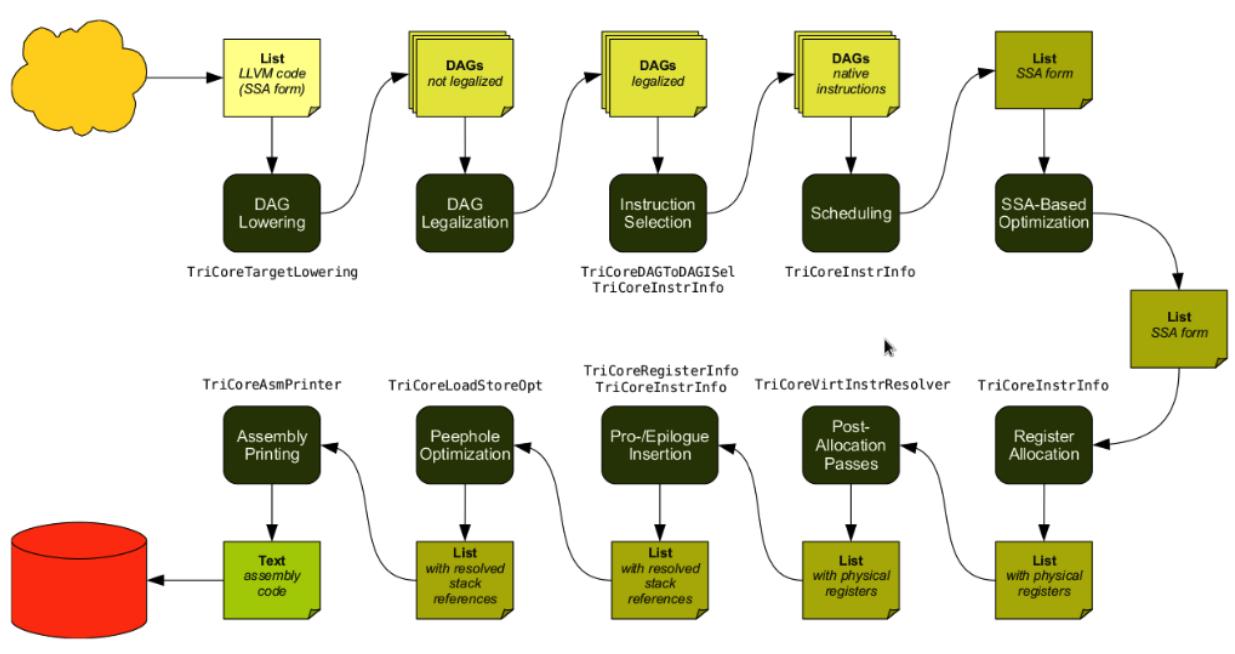


Figure 2.7: tricore_llvm.pdf: Code generation sequence. On the path from LLVM code to assembly code, numerous passes are run through and several data structures are used to represent the intermediate results.

```

; memory value to %a1
%a3 = add i32 %a2, 1 ; add %a2 and 1 and save to %a3

```

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 here⁹ and “LLVM Language Reference Manual” from here¹⁰ 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 2 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 use the result of the previous instruction st %c. So it must wait 1
// cycles more. Meaning the ld cannot follow st immediately.
=> st %b, i32* %c, i16 0

```

⁹ <http://llvm.org/docs/CodeGenerator.html>

¹⁰ <http://llvm.org/docs/LangRef.html>

```

st i32 %a, i16* %b, i16 5
%d = ld i32* %c, i16 0
// 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 need 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 llv code generation sequence also can be obtained by `llc -debug-pass=Structure` as the following. The first 4 code generation sequences from [Figure 2.7](#) 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 [Figure 2.7](#) 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>
```

```
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
  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
```

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 kept 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 use limited virtual registers to generate 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

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

Above code must run in sequence. On the other hand, the SSA form as the following can be reordered and run in

parallel with the following different version ¹¹.

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

2.2.5 DAG (Directed Acyclic Graph)

Many important techniques for local optimization begin by transforming a basic block into DAG ¹². For example, the basic block code and its corresponding DAG as Figure 2.8.

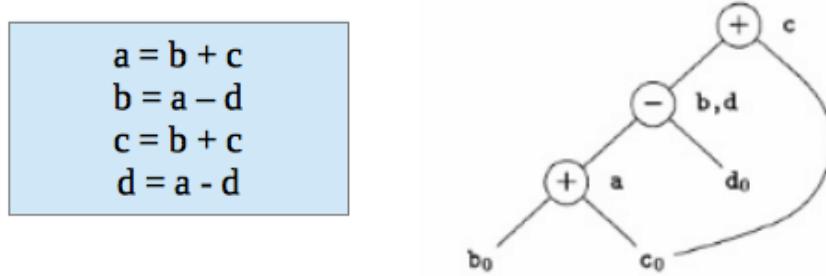


Figure 2.8: DAG example

If b is not live on exit from the block, then we can do “common expression remove” to get the following code.

```
a = b + c
d = a - d
c = d + c
```

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

DAG 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.

¹¹ Refer section 10.2.3 of book Compilers: Principles, Techniques, and Tools (2nd Edition)

¹² Refer section 8.5 of book Compilers: Principles, Techniques, and Tools (2nd Edition)

2.2.6 Instruction Selection

The major function of backend is translating IR code into machine code at stage of Instruction Selection as Figure 2.9.

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	$r_d = r_0 + c$

Figure 2.9: 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 Figure 2.10. The $r_j + rk$ is IR DAG representation (for symbol notation, not llvm SSA form). ADD is machine instruction.

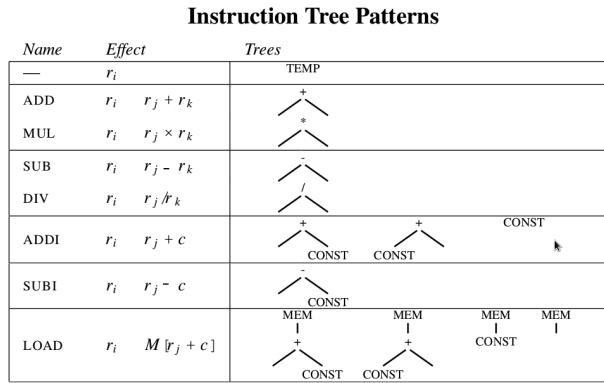


Figure 2.10: Instruction DAG representation

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

Now, let's recall the ADDiu instruction defined in Cpu0InstrInfo.td of the previous chapter. Listing them again as follows,

Ibdex/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;
}
    
```

Ibdex/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()); }]>;

$$\begin{aligned} &\text{// Arithmetic and logical instructions with 2 register operands.} \\ &\text{class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,} \\ &\quad \text{Operand Od, PatLeaf imm_type, RegisterClass RC>} : \\ &\quad \text{FL<op, (outs GPROut:$ra), (ins RC:$rb, Od:$imm16),} \\ &\quad \text{!strconcat(instr_asm, "\t$ra, $rb, $imm16"),} \\ &\quad \text{[(set GPROut:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu>} \\ &\quad \text{let isReMaterializable = 1;} \\ &\} \\ \\ &\text{// IR "add" defined in include/llvm/Target/TargetSelectionDAG.td, line 315 (def add).} \\ &\text{def ADDiu : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;} \end{aligned}$$

    
```

Figure 2.11 shows how the pattern match work in the IR node **add** and instruction **ADDiu** both defined in Cpu0InstrInfo.td. In this example, IR node “add %a, 5” will be translated to “addiu \$r1, 5” after %a is allocated to register \$r1 in register 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 issued in obj file of ELF format which will be explained at later chapter). Similarly, the machine instruction DAG node LD and ST can be translated from IR DAG node **load** and **store**. Notice that the \$r1 in this case is virtual register name (not machine register).

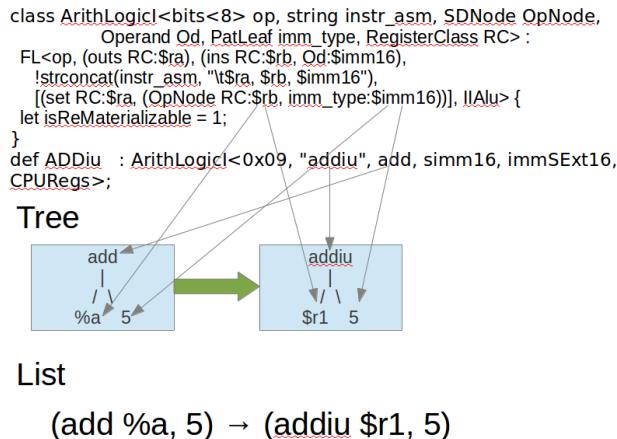


Figure 2.11: Pattern match for ADDiu instruction and IR node add

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.

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 Figure 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;
```

2.3 Create Cpu0 backend

From now on, the Cpu0 backend will be created from scratch step by step and chapter by chapter. To make readers easily understanding the backend structure step by step, Cpu0 example code can be generated with chapter by chapter through command here ¹³.

¹³ <http://jonathan2251.github.io/lbd/install.html#cpu0-document>

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,

Ibdex/src/modify/src/config-ix.cmake

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

```

Ibdex/src/modify/src/CMakeLists.txt

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

Ibdex/src/modify/src/include/llvm/ADT/Triple.h

```
...
#undef mips
#undef cpu0
...
class Triple {
public:
    enum ArchType {
        ...
        mips,           // MIPS: mips, mipsallegrex
        mipsel,         // MIPSEL: mipsel, mipsallegrexel
        mips64,         // MIPS64: mips64
        mips64el,       // MIPS64EL: mips64el
        cpu0,           // For Tutorial Backend Cpu0
        cpu0el,
        ...
    };
    ...
}
```

Ibdex/src/modify/src/include/llvm/MC/MCE Expr.h

```
class MCSymbolRefExpr : public MCE Expr {
public:
    enum VariantKind {
        ...
        VK_Cpu0_GPREL,
```

```

VK_Cpu0_GOT_CALL,
VK_Cpu0_GOT16,
VK_Cpu0_GOT,
VK_Cpu0_ABS_HI,
VK_Cpu0_ABS_LO,
VK_Cpu0_TLSDG,
VK_Cpu0_TLSDM,
VK_Cpu0_DTP_HI,
VK_Cpu0_DTP_LO,
VK_Cpu0_GOTTPREL,
VK_Cpu0_TP_HI,
VK_Cpu0_TP_LO,
VK_Cpu0_GPOFF_HI,
VK_Cpu0_GPOFF_LO,
VK_Cpu0_GOT_DISP,
VK_Cpu0_GOT_PAGE,
VK_Cpu0_GOT_OFST,
VK_Cpu0_HIGHER,
VK_Cpu0_HIGHEST,
VK_Cpu0_GOT_HI16,
VK_Cpu0_GOT_LO16,
VK_Cpu0_CALL_HI16,
VK_Cpu0_CALL_LO16,
...
};

...
};


```

[libdex/src/modify/src/include/llvm/Object/ELFObjectFile.h](#)

```

template <class ELFT>
std::error_code ELFObjectFile<ELFT>::getRelocationValueString(
    DataRefImpl Rel, SmallVectorImpl<char> &Result) const {
    ...
    switch (EF.getHeader()->e_machine) {
    ...
    case ELF::EM_MIPS:
    case ELF::EM_CPU0: // llvm-objdump -t -r
    case ELF::EM_CPU0_LE:
        res = *SymName;
        break;
    ...
    }
    ...
}

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
        case ELF::EM_CPU0_LE:
            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
        case ELF::EM_CPU0_LE:
            switch (EF.getHeader()->e_ident[ELF::EI_CLASS]) {
                case ELF::ELFCLASS32:
                    return IsLittleEndian ? Triple::cpu0el : Triple::cpu0;
                default:
                    report_fatal_error("Invalid ELFCLASS!");
            }
        ...
    }
}

```

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

```

enum {
    EM_NONE      = 0, // No machine
    EM_M32       = 1, // AT&T WE 32100
    ...
    EM_CPU0      = 998, // Document LLVM Backend Tutorial Cpu0
    EM_CPU0_LE   = 999 // EM_CPU0_LE: little endian; EM_CPU0: big endian
};

...
// Cpu0 Specific e_flags
enum {
    EF_CPU0_NOREORDER = 0x00000001, // Don't reorder instructions
    EF_CPU0_PIC      = 0x00000002, // Position independent code
    EF_CPU0_CPIC     = 0x00000004, // Call object with Position independent code
    EF_CPU0_ARCH_1   = 0x00000000, // CPU01 instruction set
    EF_CPU0_ARCH_2   = 0x10000000, // CPU02 instruction set
    EF_CPU0_ARCH_3   = 0x20000000, // CPU03 instruction set
    EF_CPU0_ARCH_4   = 0x30000000, // CPU04 instruction set
    EF_CPU0_ARCH_5   = 0x40000000, // CPU05 instruction set
    EF_CPU0_ARCH_32  = 0x50000000, // CPU032 instruction set per linux not elf.h
    EF_CPU0_ARCH_64  = 0x60000000, // CPU064 instruction set per linux not elf.h
    EF_CPU0_ARCH_32R2 = 0x70000000, // cpu032r2
    EF_CPU0_ARCH_64R2 = 0x80000000, // cpu064r2
    EF_CPU0_ARCH     = 0xf0000000 // Mask for applying EF_CPU0_ARCH_ variant
};

// ELF Relocation types for Cpu0
// .
enum {
    R_CPU0_NONE      = 0,
    R_CPU0_32        = 2,
    R_CPU0_HI16      = 5,
    R_CPU0_LO16      = 6,

```

```
R_CPU0_GPREL16      = 7,
R_CPU0_LITERAL      = 8,
R_CPU0_GOT16        = 9,
R_CPU0_PC16         = 10,
R_CPU0_CALL16       = 11,
R_CPU0_GPREL32       = 12,
R_CPU0_PC24         = 13,
R_CPU0_GOT_HI16     = 22,
R_CPU0_GOT_LO16     = 23,
R_CPU0_RELGOT       = 36,
R_CPU0_TLS_GD        = 42,
R_CPU0_TLS_LDM       = 43,
R_CPU0_TLS_DTP_HI16  = 44,
R_CPU0_TLS_DTP_LO16  = 45,
R_CPU0_TLS_GOTTPREL  = 46,
R_CPU0_TLS_TPREL32    = 47,
R_CPU0_TLS_TP_HI16    = 49,
R_CPU0_TLS_TP_LO16    = 50,
R_CPU0_GLOB_DAT      = 51,
R_CPU0_JUMP_SLOT     = 127
};


```

Ibdex/src/modify/src/lib/MC/MCELFStreamer.cpp

```
void MCELFStreamer::fixSymbolsInTLSFixups(const MCExpr *expr) {
    switch (expr->getKind()) {
    ...
    case MCExpr::SymbolRef: {
        const MCSymbolRefExpr &symRef = *cast<MCSymbolRefExpr>(expr);
        switch (symRef.getKind()) {
        ...
        case MCSymbolRefExpr::VK_Mips_TLSGD:
        case MCSymbolRefExpr::VK_Mips_GOTTPREL:
        case MCSymbolRefExpr::VK_Mips_TPREL_HI:
        case MCSymbolRefExpr::VK_Mips_TPREL_LO:
        case MCSymbolRefExpr::VK_Cpu0_TLSGD:
        case MCSymbolRefExpr::VK_Cpu0_GOTTPREL:
        case MCSymbolRefExpr::VK_Cpu0_TP_HI:
        case MCSymbolRefExpr::VK_Cpu0_TP_LO:
        ...
        break;
    }
    ...
}
```

Ibdex/src/modify/src/lib/MC/MCExpr.cpp

```
StringRef MCSymbolRefExpr::getVariantKindName(VariantKind Kind) {
    switch (Kind) {
    ...
    case VK_Cpu0_GPREL: return "GPREL";
    case VK_Cpu0_GOT_CALL: return "GOT_CALL";
    case VK_Cpu0_GOT16: return "GOT16";
    case VK_Cpu0_GOT: return "GOT";
    case VK_Cpu0_ABS_HI: return "ABS_HI";
```

```

    case VK_Cpu0_ABS_LO: return "ABS_LO";
    case VK_Cpu0_TLSGD: return "TLSGD";
    case VK_Cpu0_TLSLDM: return "TLSLDM";
    case VK_Cpu0_DTP_HI: return "DTP_HI";
    case VK_Cpu0_DTP_LO: return "DTP_LO";
    case VK_Cpu0_GOTTPREL: return "GOTTPREL";
    case VK_Cpu0_TP_HI: return "TP_HI";
    case VK_Cpu0_TP_LO: return "TP_LO";
    case VK_Cpu0_GPOFF_HI: return "GPOFF_HI";
    case VK_Cpu0_GPOFF_LO: return "GPOFF_LO";
    case VK_Cpu0_GOT_DISP: return "GOT_DISP";
    case VK_Cpu0_GOT_PAGE: return "GOT_PAGE";
    case VK_Cpu0_GOT_OFST: return "GOT_OFST";
    case VK_Cpu0_HIGHER: return "HIGHER";
    case VK_Cpu0_HIGHEST: return "HIGHEST";
    case VK_Cpu0_GOT_HI16: return "GOT_HI16";
    case VK_Cpu0_GOT_LO16: return "GOT_LO16";
    case VK_Cpu0_CALL_HI16: return "CALL_HI16";
    case VK_Cpu0_CALL_LO16: return "CALL_LO16";
    ...
}
}
}

```

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)
```

Ibdex/src/modify/src/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 unsigned getArchPointerBitWidth(llvm::Triple::ArchType Arch) {
    switch (Arch) {
    ...
    case llvm::Triple::cpu0:
    case llvm::Triple::cpu0el:
```

```

...
    return 32;
}
}

...
Triple Triple::get32BitArchVariant() const {
    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 use target description files (which use 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 ¹⁴ for more information regarding how to use tablegen.

Every backend has a .td which defines some target information, including what other .td files are used by the backend. These files have a similar syntax to C++. For Cpu0, the target description file is called Cpu0Other.td, which is shown below:

```

118-165-12-177:BackendTutorial Jonathan$ pwd
/home/cschen/test/lbd/docs/BackendTutorial
118-165-12-177:BackendTutorial Jonathan$ make genexample

```

Index/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"

//=====-----=====//

```

¹⁴ <http://llvm.org/docs/TableGen/index.html>

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

```
include "Cpu0RegisterInfo.td"
include "Cpu0RegisterInfoGPROutForOther.td" // except AsmParser
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 registers have been given names, i.e. “**def PC**” indicates that there is a register called PC. Also, there is a register class named “**CPURegs**” that contains all of the other registers. You may have multiple register classes such as CPURegs, SR, C0Regs and GPROut. GPROut defined in Cpu0RegisterInfoGPROutForOther.td which include CPURegs except SW, PC and EPC, so SW, PC and EPC won't be allocated as the output registers in register allocation stage.

Index/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<string n> : Register<n> {
    field bits<4> Num;
    let Namespace = "Cpu0";
}

// General Purpose Registers
class Cpu0GPRReg<bits<4> num, string n> : Cpu0Reg<n> {
    let Num = num;
}

// Co-processor 0 Registers
class Cpu0C0Reg<bits<4> num, string n> : Cpu0Reg<n> {
    let Num = num;
}

//=====
```

```
//@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]>;
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, PC, EPC)>

//@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)>;

```

Ibdex/chapters/Chapter2/Cpu0RegisterInfoGPROutForOther.td

```

=====//
// Register Classes
=====//

def GPROut : RegisterClass<"Cpu0", [i32], 32, (add (sub CPUREgs, SW, PC, EPC))>;

```

In C++, class typically provide 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**, however 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 describe the structure of how data is laid out, while definitions act as the specific instances of the class. If we look back at the Cpu0RegisterInfo.td file, we see a class called **Cpu0Reg<string n>** which is derived from the **Register<n>** class provided by LLVM. **Cpu0Reg** inherits all the fields that exist in the **Register** class, and also adds a new field called **Num** which is four bits wide.

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 four bit **Num** field 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 C++ code in a backend 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,

Ibdex/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  
//=====//  
  
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<i32> {
    let PrintMethod = "printMemOperand";
    let MIOperandInfo = (ops CPURegs, 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 possibly 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();
    }}];

// Load/Store PatFrgs.
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:$simm16),
    !strconcat(instr_asm, "\t$ra, $rb, $simm16"),
    [(set GPROut:$ra, (OpNode RC:$rb, imm_type:$simm16))], 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.
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>;
}

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;
}

// 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  
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>;  
  
/// Arithmetic Instructions (3-Operand, R-Type)  
  
/// Shift Instructions  
  
def RET     : RetBase<GPROut>;  
  
/// No operation  
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,

Ibdex/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

// Generic Cpu0 Format
class Cpu0Inst<dag outs, dag ins, string asmstr, list<dag> pattern,
              InstrItinClass itin, Format f>: Instruction
{
    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;

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 which inherited from FL, and 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> {
    let isReMaterializable = 1;

```

}

So,

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

To expand the td, one principle is:

- 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 2.7: ADDiu expand part I

ADDiu	ArithLogicl	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);
imm-SExt16	imm_type = immSExt16	Inst{15-0} = imm16;
CPURegs	RC = CPURegs isReMaterializable=1;	Inst{23-20} = ra; Inst{19-16} = rb;

Table 2.8: ADDiu expand part II

Cpu0Inst	instruction
Namespace = "Cpu0"	Uses = []; ...
Inst{31-24} = 0x09;	Size = 0; ...
OutOperandList = GPROut:\$ra;	
InOperandList = CPURegs:\$rb,simm16:\$imm16;	
AsmString = "addiu\$tra, \$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,

Ibdex/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 II_Load    : InstrItinClass;
def II_Store   : InstrItinClass;
def II_Branch  : 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<II_Load>    , [InstrStage<3, [ALU]>],
  InstrItinData<II_Store>   , [InstrStage<1, [ALU]>],
  InstrItinData<II_Branch>  , [InstrStage<1, [ALU]>]
] >;
```

2.3.3 Write cmake file

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

Ibdex/chapters/Chapter2/CMakeLists.txt

```
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
)

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

Index/chapters/Chapter2/LLVMBuild.txt

```
;===== ./lib/Target/Cpu0/LLVMBuild.txt -----*-- Conf -*----;;
;
; The LLVM Compiler Infrastructure
;
; This file is distributed under the University of Illinois Open Source
; License. See LICENSE.TXT for details.
;
;=====;
;
; This is an LLVMBuild description file for the components in this subdirectory.
;
; For more information on the LLVMBuild system, please see:
;
;   http://llvm.org/docs/LLVMBuild.html
;
;=====;

# Following comments extracted from http://llvm.org/docs/LLVMBuild.html

[common]
subdirectories =
    MCTargetDesc TargetInfo

[component_0]
# TargetGroup components are an extension of LibraryGroups, specifically for
# defining LLVM targets (which are handled specially in a few places).
type = TargetGroup
# The name of the component should always be the name of the target. (should
# match "def Cpu0 : Target" in Cpu0.td)
name = Cpu0
# Cpu0 component is located in directory Target/
parent = Target
# Whether this target defines an assembly parser, assembly printer, disassembler
# , and supports JIT compilation. They are optional.

[component_1]
# component_1 is a Library type and name is Cpu0CodeGen. After build it will
```

```

# in lib/libLLVMCpu0CodeGen.a of your build command directory.
type = Library
name = Cpu0CodeGen
# Cpu0CodeGen component(Library) is located in directory Cpu0/
parent = Cpu0
# If given, a list of the names of Library or LibraryGroup components which
# must also be linked in whenever this library is used. That is, the link time
# dependencies for this component. When tools are built, the build system will
# include the transitive closure of all required_libraries for the components
# the tool needs.
required_libraries =
    CodeGen Core MC
    Cpu0Desc
    Cpu0Info
    SelectionDAG
    Support
    Target
# end of required_libraries

# All LLVMBuild.txt in Target/Cpu0 and subdirectory use 'add_to_library_groups'
# = Cpu0'
add_to_library_groups = Cpu0

```

CMakeLists.txt is the make information for cmake and # is comment. File LLVMBuild.txt is written in a simple variant of the INI or configuration file format. Comments are prefixed by # in both files. We explain the setting for these two files in comments. Please read it. This book breaks the whole backend source code by function, add code chapter by chapter and even section by section. 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 implementation 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.

Both CMakeLists.txt and LLVMBuild.txt coexist in sub-directories **MCTargetDesc** and **TargetInfo**. Their contents indicate they will generate Cpu0Desc and Cpu0Info libraries. After building, you will find three libraries: **libLLVM-Cpu0CodeGen.a**, **libLLVMCpu0Desc.a** and **libLLVMCpu0Info.a** in lib/ of your build directory. For more details please see “Building LLVM with CMake”¹⁵ and “LLVMBuild Guide”¹⁶.

2.3.4 Target Registration

You must also register your target with the TargetRegistry, which is what other LLVM tools use to be 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.

Ibdex/chapters/Chapter2/Cpu0.h

```

//===== Cpu0.h - Top-level interface for Cpu0 representation ----- C++ -*====//
//
//                                     The LLVM Compiler Infrastructure

```

¹⁵ <http://llvm.org/docs/CMake.html>

¹⁶ <http://llvm.org/docs/LLVMBuild.html>

```
//  
// 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 TARGET_CPU0_H  
#define TARGET_CPU0_H  
  
#include "Cpu0Config.h"  
#include "MCTargetDesc/Cpu0MCTargetDesc.h"  
#include "llvm/Target/TargetMachine.h"  
  
namespace llvm {  
    class Cpu0TargetMachine;  
    class FunctionPass;  
  
    FunctionPass *createCpu0ISelDag(Cpu0TargetMachine &TM);  
}  
// end namespace llvm;  
  
#endif
```

Index/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");  
  
    RegisterTarget<Triple::cpu0el,  
        /*HasJIT=*/true> Y(TheCpu0elTarget, "cpu0el", "Cpu0el");  
}
```

Ibdex/chapters/Chapter2/TargetInfo/CMakeLists.txt

```
add_llvm_library(LLVMCpu0Info
  Cpu0TargetInfo.cpp
)
```

Ibdex/chapters/Chapter2/TargetInfo/LLVMBuild.txt

```
;===== ./lib/Target/Cpu0/TargetInfo/LLVMBuild.txt -----*-- Conf -*----=;
;
;           The LLVM Compiler Infrastructure
;
; This file is distributed under the University of Illinois Open Source
; License. See LICENSE.TXT for details.
;
;=====-----=;
;
; This is an LLVMBuild description file for the components in this subdirectory.
;
; For more information on the LLVMBuild system, please see:
;
;   http://llvm.org/docs/LLVMBuild.html
;
;=====-----=;

[component_0]
type = Library
name = Cpu0Info
parent = Cpu0
required_libraries = MC Support Target
add_to_library_groups = Cpu0
```

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

Ibdex/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/LegacyPassManager.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/Support/TargetRegistry.h"
```

```
using namespace llvm;

#define DEBUG_TYPE "cpu0"

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

Ibdex/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 CPU0MCTARGETDESC_H
#define CPU0MCTARGETDESC_H

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

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
```

Ibdex/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/MCCodeGenInfo.h"
#include "llvm/MC/MCELFStreamer.h"
#include "llvm/MC/MCInstPrinter.h"
#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 }
```

Ibdex/chapters/Chapter2/MCTargetDesc/CMakeLists.txt

```
# MCTargetDesc/CMakeLists.txt
add_llvm_library(LLVMCpu0Desc
    Cpu0MCTargetDesc.cpp
)
```

Ibdex/chapters/Chapter2/MCTargetDesc/LLVMBuild.txt

```
;===== ./lib/Target/Cpu0/MCTargetDesc/LLVMBuild.txt -----*-- Conf -*---=;
;
;          The LLVM Compiler Infrastructure
;
```

```
; This file is distributed under the University of Illinois Open Source
; License. See LICENSE.TXT for details.
;
;=====;
;
; This is an LLVMBuild description file for the components in this subdirectory.
;
; For more information on the LLVMBuild system, please see:
;
;   http://llvm.org/docs/LLVMBuild.html
;
;=====;
# MCTargetDesc/LLVMBuild.txt
[component_0]
type = Library
name = Cpu0Desc
parent = Cpu0
required_libraries = MC
          Cpu0Info
          Support
add_to_library_groups = Cpu0
```

Please see “Target Registration”¹⁷ for reference.

2.3.5 Build libraries and td

We set llvm source code in /Users/Jonathan/llvm/release/src and have llvm release-build in /Users/Jonathan/llvm/release/cmake_release_build. About how to build llvm, please refer here¹⁸. In appendix A, we made a copy from /Users/Jonathan/llvm/release/src to /Users/Jonathan/llvm/test/src for working with my Cpu0 target backend. Sub-directories src is for source code and cmake_debug_build is for debug build directory.

Beside directory src/lib/Target/Cpu0, there are a couple of files modified to support cpu0 new Target. It include 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/src/modify/src/* <yourllvm/workingcopy/sourcedir>/.

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

Next configure the Cpu0 example code to chapter2 as follows,

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

```
#define CH          CH2
```

¹⁷ <http://llvm.org/docs/WritingAnLLVMBackend.html#target-registration>

¹⁸ http://clang.llvm.org/get_started.html

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

```
118-165-78-230:cmake_debug_build Jonathan$ cmake -DCMAKE_CXX_COMPILER=clang++  
-DCMAKE_C_COMPILER=clang -DCMAKE_BUILD_TYPE=Debug -G "Xcode" ../src/  
  
-- Targeting Cpu0  
...  
-- Targeting XCore  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /Users/Jonathan/llvm/test/cmake_debug_build  
  
118-165-78-230:cmake_debug_build Jonathan$
```

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

```
118-165-78-230:cmake_debug_build Jonathan$ /Users/Jonathan/llvm/test/  
cmake_debug_build/Debug/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 Target-Info/Cpu0TargetInfo.cpp we made in last sub-section “Target Registration” ¹⁹.

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

To save time, we build Cpu0 target only by option -DLLVM_TARGETS_TO_BUILD=Cpu0. After cmake please open Xcode and build the Xcode project file as appendix A, or refer appendix A to build it on linux if you work on unix/linux platform. After that, you can find the *.inc files in directory /Users/Jonathan/llvm/test/cmake_debug_build/lib/Target/Cpu0 as follows,

¹⁹ <http://jonathan2251.github.io/lbd/llvmstructure.html#target-registration>

cmake_debug_build/lib/Target/Cpu0/Cpu0GenRegisterInfo.inc

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

These *.inc are created by llvm-tblgen from directory cmake_debug_build/bin where input files are the Cpu0 backend *.td files. The llvm-tblgen is invoked by **tablegen** of /Users/Jonathan/llvm/test/src/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 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 ¹⁹ ²⁰ ²¹.

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

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/llvm/test/src/lib/Target/Cpu0/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,

²⁰ <http://llvm.org/docs/TableGen/LangIntro.html>

²¹ <http://llvm.org/docs/TableGen/LangRef.html>

```
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/cmake_debug_build/  
Debug/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 development. You can review the hundreds lines of Chapter2 example code to see how to do the Target Registration.

BACKEND STRUCTURE

- TargetMachine structure
- Add AsmPrinter
- Add Cpu0DAGToDAGISel class
- Handle return register lr
- Add Prologue/Epilogue functions
- Data operands DAGs
- Summary of this Chapter

This chapter introduces the back end class inheritance tree and class members first. Next, following the back end structure, adding individual class implementation in each section. At the end of this chapter, we will have a back end to compile llvm intermediate code into cpu0 assembly code.

Many code are added in this chapter. They almost are common in every back end except the back end 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 3100 lines of code in this chapter.

3.1 TargetMachine structure

[Index/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.  
//  
//=====-----=====//  
  
#ifndef LLVM_TARGET_CPU0_TARGETOBJECTFILE_H  
#define LLVM_TARGET_CPU0_TARGETOBJECTFILE_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

```

Ibdex/chapters/Chapter3_1/Cpu0TargetObjectFile.cpp

```

//===== Cpu0TargetObjectFile.cpp - Cpu0 Object Files -----
//  

//  

//  

// 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/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"
#include "llvm/Support/ELF.h"
#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);
}

```

Ibdex/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 CPU0TARGETMACHINE_H
#define CPU0TARGETMACHINE_H
```

```
#include "Cpu0Config.h"

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

```
namespace llvm {
class formatted_raw_ostream;
class Cpu0RegisterInfo;
```

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

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

    const Cpu0Subtarget *getSubtargetImpl() const {
        if (Subtarget)
            return Subtarget;
        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,
                        Reloc::Model RM, CodeModel::Model CM,
                        CodeGenOpt::Level OL);
};

/// 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,
                        Reloc::Model RM, CodeModel::Model CM,
                        CodeGenOpt::Level OL);
};
} // End llvm namespace

#endif

```

Index/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/LegacyPassManager.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/Support/TargetRegistry.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,
                                    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;
}

// 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,
                  Reloc::Model RM, CodeModel::Model CM,
                  CodeGenOpt::Level OL,
                  bool isLittle)
    // Default is big endian
    : LLVMTargetMachine(T, computeDataLayout(TT, CPU, Options, isLittle), TT,
                        CPU, FS, Options, RM, CM, OL),
      isLittle(isLittle), TLOF(make_unique<Cpu0TargetObjectFile>()),
      ABI(Cpu0ABIInfo::computeTargetABI()),
      Subtarget(nullptr), DefaultSubtarget(TT, CPU, FS, isLittle, *this) {
    Subtarget = &DefaultSubtarget;
    // 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,
                                        Reloc::Model RM, CodeModel::Model CM,
                                        CodeGenOpt::Level OL)
: Cpu0TargetMachine(T, TT, CPU, FS, Options, RM, CM, OL, false) {}

void Cpu0elTargetMachine::anchor() { }

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

const Cpu0Subtarget *
Cpu0TargetMachine::getSubtargetImpl(const Function &F) const {
    Attribute CPUAttr = F.getFnAttribute("target-cpu");
    Attribute FSAttr = F.getFnAttribute("target-features");

    std::string CPU = !CPUAttr.hasAttribute(Attribute::None)
                      ? CPUAttr.getValueAsString().str()
                      : TargetCPU;
    std::string FS = !FSAttr.hasAttribute(Attribute::None)
                   ? FSAttr.getValueAsString().str()
                   : 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 = llvm::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) {}

    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/TargetInstrInfo.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:
    ...
}

```

Index/chapters/Chapter3_1/Cpu0.td

```

include "Cpu0CallingConv.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",
                      "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",
                        [FeatureCmp, FeatureChapterAll]>;
def FeatureCpu032II  : SubtargetFeature<"cpu032II", "Cpu0ArchVersion",
                        "Cpu032II", "Cpu032II ISA Support (slt)",
                        [FeatureCmp, FeatureSlt, FeatureChapterAll]>;

//=====//
// Cpu0 processors supported.
//=====//

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
// };
```

Ibdex/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)>;
```

Ibdex/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 CPU0_FRAMEINFO_H
#define CPU0_FRAMEINFO_H

#include "Cpu0Config.h"

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

namespace llvm {
    class Cpu0Subtarget;

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

public:
```

```

explicit Cpu0FrameLowering(const Cpu0Subtarget &sti, unsigned Alignment)
    : TargetFrameLowering(StackGrowsDown, Alignment, 0, 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

```

Index/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 "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;

// - emitPrologue() 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
// .
//           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);
}

// Must have, hasFP() is pure virtual of parent
// hasFP - Return true if the specified function should have a dedicated frame
// pointer register. This is true if the function has variable sized allocas or
// if frame pointer elimination is disabled.
bool Cpu0FrameLowering::hasFP(const MachineFunction &MF) const {
    const MachineFrameInfo *MFI = MF.getFrameInfo();
    return MF.getTarget().Options.DisableFramePointerElim(MF) ||
        MFI->hasVarSizedObjects() || MFI->isFrameAddressTaken();

```

}

Ibdex/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 CPU0SE_FRAMEINFO_H
#define CPU0SE_FRAMEINFO_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
```

Ibdex/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.
//
//=====
//=====
// 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);
}

```

Ibdex/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 CPU0INSTRUCTIONINFO_H
#define CPU0INSTRUCTIONINFO_H

```

```
#include "Cpu0Config.h"

#include "Cpu0.h"
#include "Cpu0RegisterInfo.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/Target/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;

};

const Cpu0InstrInfo *createCpu0SEInstrInfo(const Cpu0Subtarget &STI);
}

#endif
```

Index/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_INSTRINFOCTOR_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);
}

```

[Index/chapters/Chapter3_1/Cpu0InstrInfo.td](#)

```

//=====
// Cpu0 Instruction Predicate Definitions.
=====

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

```

```

        AssemblerPredicate<"FeatureChapter11_1">;
def Ch11_2      : Predicate<"Subtarget->hasChapter11_2()">,
                  AssemblerPredicate<"FeatureChapter11_2">;
def Ch12_1      : Predicate<"Subtarget->hasChapter12_1()">,
                  AssemblerPredicate<"FeatureChapter12_1">;
def Ch_all       : Predicate<"Subtarget->hasChapterAll()">,
                  AssemblerPredicate<"FeatureChapterAll">;

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

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

```

Index/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 Cpu0ISELLOWERING_H
#define 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/Target/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
        };
    }
}

```

```

// 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,

// 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.
        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:

#if 0
    // Create a TargetConstantPool node.
    SDValue getTargetNode(ConstantPoolSDNode *N, EVT Ty, SelectionDAG &DAG,
                          unsigned Flag) const;
#endif

    // Lower Operand specifics
    SDValue lowerGlobalAddress(SDValue Op, SelectionDAG &DAG) const;
    SDValue lowerJumpTable(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,
                         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,
                        SDLoc dl, SelectionDAG &DAG) const override;

};

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

#endif // Cpu0ISELLOWERING_H

```

Ibdex/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.
//
//=====
// 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 {
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::DivRem:            return "Cpu0ISD::DivRem";
        case Cpu0ISD::DivRemU:           return "Cpu0ISD::DivRemU";
        case Cpu0ISD::Wrapper:           return "Cpu0ISD::Wrapper";
        default:                         return NULL;
    }
}
//@3_1 }

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

}

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,
                                         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,
                                 SDLoc DL, SelectionDAG &DAG) const {
    return DAG.getNode(Cpu0ISD::Ret, DL, MVT::Other,
                       Chain, DAG.getRegister(Cpu0::LR, MVT::i32));
}
```

[Index/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 CPU0SEISELLWERING_H
#define CPU0SEISELLWERING_H

#include "Cpu0Config.h"
```

```

#include "Cpu0ISEISelLowering.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 // Cpu0ISEISELLWERING_H

```

Index/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.
//
//=====
#include "Cpu0MachineFunction.h"
#include "Cpu0SEISelLowering.h"

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

```

[Index/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 CPU0_MACHINE_FUNCTION_INFO_H
#define CPU0_MACHINE_FUNCTION_INFO_H

#include "Cpu0Config.h"

#include "llvm/ADT/StringMap.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineMemOperand.h"
#include "llvm/CodeGen/PseudoSourceValue.h"
#include "llvm/IR/GlobalValue.h"
#include "llvm/IR/ValueMap.h"
#include "llvm/Target/TargetFrameLowering.h"
#include "llvm/Target/TargetMachine.h"
#include <map>
#include <string>
#include <utility>

namespace llvm {

/// \brief A class derived from PseudoSourceValue that represents a GOT entry

```

```

/// resolved by lazy-binding.
class Cpu0CallEntry : public PseudoSourceValue {
public:
    explicit Cpu0CallEntry(const StringRef &N);
    explicit Cpu0CallEntry(const GlobalValue *V);
    bool isConstant(const MachineFrameInfo *) const override;
    bool isAliased(const MachineFrameInfo *) const override;
    bool mayAlias(const MachineFrameInfo *) const override;

private:
    void printCustom(raw_ostream &O) const override;
#ifndef NDEBUG
    std::string Name;
    const GlobalValue *Val;
#endif
};

//@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;

    unsigned MaxCallFrameSize;

    /// Cpu0CallEntry maps.
    StringMap<const Cpu0CallEntry *> ExternalCallEntries;
    ValueMap<const GlobalValue *, const Cpu0CallEntry *> GlobalCallEntries;
};

//@1 }

} // end of namespace llvm

#endif // CPU0_MACHINE_FUNCTION_INFO_H

```

Ibdex/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;

// class Cpu0CallEntry.
Cpu0CallEntry::Cpu0CallEntry(const StringRef &N) {
#ifndef NDEBUG
    Name = N;
    Val = nullptr;
#endif
}

Cpu0CallEntry::Cpu0CallEntry(const GlobalValue *V) {
#ifndef NDEBUG
    Val = V;
#endif
}

bool Cpu0CallEntry::isConstant(const MachineFrameInfo *) const {
    return false;
}

bool Cpu0CallEntry::isAliased(const MachineFrameInfo *) const {
    return false;
}

bool Cpu0CallEntry::mayAlias(const MachineFrameInfo *) const {
    return false;
}

void Cpu0CallEntry::printCustom(raw_ostream &O) const {
    O << "Cpu0CallEntry: ";
#ifndef NDEBUG
    if (Val)
        O << Val->getName();
    else
        O << Name;
#endif
}
```

```
Cpu0FunctionInfo::~Cpu0FunctionInfo() {
    for (StringMap<const Cpu0CallEntry *>::iterator
        I = ExternalCallEntries.begin(), E = ExternalCallEntries.end(); I != E;
        ++I)
        delete I->getValue();

    for (const auto &Entry : GlobalCallEntries)
        delete Entry.second;
}

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

[Index/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
#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();
}

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

unsigned GetEhDataReg(unsigned I) const;
};

}

#endif

```

Ibdex/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");
}

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];
}
```

Ibdex/chapters/Chapter3_1/Cpu0Subtarget.h

```
#include "Cpu0FrameLowering.h"
#include "Cpu0ISelLowering.h"
#include "Cpu0InstrInfo.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/MC/MCInstrItineraries.h"
#include "llvm/Target/TargetSelectionDAGInfo.h"
#include "llvm/Target/TargetSubtargetInfo.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
    }

    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
    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
}

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.
bool HasSlt;

InstrItineraryData InstrItins;

const Cpu0TargetMachine &TM;

Triple TargetTriple;

const TargetSelectionDAGInfo TSIInfo;

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

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

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

const InstrItineraryData *getInstrItineraryData() const { return &InstrItins; }

unsigned stackAlignment() const { return 8; }

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

const Cpu0InstrInfo *getInstrInfo() const { return InstrInfo.get(); }
const TargetFrameLowering *getFrameLowering() const {
    return FrameLowering.get();
}
const Cpu0RegisterInfo *getRegisterInfo() const {
    return &InstrInfo->getRegisterInfo();
}
const Cpu0TargetLowering *getTargetLowering() const { return TLInfo.get(); }
};

} // End llvm namespace

#endif

```

Index/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"
#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_SUBTARGETINFOCTOR
#include "Cpu0GenSubtargetInfo.inc"

extern bool FixGlobalBaseReg;

/// Select the Cpu0 CPU for the given triple and cpu name.
/// FIXME: Merge with the copy in Cpu0MCTargetDesc.cpp
static StringRef selectCpu0CPU(Triple TT, StringRef CPU) {
    if (CPU.empty() || CPU == "generic") {
        if (TT.getArch() == Triple::cpu0 || TT.getArch() == Triple::cpu0el)
            CPU = "cpu032II";
    }
    return CPU;
}

void Cpu0Subtarget::anchor() { }

//@1 {
Cpu0Subtarget::Cpu0Subtarget(const Triple &TT, const std::string &CPU,
                            const std::string &FS, bool little,
                            const Cpu0TargetMachine &_TM) :
//@1 }
    // Cpu0GenSubtargetInfo will display features by llc -march=cpu0 -mcpu=help
    Cpu0GenSubtargetInfo(TT, 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)) {

}

Cpu0Subtarget &
Cpu0Subtarget::initializeSubtargetDependencies(StringRef CPU, StringRef FS,
                                               const TargetMachine &TM) {
    std::string CPUName = selectCpu0CPU(TargetTriple, CPU);

    if (CPUName == "help")
        CPUName = "cpu032II";

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

    if (isCpu032I())
        HasCmp = true;
        HasSlt = false;
    }
    else if (isCpu032II())
        HasCmp = true;
        HasSlt = true;
}

```

```
    else {
        errs() << "-mcpu must be empty(default:cpu032II), cpu032I or cpu032II" << "\n";
    }

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

    return *this;
}

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

Index/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.
//
//=====

#ifndef CPU0REGISTERINFO_H
#define CPU0REGISTERINFO_H

#include "Cpu0Config.h"

#include "Cpu0.h"
#include "llvm/Target/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);

    /// getRegisterNumbering - Given the enum value for some register, e.g.
```

```

/// Cpu0::LR, return the number that it corresponds to (e.g. 14).
static unsigned getRegisterNumbering(unsigned RegEnum);

const MCPhysReg *
getCalleeSavedRegs(const MachineFunction *MF = nullptr) 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.
unsigned getFrameRegister(const MachineFunction &MF) const override;

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

} // end namespace llvm

#endif

```

Ibdex/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 "%u", 2, 0))>;
// llc create CSR_O32_SaveList and CSR_O32_RegMask from above defined.
const uint16_t* Cpu0RegisterInfo::
getCalleeSavedRegs(const MachineFunction *MF) const {
    return CSR_O32_SaveList;
}

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::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
unsigned Cpu0RegisterInfo::
getFrameRegister(const MachineFunction &MF) const {
    const TargetFrameLowering *TFI = MF.getSubtarget().getFrameLowering();
    return TFI->hasFP(MF) ? (Cpu0::FP) :
                                (Cpu0::SP);
}
```

[Index/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 CPU0SEREGISTERINFO_H
#define 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
```

Ibdex/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;
}
```

cmake_debug_build/lib/Target/Cpu0/Cpu0GenInstInfo.inc

```
//- Cpu0GenInstInfo.inc which generate from Cpu0InstrInfo.td
#ifndef GET_INSTRINFO_HEADER
#define 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);
};
```

Chapter3_1 add most Cpu0 backend classes. The code of Chapter3_1 can be summaried as Figure 3.1. Class Cpu0Subtarget supply the interface getInstrInfo(), setFrameLowering(), ..., to get other Cpu0 classes. Most classes (like Cpu0InstrInfo, Cpu0RegisterInfo, ...) have Subtarget reference member to allow them access other classes through the Cpu0Subtarget interface list in Figure 3.1. Classes (maybe added at later chapters) can access Subtarget class through Cpu0TargetMachine (usually use TM as symbol) by TM.getSubtargetImpl(). Once get Subtarget

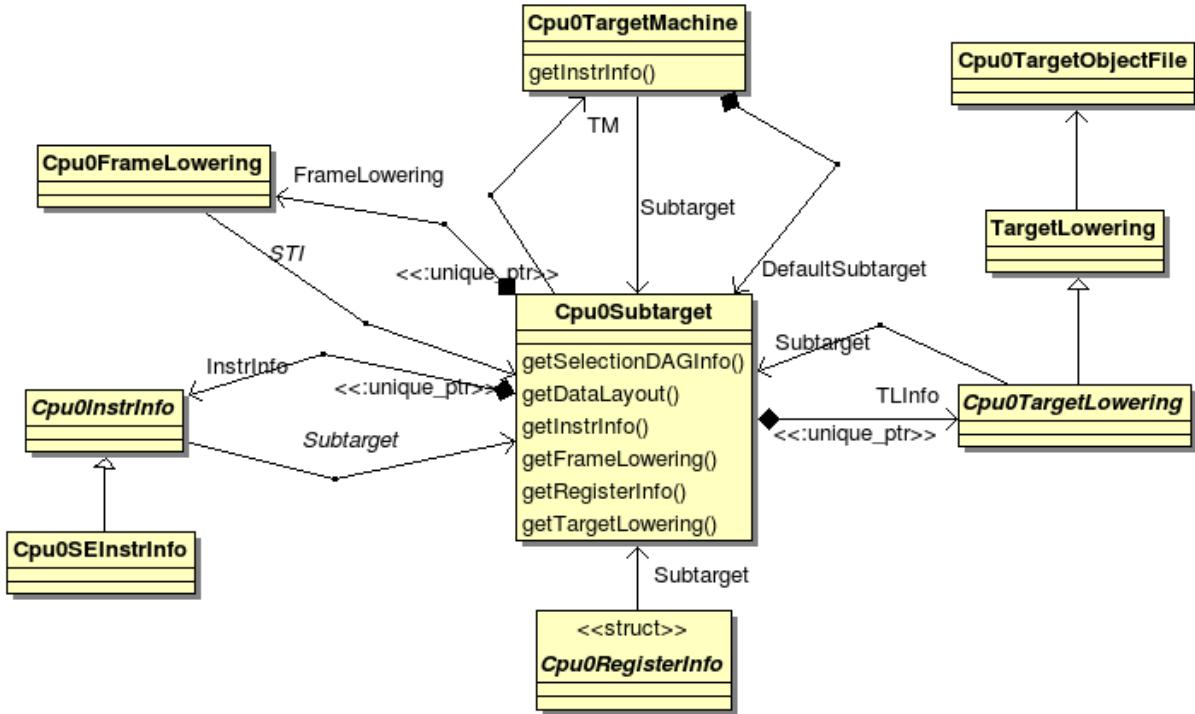


Figure 3.1: Cpu0 backend class access link

class, the backend code can accesss other classes through it. For those classes name of Cpu0SExx, they mean the standard 32 bits class. This arrangement follows llvm 3.5 Mips backend style. In Mips backend, it uses Mips16, MipsSE and Mips64 files/class name to define class function for 16, 32 and 64 bits architecture, respectively. Since Cpu0Subtarget creates Cpu0InstrInfo, Cpu0RegisterInfo, ..., at constuctor function, it can supply the class reference through the interfaces shown in Figure 3.1.

Below Figure 3.2 shows Cpu0 TableGen inheritance relationship. Last chapter mentioned llvm TableGen (llvmtblgen) is used in code generation process. Backend class can choose the TableGen generated classes and inherited from it. There are more TableGen generated classes, and they all exist in cmake_debug_build/lib/Target/Cpu0/*.inc. Through C++ inheritance mechanism, TableGen provides backend programmer a flexible way to use its generated code. Programmer has chance to override this function if it need to.

Since llvm has deep inheritance tree, they are not digged here. Benefit from the inheritance tree structure, there are not too much code need to be implemented in classes of instruction, frame/stack and select DAG, since many code are implemented by their parent class. The llvmtblgen generate Cpu0GenInstrInfo.inc 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 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.

[cmake_debug_build/lib/Target/Cpu0/Cpu0GenInstInfo.inc](#)

```

// - Cpu0GenInstInfo.inc which generate from Cpu0InstrInfo.td
#ifndef GET_INSTRINFO_HEADER
#define GET_INSTRINFO_HEADER
namespace llvm {

```

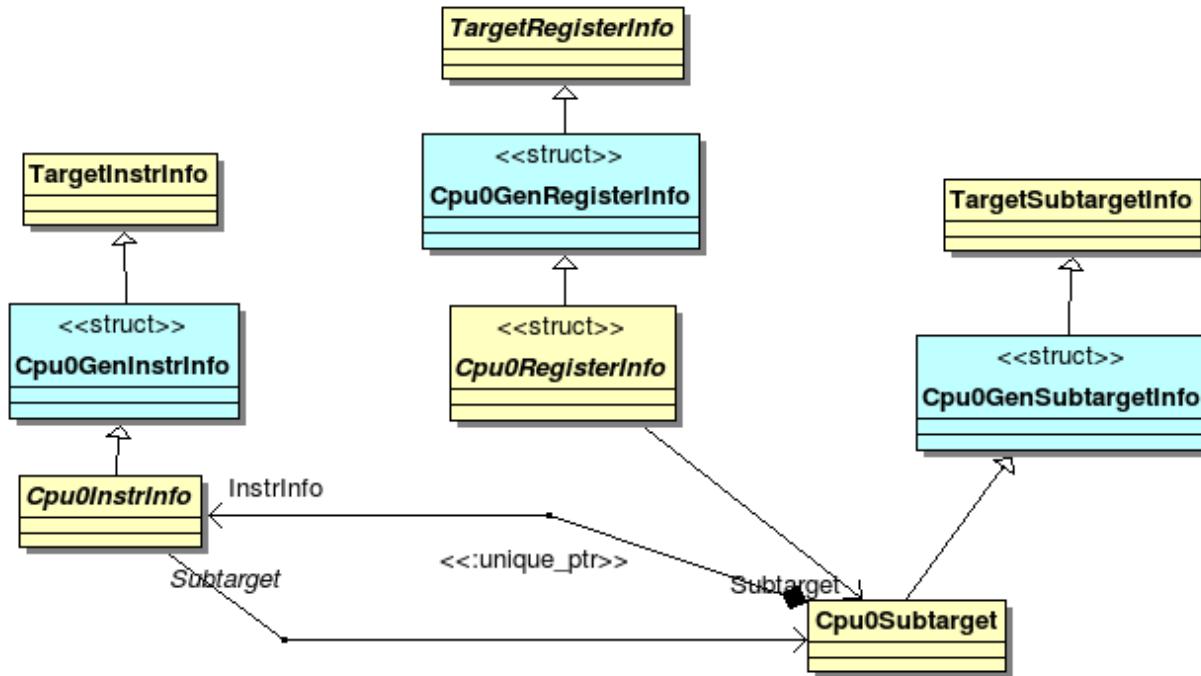


Figure 3.2: Cpu0 classes inherited from TableGen generated files

```

struct Cpu0GenInstrInfo : public TargetInstrInfoImpl {
    explicit Cpu0GenInstrInfo(int SO = -1, int DO = -1);
};

} // End llvm namespace
#endif // GET_INSTRINFO_HEADER
  
```

Reference “Write An LLVM Backend” web site ¹.

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

Ibdex/chapters/Chapter3_1/CMakeLists.txt

```

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

Cpu0FrameLowering.cpp
Cpu0InstrInfo.cpp
Cpu0ISelLowering.cpp
Cpu0MachineFunction.cpp
Cpu0RegisterInfo.cpp
Cpu0SEFrameLowering.cpp
Cpu0SEInstrInfo.cpp
Cpu0SEISelLowering.cpp
Cpu0SERegisterInfo.cpp
Cpu0Subtarget.cpp
Cpu0TargetObjectFile.cpp
  
```

¹ <http://llvm.org/docs/WritingAnLLVMBackend.html#target-machine>

Please take a look for Chapter3_1 code. After that, building Chapter3_1 by “`#define CH CH2`” in Cpu0Config.h as follows, and do building with Xcode on iMac or make on linux again.

