

---

# **Tutorial: Creating an LLVM Backend for the Cpu0 Architecture**

***Release 3.3.7***

**Chen Chung-Shu**    `gamma_chen@yahoo.com.tw`  
**Anoushe Jamshidi**    `ajamshidi@gmail.com`

November 18, 2013



# CONTENTS

<b>1</b>	<b>About</b>	<b>3</b>
1.1	Authors . . . . .	3
1.2	Contributors . . . . .	3
1.3	Acknowledgments . . . . .	3
1.4	Support . . . . .	3
1.5	Revision history . . . . .	4
1.6	Licensing . . . . .	5
1.7	Preface . . . . .	5
1.8	Prerequisites . . . . .	6
1.9	Outline of Chapters . . . . .	6
<b>2</b>	<b>Cpu0 Instruction Set and LLVM Target Description</b>	<b>9</b>
2.1	Cpu0 Processor Architecture Details . . . . .	9
2.2	LLVM Structure . . . . .	14
2.3	.td: LLVM's Target Description Files . . . . .	16
2.4	Creating the Initial Cpu0 .td Files . . . . .	16
2.5	Write cmake file . . . . .	21
2.6	Target Registration . . . . .	21
2.7	Build libraries and td . . . . .	22
<b>3</b>	<b>Backend structure</b>	<b>25</b>
3.1	TargetMachine structure . . . . .	25
3.2	Add AsmPrinter . . . . .	32
3.3	LLVM Code Generation Sequence . . . . .	35
3.4	DAG (Directed Acyclic Graph) . . . . .	39
3.5	Instruction Selection . . . . .	40
3.6	Add Cpu0DAGToDAGISel class . . . . .	43
3.7	Handle return register lr . . . . .	45
3.8	Add Prologue/Epilogue functions . . . . .	46
3.9	Summary of this Chapter . . . . .	57
<b>4</b>	<b>Arithmetic and logic lsupport</b>	<b>59</b>
4.1	Arithmetic . . . . .	59
4.2	Logic . . . . .	85
4.3	Summary . . . . .	93
<b>5</b>	<b>Generating object files</b>	<b>95</b>
5.1	Translate into obj file . . . . .	95
5.2	Backend Target Registration Structure . . . . .	96

<b>6 Global variables</b>	<b>109</b>
6.1 Global variable . . . . .	109
<b>7 Other data type</b>	<b>133</b>
7.1 Local variable pointer . . . . .	133
7.2 char, short int and bool . . . . .	134
7.3 long long . . . . .	138
7.4 float and double . . . . .	141
7.5 Array and struct support . . . . .	141
<b>8 Control flow statements</b>	<b>149</b>
8.1 Control flow statement . . . . .	149
8.2 RISC CPU knowledge . . . . .	162
<b>9 Function call</b>	<b>163</b>
9.1 Mips stack frame . . . . .	163
9.2 Load incoming arguments from stack frame . . . . .	168
9.3 Store outgoing arguments to stack frame . . . . .	176
9.4 Fix issues . . . . .	181
9.5 Support features . . . . .	194
9.6 Summary of this chapter . . . . .	217
<b>10 ELF Support</b>	<b>221</b>
10.1 ELF format . . . . .	221
10.2 ELF header and Section header table . . . . .	223
10.3 Relocation Record . . . . .	224
10.4 Cpu0 ELF related files . . . . .	227
10.5 llvm-objdump . . . . .	227
10.6 Dynamic link . . . . .	234
<b>11 Run backend</b>	<b>245</b>
11.1 AsmParser support . . . . .	245
11.2 Verilog of CPU0 . . . . .	251
11.3 Run program on CPU0 machine . . . . .	259
<b>12 Backend Optimization</b>	<b>267</b>
12.1 Cpu0 backend Optimization: Remove useless JMP . . . . .	267
12.2 Cpu0 Optimization: Redesign instruction sets . . . . .	270
<b>13 LLD for Cpu0</b>	<b>285</b>
13.1 Install lld . . . . .	285
13.2 Cpu0 lld souce code . . . . .	287
13.3 ELF to Hex . . . . .	312
13.4 Static linker . . . . .	327
13.5 Dynamic linker . . . . .	344
13.6 Summary . . . . .	357
<b>14 Appendix A: Getting Started: Installing LLVM and the Cpu0 example code</b>	<b>359</b>
14.1 Setting Up Your Mac . . . . .	359
14.2 Setting Up Your Linux Machine . . . . .	376
14.3 Install sphinx . . . . .	380
<b>15 Todo List</b>	<b>381</b>
<b>16 Book example code</b>	<b>383</b>





**Warning:** This is a work in progress. If you would like to contribution, please push updates and patches to the main github project available at <http://github.com/Jonathan2251/lbd> for review.



# ABOUT

## 1.1 Authors

陳鍾樞

**Chen Chung-Shu** [gamma\\_chen@yahoo.com.tw](mailto:gamma_chen@yahoo.com.tw)

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

**Anoushe Jamshidi** [ajamshidi@gmail.com](mailto:ajamshidi@gmail.com)

## 1.2 Contributors

Chen Wei-Ren, [chenwj@iis.sinica.edu.tw](mailto: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, [silvas@purdue.edu](mailto:silvas@purdue.edu), 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 also get the kind help from LLVM development mail list, [llvmdev@cs.uiuc.edu](mailto:llvmdev@cs.uiuc.edu), even we don't know them. So, our experience is you are not alone and can get help from the development list members in working with the LLVM project. They are:

Akira Hatanaka <[ahatanak@gmail.com](mailto:ahatanak@gmail.com)> in va\_arg question answer.

Ulrich Weigand <[Ulrich.Weigand@de.ibm.com](mailto:Ulrich.Weigand@de.ibm.com)> in AsmParser question answer.

## 1.5 Revision history

Version 3.3.8, Not release yet.

**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

---

### Todo

Add info about LLVM documentation licensing.

---

## 1.7 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 make the Cpu0 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 was written using the LLVM 3.1 Mips backend as a reference. 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.

## 1.8 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](#)

## 1.9 Outline of Chapters

### *Cpu0 Instruction Set and LLVM Target Description:*

This chapter introduces the Cpu0 architecture, a high-level view of LLVM, and how Cpu0 will be targeted 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 UML graphs, and we continue to build the Cpu0 backend. Around 2300 lines of source code are added, most of which are common from one LLVM backend to another, regardless of the target architecture. By the end of this chapter, the Cpu0 LLVM backend will support three instructions to generate some initial assembly output.

### *Arithmetic and logic lsupport:*

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 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 of which are explained as their functionality is implemented.

:ref:\_sec-othertypesupport:

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** flow control statements are added in this chapter. Around 150 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 which translate elf into hex file format is added in last section.

### *Run backend:*

Add AsmParser support for translate hand code assembly language into obj first. Next, design the CPU0 backend with Verilog language of Icarus tool. Finally feed the hex file which generated by `llvm-objdump` and see the CPU0 running result.

### *Backend Optimization:*

Introduce how to do backend optimization by a simple effective example, and redesign Cpu0 instruction sets to be a efficient RISC CPU.

### *LLD for Cpu0:*

Develop ELF linker for Cpu0 backend based on lld project.

### *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 INSTRUCTION SET AND LLVM TARGET DESCRIPTION

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 section 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 <sup>1</sup>. You can read the original article from the AOSA website if you prefer. Finally, 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 section.

## 2.1 Cpu0 Processor Architecture Details

This subsection is based on materials available here <sup>2</sup> (Chinese) and <sup>3</sup> (English).

### 2.1.1 Brief introduction

Cpu0 is a 32-bit architecture. It has 16 general purpose registers (R0, ..., R15), the Instruction Register (IR), the memory access registers MAR & MDR. Its structure is illustrated in Figure 2.1 below.

The registers are used for the following purposes:

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

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

<sup>3</sup> 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](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)

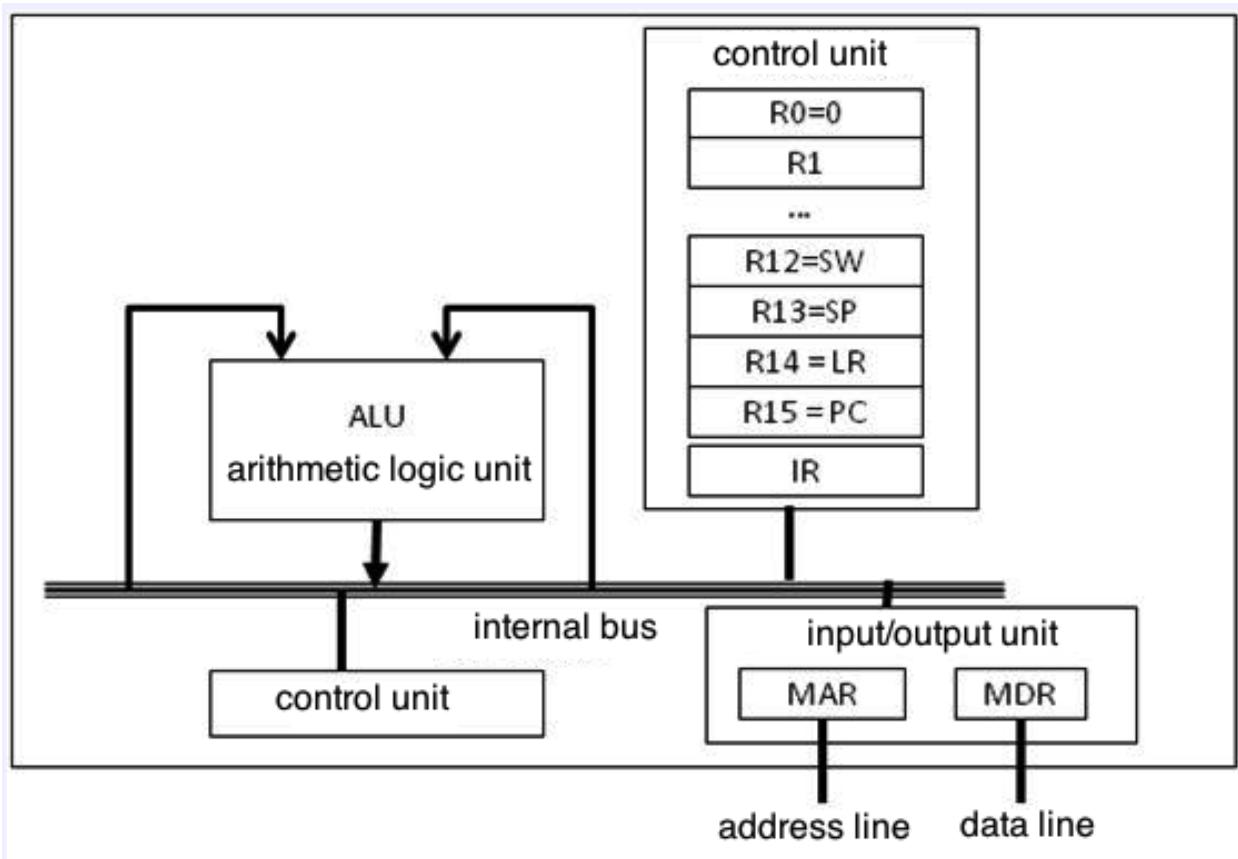


Figure 2.1: Architectural block diagram of the Cpu0 processor

Table 2.1: Cpu0 registers purposes

Register	Description
IR	Instruction register
R0	Constant register, value is 0
R1-R11	General-purpose registers
R12	Status Word register (SW)
R13	Stack Pointer register (SP)
R14	Link Register (LR)
R15	Program Counter (PC)
MAR	Memory Address Register (MAR)
MDR	Memory Data Register (MDR)
HI	High part of MULT result
LO	Low part of MULT result
SW	Status Word Register

## 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.

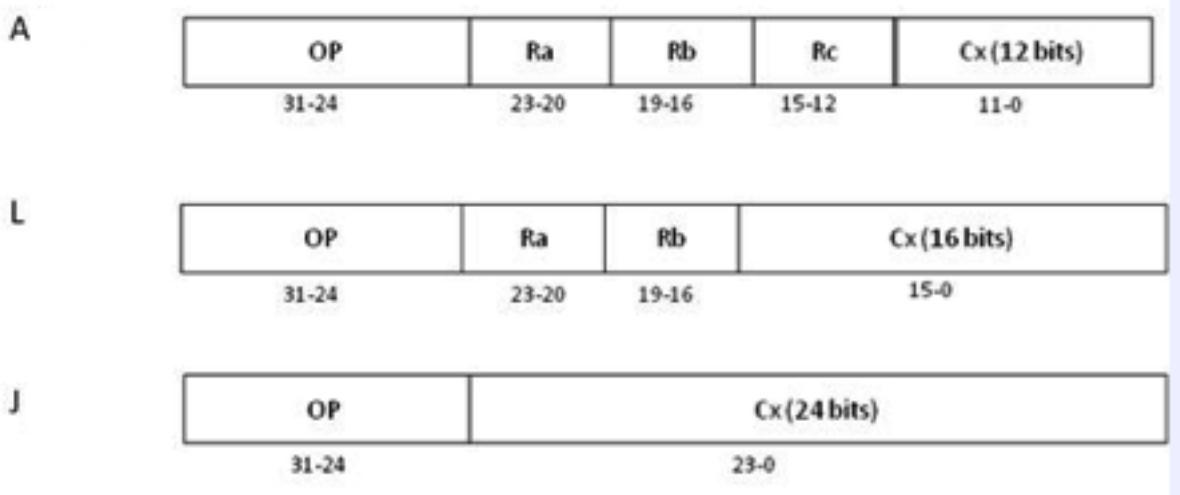


Figure 2.2: Cpu0's three instruction formats

The following table details the Cpu0 instruction set:

- First column F.: meaning Format.

Table 2.2: Cpu0 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

Continued on next page

Table 2.2 – continued from previous page

F.	Mnemonic	Opcode	Meaning	Syntax	Operation
L	LB	03	Load byte	LB Ra, [Rb+Cx]	Ra <= (byte)[Rb+Cx]
L	LBu	04	Load byte unsigned	LBu Ra, [Rb+Cx]	Ra <= (byte)[Rb+Cx]
L	SB	05	Store byte	SB Ra, [Rb+Cx]	[Rb+Cx] <= (byte)Ra
A	LH	06	Load half word unsigned	LH Ra, [Rb+Cx]	Ra <= (2bytes)[Rb+Cx]
A	LHu	07	Load half word	LHu Ra, [Rb+Cx]	Ra <= (2bytes)[Rb+Cx]
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) <sup>4</sup>
A	ADDu	11	Add unsigned	ADD Ra, Rb, Rc	Ra <= Rb + Rc
A	SUBu	12	Sub unsigned	SUB Ra, Rb, Rc	Ra <= Rb - Rc
A	ADD	13	Add	ADD Ra, Rb, Rc	Ra <= Rb + Rc
A	SUB	14	Subtract	SUB Ra, Rb, Rc	Ra <= Rb - Rc
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 <sup>5</sup>
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 <sup>4</sup>
A	SHLV	21	Shift left	SHLV Ra, Rb, Rc	Ra <= Rb << Rc
A	SHRV	22	Shift right	SHRV Ra, Rb, Rc	Ra <= Rb >> 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	SWI	3A	Software interrupt	SWI Cx	LR <= PC; PC <= Cx
J	JSUB	3B	Jump to subroutine	JSUB Cx	LR <= PC; PC <= PC + Cx
J	RET	3C	Return from subroutine	RET LR	PC <= LR
J	IRET	3D	Return from interrupt handler	IRET	PC <= LR; INT 0
J	JALR	3E	Indirect jump	JALR Rb	LR <= PC; PC <= Rb <sup>6</sup>
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	DIV Ra, Rb	HI<=Ra%Rb, LO<=Ra/Rb
L	MFHI	46	Move HI to GPR	MFHI Ra	Ra <= HI

Continued on next page

<sup>4</sup> 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.

<sup>5</sup> Rb '">>> Cx, Rb '">>> Rc: Shift with signed bit remain. It's equal to ((Rb&'h80000000)|Rb>>Cx) or ((Rb&'h80000000)|Rb>>Rc).

<sup>6</sup> 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 2.2 – continued from previous page

F.	Mnemonic	Opcode	Meaning	Syntax	Operation
L	MFLO	47	Move LO to GPR	MFLO Ra	Ra <= LO
L	MTHI	48	Move GPR to HI	MTHI Ra	HI <= Ra
L	MTLO	49	Move GPR to LO	MTLO Ra	LO <= Ra
L	MFSW	50	Move SW to GPR	MFSW Ra	Ra <= SW
L	MTSW	51	Move GPR to SW	MTSW Ra	SW <= Ra

#### Note: Cpu0 unsigned instructions

Like Mips, the mathematic unsigned instructions except DIVU, such as ADDu and SUBu, are no overflow exception instructions. The ADDu and SUBu handle both signed and unsigned integers well. For example, (ADDu 1, -2) is -3; (ADDu 0x01, 0xffffffff) is 0xffffffff = (4G - 1). If you treat the result is negative then it is -1. Otherwise, if you treat the result is positive then it's (+4G - 1).

### 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), and Interrupt (I), Trap (T), and Mode (M) boolean flags. The bit layout of the SW register is shown in Figure 2.3 below.

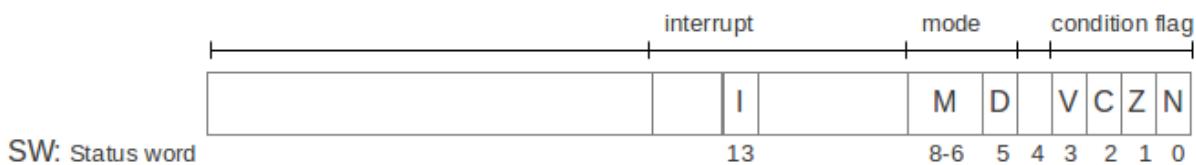


Figure 2.3: Cpu0 status word (SW) register

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

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

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

### 2.1.4 Cpu0's Stages of Instruction Execution

The Cpu0 architecture has a three-stage pipeline. The stages are instruction fetch (IF), decode (D), and execute (EX), and they occur in that order. Here is a description of what happens in the processor:

1. Instruction fetch
  - 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. Decode

- 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
- The ALU executes the operation designated by the control unit upon data in registers. After the ALU is done, the result is stored in the destination register.

### 2.1.5 Cpu0’s Interrupt Vector

Table 2.3: Cpu0’s Interrupt Vector

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

## 2.2 LLVM Structure

The text in this and the following section comes from the AOSA chapter on LLVM written by Chris Lattner<sup>6</sup>.

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.

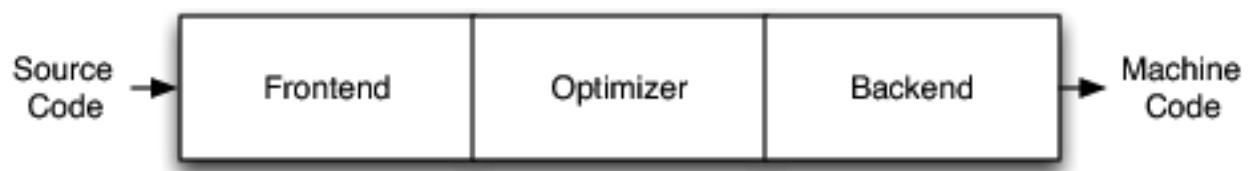


Figure 2.4: Three Major Components of a Three Phase Compiler

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](#).

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

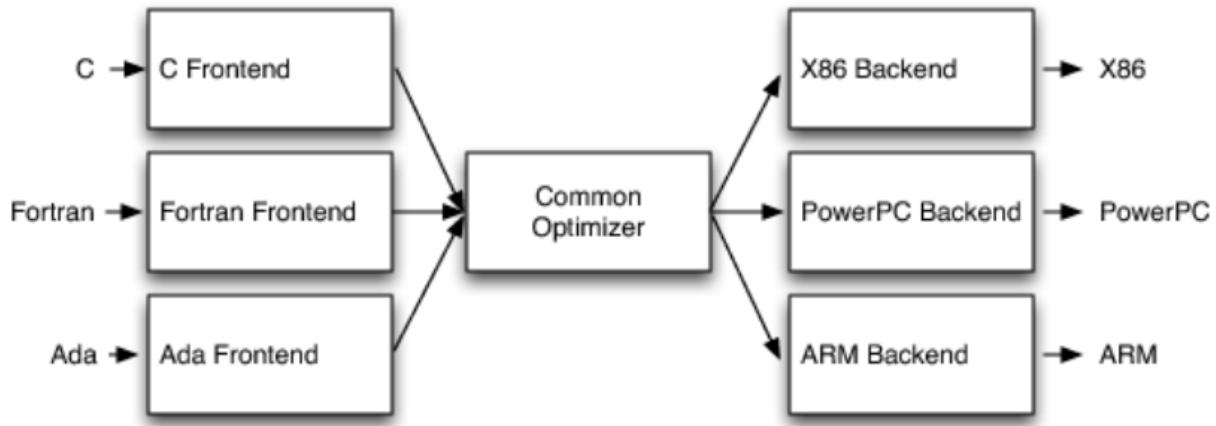


Figure 2.5: Retargetability

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 section 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.

## 2.3 .td: LLVM's Target Description Files

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](#).

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]> { ... }
```

## 2.4 Creating the Initial Cpu0 .td Files

As has been discussed in the previous section, LLVM uses 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

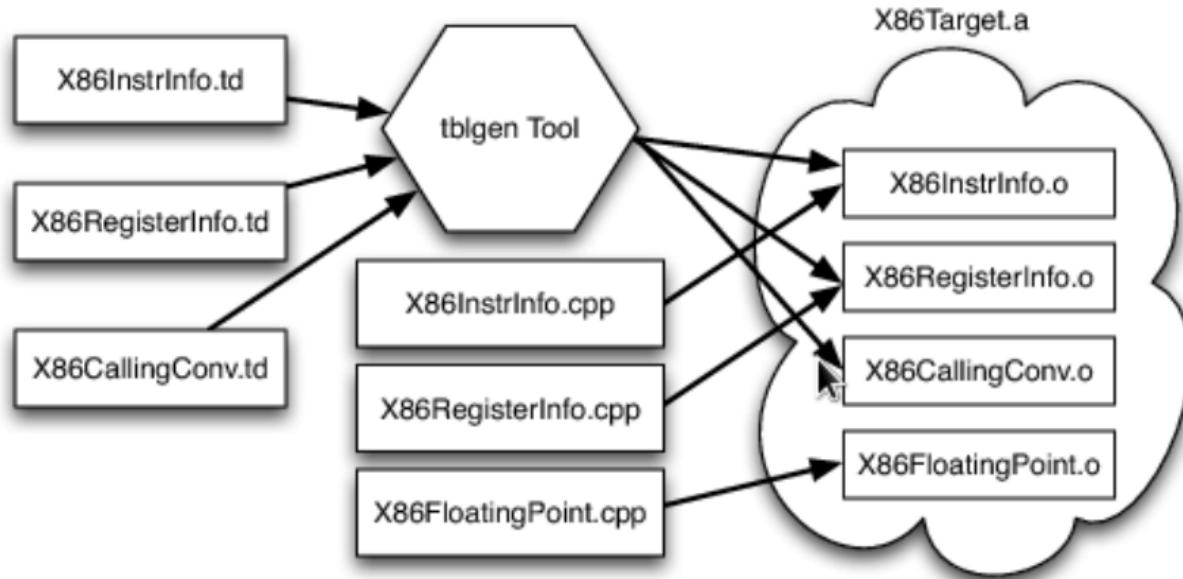


Figure 2.6: Simplified x86 Target Definition

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 <sup>7</sup> 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 Cpu0.td, which is shown below:

#### Ibdex/Chapter2/Cpu0.td

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 (see the X86 backend, for example) which can help you if certain instructions can only write to specific registers. In this case, there is only one set of general purpose registers for Cpu0, and some registers that are reserved so that they are not modified by instructions during execution.

#### Ibdex/Chapter2/Cpu0RegisterInfo.td

In C++, classes 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.

<sup>7</sup> <http://llvm.org/docs/TableGenFundamentals.html>

In .td files, classes describe the structure of how data is laid out, while definitions act as the specific instances of the classes. 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 classes. 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`.

---

### Todo

I might want to re-edit the following paragraph

---

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,

### [Index/Chapter2/Cpu0InstrInfo.td](#)

The `Cpu0InstrFormats.td` is included by `Cpu0InstrInfo.td` as follows,

### [Index/Chapter2/Cpu0InstrFormats.td](#)

`ADDiu` is class `ArithLogicI` inherited from `FL`, can be expanded and get member value 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 RC:$ra, (ins RC:$rb, Od:$simm16),  
        !strconcat(instr_asm, "\t$ra, $rb, $simm16"),  
        [(set RC:$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
```

Expand with `FL` further,

```

: FL<op, (outs RC:$ra), (ins RC:$rb, Od:$imm16),
!strconcat(instr_asm, "\t$ra, $rb, $imm16"),
[(set RC:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu>

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;
}

So,
op = 0x09
outs = CPURegs:$ra
ins = CPURegs:$rb, imm16:$imm16
asmstr = "addiu\t$ra, $rb, $imm16"
pattern = [(set CPURegs:$ra, (add RC:$rb, immSExt16:$imm16))]
itin = IIAlu

Members are,
ra = CPURegs:$ra
rb = CPURegs:$rb
imm16 = imm16:$imm16
Opcode = 0x09;
Inst{23-20} = CPURegs:$ra;
Inst{19-16} = CPURegs:$rb;
Inst{15-0} = imm16:$imm16;

```

Expand with Cpu0Inst further,

```

class FL<bits<8> op, dag outs, dag ins, string asmstr, list<dag> pattern,
InstrItinClass itin>: Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmL>

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;
}

So,
outs = CPUREgs:$ra
ins = CPUREgs:$rb,simm16:$imm16
asmstr = "addiu\t$ra, $rb, $imm16"
pattern = [(set CPUREgs:$ra, (add RC:$rb, immSExt16:$imm16))]
itin = IIAlu
f = FrmL

Members are,
Inst{31-24} = 0x09;
OutOperandList = CPUREgs:$ra
InOperandList = CPUREgs:$rb,simm16:$imm16
AsmString = "addiu\t$ra, $rb, $imm16"
Pattern = [(set CPUREgs:$ra, (add RC:$rb, immSExt16:$imm16))]
Itinerary = IIAlu

Summary with all members are,
// Inherited from parent like Instruction
Namespace = "Cpu0";
DecoderNamespace = "Cpu0";
Inst{31-24} = 0x08;
Inst{23-20} = CPUREgs:$ra;
Inst{19-16} = CPUREgs:$rb;
Inst{15-0}  = simm16:$imm16;
OutOperandList = CPUREgs:$ra
InOperandList = CPUREgs:$rb,simm16:$imm16
AsmString = "addiu\t$ra, $rb, $imm16"
Pattern = [(set CPUREgs:$ra, (add RC:$rb, immSExt16:$imm16))]
Itinerary = IIAlu
// From Cpu0Inst
Opcode = 0x09;
// From FL
ra = CPUREgs:$ra
rb = CPUREgs:$rb
imm16 = simm16:$imm16

```

It's a lousy process. Similarly, LD and ST instruction definition can be expanded in this way. Please notify the Pattern = [(set CPUREgs:\$ra, (add RC:\$rb, immSExt16:\$imm16))] which include keyword “**add**”. We will use it in DAG transformations later.

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

**Ibdex/Chapter2/Cpu0Schedule.td**

## 2.5 Write cmake file

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

**Ibdex/Chapter2/CMakeLists.txt**

**Ibdex/Chapter2/LLVMBuild.txt**

CMakeLists.txt is the make information for cmake, # is comment. LLVMBuild.txt files are 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 2 files in comments. Please spend a little time to read it.

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: `libLLVMCpu0CodeGen.a`, `libLLVMCpu0Desc.a` and `libLLVMCpu0Info.a` in lib/ of your build directory. For more details please see “Building LLVM with CMake”<sup>8</sup> and “LLVMBuild Guide”<sup>9</sup>.

## 2.6 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/Chapter2/Cpu0.h**

**Ibdex/Chapter2/TargetInfo/Cpu0TargetInfo.cpp**

**Ibdex/Chapter2/TargetInfo/CMakeLists.txt**

**Ibdex/Chapter2/TargetInfo/LLVMBuild.txt**

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

---

<sup>8</sup> <http://llvm.org/docs/CMake.html>

<sup>9</sup> <http://llvm.org/docs/LLVMBuild.html>

[lbdex/Chapter2/Cpu0TargetMachine.cpp](#)

[lbdex/Chapter2/MCTargetDesc/Cpu0MCTargetDesc.h](#)

[lbdex/Chapter2/MCTargetDesc/Cpu0MCTargetDesc.cpp](#)

[lbdex/Chapter2/MCTargetDesc/CMakeLists.txt](#)

[lbdex/Chapter2/MCTargetDesc/LLVMBuild.txt](#)

Please see “Target Registration”<sup>10</sup> for reference.

## 2.7 Build libraries and td

The llvm source code is put in /Users/Jonathan/llvm/release/src and have llvm release-build in /Users/Jonathan/llvm/release/configure\_release\_build. About how to build llvm, please refer<sup>11</sup>. We made a copy from /Users/Jonathan/llvm/release/src to /Users/Jonathan/llvm/test/src for working with my Cpu0 target back end. Sub-directories src is for source code and cmake\_debug\_build is for debug build directory.

Except directory src/lib/Target/Cpu0, there are a couple of files modified to support cpu0 new Target. Please check files in src\_files\_modified/src\_files\_modified/src/.

You can update your llvm working copy and find the modified files by command,

```
cp -rf lbdex/src_files_modified/src_files_modified/src/  
* yourllvm/workingcopy/sourcedir/.
```

```
118-165-78-230:test Jonathan$ pwd  
/Users/Jonathan/test  
118-165-78-230:test Jonathan$ grep -R "cpu0" src  
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  
src/lib/Support/Triple.cpp:  case cpu0:    return "cpu0";  
...
```

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

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

---

<sup>10</sup> <http://llvm.org/docs/WritingAnLLVMBackend.html#target-registration>

<sup>11</sup> [http://clang.llvm.org/get\\_started.html](http://clang.llvm.org/get_started.html)

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

```
118-165-78-230:test Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/bin/
Debug/llc --version
LLVM (http://llvm.org/):
...
Registered Targets:
arm      - ARM
cellspu  - STI CBEA Cell SPU [experimental]
cpp      - C++ backend
cpu0     - Cpu0
cpu0el   - Cpu0el
...
...
```

The `llc -version` can display “`cpu0`” and “`cpu0el`” message, because the code from file `Target-Info/Cpu0TargetInfo.cpp` what in “section Target Registration” <sup>12</sup> we made.

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

```
118-165-75-57:lbdex Jonathan$ pwd
/Users/Jonathan/llvm/test/src/lib/Target/Cpu0/lbdex
118-165-75-57:lbdex Jonathan$ sh removecpu0.sh
118-165-75-57:lbdex Jonathan$ cp -rf lbdex/Chapter2/
* ../.

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 lib/Target/Cpu0/*
118-165-75-57: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
```

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

### **lbdex/InputFiles/ch3.cpp**

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

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

```
118-165-78-230:InputFiles 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` borrow the ABI from `Mips`. Next step, transfer bitcode `.bc` to human readable text format as follows,

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

---

<sup>12</sup> <http://jonathan2251.github.com/lbd/llvmsstructure.html#target-registration>

```
// 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 = "x86_64-unknown-linux-gnu"

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

Now, compile ch3.bc into ch3.cpu0.s, we get the error message as follows,

```
118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
Assertion failed: (target.get() && "Could not allocate target machine!"),
function main, file /Users/Jonathan/llvm/test/src/tools/llc/llc.cpp,
...
```

Currently we just define target td files (Cpu0.td, Cpu0RegisterInfo.td, ...). According to LLVM structure, we need to define our target machine and include those td related files. The error message say we didn't define our target machine.

# BACKEND STRUCTURE

This chapter introduce the back end class inherit tree and class members first. Next, following the back end structure, adding individual class implementation in each section. There are compiler knowledge like DAG (Directed-Acyclic-Graph) and instruction selection needed in this chapter. This chapter explains these knowledge just when needed. 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. 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 3000 lines of code in this chapter.

## 3.1 TargetMachine structure

Your back end should define a TargetMachine class, for example, we define the Cpu0TargetMachine class. Cpu0TargetMachine class contains it's own instruction class, frame/stack class, DAG (Directed-Acyclic-Graph) class, and register class. The Cpu0TargetMachine contents and it's own class as follows,

`include/llvm/Target/TargetMachine.h`

```
//- TargetMachine.h
class TargetMachine {
    TargetMachine(const TargetMachine &) LLVM_DELETED_FUNCTION;
    void operator=(const TargetMachine &) LLVM_DELETED_FUNCTION;
    ...
public:
    // Interfaces to the major aspects of target machine information:
    // -- Instruction opcode and operand information
    // -- Pipelines and scheduling information
    // -- Stack frame information
    // -- Selection DAG lowering information
    //
    virtual const TargetInstrInfo *getInstrInfo() const { return 0; }
    virtual const TargetFrameLowering *getFrameLowering() const { return 0; }
    virtual const TargetLowering *getTargetLowering() const { return 0; }
    virtual const TargetSelectionDAGInfo *getSelectionDAGInfo() const { return 0; }
    virtual const DataLayout *getDataLayout() const { return 0; }
    ...
    /// getSubtarget - This method returns a pointer to the specified type of
    /// TargetSubtargetInfo. In debug builds, it verifies that the object being
    /// returned is of the correct type.
```

```
template<typename STC> const STC &getSubtarget() const {
    return *static_cast<const STC*>(getSubtargetImpl());
}

...
class LLVMTargetMachine : public TargetMachine {
protected: // Can only create subclasses.
    LLVMTargetMachine(const Target &T, StringRef TargetTriple,
                      StringRef CPU, StringRef FS, TargetOptions Options,
                      Reloc::Model RM, CodeModel::Model CM,
                      CodeGenOpt::Level OL);
    ...
};


```

[lbdex/Chapter3\\_1/Cpu0TargetObjectFile.h](#)

[lbdex/Chapter3\\_1/Cpu0TargetObjectFile.cpp](#)

[lbdex/Chapter3\\_1/Cpu0TargetMachine.h](#)

[lbdex/Chapter3\\_1/Cpu0TargetMachine.cpp](#)

[include/llvm/Target/TargetInstrInfo.h](#)

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

[lbdex/Chapter3\\_1/Cpu0.td](#)

[lbdex/Chapter3\\_1/Cpu0CallingConv.td](#)

[lbdex/Chapter3\\_1/Cpu0FrameLowering.h](#)

[lbdex/Chapter3\\_1/Cpu0FrameLowering.cpp](#)

}

[lbdex/Chapter3\\_1/Cpu0InstrInfo.h](#)

[lbdex/Chapter3\\_1/Cpu0InstrInfo.cpp](#)

[lbdex/Chapter3\\_1/Cpu0ISelLowering.h](#)

[lbdex/Chapter3\\_1/Cpu0ISelLowering.cpp](#)

}

[lbdex/Chapter3\\_1/Cpu0MachineFunction.h](#)

[lbdex/Chapter3\\_1/Cpu0SelectionDAGInfo.h](#)

[lbdex/Chapter3\\_1/Cpu0SelectionDAGInfo.cpp](#)

[lbdex/Chapter3\\_1/Cpu0Subtarget.h](#)

[lbdex/Chapter3\\_1/Cpu0Subtarget.cpp](#)

[lbdex/Chapter3\\_1/Cpu0RegisterInfo.h](#)

[lbdex/Chapter3\\_1/Cpu0RegisterInfo.cpp](#)

[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);
};


```

The Cpu0TargetMachine inherit tree is TargetMachine <- LLVMTargetMachine <- Cpu0TargetMachine. Cpu0TargetMachine has class Cpu0Subtarget, Cpu0InstrInfo, Cpu0FrameLowering, Cpu0TargetLowering and Cpu0SelectionDAGInfo. Class Cpu0Subtarget, Cpu0InstrInfo, Cpu0FrameLowering, Cpu0TargetLowering and Cpu0SelectionDAGInfo are inherited from parent class TargetSubtargetInfo, TargetInstrInfo, TargetFrameLowering, TargetLowering and TargetSelectionDAGInfo.

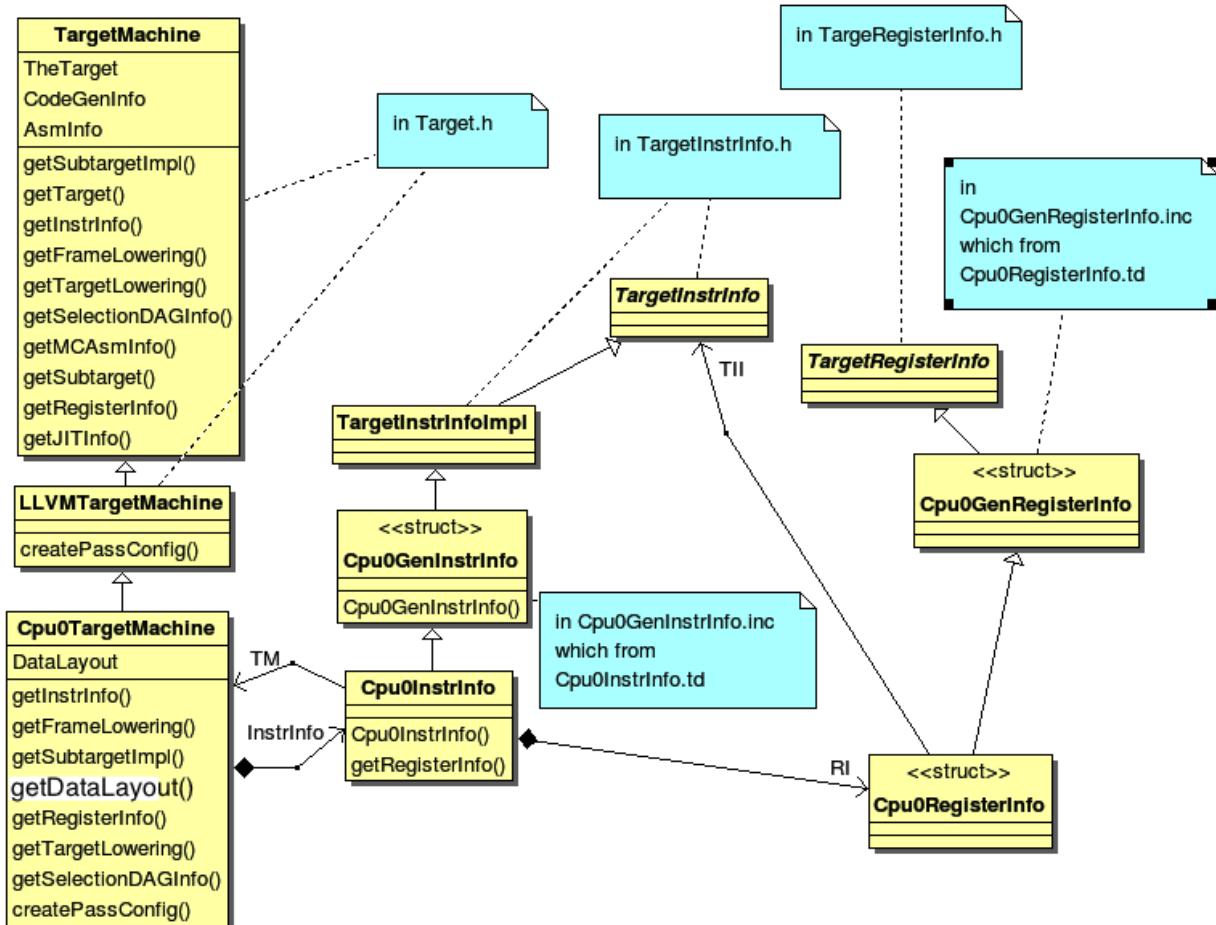


Figure 3.1: TargetMachine class diagram 1

Figure 3.1 shows Cpu0TargetMachine inherit tree and it's Cpu0InstrInfo class inherit tree. Cpu0TargetMachine contains Cpu0InstrInfo and ... other class. Cpu0InstrInfo contains Cpu0RegisterInfo class, RI. Cpu0InstrInfo.td and Cpu0RegisterInfo.td will generate Cpu0GenInstrInfo.inc and Cpu0GenRegisterInfo.inc which contain some member functions implementation for class Cpu0InstrInfo and Cpu0RegisterInfo.

Figure 3.2 as below shows Cpu0TargetMachine contains class TSInfo: Cpu0SelectionDAGInfo, FrameLowering: Cpu0FrameLowering, Subtarget: Cpu0Subtarget and TLInfo: Cpu0TargetLowering.

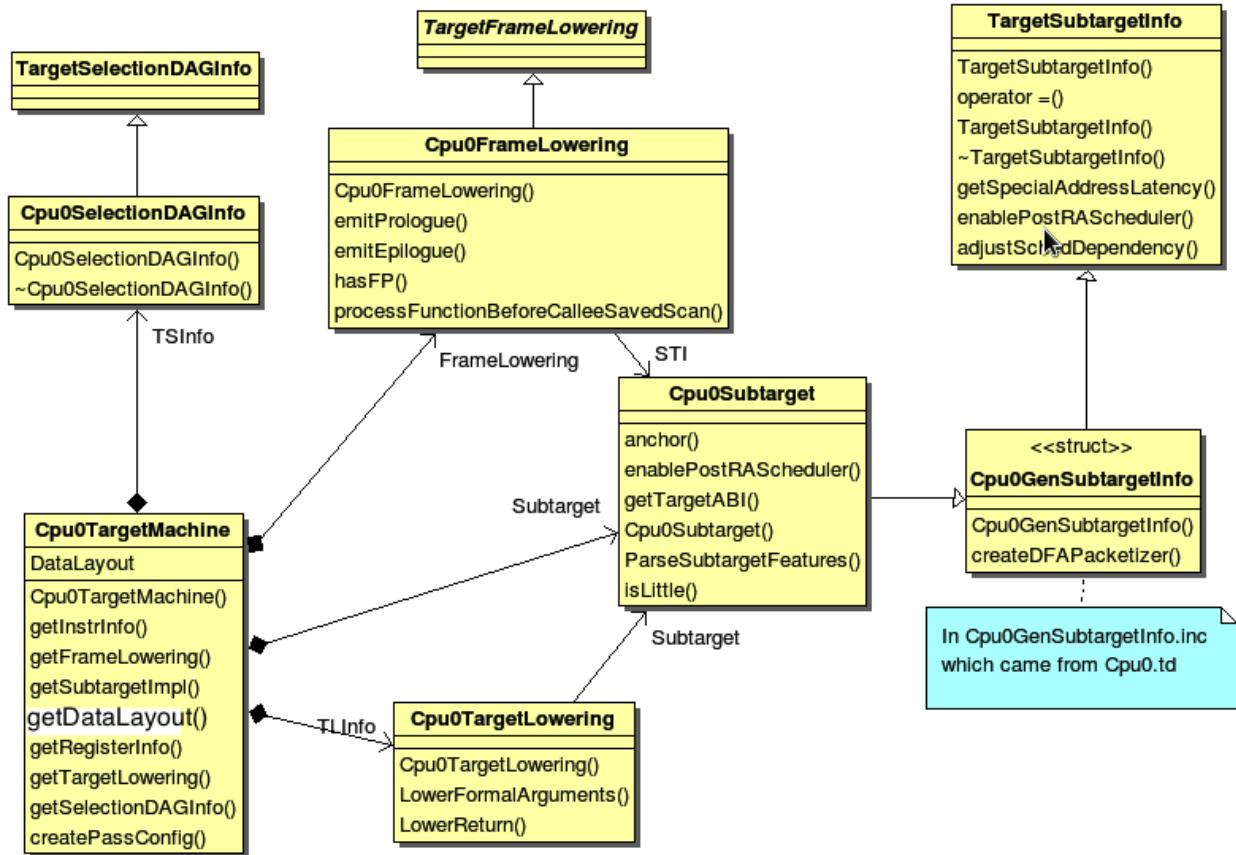


Figure 3.2: TargetMachine class diagram 2

Figure 3.3 shows some members and operators (member function) of the parent class TargetMachine's. Figure 3.4 as below shows some members of class InstrInfo, RegisterInfo and TargetLowering. Class DAGInfo is skipped here.

Benefit from the inherit tree structure, we just need to implement few code in instruction, frame/stack, select DAG class. Many code implemented by their parent class. The llvm-tblgen generate Cpu0GenInstrInfo.inc from Cpu0InstrInfo.td. Cpu0InstrInfo.h extract those code it need from Cpu0GenInstrInfo.inc by define "#define GET\_INSTRINFO\_HEADER". Following is the code fragment from Cpu0GenInstrInfo.inc. Code between "#if def GET\_INSTRINFO\_HEADER" and "#endif // GET\_INSTRINFO\_HEADER" will be extracted by Cpu0InstrInfo.h.

`cmake_debug_build/lib/Target/Cpu0/Cpu0GenInstrInfo.inc`

```
//- Cpu0GenInstrInfo.inc which generate from Cpu0InstrInfo.td
#ifndef GET_INSTRINFO_HEADER
#define GET_INSTRINFO_HEADER
namespace llvm {
```

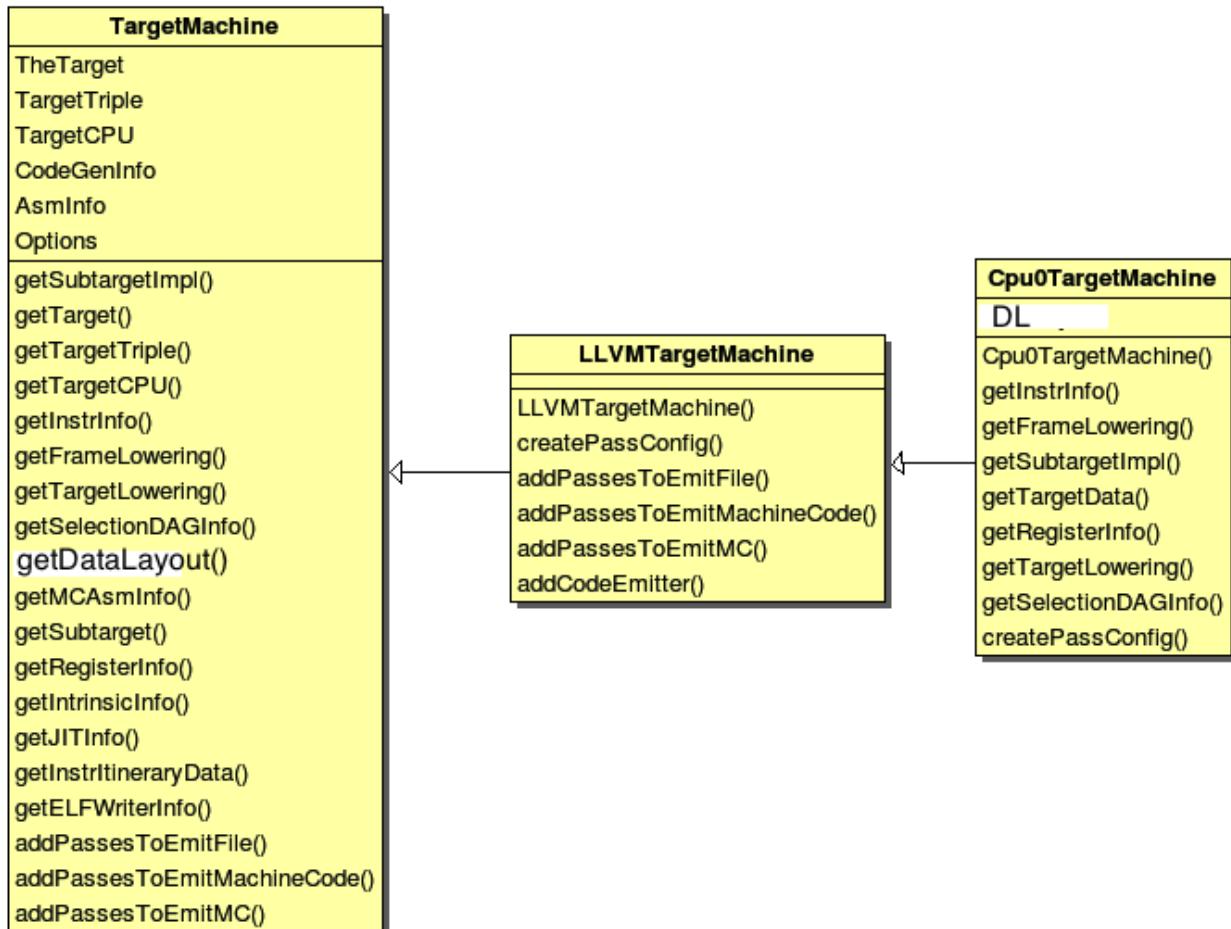


Figure 3.3: TargetMachine members and operators

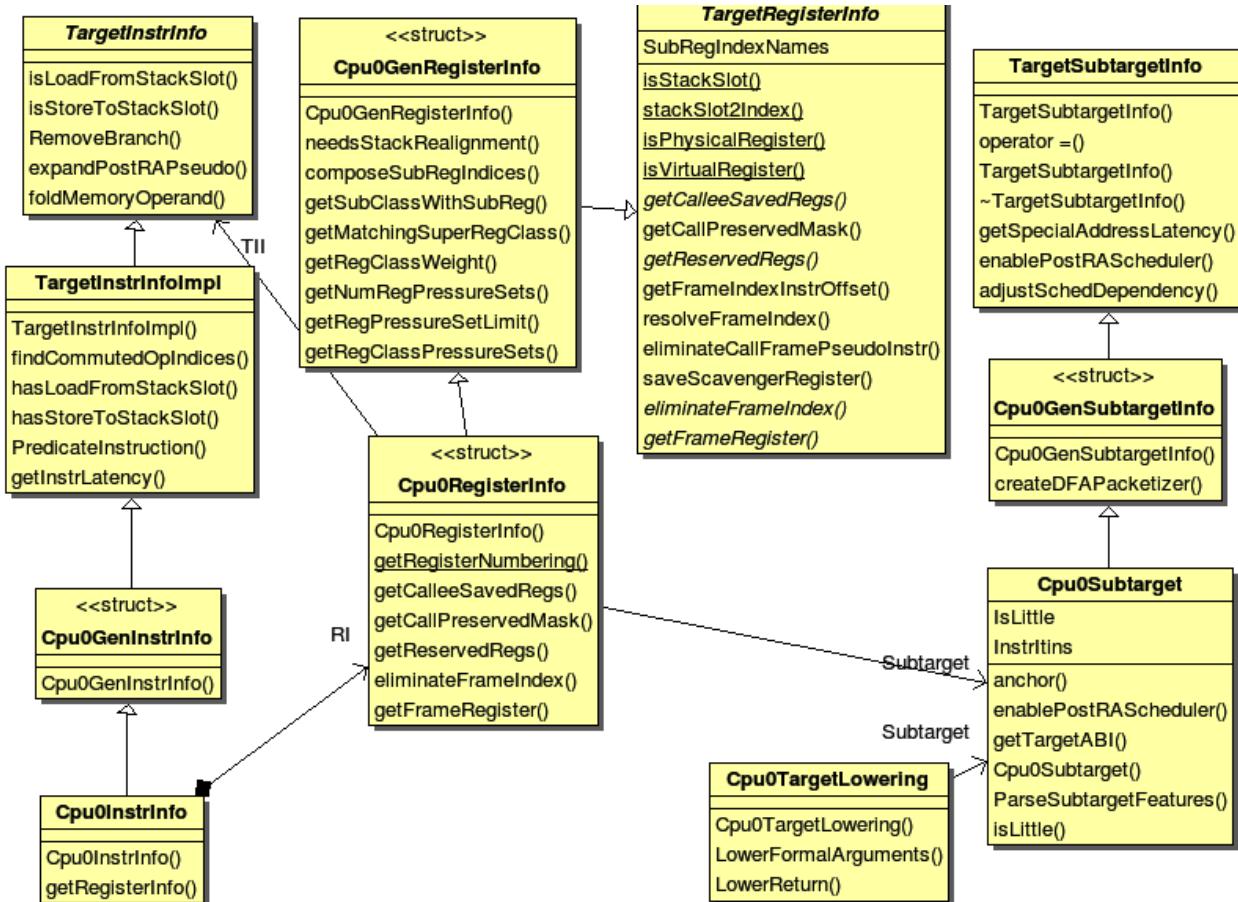


Figure 3.4: Other class members and operators

```
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 <sup>1</sup>.

Now, the code in Chapter3\_1/ add class Cpu0TargetMachine(Cpu0TargetMachine.h and .cpp), Cpu0Subtarget (Cpu0Subtarget.h and .cpp), Cpu0InstrInfo (Cpu0InstrInfo.h and .cpp), Cpu0FrameLowering (Cpu0FrameLowering.h and .cpp), Cpu0TargetLowering (Cpu0ISelLowering.h and .cpp) and Cpu0SelectionDAGInfo (Cpu0SelectionDAGInfo.h and .cpp). CMakeLists.txt modified with those new added \*.cpp as follows,

### Ibdex/Chapter3\_1/CMakeLists.txt

Please take a look for Chapter3\_1 code. After that, building Chapter3\_1 by make as chapter 2 (of course, you should remove old src/lib/Target/Cpu0 and replace them with src/lib/Target/Cpu0/lbdex/Chapter3\_1/). You can remove cmake\_debug\_build/lib/Target/Cpu0/\*.inc before do “make” to ensure your code rebuild completely. By remove \*.inc, all files those have included .inc will be rebuild, then your Target library will be regenerated. Command as follows,

```
118-165-78-230:cmake_debug_build Jonathan$ rm -rf lib/Target/Cpu0/*
```

Now, let's build Chapter3\_1 as the following command,

```
118-165-75-57:ExampleCode Jonathan$ pwd
/Users/Jonathan/llvm/test/src/lib/Target/Cpu0/lbdex
118-165-75-57:lbdex Jonathan$ sh removecpu0.sh
118-165-75-57:lbdex Jonathan$ cp -rf Chapter3_1/
* ../.
```

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

```

The errors say that we have not Target AsmPrinter. Let's add it in next section.

## 3.2 Add AsmPrinter

Chapter3\_2/cpu0 contains the Cpu0AsmPrinter definition. First, we add definitions in Cpu0.td to support Assembly-Writer. Cpu0.td is added with the following fragment,

### Ibdex/Chapter3\_2/Cpu0.td

As comments indicate, it will generate Cpu0GenAsmWrite.inc which is included by Cpu0InstPrinter.cpp as follows,

---

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

[Index/Chapter3\\_2/InstPrinter/Cpu0InstPrinter.h](#)

[Index/Chapter3\\_2/InstPrinter/Cpu0InstPrinter.cpp](#)

[Index/Chapter3\\_2/InstPrinter/CMakeLists.txt](#)

[Index/Chapter3\\_2/InstPrinter/LLVMBuild.txt](#)

Cpu0GenAsmWrite.inc has the implementation 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 need to do is add a class Cpu0InstPrinter and include them as did in Chapter3\_1.

File Chapter3\_1/Cpu0/InstPrinter/Cpu0InstPrinter.cpp include Cpu0GenAsmWrite.inc and call the auto-generated functions from TableGen.

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,

[Index/Chapter3\\_2/Cpu0MCInstLower.h](#)

[Index/Chapter3\\_2/Cpu0MCInstLower.cpp](#)

[Index/Chapter3\\_2/MCTargetDesc/Cpu0BaseInfo.h](#)

[Index/Chapter3\\_2/MCTargetDesc/Cpu0MCAsmInfo.h](#)

[Index/Chapter3\\_2/MCTargetDesc/Cpu0MCAsmInfo.cpp](#)

Finally, add code in Cpu0MCTargetDesc.cpp to register Cpu0InstPrinter as follows,

[Index/Chapter3\\_2/MCTargetDesc/Cpu0MCTargetDesc.h](#)

[Index/Chapter3\\_2/MCTargetDesc/Cpu0MCTargetDesc.cpp](#)

[Index/Chapter3\\_2/MCTargetDesc/CMakeLists.txt](#)

Cpu0MCAsmInfo.cpp

[Index/Chapter3\\_2/MCTargetDesc/LLVMBuild.txt](#)

Cpu0AsmPrinter

Now, it's time to work with AsmPrinter. According section "section Target Registration" <sup>2</sup>, we can register our AsmPrinter when we need it as the following function of LLVMInitializeCpu0AsmPrinter(),

---

<sup>2</sup> <http://jonathan2251.github.com/lbd/llvmstructure.html#target-registration>

### [Index/Chapter3\\_2/Cpu0AsmPrinter.h](#)

### [Index/Chapter3\\_2/Cpu0AsmPrinter.cpp](#)

The dynamic register mechanism is a good idea, right.

Add the following code to Cpu0ISelLowering.cpp.

### [Index/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/Chapter3\\_2/Cpu0MachineFunction.h](#)

```
class Cpu0FunctionInfo : public MachineFunctionInfo {  
  ...  
  bool EmitNOAT;  
  
public:  
  Cpu0FunctionInfo(MachineFunction& MF)  
  : ...  
  EmitNOAT(false),  
  ...  
  {}  
  
  ...  
  bool getEmitNOAT() const { return EmitNOAT; }  
  void setEmitNOAT() { EmitNOAT = true; }  
};
```

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

### [Index/Chapter3\\_2/CMakeLists.txt](#)

```

tablegen(LLVM Cpu0GenCodeEmitter.inc -gen-emitter)
tablegen(LLVM Cpu0GenMCCodeEmitter.inc -gen-emitter -mc-emitter)

tablegen(LLVM Cpu0GenAsmWriter.inc -gen-asm-writer)
...
add_llvm_target(Cpu0CodeGen
    Cpu0AsmPrinter.cpp
    ...
    Cpu0MCInstLower.cpp
    ...
)
...
add_subdirectory(InstPrinter)
...

```

### Ibdex/Chapter3\_2/LLVMBuild.txt

```

// LLVMBuild.txt
[common]
subdirectories =
    InstPrinter
    ...

[component_0]
...
# Please enable asmprinter
has_asmprinter = 1
...

[component_1]
# Add AsmPrinter Cpu0AsmPrinter
required_libraries =
    AsmPrinter
    ...
    Cpu0AsmPrinter
    ...

```

Now, run Chapter3\_2/Cpu0 for AsmPrinter support, will get error message as follows,

```

118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
/Users/Jonathan/llvm/test/cmake_debug_build/bin/Debug/llc: target does not
support generation of this file type!

```

The llc fails to compile IR code into machine code since we didn't implement class Cpu0DAGToDAGISel. Before the implementation, we will introduce the LLVM Code Generation Sequence, DAG, and LLVM instruction selection in next 3 sections.

## 3.3 LLVM Code Generation Sequence

Following diagram came 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

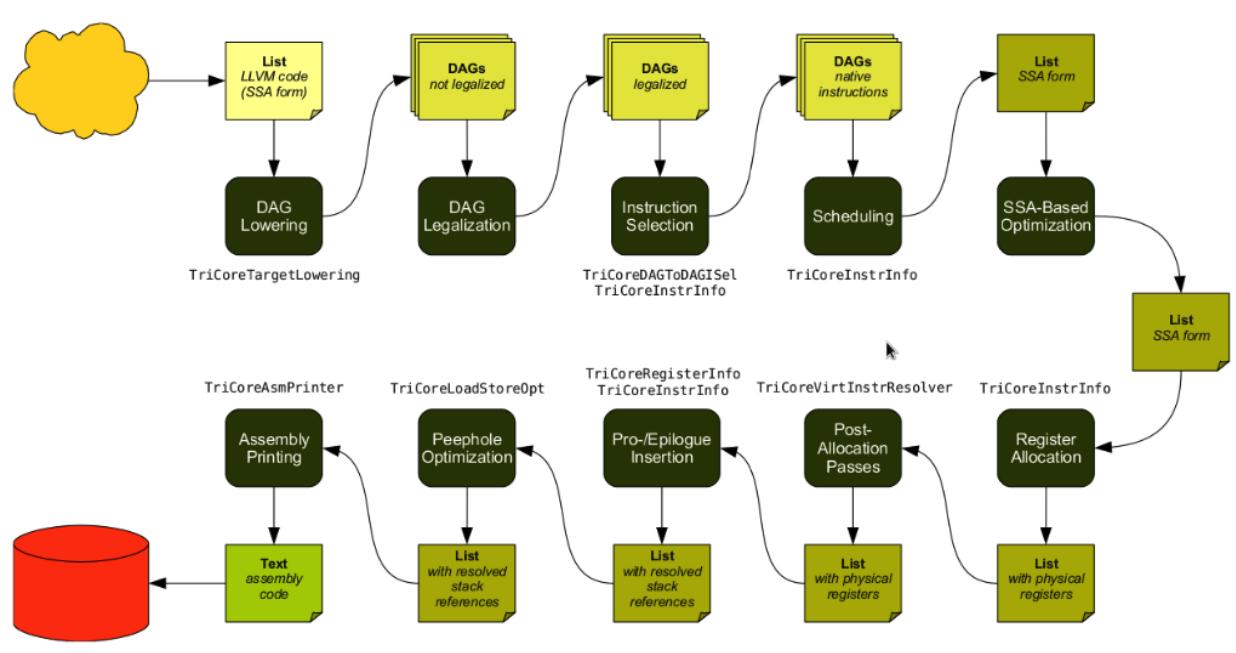


Figure 3.5: tricore\_llvm.pdf: Code generation sequence. On the path from LLVM code to assembly code, numerous passes are run through and several data structures are used to represent the intermediate results.

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
; 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 <sup>3</sup> and “LLVM Language Reference Manual” from <sup>4</sup> before go ahead, but we think read section 4.2 of tricore\_llvm.pdf is enough. We suggest you read the web site documents as above only when you are still not quite understand, even though 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
=> addiu i16 0, i32* %a

```

### 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)

```

<sup>3</sup> <http://llvm.org/docs/CodeGenerator.html>

<sup>4</sup> <http://llvm.org/docs/LangRef.html>

```

st %b, i32* %c, i16 0
%d = ld i32* %c

// Transfer above instructions order as follows. In RISC like Mips the ld %c use
// the previous instruction st %c, must wait more than 1
// cycles. Meaning the ld cannot follow st immediately.
=> st %b, i32* %c, i16 0
    st i32 %a, i16* %b, i16 5
    %d = ld i32* %c, i16 0
// 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 llc code generation sequence also can be obtained by `llc -debug-pass=Structure` as the following. The first 4 code generation sequences from Figure 3.5 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 in them (please check the \* in the following). No need to bother since the LLVM is under development and changed all the time.

```
118-165-79-200:InputFiles 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:InputFiles 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
```

## 3.4 DAG (Directed Acyclic Graph)

Many important techniques for local optimization begin by transforming a basic block into DAG. For example, the basic block code and its corresponding DAG as [Figure 3.6](#).