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

```
#define CH      CH3_1

118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
... Assertion `AsmInfo && "MCAsmInfo not initialized. "
...
...
```

With Chapter3_1 implementation, the Chapter2 error message “Could not allocate target machine!” has gone. The new errors say that we have not Target AsmPrinter. 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 design choice” to readers, this book create two CPU. Readers will realize why Mips CPU uses instruction SLT instead of CMP when they go to 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/
cmake_debug_build/Debug/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..
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 by checking variable CPU in constructor function (the variable CPU 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`, respectively, 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`. Similarly, `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,

Ibdex/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/src/lib/Target/Cpu0SetChapter.h`

```
#define CH        CH4_1
```

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

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

```
#define CH        CH3_5
```

3.2 Add AsmPrinter

`Chapter3_2/` contains the `Cpu0AsmPrinter` definition.

Ibdex/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,

Ibdex/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 CPU0INSTPRINTER_H
#define CPU0INSTPRINTER_H

#include "Cpu0Config.h"

#include "llvm/MC/MCInstPrinter.h"

namespace llvm {
// These enumeration declarations were originally 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.
    void printInstruction(const MCInst *MI, raw_ostream &O);
    static const char *getRegisterName(unsigned RegNo);

    void printRegName(raw_ostream &OS, unsigned RegNo) const override;
    void printInst(const MCInst *MI, raw_ostream &O, StringRef Annot,
                  const MCSubtargetInfo &STI) override;

    bool printAliasInstr(const MCInst *MI, raw_ostream &OS);
    void printCustomAliasOperand(const MCInst *MI, unsigned OpIdx,
                                unsigned PrintMethodIdx, raw_ostream &O);
}

```

```

private:
    void printOperand(const MCInst *MI, unsigned OpNo, raw_ostream &O);
    void printUnsignedImm(const MCInst *MI, int opNum, raw_ostream &O);
    void printMemOperand(const MCInst *MI, int opNum, raw_ostream &O);
 $\text{//\#if } \text{CH} \geq \text{CH7_1}$ 
    void printMemOperandEA(const MCInst *MI, int opNum, raw_ostream &O);
 $\text{//\#endif}$ 
};

} // 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/MCE Expr.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 came from
//   Cpu0.td indicate.
    OS << '$' << StringRef(getRegisterName(RegNo)).lower();
}

//@1 {
void Cpu0InstPrinter::printInst(const MCInst *MI, raw_ostream &O,
                              StringRef Annot, const MCSubtargetInfo &STI) {
    // Try to print any aliases first.
    if (!printAliasInstr(MI, O))
//@1 }

```

```

//- printInstruction(MI, O) defined in Cpu0GenAsmWriter.inc which came from
//  Cpu0.td indicate.
printInstruction(MI, O);
printAnnotation(O, Annot);
}

//@printExpr {
static void printExpr(const MCExpr *Expr, raw_ostream &OS) {
//@printExpr body {
int Offset = 0;
const MCSymbolRefExpr *SRE;

if (const MCBinaryExpr *BE = dyn_cast<MCBinaryExpr>(Expr)) {
    SRE = dyn_cast<MCSymbolRefExpr>(BE->getLHS());
    const MCConstantExpr *CE = dyn_cast<MCConstantExpr>(BE->getRHS());
    assert(SRE && CE && "Binary expression must be sym+const.");
    Offset = CE->getValue();
}
else if (!(SRE = dyn_cast<MCSymbolRefExpr>(Expr)))
    assert(false && "Unexpected MCExpr type.");
MCSymbolRefExpr::VariantKind Kind = SRE->getKind();

switch (Kind) {
default: llvm_unreachable("Invalid kind!");
case MCSymbolRefExpr::VK_None:
}

OS << SRE->getSymbol();

if (Offset) {
    if (Offset > 0)
        OS << '+';
    OS << Offset;
}

if ((Kind == MCSymbolRefExpr::VK_Cpu0_GPOFF_HI) ||
    (Kind == MCSymbolRefExpr::VK_Cpu0_GPOFF_LO))
    OS << ")";
else if (Kind != MCSymbolRefExpr::VK_None)
    OS << ')';
}

void Cpu0InstPrinter::printOperand(const MCInst *MI, unsigned OpNo,
                                  raw_ostream &O) {
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");
printExpr(Op.getExpr(), O);
}

```

```
}

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
```

Index/chapters/Chapter3_2/InstPrinter/CMakeLists.txt

```
add_llvm_library(LLVMCpu0AsmPrinter
                  Cpu0InstPrinter.cpp
                )
```

Index/chapters/Chapter3_2/InstPrinter/LLVMBuild.txt

```
;===== ./lib/Target/Cpu0/InstPrinter/LLVMBuild.txt -----*-- Conf -*---=;
;
;                               The LLVM Compiler Infrastructure
;
; This file is distributed under the University of Illinois Open Source
; License. See LICENSE.TXT for details.
;
;=====-----=;
;
; This is an LLVMBuild description file for the components in this subdirectory.
;
```

```

; For more information on the LLVMBuild system, please see:
;
;   http://llvm.org/docs/LLVMBuild.html
;
;=====;

[component_0]
type = Library
name = Cpu0AsmPrinter
parent = Cpu0
required_libraries = MC Support
add_to_library_groups = 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_1.

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,

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```

// Address operand
def mem : Operand<i32> {
    let PrintMethod = "printMemOperand";
    let MIOperandInfo = (ops CPUREgs, simm16);
    let EncoderMethod = "getMemEncoding";
//#if CH >= CH11_1
    let ParserMatchClass = Cpu0MemAsmOperand;
//#endif
}
...
// 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 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 `MCTarget-Desc` as follows,

Ibdex/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 CPU0MCINSTLOWER_H  
#define 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
```

Ibdex/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/MCE Expr.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();

    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);
    }
}

```

[Index/chapters/Chapter3_2/MCTargetDesc/Cpu0BaseInfo.h](#)

```

===== Cpu0BaseInfo.h - Top level definitions for CPU0 MC -----*---//  

//  

//          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 CPU0BASEINFO_H
#define CPU0BASEINFO_H

#include "Cpu0Config.h"

#include "Cpu0MCTargetDesc.h"
#include "llvm/MC/MCE Expr.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_DISP,
MO_GOT_PAGE,
MO_GOT_OFST,

// N32/64 Flags.
MO_GPOFF_HI,
MO_GPOFF_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,
/// 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
};

//@get register number
/// getCpu0RegisterNumbering - Given the enum value for some register,
/// return the number that it corresponds to.
inline static unsigned getCpu0RegisterNumbering(unsigned RegEnum)
{
    switch (RegEnum) {
//@1

```

```
case Cpu0::ZERO:
    return 0;
case Cpu0::AT:
    return 1;
case Cpu0::V0:
    return 2;
case Cpu0::V1:
    return 3;
case Cpu0::A0:
    return 4;
case Cpu0::A1:
    return 5;
case Cpu0::T9:
    return 6;
case Cpu0::T0:
    return 7;
case Cpu0::T1:
    return 8;
case Cpu0::S0:
    return 9;
case Cpu0::S1:
    return 10;
case Cpu0::GP:
    return 11;
case Cpu0::FP:
    return 12;
case Cpu0::SP:
    return 13;
case Cpu0::LR:
    return 14;
case Cpu0::SW:
    return 15;
case Cpu0::PC:
    return 0;
case Cpu0::EPC:
    return 1;
default: llvm_unreachable("Unknown register number!");
}
}

#endif
```

Ibdex/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 CPU0TARGETASMINFO_H  
#define CPU0TARGETASMINFO_H  
  
#include "Cpu0Config.h"  
#if CH >= CH3_2  
  
#include "llvm/MC/MCAsmInfo.h"  
  
namespace llvm {  
    class Triple;  
    class Target;  
  
    class Cpu0MCAsmInfo : public MCAsmInfo {  
        virtual void anchor();  
    public:  
        explicit Cpu0MCAsmInfo(const Triple &TheTriple);  
    };  
}  
// namespace llvm  
  
#endif // #if CH >= CH3_2  
  
#endif
```

Ibdex/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;  
  
    AlignmentIsInBytes      = false;  
    Data16bitsDirective    = "\t.2byte\t";  
    Data32bitsDirective    = "\t.4byte\t";
```

```

Data64bitsDirective      = "\t.8byte\t";
PrivateGlobalPrefix      = "$";
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. By the way, it register other classes (register, instruction and subtarget) defined in Chapter3_1 at this point.

[Index/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;
...
}

```

[Index/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 StringRef 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::LR); // defined in Cpu0GenRegisterInfo.inc
    return X;
}

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;
    }
    return createCpu0MCSubtargetInfoImpl(TT, CPU, ArchFS);
// createCpu0MCSubtargetInfoImpl defined in Cpu0GenSubtargetInfo.inc
}

static MCAsmInfo *createCpu0MCAsmInfo(const MCRegisterInfo &MRI,
                                         const Triple &TT) {
    MCAsmInfo *MAI = new Cpu0MCAsmInfo(TT);

    unsigned SP = MRI.getDwarfRegNum(Cpu0::SP, true);
    MCCFIInstruction Inst = MCCFIInstruction::createDefCfa(0, SP, 0);
    MAI->addInitialFrameState(Inst);

    return MAI;
}

static MCCCodeGenInfo *createCpu0MCCCodeGenInfo(const Triple &TT, Reloc::Model RM,
                                                CodeModel::Model CM,
                                                CodeGenOpt::Level OL) {
    MCCCodeGenInfo *X = new MCCCodeGenInfo();
    if (CM == CodeModel::JITDefault)
        RM = Reloc::Static;
    else if (RM == Reloc::Default)
        RM = Reloc::PIC_;
    X->initMCCCodeGenInfo(RM, CM, OL); // defined in lib/MC/MCCCodeGenInfo.cpp
    return X;
}

static MCInstPrinter *createCpu0MCInstPrinter(const Triple &T,
                                              unsigned SyntaxVariant,
                                              const MCAsmInfo &MAI,
                                              const MCInstrInfo &MII,
                                              const MCRegisterInfo &MRI) {

```

```
    return new Cpu0InstPrinter(MAI, MII, MRI);
}

//@2 {
extern "C" void LLVMInitializeCpu0TargetMC() {
    for (Target *T : {&TheCpu0Target, &TheCpu0elTarget}) {
        // Register the MC asm info.
        RegisterMCAsmInfoFn X(*T, createCpu0MCAsmInfo);

        // Register the MC codegen info.
        TargetRegistry::RegisterMCCodeGenInfo(*T,
                                              createCpu0MCCodeGenInfo);

        // 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 MCInstPrinter.
        TargetRegistry::RegisterMCInstPrinter(*T,
                                              createCpu0MCInstPrinter);
    }
}
//@2 }
```

Index/chapters/Chapter3_2/MCTargetDesc/CMakeLists.txt

Cpu0MCAsmInfo.cpp

Index/chapters/Chapter3_2/MCTargetDesc/LLVMBuild.txt

Cpu0AsmPrinter

To make the registration clearly, list it in Figure 3.3, Figure 3.4 and Figure 3.5.

According “section Target Registration”², we can register Cpu0 backend classes at LLVMInitializeCpu0TargetMC() on demand by the dynamic register mechanism as the above function, LLVMInitializeCpu0TargetMC(). In the last section of later chapter “Generating object files”, we will review this part and explain some of them more clearly with Figure display. The whole registration functions will be reviewed and explained at later chapter “Generating object files”.

Now, it’s time to work with AsmPrinter as follows,

Index/chapters/Chapter3_2/Cpu0AsmPrinter.h

² <http://jonathan2251.github.io/lbd/llvmsstructure.html#target-registration>

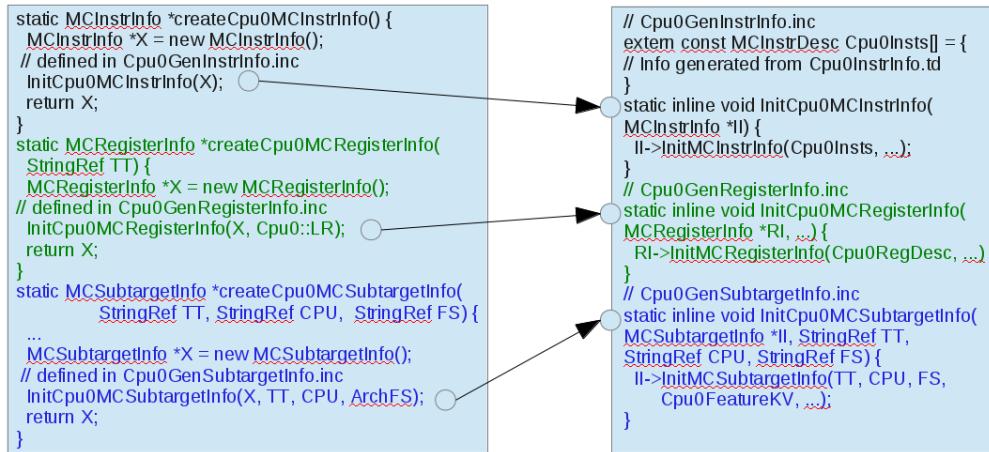


Figure 3.3: Cpu0 InstrInfo, RegisterInfo and SubtargetInfo register function

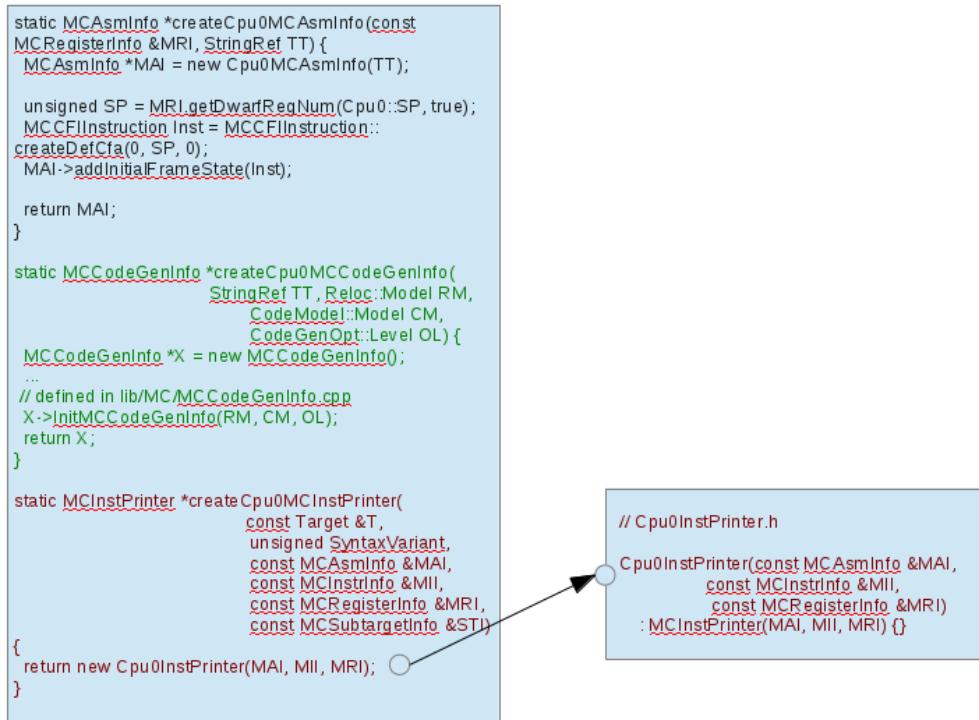


Figure 3.4: Cpu0MCAsmInfo and Cpu0InstPrinter register function

```

extern "C" void LLVMInitializeCpu0TargetMC() {
    // Register the MC asm info.
    RegisterMCAsmInfoFn X(TheCpu0Target, createCpu0MCAsmInfo);
    RegisterMCAsmInfoFn Y(TheCpu0eITarget, createCpu0MCAsmInfo);

    // Register the MC codegen info.
    TargetRegistry::RegisterMCCodeGenInfo(TheCpu0Target,
                                          createCpu0MCCodeGenInfo);
    TargetRegistry::RegisterMCCodeGenInfo(TheCpu0eITarget,
                                          createCpu0MCCodeGenInfo);

    // Register the MC instruction info.
    TargetRegistry::RegisterMCInstrInfo(TheCpu0Target, createCpu0MCInstrInfo);
    TargetRegistry::RegisterMCInstrInfo(TheCpu0eITarget, createCpu0MCInstrInfo);

    // Register the MC register info.
    TargetRegistry::RegisterMCRegInfo(TheCpu0Target, createCpu0MCRegisterInfo);
    TargetRegistry::RegisterMCRegInfo(TheCpu0eITarget, createCpu0MCRegisterInfo);

    // Register the MC subtarget info.
    TargetRegistry::RegisterMCSubtargetInfo(TheCpu0Target,
                                           createCpu0MCSubtargetInfo);
    TargetRegistry::RegisterMCSubtargetInfo(TheCpu0eITarget,
                                           createCpu0MCSubtargetInfo);

    // Register the MC InstPrinter.
    TargetRegistry::RegisterMCInstPrinter(TheCpu0Target,
                                         createCpu0MCInstPrinter);
    TargetRegistry::RegisterMCInstPrinter(TheCpu0eITarget,
                                         createCpu0MCInstPrinter);
}

```

Figure 3.5: LLVMInitializeCpu0TargetMC() dynamic register

```

//===== 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 CPU0ASMPRINTER_H
#define 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"
#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 &MOp);

public:

const Cpu0Subtarget *Subtarget;
const Cpu0FunctionInfo *Cpu0FI;
Cpu0MCInstLowering MCInstLowering;

explicit Cpu0AsmPrinter(TargetMachine &TM,
                      std::unique_ptr<MCStreamer> Streamer)
: AsmPrinter(TM, std::move(Streamer)),
MCInstLowering(*this) {
    Subtarget = static_cast<Cpu0TargetMachine &>(TM).getSubtargetImpl();
}

virtual const char *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);
};

};

#endif

```

Index/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) {
    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;
    }
}
```

```

MachineBasicBlock::const_instr_iterator I = MI;
MachineBasicBlock::const_instr_iterator E = MI->getParent()->instr_end();

MCInst TmpInst0;
do {
    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
//     .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 subtraction, 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 std::vector<CalleeSavedInfo> &CSI = MFI->getCalleeSavedInfo();
// size of stack area to which FP callee-saved regs are saved.
unsigned CPUREgSize = Cpu0::CPUREgsRegClass.getSize();
unsigned i = 0, e = CSI.size();

// Set CPU Bitmask.
for (; i != e; ++i) {
    unsigned Reg = CSI[i].getReg();
    unsigned RegNum = getCpu0RegisterNumbering(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) {
    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() {
    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->get.EmitNOAT())
            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->get.EmitNOAT())
            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
    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 according the information from Cpu0GenInstrInfo.inc and Cpu0GenRegisterInfo.inc both registered at dynamic register function, LLVMInitializeCpu0TargetMC(), as [Figure 3.5](#) and [Figure 3.3](#). Notice, file Cpu0InstPrinter.cpp only print operand while the OP code information come from Cpu0InstrInfo.td.

Add the following code to Cpu0ISelLowering.cpp.

[Index/chapters/Chapter3_2/Cpu0ISelLowering.cpp](#)

```

Cpu0TargetLowering::
Cpu0TargetLowering(Cpu0TargetMachine &TM)
: TargetLowering(TM, new Cpu0TargetObjectFile()),
Subtarget(&TM.getSubtarget<Cpu0Subtarget>()) {

    // Set up the register classes
    addRegisterClass(MVT::i32, &Cpu0::CPURegsRegClass);

    // 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().

[Index/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:
...
bool EmitNOAT;
};
```

Beyond adding these new .cpp files to CMakeLists.txt, please remember to add subdirectory InstPrinter, enable asm-printer, adding libraries AsmPrinter and Cpu0AsmPrinter to LLVMBuild.txt as follows,

Ibdex/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

...
)
...
add_subdirectory(InstPrinter)
```

Ibdex/chapters/Chapter3_2/LLVMBuild.txt

```
// LLVMBuild.txt
[common]
subdirectories =
    InstPrinter
    ...
[component_0]
...
# Please enable asmprinter
has_asmprinter = 1
...

[component_1]
required_libraries =
    AsmPrinter
    ...
    Cpu0AsmPrinter
    ...
```

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/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
/Users/Jonathan/llvm/test/cmake_debug_build/Debug/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
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 need in later section "Add Prologue/Epilogue functions" of this chapter. IR DAG is defined in file include/llvm/Target/TargetSelectionDAG.td.

Index/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,

Index/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.

Ibdex/chapters/Chapter3_3/Cpu0TargetMachine.cpp

```
class Cpu0PassConfig : public TargetPassConfig {
public:
    ...

    bool addInstSelector() override;

};

...

// Install an instruction selector pass using
// the ISelDag to gen Cpu0 code.
bool Cpu0PassConfig::addInstSelector() {
    addPass(createCpu0ISelDag(getCpu0TargetMachine()));
    return false;
}
```

Ibdex/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 CPU0ISELDAGTODAG_H
#define 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
//=====

//=====
// Cpu0DAGToDAGISel - CPU0 specific code to select CPU0 machine
```

```
// instructions for SelectionDAG operations.
//=====
namespace llvm {

class Cpu0DAGToDAGISel : public SelectionDAGISel {
public:
    explicit Cpu0DAGToDAGISel(Cpu0TargetMachine &TM)
        : SelectionDAGISel(TM), Subtarget(nullptr) {}

    // Pass Name
    const char *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);
    }

    SDNode *Select(SDNode *N) override;

    virtual std::pair<bool, SDNode*> selectNode(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;

};

/// createCpu0ISelDag - This pass converts a legalized DAG into a
/// CPU0-specific DAG, ready for instruction scheduling.
FunctionPass *createCpu0ISelDag(Cpu0TargetMachine &TM);

}

#endif
```

Ibdex/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 "Cpu0Subtarget.h"
#include "Cpu0TargetMachine.h"
#include "MCTargetDesc/Cpu0BaseInfo.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/CodeGen/MachineConstantPool.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/SelectionDAGISel.h"
#include "llvm/CodeGen/SelectionDAGNodes.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

#define DEBUG_TYPE "cpu0-isel"

//=====
// Instruction Selector Implementation
//=====

//=====
// 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 (luxcl or suxcl).  

    const LSBaseSDNode* LS = 0;  

if (Parent && (LS = dyn_cast<LSBaseSDNode>(Parent))) {  

    EVT VT = LS->getMemoryVT();  

    if (VT.getSizeInBits() / 8 > LS->getAlignment()) {  

        assert("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
SDNode* Cpu0DAGToDAGISel::Select(SDNode *Node) {  

//@Select  

    // Dump information about the Node being selected
    DEBUG(errs() << "Selecting: "; Node->dump(CurDAG); errs() << "\n");  

    // If we have a custom node, we already have selected!
    if (Node->isMachineOpcode()) {
        DEBUG(errs() << "==" "; Node->dump(CurDAG); errs() << "\n");     Node->setNodeId(-1);
        return nullptr;
    }
  

    // See if subclasses can handle this node.
    std::pair<bool, SDNode*> Ret = selectNode(Node);  

    if (Ret.first)
        return Ret.second;  

    unsigned Opcode = Node->getOpcode();
    switch(Opcode) {

```

```

default: break;

#if 0
//#ifndef NDEBUG
    case ISD::LOAD:
    case ISD::STORE:
        assert((Subtarget->systemSupportsUnalignedAccess() ||
            cast<MemSDNode>(Node)->getMemoryVT().getSizeInBits() / 8 <=
            cast<MemSDNode>(Node)->getAlignment()) &&
            "Unexpected unaligned loads/stores.");
        break;
#endif
}

// Select the default instruction
SDNode *ResNode = SelectCode(Node);

DEBUG(errs() << "=> ");
if (ResNode == NULL || ResNode == Node)
    DEBUG(Node->dump(CurDAG));
else
    DEBUG(ResNode->dump(CurDAG));
DEBUG(errs() << "\n");
return ResNode;
}

/// createCpu0ISelDag - This pass converts a legalized DAG into a
/// CPU0-specific DAG, ready for instruction scheduling.
FunctionPass *llvm::createCpu0ISelDag(Cpu0TargetMachine &TM) {
    return llvm::createCpu0SEISelDag(TM);
}

```

[Index/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 CPU0SEISELDAGTODAG_H
#define CPU0SEISELDAGTODAG_H

#include "Cpu0Config.h"

#include "Cpu0ISelDAGToDAG.h"

namespace llvm {

```

```
class Cpu0SEDAGToDAGISel : public Cpu0DAGToDAGISel {  
  
public:  
    explicit Cpu0SEDAGToDAGISel(Cpu0TargetMachine &TM) : Cpu0DAGToDAGISel(TM) {}  
  
private:  
  
    bool runOnMachineFunction(MachineFunction &MF) override;  
  
    std::pair<bool, SDNode*> selectNode(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);  
}  
  
#endif
```

Index/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"  
#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
std::pair<bool, SDNode*> Cpu0SEDAGToDAGISel::selectNode(SDNode *Node) {
    unsigned Opcode = Node->getOpcode();
    SDLoc DL(Node);

    /**
     // Instruction Selection not handled by the auto-generated
     // tablegen selection should be handled here.
     /**
     SDNode *Result;

    /**
     // 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 std::make_pair(false, nullptr);
}

FunctionPass *llvm::createCpu0SEISelDag(Cpu0TargetMachine &TM) {
    return new Cpu0SEDAGToDAGISel(TM);
}

```

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 name corresponding to Chapter2/Cpu0InstrInfo.td as follows,

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```
def addr : ComplexPattern<iPTR, 2, "SelectAddr", [frameindex], [SDNPWantParent]>;
```

The iPTR, ComplexPattern, frameindex and SDNPWantParent defined as follows,

src/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;
}
```

src/include/llvm/CodeGen/ValueTypes.td

```
// Pseudo valuetype mapped to the current pointer size.
def iPTR : Valuetype<0, 255>;
```

Chapter3_3 adding the following code in Cpu0InstInfo.cpp to enable debug information which called by llvm at proper time.

lbdex/chapters/Chapter3_3/Cpu0InstrInfo.h

```
virtual MachineInstr* emitFrameIndexDebugValue(MachineFunction &MF,
                                                int FrameIx, uint64_t Offset,
                                                const MDNode *MDPtr,
                                                DebugLoc DL) const;
```

lbdex/chapters/Chapter3_3/Cpu0InstrInfo.cpp

```
MachineInstr*
Cpu0InstrInfo::emitFrameIndexDebugValue(MachineFunction &MF, int FrameIx,
                                         uint64_t Offset, const MDNode *MDPtr,
                                         DebugLoc DL) const {
    MachineInstrBuilder MIB = BuildMI(MF, DL, get(Cpu0::DBG_VALUE))
        .addFrameIndex(FrameIx).addImm(0).addImm(Offset).addMetadata(MDPtr);
    return &*MIB;
}
```

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/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
```

```

ch3.cpu0.s
...
LLVM ERROR: Cannot select: 0x24834b8: ch = Cpu0ISD::Ret 0x24832a8, 0x24833b0 [ORD=4] [ID=6]
  0x24833b0: i32 = Register %LR [ID=4]
...
0x7f80f182d210: i32 = Register %LR [ID=4]

```

Above can display the error message DAG node “Cpu0ISD::Ret” because the following code added in Chapter3_1/Cpu0ISelLowering.cpp.

Ibdex/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::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

Ibdex/chapters/Chapter3_4/CMakeLists.txt

```

add_llvm_target(
    ...
    Cpu0AnalyzeImmediate.cpp
    ...
)

```

Ibdex/chapters/Chapter3_4/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_4

#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);

    /// 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_4

#endif

```

Ibdex/chapters/Chapter3_4/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_4

#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 sequence 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);
}

void Cpu0AnalyzeImmediate::GetInstSeqLsADDiu(uint64_t Imm, unsigned RemSize,
                                              InstSeqLs &SeqLs) {
    GetInstSeqLs((Imm + 0x8000ULL) & 0xfffffffffffff0000ULL, RemSize, SeqLs);
    AddInstr(SeqLs, Inst(ADDiu, Imm & 0xffffULL));
}

void Cpu0AnalyzeImmediate::GetInstSeqLsORi(uint64_t Imm, unsigned RemSize,
                                             InstSeqLs &SeqLs) {
    GetInstSeqLs(Imm & 0xfffffffffffff0000ULL, 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 & (0xfffffffffffffULL >> (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
// 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;

    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

```

Ibdex/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>
]>;

```

Ibdex/chapters/Chapter3_4/Cpu0FrameLowering.h

```

protected:
    uint64_t estimateStackSize(const MachineFunction &MF) const;

```

Ibdex/chapters/Chapter3_4/Cpu0FrameLowering.cpp

```
#include "Cpu0AnalyzeImmediate.h"

uint64_t Cpu0FrameLowering::estimateStackSize(const MachineFunction &MF) const {
    const MachineFrameInfo *MFI = MF.getFrameInfo();
    const TargetRegisterInfo &TRI = *MFI.getSubtarget().getRegisterInfo();

    int64_t Offset = 0;

    // Iterate over fixed sized objects.
    for (int I = MFI->getObjectIndexBegin(); I != 0; ++I)
        Offset = std::max(Offset, -MFI->getObjectOffset(I));

    // Conservatively assume all callee-saved registers will be saved.
    for (const MCPhysReg *R = TRI.getCalleeSavedRegs(&MF); *R; ++R) {
        unsigned Size = TRI.getMinimalPhysRegClass(*R)->getSize();
        Offset = RoundUpToAlignment(Offset + Size, Size);
    }

    unsigned MaxAlign = MFI->getMaxAlignment();

    // Check that MaxAlign is not zero if there is a stack object that is not a
    // callee-saved spill.
    assert(!MFI->getObjectIndexEnd() || MaxAlign);

    // Iterate over other objects.
    for (unsigned I = 0, E = MFI->getObjectIndexEnd(); I != E; ++I)
        Offset = RoundUpToAlignment(Offset + MFI->getObjectSize(I), MaxAlign);

    // Call frame.
    if (MFI->adjustsStack() && hasReservedCallFrame(MF))
        Offset = RoundUpToAlignment(Offset + MFI->getMaxCallFrameSize(),
                                   std::max(MaxAlign, getStackAlignment()));

    return RoundUpToAlignment(Offset, getStackAlignment());
}
```

Ibdex/chapters/Chapter3_4/Cpu0SEFrameLowering.cpp

```
#include "Cpu0AnalyzeImmediate.h"
```

Ibdex/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, IIIPseudo, Pseudo> {
    let isCodeGenOnly = 1;
    let isPseudo = 1;
}
```

Ibdex/chapters/Chapter3_4/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), [<
  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_4] 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_4] 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;
let shamt = shamt;
}

// 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_4] 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;
    let isReMaterializable = 1;
}
}

let Predicates = [Ch3_4] in {
def ORi      : ArithLogicI<0x0d, "ori", or, uimm16, immZExt16, CPURegs>;
}

let Predicates = [Ch3_4] in {
def LUi      : LoadUpper<0x0f, "lui", GPROut, uimm16>;
}

let Predicates = [Ch3_4] in {
let Predicates = [DisableOverflow] in {
def ADDu     : ArithLogicR<0x11, "addu", add, IIAlu, CPURegs, 1>;
}
}

let Predicates = [Ch3_4] in {
def SHL      : shift_rotate_imm32<0x1e, 0x00, "shl", shl>;
}

let Predicates = [Ch3_4] in {
let isReturn=1, isTerminator=1, hasDelaySlot=1, isBarrier=1, hasCtrlDep=1 in
    def RetLR : Cpu0Pseudo<(outs), (ins), "", [(Cpu0Ret)]>;
}

let Predicates = [Ch3_4] in {
//=====
// Instruction aliases
//=====
def : Cpu0InstAlias<"move $dst, $src",
    (ADDu GPROut:$dst, GPROut:$src, ZERO), 1>;
}

let Predicates = [Ch3_4] 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]
```

Ibdex/chapters/Chapter3_4/Cpu0InstrInfo.h

```
#include "Cpu0AnalyzeImmediate.h"

void storeRegToStackSlot (MachineBasicBlock &MBB,
                         MachineBasicBlock::iterator MBBI,
                         unsigned 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,
                           unsigned 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,
                               unsigned SrcReg, bool isKill, int FrameIndex,
                               const TargetRegisterClass *RC,
                               const TargetRegisterInfo *TRI,
                               int64_t Offset) const = 0;

virtual void loadRegFromStack (MachineBasicBlock &MBB,
                               MachineBasicBlock::iterator MI,
                               unsigned DestReg, int FrameIndex,
                               const TargetRegisterClass *RC,
                               const TargetRegisterInfo *TRI,
                               int64_t Offset) const = 0;
```

Ibdex/chapters/Chapter3_4/Cpu0SEInstrInfo.h

```
void storeRegToStack (MachineBasicBlock &MBB,
                      MachineBasicBlock::iterator MI,
                      unsigned SrcReg, bool isKill, int FrameIndex,
                      const TargetRegisterClass *RC,
                      const TargetRegisterInfo *TRI,
                      int64_t Offset) const override;

void loadRegFromStack (MachineBasicBlock &MBB,
                      MachineBasicBlock::iterator MI,
                      unsigned DestReg, int FrameIndex,
```

```

    const TargetRegisterClass *RC,
    const TargetRegisterInfo *TRI,
    int64_t Offset) const override;

bool expandPostRAPpseudo(MachineBasicBlock::iterator MI) const override;

/// Adjust SP by Amount bytes.
void adjustStackPtr(Cpu0FunctionInfo *Cpu0FI, unsigned SP, int64_t Amount,
                     MachineBasicBlock &MBB, MachineBasicBlock::iterator I)
    const;

/// 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, DebugLoc DL,
                      unsigned *NewImm) const;

private:
void ExpandRetLR(MachineBasicBlock &MBB, MachineBasicBlock::iterator I,
                   unsigned Opc) const;

```

[Index/chapters/Chapter3_4/Cpu0SEInstrInfo.cpp](#)

```

static MachineMemOperand* GetMemOperand(MachineBasicBlock &MBB, int FI,
                                         unsigned Flag) {
    MachineFunction &MF = *MBB.getParent();
    MachineFrameInfo &MFI = *MF.getFrameInfo();
    unsigned Align = MFI.getObjectAlignment(FI);

    return MF.getMachineMemOperand(MachinePointerInfo::getFixedStack(FI), Flag,
                                    MFI.getObjectSize(FI), Align);
}

void Cpu0SEInstrInfo::
storeRegToStack(MachineBasicBlock &MBB, MachineBasicBlock::iterator I,
                unsigned SrcReg, bool isKill, 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::MOSStore);

    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(0).addMemOperand(MMO);
}

void Cpu0SEInstrInfo::
loadRegFromStack(MachineBasicBlock &MBB, MachineBasicBlock::iterator I,
                 unsigned 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(0)
    .addMemOperand(MMO);
}

// Cpu0SEInstrInfo::expandPostRAPpseudo
/// Expand Pseudo instructions into real backend instructions
bool Cpu0SEInstrInfo::expandPostRAPpseudo(MachineBasicBlock::iterator MI) const {
    MachineBasicBlock &MBB = *MI->getParent();

    switch(MI->getDesc().getOpcode()) {
        default:
            return false;
        case Cpu0::RetLR:
            ExpandRetLR(MBB, MI, Cpu0::RET);
            break;
    }

    MBB.erase(MI);
    return true;
}

/// Adjust SP by Amount bytes.
void Cpu0SEInstrInfo::adjustStackPtr(Cpu0FunctionInfo *Cpu0FI, 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.
        Cpu0FI->setEmitNOAT();
        unsigned Reg = loadImmediate(Amount, MBB, I, DL, nullptr);
        BuildMI(MBB, I, DL, get(ADDu), SP).addReg(SP).addReg(Reg);
    }
}

/// 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, 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;
}

void Cpu0SEInstrInfo::ExpandRetLR(MachineBasicBlock &MBB,
                                    MachineBasicBlock::iterator I,
                                    unsigned Opc) const {
    BuildMI(MBB, I, I->getDebugLoc(), get(Opc)).addReg(Cpu0::LR);
}
```

[Index/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; }

    /// 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, const Type *OrigTy, const SDNode *CallNode,
              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;
};

```

[Index/chapters/Chapter3_4/Cpu0ISelLowering.cpp](#)

```

SDValue
Cpu0TargetLowering::LowerReturn(SDValue Chain,
                                CallingConv::ID CallConv, bool IsVarArg,
                                const SmallVectorImpl<ISD::OutputArg> &Outs,
                                const SmallVectorImpl<SDValue> &OutVals,
                                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);

// 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, RetTy, CallNode, 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, const Type *OrigTy,
                                         const SDNode *CallNode,
                                         bool IsSoftFloat) const {
    if (IsSoftFloat || IsO32)
        return VT;

    return VT;
}

```

[Index/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),

    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;

/// Frame objects for spilling eh data registers.
int EhDataRegFI[4];

}

```

Ilindex/chapters/Chapter3_4/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!");
    }

    DEBUG(errs() << "\nFunction : " << MF.getFunction()->getName() << "\n";
          errs() << "<----->\n" << MI);

    int FrameIndex = MI.getOperand(i).getIndex();
    uint64_t stackSize = MF.getFrameInfo()->getStackSize();
    int64_t spOffset = MF.getFrameInfo()->getObjectOffset(FrameIndex);

    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();

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)) {
    assert("(!MI.isDebugValue() && !isInt<16>(Offset))");
}

MI.getOperand(i).ChangeToRegister(FrameReg, false);
MI.getOperand(i+1).ChangeToImmediate(Offset);
}

```

Ibdex/chapters/Chapter3_4/Cpu0SEISelDAGToDAG.cpp

```
#include "Cpu0AnalyzeImmediate.h"
```

To handle IR ret, these code 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,

Ibdex/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.

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 (those *.inc generated by TableGen from *.td) generated by the following code at “Cpu0 Assembly Printer” stage.

Ibdex/chapters/Chapter2/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;
}

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

def RET      : RetBase<GPROut>;
```

Table 3.1: 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 specifically, 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.

Function storeRegToStack() and loadRegFromStack() of Cpu0SEInstrInfo.cpp, storeRegToStackSlot() and loadRegFromStackSlot() of Cpu0InstrInfo.cpp are for handling the registers spill during register allocation process.

Build Chapter3_4 and run with it, finding the error message in Chapter3_3 is gone. The compile result as follows,

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/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 $fp,8,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $2, $zero, 0
st $2, 4($fp)
ret $lr
.set macro
.set reorder
.end main
$tmp0:
.size main, ($tmp0)-main
```

Although Chapter3_4 can generated asm for ch3.cpp, you will find the ** st \$2, 4(\$fp) ** instruction will has trouble since the stack pointer (stack frame) has not addadjustment. This problem will be fixed at Chapter3_5.

3.5 Add Prologue/Epilogue functions

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. [Figure 3.6](#) 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.

We will explain the Prologue and Epilogue further by example code.

For the following llvm IR code of ch3.cpp, Cpu0 backend will emit the corresponding machine instructions as follows,

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

.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
```

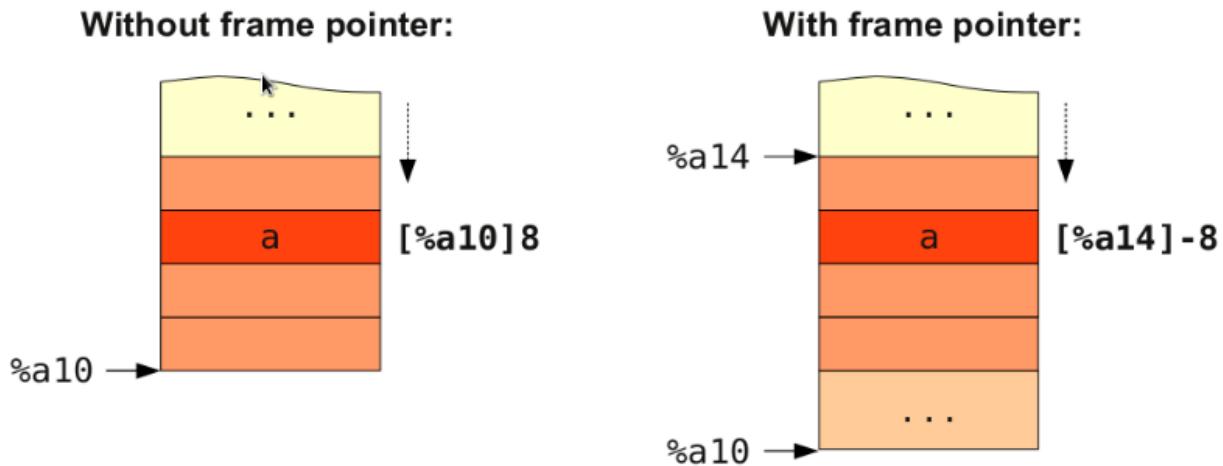


Figure 3.6: Addressing of a variable `a` located on the stack. If the stack frame has a variable size, slot must be addressed relative to the frame pointer

```
.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)
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 parsing IRs and counting how many virtual registers is assigned to local variables. After that, it calls `emitPrologue()`. This function will emit machine instructions to adjust `sp` (stack pointer register) for local variables. For our example, it will emit the instructions,

```
addiu $sp, $sp, -8
```

The `emitEpilogue` will emit “`addiu $sp, $sp, 8`”, where 8 is the stack size.

Since Instruction Selection and Register Allocation occurs before Prologue/Epilogue Code Insertion, `eliminateFrameIndex()` is called after instruction selection and register allocated. It translates the frame index of local variable (%1 and %2 in the following example) into stack offset according the frame index order upward (stack grow up downward from high address to low address, 0(\$sp) is the top, 52(\$sp) is the bottom) as follows,

```
define i32 @main() nounwind uwtable {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    ...
}
```

```

store i32 0, i32* %1
store i32 5, i32* %2, align 4
...
ret i32 0

=> # BB#0:
    addiu $sp, $sp, -56
$tmp1:
    addiu $3, $zero, 0
    st $3, 52($sp) // %1 is the first frame index local variable, so allocate
                     // in 52($sp)
    addiu $2, $zero, 5
    st $2, 48($sp) // %2 is the second frame index local variable, so
                     // allocate in 48($sp)
...
ret $lr

```

The Prologue and Epilogue functions as follows,

[Index/chapters/Chapter3_5/Cpu0SEFrameLowering.h](#)

```

bool hasReservedCallFrame(const MachineFunction &MF) const override;

void determineCalleeSaves(MachineFunction &MF, BitVector &SavedRegs,
                           RegScavenger *RS) const override;

```

[Index/chapters/Chapter3_5/Cpu0SEFrameLowering.cpp](#)

```

void Cpu0SEFrameLowering::emitPrologue(MachineFunction &MF,
                                         MachineBasicBlock &MBB) const {
    assert(&MF.front() == &MBB && "Shrink-wrapping not yet supported");
    MachineFrameInfo *MFI = MF.getFrameInfo();
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

    const Cpu0SEInstrInfo &TII =
        *static_cast<const Cpu0SEInstrInfo*>(STI.getInstrInfo());

    MachineBasicBlock::iterator MBBI = MBB.begin();
    DebugLoc dl = MBBI != MBB.end() ? MBBI->getDebugLoc() : DebugLoc();
    unsigned SP = Cpu0::SP;

    // 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();
    MachineLocation DstML, SrcML;

    // Adjust stack.
    TII.adjustStackPtr(Cpu0FI, SP, -StackSize, MBB, MBBI);

    // emit ".cfi_def_cfa_offset StackSize"

```

```

unsigned CFIIndex = MMI.addFrameInst(
    MCCFIInstruction::createDefCfaOffset(nullptr, -StackSize));
BuildMI(MBB, MBBI, dl, TII.get(TargetOpcode::CFI_INSTRUCTION))
    .addCFIIndex(CFIIndex);

const std::vector<CalleeSavedInfo> &CSI = MFI->getCalleeSavedInfo();

if (CSI.size()) {
    // Find the instruction past the last instruction that saves a callee-saved
    // register to the stack.
    for (unsigned i = 0; i < CSI.size(); ++i)
        ++MBBI;

    // Iterate over list of callee-saved registers and emit .cfi_offset
    // directives.
    for (std::vector<CalleeSavedInfo>::const_iterator I = CSI.begin(),
          E = CSI.end(); I != E; ++I) {
        int64_t Offset = MFI->getObjectOffset(I->getFrameIdx());
        unsigned Reg = I->getReg();
        {
            // Reg is in CPURegs.
            unsigned CFIIndex = MMI.addFrameInst(MCCFIInstruction::createOffset(
                nullptr, MRI->getDwarfRegNum(Reg, 1), Offset));
            BuildMI(MBB, MBBI, dl, TII.get(TargetOpcode::CFI_INSTRUCTION))
                .addCFIIndex(CFIIndex);
        }
    }
}

void Cpu0SEFrameLowering::emitEpilogue(MachineFunction &MF,
                                         MachineBasicBlock &MBB) const {
    MachineBasicBlock::iterator MBBI = MBB.getLastNonDebugInstr();
    MachineFrameInfo *MFI = MF.getFrameInfo();
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

    const Cpu0SEInstrInfo &TII =
        *static_cast<const Cpu0SEInstrInfo*>(MF.getSubtarget().getInstrInfo());

    DebugLoc dl = MBBI->getDebugLoc();
    unsigned SP = Cpu0::SP;

    // Get the number of bytes from FrameInfo
    uint64_t StackSize = MFI->getStackSize();

    if (!StackSize)
        return;

    // Adjust stack.
    TII.adjustStackPtr(Cpu0FI, 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();
}

// 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 {
    TargetFrameLowering::determineCalleeSaves(MF, SavedRegs, RS);
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
    MachineRegisterInfo& MRI = MF.getRegInfo();

    if (MF.getFrameInfo()->hasCalls())
        setAliasRegs(MF, SavedRegs, Cpu0::LR);

    return;
}

```

[Index/chapters/Chapter3_5/Cpu0MachineFunction.h](#)

```

unsigned getIncomingArgSize() const { return IncomingArgSize; }

bool callsEhReturn() const { return CallsEhReturn; }

void createEhDataRegsFI();

unsigned getMaxCallFrameSize() const { return MaxCallFrameSize; }
void setMaxCallFrameSize(unsigned S) { MaxCallFrameSize = S; }

```

After adding these Prologue and Epilogue functions, and build with Chapter3_5/. Now we are ready to compile our example code ch3.bc and ouput cpu0 assembly as follows,

```

118-165-78-12:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm -debug ch3.bc -o -
...
.section .mdebug.abi32
.previous
.file "ch3.bc"
.text
.globl main
.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)
addiu $sp, $sp, 8
ret $lr
.set macro
.set reorder
.end main
$tmp2:
.size main, ($tmp2)-main
.cfi_endproc

```

To see how the ‘**DAG->DAG Pattern Instruction Selection**’ work in llc, let’s compile with llc -debug option and get the following result. The DAGs which before and after instruction selection stage are shown as follows,

```

118-165-78-12:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm -debug ch3.bc -o -
Args: /Users/Jonathan/llvm/test/cmake_debug_build/Debug/bin/llc -march=cpu0
-relocation-model=pic -filetype=asm -debug ch3.bc -o -
...
Optimized legalized selection DAG: BB#0 'main:'
SelectionDAG has 8 nodes:
0x7fbe4082d010: i32 = Constant<0> [ORD=1] [ID=1]

0x7fbe4082d410: i32 = Register %V0 [ID=4]

0x7fbe40410668: ch = EntryToken [ORD=1] [ID=0]

0x7fbe4082d010: <multiple use>
0x7fbe4082d110: i32 = FrameIndex<0> [ORD=1] [ID=2]

0x7fbe4082d210: i32 = undef [ORD=1] [ID=3]

0x7fbe4082d310: ch = store 0x7fbe40410668, 0x7fbe4082d010, 0x7fbe4082d110,
0x7fbe4082d210<ST4[%1]> [ORD=1] [ID=5]

0x7fbe4082d410: <multiple use>
0x7fbe4082d010: <multiple use>
0x7fbe4082d510: ch,glue = CopyToReg 0x7fbe4082d310, 0x7fbe4082d410,
0x7fbe4082d010 [ID=6]

0x7fbe4082d510: <multiple use>
0x7fbe4082d410: <multiple use>
0x7fbe4082d510: <multiple use>
0x7fbe4082d610: ch = Cpu0ISD::Ret 0x7fbe4082d510, 0x7fbe4082d410,
0x7fbe4082d510:1 [ID=7]

===== Instruction selection begins: BB#0 ''
Selecting: 0x7fbe4082d610: ch = Cpu0ISD::Ret 0x7fbe4082d510, 0x7fbe4082d410,
0x7fbe4082d510:1 [ID=7]

ISEL: Starting pattern match on root node: 0x7fbe4082d610: ch = Cpu0ISD::Ret
0x7fbe4082d510, 0x7fbe4082d410, 0x7fbe4082d510:1 [ID=7]

Morphed node: 0x7fbe4082d610: ch = RET 0x7fbe4082d410, 0x7fbe4082d510,
0x7fbe4082d510:1

```

```

ISEL: Match complete!
=> 0x7fbe4082d610: ch = RET 0x7fbe4082d410, 0x7fbe4082d510, 0x7fbe4082d510:1

Selecting: 0x7fbe4082d510: ch,glue = CopyToReg 0x7fbe4082d310, 0x7fbe4082d410,
0x7fbe4082d010 [ID=6]

=> 0x7fbe4082d510: ch,glue = CopyToReg 0x7fbe4082d310, 0x7fbe4082d410,
0x7fbe4082d010

Selecting: 0x7fbe4082d310: ch = store 0x7fbe40410668, 0x7fbe4082d010,
0x7fbe4082d110, 0x7fbe4082d210<ST4[%1]> [ORD=1] [ID=5]

ISEL: Starting pattern match on root node: 0x7fbe4082d310: ch = store 0x7fbe40410668,
0x7fbe4082d010, 0x7fbe4082d110, 0x7fbe4082d210<ST4[%1]> [ORD=1] [ID=5]

    Initial Opcode index to 166
    Morphed node: 0x7fbe4082d310: ch = ST 0x7fbe4082d010, 0x7fbe4082d710,
0x7fbe4082d810, 0x7fbe40410668<Mem:ST4[%1]> [ORD=1]

ISEL: Match complete!
=> 0x7fbe4082d310: ch = ST 0x7fbe4082d010, 0x7fbe4082d710, 0x7fbe4082d810,
0x7fbe40410668<Mem:ST4[%1]> [ORD=1]

Selecting: 0x7fbe4082d410: i32 = Register %V0 [ID=4]

=> 0x7fbe4082d410: i32 = Register %V0

Selecting: 0x7fbe4082d010: i32 = Constant<0> [ORD=1] [ID=1]

ISEL: Starting pattern match on root node: 0x7fbe4082d010: i32 =
Constant<0> [ORD=1] [ID=1]

    Initial Opcode index to 1201
    Morphed node: 0x7fbe4082d010: i32 = ADDiu 0x7fbe4082d110, 0x7fbe4082d810 [ORD=1]

ISEL: Match complete!
=> 0x7fbe4082d010: i32 = ADDiu 0x7fbe4082d110, 0x7fbe4082d810 [ORD=1]

Selecting: 0x7fbe40410668: ch = EntryToken [ORD=1] [ID=0]

=> 0x7fbe40410668: ch = EntryToken [ORD=1]

===== Instruction selection ends:

```

Summary above translation into Table: Chapter 3 .bc IR instructions.

Table 3.2: Chapter 3 .bc IR instructions

.bc	Optimized legalized selection DAG	Cpu0
constant 0	constant 0	addiu
store	store	st
ret	Cpu0ISD::Ret	ret

From above llc -debug display, we see the **store** and **ret** are translated into **store** and **Cpu0ISD::Ret** in stage Optimized legalized selection DAG, and then translated into Cpu0 instructions **st** and **ret** finally. Since store use **constant 0 (store i32 0, i32* %1 in this example)**, the constant 0 will be translated into “**addiu \$2, \$zero, 0**” via the following pattern defined in Cpu0InstrInfo.td.

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

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

Ibdex/chapters/Chapter3_4/Cpu0InstrInfo.td

```
let Predicates = [Ch3_4] 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 ADDiu defined before ORi, so it takes the priority. Of course, if the ORi is defined first, the it will translate into “ori” instruction.

At this point, we have translated the very simple main() function with “return 0;” single instruction. The Cpu0AnalyzeImmediate.cpp and the Cpu0InstrInfo.td instructions defined in Chapter3_4, which take care the 32 bits stack size adjustments.

The Cpu0AnalyzeImmediate.cpp written in recursive with a little complicate in logic. Anyway, the recursive skills is used in the front end compile book, you should familiar with it. Instead 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 3.3: 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 ~ 0xffff8	• 0x8000	• addiu \$sp, \$sp, -32768;	• addiu \$1, \$zero, 1; • shl \$1, \$1, 16; • addiu \$1, \$1, -32768; • addu \$sp, \$sp, \$1;
x10000 ~ 0xffffffff8	• 0x7fffffff8	• 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, the 0x7ff9 to 0x7fff is 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 = 0xffffffff7.
2. “shl \$1, \$1, 28;” => \$1 = 0x70000000.
3. “addiu \$1, \$1, -32768” => \$1 = (0x70000000 + 0xfffff8000) => \$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 + 0xfffff8000) => \$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 3.4: 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 ~ 0xffff8	• 0x8000	• addiu \$sp, \$sp, -32768;	• ori \$1, \$zero, 32768; • addu \$sp, \$sp, \$1;
x10000 ~ 0xffffffff8	• 0x7fffffff8	• 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-proepilog.ll include the large frame test.

Run Chapter3_5 with ch3_2.cpp will get the following result.

```
118-165-78-12:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm -debug ch3_2.bc
-o -
...
.section .mdebug.abi32
.previous
.file "ch3_2.bc"
.text
.globl main
.align 2
.type main,@function
.ent main           # @main
main:
.frame $fp,24,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -24
addiu $2, $zero, 0
st $2, 20($fp)
addiu $2, $zero, 5
```

```

        st    $2, 16($fp)
        addiu $2, $zero, 2
        st    $2, 12($fp)
        ld    $2, 16($fp)
        addiu $2, $2, 2
        st    $2, 8($fp)
        ld    $2, 12($fp)
        addiu $2, $2, 1
        st    $2, 4($fp) // $2 = 3
        ld    $3, 8($fp) // $3 = 7
        addu $2, $3, $2 // $2 = 7+3
        addiu $sp, $sp, 24
        ret   $lr
        .set  macro
        .set  reorder
        .end  main
$tmp1:
.size main, ($tmp1)-main
    
```

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 leaf 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 it by grep -R "<datadag>" 'find src/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 occurred 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:
//
// 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;
}
    
```

Ibdex/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<i32> {
    let PrintMethod = "printMemOperand";
    let MIOperandInfo = (ops CPUREgs, 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
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 possibly 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/cmake_debug_build/
Debug/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
...
Post-RA pseudo instruction expansion pass
...
Cpu0 Assembly Printer

```

Table 3.5: Functions for llvm backend stage

Stage	Function
Before CPU0 DAG->DAG Pattern Instruction Selection	<ul style="list-style-type: none"> • Cpu0TargetLowering::LowerFormalArguments • Cpu0TargetLowering::LowerReturn
Instruction selection	<ul style="list-style-type: none"> • Cpu0DAGToDAGISel::Select
Prologue/Epilogue Insertion & Frame Finalization	<ul style="list-style-type: none"> • Cpu0FrameLowering.cpp • Cpu0RegisterInfo::eliminateFrameIndex()
Cpu0 Assembly Printer	<ul style="list-style-type: none"> • 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 pass 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 assembler for cpu0 which only support **ld, st, addiu, ori, lui, addu, shl** and **ret** 8 instructions.

We are satisfied with this result. But you may think “After so many codes 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 back end 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, CMakeLists.txt and LLVMBuild.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 back end 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 back end 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.

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 in some steps of optimization can be displayed by the graphic tool of Graphviz which supply useful information with graphic view. Logic instructions support will come after arithmetic section. In spite of llvm backend handle the IR only, we get the IR from the corresponding C operators with designed C example code. Through compiling with C code, readers can know exactly what C statements are handled with each chapter's appending code. Instead of focusing on classes relationship in this backend structure of last chapter, readers should focus on the mapping of C operators and 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 need them, from this chapter.

4.1 Arithmetic

The code added in Chapter4_1/ to support arithmetic instructions as follows,

[Index/chapters/Chapter4_1/MCTargetDesc/Cpu0BaseInfo.h](#)

```
/// getCpu0RegisterNumbering - Given the enum value for some register,  
/// return the number that it corresponds to.  
inline static unsigned getCpu0RegisterNumbering(unsigned RegEnum)  
{  
    switch (RegEnum) {  
        ...
```

```

    case Cpu0::HI:
        return 18;
    case Cpu0::LO:
        return 19;

    ...
}
}

```

Ibdex/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, const std::string &CPU,
                            const std::string &FS, bool little,
                            const Cpu0TargetMachine &_TM) :
    ...

    EnableOverflow = EnableOverflowOpt;

    ...
}

```

Ibdex/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 shamrt = 0;
    }
}

let Predicates = [Ch4_1] in {
// Mul, Div
class Mult<bits<8> op, string instr_asm, InstrItinClass itin,
                           RegisterClass RC, list<Register> DefRegs>:
    FL<op, (outs), (ins RC:$ra, RC:$rb),
        !strconcat(instr_asm, "\t$ra, $rb"), [], itin> {

```

```

let imm16 = 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>:
    FL<op, (outs), (ins RC:$ra, RC:$rb),
        !strconcat(instr_asm, "\t$ra, $rb"),
        [(opNode RC:$ra, RC:$rb)], itin> {
    let imm16 = 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>:
    FL<op, (outs RC:$ra), (ins),
        !strconcat(instr_asm, "\t$ra"), [], IIHiLo> {
    let rb = 0;
    let imm16 = 0;
    let Uses = UseRegs;
    let hasSideEffects = 0;
}

// Move to Lo/Hi
class MoveToLOHI<bits<8> op, string instr_asm, RegisterClass RC,
                      list<Register> DefRegs>:
    FL<op, (outs), (ins RC:$ra),
        !strconcat(instr_asm, "\t$ra"), [], IIHiLo> {
    let rb = 0;
    let imm16 = 0;
    let Defs = DefRegs;
    let hasSideEffects = 0;
}

// Move from C0 (co-processor 0) Register
class MoveFromC0<bits<8> op, string instr_asm, RegisterClass RC>:
    FL<op, (outs), (ins RC:$ra, C0Regs:$rb),
        !strconcat(instr_asm, "\t$ra, $rb"), [], IIAlu> {
    let imm16 = 0;
    let hasSideEffects = 0;
}

// Move to C0 Register
class MoveToC0<bits<8> op, string instr_asm, RegisterClass RC>:
    FL<op, (outs C0Regs:$ra), (ins RC:$rb),
        !strconcat(instr_asm, "\t$ra, $rb"), [], IIAlu> {
    let imm16 = 0;
    let hasSideEffects = 0;
}

```

```

// Move from C0 register to C0 register
class C0Move<bits<8> op, string instr_asm>:
    FL<op, (outs C0Regs:$ra), (ins C0Regs:$rb),
        !strconcat(instr_asm, "\t$ra, $rb"), [], IIAlu> {
    let imm16 = 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 ashv llvm IR instruction of .bc
def ROL       : shift_rotate_imm32<0x1b, 0x01, "rol", rotl>;
def ROR       : shift_rotate_imm32<0x1c, 0x01, "ror", rotr>;
def SRA       : shift_rotate_imm32<0x1d, 0x00, "sra", sra>;
}

let Predicates = [Ch4_1] in {
// srl is IR node for lshv llvm IR instruction of .bc
def SHR       : shift_rotate_imm32<0x1f, 0x00, "shr", srl>;
def SRAV      : shift_rotate_reg<0x20, 0x00, "srav", sra, CPURegs>;
def SHLV      : shift_rotate_reg<0x21, 0x00, "shlv", shl, CPURegs>;
def SHRV      : shift_rotate_reg<0x22, 0x00, "shrv", srl, CPURegs>;
def ROLV      : shift_rotate_reg<0x23, 0x01, "rolv", rotl, CPURegs>;
def RORV      : shift_rotate_reg<0x24, 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">;
}

```

Ibdex/chapters/Chapter4_1/Cpu0ISelLowering.h

```
SDValue PerformDAGCombine(SDNode *N, DAGCombinerInfo &DCI) const override;
```

Ibdex/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) {
  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();
}
```

Ibdex/chapters/Chapter4_1/Cpu0RegisterInfo.td

```
let Namespace = "Cpu0" in {

  // Hi/Lo registers number and name
  def HI    : Register<"hi">, DwarfRegNum<[18]>;
  def LO    : Register<"lo">, DwarfRegNum<[19]>;

}

...
// Hi/Lo Registers class
def HILO  : RegisterClass<"Cpu0", [i32], 32, (add HI, LO)>;
```

Ibdex/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]>]>,
] >;
```

Ibdex/chapters/Chapter4_1/Cpu0SEISelDAGToDAG.h

```
std::pair<SDNode*, SDNode*> SelectMULT(SDNode *N, unsigned Opc, SDLoc DL,
                                         EVT Ty, bool HasLo, bool HasHi);
```

Ibdex/chapters/Chapter4_1/Cpu0SEISelDAGToDAG.cpp

```

/// Select multiply instructions.
std::pair<SDNode*, SDNode*>
Cpu0SEISelDAGToDAGISel::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);
}

std::pair<bool, SDNode*> Cpu0SEISelDAGISel::selectNode(SDNode *Node) {
    unsigned Opcode = Node->getOpcode();
    SDLoc DL(Node);

    /**
     // Instruction Selection not handled by the auto-generated
     // tablegen selection should be handled here.
     /**
     SDNode *Result;

    /**
     // 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);
        Result = SelectMULT(Node, MultOpc, DL, NodeTy, false, true).second;
        return std::make_pair(true, Result);
    }

    case ISD::Constant: {
        const ConstantSDNode *CN = dyn_cast<ConstantSDNode>(Node);
        unsigned Size = CN->getValueSizeInBits(0);

        if (Size == 32)
            break;
    }
}

```

```

    }
    ...
}
```

[Index/chapters/Chapter4_1/Cpu0SEInstrInfo.h](#)

```

void copyPhysReg(MachineBasicBlock &MBB,
                  MachineBasicBlock::iterator MI, DebugLoc DL,
                  unsigned DestReg, unsigned SrcReg,
                  bool KillSrc) const override;
```

[Index/chapters/Chapter4_1/Cpu0SEInstrInfo.cpp](#)

```

void Cpu0SEInstrInfo::
copyPhysReg(MachineBasicBlock &MBB,
            MachineBasicBlock::iterator I, DebugLoc DL,
            unsigned DestReg, unsigned 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 like 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 compiler translate it into “shift with sign extension fill”, for example, Mips **sra** is the instruction. Following is the Microsoft web site 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 don’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	0xfffffffffe i.e. -2	0xfffffffffe i.e. -2
signed example after x >> 1	0x7fffffff i.e. 2G-1	0xffffffff i.e. -1
unsigned example before x >> 1	0xfffffffffe i.e. 4G-2	0xfffffffffe i.e. 4G-2
unsigned example after x >> 1	0x7fffffff i.e. 2G-1	0xffffffff i.e. 4G-1

lshr: Logical SHift Right

ashr: Arithmetic SHift right

srl: Shift Right Logically

sra: Shift Right Arithmetically

shr: SHift Right

¹ http://llvm.org/docs/doxygen/html/structllvm_1_1InstrStage.html

If we consider the $x \gg 1$ definition is $x = x/2$ for compiler implementation. Then as you can see from Table: C operator \gg 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 \ll implementation

Description	Shift with zero filled
symbol in .bc	shl
symbol in IR node	shl
Mips instruction	sll
Cpu0 instruction	shl
signed example before $x \ll 1$	0x40000000 i.e. 1G
signed example after $x \ll 1$	0x80000000 i.e. -2G
unsigned example before $x \ll 1$	0x40000000 i.e. 1G
unsigned example after $x \ll 1$	0x80000000 i.e. 2G

Again, consider the $x \ll 1$ definition is $x = x*2$. From Table: C operator \ll 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 $x \ll 1$ and the x result is not out of range, no matter operand x is signed or unsigned integer.

Microsoft implementation references here ².

The ‘ashr’ Instruction reference here ³, ‘lshr’ reference here ⁴.

The sra, 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,

Ibdex/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* %a, align 4
    %2 = load 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
    %7 = ash shr i32 %1, 2
    %8 = lshr i32 %1, 30
    %9 = shl i32 1, %2
    %10 = ash shr i32 128, %2
    %11 = ash shr 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
```

² <http://msdn.microsoft.com/en-us/library/336xbhc%28v=vs.80%29.aspx>

³ <http://llvm.org/docs/LangRef.html#ashr-instruction>

⁴ <http://llvm.org/docs/LangRef.html#lshr-instruction>

```
%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/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch4_math.bc -o -

```

Ibdex/output/ch4_math.s

```
.text
.section .mdebug.abi32
.previous
.file      "ch4_math.ll"
.globl    _Z9test_mathv
.align    2
.type     _Z9test_mathv,@function
.ent      _Z9test_mathv           # @_Z9test_mathv
_Z9test_mathv:
.cfi_startproc
.frame    $sp,8,$lr
.mask     0x00000000,0
.set      noreorder
.set      nomacro
# BB#0:
addiu   $sp, $sp, -8
$ttmp0:
.cfi_def_cfa_offset 8
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
shr    $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
.set      macro
.set      reorder
.end    _Z9test_mathv
```

```
$tmp1:  
.size      _Z9test_mathv, ($tmp1)-_Z9test_mathv  
.cfi_endproc
```

File ch4_1_1.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_1.cpp

```
int test_math()  
{  
    int a = 5;  
    int b = 2;  
    unsigned int a1 = -5;  
    int c, d, e, f, g, h, i;  
    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 = 0xffffffff6 = -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  
  
    return (c+d+e+f+int(f1)+g+(int)g1+h+(int)h1+i+(int)i1);  
// 7+3+10+20-10+1+3+32+4+4+0 = 74  
}
```

Cpu0 instructions add and sub will trigger overflow exception while addu and subu truncate overflow value directly. Compile ch4_1_2.cpp with llc -cpu0-enable-overflow=true will generate add and sub instructions as follows,

lbdex/input/ch4_1_2.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;
int c = 0;

c = a - b;

return 0;
}

118-165-78-12:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch4_1_2.cpp -emit-llvm -o ch4_1_2.bc
118-165-78-12:input Jonathan$ llvm-dis ch4_1_2.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/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
-cpu0-enable-overflow=true ch4_1_2.bc -o -
...
add      $3, $4, $3
...
sub      $3, $4, $3
...

```

In modern CPU, programmers are used to using truncate overflow instructions in C program regard add and sub instructions. 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 Graphviz. The llc graphic displaying with tool Graphviz is introduced in this section. The graphic displaying is more readable by eye 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.
 - view-sched-dags displays the DAG before Scheduling.
-

⁵ <http://llvm.org/docs/CodeGenerator.html#selectiondag-instruction-selection-process>

By tracking llc -debug, you can see the DAG translation steps 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/
cmake_debug_build/Debug/bin/llc -view-dag-combine1-dags -march=cpu0
-relocation-model=pic -filetype=asm ch4_2.bc -o ch4_2.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 [Figure 4.1](#).

[Figure 4.1](#) is the stage of “Initial selection DAG”. List the other view options and their corresponding DAG translation stages as follows,

Note: llc Graphviz options and corresponding DAG translation stage

- 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 with a given C operator.

4.1.3 Operator % and /

The DAG of %

Example input code ch4_2.cpp which contains the C operator “%” and it's corresponding llvm IR, as follows,

Ibdex/input/ch4_2.cpp

```
int test_mod()
{
    int b = 11;
// unsigned int b = 11;

    b = (b+1)%12;
```

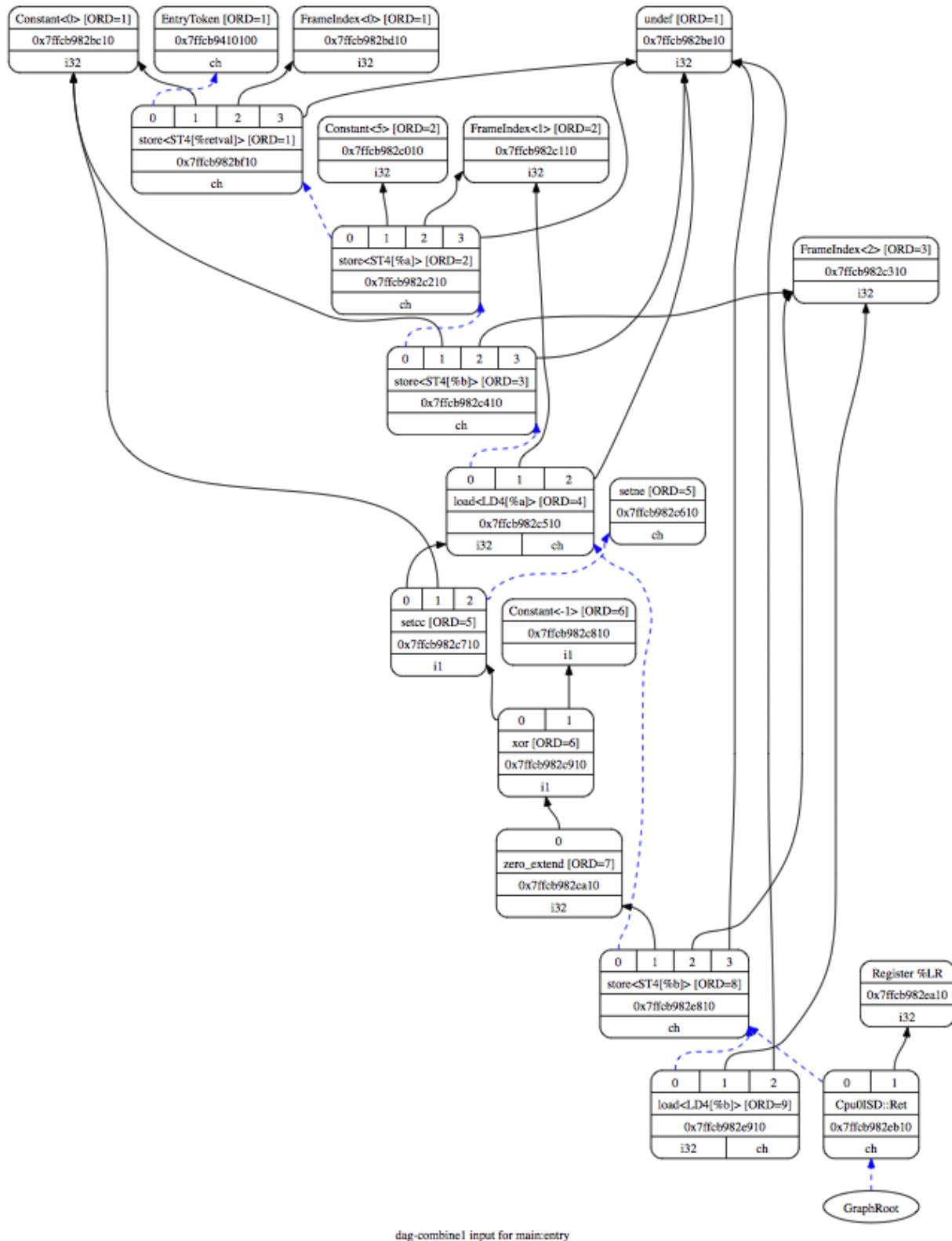


Figure 4.1: llc option -view-dag-combine1-dags graphic view

```

    return b;
}

...
define i32 @_Z8test_modv() #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 here⁶. 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_4/ with input file ch4_2.bc via option llc -view-isel-dags, will get the following error message and the llvm DAGs of Figure 4.2 below.

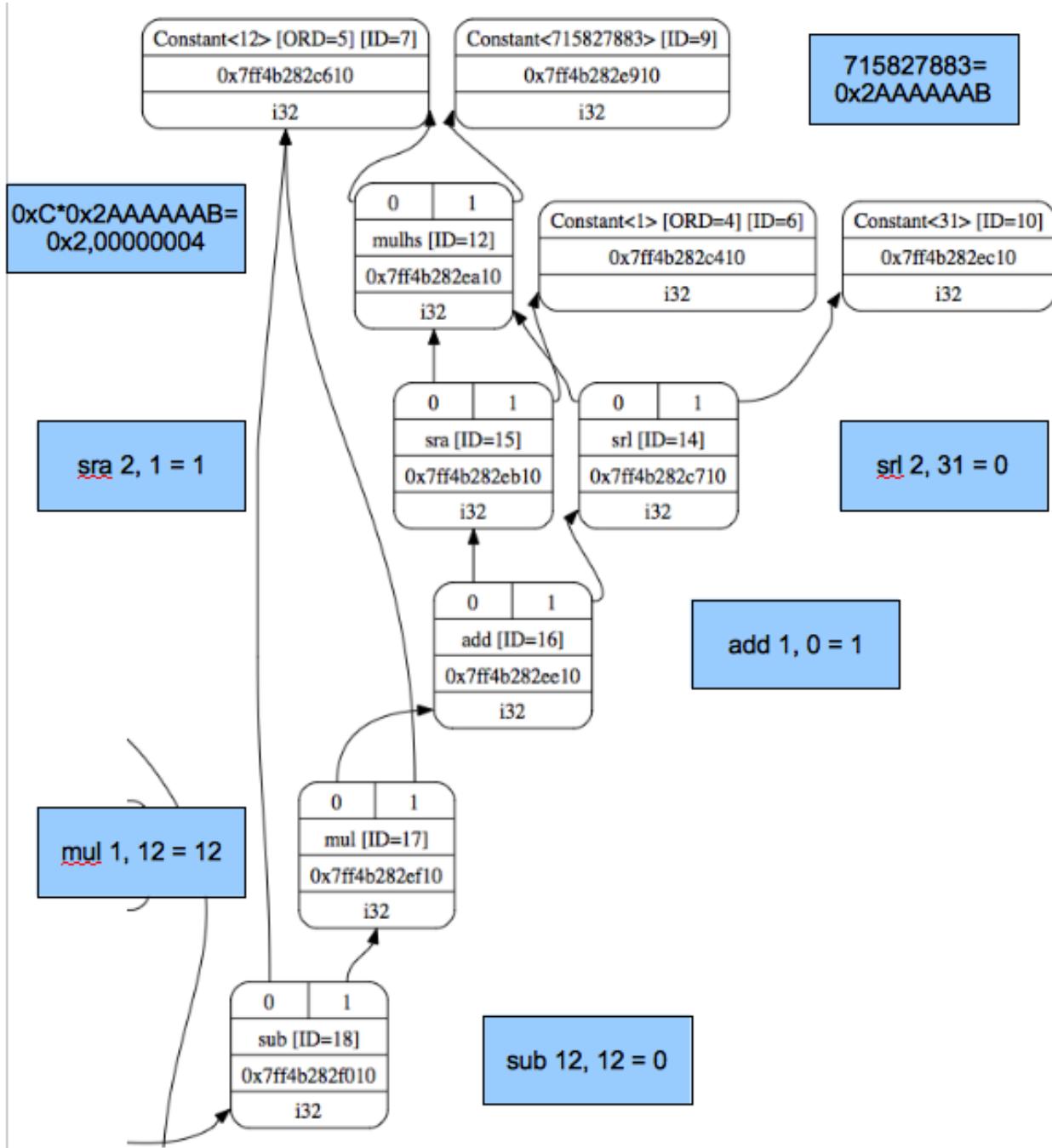
```

118-165-79-37:input Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/Debug/bin/llc -march=cpu0 -view-isel-dags -relocation-model=
pic -filetype=asm ch4_2.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]

```

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 Figure 4.2. The DAGs of generated result is verified and explained by calculating the value in each node. The 0xC*0x2AAAAAAAB=0x2,00000004, (mulhs 0xC, 0x2AAAAAAAB) meaning get the Signed mul high word (32bits). Multiply with 2 operands of 1 word

⁶ <http://llvm.org/docs/LangRef.html#srem-instruction>

Figure 4.2: `ch4_2.bc` DAG

size probably generate the 2 word size of result (0x2, 0xAAAAAAAB). The high word result, 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:  
    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_2.cpp as well as `llc -view-sched-dags` option to get [Figure 4.3](#). Instruction SMMUL will get the high word of multiply result.

The following is the result of run above changes with ch4_2.bc.

```
118-165-66-82:input Jonathan$ /Users/Jonathan/llvm/test/cmake_
debug_build/Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch4_2.bc -o -
    .section .mdebug.abi32
    .previous
    .file "ch4_2.bc"
    .text
    .globl main
    .align 2
    .type main,@function
    .ent main # @main
main:
    .cfi_startproc
    .frame $fp,8,$lr
    .mask 0x00000000,0
    .set noreorder
    .set nomacro
# BB#0: # %entry
    addiu $sp, $sp, -8
$tmp1:
    .cfi_def_cfa_offset 8
    addiu $2, $zero, 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
    .set macro
    .set reorder
    .end main
$tmp2:
    .size main, ($tmp2)-main
    .cfi_endproc
```

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_2.cpp.

Uses SMMUL instruction to get the high word of multiplication result is adopted in ARM.

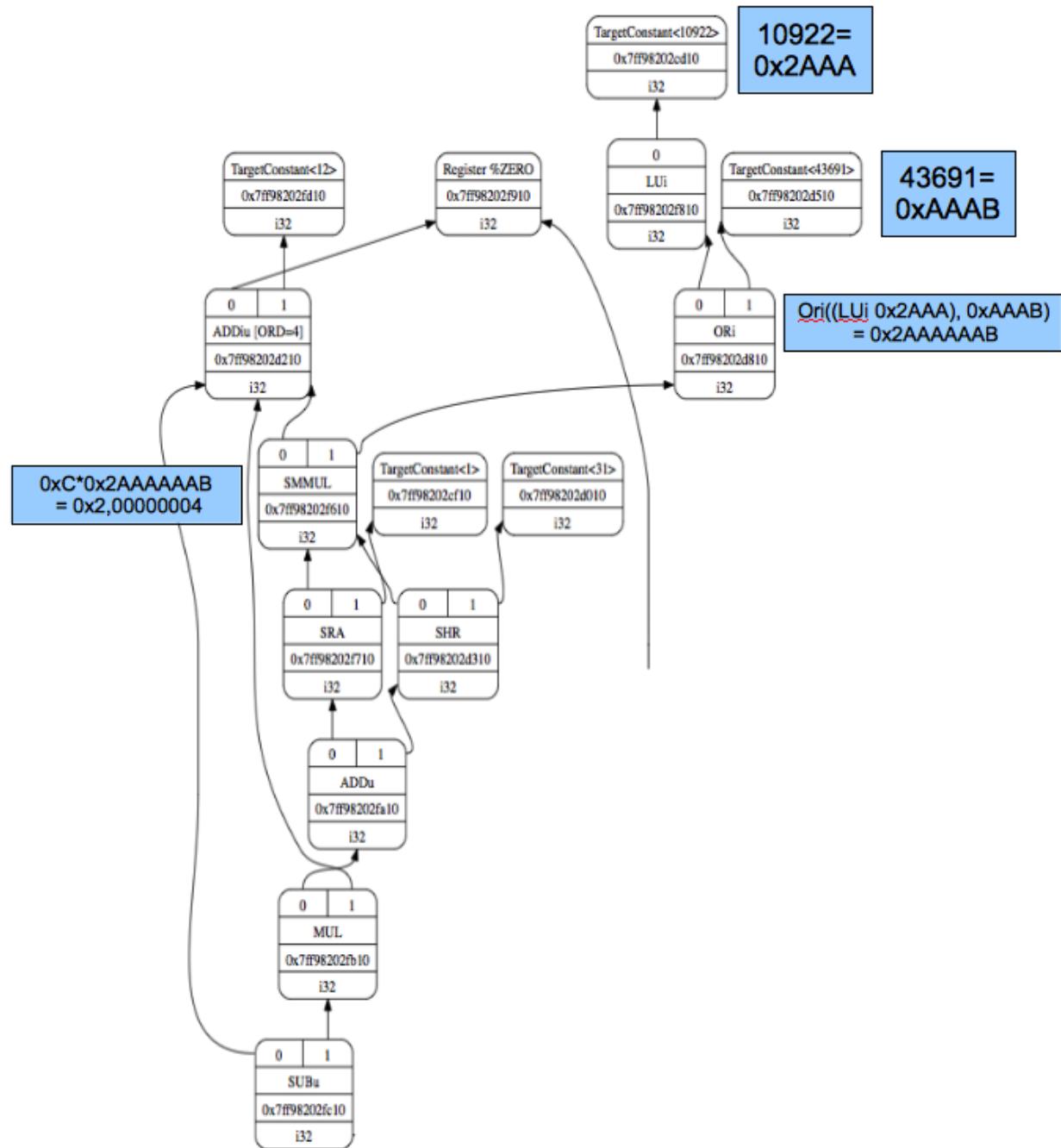


Figure 4.3: DAG for ch4_2.bc with ARM style SMMUL

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 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. [Figure 4.4](#) and ch4_2.cpu0.s are the results of compile ch4_2.bc.

```
118-165-66-82:input Jonathan$ cat ch4_2.cpu0.s
.section .mdebug.abi32
.previous
.file "ch4_2.bc"
.text
.globl _Z8test_modv
.align 2
.type _Z8test_modv,@function
.ent _Z8test_modv          # @_Z8test_modv
_Z8test_modv:
.frame $sp,8,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# 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)
```

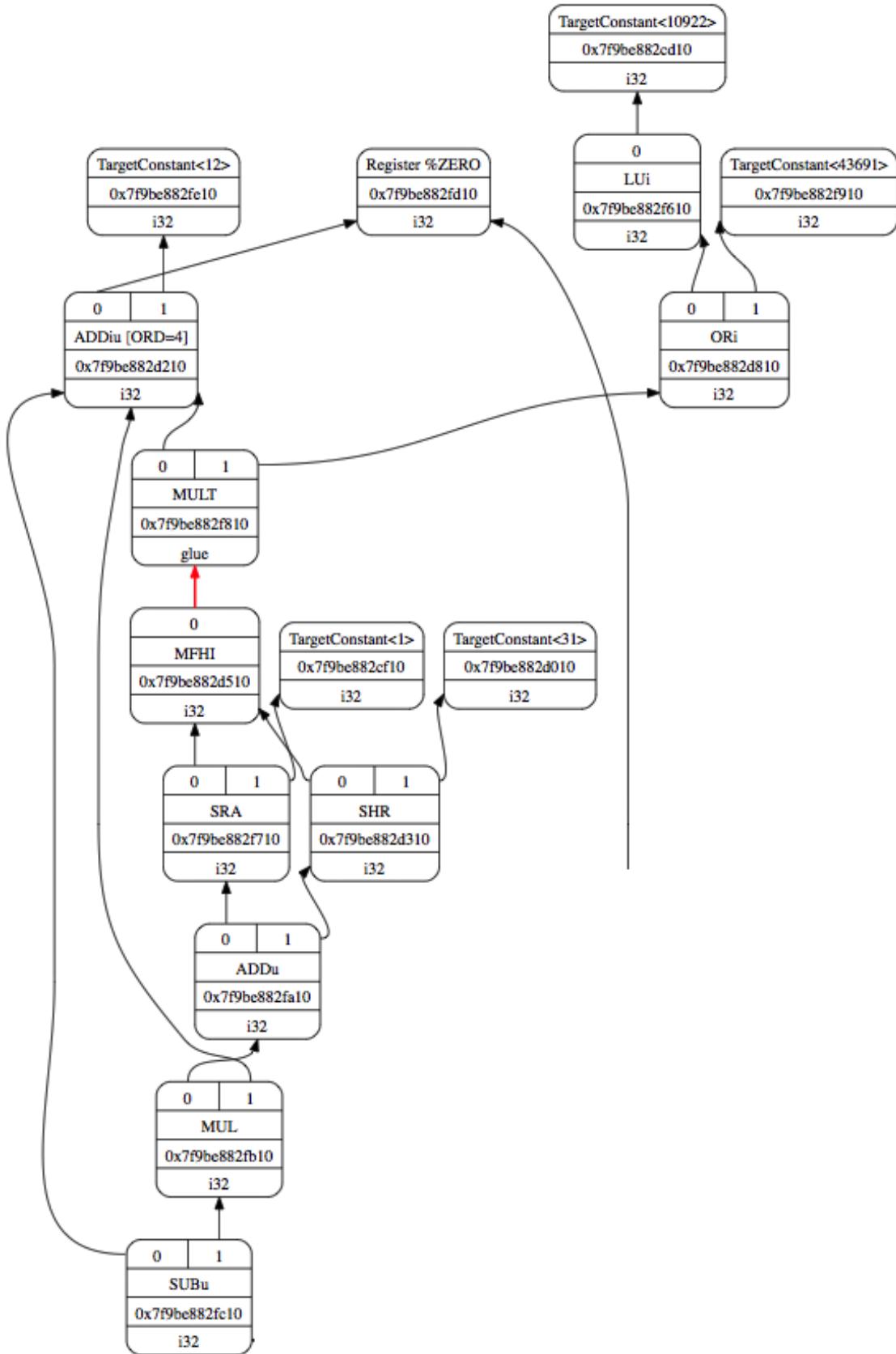


Figure 4.4: DAG for ch4_2.bc with Mips style MULT

```

addiu $sp, $sp, 8
ret $lr
.set macro
.set reorder
.end _Z8test_modv
$tmp1:
.size _Z8test_modv, ($tmp1)-_Z8test_modv

```

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.
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.

Ibdex/input/ch4_2_1.cpp

```

int test_mod()
{
    int b = 11;
    int a = 12;

    b = (b+1)%a;

    return b;
}

```

If we run with ch4_2_1.cpp, the “**div**” cannot be gotten for operator “%”. It still uses “**multiplication**” instead of “**div**” in ch4_2_1.cpp because llvm do “**Constant Propagation Optimization**” in this. The ch4_2_2.cpp will get the “div” result for operator “%” but it cannot be compiled at this point. It needs “function call argument support” in Chapter 9 of Function call. The ch4_2_2.cpp can get the “**div**” for “%” result since it makes llvm “**Constant Propagation Optimization**” useless in it.

Ibdex/input/ch4_2_2.cpp

```

int test_mod(int c)
{
    int b = 11;

    b = (b+1)%c;
}

```

```
    return b;
}

118-165-77-79:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch4_2_2.cpp -emit-llvm -o ch4_2_2.bc
118-165-77-79:input Jonathan$ /Users/Jonathan/llvm/test/cmake_
debug_build/Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch4_2_2.bc -o -
...
div $zero, $3, $2
mflo $2
...
```

To explains how work with “**div**”, let’s run Chapter9_3 with ch4_2_2.cpp as follows,

```
118-165-83-58:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch4_2_2.cpp -I/Applications/Xcode.app/Contents/Developer/Platforms/
MacOSX.platform/Developer/SDKs/MacOSX10.8.sdk/usr/include/ -emit-llvm -o
ch4_2_2.bc
118-165-83-58:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm -debug ch4_2_2.bc -o -
Args: /Users/Jonathan/llvm/test/cmake_debug_build/Debug/bin/llc -march=cpu0
-relocation-model=pic -filetype=asm -debug ch4_2_2.bc -o -
```

```
==== _Z8test_modi
Initial selection DAG: BB#0 '_Z8test_modi:'
SelectionDAG has 21 nodes:
0x7fed68410bc8: ch = EntryToken [ORD=1]

0x7fed6882cb10: i32 = undef [ORD=1]

0x7fed6882cd10: i32 = FrameIndex<0> [ORD=1]

0x7fed6882ce10: i32 = Constant<0>

0x7fed6882d110: i32 = FrameIndex<1> [ORD=2]

0x7fed68410bc8: <multiple use>
0x7fed68410bc8: <multiple use>
0x7fed6882ca10: i32 = FrameIndex<-1> [ORD=1]

0x7fed6882cb10: <multiple use>
0x7fed6882cc10: i32, ch = load 0x7fed68410bc8, 0x7fed6882ca10,
0x7fed6882cb10<LD4[FixedStack-1]> [ORD=1]

0x7fed6882cd10: <multiple use>
0x7fed6882cb10: <multiple use>
0x7fed6882cf10: ch = store 0x7fed68410bc8, 0x7fed6882cc10, 0x7fed6882cd10,
0x7fed6882cb10<ST4[%1]> [ORD=1]

0x7fed6882d010: i32 = Constant<11> [ORD=2]

0x7fed6882d110: <multiple use>
0x7fed6882cb10: <multiple use>
0x7fed6882d210: ch = store 0x7fed6882cf10, 0x7fed6882d010, 0x7fed6882d110,
0x7fed6882cb10<ST4[%b]> [ORD=2]

0x7fed6882d210: <multiple use>
```

```

0x7fed6882d110: <multiple use>
0x7fed6882cb10: <multiple use>
0x7fed6882d310: i32,ch = load 0x7fed6882d210, 0x7fed6882d110,
0x7fed6882cb10<LD4[%b]> [ORD=3]

0x7fed6882d210: <multiple use>
0x7fed6882cd10: <multiple use>
0x7fed6882cb10: <multiple use>
0x7fed6882d610: i32,ch = load 0x7fed6882d210, 0x7fed6882cd10,
0x7fed6882cb10<LD4[%1]> [ORD=5]

0x7fed6882d310: <multiple use>
0x7fed6882d610: <multiple use>
0x7fed6882d810: ch = TokenFactor 0x7fed6882d310:1, 0x7fed6882d610:1 [ORD=7]

0x7fed6882d310: <multiple use>
0x7fed6882d410: i32 = Constant<1> [ORD=4]

0x7fed6882d510: i32 = add 0x7fed6882d310, 0x7fed6882d410 [ORD=4]

0x7fed6882d610: <multiple use>
0x7fed6882d710: i32 = srem 0x7fed6882d510, 0x7fed6882d610 [ORD=6]

0x7fed6882d110: <multiple use>
0x7fed6882cb10: <multiple use>
0x7fed6882fc10: ch = store 0x7fed6882d810, 0x7fed6882d710, 0x7fed6882d110,
0x7fed6882cb10<ST4[%b]> [ORD=7]

0x7fed6882fe10: i32 = Register %V0

0x7fed6882fc10: <multiple use>
0x7fed6882fe10: <multiple use>
0x7fed6882fc10: <multiple use>
0x7fed6882d110: <multiple use>
0x7fed6882cb10: <multiple use>
0x7fed6882fd10: i32,ch = load 0x7fed6882fc10, 0x7fed6882d110,
0x7fed6882cb10<LD4[%b]> [ORD=8]

0x7fed6882ff10: ch,glue = CopyToReg 0x7fed6882fc10, 0x7fed6882fe10,
0x7fed6882fd10

0x7fed6882ff10: <multiple use>
0x7fed6882fe10: <multiple use>
0x7fed6882ff10: <multiple use>
0x7fed68830010: ch = Cpu0ISD::Ret 0x7fed6882ff10, 0x7fed6882fe10,
0x7fed6882ff10:1

Replacing.1 0x7fed6882fd10: i32,ch = load 0x7fed6882fc10, 0x7fed6882d110,
0x7fed6882cb10<LD4[%b]> [ORD=8]

With: 0x7fed6882d710: i32 = srem 0x7fed6882d510, 0x7fed6882d610 [ORD=6]
and 1 other values

Replacing.1 0x7fed6882d310: i32,ch = load 0x7fed6882d210, 0x7fed6882d110,
0x7fed6882cb10<LD4[%b]> [ORD=3]

With: 0x7fed6882d010: i32 = Constant<11> [ORD=2]
and 1 other values

```

```
Replacing.3 0x7fed6882d810: ch = TokenFactor 0x7fed6882d210,  
0x7fed6882d610:1 [ORD=7]
```

```
With: 0x7fed6882d610: i32, ch = load 0x7fed6882d210, 0x7fed6882cd10,  
0x7fed6882cb10<LD4[%1]> [ORD=5]
```

```
Replacing.3 0x7fed6882d510: i32 = add 0x7fed6882d010, 0x7fed6882d410 [ORD=4]
```

```
With: 0x7fed6882d810: i32 = Constant<12>
```

```
Replacing.1 0x7fed6882cc10: i32, ch = load 0x7fed68410bc8, 0x7fed6882ca10,  
0x7fed6882cb10<LD4[FixedStack-1] (align=8)> [ORD=1]
```

```
With: 0x7fed6882cc10: i32, ch = load 0x7fed68410bc8, 0x7fed6882ca10,  
0x7fed6882cb10<LD4[FixedStack-1] (align=8)> [ORD=1]  
and 1 other values
```

```
Optimized lowered selection DAG: BB#0 '_Z8test_modi:'
```

```
SelectionDAG has 16 nodes:
```

```
0x7fed68410bc8: ch = EntryToken [ORD=1]
```

```
0x7fed6882cb10: i32 = undef [ORD=1]
```

```
0x7fed6882cd10: i32 = FrameIndex<0> [ORD=1]
```

```
0x7fed6882d110: i32 = FrameIndex<1> [ORD=2]
```

```
0x7fed68410bc8: <multiple use>
```

```
0x7fed68410bc8: <multiple use>
```

```
0x7fed6882ca10: i32 = FrameIndex<-1> [ORD=1]
```

```
0x7fed6882cb10: <multiple use>
```

```
0x7fed6882cc10: i32, ch = load 0x7fed68410bc8, 0x7fed6882ca10,
```

```
0x7fed6882cb10<LD4[FixedStack-1] (align=8)> [ORD=1]
```

```
0x7fed6882cd10: <multiple use>
```

```
0x7fed6882cb10: <multiple use>
```

```
0x7fed6882cf10: ch = store 0x7fed68410bc8, 0x7fed6882cc10, 0x7fed6882cd10,  
0x7fed6882cb10<ST4[%1]> [ORD=1]
```

```
0x7fed6882d010: i32 = Constant<11> [ORD=2]
```

```
0x7fed6882d110: <multiple use>
```

```
0x7fed6882cb10: <multiple use>
```

```
0x7fed6882d210: ch = store 0x7fed6882cf10, 0x7fed6882d010, 0x7fed6882d110,  
0x7fed6882cb10<ST4[%b]> [ORD=2]
```

```
0x7fed6882cd10: <multiple use>
```

```
0x7fed6882cb10: <multiple use>
```

```
0x7fed6882d610: i32, ch = load 0x7fed6882d210, 0x7fed6882cd10,  
0x7fed6882cb10<LD4[%1]> [ORD=5]
```

```
0x7fed6882d810: i32 = Constant<12>
```

```
0x7fed6882d610: <multiple use>
```

```
0x7fed6882d710: i32 = srem 0x7fed6882d810, 0x7fed6882d610 [ORD=6]
```

```

0x7fed6882fe10: i32 = Register %V0

0x7fed6882d610: <multiple use>
0x7fed6882d710: <multiple use>
0x7fed6882d110: <multiple use>
0x7fed6882cb10: <multiple use>
0x7fed6882fc10: ch = store 0x7fed6882d610:1, 0x7fed6882d710, 0x7fed6882d110,
0x7fed6882cb10<ST4[%b]> [ORD=7]

0x7fed6882fe10: <multiple use>
0x7fed6882d710: <multiple use>
0x7fed6882ff10: ch,glue = CopyToReg 0x7fed6882fc10, 0x7fed6882fe10,
0x7fed6882d710

0x7fed6882ff10: <multiple use>
0x7fed6882fe10: <multiple use>
0x7fed6882ff10: <multiple use>
0x7fed68830010: ch = Cpu0ISD::Ret 0x7fed6882ff10, 0x7fed6882fe10,
0x7fed6882ff10:1

Type-legalized selection DAG: BB#0 '_Z8test_modi:'
SelectionDAG has 16 nodes:
...
0x7fed6882d610: i32, ch = load 0x7fed6882d210, 0x7fed6882cd10,
0x7fed6882cb10<LD4[%1]> [ORD=5] [ID=-3]

0x7fed6882d810: i32 = Constant<12> [ID=-3]

0x7fed6882d610: <multiple use>
0x7fed6882d710: i32 = srem 0x7fed6882d810, 0x7fed6882d610 [ORD=6] [ID=-3]
...
Legalized selection DAG: BB#0 '_Z8test_modi:'
SelectionDAG has 16 nodes:
0x7fed68410bc8: ch = EntryToken [ORD=1] [ID=0]

0x7fed6882cb10: i32 = undef [ORD=1] [ID=2]

0x7fed6882cd10: i32 = FrameIndex<0> [ORD=1] [ID=3]

0x7fed6882d110: i32 = FrameIndex<1> [ORD=2] [ID=5]

0x7fed6882fe10: i32 = Register %V0 [ID=6]
...
0x7fed6882d810: i32 = Constant<12> [ID=7]

0x7fed6882d610: <multiple use>
0x7fed6882ce10: i32,i32 = sdivrem 0x7fed6882d810, 0x7fed6882d610

Optimized legalized selection DAG: BB#0 '_Z8test_modi:'
SelectionDAG has 18 nodes:
...
0x7fed6882d510: i32 = Register %HI

0x7fed6882d810: i32 = Constant<12> [ID=7]

0x7fed6882d610: <multiple use>

```

```

0x7fed6882d410: glue = Cpu0ISD::DivRem 0x7fed6882d810, 0x7fed6882d610

0x7fed6882d310: i32,ch,glue = CopyFromReg 0x7fed68410bc8, 0x7fed6882d510,
0x7fed6882d410
...

===== Instruction selection begins: BB#0 ''
...
Selecting: 0x7fed6882d410: glue = Cpu0ISD::DivRem 0x7fed6882d810,
0x7fed6882d610 [ID=13]

ISEL: Starting pattern match on root node: 0x7fed6882d410: glue =
Cpu0ISD::DivRem 0x7fed6882d810, 0x7fed6882d610 [ID=13]

Initial Opcode index to 1355
Morphed node: 0x7fed6882d410: i32,glue = SDIV 0x7fed6882d810, 0x7fed6882d610

ISEL: Match complete!
=> 0x7fed6882d410: i32,glue = SDIV 0x7fed6882d810, 0x7fed6882d610
...

```

According above DAGs translation messages, llvm do the following things:

1. Reduce DAG nodes in stage “Optimized lowered selection DAG” (Replacing ... displayed before “Optimized lowered selection DAG: BB#0 ‘_Z8test_modi:entry’ ”). Since SSA form has some redundant nodes for store and load, them 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 “0x7fd25b830710: i32 = Register %HI” and “CopyFromReg 0x7fd25b410e18, 0x7fd25b830710, 0x7fd25b830910” 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	IR/DAG/instruction
.bc	srem	
Legalized selection DAG	sdivrem	
Optimized legalized selection DAG	Cpu0ISD::DivRem	CopyFromReg xx, Hi, xx
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)

Item 2 as above, is triggered by code “setOperationAction(ISD::SREM, MVT::i32, Expand);” in Cpu0ISelLowering.cpp. About **Expand** please ref. ⁷ and ⁸. Item 3 is triggered by code “setTargetDAGCombine(ISD::SDIVREM);” in Cpu0ISelLowering.cpp. Item 4 is did by PerformDivRemCombine() which

⁷ <http://llvm.org/docs/WritingAnLLVMBBackend.html#expand>

⁸ <http://llvm.org/docs/CodeGenerator.html#selectiondag-legalizetypes-phase>

called by PerformDAGCombine() since the `%` corresponding `srem` makes the “N->hasAnyUseOfValue(1)” to true in PerformDivRemCombine(). Then, it creates “CopyFromReg 0x7fd25b410e18, 0x7fd25b830710, 0x7fd25b830910”. When using “`/`” in C, it will make “N->hasAnyUseOfValue(0)” to true. For `sdivrem`, `sdiv` makes “N->hasAnyUseOfValue(0)” true while `srem` makes “N->hasAnyUseOfValue(1)” true.

Above items will change the DAG when `llc` running. After that, the pattern match defined in Chapter4_1/Cpu0InstrInfo.td will translate `Cpu0ISD::DivRem` into `div`; and “`CopyFromReg 0x7fd25b410e18, Register %H, 0x7fd25b830910`” to `mfhi`.

The `ch4_3.cpp` is for `/` `div` operator test.

4.1.4 Rotate instructions

Chapter4_1 include the rotate operations translation. The instructions “`rol`”, “`ror`”, “`rolv`” and “`rорv`” defined in `Cpu0InstrInfo.td` handle the translation. Compile `ch4_1_3.cpp` will get Cpu0 “`rol`” instruction.

`lbdex/input/ch4_1_3.cpp`

```
int test_rotate_left()
{
    unsigned int a = 8;
    int result = ((a << 30) | (a >> 2));

    return result;
}

#ifndef TEST_ROXV

int test_rotate_left1(unsigned int a, int n)
{
    int result = ((a << n) | (a >> (32 - n)));

    return result;
}

int test_rotate_right(unsigned int a, int n)
{
    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_3.cpp -emit-llvm -o ch4_1_3.bc
114-43-200-122:input Jonathan$ llvm-dis ch4_1_3.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 = ash r 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/cmake_debug_build/Debug/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch4_1_3.bc -o -
...
rol $2, $2, 30
...
```

Instructions “rolv” and “rorv” cannot be tested at this moment, they need function argument passing support. The “#ifdef TEST_ROXV” part of ch4_1_3.cpp can be tested after Chapter Function call. 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 **&**, **l**, **^**, **!**, **==**, **!=**, **<**, **<=**, **>** 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_5.cpp as below.

Index/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;
//#if CH >= CH10_1
let DecoderMethod = "DecodeCMPInstruction";

// 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;
}

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, uimml6, immZExt16, CPURegs>;
}

let Predicates = [Ch4_2] in {
def XORi      : ArithLogicI<0x0e, "xori", xor, uimml6, immZExt16, CPURegs>;
}

let Predicates = [Ch4_2] in {
let Predicates = [HasCmp] in {
def CMP       : CmpInstr<0x10, "cmp", 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>;
}

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:$in),
// 1: in == 0; 0: in != 0
          (XORi CPURegs:$in, 1)>;
}

// setcc patterns

let Predicates = [Ch4_2] in {
// 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 (CMP 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 (CMP 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 (CMP RC:$lhs, RC:$rhs), 1), 1)>;
}

// setcc for slt instruction
multiclass SeteqPatsSlt<RegisterClass RC, Instruction SLTiuOp, Instruction XOROp,
                           Instruction SLTuOp, Register ZEROReg> {
    // a == b
    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 SLTiOp> {
    def : Pat<(setge RC:$lhs, immSExt16:$rhs),
        (XORi (SLTiOp RC:$lhs, immSExt16:$rhs), 1)>;
    def : Pat<(setuge RC:$lhs, immSExt16:$rhs),
        (XORi (SLTiOp 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>;
defm : SetltPatsCmp<CPUREgs>;
defm : SetlePatsCmp<CPUREgs>;
defm : SetgtPatsCmp<CPUREgs>;
defm : SetgePatsCmp<CPUREgs>;
}
} // let Predicates = [Ch4_2]

```

Index/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_5.cpp

```
int test_andorxornot()
{
    int a = 5;
    int b = 3;
    int c = 0, d = 0, e = 0;

    c = (a & b); // c = 1
    d = (a | b); // d = 7
    e = (a ^ b); // e = 6
    b = !a; // b = 0

    return (c+d+e+b); // 14
}

int test_setxx()
{
    int a = 5;
    int b = 3;
    int c, d, e, f, g, h;

    c = (a == b); // seq, c = 0
    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_5.cpp -emit-llvm -o ch4_5.bc
114-43-204-152:input Jonathan$ llvm-dis ch4_5.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 il %tobool, true
%conv = zext il %lnot to i32
...
}

; Function Attrs: nounwind uwtable
define i32 @_Z10test_setxxv() #0 {
entry:
...
%cmp = icmp eq i32 %0, %1
%conv = zext il %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
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/cmake_debug_build/Debug/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -filetype=asm ch4_5.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
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/cmake_debug_build/
Debug/bin/llc -march=cpu0 -mcpu=cpu032II -relocation-model=pic -filetype=asm
ch4_5.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
!	<ul style="list-style-type: none"> %tobool = icmp ne i32 %6, 0 %lnot = xor i1 %tobool, true %conv = zext i1 %lnot to i32 	<ul style="list-style-type: none"> %lnot = (setcc %tobool, 0, seteq) %conv = (and %lnot, 1) • 	<ul style="list-style-type: none"> xor \$3, \$4, \$3
==	<ul style="list-style-type: none"> %cmp = icmp eq i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> %cmp = (setcc %0, %1, seteq) and %cmp, 1 	<ul style="list-style-type: none"> cmp \$sw, \$3, \$2 andi \$2, \$sw, 2 shr \$2, \$2, 1 andi \$2, \$2, 1
!=	<ul style="list-style-type: none"> %cmp = icmp ne i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> %cmp = (setcc %0, %1, setne) and %cmp, 1 	<ul style="list-style-type: none"> cmp \$sw, \$3, \$2 andi \$2, \$sw, 2 shr \$2, \$2, 1 andi \$2, \$2, 1
<	<ul style="list-style-type: none"> %cmp = icmp lt i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setlt) and %cmp, 1 	<ul style="list-style-type: none"> cmp \$sw, \$3, \$2 andi \$2, \$sw, 2 andi \$2, \$2, 1 andi \$2, \$2, 1
<=	<ul style="list-style-type: none"> %cmp = icmp le i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, settle) and %cmp, 1 	<ul style="list-style-type: none"> cmp \$sw, \$2, \$3 andi \$2, \$sw, 1 xori \$2, \$2, 1 andi \$2, \$2, 1
>	<ul style="list-style-type: none"> %cmp = icmp gt i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setgt) and %cmp, 1 	<ul style="list-style-type: none"> cmp \$sw, \$2, \$3 andi \$2, \$sw, 2 andi \$2, \$2, 1
>=	<ul style="list-style-type: none"> %cmp = icmp le i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, settle) and %cmp, 1 	<ul style="list-style-type: none"> 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
!	<ul style="list-style-type: none"> %tobool = icmp ne i32 %6, 0 %lnot = xor i1 %tobool, true %conv = zext i1 %lnot to i32 	<ul style="list-style-type: none"> %lnot = (setcc %tobool, 0, seteq) %conv = (and %lnot, 1) • 	<ul style="list-style-type: none"> xor \$3, \$4, \$3
==	<ul style="list-style-type: none"> %cmp = icmp eq i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> %cmp = (setcc %0, %1, seteq) and %cmp, 1 	<ul style="list-style-type: none"> xor \$2, \$3, \$2 sltiu \$2, \$2, 1 andi \$2, \$2, 1
!=	<ul style="list-style-type: none"> %cmp = icmp ne i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> %cmp = (setcc %0, %1, setne) and %cmp, 1 	<ul style="list-style-type: none"> xor \$2, \$3, \$2 sltu \$2, \$zero, 2 shr \$2, \$2, 1 andi \$2, \$2, 1
<	<ul style="list-style-type: none"> %cmp = icmp lt i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setlt) and %cmp, 1 	<ul style="list-style-type: none"> slt \$2, \$3, \$2 andi \$2, \$2, 1
<=	<ul style="list-style-type: none"> %cmp = icmp le i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, settle) and %cmp, 1 	<ul style="list-style-type: none"> slt \$2, \$3, \$2 xori \$2, \$2, 1 andi \$2, \$2, 1
>	<ul style="list-style-type: none"> %cmp = icmp gt i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setgt) and %cmp, 1 	<ul style="list-style-type: none"> slt \$2, \$3, \$2 andi \$2, \$2, 1
>=	<ul style="list-style-type: none"> %cmp = icmp le i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, settle) and %cmp, 1 	<ul style="list-style-type: none"> slt \$2, \$3, \$2 xori \$2, \$2, 1 andi \$2, \$2, 1

In relation operators ==, !=, ..., %0 = \$3 = 5, %1 = \$2 = 3 for ch4_5.cpp.

The “Optimized legalized selection DAG” is the last DAG stage just before the “instruction selection” as the 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.

Ibdex/input/ch4_6.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_6.cpp -emit-llvm -o ch4_6.bc
118-165-78-10:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch4_6.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/cmake_debug_build/
Debug/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch4_6.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_6.cpp 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 have hazard in instruction reorder since both of them use \$sw register but *llc -mcpu=cpu032II* has not⁹. The slti version can reorder as follows,

```

...
ld $2, 16($sp)
slti $2, $2, 2
andi $2, $2, 1
st $2, 8($sp)

```

⁹ See book Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)

```

ld $2, 20($sp)
sli $2, $2, 1
andi $2, $2, 1
st $2, 12($sp)
...
    
```

Chapter4_2 include instructions cmp and slt. Even 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.