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.

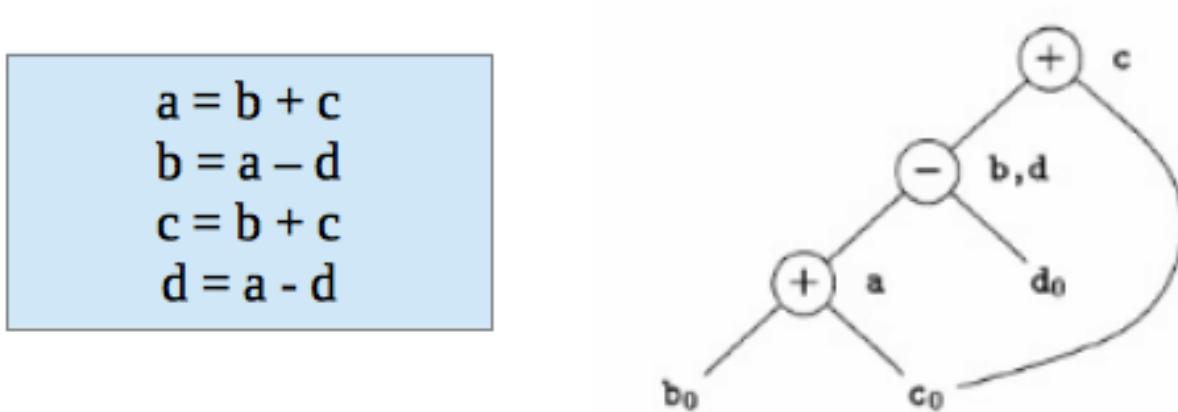


Figure 3.6: DAG example

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.

### 3.5 Instruction Selection

In back end, we need to translate IR code into machine code at Instruction Selection Process as Figure 3.7.

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 3.7: IR and it's corresponding machine instruction

For machine instruction selection, the better solution is represent IR and machine instruction by DAG. In Figure 3.8, we skip the register leaf. The  $rj + rk$  is IR DAG representation (for symbol notation, not llvm SSA form). ADD is machine instruction.

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 on Cpu0InstrInfo.td in the previous chapter. List them again as follows,

#### Index/Chapter3\_2/Cpu0InstrFormats.td

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

## Instruction Tree Patterns

Name	Effect	Trees
—	$r_i$	TEMP
ADD	$r_i = r_j + r_k$	<pre>  +   / \   *   </pre>
MUL	$r_i = r_j \times r_k$	<pre>  *   / \   /   </pre>
SUB	$r_i = r_j - r_k$	<pre>  -   / \   /   </pre>
DIV	$r_i = r_j / r_k$	<pre>  /   / \   /   </pre>
ADDI	$r_i = r_j + c$	<pre>  +   / \   CONST   +   / \   CONST   CONST</pre>
SUBI	$r_i = r_j - c$	<pre>  -   / \   CONST   </pre>
LOAD	$r_i = M[r_j + c]$	<pre>  MEM       +   / \   CONST   CONST       MEM       +   / \   CONST   CONST       MEM       CONST       MEM</pre>

Figure 3.8: Instruction DAG representation

```

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;
}

```

### Index/Chapter3\_2/Cpu0InstrInfo.td

```

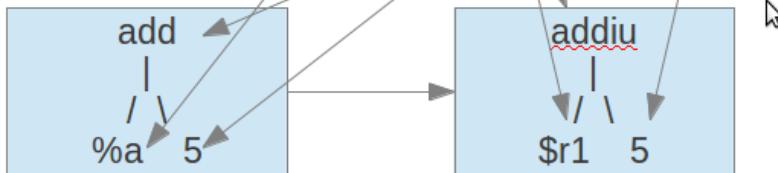
// Arithmetic and logical instructions with 2 register operands.
class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,
                  Operand Od, PatLeaf imm_type, RegisterClass RC> :
    FL<op, (outs RC:$ra), (ins RC:$rb, Od:$imm16),
        !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
        [(set RC:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {
    let isReMaterializable = 1;
}
...
def ADDiu    : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;

```

Figure 3.9 show how the pattern match work in the IR node **add** and instruction ADDiu defined in Cpu0InstrInfo.td. This example IR node “add %a, 5”, will be translated to “addiu %r1, 5” 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 in obj file of ELF format which will shown at later chapter). Similarly, the machine instruction DAG node LD and ST can be got from IR DAG node **load** and **store**.

```
class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,
    Operand Od, PatLeaf imm_type, RegisterClass RC> :
    FL<op, (outs RC:$ra), (ins RC:$rb, Od:$imm16),
    !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
    [(set RC:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {
    let isReMaterializable = 1;
}
def ADDiu : ArithLogicI<0x09, "addiu", add, simm16, immSExt16,
CPURegs>;
```

Tree



List

- $(\text{add } \%a, 5) \rightarrow (\text{addiu } \$r1, 5)$

Figure 3.9: Pattern match for ADDiu instruction and IR node add

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 node **fmul** and node **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 llvm SelectionDAG Optimization Phase (is part of Instruction Selection Process) prefered to translate this 2 IR DAG node (fmul %a, %b) (fadd %d, %c) into one machine instruction DAG node (**fmadd** %a, %c, %b), than translate them into 2 machine instruction nodes **fmul** and **fadd**.

```

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

```

As you can see, the IR notation representation is easier to read then llvm SSA IR form. So, we use the notation form 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 3.7 Instruction tree pattern 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;

```

## 3.6 Add Cpu0DAGToDAGISel class

The IR DAG to machine instruction DAG transformation is introduced in the previous section. 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 use the IR DAG node **store**, **ret**. Actually, it also use **add** for sp (stack point) register adjust. So, the definitions in Cpu0InstrInfo.td as follows is enough. IR DAG is defined in file include/llvm/Target/TargetSelectionDAG.td.

### Index/Chapter3\_2/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, isCodeGenOnly=1,  
    isBarrier=1, hasCtrlDep=1 in  
def RET : FJ <0x3c, (outs), (ins CPURegs:$target),  
    "ret\t$target", [(Cpu0Ret CPURegs:$target)], IIBranch>;  
  
//=====//
```

Add class Cpu0DAGToDAGISel (Cpu0ISelDAGToDAG.cpp) to CMakeLists.txt, and add following fragment to Cpu0TargetMachine.cpp,

### [Index/Chapter3\\_3/CMakeLists.txt](#)

```
add_llvm_target(...  
  ...  
  Cpu0ISelDAGToDAG.cpp  
  ...  
)
```

The following code in Cpu0TargetMachine.cpp will create a pass in instruction selection stage.

### [Index/Chapter3\\_3/Cpu0TargetMachine.cpp](#)

#### [Index/Chapter3\\_3/Cpu0ISelDAGToDAG.cpp](#)

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

### [Index/Chapter3\\_3/Cpu0InstrInfo.h](#)

```
class Cpu0InstrInfo : public Cpu0GenInstrInfo {  
  ...  
  virtual MachineInstr* emitFrameIndexDebugValue(MachineFunction &MF,  
                                                int FrameIx, uint64_t Offset,  
                                                const MDNode *MDPtr,  
                                                DebugLoc DL) const;
```

### [Index/Chapter3\\_3/Cpu0InstrInfo.cpp](#)

Build Chapter3\_3, run it, we find the error message in Chapter3\_2 is gone. The new error message for Chapter3\_3 as follows,

```
118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/  
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o  
ch3.cpu0.s  
...  
LLVM ERROR: Cannot select: 0x7f80f182d310: ch = <<Unknown Target Node #190>>
```

```
...
0x7f80f182d210: i32 = Register %LR [ID=4]
```

## 3.7 Handle return register Ir

[lbdex/Chapter3\\_4/Cpu0InstrFormats.td](#)

[lbdex/Chapter3\\_4/Cpu0InstrInfo.td](#)

[lbdex/Chapter3\\_4/Cpu0InstrInfo.h](#)

[lbdex/Chapter3\\_4/Cpu0InstrInfo.cpp](#)

To handle IR ret, these code in Cpu0InstrInfo.td do things as below.

1. Declare a pseudo node to take care the IR Cpu0ISD::Ret by the following code,

[lbdex/Chapter3\\_3/Cpu0InstrInfo.td](#)

2. After instruction selection, the Cpu0::Ret is replaced by Cpu0::RetLR as below. This effect came 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
```

3. Expand the Cpu0::RetLR into instruction **ret \$lr** in “Post-RA pseudo instruction expansion pass” stage by the code in Chapter3\_4/Cpu0InstrInfo.cpp as above. This stage is after the register allocation, so we can replace the V0 (\$r2) by LR (\$lr) without any side effect.
4. Print assembly or obj according the information (those \*.inc generated by TableGen from \*.td) generated by the following code at “Cpu0 Assembly Printer” stage.

**Index/Chapter2/Cpu0InstrInfo.td**

Table 3.1: Handle return register lr

Stage	Function
Write Code	Declare a pseudo node Cpu0::RetLR
•	for IR Cpu0::Ret;
Instruction selection	Cpu0::Ret is replaced by Cpu0::RetLR
Post-RA pseudo instruction expansion pass	Cpu0::RetLR -> ret \$lr
Cpu0 Assembly Printer	Print according “def RET”

Build Chapter3\_4, run it, we find the error message in Chapter3\_3 is gone. The new error message for Chapter3\_4 as follows,

```
118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/ bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o ch3.cpu0.s ... Target didn't implement TargetInstrInfo::storeRegToStackSlot! 1. Running pass 'Function Pass Manager' on module 'ch3.bc'. 2. Running pass 'Prologue/Epilogue Insertion & Frame Finalization' on function '@main' ...
```

## 3.8 Add Prologue/Epilogue functions

Following came 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 3.10 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.

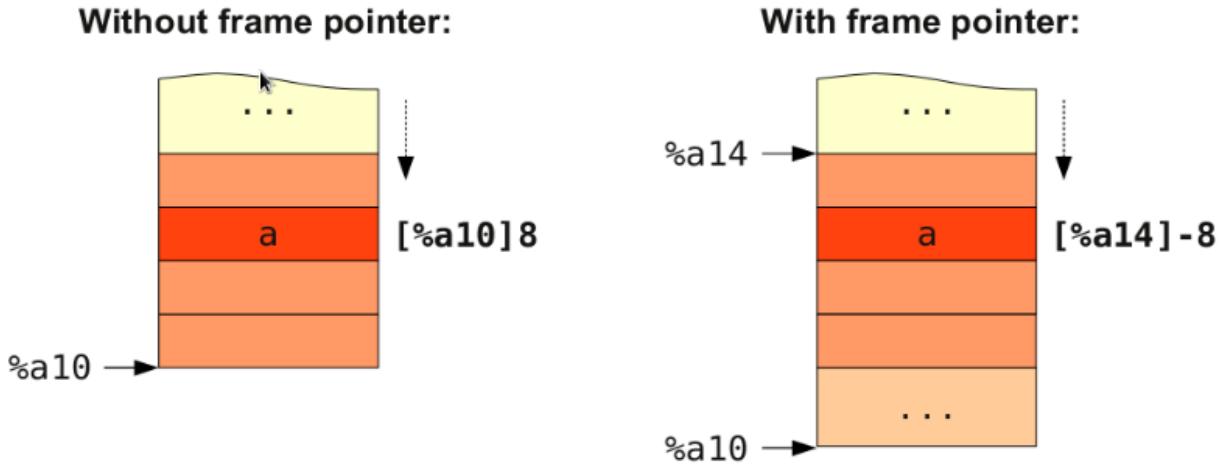


Figure 3.10: Addressing of a variable `a` located on the stack. If the stack frame has a variable size, slot must be addressed relative to the frame pointer

We will explain the Prologue and Epilogue further by example code. So for the following llvm IR code, Cpu0 back end 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
.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 IR and counting how many virtual registers is assigned to local variables. After that, it call emitPrologue(). This function will emit machine instructions to adjust sp (stack pointer register) for local variables since we don't use fp (frame pointer register). 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 machine instruction and real register allocated. It translate 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/Chapter3\\_1/Cpu0FrameLowering.h](#)

[Index/Chapter3\\_5/Cpu0FrameLowering.h](#)

```
void processFunctionBeforeCalleeSavedScan(MachineFunction &MF,
                                         RegScavenger *RS) const;
```

**lbdex/Chapter3\_5/Cpu0FrameLowering.cpp**

**lbdex/Chapter3\_5/Cpu0AnalyzeImmediate.h**

**lbdex/Chapter3\_5/Cpu0AnalyzeImmediate.cpp**

**lbdex/Chapter3\_5/Cpu0RegisterInfo.cpp**

Add these instructions to Cpu0InstrInfo.td which used in Prologue and Epilogue functions.

**lbdex/Chapter3\_5/Cpu0InstrInfo.td**

```
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);
]>; // lbd document - mark - def HI16
...
// 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);>];
...
// 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 RC:$ra), (ins RC:$rb, RC:$rc),
    !strconcat(instr_asm, "\t$ra, $rb, $rc"),
    [(set RC:$ra, (OpNode RC:$rb, RC:$rc))], itin> {
```

```

let shamt = 0;
let isCommutable = isComm; // e.g. add rb rc = add rc rb
let isReMaterializable = 1;
}
...
// 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 RC:$ra), (ins RC:$rb, ImmOpnd:$shamt),
    !strconcat(instr_asm, "\t$ra, $rb, $shamt"),
    [(set RC:$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>;

// Load Upper Immediate
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 neverHasSideEffects = 1;
let isReMaterializable = 1;
} // lbd document - mark - class LoadUpper
...
def ORI      : ArithLogicI<0x0d, "ori", or, uimm16, immZExt16, CPUREgs>;
def LUI      : LoadUpper<0x0f, "lui", CPUREgs, uimm16>;

/// Arithmetic Instructions (3-Operand, R-Type)
def ADDu    : ArithLogicR<0x11, "addu", add, IIAlu, CPUREgs, 1>;

/// Shift Instructions
def SHL      : shift_rotate_imm32<0x1e, 0x00, "shl", shl>;
...
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))>;

```

### Ibdex/Chapter3\_5/CMakeLists.txt

```

add_llvm_target(...  

...
Cpu0AnalyzeImmediate.cpp  

...
)

```

After add these Prologue and Epilogue functions, and build with Chapter3\_5/Cpu0. Now we are ready to compile our

example code ch3.bc into cpu0 assembly code. Following is the command and output file ch3.cpu0.s,

```
118-165-78-12:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm -debug ch3.bc -o -
Args: /Users/Jonathan/llvm/test/cmake_debug_build/bin/Debug/llc -march=cpu0
-relocation-model=pic -filetype=asm -debug ch3.bc -o ch3.cpu0.s
118-165-78-12:InputFiles Jonathan$ cat ch3.cpu0.s
.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 see what happens.

```
118-165-78-12:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm -debug ch3.bc -o -
Args: /Users/Jonathan/llvm/test/cmake_debug_build/bin/Debug/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/Chapter3\_5/Cpu0InstrInfo.td

```

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

```

At this point, we have translated the very simple `main()` function with return 0 single instruction. The `Cpu0AnalyzeImmediate.cpp` defined as above and the `Cpu0InstrInfo.td` instructions add as below, which takes care the 32 bits stack size adjustments.

#### Ibdex/Chapter3\_5/Cpu0InstrInfo.td

```

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);}]>;  

...  

// 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 RC:$ra, (ins RC:$rb, RC:$rc),  

        !strconcat(instr_asm, "\t$ra, $rb, $rc"),  

        [(set RC:$ra, (OpNode RC:$rb, RC:$rc))], itin> {  

    let shamt = 0;  

    let isCommutable = isComm; // e.g. add rb rc = add rc rb  

    let isReMaterializable = 1;  

}  

...  

// 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 RC:$ra), (ins RC:$rb, ImmOpnd:$shamt),  

        !strconcat(instr_asm, "\t$ra, $rb, $shamt"),  

        [(set RC:$ra, (OpNode RC:$rb, PF:$shamt))], IIAlu> {  

    let rc = isRotate;  

    let shamt = shamt;  

}  

// 32-bit shift instructions.  

class shift_rotate_imm32<bits<8> func, bits<4> isRotate, string instr_asm,  

    SDNode OpNode>:  

    shift_rotate_imm<func, isRotate, instr_asm, OpNode, immZExt5, shamt, CPURegs>;  

// Load Upper Immediate  

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 neverHasSideEffects = 1;

```

```

    let isReMaterializable = 1;
}

...
def ORI      : ArithLogicI<0x0d, "ori", or, uimm16, immZExt16, CPURegs>;
def LUI      : LoadUpper<0x0f, "lui", CPURegs, uimm16>;

/// Arithmetic Instructions (3-Operand, R-Type)
def ADDu    : ArithLogicR<0x11, "addu", add, IIAlu, CPURegs, 1>;

/// Shift Instructions
def SHL     : shift_rotate_imm32<0x1e, 0x00, "shl", shl>;
...

// Small immediates
...
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))>;

```

The Cpu0AnalyzeImmediate.cpp written in recursive and 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 (Cpu0 stack adjustment instructions after replace addiu and shl with lui instruction as below),

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 ~ 0x7fff	• 0x7fff	• addiu \$sp, \$sp, 32767;	• addiu \$sp, \$sp, 32767;
0x8000 ~ 0xfffff	• 0x8000	• addiu \$sp, \$sp, -32768;	• addiu \$1, \$zero, 1; • shl \$1, \$1, 16; • addiu \$1, \$1, -32768; • addu \$sp, \$sp, \$1;
x10000 ~ 0xffffffff	• 0x7fffffff	• addiu \$1, \$zero, -1; • shl \$1, \$1, 31; • addiu \$1, \$1, 1; • addu \$sp, \$sp, \$1;	• addiu \$1, \$zero, 1; • shl \$1, \$1, 31; • addiu \$1, \$1, -1; • addu \$sp, \$sp, \$1;
x10000 ~ 0xffffffff	• 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;

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 + 0xffffffff7) => \$1 = 0xffffffff7.
2. “shl \$1, \$1, 28;” => \$1 = 0x70000000.
3. “addiu \$1, \$1, -32768” => \$1 = (0x70000000 + 0xffff8000) => \$1 = 0x6fff8000.
4. “addu \$sp, \$sp, \$1” => \$sp = (0xa0008000 + 0x6fff8000) => \$sp = 0x10000000.

Verify with the Cpu0 Epilogue instructions with  $sp = 0x10000000$  and stack size =  $0x90008000$  as follows,

1. “addiu \$1, \$zero, -28671” => (\$1 = 0 + 0xffff9001) => \$1 = 0xffff9001.
2. “shl \$1, \$1, 16;” => \$1 = 0x90010000.
3. “addiu \$1, \$1, -32768” => \$1 = (0x90010000 + 0xffff8000) => \$1 = 0x90008000.
4. “addu \$sp, \$sp, \$1” => \$sp = (0x10000000 + 0x90008000) => \$sp = 0xa0008000.

The Cpu0AnalyzeImmediate::GetShortestSeq() will call Cpu0AnalyzeImmediate:: ReplaceADDiuSHLWithLUI() to replace addiu and shl with single instruction lui only. The effect as the following table.

Table 3.4: Cpu0 stack adjustment instructions after replace addiu and shl with lui instruction

old	x10000 ~ 0xffffffff	<ul style="list-style-type: none"> <li>• 0x90008000</li> </ul>	<ul style="list-style-type: none"> <li>• addiu \$1, \$zero, -9;</li> <li>• shl \$1, \$1, 28;</li> <li>• addiu \$1, \$1, -32768;</li> <li>• addu \$sp, \$sp, \$1;</li> </ul>	<ul style="list-style-type: none"> <li>• addiu \$1, \$zero, -28671;</li> <li>• shl \$1, \$1, 16</li> <li>• addiu \$1, \$1, -32768;</li> <li>• addu \$sp, \$sp, \$1;</li> </ul>
new	x10000 ~ 0xffffffff	<ul style="list-style-type: none"> <li>• 0x90008000</li> </ul>	<ul style="list-style-type: none"> <li>• lui \$1, 28671;</li> <li>• ori \$1, \$1, 32768;</li> <li>• addu \$sp, \$sp, \$1;</li> </ul>	<ul style="list-style-type: none"> <li>• lui \$1, 36865;</li> <li>• addiu \$1, \$1, -32768;</li> <li>• addu \$sp, \$sp, \$1;</li> </ul>

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.

## 3.9 Summary of this Chapter

Summary the functions for llvm backend stages as the following table.

```
118-165-79-200:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/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"> <li>• Cpu0TargetLowering::LowerFormalArguments</li> <li>• Cpu0TargetLowering::LowerReturn</li> </ul>
Instruction selection	<ul style="list-style-type: none"> <li>• Cpu0DAGToDAGISel::Select</li> </ul>
Prologue/Epilogue Insertion & Frame Finalization	<ul style="list-style-type: none"> <li>• Cpu0FrameLowering.cpp</li> <li>• Cpu0RegisterInfo::eliminateFrameIndex()</li> </ul>
Cpu0 Assembly Printer	<ul style="list-style-type: none"> <li>• Cpu0AsmPrinter.cpp -&gt; Cpu0MCInstLower.cpp</li> <li>• Cpu0InstPrinter.cpp</li> </ul>

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 inherit 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 LLVMBuilder.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 totally. Don't feel down with this result. In reality, write a back end is warm up slowly but run fast. 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 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 with comments. In next chapter, we will show you that add a new instruction support is as easy as 123.

# ARITHMETIC AND LOGIC LSUPPORT

This chapter adds more cpu0 arithmetic instructions support first. The logic operation “**not**” support and translation in section [Operator “not”](#) !. The section [Display llvm IR nodes with Graphviz](#) will show you the DAG optimization steps and their corresponding `l1c` display options. These DAG optimization steps result can be displayed by the graphic tool of Graphviz which supply very useful information with graphic view. You will appreciate Graphviz support in debug, we think. The section [Local variable pointer](#) introduce you the local variable pointer translation. Finally, section [Operator mod, %](#) take care the C operator %.

## 4.1 Arithmetic

The code added in Chapter4\_1/ to support arithmetic instructions as follows,

[lbdex/Chapter4\\_1/MCTargetDesc/Cpu0BaseInfo.h](#)

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

[lbdex/Chapter4\\_1/Cpu0InstrInfo.cpp](#)

[lbdex/Chapter4\\_1/Cpu0InstrInfo.h](#)

[lbdex/Chapter4\\_1/Cpu0InstrInfo.td](#)

```
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]>;
...
class shift_rotate_reg<bits<8> op, bits<4> isRotate, string instr_asm,
    SDNode OpNode, RegisterClass RC>:
    FA<op, (outs RC:$ra), (ins CPUREgs:$rb, RC:$rc),
```

```

    !strconcat(instr_asm, "\t$ra, $rb, $rc"),
    [(set RC:$ra, (OpNode RC:$rb, CPURegs:$rc))], IIAlu> {
let shamt = 0;
}

...
// 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 neverHasSideEffects = 1;
}

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 Hi/Lo
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 neverHasSideEffects = 1;
}

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 neverHasSideEffects = 1;
}

def SUBu    : ArithLogicR<0x12, "subu", sub, IIAlu, CPURegs>;
def ADD     : ArithLogicR<0x13, "add", add, IIAlu, CPURegs, 1>;
def SUB     : ArithLogicR<0x14, "sub", sub, IIAlu, CPURegs, 1>;
def MUL     : ArithLogicR<0x17, "mul", mul, IIImul, CPURegs, 1>;

/// Shift Instructions

```

```

// sra is IR node for ashr 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>;
...
// srl is IR node for lshr llvm IR instruction of .bc
def SHR      : shift_rotate_imm32<0x1f, 0x00, "shr", srl>;
def SRAV     : shift_rotate_reg<0x20, 0x00, "sra", sra, CPURegs>;
def SHLV     : shift_rotate_reg<0x21, 0x00, "shlv", shl, CPURegs>;
def SHRV     : shift_rotate_reg<0x22, 0x00, "shrv", srl, CPURegs>;

/// 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]>;

/// No operation
let addr=0 in
    def NOP     : FJ<0, (outs), (ins), "nop", [], IIAlu>;

```

### Index/Chapter4\_1/Cpu0ISelDAGToDAG.cpp

```

std::pair<SDNode*, SDNode*> SelectMULT(SDNode *N, unsigned Opc, DebugLoc dl,
                                         EVT Ty, bool HasLo, bool HasHi);
...
/// Select multiply instructions.
std::pair<SDNode*, SDNode*>
Cpu0DAGToDAGISel::SelectMULT(SDNode *N, unsigned Opc, DebugLoc 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);
}

/// Select instructions not customized! Used for
/// expanded, promoted and normal instructions
SDNode* Cpu0DAGToDAGISel::Select(SDNode *Node) {
    unsigned Opcode = Node->getOpcode();

```

```

DebugLoc dl = Node->getDebugLoc();
...
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);
    return SelectMULT(Node, MultOpc, dl, NodeTy, false, true).second;
}
...
}

```

### **lbdex/Chapter4\_1/Cpu0ISelLowering.cpp**

```

Cpu0TargetLowering::
Cpu0TargetLowering(Cpu0TargetMachine &TM)
: TargetLowering(TM, new TargetLoweringObjectFileELF()),
Subtarget(&TM.getSubtarget<Cpu0Subtarget>()) {
    ...
    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;
    DebugLoc dl = N->getDebugLoc();

    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)
{
    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/Chapter4\_1/Cpu0ISelLowering.h

```

namespace llvm {
    namespace Cpu0ISD {
        enum NodeType {
            ...
            // DivRem(u)
            DivRem,
            DivRemU
        };
    }
    ...
    virtual SDValue PerformDAGCombine(SDNode *N, DAGCombinerInfo &DCI) const;
    ...
}

```

### Ibdex/Chapter4\_1/Cpu0RegisterInfo.td

```

// Hi/Lo registers
def HI : Register<"HI">, DwarfRegNum<[18]>;
def LO : Register<"LO">, DwarfRegNum<[19]>;
...
// Hi/Lo Registers
def HILO : RegisterClass<"Cpu0", [i32], 32, (add HI, LO)>;

```

### Ibdex/Chapter4\_1/Cpu0Schedule.td

```
...
def IIHiLo      : InstrItinClass;
...
def Cpu0GenericItineraries : ProcessorItineraries<[ALU, IMULDIV], [], [
...
InstrItinData<IIHiLo      , [InstrStage<1, [IMULDIV]>]>,
...
]>;
```

### Ibdex/Chapter4\_1/Cpu0Schedule.td

```
...
def IIHiLo      : InstrItinClass;
def IIImul      : InstrItinClass;
def IIIdiv      : InstrItinClass;
...
// http://llvm.org/docs/doxygen/html/structllvm\_1\_1InstrStage.html
def Cpu0GenericItineraries : ProcessorItineraries<[ALU, IMULDIV], [], [
...
InstrItinData<IIHiLo      , [InstrStage<1, [IMULDIV]>]>,
InstrItinData<IIImul      , [InstrStage<17, [IMULDIV]>]>,
InstrItinData<IIIdiv      , [InstrStage<38, [IMULDIV]>]>
]>;
```

### 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 is separate. We think the cpu0 is the same even though no explanation in it's web site. So, these two function units can be executed at same time (instruction level parallelism). Reference <sup>1</sup> for instruction itineraries.

This version can process +, -, \*, <<, 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: ashx

**Example:** <result> = ashx i32 4, 1 ; yields {i32}:result = 2

<result> = ashx i8 -2, 1 ; yields {i8}:result = -1

<result> = ashx i32 1, 32 ; undefined

---

The 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

<sup>1</sup> [http://llvm.org/docs/doxygen/html/structllvm\\_1\\_1InstrStage.html](http://llvm.org/docs/doxygen/html/structllvm_1_1InstrStage.html)

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, node **srl** is defined for lshr instruction (I don't know why don't use ashr and lshr as the IR node name directly). Summary as the Table: C operator  $\gg$  implementation.

Table 4.1: C operator  $\gg$  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 $x \gg 1$	0xfffffffffe i.e. -2	0xfffffffffe i.e. -2
signed example after $x \gg 1$	0x7fffffff i.e. 2G-1	0xffffffff i.e. -1
unsigned example before $x \gg 1$	0xfffffffffe i.e. 4G-2	0xfffffffffe i.e. 4G-2
unsigned example after $x \gg 1$	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

If we consider the  $x \gg 1$  definition is  $x = x/2$  for compiler implementation. As you can see from Table: C operator  $\gg$  implementation, **lshr** is failed on some signed value (such as -2). In the same way, **ashr** is failed on some unsigned value (such as 4G-2). So, in order to satisfy this definition in both signed and unsigned integer 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 the unsigned  $x=1G$  but failed on signed  $x=1G$ . It's fine since the 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 the **lshr** satisfy  $x = x*2$  for  $x \ll 1$  when the x result is not out of range, no matter operand x is signed or unsigned integer.

Microsoft implementation references as <sup>2</sup>.

The sub-section ““ashr‘ Instruction” and sub-section ““lshr‘ Instruction” of <sup>3</sup>.

The sra, shlv and shrv are for two virtual registers instructions while the sra, ... are for 1 virtual registers and 1 constant input operands.

Now, let's build Chapter4\_1/ and run with input file ch4\_1.cpp as follows,

### Ibdex/InputFiles/ch4\_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
}
```

```
118-165-78-12:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch4_1.cpp -emit-llvm -o ch4_1.bc
118-165-78-12:InputFiles Jonathan$ llvm-dis ch4_1.bc -o -
...
; Function Attrs: nounwind uwtable
define i32 @_Z9test_mathv() #0 {
entry:
%a = alloca i32, align 4
%b = alloca i32, align 4
%a1 = 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
%h = alloca i32, align 4
%i = alloca i32, align 4
%f1 = alloca i32, align 4
%g1 = alloca i32, align 4
%h1 = alloca i32, align 4
%i1 = alloca i32, align 4
```

---

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

<sup>3</sup> <http://llvm.org/docs/LangRef.html>.

```
store i32 5, i32* %a, align 4
store i32 2, i32* %b, align 4
store i32 -5, i32* %a1, align 4
%0 = load i32* %a, align 4
%1 = load i32* %b, align 4
%add = add nsw i32 %0, %1
store i32 %add, i32* %c, align 4
%2 = load i32* %a, align 4
%3 = load i32* %b, align 4
%sub = sub nsw i32 %2, %3
store i32 %sub, i32* %d, align 4
%4 = load i32* %a, align 4
%5 = load i32* %b, align 4
%mul = mul nsw i32 %4, %5
store i32 %mul, i32* %e, align 4
%6 = load i32* %a, align 4
%shl = shl i32 %6, 2
store i32 %shl, i32* %f, align 4
%7 = load i32* %a1, align 4
%shl1 = shl i32 %7, 1
store i32 %shl1, i32* %f1, align 4
%8 = load i32* %a, align 4
%shr = ashr i32 %8, 2
store i32 %shr, i32* %g, align 4
%9 = load i32* %a1, align 4
%shr2 = lshr i32 %9, 30
store i32 %shr2, i32* %g1, align 4
%10 = load i32* %a, align 4
%shl3 = shl i32 1, %10
store i32 %shl3, i32* %h, align 4
%11 = load i32* %b, align 4
%shl4 = shl i32 1, %11
store i32 %shl4, i32* %h1, align 4
%12 = load i32* %a, align 4
%shr5 = ashr i32 128, %12
store i32 %shr5, i32* %i, align 4
%13 = load i32* %b, align 4
%14 = load i32* %a, align 4
%shr6 = ashr i32 %13, %14
store i32 %shr6, i32* %i1, align 4
%15 = load i32* %c, align 4
%16 = load i32* %d, align 4
%add7 = add nsw i32 %15, %16
%17 = load i32* %e, align 4
%add8 = add nsw i32 %add7, %17
%18 = load i32* %f, align 4
%add9 = add nsw i32 %add8, %18
%19 = load i32* %f1, align 4
%add10 = add nsw i32 %add9, %19
%20 = load i32* %g, align 4
%add11 = add nsw i32 %add10, %20
%21 = load i32* %g1, align 4
%add12 = add nsw i32 %add11, %21
%22 = load i32* %h, align 4
%add13 = add nsw i32 %add12, %22
%23 = load i32* %h1, align 4
%add14 = add nsw i32 %add13, %23
%24 = load i32* %i, align 4
```

```
%add15 = add nsw i32 %add14, %24
%25 = load i32* %i1, align 4
%add16 = add nsw i32 %add15, %25
ret i32 %add16
}

118-165-78-12:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch4_1.bc -o -
.section .mdebug.abi32
.previous
.file "ch4_1.bc"
.text
.globl _Z9test_mathv
.align 2
.type _Z9test_mathv,@function
.ent _Z9test_mathv           # @_Z9test_mathv
_Z9test_mathv:
.frame $fp,56,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:                      # %entry
addiu $sp, $sp, -56
addiu $2, $zero, 5
st $2, 52($fp)
addiu $2, $zero, 2
st $2, 48($fp)
addiu $2, $zero, -5
st $2, 44($fp)
ld $2, 48($fp)
ld $3, 52($fp)
addu $2, $3, $2
st $2, 40($fp)
ld $2, 48($fp)
ld $3, 52($fp)
subu $2, $3, $2
st $2, 36($fp)
ld $2, 48($fp)
ld $3, 52($fp)
mul $2, $3, $2
st $2, 32($fp)
ld $2, 52($fp)
shl $2, $2, 2
st $2, 28($fp)
ld $2, 44($fp)
shl $2, $2, 1
st $2, 12($fp)
ld $2, 52($fp)
sra $2, $2, 2
st $2, 24($fp)
ld $2, 44($fp)
shr $2, $2, 30
st $2, 8($fp)
addiu $2, $zero, 1
ld $3, 52($fp)
shlv $3, $2, $3
st $3, 20($fp)
ld $3, 48($fp)
```

```

shlv $2, $2, $3
st $2, 4($fp)
addiu $2, $zero, 128
ld $3, 52($fp)
shrv $2, $2, $3
st $2, 16($fp)
ld $2, 52($fp)
ld $3, 48($fp)
srav $2, $3, $2
st $2, 0($fp)
addiu $sp, $sp, 56
ret $lr
.set macro
.set reorder
.end _Z9test_mathv
$tmp1:
.size _Z9test_mathv, ($tmp1)-_Z9test_mathv

```

#### 4.1.2 Display llvm IR nodes with Graphviz

The previous section, display the DAG translation process in text on terminal by `llc -debug` option. The `llc` also support the graphic display. The [section Install other tools on iMac](#) mentioned the web for `llc` graphic display information. The `llc` graphic display with tool Graphviz is introduced in this section. The graphic display is more readable by eye than display text in terminal. It's not necessary, 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 <sup>4</sup> 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.

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,

<sup>4</sup> <http://llvm.org/docs/CodeGenerator.html>

```
118-165-12-177:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/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:InputFiles Jonathan$ Graphviz /tmp/llvm_84ibpm/dag.main.dot
```

It will show the /tmp/llvm\_84ibpm/dag.main.dot as [Figure 4.1](#).

From [Figure 4.1](#), we can see the -view-dag-combine1-dags option is for Initial selection DAG. We list the other view options and their corresponding DAG translation stage 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 DAG before instruction selection. The backend programmer need to know what is the DAG for writing the pattern match instruction in target description file .td.

### 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,

##### [Index/InputFiles/ch4\\_2.cpp](#)

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

    b = (b+1)%12;

    return b;
}

...
define i32 @main() nounwind ssp {
    entry:
    %retval = alloca i32, align 4
    %b = alloca i32, align 4
    store i32 0, i32* %retval
    store i32 11, i32* %b, align 4
    %0 = load i32* %b, align 4
    %add = add nsw i32 %0, 1
    %rem = srem i32 %add, 12
    store i32 %rem, i32* %b, align 4
```

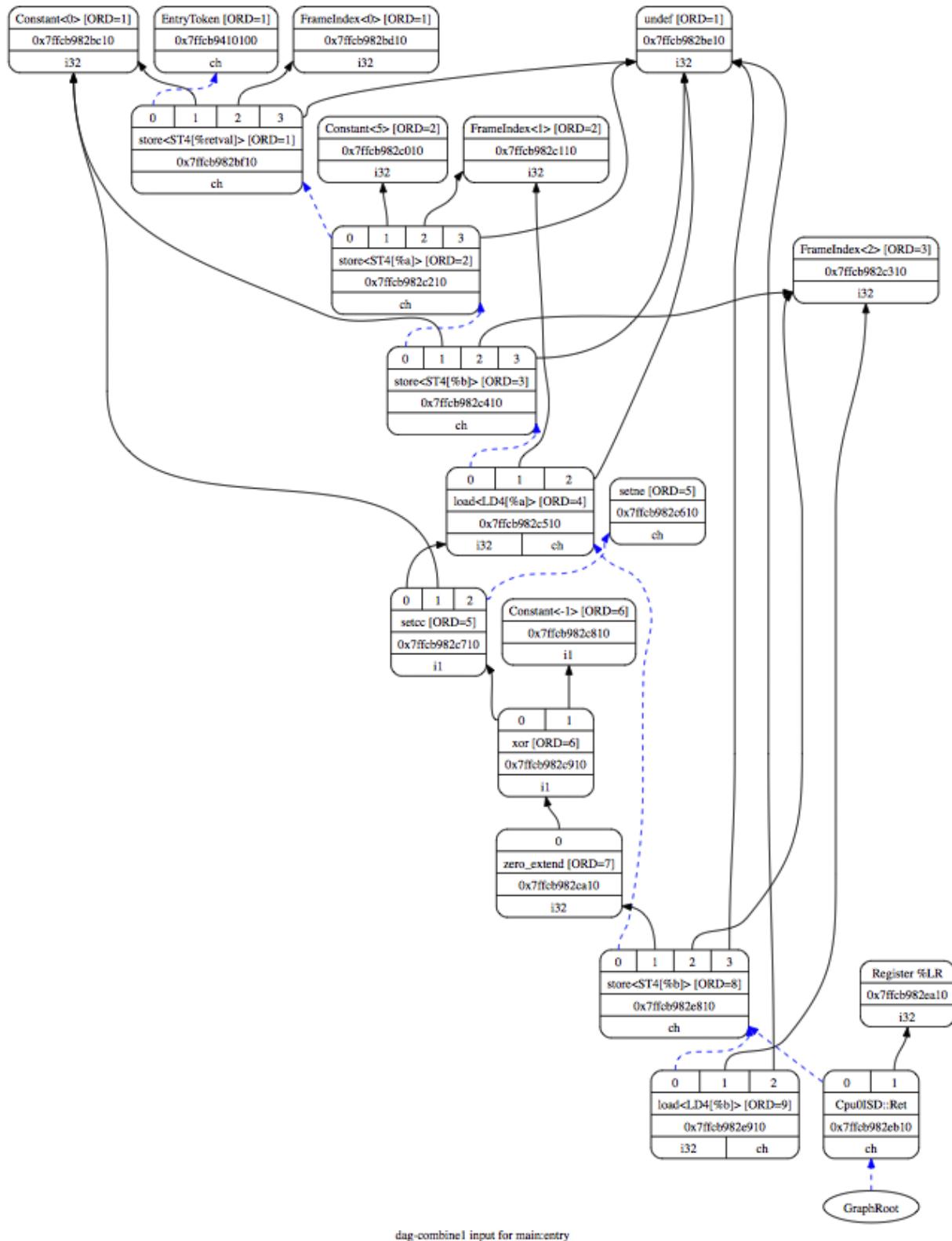


Figure 4.1: llc option -view-dag-combine1-dags graphic view

```
%1 = load i32* %b, align 4
ret i32 %1
}
```

LLVM **srem** is the IR corresponding “%”, reference sub-section “srem instruction” of <sup>3</sup>. 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 `llc` option `-view-isel-dags` as below, will get the following error message and the llvm DAG of [Figure 4.2](#) below.

```
118-165-79-37:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/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 replace srem divide operation with multiply operation in DAG optimization because DIV operation cost more in time than MUL. For example code “**int b = 11; b=(b+1)%12;**”, it translate into [Figure 4.2](#). We verify the result and explain it by calculate 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 size 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,

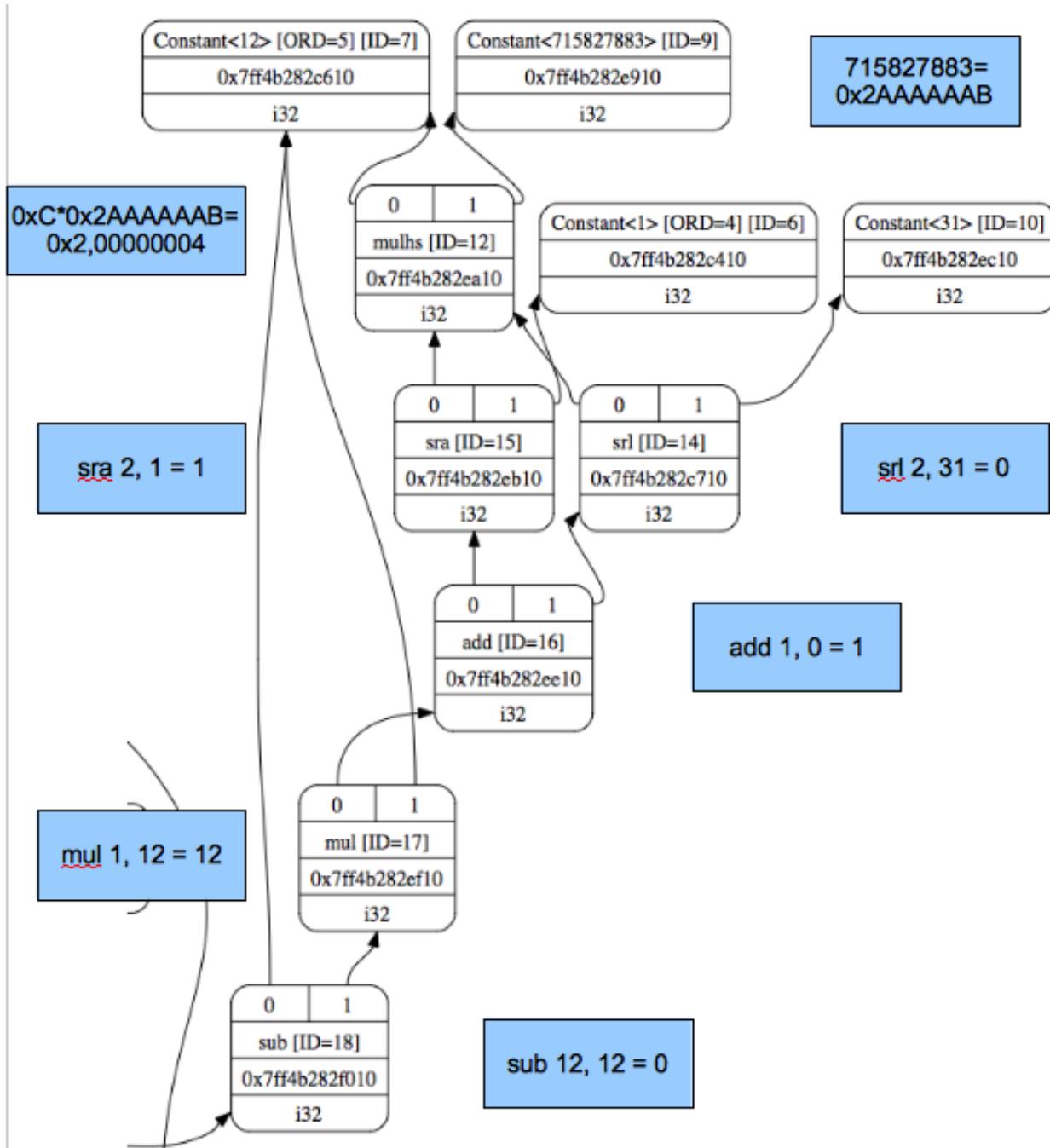


Figure 4.2: ch4\_2.bc DAG

**lbdex/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/Chapter4\_1/Cpu0ISelDAGToDAG.cpp**

```
#if 0
/// Select multiply instructions.
std::pair<SDNode*, SDNode*>
Cpu0DAGToDAGISel::SelectMULT(SDNode *N, unsigned Opc, DebugLoc 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;
#endif
    #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. Similarly, SMMUL get the high word of multiply result.

Follows is the result of run above changes with ch4\_2.bc.

```
118-165-66-82:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_
debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch4_2.bc -o -
```

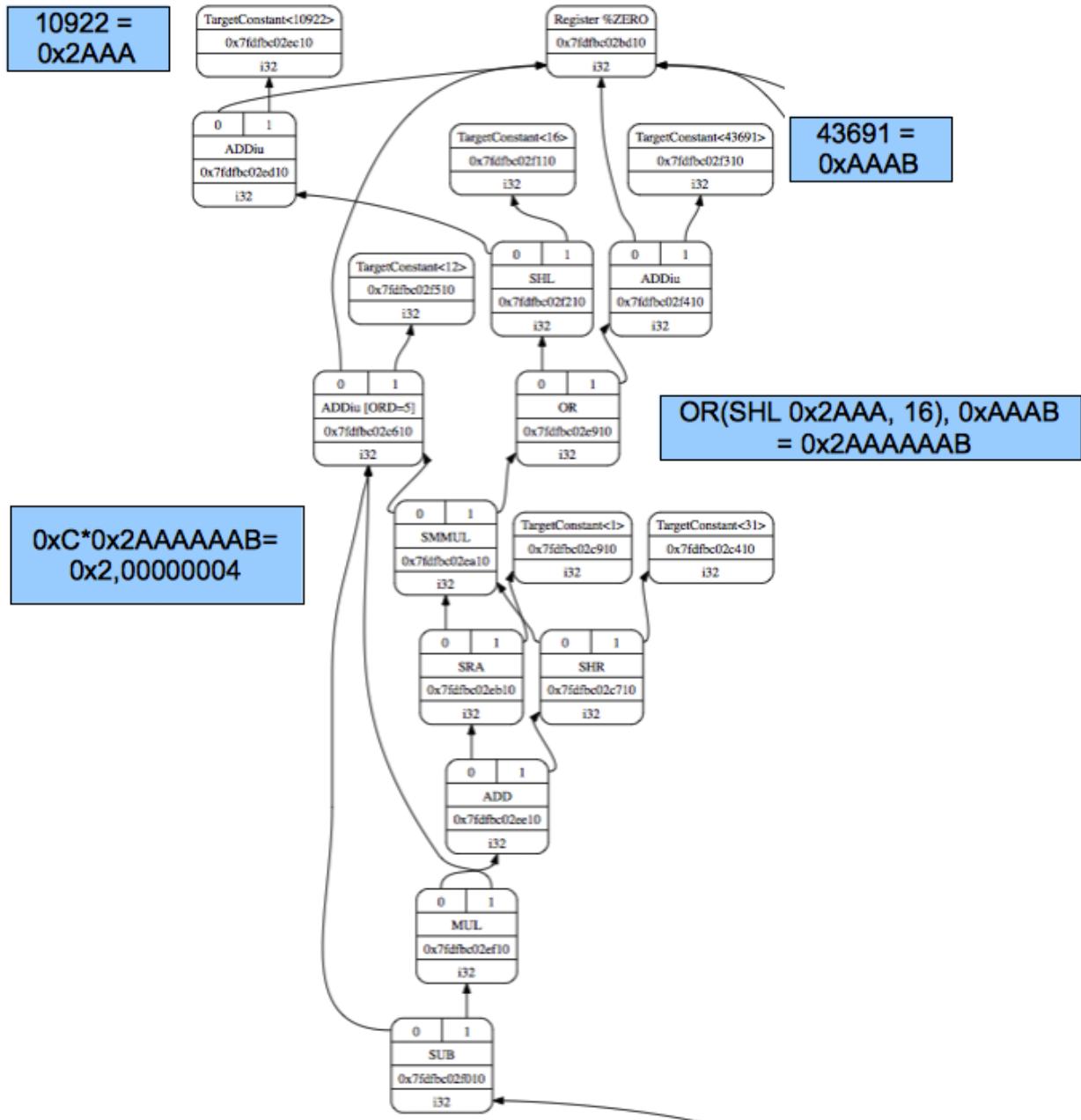


Figure 4.3: Translate ch4\_2.bc into cpu0 backend DAG

```

.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.

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

### Mips solution

Mips use MULT instruction and save the high & low part to register HI and LO. After that, use mfhi/mflo to move register HI/LO to your general purpose register. ARM SMMUL is fast if you only need the HI part of result (it ignore the LO part of operation). ARM also provide 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 which only get the LO part of result. Chapter4\_1/ is implemented with Mips MULT style. We choose it as the implementation of this book to add instructions as less as possible. This approach is better for Cpu0 to keep it as a tutorial architecture for school teaching purpose material, and apply Cpu0 as an engineer learning materials in compiler, system program and verilog CPU hardware design. The MULT, MULTu, MFHI, MFLO, MTHI, MTLO added in Chapter4\_1/Cpu0InstrInfo.td; HI, LO register 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 which are used in Chapter4\_1/ came 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 expand and promote type will call Cpu0DAGToDAGISel::Select() during instruction selection of DAG translation. In SelectMULT() which called by Select(), it return the HI part of multiplication result to HI register, for IR operations of mulhs or mulhu. After that, MFHI instruction move 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 result of compile ch4\_2.bc.

```
118-165-66-82:InputFiles 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)
addiu $sp, $sp, 8
ret $lr
.set macro
.set reorder
.end _Z8test_modv
$tmp1:
.size _Z8test_modv, ($tmp1)-_Z8test_modv
```

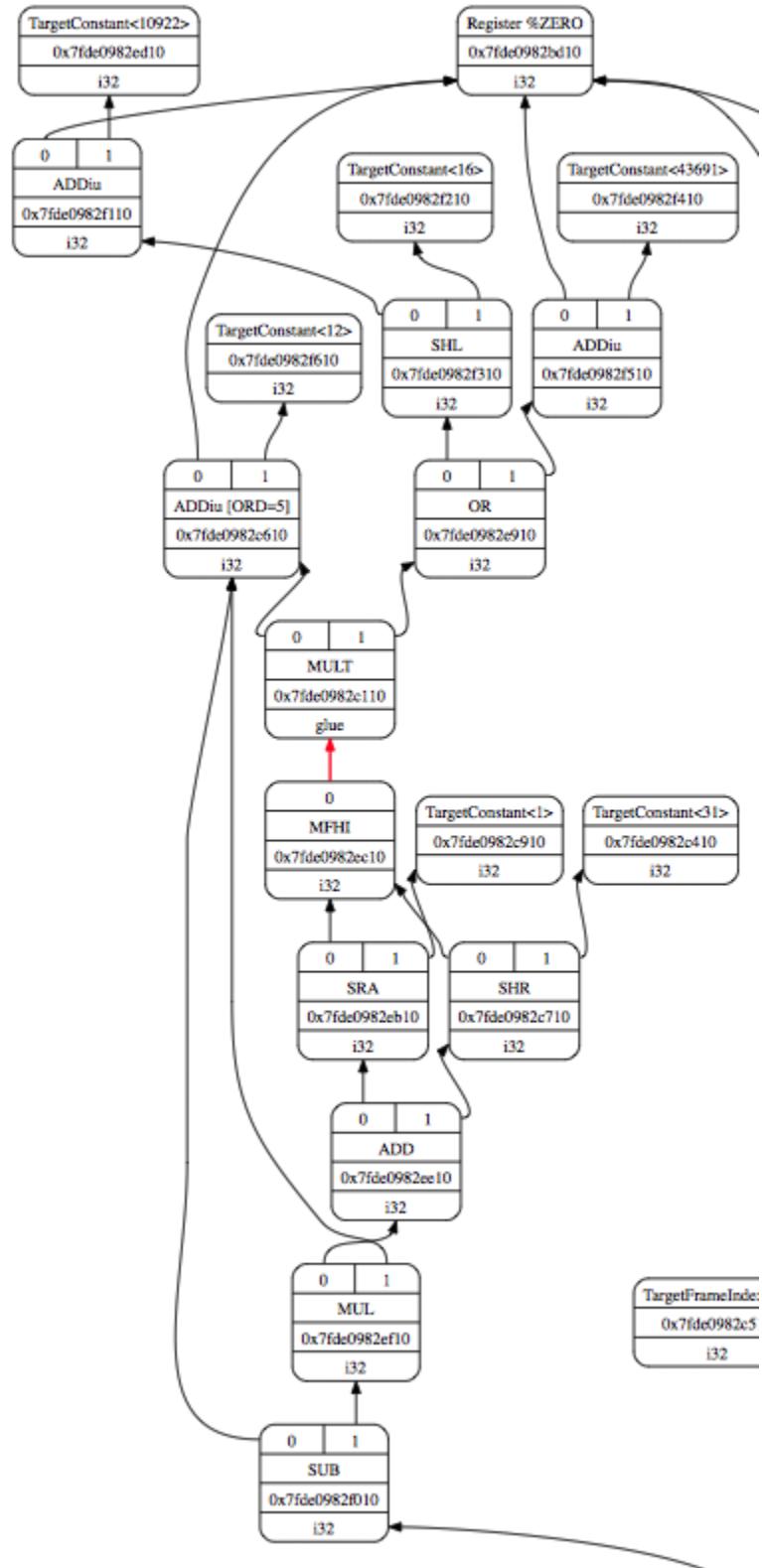


Figure 4.4: DAG for ch4\_2.bc with Mips style MULT

## Full support %, and /

The sensitive readers may find the llvm using “**multiplication**” instead of “**div**” to get the “%” result just because our example use constant as divider, “**(b+1)%12**” in our example. If programmer use variable as the divider like “**(b+1)%a**”, then what will happen in our code. The answer is our code will has error to take care this.

Cpu0 just like Mips use LO and HI registers to hold the “**quotient**” and “**remainder**”. And use instructions “**mflo**” and “**mfhi**” to get the result from LO or HI registers. With this solution, the “**c = a / b**” can be got by “**div a, b**” and “**mflo c**”; the “**c = a % b**” can be got by “**div a, b**” and “**mfhi c**”.

To support 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.

IR instruction **sdiv** stand for signed div while **udiv** is for unsigned div.

Run with ch4\_2\_2.cpp can get the “div” result for operator “%” but it cannot be compiled at this point. It need the function call argument support in Chapter 8 of Function call. If run with ch4\_2\_1.cpp as below, cannot get the “div” for operator “%”. It still use “**multiplication**” instead of “**div**” in ch4\_2\_1.cpp because llvm do “**Constant Propagation Optimization**” on this. The ch4\_2\_2.cpp can get the “div” for “%” result since it make the llvm “**Constant Propagation Optimization**” useless in this.