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
>>	<ul style="list-style-type: none"> • ashtr • lshr 	<ul style="list-style-type: none"> • sra • srl 	<ul style="list-style-type: none"> • sra • shr
!	<ul style="list-style-type: none"> • %tobool = icmp ne i32 %0, 0 • %lnot = xor i1 %tobool, true 	<ul style="list-style-type: none"> • %lnot = (setcc %tobool, 0, seteq) • %conv = (and %lnot, 1) 	<ul style="list-style-type: none"> • %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)
%	<ul style="list-style-type: none"> • srem • sremu 	<ul style="list-style-type: none"> • Cpu0ISD::DivRem • Cpu0ISD::DivRemU 	<ul style="list-style-type: none"> • div • divu
(x<<n) (x>>32-n)	shl + lshr	rotl, rotr	rol, rolv, ror, rorv

GENERATING OBJECT FILES

- Translate into obj file
- ELF obj related code
- 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 to run Chapter4_2/ to translate it into obj code will get the error message as follows,

```
[Gamma@localhost 3]$ ~/llvm/test/cmake_debug_build/bin/
llc -march=cpu0 -relocation-model=pic -filetype=obj ch4_1.bc -o ch4_1.cpu0.o
~/llvm/test/cmake_debug_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,

```
[Gamma@localhost input]$ cat ch4_1.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)
    addiu $2, $zero, 0
    st $2, 28($fp)
...
[Gamma@localhost 3]$ ~/llvm/test/cmake_debug_build/bin/
llc -march=cpu0 -relocation-model=pic -filetype=obj ch4_1.bc -o ch4_1.cpu0.o
[Gamma@localhost input]$ objdump -s ch4_1.cpu0.o
```

```
ch4_1.cpu0.o:      file format elf32-big

Contents of section .text:
0000 09ddfffc8 09200005 022d0034 09200002 ..... . .-.4. ...
0010 022d0030 0920ffffb 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<....
```

```
[Gamma@localhost input]$ ~/llvm/test/
cmake_debug_build/bin/llc -march=cpu0el -relocation-model=pic -filetype=obj
ch4_1.bc -o ch4_1.cpu0el.o
[Gamma@localhost input]$ objdump -s ch4_1.cpu0el.o
```

```
ch4_1.cpu0el.o:      file format elf32-little
```

```
Contents of section .text:
0000 c8ffd09 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 0x09ddfffc8. 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(immediate 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 0x09ddfffc8 and the little endian is (B3, B2, B1, B0) = (09, dd, ff, c8), objdump from B0 to B3 is 0xc8ffd09.

5.2 ELF obj related code

To support elf obj generation, the following code changed and added to Chapter5_1.

[Index/chapters/Chapter5_1/MCTargetDesc/CMakeLists.txt](#)

```
Cpu0AsmBackend.cpp
Cpu0MCCodeEmitter.cpp
```

Cpu0ELFObjectWriter.cpp
Cpu0TargetStreamer.cpp

Ibdex/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 CPU0ASMBACKEND_H
#define CPU0ASMBACKEND_H

#include "Cpu0Config.h"

#include "MCTargetDesc/Cpu0FixupKinds.h"
#include "llvm/MC/MCAsmBackend.h"
#include "llvm/ADT/Triple.h"

namespace llvm {

class MCAssembler;
struct MCFixupKindInfo;
class Target;
class MCObjectWriter;

class Cpu0AsmBackend : public MCAsmBackend {
    Triple::OSType OSType;
    bool IsLittle; // Big or little endian

public:
    Cpu0AsmBackend(const Target &T, Triple::OSType _OSType, bool _isLittle)
        : MCAsmBackend(), OSType(_OSType), IsLittle(_isLittle) {}

    MCObjectWriter *createObjectWriter(raw_pwrite_stream &OS) const override;

    void applyFixup(const MCFixup &Fixup, char *Data, unsigned DataSize,
                   uint64_t Value, bool IsPCRel) 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 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;
}

/// RelaxInstruction - Relax the instruction in the given fragment
/// to the next wider instruction.
///
/// \param Inst - The instruction to relax, which may be the same
/// as the output.
/// \param [out] Res On return, the relaxed instruction.
void relaxInstruction(const MCInst &Inst, MCInst &Res) const override {}

/// @}

bool writeNopData(uint64_t Count, MCObjectWriter *OW) const override;
}; // class Cpu0AsmBackend

} // namespace

#endif

```

Index/chapters/Chapter5_1/MCTargetDesc/Cpu0AsmBackend.cpp

```

===== Cpu0ASMBbackend.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 and Cpu0ELFObjectWriter classes.
//
=====//
//

#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/MCOBJObjectWriter.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

//@adjustFixupValue {
// Prepare value for the target space for it
static unsigned adjustFixupValue(const MCFixup &Fixup, uint64_t Value,
                                MCContext *Ctx = nullptr) {

    unsigned Kind = Fixup.getKind();

    // 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_Local:
        // Get the higher 16-bits. Also add 1 if bit 15 is 1.
        Value = ((Value + 0x8000) >> 16) & 0xffff;
        break;
    }
}

return Value;
}
//@adjustFixupValue }

MCObjectWriter *
Cpu0AsmBackend::createObjectWriter(raw_pwrite_stream &OS) const {
    return createCpu0ELFObjectWriter(OS,
                                    MCELFObjectTargetWriter::getOSABI(OSType), IsLittle);
}

/// ApplyFixup - Apply the \arg Value for given \arg Fixup into the provided
/// data fragment, at the offset specified by the fixup and following the
/// fixup kind as appropriate.
void Cpu0AsmBackend::applyFixup(const MCFixup &Fixup, char *Data,
                               unsigned DataSize, uint64_t Value,
                               bool IsPCRel) const {
    MCFixupKind Kind = Fixup.getKind();
    Value = adjustFixupValue(Fixup, Value);

    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;
}

// Grab current value, if any, from bits.
uint64_t CurVal = 0;

for (unsigned i = 0; i != NumBytes; ++i) {
    unsigned Idx = IsLittle ? 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 = IsLittle ? i : (FullSize - 1 - i);
    Data[Offset + Idx] = (uint8_t) ((CurVal >> (i*8)) & 0xff);
}
}

//@getFixupKindInfo {
const MCFixupKindInfo &Cpu0AsmBackend::  

getFixupKindInfo(MCFixupKind Kind) const {
    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_Global",   0,      16,     0 },
        { "fixup_Cpu0_GOT_Local",    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 }

/// 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(uint64_t Count, MCObjectWriter *OW) const {
    return true;
}

// MCasmBackend
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);
}

```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0BaseInfo.h

```
#include "Cpu0FixupKinds.h"
```

Ibdex/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;

namespace {
    class Cpu0ELFOBJECTWriter : public MCELFObjectTargetWriter {
    public:
        Cpu0ELFOBJECTWriter(uint8_t OSABI);

        virtual ~Cpu0ELFOBJECTWriter();
    }
}
```

```

    unsigned GetRelocType(const MCValue &Target, const MCFixup &Fixup,
                         bool IsPCRel) const override;
    bool needsRelocateWithSymbol(const MCSymbol &Sym,
                                 unsigned Type) const override;
};

}

Cpu0ELFObjectWriter::Cpu0ELFObjectWriter(uint8_t OSABI)
: MCELFObjectTargetWriter(/*_is64Bit=false*/ false, OSABI, ELF::EM_CPU0,
  /*HasRelocationAddend*/ false) {}

Cpu0ELFObjectWriter::~Cpu0ELFObjectWriter() {}

//@GetRelocType {
unsigned Cpu0ELFObjectWriter::GetRelocType(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_Global:
    case Cpu0::fixup_Cpu0_GOT_Local:
        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;
    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 extremely conservative. This really needs to use a
// whitelist with a clear explanation for why each relocation 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;
}
}

MCObjectWriter *llvm::createCpu0ELFObjectWriter(raw_pwrite_stream &OS,
                                                uint8_t OSABI,
                                                bool IsLittleEndian) {
MCELFObjectTargetWriter *MOTW = new Cpu0ELFObjectWriter(OSABI);
return createELFObjectWriter(MOTW, OS, IsLittleEndian);
}

```

Ibdex/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_CPU0_CPU0FIXUPKINDS_H
#define LLVM_CPU0_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 same 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,

    // Pure lower 16 bit fixup resulting in - R_CPU0_GPREL16.
    fixup_Cpu0_GPREL16,

    // Global symbol fixup resulting in - R_CPU0_GOT16.
    fixup_Cpu0_GOT_Global,

    // Local symbol fixup resulting in - R_CPU0_GOT16.
    fixup_Cpu0_GOT_Local,

    // resulting in - R_CPU0_GOT_HI16
    fixup_Cpu0_GOT_HI16,

    // resulting in - R_CPU0_GOT_LO16
    fixup_Cpu0_GOT_LO16,

    // Marker
    LastTargetFixupKind,
    NumTargetFixupKinds = LastTargetFixupKind - FirstTargetFixupKind
};

//@Fixups }
} // namespace Cpu0
} // namespace llvm

#endif // LLVM_CPU0_CPU0FIXUPKINDS_H
```

Ibdex/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.
//
//=====//



#include "Cpu0Config.h"

#ifndef CPU0_MC_CODE_EMITTER_H
#define CPU0_MC_CODE_EMITTER_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;
    const MCInstrInfo &MCII;
    bool IsLittleEndian;

public:
    Cpu0MCCodeEmitter(const MCInstrInfo &mcii, MCContext &Ctx_, bool IsLittle)
        : MCII(mcii), 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'reated 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 machine
// operand requires relocation, record the relocation and return zero.
unsigned getMachineOpValue(const MCInst &MI, const MCOperand &MO,
                           SmallVectorImpl<MCFixup> &Fixups,
                           const MCSubtargetInfo &STI) const;

unsigned getMemEncoding(const MCInst &MI, unsigned OpNo,
                        SmallVectorImpl<MCFixup> &Fixups,
                        const MCSubtargetInfo &STI) const;

unsigned getExprOpValue(const MCE Expr *Expr, SmallVectorImpl<MCFixup> &Fixups,
                        const MCSubtargetInfo &STI) const;
}; // class Cpu0MCCodeEmitter
} // namespace llvm.

#endif // #if CH >= CH5_1

```

Ibdex/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/Cpu0MCTargetDesc.h"
#include "llvm/ADT/APFloat.h"
#include "llvm/MC/MCCodeEmitter.h"
#include "llvm/MC/MCE Expr.h"

```

```

#include "llvm/MC/MCInst.h"
#include "llvm/MC/MCInstrInfo.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

#define DEBUG_TYPE "mccodeemitter"

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
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::Binary) {
        Expr = static_cast<const MCBinaryExpr*>(Expr)->getLHS();
        Kind = Expr->getKind();
    }

    assert (Kind == MCExpr::SymbolRef);

    // 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 = getCPU0RegisterNumbering(Reg);
        return RegNo;
    } else if (MO.isImm()) {
        return static_cast<unsigned>(MO.getImm());
    } else if (MO.isFPIImm()) {
        return static_cast<unsigned>(APFloat(MO.getFPIImm())
            .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;
}

#include "Cpu0GenMCCodeEmitter.inc"

```

Ibdex/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 *createCpu0AsmBackendEB32(const Target &T,
                                         const MCRegisterInfo &MRI,
                                         const Triple &TT, StringRef CPU);
MCAsmBackend *createCpu0AsmBackendEL32(const Target &T,
                                         const MCRegisterInfo &MRI,
                                         const Triple &TT, StringRef CPU);

MCObjectWriter *createCpu0ELFObjectWriter(raw_pwrite_stream &OS,
                                           uint8_t OSABI,
                                           bool IsLittleEndian);

```

Ibdex/chapters/Chapter5_1/MCTargetDesc/Cpu0MCTargetDesc.cpp

```

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);
}

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);

#if CH >= CH5_1 //3
    // Register the MC Code Emitter
    TargetRegistry::RegisterMCCodeEmitter(TheCpu0Target,
                                         createCpu0MCCodeEmitterEB);
    TargetRegistry::RegisterMCCodeEmitter(TheCpu0elTarget,
                                         createCpu0MCCodeEmitterEL);

    // Register the asm backend.
    TargetRegistry::RegisterMCAsmBackend(TheCpu0Target,
                                         createCpu0AsmBackendEB32);
    TargetRegistry::RegisterMCAsmBackend(TheCpu0elTarget,
                                         createCpu0AsmBackendEL32);
#endif
}

```

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 whether 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.

Ibdex/chapters/Chapter2/Cpu0InstrInfo.td

```

// Address operand
def mem : Operand<i32> {
    let PrintMethod = "printMemOperand";
    let MIOperandInfo = (ops CPURegs, simm16);
    let EncoderMethod = "getMemEncoding";

```

```

}

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>;
}

```

The “let EncoderMethod = “getMemEncoding”;” in Cpu0InstrInfo.td as above will make 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.

5.3 Backend Target Registration Structure

Now, let’s examine Cpu0MCTargetDesc.cpp. Cpu0MCTargetDesc.cpp do the target registration as mentioned in “section Target Registration”¹ of the last chapter. Drawing the register functions and those classes it registered from [Figure 5.1](#) to [Figure 5.9](#) for explanation.

[Figure 5.1](#) to [Figure 5.4](#) had been defined in [Chapter4_2/](#) for assembly file generated support.

In [Figure 5.1](#), 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 its parent, back end reuses those code by inheritance.

In [Figure 5.2](#), instancing MCCCodeGenInfo, and initialize it by pass RM=Reloc::PIC when user compiling with position-independent code mode through command `llc -relocation-model=pic` or `llc` (default relocation mode is pic). Recall there are two addressing mode in system program book, one is PIC mode, the other is absolute addressing mode. MC stands for Machine Code.

In [Figure 5.3](#), 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. [Figure 5.4](#) is similar to [Figure 5.3](#), but it initializes the register information specified in Cpu0RegisterInfo.td. They share some values from instruction/register td description. No need to specify them again in Initialize routine if they are consistent with td description files.

[Figure 5.5](#), instancing two objects Cpu0MCCodeEmitter, one is for big endian and the other is for little endian. They take care the obj format generated. These functions are not defined in [Chapter4_2/](#) which support assembly code only.

[Figure 5.6](#), MCELFStreamer takes care the obj format also. [Figure 5.5](#) Cpu0MCCodeEmitter takes care code emitter while MCELFStreamer takes care the obj output streamer. [Figure 5.10](#) is MCELFStreamer inheritance tree. You can find a lot of operations in that inheritance tree.

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()`.

¹ <http://jonathan2251.github.io/lbd/llvmstructure.html#target-registration>

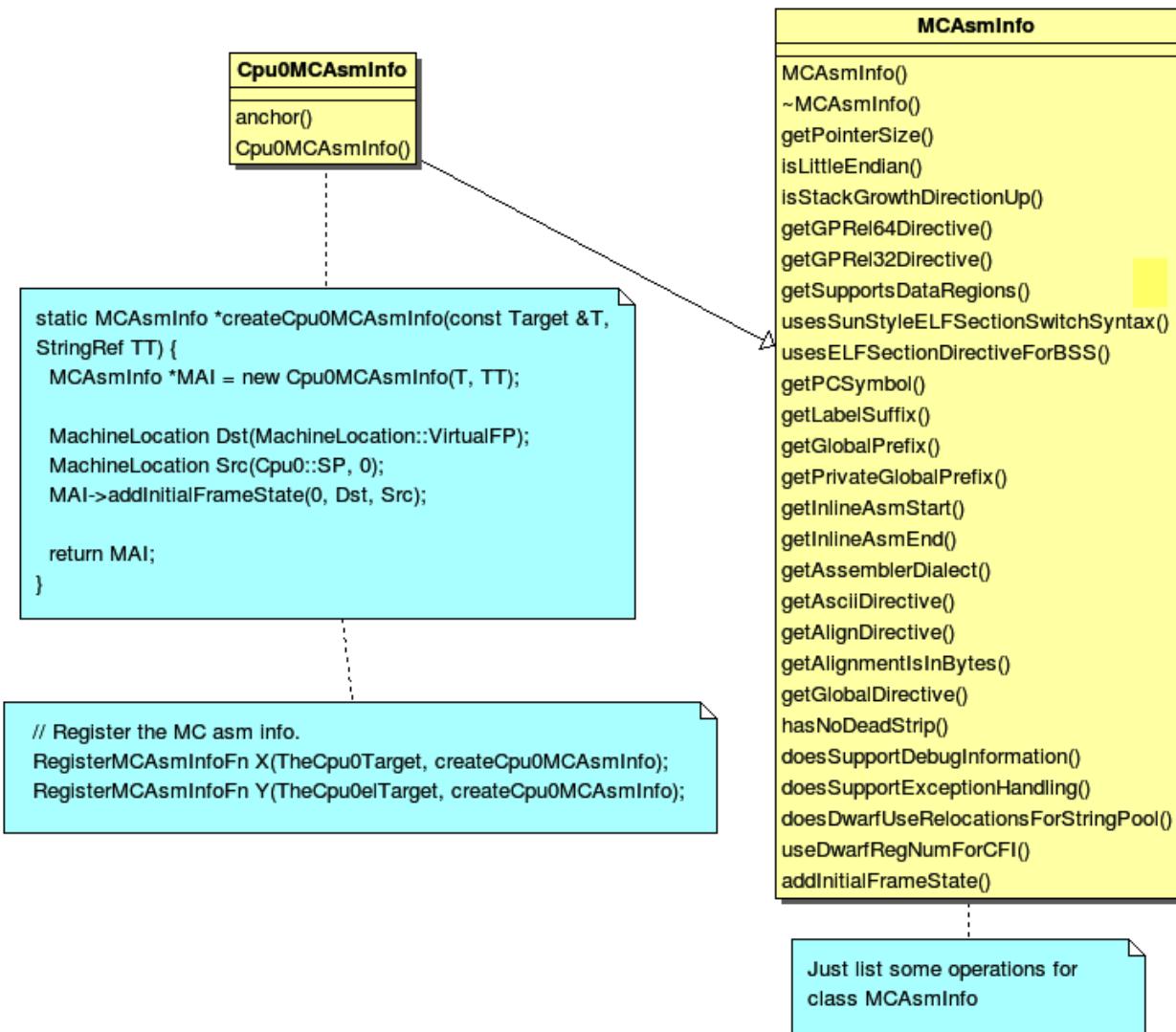


Figure 5.1: Register Cpu0MCAsmInfo

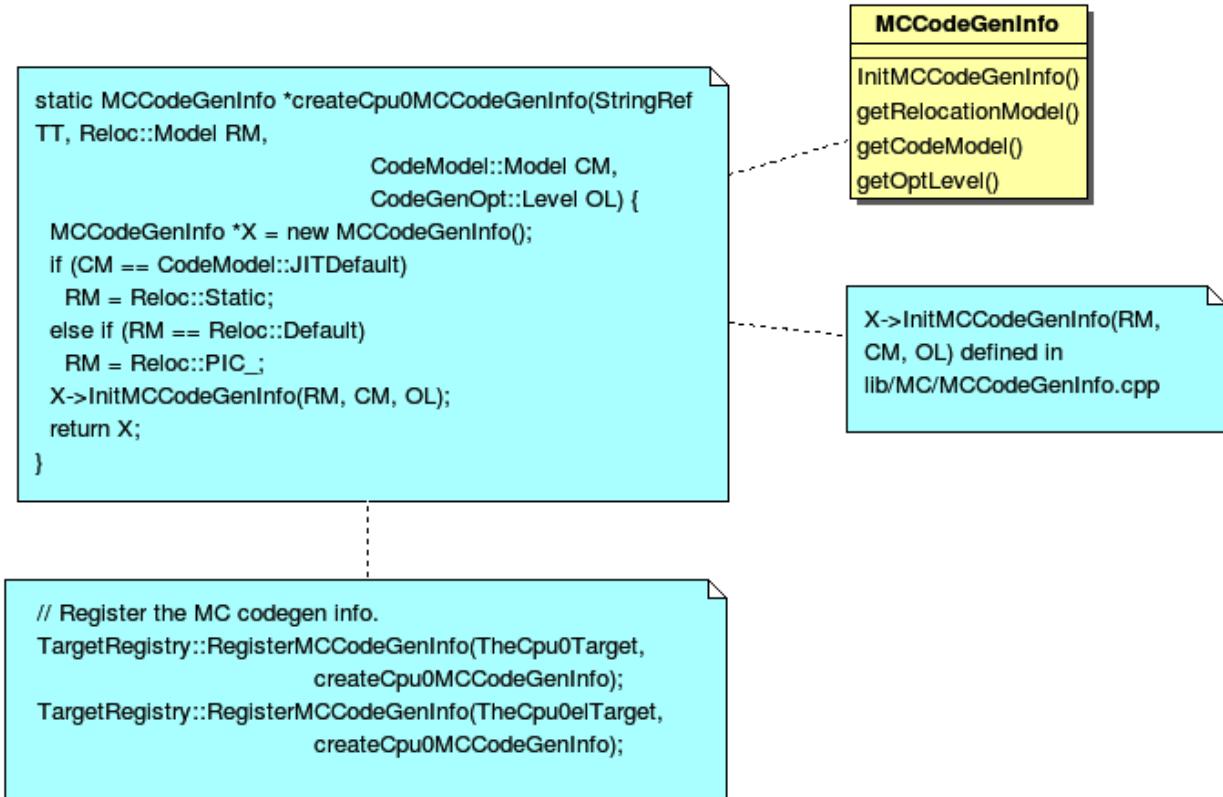


Figure 5.2: Register MCCCodeGenInfo

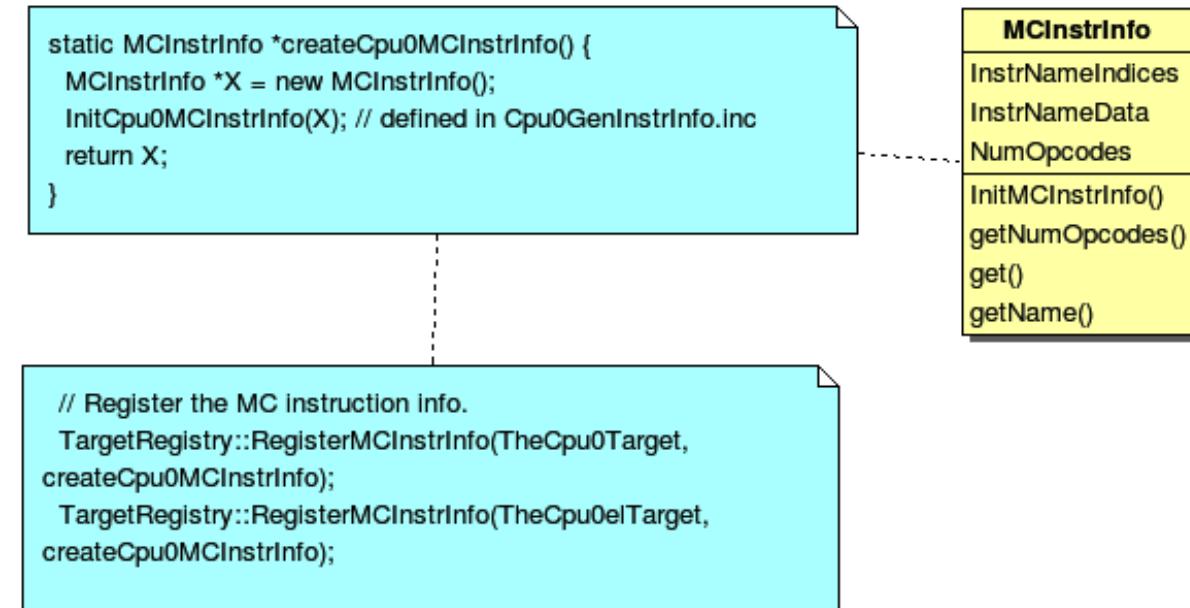


Figure 5.3: Register MCInstrInfo



Figure 5.4: Register MCRegisterInfo

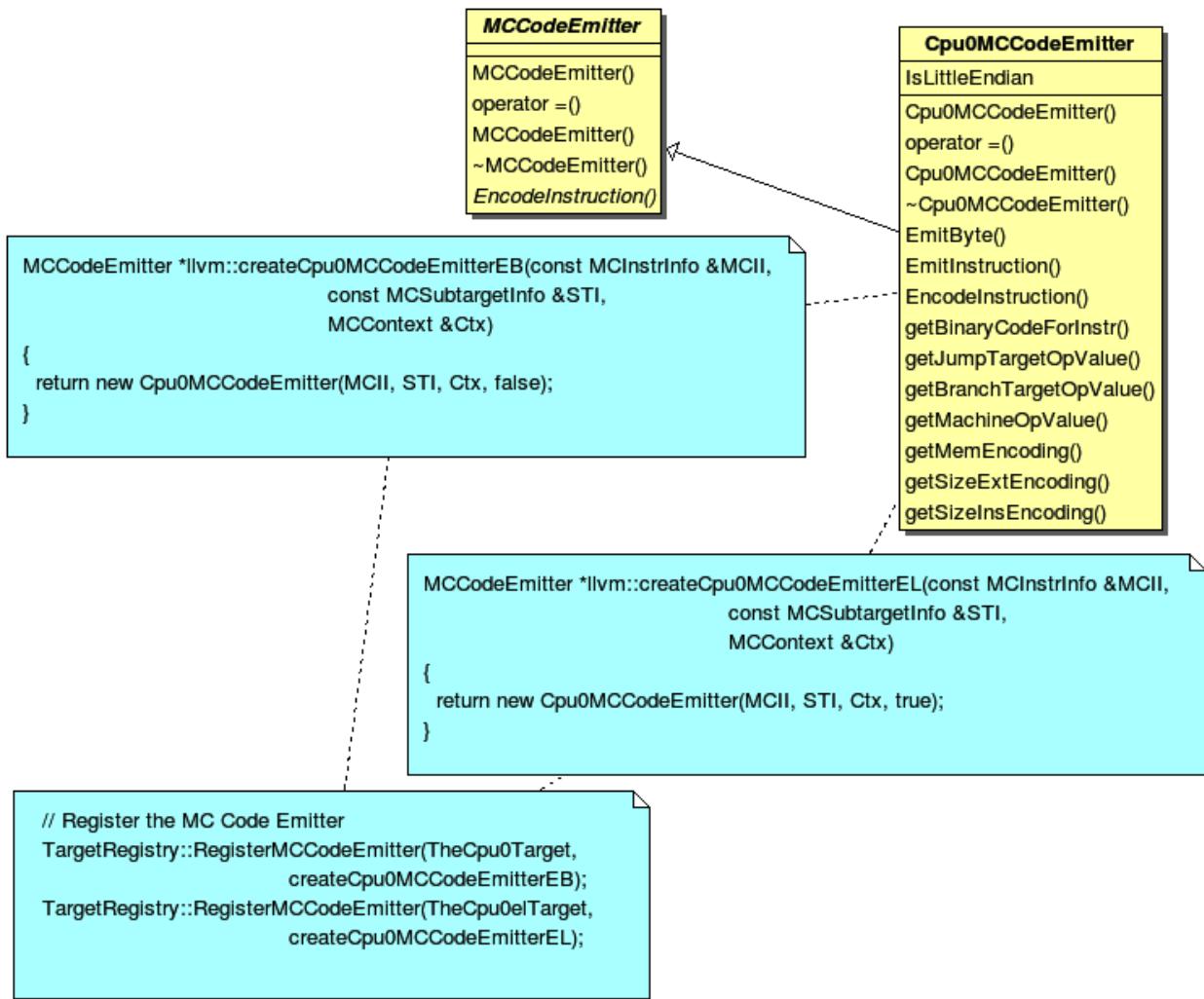


Figure 5.5: Register Cpu0MCCodeEmitter

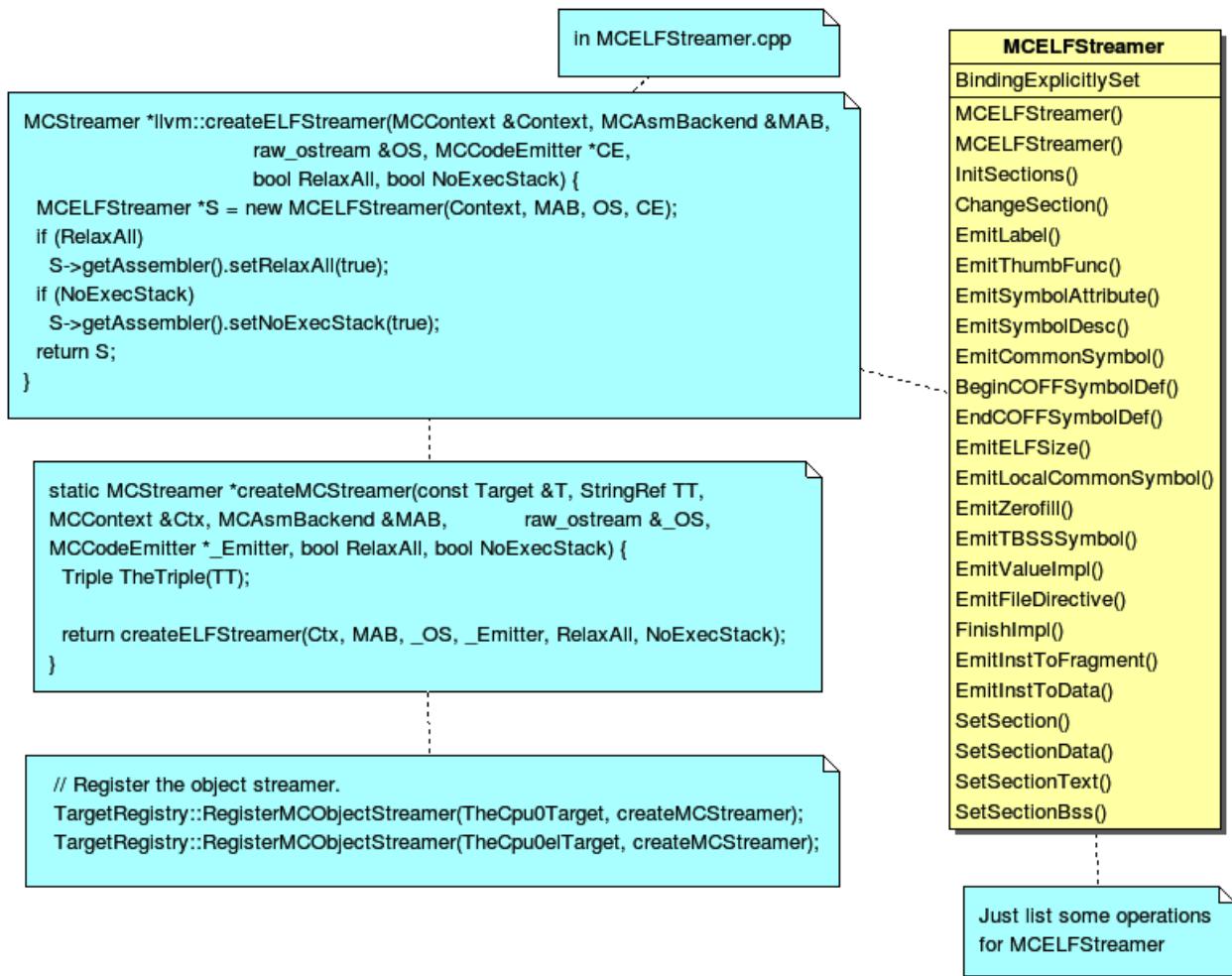


Figure 5.6: Register MCELFStreamer



Figure 5.7: Register Cpu0AsmBackend

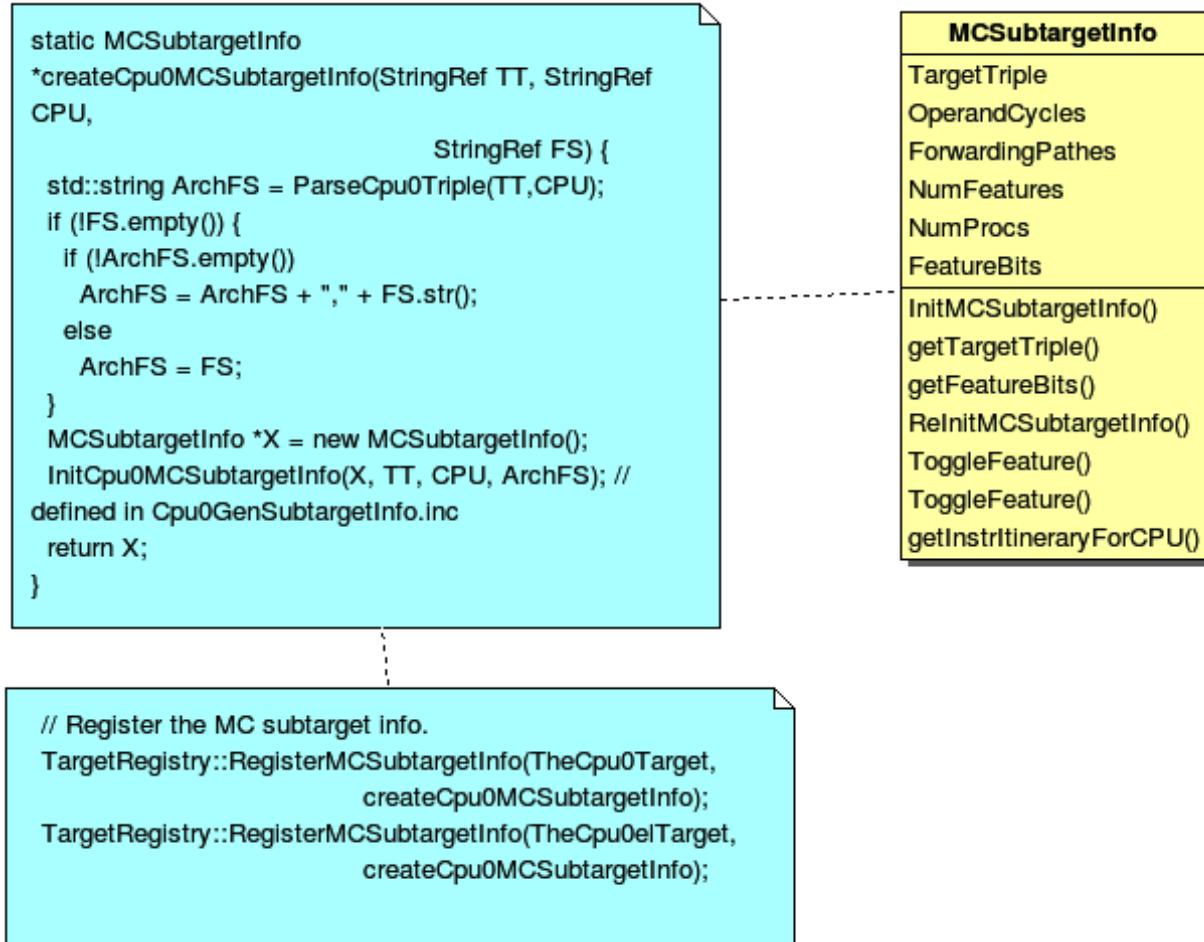


Figure 5.8: Register Cpu0MCSubtargetInfo

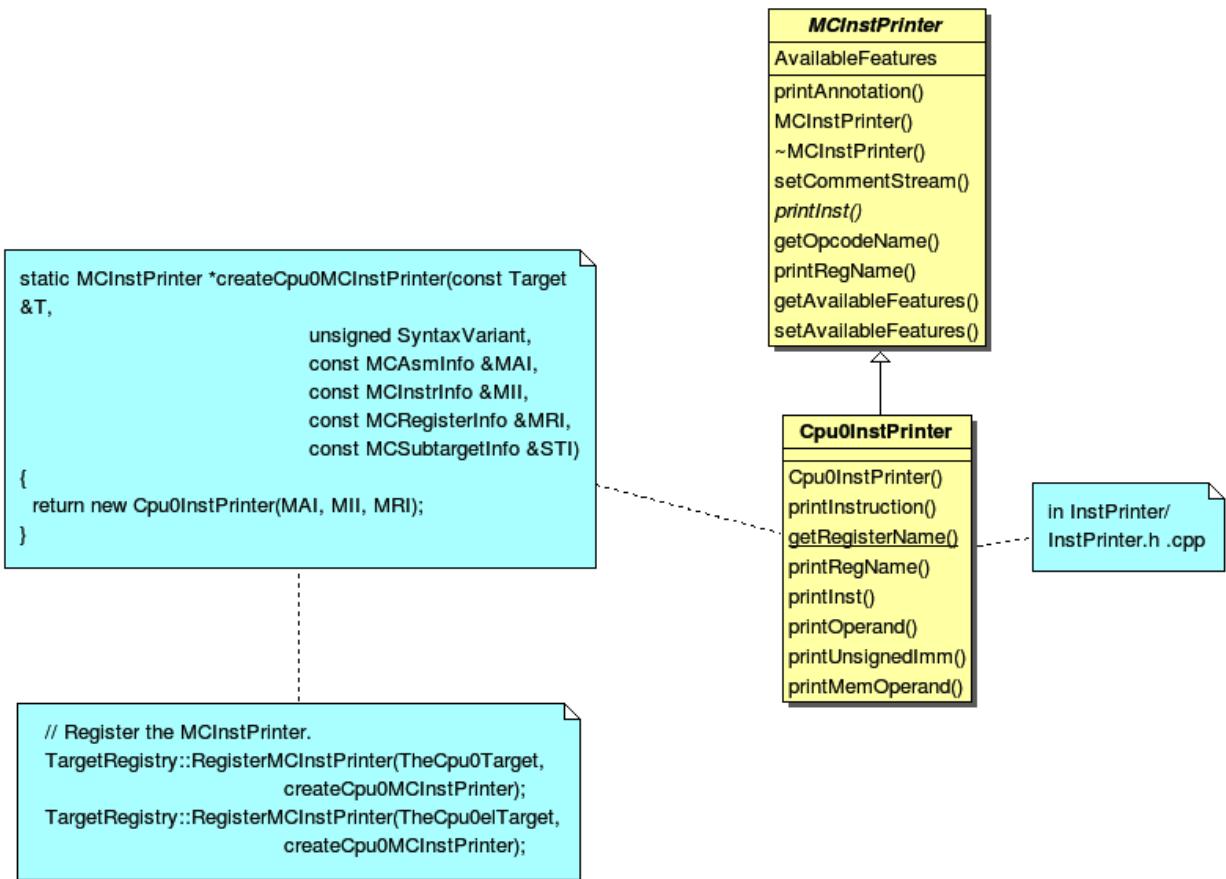


Figure 5.9: Register Cpu0InstPrinter



Figure 5.10: MCELFStreamer inherit tree

Figure 5.7, 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 MCELF-Streamer and it's parent, MCAsmBackend.

Figure 5.8, instancing MCSubtargetInfo object and initialize with Cpu0.td information. Figure 5.9, instancing Cpu0InstPrinter to take care printing function for instructions. Like Figure 5.1 to Figure 5.4, it has been defined before in Chapter4_2/ code for assembly file generated support.

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 very 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.

Index/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=false` 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/cmake_debug_build/
Debug/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,
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/cmake_debug_build/
Debug/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]
...
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/cmake_debug_build/
Debug/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]

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, %hi(gI)
ori $2, $2, %lo(gI)
ld $2, 0($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/cmake_debug_build/
Debug/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:
...
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
.cupload $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

```

Summary above information to Table: Cpu0 global variable options.

Table 6.1: Cpu0 global variable options

option name	default	other option value	description
-relocation-model	pic	static	<ul style="list-style-type: none"> • pic: Position Independent Address • static: Absolute Address
-cpu0-use-small-section	false	true	<ul style="list-style-type: none"> • 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	ori \$2, \$zero, %hi(gI); shl \$2, \$2, 16; 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);	ori \$2, \$zero, %got_hi(gI); shl \$2, \$2, 16; 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 statically. In C++, the overload variable or function are binding dynamically.

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.

Ibdex/chapters/Chapter6_1/Cpu0Subtarget.h

```
extern bool Cpu0ReserveGP;
extern bool Cpu0NoCpload;
```

¹ <http://llvm.org/docs/CommandLine.html>

```
class Cpu0Subtarget : public Cpu0GenSubtargetInfo {
    ...
    // UseSmallSection - Small section is used.
    bool UseSmallSection;
    ...
    bool useSmallSection() const { return UseSmallSection; }
    ...
};
```

Ibdex/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, const std::string &CPU,
                            const std::string &FS, bool little,
                            const Cpu0TargetMachine &_TM) :
    ...
    // Set UseSmallSection.
    UseSmallSection = UseSmallSectionOpt;
    Cpu0ReserveGP = ReserveGPOpt;
    Cpu0NoCpload = NoCploadOpt;
    ...
}
```

The options ReserveGPOpt and NoCploadOpt will be 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.

Ibdex/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 {
```

Ibdex/chapters/Chapter6_1/Cpu0TargetObjectFile.h

```

/// IsGlobalInSmallSection - Return true if this global address should be
/// placed into small data/bss section.
bool IsGlobalInSmallSection(const GlobalValue *GV,
                           const TargetMachine &TM, SectionKind Kind) const;
bool IsGlobalInSmallSection(const GlobalValue *GV,
                           const TargetMachine &TM) const;
bool IsGlobalInSmallSectionImpl(const GlobalValue *GV,
                               const TargetMachine &TM) const;

MCSection *SelectSectionForGlobal(const GlobalValue *GV, SectionKind Kind,
                                 Mangler &Mang,
                                 const TargetMachine &TM) const override;

```

Ibdex/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 GlobalValue *GV,
                                                const TargetMachine &TM) const {
    if (GV->isDeclaration() || GV->hasAvailableExternallyLinkage())
        return false;

    return IsGlobalInSmallSection(GV, TM, getKindForGlobal(GV, TM));
}

/// IsGlobalInSmallSection - Return true if this global address should be
/// placed into small data/bss section.
bool Cpu0TargetObjectFile::
IsGlobalInSmallSection(const GlobalValue *GV, const TargetMachine &TM,
                      SectionKind Kind) const {
    return (IsGlobalInSmallSectionImpl(GV, TM) &&
            (Kind.isDataRel() || Kind.isBSS() || Kind.isCommon()));
}

/// 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 GlobalValue *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->getType()->getElementType();
return IsInSmallSection(TM.getDataLayout()->getTypeAllocSize(Ty));
}

MCSection *
Cpu0TargetObjectFile::SelectSectionForGlobal(const GlobalValue *GV,
                                             SectionKind Kind, Mangler &Mang,
                                             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(GV, TM, Kind))
    return SmallBSSSection;
if (Kind.isDataNoRel() && IsGlobalInSmallSection(GV, TM, Kind))
    return SmallDataSection;

// Otherwise, we work the same as ELF.
return TargetLoweringObjectFileELF::SelectSectionForGlobal(GV, Kind, Mang, TM);
}

```

[Index/chapters/Chapter6_1/Cpu0RegisterInfo.cpp](#)

```

BitVector Cpu0RegisterInfo::
getReservedRegs(const MachineFunction &MF) const {
    ...
    Reserved.set(Cpu0::GP);
    ...
}

```

[Index/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 getAddressLocal(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(), false, false,
                               false, 0);
    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, false, false, false, 0);
}
//@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, false, false, false,
                        0);
}
//@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));
}

```

Index/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);
}

//@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 (getTargetMachine().getRelocationModel() != Reloc::PIC_) {
        //@ %gp_rel relocation
        if (TLOF->IsGlobalInSmallSection(GV, getTargetMachine())) {
            SDValue GA = DAG.getTargetGlobalAddress(GV, DL, MVT::i32, 0,

```

```

        Cpu0II::MO_GPREL) ;
SDValue GPRelNode = DAG.getNode(Cpu0ISD::GPRel, DL,
                                DAG.getVList(MVT::i32), GA) ;
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
if (!TLOF->IsGlobalInSmallSection(GV, getTargetMachine()))
return getAddrGlobalLargeGOT(N, Ty, DAG, Cpu0II::MO_GOT_HI16,
                             Cpu0II::MO_GOT_LO16, DAG.getEntryNode(),
                             MachinePointerInfo::getGOT()) ;
return getAddrGlobal(N, Ty, DAG, Cpu0II::MO_GOT16, DAG.getEntryNode(),
                     MachinePointerInfo::getGOT()) ;
}

```

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.

Index/chapters/Chapter6_1/Cpu0ISelDAGToDAG.h

```
SDNode *getGlobalBaseReg();
```

Index/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
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
SDNode* Cpu0DAGToDAGISel::Select(SDNode *Node) {

    // Get target GOT address.
    case ISD::GLOBAL_OFFSET_TABLE:
        return getGlobalBaseReg();

    ...
}

```

Index/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<(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

.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 .data and instruction “ori \$2, \$zero, %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 “ori \$2, \$zero, %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 loading 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 (GlobalAd-

dress<i32* @gI> 0) into (add Cpu0ISD::Hi<gI offset Hi16> Cpu0ISD::Lo<gI offset Lo16>) in stage “Legalized selection DAG” as below.

Ibdex/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));
}
```

Ibdex/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$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/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]

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.

Index/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<(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.

[Index/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 (getTargetMachine().getRelocationModel() != Reloc::PIC_) {
        //@@ %gp_rel relocation
        if (TLOF->IsGlobalInSmallSection(GV, 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);
        }
    }
```

```

    ...
}

...
}

...
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.

[Index/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 specific 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.

Ibdex/chapters/Chapter6_1/Cpu0Subtarget.cpp

```
Cpu0Subtarget::Cpu0Subtarget(const Triple &TT, const std::string &CPU,
                            const std::string &FS, bool little,
                            const Cpu0TargetMachine &_TM) :

    // Set UseSmallSection.
    UseSmallSection = UseSmallSectionOpt;
    Cpu0ReserveGP = ReserveGPOpt;
    Cpu0NoCpload = NoCploadOpt;
#ifdef ENABLE_GPRESTORE
    if (_TM.getRelocationModel() == Reloc::Static == Reloc::Static && !UseSmallSection && !Cpu0ReserveGP)
        FixGlobalBaseReg = false;
    else
#endif
    FixGlobalBaseReg = true;
}

}
```

Ibdex/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
.cupload $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 **.cupload** asm pseudo instruction at function entry point as below.

Index/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),
        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

    ...
};

```

Index/chapters/Chapter6_1/Cpu0MachineFunction.cpp

```

bool Cpu0FunctionInfo::globalBaseRegFixed() const {
    return FixGlobalBaseReg;
}

```

```
bool Cpu0FunctionInfo::globalBaseRegSet() const {
    return GlobalBaseReg;
}

unsigned Cpu0FunctionInfo::getGlobalBaseReg() {
    return GlobalBaseReg = Cpu0::GP;
}
```

Index/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() &&
        Cpu0FI->globalBaseRegFixed();
    if (Cpu0NoCpload)
        EmitCPLoad = false;

    // Emit .cupload directive if needed.
    if (EmitCPLoad)
        OutStreamer->EmitRawText(StringRef("\t.cupload\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
.cupload $6
.set      nomacro
...
```

The **.cupload** is the assembly directive (macro) which will expand to several instructions. Issue **.cupload** 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 ABI².

Following code will expand **.cupload** into machine instructions as below. “0fa00000 09aa0000 13aa6000” is the **.cupload** machine instructions displayed in comments of Cpu0MCInstLower.cpp.

Index/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);
```

² <http://www.linux-mips.org/pub/linux/mips/doc/ABI/mipsabi.pdf>

```

private:
    MCOperand LowerSymbolOperand(const MachineOperand &MO,
                                MachineOperandType MOTy, unsigned Offset) const;
    ...
}

```

Ibdex/chapters/Chapter6_1/Cpu0MCInstLower.cpp

```

// Lower ".cupload $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 MCSymbolRefExpr *MCSym;

    MCSym = MCSymbolRefExpr::create(Sym, MCSymbolRefExpr::VK_Cpu0_ABS_HI, *Ctx);
    MCOperand SymHi = MCOperand::createExpr(MCSym);
    MCSym = MCSymbolRefExpr::create(Sym, MCSymbolRefExpr::VK_Cpu0_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/
cmake_debug_build/Debug/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 offset 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 reloaction 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.

[Index/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, false, false, false, 0);
}
```

[Index/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();

    if (TLOF->IsGlobalInSmallSection(GV, 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);
    }
    ...
}
```

Ibdex/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

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 **Id** 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 ".cupload" 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 ".cupload" can be removed here and have only one instruction cost in global variable access. The advantage of ".cupload" removing come from losing one general purpose register \$gp which can be allocated for variables. In last sub-section, .sdata mode, we use ".cupload" removing since it is static link. In pic mode, the dynamic loading takes too much time. Remove ".cupload" with the cost of losing one general purpose register at all functions is not deserved here. The relocation records of ".cupload" 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.

Index/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, false, false, false,
                      0);
}
```

Index/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();

    if (!TLOF->IsGlobalInSmallSection(GV, getTargetMachine()))
        return getAddrGlobalLargeGOT(N, Ty, DAG, Cpu0II::MO_GOT_HI16,
                                     Cpu0II::MO_GOT_LO16, DAG.getEntryNode(),
                                     MachinePointerInfo::getGOT());
    return getAddrGlobal(N, Ty, DAG, Cpu0II::MO_GOT16, DAG.getEntryNode(),
```

```

        MachinePointerInfo::getGOT() );
}

...
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

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 **Id** 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_5.cpp. Since ch8_5.cpp uses llvm IR **select**, it cannot be run at this point. It will be run in later Chapter Control flow statements.

Ibdex/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>;  
}
```

Ibdex/input/ch8_5.cpp

```
int test_select_global_pic()  
{  
    if (a1 < b1)  
        return gI1;  
    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, Cpu0InstPrinter.cpp and Cpu0ISelLowering.cpp for global variable printing operand function.

Ibdex/chapters/Chapter6_1/Cpu0MCInstLower.cpp

```
MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,  
                                              MachineOperandType MOTy,  
                                              unsigned Offset) const {  
    MCSymbolRefExpr::VariantKind Kind;  
    const MCSymbol *Symbol;  
  
    switch(MO.getTargetFlags()) {  
    default: llvm_unreachable("Invalid target flag!");  
    case Cpu0II::MO_NO_FLAG: Kind = MCSymbolRefExpr::VK_None; break;  
  
    // Cpu0_GPREL is for l1c -march=cpu0 -relocation-model=static -cpu0-islinux-  
    // format=false (global var in .sdata).  
    case Cpu0II::MO_GPREL: Kind = MCSymbolRefExpr::VK_Cpu0_GPREL; break;  
  
    case Cpu0II::MO_GOT16: Kind = MCSymbolRefExpr::VK_Cpu0_GOT16; break;  
    case Cpu0II::MO_GOT: Kind = MCSymbolRefExpr::VK_Cpu0_GOT; break;  
    // ABS_HI and ABS_LO is for l1c -march=cpu0 -relocation-model=static (global  
    // var in .data).  
    case Cpu0II::MO_ABS_HI: Kind = MCSymbolRefExpr::VK_Cpu0_ABS_HI; break;  
    case Cpu0II::MO_ABS_LO: Kind = MCSymbolRefExpr::VK_Cpu0_ABS_LO; break;  
    case Cpu0II::MO_GOT_HI16: Kind = MCSymbolRefExpr::VK_Cpu0_GOT_HI16; break;  
    case Cpu0II::MO_GOT_LO16: Kind = MCSymbolRefExpr::VK_Cpu0_GOT_LO16; break;  
    }  
}
```

```

switch (MOTy) {
    case MachineOperand::MO_GlobalAddress:
        Symbol = AsmPrinter.getSymbol(MO.getGlobal());
        break;

    default:
        llvm_unreachable("<unknown operand type>");
}

const MCSymbolRefExpr *MCSym = MCSymbolRefExpr::create(Symbol, Kind, *Ctx);

if (!Offset)
    return MCOperand::createExpr(MCSym);

// Assume offset is never negative.
assert(Offset > 0);

const MCConstantExpr *OffsetExpr = MCConstantExpr::create(Offset, *Ctx);
const MCBinaryExpr *AddExpr = MCBinaryExpr::createAdd(MCSym, OffsetExpr, *Ctx);
return MCOperand::createExpr(AddExpr);
}

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);

        ...
    }
}

```

Index/chapters/Chapter6_1/InstPrinter/Cpu0InstPrinter.cpp

```

static void printExpr(const MCEexpr *Expr, raw_ostream &OS) {

// Cpu0_GPREL is for llc -march=cpu0 -relocation-model=static
    case MCSymbolRefExpr::VK_Cpu0_GPREL:      OS << "%gp_rel("; break;

    case MCSymbolRefExpr::VK_Cpu0_GOT16:       OS << "%got(";      break;
    case MCSymbolRefExpr::VK_Cpu0_GOT:         OS << "%got(";      break;
    case MCSymbolRefExpr::VK_Cpu0_ABS_HI:      OS << "%hi(";       break;
    case MCSymbolRefExpr::VK_Cpu0_ABS_LO:      OS << "%lo(";       break;

    ...
}
...
}

```

The following function is for llc -debug this chapter DAG node name printing. It is added at Chapter3_1 already.

Ibdex/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 set 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
l1c -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
l1c -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>

OTHER DATA TYPE

- Local variable pointer
- char, short int and bool
- long long
- float and double
- Array and struct 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 include 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,

[Index/chapters/Chapter7_1/Cpu0InstrInfo.td](#)

```
def mem_ea : Operand<i32> {
    let PrintMethod = "printMemOperandEA";
    let MIOperandInfo = (ops CPURegs, 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> {
    let isCodeGenOnly = 1;
}
```

Ibdex/chapters/Chapter3_2/InstPrinter/Cpu0InstPrinter.h

```
void printMemOperandEA(const MCInst *MI, int opNum, raw_ostream &O);
```

Ibdex/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, only opcode node can be disabled. Run ch7_1.cpp with code Chapter7_1/ which support pointer to local variable, will get result as follows,

Ibdex/input/ch7_1.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.cpp -emit-llvm -o ch7_1.bc
118-165-66-82:input Jonathan$ llvm-dis ch7_1.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
    ret i32 %2
}
...
118-165-66-82:input Jonathan$ /Users/Jonathan/llvm/test/cmake_
debug_build/Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch7_1.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/.

Ibdex/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 {
defm LB      : LoadM32<0x03, "lb",  sextloadi8>;
defm LBu     : LoadM32<0x04, "lbu", zextloadi8>;
defm SB      : StoreM32<0x05, "sb",  truncstorei8>;
defm LH      : LoadM32<0x06, "lh",  sextloadi16_a>;
defm LHu     : LoadM32<0x07, "lhu", zextloadi16_a>;
defm SH      : StoreM32<0x08, "sh",  truncstorei16_a>;
}

```

Run Chapter7_1/ with ch7_2.cpp will get the following result.