### lbdex/InputFiles/ch4\_2\_1.cpp

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

    b = (b+1)%a;

    return b;
}
```

### lbdex/InputFiles/ch4\_2\_2.cpp

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

    b = (b+1)%c;

    return b;
}
```

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

```
div $zero, $3, $2
mflo $2
...
```

To explain how work with “**div**”, let’s run Chapter8\_4 with ch4\_2\_2.cpp as follows,

```
118-165-83-58:InputFiles 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:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/bin/
Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm -debug ch4_2_2.bc -o -
Args: /Users/Jonathan/llvm/test/cmake_debug_build/bin/Debug/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 DAG translation message from `llc -debug`, it 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. <sup>5</sup> and <sup>6</sup>. Item 3 is triggered by code “`setTargetDAGCombine(ISD::SDIVREM);`” in `Cpu0ISelLowering.cpp`. Item 4 is triggered by `PerformDivRemCombine()` which called by `PerformDAGCombine()` since the % corresponding **srem** make the “N->hasAnyUseOfValue(1)” to true in `PerformDivRemCombine()`. Then, it create “`CopyFromReg 0x7fd25b410e18, 0x7fd25b830710, 0x7fd25b830910`”. When use “%” in C, it will make “N->hasAnyUseOfValue(0)” to true. For sdivrem, **sdiv** make “N->hasAnyUseOfValue(0)” true while **srem** make “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** to **div**; and “**CopyFromReg 0x7fd25b410e18, Register %H, 0x7fd25b830910**” to **mfhi**.

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

<sup>5</sup> <http://llvm.org/docs/WritingAnLLVMBBackend.html#expand>

<sup>6</sup> <http://llvm.org/docs/CodeGenerator.html#selectiondag-legalizetypes-phase>

## 4.2 Logic

Chapter4\_2 support logic operators `&`, `|`, `^`, `!`, `==`, `!=`, `<`, `<=`, `>` and `>=`. They are trivial and easy. Listing the added code with comment and table for these operators IR, DAG and instructions as below. You check them with the run result of bc and asm instructions for ch4\_5.cpp as below.

### lbdex/Chapter4\_2/Cpu0InstrInfo.cpp

```
void Cpu0InstrInfo::  
copyPhysReg(MachineBasicBlock &MBB,  
            MachineBasicBlock::iterator I, DebugLoc DL,  
            unsigned DestReg, unsigned SrcReg,  
            bool KillSrc) const {  
    ...  
    if (Cpu0::CPUREgsRegClass.contains(DestReg)) { // Copy to CPU Reg.  
        ...  
        if (SrcReg == Cpu0::SW)  
            Opc = Cpu0::MFSW, SrcReg = 0;  
    }  
    else if (Cpu0::CPUREgsRegClass.contains(SrcReg)) { // Copy from CPU Reg.  
        ...  
        if (DestReg == Cpu0::SW)  
            Opc = Cpu0::MTSW, DestReg = 0;  
    }  
    ...  
}
```

### lbdex/Chapter4\_2/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:$rb),  
  !strconcat(instr_asm, "\t$rc, $ra, $rb"), [], itin> {  
let rc = 0;  
let shamt = 0;  
let isCommutable = isComm;  
}  
...  
// Move from SW  
class MoveFromSW<bits<8> op, string instr_asm, RegisterClass RC,  
               list<Register> UseRegs>:  
FL<op, (outs RC:$ra), (ins),  
  !strconcat(instr_asm, "\t$ra"), [], IIAlu> {  
let rb = 0;  
let imm16 = 0;  
let Uses = UseRegs;  
let neverHasSideEffects = 1;  
}  
...  
// Move to SW  
class MoveToSW<bits<8> op, string instr_asm, RegisterClass RC,  
               list<Register> DefRegs>:  
FL<op, (outs), (ins RC:$ra),
```

```

    !strconcat(instr_asm, "\t$ra"), [], IIAlu> {
let rb = 0;
let imm16 = 0;
let Defs = DefRegs;
let neverHasSideEffects = 1;
}
...
/// Arithmetic Instructions (ALU Immediate)
...
def ANDi      : ArithLogicI<0x0c, "andi", and, uimm16, immZExt16, CPURegs>;
...
def XORi      : ArithLogicI<0x0e, "xori", xor, uimm16, immZExt16, CPURegs>;
...
/// Arithmetic Instructions (3-Operand, R-Type)
def CMP       : CmpInstr<0x10, "cmp", IIAlu, CPURegs, SR, 0>;
...
def AND       : ArithLogicR<0x18, "and", and, IIAlu, CPURegs, 1>;
def OR        : ArithLogicR<0x19, "or", or, IIAlu, CPURegs, 1>;
def XOR       : ArithLogicR<0x1a, "xor", xor, IIAlu, CPURegs, 1>;
...
def MFSW      : MoveFromSW<0x50, "mfsw", CPURegs, [SW]>;
def MTSW      : MoveToSW<0x51, "mtsw", CPURegs, [SW]>;
...
def : Pat<(not CPURegs:$in),
// 1: in == 0; 0: in != 0
(XORi CPURegs:$in, 1)>;
...
// setcc patterns
multiclass SeteqPats<RegisterClass RC> {
// a == b
def : Pat<(seteq RC:$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:$rhs), 2), 1), 1)>;
}
...
// a < b
multiclass SetltPats<RegisterClass RC> {
def : Pat<(setlt RC:$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:$rhs), 1)>;
}
...
// a <= b
multiclass SetlePats<RegisterClass RC> {
def : Pat<(setle RC:$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 SetgtPats<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 SetgePats<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)>;
}

defm : SeteqPats<CPUREgs>;
defm : SetltPats<CPUREgs>;
defm : SetlePats<CPUREgs>;
defm : SetgtPats<CPUREgs>;
defm : SetgePats<CPUREgs>;

```

### Index/Chapter4\_2/Cpu0ISelLowering.cpp

```

Cpu0TargetLowering::
Cpu0TargetLowering(Cpu0TargetMachine &TM)
    : TargetLowering(TM, new Cpu0TargetObjectFile(),
        Subtarget(&TM.getSubtarget<Cpu0Subtarget>()) {
    ...
    // Cpu0 doesn't have sext_inreg, replace them with shl/sra.
    setOperationAction(ISO::SIGN_EXTEND_INREG, MVT::il, Expand);
    ...
}

```

### Index/InputFiles/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:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch4_5.cpp -emit-llvm -o ch4_5.bc
114-43-204-152:InputFiles Jonathan$ llvm-dis ch4_5.bc -o -
...
; Function Attrs: nounwind uwtable
define i32 @_Z16test_andorxornotv() #0 {
entry:
%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
store i32 5, i32* %a, align 4
store i32 3, i32* %b, align 4
store i32 0, i32* %c, align 4
store i32 0, i32* %d, align 4
store i32 0, i32* %e, align 4
%0 = load i32* %a, align 4
%1 = load i32* %b, align 4
%and = and i32 %0, %1
store i32 %and, i32* %c, align 4
%2 = load i32* %a, align 4
%3 = load i32* %b, align 4
%or = or i32 %2, %3
store i32 %or, i32* %d, align 4
%4 = load i32* %a, align 4
%5 = load i32* %b, align 4
%xor = xor i32 %4, %5
store i32 %xor, i32* %e, align 4
%6 = load i32* %a, align 4
%tobool = icmp ne i32 %6, 0
%lnot = xor i1 %tobool, true
%conv = zext i1 %lnot to i32
store i32 %conv, i32* %b, align 4
%7 = load i32* %c, align 4
%8 = load i32* %d, align 4
%add = add nsw i32 %7, %8
%9 = load i32* %e, align 4
%add1 = add nsw i32 %add, %9
%10 = load i32* %b, align 4
%add2 = add nsw i32 %add1, %10
ret i32 %add2
}

; Function Attrs: nounwind uwtable
define i32 @_Z10test_setxxv() #0 {
```

```
entry:
  %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
  %h = alloca i32, align 4
  store i32 5, i32* %a, align 4
  store i32 3, i32* %b, align 4
  %0 = load i32* %a, align 4
  %1 = load i32* %b, align 4
  %cmp = icmp eq i32 %0, %1
  %conv = zext i1 %cmp to i32
  store i32 %conv, i32* %c, align 4
  %2 = load i32* %a, align 4
  %3 = load i32* %b, align 4
  %cmp1 = icmp ne i32 %2, %3
  %conv2 = zext i1 %cmp1 to i32
  store i32 %conv2, i32* %d, align 4
  %4 = load i32* %a, align 4
  %5 = load i32* %b, align 4
  %cmp3 = icmp slt i32 %4, %5
  %conv4 = zext i1 %cmp3 to i32
  store i32 %conv4, i32* %e, align 4
  %6 = load i32* %a, align 4
  %7 = load i32* %b, align 4
  %cmp5 = icmp sle i32 %6, %7
  %conv6 = zext i1 %cmp5 to i32
  store i32 %conv6, i32* %f, align 4
  %8 = load i32* %a, align 4
  %9 = load i32* %b, align 4
  %cmp7 = icmp sgt i32 %8, %9
  %conv8 = zext i1 %cmp7 to i32
  store i32 %conv8, i32* %g, align 4
  %10 = load i32* %a, align 4
  %11 = load i32* %b, align 4
  %cmp9 = icmp sge i32 %10, %11
  %conv10 = zext i1 %cmp9 to i32
  store i32 %conv10, i32* %h, align 4
  %12 = load i32* %c, align 4
  %13 = load i32* %d, align 4
  %add = add nsw i32 %12, %13
  %14 = load i32* %e, align 4
  %add11 = add nsw i32 %add, %14
  %15 = load i32* %f, align 4
  %add12 = add nsw i32 %add11, %15
  %16 = load i32* %g, align 4
  %add13 = add nsw i32 %add12, %16
  %17 = load i32* %h, align 4
  %add14 = add nsw i32 %add13, %17
  ret i32 %add14
}
```

```
114-43-204-152:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch4_5.bc -o -
-filetype=asm ch4_5.bc -o -
```

```
.section .mdebug.abi32
.previous
.file "ch4_5.bc"
.text
.globl _Z16test_andorxornotv
.align 2
.type _Z16test_andorxornotv,@function
.ent _Z16test_andorxornotv      # @_Z16test_andorxornotv
_Z16test_andorxornotv:
.frame $fp,24,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:                                # %entry
addiu $sp, $sp, -24
addiu $2, $zero, 5
st $2, 20($fp)
addiu $2, $zero, 3
st $2, 16($fp)
addiu $2, $zero, 0
st $2, 12($fp)
st $2, 8($fp)
st $2, 4($fp)
ld $3, 16($fp)
ld $4, 20($fp)
and $3, $4, $3
st $3, 12($fp)
ld $3, 16($fp)
ld $4, 20($fp)
or $3, $4, $3
st $3, 8($fp)
ld $3, 16($fp)
ld $4, 20($fp)
xor $3, $4, $3
st $3, 4($fp)
ld $3, 20($fp)
cmp $sw, $3, $2
mfsw $2
andi $2, $2, 2
shr $2, $2, 1
andi $2, $2, 1
st $2, 16($fp)
addiu $sp, $sp, 24
ret $lr
.set macro
.set reorder
.end _Z16test_andorxornotv
$tmp1:
.size _Z16test_andorxornotv, ($tmp1)-_Z16test_andorxornotv

.globl _Z10test_setxxv
.align 2
.type _Z10test_setxxv,@function
.ent _Z10test_setxxv      # @_Z10test_setxxv
_Z10test_setxxv:
.frame $fp,32,$lr
.mask 0x00000000,0
.set noreorder
```

```
.set nomacro
# BB#0:                                     # %entry
addiu $sp, $sp, -32
addiu $2, $zero, 5
st $2, 28($fp)
addiu $2, $zero, 3
st $2, 24($fp)
ld $3, 28($fp)
cmp $sw, $3, $2
mfsw $2
andi $2, $2, 2
shr $2, $2, 1
andi $2, $2, 1
st $2, 20($fp)
ld $2, 24($fp)
ld $3, 28($fp)
cmp $sw, $3, $2
mfsw $2
andi $2, $2, 2
shr $2, $2, 1
xori $2, $2, 1
andi $2, $2, 1
st $2, 16($fp)
ld $2, 24($fp)
ld $3, 28($fp)
cmp $sw, $3, $2
mfsw $2
andi $2, $2, 1
andi $2, $2, 1
st $2, 12($fp)
ld $2, 28($fp)
ld $3, 24($fp)
cmp $sw, $3, $2
mfsw $2
andi $2, $2, 1
xori $2, $2, 1
andi $2, $2, 1
st $2, 8($fp)
ld $2, 28($fp)
ld $3, 24($fp)
cmp $sw, $3, $2
mfsw $2
andi $2, $2, 1
andi $2, $2, 1
st $2, 4($fp)
ld $2, 24($fp)
ld $3, 28($fp)
cmp $sw, $3, $2
mfsw $2
andi $2, $2, 1
xori $2, $2, 1
andi $2, $2, 1
st $2, 0($fp)
addiu $sp, $sp, 32
ret $lr
.set macro
.set reorder
.end _Z10test_setxxv
```

```
$tmp3:
.size _Z10test_setxxv, ($tmp3)-_Z10test_setxxv
```

Table 4.5: Logic operators

C	.bc	Optimized legalized selection DAG	Cpu0
&, &&	and	and	and
!,	or	or	or
^	xor	xor	xor
!	<ul style="list-style-type: none"> <li>%tobool = icmp ne i32 %6, 0</li> <li>%lnot = xor i1 %tobool, true</li> <li>%conv = zext i1 %lnot to i32</li> </ul>	<ul style="list-style-type: none"> <li>%lnot = (setcc %tobool, 0, seteq)</li> <li>%conv = (and %lnot, 1)</li> <li>•</li> </ul>	<ul style="list-style-type: none"> <li>xor \$3, \$4, \$3</li> </ul>
==	<ul style="list-style-type: none"> <li>%cmp = icmp eq i32 %0, %1</li> <li>%conv = zext i1 %cmp to i32</li> </ul>	<ul style="list-style-type: none"> <li>(setcc %0, %1, seteq)</li> </ul>	<ul style="list-style-type: none"> <li>cmp \$sw, \$3, \$2</li> <li>mfsw \$2</li> <li>andi \$2, \$2, 2</li> <li>shr \$2, \$2, 1</li> </ul>
!=	<ul style="list-style-type: none"> <li>%cmp = icmp ne i32 %0, %1</li> <li>%conv = zext i1 %cmp to i32</li> </ul>	<ul style="list-style-type: none"> <li>(setcc %0, %1, setne)</li> </ul>	<ul style="list-style-type: none"> <li>cmp \$sw, \$3, \$2</li> <li>mfsw \$2</li> <li>andi \$2, \$2, 2</li> <li>shr \$2, \$2, 1</li> </ul>
<	<ul style="list-style-type: none"> <li>%cmp = icmp lt i32 %0, %1</li> <li>%conv = zext i1 %cmp to i32</li> </ul>	<ul style="list-style-type: none"> <li>(setcc %0, %1, setlt)</li> </ul>	<ul style="list-style-type: none"> <li>cmp \$sw, \$3, \$2</li> <li>mfsw \$2</li> <li>andi \$2, \$2, 2</li> </ul>
<=	<ul style="list-style-type: none"> <li>%cmp = icmp le i32 %0, %1</li> <li>%conv = zext i1 %cmp to i32</li> </ul>	<ul style="list-style-type: none"> <li>(setcc %0, %1, settle)</li> </ul>	<ul style="list-style-type: none"> <li>cmp \$sw, \$2, \$3</li> <li>mfsw \$2</li> <li>andi \$2, \$2, 1</li> <li>xori \$2, \$2, 1</li> </ul>
>	<ul style="list-style-type: none"> <li>%cmp = icmp gt i32 %0, %1</li> <li>%conv = zext i1 %cmp to i32</li> </ul>	<ul style="list-style-type: none"> <li>(setcc %0, %1, setgt)</li> </ul>	<ul style="list-style-type: none"> <li>cmp \$sw, \$2, \$3</li> <li>mfsw \$2</li> <li>andi \$2, \$2, 2</li> </ul>
>=	<ul style="list-style-type: none"> <li>%cmp = icmp le i32 %0, %1</li> <li>%conv = zext i1 %cmp to i32</li> </ul>	<ul style="list-style-type: none"> <li>(setcc %0, %1, settle)</li> </ul>	<ul style="list-style-type: none"> <li>cmp \$sw, \$3, \$2</li> <li>mfsw \$2</li> <li>andi \$2, \$2, 1</li> <li>xori \$2, \$2, 1</li> </ul>

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.

## 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 20 operators totally in mathmetic and logic support in this chapter and spend 360 lines of source code.

Table 4.6: 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"> <li>ashr</li> <li>lshr</li> </ul>	<ul style="list-style-type: none"> <li>sra</li> <li>srl</li> </ul>	<ul style="list-style-type: none"> <li>sra</li> <li>shr</li> </ul>
!	<ul style="list-style-type: none"> <li>%tobool = icmp ne i32 %0, 0</li> <li>%lnot = xor i1 %tobool, true</li> </ul>	<ul style="list-style-type: none"> <li>%lnot = (setcc %tobool, 0, seteq)</li> <li>%conv = (and %lnot, 1)</li> </ul>	<ul style="list-style-type: none"> <li>%1 = (xor %tobool, 0)</li> <li>%true = (addiu \$r0, 1)</li> <li>%lnot = (xor %1, %true)</li> </ul>
•	<ul style="list-style-type: none"> <li>%conv = zext i1 %lnot to i32</li> </ul>	<ul style="list-style-type: none"> <li>%conv = (and %lnot, 1)</li> </ul>	<ul style="list-style-type: none"> <li>%conv = (and %lnot, 1)</li> </ul>
%	<ul style="list-style-type: none"> <li>srem</li> <li>sremu</li> </ul>	<ul style="list-style-type: none"> <li>Cpu0ISD::DivRem</li> <li>Cpu0ISD::DivRemU</li> </ul>	<ul style="list-style-type: none"> <li>div</li> <li>divu</li> </ul>



# GENERATING OBJECT FILES

The previous chapters only introduce the assembly code generated. This chapter will introduce you the obj support first, and display the 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 will be introduced in this chapter.

## 5.1 Translate into obj file

Currently, we only support translate llvm IR code into assembly code. If you try to run Chapter4\_2/ to translate obj code will get the error message as follows,

```
[Gamma@localhost 3]$ /usr/local/llvm/test/cmake_debug_build/bin/
llc -march=cpu0 -relocation-model=pic -filetype=obj ch4_1.bc -o ch4_1.cpu0.o
/usr/local/llvm/test/cmake_debug_build/bin/llc: target does not
support generation of this file type!
```

The Chapter5\_1/ support obj file generated. It can get result for big endian and little endian with command llc -march=cpu0 and llc -march=cpu0el. Run it will get the obj files as follows,

```
[Gamma@localhost InputFiles]$ 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]$ /usr/local/llvm/test/cmake_debug_build/bin/
llc -march=cpu0 -relocation-model=pic -filetype=obj ch4_1.bc -o ch4_1.cpu0.o
[Gamma@localhost InputFiles]$ 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 InputFiles]$ /usr/local/llvm/test/
cmake_debug_build/bin/llc -march=cpu0el -relocation-model=pic -filetype=obj
ch4_1.bc -o ch4_1.cpu0el.o
[Gamma@localhost InputFiles]$ objdump -s ch4_1.cpu0el.o

```

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

```

Contents of section .text:
0000 c8fffd09 05002009 34002d02 02002009  ..... .4.-.... .
0010 30002d02 fbf02009 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 it’s corresponding obj is 0x09ddff8. The addiu opcode is 0x09, 8 bits, \$sp register number is 13(0xd), 4bits, and the immediate is 16 bits -56(=0xffc8), so it’s 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 as 0x09ddff8 and the little endian is (B3, B2, B1, B0) = (09, dd, ff, c8), objdump from B0 to B3 as 0xc8fffd09.

## 5.2 Backend Target Registration Structure

To support elf obj generated, the following code changed and added to Chapter5\_1.

### Ibdex/Chapter5\_1/MCTargetDesc/CMakeLists.txt

```

add_llvm_library(LLVMCpu0Desc
  Cpu0AsmBackend.cpp
  ...
  Cpu0MCCodeEmitter.cpp
  ...
  Cpu0ELFObjectWriter.cpp
)

```

### Ibdex/Chapter5\_1/MCTargetDesc/Cpu0AsmBackend.cpp

### Ibdex/Chapter5\_1/MCTargetDesc/Cpu0BaseInfo.h

```

inline static std::pair<const MCSymbolRefExpr*, int64_t>
Cpu0GetSymAndOffset(const MCFixup &Fixup) {
    MCFixupKind FixupKind = Fixup.getKind();

    if ((FixupKind < FirstTargetFixupKind) ||
        (FixupKind >= MCFixupKind(Cpu0::LastTargetFixupKind)))
        return std::make_pair((const MCSymbolRefExpr*)0, (int64_t)0);

    const MCExpr *Expr = Fixup.getValue();
    MCExpr::ExprKind Kind = Expr->getKind();

    if (Kind == MCExpr::Binary) {
        const MCBinaryExpr *BE = static_cast<const MCBinaryExpr*>(Expr);
        const MCExpr *LHS = BE->getLHS();
        const MCConstantExpr *CE = dyn_cast<MCConstantExpr>(BE->getRHS());

        if ((LHS->getKind() != MCExpr::SymbolRef) || !CE)
            return std::make_pair((const MCSymbolRefExpr*)0, (int64_t)0);

        return std::make_pair(cast<MCSymbolRefExpr>(LHS), CE->getValue());
    }

    if (Kind != MCExpr::SymbolRef)
        return std::make_pair((const MCSymbolRefExpr*)0, (int64_t)0);

    return std::make_pair(cast<MCSymbolRefExpr>(Expr), 0);
} // Cpu0GetSymAndOffset

```

### Ibdex/Chapter5\_1/MCTargetDesc/Cpu0ELFObjectWriter.cpp

```

case Cpu0::fixup_Cpu0_GOT_Global:
case Cpu0::fixup_Cpu0_GOT_Local:
    Type = ELF::R_CPU0_GOT16;

```

### Ibdex/Chapter5\_1/MCTargetDesc/Cpu0FixupKinds.h

### Ibdex/Chapter5\_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

### Ibdex/Chapter5\_1/MCTargetDesc/Cpu0MCTargetDesc.cpp

### Ibdex/Chapter5\_1/MCTargetDesc/Cpu0MCTargetDesc.h

Now, let's examine Cpu0MCTargetDesc.cpp. Cpu0MCTargetDesc.cpp do the target registration as mentioned in “section Target Registration”<sup>1</sup> of the last chapter. Drawing the register function and those class it registered in Figure 5.1 to Figure 5.9 for explanation.

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

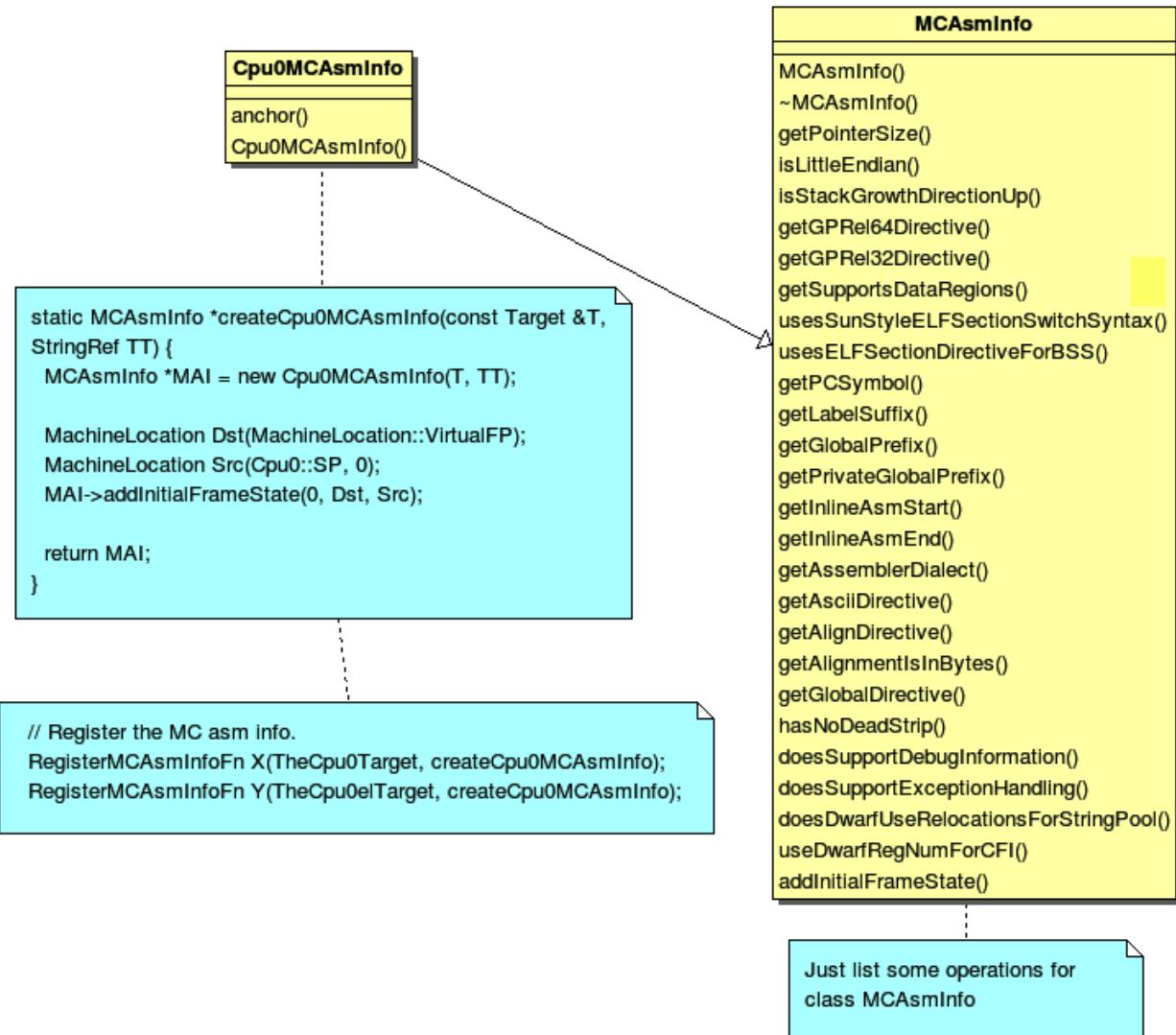


Figure 5.1: Register Cpu0MCAsmInfo

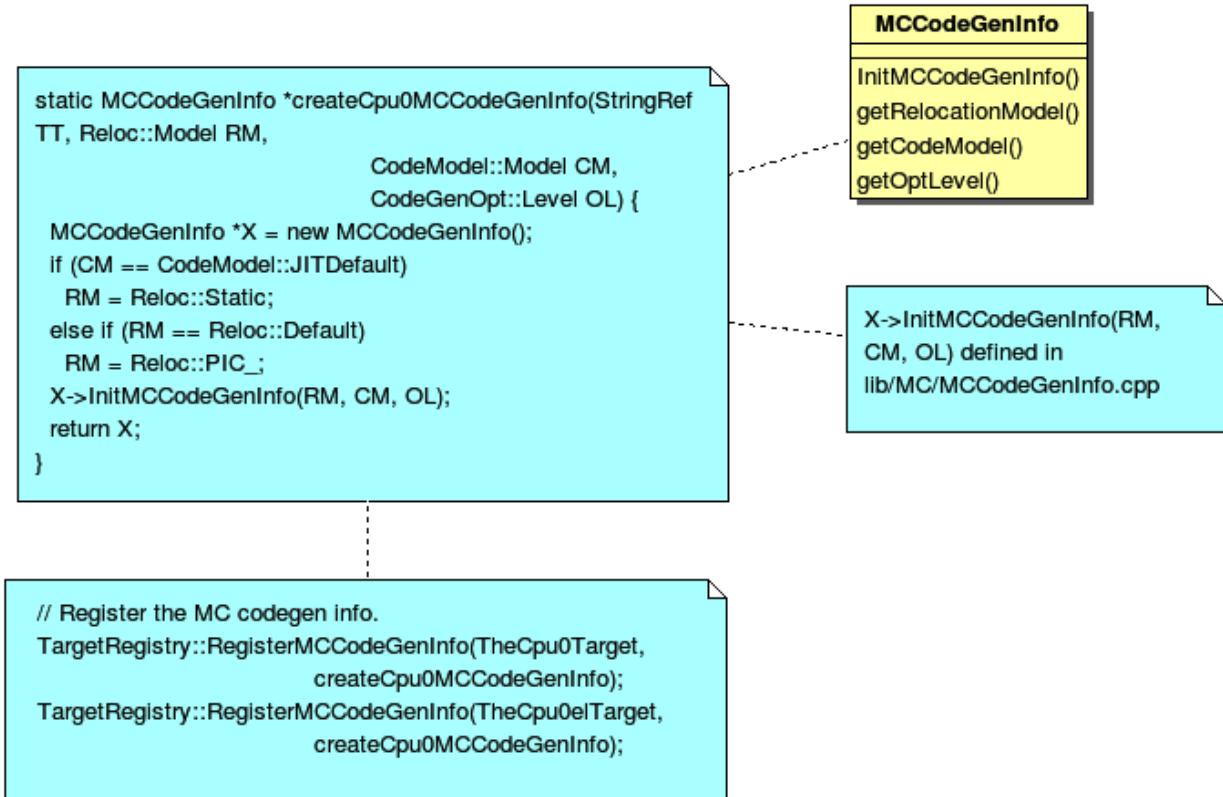


Figure 5.2: Register MCCCodeGenInfo

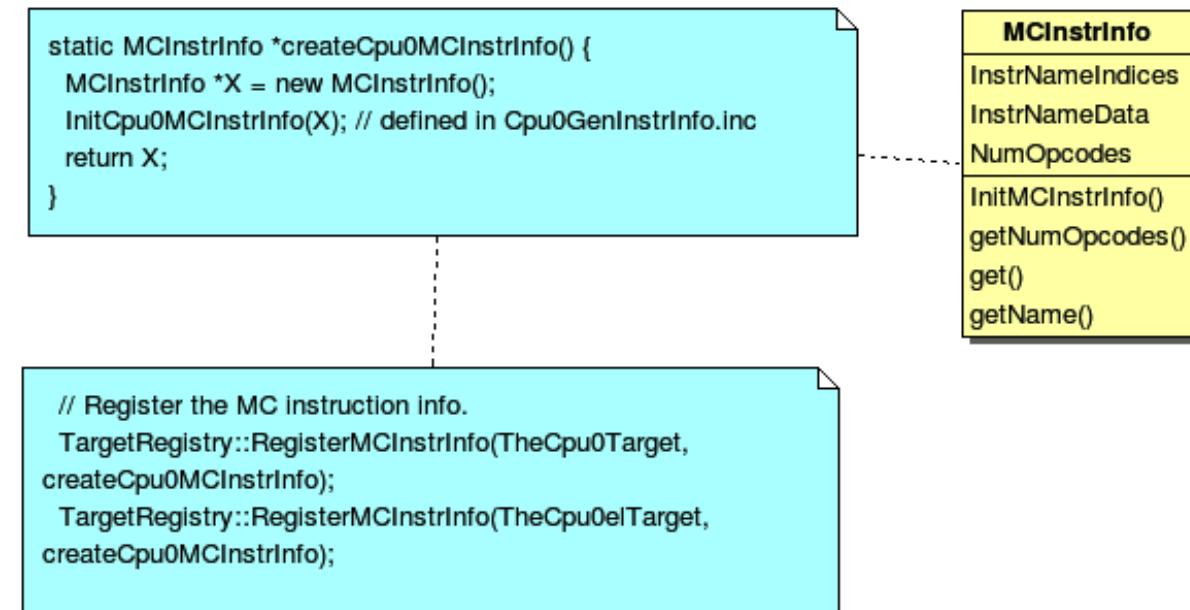


Figure 5.3: Register MCInstrInfo

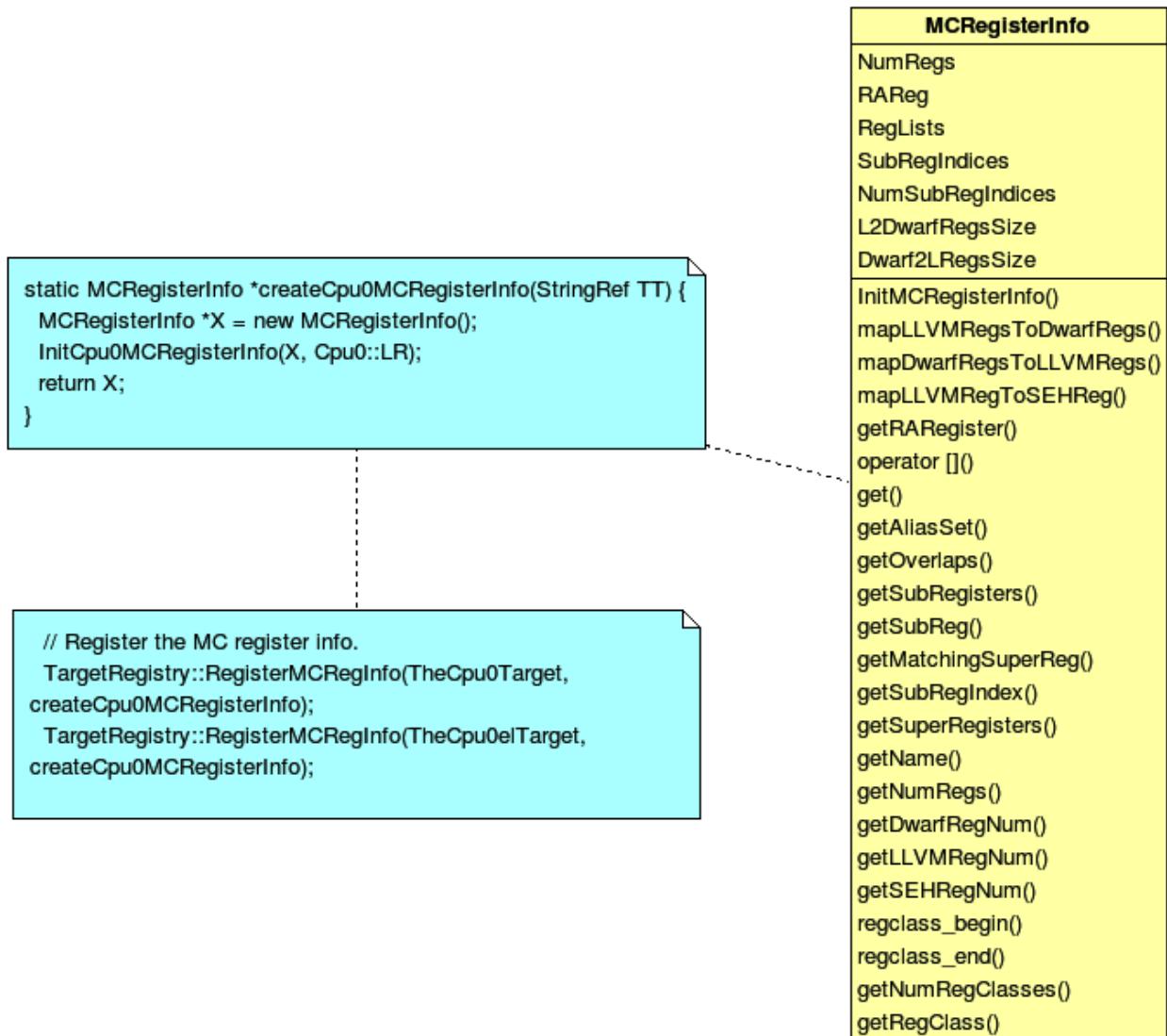


Figure 5.4: Register MCRegisterInfo

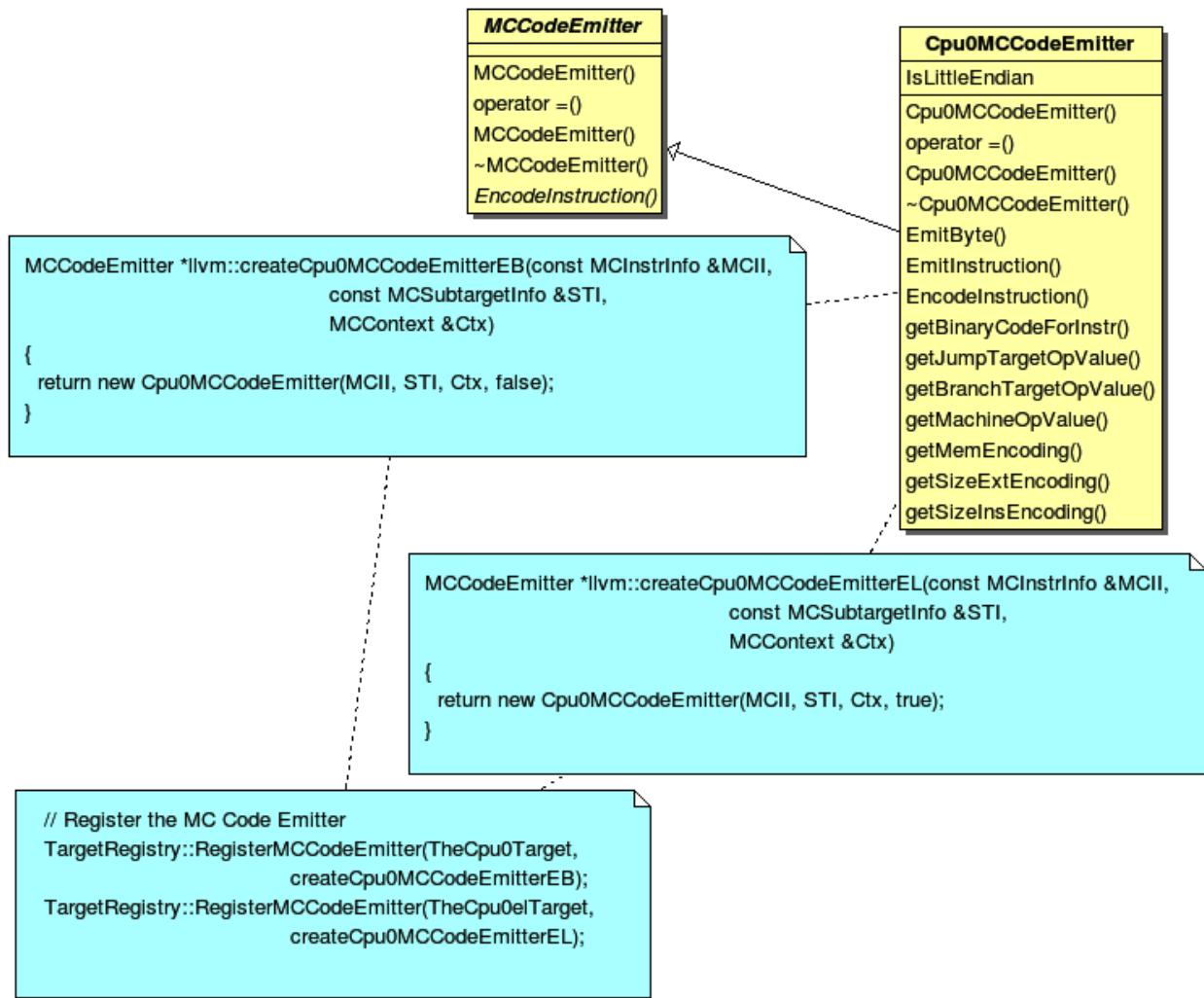


Figure 5.5: Register Cpu0MCCodeEmitter

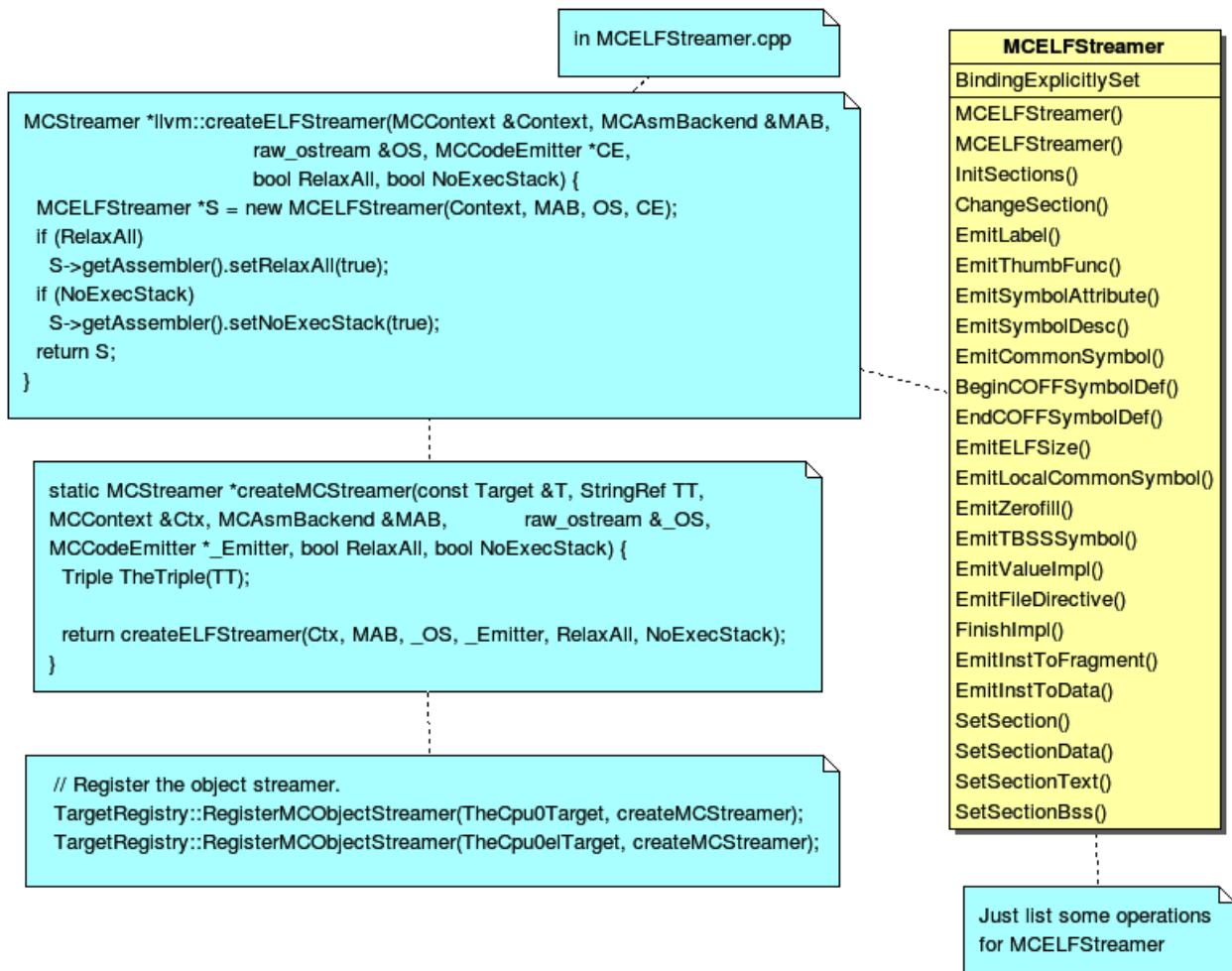


Figure 5.6: Register MCELFStreamer

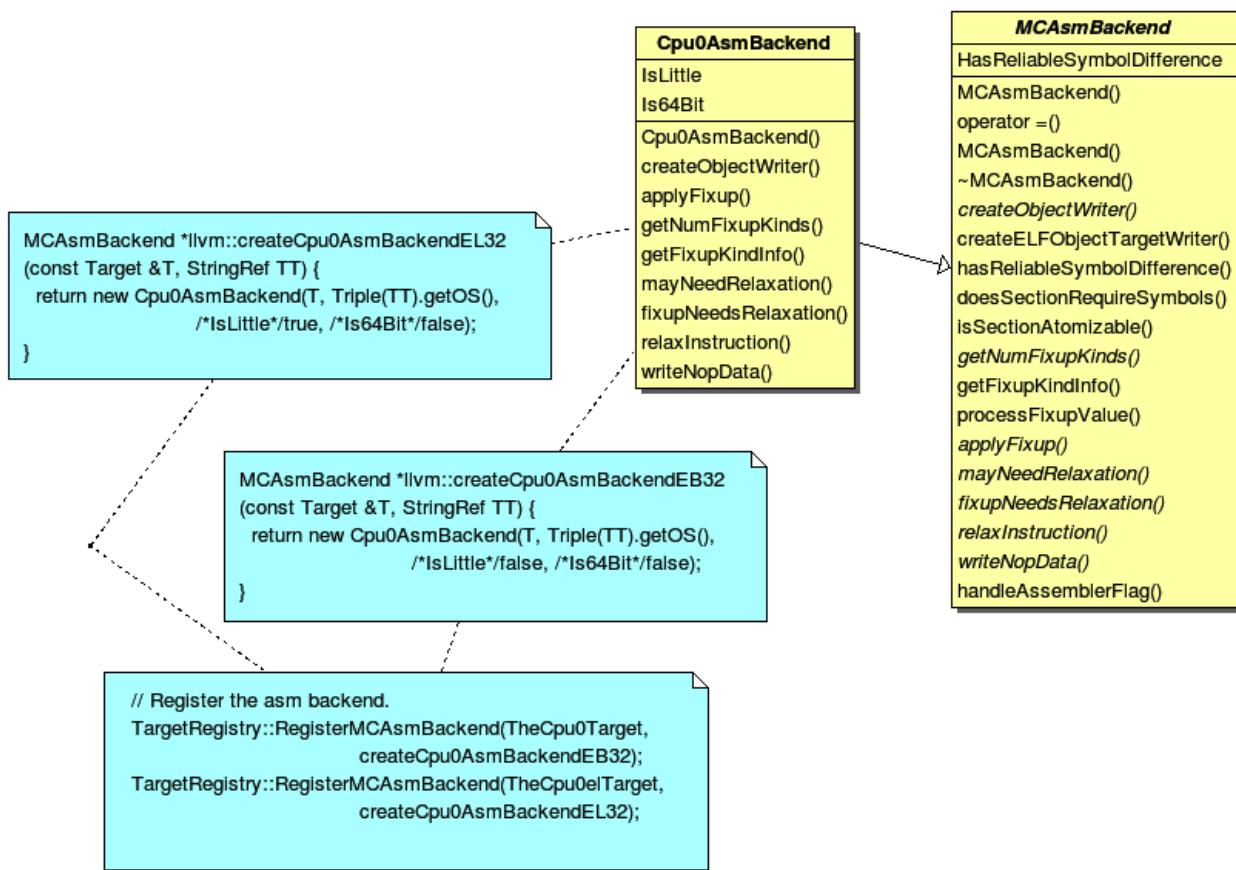


Figure 5.7: Register Cpu0AsmBackend

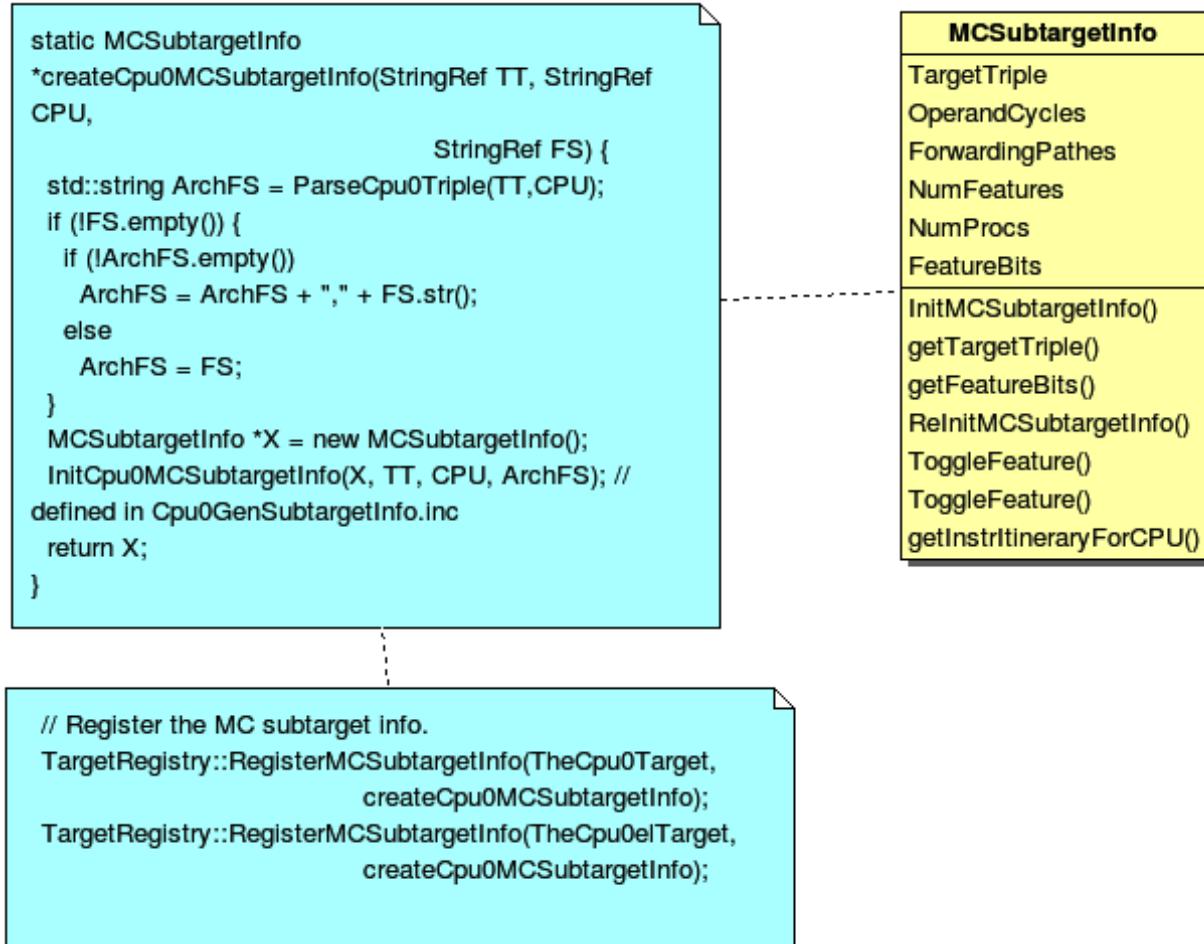


Figure 5.8: Register Cpu0MCSubtargetInfo

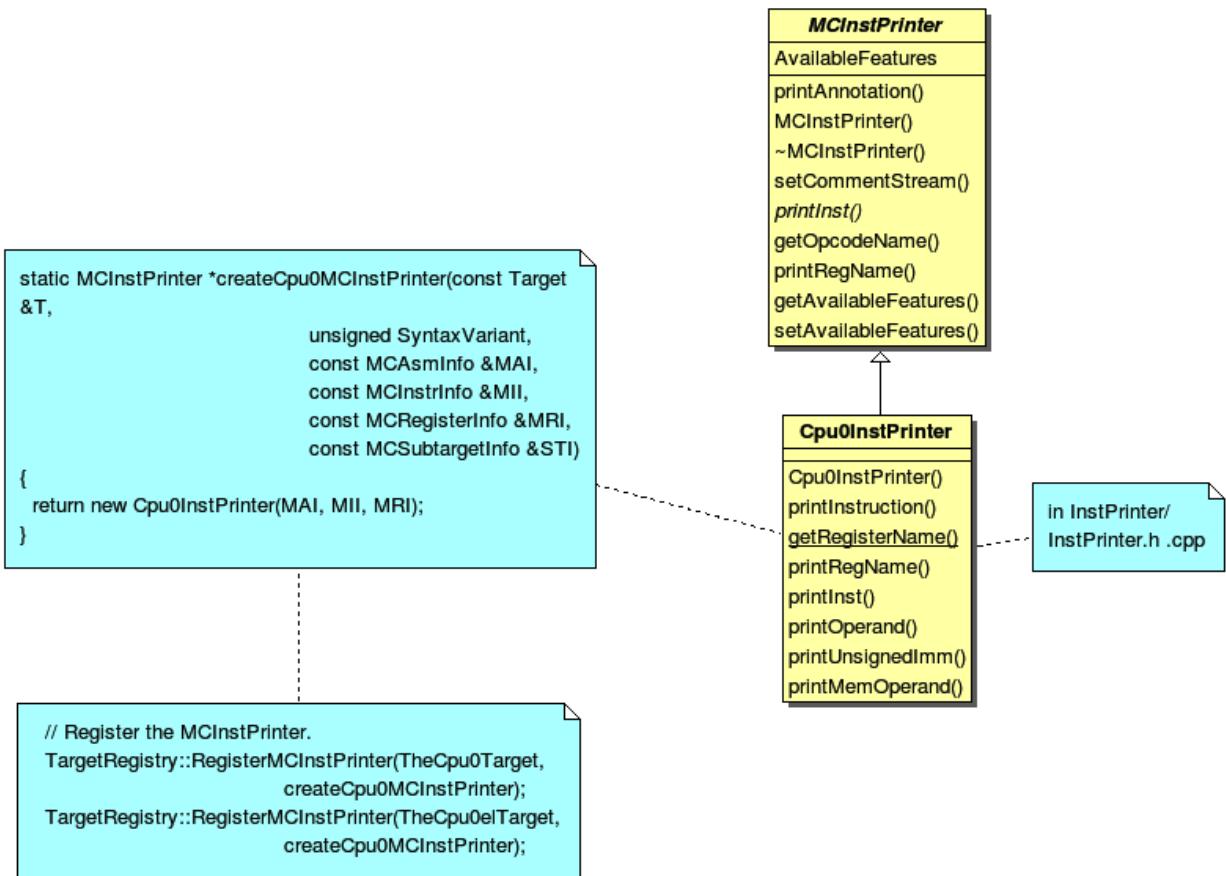


Figure 5.9: Register Cpu0InstPrinter



Figure 5.10: MCELFStreamer inherit tree

In Figure 5.1, registering the object of class Cpu0AsmInfo for target TheCpu0Target and TheCpu0elTarget. TheCpu0Target is for big endian and TheCpu0elTarget is for little endian. Cpu0AsmInfo is derived from MCAsmInfo which is llvm built-in class. Most code is implemented in it's parent, back end reuse those code by inherit.

In Figure 5.2, instancing MCCCodeGenInfo, and initialize it by pass `Roloc::PIC` because we use command `llc -relocation-model=pic` to tell `llc` compile using position-independent code mode. Recall the addressing mode in system program book has two mode, 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`, it will add the information from `Cpu0InstrInfo.td` we specified. Figure 5.4 is similar to Figure 5.3, but it initialize the register information specified in `Cpu0RegisterInfo.td`. They share a lot of code with instruction/register td description.

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. So, it's not defined in `Chapter4_2/` which support assembly code only.

Figure 5.6, MCELFStreamer take care the obj format also. Figure 5.5 `Cpu0MCCodeEmitter` take care code emitter while MCELFStreamer take care the obj output streamer. Figure 5.10 is MCELFStreamer inherit tree. You can find a lot of operations in that inherit tree.

Reader maybe has the question for what are the actual arguments in `createCpu0MCCodeEmitterEB(const MCInstrInfo &MCII, const MCSubtargetInfo &STI, MCContext &Ctx)` and at when they are assigned. Yes, we didn't assign it, we register the `createXXX()` function by function pointer only (according C, `TargetRegistry::RegisterXXX(TheCpu0Target, createXXX())` where `createXXX` is function pointer). LLVM keep 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()`.

Figure 5.7, `Cpu0AsmBackend` class is the bridge for asm to obj. Two objects take care big endian and little endian also. It derived from `MCAsmBackend`. Most of code for object file generated is implemented by `MCELFStreamer` 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 in `Chapter4_2/` code for assembly file generated support.



# GLOBAL VARIABLES

In the previous two chapters, we only access the local variables. This chapter will deal global variable access translation.

The global variable DAG translation is different from the previous DAG translation we have now. It create DAG nodes at run time in our 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 IR DAG.

## 6.1 Global variable

Chapter6\_1/ support the global variable, let's compile ch6\_1.cpp with this version first, and explain the code changes after that.

### Index/InputFiles/ch6\_1.cpp

```
int gStart = 3;
int gI = 100;
int fun()
{
    int c = 0;

    c = gI;

    return c;
}

118-165-78-166:InputFiles Jonathan$ llvm-dis ch6_1.bc -o -
; ModuleID = 'ch6_1.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"

@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
}
```

### 6.1.1 Cpu0 global variable options

Cpu0 like Mips 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/ support the global variable translation. Let's run Chapter6\_1/ with `ch6_1.cpp` via three different options `llc -relocation-model=static -cpu0-use-small-section=false`, `llc -relocation-model=static -cpu0-use-small-section=true` and `llc -relocation-model=pic` to trace the DAG and Cpu0 instructions.

```
118-165-78-166:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch6_1.cpp -emit-llvm -o ch6_1.bc
118-165-78-166:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/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]
```

```

...
lui $2, %hi(gI)
addiu $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:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=static -cpu0-use-small-section=true
-filetype=asm -debug ch6_1.bc -o -
.

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 = GLOBAL_OFFSET_TABLE

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]
...
addiu $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:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm -debug ch6_1.bc
-o -

...
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]
...

```

```

.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"> <li>pic: Position Independent Address</li> <li>static: Absolute Address</li> </ul>
-cpu0-use-small-section	false	true	<ul style="list-style-type: none"> <li>false: .data or .bss, 16 bits addressable</li> <li>true: .sdata or .sbss, 32 bits addressable</li> </ul>

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 GLOBAL_OFFSET_TABLE, Cpu0ISD::GPRel<gI offset>)
Cpu0	addiu \$2, \$zero, %hi(gI); shl \$2, \$2, 16; addiu \$2, \$2, %lo(gI);	addiu \$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 %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);	addiu \$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 compiled with Position Independent Addressing Mode. In principle, option -relocation-model is used to generate 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 use \$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 <sup>1</sup> further.

### lbdex/Chapter6\_1/Cpu0Subtarget.h

```
extern bool Cpu0NoCpload;
...
class Cpu0Subtarget : public Cpu0GenSubtargetInfo {
...
    // UseSmallSection - Small section is used.
    bool UseSmallSection;
    ...
    bool useSmallSection() const { return UseSmallSection; }
};
```

### lbdex/Chapter6\_1/Cpu0Subtarget.cpp

```
#include "llvm/Support/CommandLine.h"
...
static cl::opt<bool>
UseSmallSectionOpt("cpu0-use-small-section", cl::Hidden, cl::init(false),
                   cl::desc("Use small section. Only work with -relocation-model="
                           "'static. pic always not use small section.'"));
static cl::opt<bool>
```

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

```

ReserveGPOpt ("cpu0-reserve-gp", cl::Hidden, cl::init(false),
              cl::desc("Never allocate $gp to variable"));

static cl::opt<bool>
NoCuploadOpt ("cpu0-no-cupload", cl::Hidden, cl::init(false),
              cl::desc("No issue .cupload"));

bool Cpu0ReserveGP;
bool Cpu0NoCupload;

extern bool FixGlobalBaseReg;

```

The ReserveGPOpt and NoCuploadOpt are used in Cpu0 linker at later Chapter. Next add file Cpu0TargetObjectFile.h, Cpu0TargetObjectFile.cpp and the following code to Cpu0RegisterInfo.cpp and Cpu0ISelLowering.cpp.

[lbdex/Chapter6\\_1/Cpu0TargetObjectFile.h](#)

[lbdex/Chapter6\\_1/Cpu0TargetObjectFile.cpp](#)

[lbdex/Chapter6\\_1/Cpu0RegisterInfo.cpp](#)

```

// pure virtual method
BitVector Cpu0RegisterInfo::
getReservedRegs(const MachineFunction &MF) const {
    ...
// Reserve GP if small section is used.
if (Subtarget.useSmallSection()) {
    Reserved.set(Cpu0::GP);
}
...
}

```

[lbdex/Chapter6\\_1/Cpu0ISelLowering.cpp](#)

```

#include "Cpu0MachineFunction.h"
...
#include "Cpu0TargetObjectFile.h"
...
#include "MCTargetDesc/Cpu0BaseInfo.h"
...
#include "llvm/Support/CommandLine.h"
SDValue Cpu0TargetLowering::getGlobalReg(SelectionDAG &DAG, EVT Ty) const {
    Cpu0FunctionInfo *FI = DAG.getMachineFunction().getInfo<Cpu0FunctionInfo>();
    return DAG.getRegister(FI->getGlobalBaseReg(), Ty);
}

static SDValue getTargetNode(SDValue Op, SelectionDAG &DAG, unsigned Flag) {
    EVT Ty = Op.getValueType();

    if (GlobalAddressSDNode *N = dyn_cast<GlobalAddressSDNode>(Op))
        return DAG.getTargetGlobalAddress(N->getGlobal(), Op.getDebugLoc(), Ty, 0,
                                         Flag);
    if (ExternalSymbolSDNode *N = dyn_cast<ExternalSymbolSDNode>(Op))
        return DAG.getTargetExternalSymbol(N->getSymbol(), Ty, Flag);
}

```

```

if (BlockAddressSDNode *N = dyn_cast<BlockAddressSDNode>(Op))
    return DAG.getTargetBlockAddress(N->getBlockAddress(), Ty, 0, Flag);
if (JumpTableSDNode *N = dyn_cast<JumpTableSDNode>(Op))
    return DAG.getTargetJumpTable(N->getIndex(), Ty, Flag);
if (ConstantPoolSDNode *N = dyn_cast<ConstantPoolSDNode>(Op))
    return DAG.getTargetConstantPool(N->getConstVal(), Ty, N->getAlignment(),
                                    N->getOffset(), Flag);

    llvm_unreachable("Unexpected node type.");
    return SDValue();
}

SDValue Cpu0TargetLowering::getAddrLocal(SDValue Op, SelectionDAG &DAG) const {
    DebugLoc DL = Op.getDebugLoc();
    EVT Ty = Op.getValueType();
    unsigned GOTFlag = Cpu0II::MO_GOT;
    SDValue GOT = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, getGlobalReg(DAG, Ty),
                             getTargetNode(Op, 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(Op, DAG, LoFlag));
    return DAG.getNode(ISD::ADD, DL, Ty, Load, Lo);
}

SDValue Cpu0TargetLowering::getAddrGlobal(SDValue Op, SelectionDAG &DAG,
                                         unsigned Flag) const {
    DebugLoc DL = Op.getDebugLoc();
    EVT Ty = Op.getValueType();
    SDValue Tgt = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, getGlobalReg(DAG, Ty),
                             getTargetNode(Op, DAG, Flag));
    return DAG.getLoad(Ty, DL, DAG.getEntryNode(), Tgt,
                       MachinePointerInfo::getGOT(), false, false, false, 0);
}

SDValue Cpu0TargetLowering::getAddrGlobalLargeGOT(SDValue Op, SelectionDAG &DAG,
                                                unsigned HiFlag,
                                                unsigned LoFlag) const {
    DebugLoc DL = Op.getDebugLoc();
    EVT Ty = Op.getValueType();
    SDValue Hi = DAG.getNode(Cpu0ISD::Hi, DL, Ty, getTargetNode(Op, DAG, HiFlag));
    Hi = DAG.getNode(ISD::ADD, DL, Ty, Hi, getGlobalReg(DAG, Ty));
    SDValue Wrapper = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, Hi,
                                 getTargetNode(Op, DAG, LoFlag));
    return DAG.getLoad(Ty, DL, DAG.getEntryNode(), Wrapper,
                       MachinePointerInfo::getGOT(), false, false, false, 0);
}

const char *Cpu0TargetLowering::getTargetNodeName(unsigned Opcode) const {
    switch (Opcode) {
        case Cpu0ISD::JmpLink:           return "Cpu0ISD::JmpLink";
        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;
}

}

Cpu0TargetLowering::
Cpu0TargetLowering(Cpu0TargetMachine &TM)
: TargetLowering(TM, new Cpu0TargetObjectFile(),
  Subtarget(&TM.getSubtarget<Cpu0Subtarget>()) {
  ...
  // Cpu0 Custom Operations
  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();
}

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

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

SDValue Cpu0TargetLowering::LowerGlobalAddress(SDValue Op,
                                              SelectionDAG &DAG) const {
  // FIXME there isn't actually debug info here
  DebugLoc dl = Op.getDebugLoc();
  const GlobalValue *GV = cast<GlobalAddressSDNode>(Op)->getGlobal();

  if (getTargetMachine().getRelocationModel() != Reloc::PIC_) {
    SDVTList VTs = DAG.getVTList(MVT::i32);

    Cpu0TargetObjectFile &TLOF = (Cpu0TargetObjectFile&)getObjFileLowering();

    // %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, VTs, &GA, 1);
      SDValue GOT = DAG.getGLOBAL_OFFSET_TABLE(MVT::i32);
      return DAG.getNode(ISD::ADD, dl, MVT::i32, GOT, GPRelNode);
    }
    // %hi/%lo relocation
    SDValue GAHi = DAG.getTargetGlobalAddress(GV, dl, MVT::i32, 0,
                                              Cpu0II::MO_ABS_HI);
    SDValue GALo = DAG.getTargetGlobalAddress(GV, dl, MVT::i32, 0,
                                              Cpu0II::MO_ABS_LO);
    SDValue HiPart = DAG.getNode(Cpu0ISD::Hi, dl, VTs, &GAHi, 1);
  }
}

```

```

SDValue Lo = DAG getNode(Cpu0ISD::Lo, dl, MVT::i32, GALo);
return DAG getNode(ISD::ADD, dl, MVT::i32, HiPart, Lo);
}

if (GV->hasInternalLinkage() || (GV->hasLocalLinkage() && !isa<Function>(GV)))
    return getAddrLocal(Op, DAG);

if (TLOF.IsGlobalInSmallSection(GV, getTargetMachine()))
    return getAddrGlobal(Op, DAG, Cpu0II::MO_GOT16);
else
    return getAddrGlobalLargeGOT(Op, DAG, Cpu0II::MO_GOT_HI16,
                                Cpu0II::MO_GOT_LO16);
}

```

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. Since there are many Custom type of setOperationAction(ISD::XXX, MVT::XXX, Custom) in construction function Cpu0TargetLowering(), and each of them will trigger llvm calling Cpu0TargetLowering::LowerOperation() in stage “Legalized selection DAG”. The global address access can be identified by check if the DAG node of opcode is equal to ISD::GlobalAddress.

Finally, add the following code in Cpu0InstrInfo.td.

### Ibdex/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>;
...
// hi/lo relocs
def : Pat<(Cpu0Hi tglobaladdr:$in), (SHL (ADDiu ZERO, tglobaladdr:$in), 16)>;
// Expect cpu0 add LUi support, like Mips
//def : Pat<(Cpu0Hi tglobaladdr:$in), (LUi tglobaladdr:$in)>;
def : Pat<(Cpu0Lo tglobaladdr:$in), (ADDiu ZERO, tglobaladdr:$in)>

def : Pat<(add CPUREgs:$hi, (Cpu0Lo tglobaladdr:$lo)),
        (ADDiu CPUREgs:$hi, tglobaladdr:$lo)>

// gp_rel relocs
def : Pat<(add CPUREgs:$gp, (Cpu0GPRel tglobaladdr:$in)),
        (ADDiu CPUREgs:$gp, tglobaladdr:$in)>;

```

### 6.1.2 Static mode

From Table: Cpu0 global variable options, option cpu0-use-small-section=false put the global varibile in data/bss while cpu0-use-small-section=true in sdata/sbss. The sdata stands for small data area. Section data and sdata are areas for global variable 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 example, int gI;).

## 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)
addiu $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

```

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 add register \$2 and low part of gI absolute address into \$2. At this point, it get the gI memory address. Finally, it get 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. In static, the function `fun()` is included to the whole execution file, ELF. The offset between `.data` and instruction “`addiu $2, $zero, %hi(gI)`” can be caculated. Since use PC relative address coding, this program can be loaded to any address and run well there. If this program use absolute address and will be loaded at a specific address known at link stage, the relocation record of gI variable access instruction such as “`addiu $2, $zero, %hi(gI)`” and “`addiu $2, $2, %lo(gI)`” can be solved at link time. 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()` return true or false depends on `UseSmallSectionOpt`.

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 Cpu0ISD::Hi<gI offset Hi16> Cpu0ISD::Lo<gI offset Lo16>) in stage “Legalized selection DAG” as below.

## Index/Chapter6\_1/Cpu0ISelLowering.cpp

```

// Cpu0ISelLowering.cpp
...
// %hi/%lo relocation
SDValue GAHi = DAG.getTargetGlobalAddress(GV, dl, MVT::i32, 0,
                                           Cpu0II::MO_ABS_HI);
SDValue GALo = DAG.getTargetGlobalAddress(GV, dl, MVT::i32, 0,
                                           Cpu0II::MO_ABS_LO);
SDValue HiPart = DAG.getNode(Cpu0ISD::Hi, dl, VTs, &GAHi, 1);
SDValue Lo = DAG.getNode(Cpu0ISD::Lo, dl, MVT::i32, GALo);
return DAG.getNode(ISD::ADD, dl, MVT::i32, HiPart, Lo);

```

```

118-165-78-166:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch6_1.cpp -emit-llvm -o ch6_1.bc
118-165-78-166:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/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.

### Ibdex/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
def : Pat<(Cpu0Hi tglobaladdr:$in), (LUI tglobaladdr:$in)>;
def : Pat<(Cpu0Lo tglobaladdr:$in), (ADDiu ZERO, tglobaladdr:$in)>;

```

```

def : Pat<(add CPURegs:$hi, (Cpu0Lo tglobaladdr:$lo)),  

          (ADDiu CPURegs:$hi, tglobaladdr:$lo)>;  

...  

    addiu  $2, $zero, %hi(gI)  

    shl    $2, $2, 16  

    addiu  $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), (SHL (ADDiu ZERO, tglobaladdr:\$in), 16)>; will translate DAG (Cpu0ISD::Hi tglobaladdr) into (shl (addiu ZERO, tglobaladdr), 16). Pat<(Cpu0Lo tglobaladdr:\$in), (ADDiu ZERO, tglobaladdr:\$in)>; will translate (Cpu0ISD::Hi tglobaladdr) into (addiu ZERO, tglobaladdr). Pat<(add CPURegs:\$hi, (Cpu0Lo tglobaladdr:\$lo)), (ADDiu CPURegs:\$hi, tglobaladdr:\$lo)>; will translate DAG (add Cpu0ISD::Hi, Cpu0ISD::Lo) into Cpu0 instruction (add Cpu0ISD::Hi, Cpu0ISD::Lo).

## sdata or sbss

The sdata/sbss are 16 bits addressable areas which planed in ELF for fast access. Option cpu0-use-small-section=true will generate the following instructions.

```

addiu  $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 GLOBAL\_OFFSET\_TABLE Cpu0ISD::GPREL<gI offset>) in stage “Legalized selection DAG” as below.

## Index/Chapter6\_1/Cpu0ISelLowering.cpp

```

// Cpu0ISelLowering.cpp
...
// %gp_rel relocation
if (TLOF.IsGlobalInSection(GV, getTargetMachine())) {
    SDValue GA = DAG.getTargetGlobalAddress(GV, dl, MVT::i32, 0,
                                             Cpu0II::MO_GPREL);
    SDValue GPRELNode = DAG.getNode(Cpu0ISD::GPREL, dl, VTs, &GA, 1);
    SDValue GOT = DAG.getGLOBAL_OFFSET_TABLE(MVT::i32);
    return DAG.getNode(ISD::ADD, dl, MVT::i32, GOT, 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 = GLOBAL_OFFSET_TABLE

  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 GLOBAL\_OFFSET\_TABLE Cpu0ISD::GPRel<gI offset>) into Cpu0 instruction as below. The following code in Cpu0ISelDAGToDAG.cpp make the GLOBAL\_OFFSET\_TABLE translate into \$gp as below.

### [Index/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, TLI.getPointerTy()).getNode();
}

/// Select instructions not customized! Used for
/// expanded, promoted and normal instructions
SDNode* Cpu0DAGToDAGISel::Select(SDNode *Node) {
  ...
  // Get target GOT address.
  // For global variables as follows,
  // - @gI = global i32 100, align 4
  // - %2 = load i32* @gI, align 4
  // =>
  // - .cupload $gp

```

```

//- 1d      $2, %got(gI)($gp)
case ISD::GLOBAL_OFFSET_TABLE:
    return getGlobalBaseReg();
...
}

```

### Ibdex/Chapter6\_1/Cpu0InstrInfo.td

```

// Cpu0InstrInfo.td
def Cpu0GPRel : SDNode<"Cpu0ISD::GPRel", SDTIntUnaryOp>;
...
// gp_rel relocs
def : Pat<(add CPUREgs:$gp, (Cpu0GPRel tglobaladdr:$in)),
          (ADD CPUREgs:$gp, (ADDiu ZERO, tglobaladdr:$in))>;
addiu  $2, $gp, %gp_rel(gI)
...

```

Pat<(add CPUREgs:\$gp, (Cpu0GPRel tglobaladdr:\$in)), (ADD CPUREgs:\$gp, (ADDiu ZERO, tglobaladdr:\$in))>;  
 will translate (add \$gp Cpu0ISD::GPRel tglobaladdr) into (add \$gp, (addiu 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; while the -relocation-model=pic the \$gp can be changed during program running. For this example, 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 both linker and loader agree to. So, the \$gp cannot be allocated as a general purpose for variables. The following code tells llvm never allocate \$gp for variables.

### Ibdex/Chapter6\_1/Cpu0Subtarget.cpp

```

Cpu0Subtarget::Cpu0Subtarget(const std::string &TT, const std::string &CPU,
                           const std::string &FS, bool little,
                           Reloc::Model _RM) :
    Cpu0GenSubtargetInfo(TT, CPU, FS),
    Cpu0ABI(UnknownABI), IsLittle(little), RM(_RM)
{
    ...
    // Set UseSmallSection.
    UseSmallSection = UseSmallSectionOpt;
    Cpu0ReserveGP = ReserveGPOpt;
    Cpu0NoCpload = NoCploadOpt;
    if (_RM == Reloc::Static && !UseSmallSection && !Cpu0ReserveGP)
        FixGlobalBaseReg = false;
    else
        FixGlobalBaseReg = true;
}

```

### Ibdex/Chapter6\_1/Cpu0RegisterInfo.cpp

```

// pure virtual method
BitVector Cpu0RegisterInfo::

```

```
getReservedRegs(const MachineFunction &MF) const {
    ...
    const Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
    // Reserve GP if globalBaseRegFixed()
    if (Cpu0FI->globalBaseRegFixed())
        Reserved.set(Cpu0::GP);
    }
    ...
}
```

### 6.1.3 pic mode

#### 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.

#### lbdex/Chapter6\_1/Cpu0MachineFunction.h

```
//===== Cpu0MachineFunction.h - Private data used for Cpu0 -----*-- C++ -*--//  

...
class Cpu0FunctionInfo : public MachineFunctionInfo {
    virtual void anchor();
    ...

    /// 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
...

public: Cpu0FunctionInfo(MachineFunction& MF)
: ..., GlobalBaseReg(0), ...
{ }

bool globalBaseRegFixed() const;
bool globalBaseRegSet() const;
unsigned getGlobalBaseReg();
};

} // end of namespace llvm

#endif // CPU0_MACHINE_FUNCTION_INFO_H

```

**Ibdex/Chapter6\_1/Cpu0MachineFunction.cpp**

**Ibdex/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 (OutStreamer.hasRawTextSupport()) {
        ...
        OutStreamer.EmitRawText(StringRef("\t.set\tnoreorder"));
        // Emit .cupload directive if needed.
        if (EmitCPLoad)
            OutStreamer.EmitRawText(StringRef("\t.cupload\t$6"));
        OutStreamer.EmitRawText(StringRef("\t.set\tnomacro"));
        if (Cpu0FI->getEmitNOAT())
            OutStreamer.EmitRawText(StringRef("\t.set\tnoat"));
    } 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);
    }
}

...
.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 whenever an assembler operation generates more than one machine language instruction, reference Mips ABI<sup>2</sup>.

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

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

### Ibdex/Chapter6\_1/Cpu0MCInstLower.cpp

```
118-165-76-131:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=
obj ch6_1.bc -o ch6_1.cpu0.o
118-165-76-131:InputFiles Jonathan$ gobjdump -s ch6_1.cpu0.o

ch6_1.cpu0.o:      file format elf32-big

Contents of section .text:
0000 0fa00000 09aa0000 13aa6000 ...
...

118-165-76-131:InputFiles 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 09a00000 (offset 0) which equal to assembly “addiu \$gp, \$zero, %hi(\_gp\_disp)” and 09aa0000 (offset 8) which equal to assembly “addiu \$gp, \$gp, %lo(\_gp\_disp)” 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 adjust these two instructions offset correctly. Since shared function is loaded when this function be 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. No need to search the variable name at run time. The ELF relocation records will be introduced in Chapter ELF Support. 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.

### Ibdex/Chapter6\_1/Cpu0ISelLowering.cpp

```
SDValue Cpu0TargetLowering::getAddrGlobal(SDValue Op, SelectionDAG &DAG,
                                         unsigned Flag) const {
    DebugLoc DL = Op.getDebugLoc();
    EVT Ty = Op.getValueType();
```

```

SDValue Tgt = DAG getNode(Cpu0ISD::Wrapper, DL, Ty, getGlobalReg(DAG, Ty),
                           getTargetNode(Op, DAG, Flag));
return DAG.getLoad(Ty, DL, DAG.getEntryNode(), Tgt,
                     MachinePointerInfo::getGOT(), false, false, false, 0);
}

SDValue Cpu0TargetLowering::LowerGlobalAddress(SDValue Op,
                                              SelectionDAG &DAG) const {
    ...
    if (TLOF.IsGlobalInSmallSection(GV, getTargetMachine()))
        return getAddrGlobal(Op, DAG, Cpu0II::MO_GOT16);
    ...
}

```

### Ibdex/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 below.

```

...
ld      $2, %got(gI)($gp)
...

```

Remind in pic mode, Cpu0 use ".cupload" and "ld \$2, %got(gI)(\$gp)" to access global variable. It take 5 instructions in Cpu0 and 4 instructions in Mips. The cost came from we didn't assume the 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 preserve 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 came 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, and without ".cupload" will save four instructions which has the faster result in speed. 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. Anyway, in pic mode and used in static link, you can choose ".cupload" removing. But we prefer use \$gp for general purpose register as the solution. 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.

### 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.

#### Ibdex/Chapter6\_1/Cpu0ISelLowering.cpp

```

SDValue Cpu0TargetLowering::getAddrGlobalLargeGOT(SDValue Op, SelectionDAG &DAG,
                                                 unsigned HiFlag,
                                                 unsigned LoFlag) const {
    DebugLoc DL = Op.getDebugLoc();
    EVT Ty = Op.getValueType();
    SDValue Hi = DAG.getNode(Cpu0ISD::Hi, DL, Ty, getTargetNode(Op, DAG, HiFlag));
    Hi = DAG.getNode(ISD::ADD, DL, Ty, Hi, getGlobalReg(DAG, Ty));
    SDValue Wrapper = DAG.getNode(Cpu0ISD::Wrapper, DL, Ty, Hi,
                                   getTargetNode(Op, DAG, LoFlag));
    return DAG.getLoad(Ty, DL, DAG.getEntryNode(), Wrapper,
                       MachinePointerInfo::getGOT(), false, false, false, 0);
}

SDValue Cpu0TargetLowering::LowerGlobalAddress(SDValue Op,
                                              SelectionDAG &DAG) const {
...

```

```

if (TLOF.IsGlobalInSmallSection(GV, getTargetMachine()))
    ...
else
    return getAddrGlobalLargeGOT(Op, DAG, Cpu0II::MO_GOT_HI16,
                                Cpu0II::MO_GOT_LO16);
}

...
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.

```

...
    addiu $2, $zero, %got_hi(gI)
    shl  $2, $2, 16
    add  $2, $2, $gp
    ld   $2, %got_lo(gI)($2)
...

```

### 6.1.4 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.

#### lbdex/Chapter6\_1/Cpu0MCInstLower.cpp

```
MCOperand Cpu0MCInstLower::LowerOperand(const MachineOperand& MO,
                                         unsigned offset) const {
    MachineOperandType MOTy = MO.getType();
    switch (MOTy) {
    ...
    case MachineOperand::MO_GlobalAddress:
        return LowerSymbolOperand(MO, MOTy, offset);
    ...
}
```

#### lbdex/Chapter6\_1/InstPrinter/Cpu0InstPrinter.cpp

```
static void printExpr(const MCExpr *Expr, raw_ostream &OS) {
    ...
    switch (Kind) {
    default: llvm_unreachable("Invalid kind!");
    case MCSymbolRefExpr::VK_None: break;
// 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 DAG node name printing.

#### lbdex/Chapter6\_1/Cpu0ISelLowering.cpp

```
const char *Cpu0TargetLowering::getTargetNodeName(unsigned Opcode) const {
    switch (Opcode) {
    case Cpu0ISD::JmpLink: return "Cpu0ISD::JmpLink";
    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 "MipsISD::DivRem";
    case Cpu0ISD::DivRemU: return "MipsISD::DivRemU";
    case Cpu0ISD::Wrapper: return "Cpu0ISD::Wrapper";
    default: return NULL;
    }
}
```

OS is the output stream which output to the assembly file.

### 6.1.5 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() take 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 has it's own DAG list be created. 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 type for setXXXAction(), Promote, Expand and Custom. Except Custom, the other two maybe no need to coding. The section “Instruction Selector” of <sup>3</sup> is the references.

As shown in the section, the global variable can be laid in .sdata/.sbss by option `-cpu0-use-small-section=true`. It is possible, the small data section (16 bits addressable) is full out at link stage. When this happens, linker will highlight this error and force the toolchain user to fix it. The toolchain user, need to reconsider which global variables should be move from .sdata/.sbss to .data/.bss by set option `-cpu0-use-small-section=false` for that global variables declared file. The rule for global variables allocation is “set the small and frequent variables in small 16 addressable area”.

---

<sup>3</sup> <http://llvm.org/docs/WritingAnLLVMBackend.html>



# OTHER DATA TYPE

Until now, we only handle the type int and long of 32 bits long. This chapter introduce other type such as pointer, char, long long which are not 32 bits size.

## 7.1 Local variable pointer

To support pointer to local variable, add this code fragment in Cpu0InstrInfo.td and Cpu0InstPrinter.cpp as follows,

### lbdex/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;
}
```

### lbdex/Chapter7\_1/Cpu0InstPrinter.td

```
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;
}
```

Run ch7\_1.cpp with code Chapter7\_1/ which support pointer to local variable, will get result as follows,

#### lbdex/InputFiles/ch7\_1.cpp

```
int test_local_pointer()
{
    int b = 3;

    int* p = &b;

    return *p;
}

118-165-66-82:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_1.cpp -emit-llvm -o ch7_1.bc
118-165-66-82:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_
debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch7_1.bc -o ch7_1.cpu0.s
118-165-66-82:InputFiles Jonathan$ cat ch7_1.cpu0.s
.section .mdebug.abi32
.previous
.file "ch7_1.bc"
.text
.globl _Z18test_local_pointerv
.align 2
.type _Z18test_local_pointerv,@function
.ent _Z18test_local_pointerv # @_Z18test_local_pointerv
_Z18test_local_pointerv:
.frame $sp,8,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -8
addiu $2, $zero, 3
st $2, 4($sp)
addiu $2, $sp, 4
st $2, 0($sp)
addiu $sp, $sp, 8
ret $lr
.set macro
.set reorder
.end _Z18test_local_pointerv
$tmp1:
.size _Z18test_local_pointerv, ($tmp1)-_Z18test_local_pointerv
```

## 7.2 char, short int and bool

To support signed/unsigned char and short int, we add the following code to Chapter7\_1/.

**lbdex/Chapter7\_1/Cpu0InstrInfo.td**

```
def sextloadi16_a : AlignedLoad<sextloadi16>;
def zextloadi16_a : AlignedLoad<zextloadi16>;
def extloadi16_a : AlignedLoad<extloadi16>;
...
def truncstorei16_a : AlignedStore<truncstorei16>;
...
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.

**lbdex/InputFiles/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:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_3.cpp -emit-llvm -o ch7_2.bc
118-165-64-245:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch7_2.bc -o -
.section .mdebug.abi32
.previous
.file "ch7_2.bc"
.text
.globl _Z9test_charv
.align 2
.type _Z9test_charv,@function
.ent _Z9test_charv          # @_Z9test_charv
_Z9test_charv:
.frame $fp,24,$lr
.mask 0x00000000,0
```

```

.set noreorder
.cupload $t9
.set nomacro
# BB#0:                                # %entry
addiu $sp, $sp, -24
lui $2, %got_hi(b)
addu $2, $2, $gp
ld $2, %got_lo(b)($2)
lbu $3, 1($2)
sb $3, 20($fp)
lbu $2, 1($2)
sb $2, 16($fp)
ld $2, %got($_ZZ9test_charvE5date1)($gp)
addiu $2, $2, %lo($_ZZ9test_charvE5date1)
lhu $3, 4($2)
shl $3, $3, 16
lhu $4, 6($2)
or $3, $3, $4
st $3, 12($fp) // store hour, minute and second on 12($sp)
lhu $3, 2($2)
lhu $2, 0($2)
shl $2, $2, 16
or $2, $2, $3
st $2, 8($fp) // store year, month and day on 8($sp)
lbu $2, 10($fp) // m = date1.month;
sb $2, 4($fp)
lbu $2, 14($fp) // s = date1.second;
sb $2, 0($fp)
addiu $sp, $sp, 24
ret $lr
.set macro
.set reorder
.end _Z9test_charv
$tmp1:
.size _Z9test_charv, ($tmp1)-_Z9test_charv

.type b,@object          # @b
.data
.globl b
b:
.asciz "abc"
.size b, 4

.type $_ZZ9test_charvE5date1,@object # @_ZZ9test_charvE5date1
.section .rodata.cst8,"aM",@progbits,8
.align 1
$_ZZ9test_charvE5date1:
.2byte 2012          # 0x7dc
.byte 11             # 0xb
.byte 25             # 0x19
.byte 9              # 0x9
.byte 40             # 0x28
.byte 15             # 0xf
.space 1
.size $_ZZ9test_charvE5date1, 8

```

To support load bool type, the following code added.

### Index/Chapter7\_1/Cpu0ISelLowering.cpp

```
Cpu0TargetLowering::
Cpu0TargetLowering(Cpu0TargetMachine &TM)
: TargetLowering(TM, new Cpu0TargetObjectFile()),
  Subtarget(&TM.getSubtarget<Cpu0Subtarget>()) {
...
// 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
setLoadExtAction(ISD::EXTLOAD, MVT::i1, Promote);
setLoadExtAction(ISD::ZEXTLOAD, MVT::i1, Promote);
setLoadExtAction(ISD::SEXTLOAD, MVT::i1, Promote);
...
}
```

Above code setLoadExtAction() are work enough. 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 support here. File ch\_run\_backend.cpp include the test fragment 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;
}
```

### Index/InputFiles/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:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch7_3.ll -o -
```

```

.section .mdebug.abi32
.previous
.file "ch7_3.11"
.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

```

#### Ibdex/InputFiles/ch\_run\_backend.cpp

```

...
bool test_load_bool()
{
    int a = 1;

    if (a < 0)
        return false;

    return true;
}

```

## 7.3 long long

Cpu0 borrow the Mips ABI which long is 32-bits and long long is 64-bits for C language type. To support long long, we add the following code to Chapter7\_1/.

#### Ibdex/Chapter7\_1/Cpu0ISelDAGToDAG.cpp

```

/// Select instructions not customized! Used for
/// expanded, promoted and normal instructions
SDNode* Cpu0DAGToDAGISel::Select(SDNode *Node) {
    unsigned Opcode = Node->getOpcode();

```

```

...
switch(Opcode) {
default: break;

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

    unsigned MOOp;
    if (Opcode == ISD::ADDE) {
        CmpLHS = InFlag.getValue(0);
        MOOp = Cpu0::ADDu;
    } else {
        CmpLHS = InFlag.getOperand(0);
        MOOp = Cpu0::SUBu;
    }

    SDValue Ops[] = { CmpLHS, InFlag.getOperand(1) };

    SDValue LHS = Node->getOperand(0);
    SDValue RHS = Node->getOperand(1);

    EVT VT = LHS.getValueType();
    SDNode *StatusWord = CurDAG->getMachineNode(Cpu0::CMP, dl, VT, Ops);
    SDValue Constant1 = CurDAG->getTargetConstant(1, VT);
    SDNode *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, MOOp, VT, MVT::Glue,
                                  LHS, SDValue(AddCarry, 0));
}

/// Mul with two results
case ISD::SMUL_LOHI:
case ISD::UMUL_LOHI: {
    if (NodeTy == MVT::i32)
        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));

    return NULL;
}
...
}

```

Run Chapter7\_1 with ch7\_4.cpp to get the result as follows,

lbdex/InputFiles/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:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_4.cpp -emit-llvm -o ch7_4.bc
1-160-134-62:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch7_4.bc -o -
.section .mdebug.abi32
.previous
.file "ch7_4.bc"
.text
.globl _Z13test_longlongv
.align 2
.type _Z13test_longlongv,@function
.ent _Z13test_longlongv      # @_Z13test_longlongv
_Z13test_longlongv:
.frame $fp,56,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:                                # %entry
addiu $sp, $sp, -56
addiu $2, $zero, 2
st $2, 52($fp)
addiu $2, $zero, 3
st $2, 48($fp)
addiu $2, $zero, 1
st $2, 44($fp)
st $2, 40($fp)
lui $2, 768
ori $2, $2, 4096
st $2, 36($fp)
lui $2, 512
ori $2, $2, 4096
st $2, 32($fp)
ld $3, 44($fp)
ld $2, 52($fp)
addu $5, $2, $3
ld $2, 48($fp)
ld $4, 40($fp)
st $5, 28($fp)
cmp $sw, $5, $3
mfsw $3
andi $3, $3, 1
```

```
addu $3, $3, $4
addu $2, $2, $3
st $2, 24($fp)
ld $3, 44($fp)
ld $4, 52($fp)
subu $t9, $4, $3
ld $2, 48($fp)
ld $5, 40($fp)
st $t9, 20($fp)
cmp $sw, $4, $3
mfsw $3
andi $3, $3, 1
addu $3, $3, $5
subu $2, $2, $3
st $2, 16($fp)
ld $2, 44($fp)
ld $3, 52($fp)
multu $3, $2
mflo $t9
mfhi $4
ld $5, 48($fp)
ld $t0, 40($fp)
st $t9, 12($fp)
mul $3, $3, $t0
addu $3, $4, $3
mul $2, $5, $2
addu $2, $3, $2
st $2, 8($fp)
ld $2, 32($fp)
ld $3, 36($fp)
mult $3, $2
mflo $2
mfhi $3
st $2, 4($fp)
st $3, 0($fp)
addiu $sp, $sp, 56
ret $lr
.set macro
.set reorder
.end _Z13test_longlongv
$tmp1:
.size _Z13test_longlongv, ($tmp1)-_Z13test_longlongv
```

## 7.4 float and double

Cpu0 only has integer instructions at this point. For float operations, the clang 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.

## 7.5 Array and struct support

LLVM use getelementptr to represent the array and struct type in C. Please reference section getelementptr of <sup>1</sup>. For ch7\_5.cpp, the llvm IR as follows,

---

<sup>1</sup> <http://llvm.org/docs/LangRef.html>

### Ibdex/InputFiles/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); // 12+2012=2024
}

// ch7_5.ll1
; 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

define i32 @main() nounwind ssp {
entry:
    %retval = alloca i32, align 4
    %day = alloca i32, align 4
    %i = alloca i32, align 4
    store i32 0, i32* %retval
    %0 = load i32* getelementptr inbounds (%struct.Date* @date, i32 0, i32 2),
    align 4
    store i32 %0, i32* %day, align 4
    %1 = load i32* getelementptr inbounds ([3 x i32]* @a, i32 0, i32 1), align 4
    store i32 %1, i32* %i, align 4
    ret i32 0
}
```

Run Chapter6\_1/ with ch7\_5.bc on static mode will get the incorrect asm file as follows,

```
1-160-134-62:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/bin/
Debug/llc -march=cpu0 -relocation-model=static -filetype=asm ch7_5.bc -o -
.section .mdebug.abi32
.previous
.file "ch7_5.bc"
.text
.globl _Z11test_structv
.align 2
.type _Z11test_structv,@function
.ent _Z11test_structv      # @_Z11test_structv
_Z11test_structv:
    .frame $fp,8,$lr
    .mask 0x00000000,0
    .set noreorder
```

```

.set nomacro
# BB#0:                                # %entry
addiu $sp, $sp, -8
lui $2, %hi(date)
addiu $2, $2, %lo(date)
    ld $2, 0($2)    // the correct one is    ld $2, 8($2)
    st $2, 4($fp)
    lui $2, %hi(a)
    addiu $2, $2, %lo(a)
    ld $2, 4($2)
    st $2, 0($fp)
    addiu $sp, $sp, 8
    ret $lr
.set macro
.set reorder
.end _Z11test_structv
$tmp1:
.size _Z11test_structv, ($tmp1)-_Z11test_structv

.type date,@object          # @date
.data
.globl date
.align 2
date:
    .4byte 2012             # 0x7dc
    .4byte 10               # 0xa
    .4byte 12               # 0xc
.size date, 12

.type a,@object            # @a
.globl a
.align 2
a:
    .4byte 2012             # 0x7dc
    .4byte 10               # 0xa
    .4byte 12               # 0xc
.size a, 12

```

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 use debug option in llc to see what’s wrong,

```

jonathantekiimac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/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]

...

```

By `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;
}

```

### lib/CodeGen/SelectionDAG/TargetLowering.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/Chapter7\_1/Cpu0ISelDAGToDAG.cpp

```
// 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(), ValTy);
        return true;
    }
}
}
```

Recall we have translated DAG list for date.day (add GlobalAddress<[3 x i32]\* @a> 0, Constant<8>) into (add (add Cpu0ISD::Hi (Cpu0II::MO\_ABS\_HI), Cpu0ISD::Lo(Cpu0II::MO\_ABS\_LO)), Constant<8>) by the following code in Cpu0ISelLowering.cpp.

### lbdex/Chapter6\_1/Cpu0ISelLowering.cpp

```
// Cpu0ISelLowering.cpp
SDValue Cpu0TargetLowering::LowerGlobalAddress(SDValue Op,
                                              SelectionDAG &DAG) const {
...
// %hi/%lo relocation
SDValue GAHi = DAG.getTargetGlobalAddress(GV, dl, MVT::i32, 0,
                                           Cpu0II::MO_ABS_HI);
SDValue GALo = DAG.getTargetGlobalAddress(GV, dl, MVT::i32, 0,
                                           Cpu0II::MO_ABS_LO);
SDValue HiPart = DAG.getNode(Cpu0ISD::Hi, dl, VTs, &GAHi, 1);
SDValue Lo = DAG.getNode(Cpu0ISD::Lo, dl, MVT::i32, GALo);
return DAG.getNode(ISD::ADD, dl, MVT::i32, HiPart, 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.

```
...
ld  $2, 8($2)
st  $2, 8($sp)
addiu $2, $zero, %hi(a)
shl $2, $2, 16
addiu $2, $2, %lo(a)
ld  $2, 4($2)
```

The ch7\_5\_2.cpp is for local variable initialization test. The result as follows,

#### Ibdex/InputFiles/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); // 12+2012=2024
}

118-165-79-206:InputFiles 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]* @_ZZ4mainE1a to i8*), i32 12, i32 4, i1 false)
    ret i32 0
}
; Function Attrs: nounwind
declare void @llvm.memcpy.p0i8.p0i8.i32(i8* nocapture, i8* nocapture, i32, i32, i1) #1

118-165-79-206:InputFiles Jonathan$ /usr/local/llvm/test/cmake_debug_build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch7_5_2.bc -o -
    .section .mdebug.abi32
    .previous
    .file "ch7_5_2.bc"
    .text
    .globl      main
    .align      2
    .type main,@function
    .ent main          # @_main
```

```
main:
    .frame      $fp,16,$lr
    .mask      0x00000000,0
    .set  noreorder
    .cupload   $t9
    .set  nomacro
# BB#0:                                # %entry
    addiu $sp, $sp, -16
    addiu $2, $zero, 0
    st   $2, 12($fp)
    ld   $2, %got($_ZZ4mainE1a)($gp)
    addiu $2, $2, %lo($_ZZ4mainE1a)
    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
    .set  macro
    .set  reorder
    .end  main
$tmp1:
    .size main, ($tmp1)-main

    .type $_ZZ4mainE1a,@object      # @_ZZ4mainE1a
    .section     .rodata,"a",@progbits
    .align        2
$_ZZ4mainE1a:
    .4byte      0                  # 0x0
    .4byte      1                  # 0x1
    .4byte      2                  # 0x2
    .size $_ZZ4mainE1a, 12
```

# CONTROL FLOW STATEMENTS

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.

## 8.1 Control flow statement

Run ch8\_1\_1.cpp with clang will get result as follows,

**lbdex/InputFiles/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
}

; Function Attrs: nounwind uwtable
define i32 @_Z13test_control1v() #0 {
entry:
%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
%h = alloca i32, align 4
%i = alloca i32, align 4
%j = alloca i32, align 4
store i32 0, i32* %a, align 4
store i32 1, i32* %b, align 4
store i32 2, i32* %c, align 4
store i32 3, i32* %d, align 4
store i32 4, i32* %e, align 4
store i32 5, i32* %f, align 4
store i32 6, i32* %g, align 4
store i32 7, i32* %h, align 4
store i32 8, i32* %i, align 4
store i32 9, i32* %j, align 4
%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

if.end: ; preds = %if.then, %entry
%2 = load i32* %b, align 4
%cmp1 = icmp ne i32 %2, 0
br i1 %cmp1, label %if.then2, label %if.end4

if.then2: ; preds = %if.end
%3 = load i32* %b, align 4
%inc3 = add nsw i32 %3, 1
store i32 %inc3, i32* %b, align 4

```

```
br label %if.end4

if.end4: ; preds = %if.then2, %if.end
%4 = load i32* %c, align 4
%cmp5 = icmp sgt i32 %4, 0
br i1 %cmp5, label %if.then6, label %if.end8

if.then6: ; preds = %if.end4
%5 = load i32* %c, align 4
%inc7 = add nsw i32 %5, 1
store i32 %inc7, i32* %c, align 4
br label %if.end8

if.end8: ; preds = %if.then6, %if.end4
%6 = load i32* %d, align 4
%cmp9 = icmp sge i32 %6, 0
br i1 %cmp9, label %if.then10, label %if.end12

if.then10: ; preds = %if.end8
%7 = load i32* %d, align 4
%inc11 = add nsw i32 %7, 1
store i32 %inc11, i32* %d, align 4
br label %if.end12

if.end12: ; preds = %if.then10, %if.end8
%8 = load i32* %e, align 4
%cmp13 = icmp slt i32 %8, 0
br i1 %cmp13, label %if.then14, label %if.end16

if.then14: ; preds = %if.end12
%9 = load i32* %e, align 4
%inc15 = add nsw i32 %9, 1
store i32 %inc15, i32* %e, align 4
br label %if.end16

if.end16: ; preds = %if.then14, %if.end12
%10 = load i32* %f, align 4
%cmp17 = icmp sle i32 %10, 0
br i1 %cmp17, label %if.then18, label %if.end20

if.then18: ; preds = %if.end16
%11 = load i32* %f, align 4
%inc19 = add nsw i32 %11, 1
store i32 %inc19, i32* %f, align 4
br label %if.end20

if.end20: ; preds = %if.then18, %if.end16
%12 = load i32* %g, align 4
%cmp21 = icmp sle i32 %12, 1
br i1 %cmp21, label %if.then22, label %if.end24

if.then22: ; preds = %if.end20
%13 = load i32* %g, align 4
%inc23 = add nsw i32 %13, 1
store i32 %inc23, i32* %g, align 4
br label %if.end24

if.end24: ; preds = %if.then22, %if.end20
```

```

%14 = load i32* %h, align 4
%cmp25 = icmp sge i32 %14, 1
br i1 %cmp25, label %if.then26, label %if.end28

if.then26: ; preds = %if.end24
%15 = load i32* %h, align 4
%inc27 = add nsw i32 %15, 1
store i32 %inc27, i32* %h, align 4
br label %if.end28

if.end28: ; preds = %if.then26, %if.end24
%16 = load i32* %i, align 4
%17 = load i32* %h, align 4
%cmp29 = icmp slt i32 %16, %17
br i1 %cmp29, label %if.then30, label %if.end32

if.then30: ; preds = %if.end28
%18 = load i32* %i, align 4
%inc31 = add nsw i32 %18, 1
store i32 %inc31, i32* %i, align 4
br label %if.end32

if.end32: ; preds = %if.then30, %if.end28
%19 = load i32* %a, align 4
%20 = load i32* %b, align 4
%cmp33 = icmp ne i32 %19, %20
br i1 %cmp33, label %if.then34, label %if.end36

if.then34: ; preds = %if.end32
%21 = load i32* %j, align 4
%inc35 = add nsw i32 %21, 1
store i32 %inc35, i32* %j, align 4
br label %if.end36

if.end36: ; preds = %if.then34, %if.end32
%22 = load i32* %a, align 4
%23 = load i32* %b, align 4
%add = add i32 %22, %23
%24 = load i32* %c, align 4
%add37 = add i32 %add, %24
%25 = load i32* %d, align 4
%add38 = add i32 %add37, %25
%26 = load i32* %e, align 4
%add39 = add i32 %add38, %26
%27 = load i32* %f, align 4
%add40 = add i32 %add39, %27
%28 = load i32* %g, align 4
%add41 = add i32 %add40, %28
%29 = load i32* %h, align 4
%add42 = add i32 %add41, %29
%30 = load i32* %i, align 4
%add43 = add i32 %add42, %30
%31 = load i32* %j, align 4
%add44 = add i32 %add43, %31
ret i32 %add44
}

```

The “**icmp ne**” stand for integer compare NotEqual, “**slt**” stands for Set Less Than, “**sle**” stands for Set Less Equal.

Run version Chapter6\_2/ with `llc -view-isel-dags` or `-debug` option, you can see it has translated **if** statement into `(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, We list 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 first addiu instruction as above which move `Constant<c>` into register, we have defined it before by the following code,

#### **lbdex/Chapter3\_5/Cpu0InstrInfo.td**

```
// Small immediates
def : Pat<(i32 immSExt16:$in),  
      (ADDiu ZERO, imm:$in)>;  
  
// Arbitrary immediates
def : Pat<(i32 imm:$imm),  
      (ORi (LUI (HI16 imm:$imm)), (LO16 imm:$imm))>;
```

For the last IR br, we translate unconditional branch `(br BasicBlock_01)` into `jmp BasicBlock_01` by the following pattern definition,

#### **lbdex/Chapter8\_1/Cpu0InstrInfo.td**

```
// Unconditional branch
class UncondBranch<bits<8> op, string instr_asm>:
    BranchBase<op, (outs), (ins brtarget:$imm24),
    !strconcat(instr_asm, "\t$imm24"), [(br bb:$imm24)], IIBranch> {
    let isBranch = 1;
    let isTerminator = 1;
    let isBarrier = 1;
    let hasDelaySlot = 0;
}
...
def JMP      : UncondBranch<0x26, "jmp">;
```

The pattern `[(br bb:$imm24)]` in class `UncondBranch` is translated into `jmp` machine instruction. The other two 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,

#### **lbdex/Chapter8\_1/Cpu0InstrInfo.td**

```
// brcond patterns
multiclass BrcondPats<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)>;

```

Above definition support (setne RC:\$lhs, RC:\$rhs) register to register compare. There are other compare pattern like, seteq, setlt, ... . In addition to seteq, setne, ..., we define setueq, setune, ..., by reference Mips code even though we didn't find how setune came from. We have tried to define unsigned int type, but clang still generate setne instead of setune. Pattern search order is according their appear order in context. The last pattern (brcond RC:\$cond, bb:\$dst) is meaning branch to \$dst if \$cond != 0, it is equal to (JNEOp (CMPOp RC:\$cond, ZEROReg), bb:\$dst) in cpu0 translation.

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 is correct even an instruction is inserted between CMP and JNE as follows,

```

cmp %2, %3
addiu $r1, $r2, 3 // $r1 register never be allocated to $SW
jne BasicBlock_02

```

The reserved registers setting by the following function code we defined before,

### Index/Chapter8\_1/Cpu0RegisterInfo.cpp

```

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

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

    const Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
    // Reserve GP if globalBaseRegFixed()
    if (Cpu0FI->globalBaseRegFixed())
        Reserved.set(Cpu0::GP);

    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 into another register class to prevent the SW register allocated to the register used by other instruction. The copyPhysReg() is called when DestReg and SrcReg belong to different Register Class.

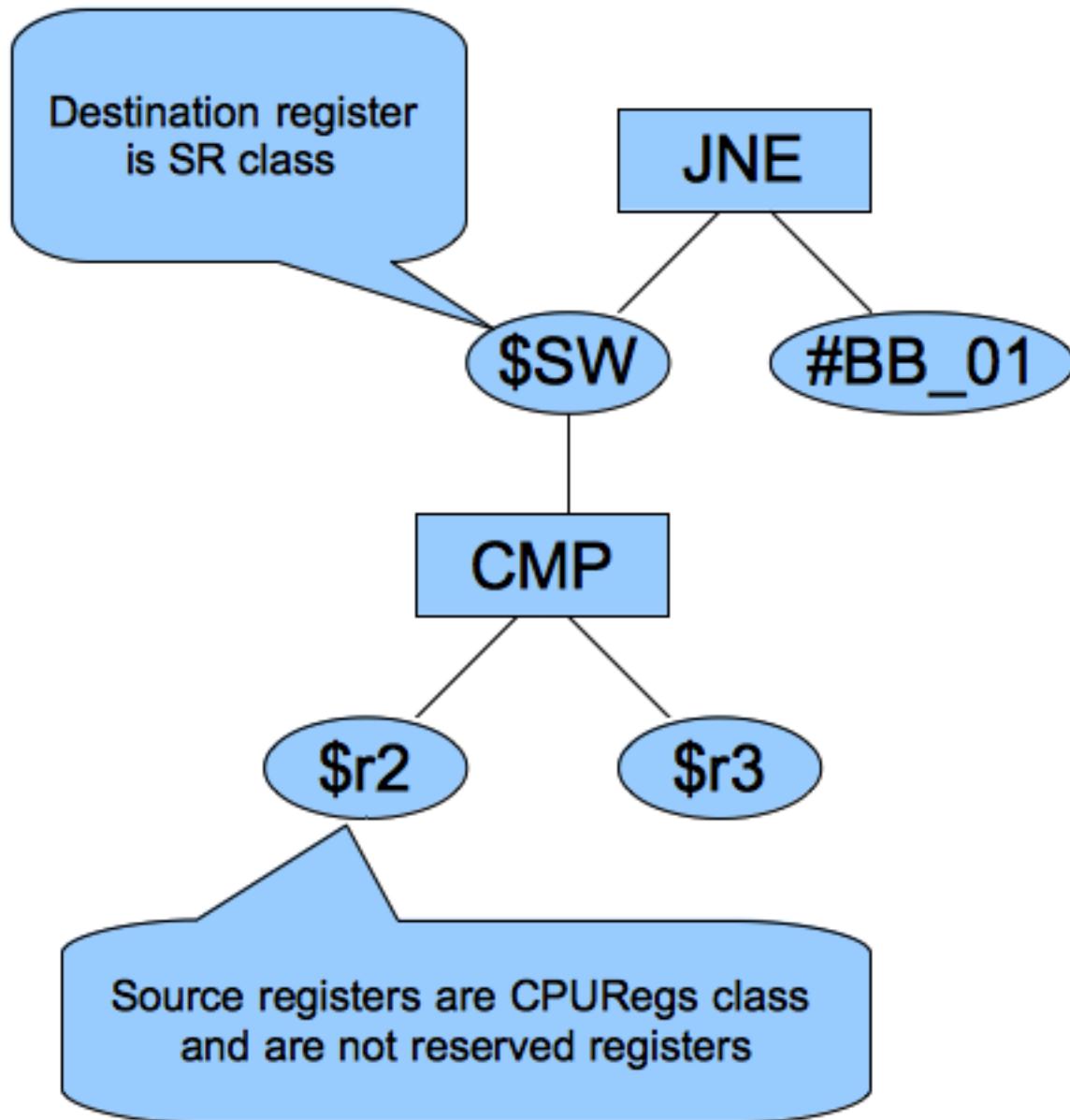


Figure 8.1: JNE (CMP \$r2, \$r3),

### lbdex/Chapter2/Cpu0RegisterInfo.td

```
//=====
// 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,
  // Callee save
  S0, S1, S2,
  // Reserved
  GP, FP,
  SP, LR, PC)>;
...
// Status Registers
def SR : RegisterClass<"Cpu0", [i32], 32, (add SW)>;
```

### lbdex/Chapter4\_2/Cpu0InstrInfo.cpp

```
//- Called when DestReg and SrcReg belong to different Register Class.
void Cpu0InstrInfo::
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 (SrcReg == Cpu0::SW)
    Opc = Cpu0::MFSW, SrcReg = 0;
}
else if (Cpu0::CPUREgsRegClass.contains(SrcReg)) { // Copy from CPU Reg.
...
  if (DestReg == Cpu0::SW)
    Opc = Cpu0::MTSW, DestReg = 0
}
```

Chapter8\_1/ include support for control flow statement. Run with it as well as the following llc option, you can get the obj file and dump it's content by gobjdump or hexdump as follows,

```
118-165-79-206:InputFiles Jonathan$ cat ch8_1_1.cpu0.s
...
ld  $4, 36($fp)
cmp $sw, $4, $3
jne $BB0_2
jmp $BB0_1
$BB0_1:          # %if.then
ld  $4, 36($fp)
addiu $4, $4, 1
st  $4, 36($fp)
```

```

$BB0_2:                                # %if.end
    ld $4, 32($fp)
    ...

118-165-79-206:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=obj
ch8_1_1.bc -o ch8_1_1.cpu0.o

118-165-79-206:InputFiles Jonathan$ hexdump ch8_1_1.cpu0.o
// jmp offset is 0x10=16 bytes which is correct
0000080 ..... 10 43 00 00
0000090 31 00 00 10 36 00 00 00 .....

```

The immediate value of jne (op 0x31) is 16; The offset between jne and \$BB0\_2 is 20 (5 words = 5\*4 bytes). Suppose the jne address is X, then the label \$BB0\_2 is X+20. Cpu0 is a RISC cpu0 with 3 stages of pipeline which are fetch, decode and execution according to cpu0 web site information. The cpu0 do branch instruction execution at decode stage which like mips. After the jne instruction fetched, the PC (Program Counter) is X+4 since cpu0 update PC at fetch stage. The \$BB0\_2 address is equal to PC+16 for the jne branch instruction execute at decode stage. List and explain this again as follows,

```

// Fetch instruction stage for jne instruction. The fetch stage
// can be divided into 2 cycles. First cycle fetch the
// instruction. Second cycle adjust PC = PC+4.
jne $BB0_2 // Do jne compare in decode stage. PC = X+4 at this stage.
// When jne immediate value is 16, PC = PC+16. It will fetch
// X+20 which equal to label $BB0_2 instruction, ld $2, 28($sp).
    jmp $BB0_1
$BB0_1:                                # %if.then
    ld $4, 36($fp)
    addiu $4, $4, 1
    st $4, 36($fp)
$BB0_2:                                # %if.end
    ld $4, 32($fp)

```

If cpu0 do “**jne**” compare in execution stage, then we should set PC=PC+12, offset of (\$BB0\_2, jne \$BB02) – 8, in this example.

Cpu0 is for teaching purpose and didn’t consider the performance with design. 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. Cpu0 take 2 instructions for conditional branch, (jne(cmp...)), while Mips use one instruction (bne).

Finally we list the code added for full support of control flow statement,

[Index/Chapter8\\_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp](#)

[Index/Chapter8\\_1/Cpu0ISelLowering.cpp](#)

```

Cpu0TargetLowering::
Cpu0TargetLowering(Cpu0TargetMachine &TM)
: TargetLowering(TM, new Cpu0TargetObjectFile()),
  Subtarget(&TM.getSubtarget<Cpu0Subtarget>()) {
...
// Used by legalize types to correctly generate the setcc result.
// Without this, every float setcc comes with a AND/OR with the result,
// we don't want this, since the fpcmp result goes to a flag register,
// which is used implicitly by brcond and select operations.

```

```
    AddPromotedToType(ISD::SETCC, MVT::i1, MVT::i32);  
    ...  
    setOperationAction(ISD::BRCOND, MVT::Other, Custom);  
    ...  
    // Operations not directly supported by Cpu0.  
    setOperationAction(ISD::BR_CC, MVT::i32, Expand);  
    ...  
}  
...  
SDValue Cpu0TargetLowering::  
LowerOperation(SDValue Op, SelectionDAG &DAG) const  
{  
    switch (Op.getOpcode())  
    {  
        case ISD::BRCOND: return LowerBRCOND(Op, DAG);  
        ...  
    }  
    ...  
}  
...  
SDValue Cpu0TargetLowering::  
LowerBRCOND(SDValue Op, SelectionDAG &DAG) const  
{  
    return Op;  
}
```

### [Index/Chapter8\\_1/Cpu0ISelLowering.h](#)

```
SDValue LowerBRCOND(SDValue Op, SelectionDAG &DAG) const;
```

### [Index/Chapter8\\_1/Cpu0MCInstLower.cpp](#)

```
MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,  
                                              MachineOperandType MOTy,  
                                              unsigned Offset) const {  
    ...  
    switch (MOTy) {  
        ...  
        case MachineOperand::MO_MachineBasicBlock:  
            Symbol = MO.getMBB()->getSymbol();  
            break;  
        ...  
        case MachineOperand::MO_BlockAddress:  
            Symbol = AsmPrinter.GetBlockAddressSymbol(MO.getBlockAddress());  
            Offset += MO.getOffset();  
            break;  
        ...  
    }  
  
    MCOperand Cpu0MCInstLower::LowerOperand(const MachineOperand& MO,  
                                           unsigned offset) const {  
        MachineOperandType MOTy = MO.getType();  
  
        switch (MOTy) {  
            ...  
        }
```

```

case MachineOperand::MO_MachineBasicBlock:
...
case MachineOperand::MO_BlockAddress:
...
}
...
}

```

### Ibdex/Chapter8\_1/Cpu0InstrFormats.td

```

//=====
// 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;
}

```

### Ibdex/Chapter8\_1/Cpu0InstrInfo.td

```

// Cpu0InstrInfo.td
// Instruction operand types
def brtarget24 : Operand<OtherVT> {
    let EncoderMethod = "getBranchTargetOpValue";
    let OperandType = "OPERAND_PCREL";
    let DecoderMethod = "DecodeBranchTarget";
}
// JMP
def jmptarget : Operand<OtherVT> {
    let EncoderMethod = "getJumpTargetOpValue";
    let OperandType = "OPERAND_PCREL";
    let DecoderMethod = "DecodeJumpRelativeTarget";
}
...
/// Conditional Branch
class CBranch24<bits<8> op, string instr_asm, RegisterClass RC,
                  list<Register> UseRegs>:
    FJ<op, (outs), (ins RC:$ra, brtarget:$addr),
                  !strconcat(instr_asm, "\t$addr"),
                  [], IIBranch> {
        let isBranch = 1;
        let isTerminator = 1;
        let hasDelaySlot = 0;
        let neverHasSideEffects = 1;
    }

    // Unconditional branch, such as JMP
    class UncondBranch<bits<8> op, string instr_asm>:
        FJ<op, (outs), (ins brtarget:$addr),

```

```

        !strconcat(instr_asm, "\t$addr"), [(br bb:$addr)], IIBranch> {
let isBranch = 1;
let isTerminator = 1;
let isBarrier = 1;
let hasDelaySlot = 0;
}

...
/// Jump and Branch Instructions
def JEQ      : CBranch<0x30, "jeq", CPURegs>;
def JNE      : CBranch<0x31, "jne", CPURegs>;
def JLT      : CBranch<0x32, "jlt", CPURegs>;
def JGT      : CBranch<0x33, "jgt", CPURegs>;
def JLE      : CBranch<0x34, "jle", CPURegs>;
def JGE      : CBranch<0x35, "jge", CPURegs>;
def JMP      : UncondBranch<0x36, "jmp">;
...

// brcond patterns
multiclass BrcondPats<RegisterClass RC, Instruction JEQOp,
Instruction JNEOp, Instruction JLTOp, Instruction JGTOp,
Instruction JLEOp, Instruction JGEOp, Instruction CMPOp,
Register ZEROReg> {
def : Pat<(brcond (i32 (seteq RC:$lhs, RC:$rhs)), bb:$dst),
(JEQOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;
def : Pat<(brcond (i32 (setueq RC:$lhs, RC:$rhs)), bb:$dst),
(JEQOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;
def : Pat<(brcond (i32 (setne RC:$lhs, RC:$rhs)), bb:$dst),
(JNEOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;
def : Pat<(brcond (i32 (setune RC:$lhs, RC:$rhs)), bb:$dst),
(JNEOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;
def : Pat<(brcond (i32 (setlt RC:$lhs, RC:$rhs)), bb:$dst),
(JLTOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;
def : Pat<(brcond (i32 (setgt RC:$lhs, RC:$rhs)), bb:$dst),
(JGTOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;
def : Pat<(brcond (i32 (setult RC:$lhs, RC:$rhs)), bb:$dst),
(JLTOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;
def : Pat<(brcond (i32 (setgt RC:$lhs, RC:$rhs)), bb:$dst),
(JGTOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;
def : Pat<(brcond (i32 (setugt RC:$lhs, RC:$rhs)), bb:$dst),
(JGTOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;
def : Pat<(brcond (i32 (setle RC:$lhs, RC:$rhs)), bb:$dst),
(JLEOp (CMPOp RC:$rhs, RC:$lhs), bb:$dst)>;
def : Pat<(brcond (i32 (setule RC:$lhs, RC:$rhs)), bb:$dst),
(JLEOp (CMPOp RC:$rhs, RC:$lhs), bb:$dst)>;
def : Pat<(brcond (i32 (setge RC:$lhs, RC:$rhs)), bb:$dst),
(JGEOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;
def : Pat<(brcond (i32 (setuge RC:$lhs, RC:$rhs)), bb:$dst),
(JGEOp (CMPOp RC:$lhs, RC:$rhs), bb:$dst)>;

def : Pat<(brcond RC:$cond, bb:$dst),
(JNEOp (CMPOp RC:$cond, ZEROReg), bb:$dst)>;
}

defm : BrcondPats<CPURegs, JEQ, JNE, JLT, JGT, JLE, JGE, CMP, ZERO>;

```

The ch8\_1\_2.cpp is for “**nest if**” test. The ch8\_1\_3.cpp is the “**for loop**” as well as “**while loop**”, “**continue**”, “**break**”, “**goto**” test. The ch8\_1\_5.cpp is for “**goto**” test. You can run with them if you like to test more.

The ch8\_1\_4.cpp is for test C operators ==, !=, &&, ||. No code need to add since we have take care them before. But it can be test only when the control flow statement support is ready, as follows,

**Ibdex/InputFiles/ch8\_1\_4.cpp**

```
int main()
{
    int a[3]={0, 1, 2};

    return 0;
}

118-165-78-230:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch8_1_4.cpp -emit-llvm -o ch8_1_4.bc
118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch8_1_4.bc -o
ch8_1_4.cpu0.s
118-165-78-230:InputFiles Jonathan$ cat ch8_1_4.cpu0.s
    .section .mdebug.abi32
    .previous
    .file "ch8_1_4.bc"
    .text
    .globl main
    .align 2
    .type main,@function
    .ent main                      # @main
main:
    .cfi_startproc
    .frame $sp,16,$lr
    .mask 0x00000000,0
    .set noreorder
    .set nomacro
# BB#0:
    addiu $sp, $sp, -16
$tmp1:
    .cfi_def_cfa_offset 16
    addiu $3, $zero, 0
    st $3, 12($sp)
    st $3, 8($sp)
    addiu $2, $zero, 1
    st $2, 4($sp)
    addiu $2, $zero, 2
    st $2, 0($sp)
    ld $4, 8($sp)
    cmp $4, $3
    jne $BB0_2                  // a != 0
    jmp $BB0_1
$BB0_1:                      // a == 0
    ld $3, 4($sp)
    cmp $3, $2
    jeq $BB0_3                  // b == 2
    jmp $BB0_2
$BB0_2:
    ld $3, 0($sp)
    cmp $3, $2                  // c == 2
    jeq $BB0_4
    jmp $BB0_3
$BB0_3:                      // (a == 0 && b == 2) || (c != 2)
    ld $2, 8($sp)
    addiu $2, $2, 1            // a++
    st $2, 8($sp)
```

```
$BB0_4:  
    addiu $sp, $sp, 16  
    ret $lr  
.set macro  
.set reorder  
.end main  
$tmp2:  
.size main, ($tmp2)-main  
.cfi_endproc
```

## 8.2 RISC CPU knowledge

As mentioned in the previous section, cpu0 is a RISC (Reduced Instruction Set Computer) CPU with 3 stages of pipeline. RISC CPU is full in world. Even the X86 of CISC (Complex Instruction Set Computer) is RISC inside. (It translate CISC instruction into micro-instruction 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 what we read.

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 until the book is written) is for the introduction (simple). “Computer Architecture: A Quantitative Approach” (there are 5 editions until the book is written) is more complicate and deep in CPU architecture.

Above two books use Mips CPU as 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

The subroutine/function call of backend code translation is supported in this chapter. A lots of code needed in function call. We break it down according llvm supplied interface for easy to explanation. This chapter start from introducing the Mips stack frame structure since we borrow many part of ABI from it. Although each CPU has it's own ABI, most of RISC CPUs ABI are similar. In addition to support fixed number of arguments function call, cpu0 also support variable number of arguments 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 some knowledge about Lowering process. Section “4.5.1 Calling Conventions” of tricore\_llvm.pdf is the related materials you can reference.

This chapter is more complicate than any of the previous chapter. 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” which listed in section “RISC CPU knowledge” of chapter “Control flow statement”<sup>1</sup>, “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 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.

Run `llc -march=mips` for `ch9_1.bc`, you will get the following result. See comment “//”.

### lbdex/InputFiles/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()
```

---

<sup>1</sup> <http://jonathan2251.github.com/lbd/ctrlflow.html#risc-cpu-knowledge>

Base	Offset	Contents	Frame
old \$sp	+16	unspecified	<i>High addresses</i>
		...	
		variable size	
		(if present) incoming arguments passed in stack frame	
\$sp	+0	space for incoming arguments 1-4	Previous
		locals and temporaries	
		general register save area	
		floating-point register save area	
		argument build area	
			<i>Low addresses</i>

Figure 9.1: Mips stack frame

```
{  
    char str[81] = "Hello world";  
    int a = sum_i(1, 2, 3, 4, 5, 6);  
  
    return a;  
}  
  
118-165-78-230:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c  
ch9_1.cpp -emit-llvm -o ch9_1.bc  
118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/  
bin/Debug/llc -march=mips -relocation-model=pic -filetype=asm ch9_1.bc -o  
ch9_1.mips.s  
118-165-78-230:InputFiles Jonathan$ cat ch9_1.mips.s  
.section .mdebug.abi32  
.previous  
.file "ch9_1.bc"  
.text  
.globl _Z5sum_iiiiiii  
.align 2  
.type _Z5sum_iiiiiii,@function  
.set nomips16 # @_Z5sum_iiiiiii  
.ent _Z5sum_iiiiiii  
_Z5sum_iiiiiii:  
.cfi_startproc  
.frame $sp,32,$ra  
.mask 0x00000000,0  
.fmask 0x00000000,0  
.set noreorder  
.set nomacro  
.set noat  
# BB#0:  
    addiu $sp, $sp, -32  
$tmp1:  
.cfi_def_cfa_offset 32  
    sw $4, 28($sp)  
    sw $5, 24($sp)  
    sw $t9, 20($sp)  
    sw $7, 16($sp)  
    lw $1, 48($sp) // load argument 5  
    sw $1, 12($sp)  
    lw $1, 52($sp) // load argument 6  
    sw $1, 8($sp)  
    lw $2, 24($sp)  
    lw $3, 28($sp)  
    addu $2, $3, $2  
    lw $3, 20($sp)  
    addu $2, $2, $3  
    lw $3, 16($sp)  
    addu $2, $2, $3  
    lw $3, 12($sp)  
    addu $2, $2, $3  
    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)
addiu $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 load 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.

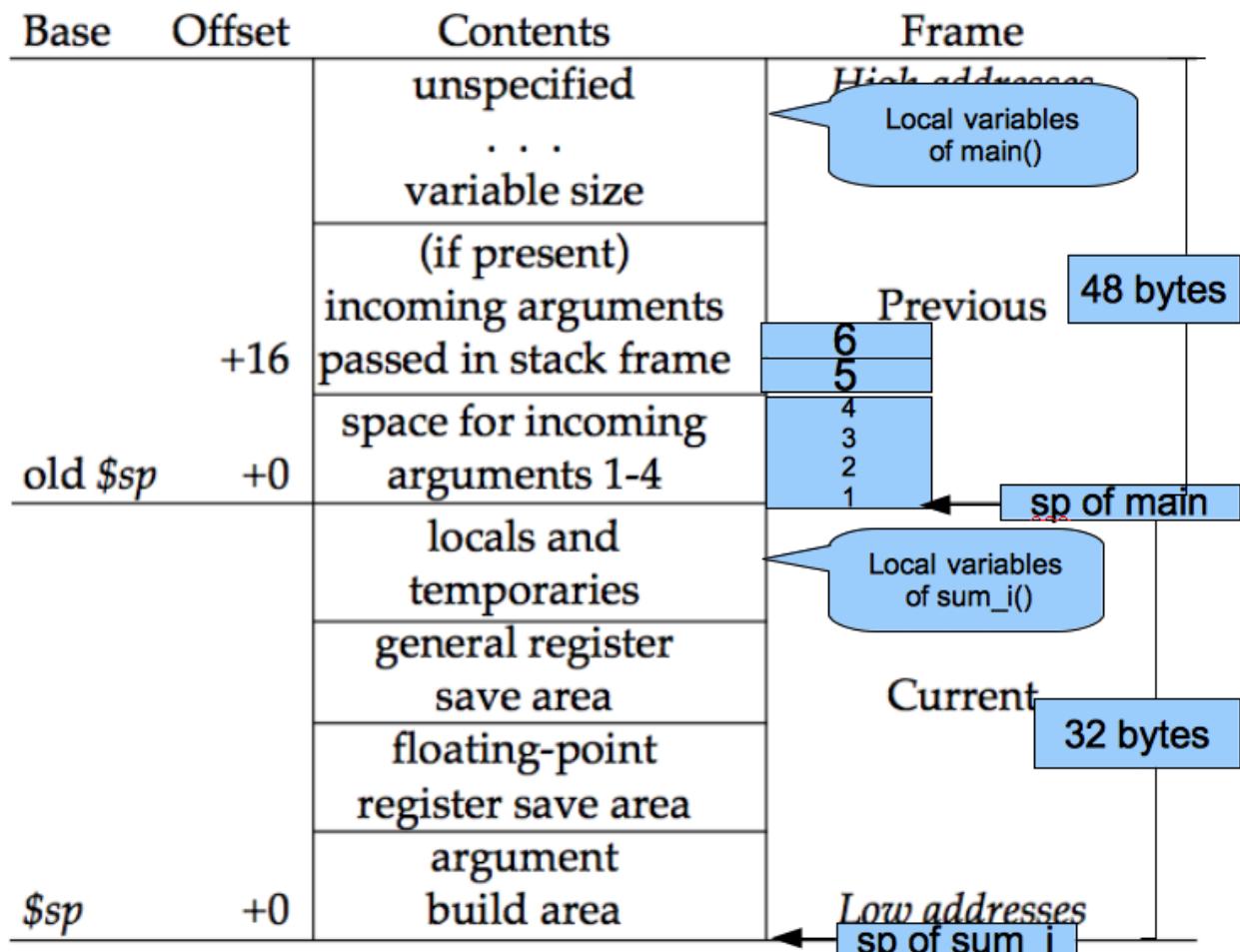


Figure 9.2: Mips arguments location in stack frame

The 007-2418-003.pdf in <sup>2</sup> is the Mips assembly language manual. <sup>3</sup> 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\_1/ with ch9\_1.cpp and see what happens.

```
118-165-79-31:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/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/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_1.bc -o
ch9_1.cpu0.s
1. Running pass 'Function Pass Manager' on module 'ch9_1.bc'.
2. Running pass 'CPU0 DAG->DAG Pattern Instruction Selection' on function
'@_Z5sum_iiiiiii'.
Illegal instruction: 4
```

Since Chapter8\_1/ 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. We choose pass arguments all in stack frame. We don't reserve any dedicated register for arguments passing since cpu0 has only 16 registers while Mips has 32 registers. Cpu0CallingConv.td is defined for cpu0 passing rule as follows,

### Ibdex/Chapter9\_1/Cpu0CallingConv.td

As above, CC\_Cpu0 is the cpu0 Calling Convention which delegate to CC\_Cpu0EABI and define the CC\_Cpu0EABI. The reason we don't define the Calling Convention directly in CC\_Cpu0 is that a real general CPU like Mips can have several Calling Convention. Combine with the mechanism of "section Target Registration" <sup>4</sup> which llvm supplied, we can use different Calling Convention in different target. Although cpu0 only have a Calling Convention right now, define with a dedicate Call Convention name (CC\_Cpu0EABI in this example) is a better solution for system expand, and naming your Calling Convention. CC\_Cpu0EABI as above, say it pass arguments in stack frame.