Ibdex/input/ch7_2.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/cmake_debug_build/
Debug/bin/llvm-dis ch7_2.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_2.cpp -emit-llvm -o ch7_2.bc
118-165-64-245:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch7_2.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 _$Z9test_charvE5date1,@object # @_Z9test_charvE5date1
.section .rodata.cst8,"aM",@progbits,8
.align 1
$_Z9test_charvE5date1:
.2byte 2012             # 0x7dc
.byte 11                # 0xb
.byte 25                # 0x19
.byte 9                 # 0x9
.byte 40                # 0x28
.byte 15                # 0xf
.space 1
.size _$Z9test_charvE5date1, 8

```

Run Chapter7_1/ with ch7_2_2.cpp will get the following result.

lbdex/input/ch7_2_2.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/cmake_debug_build/
Debug/bin/llvm-dis ch7_2_2.bc -o -
...
define i32 @_Z16test_signed_charv() #0 {
...
%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/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=static -filetype=asm ch7_2_2.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

.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.

[Index/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_3.ll still works as below. The IR input file ch7_3.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  
  /// 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;  
  }
```

Ibdex/input/ch7_3.ll

```
define zeroext i1 @verify_load_bool() #0 {  
entry:  
  %retval = alloca i1, align 1  
  store i1 1, i1* %retval, align 1  
  %0 = load i1* %retval  
  ret i1 %0  
}  
  
118-165-64-245:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/  
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch7_3.ll -o -  
  
.section .mdebug.abi32  
.previous  
.file "ch7_3.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_3.cpp is the bool test version for C language. You can run with it at Chapter8_1 to get the similar result with ch7_3.ll.

Ibdex/input/ch7_3.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_2_2.cpp and ch7_3.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_2_2.cpp and ch7_3.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/.

Index/chapters/Chapter7_1/Cpu0SEISelDAGToDAG.cpp

```

SDNode *Cpu0SEIDAGToDAGISel::selectAddESubE(unsigned MOp, SDValue InFlag,
                                             SDValue CmpLHS, 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);

    return CurDAG->SelectNodeTo(Node, MOp, VT, MVT::Glue,
                                  LHS, SDValue(AddCarry, 0));
}

std::pair<bool, SDNode*> Cpu0SEIDAGToDAGISel::selectNode(SDNode *Node) {
    unsigned Opcode = Node->getOpcode();
    SDLoc DL(Node);

    /**
     // Instruction Selection not handled by the auto-generated
     // tablegen selection should be handled here.
     /**
     SDNode *Result;

    /**
     // 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::SUBE: {
        SDValue InFlag = Node->getOperand(2);
        Result = selectAddESubE(Cpu0::SUBu, InFlag, InFlag.getOperand(0), DL, Node);
        return std::make_pair(true, Result);
    }

    case ISD::ADDE: {
        SDValue InFlag = Node->getOperand(2);

```

```

        Result = selectAddESubE(Cpu0::ADDu, InFlag, InFlag.getValue(0), DL, Node);
        return std::make_pair(true, Result);
    }

    /// 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));
        Result = NULL;
        return std::make_pair(true, Result);
    }

    ...
}

```

Ibdex/chapters/Chapter7_1/Cpu0ISelLowering.h

```

class Cpu0TargetLowering : public TargetLowering {

    bool isOffsetFoldingLegal(const GlobalAddressSDNode *GA) const override;
    ...
}

```

Ibdex/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_4.cpp to get the result as follows,

Ibdex/input/ch7_4.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

    return (c+d+e+f); // (0x0006005b,01000006) = (393307,16777222)
}
```

```
1-160-134-62:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_4.cpp -emit-llvm -o ch7_4.bc
1-160-134-62:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -filetype=asm
ch7_4.bc -o -
...
```

```
# BB#0:
    addiu $sp, $sp, -72
    st    $8, 68($fp)           # 4-byte Folded Spill
    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
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. 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 need float intended application.

7.5 Array and struct support

LLVM uses getelementptr to represent the array and struct type in C. Please reference here ¹. For ch7_5.cpp, the llvm IR as follows,

Ibdex/input/ch7_5.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_5.ll
; ModuleID = 'ch7_5.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_5.bc on static mode will get the incorrect asm file as follows,

```
1-160-134-62:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/bin/
Debug/llc -march=cpu0 -relocation-model=static -filetype=asm ch7_5.bc -o -
```

¹ <http://llvm.org/docs/LangRef.html#getelementptr-instruction>

```

...
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,

```

jonathantekimac:input Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/Debug/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]

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>
```

```
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.

Ibdex/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_5.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_5_2.cpp is for local variable initialization test. The result as follows,

Ibdex/input/ch7_5_2.cpp

```
int main()
{
    int a[3]={0, 1, 2};

    return 0;
}
```

```
118-165-79-206:input Jonathan$ llvm-dis ch7_5_2.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]*
 @_ZZ4mainEla 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/cmake_debug_build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch7_5_2.bc -o -
...
# BB#0:                                # %entry
    addiu $sp, $sp, -16
    addiu $2, $zero, 0
    st   $2, 12($fp)
    ld   $2, %got($_ZZ4mainEla)($gp)
    ori  $2, $2, %lo($_ZZ4mainEla)
    ld   $3, 8($2)
    st   $3, 8($fp)
    ld   $3, 4($2)
    st   $3, 4($fp)
    ld   $2, 0($2)
    st   $2, 0($fp)
    addiu $sp, $sp, 16
    ret  $lr
...
.type $_ZZ4mainEla,@object      # @_ZZ4mainEla
.section .rodata,"a",@progbits
.align 2
$_ZZ4mainEla:
    .4byte 0                      # 0x0
    .4byte 1                      # 0x1
    .4byte 2                      # 0x2
.size $_ZZ4mainEla, 12
```


CONTROL FLOW STATEMENTS

- Control flow statement
- Cpu0 backend Optimization: Remove useless JMP
- Fill Branch Delay Slot
- Conditional instruction
- RISC CPU knowledge

This chapter illustrates the corresponding IR for control flow statements, like “**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 II, 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 III, include the conditional instructions handle since the clang will generate specific IRs, select and select_cc, to support the backend optimiation in control flow statement.

8.1 Control flow statement

Run ch8_1_1.cpp with clang will get result as follows,

lbdex/input/ch8_1_1.cpp

```
int test_control1()
{
    unsigned int a = 0;
    int b = 1;
    int c = 2;
    int d = 3;
    int e = 4;
    int f = 5;
    int g = 6;
    int h = 7;
    int i = 8;
    int j = 9;

    if (a == 0) {
        a++; // a = 1
    }
    if (b != 0) {
        b++; // b = 2
    }
    if (c > 0) {
```

```

        c++; // c = 3
    }
    if (d >= 0) {
        d++; // d = 4
    }
    if (e < 0) {
        e++; // e = 4
    }
    if (f <= 0) {
        f++; // f = 5
    }
    if (g <= 1) {
        g++; // g = 6
    }
    if (h >= 1) {
        h++; // h = 8
    }
    if (i < h) {
        i++; // i = 8
    }
    if (a != b) {
        j++; // j = 10
    }

    return (a+b+c+d+e+f+g+h+i+j); // 1+2+3+4+4+5+6+8+8+10 = 51
}

...
%0 = load i32* %a, align 4
%cmp = icmp eq i32 %0, 0
br i1 %cmp, label %if.then, label %if.end

if.then:                                ; preds = %entry
    %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, we get the form `(br (brcond (setcc(%2, Constant<c>, setne)), BasicBlock_02), BasicBlock_01)`. For explanation, listing the IR DAG as follows,

```
%cond=setcc(%2, Constant<c>, setne)
brcond %cond, BasicBlock_02
br BasicBlock_01
```

We want to translate them into Cpu0 instructions DAG 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,

Ibdex/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 = 1;
}
}

...
def JMP      : UncondBranch<0x26, "jmp">;
```

The pattern [(br bb:\$imm24)] in class UncondBranch is translated into jmp machine instruction. The pair of **cmp** and **jne** Cpu0 instructions translation is more complicate than simple one-to-one IR to machine instruction translation we have experienced until now. To solve this chained IR to machine instructions translation, we define the following pattern,

Ibdex/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),
  (JNEOp (CMPOp RC:$cond, ZEROReg), bb:$dst)>;
...
}
```

Since the BrcondPats pattern as above 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.

Ibdex/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 reference Mips code even though we didn'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. Pattern search order come along with their appear order in context. The last pattern (brcond RC:\$cond, bb:\$dst) meaning branch to \$dst if \$cond != 0. So we set the corresponding translation to (JNEOp (CMPOp RC:\$cond, ZEROReg), bb:\$dst).

The CMP instruction will set the result to register SW, and then JNE check the condition based on SW status as Figure 8.1. Since SW belongs to a different register class, it will be correct even an instruction is inserted between CMP and JNE as follows,

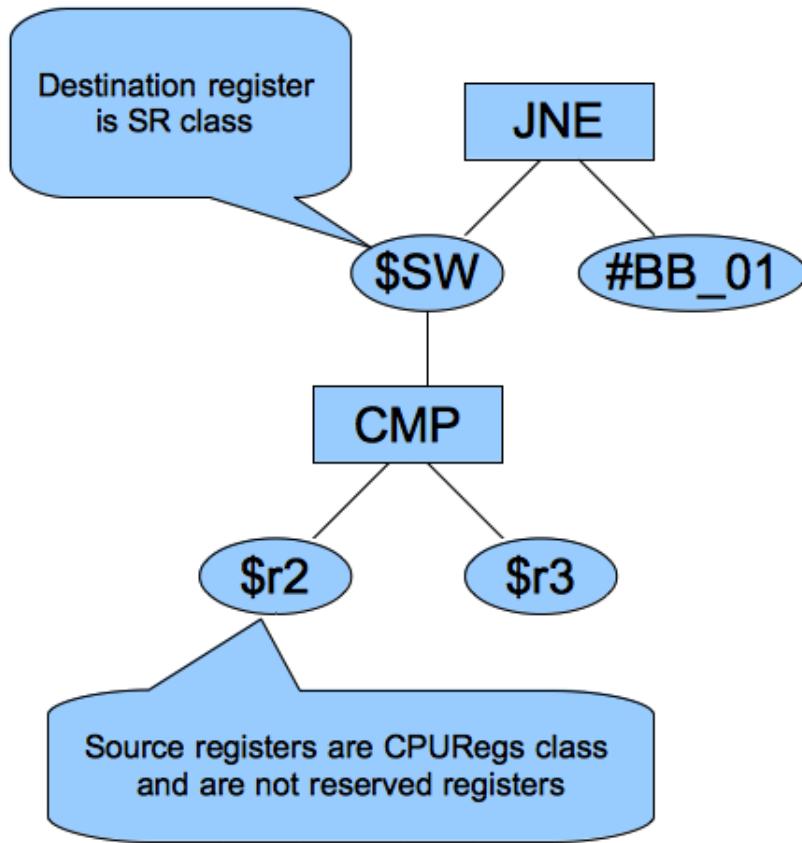


Figure 8.1: JNE (CMP \$r2, \$r3),

```

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
// Cpu0RegisterInfoForGPROutForOther.td
jne BasicBlock_02

```

The reserved registers setting by the following function code we defined before,

Ibdex/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::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, you should comment the Reserved Registers in it for readability. Setting SW both in register class CPUREgs and SR to allow the SW to be accessed by RISC instructions like andi and allow programmer use traditional assembly instruction cmp. The copyPhysReg() is called when DestReg and SrcReg are belonging to different Register Class.

Ibdex/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, PC, EPC)>;

def SR      : RegisterClass<"Cpu0", [i32], 32, (add SW)>;

```

Ibdex/chapters/Chapter2/Cpu0RegisterInfoGPROutForOther.td

```

//=====
// Register Classes
//=====

def GPROut : RegisterClass<"Cpu0", [i32], 32, (add (sub CPUREgs, SW, PC, EPC))>;

```

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/
cmake_debug_build/Debug/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
nop
jmp $BB0_1
nop
$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/
cmake_debug_build/Debug/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
00000080 ..... 10 43 00 00
00000090 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 immeddiate 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)
    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 follows,

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/Debug/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic
-filetype=asm ch8_1_1.bc -o -
...
    cmp      $sw, $4, $3
    jne      $sw, $BB0_2
    nop
    jmp      $BB0_1
    nop
$BB0_1:
```

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/Debug/bin/llc -march=cpu0 -mcpu=cpu032II -relocation-model=pic
-filetype=asm ch8_1_1.bc -o -
...
    bne      $4, $zero, $BB0_2
    nop
    jmp      $BB0_1
    nop
```

`$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_2.cpp is for “**nest if**” test. The ch8_1_3.cpp is for the test of “**for loop**” as well as “**while loop**”, “**continue**”, “**break**”, “**goto**”. The ch8_1_4.cpp is for the test of “**goto**”. The ch8_1_5.cpp is for **br_jt** and **JumpTable** 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, settle) + brcond,
•	(setueq, setune, setugt, setuge, setult, setule) + brcond
cpu032I	CMP + (JEQ, JNE, JGT, JGE, JLT, JLE)
cpu032II	(SLT, SLTu, SLTi, SLTi) + (BEG, BNE)

8.2 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 much 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 migrate 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 complicate optimization algorithm on your backend in real project.

Chapter8_2/ supports “delete useless jmp” optimization algorithm which add codes as follows,

Ibdex/chapters/Chapter8_2/CMakeLists.txt

```
Cpu0DelUselessJMP.cpp
```

Ibdex/chapters/Chapter8_2/Cpu0.h

```
FunctionPass *createCpu0DelJmpPass(Cpu0TargetMachine &TM);
```

Ibdex/chapters/Chapter8_2/Cpu0TargetMachine.cpp

```
/// Cpu0 Code Generator Pass Configuration Options.
class Cpu0PassConfig : public TargetPassConfig {

    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(createCpu0DelJmpPass(TM));

}
```

Ibdex/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/Support/CommandLine.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/TargetInstrInfo.h"
#include "llvm/ADT/SmallSet.h"
#include "llvm/ADT/Statistic.h"

using namespace llvm;

#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) { }

        virtual const char *getPassName() const {
            return "Cpu0 Del Useless jmp";
    };
}
```

```

}

bool runOnMachineBasicBlock(MachineBasicBlock &MBB, MachineBasicBlock &MBBN);
bool runOnMachineFunction(MachineFunction &F) {
    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();
              FI != 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
        return Changed;

    if (I->getOpcode() == Cpu0::JMP && I->getOperand(0).getMBB() == &MBBN) {
        // I is the instruction of "jmp #offset=0", as follows,
        //      jmp      $B0_3
        // $B0_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 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 if it is the JMP instruction as well as 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.cpp for explanation.

Ibdex/input/ch8_2.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;
}

118-165-78-10:input Jonathan$ clang -target mips-unknown-linux-gnu
-c ch8_2.cpp -emit-llvm -o ch8_2.bc
118-165-78-10:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=static -filetype=asm -stats
ch8_2.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, will find 10 jmp instructions are deleted from 120 lines of assembly code, which meaning 8% improvement in speed and code size¹.

¹ On a platform with cache and DRAM, the cache miss costs several tens time of instruction cycle. Usually, the compiler engineers who work

8.3 Fill Branch Delay Slot

Cpu0 instruction set is designed to be a classical RISC pipeline machine. Classical machine has many perfect features²³. Since Cpu0 has delay slot same with 5 stages of Mips machine, the backend needs filling the NOP instruction in the branch delay slot. In order to make this tutorial as simple for learning, Cpu0 backend code not fill the branch delay slot with useful instruction for optimization. Readers can read the MipsDelaySlotFiller.cpp to know how to implement this optimization. Following code added in Chapter8_2 for NOP fill in Branch Delay Slot.

Ibdex/chapters/Chapter8_2/CMakeLists.txt

```
Cpu0DelaySlotFiller.cpp
```

Ibdex/chapters/Chapter8_2/Cpu0.h

```
FunctionPass *createCpu0DelaySlotFillerPass(Cpu0TargetMachine &TM);
```

Ibdex/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));

}
```

Ibdex/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
```

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.

² See book Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)

³ http://en.wikipedia.org/wiki/Classic_RISC_pipeline

```

#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"
#include "llvm/CodeGen/PseudoSourceValue.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Target/TargetInstrInfo.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/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) { }

    const char *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) {
    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 are bundled into one instruction (first part is branch instruction and second part is NOP).

[Index/chapters/Chapter3_2/Cpu0AsmPrinter.cpp](#)

```

//- EmitInstruction() must exists or will have run time error.
void Cpu0AsmPrinter::EmitInstruction(const MachineInstr *MI) {

    MachineBasicBlock::const_instr_iterator I = MI;
    MachineBasicBlock::const_instr_iterator E = MI->getParent()->instr_end();

    MCInst TmpInst0;
    do {

        MCInstLowering.Lower(I, TmpInst0);
        OutStreamer->EmitInstruction(TmpInst0, getSubtargetInfo());
    } while ((++I != E) && I->isInsideBundle()); // Delay slot check
}

```

To print the NOP, the Cpu0AsmPrinter.cpp of Chapter3_2 has printed all bundle instructions in loop. Without the loop, only the first part of the bundle instruction (branch instruction only) is printed. The result is NOP is missing and not be filled in branch delay slot. 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 some later version and you need doing "bundle" in order to keep block label unchanged at later llvm phase.

8.4 Conditional instruction

lbdex/input/ch8_3.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;
}
```

If you run Chapter8_1 with ch8_3.cpp will get the following result.

```
114-37-150-209:input Jonathan$ clang -O1 -target mips-unknown-linux-gnu
-c ch8_3.cpp -emit-llvm -o ch8_3.bc
114-37-150-209:input Jonathan$ ~/llvm/test/cmake_debug_build/Debug/bin/
 llvm-dis ch8_3.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
    %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/cmake_debug_build/Debug/bin/llc
-march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm ch8_3.bc -o -
...
LLVM ERROR: Cannot select: 0x39f47c0: i32 = select_cc ...
```

As llvm IR of ch8_3.bc as above, clang generates **select** IR for small basic control block (if statement only include one assign statement). This **select** IR is optimization result for CPU which has conditional instructions support. And from above llc command debug trace message, IR **select** is changed to **select_cc** during DAG optimization stages.

Chapter8_2 supports **select** with the following code added and changed.

Ibdex/chapters/Chapter8_2/Cpu0InstrInfo.td

```
let Predicates = [Ch8_2] in {
include "Cpu0CondMov.td"
} // let Predicates = [Ch8_2]
```

Ibdex/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
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:$T, (SLTiOp CRC:$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 {
defm : MovzPats0Slt<CPURegs, CPURegs, MOVZ_I_I, SLT, SLTu, SLTi, SLTi>;
}

defm : MovzPats1<CPURegs, CPURegs, MOVZ_I_I, XOR>;
defm : MovnPats<CPURegs, CPURegs, MOVN_I_I, XOR>;

```

Ibdex/chapters/Chapter8_2/Cpu0ISelLowering.h

```
SDValue lowerSELECT(SDValue Op, SelectionDAG &DAG) const;
```

Ibdex/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 to merge setcc and select into one IR select_cc⁴. Next the LowerSELECT() return ISD::SELECT as Op code directly. Finally the pattern define in Cpu0CondMov.td will translate the **select** IR into **movz** or **movn** conditional instruction. Let's run Chapter8_2 with ch8_3.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/cmake_debug_build/Debug/bin/llc
-march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm ch8_3.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 support³. If your backend has no conditional instruction and need clang compiler with optimization option **O1** level above, you can change clang to force it generate traditional branch basic block instead of IR **select**. RISC CPU came from pipeline advantage and add more and more instruction as 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 RISC pipeline machine as well as simple instructions for compiler toolchain tutorial. Anyway the **cmp** instruction hired because many programmer is used to it in past and now (ARM use it). It match the thinking in assembly programming, but the **slt** instruction is more efficient in RISC pipeline. If you designed a backend aimed for C/C++ highlevel language, you should consider **slt** instead **cmp**. Assembly code are rare used in programming, beside, the assembly programmer can accept **slt** either since usually they are professional.

File ch8_3_2.cpp will generate IR **select** if compile with clang -O1.

⁴ <http://llvm.org/docs/WritingAnLLVMBBackend.html#expand>

Ibdex/input/ch8_3_2.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;

    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_5.cpp for wrapper pic mode of global variable support which mentioned in Chapter Global variables can be tested now as follows.

Ibdex/input/ch8_5.cpp

```
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_5.cpp -emit-llvm -o ch8_5.bc
JonathantekiiMac:input Jonathan$ ~/llvm/test/cmake_debug_build/Debug/bin/
```

```

llvm-dis ch8_5.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 {
    %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
}
...
Jonathan@tekiMac:~/input Jonathan$ ~/llvm/test/cmake_debug_build/Debug/bin/
llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -filetype=asm ch8_5.bc -o -
.section .mdebug.abi32
.previous
.file "ch8_5.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
.cupload $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  
  
.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
```

8.5 RISC CPU knowledge

As mentioned in the previous section, Cpu0's instruction set is a RISC (Reduced Instruction Set Computer) CPU with 5 stages of pipeline (Even though it is not a pipeline as the Verilog designed at later chapter at this point). 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 will make you satisfied in compiler design. List these two excellent books we have read which include the real RISC CPU knowledge needed 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 (simple). "Computer Architecture: A Quantitative Approach" (there are 5 editions at the book is written) is more complicate and deep in CPU architecture.

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).

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 Graphviz's help
 - Long and short string initialization
- Structure type support
 - Ordinary struct type
 - byval struct type
- Function call optimization
 - 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
- Summary

The subroutine/function call of backend translation is supported in this chapter. A lot of code needed in function call. They are added according llvm supplied interface for easy to explanation. This chapter starts from introducing the Mips stack frame structure since we borrow many part of ABI from it. Although each CPU has its own ABI, most of RISC CPUs ABI are similar. In addition to support fixed number of arguments in function call, Cpu0 supports variable number of arguments either since C/C++ support this feature. Supply Mips ABI and assemble language manual on internet link in this chapter for your reference. 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 materials you can reference furth.

This chapter is more complicate than any of the previous chapters. It include stack frame and the related ABI support. 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” which listed in section “RISC CPU knowledge” of chapter “Control flow statement”¹, “Run Time Memory” of compiler book, or “Function Call Sequence” and “Stack Frame” of Mips ABI.

9.1 Mips stack frame

The first thing for design the Cpu0 function call is deciding how to pass arguments in function call. There are two options. The first is pass arguments all in stack. Second is pass arguments in the registers which are reserved for

¹ <http://jonathan2251.github.io/lbd/ctrlflow.html#risc-cpu-knowledge>

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. Figure 9.1 is the Mips stack frame.

Base	Offset	Contents	Frame
		unspecified ...	<i>High addresses</i>
		variable size	
		(if present)	
	+16	incoming arguments passed in stack frame	
old \$sp	+0	space for incoming arguments 1-4	<i>Previous</i>
		locals and temporaries	
		general register save area	
		floating-point register save area	
		argument build area	
\$sp	+0		

Base	Offset	Contents	Frame
		unspecified ...	<i>High addresses</i>
		variable size	
		(if present)	
	+16	incoming arguments passed in stack frame	
old \$sp	+0	space for incoming arguments 1-4	<i>Previous</i>
		locals and temporaries	
		general register save area	
		floating-point register save area	
		argument build area	
\$sp	+0		

Base	Offset	Contents	Frame
		unspecified ...	<i>High addresses</i>
		variable size	
		(if present)	
	+16	incoming arguments passed in stack frame	
old \$sp	+0	space for incoming arguments 1-4	<i>Previous</i>
		locals and temporaries	
		general register save area	
		floating-point register save area	
		argument build area	
\$sp	+0		

Figure 9.1: Mips stack frame

Run `llc -march=mips` for `ch9_1.bc`, you will get the following result. See comment “//”.

Ibdex/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;
}

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/cmake_debug_build/
Debug/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_iiiiii
.ent _Z5sum_iiiiii
_Z5sum_iiiiii:
.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
addu $2, $2, $1
sw $2, 4($sp)
jr $ra
addiu $sp, $sp, 32
.set at
.set macro
.set reorder
.end _Z5sum_iiiiii
$tmp2:
.size _Z5sum_iiiiii, ($tmp2)-_Z5sum_iiiiii
.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 $25, %call16(_Z5sum_iiiiii)($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
.set reorder
.end main
$tmp7:
.size main, ($tmp7)-main
.cfi_endproc
    
```

From the mips assembly code generated as above, we know it save the first 4 arguments to \$a0..\$a3 and last 2 arguments to 16(\$sp) and 20(\$sp). Figure 9.2 is the arguments location 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.

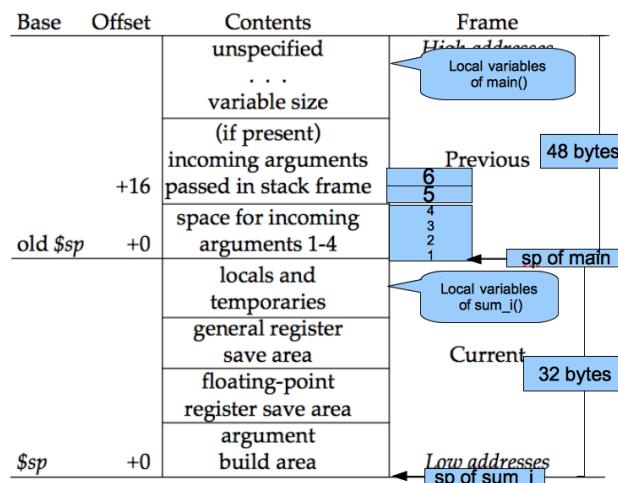


Figure 9.2: Mips arguments location in stack frame

The 007-2418-003.pdf in here ² is the Mips assembly language manual. Here ³ is Mips Application Binary Interface which include the Figure 9.1.

9.2 Load incoming arguments from stack frame

From last section, to support function call, we need implementing the arguments pass mechanism with stack frame. Before do that, 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/
cmake_debug_build/Debug/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/src/lib/CodeGen/SelectionDAG/
SelectionDAGBuilder.cpp, ...
...
0. Program arguments: /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/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_iiiiii'
Illegal instruction: 4
```

Since Chapter8_2/ define the LowerFormalArguments() with empty, we get the error message as above. Before define LowerFormalArguments(), we have to choose how to pass arguments in function call. For demonstration, Cpu0 pass arguments first two arguments in registers as default setting of llc -cpu0-s32-calls=false. When llc -cpu0-s32-calls=true, Cpu0 pass all it's arguments in stack.

Function LowerFormalArguments() is in charge of incoming arguments creation. We define it as follows,

Ibdex/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;
```

² <https://www.dropbox.com/sh/2pkh1fewlq2zag9/OHnrYn2nOs/doc/MIPSproAssemblyLanguageProgrammerGuide>

³ <http://www.linux-mips.org/pub/linux/mips/doc/ABI/mipsabi.pdf>

```

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, 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;

...
}

```

Index/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;

    // 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 OrigAlign = ArgFlags.getOrigAlign();
    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;
    unsigned OrigAlign = ArgFlags.getOrigAlign();
    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.
    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,
                                         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 MCPPhysReg 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,
                                         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);
Cpu0FI->setFormalArgInfo(CCInfo.getNextStackOffset(),
                           Cpu0CCInfo.hasByValArg());

Function::const_arg_iterator FuncArg =
    DAG.getMachineFunction().getFunction()->arg_begin();
bool UseSoftFloat = Subtarget.abiUsesSoftFloat();

Cpu0CCInfo.analyzeFormalArguments(Ins, UseSoftFloat, FuncArg);

// Used with vargs to accumulate 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];
    std::advance(FuncArg, Ins[i].OrigArgIndex - CurArgIdx);
    CurArgIdx = Ins[i].OrigArgIndex;
    EVT ValVT = VA.getValVT();
    ISD::ArgFlagsTy Flags = Ins[i].Flags;
    bool IsRegLoc = VA.isRegLoc();

    //@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()

    // 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);

    // Create load nodes to retrieve arguments from the stack
    SDValue FIN = DAG.getFrameIndex(FI, getPointerTy(DAG.getDataLayout()));
    SDValue Load = DAG.getLoad(ValVT, DL, Chain, FIN,
                               MachinePointerInfo::getFixedStack(FI),
                               false, false, false, 0);
    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;
        std::advance(FuncArg, Args[I].OrigArgIndex - CurArgIdx);
        CurArgIdx = Args[I].OrigArgIndex;

        if (ArgFlags.isByVal()) {
            handleByValArg(I, ArgVT, ArgVT, CCValAssign::Full, ArgFlags);
            continue;
        }

        MVT RegVT = getRegVT(ArgVT, FuncArg->getType(), nullptr, IsSoftFloat);

        if (!FixedFn(I, ArgVT, RegVT, CCValAssign::Full, ArgFlags, CCIInfo))
            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 = RoundUpToAlignment(ArgFlags.getByValSize(), RegSize);
unsigned Align = std::min(std::max(ArgFlags.getByValAlign(), RegSize),
                        RegSize * 2);

if (useRegsForByval())
    allocateRegs(ByVal, ByValSize, Align);

// Allocate space on caller's stack.
ByVal.Address = CCInfo.AllocateStack(ByValSize - RegSize * ByVal.NumRegs,
                                      Align);
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 create 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 get incoming arguments information by CCInfo(CallConv,..., ArgLocs, ...) before enter “**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 argument pass in register. On the contrary, when VA.isMemLoc() is true, meaning the argument pass in memory stack. When pass in register, it marks the register “live in” and copy directly from the register. When pass 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 first all arguments in stack frame.

Before take care the arguments as above, it calls analyzeFormalArguments(). In analyzeFormalArguments() it calls fixedArgFn() which returns the function pointer of CC_Cpu0O32() or CC_Cpu0S32(). ArgFlags.isByVal() will be true if when meet “struct pointer byval” keyword such as “%struct.S* byval” in tailcall.ll. When llc -cpu0-s32-calls=false the stack offset begins from 8 (in case the argument registers need spill out) while llc -cpu0-s32-calls=true stack offset begins from 0.

For instance of example code ch9_1.cpp with llc -cpu0-s32-calls=true (use 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 for 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 circled parts of DAG nodes as Figure 9.3 and Figure 9.4 for llc -cpu0-s32-calls=true and llc -cpu0-s32-calls=false, respectively. The root node at bottom is created by

Ibdex/input/ch9_incoming.cpp

```
int sum_i(int x1, int x2, int x3)
{
    int sum = x1 + x2 + x3;

    return sum;
}

Jonathan@MacBook-Pro:~/Documents$ clang -O3 -target mips-unknown-linux-gnu -c
ch9_incoming.cpp -emit-llvm -o ch9_incoming.bc
Jonathan@MacBook-Pro:~/Documents$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/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).

⁴ <http://jonathan2251.github.io/lbd/globalvar.html#global-variable>

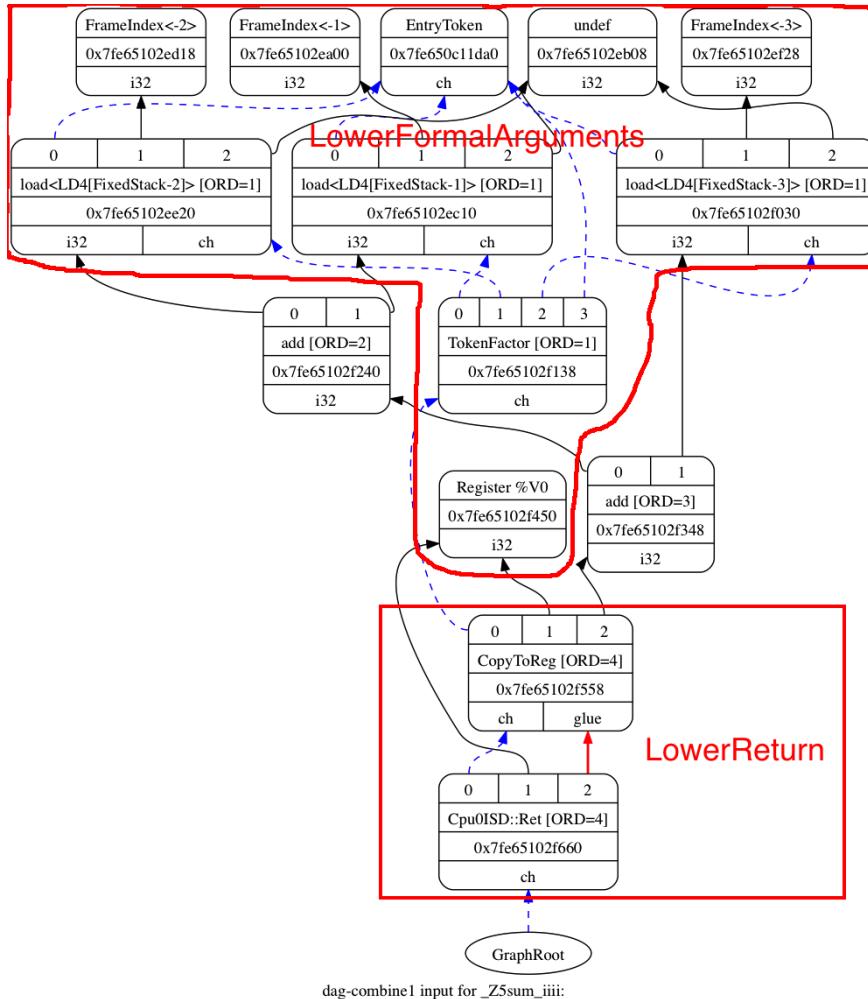
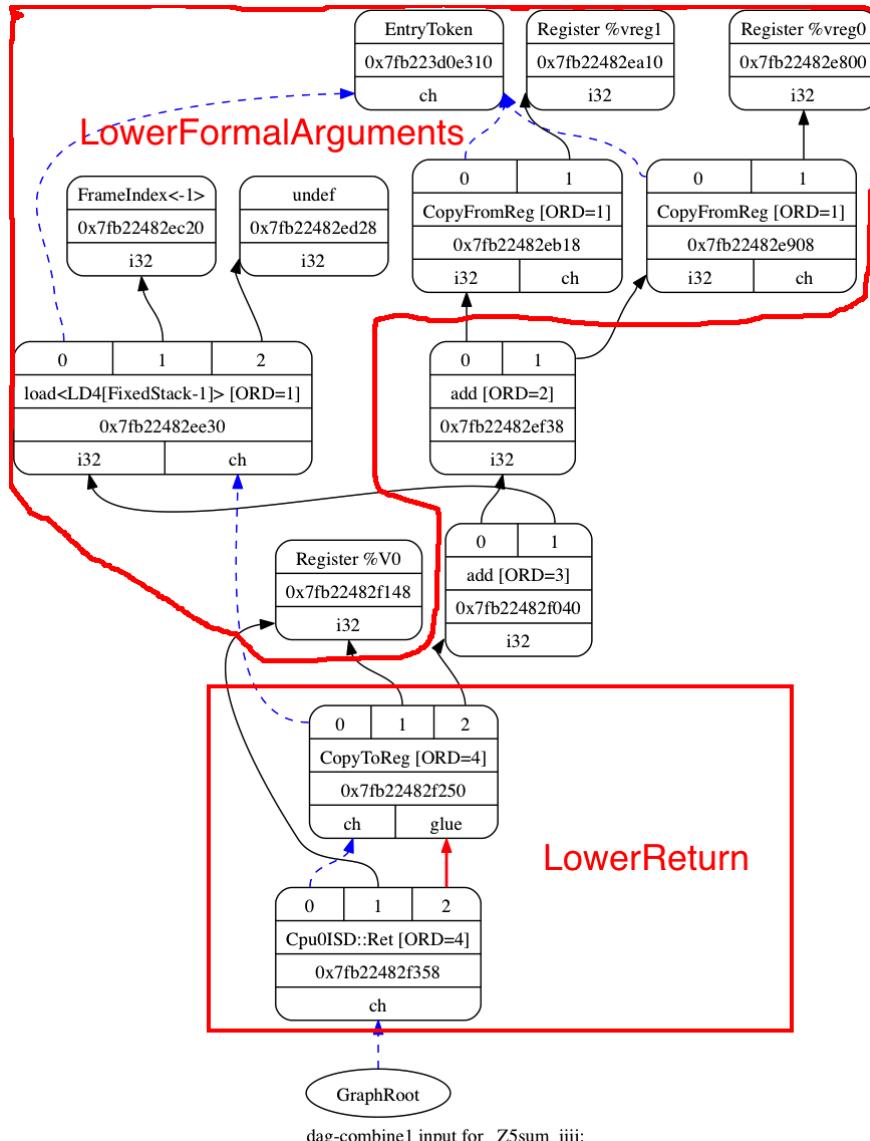


Figure 9.3: Incoming arguments DAG created for ch9_incoming.cpp with `-cpu0-s32-calls=true`

Figure 9.4: Incoming arguments DAG created for `ch9_incoming.cpp` with `-cpu0-s32-calls=false`

Ibdex/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;
    bit isCodeGenOnly = 1;
}

def calltarget : Operand<iPTR> {
    let EncoderMethod = "getJumpTargetOpValue";
}

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
        let DecoderMethod = "DecodeJumpAbsoluteTarget";
    // #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 = [Ch10_1] in {
def JR     : JumpFR<0x3c, "ret", GPROut>;
}

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;
    const MCSymbol *Symbol;

    switch (MO.getTargetFlags()) {
        case Cpu0II::MO_GOT_CALL: Kind = MCSymbolRefExpr::VK_Cpu0_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/InstPrinter/Cpu0InstPrinter.cpp

```

static void printExpr(const MCEexpr *Expr, raw_ostream &OS) {

```

```
...
switch (Kind) {
...
case MCSymbolRefExpr::VK_Cpu0_GOT_CALL: OS << "%call16("; break;
...
}
...
```

Ibdex/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 = nullptr) {

    unsigned Kind = Fixup.getKind();

    // Add/subtract and shift
    switch (Kind) {

        case Cpu0::fixup_Cpu0_CALL16:
        ...
    }
    ...
}
```

Ibdex/chapters/Chapter9_1/MCTargetDesc/Cpu0ELFOBJECTWriter.cpp

```
unsigned Cpu0ELFOBJECTWriter::GetRelocType(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;

        ...
    }
    ...
}
```

Ibdex/chapters/Chapter9_1/MCTargetDesc/Cpu0FixupKinds.h

```
enum Fixups {
```

```

// resulting in - R_CPU0_CALL16.
fixup_Cpu0_CALL16,
...
.
}

```

Ibdex/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)

        Fixups.push_back(MCFixup::create(0, Expr,
                                         MCFixupKind(Cpu0::fixup_Cpu0_PC24)));

    ...
}

unsigned Cpu0MCCodeEmitter::
getExprOpValue(const MCEexpr *Expr, SmallVectorImpl<MCFixup> &Fixups,
               const MCSubtargetInfo &STI) const {

    switch(cast<MCsymbolRefExpr>(Expr)->getKind()) {

        case MCSymbolRefExpr::VK_Cpu0_GOT_CALL:
            FixupKind = Cpu0::fixup_Cpu0_CALL16;
            break;

        ...
    }
    ...
}

```

Ibdex/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),

        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; }
        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.

...
};

The JSUB and JALR defined in Cpu0InstrInfo.td as above all use Cpu0JmpLink node. They are distinguishable since JSUB use “imm” operand while JALR uses register operand.


```

[Index/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 belong 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. The Cpu0GenDAGISel.inc contains the TablGen generated information about JSUB and JALR pattern match information as follows,
```

```

/*SwitchOpcode*/ 74, TARGET_VAL(Cpu0ISD::JmpLink), // ->734
/*660*/
OPC_RecordNode, // #0 = 'Cpu0JmpLink' chained node
/*661*/
OPC_CaptureGlueInput,
/*662*/
OPC_RecordChild1, // #1 = $target
/*663*/
OPC_Scope, 57, /*->722*/ // 2 children in Scope
/*665*/
OPC_MoveChild, 1,
/*667*/
OPC_SwitchOpcode /*3 cases */, 22, TARGET_VAL(ISD::Constant),
// ->693
/*671*/
OPC_MoveParent,
/*672*/
OPC_EmitMergeInputChains1_0,
/*673*/
OPC_EmitConvertToTarget, 1,
/*675*/
OPC_Scope, 7, /*->684*/ // 2 children in Scope
/*684*/
/*Scope*/ 7, /*->692*/
/*685*/
OPC_MorphNodeTo, TARGET_VAL(Cpu0::JSUB), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
    0/*#VTS*/, 1/*#Ops*/, 2,
    // Src: (Cpu0JmpLink (imm:iPTR):$target) - Complexity = 6
    // Dst: (JSUB (imm:iPTR):$target)
    0, /*End of Scope*/
/*SwitchOpcode*/ 11, TARGET_VAL(ISD::TargetGlobalAddress), // ->707

```

```

/*696*/          OPC_CheckType, MVT::i32,
/*698*/          OPC_MoveParent,
/*699*/          OPC_EmitMergeInputChainsl_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_EmitMergeInputChainsl_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_EmitMergeInputChainsl_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/
cmake_debug_build/Debug/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/src/lib/CodeGen/SelectionDAG/SelectionDAGBuilder.
cpp, ...
...
0. Program arguments: /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/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

Figure 9.2 depicts two steps to take care arguments passing. One is store outgoing arguments in caller function, and the other is load incoming arguments in 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 to do this. The implementation as follows,

lbdex/chapters/Chapter9_2/Cpu0MachineFunction.h

```
/// \brief Create a MachinePointerInfo that has a Cpu0CallEntry object
/// representing a GOT entry for an external function.
MachinePointerInfo callPtrInfo(const StringRef &Name);

/// \brief Create a MachinePointerInfo that has a Cpu0CallEntry object
/// representing a GOT entry for a global function.
MachinePointerInfo callPtrInfo(const GlobalValue *Val);
```

lbdex/chapters/Chapter9_2/Cpu0MachineFunction.cpp

```
MachinePointerInfo Cpu0FunctionInfo::callPtrInfo(const StringRef &Name) {
    const Cpu0CallEntry *&E = ExternalCallEntries[Name];

    if (!E)
        E = new Cpu0CallEntry(Name);

    return MachinePointerInfo(E);
}

MachinePointerInfo Cpu0FunctionInfo::callPtrInfo(const GlobalValue *Val) {
    const Cpu0CallEntry *&E = GlobalCallEntries[Val];

    if (!E)
        E = new Cpu0CallEntry(Val);

    return MachinePointerInfo(E);
}
```

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,
                       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, 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, 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;

```

Ibdex/chapters/Chapter9_2/Cpu0SelLowering.cpp

```

SDValue
Cpu0TargetLowering::passArgOnStack(SDValue StackPtr, unsigned Offset,
                                    SDValue Chain, SDValue Arg, 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(), false,
                            false, 0);
    }

    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(),
                        /*isVolatile=*/ true, false, 0);
}

void Cpu0TargetLowering::
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
    // -relocation-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 is 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 = getTargetMachine().getRelocationModel() == Reloc::PIC_;
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.CS && CLI.CS->isMustTailCall())
    report_fatal_error("failed to perform tail call elimination on a call "
                      "site marked musttail");

if (IsTailCall)
    ++NumTailCalls;
//@TailCall 1 }

// 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 = RoundUpToAlignment(NextStackOffset, StackAlignment);
SDValue NextStackOffsetVal = DAG.getIntPtrConstant(NextStackOffset, DL, true);

//@TailCall 2 {
if (!IsTailCall)
    Chain = DAG.getCALLSEQ_START(Chain, NextStackOffsetVal, 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);
    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;
SDValue CalleeLo;
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);
    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,
                                    SDLoc DL, SelectionDAG &DAG,
                                    SmallVectorImpl<SDValue> &InVals,
                                    const SDNode *CallNode,
                                    const Type *RetTy) const {
    // Assign locations to each value returned by this call.
    SmallVector<CCValAssign, 16> RVLocs;
    CCState CCInfo(CallConv, IsVarArg, DAG.getMachineFunction(),
                   RVLocs, *DAG.getContext());

    Cpu0CC Cpu0CCInfo(CallConv, ABI.IsQ32(), 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.");

    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, FuncArgs[Args[I].OrigArgIndex].Ty, CallNode,
                             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 enter “**for loop**” and set stack alignment with 8 bytes. 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 load 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.

Ibdex/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:$amt),
                         "!ADJCALLSTACKDOWN $amt",
                         [(callseq_start timm:$amt)]>;
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 just discard these 2 pseudo instructions, and LLVM will add offset to stack.

Ibdex/chapters/Chapter9_2/Cpu0InstrInfo.cpp

```

Cpu0InstrInfo::Cpu0InstrInfo(const Cpu0Subtarget &STI)
:
Cpu0GenInstrInfo(Cpu0::ADJCALLSTACKDOWN, Cpu0::ADJCALLSTACKUP),

```

Ibdex/chapters/Chapter9_2/Cpu0SEFrameLowering.h

```

void eliminateCallFramePseudoInstr(MachineFunction &MF,
                                    MachineBasicBlock &MBB,
                                    MachineBasicBlock::iterator I) const override;

```

Ibdex/chapters/Chapter9_2/Cpu0SEFrameLowering.cpp

```
// Eliminate ADJCALLSTACKDOWN, ADJCALLSTACKUP pseudo instructions
void Cpu0SEFrameLowering::eliminateCallFramePseudoInstr(MachineFunction &MF, MachineBasicBlock &MBB,
                                                          MachineBasicBlock::iterator I) const {
    // Simply discard ADJCALLSTACKDOWN, ADJCALLSTACKUP instructions.
    MBB.erase(I);
}
```

9.3.2 Read Lowercall() with Graphviz's help

The whole DAGs created for outgoing arguments as Figure 9.5 for ch9_outgoing.cpp with cpu032I. LowerCall() will generate the DAG nodes as Figure 9.6 for ch9_outgoing.cpp with cpu032I. The corresponding code of DAGs Store and TargetGlobalAddress are listed in the figure, user can match the other DAGs to function LowerCall() easy. Through Graphviz tool with llc option -view-dag-combine1-dags, you can design a small input C or llvm IR source code and check the DAGs to understand the code in LowerCall() and LowerFormalArguments(). In later section of this chapter, there are variable arguments and dynamic stack allocation support. You can design the input example with this features and check the DAGs with these two function again to make sure you know the code in these two function. About Graphviz, please reference 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,

Ibdex/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/cmake_debug_build/
Debug/bin/llvm-dis ch9_outgoing.bc -o -
...
define i32 @_Z10call_sum_iv() #0 {
    %1 = tail call i32 @_Z5sum_ii(i32 1)
    ret i32 %1
}
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/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_zycd0000gn/T/dag._Z10call_sum_iv-
0dfa1.dot'... done.
Running 'Graphviz' program...
```

Mentioned in last section, llc -cpu0-s32-calls=true uses S32 calling convention which pass all arguments at registers while llc -cpu0-s32-calls=false uses O32 pass first two arguments at registers and other arguments at stack. The result as follows,

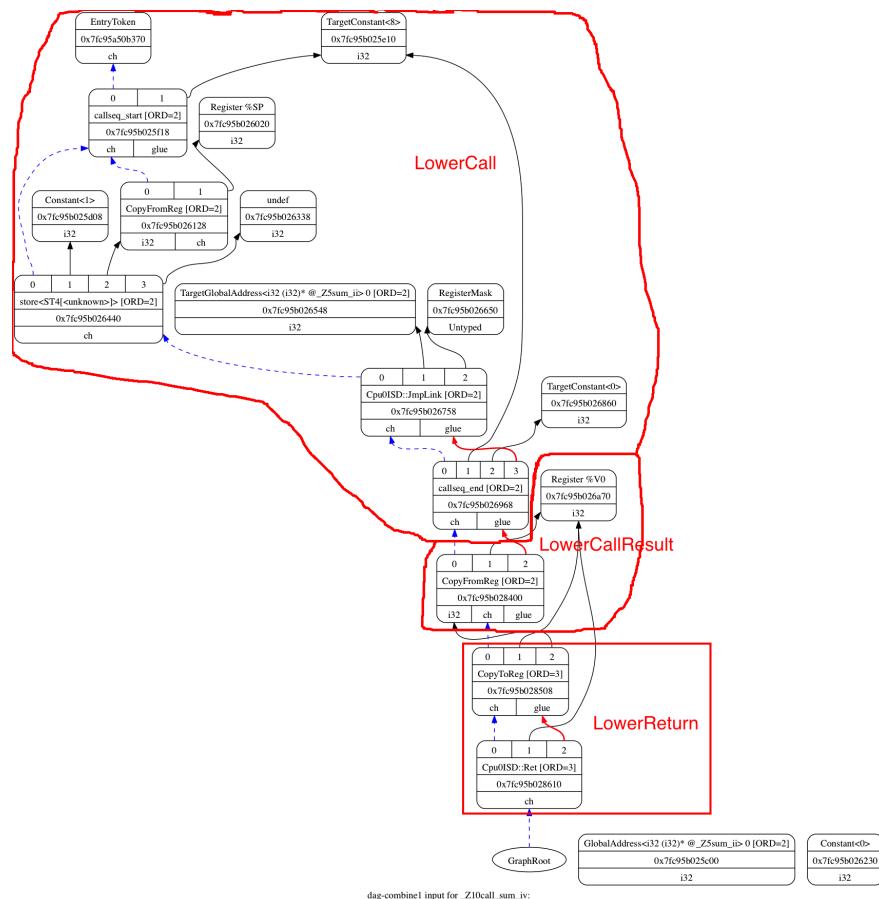


Figure 9.5: Outgoing arguments DAG created for ch9_outgoing.cpp with `-cpu0-s32-calls=true`

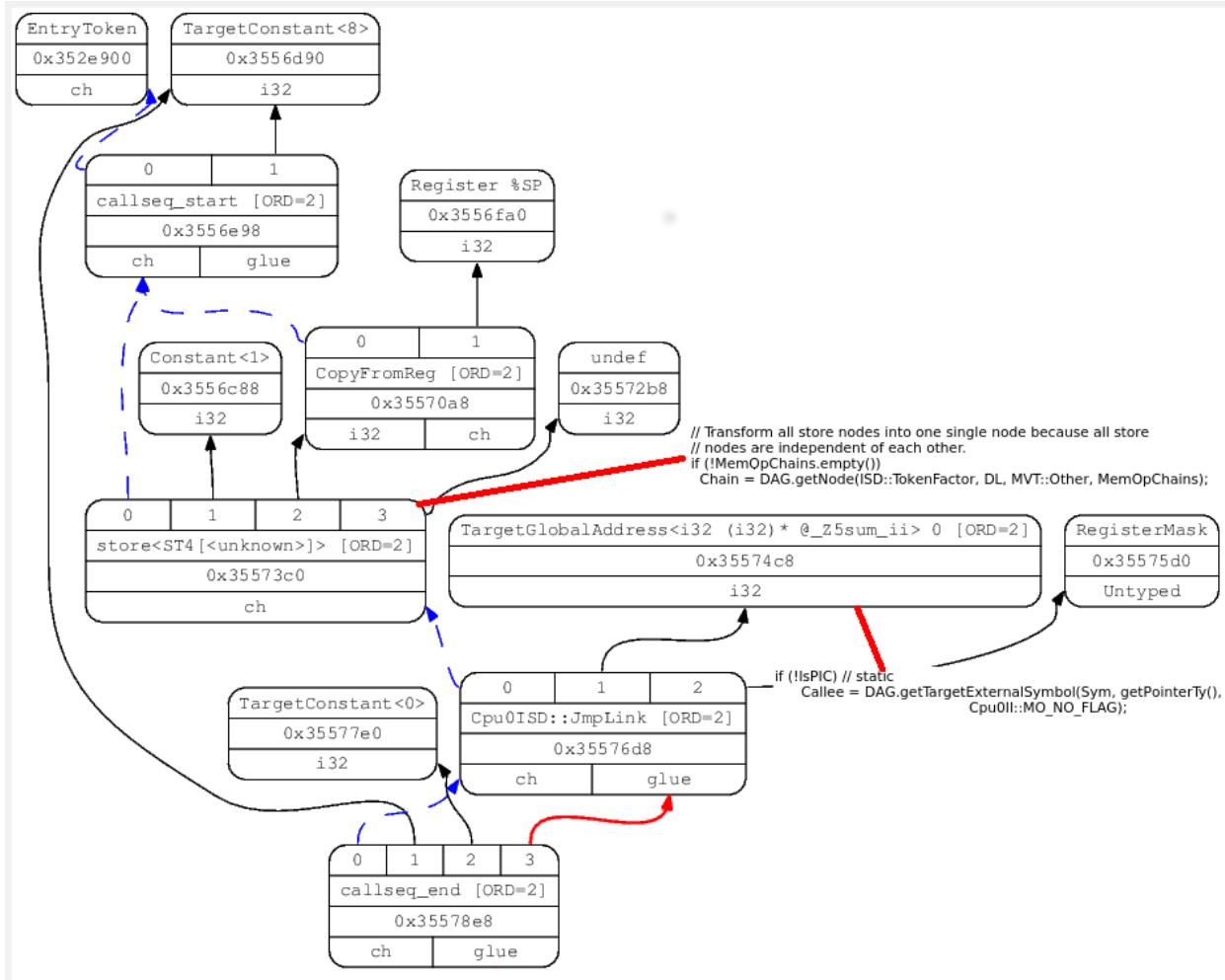


Figure 9.6: Outgoing arguments DAG created by LowerCall() for ch9_outgoing.cpp with -cpu0-s32-calls=true

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/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_iuiiii
.align 2
.type _Z5sum_iuiiii,@function
.ent _Z5sum_iuiiii      # @_Z5sum_iuiiii
_Z5sum_iuiiii:
.frame $fp,32,$lr
.mask 0x00000000,0
.set noreorder
.cupload $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)
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_iuiiii
$tmp0:
.size _Z5sum_iuiiii, ($tmp0)-_Z5sum_iuiiii

.globl main
.align 2
```

```

.type main,@function
.ent main                               # @main
main:
.frame      $fp,40,$lr
.mask       0x00004000,-4
.set noreorder
.cupload   $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   $t9, %call16(_Z5sum_iiiiii)($gp)
jalr $t9
nop
st   $2, 28($sp)
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/cmake_debug_build/
Debug/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
.cupload      $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   $t9, %call16(_Z5sum_iiiiii) ($gp)
addiu $4, $zero, 1
addiu $5, $zero, 2
jalr $t9
nop
st   $2, 28($sp)
ld   $lr, 36($sp)          # 4-byte Folded Reload
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.