Function LowerFormalArguments() charge function incoming arguments creation. We define it as follows,

### Ibdex/Chapter9\_1/Cpu0ISelLowering.cpp

```
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                               SmallVectorImpl<SDValue> &InVals) const {
    return CLI.Chain;
}
...
/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
```

<sup>2</sup> <https://www.dropbox.com/sh/2pkh1fewlq2zag9/OHnrYn2nOs/doc/MIPSproAssemblyLanguageProgrammerGuide>

<sup>3</sup> <http://www.linux-mips.org/pub/linux/mips/doc/ABI/mipsabi.pdf>

<sup>4</sup> <http://jonathan2251.github.com/lbd/llvmstructure.html#target-registration>

```

SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool isVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         DebugLoc dl, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                         const {
MachineFunction &MF = DAG.getMachineFunction();
MachineFrameInfo *MFI = MF.getFrameInfo();
Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

Cpu0FI->setVarArgsFrameIndex(0);

// Used with vargs to accumulate store chains.
std::vector<SDValue> OutChains;

// Assign locations to all of the incoming arguments.
SmallVector<CCValAssign, 16> ArgLocs;
CCState CCInfo(CallConv, isVarArg, DAG.getMachineFunction(),
               getTargetMachine(), ArgLocs, *DAG.getContext());

CCInfo.AnalyzeFormalArguments(Ins, CC_Cpu0);

Function::const_arg_iterator FuncArg =
  DAG.getMachineFunction().getFunction()->arg_begin();
int LastFI = 0;// Cpu0FI->LastInArgFI is 0 at the entry of this function.

for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i, ++FuncArg) {
  CCValAssign &VA = ArgLocs[i];
  EVT ValVT = VA.getValVT();
  ISD::ArgFlagsTy Flags = Ins[i].Flags;
  bool IsRegLoc = VA.isRegLoc();

  if (Flags.isByVal()) {
    assert(Flags.getByValSize() &&
           "ByVal args of size 0 should have been ignored by front-end.");
    continue;
  }
  // sanity check
  assert(VA.isMemLoc());

  // The stack pointer offset is relative to the caller stack frame.
  LastFI = MFI->CreateFixedObject(ValVT.getSizeInBits()/8,
                                   VA.getLocMemOffset(), true);

  // Create load nodes to retrieve arguments from the stack
  SDValue FIN = DAG.getFrameIndex(LastFI, getPointerTy());
  InVals.push_back(DAG.getLoad(ValVT, dl, Chain, FIN,
                               MachinePointerInfo::getFixedStack(LastFI),
                               false, false, false, 0));
}

Cpu0FI->setLastInArgFI(LastFI);
// 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[0], OutChains.size());
    }
    return Chain;
}

```

### Ibdex/Chapter9\_1/Cpu0ISelLowering.h

```

virtual SDValue
LowerCall(TargetLowering::CallLoweringInfo &CLI,
          SmallVectorImpl<SDValue> &InVals) const;
// LowerCall: outgoing arguments

```

Refresh “section Global variable”<sup>5</sup>, we handled global variable translation by create the IR DAG in LowerGlobalAddress() first, and then do the Instruction Selection by their corresponding machine instruction DAG in Cpu0InstrInfo.td. LowerGlobalAddress() is called when `l1c` meet the global variable access. LowerFormalArguments() work with 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 and we use the stack frame only for arguments passing without any arguments pass in registers. So `ArgLocs.size()` is 6, each argument information is in `ArgLocs[i]` and `ArgLocs[i].isMemLoc()` is true. In “**for loop**”, it create each frame index object by `LastFI = MFI->CreateFixedObject(ValVT, getSizeInBits()/8, VA.getLocMemOffset(), true)` and `FIN = DAG.getFrameIndex>LastFI, getPointerTy()`. And then create IR DAG load node and put the load node into vector `InVals` by `InVals.push_back(DAG.getLoad(ValVT, dl, Chain, FIN, MachinePointerInfo::getFixedStack>LastFI, false, false, false, 0))`. `Cpu0FI->setVarArgsFrameIndex(0)` and `Cpu0FI->setLastInArgFI>LastFI` are called when before and after above work. In `ch9_1.cpp` example, `LowerFormalArguments()` will be called twice. First time is for `sum_i()` which will create 6 load DAG for 6 incoming arguments passing into this function. Second time is for `main()` which didn’t create any load DAG for no incoming argument passing into `main()`. In addition to `LowerFormalArguments()` which create the load DAG, we need to define the `loadRegFromStackSlot()` 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.

### Ibdex/Chapter9\_1/Cpu0InstrInfo.cpp

#### Ibdex/Chapter9\_1/Cpu0InstrInfo.h

```

virtual void loadRegFromStackSlot(MachineBasicBlock &MBB,
                                MachineBasicBlock::iterator MBBI,
                                unsigned DestReg, int FrameIndex,
                                const TargetRegisterClass *RC,
                                const TargetRegisterInfo *TRI) const;

```

In addition to Calling Convention and `LowerFormalArguments()`, `Chapter9_1/` add the following code for `cpu0` instructions **swi** (Software Interrupt), **jsub** and **jalr** (function call) definition and printing.

### Ibdex/Chapter9\_1/Cpu0InstrInfo.td

```

def SDT_Cpu0JmpLink      : SDTypeProfile<0, 1, [SDTCisVT<0, iPTR>]>;
...
// Call
def Cpu0JmpLink : SDNode<"Cpu0ISD::JmpLink", SDT_Cpu0JmpLink,
                      [SDNPHasChain, SDNPOutGlue, SDNPOptInGlue,

```

<sup>5</sup> <http://jonathan2251.github.com/lbd/globalvar.html#global-variable>

```

        SDNPVariadic] >;
    ...
def calltarget : Operand<iPTR> {
    let EncoderMethod = "getJumpTargetOpValue";
}
...
// Jump and Link (Call)
let isCall=1, hasDelaySlot=0 in {
    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> {
    }

    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:$rb)], IIBranch> {
            let rc = 0;
            let ra = 14;
            let shamt = 0;
        }
    }
}
...
/// Jump and Branch Instructions
def SWI : JumpLink<0x2a, "swi">;
def JSUB : JumpLink<0x2b, "jsub">;
...
def JALR : JumpLinkReg<0x2e, "jalr", CPURegs>;
...
def : Pat<(Cpu0JmpLink (i32 tglobaladdr:$dst)),
        (JSUB tglobaladdr:$dst)>;
def : Pat<(Cpu0JmpLink (i32 texternalsym:$dst)),
        (JSUB texternalsym:$dst)>;
...

```

### Ibdex/Chapter9\_1/InstPrinter/Cpu0InstPrinter.cpp

```

static void printExpr(const MCExpr *Expr, raw_ostream &OS) {
    switch (Kind) {
    ...
    case MCSymbolRefExpr::VK_Cpu0_GOT_CALL: OS << "%call16("; break;
    ...
}
...
}

```

### Ibdex/Chapter9\_1/Cpu0MCInstLower.cpp

```

MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,
                                              MachineOperandType MOTy,
                                              unsigned Offset) const {
    ...
    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 {
...
switch (MOTy) {
...
case MachineOperand::MO_ExternalSymbol:
case MachineOperand::MO_BlockAddress:
    return LowerSymbolOperand(MO, MOTy, offset);
...
}
...
}

```

lbdex/Chapter9\_1/MCTargetDesc/Cpu0AsmBackend.cpp

```
case Cpu0::fixup_Cpu0_CALL16:
```

lbdex/Chapter9\_1/MCTargetDesc/Cpu0ELFObjectWriter.cpp

```
case Cpu0::fixup_Cpu0_CALL16:
    Type = ELF::R_CPU0_CALL16;
    break;
```

lbdex/Chapter9\_1/MCTargetDesc/Cpu0FixupKinds.h

```
// resulting in - R_CPU0_CALL16.
fixup_Cpu0_CALL16,
```

lbdex/Chapter9\_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
unsigned Cpu0MCCodeEmitter::
getJumpTargetOpValue(const MCInst &MI, unsigned OpNo,
                     SmallVectorImpl<MCFixup> &Fixups) const {
...
    if (Opcode == Cpu0::JSUB || Opcode == Cpu0::JMP)
        Fixups.push_back(MCFixup::Create(0, Expr,
                                         MCFixupKind(Cpu0::fixup_Cpu0_PC24)));
    else if (Opcode == Cpu0::SWI)
        Fixups.push_back(MCFixup::Create(0, Expr,
                                         MCFixupKind(Cpu0::fixup_Cpu0_24)));
}
```

```

...
} // lbd document - mark - getJumpTargetOpValue
...
unsigned Cpu0MCCodeEmitter::
getMachineOpValue(const MCInst &MI, const MCOperand &MO,
                 SmallVectorImpl<MCFixup> &Fixups) const {
...
switch(cast<MCSymbolRefExpr>(Expr)->getKind()) {
...
case MCSymbolRefExpr::VK_Cpu0_GOT_CALL:
    FixupKind = Cpu0::fixup_Cpu0_CALL16;
    break;
...
}
...
}

unsigned Cpu0MCCodeEmitter::
getMachineOpValue(const MCInst &MI, const MCOperand &MO,
                 SmallVectorImpl<MCFixup> &Fixups) const {
...
case MCSymbolRefExpr::VK_Cpu0_GOT_CALL:
    FixupKind = Cpu0::fixup_Cpu0_CALL16;
    break;
...
}

```

### Ibdex/Chapter9\_1/Cpu0MachineFucntion.h

```

class Cpu0FunctionInfo : public MachineFunctionInfo {
...
    /// VarArgsFrameIndex - FrameIndex for start of varargs area.
int VarArgsFrameIndex;

    // 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.
...

public:
    Cpu0FunctionInfo(MachineFunction& MF)
    : ...
        VarArgsFrameIndex(0), InArgFIRange(std::make_pair(-1, 0)),
        OutArgFIRange(std::make_pair(-1, 0)), GPFI(0), DynAllocFI(0),
        ...
        {}

    bool isInArgFI(int FI) const {
        return FI <= InArgFIRange.first && FI >= InArgFIRange.second;
    }
    void setLastInArgFI(int FI) { InArgFIRange.second = FI; }

```

```

void extendOutArgFIRange(int FirstFI, int LastFI) {
    if (!OutArgFIRange.second)
        // this must be the first time this function was called.
        OutArgFIRange.first = FirstFI;
        OutArgFIRange.second = LastFI;
}

int getGPFI() const { return GPFI; }
void setGPFI(int FI) { GPFI = FI; }
bool needGPSaveRestore() const { return getGPFI(); }
bool isGPFI(int FI) const { return GPFI && GPFI == FI; }

// The first call to this function creates a frame object for dynamically
// allocated stack area.
int getDynAllocFI() const {
    if (!DynAllocFI)
        DynAllocFI = MF.getFrameInfo()->CreateFixedObject(4, 0, true);

    return DynAllocFI;
}
bool isDynAllocFI(int FI) const { return DynAllocFI && DynAllocFI == FI; }
...
int getVarArgsFrameIndex() const { return VarArgsFrameIndex; }
void setVarArgsFrameIndex(int Index) { VarArgsFrameIndex = Index; }
...
};


```

The SWI, JSUB and JALR defined in Cpu0InstrInfo.td as above all use Cpu0JmpLink node. They are distinguishable since both SWI and JSUB use “imm” operand while JALR use register operand. JSUB take the priority to match since we set the following code in Cpu0InstrInfo.td.

### Ibdex/Chapter9\_1/Cpu0InstrInfo.td

```

def : Pat<(Cpu0JmpLink (i32 tglobaladdr:$dst)),
        (JSUB tglobaladdr:$dst)>;
def : Pat<(Cpu0JmpLink (i32 texternalsym:$dst)),
        (JSUB texternalsym:$dst)>;

```

The code tells TableGen generate pattern match pattern to match the “imm” for “tglobaladdr” pattern first. If it fails then try to match “texternalsym” next. The function you declared is “tglobaladdr”, the function which implicit used by llvm most are “texternalsym” such as “memcpy”. The “memcpy” will be generated when define a long string. The ch9\_1\_2.cpp is an example to generate “memcpy” function call. It will be shown in next section of Chapter9\_2 example code. Even though SWI have no chance to match in C/C++ language. We define it for easy to implement assembly parser which introduced in Chapter 11. This SWI definition will save us to implement the assembly parser for this instruction. TableGen will generate information for SWI instruction in assembly and ELF obj encode automatically. 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
/*677*/          OPC_MorphNodeTo, TARGET_VAL(Cpu0::SWI), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
    0/*#VTS*/, 1/*#Ops*/, 2,
    // Src: (Cpu0JmpLink (imm:iPTR):$target) - Complexity = 6
    // Dst: (SWI (imm:iPTR):$target)
/*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)
/*692*/          0, /*End of Scope*/
    /*SwitchOpcode*/ 11, TARGET_VAL(ISD::TargetGlobalAddress), // ->707
/*696*/          OPC_CheckType, MVT::i32,
/*698*/          OPC_MoveParent,
/*699*/          OPC_EmitMergeInputChains1_0,
/*700*/          OPC_MorphNodeTo, TARGET_VAL(Cpu0::JSUB), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
    0/*#VTS*/, 1/*#Ops*/, 1,
    // Src: (Cpu0JmpLink (tglobaladdr:i32):$dst) - Complexity = 6
    // Dst: (JSUB (tglobaladdr:i32):$dst)
    /*SwitchOpcode*/ 11, TARGET_VAL(ISD::TargetExternalSymbol), // ->721
/*710*/          OPC_CheckType, MVT::i32,
/*712*/          OPC_MoveParent,
/*713*/          OPC_EmitMergeInputChains1_0,
/*714*/          OPC_MorphNodeTo, TARGET_VAL(Cpu0::JSUB), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
    0/*#VTS*/, 1/*#Ops*/, 1,
    // Src: (Cpu0JmpLink (texternalsym:i32):$dst) - Complexity = 6
    // Dst: (JSUB (texternalsym:i32):$dst)
    0, // EndSwitchOpcode
/*722*/          /*Scope*/ 10, /*->733*/
/*723*/          OPC_CheckChild1Type, MVT::i32,
/*725*/          OPC_EmitMergeInputChains1_0,
/*726*/          OPC_MorphNodeTo, TARGET_VAL(Cpu0::JALR), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
    0/*#VTS*/, 1/*#Ops*/, 1,
    // Src: (Cpu0JmpLink CPURegs:i32:$rb) - Complexity = 3
    // Dst: (JALR CPURegs:i32:$rb)
/*733*/          0, /*End of Scope*/

```

After above changes, you can run Chapter9\_1/ with ch9\_1.cpp and see what happens in the following,

```

118-165-79-83:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/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/
bin/Debug/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 depicted 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/Chapter9\\_2/Cpu0ISelLowering.cpp](#)

[lbdex/Chapter9\\_2/Cpu0ISelLowering.h](#)

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() create 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. DAG.getCALLSEQ\_START() and DAG.getCALLSEQ\_END() are set before the “**for loop**” and after call subroutine, they insert CALLSEQ\_START, CALLSEQ\_END, and translate into pseudo machine instructions !ADJCALLSTACKDOWN, !ADJCALLSTACKUP later according Cpu0InstrInfo.td definition as follows.

[lbdex/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
//=====

// 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)]>;
}

```

Like load incoming arguments, we need to implement storeRegToStackSlot() for store outgoing arguments to stack frame offset.

### Ibdex/Chapter9\_2/Cpu0InstrInfo.cpp

```
//- st SrcReg, MMO(FI)
void Cpu0InstrInfo::
storeRegToStackSlot(MachineBasicBlock &MBB, MachineBasicBlock::iterator I,
                    unsigned SrcReg, bool isKill, int FI,
                    const TargetRegisterClass *RC,
                    const TargetRegisterInfo *TRI) const {
    DebugLoc DL;
    if (I != MBB.end()) DL = I->getDebugLoc();
    MachineMemOperand *MMO = GetMemOperand(MBB, FI, MachineMemOperand::MOStore);

    unsigned Opc = 0;

    if (RC == Cpu0::CPURegsRegisterClass)
        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);
} // lbd document - mark - storeRegToStackSlot
```

Now, let's run Chapter9\_2/ with ch9\_1.cpp to get result as follows (see comment //),

### Ibdex/Chapter9\_2/Cpu0InstrInfo.h

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

```
118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_1.bc -o
ch9_1.cpu0.s
118-165-78-230:InputFiles Jonathan$ cat ch9_1.cpu0.s
    .section .mdebug.abi32
    .previous
    .file "ch9_1.bc"
    .text
    .globl _Z5sum_iiiiii
    .align 2
    .type _Z5sum_iiiiii,@function
    .ent _Z5sum_iiiiii          # @_Z5sum_iiiiii
_Z5sum_iiiiii:
    .cfi_startproc
    .frame $sp,32,$lr
    .mask 0x00000000,0
    .set noreorder
    .cupload $t9
    .set nomacro
# BB#0:
    addiu $sp, $sp, -32
$tmp1:
```

```

.cfi_def_cfa_offset 32
ld    $2, 32($sp)
st    $2, 28($sp)
ld    $2, 36($sp)
st    $2, 24($sp)
ld    $2, 40($sp)
st    $2, 20($sp)
ld    $2, 44($sp)
st    $2, 16($sp)
ld    $2, 48($sp)
st    $2, 12($sp)
ld    $2, 52($sp)
st    $2, 8($sp)
addiu $3, $zero, %got_hi(gI)
shl   $3, $3, 16
addu  $3, $3, $gp
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
.set  macro
.set  reorder
.end  _Z5sum_iuiuii
$tmp2:
.size  _Z5sum_iuiuii, ($tmp2)-_Z5sum_iuiuii
.cfi_endproc

.globl main
.align 2
.type  main,@function
.ent   main          # @main
main:
.cfi_startproc
.frame $sp,40,$lr
.mask  0x00004000,-4
.set   noreorder
.cupload $t9
.set   nomacro
# BB#0:
addiu $sp, $sp, -40
$tmp5:
.cfi_def_cfa_offset 40
st    $lr, 36($sp)      # 4-byte Folded Spill
$tmp6:
.cfi_offset 14, -4
addiu $2, $zero, 0

```

```

st      $2, 32($sp)
!ADJCALLSTACKDOWN 24
addiu  $2, $zero, 6
st      $2, 60($sp)
addiu  $2, $zero, 5
st      $2, 56($sp)
addiu  $2, $zero, 4
st      $2, 52($sp)
addiu  $2, $zero, 3
st      $2, 48($sp)
addiu  $2, $zero, 2
st      $2, 44($sp)
addiu  $2, $zero, 1
st      $2, 40($sp)
ld      $t9, %call16(_Z5sum_iiiiiii)($gp)
jalr   $t9
!ADJCALLSTACKUP 24
st      $2, 28($sp)
ld      $lr, 36($sp)          # 4-byte Folded Reload
addiu $sp, $sp, 40
ret    $lr
.set   macro
.set   reorder
.end   main

$tmp7:
.size  main, ($tmp7)-main
.cfi_endproc

.type  gI,@object          # @gI
.data
.globl gI
.align 2

gI:
.4byte 100                 # 0x64
.size  gI, 4

```

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/InputFiles/ch9\_1\_2.cpp

The “call memcpy” for short string is optimized by llvm before “DAG->DAG Pattern Instruction Selection” stage and translate it into “store with contant” as follows,

```
Jonathan@tekkiMac:~/InputFiles$ /Users/Jonathan/llvm/test/cmake_debug_build/bin/Debug/llc -march=cpu0 -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="Hello"

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 “isTailCall = false;” set in LowerCall() of Cpu0ISelLowering.cpp meaning Cpu0 don’t support tail call optimization at this moment. About tail call optimization please reference <sup>6</sup>.

## 9.4 Fix issues

Run Chapter9\_2/ with ch7\_5.cpp to get the incorrect main return (return register \$2 is not 0) as follows,

### Index/InputFiles/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); // 12+2012=2024
}

118-165-78-31:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch7_5.cpp -emit-llvm -o ch7_5.bc
118-165-78-31:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=asm ch7_5.bc -o
ch7_5.cpu0.static.s
118-165-78-31:InputFiles Jonathan$ cat ch7_5.cpu0.static.s

```

---

<sup>6</sup> [http://en.wikipedia.org/wiki/Tail\\_call](http://en.wikipedia.org/wiki/Tail_call)

```

.section .mdebug.abi32
.previous
.file "ch7_5.bc"
...
.cfi_startproc
.frame $sp,16,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -16
$tmp1:
.cfi_def_cfa_offset 16
addiu $2, $zero, 0
st $2, 12($sp)
addiu $2, $zero, %hi(date)
shl $2, $2, 16
addiu $2, $2, %lo(date)
ld $2, 8($2)
st $2, 8($sp)
addiu $2, $zero, %hi(a)
shl $2, $2, 16
addiu $2, $2, %lo(a)
ld $2, 4($2)
st $2, 4($sp)
addiu $sp, $sp, 16
ret $lr
.set macro
...

```

Summary the issues for the code generated as above and in last section as follows:

1. It store the arguments to wrong offset.
2. !ADJCALLSTACKUP and !ADJCALLSTACKDOWN.
3. The \$gp is caller saved register. The caller main() didn't save \$gp will has bug if the callee sum\_i() has changed \$gp. Programmer can change \$gp with assembly code in sum\_i().
4. Return value of main().

Solve these issues in each sub-section.

### 9.4.1 Fix the wrong offset in storing arguments to stack frame

To fix the wrong offset in storing arguments, we modify the following code in eliminateFrameIndex() as follows. The code as below is modified in Chapter9\_3/ to set the caller outgoing arguments into spOffset(\$sp) (Chapter9\_2/ set them to pOffset+stackSize(\$sp)).

#### Ibdex/Chapter9\_3/Cpu0RegisterInfo.cpp

```

void Cpu0RegisterInfo::
eliminateFrameIndex(MachineBasicBlock::iterator II, int SPAdj,
                     RegScavenger *RS) const {
...
Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
...

```

```

if (Cpu0FI->isOutArgFI(FrameIndex) || Cpu0FI->isDynAllocFI(FrameIndex) ||
    (FrameIndex >= MinCSFI && FrameIndex <= MaxCSFI))
    FrameReg = Cpu0:::SP;
else
    FrameReg = getFrameRegister(MF);
...
// 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.
if (Cpu0FI->isOutArgFI(FrameIndex) || Cpu0FI->isGPFI(FrameIndex) ||
    Cpu0FI->isDynAllocFI(FrameIndex))
    Offset = spOffset;
else
    Offset = spOffset + (int64_t)stackSize;
Offset += MI.getOperand(i+1).getImm();
...
}

```

### Ibdex/Chapter9\_3/Cpu0MachineFunction.h

```

/// 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;
...
Cpu0FunctionInfo(MachineFunction& MF)
: ...
    SRetReturnReg(0)
...
bool isOutArgFI(int FI) const {
    return FI <= OutArgFIRange.first && FI >= OutArgFIRange.second;
}
...
unsigned getSRetReturnReg() const { return SRetReturnReg; }
void setSRetReturnReg(unsigned Reg) { SRetReturnReg = Reg; }
...

```

Run Chapter9\_3/ with ch9\_1.cpp will get the following result. It correct arguments offset im main() from (0+40)\$sp, (8+40)\$sp, ..., to (0)\$sp, (8)\$sp, ..., where the stack size is 40 in main().

```

118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_1.bc -o
ch9_1.cpu0.s
118-165-78-230:InputFiles Jonathan$ cat ch9_1.cpu0.s
...
addiu $2, $zero, 6
st $2, 20($sp)           // Correct offset
addiu $2, $zero, 5
st $2, 16($sp)
addiu $2, $zero, 4
st $2, 12($sp)
addiu $2, $zero, 3
st $2, 8($sp)

```

```

addiu $2, $zero, 2
st $2, 4($sp)
addiu $2, $zero, 1
st $2, 0($sp)
ld $t9, %call16(_Z5sum_iiiiiii) ($gp)
jalr $t9
...

```

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
Arguments location	spOffset + stackSize	spOffset

#### 9.4.2 Pseudo hook instruction ADJCALLSTACKDOWN and ADJCALLSTACKUP

To fix the !ADJSTACKDOWN and !ADJSTACKUP, we call Cpu0GenInstrInfo(Cpu0:: ADJCALLSTACKDOWN, Cpu0::ADJCALLSTACKUP) in Cpu0InstrInfo() constructor function and define eliminateCallFramePseudoInstr() as follows,

##### Ibdex/Chapter9\_3/Cpu0InstrInfo.cpp

```

Cpu0InstrInfo::Cpu0InstrInfo(Cpu0TargetMachine &tm)
:
  Cpu0GenInstrInfo(Cpu0::ADJCALLSTACKDOWN, Cpu0::ADJCALLSTACKUP),
...

```

##### Ibdex/Chapter9\_3/Cpu0FrameLowering.h

```

void eliminateCallFramePseudoInstr(MachineFunction &MF,
                                    MachineBasicBlock &MBB,
                                    MachineBasicBlock::iterator I) const;

```

##### Ibdex/Chapter9\_3/Cpu0FrameLowering.cpp

```

...
// Cpu0
// This function eliminate ADJCALLSTACKDOWN,
// ADJCALLSTACKUP pseudo instructions
void Cpu0FrameLowering::
eliminateCallFramePseudoInstr(MachineFunction &MF, MachineBasicBlock &MBB,
                             MachineBasicBlock::iterator I) const {
  // Simply discard ADJCALLSTACKDOWN, ADJCALLSTACKUP instructions.
  MBB.erase(I);
}

```

With above definition, `eliminateCallFramePseudoInstr()` will be called when llvm meet pseudo instructions `ADJCALLSTACKDOWN` and `ADJCALLSTACKUP`. We just discard these 2 pseudo instructions. Run `Chapter9_3/` with `ch9_1.cpp` will these two Pseudo hook instructions.

### 9.4.3 Handle \$gp register in PIC addressing mode

In “section Global variable”<sup>5</sup>, 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. But, we can caculate 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_2.cpp` to get the following correct result. We putting the comments in the result for explanation.

```
118-165-78-230:InputFiles Jonathan$ cat ch9_1.cpu0.s
_Z5sum_iuiuii:
...
    .cupload $t9 // assign $gp = $t9 by loader when loader load re-entry
                  // function (shared library) of _Z5sum_iuiuii
    .set    nomacro
# BB#0:
    addiu  $sp, $sp, -32
$tmp1:
    .cfi_def_cfa_offset 32
...
    ld    $3, %got(gI)($gp) // %got(gI) is offset of (gI - _Z5sum_iuiuii)
...
    ret  $lr
    .set    macro
    .set    reorder
    .end   _Z5sum_iuiuii
...
    .ent   main           # @main
main:
    .cfi_startproc
...
    .cupload $t9
    .set    nomacro
...
    .cprestore 24 // save $gp to 24($sp)
    addiu  $2, $zero, 0
...
    ld    $t9, %call16(_Z5sum_iuiuii)($gp)
    jalr  $t9           // $t9 register is the alias of $6
    ld    $gp, 24($sp) // restore $gp from 24($sp)
...
    .end   main
$tmp7:
    .size   main, ($tmp7)-main
    .cfi_endproc
    .type   gI,@object          # @gI
    .data
    .globl   gI
    .align  2
```

```
gI:
    .4byte 100          # 0x64
    .size   gI, 4
```

As above code comment, “**.cprestore 24**” is a pseudo instruction for saving **\$gp** to **24(\$sp)** while Instruction “**ld \$gp, 24(\$sp)**” will restore the **\$gp**. In other word, **\$gp** is a caller saved register, so **main()** need to save/restore **\$gp** before/after call the shared library **\_Z5sum\_iiiiii()** function. In **\_Z5sum\_iiiiii()** function, we translate global variable **gI** address by “**ld \$3, %got(gI(\$gp))**” where **%got(gI)** is the offset value of **(gI - \_Z5sum\_iiiiii)** which can be caculated at link time.

According the original cpu0 web site information, it only support “**jsub**” 24 bits address range access. We add “**jalr**” to cpu0 and expand it to 32 bit address. We did this change for two reason. One is cpu0 can be expand to 32 bit address space by only add 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 demonstrate:

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.

Now, after the following code added in Chapter9\_3/, we can issue “**.cprestore**” in **emitPrologue()** and emit “**ld \$gp, (\$gp save slot on stack)**” after **jalr** by create file **Cpu0EmitGPRestore.cpp** which run as a function pass.

### Ibdex/Chapter9\_3/CMakeLists.txt

```
add_llvm_target(Cpu0CodeGen
...
Cpu0EmitGPRestore.cpp
...
```

### Ibdex/Chapter9\_3/Cpu0TargetMachine.cpp

```
Cpu0elTargetMachine::
Cpu0elTargetMachine(const Target &T, StringRef 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) {}
namespace {
...
virtual bool addPreRegAlloc();
...
}

bool Cpu0PassConfig::addPreRegAlloc() {
    // Do not restore $gp if target is Cpu064.
    // In N32/64, $gp is a callee-saved register.

    addPass(createCpu0EmitGPRestorePass(getCpu0TargetMachine()));
    return true;
}
```

### Ibdex/Chapter9\_3/Cpu0.h

```
FunctionPass *createCpu0EmitGPRestorePass(Cpu0TargetMachine &TM);
```

### Ibdex/Chapter9\_3/Cpu0FrameLowering.cpp

```
void Cpu0FrameLowering::emitPrologue(MachineFunction &MF) const {
    ...
    unsigned RegSize = 4;
    unsigned LocalVarAreaOffset =
        Cpu0FI->needGPSaveRestore() ?
        (MFI->getObjectOffset(Cpu0FI->getGPFI()) + RegSize) :
        Cpu0FI->getMaxCallFrameSize();
    ...
    // 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);
    }
}
```

### Ibdex/Chapter9\_3/Cpu0InstrInfo.td

```
let neverHasSideEffects = 1 in
def CPRESTORE : Cpu0Pseudo<(outs), (ins i32imm:$loc, CPUREgs:$gp),
    ".cprestore\t$loc", []>;
```

### Ibdex/Chapter9\_3/Cpu0ISelLowering.cpp

```
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                               SmallVectorImpl<SDValue> &InVals) const {
    ...
    // If this is the first call, create a stack frame object that points to
    // a location to which .cprestore saves $gp.
    if (IsPIC && Cpu0FI->globalBaseRegFixed() && !Cpu0FI->getGPFI())
    ...
    if (MaxCallFrameSize < NextStackOffset) {
        ...
        if (Cpu0FI->needGPSaveRestore())
            MFI->setObjectOffset(Cpu0FI->getGPFI(), NextStackOffset);
        }
        ...
    }
    ...
}
```

[lbdex/Chapter9\\_3/Cpu0.EmitGPRestore.cpp](#)

[lbdex/Chapter9\\_3/Cpu0.AsmPrinter.cpp](#)

```
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);
    if (Cpu0FI->getEmitNOAT())
        OutStreamer.EmitRawText(StringRef("\t.set\tnoat"));
    OutStreamer.EmitRawText(StringRef("\t.set\tnomacro"));
}

...
void Cpu0AsmPrinter::EmitInstruction(const MachineInstr *MI) {
    ...
    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);

            return;
        }
        break;
    }
    default:
        break;
    }
    ...
}
```

[Index/Chapter9\\_3/Cpu0MCInstLower.cpp](#)

[Index/Chapter9\\_3/Cpu0MCInstLower.h](#)

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 take care the .cprestore machine instructions.

```
118-165-76-131:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=
obj ch9_1.bc -o ch9_1.cpu0.o
118-165-76-131:InputFiles Jonathan$ hexdump ch9_2.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:InputFiles 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:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=
asm ch9_1.bc -o ch9_1.cpu0.s
118-165-76-131:InputFiles 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
```

#### 9.4.4 Correct the return of main()

The `LowerReturn()` modified in `Chapter9_3/` as follows,

[Index/Chapter9\\_3/Cpu0ISelLowering.cpp](#)

The `LowerReturn` work with `RET` defined in `Chapter3_4` as follows,

**lbdex/Chapter3\_4/Cpu0InstrInfo.h**

**lbdex/Chapter3\_4/Cpu0InstrInfo.cpp**

**lbdex/Chapter3\_4/Cpu0InstrInfo.td**

```
// Return
def Cpu0Ret : SDNode<"Cpu0ISD::Ret", SDTNone,
               [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;
...
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<0x2C, "ret", RC> {
    let isReturn = 1;
    let isCodeGenOnly = 1;
    let hasCtrlDep = 1;
    let hasExtraSrcRegAllocReq = 1;
}
...
let isReturn=1, isTerminator=1, hasDelaySlot=1, isCodeGenOnly=1,
    isBarrier=1, hasCtrlDep=1, addr=0 in
def RetLR : Cpu0Pseudo<(outs), (ins), "", [(Cpu0Ret)]>;
def RET      : RetBase<CPUREgs>;
```

Above code do the following:

1. Declare a pseudo node by the following code,

**lbdex/Chapter3\_4/Cpu0InstrInfo.td**

```
// Return
def Cpu0Ret : SDNode<"Cpu0ISD::Ret", SDTNone,
               [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;
...
let isReturn=1, isTerminator=1, hasDelaySlot=1, isCodeGenOnly=1,
    isBarrier=1, hasCtrlDep=1, addr=0 in
def RetLR : Cpu0Pseudo<(outs), (ins), "", [(Cpu0Ret)]>;
```

2. Create Cpu0ISD::Ret node in LowerReturn() which is called when meet function return as above code in Chapter9\_3/Cpu0ISelLowering.cpp. More specific, it create 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 in any later optimization step. 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());" the V0 register won't be live out, the previous DAG (CopyToReg %X, %V0, %Y) will be removed in later optimization stage. Then the result is same with Chapter9\_2 which the return value is error.

```

Initial selection DAG: BB#0 'main:entry'
SelectionDAG has 21 nodes:
...
0x1e1e50: i32 = Register %V0

0x1e9fd20: <multiple use>
0x1e9fd20: <multiple use>
0x1e1c50: i32 = FrameIndex<2> [ORD=7]

0x1e9f120: <multiple use>
0x1e1d50: ch = store 0x1e9fd20:1, 0x1e9fd20, 0x1e1c50,
0x1e9f120<ST4[%i]> [ORD=7]

0x1e1e50: <multiple use>
0x1e9ef20: <multiple use>
0x1e1f50: ch,glue = CopyToReg 0x1e1d50, 0x1e1e50, 0x1e9ef20

0x1e1f50: <multiple use>
0x1e1e50: <multiple use>
0x1e1f50: <multiple use>
0x1ea2050: ch = Cpu0ISD::Ret 0x1e1f50, 0x1e1e50, 0x1e1f50:1

```

3. After instruction selection, the Cpu0::Ret is replaced by Cpu0::RetLR as below. This effect came 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 Cpu0::RetLR into instruction **ret \$lr** in “Post-RA pseudo instruction expansion pass” stage by the code in Chapter9\_3/Cpu0InstrInfo.cpp as above. This stage is 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/Chapter3\_4/Cpu0InstrInfo.td

```

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<CPUREgs>;

```

List the stages mentioned in Chapter 3 and sub-stages in Chapter 4 again as below. Step 2 as above is before “CPU0 DAG->DAG Pattern Instruction Selection” stage, step 3 is in “Instruction selection” stage, step 4 is in “Expand ISel Pseudo-instructions” stage and step 5 is “Cpu0 Assembly Printer” stage.

```

118-165-79-200:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=asm ch7_5.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

```

Summary to Table: Correct the return value in each stage.

Table 9.2: Correct the return value in each stage

Stage	Function
Write Code	Declare a pseudo node 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 -> ret \$lr
Cpu0 Assembly Printer	Print according “def RET”

Run Chapter9\_3/ to get the correct result (return register \$2 is 0) as follows,

```

118-165-78-31:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=asm ch7_5.bc -o

```

```
ch7_5.cpu0.static.s
118-165-78-31:InputFiles Jonathan$ cat ch7_5.cpu0.static.s
.section .mdebug.abi32
.previous
.file "ch7_5.bc"
.text
.globl main
.align 2
.type main,@function
.ent main                                # @main
main:
.cfi_startproc
.frame $sp,16,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
    addiu $sp, $sp, -16
$tmp1:
.cfi_def_cfa_offset 16
    addiu $2, $zero, 0
    st $2, 12($sp)
    addiu $3, $zero, %hi(date)
    shl $3, $3, 16
    addiu $3, $3, %lo(date)
    ld $3, 8($3)
    st $3, 8($sp)
    addiu $3, $zero, %hi(a)
    shl $3, $3, 16
    addiu $3, $3, %lo(a)
    ld $3, 4($3)
    st $3, 4($sp)
    addiu $sp, $sp, 16
    ret $lr
.set macro
.set reorder
.end main
$tmp2:
.size main, ($tmp2)-main
.cfi_endproc

.type date,@object                      # @date
.data
.globl date
.align 2
date:
    .4byte 2012                         # 0x7dc
    .4byte 10                           # 0xa
    .4byte 12                           # 0xc
.size date, 12

.type a,@object                         # @a
.globl a
.align 2
a:
    .4byte 2012                         # 0x7dc
    .4byte 10                           # 0xa
    .4byte 12                           # 0xc
```

```
.size a, 12
```

## 9.5 Support features

This section support features of struct type, variable number of arguments and dynamic stack allocation.

Run Chapter9\_3 with ch9\_2\_1.cpp will get the error message as follows,

**lbdex/InputFiles/ch9\_2\_1.cpp**

```
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 main()
```

```

{
    Time time1 = {1, 10, 12};
    Date date1 = getDate();
    Date date2 = copyDate(date1);
    Date date3 = copyDate(&date1);
    Time time2 = copyTime(time1);
    Time time3 = copyTime(&time1);

    return 0;
}

JonathantekiiMac:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_2_1.cpp -emit-llvm -o ch9_2_1.bc
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch9_2_1.bc -o ch9_2_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,
line 6712.
...

```

Run Chapter9\_3/ with ch9\_3.cpp to get the following error,

### Ibdex/InputFiles/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 main()
{
    int a = sum_i(6, 0, 1, 2, 3, 4, 5);

    return a;
}

```

```

118-165-78-230:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3.cpp -emit-llvm -o ch9_3.bc
118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/

```

```
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_3.bc -o
ch9_3.cpu0.s
LLVM ERROR: Cannot select: 0x7f8b6902fd10: ch = vastart 0x7f8b6902fa10,
0x7f8b6902fb10, 0x7f8b6902fc10 [ORD=9] [ID=22]
0x7f8b6902fb10: i32 = FrameIndex<5> [ORD=7] [ID=9]
In function: _Z5sum_iiz
```

### lbdex/InputFiles/ch9\_4.cpp

```
#include <alloca.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:InputFiles 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:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_4.bc -o
ch9_4.cpu0.s
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
```

### 9.5.1 Structure type support

Chapter9\_4/ with the following code added to support the structure type in function call.

#### lbdex/Chapter9\_4/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)
{
    assert(RC->contains(PReg) && "Not the correct regclass!");
    unsigned VReg = MF.getRegInfo().createVirtualRegister(RC);
    MF.getRegInfo().addLiveIn(PReg, VReg);
    return VReg;
}
...
//===== Call Calling Convention Implementation =====
//=====

static const unsigned IntRegsSize = 2;

static const uint16_t IntRegs[] = {
    Cpu0::A0, Cpu0::A1
};

// Write ByVal Arg to arg registers and stack.
static void
WriteByValArg(SDValue& ByValChain, SDValue Chain, DebugLoc dl,
    SmallVector<std::pair<unsigned, SDValue>, 16>& RegsToPass,
    SmallVector<SDValue, 8>& MemOpChains, int& LastFI,
    MachineFrameInfo *MFI, SelectionDAG &DAG, SDValue Arg,
    const CCValAssign &VA, const ISD::ArgFlagsTy& Flags,
    MVT PtrType, bool isLittle) {
    unsigned LocMemOffset = VA.getLocMemOffset();
    unsigned Offset = 0;
    uint32_t RemainingSize = Flags.getByValSize();
    unsigned ByValAlign = Flags.getByValAlign();

    if (RemainingSize == 0)
        return;

    // Create a fixed object on stack at offset LocMemOffset and copy
    // remaining part of byval arg to it using memcpy.
    SDValue Src = DAG.getNode(ISD::ADD, dl, MVT::i32, Arg,
        DAG.getConstant(Offset, MVT::i32));
    LastFI = MFI->CreateFixedObject(RemainingSize, LocMemOffset, true);
    SDValue Dst = DAG.getFrameIndex(LastFI, PtrType);
    ByValChain = DAG.getMemcpy(ByValChain, dl, Dst, Src,
        DAG.getConstant(RemainingSize, MVT::i32),
        std::min(BindViewAlign, (unsigned)4),
        /*isVolatile=*/false, /*AlwaysInline=*/false,
        MachinePointerInfo(0), MachinePointerInfo(0));
}
...

```

```

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) {
        ...
        // ByVal Arg.
        if (Flags.isByVal()) {
            ...
            WriteByValArg( ByValChain, Chain, dl, RegsToPass, MemOpChains, LastFI,
                           MFI, DAG, Arg, VA, Flags, getPointerTy(),
                           Subtarget->isLittle());
            ...
        }
        ...
    }
    ...
}

//=====
//      Formal Arguments Calling Convention Implementation
//=====

static void ReadByValArg(MachineFunction &MF, SDValue Chain, DebugLoc dl,
                           std::vector<SDValue> &OutChains,
                           SelectionDAG &DAG, unsigned NumWords, SDValue FIN,
                           const CCValAssign &VA, const ISD::ArgFlagsTy &Flags,
                           const Argument *FuncArg) {
    unsigned LocMem = VA.getLocMemOffset();
    unsigned FirstWord = LocMem / 4;

    // copy register A0 - A1 to frame object
    for (unsigned i = 0; i < NumWords; ++i) {
        unsigned CurWord = FirstWord + i;
        if (CurWord >= IntRegsSize)
            break;

        unsigned SrcReg = IntRegs[CurWord];
        unsigned Reg = AddLiveIn(MF, SrcReg, &Cpu0::CPURegsRegClass);
        SDValue StorePtr = DAG.getNode(ISD::ADD, dl, MVT::i32, FIN,
                                       DAG.getConstant(i * 4, MVT::i32));
        SDValue Store = DAG.getStore(Chain, dl, DAG.getRegister(Reg, MVT::i32),
                                     StorePtr, MachinePointerInfo(FuncArg, i * 4),
                                     false, false, 0);
        OutChains.push_back(Store);
    }
} // lbd document - mark - ReadByValArg
...
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool isVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         DebugLoc dl, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                         const {
    ...
    for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i, ++FuncArg) {

```

```

...
if (Flags.isByVal()) {
    assert(Flags.getByValSize() &&
        "ByVal args of size 0 should have been ignored by front-end.");
    unsigned NumWords = (Flags.getByValSize() + 3) / 4;
    LastFI = MFI->CreateFixedObject(NumWords * 4, VA.getLocMemOffset(),
        true);
    SDValue FIN = DAG.getFrameIndex(LastFI, getPointerTy());
    InVals.push_back(FIN);
    ReadByValArg(MF, Chain, dl, OutChains, DAG, NumWords, FIN, VA, Flags,
        &*FuncArg);
    continue;
}
...
}
// 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 (DAG.getMachineFunction().getFunction()->hasStructRetAttr()) {
    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[0]);
    Chain = DAG.getNode(ISD::TokenFactor, dl, MVT::Other, Copy, Chain);
}
...
}
...
SDValue
Cpu0TargetLowering::LowerReturn(SDValue Chain,
    CallingConv::ID CallConv, bool isVarArg,
    const SmallVectorImpl<ISD::OutputArg> &Outs,
    const SmallVectorImpl<SDValue> &OutVals,
    DebugLoc 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 (DAG.getMachineFunction().getFunction()->hasStructRetAttr()) {
    MachineFunction &MF = DAG.getMachineFunction();
    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());

    Chain = DAG.getCopyToReg(Chain, dl, Cpu0::V0, Val, Flag);
    Flag = Chain.getValue(1);
    RetOps.push_back(DAG.getRegister(Cpu0::V0, getPointerTy()));
}
...
}

```

In addition to above code, we have defined the calling convention at early of this chapter as follows,

### lbdex/Chapter9\_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\_4/ with ch9\_2\_1.cpp and explain with this example.

```
JonathantekiiMac:InputFiles Jonathan$ cat ch9_2_1.cpu0.s
.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:
ld $2, 0($sp)           // $2 is 192($sp)
ld $3, %got(gDate)($gp) // $3 is &gDate
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
.set macro
.set reorder
.end _Z7getDatev
$tmp0:
.size _Z7getDatev, ($tmp0)-_Z7getDatev
.cfi_endproc

.globl _Z8copyDate4Date
.align 2
.type _Z8copyDate4Date,@function
.ent _Z8copyDate4Date           # @_Z8copyDate4Date
_Z8copyDate4Date:
.cfi_startproc
.frame $sp,0,$lr
.mask 0x00000000,0
```

```
.set noreorder
.set nomacro
# BB#0:
st $5, 4($sp)
ld $2, 0($sp)           // $2 = 168($sp)
ld $3, 24($sp)
st $3, 20($2)           // copy date1, 24..4($sp), to date2,
ld $3, 20($sp)           // 188..168($sp)
st $3, 16($2)
ld $3, 16($sp)
st $3, 12($2)
ld $3, 12($sp)
st $3, 8($2)
ld $3, 8($sp)
st $3, 4($2)
ld $3, 4($sp)
st $3, 0($2)
ret $lr
.set macro
.set reorder
.end _Z8copyDate4Date
$tmp1:
.size _Z8copyDate4Date, ($tmp1)-_Z8copyDate4Date
.cfi_endproc

.globl _Z8copyDateP4Date
.align 2
.type _Z8copyDateP4Date, @function
.ent _Z8copyDateP4Date      # @_Z8copyDateP4Date
_Z8copyDateP4Date:
.cfi_startproc
.frame $sp,8,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -8
$tmp3:
.cfi_def_cfa_offset 8
ld $2, 8($sp)           // $2 = 120($sp of main) date2
ld $3, 12($sp)           // $3 = 192($sp of main) date1
st $3, 0($sp)
ld $4, 20($3)           // copy date1, 212..192($sp of main),
st $4, 20($2)           // to date2, 140..120($sp of main)
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)
addiu $sp, $sp, 8
ret $lr
.set macro
.set reorder
```

```

.end _Z8copyDateP4Date
$tmp4:
.size _Z8copyDateP4Date, ($tmp4)-_Z8copyDateP4Date
.cfi_endproc

.globl _Z8copyTime4Time
.align 2
.type _Z8copyTime4Time, @function
.ent _Z8copyTime4Time      # @_Z8copyTime4Time
_Z8copyTime4Time:
.cfi_startproc
.frame $sp,64,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -64
$tmp6:
.cfi_def_cfa_offset 64
ld $2, 68($sp)      // save 8..0 ($sp of main) to 24..16($sp)
st $2, 20($sp)
ld $2, 64($sp)
st $2, 16($sp)
ld $2, 72($sp)
st $2, 24($sp)
st $2, 40($sp)      // save 8($sp of main) to 40($sp)
ld $2, 20($sp)      // timel.minute, save timel.minute and
st $2, 36($sp)      // timel.second to 36..32($sp)
ld $2, 16($sp)      // timel.second
st $2, 32($sp)
ld $2, 40($sp)      // $2 = 8($sp of main) = timel.hour
st $2, 56($sp)      // copy timel to 56..48($sp)
ld $2, 36($sp)
st $2, 52($sp)
ld $2, 32($sp)
st $2, 48($sp)
ld $2, 48($sp)      // copy timel to 8..0($sp)
ld $3, 52($sp)
ld $4, 56($sp)
st $4, 8($sp)
st $3, 4($sp)
st $2, 0($sp)
ld $2, 0($sp)        // put timel to $2, $3 and $4 ($v0, $v1 and $a0)
ld $3, 4($sp)
ld $4, 8($sp)
addiu $sp, $sp, 64
ret $lr
.set macro
.set reorder
.end _Z8copyTime4Time
$tmp7:
.size _Z8copyTime4Time, ($tmp7)-_Z8copyTime4Time
.cfi_endproc

.globl _Z8copyTimeP4Time
.align 2
.type _Z8copyTimeP4Time, @function
.ent _Z8copyTimeP4Time      # @_Z8copyTimeP4Time

```

```
_Z8copyTimeP4Time:
.cfi_startproc
.frame $sp,40,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -40
$tmp9:
.cfi_def_cfa_offset 40
ld $2, 40($sp)           // 216($sp of main)
st $2, 16($sp)
ld $3, 8($2)             // copy time1, 224..216($sp of main) to
st $3, 32($sp)           // 32..24($sp), 8..0($sp) and $2, $3, $4
ld $3, 4($2)
st $3, 28($sp)
ld $2, 0($2)
st $2, 24($sp)
ld $2, 24($sp)
ld $3, 28($sp)
ld $4, 32($sp)
st $4, 8($sp)
st $3, 4($sp)
st $2, 0($sp)
ld $2, 0($sp)
ld $3, 4($sp)
ld $4, 8($sp)
addiu $sp, $sp, 40
ret $lr
.set macro
.set reorder
.end _Z8copyTimeP4Time
$tmp10:
.size _Z8copyTimeP4Time, ($tmp10)-_Z8copyTimeP4Time
.cfi_endproc

.globl main
.align 2
.type main,@function
.ent main           # @main
main:
.cfi_startproc
.frame $sp,248,$lr
.mask 0x00004180,-4
.set noreorder
.cupload $t9
.set nomacro
# BB#0:
addiu $sp, $sp, -248
$tmp13:
.cfi_def_cfa_offset 248
st $lr, 244($sp)      # 4-byte Folded Spill
st $8, 240($sp)       # 4-byte Folded Spill
st $7, 236($sp)       # 4-byte Folded Spill
$tmp14:
.cfi_offset 14, -4
$tmp15:
.cfi_offset 8, -8
```

```
$tmp16:
.cfi_offset 7, -12
.cprestore 16
addiu $7, $zero, 0
st $7, 232($sp)
ld $2, %got($_Z4mainE5time1)($gp)
addiu $2, $2, %lo($_Z4mainE5time1)
ld $3, 8($2)      // save initial value to time1, 224..216($sp)
st $3, 224($sp)
ld $3, 4($2)
st $3, 220($sp)
ld $2, 0($2)
st $2, 216($sp)
addiu $8, $sp, 192
st $8, 0($sp)      // *0($sp) = 192($sp)
ld $t9, %call16(_Z7getDatev)($gp) // copy gDate contents to date1, 212..192($sp)
jalr $t9
ld $gp, 16($sp)
ld $2, 212($sp)    // copy 212..192($sp) to 164..144($sp)
st $2, 164($sp)
ld $2, 208($sp)
st $2, 160($sp)
ld $2, 204($sp)
st $2, 156($sp)
ld $2, 200($sp)
st $2, 152($sp)
ld $2, 196($sp)
st $2, 148($sp)
ld $2, 192($sp)
st $2, 144($sp)
ld $2, 164($sp)    // copy 164..144($sp) to 24..4($sp)
st $2, 24($sp)
ld $2, 160($sp)
st $2, 20($sp)
ld $2, 156($sp)
st $2, 16($sp)
ld $2, 152($sp)
st $2, 12($sp)
ld $2, 148($sp)
st $2, 8($sp)
ld $2, 144($sp)
st $2, 4($sp)
addiu $2, $sp, 168
st $2, 0($sp)      // *0($sp) = 168($sp)
ld $t9, %call16(_Z8copyDate4Date)($gp)
jalr $t9
ld $gp, 16($sp)
st $8, 4($sp)      // 4($sp) = 192($sp) date1
addiu $2, $sp, 120
st $2, 0($sp)      // *0($sp) = 120($sp) date2
ld $t9, %call16(_Z8copyDateP4Date)($gp)
jalr $t9
ld $gp, 16($sp)
ld $2, 224($sp)    // save time1 to arguments passing location,
st $2, 96($sp)      // 8..0($sp)
ld $2, 220($sp)
st $2, 92($sp)
ld $2, 216($sp)
```

```
st  $2, 88($sp)
ld  $2, 88($sp)
ld  $3, 92($sp)
ld  $4, 96($sp)
st  $4, 8($sp)
st  $3, 4($sp)
st  $2, 0($sp)
ld  $t9, %call16(_Z8copyTime4Time) ($gp)
jalr $t9
ld  $gp, 16($sp)
st  $3, 76($sp)      // save return value time2 from $2, $3, $4 to
st  $2, 72($sp)      // 80..72($sp) and 112..104($sp)
st  $4, 80($sp)
ld  $2, 72($sp)
ld  $3, 76($sp)
ld  $4, 80($sp)
st  $4, 112($sp)
st  $3, 108($sp)
st  $2, 104($sp)
addiu $2, $sp, 216
st  $2, 0($sp)      // *(0($sp)) = 216($sp)
ld  $t9, %call16(_Z8copyTimeP4Time) ($gp)
jalr $t9
ld  $gp, 16($sp)
st  $3, 44($sp)      // save return value time3 from $2, $3, $4 to
st  $2, 40($sp)      // 48..44($sp) 64..56($sp)
st  $4, 48($sp)
ld  $2, 40($sp)
ld  $3, 44($sp)
ld  $4, 48($sp)
st  $4, 64($sp)
st  $3, 60($sp)
st  $2, 56($sp)
add $2, $zero, $7    // return 0 by $2, ($7 is 0)

ld  $7, 236($sp)      # 4-byte Folded Reload // restore callee saved
ld  $8, 240($sp)      # 4-byte Folded Reload // registers $s0, $s1
ld  $lr, 244($sp)      # 4-byte Folded Reload // ($7, $8)
addiu $sp, $sp, 248
ret $lr
.set  macro
.set  reorder
.end  main
$tmp17:
.size main, ($tmp17)-main
.cfi_endproc

.type gDate,@object          # @gDate
.data
.globl gDate
.align 2
gDate:
.4byte 2012                # 0x7dc
.4byte 10                  # 0xa
.4byte 12                  # 0xc
.4byte 1                   # 0x1
.4byte 2                   # 0x2
.4byte 3                   # 0x3
```

```

.size gDate, 24

.type gTime, @object          # @_gTime
.globl gTime
.align 2
gTime:
    .4byte 2                  # 0x2
    .4byte 20                 # 0x14
    .4byte 30                 # 0x1e
.size gTime, 12

.type _$ZZ4mainE5time1, @object # @_ZZ4mainE5time1
.section .rodata, "a", @progbits
.align 2
$ZZ4mainE5time1:
    .4byte 1                  # 0x1
    .4byte 10                 # 0xa
    .4byte 12                 # 0xc
.size _$ZZ4mainE5time1, 12

```

In LowerCall(), Flags.isByVal() will be true if the outgoing arguments over 4 registers size, then it will call WriteByValArg(..., getPointerTy(), ...) to save those arguments to stack as offset. For example code of ch9\_2\_1.cpp, Flags.isByVal() is true for copyDate(date1) outgoing arguments, since the date1 is type of Date which contains 6 integers (year, month, day, hour, minute, second). But Flags.isByVal() is false for copyTime(time1) since type Time is a struct contains 3 integers (hour, minute, second). So, if you mark WriteByValArg(..., getPointerTy(), ...), the result will missing the following code in caller, main(),

```

ld $2, 164($sp)      // copy 164..144($sp) to 24..4($sp)
st $2, 24($sp)
ld $2, 160($sp)
st $2, 20($sp)
ld $2, 156($sp)
st $2, 16($sp)
ld $2, 152($sp)
st $2, 12($sp)
ld $2, 148($sp)
st $2, 8($sp)
ld $2, 144($sp)
st $2, 4($sp)          // will missing the above code

addiu $2, $sp, 168
st $2, 0($sp)          // *0($sp) = 168($sp)
ld $t9, %call16(_Z8copyDate4Date) ($gp)

```

In LowerFormalArguments(), the “if (Flags.isByVal())” getting the incoming arguments which corresponding the outgoing arguments of LowerCall().

LowerFormalArguments() is called when a function is entered while LowerReturn() is called when a function is left, reference <sup>7</sup>. The former save the return register to virtual register while the later load the virtual register back to return register. Since the return value is “struct type” and over 4 registers size, it save pointer (struct address) to return register. List the code and their effect as follows,

---

<sup>7</sup> <http://developer.mips.com/clang-llvm/>

### Ibdex/Chapter9\_4/Cpu0ISelLowering.cpp

```

SDValue
Cpu0TargetLowering::LowerFormalArguments (SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool isVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         DebugLoc dl, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                         const {

...
// 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 (DAG.getMachineFunction().getFunction() ->hasStructRetAttr()) {
    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[0]);
    Chain = DAG.getNode(ISD::TokenFactor, dl, MVT::Other, Copy, Chain);
}
...
}

addiu $2, $sp, 168
st $2, 0($sp)      // *0($sp) = 168($sp); LowerFormalArguments():
                    // return register is $2, virtual register is
                    // 0($sp)
ld $t9, %call16(_Z8copyDate4Date)($gp)

```

### Ibdex/Chapter9\_4/Cpu0ISelLowering.cpp

```

SDValue
Cpu0TargetLowering::LowerReturn (SDValue Chain,
                                 CallingConv::ID CallConv, bool isVarArg,
                                 const SmallVectorImpl<ISD::OutputArg> &Outs,
                                 const SmallVectorImpl<SDValue> &OutVals,
                                 DebugLoc 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 (DAG.getMachineFunction().getFunction() ->hasStructRetAttr()) {
    MachineFunction &MF      = DAG.getMachineFunction();
    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());

    Chain = DAG.getCopyToReg(Chain, dl, Cpu0::V0, Val, Flag);
    Flag = Chain.getValue(1);
}

```

```

        RetOps.push_back(DAG.getRegister(Cpu0::V0, getPointerTy()));
    }
    ...
}

.globl _Z8copyDateP4Date
.align 2
.type _Z8copyDateP4Date, @function
.ent _Z8copyDate4Date      # @_Z8copyDate4Date
_Z8copyDate4Date:
.cfi_startproc
.frame $sp, 0, $lr
.mask 0x00000000, 0
.set noreorder
.set nomacro
# BB#0:
st $5, 4($sp)
ld $2, 0($sp)           // $2 = 168($sp); LowerReturn(): virtual
                        // register is 0($sp), return register is $2
ld $3, 24($sp)
st $3, 20($2)           // copy date1, 24..4($sp), to date2,
ld $3, 20($sp)           // 188..168($sp)
st $3, 16($2)
ld $3, 16($sp)
st $3, 12($2)
ld $3, 12($sp)
st $3, 8($2)
ld $3, 8($sp)
st $3, 4($2)
ld $3, 4($sp)
st $3, 0($2)
ret $lr
.set macro
.set reorder
.end _Z8copyDate4Date

```

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. You can also mark the “hasStructRetAttr() if” part from both of above functions, the output cpu0 code will use \$3 instead of \$2 as return register as follows,

```

.globl _Z8copyDateP4Date
.align 2
.type _Z8copyDateP4Date, @function
.ent _Z8copyDateP4Date      # @_Z8copyDateP4Date
_Z8copyDateP4Date:
.cfi_startproc
.frame $sp, 8, $lr
.mask 0x00000000, 0
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -8
$tmp3:
.cfi_def_cfa_offset 8
ld $2, 12($sp)
st $2, 0($sp)
ld $4, 20($2)
ld $3, 8($sp)

```

```

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)
addiu $sp, $sp, 8
ret $lr
.set  macro
.set  reorder
.end  _Z8copyDateP4Date

```

## 9.5.2 Variable number of arguments

Until now, we support fixed number of arguments in formal function definition (Incoming Arguments). This section support variable number of arguments since C language support this feature.

Run Chapter9\_4/ with ch9\_3.cpp as well as clang option, **clang -target mips-unknown-linux-gnu**, to get the following result,

```

118-165-76-131:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3.cpp -emit-llvm -o ch9_3.bc
118-165-76-131:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch9_3.bc -o ch9_3.cpu0.s
118-165-76-131:InputFiles 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 $sp,24,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -24
ld $2, 24($sp)      // amount
st $2, 20($sp)      // amount
addiu $2, $zero, 0
st $2, 16($sp)      // i
st $2, 12($sp)      // val
st $2, 8($sp)       // sum
addiu $3, $sp, 28
st $3, 4($sp)       // arg_ptr = 2nd argument = &arg[1],
                     // since &arg[0] = 24($sp)
st $2, 16($sp)
$BB0_1:             # =>This Inner Loop Header: Depth=1

```

```

ld  $2, 20($sp)
ld  $3, 16($sp)
cmp $3, $2          // compare(i, amount)
jge $BB0_4
jmp $BB0_2
$BB0_2:                      #  in Loop: Header=BB0_1 Depth=1
    // i < amount
    ld  $2, 4($sp)
    addiu $3, $2, 4    // arg_ptr + 4
    st   $3, 4($sp)
    ld   $2, 0($2)    // *arg_ptr
    st   $2, 12($sp)
    ld   $3, 8($sp)    // sum
    add $2, $3, $2      // sum += *arg_ptr
    st   $2, 8($sp)
# BB#3:                      #  in Loop: Header=BB0_1 Depth=1
    // i >= amount
    ld  $2, 16($sp)
    addiu $2, $2, 1    // i++
    st   $2, 16($sp)
    jmp $BB0_1
$BB0_4:
    addiu $sp, $sp, 24
    ret $lr
    .set macro
    .set reorder
    .end _Z5sum_iiz
$tmp1:
    .size _Z5sum_iiz, ($tmp1)-_Z5sum_iiz

.globl main
.align 2
.type main,@function
.ent main          # @main
main:
    .frame $sp,88,$lr
    .mask 0x00004000,-4
    .set noreorder
    .cupload $t9
    .set nomacro
# BB#0:
    addiu $sp, $sp, -88
    st $lr, 84($sp)      # 4-byte Folded Spill
    .cprestore 32
    addiu $2, $zero, 0
    st $2, 80($sp)
    addiu $3, $zero, 5
    st $3, 24($sp)
    addiu $3, $zero, 4
    st $3, 20($sp)
    addiu $3, $zero, 3
    st $3, 16($sp)
    addiu $3, $zero, 2
    st $3, 12($sp)
    addiu $3, $zero, 1
    st $3, 8($sp)
    st $2, 4($sp)
    addiu $2, $zero, 6

```

```

st  $2, 0($sp)
ld  $t9, %call16(_Z5sum_iiz)($gp)
jalr $t9
ld  $gp, 32($sp)
st  $2, 76($sp)
ld  $lr, 84($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 88
ret $lr
.set  macro
.set  reorder
.end  main
$tmp4:
.size main, ($tmp4)-main

```

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 “**ld \$2, 24(\$sp)**” since the stack size of the callee function “**\_Z5sum\_iiz()**” is 24. And set argument pointer, **arg\_ptr**, to **28(\$sp)**, **&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\_4/. The ch9\_3\_2.cpp is C++ template example code, it can be translated into cpu0 backend code too.

#### Index/Chapter9\_4/Cpu0ISelLowering.h

```

class Cpu0TargetLowering : public TargetLowering {
...
private:
...
SDValue LowerVASTART(SDValue Op, SelectionDAG &DAG) const;
...
}

```

#### Index/Chapter9\_4/Cpu0ISelLowering.cpp

```

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

    DebugLoc dl = Op.getDebugLoc();
    SDValue FI = DAG.getFrameIndex(FuncInfo->getVarArgsFrameIndex(),
                                    getPointerTy());

    // vstart 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);
}
...
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool isVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         DebugLoc dl, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                         const {
...
if (isVarArg) {
    unsigned NumOfRegs = 0;
    int FirstRegSlotOffset = 0; // offset of $a0's slot.
    unsigned RegSize = Cpu0::CPURegsRegClass.getSize();
    int RegSlotOffset = FirstRegSlotOffset + ArgLocs.size() * RegSize;

    // Offset of the first variable argument from stack pointer.
    int FirstVaArgOffset;

    FirstVaArgOffset = RegSlotOffset;

    // Record the frame index of the first variable argument
    // which is a value necessary to VASTART.
    LastFI = MFI->CreateFixedObject(RegSize, FirstVaArgOffset, true);
    Cpu0FI->setVarArgsFrameIndex(LastFI);
}
...
}

```

### Index/InputFiles/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 main()
{
    int a = sum<int>(6, 0, 1, 2, 3, 4, 5);

    return a;
}

```

Mips qemu reference <sup>8</sup>, you can download and run it with gcc to verify the result with printf() function. We will verify the code correction in chapter “Run backend” through the CPU0 Verilog language machine.

### 9.5.3 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\_4 support dynamic stack allocation with the following code added.

#### lbdex/Chapter9\_4/Cpu0FrameLowering.cpp

```

void Cpu0FrameLowering::emitPrologue(MachineFunction &MF) const {
    ...
    unsigned FP = Cpu0::FP;
    unsigned ZERO = Cpu0::ZERO;
    unsigned ADDu = Cpu0::ADDu;
    ...
    // if framepointer enabled, set it to point to the stack pointer.
    if (hasFP(MF)) {
        // Insert instruction "move $fp, $sp" at this location.
        BuildMI(MBB, MBBI, dl, TII.get(ADDu), FP).addReg(SP).addReg(ZERO);

        // emit ".cfi_def_cfa_register $fp"
        MCSymbol *SetFPLabel = MMI.getContext().CreateTempSymbol();
        BuildMI(MBB, MBBI, dl,
            TII.get(TargetOpcode::PROLOG_LABEL)).addSym(SetFPLabel);
        DstML = MachineLocation(FP);
        SrcML = MachineLocation(MachineLocation::VirtualFP);
        Moves.push_back(MachineMove(SetFPLabel, DstML, SrcML));
    }
}

```

<sup>8</sup> section “4.5.1 Calling Conventions” of tricore\_llvm.pdf

```

    ...
}

void Cpu0FrameLowering::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);
    } // lbd document - mark - emitEpilogue() if (hasFP(MF))
    ...
}

```

### lbdex/Chapter9\_4/Cpu0ISelLowering.cpp

```

Cpu0TargetLowering::
Cpu0TargetLowering(Cpu0TargetMachine &TM)
    : TargetLowering(TM, new Cpu0TargetObjectFile()),
      Subtarget(&TM.getSubtarget<Cpu0Subtarget>()) {
    ...
    setOperationAction(ISD::DYNAMIC_STACKALLOC, MVT::i32, Expand);
    ...
    setStackPointerRegisterToSaveRestore(Cpu0::SP);
    ...
}

```

### lbdex/Chapter9\_4/Cpu0RegisterInfo.cpp

```

// pure virtual method
BitVector Cpu0RegisterInfo:::
getReservedRegs(const MachineFunction &MF) const {
    ...
    // Reserve FP if this function should have a dedicated frame pointer register.
    if (MF.getTarget().getFrameLowering()->hasFP(MF)) {
        Reserved.set(Cpu0::FP);
    }
    ...
}

```

Run Chapter9\_4 with ch9\_4.cpp will get the following correct result.

```

118-165-72-242:InputFiles 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:InputFiles Jonathan$ llvm-dis ch9_4.bc -o ch9_4.ll
118-165-72-242:InputFiles Jonathan$ cat ch9_4.ll
; ModuleID = 'ch9_4.bc'
target datalayout = "e-p:64:64:64-i1:8:8-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:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch9_4.bc -o
ch9_4.cpu0.s
118-165-72-242:InputFiles 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
    add $fp, $sp, $zero
$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
# BB#1:                      # %SP_return
add    $sp, $fp, $zero
ld      $7, 72($sp)           # 4-byte Folded Reload
ld      $lr, 76($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 80
ret    $2
$BB1_2:                      # %CallStackCheckFailBlk
ld      $t9, %call16(__stack_chk_fail)($gp)
jalr   $t9
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 need frame pointer register **fp** support. As Figure 9.3, 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 just below of alloca() area. 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 9.4 depicted how the stack pointer changes back to the caller stack bottom. As above, the `fp` is set to the 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`.

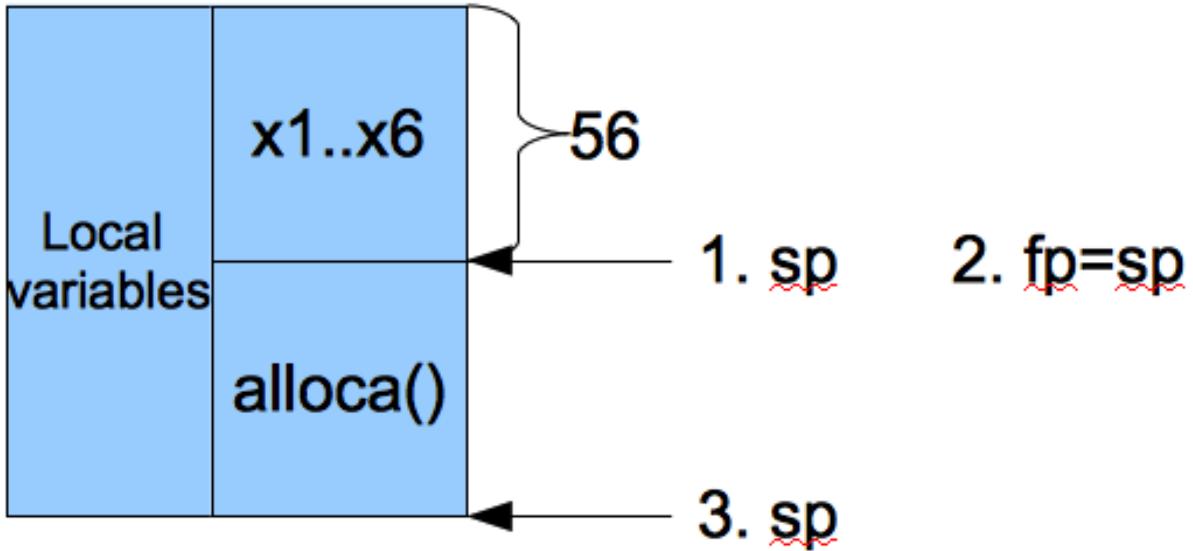


Figure 9.3: Frame pointer changes when enter function

Use `fp` to keep the old stack pointer value is not necessary. Actually, 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 9.5 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.

## 9.6 Summary of this chapter

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 the C++ OOP language into `cpu0` instructions without much effort. Because the most complex things in language, such as C++ syntax, is handled by front end. LLVM is an real structure following the compiler theory, any backend of LLVM can benefit from this structure. A couple of thousands lines of code make OOP language translated into your backend. And your backend will grow up automatically through the front end support languages more and more.

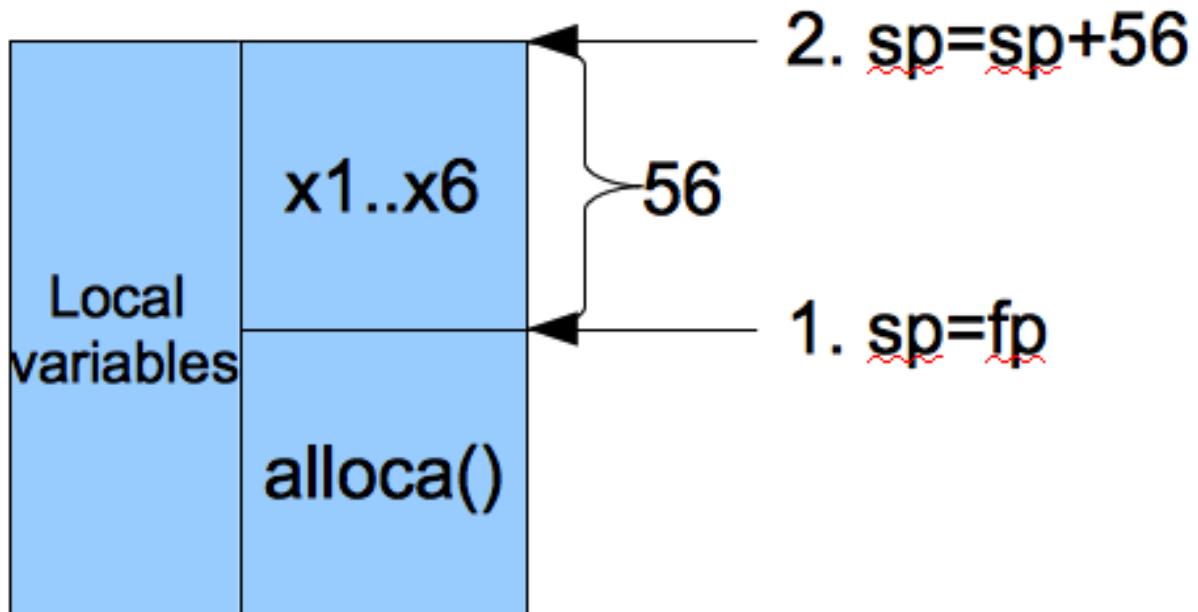


Figure 9.4: Stack pointer changes when exit function

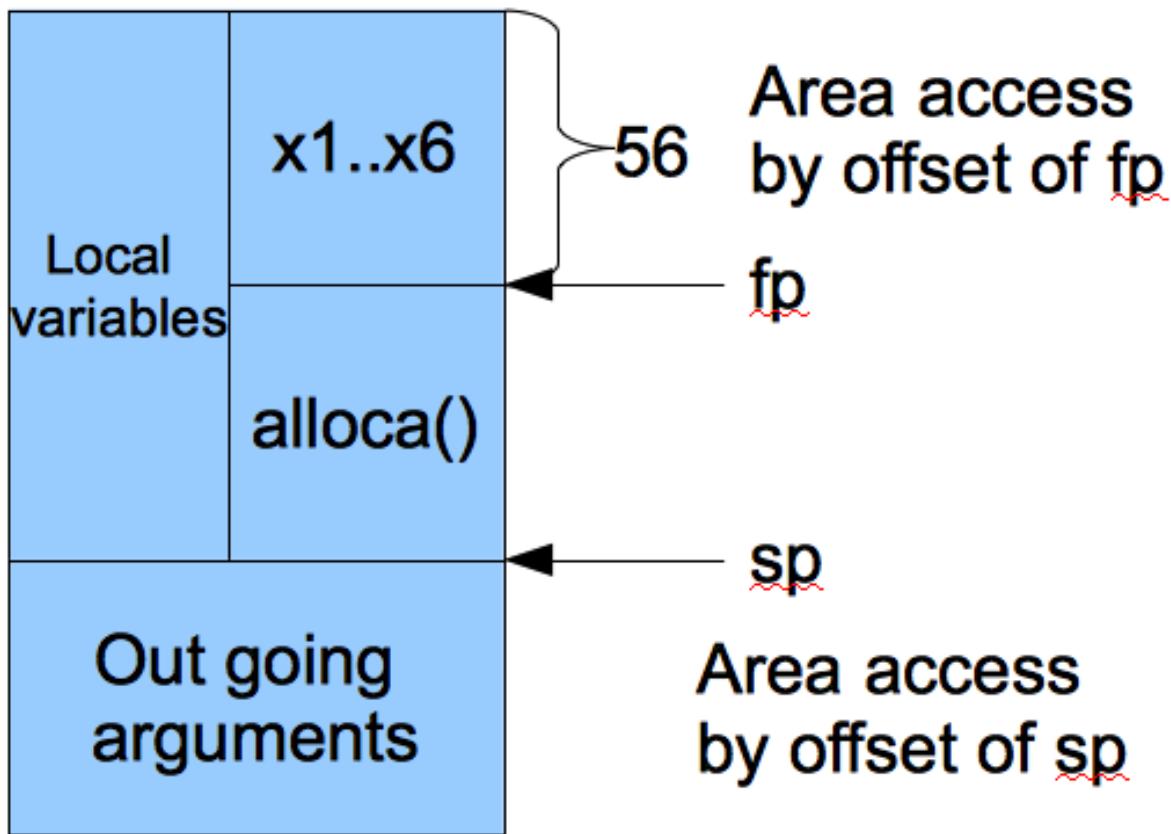


Figure 9.5: fp and sp access areas



# ELF SUPPORT

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 <sup>1</sup>.

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” <sup>2</sup> to analysis 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 in this chapter. The binutils support other CPU ELF dump as a cross compiler tool chains. Linux platform has binutils already and no need to install it further. We use Linux binutils in this chapter just because iMac will display Chinese text. The iMac corresponding binutils have no problem except it use add g in command. For example, use gobjdump instead of objdump, and display with your area language instead of pure English.

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 introduce the topics of assembler, linker, loader, compiler and OS in concept, and at same time demonstrate 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” <sup>3</sup> 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.

<sup>4</sup>, <sup>5</sup>, <sup>6</sup> 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](#).

---

<sup>1</sup> [http://en.wikipedia.org/wiki/Executable\\_and\\_Linkable\\_Format](http://en.wikipedia.org/wiki/Executable_and_Linkable_Format)

<sup>2</sup> <http://jonathan2251.github.com/lbd/install.html#install-other-tools-on-imac>

<sup>3</sup> Leland Beck, System Software: An Introduction to Systems Programming.

<sup>4</sup> <http://ccckmit.wikidot.com/lk:aout>

<sup>5</sup> <http://ccckmit.wikidot.com/lk:objfile>

<sup>6</sup> <http://ccckmit.wikidot.com/lk:elf>

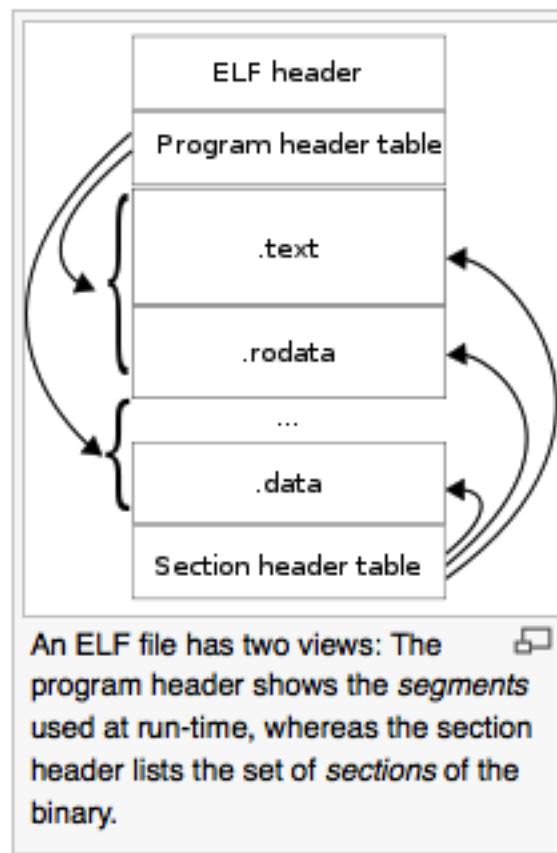


Figure 10.1: ELF file format overview

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 include run time code and data. The definition of segments is run time layout for code and data while sections is link time layout for code and data.

## 10.2 ELF header and Section header table

Let's run Chapter9\_4/ with ch6\_1.cpp, and dump ELF header information by `readelf -h` to see what information the ELF header contains.

```
[Gamma@localhost InputFiles]$ /usr/local/llvm/test/cmake_debug_build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch6_1.bc -o ch6_1.cpu0.o
```

```
[Gamma@localhost InputFiles]$ 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 InputFiles]$
```

```
[Gamma@localhost InputFiles]$ /usr/local/llvm/test/cmake_debug_build/
bin/llc -march=mips -relocation-model=pic -filetype=obj ch6_1.bc -o ch6_1.mips.o
```

```
[Gamma@localhost InputFiles]$ 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 InputFiles]$
```

As above ELF header display, it contains information of magic number, version, ABI, ..., . The Machine field of cpu0 is unknown while mips is MIPS3000. It is because cpu0 is not a popular CPU recognized by utility readelf. Let's check ELF segments information as follows,

```
[Gamma@localhost InputFiles]$ readelf -l ch6_1.cpu0.o
```

There are no program headers in this file.  
[Gamma@localhost InputFiles]\$

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. It contains offset and size information for every section.

```
[Gamma@localhost InputFiles]$ 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] .text          PROGBITS 00000000 000000 000000 00  AX 0  0  4
[ 1] .rel.text      REL       00000000 000310 000018 08  8  1  4
[ 2] .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)
[Gamma@localhost InputFiles]$
```

## 10.3 Relocation Record

The cpu0 backend translate global variable as follows,

```
[Gamma@localhost InputFiles]$ clang -target mips-unknown-linux-gnu -c ch6_1.cpp
-emit-llvm -o ch6_1.bc
[Gamma@localhost InputFiles]$ /usr/local/llvm/test/cmake_debug_build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch6_1.bc -o ch6_1.cpu0.s
[Gamma@localhost InputFiles]$ 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 InputFiles]$ /usr/local/llvm/test/cmake_debug_build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch6_1.bc -o ch6_1.cpu0.o
[Gamma@localhost InputFiles]$ objdump -s ch6_1.cpu0.o

ch6_1.cpu0.o:      file format elf32-big

Contents of section .text:
// .cupload machine instruction
0000 0fa00000 09aa0000 13aa6000 ..... .
...
0020 002a0000 00220000 012d0000 09dd0008 .*...."-.....
...
[Gamma@localhost InputFiles]$ Jonathan$


[Gamma@localhost InputFiles]$ 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
[ 0]             NULL            00000000 0000000 0000000 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 0000000 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 InputFiles]$ readelf -tr ch6_1.mips.o
There are 9 section headers, starting at offset 0xc8:
```

Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Lk	Inf	Al
		Flags							
[ 0]	NULL		00000000	000000	000000	00	0	0	0
	[00000000]:								
[ 1]	.text	PROGBITS	00000000	000034	000038	00	0	0	4
	[00000006]:	ALLOC, EXEC							
[ 2]	.rel.text	REL	00000000	0002f8	000018	08	7	1	4
	[00000000]:								
[ 3]	.data	PROGBITS	00000000	00006c	000008	00	0	0	4
	[00000003]:	WRITE, ALLOC							
[ 4]	.bss	NOBITS	00000000	000074	000000	00	0	0	4
	[00000003]:	WRITE, ALLOC							
[ 5]	.reginfo	MIPS_REGINFO	00000000	000074	000018	00	0	0	1
	[00000002]:	ALLOC							
[ 6]	.shstrtab	STRTAB	00000000	00008c	000039	00	0	0	1
	[00000000]:								
[ 7]	.symtab	SYMTAB	00000000	000230	0000a0	10	8	6	4
	[00000000]:								
[ 8]	.strtab	STRTAB	00000000	0002d0	000025	00	0	0	1
	[00000000]:								

Relocation section '.rel.text' at offset 0x2f8 contains 3 entries:

Offset	Info	Type	Sym. Value	Sym. Name
00000000	00000905	R_MIPS_HI16	00000000	_gp_disp
00000004	00000906	R_MIPS_LO16	00000000	_gp_disp
0000001c	00000709	R_MIPS_GOT16	00000004	gI

As depicted in section Handle \$gp register in PIC addressing mode, it translates “.cupload %reg” into the following.

```
// Lower ".cupload $reg" to
// "lui $gp, %hi(_gp_disp)"
// "addiu $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 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 “addiu \$gp, \$gp, %lo(\_gp\_disp)” which their corresponding obj encode are 0fa00000 and 09aa0000. 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 depend on the program is static link or dynamic link.

## 10.4 Cpu0 ELF related files

Files Cpu0ELFObjectWrite.cpp and Cpu0MC\*.cpp are the files take care the obj format. Most obj code translation are defined by Cpu0InstrInfo.td and Cpu0RegisterInfo.td. With these td description, LLVM translate Cpu0 instruction into obj format automatically.

## 10.5 llvm-objdump

### 10.5.1 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:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch9_3.cpp -emit-llvm -o ch9_3.bc
118-165-83-10:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=obj ch9_3.bc -o
ch9_3.cpu0.o

118-165-78-12:InputFiles Jonathan$ gobjdump -t -r ch9_3.cpu0.o

ch9_3.cpu0.o:      file format elf32-big

SYMBOL TABLE:
00000000 1  df  *ABS*          00000000 ch9_3.bc
00000000 1  d   .text          00000000 .text
00000000 1  d   .data          00000000 .data
00000000 1  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:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llvm-objdump -t -r ch9_3_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 1 df *ABS* 00000000 ch9_3.bc
00000000 1 d .text 00000000 .text
00000000 1 d .data 00000000 .data
00000000 1 d .bss 00000000 .bss
00000000 g F .text 00000084 _Z5sum_iiz
00000084 g F .text 00000080 main
00000000 *UND* 00000000 _gp_disp
```

The latter llvm-objdump can display the file format and relocation records information since we add the relocation records information in ELF.h as follows,

### include/support/ELF.h

```
// Machine architectures
enum {
    ...
    EM_CPU0      = 201, // Document Write An LLVM Backend Tutorial For Cpu0
    ...
}

// include/object/ELF.h
...
template<support::endianness target_endianness, bool is64Bits>
error_code ELFObjectFile<target_endianness, is64Bits>
    ::getRelocationTypeName(DataRefImpl Rel,
                           SmallVectorImpl<char> &Result) const {
    ...
    switch (Header->e_machine) {
    case ELF::EM_CPU0: // llvm-objdump -t -r
        switch (type) {
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_NONE);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_16);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_32);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_REL32);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_24);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_HI16);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_LO16);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_GPREL16);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_LITERAL);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_GOT16);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_PC24);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_CALL16);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_GPREL32);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_SHIFT5);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_SHIFT6);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_64);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_GOT_DISP);
            LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_GOT_PAGE);
```

```

LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_GOT_OFST);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_GOT_HI16);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_GOT_LO16);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_SUB);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_INSERT_A);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_INSERT_B);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_DELETE);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_HIGHER);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_HIGHEST);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_CALL_HI16);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_CALL_LO16);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_SCN_DISP);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_REL16);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_ADD_IMMEDIATE);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_PJUMP);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_RELGOT);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_JALR);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_DTPMOD32);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_DTPREL32);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_DTPMOD64);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_DTPREL64);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_GD);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_LDM);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_DTPREL_HI16);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_DTPREL_LO16);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_GOTTPREL);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_TPREL32);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_TPREL64);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_TPREL_HI16);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_TLS_TPREL_LO16);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_GLOB_DAT);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_COPY);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_JUMP_SLOT);
LLVM_ELF_SWITCH_RELLOC_TYPE_NAME(R_CPU0_NUM);

default:
    res = "Unknown";
}
break;
...
}

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 -
    res = symname;
break;
...
}

template<support::endianness target_endianness, bool is64Bits>
StringRef ELFObjectFile<target_endianness, is64Bits>
    ::getFileName() 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;
...
}
```

### 10.5.2 llvm-objdump -d

Run the last Chapter example code with command `llvm-objdump -d` for dump file from elf to hex as follows,

```
JonathantekiiMac:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c
ch8_1_1.cpp -emit-llvm -o ch8_1_1.bc
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=obj ch8_1_1.bc
-o ch8_1_1.cpu0.o
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/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/Chapter10\_1/CMakeLists.txt

```
tablegen(LLVM Cpu0GenDisassemblerTables.inc -gen-disassembler)
...
```

#### Ibdex/Chapter10\_1/LLVMBuild.txt

```
[common]
subdirectories = Disassembler ...
...
has_disassembler = 1
...
```

### Ibdex/Chapter8\_1/Cpu0InstrInfo.td

```
def brtarget24      : Operand<OtherVT> {
    ...
    let DecoderMethod = "DecodeBranch24Target";
}
```

### Ibdex/Chapter10\_1/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> {
    ...
    let DecoderMethod = "DecodeCMPInstruction";
}

...
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;
}

let isCall=1, hasDelaySlot=0 in {
    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> {
        let DecoderMethod = "DecodeJumpAbsoluteTarget";
    }
}

def JR      : JumpFR<0x2C, "ret", CPURegs>;
```

### Ibdex/Chapter10\_1/Disassembler/CMakeLists.txt

### Ibdex/Chapter10\_1/Disassembler/LLVMBuild.txt

### Ibdex/Chapter10\_1/Disassembler/Cpu0Disassembler.cpp

As above code, it add directory Disassembler for handling the obj to assembly code reverse translation. 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 DecodeCMPInstruction(), DecodeBranchTarget() 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, like llvmm-objdump -d. You can check the comments above these DecodeMethod functions to see how it work. For

the CMP instruction, since there are 3 operand \$rc, \$ra and \$rb occurs in CmpInstr<...>, and the assembler print \$ra and \$rb. LLVM table generate system will print operand 1 and 2 (\$ra and \$rb) in the table generated function printInstruction(). The operand 0 (\$rc) didn't be printed in printInstruction() since assembly print \$ra and \$rb only. In the CMP decode function, we didn't decode shamt field because we don't want it to be displayed and it's not in the assembler print pattern of Cpu0InstrInfo.td.

The RET (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. The IR node Cpu0ISD::Ret is created in LowerReturn() which called at function exit point.

Now, run Chapter10\_1/ with command `llvm-objdump -d ch8_1_1.cpu0.o` will get the following result.

```
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=obj
ch8_1_1.bc -o ch8_1_1.cpu0.o
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llvm-objdump -d ch8_1_1.cpu0.o

ch8_1_1.cpu0.o:          file format ELF32-CPU0

Disassembly of section .text:
_Z13test_control1v:
_Z13test_control1v:
 0: 09 dd ff d0          addiu $sp, $sp, -48
 4: 02 cd 00 2c          st $fp, 44($sp)
 8: 11 cd 00 00          addu $fp, $sp, $zero
 c: 09 30 00 00          addiu $3, $zero, 0
10: 02 3c 00 28          st $3, 40($fp)
14: 09 20 00 01          addiu $2, $zero, 1
18: 02 2c 00 24          st $2, 36($fp)
1c: 09 40 00 02          addiu $4, $zero, 2
20: 02 4c 00 20          st $4, 32($fp)
24: 09 40 00 03          addiu $4, $zero, 3
28: 02 4c 00 1c          st $4, 28($fp)
2c: 09 40 00 04          addiu $4, $zero, 4
30: 02 4c 00 18          st $4, 24($fp)
34: 09 40 00 05          addiu $4, $zero, 5
38: 02 4c 00 14          st $4, 20($fp)
3c: 09 40 00 06          addiu $4, $zero, 6
40: 02 4c 00 10          st $4, 16($fp)
44: 09 40 00 07          addiu $4, $zero, 7
48: 02 4c 00 0c          st $4, 12($fp)
4c: 09 40 00 08          addiu $4, $zero, 8
50: 02 4c 00 08          st $4, 8($fp)
54: 09 40 00 09          addiu $4, $zero, 9
58: 02 4c 00 04          st $4, 4($fp)
5c: 01 4c 00 28          ld $4, 40($fp)
60: 10 43 00 00          cmp $zero, $4, $3
64: 31 00 00 10          jne $zero, 16
68: 36 00 00 00          jmp 0
6c: 01 4c 00 28          ld $4, 40($fp)
70: 09 44 00 01          addiu $4, $4, 1
74: 02 4c 00 28          st $4, 40($fp)
78: 01 4c 00 24          ld $4, 36($fp)
7c: 10 43 00 00          cmp $zero, $4, $3
80: 30 00 00 10          jeq $zero, 16
84: 36 00 00 00          jmp 0
88: 01 4c 00 24          ld $4, 36($fp)
8c: 09 44 00 01          addiu $4, $4, 1
```

```
90: 02 4c 00 24          st  $4, 36($fp)
94: 01 4c 00 20          ld  $4, 32($fp)
98: 10 42 00 00          cmp $zero, $4, $2
9c: 32 00 00 10          jlt $zero, 16
a0: 36 00 00 00          jmp 0
a4: 01 4c 00 20          ld  $4, 32($fp)
a8: 09 44 00 01          addiu $4, $4, 1
ac: 02 4c 00 20          st  $4, 32($fp)
b0: 01 4c 00 1c          ld  $4, 28($fp)
b4: 10 43 00 00          cmp $zero, $4, $3
b8: 32 00 00 10          jlt $zero, 16
bc: 36 00 00 00          jmp 0
c0: 01 4c 00 1c          ld  $4, 28($fp)
c4: 09 44 00 01          addiu $4, $4, 1
c8: 02 4c 00 1c          st  $4, 28($fp)
cc: 09 40 ff ff          addiu $4, $zero, -1
d0: 01 5c 00 18          ld  $5, 24($fp)
d4: 10 54 00 00          cmp $zero, $5, $4
d8: 33 00 00 10          jgt $zero, 16
dc: 36 00 00 00          jmp 0
e0: 01 4c 00 18          ld  $4, 24($fp)
e4: 09 44 00 01          addiu $4, $4, 1
e8: 02 4c 00 18          st  $4, 24($fp)
ec: 01 4c 00 14          ld  $4, 20($fp)
f0: 10 43 00 00          cmp $zero, $4, $3
f4: 33 00 00 10          jgt $zero, 16
f8: 36 00 00 00          jmp 0
fc: 01 3c 00 14          ld  $3, 20($fp)
100: 09 33 00 01         addiu $3, $3, 1
104: 02 3c 00 14         st  $3, 20($fp)
108: 01 3c 00 10         ld  $3, 16($fp)
10c: 10 32 00 00         cmp $zero, $3, $2
110: 33 00 00 10         jgt $zero, 16
114: 36 00 00 00         jmp 0
118: 01 3c 00 10         ld  $3, 16($fp)
11c: 09 33 00 01         addiu $3, $3, 1
120: 02 3c 00 10         st  $3, 16($fp)
124: 01 3c 00 0c         ld  $3, 12($fp)
128: 10 32 00 00         cmp $zero, $3, $2
12c: 32 00 00 10         jlt $zero, 16
130: 36 00 00 00         jmp 0
134: 01 2c 00 0c         ld  $2, 12($fp)
138: 09 22 00 01         addiu $2, $2, 1
13c: 02 2c 00 0c         st  $2, 12($fp)
140: 01 2c 00 0c         ld  $2, 12($fp)
144: 01 3c 00 08         ld  $3, 8($fp)
148: 10 32 00 00         cmp $zero, $3, $2
14c: 35 00 00 10         jge $zero, 16
150: 36 00 00 00         jmp 0
154: 01 2c 00 08         ld  $2, 8($fp)
158: 09 22 00 01         addiu $2, $2, 1
15c: 02 2c 00 08         st  $2, 8($fp)
160: 01 2c 00 24         ld  $2, 36($fp)
164: 01 3c 00 28         ld  $3, 40($fp)
168: 10 32 00 00         cmp $zero, $3, $2
16c: 30 00 00 10         jeq $zero, 16
170: 36 00 00 00         jmp 0
174: 01 2c 00 04         ld  $2, 4($fp)
```

```

178: 09 22 00 01      addiu $2, $2, 1
17c: 02 2c 00 04      st $2, 4($fp)
180: 01 2c 00 24      ld $2, 36($fp)
184: 01 3c 00 28      ld $3, 40($fp)
188: 11 23 20 00      addu $2, $3, $2
18c: 01 3c 00 20      ld $3, 32($fp)
190: 11 22 30 00      addu $2, $2, $3
194: 01 3c 00 1c      ld $3, 28($fp)
198: 11 22 30 00      addu $2, $2, $3
19c: 01 3c 00 18      ld $3, 24($fp)
1a0: 11 22 30 00      addu $2, $2, $3
1a4: 01 3c 00 14      ld $3, 20($fp)
1a8: 11 22 30 00      addu $2, $2, $3
1ac: 01 3c 00 10      ld $3, 16($fp)
1b0: 11 22 30 00      addu $2, $2, $3
1b4: 01 3c 00 0c      ld $3, 12($fp)
1b8: 11 22 30 00      addu $2, $2, $3
1bc: 01 3c 00 08      ld $3, 8($fp)
1c0: 11 22 30 00      addu $2, $2, $3
1c4: 01 3c 00 04      ld $3, 4($fp)
1c8: 11 22 30 00      addu $2, $2, $3
1cc: 11 dc 00 00      addu $sp, $fp, $zero
1d0: 01 cd 00 2c      ld $fp, 44($sp)
1d4: 09 dd 00 30      addiu $sp, $sp, 48
1d8: 3c e0 00 00      ret $lr

```

## 10.6 Dynamic link

We explain how the dynamic link work for Cpu0 even though the Cpu0 linker and dynamic linker not exist at this point. Same with other parts, Cpu0 dynamic link implementation borrowed from Mips ABI. We trace the dynamic link implementation by lldb on X86 platform. Finding X86 and Mips all use the plt as the dynamic link implementation.

### 10.6.1 Linker support

In this section, it shows what's the code that compiler generate to support dynamic link. And what's the code generated by linker for dynamic link.

Compile main.cpp to get the Cpu0 PIC assembly code as follows,

lbdex/InputFiles/main.cpp

```

extern int foo(int x1, int x2);
extern int bar();

int main()
{
//  ENABLE_TRACE;
    int a = 0;

    a = foo(1, 2);
    a = bar();
}

```

```
    return 0;
}

18:165-77-200:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm main.bc -o -
    .section .mdebug.abi32
    .previous
    .file "main.bc"
    .text
    .globl main
    .align 2
    .type main,@function
    .ent main # @main
main:
    .cfi_startproc
    .frame $fp,40,$lr
    .mask 0x00004880,-4
    .set noreorder
    .cupload $t9
    .set nomacro
# BB#0:                                # %entry
    addiu $sp, $sp, -40
$tmp3:
    .cfi_def_cfa_offset 40
    st $lr, 36($sp) # 4-byte Folded Spill
    st $fp, 32($sp) # 4-byte Folded Spill
    st $7, 28($sp) # 4-byte Folded Spill
$tmp4:
    .cfi_offset 14, -4
$tmp5:
    .cfi_offset 11, -8
$tmp6:
    .cfi_offset 7, -12
    addu $fp, $sp, $zero
$tmp7:
    .cfi_def_cfa_register 11
    .cprestore 8
    addiu $2, $zero, 0
    st $2, 24($fp)
    addiu $2, $zero, 2
    st $2, 4($sp)
    addiu $2, $zero, 1
    st $2, 0($sp)
    ld $7, %call16(_Z3fooii)($gp)
    add $t9, $zero, $7
    jalr $t9
    ld $gp, 8($fp)
    st $2, 20($fp)
    addiu $2, $zero, 4
    st $2, 4($sp)
    addiu $2, $zero, 3
    st $2, 0($sp)
    add $t9, $zero, $7
    jalr $t9
    ld $gp, 8($fp)
    ld $3, 20($fp)
    addu $2, $3, $2
    st $2, 20($fp)
```

```

ld      $t9, %call16(_Z3barv) ($gp)
jalr  $t9
ld      $gp, 8($fp)
ld      $3, 20($fp)
addu  $2, $3, $2
st      $2, 20($fp)
addu  $sp, $fp, $zero
ld      $7, 28($sp)           # 4-byte Folded Reload
ld      $fp, 32($sp)           # 4-byte Folded Reload
ld      $lr, 36($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 40
ret    $lr
.set  macro
.set  reorder
.end  main

$tmp8:
.size main, ($tmp8)-main
.cfi_endproc

118-165-77-200:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=obj main.bc -o
main.cpu0.o
118-165-77-200:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llvm-objdump -r main.cpu0.o

main.cpu0.o: file format ELF32-CPU0

RELOCATION RECORDS FOR [.text]:
4 R_CPU0_L016 _gp_disp
52 R_CPU0_CALL16 _Z3fooii
112 R_CPU0_CALL16 _Z3barv

RELOCATION RECORDS FOR [.eh_frame]:
28 R_CPU0_32 .text

```

Suppost we have the linker which support dynamic link of Cpu0. After linker, the plt for dynamic link function \_Z3fooii and \_Z3barv are solved and the ELF file looks like the following,

SYMBOL TABLE:

```

...
0040035c 1      d  .dynsym      00000000          .dynsym
...

```

Disassembly of section .plt:

```

...
00400720 <_Z3barv@plt>:
400720:    lui      $8,0x41
400724:    ld      $t9,0x0acc($8)
400738:    addiu  $9,$zero,0x18
40073c:    jr      $t9

```

```

00400730 <_Z3fooii@plt>:
400730:    lui      $8,0x41
400734:    ld      $t9,0x0ado($8)
400738:    addiu  $9,$zero,0x08
40073c:    jr      $t9
...

```

```
004009f0 <.CPU0.stubs>:
```

```

4009F0: 8f998010  ld      $t9,-32752 (gp)
4009F4: 03e07821  add    $8,$zero, lr
4009F8: 0320f809  jalr   $t9
    
```

## 10.6.2 Principle

To support dynamic link, Cpu0 set the protocol as Table registers changed for call dynamic link function `_Z3fooii()`. The `.dynsym+0x08` include the dynamic link function information. Usually the information include which library and the offset value in this library. This information can be got and saved in ELF file in link time.

After the ELF is loaded to memory, it looks like Figure 10.2

Table 10.1: registers changed for call dynamic link function `_Z3fooii()`

register/memory	call <code>_Z3fooii</code> first time	call <code>_Z3fooii</code> second time
0x410ad0	point to CPU0.stubs	point to <code>_Z3fooii</code>
<code>.dynsym+0x08</code>	(libfoobar.so, offset, length) about <code>_Z3fooii</code>	useless
-32752(gp)	point to dynamic_linker	useless
\$8	the next instruction of <code>_Z3fooii()</code>	useless
\$9	<code>.dynsym+0x08</code>	useless
\$t9 (at the end of <code>_Z3fooii@plt</code> )	point to CPU0.stubs0	point to <code>_Z3fooii</code>
\$t9 (at the end of CPU0.stubs)	point to dynamic_linker	useless

Explains it as follows,

1. As you can see, the first time of function call, `a = foo(1,2)`, which is implemented by instructions “`ld $7, %call16(_Z3fooii@plt)($gp)`”, “`add $t9, $zero, $7`” and “`jalr $t9`”. Remember, `.dynsym+0x08` contains information (libfoobar.so, offset, length) which is set by linker at link to dynamic shared library. After “`jalr $t9`”, PC counter jump to “`00400730 <_Z3fooii@plt>`”.
2. The memory `0x410ad0` contents is the address of CPU0.stubs when the program, `main()`, is loaded.
3. After `_Z3fooii@plt` instructions executed, it jump to CPU0.stubs since `$t9` = the address of CPU0.stubs. Register `$9` = the contents of address `.dynsym+0x08` since it is set in step 1.
4. After CPU0.stubs is executed, register `$8 = 0x004008a4` which point to the caller next instruction in step 1.
5. Dynamic linker looks into register `$9` which value is `0x08`. It ask OS for the caller process address information of `.dynsym + offset 0x08`. This address include information (libfoobar.so, offset, length). With this information, dynamic linker knows where can get `_Z3fooii` function body. Dynamic linker loads `_Z3fooii()` function body to an available address where from asking OS. After load `_Z3fooii()`, it call `_Z3fooii()` and save and restore the registers `$t9, $8, $9` and caller saved registers just before and after call `_Z3fooii()`.
6. After `_Z3fooii()` return, dynamic linker set the contents of address `0x410ad0` to the entry address of `_Z3fooii` in memory.
7. Dynamic linker execute `jr $8`. It jump to the next instruction of “`a = _Z3fooii();`” in caller.

After the `_Z3fooii()` is called at second time, it looks like Figure 10.3. It jump to `_Z3fooii()` directly in `<_Z3fooii@plt>` since the contents of address `0x410ad0` is changed to the memory address of `_Z3fooii()` at step 6 of Figure 10.2. From now on, any call `_Z3fooii()` will jump to `_Z3fooii()` directly from `_Z3fooii@plt` instructions.

According Mips Application Binary Interface (ABI), `$t9` is register alias for `$25` in Mips. The `%t9` is the register used in `jalr $25` for long distance function pointer (far subroutine call). Cpu0 use register `$t9($6)` as the `$t9 ($25)` register of Mips. The `jal %subroutine` has 24 bits range of address offset relative to Program Counter (PC) while `jalr` has 32 bits address range in register size of 32 bits. One example of PIC mode is used in share library just like this example. Share library is re-entry code which can be loaded in different memory address decided on run time. The `jalr` make the implementation of dynamic link function easier and faster as above.

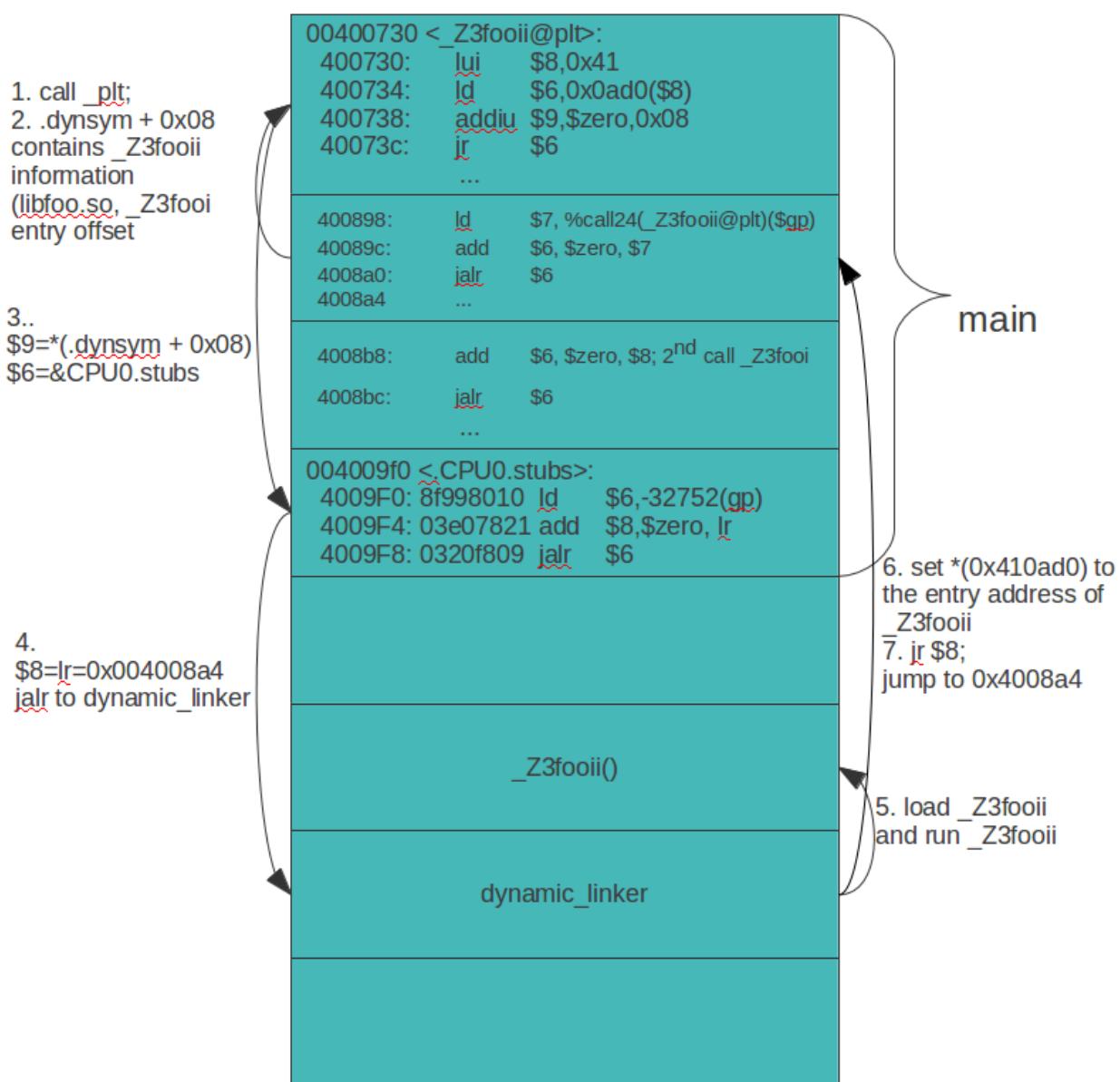
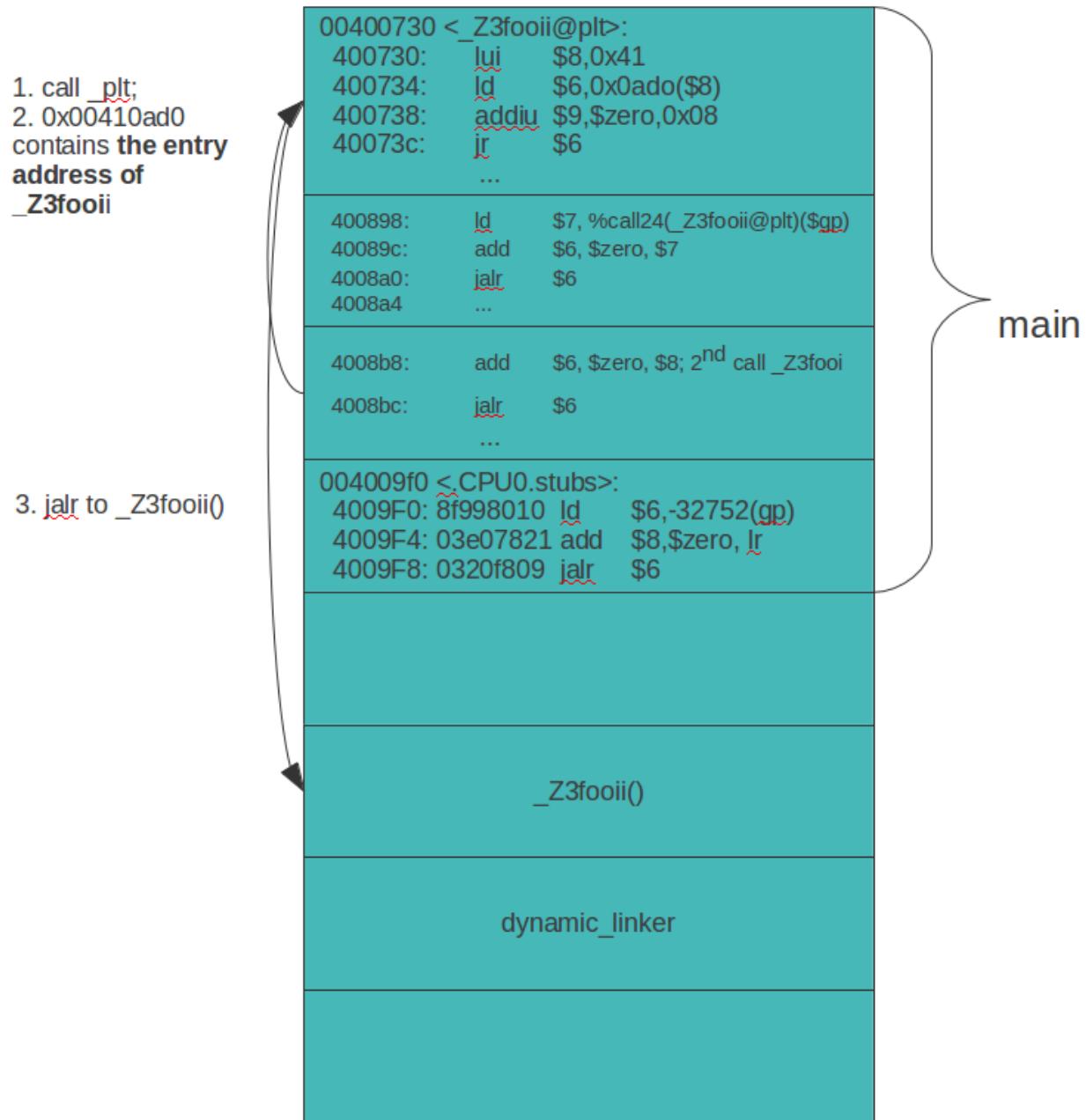


Figure 10.2: Call dynamic function `_Z3fooii()` first time

Figure 10.3: Call dynamic function `_Z3fooii()` second time

### 10.6.3 Trace with lldb

We tracking the dynamic link on X86 as below. You can skip it if you have no interest or already know how to track it via lldb or gdb.

```
118-165-77-200:InputFiles Jonathan$ clang -fPIC -g -c foobar.cpp
118-165-77-200:InputFiles Jonathan$ clang -shared -g foobar.o -o libfoobar.so
118-165-77-200:InputFiles Jonathan$ clang -g -c main.cpp
118-165-77-200:InputFiles Jonathan$ clang -g main.o libfoobar.so
118-165-77-200:InputFiles Jonathan$ gobjdump -d main.o
```

```
main.o:      fileformat mach-o-x86-64
```

Disassembly of section .text:

```
0000000000000000 <_main>:
 0: 55                      push   %rbp
 1: 48 89 e5                mov    %rsp,%rbp
 4: 48 83 ec 10             sub    $0x10,%rsp
 8: bf 01 00 00 00           mov    $0x1,%edi
 d: be 02 00 00 00           mov    $0x2,%esi
12: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
19: e8 00 00 00 00           callq  1e <_main+0x1e>
1e: bf 03 00 00 00           mov    $0x3,%edi
23: be 04 00 00 00           mov    $0x4,%esi
28: 89 45 f8                mov    %eax,-0x8(%rbp)
2b: e8 00 00 00 00           callq  30 <_main+0x30>
30: 8b 75 f8                mov    -0x8(%rbp),%esi
33: 01 c6                  add    %eax,%esi
35: 89 75 f8                mov    %esi,-0x8(%rbp)
38: e8 00 00 00 00           callq  3d <_main+0x3d>
3d: 8b 75 f8                mov    -0x8(%rbp),%esi
40: 01 c6                  add    %eax,%esi
42: 89 75 f8                mov    %esi,-0x8(%rbp)
45: 8b 45 f8                mov    -0x8(%rbp),%eax
48: 48 83 c4 10             add    $0x10,%rsp
4c: 5d                      pop    %rbp
4d: c3                      retq
```

```
118-165-77-200:InputFiles Jonathan$ gobjdump -d a.out
```

```
...
```

```
main.o:      fileformat mach-o-x86-64
```

Disassembly of section .text:

```
0000000100000ef0 <_main>:
100000ef0: 55                      push   %rbp
100000ef1: 48 89 e5                mov    %rsp,%rbp
100000ef4: 48 83 ec 10             sub    $0x10,%rsp
100000ef8: bf 01 00 00 00           mov    $0x1,%edi
100000efd: be 02 00 00 00           mov    $0x2,%esi
100000f02: c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
100000f09: e8 36 00 00 00           callq  100000f44 <__Z3fooii$stub>
100000f0e: bf 03 00 00 00           mov    $0x3,%edi
100000f13: be 04 00 00 00           mov    $0x4,%esi
100000f18: 89 45 f8                mov    %eax,-0x8(%rbp)
100000f1b: e8 24 00 00 00           callq  100000f44 <__Z3fooii$stub>
100000f20: 8b 75 f8                mov    -0x8(%rbp),%esi
```

```

100000f23: 01 c6          add    %eax,%esi
100000f25: 89 75 f8      mov    %esi,-0x8(%rbp)
100000f28: e8 11 00 00 00  callq 100000f3e <__Z3barv$stub>
100000f2d: 8b 75 f8      mov    -0x8(%rbp),%esi
100000f30: 01 c6          add    %eax,%esi
100000f32: 89 75 f8      mov    %esi,-0x8(%rbp)
100000f35: 8b 45 f8      mov    -0x8(%rbp),%eax
100000f38: 48 83 c4 10  add    $0x10,%rsp
100000f3c: 5d            pop    %rbp
100000f3d: c3            retq

```

Disassembly of section \_\_TEXT.\_\_stubs:

```

0000000100000f3e <__Z3barv$stub>:
100000f3e: ff 25 cc 00 00 00  jmpq   *0xcc(%rip)      # 100001010
<__Z3barv$stub>

```

```

0000000100000f44 <__Z3fooii$stub>:
100000f44: ff 25 ce 00 00 00  jmpq   *0xce(%rip)      # 100001018
<__Z3fooii$stub>

```

Disassembly of section \_\_TEXT.\_\_stub\_helper:

```

0000000100000f4c <__TEXT.__stub_helper>:
100000f4c: 4c 8d 1d b5 00 00 00  lea    0xb5(%rip),%r11      # 100001008 <>
100000f53: 41 53          push   %r11
100000f55: ff 25 a5 00 00 00  jmpq   *0xa5(%rip)      # 100001000
<dyld_stub_binder$stub>
100000f5b: 90            nop
100000f5c: 68 00 00 00 00  pushq $0x0
100000f61: e9 e6 ff ff ff  jmpq   100000f4c <__Z3fooii$stub+0x8>
100000f66: 68 0f 00 00 00  pushq $0xf
100000f6b: e9 dc ff ff ff  jmpq   100000f4c <__Z3fooii$stub+0x8>

```

Disassembly of section \_\_TEXT.\_\_ unwind\_info:

...

```

118-165-77-200:InputFiles Jonathan$ lldb a.out
Current executable set to 'a.out' (x86_64).
(lldb) run main
Process 702 launched: '/Users/Jonathan/test/lbd/docs/BackendTutorial/
lbdex/InputFiles/a.out' (x86_64)
Process 702 exited with status = 15 (0x0000000f)
(lldb) b main
Breakpoint 1: where = a.out`main + 25 at main.cpp:7, address = 0x0000000100000f09
(lldb) target stop-hook add
Enter your stop hook command(s). Type 'DONE' to end.
> disassemble --pc
> DONE
Stop hook #1 added.
(lldb) run
Process 705 launched: '/Users/Jonathan/test/lbd/docs/BackendTutorial/
lbdex/InputFiles/a.out' (x86_64)
dyld`_dyld_start:
-> 0x7fff5fc01028: popq   %rdi

```

```

0x7fff5fc01029: pushq  $0
0x7fff5fc0102b: movq    %rsp, %rbp
0x7fff5fc0102e: andq    $-16, %rsp
Process 753 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f09 a.out 'main + 25 at main.cpp:7,
stop reason = breakpoint 1.1
    frame #0: 0x0000000100000f09 a.out 'main + 25 at main.cpp:7
4
5         int main()
6         {
-> 7             int a = foo(1, 2);
8             a += foo(3, 4);
9             a += bar();
10
a.out `main + 25 at main.cpp:7:
-> 0x100000f09: callq  0x100000f44           ; symbol stub for: foo(int, int)
 0x100000f0e: movl   $3, %edi
 0x100000f13: movl   $4, %esi
 0x100000f18: movl   %eax, -8(%rbp)
(lldb) stepi
Process 753 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f44 a.out 'foo(int, int), stop reason
= instruction step into
    frame #0: 0x0000000100000f44 a.out 'foo(int, int)
a.out `symbol stub for: foo(int, int):
-> 0x100000f44: jmpq   *206(%rip)           ; (void *)0x0000000100000f66
a.out `symbol stub for: foo(int, int):
-> 0x100000f44: jmpq   *206(%rip)           ; (void *)0x0000000100000f66
 0x100000f4a: addb   %al, (%rax)
 0x100000f4c: addb   %al, (%rax)
 0x100000f4e: addb   %al, (%rax)
(lldb) p $rip
(unsigned long) $1 = 4294971204
(lldb) memory read/4xw 4294971410
0x100001012: 0x00010000 0x0f660000 0x00010000 0x00000000
(lldb) stepi
Process 859 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f66 a.out, stop reason = instruction
    step into frame #0: 0x0000000100000f66 a.out
-> 0x100000f66: pushq  $15
 0x100000f6b: jmpq   0x100000f4c
-> 0x100000f66: pushq  $15
 0x100000f6b: jmpq   0x100000f4c
 0x100000f70: addb   %al, (%rax)
 0x100000f72: addb   %al, (%rax)
) stepi
Process 859 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f6b a.out, stop reason = instruction
    step into frame #0: 0x0000000100000f6b a.out
-> 0x100000f6b: jmpq   0x100000f4c
-> 0x100000f6b: jmpq   0x100000f4c
 0x100000f70: addb   %al, (%rax)
 0x100000f72: addb   %al, (%rax)
 0x100000f74: addb   %al, (%rax)
(lldb) stepi
Process 859 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f4c a.out, stop reason = instruction
    step into frame #0: 0x0000000100000f4c a.out

```

```

-> 0x100000f4c: leaq    181(%rip), %r11          ; (void *)0x0000000000000000
0x100000f53: pushq  %r11
0x100000f55: jmpq   *165(%rip)          ; (void *)0x00007fff978da878:
dyld_stub_binder
0x100000f5b: nop
-> 0x100000f4c: leaq    181(%rip), %r11          ; (void *)0x0000000000000000
0x100000f53: pushq  %r11
0x100000f55: jmpq   *165(%rip)          ; (void *)0x00007fff978da878:
dyld_stub_binder
0x100000f5b: nop
(lldb)
Process 859 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f53 a.out, stop reason = instruction
  step into frame #0: 0x0000000100000f53 a.out
-> 0x100000f53: pushq  %r11
0x100000f55: jmpq   *165(%rip)          ; (void *)0x00007fff978da878:
dyld_stub_binder
0x100000f5b: nop
0x100000f5c: pushq  $0
-> 0x100000f53: pushq  %r11
0x100000f55: jmpq   *165(%rip)          ; (void *)0x00007fff978da878:
dyld_stub_binder
0x100000f5b: nop
0x100000f5c: pushq  $0
(lldb)
Process 859 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f55 a.out, stop reason = instruction
  step into frame #0: 0x0000000100000f55 a.out
-> 0x100000f55: jmpq   *165(%rip)          ; (void *)0x00007fff978da878:
dyld_stub_binder
0x100000f5b: nop
0x100000f5c: pushq  $0
0x100000f61: jmpq   0x100000f4c
-> 0x100000f55: jmpq   *165(%rip)          ; (void *)0x00007fff978da878:
dyld_stub_binder
0x100000f5b: nop
0x100000f5c: pushq  $0
0x100000f61: jmpq   0x100000f4c
(lldb)
Process 859 stopped
* thread #1: tid = 0x1c03, 0x00007fff978da878 libdyld.dylib`dyld_stub_binder,
  stop reason = instruction step into
  frame #0: 0x00007fff978da878 libdyld.dylib`dyld_stub_binder
libdyld.dylib`dyld_stub_binder:
-> 0x7fff978da878: pushq  %rbp
0x7fff978da879: movq   %rsp, %rbp
0x7fff978da87c: subq   $192, %rsp
0x7fff978da883: movq   %rdi, (%rsp)
libdyld.dylib`dyld_stub_binder:
-> 0x7fff978da878: pushq  %rbp
0x7fff978da879: movq   %rsp, %rbp
0x7fff978da87c: subq   $192, %rsp
0x7fff978da883: movq   %rdi, (%rsp)
(lldb) cont
Process 753 resuming
Process 753 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f1b a.out`main + 43 at main.cpp:8,
  stop reason = breakpoint 2.1

```

```

frame #0: 0x0000000100000f1b a.out `main + 43 at main.cpp:8
5         int main()
6         {
7             int a = foo(1, 2);
-> 8             a += foo(3, 4);
9             a += bar();
10
11         return a;
a.out `main + 43 at main.cpp:8:
-> 0x100000f1b: callq  0x100000f44          ; symbol stub for: foo(int, int)
  0x100000f20: movl    -8(%rbp), %esi
  0x100000f23: addl    %eax, %esi
  0x100000f25: movl    %esi, -8(%rbp)
(lldb) stepi
Process 753 stopped
* thread #1: tid = 0x1c03, 0x0000000100000f44 a.out `foo(int, int), stop reason =
  instruction step into frame #0: 0x0000000100000f44 a.out `foo(int, int)
a.out `symbol stub for: foo(int, int):
-> 0x100000f44: jmpq   *206(%rip)          ; (void *)0x0000000100003f20:
  foo(int, int) at /Users/Jonathan/test/lbd/docs/BackendTutorial/LLVMBackendT
  utorialExampleCode/InputFiles/foobar.cpp:3
a.out `symbol stub for: foo(int, int):
-> 0x100000f44: jmpq   *206(%rip)          ; (void *)0x0000000100003f20:
  foo(int, int) at /Users/Jonathan/test/lbd/docs/BackendTutorial/LLVMBackendT
  utorialExampleCode/InputFiles/foobar.cpp:3
  0x100000f4a: addb    %al, (%rax)
  0x100000f4c: addb    %al, (%rax)
  0x100000f4e: addb    %al, (%rax)
(lldb) p $rip
(unsigned long) $2 = 4294971204
(lldb) memory read/4xw 4294971410
0x100001012: 0x00010000 0x3f200000 0x00010000 0x00000000
(lldb) stepi
Process 753 stopped
* thread #1: tid = 0x1c03, 0x0000000100003f20 libfoobar.so `foo(x1=0, x2=3) at
  foobar.cpp:3, stop reason = instruction step into
  frame #0: 0x0000000100003f20 libfoobar.so `foo(x1=0, x2=3) at foobar.cpp:3
1
2         int foo(int x1, int x2)
-> 3         {
4             int sum = x1 + x2;
5
6         return sum;
libfoobar.so `foo(int, int) at foobar.cpp:3:
-> 0x100003f20: pushq  %rbp
  0x100003f21: movq    %rsp, %rbp
  0x100003f24: movl    %edi, -4(%rbp)
  0x100003f27: movl    %esi, -8(%rbp)
(lldb)

```

# RUN BACKEND

This chapter will add LLVM AsmParser support first. With AsmParser support, we can hand code the assembly language in C/C++ file and translate it into obj (elf format). We can write a C++ main function as well as the boot code by assembly hand code, and translate this main() + bootcode() into obj file. Combined with llvm-objdump support in last chapter, this main() + bootcode() elf can be translated into hex file format which include the disassemble code as comment. Furthermore, we can design the Cpu0 with Verilog language tool and run the Cpu0 backend on PC by feed the hex file and see the Cpu0 instructions execution result.