Ibdex/input/ch9_1_2.cpp


```
.type $_ZZ4mainEls,@object      # @_ZZ4mainEls
.section     .rodata.str1.1,"aMS",@progbits,1
$_ZZ4mainEls:
    .asciz      "Hello"
    .size $_ZZ4mainEls, 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/cmake_debug_build
/Debug/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,
                                         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,
                               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 had defined the calling convention in early chapter as follows,

Ibdex/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 return value didn't over 4 registers size; If it over 4 registers size, cpu0 will save them with pointer. For explanation, let's run Chapter9_2/ with ch9_2_1.cpp and explain with this example.

Ibdex/input/ch9_2_1.cpp

```
extern "C" int printf(const char *format, ...);  
  
struct Date  
{  
    int year;  
    int month;  
    int day;  
    int hour;  
    int minute;  
    int second;  
};  
Date gDate = {2012, 10, 12, 1, 2, 3};  
  
struct Time  
{  
    int hour;  
    int minute;  
    int second;  
};  
Time gTime = {2, 20, 30};  
  
Date getDate()  
{  
    return gDate;  
}  
  
Date copyDate(Date date)  
{  
    return date;  
}  
  
Date copyDate(Date* date)  
{  
    return *date;  
}  
  
Time copyTime(Time time)  
{  
    return time;  
}  
  
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);
#ifdef PRINT_TEST
    printf("date1 = %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", 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);
    if (time3.hour == 1 && time3.minute == 10 && time3.second == 12)
        printf(", PASS\n");
    else
        printf(", FAIL\n");
#endif

    return 0;
}

```

```

Jonathan:~/input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -mcpu=cpu032I -relocation-model=pic -filetype=asm
ch9_2_1.bc -o -
.section .mdebug.abi32
.previous
.file "ch9_2_1.bc"
.text
.globl _Z7getDatev
.align 2
.type _Z7getDatev,@function
.ent _Z7getDatev          # @_Z7getDatev
_Z7getDatev:
.cfi_startproc
.frame $sp,0,$lr
.mask 0x00000000,0

```

```

.set noreorder
.cupload $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
...
.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
.cupload $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_2_2.cpp include C++ class “Date” implementation. It can be translated into cpu0 backend too since the front end (clang in this example) translate them into C language form. If you mark the “if hasStructRetAttr()” part from both of above functions, the output cpu0 code for ch9_2_1.cpp will use \$3 instead of \$2 as return register as follows,

```

.text
.section .mdebug.abiS32
.previous
.file "ch9_2_1.bc"
.globl _Z7getDatev
.align 2
.type _Z7getDatev,@function
.ent _Z7getDatev           # @_Z7getDatev

_Z7getDatev:
.frame $fp,0,$lr
.mask 0x00000000,0
.set noreorder
.cupload $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 ask return struct varaible address 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.

Ibdex/chapters/Chapter9_1/Cpu0ISelLowering.cpp

```

void Cpu0TargetLowering::
copyByValRegs(SDValue Chain, 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);
    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),
                                     false, false, 0);
        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,
                                         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.");
        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;
    }
}
...
}
    
```

Index/chapters/Chapter9_2/Cpu0ISelLowering.cpp

```

// Copy ByVal arg to registers and stack.
void Cpu0TargetLowering::
passByValArg(SDValue Chain, 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(Flags.getByValAlign(), RegSizeInBytes);
    EVT PtrTy = getPointerTy(DAG.getDataLayout()),
    RegTy = MVT::getIntegerVT(RegSizeInBytes * 8);

    if (ByVal.NumRegs) {
        const ArrayRef<MCPhysReg> ArgRegs = CC.intArgRegs();
        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(), false, false, false,
                                 Alignment);
    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), false, false, false,
            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));

        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),
                      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,

Ibdex/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 if it meet **byval** for in callee function as follows,

Ibdex/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 optimization

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 here⁶.

The **tailcall** appeared in Cpu0ISelLowering.cpp and Cpu0InstrInfo.td are used to make tail call optimization.

Ibdex/input/ch9_2_3_tailcall.cpp

```
int factorial(int x)
{
  if (x > 0)
    return x*factorial(x-1);
  else
    return 1;
}

int test_tailcall(int a)
{
  return factorial(a);
}
```

Run Chapter9_2/ with ch9_2_3_tailcall.cpp will get the following result.

```
JonathantekiiMac:input Jonathan$ clang -O1 -target mips-unknown-linux-gnu -c
ch9_2_3_tailcall.cpp -emit-llvm -o ch9_2_3_tailcall.bc
JonathantekiiMac:input Jonathan$ ~/llvm/test/cmake_debug_build/bin/
llvm-dis ch9_2_3_tailcall.bc -o -
...
; Function Attrs: nounwind readnone
define i32 @_Z9factoriali(i32 %x) #0 {
  %1 = icmp sgt i32 %x, 0
  br il %1, label %tailrecurse, label %tailrecurse._crit_edge
```

⁵ http://en.wikipedia.org/wiki/Tail_call

⁶ <http://llvm.org/docs/CodeGenerator.html#tail-call-optimization>

```

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
}
...
Jonathan@tekiiMac:~/input Jonathan$ ~/llvm/test/cmake_debug_build/bin/
llc -march=cpu0 -mcpu=cpu032II -relocation-model=static -filetype=asm
-enable-cpu0-tail-calls ch9_2_3_tailcall.bc -stats -o -
.text
.section .mdebug.abi32
.previous
.file "ch9_2_3_tailcall.bc"
.globl _Z9factoriali
.align 2
.type _Z9factoriali,@function
.ent _Z9factoriali           # @_Z9factoriali
_Z9factoriali:
.frame $sp,0,$lr
.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 is applied in cpu032II only for this example (it use “jmp _Z9factoriali” instead of “jsub _Z9factoriali”). Tail call share caller and callee stack but cpu032I (pass all arguments in stack) not satisfy the following statement, NextStackOffset <= FI.getIncomingArgSize() in isEligibleForTailCallOptimization(), and return false for this function as follows,

[Index/chapters/Chapter9_2/Cpu0SEISelLowering.cpp](#)

```

bool Cpu0SEITargetLowering::
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();
}

```

[Index/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.CS && CLI.CS->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, NextStackOffsetVal, 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 DAGs of LowerCallResult() and LowerReturn() created are needless. It creates DAGs as [Figure 9.7](#) for ch9_2_3_tailcall.cpp as follows,

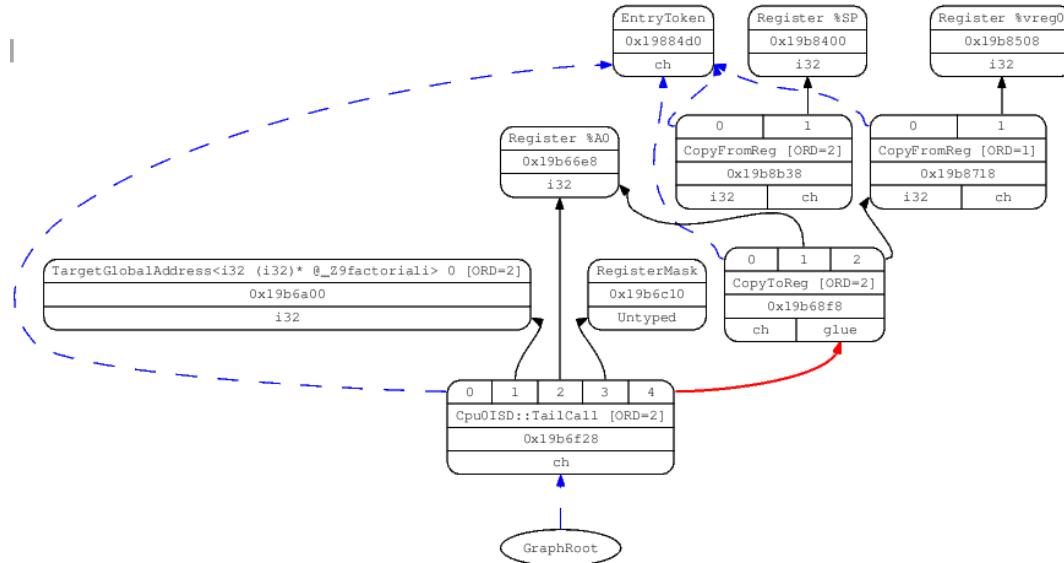


Figure 9.7: Outgoing arguments DAG created for ch9_2_3_tailcall.cpp

Finally, the tail call DAG process as the following table.

Table 9.2: tail call DAG translation

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,

Ibdex/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,

Ibdex/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>;
}
```

Ibdex/chapters/Chapter9_1/Cpu0AsmPrinter.h

```
// tblgen'ered function.
bool emitPseudoExpansionLowering(MCStreamer &OutStreamer,
    const MachineInstr *MI);
```

Ibdex/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);
    return;
}

MachineBasicBlock::const_instr_iterator I = MI;
MachineBasicBlock::const_instr_iterator E = MI->getParent()->instr_end();

MCInst TmpInst0;
do {
    // Do any auto-generated pseudo lowerings.
    if (emitPseudoExpansionLowering(*OutStreamer, &*I))
        continue;
    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 do tail call optimization in ch9_2_3_tailcall.cpp since the limitation of arguments size not satisfied. If run with clang -O3 option, it can get the same or better performance than tail call as follows,

```

JonathanMac:input Jonathan$ clang -O1 -target mips-unknown-linux-gnu -c
ch9_2_3_tailcall.cpp -emit-llvm -o ch9_2_3_tailcall.bc
JonathanMac:input Jonathan$ ~/llvm/test/cmake_debug_build/bin/
llvm-dis ch9_2_3_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

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 %tailrecuse.i.preheader, label %_Z9factoriali.exit

tailrecuse.i.preheader:                                ; preds = %0
    br label %tailrecuse.i

tailrecuse.i:                                         ; preds = %tailrecuse.i,
    %tailrecuse.i.preheader
    %x.tr2.i = phi i32 [ %2, %tailrecuse.i ], [ %a, %tailrecuse.i.preheader ]
    %accumulator.tr1.i = phi i32 [ %3, %tailrecuse.i ], [ 1, %tailrecuse.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 %tailrecuse.i, label %_Z9factoriali.exit.loopexit

_Z9factoriali.exit.loopexit:                          ; preds = %tailrecuse.i
    %.lcssa = phi i32 [ %3, %tailrecuse.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
}
...
Jonathan@tekiMac:~/input$ Jonathan$ ~/llvm/test/cmake_debug_build/bin/
llc -march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm
ch9_2_3_tailcall.bc -o -
    .text
    .section .mdebug.abiS32
    .previous
    .file "ch9_2_3_tailcall.bc"
    .globl _Z9factoriali
    .align 2
    .type _Z9factoriali,@function
    .ent _Z9factoriali      # @_Z9factoriali
_Z9factoriali:
    .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:                                     # %tailrecuse
                                                # => 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:                                # %tailrecuse._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:                                # %tailrecuse.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
    nop
    .set macro
    .set reorder
    .end _Z13test_tailcalli
$tmp1:
    .size _Z13test_tailcalli, ($tmp1)-_Z13test_tailcalli

```

According above llvm IR, clang -O3 option remove recursion into loop by inline the callee recursion function. This is a front end optimization through cross over function analysis.

9.6 Other features supporting

This section support features of \$gp register caller saved register in PIC addressing mode, variable number of arguments and dynamic stack allocation.

Run Chapter9_2/ with ch9_3.cpp to get the following error,

Ibdex/input/ch9_3.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;
}

int test_vararg()
{
    int a = sum_i(6, 0, 1, 2, 3, 4, 5);

    return a;
}

118-165-78-230:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3.cpp -emit-llvm -o ch9_3.bc
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_3.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
```

Ibdex/input/ch9_4.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) * x1);
    *b = 1111;
    int weight = sum(6*x1, x2, x3, x4, 2*x5, x6);

    return weight;
}

int test_alloc()
{
    int a = weight_sum(1, 2, 3, 4, 5, 6); // 31

    return a;
}

```

Run Chapter9_3 with ch9_4.cpp will get the following error.

```

118-165-72-242:input Jonathan$ clang -target mips-unknown-linux-gnu -I/
Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/
SDKs/MacOSX10.8.sdk/usr/include/ -c ch9_4.cpp -emit-llvm -o ch9_4.bc
118-165-72-242:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_4.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 support “**jsub**” bits address range access. We add “**jalr**” to cpu0 and expand it to 32 bit address. We did this change for two reasons. One is cpu0 can be expanded to 32 bit address space by only adding this instruction. The other is cpu0 as well as this book are designed for teaching purpose. We reserve “**jalr**” as PIC mode for dynamic linking function to demonstrates:

1. How caller handle the caller saved register \$gp in calling the function
2. How the code in the shared libray function use \$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 change 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 example of dynamic link function is used in share library. Share library include 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 it access

cannot be decided at link time. Even so, we still can calculate the distance between the global variable address and the start address of shared library function when it be loaded.

Let's run Chapter9_3/ with ch9_gprestore.cpp to get the following result. We putting the comments in the result for explanation.

Ibdex/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/cmake_debug_build/
Debug/bin/llc -march=cpu0 -mcpu=cpu032II-cpu0-s32-calls=true
-relocation-model=pic -filetype=asm ch9_gprestore.bc -o -
...
.cupload      $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
    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**, reference Table 8-1 of “MIPSpro TM Assembly Language Programmer’s Guide”⁷. In other word, **\$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. Anyway, it is kept in Cpu0 and make this feature can be removed by not define it in Cpu0Config.h.

⁷ <http://math-atlas.sourceforge.net-devel/assembly/007-2418-003.pdf>

The #ifdef ENABLE_GPRESTORE part of code in Cpu0 can be removed but it come with the cost of reserve \$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 rare and the performance advantage can be ignored in dynamic link, so we keep this feature in Cpu0 and provide readers this feature. Even with this point, I really prefer to reserve \$gp as a specific register in PIC. It will save a lot of trouble in programming. When reserve \$gp, .cupload can be disabled by option “-cpu0-reserve-gp”. The .cupload is need even reserve \$gp. Consider prgrammer implement a boot code function with C and assembly mixed, programmer can set \$gp value through .cupload issue.

If enable “-cpu0-no-cupload”, and undefine ENABLE_GPRESTORE or enable “-cpu0-reserve-gp”, .cupload and \$gp save/restore won’t be issue as follow,

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -mcpu=cpu032II-cpu0-s32-calls=true
-relocation-model=pic -filetype=asm ch9_gprestore.bc -cpu0-no-cupload
-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
    ret   $lr
    nop
```

LLVM Mips 3.1 issues the .cupload and .cprestore and Cpu0 borrow it from that version. But now, llvm Mips replace .cupload with real instructions and remove .cprestore. It treats \$gp as reserved register in PIC mode. Since the Mips assembly document I reference say \$gp is caller save register, Cpu0 stay and follow this document at this point and supply reserve \$gp register as option.

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/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/ to issue “**.cprestore**” or the corresponding machine code before the first time of PIC function call.

Index/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 .cp	restore saves $gp.
        if (IsPIC && Cpu0FI->globalBaseRegFixed() && !Cpu0FI->getGPFI())
            Cpu0FI->setGPFI(MFI->CreateFixedObject(4, 0, true));
        if (Cpu0FI->needGPSaveRestore())
            MFI->setObjectOffset(Cpu0FI->getGPFI(), NextStackOffset);
    }
#endif

```

```
...
}
```

Ibdex/chapters/Chapter9_3/Cpu0MachineFunction.h

```
#ifdef ENABLE_GPRESTORE
    bool needGPSaveRestore() const { return getGPFI(); }
#endif
```

Ibdex/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
}
```

Ibdex/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
    ...
}
```

Ibdex/chapters/Chapter9_3/Cpu0InstrInfo.td

```
// When handling PIC code the assembler needs .cupload 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", []>;
```

Ibdex/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

//EmitInstruction() must exists or will have run time error.
void Cpu0AsmPrinter::EmitInstruction(const MachineInstr *MI) {

#ifdef ENABLE_GPRESTORE
    unsigned Opc = MI->getOpcode();
    SmallVector<MCInst, 4> MCInsts;

    switch (Opc) {
        case Cpu0::CPRESTORE: {
            const MachineOperand &MO = MI->getOperand(0);
            assert(MO.isImm() && "CPRESTORE's operand must be an immediate.");
            int64_t Offset = MO.getImm();

            if (OutStreamer->hasRawTextSupport()) {
                if (!isInt<16>(Offset)) {
                    EmitInstrWithMacroNoAT(MI);
                    return;
                }
            } else {
                MCInstLowering.LowerCPRESTORE(Offset, MCInsts);

                for (SmallVector<MCInst, 4>::iterator I = MCInsts.begin();
                     I != MCInsts.end(); ++I)
                    OutStreamer->EmitInstruction(*I, getSubtargetInfo());

                return;
            }
            break;
        }
        default:
            break;
    }
#endif
}
```

```
    ...
}
```

Ibdex/chapters/Chapter9_3/Cpu0MCInstLower.h

```
#ifdef ENABLE_GPRESTORE
void LowerCPRESTORE(int64_t Offset, SmallVector<MCInst, 4>& MCInsts);
#endif
```

Ibdex/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 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/
cmake_debug_build/Debug/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
```

```
...
.ent _Z5sum_iiiiiii          # @_Z5sum_iiiiiii
_Z5sum_iiiiiii:
...
.cupload $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/
cmake_debug_build/Debug/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 with llc -filetype=obj, you can 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. The ch9_1_3.cpp and ch9_1_4.cpp are example code more for test.

```
// 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 create file Cpu0EmitGPRestore.cpp which run as a function pass.

Ibdex/chapters/Chapter9_3/CMakeLists.txt

```
Cpu0EmitGPRestore.cpp
```

Ibdex/chapters/Chapter9_3/Cpu0TargetMachine.cpp

```
/// Cpu0 Code Generator Pass Configuration Options.
class Cpu0PassConfig : public TargetPassConfig {

#define ENABLE_GPRESTORE
    void addPreRegAlloc() override;
#endif

#define ENABLE_GPRESTORE
void Cpu0PassConfig::addPreRegAlloc() {
    if (!Cpu0ReserveGP) {
        // $gp is a caller-saved register.
        addPass(createCpu0EmitGPRestorePass(getCpu0TargetMachine()));
    }
    return;
}
#endif
```

Ibdex/chapters/Chapter9_3/Cpu0.h

```
#ifdef ENABLE_GPRESTORE
    FunctionPass *createCpu0EmitGPRestorePass (Cpu0TargetMachine &TM) ;
#endif
```

Ibdex/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
#define ENABLE_GPRESTORE

#include "Cpu0TargetMachine.h"
#include "Cpu0MachineFunction.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/Target/TargetInstrInfo.h"
#include "llvm/ADT/Statistic.h"

using namespace llvm;

#define DEBUG_TYPE "emit-gp-restore"

namespace {
    struct Inserter : public MachineFunctionPass {

        TargetMachine &TM;

        static char ID;
        Inserter(TargetMachine &tm)
            : MachineFunctionPass(ID), TM(tm) { }

        virtual const char *getPassName() const {
            return "Cpu0 Emit GP Restore";
        }

        bool runOnMachineFunction(MachineFunction &F);
    };
    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->getGPI();

for (MachineFunction::iterator MFI = F.begin(), MFE = F.end();
      MFI != MFE; ++MFI) {
    MachineBasicBlock& MBB = *MFI;
    MachineBasicBlock::iterator I = MFI->begin();

    /// IsLandingPad - 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.isLandingPad()) {
        // 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();
        // 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 support variable number of arguments since C language support this feature.

Run Chapter9_3/ with ch9_3.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.cpp -emit-llvm -o ch9_3.bc
118-165-76-131:input Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/Debug/bin/llc -march=cpu0 -mcpu=cpu032I -cpu0-s32-calls=false
-relocation-model=pic -filetype=asm ch9_3.bc -o ch9_3.cpu0.s
118-165-76-131:input Jonathan$ cat ch9_3.cpu0.s
    .section .mdebug.abi32
    .previous
    .file "ch9_3.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
    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)             // sum
    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:
    .frame $sp,88,$lr
    .mask 0x00004000,-4
    .set noreorder
    .cupload $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 $1r
nop
.set macro
.set reorder
.end _Z11test_varargv
$tmp1:
.size _Z11test_varargv, ($tmp1)-_Z11test_varargv

```

The analysis of output ch9_3.cpu0.s as above in comment. As above code, in # BB#0, we get the first argument “**amount**” from “**Id \$2, 24(\$fp)**” since the stack size of the callee function “**_Z5sum_iiz()**” is 24. And then set argument pointer, arg_ptr, to 0(\$fp), &arg[1]. Next, check i < amount in block \$BB0_1. If i < amount, than enter into \$BB0_2. In \$BB0_2, it do sum += *arg_ptr as well as arg_ptr+=4. In # BB#3, do i+=1.

To support variable number of arguments, the following code needed to add in Chapter9_3/. The ch9_3_2.cpp is C++ template example code, it can be translated into cpu0 backend code too.

Ibdex/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;
    /// 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, SDLoc DL, SelectionDAG &DAG) const;
    ...
};

```

Ibdex/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);
}

```

```

    ...
}

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), false, false, 0);
}

/// 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,
                                         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,
                                         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 = RoundUpToAlignment(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.
    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 = DAGgetStore(Chain, DL, ArgValue, PtrOff,
                                     MachinePointerInfo(), false, false, 0);
        cast<StoreSDNode>(Store.getNode())->getMemOperand()->setValue(
            (Value *)nullptr);
        OutChains.push_back(Store);
    }
}

```

Ibdex/input/ch9_3_2.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 = sum<int>(6, 0, 1, 2, 3, 4, 5);

    return a;
}
```

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 code correction in chapter “Run backend” through the CPU0 Verilog language machine.

9.6.3 Dynamic stack allocation support

Even though C language very rare to use dynamic stack allocation, there are languages use it frequently. The following C example code use it.

Chapter9_3 support dynamic stack allocation with the following code added.

Ibdex/chapters/Chapter9_3/Cpu0SEFrameLowering.h

```
bool spillCalleeSavedRegisters(MachineBasicBlock &MBB,
                                MachineBasicBlock::iterator MI,
                                const std::vector<CalleeSavedInfo> &CSI,
                                const TargetRegisterInfo *TRI) const override;
```

Ibdex/chapters/Chapter9_3/Cpu0SEFrameLowering.cpp

```
void Cpu0SEFrameLowering::emitPrologue(MachineFunction &MF,
                                         MachineBasicBlock &MBB) const {
```

⁸ <http://developer.mips.com/clang-llvm/>

```

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)) {
    // Insert instruction "move $fp, $sp" at this location.
    BuildMI(MBB, MBBI, dl, TII.get(ADDu), FP).addReg(SP).addReg(ZERO)
        .setMIFlag(MachineInstr::FrameSetup);

    // emit ".cfi_def_cfa_register $fp"
    unsigned CFIIndex = MMI.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);
    }
}

bool Cpu0SEFrameLowering::
spillCalleeSavedRegisters(MachineBasicBlock &MBB,
                         MachineBasicBlock::iterator MI,
                         const std::vector<CalleeSavedInfo> &CSI,
                         const TargetRegisterInfo *TRI) const {

    MachineFunction *MF = MBB.getParent();
    MachineBasicBlock *EntryBlock = MF->begin();
    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
        // RA 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();
        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;
}

```

[Index/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);

}

```

[Index/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;
else

}

```

Run Chapter9_3 with ch9_4.cpp will get the following correct result.

```

118-165-72-242:input Jonathan$ clang -I/Applications/Xcode.app/Contents/
Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.9.sdk/usr/include/
-c ch9_4.cpp -emit-llvm -o ch9_4.bc
118-165-72-242:input Jonathan$ llvm-dis ch9_4.bc -o ch9_4.ll
118-165-72-242:input Jonathan$ cat ch9_4.ll
; ModuleID = 'ch9_4.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_iiiiii(i32 %x1, i32 %x2, i32 %x3, i32 %x4, i32 %x5, i32 %x6)
nounwind uwtable ssp {
    ...
    %10 = alloca i8, i64 %9      // int *b = (int*)alloca(sizeof(int) * x1);
    %11 = bitcast i8* %10 to i32*
    store i32* %11, i32** %b, align 8
    %12 = load i32** %b, align 8
    store i32 1111, i32* %12, align 4    // *b = 1111;
    ...
}
...
.

118-165-72-242:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -mcpu=cpu032I -cpu0-s32-calls=false
-relocation-model=pic -filetype=asm ch9_4.bc -o ch9_4.cpu0.s
118-165-72-242:input Jonathan$ cat ch9_4.cpu0.s
...
_Z10weight_sumiiiii:
.cfi_startproc
.frame $fp,80,$lr
.mask 0x00004080,-4
.set noreorder
.cupload $t9
.set nomacro
# BB#0:
    addiu $sp, $sp, -80
$tmp6:
    .cfi_def_cfa_offset 80
    st    $lr, 76($sp)          # 4-byte Folded Spill
    st    $7, 72($sp)          # 4-byte Folded Spill
$tmp7:
    .cfi_offset 14, -4
$tmp8:
    .cfi_offset 7, -8
    move $sp, $fp
$tmp9:
    .cfi_def_cfa_register 11
    cprestore 24
    ld    $7, %got(__stack_chk_guard) ($gp)
    ld    $2, 0($7)
    st    $2, 68($fp)
    ld    $2, 80($fp)
    st    $2, 64($fp)
    ld    $2, 84($fp)
    st    $2, 60($fp)
    ld    $2, 88($fp)
    st    $2, 56($fp)

```

```

ld      $2, 92($fp)
st      $2, 52($fp)
ld      $2, 96($fp)
st      $2, 48($fp)
ld      $2, 100($fp)
st      $2, 44($fp)
ld      $2, 64($fp)      // int *b = (int*)alloca(sizeof(int) * x1);
shl     $2, $2, 2
addiu   $2, $2, 7
addiu   $3, $zero, -8
and     $2, $2, $3
subu   $2, $sp, $2
add     $sp, $zero, $2 // set sp to the bottom of alloca area
st      $2, 40($fp)
addiu   $3, $zero, 1111
st      $3, 0($2)
ld      $2, 64($fp)
ld      $3, 60($fp)
ld      $4, 56($fp)
ld      $5, 52($fp)
ld      $t9, 48($fp)
ld      $t0, 44($fp)
st      $t0, 20($sp)
shl     $t9, $t9, 1
st      $t9, 16($sp)
st      $5, 12($sp)
st      $4, 8($sp)
st      $3, 4($sp)
addiu   $3, $zero, 6
mul    $2, $2, $3
st      $2, 0($sp)
ld      $t9, %call16(_Z3sumiiiiii) ($gp)
jalr   $t9
ld      $gp, 24($fp)
st      $2, 36($fp)
ld      $3, 0($7)
ld      $4, 68($fp)
bne    $3, $4, $BB1_2
nop
# BB#1:                                # %SP_return
move   $fp, $sp
ld      $7, 72($sp)          # 4-byte Folded Reload
ld      $lr, 76($sp)          # 4-byte Folded Reload
addiu   $sp, $sp, 80
ret    $2
nop
$BB1_2:                                # %CallStackCheckFailBlk
ld      $t9, %call16(__stack_chk_fail) ($gp)
jalr   $t9
nop
ld      $gp, 24($fp)
.set   macro
.set   reorder
.end   _Z10weight_sumiiiiii
$tmp10:
.size  _Z10weight_sumiiiiii, ($tmp10)-_Z10weight_sumiiiiii
.cfi_endproc
...

```

As you can see, the dynamic stack allocation needs frame pointer register **fp** support. As Figure 9.8, the sp is adjusted to (sp - 56) when it entered the function as usual by instruction **addiu \$sp, \$sp, -56**. Next, the fp is set to sp where is the position just above alloca() spaces area when meet instruction **addu \$fp, \$sp, \$zero**. After that, the sp is changed to the area just below of alloca(). Remind, the alloca() area which the b point to, “*****b = (int*)alloca(sizeof(int) * x1)***”, is allocated at run time since the spaces is variable size which depend on x1 variable and cannot be calculated at link time.

Figure 9.9 depicted 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 **addu \$sp, \$fp, \$zero**. Next, sp is changed back to caller stack bottom by instruction **addiu \$sp, \$sp, 56**.

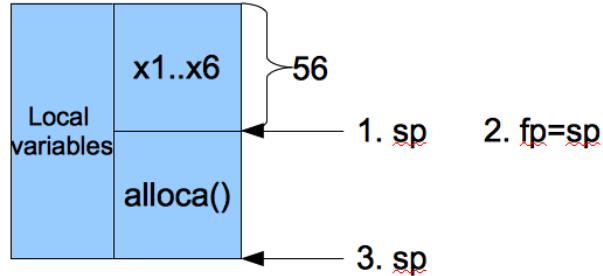


Figure 9.8: Frame pointer changes when enter function

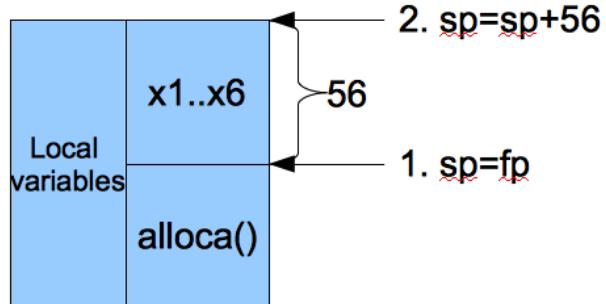


Figure 9.9: Stack pointer changes when exit function

Use fp to keep the old stack pointer value is not the only solution. Actually, we can keep the alloca() spaces size on a specific memory address and the sp can back to the the old sp by add the alloca() spaces size. Most ABI like Mips and ARM access the above area of alloca() by fp and the below area of alloca() by sp, as Figure 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 compare to use the sp only.

```
ld      $2, 64($fp)
st      $3, 4($sp)
```

Cpu0 use fp and sp to access the above and below areas of alloca() too. As ch9_4.cpu0.s, it access local variable (above of alloca()) by fp offset and outgoing arguments (below of alloca()) by sp offset.

Finally, the “move \$sp, \$fp” is the alias instruction of “addu \$fp, \$sp, \$zero”. The machine code is the latter, and the former is only for easy understand by user only. This alias come from added code in Chapter3_2 and Chapter3_4 as follows,

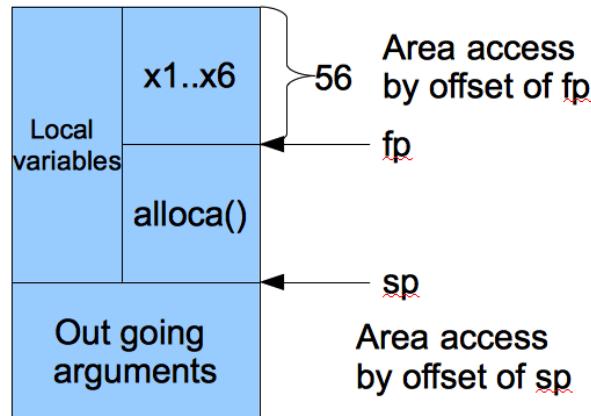


Figure 9.10: fp and sp access areas

lbdex/chapters/Chapter3_2/InstPrinter/Cpu0InstPrinter.cpp

```
void Cpu0InstPrinter::printInst(const MCInst *MI, raw_ostream &O,
                               StringRef Annot, const MCSubtargetInfo &STI) {
    // Try to print any aliases first.
    if (!printAliasInstr(MI, O))
```

lbdex/chapters/Chapter3_5/Cpu0InstrInfo.td

```
class Cpu0InstAlias<string Asm, dag Result, bit Emit = 0b1> :
    InstAlias<Asm, Result, Emit>;

let Predicates = [Ch3_4] in {
//=====
// Instruction aliases
//=====
def : Cpu0InstAlias<"move $dst, $src",
                  (ADDu GPROut:$dst, GPROut:$src, ZERO), 1>;
}
```

File ch9_7.cpp which is for type “long long shift operations” support can be tested now as follows.

lbdex/input/ch9_7.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
```

```

    return (c+d); // 0x121 = 289
}

long long test_longlong_shift2()
{
    long long a = 48;
    long long b = 0x001666660000000a;
    long long c;

    c = (b >> a);

    return c; // 22
}

114-37-150-209:input Jonathan$ clang -O0 -target mips-unknown-linux-gnu
-c ch9_7.cpp -emit-llvm -o ch9_7.bc

114-37-150-209:input Jonathan$ ~/llvm/test/cmake_debug_build/Debug/bin/
llvm-dis ch8_4.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 = ash r 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/cmake_debug_build/Debug/bin/llc
-march=cpu0 -mcpu=cpu032I -relocation-model=static -filetype=asm ch9_7.bc -o -
.text
.section .mdebug.abi32
.previous
.file "ch9_7.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:

```

```

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
.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
```

9.7 Summary

Until now, we have 6,000 lines of source code around in the end of this chapter. The cpu0 backend code now can take care the integer function call and control statement just like the llvm front end tutorial example code. Look back the chapter of “Back end structure”, there are 3,100 lines of source code with taking three instructions only. With this 95% more of code, it can translate tens of instructions, global variable, control flow statement and function call. Now the cpu0 backend is not just a toy. It can translate some of the C++ OOP language into Cpu0 instructions without much effort in backend. Because the most complex things in language, such as C++ syntax, is handled by front end. LLVM is a real structure following the compiler theory, any backend of LLVM can benefit from this structure. The best part of 3 tiers compiler structure is the backend will grow up automatically through the front end support languages more and more if the front end has not add any new IR for a new language.

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. This chapter will use the binutils which has been installed in "sub-section Install other tools on iMac" of Appendix A: "Installing LLVM" ² to check the generated cpu0 ELF file. You will learn the objdump, readelf, ..., tools and understand the ELF file format itself through using these tools to analyze the cpu0 generated obj in this chapter. LLVM has the llvm-objdump tool which like objdump. We will make cpu0 support llvm-objdump tool further in this chapter. The binutils is a cross compiler tool chains include a couple of CPU ELF dump function support. Linux platform has binutils already and no need to install it further. The reason we use Linux binutils in this chapter just because my iMac will display Chinese text. The iMac corresponding binutils have no problem except it add g in command name and and display with your area language instead of pure English on iMac. For example, to use gobjdump instead of objdump and I have the result of chinese language unicode display instead of pure English on my iMac.

The binutils tool we use is not a part of llvm tools, but it's a powerful tool in ELF analysis. This chapter introduce 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 analyze the ELF since the obj is need to be handled by linker or loader later. 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" ³ written by Beck is a famous book in concept of telling readers what is the compiler output, what is the linker output, what is the loader output, and how they work together. But it covers the concept only. You can reference it to understand how the "**Relocation Record**" works if you need to refresh or learning this knowledge for this chapter.

¹ http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

² <http://jonathan2251.github.io/lbd/install.html#install-other-tools-on-imac>

³ Leland Beck, System Software: An Introduction to Systems Programming.

⁴, ⁵, ⁶ are the Chinese documents available from the cpu0 author on web site.

10.1 ELF format

ELF is a format used both in obj and executable file. So, there are two views in it as Figure 10.1.

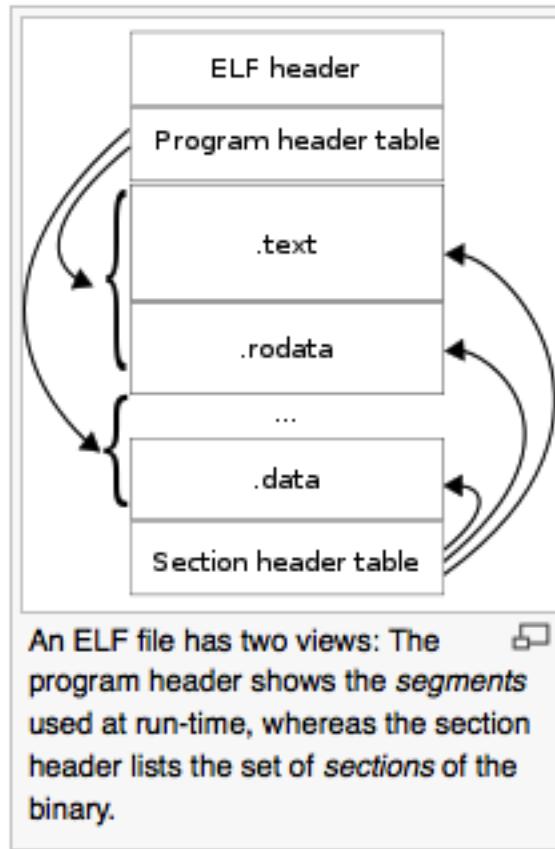


Figure 10.1: ELF file format overview

As Figure 10.1, the “Section header table” include sections .text, .rodata, ..., .data which are sections layout for code, read only data, ..., and read/write data. “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 `readelf -h` to see what information the ELF header contains.

```
[Gamma@localhost input]$ ~/llvm/test/cmake_debug_build/bin/llc -march=cpu0  
-relocation-model=pic -filetype=obj ch6_1.bc -o ch6_1.cpu0.o
```

⁴ <http://ccckmit.wikidot.com/lk:aout>

⁵ <http://ccckmit.wikidot.com/lk:objfile>

⁶ <http://ccckmit.wikidot.com/lk:elf>

```
[Gamma@localhost input]$ 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)
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
[Gamma@localhost input]$
```

```
[Gamma@localhost input]$ ~/llvm/test/cmake_debug_build/
bin/llc -march=mips -relocation-model=pic -filetype=obj ch6_1.bc -o ch6_1.mips.o
```

```
[Gamma@localhost input]$ 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
[Gamma@localhost input]$
```

As above ELF header display, it contains information of magic number, version, ABI, The Machine field of cpu0 is unknown while mips is MIPS R3000. It is because cpu0 is not a popular CPU recognized by utility readelf. Let's check ELF segments information as follows,

```
[Gamma@localhost input]$ readelf -l ch6_1.cpu0.o
```

```
There are no program headers in this file.
[Gamma@localhost 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.

```
[Gamma@localhost input]$ readelf -S ch6_1.cpu0.o
There are 10 section headers, starting at offset 0xd4:

Section Headers:
[Nr] Name           Type      Addr     Off      Size    ES Flg Lk Inf Al
[ 0] .null         NULL      00000000 000000 000000 00 0   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)
```

10.1.2 Relocation Record

Cpu0 backend translate global variable as follows,

```
[Gamma@localhost input]$ clang -target mips-unknown-linux-gnu -c ch6_1.cpp
-emit-llvm -o ch6_1.bc
[Gamma@localhost input]$ ~/llvm/test/cmake_debug_build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch6_1.bc -o ch6_1.cpu0.s
[Gamma@localhost 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
.cupload $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

[Gamma@localhost input]$ ~/llvm/test/cmake_debug_build/
```

```
bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch6_1.bc -o ch6_1.cpu0.o
[Gamma@localhost input]$ 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 .*...".-.....
...
[Gamma@localhost input]$ Jonathan$
```

```
[Gamma@localhost input]$ 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	

```
[Gamma@localhost input]$ readelf -tr ch6_1.mips.o
There are 9 section headers, starting at offset 0xc8:
```

Section Headers:

[Nr]	Name

Type Flags	Addr	Off	Size	ES	Lk	Inf	Al
[0] NULL [00000000]:		00000000 000000 000000 00			0	0	0
[1] .text PROGBITS [00000006]: ALLOC, EXEC		00000000 000034 000038 00			0	0	4
[2] .rel.text REL [00000000]:		00000000 0002f8 000018 08			7	1	4
[3] .data PROGBITS [00000003]: WRITE, ALLOC		00000000 00006c 000008 00			0	0	4
[4] .bss NOBITS [00000003]: WRITE, ALLOC		00000000 000074 000000 00			0	0	4
[5] .reginfo MIPS_REGINFO [00000002]: ALLOC		00000000 000074 000018 00			0	0	1
[6] .shstrtab STRTAB [00000000]:		00000000 00008c 000039 00			0	0	1
[7] .symtab SYMTAB [00000000]:		00000000 000230 0000a0 10			8	6	4
[8] .strtab STRTAB [00000000]:		00000000 0002d0 000025 00			0	0	1

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 “**.cupload %reg**” into the following.

```
// Lower ".cupload $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 referred 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 translate the %hi(_gp_disp) and %lo(_gp_disp) into 0 since when loader load this obj into memory, loader will know the _gp_disp value at run time and will update these two offset relocation records into 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 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, 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

In iMac, gobjdump -tr can display the information of relocation records like readelf -tr. LLVM tool llvm-objdump is the same tool as objdump. Let's run gobjdump and llvm-objdump 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/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch9_3.bc -o
ch9_3.cpu0.o

118-165-78-12:input Jonathan$ gobjdump -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/cmake_debug_build/
Debug/bin/llvm-objdump -t -r ch9_3.cpu0.o

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 llvmdump 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) {
        ...
        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 above EM_CPU0 defined.

10.2.2 LLVM-OBJDUMP -d

Run the last Chapter example code with command `llvm-objdump -d` for dump 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/cmake_debug_

```

```
build/Debug/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/cmake_debug_  
build/Debug/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).

Ibdex/chapters/Chapter10_1/CMakeLists.txt

```
tablegen(LLVM Cpu0GenDisassemblerTables.inc -gen-disassembler)  
add_subdirectory(Disassembler)
```

Ibdex/chapters/Chapter10_1/LLVMBuild.txt

```
subdirectories =  
    Disassembler  
has_disassembler = 1
```

Ibdex/chapters/Chapter10_1/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;  
        //#if CH >= CH10_1  
        let DecoderMethod = "DecodeCMPInstruction";  
        //#endif  
        let Predicates = [HasCmp];  
    }  
}  
  
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  
    let DecoderMethod = "DecodeJumpAbsoluteTarget";  
    //#endif  
}
```

Ibdex/chapters/Chapter10_1/Disassembler/CMakeLists.txt

```
add_llvm_library(LLVMCpu0Disassembler
    Cpu0Disassembler.cpp
)
```

Ibdex/chapters/Chapter10_1/Disassembler/LLVMBuild.txt

```
;===== ./lib/Target/Cpu0/Disassembler/LLVMBuild.txt -----*- Conf -*---;;
;
;           The LLVM Compiler Infrastructure
;
; This file is distributed under the University of Illinois Open Source
; License. See LICENSE.TXT for details.
;
;=====-----;
;
; This is an LLVMBuild description file for the components in this subdirectory.
;
; For more information on the LLVMBuild system, please see:
;
;   http://llvm.org/docs/LLVMBuild.html
;
;=====-----;
```

[component_0]
type = Library
name = Cpu0Disassembler
parent = Cpu0
required_libraries = MCDisassembler Support Cpu0Info
add_to_library_groups = Cpu0

Ibdex/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.  
//  
//=====-----//  
  
#include "Cpu0.h"  
  
#include "Cpu0RegisterInfo.h"  
#include "Cpu0Subtarget.h"  
#include "llvm/MC/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/MemoryObject.h"
#include "llvm/Support/TargetRegistry.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0-disassembler"

typedef MCDisassembler::DecodeStatus DecodeStatus;

namespace {

/// Cpu0DisassemblerBase - a disassembler 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 disassembler 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 &VStream,
                               raw_ostream &CStream) const override;
};

} // 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 DecodeC0RegsRegisterClass (MCInst &Inst,
                                              unsigned RegNo,
                                              uint64_t Address,
                                              const void *Decoder);
static DecodeStatus DecodeCMPInstruction (MCInst &Inst,
                                         unsigned Insn,
                                         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 DecodeJumpRelativeTarget (MCInst &Inst,
                                             unsigned Insn,
                                             uint64_t Address,
                                             const void *Decoder);
static DecodeStatus DecodeJumpAbsoluteTarget (MCInst &Inst,
                                             unsigned Insn,
                                             uint64_t Address,
                                             const void *Decoder);

static DecodeStatus DecodeMem (MCInst &Inst,
                             unsigned Insn,
                             uint64_t Address,
                             const void *Decoder);
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);
    }
    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 &VStream,
                                 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);
}

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;
}

/* CMP instruction define $rc and then $ra, $rb; The printOperand() print

```

operand 1 and operand 2 (operand 0 is \$rc and operand 1 is \$ra), so we Create register \$rc first and create \$ra next, as follows,

```
// Cpu0InstrInfo.td
class CmpInstr<bits<8> op, string instr_asm,
               InstrItinClass itin, RegisterClass RC, RegisterClass RD, bit isComm = 0>:
    FA<op, (outs RD:$rc), (ins RC:$ra, RC:$rb),
      !strconcat(instr_asm, "\t$ra, $rb"), [], itin> {

// 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;
...
case 1:
    // CMP
    printOperand(MI, 2, O);
    return;
break;
*/
static DecodeStatus DecodeCMPInstruction(MCInst &Inst,
                                         unsigned Insn,
                                         uint64_t Address,
                                         const void *Decoder) {
    int Reg_a = (int)fieldFromInstruction(Insn, 20, 4);
    int Reg_b = (int)fieldFromInstruction(Insn, 16, 4);
    int Reg_c = (int)fieldFromInstruction(Insn, 12, 4);

    Inst.addOperand(MCOperand::createReg(CPURegsTable[Reg_a]));
    Inst.addOperand(MCOperand::createReg(CPURegsTable[Reg_b]));
    Inst.addOperand(MCOperand::createReg(CPURegsTable[Reg_c]));
    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,

```

```
// 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 DecodeJumpRelativeTarget (MCInst &Inst,
                                                unsigned Insn,
                                                uint64_t Address,
                                                const void *Decoder) {

        int JumpOffset = fieldFromInstruction(Insn, 0, 24);
        if (JumpOffset > 0x8fffff)
            JumpOffset = -1*(0x1000000 - JumpOffset);
        Inst.addOperand(MCOperand::createImm(JumpOffset));
        return MCDisassembler::Success;
    }

    static DecodeStatus DecodeJumpAbsoluteTarget (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 DecodeSimm16 (MCInst &Inst,
                                     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 of obj to assembly. So, add Disassembler/Cpu0Disassembler.cpp and modify the CMakeList.txt and LLVMBuild.txt to build with directory Disassembler and enable the disassembler table generated by “has_disassembler = 1”. Most of code is handled by the table of *.td files defined. 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. In Cpu0 example, we define function DecodeCMPIInstruction(), DecodeBranch24Target() and DecodeJumpAbsoluteTarget() in Cpu0Disassembler.cpp and

tell the LLVM table driven system by write “**let DecoderMethod = ...**” in the corresponding instruction definitions or ISD node of Cpu0InstrInfo.td. LLVM will call these DecodeMethod when user use Disassembler job in tools, such as `llvm-objdump -d`. You can check the comments above these DecodeMethod functions to see how it works. For the CMP instruction, according the definiton of `CmpInstr<...>` in Cpu0InstrInfo.td, the assembler will print as `$sw, $ra, $rb.` (`$sw` is a fixed name operand and won’t exists in instruction), and encode as “10230000” in elf binary. Since `$sw` is a fixed operand, we choose assigning `$rc` to 0. You can define the `CmpInstr` as follows,

```
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;
}
```

Above definition will encode this CMP instruction “`cmp $sw, $2, $3`” into “10f23000”.

`RetLR` (Cpu0ISD::Ret) and `JR` (ISD::BRIND) are both for “ret” instruction. The former is for instruction encode in assembly and obj while the latter is for decode in disassembler. IR node `Cpu0ISD::Ret` is created in `LowerReturn()` which called at function exit point. Details is explained in Chapter 3 here ⁷.

Finally `cpu032II` include all `cpu032I` instruction set and adds some instructions. 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 -mcpu=cpu032I`, so the variable `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).

Index/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 StringRef 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/cmake_debug_
build/Debug/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/cmake_debug_
build/Debug/bin/llvm-objdump -d ch8_1_1.cpu0.o
```

⁷ <http://jonathan2251.github.io/lbd/backendstructure.html#handle-return-register-lr>

```
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)
...
...
```


ASSEMBLER

- AsmParser support
- Inline assembly

This chapter will add LLVM AsmParser support first and introduce inline assembly handler next. With AsmParser and inline assembly support, 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.

Ibdex/input/ch11_1.cpp

```
asm("ld      $2, 8($sp)");
asm("st      $0, 4($sp)");
asm("addiu $3,      $ZERO, 0");
asm("add $3, $1, $2");
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");
```

¹ <http://www.embecosm.com/appnotes/ean10/ean10-howto-llvmas-1.0.html>

```

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("ret $lr");
asm("jalr $t9");
asm("li $3, 0x00700000");
asm("la $3, 0x00800000($6)");
asm("la $3, 0x00900000");

```

```

Jonathan@tekiiMac:~/input Jonathan$ clang -c ch11_1.cpp -emit-llvm -o
ch11_1.bc
Jonathan@tekiiMac:~/input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch11_1.bc
-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 don't implement 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. Directory AsmParser handle the assembly to obj translation. The Chapter11_1/include AsmParser implementation as follows,

[Index/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/Cpu0MCTargetDesc.h"
#include "Cpu0RegisterInfo.h"
#include "llvm/ADT/APInt.h"
#include "llvm/ADT/StringSwitch.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCEExpr.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/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/MCTargetAsmParser.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;}
    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 {
    MCSubtargetInfo &STI;
    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;

    bool ParseInstruction(ParseInstructionInfo &Info, StringRef Name,
                          SMLoc NameLoc, OperandVector &Operands) override;

    bool parseMathOperation(StringRef Name, SMLoc NameLoc,
                           OperandVector &Operands);

    bool ParseDirective(AsmToken DirectiveID) override;

    Cpu0AsmParser::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);
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 MCEexpr *&Res);
bool parseRelocOperand(const MCEexpr *&Res);

bool parseDirectiveSet();

bool parseSetAtDirective();
bool parseSetNoAtDirective();
bool parseSetMacroDirective();
bool parseSetNoMacroDirective();
bool parseSetReorderDirective();
bool parseSetNoReorderDirective();

MCSymbolRefExpr::VariantKind getVariantKind(StringRef Symbol);

int matchRegisterName(StringRef Symbol);

int matchRegisterByNumber(unsigned RegNum, StringRef Mnemonic);

unsigned getReg(int RC, int RegNo);

public:
    Cpu0AsmParser(MCSubtargetInfo &sti, MCAsmParser &parser,
                  const MCInstrInfo &MII, const MCTargetOptions &Options)
    : MCTargetAsmParser(), STI(sti), Parser(parser) {
        // Initialize the set of available features.
        setAvailableFeatures(ComputeAvailableFeatures(STI.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_CondCode,
    k_CoprocNum,
    k_Immediate,
    k_Memory,
    k_PostIndexRegister,
    k_Register,
    k_Token
} Kind;

public:
Cpu0Operand(KindTy K) : MCParsedAsmOperand(), Kind(K) {}

struct Token {
    const char *Data;
    unsigned Length;
};

struct PhysRegOp {
    unsigned RegNum; // Register Number
};

struct ImmOp {
    const MCEExpr *Val;
};

struct MemOp {
    unsigned Base;
    const MCEExpr *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(MCOOperand::createReg(getReg()));
}

void addExpr(MCInst &Inst, const MCEExpr *Expr) const {
    // Add as immediate when possible. Null MCEExpr = 0.
    if (Expr == 0)
        Inst.addOperand(MCOOperand::createImm(0));
    else if (const MCConstantExpr *CE = dyn_cast<MCConstantExpr>(Expr))
        Inst.addOperand(MCOOperand::createImm(CE->getValue()));
    else
        Inst.addOperand(MCOOperand::createExpr(Expr));
}

void addImmOperands(MCInst &Inst, unsigned N) const {
    assert(N == 1 && "Invalid number of operands!");
    const MCEExpr *Expr = getImm();
    addExpr(Inst, Expr);
}

```

```

void addMemOperands(MCInst &Inst, unsigned N) const {
    assert(N == 2 && "Invalid number of operands!");

    Inst.addOperand(MCOperand::createReg(getMemBase()));

    const MCExpr *Expr = getMemOff();
    addExpr(Inst, Expr);
}

bool isReg() const { return Kind == k_Register; }
bool isImm() const { return Kind == k_Immediate; }
bool isToken() const { return Kind == k_Token; }
bool isMem() const { return Kind == k_Memory; }

StringRef getToken() const {
    assert(Kind == k_Token && "Invalid access!");
    return StringRef(Tok.Data, Tok.Length);
}

unsigned getReg() const {
    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 = 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 = make_unique<Cpu0Operand>(k_Register);
    Op->Reg.RegNum = RegNum;
    Op->StartLoc = S;
    Op->EndLoc = E;
    return Op;
}

```

```

static std::unique_ptr<Cpu0Operand> CreateImm(const MCExpr *Val, SMLoc S, SMLoc E) {
    auto Op = 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 = 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 { return StartLoc; }

/// getEndLoc - Get the location of the last token of this operand.
SMLoc getEndLoc() const { return EndLoc; }

virtual void print(raw_ostream &OS) const {
    llvm_unreachable("unimplemented!");
}
};

//@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,$zero,j
    tmpInst.setOpcode(Cpu0::ORI);
    tmpInst.addOperand(MCOperand::createReg(RegOp.getReg()));
    tmpInst.addOperand(
        MCOperand::createReg(Cpu0::ZERO));
    tmpInst.addOperand(MCOperand::createImm(ImmValue));
    Instructions.push_back(tmpInst);
} else if ( ImmValue < 0 && ImmValue >= -32768) {
    // for -32768 <= j < 0.
    // li d,j => addiu d,$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
    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);
}
}

```

```

        }

    }

//@2 {
bool Cpu0AsmParser::MatchAndEmitInstruction(SMLoc IDLoc, unsigned &Opcode,
                                             OperandVector &Operands,
                                             MCStreamer &Out,
                                             uint64_t &ErrorInfo,
                                             bool MatchingInlineAsm) {
    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], STI);
            }
        } else {
            Inst.setLoc(IDLoc);
            Out.EmitInstruction(Inst, STI);
        }
        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)
        .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;
}

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) {
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;

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;

            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);

Operands.push_back(Cpu0Operand::CreateImm(IdVal, S, E));
return false;
} // case AsmToken::Percent
} // switch(getLexer().getKind())
return true;
}

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.Identifier().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) {
    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

// Check the type of the expression
if (const MCConstantExpr *MCE = dyn_cast<MCConstantExpr>(IdVal)) {
    // it's a constant, evaluate lo or hi value
    int Val = MCE->getValue();
    if (Str == "lo") {
        Val = Val & 0xffff;
    } else if (Str == "hi") {
        Val = (Val & 0xffff0000) >> 16;
    }
    Res = MCConstantExpr::create(Val, getContext());
    return false;
}

if (const MCSymbolRefExpr *MSRE = dyn_cast<MCSymbolRefExpr>(IdVal)) {
    // it's a symbol, create symbolic expression from symbol
    StringRef Symbol = MSRE->getSymbol().getName();
    MCSymbolRefExpr::VariantKind VK = getVariantKind(Str);
    Res = MCSymbolRefExpr::create(Symbol,VK,getContext());
    return false;
}
return true;
}

bool Cpu0AsmParser::ParseRegister(unsigned &RegNo, SMLoc &StartLoc,
                                    SMLoc &EndLoc) {

    StartLoc = Parser.getTok().getLoc();
    RegNo = tryParseRegister("");
    EndLoc = Parser.getTok().getLoc();
    return (RegNo == (unsigned) -1);
}

bool Cpu0AsmParser::parseMemOffset(const MCE Expr *&Res) {
    SMLoc S;

```

```

switch(getLexer().getKind()) {
default:
    return true;
case AsmToken::Integer:
case AsmToken::Minus:
case AsmToken::Plus:
    return (getParser().parseExpression(Res));
case AsmToken::Percent:
    return parseRelocOperand(Res);
case AsmToken::LParen:
    return false; // it's probably assuming 0
}
return true;
}

// eg, 12($sp) or 12(la)
Cpu0AsmParser::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;
}

```

```

}

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;
}

//@getVariantKind {
MCSymbolRefExpr::VariantKind Cpu0AsmParser::getVariantKind(StringRef Symbol) {
//@getVariantKind body {
    MCSymbolRefExpr::VariantKind VK
        = StringSwitch<MCSymbolRefExpr::VariantKind>(Symbol)
            .Case("hi", MCSymbolRefExpr::VK_Cpu0_ABS_HI)
            .Case("lo", MCSymbolRefExpr::VK_Cpu0_ABS_LO)
            .Case("gp_rel", MCSymbolRefExpr::VK_Cpu0_GPREL)
            .Case("call16", MCSymbolRefExpr::VK_Cpu0_GOT_CALL)
            .Case("got", MCSymbolRefExpr::VK_Cpu0_GOT)
#if CH >= CH12_1
            .Case("tlsd", MCSymbolRefExpr::VK_Cpu0_TLSGD)
            .Case("tlsldm", MCSymbolRefExpr::VK_Cpu0_TLSLDM)
            .Case("dtp_hi", MCSymbolRefExpr::VK_Cpu0_DTP_HI)
            .Case("dtp_lo", MCSymbolRefExpr::VK_Cpu0_DTP_LO)
            .Case("gottp", MCSymbolRefExpr::VK_Cpu0_GOTTPREL)
            .Case("tp_hi", MCSymbolRefExpr::VK_Cpu0_TP_HI)
            .Case("tp_lo", MCSymbolRefExpr::VK_Cpu0_TP_LO)
#endif
            .Case("got_disp", MCSymbolRefExpr::VK_Cpu0_GOT_DISP)
            .Case("got_page", MCSymbolRefExpr::VK_Cpu0_GOT_PAGE)
            .Case("got_ofst", MCSymbolRefExpr::VK_Cpu0_GOT_OFST)
            .Case("hi(%neg(%gp_rel)", MCSymbolRefExpr::VK_Cpu0_GPOFF_HI)
            .Case("lo(%neg(%gp_rel", MCSymbolRefExpr::VK_Cpu0_GPOFF_LO)
            .Default(MCSymbolRefExpr::VK_None);

return VK;
}

bool Cpu0AsmParser::
parseMathOperation(StringRef Name, SMLoc NameLoc,
                  OperandVector &Operands) {
    // split the format
    size_t Start = Name.find('.'), Next = Name.rfind('.');
    StringRef Format1 = Name.slice(Start, Next);
    // and add the first format to the operands
    Operands.push_back(Cpu0Operand::CreateToken(Format1, NameLoc));
    // now for the second format
}

```

```

StringRef Format2 = Name.slice(Next, StringRef::npos);
Operands.push_back(Cpu0Operand::CreateToken(Format2, NameLoc));

// set the format for the first register
// setFpFormat(Format1);

// 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");
    }

    if (getLexer().isNot(AsmToken::Comma)) {
        SMLoc Loc = getLexer().getLoc();
        Parser.eatToEndOfStatement();
        return Error(Loc, "unexpected token in argument list");
    }
    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");
    }
}

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::
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);

    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) ) {
    ParserLex(); // 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");
    }
}

if (getLexer().isNot(AsmToken::EndOfStatement)) {
    SMLoc Loc = getLexer().getLoc();
    Parser.eatToEndOfStatement();
    return Error(Loc, "unexpected token in argument list");
}

ParserLex(); // Consume the EndOfStatement
return false;
}

bool Cpu0AsmParser::reportParseError(StringRef ErrorMsg) {
    SMLoc Loc = getLexer().getLoc();
    Parser.eatToEndOfStatement();
    return Error(Loc, ErrorMsg);
}

bool Cpu0AsmParser::parseSetReorderDirective() {
    ParserLex();
    // 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();
    ParserLex(); // Consume the EndOfStatement
    return false;
}

bool Cpu0AsmParser::parseSetNoReorderDirective() {
    ParserLex();
    // 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();
    ParserLex(); // Consume the EndOfStatement
    return false;
}

bool Cpu0AsmParser::parseSetMacroDirective() {
    ParserLex();
    // 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;
}

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;
    }
}

```

```
}

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
```

Index/chapters/Chapter11_1/AsmParser/CMakeLists.txt

```
add_llvm_library(LLVMCpu0AsmParser
    Cpu0AsmParser.cpp
)
```

Index/chapters/Chapter11_1/AsmParser/LLVMBuild.txt

```
;===== ./lib/Target/Cpu0/AsmParser/LLVMBuild.txt -----*-- Conf -*---=;
;
;                               The LLVM Compiler Infrastructure
;
; This file is distributed under the University of Illinois Open Source
; License. See LICENSE.TXT for details.
;
```

```

;=====;
;
; This is an LLVMBuild description file for the components in this subdirectory.
;
; For more information on the LLVMBuild system, please see:
;
;   http://llvm.org/docs/LLVMBuild.html
;
;=====;

[component_0]
type = Library
name = Cpu0AsmParser
parent = Cpu0
required_libraries = Cpu0Desc Cpu0Info MC MCParser Support
add_to_library_groups = Cpu0

```

The Cpu0AsmParser.cpp contains one thousand lines of code which do the assembly language parsing. You can understand it with a little patient only. To let file directory of AsmParser be built, modify CMakeLists.txt and LLVMBuild.txt as follows,

Index/chapters/Chapter11_1/CMakeLists.txt

```

set(LLVM_TARGET_DEFINITIONS Cpu0Asm.td)
tablegen(LLVM Cpu0GenAsmMatcher.inc -gen-asm-matcher)

```

Index/chapters/Chapter11_1/LLVMBuild.txt

```

subdirectories =
  AsmParser
has_asmparser = 1

```

Index/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"
```

Ibdex/chapters/Chapter11_1/Cpu0RegisterInfoGPROutForAsm.td

```
//=====//  
// Register Classes  
//=====//  
  
def GPROut : RegisterClass<"Cpu0", [i32], 32, (add CPURegs)>;
```

The CMakeLists.txt add 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 but SW. Cpu0Other.td is used when translating llvm IR to Cpu0 instruction. In this 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 compile with C statement “a = (b & c);” and generate “and \$sw, \$1, \$2” instruction, then the \$sw of interrupt status will be destroyed. When do assembling, instruction “andi \$sw, \$sw, 0xffff” is allowed. This assembly program is accepted and Cpu0 backend treats it is safe. For instance, assembler programmer can disable trace debug message by “andi \$sw, \$sw, 0xffff” and enable debug message by “ori \$sw, \$sw, 0x0020” as the dynamic linker example code using them in later chapter. Beside this, 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(“mfcc \$pc, \$epc”);”.

Ibdex/chapters/Chapter2/Cpu0RegisterInfo.td

```
def CPURegs [RegisterClass<"Cpu0", [i32], 32, (add) ... , PC, EPC>;
```

Ibdex/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];
```

}

Ibdex/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 = [];
    }
```

Ibdex/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 declare the **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.

[cmake_debug_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,

[Index/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) {
    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], STI);
                }
            } else {
                Inst.setLoc(IDLoc);
                Out.EmitInstruction(Inst, STI);
            }
            return false;
        }
    ...
}

```

Finally, remind the CPURegs as below must follow the order of register number because AsmParser uses them when do register number encode.

[Index/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]>;

#ifndef CH >= CH4_1
// Hi/Lo registers number and name
def HI     : Register<"hi">, DwarfRegNum<[18]>;
def LO     : Register<"lo">, 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
GP, FP,
SP, LR, SW, PC, EPC)>;

```

Run Chapter11_1/ with ch11_1.cpp to get the correct result as follows,

```

Jonathan@Mac:~/input$ Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch11_1.bc -o
ch11_1.cpu0.o
Jonathan@Mac:~/input$ Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/Debug/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 code in AsmParser and Dissasembly (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,

src/lib/MC/MCParser/AsmParser.cpp

```
class AsmParser : public MCAsmParser {
    ...
}
```

AsmParser will call functions ParseInstruction() and MatchAndEmitInstruction() of Cpu0AsmParser as follows,

src/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;
        ...
    }
    ...
    bool HadError = getTargetParser().ParseInstruction(IInfo, OpcodeStr, IDLoc,
                                                       Info.ParsedOperands);
    ...
    // If parsing succeeded, match the instruction.
    if (!HadError) {
        unsigned ErrorInfo;
        getTargetParser().MatchAndEmitInstruction(IDLoc, Info.Opcde,
                                                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(), matchRegisterByName()
- 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;
}

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"
        : "=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;
}

#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/cmake_debug_build/Debug/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 can be allocated by compiler to register \$2, \$3 or other. The inline assembly fills the gap between high level language and assembly language. Reference here ². Chapter11_2 support inline assembly as follows,

² <http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>

Ibdex/chapters/Chapter11_2/Cpu0AsmPrinter.h

```

bool PrintAsmOperand(const MachineInstr *MI, unsigned OpNo,
                     unsigned AsmVariant, const char *ExtraCode,
                     raw_ostream &O) override;
bool PrintAsmMemoryOperand(const MachineInstr *MI, unsigned OpNum,
                           unsigned AsmVariant, const char *ExtraCode,
                           raw_ostream &O) override;
void printOperand(const MachineInstr *MI, int opNum, raw_ostream &O);

```

Ibdex/chapters/Chapter11_2/Cpu0AsmPrinter.cpp

```

// Print out an operand for an inline asm expression.
bool Cpu0AsmPrinter::PrintAsmOperand(const MachineInstr *MI, unsigned OpNum,
                                     unsigned AsmVariant, 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, AsmVariant, 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";
                return false;
        }
    }
}

```

```

        }

    }

printOperand(MI, OpNum, O);
return false;
}

bool Cpu0AsmPrinter::PrintAsmMemoryOperand(const MachineInstr *MI,
                                            unsigned OpNum, unsigned AsmVariant,
                                            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_GOT16: O << "%got16("; 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();
            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 << ")";
}

```

Ibdex/chapters/Chapter11_2/Cpu0InstrInfo.h

```

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

```

Ibdex/chapters/Chapter11_2/Cpu0InstrInfo.cpp

```

/// Return the number of bytes of code the specified instruction may be.
unsigned Cpu0InstrInfo::GetInstSizeInBytes(const MachineInstr *MI) const {
    switch (MI->getOpcode()) {
    default:
        return MI->getDesc().getSize();
    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());
    }
}

```

Ibdex/chapters/Chapter11_2/Cpu0ISelDAGToDAG.h

```
bool SelectInlineAsmMemoryOperand(const SDValue &Op,
                                  unsigned ConstraintID,
                                  std::vector<SDValue> &OutOps) override;
```

Ibdex/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
```

Ibdex/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-asn 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,
                                    SelectionDAG &DAG) const override;

bool isLegalAddressingMode(const DataLayout &DL, const AddrMode &AM,
                           Type *Ty, unsigned AS) const override;
```

Ibdex/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);
            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, std::ptr_fun(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;

    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("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.
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;
    }
}
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) 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;
    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 others' function are same with backend. The Cpu0ISelLowering.cpp inline asm is explained after the result of run with ch11_2.cpp. Cpu0ISelDAGToDAG.cpp just save OP code in SelectInlineAssemblyMemoryOperand(). Since the the OP code is Cpu0 inline assembly instruction, no llvm IR DAG translation needed further. Save OP directly and return false to notiy 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/cmake_debug_build/Debug/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
    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
%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
}

; 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/cmake_debug_build/Debug/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
_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)
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
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)
#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
.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
addu $3, $4, $4
ld $4, 24($fp)
addu $3, $3, $4
ld $4, 20($fp)
addu $3, $3, $4
ld $4, 16($fp)
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

```

The clang translate gcc style inline assembly `__asm__` into llvm IR Inline Assembler Expressions first³, 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)
                     );

```

³ <http://llvm.org/docs/LangRef.html#inline-assembler-expressions>

```
%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)
                     );
%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 positive or negative const operand value range to 16 bits since FL type immediate operand is 16 bits in Cpu0 instruction. The range of N is -65535 to -1 and 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 16 bits limitation.

C++ SUPPORT

- Exception handle
- Thread variable
- Atomic

This chapter supports 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,

Ibdex/input/ch12_inherit.cpp

```
#ifdef COUT_TEST
#include <iostream>
using namespace std;
#endif

extern "C" int printf(const char *format, ...);
extern "C" int sprintf(char *out, const char *format, ...);

class CPolygon { // _ZTVN10_cxxabiv117_class_type_infoE for parent class
protected:
    int width, height;
public:
    void set_values (int a, int b)
    { width=a; height=b; }
//    virtual int area (void) =0; // __cxa_pure_virtual
    virtual int area (void) { return 0; };
    void printarea (void)
#endif COUT_TEST
// generate IR nvoke, landing, resume and unreachable on iMac
    { cout << this->area() << endl; }
#else
    { printf("%d\n", this->area()); }
#endif
};

// _ZTVN10_cxxabiv120_si_class_type_infoE for derived class
class CRectangle: public CPolygon {
```

```

public:
    int area (void)
    { return (width * height); }
};

class CTriangle: public CPolygon {
public:
    int area (void)
    { return (width * height / 2); }
};

class CAngle: public CPolygon {
public:
    int area (void)
    { return (width * height / 4); }
};

#if 0
int test_cpp_polymorphism() {
    CPolygon * ppoly1 = new CRectangle;           // _Znwm
    CPolygon * ppoly2 = new CTriangle;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly1->printarea();
    ppoly2->printarea();
    delete ppoly1;                // _ZdlPv
    delete ppoly2;
    return 0;
}
#else
int test_cpp_polymorphism() {
    CRectangle poly1;
    CTriangle poly2;
    CAngle poly3;

    CPolygon * ppoly1 = &poly1;
    CPolygon * ppoly2 = &poly2;
    CPolygon * ppoly3 = &poly3;

    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);
    ppoly1->printarea();
    ppoly2->printarea();
    ppoly3->printarea();
    if (ppoly1->area() == 20 && ppoly2->area() == 10 && ppoly3->area() == 5)
        return 0;

    return 0;
}
#endif

```

If using cout instead of printf in ch12_inherit.cpp on Linux won't generate exception handler IRs. But on iMac, ch12_inherit.cpp will generate invoke, landing, resume and unreachable exception handler IRs. Example code, ch12_eh.cpp, which supports **try** and **catch** exception handler as the following will generate these exception handler IRs both on iMac and Linux.

Ibdex/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/cmake_debug_build/Debug/bin/llvm-dis ch12_eh.bc -o -

```

; ModuleID = 'ch12_eh.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"

%class.Ex1 = type { i8 }

$_ZTS3Ex1 = comdat any

$_ZTI3Ex1 = comdat any

 @_ZTVN10__cxxabiv117__class_type_infoE = external global i8*
 @_ZTS3Ex1 = linkonce_ode r constant [5 x i8] c"3Ex1\00", comdat
 @_ZTI3Ex1 = linkonce_ode r constant { i8*, i8* } { i8* bitcast (i8** getelementptr
    inbounds (i8*, i8**) @_ZTVN10__cxxabiv117__class_type_infoE, i32 2) to i8*}, i8*
    getelementptr inbounds ([5 x i8], [5 x i8]* @_ZTS3Ex1, i32 0, i32 0) }, comdat

define void @_Z15throw_exceptionii(i32 signext %a, i32 signext %b) #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %ex1 = alloca %class.Ex1, align 1
    store i32 %a, i32* %1, align 4
    store i32 %b, i32* %2, align 4
    %3 = load i32, i32* %1, align 4
    %4 = load i32, i32* %2, align 4
    %5 = icmp sgt i32 %3, %4
    br i1 %5, label %6, label %9

; <label>:6 ; preds = %0
    %7 = call i8* @_cxa_allocate_exception(i32 1) #1
    %8 = bitcast i8* %7 to %class.Ex1*
    call void @_cxa_throw(i8* %7, i8* bitcast ({ i8*, i8* }* @_ZTI3Ex1 to i8*), i
    8* null) #2

```

```
unreachable

; <label>:9                                ; preds = %0
    ret void
}

declare i8* @_cxa_allocate_exception(i32)

declare void @_cxa_throw(i8*, i8*, i8*)

define i32 @_Z14test_try_catchv() #0 personality i8* bitcast (i32 (...)* @_gxx_
personality_v0 to i8*) {
    %1 = alloca i32, align 4
    %2 = alloca i8*
    %3 = alloca i32
    invoke void @_Z15throw_exceptionii(i32 signext 2, i32 signext 1)
        to label %4 unwind label %5

; <label>:4                                ; preds = %0
    br label %12

; <label>:5                                ; preds = %0
    %6 = landingpad { i8*, i32 }
        catch i8* null
    %7 = extractvalue { i8*, i32 } %6, 0
    store i8* %7, i8** %2
    %8 = extractvalue { i8*, i32 } %6, 1
    store i32 %8, i32* %3
    br label %9

; <label>:9                                ; preds = %5
    %10 = load i8*, i8** %2
    %11 = call i8* @_cxa_begin_catch(i8* %10) #1
    store i32 1, i32* %1
    call void @_cxa_end_catch()
    br label %13

; <label>:12                               ; preds = %4
    store i32 0, i32* %1
    br label %13

; <label>:13                               ; preds = %12, %9
    %14 = load i32, i32* %1
    ret i32 %14
}

declare i32 @_gxx_personality_v0(...)

declare i8* @_cxa_begin_catch(i8*)

declare void @_cxa_end_catch()

attributes #0 = { "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-
frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="f
alse" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="
mips32r2" "target-features"="+mips32r2" "unsafe-fp-math"="false" "use-soft-float
"="false" }
attributes #1 = { nounwind }
```

```

attributes #2 = { noreturn }

!llvm.ident = !{!0}

!0 = !{!"clang version 3.7.0 (tags/RELEASE_370/final)"}

JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=static -filetype=asm ch12_eh.bc -o -
    .section .mdebug.abi32
    .previous
    .file "ch12_eh.bc"
llc: /Users/Jonathan/llvm/test/src/lib/CodeGen/LiveVariables.cpp:133: void llvm::
LiveVariables::HandleVirtRegUse(unsigned int, llvm::MachineBasicBlock *, llvm
::MachineInstr *): Assertion `MRI->getVRegDef(reg) && "Register use before
def!"' failed.

```

About the IRs of LLVM exception handling, please reference here¹. Chapter12_1 supports the llvm IRs of corresponding **try** and **catch** exception C++ keywords. It can compile ch12_eh.bc as follows,

Ibdex/chapters/Chapter12_1/Cpu0ISelLowering.cpp

```

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

    setExceptionPointerRegister(Cpu0::A0);
    setExceptionSelectorRegister(Cpu0::A1);

}

JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=static -filetype=asm ch12_eh.bc -o -
    .text
    .section .mdebug.abi032
    .previous
    .file "ch12_eh.bc"
    .globl _Z15throw_exceptionii
    .align 2
    .type _Z15throw_exceptionii,@function
    .ent _Z15throw_exceptionii    # @_Z15throw_exceptionii
_Z15throw_exceptionii:
    .cfi_startproc
    .frame $fp,40,$lr
    .mask 0x00005000,-4
    .set noreorder
    .set nomacro
# BB#0:
    addiu $sp, $sp, -40
$tmp0:
    .cfi_def_cfa_offset 40
    st $lr, 36($sp)          # 4-byte Folded Spill
    st $fp, 32($sp)          # 4-byte Folded Spill
$tmp1:
    .cfi_offset 14, -4

```

¹ <http://llvm.org/docs/ExceptionHandling.html>

```
$tmp2:  
    .cfi_offset 12, -8  
    move    $fp, $sp  
$tmp3:  
    .cfi_def_cfa_register 12  
    st     $4, 28($fp)  
    st     $5, 24($fp)  
    ld     $2, 28($fp)  
    cmp    $sw, $2, $5  
    jle    $sw, LBB0_2  
    nop  
    jmp    LBB0_1  
    nop  
LBB0_2:  
    move    $sp, $fp  
    ld     $fp, 32($sp)           # 4-byte Folded Reload  
    ld     $lr, 36($sp)           # 4-byte Folded Reload  
    addiu $sp, $sp, 40  
    ret    $lr  
    nop  
LBB0_1:  
    addiu $4, $zero, 1  
    jsub  __cxa_allocate_exception  
    nop  
    addiu $3, $zero, 0  
    st     $3, 8($sp)  
    lui   $3, %hi(_ZTI3Ex1)  
    ori   $5, $3, %lo(_ZTI3Ex1)  
    addu $4, $zero, $2  
    jsub  __cxa_throw  
    nop  
.set  macro  
.set  reorder  
.end _Z15throw_exceptionii  
$func_end0:  
    .size _Z15throw_exceptionii, ($func_end0)-_Z15throw_exceptionii  
.cfi_endproc  
  
.globl _Z14test_tryCatchv  
.align 2  
.type _Z14test_tryCatchv,@function  
.ent _Z14test_tryCatchv      # @_Z14test_tryCatchv  
_Z14test_tryCatchv:  
$tmp7:  
$func_begin0 = ($tmp7)  
.cfi_startproc  
.cfi_personality 0, __gxx_personality_v0  
.cfi_lsda 0, $exception0  
.frame $fp,32,$lr  
.mask 0x00005200,-4  
.set noreorder  
.set nomacro  
# BB#0:  
    addiu $sp, $sp, -32  
$tmp8:  
.cfi_def_cfa_offset 32  
    st $lr, 28($sp)           # 4-byte Folded Spill  
    st $fp, 24($sp)           # 4-byte Folded Spill
```

```

        st  $9, 20($sp)          # 4-byte Folded Spill
$tmp9:
        .cfi_offset 14, -4
$tmp10:
        .cfi_offset 12, -8
$tmp11:
        .cfi_offset 9, -12
        move  $fp, $sp
$tmp12:
        .cfi_def_cfa_register 12
$tmp4:
        addiu $4, $zero, 2
        addiu $9, $zero, 1
        addu  $5, $zero, $9
        jsub  _Z15throw_exceptionii
        nop
$tmp5:
# BB#2:
        addiu $2, $zero, 0
        st   $2, 16($fp)
LBB1_3:
        ld   $2, 16($fp)
        move $sp, $fp
        ld   $9, 20($sp)          # 4-byte Folded Reload
        ld   $fp, 24($sp)          # 4-byte Folded Reload
        ld   $lr, 28($sp)          # 4-byte Folded Reload
        addiu $sp, $sp, 32
        ret   $lr
        nop
LBB1_1:
$tmp6:
        st   $4, 12($fp)
        st   $5, 8($fp)
        ld   $4, 12($fp)
        jsub __cxa_begin_catch
        nop
        st   $9, 16($fp)
        jsub __cxa_end_catch
        nop
        jmp  LBB1_3
        nop
        .set  macro
        .set  reorder
        .end  _Z14test_tryCatchv
$func_end1:
        .size _Z14test_tryCatchv, ($func_end1)-_Z14test_tryCatchv
        .cfi_endproc
        .section .gcc_except_table,"a",@progbits
        .align 2
GCC_except_table1:
$exception0:
        .byte 255                 # @LPStart Encoding = omit
        .byte 0                  # @TTType Encoding = absptr
        .asciz "\242\200\200"      # @TTType base offset
        .byte 3                  # Call site Encoding = udata4
        .byte 26                 # Call site table length
        .4byte ($tmp4)-($func_begin0) # >> Call Site 1 <<
        .4byte ($tmp5)-($tmp4)      # Call between $tmp4 and $tmp5

```

```

    .4byte  ($tmp6)-($func_begin0)  #      jumps to $tmp6
    .byte  1                      #      On action: 1
    .4byte  ($tmp5)-($func_begin0)  # >> Call Site 2 <<
    .4byte  ($func_end1)-($tmp5)   #      Call between $tmp5 and $func_end1
    .4byte  0                      #      has no landing pad
    .byte  0                      #      On action: cleanup
    .byte  1                      # >> Action Record 1 <<
                                    #      Catch TypeInfo 1
    .byte  0                      #      No further actions
                                    # >> Catch TypeInfos <<
    .4byte  0                      # TypeInfo 1
    .align  2

.type  _ZTS3Ex1,@object          # @_ZTS3Ex1
.section .rodata._ZTS3Ex1,"aG",@progbits,_ZTS3Ex1,comdat
.weak  _ZTS3Ex1
.align  2
_ZTS3Ex1:
.asciz  "3Ex1"
.size  _ZTS3Ex1, 5

.type  _ZTI3Ex1,@object          # @_ZTI3Ex1
.section .rodata._ZTI3Ex1,"aG",@progbits,_ZTI3Ex1,comdat
.weak  _ZTI3Ex1
.align  3
_ZTI3Ex1:
.4byte  _ZTVN10__cxxabiv117__class_type_infoE+
.4byte  _ZTS3Ex1
.size  _ZTI3Ex1, 8

```

12.2 Thread variable

C++ include the thread variable as the following file ch12_thread_var.cpp.

Ibdex/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 name².

² http://en.wikipedia.org/wiki/Thread-local_storage

To support thread variable, the following code added to Chapter12_1. 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.

Ibdex/chapters/Chapter12_1/AsmParser/Cpu0AsmParser.cpp

```
MCSymbolRefExpr::VariantKind Cpu0AsmParser::getVariantKind(StringRef Symbol) {
    .Case("tlsldm", MCSymbolRefExpr::VK_Cpu0_TLSLDM)
    .Case("dtp_hi", MCSymbolRefExpr::VK_Cpu0_DTP_HI)
    .Case("dtp_lo", MCSymbolRefExpr::VK_Cpu0_DTP_LO)
    .Case("gottp", MCSymbolRefExpr::VK_Cpu0_GOTTPREL)
    .Case("tp_hi", MCSymbolRefExpr::VK_Cpu0_TP_HI)
    .Case("tp_lo", MCSymbolRefExpr::VK_Cpu0_TP_LO)

    ...
}
```

Ibdex/chapters/Chapter12_1/InstPrinter/Cpu0InstPrinter.cpp

```
static void printExpr(const MCExpr *Expr, raw_ostream &OS) {
    case MCSymbolRefExpr::VK_Cpu0_TLSGD: OS << "%tlsldm("; break;
    case MCSymbolRefExpr::VK_Cpu0_TLSLDM: OS << "%dtp_hi("; break;
    case MCSymbolRefExpr::VK_Cpu0_DTP_HI: OS << "%dtp_lo("; break;
    case MCSymbolRefExpr::VK_Cpu0_DTP_LO: OS << "%gottprel("; break;
    case MCSymbolRefExpr::VK_Cpu0_TP_HI: OS << "%tp_hi("; break;
    case MCSymbolRefExpr::VK_Cpu0_TP_LO: OS << "%tp_lo("; break;

    ...
}
```

Ibdex/chapters/Chapter12_1/MCTargetDesc/Cpu0AsmBackend.cpp

```
const MCFixupKindInfo &Cpu0AsmBackend::getFixupKindInfo(MCFixupKind Kind) const {
    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 },
        ...
    };
    ...
}
```

Ibdex/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,
        /// 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,
        ...
    };
    ...
}
```

Ibdex/chapters/Chapter12_1/MCTargetDesc/Cpu0ELFOBJECTWriter.cpp

```
unsigned Cpu0ELFOBJECTWriter::GetRelocType(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;
        ...
    }
}
```

Ibdex/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,
    // 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,
    ...
};
```

Ibdex/chapters/Chapter12_1/MCTargetDesc/Cpu0MCCCodeEmitter.cpp

```
unsigned Cpu0MCCCodeEmitter::
getExprOpValue(const MCExpr *Expr, SmallVectorImpl<MCFixup> &Fixups,
               const MCSubtargetInfo &STI) const {
    case MCSymbolRefExpr::VK_Cpu0_TLSGD:
        FixupKind = Cpu0::fixup_Cpu0_TLSGD;
        break;
    case MCSymbolRefExpr::VK_Cpu0_TLSLDM:
        FixupKind = Cpu0::fixup_Cpu0_TLSLDM;
        break;
    case MCSymbolRefExpr::VK_Cpu0_DTP_HI:
        FixupKind = Cpu0::fixup_Cpu0_DTP_HI;
        break;
    case MCSymbolRefExpr::VK_Cpu0_DTP_LO:
        FixupKind = Cpu0::fixup_Cpu0_DTP_LO;
        break;
    case MCSymbolRefExpr::VK_Cpu0_GOTTPREL:
        FixupKind = Cpu0::fixup_Cpu0_GOTTPREL;
        break;
    case MCSymbolRefExpr::VK_Cpu0_TP_HI:
        FixupKind = Cpu0::fixup_Cpu0_TP_HI;
        break;
    case MCSymbolRefExpr::VK_Cpu0_TP_LO:
        FixupKind = Cpu0::fixup_Cpu0_TP_LO;
        break;
```

```
    ...
}
```

Ibdex/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 : WrapperPat<tglobaltlsaddr, ORi, CPURegs>;
}
```

Ibdex/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);

    ...
}
}

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);
```

```

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_TLSDM
                                                    : Cpu0II::MO_TLSD;
}

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), 0);
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);
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(),
                         false, false, false, 0);
} 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;
}

```

Ibdex/chapters/Chapter12_1/Cpu0SelLowering.h

```

MachineBasicBlock *
EmitInstrWithCustomInserter(MachineInstr *MI,
                           MachineBasicBlock *MBB) const override;

```

Ibdex/chapters/Chapter12_1/Cpu0MCInstLower.cpp

```

MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,
                                              MachineOperandType MOTy,
                                              unsigned Offset) const {
    MCSymbolRefExpr::VariantKind Kind;
    const MCSymbol *Symbol;

    switch (MO.getTargetFlags()) {
        case Cpu0II::MO_TLSGD:      Kind = MCSymbolRefExpr::VK_Cpu0_TLSGD; break;
        case Cpu0II::MO_TSLDM:      Kind = MCSymbolRefExpr::VK_Cpu0_TSLDM; break;
        case Cpu0II::MO_DTP_HI:     Kind = MCSymbolRefExpr::VK_Cpu0_DTP_HI; break;
        case Cpu0II::MO_DTP_LO:     Kind = MCSymbolRefExpr::VK_Cpu0_DTP_LO; break;
        case Cpu0II::MO_GOTTPREL:   Kind = MCSymbolRefExpr::VK_Cpu0_GOTTPREL; break;
        case Cpu0II::MO_TP_HI:      Kind = MCSymbolRefExpr::VK_Cpu0_TP_HI; break;
        case Cpu0II::MO_TP_LO:      Kind = MCSymbolRefExpr::VK_Cpu0_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/cmake_debug_build/
Debug/bin/llvm-dis ch12_thread_var.bc -o -

```

```

; ModuleID = 'ch12_thread_var.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"

@a = thread_local global i32 0, align 4
@b = thread_local global i32 0, align 4

; Function Attrs: nounwind
define i32 @_Z15test_thread_varv() #0 {
    store i32 2, i32* @a, align 4
}
```

```
%1 = load i32, i32* @a, align 4
ret i32 %1
}

define i32 @_Z17test_thread_var_2v() #1 {
%1 = call i32* @_ZTW1b()
store i32 3, i32* %1, align 4
%2 = call i32* @_ZTW1b()
%3 = load i32, i32* %2, align 4
ret i32 %3
}

define weak_odr hidden i32* @_ZTW1b() {
ret i32* @b
}

attributes #0 = { nounwind "disable-tail-calls=false" "less-precise-fpmad=false"
"no-frame-pointer-elim=true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-
-math=false" "no-nans-fp-math=false" "stack-protector-buffer-size=8" "target-
-cpu=mips32r2" "target-features=+mips32r2" "unsafe-fp-math=false" "use-sof-
-float=false" }
attributes #1 = { "disable-tail-calls=false" "less-precise-fpmad=false" "no-
frame-pointer-elim=true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math="fa
lse" "no-nans-fp-math=false" "stack-protector-buffer-size=8" "target-cpu="
mips32r2" "target-features=+mips32r2" "unsafe-fp-math=false" "use-soft-float
=false" }

!llvm.ident = !{!0}

!0 = !{!"clang version 3.7.0 (tags/RELEASE_370/final)"}

JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch12_thread_var.bc
-o -

.text
.section .mdebug.abi032
.previous
.file "ch12_thread_var.bc"
.globl _Z15test_thread_varv
.align 2
.type _Z15test_thread_varv,@function
.ent _Z15test_thread_varv    # @_Z15test_thread_varv
_Z15test_thread_varv:
.frame $fp,16,$lr
.mask 0x00005000,-4
.set noreorder
.cupload $t9
.set nomacro
# BB#0:
addiu $sp, $sp, -16
st $lr, 12($sp)          # 4-byte Folded Spill
st $fp, 8($sp)           # 4-byte Folded Spill
move $fp, $sp
.cprestore 8
ld $t9, %call16(__tls_get_addr)($gp)
ori $4, $gp, %tlsgd(a)
jalr $t9
```

```
nop
ld $gp, 8($fp)
addiu $3, $zero, 2
st $3, 0($2)
addu $2, $zero, $3
move $sp, $fp
ld $fp, 8($sp)           # 4-byte Folded Reload
ld $lr, 12($sp)          # 4-byte Folded Reload
addiu $sp, $sp, 16
ret $lr
nop
.set macro
.set reorder
.end _Z15test_thread_varv
$func_end0:
.size _Z15test_thread_varv, ($func_end0)-_Z15test_thread_varv

.globl _Z17test_thread_var_2v
.align 2
.type _Z17test_thread_var_2v,@function
.ent _Z17test_thread_var_2v # @_Z17test_thread_var_2v
_Z17test_thread_var_2v:
.cfi_startproc
.frame $fp,16,$lr
.mask 0x00005000,-4
.set noreorder
.cupload $t9
.set nomacro
# BB#0:
addiu $sp, $sp, -16
$tmp0:
.cfi_def_cfa_offset 16
st $lr, 12($sp)          # 4-byte Folded Spill
st $fp, 8($sp)           # 4-byte Folded Spill
$tmp1:
.cfi_offset 14, -4
$tmp2:
.cfi_offset 12, -8
move $fp, $sp
$tmp3:
.cfi_def_cfa_register 12
.cprestore 8
ld $t9, %call16(_ZTW1b)($gp)
jalr $t9
nop
ld $gp, 8($fp)
addiu $3, $zero, 3
st $3, 0($2)
ld $t9, %call16(_ZTW1b)($gp)
jalr $t9
nop
ld $gp, 8($fp)
ld $2, 0($2)
move $sp, $fp
ld $fp, 8($sp)           # 4-byte Folded Reload
ld $lr, 12($sp)          # 4-byte Folded Reload
addiu $sp, $sp, 16
ret $lr
```

```

nop
.set macro
.set reorder
.end _Z17test_thread_var_2v
$func_end1:
.size _Z17test_thread_var_2v, ($func_end1)-_Z17test_thread_var_2v
.cfi_endproc

.hidden _ZTW1b
.weak _ZTW1b
.align 2
.type _ZTW1b,@function
.ent _ZTW1b                      # @_ZTW1b
_ZTW1b:
.cfi_startproc
.frame $sp,16,$lr
.mask 0x00004000,-4
.set noreorder
.cupload $t9
.set nomacro
# BB#0:
addiu $sp, $sp, -16
$tmp4:
.cfi_def_cfa_offset 16
st $lr, 12($sp)                  # 4-byte Folded Spill
$tmp5:
.cfi_offset 14, -4
.cprestore 8
ld $t9, %call16(__tls_get_addr)($gp)
ori $4, $gp, %tlsgd(b)
jalr $t9
nop
ld $gp, 8($sp)
ld $lr, 12($sp)                  # 4-byte Folded Reload
addiu $sp, $sp, 16
ret $lr
nop
.set macro
.set reorder
.end _ZTW1b
$func_end2:
.size _ZTW1b, ($func_end2)-_ZTW1b
.cfi_endproc

.type a,@object                 # @a
.section .tbss,"awT",@nobits
.globl a
.align 2
a:
.4byte 0                         # 0x0
.size a, 4

.type b,@object                 # @b
.globl b
.align 2
b:
.4byte 0                         # 0x0
.size b, 4

```

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 call function _ZTW1b which call the function __tls_get_addr too to get the thread_local variable address. In static mode, the thread variable is accessed by machine instructions as follows,

```
JonathantekiiMac:input Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
Debug/bin/llc -march=cpu0 -relocation-model=static -filetype=asm
ch12_thread_var.bc -o -

.text
.section .mdebug.abi032
.previous
.file "ch12_thread_var.bc"
.globl _Z15test_thread_varv
.align 2
.type _Z15test_thread_varv,@function
.ent _Z15test_thread_varv # @_Z15test_thread_varv
_Z15test_thread_varv:
.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
 ori $2, $zero, %tp_lo(a)
 lui $3, %tp_hi(a)
 addu $3, $3, $2
 addiu $2, $zero, 2
.st $2, 0($3)
.move $sp, $fp
 ld $fp, 4($sp)           # 4-byte Folded Reload
.addiu $sp, $sp, 8
.ret $lr
.nop
.set macro
.set reorder
.end _Z15test_thread_varv
$func_end0:
.size _Z15test_thread_varv, ($func_end0)-_Z15test_thread_varv

.globl _Z17test_thread_var_2v
.align 2
.type _Z17test_thread_var_2v,@function
.ent _Z17test_thread_var_2v # @_Z17test_thread_var_2v
_Z17test_thread_var_2v:
.cfi_startproc
.frame $fp,16,$lr
.mask 0x00005000,-4
.set noreorder
.set nomacro
# BB#0:
.addiu $sp, $sp, -16
$tmp0:
.cfi_def_cfa_offset 16
.st $lr, 12($sp)           # 4-byte Folded Spill
.st $fp, 8($sp)             # 4-byte Folded Spill
$tmp1:
```

```

.cfi_offset 14, -4
$tmp2:
.cfi_offset 12, -8
move $fp, $sp
$tmp3:
.cfi_def_cfa_register 12
jsub _ZTW1b
nop
addiu $3, $zero, 3
st $3, 0($2)
jsub _ZTW1b
nop
ld $2, 0($2)
move $sp, $fp
ld $fp, 8($sp)           # 4-byte Folded Reload
ld $lr, 12($sp)          # 4-byte Folded Reload
addiu $sp, $sp, 16
ret $lr
nop
.set macro
.set reorder
.end _Z17test_thread_var_2v
$func_end1:
.size _Z17test_thread_var_2v, ($func_end1)-_Z17test_thread_var_2v
.cfi_endproc

.hidden _ZTW1b
.weak _ZTW1b
.align 2
.type _ZTW1b,@function
.ent _ZTW1b               # @_ZTW1b
_ZTW1b:
.cfi_startproc
.frame $sp,0,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
ori $2, $zero, %tp_lo(b)
lui $3, %tp_hi(b)
addu $2, $3, $2
ret $lr
nop
.set macro
.set reorder
.end _ZTW1b
$func_end2:
.size _ZTW1b, ($func_end2)-_ZTW1b
.cfi_endproc

.type a,@object           # @a
.section .tbss,"awT",@nobits
.globl a
.align 2
a:
.4byte 0                  # 0x0
.size a, 4

```

```
.type b,@object          # @b
.globl b
.align 2
b:
    .4byte 0             # 0x0
    .size b, 4
```

While Mips uses rdhwr instruction to access thread variable as below, Cpu0 access thread variable without inventing any new instruction. The thread variables are kept in thread variable memory location which accessed through %tp_hi and %tp_lo. Furthermore, this section of memory is protected through kernel mode program. As a result, 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/cmake_debug_build/
Debug/bin/llc -march=mips -relocation-model=static -filetype=asm
ch12_thread_var.bc -o -
...
    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 variable 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 variable 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 use 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 benefit is the compiler can generate high performance code by the target hardware instruction rather than counting on lock() function only^{3 4 5}.

In order to support atomic in C++ and java, llvm provides the atomic IRs here^{6 7}.

To support llvm atomic IRs, the following code added to Chapter12_1.

Ibdex/chapters/Chapter12_1/AsmParser/Cpu0AsmParser.cpp

```
MCSymbolRefExpr::VariantKind Cpu0AsmParser::getVariantKind(StringRef Symbol) {
    .Case("tlsldm", MCSymbolRefExpr::VK_Cpu0_TLSLDM)
    .Case("dtp_hi", MCSymbolRefExpr::VK_Cpu0_DTP_HI)
    .Case("dtp_lo", MCSymbolRefExpr::VK_Cpu0_DTP_LO)
    .Case("gottp", MCSymbolRefExpr::VK_Cpu0_GOTTPREL)
    .Case("tp_hi", MCSymbolRefExpr::VK_Cpu0_TP_HI)
    .Case("tp_lo", MCSymbolRefExpr::VK_Cpu0_TP_LO)
    ...
}
```

Ibdex/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));
    }
    ...
}
```

Ibdex/chapters/Chapter12_1/Cpu0InstrInfo.td

```
def SDT_Sync : SDTypeProfile<0, 1, [SDTCisVT<0, i32]>;
def Cpu0Sync : SDNode<"Cpu0ISD::Sync", SDT_Sync, [SDNPHasChain]>;
```

³ https://en.wikipedia.org/wiki/Memory_model_%28programming%29

⁴ <http://stackoverflow.com/questions/6319146/c11-introduced-a-standardized-memory-model-what-does-it-mean-and-how-is-it-g>

⁵ <http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-model-and-modern-hardware/>

⁶ <http://llvm.org/docs/Atomics.html>

⁷ <http://llvm.org/docs/LangRef.html#ordering>

```

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>;

```

[Index/chapters/Chapter12_1/Cpu0SelLowering.h](#)

```

MachineBasicBlock *
EmitInstrWithCustomInserter(MachineInstr *MI,
                            MachineBasicBlock *MBB) const override;

SDValue lowerATOMIC_FENCE(SDValue Op, SelectionDAG& DAG) const;

/// 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;

```

[Index/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);

setInsertFencesForAtomic(true);

SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
    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);
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
    const BasicBlock *LLVM_BB = BB->getBasicBlock();
    MachineBasicBlock *loopMBB = MF->CreateMachineBasicBlock(LLVM_BB);
    MachineBasicBlock *exitMBB = MF->CreateMachineBasicBlock(LLVM_BB);

```

```

MachineFunction::iterator It = BB;
++It;
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();

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
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;
++It;
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    mask1sb2,$0,-4          # 0xfffffffffc
// and     alignedaddr,ptr,mask1sb2
// andi   ptr1sb2,ptr,3
// sll    shiftamt,ptr1sb2,3
// 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);

// 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))
    .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;
++It;
MF->insert(It, loop1MBB);
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();

    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;
    ++It;
    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           # 0xffffffffc
// 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);

// 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));
}

```

Index/chapters/Chapter12_1/Cpu0RegisterInfo.h

```

/// Code Generation virtual methods...
const TargetRegisterClass *getPointerRegClass(const MachineFunction &MF,
                                              unsigned Kind) const override;

```

Ibdex/chapters/Chapter12_1/Cpu0RegisterInfo.cpp

```
const TargetRegisterClass *
Cpu0RegisterInfo::getPointerRegClass(const MachineFunction &MF,
                                     unsigned Kind) const {
    return &Cpu0::CPURegsRegClass;
}
```

Ibdex/chapters/Chapter12_1/Cpu0SEISelLowering.cpp

```
Cpu0SEITargetLowering::Cpu0SEITargetLowering(const Cpu0TargetMachine &TM,
                                              const Cpu0Subtarget &STI)
    : Cpu0TargetLowering(TM, STI) {

    setOperationAction(ISD::ATOMIC_FENCE, MVT::Other, Custom);
    ...
}
```

Ibdex/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(&getCpu0TargetMachine()));
}
```

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 change as above.

The atomic node defined in “let usesCustomInserter = 1 in” of Cpu0InstrInfo.td tells llvm calling EmitInstrWithCustomInserter() of Cpu0ISelLowering.cpp. 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 createAtomicExpandPass() in 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 12.3: 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++ source files for generating llvm atomic IRs. The C++ files need to run with clang options “clang++ -pthread -std=c++11”.

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 blue part of Figure 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.

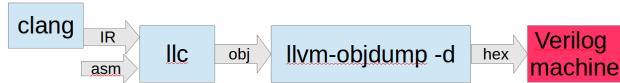


Figure 13.1: Cpu0 backend without linker

This chapter will implement Cpu0 instructions by Verilog language as the red part of Figure 13.1. With this Verilog machine, we can write a C++ main function as well as the assembly boot code, and translate this `main() + bootcode()` into obj file. Combined with `llvm-objdump` support in chapter ELF, this `main() + bootcode()` elf can be translated into hex file format which include the disassemble code as comment. Furthermore, 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 existed 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` and `cpu0Is.v` are simple design with only few hundreds lines of code totally. Although it has not the pipeline features, we can assume the Cpu0 backend code run on the pipeline machine with NOP instruction fill in branch delay slot because the pipeline version use the same machine instructions. Verilog is a C like language in syntax and this book is a compiler book, so we list the `cpu0.v` as well as the building command directly as below. We expect readers can understand the Verilog code just with a little patient and no further explanation needed. According computer architecture, there are two type of I/O. One is memory mapped I/O, the other is instruction I/O. Cpu0 use

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>

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 display the content as follows,

```
ST : begin
  ...
  if (R[b]+c16 == `IOADDR) begin
    outw(R[a]);
```

Ibdex/verilog/cpu0.v

```
'define MEMSIZE      'h80000
'define MEMEMPTY     8'hFF
'define NULL         8'h00
'define IOADDR       'h80000 // IO mapping address

// 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] COR [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, uc16, c24, Ra, Rb, Rc, pc0; // pc0: instruction pc
  reg [31:0] URa, URb, URc, HI, LO, CF, tmp;
  reg [63:0] cycles;

  // 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   COR[0]   // Program Counter
  'define EPC  COR[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 IO   '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 IOE    '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,
CMP=8'h10,
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,
ROL=8'h1B,ROR=8'h1C,SRA=8'h1D,SHL=8'h1E,SHR=8'h1F,
SRAV=8'h20,SHLV=8'h21,SHRV=8'h22,ROLV=8'h23,RORV=8'h24,
`ifdef CPU0II
    SLTi=8'h26,SLTi=8'h27, SLT=8'h28,SLTu=8'h29,
    BEQ=8'h37,BNE=8'h38,
`endif
    JEQ=8'h30,JNE=8'h31,JLT=8'h32,JGT=8'h33,JLE=8'h34,JGE=8'h35,
    JMP=8'h36,
    JALR=8'h39,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,COMOV=8'h52;

reg [0:0] inExe = 0;
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;

//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

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; 'IOE = 1; 'I1E = 1; 'I2E = 1;
            'I = 0; 'IO = 0; 'I1 = 0; 'I2 = 0; inExe = 1;
            '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;
    cycles = cycles+1;
    case (state)
        Fetch: begin // Tick 1 : instruction fetch, throw PC to address bus,
                  // memory.read(m[PC])
            memReadStart(`PC, `INT32);
            pc0 = `PC;
            `PC = `PC+4;
            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:   ;
                // 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) R[a]=Rb;           // move if Rc equal to 0
                MOVN:  if (Rc!=0) R[a]=Rb;           // move if Rc not equal to 0
                // Mathematic
                ADDiu: R[a] = Rb+c16;           // ADDiu Ra, Rb+Cx; Ra<=Rb+Cx

```

```

CMP:    begin `N=(Rb-Rc<0); `Z=(Rb-Rc==0); end // CMP Rb, Rc; SW=(Rb >= Rc)
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
                                // 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)
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&'h80000000) | (Rb>>c5));
                                // Shift Right with signed bit fill;
                                // SHR Ra,Rb,Cx; Ra<=(Rb&0x80000000) | (Rb>>Cx)
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&'h80000000) | (Rb>>Rc));
                                // Shift Right with signed bit fill;
                                // SHRV Ra,Rb,Rc; Ra<=(Rb&0x80000000) | (Rb>>Rc)
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 efficiently.
        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 0x00000000ffffffffff);
                                // with exception overflow
MULTu: {HI, LO}=URa*URb;       // MULT URa,URb; HI<=((URa*URb)>>32);
                                // LO<=((URa*URb) and 0x00000000ffffffffff);
                                // without exception overflow
MFC0: regSet(a, C0R[b]);       // MFC0 a, b; Ra<=C0R[Rb]
MTC0: C0regSet(a, Rb);         // MTC0 a, b; C0R[a]<=Rb
COMOV: C0regSet(a, C0R[b]);    // COMOV a, b; C0R[a]<=C0R[b]
`ifdef CPU0II
// set
SLT: if (Rb < Rc) R[a]=1; else R[a]=0;
SLTu: if (Rb < Rc) R[a]=1; else R[a]=0;
SLTi: if (Rb < c16) R[a]=1; else R[a]=0;
SLTi: if (Rb < c16) R[a]=1; else R[a]=0;
// Branch Instructions
BEQ: if (Ra==Rb) `PC='PC+c16;
BNE: if (Ra!=Rb) `PC='PC+c16;
`endif
// Jump Instructions
JEQ: if ('Z) `PC='PC+c24;      // JEQ Cx; if SW(=) PC PC+Cx
JNE: if (!'Z) `PC='PC+c24;      // JNE Cx; if SW(!=) PC PC+Cx
JLT: if ('N) `PC='PC+c24;       // JLT Cx; if SW(<) PC PC+Cx
JGT: if (!'N&!`Z) `PC='PC+c24; // JGT Cx; if SW(>) PC PC+Cx
JLE: if ('N || 'Z) `PC='PC+c24; // JLE Cx; if SW(<=) PC PC+Cx
JGE: if (!'N || 'Z) `PC='PC+c24; // JGE Cx; if SW(>=) PC PC+Cx
JMP: `PC = 'PC+c24;            // JMP Cx; PC <= PC+Cx
JSUB: begin `LR='PC; `PC='PC + c24; end // JSUB Cx; LR<=PC; PC<=PC+Cx
JALR: begin R[a] = 'PC; `PC=Rb; end // JALR Ra,Rb; Ra<=PC; PC<=Rb
RET: begin `PC=Ra; end         // RET; PC <= Ra
default :
    $display("%4dns %8x : OP code %8x not support", $stime, pc0, op);
endcase
if ('IE && 'I && ('IOE && 'IO || '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
    endcase
end

```

```

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'hffff80;
end
LH : begin
    if (R[a] > 16'hffff) 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);
COMOV :
    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) && ('IOE && 'IO)) begin
        // 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
        // 'D = 1; // Trace register content at beginning
        taskExecute();
        state = next_state;
    end
    pc = 'PC;
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];
`ifdef DLINKER
    reg [7:0] flash [0:'MEMSIZE-1];
    reg [7:0] dsym [0:192-1];
    reg [7:0] dstr [0:96-1];
    reg [7:0] so_func_offset[0:384-1];
    reg [7:0] globalAddr [0:3];
    reg [31:0] pltAddr [0:0];
    reg [31:0] gp;
    reg [31:0] gpPlt;
    reg [31:0] fabus;
    integer j;
    integer k;
    integer l;
    reg [31:0] j32;
    integer numDynEntry;
`endif
    reg [31:0] data;

    integer i;

```

```

`define LE  mconfig[0][0:0]    // Endian bit, Big Endian:0, Little Endian:1

`ifdef DLINKER
`include "dynlinker.v"
`endif
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
    $display("%8x: %8x", i, {m[i], m[i+1], m[i+2], m[i+3]});
  end
`endif
`endif
`ifdef DLINKER
  loadToFlash();
  createDynInfo();
`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 // LittleEndian
        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 // BigEndian
        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 // LittleEndian
        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 // BigEndian

```

```

        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;
`ifdef DLINKER
`include "flashio.v"
`endif
end else
    data = 32'hZZZZZZZZ;
end
assign dbus_out = data;
assign cfg = mconfig[0][0:0];
endmodule

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;
    #300000000 $finish;
end

always #10 clock=clock+1;

endmodule

```

Ibdex/verilog/cpu0ls.v

```

// TRACE: Display the memory contents of the loaded program and data
//`define TRACE

`include "cpu0.v"

```

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 set endian tranfer in memory module, the endian transfer can 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 Verilog. In system with bus, it suppose set in bus system I think.