## 11.1 AsmParser support

Run Chapter10\_1/ with ch11\_1.cpp will get the following error message.

### lbdex/InputFiles/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");
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("swi 0x00000400");
asm("jsub 0x000010000");
asm("ret $lr");
asm("jalr $t9");
asm("li $3, 0x00700000");
asm("la $3, 0x00800000($6)");
asm("la $3, 0x00900000");
```

```
JonathantekiiMac:InputFiles Jonathan$ clang -c ch11_1.cpp -emit-llvm -o ch11_1.bc
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/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 didn't implement cpu0 assembly, 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 into obj. Directory AsmParser handle the assembly to obj translation. The Chapter11\_1/include AsmParser implementation as follows,

[Index/Chapter11\\_1/AsmParser/Cpu0AsmParser.cpp](#)

[Index/Chapter11\\_1/AsmParser/CMakeLists.txt](#)

[Index/Chapter11\\_1/AsmParser/LLVMBuild.txt](#)

The Cpu0AsmParser.cpp contains one thousand of code which do the assembly language parsing. You can understand it with a little patient only. To let directory AsmParser be built, modify CMakeLists.txt and LLVMBuild.txt as follows,

[Index/Chapter11\\_1/CMakeLists.txt](#)

```
tablegen(LLVM Cpu0GenAsmMatcher.inc -gen-asm-matcher)
...
add_subdirectory(AsmParser)
```

[Index/Chapter11\\_1/LLVMBuild.txt](#)

```
subdirectories = AsmParser ...
...
has_asmparser = 1
```

The other files change as follows,

[Index/Chapter11\\_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp](#)

```
unsigned Cpu0MCCodeEmitter::
getBranchTargetOpValue(const MCInst &MI, unsigned OpNo,
SmallVectorImpl<MCFixup> &Fixups) const {
```

```

    ...
    // If the destination is an immediate, we have nothing to do.
    if (MO.isImm()) return MO.getImm();
    ...
}

/// getJumpAbsoluteTargetOpValue - Return binary encoding of the jump
/// target operand. Such as SWI.
unsigned Cpu0MCCodeEmitter::
getJumpAbsoluteTargetOpValue(const MCInst &MI, unsigned OpNo,
    SmallVectorImpl<MCFixup> &Fixups) const {
    ...
    // If the destination is an immediate, we have nothing to do.
    if (MO.isImm()) return MO.getImm();
    ...
}

```

### lbdex/Chapter11\_1/Cpu0.td

```

def Cpu0AsmParser : AsmParser {
    let ShouldEmitMatchRegisterName = 0;
}

def Cpu0AsmParserVariant : AsmParserVariant {
    int Variant = 0;

    // Recognize hard coded registers.
    string RegisterPrefix = "$";
}

def Cpu0 : Target {
    ...
    let AssemblyParsers = [Cpu0AsmParser];
    ...
    let AssemblyParserVariants = [Cpu0AsmParserVariant];
}

```

### lbdex/Chapter11\_1/Cpu0InstrFormats.td

```

// Pseudo-instructions for alternate assembly syntax (never used by codegen).
// These are aliases that require C++ handling to convert to the target
// instruction, while InstAliases can be handled directly by tblgen.
class Cpu0AsmPseudoInst<dag outs, dag ins, string asmstr>:
    Cpu0Inst<outs, ins, asmstr, [], IIPseudo, Pseudo> {
        let isPseudo = 1;
        let Pattern = [];
    }

```

### lbdex/Chapter11\_1/Cpu0InstrInfo.td

```

// Cpu0InstrInfo.td
def Cpu0MemAsmOperand : AsmOperandClass {
    let Name = "Mem";

```

```

let ParserMethod = "parseMemOperand";
}

// Address operand
def mem : Operand<i32> {
    ...
    let ParserMatchClass = Cpu0MemAsmOperand;
}
...
class CmpInstr<...
    !strconcat(instr_asm, "\t$rc, $ra, $rb"), [], itin> {
    ...
}
...
class CBranch<...
    !strconcat(instr_asm, "\t$ra, $addr"), ...> {
    ...
}
...
//=====
// Pseudo Instruction definition
//=====

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, CPURegs>;

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, CPURegs>;

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, CPURegs>;

```

Above define the **ParserMethod = “parseMemOperand”** and implement the `parseMemOperand()` in `Cpu0AsmParser.cpp` to handle the **“mem”** operand which used in `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(
    SmallVectorImpl<MCParsedAsmOperand*> &Operands,
    StringRef Mnemonic);
OperandMatchResultTy tryCustomParseOperand(
    SmallVectorImpl<MCParsedAsmOperand*> &Operands,

```

```
    unsigned MCK);  
  
Cpu0AsmParser::OperandMatchResultTy Cpu0AsmParser::  
tryCustomParseOperand(SmallVectorImpl<MCParsedAsmOperand*> &Operands,  
    unsigned MCK) {  
  
    switch(MCK) {  
    case MCK_Mem:  
        return parseMemOperand(Operands);  
    default:  
        return MatchOperand_NoMatch;  
    }  
    return MatchOperand_NoMatch;  
}  
  
Cpu0AsmParser::OperandMatchResultTy Cpu0AsmParser::  
MatchOperandParserImpl(SmallVectorImpl<MCParsedAsmOperand*> &Operands,  
    StringRef Mnemonic) {  
    ...  
}  
  
/// MatchClassKind - The kinds of classes which participate in  
/// instruction matching.  
enum MatchClassKind {  
    ...  
    MCK_Mem, // user defined class 'Cpu0MemAsmOperand'  
    ...  
};
```

Above 3 Pseudo Instruction definitions in Cpu0InstrInfo.td such as LoadImm32Reg are handled by Cpu0AsmParser.cpp as follows,

#### [Index/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,  
                      SmallVectorImpl<MCParsedAsmOperand*> &Operands,  
                      MCStreamer &Out, unsigned &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);  
    ...  
}  
...  
}  
}
```

Finally, remind the CPURegs as below must follow the order of register number because AsmParser use this when do register number encode.

## Ibdex/Chapter11\_1/Cpu0RegisterInfo.td

Run Chapter11\_1/ with ch11\_1.cpp to get the correct result as follows,

```
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=obj ch11_1.bc -o
ch11_1.cpu0.o
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llvm-objdump -d ch11_1.cpu0.o
```

ch11\_1.cpu0.o: file format ELF32-unknown

Disassembly of section .text:

.text:

0: 00 2d 00 08	ld \$2, 8(\$sp)
4: 01 0d 00 04	st \$zero, 4(\$sp)
8: 09 30 00 00	addiu \$3, \$zero, 0
c: 13 31 20 00	add \$3, \$at, \$2
10: 14 32 30 00	sub \$3, \$2, \$3
14: 15 21 30 00	mul \$2, \$at, \$3
18: 16 32 00 00	div \$3, \$2
1c: 17 23 00 00	divu \$2, \$3
20: 18 21 30 00	and \$2, \$at, \$3
24: 19 31 20 00	or \$3, \$at, \$2
28: 1a 12 30 00	xor \$at, \$2, \$3
2c: 50 43 00 00	mult \$4, \$3
30: 51 32 00 00	multu \$3, \$2
34: 40 30 00 00	mfhi \$3
38: 41 20 00 00	mflo \$2
3c: 42 20 00 00	mthi \$2
40: 43 20 00 00	mtlo \$2
44: 1b 22 00 02	sra \$2, \$2, 2
48: 1c 21 10 03	rol \$2, \$at, 3
4c: 1d 33 10 04	ror \$3, \$3, 4

```

50: 1e 22 00 02          shl $2, $2, 2
54: 1f 23 00 05          shr $2, $3, 5
58: 10 23 00 00          cmp $zero, $2, $3
5c: 20 00 00 14          jeq $zero, 20
60: 21 00 00 10          jne $zero, 16
64: 22 ff ff ec          jlt $zero, -20
68: 24 ff ff f0          jle $zero, -16
6c: 23 ff ff fc          jgt $zero, -4
70: 25 ff ff f4          jge $zero, -12
74: 2a 00 04 00          swi 1024
78: 2b 01 00 00          jsub 65536
7c: 2c e0 00 00          ret $lr
80: 2d e6 00 00          jalr $6
84: 09 30 00 70          addiu $3, $zero, 112
88: 1e 33 00 10          shl $3, $3, 16
8c: 09 10 00 00          addiu $at, $zero, 0
90: 19 33 10 00          or $3, $3, $at
94: 09 30 00 80          addiu $3, $zero, 128
98: 1e 36 00 10          shl $3, $6, 16
9c: 09 10 00 00          addiu $at, $zero, 0
a0: 19 36 10 00          or $3, $6, $at
a4: 13 33 60 00          add $3, $3, $6
a8: 09 30 00 90          addiu $3, $zero, 144
ac: 1e 33 00 10          shl $3, $3, 16
b0: 09 10 00 00          addiu $at, $zero, 0
b4: 19 33 10 00          or $3, $3, $at

```

We replace cmp and jeg with explicit \$sw in assembly and \$zero in disassembly for AsmParser support. It's OK with just a little bad in readability and in assembly programing than implicit representation.

## 11.2 Verilog of CPU0

Verilog language is an IEEE standard in IC design. There are a lot of book and documents for this language. Web site <sup>1</sup> has a pdf <sup>2</sup> in this. Example code lbdex/cpu0\_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 is a simple design with only few hundreds lines of code. Alough it has not the pipeline features, we can assume the cpu0 backend code run on the pipeline machine because the pipeline version use the same machine instructions. Verilog is 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 need further explanation. There are two type of I/O. One is memory mapped I/O, the other is instruction I/O. CPU0 use memory mapped I/O, we set the memory address 0x7000 as the output port. When meet the instruction “**st \$ra, cx(\$rb)**”, where cx(\$rb) is 0x7000 (28672), CPU0 display the content as follows,

```

ST :
if (R[b]+c16 == 28672)
$display("%4dns %8x : %8x OUTPUT=%-d", $stime, pc0, ir, R[a]);

```

### lbdex/cpu0\_verilog/cpu0.v

```

`define MEMSIZE    'h80000
`define MEMEMPTY   8'hFF

```

<sup>1</sup> <http://www.ece.umd.edu/courses/enee359a/>

<sup>2</sup> [http://www.ece.umd.edu/courses/enee359a/verilog\\_tutorial.pdf](http://www.ece.umd.edu/courses/enee359a/verilog_tutorial.pdf)

```

#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);
    reg signed [31:0] R [0:15];
    // 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, SW;

    // register name
    #define PC    R[15]    // Program Counter
    #define LR    R[14]    // Link Register
    #define SP    R[13]    // Stack Pointer
    // 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
    // Instruction Opcode
    parameter [7:0] 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, 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, 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,
    `ifdef CPU0II

```

```

SLTi=8'h26,SLTiU=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,
SWI=8'h3A,JSUB=8'h3B,RET=8'h3C,IRET=8'h3D,JALR=8'h3E,
MULT=8'h41,MULTu=8'h42,DIV=8'h43,DIVu=8'h44,
MFHI=8'h46,MFLO=8'h47,MTHI=8'h48,MTLO=8'h49,
MFSW=8'h50,MTSW=8'h51;

reg [0:0] inInt = 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, WriteBack=3'h4;
integer i;

// 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

task memReadEnd(output [31:0] data); begin // Read Memory Finish, get data
    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 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 (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[7:0]);
end endtask

task taskInterrupt(input [2:0] iMode); begin
if (inInt == 0) begin
    case (iMode)
        'RESET: begin
            'PC = 0; tick = 0; R[0] = 0; SW = 0; 'LR = -1;
            'IE = 0; 'IOE = 0; 'I1E = 0; 'I2E = 0; 'I = 0; 'IO = 0; 'I1 = 0;
            'I2 = 0;
        end
        'ABORT: begin 'LR = 'PC; 'PC = 4; end
        'IRQ: begin 'LR = 'PC; 'PC = 8; end
        'ERROR: begin 'LR = 'PC; 'PC = 12; end
    endcase
    $display("taskInterrupt(%3b)", iMode);
    inInt = 1;
end
end endtask

task taskExecute; begin
    m_en = 0;
    tick = tick+1;
    case (state)
        Fetch: begin // Tick 1 : instruction fetch, throw PC to address bus,
            // memory.read(m[PC])
            memReadStart('PC, 'INT32);
            pc0 = 'PC;
            '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
    end
end

```

```

case (op)
// 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);
// Mathematic
ADDiu: R[a] = Rb+c16;                      // ADDiu Ra, Rb+Cx; Ra<=Rb+Cx
CMP:   begin 'N=(Ra-Rb<0); 'Z=(Ra-Rb==0); end // CMP Ra, Rb; SW=(Ra >= Rb)
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; 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; end
                // SUB Ra,Rb,Rc; Ra<=Rb-Rc
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;
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
MFSW:  regSet(a, SW);                   // MFSW Ra; Ra<=SW
MTSW:  SW = Ra;                         // MTSW Ra; SW<=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

`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;
    SLTiu: 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
    SWI:   begin
        'LR='PC; 'PC= c24; 'I0 = 1'b1; 'I = 1'b1;
    end // Software Interrupt; SWI Cx; LR <= PC; PC <= Cx; INT<=1
    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
    IRET:  begin
        'PC=Ra; 'I = 1'b0; 'M = 'EXE;
    end // Interrupt Return; IRET; PC <= LR; INT<=0
    default :
        $display("%4dns %8x : OP code %8x not support", $stime, pc0, op);
    endcase
    next_state = WriteBack;
end
WriteBack: begin // Read/Write finish, close memory
    case (op)
        LD, LB, LBu, LH, LHu : memReadEnd(R[a]);
                                //read memory complete
        ST, SB, SH : memWriteEnd();
                                // write memory complete
    endcase
    case (op)
        MULT, MULTu, DIV, DIVu, MTHI, MTLO, MTSW :
            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[%-04d+%-04d]=%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[%-04d+%-04d]=%c SW=%8x", $stime, pc0, ir,
            R[b], c16, R[a][7:0], SW);
        if (R[b]+c16 == 'IOADDR) begin
            outc(R[a][7:0]);
        end
    end
    default :
        if ('D) // Display the written register content
            $display("%4dns %8x : %8x R[%02d]=-8x=%-d SW=%8x", $stime, pc0, ir,
            a, R[a], R[a], SW);
    endcase
    if ('PC < 0) begin
        $display("RET to PC < 0, finished!");
        $finish;
    end
    next_state = Fetch;
end
endcase
end endtask

always @(posedge clock) begin
    if (inInt == 0 && itype == 'RESET) begin
        taskInterrupt('RESET);
        'M = 'RESET;
        state = Fetch;
    end else if (inInt == 0 && (state == Fetch) && ('IE && 'I) &&
        (('I0E && 'I0) || ('I1E && 'I1) || ('I2E && 'I2)) ) begin
        'M = 'IRQ;
        taskInterrupt('IRQ);
        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);
    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] 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;

`ifdef DLINKER
`include "dynlinker.v"
`endif
initial begin
// erase memory
for (i=0; i < `MEMSIZE; i=i+1) begin
    m[i] = 'MEMEMPTY;
end
// 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;
            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[24:0];
                'INT32: {m[abus], m[abus+1], m[abus+2], m[abus+3]} = dbus_in;
            endcase
        end else if (en == 1 && rw == 1) begin// r_w==1:read
            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 else
            data = 32'hZZZZZZZZ;
    `ifdef DLINKER
    `include "flashio.v"
    `endif
    end else
        data = 32'hZZZZZZZZ;
    end
    assign dbus_out = data;
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;

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));

memory0 mem(.clock(clock), .reset(reset), .en(m_en), .rw(m_rw),
.m_size(m_size), .abus(mar), .dbus_in(mdr), .dbus_out(dbus));

initial
begin
    clock = 0;
    itype = 'RESET;
    #300000000 $finish;
end

always #10 clock=clock+1;

endmodule

```

#### Ibdex/cpu0\_verilog/cpu0Is.v

```

// TRACE: Display the memory contents of the loaded program and data
//`define TRACE

`include "cpu0.v"

JonathantekiiMac:raw Jonathan$ pwd
/Users/Jonathan/test/2/lbd/lbdex/cpu0_verilog/raw
JonathantekiiMac:raw Jonathan$ iverilog -o cpu0Is cpu0Is.v

```

## 11.3 Run program on CPU0 machine

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 cpu0.v use 0x7000 bytes of memory.

#### Ibdex/InputFiles/InitRegs.cpp

```

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 $6,      $ZERO, 0");
asm("addiu $7,      $ZERO, 0");
asm("addiu $8,      $ZERO, 0");
asm("addiu $9,      $ZERO, 0");
asm("addiu $10, $ZERO, 0");
asm("addiu $gp, $ZERO, 0");

```

```
asm("addiu $fp, $ZERO, 0");
asm("addiu $lr, $ZERO, -1");
```

### lbdex/InputFiles/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/InputFiles/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/InputFiles/ch\_run\_backend.cpp**

```
#include "boot.cpp"

#include "print.h"

int test_math();
int test_div();
int test_local_pointer();
int test_andorxornot();
int test_setxx();
bool test_load_bool();
long long test_longlong();
int test_control1();
int sum_i(int amount, ...);

int main()
{
    int a = 0;
    a = test_math();
    print_integer(a); // a = 74
    a = test_div();
    print_integer(a); // a = 253
    a = test_local_pointer();
    print_integer(a); // a = 3
    a = (int)test_load_bool();
    print_integer(a); // a = 1
    a = test_andorxornot(); // a = 14
    print_integer(a);
    a = test_setxx(); // a = 3
    print_integer(a);
    long long b = test_longlong(); // 0x8000000002
    print_integer((int)(b >> 32)); // 393307
    print_integer((int)b); // 16777222
    a = test_control1();
    print_integer(a); // a = 51
    print_integer(2147483647); // test mod % (mult) from itoa.cpp
    print_integer(-2147483648); // test mod % (multu) from itoa.cpp
    a = sum_i(6, 0, 1, 2, 3, 4, 5);
    print_integer(a); // a = 15

    return a;
}

#include "print.cpp"

void print1_integer(int x)
{
    asm("ld $at, 8($sp)");
    asm("st $at, 28672($0)");

    return;
}

#if 0
// For instruction IO
void print2_integer(int x)
{
```

```

asm("ld $at, 8($sp)");
asm("outw $stat");
return;
}
#endif

bool test_load_bool()
{
    int a = 1;

    if (a < 0)
        return false;

    return true;
}

#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;
}

#include "ch4_1.cpp"
#include "ch4_3.cpp"
#include "ch4_5.cpp"
#include "ch7_1.cpp"
#include "ch7_4.cpp"
#include "ch8_1_1.cpp"

```

```

Jonathan@Mac:InputFiles Jonathan$ pwd
/Users/Jonathan/test/2/lbd/lbdex/InputFiles
Jonathan@Mac:InputFiles Jonathan$ clang -c ch_run_backend.cpp -emit-llvm -o
ch_run_backend.bc
Jonathan@Mac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=obj
ch_run_backend.bc -o ch_run_backend.cpu0.o
Jonathan@Mac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llvm-objdump -d ch_run_backend.cpu0.o | tail -n +6 | awk '{print /* '
$1 " */\t" $2 " " $3 " " $4 " " $5 " \t/* " $6"\t" $7" " $8" " $9" " $10 "\t*/"}'
> ../cpu0_verilog/raw/cpu0.hex

```

```

118-165-81-39:raw Jonathan$ cat cpu0.hex
...
/* 4c: */ 2b 00 00 20 /* jsub 0      */
/* 50: */ 01 2d 00 04 /* st $2, 4($sp)    */

```

```
/* 54: */ 2b 00 01 44 /* jsub 0      */
```

As above code the subroutine address for “**jsub #offset**” are 0. This is correct since C language support separate compile and the subroutine address is decided at link time for static address mode or at load time for PIC address mode. Since our backend didn’t implement the linker and loader, we change the “**jsub #offset**” encode in Chapter11\_2/ as follow,

#### Ibdex/Chapter11\_2/MCTargetDesc/Cpu0MCCCodeEmitter.cpp

```
unsigned Cpu0MCCCodeEmitter::  
getJumpTargetOpValue(const MCInst &MI, unsigned OpNo,  
                    SmallVectorImpl<MCFixup> &Fixups) const {  
  
    unsigned Opcode = MI.getOpcode();  
    ...  
    if (Opcode == Cpu0::JSUB)  
        Fixups.push_back(MCFixup::Create(0, Expr,  
                                         MCFixupKind(Cpu0::fixup_Cpu0_PC24)));  
    else if (Opcode == Cpu0::JSUB)  
        Fixups.push_back(MCFixup::Create(0, Expr,  
                                         MCFixupKind(Cpu0::fixup_Cpu0_24)));  
    else  
        llvm_unreachable("unexpect opcode in getJumpAbsoluteTargetOpValue()");  
  
    return 0;  
}
```

We change JSUB from Relocation Records fixup\_Cpu0\_24 to Non-Relocation Records fixup\_Cpu0\_PC24 as the definition below. This change is fine since if call a outside defined subroutine, it will add a Relocation Record for this “**jsub #offset**”. At this point, we set it to Non-Relocation Records for run on CPU0 Verilog machine. If one day, the CPU0 linker is appeared and the linker do the sections arrangement, we should adjust it back to Relocation Records. A good linker will reorder the sections for optimization in data/function access. In other word, keep the global variable access as close as possible to reduce cache miss possibility.

#### Ibdex/Chapter11\_2/MCTargetDesc/Cpu0AsmBackend.cpp

```
const MCFixupKindInfo &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_24",          0,      24,    0 },  
        ...  
        { "fixup_Cpu0_PC24",        0,      24,  MCFixupKindInfo::FKF_IsPCRel },  
        ...  
    }  
    ...  
}
```

Let’s run the Chapter11\_2/ with `llvm-objdump -d` for input files `ch_run_backend.cpp` to generate the hex file and input to `cpu0Is` Verilog simulator to get the output result as below. Remind `ch_run_backend.cpp` have to compile with option `clang -target mips-unknown-linux-gnu` and use the `clang` of your build instead of download from Xcode on iMac. The `~/llvm/release/cmake_debug_build/bin/Debug` is my build `clang` from source code.

```
JonathantekiiMac:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu -c ch_run_backend.cpp -emit-llvm -o ch_run_backend.bc
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=obj ch_run_backend.bc -o ch_run_backend.cpu0.o
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/bin/Debug/llvm-objdump -d ch_run_backend.cpu0.o | tail -n +6 | awk '{print /* /* $1 " */\t" $2 " " $3 " " $4 " " $5 "\t/* " $6 "\t" $7 " " $8 " " $9 " " $10 "\t*/"}'
> ..../cpu0_verilog/raw/cpu0.hex

JonathantekiiMac:raw Jonathan$ ./cpu0Is
WARNING: cpu0Is.v:386: $readmemh(cpu0.hex): Not enough words in the file for the
taskInterrupt(001)
74
253
3
1
14
3
393307
16777222
51
2147483647
-2147483648
15
RET to PC < 0, finished!
```

From the result as below, you can find the `print_integer()` which implemented by C language has more instructions while the `print1_integer()` which implemented by assembly has less instructions. But the C version is better in portability since the assembly version is binding with machine assembly language and make the assumption that the stack size of `print1_integer()` is 8.

```
.....          ld  $2, 8($fp)
.....          st  $2, 0($fp)
.....          ld  $1, 8($sp)
.....          st  $1, 28672($zero)
.....          add $sp, $fp, $zero
.....          ld  $fp, 4($sp)
.....          addiu $sp, $sp, 8
.....          ret $lr
```

You can trace the memory binary code and destination register change at every instruction execution by the following change and get the result as below,

#### **lbdex/cpu0\_verilog/cpu0Is.v**

```
'define TRACE
```

#### **lbdex/cpu0\_verilog/cpu0.v**

```
...
`TR = 1; // Trace register content at beginning

Jonathan@Mac:~/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 dump the memory first after read input cpu0.hex. Next, it run 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. We have checked many example code is correct by print the variable with print\_integer().

This chapter show Verilog PC output by display the I/O memory mapped address but didn't implementing the output hardware interface or port. 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.



# BACKEND OPTIMIZATION

This chapter introduce how to do backend optimization in LLVM first. Next we do optimization via redesign instruction sets with hardware level to do optimization by create a efficient RISC CPU which aim to C/C++ high level language.

## 12.1 Cpu0 backend Optimization: Remove useless JMP

LLVM use functional pass in code generation and optimization. Following the 3 tiers of compiler architecture, LLVM did much optimization in middle tier of which is LLVM IR, SSA form. In spite of this middle tier optimization, there are opportunities in optimization which depend on backend features. Mips fill delay slot is an example of backend optimization used in pipeline RISC machine. You can modify from Mips this part if your backend is a pipeline RISC with delay slot. We apply the “delete useless jmp” unconditional branch instruction in Cpu0 backend optimization in this section. This algorithm is simple and effective as a perfect tutorial in optimization. You can understand how to add a optimization pass and design your complicate optimization algorithm on your backend in real project.

Chapter12\_1/ support this optimization algorithm include the added codes as follows,

### lbdex/Chapter12\_1/CMakeLists.txt

```
add_llvm_target(Cpu0CodeGen
...
Cpu0DelUselessJMP.cpp
...
)
```

### lbdex/Chapter12\_1/Cpu0.h

```
...
FunctionPass *createCpu0DelJmpPass(Cpu0TargetMachine &TM);

// Cpu-TargetMachine.cpp
class Cpu0PassConfig : public TargetPassConfig {
...
    virtual bool addPreEmitPass();
};

// 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.
```

```
bool Cpu0PassConfig::addPreEmitPass() {
    Cpu0TargetMachine &TM = getCpu0TargetMachine();
    addPass(createCpu0DelJmpPass(TM));
    return true;
}
```

### Index/Chapter12\_1/Cpu0DelUselessJMP.cpp

As above code, except Cpu0DelUselessJMP.cpp, other files changed for register class DelJmp as a functional pass. As comment of above code, MBB is the current block and MBBN is the next block. For the 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 function of MachineOperand, please check include/llvm/CodeGen/MachineOperand.h Let's run Chapter12\_1/ with ch12\_1.cpp to explain it easier.

### Index/InputFiles/ch12\_1.cpp

```
int main()
{
    int a = 0;
    int b = 1;
    int c = 2;

    if (a == 0) {
        a++;
    }
    if (b == 0) {
        a = a + b;
    } else if (b < 0) {
        a = a--;
    }
    if (c > 0) {
        c++;
    }

    return a;
}
```

```
118-165-78-10:InputFiles Jonathan$ clang -target mips-unknown-linux-gnu
-c ch12_1.cpp -emit-llvm -o ch12_1.bc
118-165-78-10:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=asm -stats
ch12_1.bc -o ch12_1.cpu0.s
=====
... Statistics Collected ...
=====
...
2 del-jmp      - Number of useless jmp deleted
...
.section .mdebug.abi32
.previous
.file "ch12_1.bc"
.text
.globl main
.align 2
```

```
.type main,@function
.ent main                               # @main
main:
.frame $sp,16,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro
# BB#0:
addiu $sp, $sp, -16
addiu $2, $zero, 0
st $2, 12($sp)
st $2, 8($sp)
addiu $2, $zero, 1
st $2, 4($sp)
addiu $2, $zero, 2
st $2, 0($sp)
ld $2, 8($sp)
bne $2, $zero, $BB0_2
# BB#1:
ld $2, 8($sp)
addiu $2, $2, 1
st $2, 8($sp)
$BB0_2:
ld $2, 4($sp)
bne $2, $zero, $BB0_4
jmp $BB0_3
$BB0_4:
ld $2, 4($sp)
addiu $3, $zero, -1
slt $2, $3, $2
bne $2, $zero, $BB0_6
jmp $BB0_5
$BB0_3:
ld $2, 4($sp)
ld $3, 8($sp)
addu $2, $3, $2
st $2, 8($sp)
jmp $BB0_6
$BB0_5:
ld $2, 8($sp)
addiu $3, $2, -1
st $3, 8($sp)
st $2, 8($sp)
$BB0_6:
ld $2, 0($sp)
slti $2, $2, 1
bne $2, $zero, $BB0_8
# BB#7:
ld $2, 0($sp)
addiu $2, $2, 1
st $2, 0($sp)
$BB0_8:
ld $2, 8($sp)
addiu $sp, $sp, 16
ret $lr
.set macro
.set reorder
.end main
```

```
$tmp1:  
.size main, ($tmp1)-main
```

The terminal display “Number of useless jmp deleted” by `llc -stats` option because we set the “STATISTIC(NumDelJmp, “Number of useless jmp deleted”)” in code. It delete 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 from no optimization version. If you run with `ch8_1_1.cpp`, will find 10 jmp instructions are deleted in 100 lines of assembly code, which meaning 10% enhance in speed and code size.

## 12.2 Cpu0 Optimization: Redesign instruction sets

If you compare the cpu0 and Mips instruction sets, you will find the following,

1. Mips has **addu** and **add** two different instructions for No Trigger Exception and Trigger Exception.
2. Mips use SLT, BEQ and set the status in explicit/general register while Cpu0 use CMP, JEQ and set status in implicit/specific register.

According RISC spirits, this section will replace CMP, JEQ with Mips style instructions and support both Trigger and No Trigger Exception operators. Mips style BEQ instructions will reduce the number of branch instructions too. Which means optimization in speed and code size.

### 12.2.1 Cpu0 new instruction sets table

Add Cpu0 instructions as follows,

- First column F.: meaning Format.

Table 12.1: Cpu0 Instruction Set :widths: 1 4 3 11 7 10 :header-rows: 1

F.	Mnemonic	Opcode	Meaning	Syntax	Operation
L	SLTi	26	Set less Then	SLTi Ra, Rb, Cx	$Ra \leq (Rb < Cx)$
L	SLTi <sub>u</sub>	27	SLTi unsigned	SLTi <sub>u</sub> Ra, Rb, Cx	$Ra \leq (Rb < Cx)$
A	SLT	28	Set less Then	SLT Ra, Rb, Rc	$Ra \leq (Rb < Rc)$
A	SLT <sub>u</sub>	29	SLT unsigned	SLT <sub>u</sub> Ra, Rb, Rc	$Ra \leq (Rb < Rc)$
L	BEQ	37	Jump if equal	BEQ Ra, Rb, Cx	if ( $Ra == Rb$ ), $PC \leq PC + Cx$
L	BNE	38	Jump if not equal	BNE Ra, Rb, Cx	if ( $Ra \neq Rb$ ), $PC \leq PC + Cx$

### 12.2.2 Cpu0 code changes

Chapter12\_2/ include the changes for new instruction sets as follows,

#### lbdex/Chapter12\_2/Disassembler/Cpu0Disassembler.cpp

```
static DecodeStatus DecodeBranch16Target (MCInst &Inst,
                                         unsigned Insn,
                                         uint64_t Address,
                                         const void *Decoder);  
...  
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;
}
```

### Ibidx/Chapter12\_2/MCTargetDesc/Cpu0AsmBackend.cpp

```
static unsigned adjustFixupValue(unsigned Kind, uint64_t Value) {
...
// Add/subtract and shift
switch (Kind) {
...
case Cpu0::fixup_Cpu0_PC16:
case Cpu0::fixup_Cpu0_PC24:
    // So far we are only using this type for branches.
    // For branches we start 1 instruction after the branch
    // so the displacement will be one instruction size less.
    Value -= 4;
    break;
...
}
...
const MCFixupKindInfo &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_PC16",          0,      16,  MCFixupKindInfo::FKF_IsPCRel },
...
}
```

### Ibidx/Chapter12\_2/MCTargetDesc/Cpu0ELFObjectWriter.cpp

```
unsigned Cpu0ELFObjectWriter::GetRelocType(const MCValue &Target,
                                            const MCFixup &Fixup,
                                            bool IsPCRel,
                                            bool IsRelocWithSymbol,
                                            int64_t Addend) const {
...
switch (Kind) {
...
case Cpu0::fixup_Cpu0_PC16:
Type = ELF::R_CPU0_PC16;
break;
...
}
...
}
```

### lbdex/Chapter12\_2/MCTargetDesc/Cpu0FixupKinds.cpp

```
enum Fixups {
    ...
    // PC relative branch fixup resulting in - R_CPU0_PC16.
    // cpu0 PC16, e.g. beq
    fixup_Cpu0_PC16,
    ...
};
```

### lbdex/Chapter12\_2/MCTargetDesc/Cpu0MCCCodeEmitter.cpp

```
// 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;

...
/// getBranch16TargetOpValue - Return binary encoding of the branch
/// target operand. If the machine operand requires relocation,
/// record the relocation and return zero.
unsigned Cpu0MCCCodeEmitter:::
getBranch16TargetOpValue(const MCInst &MI, unsigned OpNo,
                        SmallVectorImpl<MCFixup> &Fixups) const {
    const MCOperand &MO = MI.getOperand(OpNo);

    // If the destination is an immediate, we have nothing to do.
    if (MO.isImm()) return MO.getImm();
    assert(MO.isExpr() && "getBranch16TargetOpValue expects only expressions");

    const MCExpr *Expr = MO.getExpr();
    Fixups.push_back(MCFixup::Create(0, Expr,
                                      MCFixupKind(Cpu0::fixup_Cpu0_PC16)));
    return 0;
}
```

### lbdex/Chapter12\_2/MCTargetDesc/Cpu0TargetDesc.cpp

```
static std::string ParseCpu0Triple(StringRef TT, StringRef CPU) {
    ...
    if (TheTriple == "cpu0" || TheTriple == "cpu0el") {
        ...
    } else if (CPU == "cpu032II") {
        Cpu0ArchFeature = "+cpu032II";
    }
}
return Cpu0ArchFeature;
}
```

**lbdex/Chapter12\_2/Cpu0InstrInfo.cpp**

```

//=====//
// Cpu0 Subtarget features
//=====//
...
def FeatureCpu032II : SubtargetFeature<"cpu032II", "Cpu0ArchVersion",
    "Cpu032II", "Cpu032II ISA Support (use instruction slt)">;
def FeatureCpu032III : SubtargetFeature<"cpu032III", "Cpu0ArchVersion",
    "Cpu032III", "Cpu032III ISA Support (use instruction slt)">;
//=====//
// Cpu0 processors supported.
//=====//
...
def : Proc<"cpu032I", [FeatureCpu032I]>;
def : Proc<"cpu032II", [FeatureCpu032II]>;
def : Proc<"cpu032III", [FeatureCpu032III]>;

```

**lbdex/Chapter12\_2/Cpu0InstrInfo.cpp**

```

// Cpu0InstrInfo::copyPhysReg()
void Cpu0InstrInfo::
copyPhysReg(MachineBasicBlock &MBB,
            MachineBasicBlock::iterator I, DebugLoc DL,
            unsigned DestReg, unsigned SrcReg,
            bool KillSrc) const {
...
const Cpu0Subtarget &Subtarget = TM.getSubtarget<Cpu0Subtarget>();

if (Cpu0::CPURegsRegClass.contains(DestReg)) { // Copy to CPU Reg.
...
if (!Subtarget.hasCpu032II()) {
    if (SrcReg == Cpu0::SW)
        Opc = Cpu0::MFSW, SrcReg = 0;
}
}
else if (Cpu0::CPURegsRegClass.contains(SrcReg)) { // Copy from CPU Reg.
...
if (!Subtarget.hasCpu032II()) {
    if (DestReg == Cpu0::SW)
        Opc = Cpu0::MTSW, 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));
}

```

### Ibdex/Chapter12\_2/Cpu0InstrInfo.td

```

def NotCpu032II : Predicate<"!Subtarget.hasCpu032II()">,
    AssemblerPredicate<"FeatureCpu032I">;
def HasCpu032II : Predicate<"Subtarget.hasCpu032II()">,
    AssemblerPredicate<"!FeatureCpu032III">;
// !FeatureCpu032III is for disassembler in "llvm-objdump -d"

/*
In Cpu0GenSubtargetInfo.inc,
namespace llvm {
namespace Cpu0 {
enum {
    FeatureCpu032I = 1ULL << 0,
    FeatureCpu032II = 1ULL << 1,
    FeatureCpu032III = 1ULL << 2
};
}
} // End llvm namespace

static bool checkDecoderPredicate(unsigned Idx, uint64_t Bits) {
    switch (Idx) {
    default: llvm_unreachable("Invalid index!");
    case 0:
        return ((Bits & Cpu0::FeatureCpu032I)); // came from "FeatureCpu032I"
    case 1:
        return (!(Bits & Cpu0::FeatureCpu032III)); // came from !FeatureCpu032III"
    }
}

To let disassembler work, the function
checkDecoderPredicate(unsigned Idx, uint64_t Bits) must return true(=1).
As above code, the argument Bits always is 1. Set !FeatureCpu032III" to do
disassembler for expectation.
*/
...
// BEQ, BNE
def brtarget16 : Operand<OtherVT> {
    let EncoderMethod = "getBranch16TargetOpValue";
    let OperandType = "OPERAND_PCREL";
    let DecoderMethod = "DecodeBranch16Target";
}
...
class ArithOverflowR<bits<8> op, string instr_asm,
    InstrItinClass itin, RegisterClass RC, bit isComm = 0>:
    FA<op, (outs RC:$ra), (ins RC:$rb, RC:$rc),
        !strconcat(instr_asm, "\t$ra, $rb, $rc"), [], itin> {
    let shamt = 0;
    let isCommutable = isComm;
}
class CmpInstr<bits<8> op, string instr_asm,
    InstrItinClass itin, RegisterClass RC, RegisterClass RD,

```

```

        bit isComm = 0>:
        ...
        let Predicates = [NotCpu032II];
    }
    // Conditional Branch, e.g. JEQ brtarget24
    class CBranch24<bits<8> op, string instr_asm, RegisterClass RC,
        list<Register> UseRegs>:
        FJ<op, (outs), (ins RC:$ra, brtarget24:$addr),
            !strconcat(instr_asm, "\t$ra, $addr"),
            [], IIBranch>, Requires<[NotCpu032II]> {
        ...
        // let Predicates = [HasCpu032II]; // same effect as Requires
    }

    // Conditional Branch, e.g. BEQ $r1, $r2, brtarget16
    class CBranch16<bits<8> op, string instr_asm, PatFrag cond_op, RegisterClass RC>:
        FL<op, (outs), (ins RC:$ra, RC:$rb, brtarget16:$imm16),
            !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
            [(brcond (i32 (cond_op RC:$ra, RC:$rb)), bb:$imm16)], IIBranch>,
            Requires<[HasCpu032II]> {
        let isBranch = 1;
        let isTerminator = 1;
        let hasDelaySlot = 1;
        let Defs = [AT];
    }

    // SetCC
    class SetCC_R<bits<8> op, string instr_asm, PatFrag cond_op,
        RegisterClass RC>:
        FA<op, (outs CPURegs:$ra), (ins RC:$rb, RC:$rc),
            !strconcat(instr_asm, "\t$ra, $rb, $rc"),
            [(set CPURegs:$ra, (cond_op RC:$rb, RC:$rc))],
            IIAlu>, Requires<[HasCpu032II]> {
        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 CPURegs:$ra), (ins RC:$rb, Od:$imm16),
            !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
            [(set CPURegs:$ra, (cond_op RC:$rb, imm_type:$imm16))],
            IIAlu>, Requires<[HasCpu032II]> {
    }

    ...
    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>;
    ...
    /// Jump and Branch Instructions
    def BEQ      : CBranch<0x30, "beg", seteq, CPURegs>;
    def BNE      : CBranch<0x31, "bne", setne, CPURegs>;
    ...
    // brcond for slt instruction
    multiclass BrcondPatsSlt<RegisterClass RC, Instruction BEQOp, Instruction BNEOp,
        Instruction SLTOp, Instruction SLTuOp, Instruction SLTiOp,
        Instruction SLTiuOp, Register ZEROReg> {
        def : Pat<(brcond (i32 (setne RC:$lhs, 0)), bb:$dst),
    }

```

```

        (BNEOp RC:$lhs, ZEROReg, bb:$dst)>;
def : Pat<(brcond (i32 (seteq RC:$lhs, 0)), bb:$dst),
            (BEQOp RC:$lhs, ZEROReg, bb:$dst)>;

def : Pat<(brcond (i32 (setge RC:$lhs, RC:$rhs)), bb:$dst),
            (BEQ (SLTOp RC:$lhs, RC:$rhs), ZERO, bb:$dst)>;
def : Pat<(brcond (i32 (setuge RC:$lhs, RC:$rhs)), bb:$dst),
            (BEQ (SLTuOp RC:$lhs, RC:$rhs), ZERO, bb:$dst)>;
def : Pat<(brcond (i32 (setge RC:$lhs, immSExt16:$rhs)), bb:$dst),
            (BEQ (SLTiOp RC:$lhs, immSExt16:$rhs), ZERO, bb:$dst)>;
def : Pat<(brcond (i32 (setuge RC:$lhs, immSExt16:$rhs)), bb:$dst),
            (BEQ (SLTiOp RC:$lhs, immSExt16:$rhs), ZERO, bb:$dst)>;

def : Pat<(brcond (i32 (setle RC:$lhs, RC:$rhs)), bb:$dst),
            (BEQ (SLTOp RC:$rhs, RC:$lhs), ZERO, bb:$dst)>;
def : Pat<(brcond (i32 (setule RC:$lhs, RC:$rhs)), bb:$dst),
            (BEQ (SLTuOp RC:$rhs, RC:$lhs), ZERO, bb:$dst)>;

def : Pat<(brcond RC:$cond, bb:$dst),
            (BNEOp RC:$cond, ZEROReg, bb:$dst)>;
}

let Predicates = [NotCpu032II] in {
defm : BrcondPatsCmp<CPUREgs, JEQ, JNE, JLT, JGT, JLE, JGE, CMP, ZERO>;
}

let Predicates = [HasCpu032II] in {
defm : BrcondPatsSlt<CPUREgs, BEQ, BNE, SLT, SLTu, SLTi, SLTi, ZERO>;
}
// setcc for slt instruction
multiclass SeteqPatsSlt<RegisterClass RC, Instruction SLTiOp, Instruction XOROp,
                           Instruction SLTuOp, Register ZEROReg> {
// a == b
def : Pat<(seteq RC:$lhs, RC:$rhs),
            (SLTiOp (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 = [NotCpu032II] in {
defm : SeteqPatsCmp<CPUREgs>;
defm : SetltPatsCmp<CPUREgs>;
defm : SetlePatsCmp<CPUREgs>;
defm : SetgtPatsCmp<CPUREgs>;
defm : SetgePatsCmp<CPUREgs>;
}

let Predicates = [HasCpu032II] in {
defm : SeteqPatsSlt<CPUREgs, SLTi, XOR, SLTu, ZERO>;
defm : SetlePatsSlt<CPUREgs, SLT, SLTu>;
defm : SetgtPatsSlt<CPUREgs, SLT, SLTu>;
defm : SetgePatsSlt<CPUREgs, SLT, SLTu>;
defm : SetgeImmPatsSlt<CPUREgs, SLTi, SLTi>;
}

```

### Index/Chapter12\_2/Cpu0ISelDAGToDAG.cpp

```

/// Select instructions not customized! Used for
/// expanded, promoted and normal instructions
SDNode* Cpu0DAGToDAGISel::Select( SDNode *Node) {
    ...
    case ISD::SUBE:
    case ISD::ADDE: {
        ...
        const Cpu0TargetMachine &TM = getTargetMachine();
        const Cpu0Subtarget &Subtarget = TM.getSubtarget<Cpu0Subtarget>();
        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, VT);
            Carry = CurDAG->getMachineNode(Cpu0::ANDi, dl, VT,
                                              SDValue(StatusWord, 0), Constant1);
        }
        ...
    }
}

```

### [Index](#)/[Chapter12\\_2](#)/[Cpu0Subtarget.h](#)

```
class Cpu0Subtarget : public Cpu0GenSubtargetInfo {
    ...
    enum Cpu0ArchEnum {
        Cpu032I,
        Cpu032II,
        Cpu032III
    };
    ...
    bool hasCpu032I() const { return Cpu0ArchVersion >= Cpu032I; }
    bool hasCpu032II() const { return Cpu0ArchVersion == Cpu032II; }
    ...
}
```

As modified from above, the last Chapter instruction is work for cpu032I and the added instructions is for cpu032II. The llc will generate cpu032I cmp, jeq, ..., instructions when *llc -mcpu=cpu032I* while *llc -mcpu=cpu032II* will generate slt, beq when meet “if else”, “while” and “for” flow control statements.

### 12.2.3 Cpu0 Verilog language changes

#### [Index](#)/[cpu0\\_verilog](#)/[cpu0II.s.v](#)

```
'define CPU0II
// TRACE: Display the memory contents of the loaded program and data
//`define TRACE

`include "cpu0.v"
```

In addition to cpu0II.s.v, the “`ifdef CPU0II” in cpu0.v is added for extended instructions, slt, beq and bne.

### 12.2.4 Run the Cpu0II

Run Chapter12\_2/ with ch\_run\_backend.cpp to get result as below. It match the expect value as comment in ch\_run\_backend.cpp.

#### [Index](#)/[InputFiles](#)/[ch\\_run\\_backend.cpp](#)

```
#include "boot.cpp"

#include "print.h"

int test_math();
int test_div();
int test_local_pointer();
int test_andorxornot();
int test_setxx();
bool test_load_bool();
long long test_longlong();
int test_control();
int sum_i(int amount, ...);

int main()
```

```
{  
    int a = 0;  
    a = test_math();  
    print_integer(a); // a = 74  
    a = test_div();  
    print_integer(a); // a = 253  
    a = test_local_pointer();  
    print_integer(a); // a = 3  
    a = (int)test_load_bool();  
    print_integer(a); // a = 1  
    a = test_andorxornot(); // a = 14  
    print_integer(a);  
    a = test_setxx(); // a = 3  
    print_integer(a);  
    long long b = test_longlong(); // 0x8000000002  
    print_integer((int)(b >> 32)); // 393307  
    print_integer((int)b); // 16777222  
    a = test_control1();  
    print_integer(a); // a = 51  
    print_integer(2147483647); // test mod % (mult) from itoa.cpp  
    print_integer(-2147483648); // test mod % (multu) from itoa.cpp  
    a = sum_i(6, 0, 1, 2, 3, 4, 5);  
    print_integer(a); // a = 15  
  
    return a;  
}  
  
#include "print.cpp"  
  
void print1_integer(int x)  
{  
    asm("ld $at, 8($sp)");  
    asm("st $at, 28672($0)");  
  
    return;  
}  
  
#if 0  
// For instruction IO  
void print2_integer(int x)  
{  
    asm("ld $at, 8($sp)");  
    asm("outw $stat");  
    return;  
}  
#endif  
  
bool test_load_bool()  
{  
    int a = 1;  
  
    if (a < 0)  
        return false;  
  
    return true;  
}  
  
#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;
}

#include "ch4_1.cpp"
#include "ch4_3.cpp"
#include "ch4_5.cpp"
#include "ch7_1.cpp"
#include "ch7_4.cpp"
#include "ch8_1_1.cpp"

118-165-77-203:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=obj -stats
ch_run_backend.bc -o ch_run_backend.cpu0.o
=====
          ... Statistics Collected ...
=====
...
5 del-jmp      - Number of useless jmp deleted
...
118-165-77-203:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llvm-objdump -d ch_run_backend.cpu0.o | tail -n +6 | awk '{print /* " $1
" */\t" $2 " " $3 " " $4 " " $5 "\t/* " $6"\t" $7" " $8" " $9" " $10 "\t*/"}' >
../cpu0_verilog/redesign/cpu0.hex

JonathantekiiMac:InputFiles Jonathan$ cd ../cpu0_verilog/
JonathantekiiMac:redesign Jonathan$ iverilog -o cpu0IIs cpu0IIs.v
JonathantekiiMac:redesign Jonathan$ ./cpu0IIs
taskInterrupt(001)
74
253
3
1
14
3
393307
16777222
51
2147483647
-2147483648
15
RET to PC < 0, finished!

```

Run with ch8\_1\_1.cpp, it reduce some branch from pair instructions “CMP, JXX” to 1 single instruction ether is BEQ or BNE, as follows,

```

118-165-77-203:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -mcpu=cpu032II -relocation-model=static -filetype=asm
ch8_1_1.bc -o ch8_1_1.cpu0.s
118-165-77-203:InputFiles Jonathan$ cat ch8_1_1.cpu0.s
    .section .mdebug.abi32
    .previous
    .file "ch8_1_1.bc"
    .text
    .globl      _Z13test_control1v
    .align      2
    .type _Z13test_control1v,@function
    .ent _Z13test_control1v      # @_Z13test_control1v
_Z13test_control1v:
    .cfi_startproc
    .frame      $fp,48,$lr
    .mask       0x00000800,-4
    .set  noreorder
    .set  nomacro
# BB#0:                      # %entry
    addiu $sp, $sp, -48
$tmp3:
    .cfi_def_cfa_offset 48
    st   $fp, 44($sp)          # 4-byte Folded Spill
$tmp4:
    .cfi_offset 11, -4
    addu $fp, $sp, $zero
$tmp5:
    .cfi_def_cfa_register 11
    addiu $3, $zero, 0
    st   $3, 40($fp)
    addiu $2, $zero, 1
    st   $2, 36($fp)
    addiu $4, $zero, 2
    st   $4, 32($fp)
    addiu $4, $zero, 3
    st   $4, 28($fp)
    addiu $4, $zero, 4
    st   $4, 24($fp)
    addiu $4, $zero, 5
    st   $4, 20($fp)
    addiu $4, $zero, 6
    st   $4, 16($fp)
    addiu $4, $zero, 7
    st   $4, 12($fp)
    addiu $4, $zero, 8
    st   $4, 8($fp)
    addiu $4, $zero, 9
    st   $4, 4($fp)
    ld   $4, 40($fp)
    bne $4, $zero, $BB0_2
# BB#1:                      # %if.then
    ld   $4, 40($fp)
    addiu $4, $4, 1
    st   $4, 40($fp)
$BB0_2:                      # %if.end
    ld   $4, 36($fp)

```

```

        beq    $4, $zero, $BB0_4
# BB#3:                                     # %if.then2
        ld     $4, 36($fp)
        addiu $4, $4, 1
        st    $4, 36($fp)
$BB0_4:                                     # %if.end4
        ld     $4, 32($fp)
        slti  $4, $4, 1
        bne   $4, $zero, $BB0_6
# BB#5:                                     # %if.then6
        ld     $4, 32($fp)
        addiu $4, $4, 1
        st    $4, 32($fp)
$BB0_6:                                     # %if.end8
        ld     $4, 28($fp)
        slti  $4, $4, 0
        bne   $4, $zero, $BB0_8
# BB#7:                                     # %if.then10
        ld     $4, 28($fp)
        addiu $4, $4, 1
        st    $4, 28($fp)
$BB0_8:                                     # %if.end12
        ld     $4, 24($fp)
        addiu $5, $zero, -1
        slt   $5, $4, $4
        bne   $4, $zero, $BB0_10
# BB#9:                                     # %if.then14
        ld     $4, 24($fp)
        addiu $4, $4, 1
        st    $4, 24($fp)
$BB0_10:                                    # %if.end16
        ld     $4, 20($fp)
        slt   $3, $4, $4
        bne   $3, $zero, $BB0_12
# BB#11:                                    # %if.then18
        ld     $3, 20($fp)
        addiu $3, $3, 1
        st    $3, 20($fp)
$BB0_12:                                    # %if.end20
        ld     $3, 16($fp)
        slt   $2, $3, $3
        bne   $2, $zero, $BB0_14
# BB#13:                                    # %if.then22
        ld     $2, 16($fp)
        addiu $2, $2, 1
        st    $2, 16($fp)
$BB0_14:                                    # %if.end24
        ld     $2, 12($fp)
        slti  $2, $2, 1
        bne   $2, $zero, $BB0_16
# BB#15:                                    # %if.then26
        ld     $2, 12($fp)
        addiu $2, $2, 1
        st    $2, 12($fp)
$BB0_16:                                    # %if.end28
        ld     $2, 12($fp)
        ld     $3, 8($fp)
        slt   $2, $3, $2
    
```

```
    beq    $2, $zero, $BB0_18
# BB#17:                                # %if.then30
    ld     $2, 8($fp)
    addiu $2, $2, 1
    st    $2, 8($fp)
$BB0_18:                                # %if.end32
    ld     $2, 36($fp)
    ld    $3, 40($fp)
    beq    $3, $2, $BB0_20
# BB#19:                                # %if.then34
    ld     $2, 4($fp)
    addiu $2, $2, 1
    st    $2, 4($fp)
$BB0_20:                                # %if.end36
    ld     $2, 36($fp)
    ld    $3, 40($fp)
    addu  $2, $3, $2
    ld     $3, 32($fp)
    addu  $2, $2, $3
    ld     $3, 28($fp)
    addu  $2, $2, $3
    ld     $3, 24($fp)
    addu  $2, $2, $3
    ld     $3, 20($fp)
    addu  $2, $2, $3
    ld     $3, 16($fp)
    addu  $2, $2, $3
    ld     $3, 12($fp)
    addu  $2, $2, $3
    ld     $3, 8($fp)
    addu  $2, $2, $3
    ld     $3, 4($fp)
    addu  $2, $2, $3
    addu  $sp, $fp, $zero
    ld     $fp, 44($sp)          # 4-byte Folded Reload
    addiu $sp, $sp, 48
    ret   $lr
    .set  macro
    .set  reorder
    .end  _Z13test_controllv
$tmp6:
.size _Z13test_controllv, ($tmp6)-_Z13test_controllv
.cfi_endproc
```

The ch12\_3.cpp is written in assembly for AsmParser test. You can check if it will generate the obj.



# LLD FOR CPU0

This chapter add Cpu0 backend in lld. With this lld Cpu0 for ELF linker support, the program with global variables can be allocated in ELF file format layout. Meaning the relocation records of global variable can be solved. In addition, llvm-objdump driver is modified for support generate Hex file from ELF. With these two tools supported, the program with global variables exist in section.data and .rodata can be accessed and transferred to Hex file which feed to Verilog Cpu0 machine and run on your PC/Laptop.

LLD web site <sup>1</sup>. LLD install requirement on Linux <sup>2</sup>. In spite of the requirement, we only can build with gcc4.7 above (clang will fail) on Linux. If you run with Virtual Machine (VM), please keep your phisical memory size setting over 1GB to avoid link error with insufficient memory.

## 13.1 Install lld

LLD project is underdevelopment and can be compiled with c++11 standard (C++ 2011 year announced standard). Currently, we only know how to build lld with llvm on Linux platform or Linux VM. Please let us know if you know how to build it on iMac with Xcode. So, if you got iMac only, please install VM (such as Virtual Box). We porting lld Cpu0 at 2013/10/30, so please checkout the commit id 99a43d3b8f5cf86b333055a56220c6965fd9ece4(llvm) and 5d1737ac704352357fd28cfe3b2daf9aa308fb86(lld) which committed at 2013/10/30 as follows,

```
[Gamma@localhost test]$ mkdir lld
[Gamma@localhost test]$ cd lld
[Gamma@localhost lld]$ git clone http://llvm.org/git/llvm.git src
Cloning into 'src'...
remote: Counting objects: 780029, done.
remote: Compressing objects: 100% (153947/153947), done.
remote: Total 780029 (delta 637206), reused 764781 (delta 622170)
Receiving objects: 100% (780029/780029), 125.74 MiB | 243 KiB/s, done.
Resolving deltas: 100% (637206/637206), done.
[Gamma@localhost lld]$ cd src/
```

```
[Gamma@localhost src]$ git checkout 99a43d3b8f5cf86b333055a56220c6965fd9ece4
Note: checking out '99a43d3b8f5cf86b333055a56220c6965fd9ece4'.
```

You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may do so (now or later) by using -b with the checkout command again. Example:

---

<sup>1</sup> <http://lld.llvm.org/>

<sup>2</sup> [http://lld.llvm.org/getting\\_started.html#on-unix-like-systems](http://lld.llvm.org/getting_started.html#on-unix-like-systems)

```
git checkout -b new_branch_name

HEAD is now at da44b4f... CMake: polish the Windows packaging rules

[Gamma@localhost src]$ cd tools/
[Gamma@localhost tools]$ git clone http://llvm.org/git/lld.git lld
...
Resolving deltas: 100% (6422/6422), done.
[Gamma@localhost tools]$ cd lld/
[Gamma@localhost lld]$ git checkout 5d1737ac704352357fd28cfe3b2daf9aa308fb86
Note: checking out '5d1737ac704352357fd28cfe3b2daf9aa308fb86'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.
```

If you want to create a new branch to retain commits you create, you may **do** so (now or later) by using `-b` with the checkout `command` again. Example:

```
git checkout -b new_branch_name

HEAD is now at 014d684... [PECOFF] Handle "—" option explicitly
```

Next, update llvm 2013/10/30 source code to support Cpu0 as follows,