```
JonathantekiiMac:raw Jonathan$ pwd
/Users/Jonathan/test/2/lbd/lbdex/verilog/raw
JonathantekiiMac:raw Jonathan$ iverilog -o cpu0Is cpu0Is.v
```

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. We set \$sp at 0x6ffc because assuming cpu0.v uses 0x7000 bytes of memory.

lbdex/input/start.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");
```

lbdex/input/boot.cpp

```
#include "start.h"
```

```

// boot:
asm("boot:");
// _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");

asm("addiu $sp, $zero, 0x6ffc");
asm("mfco $3, $pc");
asm("addiu $3, $3, 0x8"); // Assume main() entry point is at the next next
                           // instruction.
asm("ret $3");
asm("nop");

```

lbdex/input/print.h

```

#ifndef _PRINT_H_
#define _PRINT_H_

#define OUT_MEM 0x80000

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*)OUT_MEM;
    *p = c;

    return;
}

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_noll.h

```
#include "debug.h"
#include "boot.cpp"

#include "print.h"

int test_noll();
```

lbdex/input/ch_noll.cpp

```
#define TEST_ROXV
#define RUN_ON_VERILOG

#include "print.cpp"

#include "ch4_1_1.cpp"
#include "ch4_1_3.cpp"
#include "ch4_3.cpp"
#include "ch4_5.cpp"
#include "ch7_1.cpp"
#include "ch7_2_2.cpp"
#include "ch7_3.cpp"
#include "ch7_4.cpp"
#include "ch8_1_1.cpp"
#include "ch8_2.cpp"
#include "ch8_3.cpp"
#include "ch9_1_4.cpp"
#include "ch9_2_3_tailcall.cpp"
#include "ch9_3.cpp"
#include "ch11_2.cpp"

// Test build only for the following files since it needs lld linker support.
#include "ch6_1.cpp"
#include "ch9_2_1.cpp"
#include "ch9_2_2.cpp"
#include "ch9_3_2.cpp"
#include "ch12_inherit.cpp"

void test_asm_build()
{
```

```

#include "ch11_1.cpp"
#ifndef CPU032II
#include "ch11_1_2.cpp"
#endif
}

int test_rotate()
{
    int a = test_rotate_left1(4, 30); // rolv 4, 30 = 1
    int b = test_rotate_left(); // rol 8, 30 = 2
    int c = test_rotate_right(1, 30); // rorv 1, 30 = 4
    int d = test_rotate_left1(1, 3); // rolv 1, 3 = 8

    return (a+b+c+d);
}

int test_nollid()
{
    bool pass = true;
    int a = 0;

    a = test_math();
    print_integer(a); // a = 74
    if (a != 74) pass = false;
    a = test_rotate();
    print_integer(a); // a = 15
    if (a != 15) 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_andorxornot();
    print_integer(a); // a = 14
    if (a != 14) 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); // 16777222
}

```

```
if ((int)(b) != 16777222) 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;
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_madd();
print_integer(a); // a = 7
if (a != 7) pass = false;
a = test_tailcall(5);
print_integer(a); // a = 120
if (a != 120) pass = false;
a = test_vararg();
print_integer(a); // a = 15
if (a != 15) pass = false;
a = test_inlineasm();
print_integer(a); // a = 49
if (a != 49) pass = false;

    return pass;
}

/* result:
74
15
253
3
1
14
3
-126
130
-32766
32770
393307
16777222
51
2
3
1
2147483647
-2147483648
7
120
15
49
...
RET to PC < 0, finished!
*/
```

lbdex/input/ch_run_backend.cpp

```
#include "ch_nollid.h"

int main()
{
    bool pass = true;
    pass = test_nollid();

    return pass;
}

#include "ch_nollid.cpp"
```

lbdex/input/functions.sh

```
prologue() {
    if [ $argNum == 0 ]; then
        echo "usage: bash $sh_name cpu_type endian"
        echo "  cpu_type: cpu032I or cpu032II"
        echo "  endian: be (big endian, default) or le (little endian)"
        echo "for example:"
        echo "  bash build-slinker.sh cpu032I be"
        exit 1;
    fi
    if [ $arg1 != cpu032I ] && [ $arg1 != cpu032II ]; then
        echo "1st argument is cpu032I or cpu032II"
        exit 1
    fi

    OS=`uname -s`
    echo "OS =" ${OS}

    if [ "$OS" == "Linux" ]; then
        TOOLDIR=~/llvm/test/cmake_debug_build/bin
    else
        TOOLDIR=~/llvm/test/cmake_debug_build/Debug/bin
    fi

    CPU=$arg1
    echo "CPU =" "${CPU}"

    if [ "$arg2" != "" ] && [ $arg2 != le ] && [ $arg2 != be ]; then
        echo "2nd argument is be (big endian, default) or le (little endian)"
        exit 1
    fi
    if [ "$arg2" == "" ] || [ $arg2 == be ]; then
        endian=
    else
        endian=el
    fi
    echo "endian =" "${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
    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;
}
```

Ibdex/input/build-run_backend.sh

```
#!/usr/bin/env bash

source functions.sh

sh_name=build-run_backend.sh
argNum=$#
arg1=$1
arg2=$2

DEFFLAGS=""
if [ "$arg1" == cpu032II ] ; then
    DEFFLAGS="${DEFFLAGS} -DCPU032II"
fi
echo ${DEFFLAGS}

prologue;

# ch8_5.cpp just for compile build test only, without running on verilog.
clang ${DEFFLAGS} -target mips-unknown-linux-gnu -c ch8_5.cpp \
-emit-llvm -o ch8_5.bc
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=pic \
-filetype=obj ch8_5.bc -o ch8_5.cpu0.o

clang ${DEFFLAGS} -target mips-unknown-linux-gnu -c ch_run_backend.cpp \
-emit-llvm -o ch_run_backend.bc
${TOOLDIR}/llc -march=cpu0${endian} -mcpu=${CPU} -relocation-model=static \
```

```

-filetype=obj -enable-cpu0-tail-calls ch_run_backend.bc -o ch_run_backend.cpu0.o
${TOOLDIR}/llvm-objdump -d ch_run_backend.cpu0.o | tail -n +12| awk \
'{print /* $1 */ $2 " " $3 " " $4 " " $5 "\t/* $6\t" $7" " $8" \
" $9" " $10 "\t*/"}' > ../verilog/cpu0.hex

if [ "$arg2" == le ] ; 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 is arranged at the beginning of code and the main() of ch_run_backend.cpp is immediately after. 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 it 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.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$ bash build-run_backend.sh cpu032I be
JonathantekiiMac:input Jonathan$ cd ../verilog
JonathantekiiMac:verilog Jonathan$ ./cpu0Is
WARNING: cpu0Is.v:386: $readmemh(cpu0.hex): Not enough words in the file for the
taskInterrupt(001)
74
15
253
3
1
14
3
-126
130
-32766
32770
393307
16777222
51
2
2147483647
-2147483648
7
120
15
49
total cpu cycles = 41670
RET to PC < 0, finished!

```

```

JonathantekiiMac:input Jonathan$ bash build-run_backend.sh cpu032II be
JonathantekiiMac:input Jonathan$ cd ../verilog
JonathantekiiMac:verilog Jonathan$ ./cpu0II
...
total cpu cycles = 39790
RET to PC < 0, finished!

```

The “total cpu cycles” can be calculated in this verilog simualtor, and the backend compiler and CPU performance can be reviewed. Only the CPU cycles are counted, it exclude I/O cycles since I/O or display cycles time is unknown. As explained in chapter “Control flow statements”, cpu032II uses slt and beq has better performance than cmp and jeq in

cpu032I. You can trace the memory binary code and destination register changed at every instruction execution by the following change and get the result as below,

Ibdex/verilog/cpu0Is.v

```
'define TRACE
```

Ibdex/verilog/cpu0.v

```
...
`D = 1; // Trace register content at beginning

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 reads 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().

This chapter shows Verilog PC output by displays the integer value located at I/O memory mapped address directly. Since the cpu0.v machine is created by Verilog language, it suppose can run on real FPGA device. 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.

To generate cpu032II 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 le
```

./verilog/cpu0.config

```
1 /* 0: big endian, 1: little endian */
```

The following files test more features.

Ibdex/input/ch_noll2.h

```
#include "debug.h"
#include "boot.cpp"

#include "print.h"

int test_noll2();
```

Ibdex/input/ch_noll2.cpp

```
#include "print.cpp"

#include "ch9_4.cpp"

int test_noll2()
{
    bool pass = true;
    int a = 0;

    a = test_alloc(); // 31
    print_integer(a); // a = 1
    if (a != 31) pass = false;
    return pass;
}

/* result:
31
...
RET to PC < 0, finished!
*/
```

Ibdex/input/ch_run_backend2.cpp

```
#include "ch_noll2.h"

int main()
{
    bool pass = true;
    pass = test_noll2();

    return pass;
}

#include "ch_noll2.cpp"
```

Ibdex/input/build-run_backend2.sh

```
#!/usr/bin/env bash

source functions.sh

sh_name=build-run_backend.sh
argNum=$#
arg1=$1
```

```
arg2=$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 +12| awk \
'{print /* $1 */\t$2 " " $3 " " $4 " " $5 "\t/* $6"\t" $7" " $8" \
" $9" " $10 "\t*/"}' > ../verilog/cpu0.hex

if [ "$arg2" == le ] ; 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 le
...
JonathantekiiMac:input Jonathan$ cd ../verilog
JonathantekiiMac:verilog Jonathan$ ./cpu0II
...
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 extended from llvm-objdump driver at web: <http://jonathan2251.github.io/lbt/index.html>.

APPENDIX A: GETTING STARTED: INSTALLING LLVM AND THE CPU0 EXAMPLE CODE

- Setting Up Your Mac
 - Installing LLVM, Xcode and cmake
 - Create LLVM.xcodeproj by terminal cmake command
 - Build llvm by Xcode
 - Create LLVM.xcodeproj of supporting Cpu0 by terminal cmake command
 - Install Icarus Verilog tool on iMac
 - Install other tools on iMac
- Setting Up Your Linux Machine
 - Install LLVM 3.7 release build on Linux
 - Install Cpu0 debug build on Linux
 - Install Icarus Verilog tool on Linux
 - Install other tools on Linux

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>.

In this chapter, we will run through how to set up LLVM using if you are using Mac OS X or Linux. For information on using cmake to build LLVM, please refer to the “Building LLVM with CMake”¹ documentation for further information.

We will install two llvm directories in this chapter. One is the directory llvm/release/ 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 without clang and clang++.

14.1 Setting Up Your Mac

The Xcode include clang and llvm already. The following three sub-sections are needless. List them just for readers who like to build clang and llvm with cmake GUI interface.

14.1.1 Installing LLVM, Xcode and cmake

Todo

Fix centering for figure captions.

¹ <http://llvm.org/docs/CMake.html?highlight=cmake>

Please download LLVM latest release version 3.7 (llvm, clang) from the “LLVM Download Page”². Then extract them using `tar -xvf {llvm-3.7.0.src.tar.xz, cfe-3.7.0.src.tar.xz}`, and change the llvm source code root directory into src. After that, move the clang source code to src/tools/clang as shown as follows. The compiler-rt should not installed in iMac OS X 10.9 and Xcode 5.x. If you did as clang installation web document, it will has compiler error.

```
118-165-78-111:Downloads Jonathan$ tar -xvf cfe-3.7.0.src.tar.xz
118-165-78-111:Downloads Jonathan$ tar -xvf llvm-3.7.0.src.tar.xz
118-165-78-111:Downloads Jonathan$ mv llvm-3.7.0.src src
118-165-78-111:Downloads Jonathan$ mv cfe-3.7.0.src src/tools/clang
118-165-78-111:Downloads Jonathan$ pwd
/Users/Jonathan/Downloads
118-165-78-111:Downloads Jonathan$ ls
cfe-3.7.0.src.tar.xz      llvm-3.7.0.src.tar.xz
src
118-165-78-111:Downloads Jonathan$ ls src/tools/
CMakeLists.txt  clang      llvm-as      llvm-dis      llvm-mcmarkup
llvm-readobj   llvm-stub   LLVMBuild.txt  gold         llvm-bcanalyzer
llvm-dwarfdump llvm-nm     llvm-rtdyld   lto          Makefile
l1c            llvm-config  llvm-extract  llvm-objdump  llvm-shlib
macho-dump    bugpoint    lli          llvm-cov      llvm-link
llvm-prof     llvm-size   opt          bugpoint-passes  llvm-ar
llvm-diff     llvm-mc    llvm-ranlib   llvm-stress
```

Next, copy the LLVM source to `/Users/Jonathan/llvm/release/src` by executing the terminal command `cp -rf /Users/Jonathan/Downloads/src /Users/Jonathan/llvm/release/..`

Install Xcode from the Mac App Store. Then install cmake, which can be found here:³. Before installing cmake, ensure you can install applications you download from the Internet. Open *System Preferences* → *Security & Privacy*. Click the **lock** to make changes, and under “Allow applications downloaded from:” select the radio button next to “Anywhere.” See [Figure 14.1](#) below for an illustration. You may want to revert this setting after installing cmake.

Alternatively, you can mount the cmake .dmg image file you downloaded. Untar the latest cmake for Darwin, copy the cmake /Applications/ and set symbolic links as follows,

```
114-43-213-176:src Jonathan$ sudo rm -rf /usr/bin/cmake
114-43-213-176:src Jonathan$ sudo rm -rf /usr/bin/cmake-gui
114-43-213-176:src Jonathan$ sudo rm -rf /usr/bin/cmakexbuild
114-43-213-176:src Jonathan$ sudo ln -s /Applications/CMake.app/Contents/bin/
cmake /usr/bin/cmake
114-43-213-176:src Jonathan$ sudo ln -s /Applications/CMake.app/Contents/bin/
cmake-gui /usr/bin/cmake-gui
114-43-213-176:src Jonathan$ sudo ln -s /Applications/CMake.app/Contents/bin/
cmakexbuild /usr/bin/cmakexbuild
```

14.1.2 Create LLVM.xcodeproj by terminal cmake command

We installed llvm source code with clang on directory `/Users/Jonathan/llvm/release/` in last section. Now, will generate the LLVM.xcodeproj in this chapter.

```
114-43-213-176:release Jonathan$ pwd
/Users/Jonathan/llvm/release
114-43-213-176:release Jonathan$ mkdir cmake_release_build
114-43-213-176:release Jonathan$ cd cmake_release_build
114-43-213-176:cmake_release_build Jonathan$ cmake -DCMAKE_CXX_COMPILER=clang++
```

² <http://llvm.org/releases/download.html#3.7>

³ <http://www.cmake.org/cmake/resources/software.html>



Figure 14.1: Adjusting Mac OS X security settings to allow cmake installation.

```
-DCMAKE_C_COMPILER=clang -DCMAKE_CXX_FLAGS=-std=c++11 -DCMAKE_BUILD_TYPE=Debug
-G "Xcode" ../src
...
114-43-213-176:cmake_release_build Jonathan$ ls
... LLVM.xcodeproj
```

14.1.3 Build llvm by Xcode

Now, LLVM.xcodeproj is created. Open the cmake_debug_build/LLVM.xcodeproj by Xcode and click menu “**Product – Build**” as Figure 14.2.

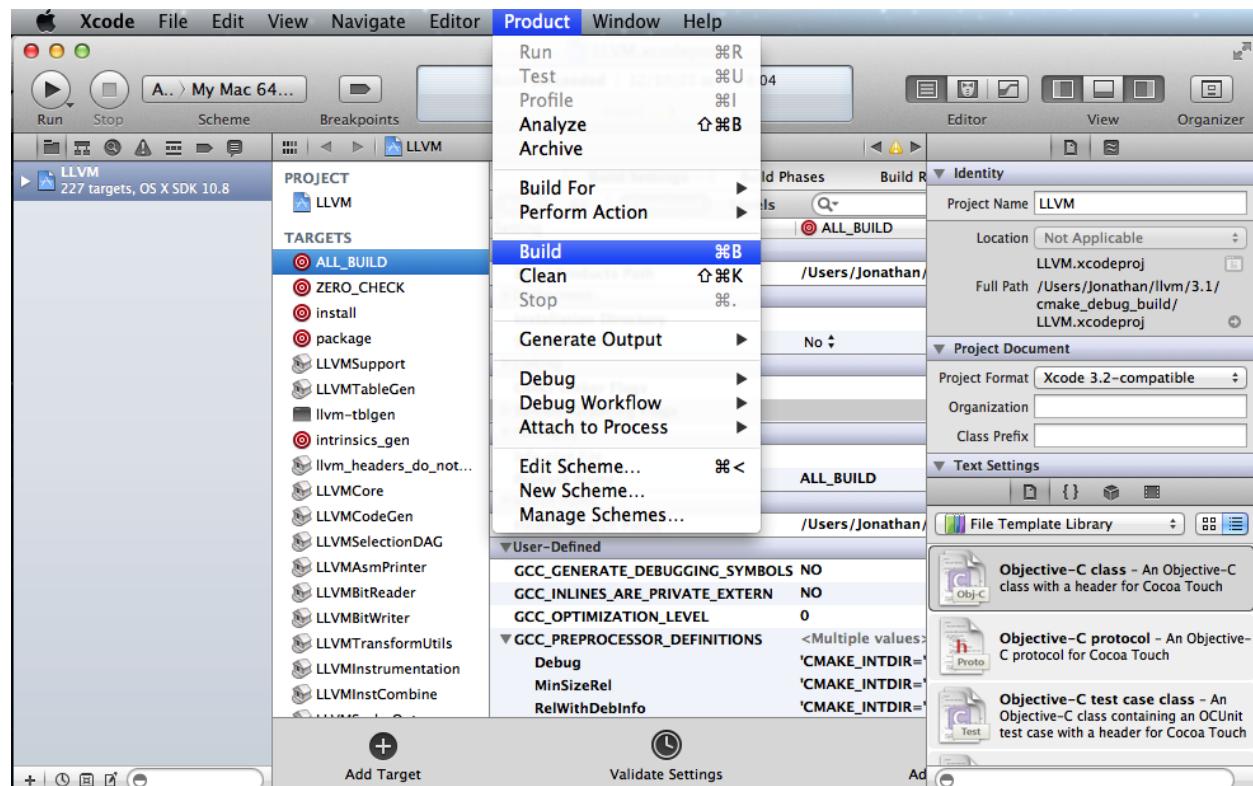


Figure 14.2: Click Build button to build LLVM.xcodeproj by Xcode

After few minutes of build, the clang, llc, llvm-as, ..., can be found in cmake_release_build/Debug/bin/ as follows.

```
118-165-78-111:cmake_release_build Jonathan$ cd Debug/bin/
118-165-78-111:bin Jonathan$ pwd
/Users/Jonathan/llvm/release/cmake_release_build/Debug/bin
118-165-78-111:bin Jonathan$ ls
...
clang
...
llc
...
llvm-as
...
```

To access those execution files, edit .profile (if you .profile not exists, please create file .profile), save .profile to

/Users/Jonathan/, and enable \$PATH by command source .profile as follows. Please add path /Applications//Xcode.app/Contents/Developer/usr/bin to .profile if you didn't add it after Xcode download.

```
118-165-65-128:~ Jonathan$ pwd  
/Users/Jonathan  
118-165-65-128:~ Jonathan$ cat .profile  
export PATH=$PATH:/Applications/Xcode.app/Contents/Developer/usr/bin:/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin:/Applications/Graphviz.app/Contents/MacOS:/Users/Jonathan/llvm/release/cmake_release_build/Debug/bin  
export WORKON_HOME=$HOME/.virtualenvs  
source /usr/local/bin/virtualenvwrapper.sh # where Homebrew places it  
export VIRTUALENVWRAPPER_VIRTUALENV_ARGS='--no-site-packages' # optional  
118-165-65-128:~ Jonathan$
```

14.1.4 Create LLVM.xcodeproj of supporting Cpu0 by terminal cmake command

We have installed llvm with clang on directory llvm/release/. Now, we want to install llvm with our cpu0 backend code on directory llvm/test/ in this section.

This book is on the process of merging into llvm trunk but not finished yet. The merged llvm trunk version on lbd git hub is LLVM 3.7 released version. The lbd of Cpu0 example code is also based on llvm 3.7. So, please install the llvm 3.7 debug version as the llvm release 3.7 installation, but without clang since the clang will waste time in build the Cpu0 backend tutorial code. Steps as follows,

The details of installing Cpu0 backend example code as follows,

```
118-165-78-111:llvm Jonathan$ mkdir test  
118-165-78-111:llvm Jonathan$ cd test  
118-165-78-111:test Jonathan$ pwd  
/Users/Jonathan/llvm/test  
118-165-78-111:test Jonathan$ cp /Users/Jonathan/Downloads/llvm-3.7.0.src.tar.xz .  
118-165-78-111:test Jonathan$ tar -xvf llvm-3.7.0.src.tar.xz  
118-165-78-111:test Jonathan$ mv llvm-3.7.0.src src  
118-165-78-111:test Jonathan$ cp /Users/Jonathan/Downloads/  
lbdex.tar.gz .  
118-165-78-111:test Jonathan$ tar -zxvf lbdex.tar.gz  
118-165-78-111:test Jonathan$ cp -rf lbdex/src/modify/src/* src/.  
118-165-78-111:test Jonathan$ grep -R "Cpu0" src/include  
...  
src/include/llvm/MC/MCEExpr.h:      VK_Cpu0_GPREL,  
src/include/llvm/MC/MCEExpr.h:      VK_Cpu0_GOT_CALL,  
src/include/llvm/MC/MCEExpr.h:      VK_Cpu0_GOT16,  
src/include/llvm/MC/MCEExpr.h:      VK_Cpu0_GOT,  
src/include/llvm/MC/MCEExpr.h:      VK_Cpu0_ABS_HI,  
src/include/llvm/MC/MCEExpr.h:      VK_Cpu0_ABS_LO,  
...  
src/lib/MC/MCEExpr.cpp:  case VK_Cpu0_GOT_PAGE: return "GOT_PAGE";  
src/lib/MC/MCEExpr.cpp:  case VK_Cpu0_GOT_OFST: return "GOT_OFST";  
src/lib/Target/LLVMBuild.txt:subdirectories = ARM CellSPU CppBackend Hexagon  
MBLaze MSP430 NVPTX Mips Cpu0 PowerPC Sparc X86 XCore  
118-165-78-111:test Jonathan$
```

Next, please copy Cpu0 example code according the following commands,

```
118-165-78-111:test Jonathan$ pwd  
/Users/Jonathan/llvm/test  
118-165-78-111:test Jonathan$ cp -rf lbdex/Cpu0 src/lib/Target/.
```

```
118-165-78-111:test Jonathan$ ls src/lib/Target/Cpu0
CMakeLists.txt          Cpu0InstrInfo.td      Cpu0TargetMachine.cpp  TargetInfo
Cpu0.h                  Cpu0RegisterInfo.td  ExampleCode        readme
Cpu0.td                 Cpu0Schedule.td     LLVMBuild.txt
Cpu0InstrFormats.td    Cpu0Subtarget.h    MCTargetDesc
118-165-80-55:Cpu0 Jonathan$
```

Now, it's ready for building llvm/test/src code by command `cmake` as follows.

```
118-165-78-111:test Jonathan$ pwd
/Users/Jonathan/llvm/test
118-165-78-111:test Jonathan$ ls
src
118-165-78-111:test Jonathan$ mkdir cmake_debug_build
118-165-78-111:test Jonathan$ cd cmake_debug_build/
118-165-78-111:cmake_debug_build Jonathan$ cmake -DCMAKE_CXX_COMPILER=clang++
-DCMAKE_C_COMPILER=clang -DCMAKE_BUILD_TYPE=Debug -DLLVM_TARGETS_TO_BUILD=Cpu0
-G "Xcode" ../src/
-- The C compiler identification is Clang 5.0
-- The CXX compiler identification is Clang 5.0
-- Check for working C compiler using: Xcode
...
-- Targeting Cpu0
...
-- Performing Test SUPPORTS_GLINE_TABLES_ONLY_FLAG
-- Performing Test SUPPORTS_GLINE_TABLES_ONLY_FLAG - Success
-- Performing Test SUPPORTS_NO_C99_EXTENSIONS_FLAG
-- Performing Test SUPPORTS_NO_C99_EXTENSIONS_FLAG - Success
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/Jonathan/llvm/test/cmake_debug_build
118-165-78-111:cmake_debug_build Jonathan$
```

Now, you can build this llvm build with Cpu0 backend only by Xcode.

On iMac, tt also can do `cmake` and `make` with '`cmake -G "Unix Makefiles"`' same as the Linux as the following section.

Since Xcode use clang compiler and lldb instead of gcc and gdb, we can run lldb debug as follows,

```
118-165-65-128:InputFiles Jonathan$ pwd
/Users/Jonathan/lbdex/InputFiles
118-165-65-128:InputFiles Jonathan$ clang -c ch3.cpp -emit-llvm -o ch3.bc
118-165-65-128:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/Debug/bin/llc -march=mips -relocation-model=pic -filetype=asm
ch3.bc -o ch3.mips.s
118-165-65-128:InputFiles Jonathan$ lldb -- /Users/Jonathan/llvm/test/
cmake_debug_build/Debug/bin/llc -march=mips -relocation-model=pic -filetype=
asm ch3.bc -o ch3.mips.s
Current executable set to '/Users/Jonathan/llvm/test/cmake_debug_build/bin/
Debug/llc' (x86_64).
(lldb) b MipsTargetInfo.cpp:19
breakpoint set --file 'MipsTargetInfo.cpp' --line 19
Breakpoint created: 1: file ='MipsTargetInfo.cpp', line = 19, locations = 1
(lldb) run
Process 6058 launched: '/Users/Jonathan/llvm/test/cmake_debug_build/Debug/bin/
llc' (x86_64)
Process 6058 stopped
* thread #1: tid = 0x1c03, 0x000000010077f231 llc'LLVMInitializeMipsTargetInfo
```

```
+ 33 at MipsTargetInfo.cpp:20, stop reason = breakpoint 1.1
frame #0: 0x000000010077f231 llc`LLVMInitializeMipsTargetInfo + 33 at
MipsTargetInfo.cpp:20
 17
 18     extern "C" void LLVMInitializeMipsTargetInfo() {
 19         RegisterTarget<Triple::mips,
-> 20             /*HasJIT=*/true> X(TheMipsTarget, "mips", "Mips");
 21
 22         RegisterTarget<Triple::mipsel,
 23             /*HasJIT=*/true> Y(TheMipselTarget, "mipsel", "Mipsel");
(lldb) n
Process 6058 stopped
* thread #1: tid = 0x1c03, 0x000000010077f24f llc`LLVMInitializeMipsTargetInfo
+ 63 at MipsTargetInfo.cpp:23, stop reason = step over
frame #0: 0x000000010077f24f llc`LLVMInitializeMipsTargetInfo + 63 at
MipsTargetInfo.cpp:23
 20             /*HasJIT=*/true> X(TheMipsTarget, "mips", "Mips");
 21
 22         RegisterTarget<Triple::mipsel,
-> 23             /*HasJIT=*/true> Y(TheMipselTarget, "mipsel", "Mipsel");
 24
 25         RegisterTarget<Triple::mips64,
 26             /*HasJIT=*/false> A(TheMips64Target, "mips64", "Mips64
 [experimental]");
(lldb) print X
(llvm::RegisterTarget<llvm::Triple::ArchType, true>) $0 = {}
(lldb) quit
118-165-65-128:InputFiles Jonathan$
```

About the lldb debug command, please reference ⁴ or lldb portal ⁵.

14.1.5 Install Icarus Verilog tool on iMac

Install Icarus Verilog tool by command brew install icarus-verilog as follows,

```
JonathantekiiMac:~ Jonathan$ 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
```

14.1.6 Install other tools on iMac

These tools mentioned in this section is for coding and debug. You can work even without these tools. Files compare tools Kdiff3 came from web site ⁶. FileMerge is a part of Xcode, you can type FileMerge in Finder – Applications as Figure 14.3 and drag it into the Dock as Figure 14.4.

⁴ <http://lldb.llvm.org/lldb-gdb.html>

⁵ <http://lldb.llvm.org/>

⁶ <http://kdiff3.sourceforge.net>

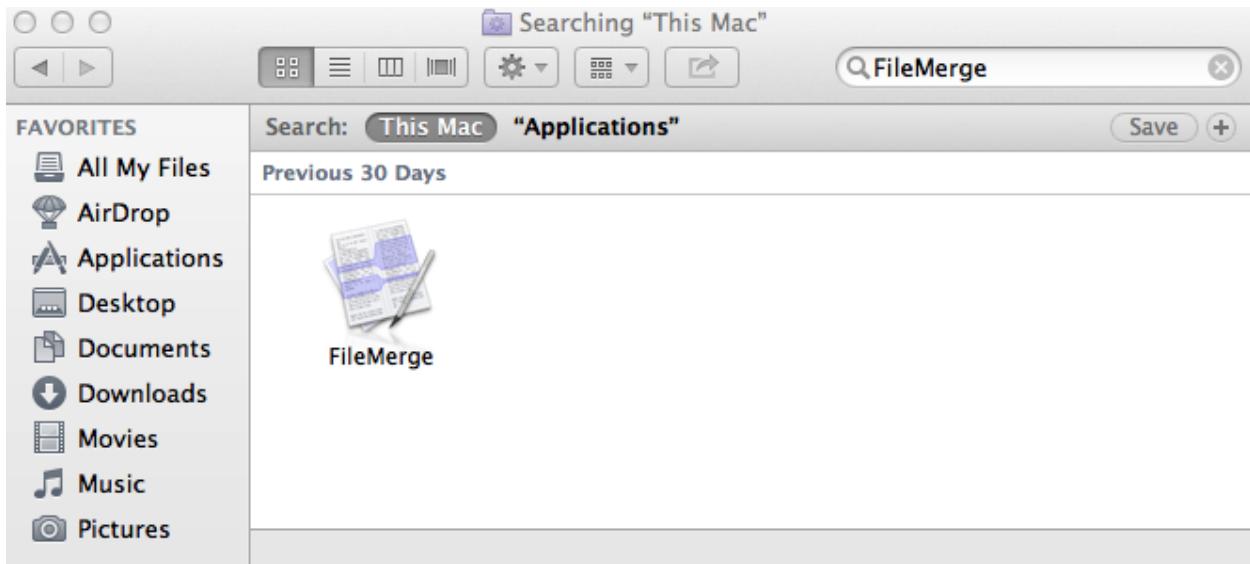


Figure 14.3: Type FileMerge in Finder – Applications



Figure 14.4: Drag FileMege into the Dock

Download tool Graphviz for display llvm IR nodes in debugging,⁷. We choose mountainlion as Figure 14.5 since our iMac is Mountain Lion.

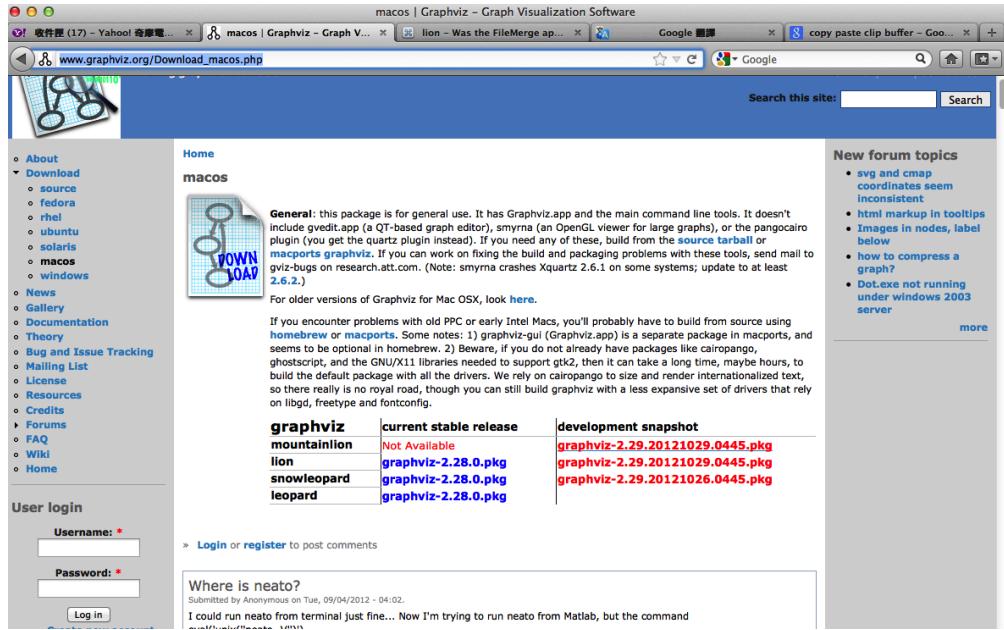


Figure 14.5: Download graphviz for llvm IR node display

After install Graphviz, please set the path to .profile. For example, we install the Graphviz in directory /Applications/Graphviz.app/Contents/MacOS/, so add this path to /User/Jonathan/.profile as follows,

```
118-165-12-177:InputFiles Jonathan$ cat /Users/Jonathan/.profile
export PATH=$PATH:/Applications/Xcode.app/Contents/Developer/usr/bin:
/Applications/Graphviz.app/Contents/MacOS:/Users/Jonathan/llvm/release/
cmake_release_build/Debug/bin
```

The Graphviz information for llvm is at section “SelectionDAG Instruction Selection Process” of “The LLVM Target-Independent Code Generator” here⁸ and at section “Viewing graphs while debugging code” of “LLVM Programmer’s Manual” here⁹. TextWrangler is for edit file with line number display and dump binary file like the obj file, *.o, that will be generated in chapter of Generating object files if you havn’t gobjdump available. You can download from App Store. To dump binary file, first, open the binary file, next, select menu “File – Hex Front Document” as Figure 14.6. Then select “Front document’s file” as Figure 14.7.

Install binutils by command brew install binutils as follows,

```
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/local
==> 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
```

⁷ http://www.graphviz.org/Download_macos.php

⁸ <http://llvm.org/docs/CodeGenerator.html#selectiondag-instruction-selection-process>

⁹ <http://llvm.org/docs/ProgrammersManual.html#viewing-graphs-while-debugging-code>

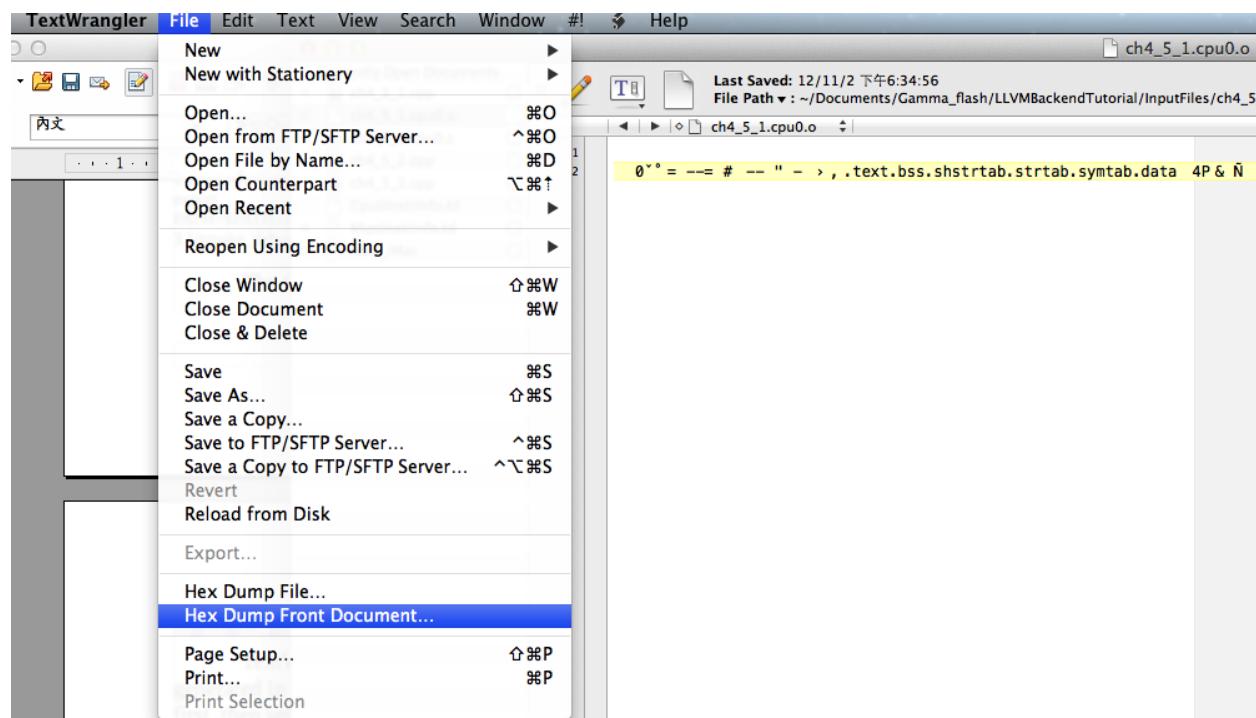


Figure 14.6: Select Hex Dump menu

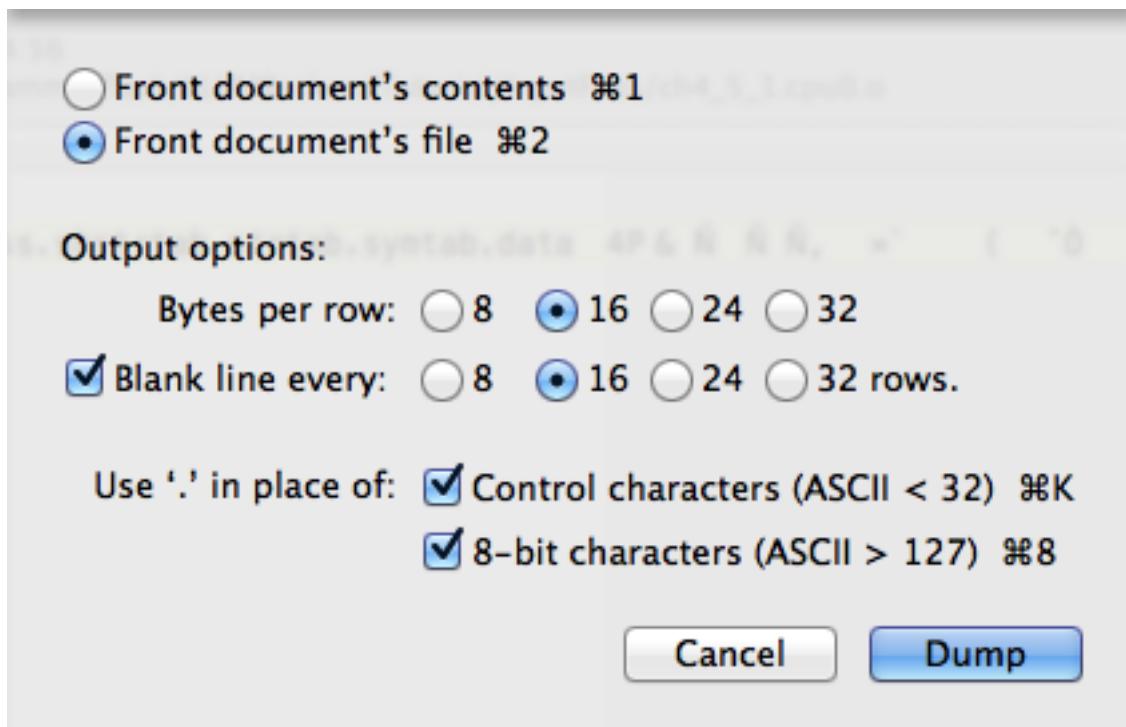


Figure 14.7: Select Front document's file in TextWrangler

```
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
```

14.2 Setting Up Your Linux Machine

14.2.1 Install LLVM 3.7 release build on Linux

First, install the llvm release build by,

1. Untar llvm source, rename llvm source with src.
2. Untar clang and move it src/tools/clang.

Next, build with cmake command, `cmake -DCMAKE_BUILD_TYPE=Release -DCLANG_BUILD_EXAMPLES=ON -DLLVM_BUILD_EXAMPLES=ON -G "Unix Makefiles" ..src/`, as follows.

```
[Gamma@localhost cmake_release_build]$ pwd
/home/cschen/llvm/release/cmake_release_build
[Gamma@localhost cmake_release_build]$ cmake -DCMAKE_BUILD_TYPE=Release
-DCLANG_BUILD_EXAMPLES=ON -DLLVM_BUILD_EXAMPLES=ON -G "Unix Makefiles" ..src/
-- The C compiler identification is GNU 4.8.2
...
-- Constructing LLVMBuild project information
...
-- Targeting XCore
-- Clang version: 3.7
-- Found Subversion: /usr/bin/svn (found version "1.7.6")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/cschen/llvm/release/cmake_release_build
```

After cmake, run command make, then you can get clang, llc, llvm-as, ..., in `cmake_release_build/bin/` after a few tens minutes of build. To speed up make process via SMP power, please check your core numbers by the following command then do make the next.

```
[Gamma@localhost cmake_release_build]$ cat /proc/cpuinfo | grep processor | wc -l
8
[Gamma@localhost cmake_release_build]$ make -j8 -l8
```

Next, edit `/home/Gamma/.bash_profile` with adding `/home/cschen/llvm/release/cmake_release_build/bin` to PATH to enable the clang, llc, ..., command search path, as follows,

```
[Gamma@localhost ~]$ pwd
/home/Gamma
[Gamma@localhost ~]$ cat .bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs
```

```
PATH=$PATH:/usr/local/sphinx/bin:~/llvm/release/cmake_release_build/bin:  
/opt/mips_linux_toolchain_clang/mips_linux_toolchain/bin:$HOME/.local/bin:  
$HOME/bin  
  
export PATH  
[Gamma@localhost ~]$ source .bash_profile  
[Gamma@localhost ~]$ $PATH  
bash: /usr/lib64/qt-3.3/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:  
/usr/sbin:/usr/local/sphinx/bin:/home/Gamma/.local/bin:/home/Gamma/bin:  
/usr/local/sphinx/bin:/home/cschen/llvm/release/cmake_release_build/bin
```

14.2.2 Install Cpu0 debug build on Linux

This book is on the process of merging into llvm trunk but not finished yet. The merged llvm trunk version on lbd git hub is LLVM 3.7 released version. The Cpu0 example code is also based on llvm 3.7. So, please install the llvm 3.7 debug version as the llvm release 3.7 installation, but without clang since the clang will waste time in build the Cpu0 backend tutorial code. Steps as follows,

The details of installing Cpu0 backend example code according the following list steps, and the corresponding commands shown as below,

1. Enter ~/llvm/test/ and get Cpu0 example code as well as the llvm 3.7.
2. Make dir Cpu0 in src/lib/Target and download example code.
3. Update llvm modified source files to support cpu0 by command `cp -rf lbdex/src/modify/src/* src/..`.
4. Check step 3 is effective by command `, grep -R "Cpu0" . | more`. We add the Cpu0 backend support, so check with grep.
5. Copy Cpu0 bakend code by command, `cp -rf lbdex/Cpu0 src/lib/Target/..`
6. Remove clang from ~/llvm/test/src/tools/clang, and mkdir test/cmake_debug_build. Otherwise you will waste extra time for command make in Cpu0 example code build with clang.

```
[Gamma@localhost llvm]$ mkdir test  
[Gamma@localhost llvm]$ cd test  
[Gamma@localhost test]$ pwd  
/home/cschen/llvm/test  
[Gamma@localhost test]$ cp /home/Gamma/Downloads/llvm-3.7.0.src.tar.xz .  
[Gamma@localhost test]$ tar -xvf llvm-3.7.0.src.tar.xz  
[Gamma@localhost test]$ mv llvm-3.7.0.src src  
[Gamma@localhost test]$ cp /Users/Jonathan/Downloads/  
lbdex.tar.gz .  
[Gamma@localhost test]$ tar -zxf lbdex.tar.gz  
...  
[Gamma@localhost test]$ cp -rf lbdex/src/modify/src/* src/.  
[Gamma@localhost test]$ grep -R "cpu0" src/include  
src/include//llvm/ADT/Triple.h:      cpu0,      // For Tutorial Backend Cpu0  
src/include//llvm/MC/MCEExpr.h:      VK_Cpu0_GPREL,  
src/include//llvm/MC/MCEExpr.h:      VK_Cpu0_GOT_CALL,  
...  
[Gamma@localhost test]$ cp -rf lbdex/Cpu0 src/lib/Target/.  
[Gamma@localhost test]$ ls src/lib/Target/Cpu0  
AsmParser  
CMakeLists.txt  
Cpu0RegisterInfoGPROutForAsm.td  
Cpu0RegisterInfoGPROutForOther.td  
...
```

Now, create directory cmake_debug_build and do cmake just like build the llvm/release, except we do Debug build with Cpu0 backend only, and use clang as our compiler instead, as follows,

```
[Gamma@localhost test]$ pwd
/home/cschen/llvm/test
[Gamma@localhost test]$ mkdir cmake_debug_build
[Gamma@localhost test]$ cd cmake_debug_build/
[Gamma@localhost cmake_debug_build]$ cmake -DCMAKE_CXX_COMPILER=clang++
-DCMAKE_C_COMPILER=clang -DCMAKE_BUILD_TYPE=Debug -DLLVM_TARGETS_TO_BUILD=Cpu0
-G "Unix Makefiles" ../src/
-- The C compiler identification is Clang 3.7.0
-- The CXX compiler identification is Clang 3.7.0
-- Check for working C compiler: /home/cschen/llvm/release/cmake_release_build/bin/
clang
-- Check for working C compiler: /home/cschen/llvm/release/cmake_release_build/bin/
clang
-- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /home/cschen/llvm/release/cmake_release_build/
bin/clang++
-- Check for working CXX compiler: /home/cschen/llvm/release/cmake_release_build/
bin/clang++
-- works
...
-- Targeting Mips
-- Targeting Cpu0
...
-- Configuring done
-- Generating done
-- Build files have been written to: /home/cschen/llvm/test/cmake_debug
_build
[Gamma@localhost cmake_debug_build]$
```

Then do make as follows,

```
[Gamma@localhost cmake_debug_build]$ make -j8 -l8
Scanning dependencies of target LLVMSupport
[ 0%] Building CXX object lib/Support/CMakeFiles/LLVMSupport.dir/APFloat.cpp.o
[ 0%] Building CXX object lib/Support/CMakeFiles/LLVMSupport.dir/APIInt.cpp.o
[ 0%] Building CXX object lib/Support/CMakeFiles/LLVMSupport.dir/APSInt.cpp.o
[ 0%] Building CXX object lib/Support/CMakeFiles/LLVMSupport.dir/Allocator.cpp.o
[ 1%] Building CXX object lib/Support/CMakeFiles/LLVMSupport.dir/BlockFrequency.
cpp.o ...
Linking CXX static library ../../lib/libgtest.a
[100%] Built target gtest
Scanning dependencies of target gtest_main
[100%] Building CXX object utils/unittest/CMakeFiles/gtest_main.dir/UnitTestMain
/
TestMain.cpp.o Linking CXX static library ../../lib/libgtest_main.a
[100%] Built target gtest_main
[Gamma@localhost cmake_debug_build]$
```

Since clang invoke the ~/llvm/cmake_release_build/bin/clang where is built by cmake -DCMAKE_BUILD_TYPE=Release, it is 4 times speed up more than make (default use 1 thread only). But if you make with debug clang build, it won't speed up too much.

Now, we are ready for the cpu0 backend development. We can run gdb debug as follows. If your setting has anything about gdb errors, please follow the errors indication (maybe need to download gdb again). Finally, try gdb as follows.

```
[Gamma@localhost InputFiles]$ pwd
~/llvm/test/src/lib/Target/Cpu0/ExampleCode/
1bdex/InputFiles
[Gamma@localhost InputFiles]$ clang -c ch3.cpp -emit-llvm -o ch3.bc
[Gamma@localhost InputFiles]$ gdb -args ~/llvm/test/
cmake_debug_build/bin/llc -march=cpu0 -relocation-model=pic -filetype=obj
ch3.bc -o ch3.cpu0.o
GNU gdb (GDB) Fedora (7.4.50.20120120-50.fc17)
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/cschen/llvm/test/cmake_debug_build/bin/llc.
..done.
(gdb) break MipsTargetInfo.cpp:19
Breakpoint 1 at 0xd54441: file /home/cschen/llvm/test/src/lib/Target/
Mips/TargetInfo/MipsTargetInfo.cpp, line 19.
(gdb) run
Starting program: /home/cschen/llvm/test/cmake_debug_build/bin/llc
-march=cpu0 -relocation-model=pic -filetype=obj ch3.bc -o ch3.cpu0.o
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, LLVMInitializeMipsTargetInfo ()
  at /home/cschen/llvm/test/src/lib/Target/Mips/TargetInfo/MipsTargetInfo.cpp:20
20      /*HasJIT=*/true> X(TheMipsTarget, "mips", "Mips");
(gdb) next
23      /*HasJIT=*/true> Y(TheMipselTarget, "mipsel", "Mipsel");
(gdb) print X
$1 = {<No data fields>}
(gdb) quit
A debugging session is active.

Inferior 1 [process 10165] will be killed.

Quit anyway? (y or n) y
[Gamma@localhost InputFiles]$
```

14.2.3 Install Icarus Verilog tool on Linux

Download the snapshot version of Icarus Verilog tool from web site, <ftp://icarus.com/pub/eda/verilog/snapshots> or go to <http://iverilog.icarus.com/> and click snapshot version link. Follow the INSTALL file guide to install it.

14.2.4 Install other tools on Linux

Download Graphviz from ¹⁰ according your Linux distribution. Files compare tools Kdiff3 came from web site [[kdiff3](#)].

¹⁰ <http://www.graphviz.org/Download.php>

APPENDIX B: CPU0 DOCUMENT AND TEST

- Cpu0 document
 - Install sphinx
 - Generate Cpu0 document
 - About Cpu0 document
- Cpu0 Regression Test

15.1 Cpu0 document

This section illustrates how to generate Cpu0 backend document.

15.1.1 Install sphinx

LLVM and this book use sphinx to generate html document. This book uses Sphix to generate pdf and epub format of document further. Sphinx uses restructured text format here [1](#) [2](#) [3](#). The installation of Sphinx reference [4](#).

On iMac or linux you can install as follows,

```
sudo easy_install sphinx
```

Above installaton can generate html document but not for pdf. To support pdf/latex document generated as follows,

On iMac, install MacTex.pkg from here [5](#).

On Linux, install texlive as follows,

```
sudo apt-get install texlive texlive-latex-extra
```

or

```
sudo yum install texlive texlive-latex-extra
```

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,

¹ <http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html>

² <http://docutils.sourceforge.net/docs/ref/rst/directives.html>

³ <http://docutils.sourceforge.net/rst.html>

⁴ <http://docs.geoserver.org/latest/en/docguide/install.html>

⁵ <http://www.tug.org/mactex/>

```
[root@localhost BackendTutorial]$ yum list pdfjam
Loaded plugins: langpacks, presto, refresh-packagekit
Installed Packages
pdfjam.noarch           2.08-3.fc17          @fedora
[root@localhost BackendTutorial]$
```

On Fedora 18, the error as follows,

```
[root@localhost BackendTutorial]$ make latexpdf
...
LaTeX Error: File 'titlesec.sty' not found
```

Install all texlive-* (full) as follows,

```
[root@localhost BackendTutorial]$ yum install texlive-*
```

15.1.2 Generate Cpu0 document

Cpu0 example code is added step by step and 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:BackendTutorial Jonathan$ pwd
/home/Jonathan/test/lbd/docs/BackendTutorial
118-165-12-177:BackendTutorial Jonathan$ make genexample
...
118-165-12-177:BackendTutorial 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
```

Then, this book html/pdf can be generated by the following commands.

```
118-165-12-177:BackendTutorial Jonathan$ pwd
/home/Jonathan/test/lbd/docs/BackendTutorial
118-165-12-177:BackendTutorial Jonathan$ make html
...
118-165-12-177:BackendTutorial Jonathan$ make latexpdf
...
```

Beside chapters example code, above html and pdf of Cpu0 documents also include files *.ll and *.s in BackendTutorial/output. So, when change Cpu0 code or porting to new llvm version, please do the following before generate documents.

```
118-165-12-177:BackendTutorial Jonathan$ pwd
/home/Jonathan/test/lbd/docs/BackendTutorial
118-165-12-177:BackendTutorial Jonathan$ bash gen-ch12-output.sh
118-165-12-177:BackendTutorial Jonathan$ git add output/.
118-165-12-177:BackendTutorial Jonathan$ git commit -m "update output"
118-165-12-177:BackendTutorial Jonathan$ git push
```

15.1.3 About Cpu0 document

Since llvm have a new release version around every 6 months and every name of file, function, class, variable, ..., etc, can be changed, the Cpu0 document maintains is an effort because it add the code step by step, chapter by chapter. In order to make the document as correct and easy to 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" is refered by document files *.rst.

In lbdex/src/modify/src, the *.rst refer the code by copy them directly. Most of references exist in llvmstructure.rst and elf.rst.

15.2 Cpu0 Regression Test

The last chapter can verify code by Verilog simulator without including global variable and some data which are put beyond stack. The chapter lld in web <https://github.com/Jonathan2251/lbt.git> will include llvm ELF linker implementation and can verify those test items which include global variable access. Beside these, LLVM has its test cases (regression test) for each backend to verify the code generation⁶. 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/src/test/CodeGen/Cpu0.

Then run as follows for single test case and the whole test cases on iMac.

```
1-160-130-77:Cpu0 Jonathan$ pwd  
/Users/Jonathan/llvm/test/src/test/CodeGen/Cpu0  
1-160-130-77:Cpu0 Jonathan$ ~/llvm/test/cmake_debug_build/Debug/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/cmake_debug_build/Debug/bin/llvm-lit .  
...  
PASS: LLVM :: CodeGen/Cpu0/zeroreg.ll (126 of 127)  
PASS: LLVM :: CodeGen/Cpu0/tailcall.ll (127 of 127)  
...
```

Run as follows for single test case and the whole test cases on Linux.

```
[Gamma@localhost Cpu0]$ pwd  
/home/cschen/llvm/test/src/test/CodeGen/Cpu0  
[Gamma@localhost Cpu0]$ ~/llvm/test/cmake_debug_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  
[Gamma@localhost Cpu0]$ ~/llvm/test/cmake_debug_build/bin/llvm-lit .  
...  
PASS: LLVM :: CodeGen/Cpu0/zeroreg.ll (126 of 127)  
PASS: LLVM :: CodeGen/Cpu0/tailcall.ll (127 of 127)  
...
```

In order to understand which chapter the regression test item test for, listing the chapters first as follows,

⁶ <http://llvm.org/docs/TestingGuide.html>

Table 15.1: 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

Then the regression test items for Cpu0 list as follows,

Table 15.2: Regression test items for Cpu0

File	v:pass x:fail	test ir, -> output asm	chapter
addc.ll	v	64-bit add	7
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
cmove.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.ll	v	c++ exception handling	12
ex2.ll	v	c++ exception handling	12
fneg.ll	v	verify Cpu0 don't uses hard float instruction	7
fp-spill-reload.ll	v	-> st \$fp, ld \$fp	9

Continued on next page

Table 15.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
global-address.ll	v	global address, global variable	6
global-pointer.ll	v	global register load and restore, -> .cupload, .cprestore	9
gprestore.ll	v	global register restore, -> .cprestore	9
helloworld.ll	v	global register load and restore, -> .cupload, .cprestore	9
hf16_1.ll	v	function call in PIC, -> ld, ja lr	9
i32k.ll	v	argument of constant int passing in register	9
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
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
largefr1.ll	v	large frame	3
largeimm1.ll	v	large immediate (32-bit, not 16-bit), -> lui, addiu	3
largeimprinting.ll	v	large imm passing in register	3
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
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
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

Continued on next page

Table 15.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
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
repmat-immed-load.ll	v	immediate load	3
remul.ll	v	urem, -> div, mfhi	4
return-vector.ll	v	return vector, -> ld ld ..., st st ...	3
return-vector-float4.ll	v	return vector, -> lui lui ...	3
rotate.ll	v	rotl, rotr, -> rolv, rol, rorv	4
sb1.ll	v	store i8, sb	7
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

Continued on next page

Table 15.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
setle.ll	v	•	8
setlt.ll	v	•	8
setltk.ll	v	•	8
setne.ll	v	•	8
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, ash, -> 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
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-models.ll	v	ir external/internal thread_local global	12

Continued on next page

Table 15.2 – continued from previous page

File	v:pass x:fail	test ir, -> output asm	chapter
uitofp.ll	v	integer2float, uitofp, -> jsub __floatunisf	9
uli.ll	v	unalignment init, -> sb sb ...	6
unalignedload.ll	v	unalignment init, -> sb sb ...	6
weak.ll	v	extern_weak function, -> .weak	9
xor1.ll	v	xor, -> xor	4
zeroreg.ll	v	check register \$zero	4

The Cpu0 regression test items copied from Mips. Cpu0 doesn't support the following test items at this point.

Table 15.3: Not supported regression test items for Cpu0

File	v:pass x:fail	comment
blockaddr.ll	x	Not C/C++ language
2008-08-08-bswap.ll	x	Not support llvm.bswap. The c++ STL swap didn't use it. ?
2010-04-07- DbgValueOtherTargets.ll	x	Mips cannot pass either ?
eh-dwraf-cfa.ll	x	
eh-return32.ll	x	What is the IR of llvm.eh.return.i32? (not from c++ try & catch).
eh-return64.ll	x	Same as above.
fastcc.ll	x	Fast Call. No need in Cpu0
frame-address.ll	x	Not support @llvm.frameaddress intrinsic function.
init-array.ll	x	Don't know the purpose.
longbranch.ll	x	Not support and not intend to support.
tls-alias.ll	x	Mips 3.5 cannot pass either.
vector-setcc.ll	x	Not support vector.

These test items of supported and not supported are in lbdex/regression-test/Cpu0 and lbdex/regression-test/not-support, respectively, which can be got from untar lbdex.tar.gz.

CHAPTER
SIXTEEN

TODO LIST

Todo

Fix centering for figure captions.

(The *original entry* is located in /home/cschen/test/4/lbd/source/install.rst, line 35.)

CHAPTER
SEVENTEEN

BOOK EXAMPLE CODE

The example code lbdex.tar.gz is available in:

<http://jonathan2251.github.io/lbd/lbdex.tar.gz>

**CHAPTER
EIGHTEEN**

ALTERNATE FORMATS

The book is also available in the following formats:

CHAPTER
NINETEEN

PRESENTATION FILES