```
[Gamma@localhost src]$ pwd
/home/Gamma/test/lld/src
[Gamma@localhost src]$ cp -rf ~/test/lbd/docs/BackendTutorial/
lbdex/3.4_1030_src_files_modify/modify/src/* .
[Gamma@localhost src]$ grep -R "cpu0" include/
include/llvm/ADT/Triple.h:#undef cpu0
include/llvm/ADT/Triple.h:    cpu0,      // For Tutorial Backend Cpu0
include/llvm/ADT/Triple.h:    cpu0el,
include/llvm/Object/ELFObjectFile.h:           Triple::cpu0el : Triple::cpu0;
include/llvm/Support/ELF.h:  EF_CPU0_ARCH_32R2 = 0x70000000, // cpu032r2
include/llvm/Support/ELF.h:  EF_CPU0_ARCH_64R2 = 0x80000000, // cpu064r2
[Gamma@localhost src]$ cd lib/Target/
[Gamma@localhost Target]$ ls
AArch64          MSP430           TargetJITInfo.cpp
ARM              NPVTX            TargetLibraryInfo.cpp
CMakeLists.txt   PowerPC          TargetLoweringObjectFile.cpp
CppBackend       R600             TargetMachineC.cpp
Hexagon          README.txt       TargetMachine.cpp
LLVMBuild.txt   Sparc            TargetSubtargetInfo.cpp
Makefile         SystemZ          X86
Mangler.cpp     Target.cpp       XCore
Mips             TargetIntrinsicInfo.cpp
[Gamma@localhost Target]$ mkdir Cpu0
[Gamma@localhost Target]$ cd Cpu0/
[Gamma@localhost Cpu0]$ cp -rf ~/test/lbd/docs/BackendTutorial/
lbdex/3.4_0830_Chapter12_2/* .
[Gamma@localhost Cpu0]$ ls
AsmParser          Cpu0InstrInfo.h      Cpu0SelectionDAGInfo.h
CMakeLists.txt     Cpu0InstrInfo.td     Cpu0Subtarget.cpp
Cpu0AnalyzeImmediate.cpp Cpu0ISelDAGToDAG.cpp Cpu0Subtarget.h
Cpu0AnalyzeImmediate.h Cpu0ISelLowering.cpp Cpu0TargetMachine.cpp
Cpu0AsmPrinter.cpp Cpu0ISelLowering.h  Cpu0TargetMachine.h
Cpu0AsmPrinter.h   Cpu0MachineFunction.cpp Cpu0TargetObjectFile.cpp
```

Cpu0CallingConv.td	Cpu0MachineFunction.h	Cpu0TargetObjectFile.h
Cpu0DelUselessJMP.cpp	Cpu0MCInstLower.cpp	Cpu0.td
Cpu0EmitGPRestore.cpp	Cpu0MCInstLower.h	Disassembler
Cpu0FrameLowering.cpp	Cpu0RegisterInfo.cpp	InstPrinter
Cpu0FrameLowering.h	Cpu0RegisterInfo.h	LLVMBuild.txt
Cpu0.h	Cpu0RegisterInfo.td	MCTargetDesc
Cpu0InstrFormats.td	Cpu0Schedule.td	TargetInfo
Cpu0InstrInfo.cpp	Cpu0SelectionDAGInfo.cpp	

Next, copy lld Cpu0 architecture ELF support as follows,

```
[Gamma@localhost Cpu0]$ cd ../../../../tools/lld/lib/ReaderWriter/ELF/
[Gamma@localhost ELF]$ pwd
/home/Gamma/test/lld/src/tools/lld/lib/ReaderWriter/ELF
[Gamma@localhost ELF]$ cp -rf ~/test/lbd/docs/BackendTutorial/
1bdex/Cpu0_lld_1030/Cpu0 .
[Gamma@localhost ELF]$ cp -f ~/test/lbd/docs/BackendTutorial/
1bdex/Cpu0_lld_1030/CMakeLists.txt .
[Gamma@localhost ELF]$ cp -f ~/test/lbd/docs/BackendTutorial/
1bdex/Cpu0_lld_1030/ELFLinkingContext.cpp .
[Gamma@localhost ELF]$ cp -f ~/test/lbd/docs/BackendTutorial/
1bdex/Cpu0_lld_1030/Targets.h .
[Gamma@localhost ELF]$ cp -f ~/test/lbd/docs/BackendTutorial/
1bdex/Cpu0_lld_1030/Resolver.cpp ../../Core/.
```

Finally, update llvm-objdump to support convert ELF file to Hex file as follows,

```
[Gamma@localhost ELF]$ cd ../../../../llvm-objdump/
[Gamma@localhost llvm-objdump]$ pwd
/home/Gamma/test/lld/src/tools/llvm-objdump
[Gamma@localhost llvm-objdump]$ cp -rf ~/test/lbd/docs/BackendTutorial/
1bdex/llvm-objdump/* .
```

Now, build llvm/lld with Cpu0 support as follows,

```
[Gamma@localhost cmake_debug_build]$ cmake -DCMAKE_CXX_COMPILER=g++ -
DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_FLAGS=-std=c++11 -DCMAKE_BUILD_TYPE=Debug
-G "Unix Makefiles" ../src
-- The C compiler identification is GNU 4.7.2
-- The CXX compiler identification is GNU 4.7.2
...
-- Targeting Cpu0
...
-- Configuring done
-- Generating done
-- Build files have been written to: /home/Gamma/test/lld/cmake_debug_build
```

## 13.2 Cpu0 lld souce code

The code added on lld to support Cpu0 ELF as follows,

### 1bdex/Cpu0\_lld\_1030/CMakeLists.txt

```
target_link_libraries(lldELF
  ...
  
```

```
lldCpu0ELFTarget
)
```

### lbdex/Cpu0\_lld\_1030/ELFLinkingContext.cpp

```
uint16_t ELFLinkingContext::getOutputMachine() const {
    switch (getTriple().getArch()) {
    ...
    case llvm::Triple::cpu0:
        return llvm::ELF::EM_CPU0;
    ...
}
}
```

### lbdex/Cpu0\_lld\_1030/Targets.h

```
#include "Cpu0/Cpu0Target.h"
```

### lbdex/Cpu0\_lld\_1030/Resolver.cpp

```
bool Resolver::checkUndefines(bool final) {
    ...
    if (_context.printRemainingUndefines() ) {
        if (undefAtom->name() == "_gp_disp") { // cschen debug
            foundUndefines = false;
            continue;
        }
        ...
    }
    ...
}
```

### lbdex/Cpu0\_lld\_1030/Cpu0/CMakeLists.txt

```
add_lld_library(lldCpu0ELFTarget
    Cpu0LinkingContext.cpp
    Cpu0TargetHandler.cpp
    Cpu0RelocationHandler.cpp
    Cpu0RelocationPass.cpp
)

target_link_libraries(lldCpu0ELFTarget
    lldCore
)
```

### lbdex/Cpu0\_lld\_1030/Cpu0/Cpu0LinkingContext.h

```
===== lib/ReaderWriter/ELF/Cpu0/Cpu0LinkingContext.h =====
//
//          The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
=====

#ifndef LLD_READER_WRITER_ELF_CPU0_LINKER_CONTEXT_H
#define LLD_READER_WRITER_ELF_CPU0_LINKER_CONTEXT_H

#include "Cpu0TargetHandler.h"

#include "lld/ReaderWriter/ELFLinkingContext.h"

#include "llvm/Object/ELF.h"
#include "llvm/Support/ELF.h"

namespace lld {
namespace elf {
/// \brief cpu0 internal references.
enum {
    /// \brief The 32 bit index of the relocation in the got this reference refers
    /// to.
    LLD_R_CPU0_GOTRELINDEX = 1024,
};

class Cpu0LinkingContext LLVM_FINAL : public ELFLinkingContext {
public:
    Cpu0LinkingContext(llvm::Triple triple)
        : ELFLinkingContext(triple, std::unique_ptr<TargetHandlerBase>(
            new Cpu0TargetHandler(*this))) {}

    virtual bool isLittleEndian() const { return false; }

    virtual void addPasses(PassManager &);

    // Cpu0 run begin from address 0 while X86 from 0x400000
    virtual uint64_t getBaseAddress() const {
        if (_baseAddress == 0)
            return 0x000000;
        return _baseAddress;
    }

    virtual bool isDynamicRelocation(const DefinedAtom &,
                                    const Reference &r) const {
        switch (r.kind()) {
        case llvm::ELF::R_CPU0_GLOB_DAT:
            return true;
        default:
            return false;
        }
    }

    virtual bool isPLTRelocation(const DefinedAtom &,
                                const Reference &r) const {
        switch (r.kind()) {
```

```

    case llvm::ELF::R_CPU0_JUMP_SLOT:
    case llvm::ELF::R_CPU0_RELGOT:
        return true;
    default:
        return false;
    }
}

/// \brief Cpu0 has two relative relocations
/// a) for supporting IFUNC - R_CPU0_RELGOT
/// b) for supporting relative relocs - R_CPU0_RELATIVE
virtual bool isRelativeReloc(const Reference &r) const {
    switch (r.kind()) {
    case llvm::ELF::R_CPU0_RELGOT:
        return true;
    default:
        return false;
    }
}

/// \brief Create Internal files for Init/Fini
bool createInternalFiles(std::vector<std::unique_ptr<File> > &) const;

virtual ErrorOr<Reference::Kind> relocKindFromString(StringRef str) const;
virtual ErrorOr<std::string> stringFromRelocKind(Reference::Kind kind) const;

bool isStaticExecutable() const { return _isStaticExecutable; }

};

} // end namespace elf
} // end namespace lld

#endif

```

### Ibdex/Cpu0\_lld\_1030/Cpu0/Cpu0LinkingContext.cpp

```

===== lib/ReaderWriter/ELF/Cpu0/Cpu0LinkingContext.cpp =====
//
//          The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
=====

#include "Cpu0LinkingContext.h"

#include "lld/Core/File.h"
#include "lld/Core/Instrumentation.h"

#include "llvm/ADT/ArrayRef.h"
#include "llvm/ADT/StringSwitch.h"

#include "Atoms.h"
#include "Cpu0RelocationPass.h"

```

```
using namespace lld;

using namespace lld::elf;

namespace {
using namespace llvm::ELF;
const uint8_t cpu0InitFiniAtomContent[8] = { 0 };

// Cpu0_64InitFini Atom
class Cpu0InitAtom : public InitFiniAtom {
public:
    Cpu0InitAtom(const File &f, StringRef function)
        : InitFiniAtom(f, ".init_array") {
#ifndef NDEBUG
        _name = "__init_fn_";
        _name += function;
#endif
    }
    virtual ArrayRef<uint8_t> rawContent() const {
        return ArrayRef<uint8_t>(cpu0InitFiniAtomContent, 8);
    }
    virtual Alignment alignment() const { return Alignment(3); }
};

class Cpu0FiniAtom : public InitFiniAtom {
public:
    Cpu0FiniAtom(const File &f, StringRef function)
        : InitFiniAtom(f, ".fini_array") {
#ifndef NDEBUG
        _name = "__fini_fn_";
        _name += function;
#endif
    }
    virtual ArrayRef<uint8_t> rawContent() const {
        return ArrayRef<uint8_t>(cpu0InitFiniAtomContent, 8);
    }
    virtual Alignment alignment() const { return Alignment(3); }
};

class Cpu0InitFiniFile : public SimpleFile {
public:
    Cpu0InitFiniFile(const ELFLinkingContext &context)
        : SimpleFile(context, "command line option -init/-fini"), _ordinal(0) {}

    void addInitFunction(StringRef name) {
        Atom *initFunctionAtom = new (_allocator) SimpleUndefinedAtom(*this, name);
        Cpu0InitAtom *initAtom =
            (new (_allocator) Cpu0InitAtom(*this, name));
        initAtom->addReference(llvm::R_CPU0_32, 0, initFunctionAtom, 0);
        initAtom->setOrdinal(_ordinal++);
        addAtom(*initFunctionAtom);
        addAtom(*initAtom);
    }

    void addFiniFunction(StringRef name) {
        Atom *finiFunctionAtom = new (_allocator) SimpleUndefinedAtom(*this, name);
        Cpu0FiniAtom *finiAtom =

```

```

        (new (_allocator) Cpu0FiniAtom(*this, name));
finiAtom->addReference(llvm::ELF::R_CPU0_32, 0, finiFunctionAtom, 0);
finiAtom->setOrdinal(_ordinal++);
addAtom(*finiFunctionAtom);
addAtom(*finiAtom);
}

private:
    llvm::BumpPtrAllocator _allocator;
    uint64_t _ordinal;
};

} // end anon namespace

void elf::Cpu0LinkingContext::addPasses(PassManager &pm) {
    auto pass = createCpu0RelocationPass(*this);
    if (pass)
        pm.add(std::move(pass));
    ELFLinkingContext::addPasses(pm);
}

bool elf::Cpu0LinkingContext::createInternalFiles(
    std::vector<std::unique_ptr<File> > &result) const {
    ELFLinkingContext::createInternalFiles(result);
    std::unique_ptr<Cpu0InitFiniFile> initFiniFile(
        new Cpu0InitFiniFile(*this));
    for (auto ai : initFunctions())
        initFiniFile->addInitFunction(ai);
    for (auto ai:finiFunctions())
        initFiniFile->addFiniFunction(ai);
    result.push_back(std::move(initFiniFile));
    return true;
}

#define LLD_CASE(name) .Case(#name, llvm::ELF::name)

ErrorOr<Reference::Kind>
elf::Cpu0LinkingContext::relocKindFromString(StringRef str) const {
    int32_t ret = llvm::StringSwitch<int32_t>(str)
        .Case(R_CPU0_NONE)
        .Case(R_CPU0_24)
        .Case(R_CPU0_32)
        .Case(R_CPU0_HI16)
        .Case(R_CPU0_LO16)
        .Case(R_CPU0_GPREL16)
        .Case(R_CPU0_LITERAL)
        .Case(R_CPU0_GOT16)
        .Case(R_CPU0_PC24)
        .Case(R_CPU0_CALL16)
        .Case(R_CPU0_JUMP_SLOT)
        .Case("LLD_R_CPU0_GOTRELINDEX", LLD_R_CPU0_GOTRELINDEX)
        .Default(-1);

    if (ret == -1)
        return make_error_code(YamlReaderError::illegal_value);
    return ret;
}

```

```
#undef LLD_CASE

#define LLD_CASE(name) case llvm::ELF::name: return std::string(#name);

ErrorOr<std::string>
elf::Cpu0LinkingContext::stringFromRelocKind(Reference::Kind kind) const {
    switch (kind) {
        LLD_CASE(R_CPU0_NONE)
        LLD_CASE(R_CPU0_24)
        LLD_CASE(R_CPU0_32)
        LLD_CASE(R_CPU0_HI16)
        LLD_CASE(R_CPU0_LO16)
        LLD_CASE(R_CPU0_GPREL16)
        LLD_CASE(R_CPU0_LITERAL)
        LLD_CASE(R_CPU0_GOT16)
        LLD_CASE(R_CPU0_PC24)
        LLD_CASE(R_CPU0_CALL16)
        LLD_CASE(R_CPU0_JUMP_SLOT)
        case LLD_R_CPU0_GOTRELINDEX:
            return std::string("LLD_R_CPU0_GOTRELINDEX");
    }

    return make_error_code(YamlReaderError::illegal_value);
}
```

### lbdex/Cpu0\_lld\_1030/Cpu0/Cpu0RelocationHandler.h

```
===== lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationHandler.h
=====//
// The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//=====

#ifndef Cpu0_RELOCATION_HANDLER_H
#define Cpu0_RELOCATION_HANDLER_H

#include "Cpu0TargetHandler.h"

namespace lld {
namespace elf {
typedef llvm::object::ELFType<llvm::support::big, 4, false> Cpu0ELFType;
class Cpu0LinkingContext;

class Cpu0TargetRelocationHandler LLVM_FINAL
    : public TargetRelocationHandler<Cpu0ELFType> {
public:
    Cpu0TargetRelocationHandler(const Cpu0LinkingContext &context)
        : _tlsSize(0), _context(context) {}

    virtual ErrorOr<void> applyRelocation(ELFWriter &, llvm::FileOutputBuffer &,
                                           const lld::AtomLayout &,
                                           const Reference &) const;
```

```

virtual int64_t relocAddend(const Reference &) const;

private:
    // Cached size of the TLS segment.
    mutable uint64_t _tlsSize;
    const Cpu0LinkingContext &_context;
};

} // end namespace elf
} // end namespace lld

#endif

```

### lbdex/Cpu0\_lld\_1030/Cpu0/Cpu0RelocationHandler.cpp

```

//===== lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationHandler.cpp =====//
//
//                                     The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//

```

```

#include "Cpu0TargetHandler.h"
#include "Cpu0LinkingContext.h"
#include "llvm/Object/ObjectFile.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/system_error.h"

using namespace lld;
using namespace elf;
using namespace llvm;
using namespace object;

static bool error(error_code ec) {
    if (!ec) return false;

    outs() << "Cpu0RelocationHandler.cpp : error reading file: "
        << ec.message() << ".\n";
    outs().flush();
    return true;
}

namespace {
/// \brief R_CPU0_HI16 - word64: (S + A) >> 16
void relocHI16(uint8_t *location, uint64_t P, uint64_t S, int64_t A) {
    // Don't know why A, ref.addend(), = 9
    uint32_t result = (uint32_t)(S >> 16);
    *reinterpret_cast<llvm::support::ubig32_t *>(location) =
        result |
        (uint32_t) * reinterpret_cast<llvm::support::ubig32_t *>(location);
}

void relocLO16(uint8_t *location, uint64_t P, uint64_t S, uint64_t A) {
    // Don't know why A, ref.addend(), = 9

```

```

    uint32_t result = (uint32_t)(S & 0x0000ffff);
    *reinterpret_cast<llvm::support::ubig32_t *>(location) =
        result |
        (uint32_t) * reinterpret_cast<llvm::support::ubig32_t *>(location);
}

/// \brief R_CPU0_GOT16 - word32: S
void relocGOT16(uint8_t *location, uint64_t P, uint64_t S, int64_t A) {
    uint32_t result = (uint32_t)(S);
    *reinterpret_cast<llvm::support::ubig32_t *>(location) =
        result |
        (uint32_t) * reinterpret_cast<llvm::support::ubig32_t *>(location);
}

/// \brief R_CPU0_PC24 - word32: S + A - P
void relocPC24(uint8_t *location, uint64_t P, uint64_t S, int64_t A) {
    uint32_t result = (uint32_t)(S - P);
    uint32_t machinecode = (uint32_t) *
        reinterpret_cast<llvm::support::ubig32_t *>(location);
    uint32_t opcode = (machinecode & 0xffff0000);
    uint32_t offset = (machinecode & 0x00ffff00);
    *reinterpret_cast<llvm::support::ubig32_t *>(location) =
        (((result + offset) & 0x00ffff00) | opcode);
}

/// \brief R_CPU0_32 - word24: S
void reloc24(uint8_t *location, uint64_t P, uint64_t S, int64_t A) {
    int32_t addr = (uint32_t)(S & 0x00ffff00);
    uint32_t machinecode = (uint32_t) *
        reinterpret_cast<llvm::support::ubig32_t *>(location);
    uint32_t opcode = (machinecode & 0xffff0000);
    *reinterpret_cast<llvm::support::ubig32_t *>(location) =
        (opcode | addr);
    // TODO: Make sure that the result zero extends to the 64bit value.
}

/// \brief R_CPU0_32 - word32: S
void reloc32(uint8_t *location, uint64_t P, uint64_t S, int64_t A) {
    int32_t result = (uint32_t)(S);
    *reinterpret_cast<llvm::support::ubig32_t *>(location) =
        result |
        (uint32_t) * reinterpret_cast<llvm::support::ubig32_t *>(location);
    // TODO: Make sure that the result zero extends to the 64bit value.
}
} // end anon namespace

int64_t Cpu0TargetRelocationHandler::relocAddend(const Reference &ref) const {
    switch (ref.kind()) {
    case R_CPU0_PC24:
        return 4;
    default:
        return 0;
    }
    return 0;
}

#endif DLINKER
class Cpu0SoPlt {

```

```

private:
    uint32_t funAddr[100];
    int funAddrSize = 0;
public:
    void createFunAddr(const Cpu0LinkingContext &context,
                        llvm::FileOutputBuffer &buf);
    // Return function index, 1: 1st function appear on section .text of .so.
    // 2: 2nd function ...
    // For example: 3 functions _Z2laii, _Z3fooii and _Z3barv. 1: is _Z2laii
    // 2 is _Z3fooii, 3: is _Z3barv.
    int getDynFunIndexByTargetAddr(uint64_t fAddr);
};

void Cpu0SoPlt::createFunAddr(const Cpu0LinkingContext &context,
                               llvm::FileOutputBuffer &buf) {
    auto dynsymSection = context.getTargetHandler<Cpu0ELFType>().targetLayout() .
        findOutputSection(".dynsym");
    uint64_t dynsymFileOffset, dynsymSize;
    if (dynsymSection) {
        dynsymFileOffset = dynsymSection->fileOffset();
        dynsymSize = dynsymSection->memSize();
        uint8_t *atomContent = buf.getBufferStart() + dynsymFileOffset;
        for (uint64_t i = 4; i < dynsymSize; i += 16) {
            funAddr[funAddrSize] =
                *reinterpret_cast<llvm::support::ubig32_t*>((uint32_t*) (atomContent + i));
            funAddrSize++;
        }
    }
}

int Cpu0SoPlt::getDynFunIndexByTargetAddr(uint64_t fAddr) {
    for (int i = 0; i < funAddrSize; i++) {
        // Below statement fix the issue that both __tls_get_addr and first
        // function has the same file offset 0 issue.
        if (i < (funAddrSize-1) && funAddr[i] == funAddr[i+1])
            continue;

        if (fAddr == funAddr[i]) {
            return i;
        }
    }
    return -1;
}

Cpu0SoPlt cpu0SoPlt;
#endif // DLINKER

ErrorOr<void> Cpu0TargetRelocationHandler::applyRelocation(
    ELFWriter &writer, llvm::FileOutputBuffer &buf, const lld::AtomLayout &atom,
    const Reference &ref) const {
#ifdef DLINKER
    static bool firstTime = true;
    std::string soName("libfoobar.cpu0.so");
    int idx = 0;
    if (firstTime) {
        if (_context.getOutputELFType() == llvm::ELF::ET_DYN) {
            cpu0SoPlt.createFunAddr(_context, buf);
    }
}

```

```

    }
    else if (_context.getOutputELFType() == llvm::ELF::ET_EXEC &&
        !_context.isStaticExecutable()) {
        cpu0SoPlt.createFunAddr(_context, buf);
    }
    firstTime = false;
}
#endif // DLINKER
uint8_t *atomContent = buf.getBufferStart() + atom._fileOffset;
uint8_t *location = atomContent + ref.offsetInAtom();
uint64_t targetVAddress = writer.addressOfAtom(ref.target());
uint64_t relocVAddress = atom._virtualAddr + ref.offsetInAtom();
#if 1 // For case R_CPU0_GOT16:
// auto gotAtomIter = _context.getTargetHandler<Cpu0ELFType>().targetLayout() .
//     findAbsoluteAtom("_GLOBAL_OFFSET_TABLE_");
// uint64_t globalOffsetTableAddress = writer.addressOfAtom(*gotAtomIter);
// .got.plt start from _GLOBAL_OFFSET_TABLE_
auto gotpltSection = _context.getTargetHandler<Cpu0ELFType>().targetLayout() .
    findOutputSection(".got.plt");
uint64_t gotPltFileOffset;
if (gotpltSection)
    gotPltFileOffset = gotpltSection->fileOffset();
else
    gotPltFileOffset = 0;
#endif

switch (ref.kind()) {
case R_CPU0_NONE:
    break;
case R_CPU0_HI16:
    relocHI16(location, relocVAddress, targetVAddress, ref.addend());
    break;
case R_CPU0_LO16:
    relocLO16(location, relocVAddress, targetVAddress, ref.addend());
    break;
#if 0 // Not support yet
case R_CPU0_GOT16:
if 1
    idx = cpu0SoPlt.getDynFunIndexByTargetAddr(targetVAddress);
    relocGOT16(location, relocVAddress, idx, ref.addend());
else
    relocGOT16(location, relocVAddress, (targetVAddress - gotPltFileOffset),
        ref.addend());
endif
    break;
endif
case R_CPU0_PC24:
    relocPC24(location, relocVAddress, targetVAddress, ref.addend());
    break;
#endif // DLINKER
case R_CPU0_CALL16:
// offset at _GLOBAL_OFFSET_TABLE_ and $gp point to _GLOBAL_OFFSET_TABLE_.
idx = cpu0SoPlt.getDynFunIndexByTargetAddr(targetVAddress);
reloc32(location, relocVAddress, idx*0x04+16, ref.addend());
break;
#endif // DLINKER
case R_CPU0_24:
    reloc24(location, relocVAddress, targetVAddress, ref.addend());

```

```

break;
case R_CPU0_32:
    reloc32(location, relocVAddress, targetVAddress, ref.addend());
    break;

    // Runtime only relocations. Ignore here.
case R_CPU0_JUMP_SLOT:
    break;
case lld::Reference::kindLayoutAfter:
case lld::Reference::kindLayoutBefore:
case lld::Reference::kindInGroup:
    break;

default: {
    std::string str;
    llvm::raw_string_ostream s(str);
    auto name = _context.stringFromRelocKind(ref.kind());
    s << "Unhandled relocation: " << atom._atom->file().path() << ":" 
    << atom._atom->name() << "@" << ref.offsetInAtom() << " "
    << (name ? *name : "<unknown>") << " (" << ref.kind() << ")";
    s.flush();
    llvm_unreachable(str.c_str());
}
}

return error_code::success();
}

```

### Ibdex/Cpu0\_Lld\_1030/Cpu0/Cpu0RelocationPass.h

```

===== lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationPass.h =====
//
//          The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
=====

/// \file
/// \brief Declares the relocation processing pass for cpu0. This includes
/// GOT and PLT entries, TLS, COPY, and ifunc.
///
=====

#ifndef LLD_READER_WRITER_ELF_CPU0_RELOCATION_PASS_H
#define LLD_READER_WRITER_ELF_CPU0_RELOCATION_PASS_H

#include <memory>

namespace lld {
class Pass;
namespace elf {
class Cpu0LinkingContext;

/// \brief Create cpu0 relocation pass for the given linking context.

```

```
std::unique_ptr<Pass>
createCpu0RelocationPass(const Cpu0LinkingContext &);
}

}

#endif
```

### **lbdex/Cpu0\_lld\_1030/Cpu0/Cpu0RelocationPass.cpp**

```
===== lib/ReaderWriter/ELF/Cpu0/Cpu0RelocationPass.cpp =====
//
// The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
=====

/// \file
/// \brief Defines the relocation processing pass for Cpu0. This includes
/// GOT and PLT entries, TLS, and ifunc.
///
/// This also includes additional behavior that gnu-ld and gold implement but
/// which is not specified anywhere.
///
=====

#include "Cpu0RelocationPass.h"

#include "lld/ReaderWriter/Simple.h"

#include "llvm/ADT/DenseMap.h"

#include "Atoms.h"
#include "Cpu0LinkingContext.h"

using namespace lld;
using namespace lld::elf;
using namespace llvm::ELF;

namespace {

// .plt value (entry 0)
const uint8_t cpu0BootAtomContent[16] = {
  0x36, 0xff, 0xff, 0xfc, // jmp _start
  0x36, 0x00, 0x00, 0x04, // jmp 4
  0x36, 0x00, 0x00, 0x04, // jmp 4
  0x36, 0xff, 0xff, 0xfc // jmp -4
};

#ifdef DLINKER
// .got values
const uint8_t cpu0GotAtomContent[16] = { 0 };

// .plt value (entry 0)
const uint8_t cpu0Plt0AtomContent[16] = {
  0x02, 0xeb, 0x00, 0x04, // st $lr, $zero, reloc-index ($gp)
```

```

0x02, 0xcb, 0x00, 0x08, // st $fp, $zero, reloc-index ($gp)
0x02, 0xdb, 0x00, 0x0c, // st $sp, $zero, reloc-index ($gp)
0x36, 0xff, 0xff, 0xfc // jmp dynamic_linker
};

// .plt values (other entries)
const uint8_t cpu0PltAtomContent[16] = {
    0x09, 0x60, 0x00, 0x00, // addiu $t9, $zero, reloc-index (=.dynsym_index)
    0x02, 0x6b, 0x00, 0x00, // st $t9, $zero, reloc-index ($gp)
    0x01, 0x6b, 0x00, 0x10, // ld $t9, 0x10($gp) (0x10($gp) point to plt0
    0x3c, 0x60, 0x00, 0x00 // ret $t9 // jump to Cpu0.Stub
};
#endif // DLINKER

/// boot record
class Cpu0BootAtom : public PLT0Atom {
public:
    Cpu0BootAtom(const File &f) : PLT0Atom(f) {
#ifndef NDEBUG
        _name = ".PLT0";
#endif
    }
    virtual ArrayRef<uint8_t> rawContent() const {
        return ArrayRef<uint8_t>(cpu0BootAtomContent, 16);
    }
};

#endif DLINKER
/// \brief Atoms that are used by Cpu0 dynamic linking
class Cpu0GOTAtom : public GOTAtom {
public:
    Cpu0GOTAtom(const File &f, StringRef secName) : GOTAtom(f, secName) {}

    virtual ArrayRef<uint8_t> rawContent() const {
        return ArrayRef<uint8_t>(cpu0GotAtomContent, 16);
    }
};

class Cpu0PLT0Atom : public PLT0Atom {
public:
    Cpu0PLT0Atom(const File &f) : PLT0Atom(f) {
#ifndef NDEBUG
        _name = ".PLT0";
#endif
    }
    virtual ArrayRef<uint8_t> rawContent() const {
        return ArrayRef<uint8_t>(cpu0Plt0AtomContent, 16);
    }
};

class Cpu0PLTAtom : public PLTAtom {
public:
    Cpu0PLTAtom(const File &f, StringRef secName) : PLTAtom(f, secName) {}

    virtual ArrayRef<uint8_t> rawContent() const {
        return ArrayRef<uint8_t>(cpu0PltAtomContent, 16);
    }
};

```

```
#endif // DLINKER

class ELFPassFile : public SimpleFile {
public:
    ELFPassFile(const ELFLinkingContext &eti) : SimpleFile(eti, "ELFPassFile") {
        setOrdinal(eti.getNextOrdinalAndIncrement());
    }

    llvm::BumpPtrAllocator _alloc;
};

/// \brief CRTP base for handling relocations.
template <class Derived> class RelocationPass : public Pass {
    /// \brief Handle a specific reference.
    void handleReference(const DefinedAtom &atom, const Reference &ref) {
        switch (ref.kind()) {
        case R_CPU0_CALL16:
            static_cast<Derived *>(this)->handlePLT32(ref);
            break;

        case R_CPU0_PC24:
            static_cast<Derived *>(this)->handlePlain(ref);
            break;
        }
    }
}

protected:
#ifndef DLINKER
    /// \brief get the PLT entry for a given IFUNC Atom.
    ///
    /// If the entry does not exist. Both the GOT and PLT entry is created.
    const PLTAtom *getIFUNCPLTEEntry(const DefinedAtom *da) {
        auto plt = _pltMap.find(da);
        if (plt != _pltMap.end())
            return plt->second;
        auto ga = new (_file._alloc) Cpu0GOTAtom(_file, ".got.plt");
        ga->addReference(R_CPU0_RELGOT, 0, da, 0);
        auto pa = new (_file._alloc) Cpu0PLTAtom(_file, ".plt");
        pa->addReference(R_CPU0_PC24, 2, ga, -4);
        #ifndef NDEBUG
            ga->_name = "__got_ifunc_";
            ga->_name += da->name();
            pa->_name = "__plt_ifunc_";
            pa->_name += da->name();
        #endif
        _gotMap[da] = ga;
        _pltMap[da] = pa;
        _gotVector.push_back(ga);
        _pltVector.push_back(pa);
        return pa;
    }
#endif // DLINKER

    /// \brief Redirect the call to the PLT stub for the target IFUNC.
    ///
    /// This create a PLT and GOT entry for the IFUNC if one does not exist. The
    /// GOT entry and a IRELATIVE relocation to the original target resolver.
    ErrorOr<void> handleIFUNC(const Reference &ref) {
```

```

    auto target = dyn_cast_or_null<const DefinedAtom>(ref.target());
#endif // DLINKER
    if (target && target->contentType() == DefinedAtom::typeResolver)
        const_cast<Reference &>(ref).setTarget(getIFUNCPLTEEntry(target));
#endif // DLINKER
    return error_code::success();
}

#ifndef DLINKER
/// \brief Create a GOT entry for the TP offset of a TLS atom.
const GOTAtom *getGOTPOFF(const Atom *atom) {
    auto got = _gotMap.find(atom);
    if (got == _gotMap.end()) {
        auto g = new (_file._alloc) Cpu0GOTAtom(_file, ".got");
        g->addReference(R_CPU0_TLS_TPREL32, 0, atom, 0);
#ifndef NDEBUG
        g->_name = "__got_tls_";
        g->_name += atom->name();
#endif
        _gotMap[atom] = g;
        _gotVector.push_back(g);
        return g;
    }
    return got->second;
}

/// \brief Create a GOT entry containing 0.
const GOTAtom *getNullGOT() {
    if (!_null) {
        _null = new (_file._alloc) Cpu0GOTAtom(_file, ".got.plt");
#ifndef NDEBUG
        _null->_name = "__got_null";
#endif
        return _null;
    }
}

const GOTAtom *getGOT(const DefinedAtom *da) {
    auto got = _gotMap.find(da);
    if (got == _gotMap.end()) {
        auto g = new (_file._alloc) Cpu0GOTAtom(_file, ".got");
        g->addReference(R_CPU0_32, 0, da, 0);
#ifndef NDEBUG
        g->_name = "__got_";
        g->_name += da->name();
#endif
        _gotMap[da] = g;
        _gotVector.push_back(g);
        return g;
    }
    return got->second;
}
#endif // DLINKER

public:
    RelocationPass(const ELFLinkingContext &ctx)
        : _file(ctx), _ctx(ctx), _null(nullptr), _PLT0(nullptr), _got0(nullptr),
        _boot(new Cpu0BootAtom(_file)) {}

```

```

/// \brief Do the pass.
///
/// The goal here is to first process each reference individually. Each call
/// to handleReference may modify the reference itself and/or create new
/// atoms which must be stored in one of the maps below.
///
/// After all references are handled, the atoms created during that are all
/// added to mf.
virtual void perform(std::unique_ptr<MutableFile> &mf) {
    ScopedTask task(getDefaultDomain(), "Cpu0 GOT/PLT Pass");
    // Process all references.
    for (const auto &atom : mf->defined())
        for (const auto &ref : *atom)
            handleReference(*atom, *ref);

    // Add all created atoms to the link.
    uint64_t ordinal = 0;
    if (_ctx.getOutputELFType() == llvm::ELF::ET_EXEC) {
        MutableFile::DefinedAtomRange atomRange = mf->definedAtoms();
        auto it = atomRange.begin();
        bool find = false;
        for (it = atomRange.begin(); it < atomRange.end(); it++) {
            if ((*it)->name() == "_Z5startv") {
                find = true;
                break;
            }
        }
        assert(find && "not found _Z5startv\n");
        _boot->addReference(R_CPU0_PC24, 0, *it, -3);
        _boot->setOrdinal(ordinal++);
        mf->addAtom(*_boot);
    }
#endif _LINKER
    if (_PLT0) {
        MutableFile::DefinedAtomRange atomRange = mf->definedAtoms();
        auto it = atomRange.begin();
        bool find = false;
        for (it = atomRange.begin(); it < atomRange.end(); it++) {
            if ((*it)->name() == "_Z14dynamic_linkerv") {
                find = true;
                break;
            }
        }
        assert(find && "Cannot find _Z14dynamic_linkerv()");
        _PLT0->addReference(R_CPU0_PC24, 12, *it, -3);
        _PLT0->setOrdinal(ordinal++);
        mf->addAtom(*_PLT0);
    }
    for (auto &plt : _pltVector) {
        plt->setOrdinal(ordinal++);
        mf->addAtom(*plt);
    }
    if (_null) {
        _null->setOrdinal(ordinal++);
        mf->addAtom(*_null);
    }
    if (_PLT0) {
        _got0->setOrdinal(ordinal++);
    }
}

```

```

        mf->addAtom(*_got0);
    }
    for (auto &got : _gotVector) {
        got->setOrdinal(ordinal++);
        mf->addAtom(*got);
    }
#endif // DLINKER
}

protected:
    /// \brief Owner of all the Atoms created by this pass.
    ELFPassFile _file;
    const ELFLinkingContext &_ctx;

    /// \brief Map Atoms to their GOT entries.
    llvm::DenseMap<const Atom *, GOTAtom *> _gotMap;

    /// \brief Map Atoms to their PLT entries.
    llvm::DenseMap<const Atom *, PLTAtom *> _pltMap;
    /// \brief the list of GOT/PLT atoms
    std::vector<GOTAtom *> _gotVector;
    std::vector<PLTAtom *> _pltVector;
    PLT0Atom *_boot;

    /// \brief GOT entry that is always 0. Used for undefined weaks.
    GOTAtom *_null;

    /// \brief The got and plt entries for .PLT0. This is used to call into the
    /// dynamic linker for symbol resolution.
    /// @{
    PLT0Atom *_PLT0;
    GOTAtom *_got0;
    /// @@
};

/// This implements the static relocation model. Meaning GOT and PLT entries are
/// not created for references that can be directly resolved. These are
/// converted to a direct relocation. For entries that do require a GOT or PLT
/// entry, that entry is statically bound.
///
/// TLS always assumes module 1 and attempts to remove indirection.
class StaticRelocationPass LLVM_FINAL
    : public RelocationPass<StaticRelocationPass> {
public:
    StaticRelocationPass(const elf::Cpu0LinkingContext &ctx)
        : RelocationPass(ctx) {}

    ErrorOr<void> handlePlain(const Reference &ref) { return handleIFUNC(ref); }

    ErrorOr<void> handlePLT32(const Reference &ref) {
        // __tls_get_addr is handled elsewhere.
        if (ref.target() && ref.target()->name() == "__tls_get_addr") {
            const_cast<Reference &>(ref).setKind(R_CPU0_NONE);
            return error_code::success();
        } else
            // Static code doesn't need PLTs.
            const_cast<Reference &>(ref).setKind(R_CPU0_PC24);
        // Handle IFUNC.
    }
}

```

```

if (const DefinedAtom *da =
    dyn_cast_or_null<const DefinedAtom>(ref.target()))
if (da->contentType() == DefinedAtom::typeResolver)
    return handleIFUNC(ref);
return error_code::success();
}

ErrorOr<void> handleGOT(const Reference &ref) {
    if (isa<UndefinedAtom>(ref.target()))
        const_cast<Reference &>(ref).setTarget(getNullGOT());
    else if (const DefinedAtom *da = dyn_cast<const DefinedAtom>(ref.target()))
        const_cast<Reference &>(ref).setTarget(getGOT(da));
    return error_code::success();
}
};

#ifdef DLINKER
class DynamicRelocationPass LLVM_FINAL
    : public RelocationPass<DynamicRelocationPass> {
public:
    DynamicRelocationPass(const elf::Cpu0LinkingContext &ctx)
        : RelocationPass(ctx) {}

    const PLT0Atom *getPLT0() {
        if (_PLT0)
            return _PLT0;
        // Fill in the null entry.
        getNullGOT();
        _PLT0 = new (_file._alloc) Cpu0PLT0Atom(_file);
        _got0 = new (_file._alloc) Cpu0GOTAtom(_file, ".got.plt");
#ifndef NDEBUG
        _got0->_name = "__got0";
#endif
        return _PLT0;
    }

    const PLTAtom *getPLTEEntry(const Atom *a) {
        auto plt = _pltMap.find(a);
        if (plt != _pltMap.end())
            return plt->second;
        auto ga = new (_file._alloc) Cpu0GOTAtom(_file, ".got.plt");
        ga->addReference(R_CPU0_JUMP_SLOT, 0, a, 0);
        auto pa = new (_file._alloc) Cpu0PLTAtom(_file, ".plt");
        getPLT0(); // add _PLT0 and _got0
        // Set the starting address of the got entry to the second instruction in
        // the plt entry.
        ga->addReference(R_CPU0_32, 0, pa, 4);
#ifndef NDEBUG
        ga->_name = "__got_";
        ga->_name += a->name();
        pa->_name = "__plt_";
        pa->_name += a->name();
#endif
        _gotMap[a] = ga;
        _pltMap[a] = pa;
        _gotVector.push_back(ga);
        _pltVector.push_back(pa);
        return pa;
    }
}

```

```

    }

ErrorOr<void> handlePlain(const Reference &ref) {
    if (!ref.target())
        return error_code::success();
    if (auto sla = dyn_cast<SharedLibraryAtom>(ref.target())) {
        if (sla->type() == SharedLibraryAtom::Type::Code) {
            const_cast<Reference &>(ref).setTarget(getPLTEEntry(sla));
            // When caller of execution file call shared library function
            // Turn this into a PC24 to the PLT entry.
            const_cast<Reference &>(ref).setKind(R_CPU0_PC24);
        }
    } else
        return handleIFUNC(ref);
    return error_code::success();
}

ErrorOr<void> handlePLT32(const Reference &ref) {
    // Handle IFUNC.
    if (const DefinedAtom *da =
        dyn_cast_or_null<const DefinedAtom>(ref.target()))
        if (da->contentType() == DefinedAtom::typeResolver)
            return handleIFUNC(ref);
    if (isa<const SharedLibraryAtom>(ref.target())) {
        const_cast<Reference &>(ref).setTarget(getPLTEEntry(ref.target()));
        // Turn this into a PC24 to the PLT entry.
#ifndef 1
        const_cast<Reference &>(ref).setKind(R_CPU0_PC24);
#endif
    }
    return error_code::success();
}

const GOTAtom *getSharedGOT(const SharedLibraryAtom *sla) {
    auto got = _gotMap.find(sla);
    if (got == _gotMap.end()) {
        auto g = new (_file._alloc) Cpu0GOTAtom(_file, ".got.dyn");
        g->addReference(R_CPU0_GLOB_DAT, 0, sla, 0);
#ifndef NDEBUG
        g->_name = "__got_";
        g->_name += sla->name();
#endif
        _gotMap[sla] = g;
        _gotVector.push_back(g);
        return g;
    }
    return got->second;
}

ErrorOr<void> handleGOT(const Reference &ref) {
    if (isa<UndefinedAtom>(ref.target()))
        const_cast<Reference &>(ref).setTarget(getNullGOT());
    else if (const DefinedAtom *da = dyn_cast<const DefinedAtom>(ref.target()))
        const_cast<Reference &>(ref).setTarget(getGOT(da));
    else if (const auto sla = dyn_cast<const SharedLibraryAtom>(ref.target()))
        const_cast<Reference &>(ref).setTarget(getSharedGOT(sla));
    return error_code::success();
}
;

```

```

#endif // DLINKER
} // end anon namespace

std::unique_ptr<Pass>
lld::elf::createCpu0RelocationPass(const Cpu0LinkingContext &ctx) {
    switch (ctx.getOutputELFType()) {
        case llvm::ELF::ET_EXEC:
#ifndef DLINKER
            if (ctx.isDynamic())
                return std::unique_ptr<Pass> (new DynamicRelocationPass(ctx));
            else
#endif // DLINKER
                return std::unique_ptr<Pass> (new StaticRelocationPass(ctx));
#ifndef DLINKER
        case llvm::ELF::ET_DYN:
            return std::unique_ptr<Pass> (new DynamicRelocationPass(ctx));
#endif // DLINKER
        case llvm::ELF::ET_REL:
            return std::unique_ptr<Pass> ();
        default:
            llvm_unreachable("Unhandled output file type");
    }
}

```

### lbdex/Cpu0\_lld\_1030/Cpu0/Cpu0LinkingContext.cpp

```

//===== lib/ReaderWriter/ELF/Cpu0/Cpu0LinkingContext.cpp -----
// The LLVM Linker
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//=====

#include "Cpu0LinkingContext.h"

#include "lld/Core/File.h"
#include "lld/Core/Instrumentation.h"

#include "llvm/ADT/ArrayRef.h"
#include "llvm/ADT/StringSwitch.h"

#include "Atoms.h"
#include "Cpu0RelocationPass.h"

using namespace lld;

using namespace lld::elf;

namespace {
using namespace llvm::ELF;
const uint8_t cpu0InitFiniAtomContent[8] = { 0 };

// Cpu0_64InitFini Atom

```

```

class Cpu0InitAtom : public InitFinAtom {
public:
    Cpu0InitAtom(const File &f, StringRef function)
        : InitFinAtom(f, ".init_array") {
#ifndef NDEBUG
        _name = "__init_fn__";
        _name += function;
#endif
    }
    virtual ArrayRef<uint8_t> rawContent() const {
        return ArrayRef<uint8_t>(cpu0InitFinAtomContent, 8);
    }
    virtual Alignment alignment() const { return Alignment(3); }
};

class Cpu0FinAtom : public InitFinAtom {
public:
    Cpu0FinAtom(const File &f, StringRef function)
        : InitFinAtom(f, ".fini_array") {
#ifndef NDEBUG
        _name = "__fini_fn__";
        _name += function;
#endif
    }
    virtual ArrayRef<uint8_t> rawContent() const {
        return ArrayRef<uint8_t>(cpu0InitFinAtomContent, 8);
    }
    virtual Alignment alignment() const { return Alignment(3); }
};

class Cpu0InitFinFile : public SimpleFile {
public:
    Cpu0InitFinFile(const ELFLinkingContext &context)
        : SimpleFile(context, "command line option -init/-fini"), _ordinal(0) {}

    void addInitFunction(StringRef name) {
        Atom *initFunctionAtom = new (_allocator) SimpleUndefinedAtom(*this, name);
        Cpu0InitAtom *initAtom =
            (new (_allocator) Cpu0InitAtom(*this, name));
        initAtom->addReference(llvm::ELF::R_CPU0_32, 0, initFunctionAtom, 0);
        initAtom->setOrdinal(_ordinal++);
        addAtom(*initFunctionAtom);
        addAtom(*initAtom);
    }

    void addFiniFunction(StringRef name) {
        Atom *finiFunctionAtom = new (_allocator) SimpleUndefinedAtom(*this, name);
        Cpu0FinAtom *finiAtom =
            (new (_allocator) Cpu0FinAtom(*this, name));
        finiAtom->addReference(llvm::ELF::R_CPU0_32, 0, finiFunctionAtom, 0);
        finiAtom->setOrdinal(_ordinal++);
        addAtom(*finiFunctionAtom);
        addAtom(*finiAtom);
    }

private:
    llvm::BumpPtrAllocator _allocator;
}

```

```

        uint64_t _ordinal;
    };

} // end anon namespace

void elf::Cpu0LinkingContext::addPasses(PassManager &pm) {
    auto pass = createCpu0RelocationPass(*this);
    if (pass)
        pm.add(std::move(pass));
    ELFLinkingContext::addPasses(pm);
}

bool elf::Cpu0LinkingContext::createInternalFiles(
    std::vector<std::unique_ptr<File> > &result) const {
    ELFLinkingContext::createInternalFiles(result);
    std::unique_ptr<Cpu0InitFiniFile> initFiniFile(
        new Cpu0InitFiniFile(*this));
    for (auto ai : initFunctions())
        initFiniFile->addInitFunction(ai);
    for (auto ai : finiFunctions())
        initFiniFile->addFiniFunction(ai);
    result.push_back(std::move(initFiniFile));
    return true;
}

#define LLD_CASE(name) .Case(#name, llvm::ELF::name)

ErrorOr<Reference::Kind>
elf::Cpu0LinkingContext::relocKindFromString(StringRef str) const {
    int32_t ret = llvm::StringSwitch<int32_t>(str)
        LLD_CASE(R_CPU0_NONE)
        LLD_CASE(R_CPU0_24)
        LLD_CASE(R_CPU0_32)
        LLD_CASE(R_CPU0_HI16)
        LLD_CASE(R_CPU0_LO16)
        LLD_CASE(R_CPU0_GPREL16)
        LLD_CASE(R_CPU0_LITERAL)
        LLD_CASE(R_CPU0_GOT16)
        LLD_CASE(R_CPU0_PC24)
        LLD_CASE(R_CPU0_CALL16)
        LLD_CASE(R_CPU0_JUMP_SLOT)
            .Case("LLD_R_CPU0_GOTRELINDEX", LLD_R_CPU0_GOTRELINDEX)
            .Default(-1);

    if (ret == -1)
        return make_error_code(YamlReaderError::illegal_value);
    return ret;
}

#define LLD_CASE(name) case llvm::ELF::name: return std::string(#name);

ErrorOr<std::string>
elf::Cpu0LinkingContext::stringFromRelocKind(Reference::Kind kind) const {
    switch (kind) {
        LLD_CASE(R_CPU0_NONE)
        LLD_CASE(R_CPU0_24)

```

```
LLD_CASE(R_CPU0_32)
LLD_CASE(R_CPU0_HI16)
LLD_CASE(R_CPU0_LO16)
LLD_CASE(R_CPU0_GPREL16)
LLD_CASE(R_CPU0_LITERAL)
LLD_CASE(R_CPU0_GOT16)
LLD_CASE(R_CPU0_PC24)
LLD_CASE(R_CPU0_CALL16)
LLD_CASE(R_CPU0_JUMP_SLOT)
case LLD_R_CPU0_GOTRELINDEX:
    return std::string("LLD_R_CPU0_GOTRELINDEX");
}

return make_error_code(YamlReaderError::illegal_value);
}
```

### lbdex/Cpu0\_lld\_1030/Cpu0/Cpu0Target.h

```
===== lib/ReaderWriter/ELF/Cpu0/Cpu0Target.h =====
//
//          The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
=====

#include "Cpu0LinkingContext.h"
```

### lbdex/Cpu0\_lld\_1030/Cpu0/Cpu0TargetHandler.h

```
===== lib/ReaderWriter/ELF/Cpu0/Cpu0TargetHandler.h =====
//
//          The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
=====

#ifndef LLD_READER_WRITER_ELF_Cpu0_TARGET_HANDLER_H
#define LLD_READER_WRITER_ELF_Cpu0_TARGET_HANDLER_H

#include "DefaultTargetHandler.h"
#include "File.h"
#include "Cpu0RelocationHandler.h"
#include "TargetLayout.h"

#include "lld/ReaderWriter/Simple.h"

#include "lld/Core/Atom.h"

#define DLINKER

namespace lld {
```

```

namespace elf {
typedef llvm::object::ELFType<llvm::support::big, 4, false> Cpu0ELFType;
class Cpu0LinkingContext;

class Cpu0TargetHandler LLVM_FINAL
    : public DefaultTargetHandler<Cpu0ELFType> {
public:
    Cpu0TargetHandler(Cpu0LinkingContext &targetInfo);

    virtual TargetLayout<Cpu0ELFType> &targetLayout() {
        return _targetLayout;
    }

    virtual const Cpu0TargetRelocationHandler &getRelocationHandler() const {
        return _relocationHandler;
    }
    virtual bool createImplicitFiles(std::vector<std::unique_ptr<File> > &);

private:
    class GOTFile : public SimpleFile {
    public:
        GOTFile(const ELFLinkingContext &eti) : SimpleFile(eti, "GOTFile") {}
        llvm::BumpPtrAllocator _alloc;
    };

    std::unique_ptr<GOTFile> _gotFile;

    Cpu0TargetRelocationHandler _relocationHandler;
    TargetLayout<Cpu0ELFType> _targetLayout;
};

} // end namespace elf
} // end namespace lld

#endif

```

### Ibdex/Cpu0\_lld\_1030/Cpu0/Cpu0TargetHandler.cpp

```

===== lib/ReaderWriter/ELF/Cpu0/Cpu0TargetHandler.cpp =====
//
//          The LLVM Linker
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
=====

#include "Atoms.h"
#include "Cpu0TargetHandler.h"
#include "Cpu0LinkingContext.h"

using namespace lld;
using namespace elf;

uint64_t textSectionAddr;

Cpu0TargetHandler::Cpu0TargetHandler(Cpu0LinkingContext &context)
    : DefaultTargetHandler(context), _gotFile(new GOTFile(context)),

```

```

        _relocationHandler(context), _targetLayout(context) {}

bool Cpu0TargetHandler::createImplicitFiles(
    std::vector<std::unique_ptr<File> > &result) {
    _gotFile->addAtom(*new (_gotFile->_alloc) GLOBAL_OFFSET_TABLEAtom(*_gotFile));
    _gotFile->addAtom(*new (_gotFile->_alloc) TLSGETADDRAtom(*_gotFile));
    if (_context.isDynamic())
        _gotFile->addAtom(*new (_gotFile->_alloc) DYNAMICAtom(*_gotFile));
    result.push_back(std::move(_gotFile));
    return true;
}

```

Above code in Cpu0 lld support both the static and dynamic link. The “#ifdef DLINKER” is for dynamic link support. There are only just over 1 thousand of code in it. Half of the code size is for the dynamic linker.

### 13.3 ELF to Hex

Add elf2hex.h and update llvm-objdump driver to support ELF to Hex for Cpu0 backend as follows,

#### lbdex/llvm-objdump/elf2hex.h

```

//===== elf2hex.cpp =====//
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===== //
//
// This program is a utility that works with llvm-objdump.
//
//===== //

#include <stdio.h>

static cl::opt<bool>
ConvertElf2Hex("elf2hex",
    cl::desc("Display the hex content of verilog cpu0 needed sections"));

static cl::opt<bool>
DumpSo("cpu0dumpso",
    cl::desc("Dump shared library .so"));

static cl::opt<bool>
LinkSo("cpu0linkso",
    cl::desc("Link shared library .so"));

// Modified from PrintSectionHeaders()
static uint64_t GetSectionHeaderStartAddress(const ObjectFile *o,
    StringRef sectionName) {
// outs() << "Sections:\n"
//     "Idx Name          Size      Address          Type\n";
    error_code ec;
    unsigned i = 0;

```

```
for (section_iterator si = o->begin_sections(), se = o->end_sections();
                                         si != se; si.increment(ec)) {
    if (error(ec)) return 0;
    StringRef Name;
    if (error(si->getName(Name))) return 0;
    uint64_t Address;
    if (error(si->getAddress(Address))) return 0;
    uint64_t Size;
    if (error(si->getSize(Size))) return 0;
    bool Text, Data, BSS;
    if (error(si->isText(Text))) return 0;
    if (error(si->isData(Data))) return 0;
    if (error(si->isBSS(BSS))) return 0;
    if (Name == sectionName)
        return Address;
    else
        return 0;
    ++i;
}
return 0;
}

// Modified from PrintSymbolTable()
static void GetSymbolTableStartAddress(const ObjectFile *o, StringRef sectionName) {
    outs() << "SYMBOL TABLE:\n";

    if (const COFFObjectFile *coff = dyn_cast<const COFFObjectFile>(o))
        PrintCOFFSymbolTable(coff);
    else {
        error_code ec;
        for (symbol_iterator si = o->begin_symbols(),
                         se = o->end_symbols(); si != se; si.increment(ec)) {
            if (error(ec)) return;
            StringRef Name;
            uint64_t Address;
            SymbolRef::Type Type;
            uint64_t Size;
            uint32_t Flags;
            section_iterator Section = o->end_sections();
            if (error(si->getName(Name))) continue;
            if (error(si->getAddress(Address))) continue;
            if (error(si->getFlags(Flags))) continue;
            if (error(si->getType(Type))) continue;
            if (error(si->getSize(Size))) continue;
            if (error(si->getSection(Section))) continue;

            bool Global = Flags & SymbolRef::SF_Global;
            bool Weak = Flags & SymbolRef::SF_Weak;
            bool Absolute = Flags & SymbolRef::SF_Absolute;

            if (Address == UnknownAddressOrSize)
                Address = 0;
            if (Size == UnknownAddressOrSize)
                Size = 0;
            char GlobLoc = ' ';
            if (Type != SymbolRef::ST_Unknown)
                GlobLoc = Global ? 'g' : 'l';
            char Debug = (Type == SymbolRef::ST_Debug || Type == SymbolRef::ST_File)
```

```

        ? 'd' : ' ';
    char FileFunc = ' ';
    if (Type == SymbolRef::ST_File)
        FileFunc = 'f';
    else if (Type == SymbolRef::ST_Function)
        FileFunc = 'F';

const char *Fmt = o->getBytesInAddress() > 4 ? "%016" PRIx64 :
                                         "%08" PRIx64;

outs() << format(Fmt, Address) << " "
    << GlobLoc // Local -> 'l', Global -> 'g', Neither -> ''
    << (Weak ? 'w' : ' ') // Weak?
    << ' ' // Constructor. Not supported yet.
    << ' ' // Warning. Not supported yet.
    << ' ' // Indirect reference to another symbol.
    << Debug // Debugging (d) or dynamic (D) symbol.
    << FileFunc // Name of function (F), file (f) or object (O).
    << ' ';

if (Absolute)
    outs() << "*ABS*";
else if (Section == o->end_sections())
    outs() << "*UND*";
else {
    if (const MachOObjectFile *MachO =
        dyn_cast<const MachOObjectFile>(o)) {
        DataRefImpl DR = Section->getRawDataRefImpl();
        StringRef SegmentName = MachO->getSectionFinalSegmentName(DR);
        outs() << SegmentName << ",";
    }
    StringRef SectionName;
    if (error(Section->getName(SectionName)))
        SectionName = "";
    outs() << SectionName;
}
outs() << '\t'
    << format("%08" PRIx64 " ", Size)
    << Name
    << '\n';
}

}

class Cpu0DynFunIndex {
private:
    char soStrtab[20][100];
    int soStrtabSize = 0;

    char exePltName[20][100];
    int exePltNameSize = 0;

    int findPltName(const char* pltName);
public:
    void createPltName(const ObjectFile *o);
    void createStrtab();
    uint16_t correctDynFunIndex(const char* pltName);
};

```

```

int Cpu0DynFunIndex::findPltName(const char* pltName) {
    for (int i = 0; i < exePltNameSize; i++)
        if (strcmp(pltName, exePltName[i]) == 0)
            return i;
    return -1;
}

void Cpu0DynFunIndex::createPltName(const ObjectFile *o) {
    error_code ec;
    std::string Error;

    for (section_iterator si = o->begin_sections(),
         se = o->end_sections();
         si != se; si.increment(ec)) {
        if (error(ec)) return;
        StringRef Name;
        StringRef Contents;
        uint64_t BaseAddr;
        bool BSS;
        if (error(si->getName(Name))) continue;
        if (error(si->getContents(Contents))) continue;
        if (error(si->getAddress(BaseAddr))) continue;
        if (error(si->isBSS(BSS))) continue;

        if (Name == ".strtab") {
            int num_dyn_entry = 0;
            FILE *fd_num_dyn_entry;
            fd_num_dyn_entry = fopen("num_dyn_entry", "r");
            if (fd_num_dyn_entry != NULL) {
                fscanf(fd_num_dyn_entry, "%d", &num_dyn_entry);
            }
            fclose(fd_num_dyn_entry);

            for (std::size_t addr = 2+strlen(".PLT0"), end = Contents.size();
                 addr < end; ) {
                if (Contents.substr(addr, strlen("__plt_")) != "__plt_")
                    break;
                strcpy(exePltName[exePltNameSize], Contents.data() + addr);
                addr = addr + strlen(exePltName[exePltNameSize]) + 1;
                exePltNameSize++;
            }
            break;
        }
    }
}

void Cpu0DynFunIndex::createStrtab() {
    FILE *fd_dynstrAscii;

    fd_dynstrAscii = fopen("dynstrAscii", "r");
    if (fd_dynstrAscii == NULL)
        fclose(fd_dynstrAscii);
    assert(fd_dynstrAscii != NULL && "fd_dynstr == NULL");
    int i = 0;
    // function                                result on EOF or error
    // -----                                     -----
    // fgets()                                     NULL
    // fscanf()                                    number of successful conversions
}

```

```

//                                     less than expected
// fgetc()                           EOF
// fread()                            number of elements read
//                                     less than expected
int j = 0;
for (i=0; 1; i++) {
    j=fscanf(fd_dynstrAscii, "%s", soStrtab[i]);
    if (j != 1)
        break;
}
soStrtabSize = i;
fclose(fd_dynstrAscii);
}

uint16_t Cpu0DynFunIndex::correctDynFunIndex(const char* pltName) {
    int i = findPltName(pltName);
    if (i != -1) {
        int j = 0;
        for (j=0; j < soStrtabSize; j++)
            if (strcmp(soStrtab[j], (const char*)exePltName[i]+strlen("__plt_")) == 0)
                break;
        if (j == soStrtabSize) {
            outs() << "cannot find " << exePltName[i] << "\n";
            exit(1);
        }
        j++;
        return (uint16_t)(j & 0xffff);
    }
    return (uint16_t)0;
}

Cpu0DynFunIndex cpu0DynFunIndex;

// Modified from DisassembleObject()
static void DisassembleObjectInHexFormat(const ObjectFile *Obj
/*, bool InlineRelocs*/ , uint64_t& lastDumpAddr) {
    std::string Error;
    uint64_t soLastPrintAddr = 0;
    FILE *fd_so_func_offset;
    int num_dyn_entry = 0;
    if (DumpSo) {
        fd_so_func_offset = fopen("so_func_offset", "w");
        if (fd_so_func_offset == NULL)
            fclose(fd_so_func_offset);
        assert(fd_so_func_offset != NULL && "fd_so_func_offset == NULL");
    }
    if (LinkSo) {
        cpu0DynFunIndex.createStrtab();
    }

    const Target *TheTarget = getTarget(Obj);
    // getTarget() will have already issued a diagnostic if necessary, so
    // just bail here if it failed.
    if (!TheTarget)
        return;

    // Package up features to be passed to target/subtarget
    std::string FeaturesStr;

```

```

if (MAttrs.size()) {
    SubtargetFeatures Features;
    for (unsigned i = 0; i != MAttrs.size(); ++i)
        Features.AddFeature(MAttrs[i]);
    FeaturesStr = Features.getString();
}

OwningPtr<const MCRegisterInfo> MRI (TheTarget->createMCRegInfo(TripleName));
if (!MRI) {
    errs() << "error: no register info for target " << TripleName << "\n";
    return;
}

// Set up disassembler.
OwningPtr<const MCAsmInfo> AsmInfo(
    TheTarget->createMCAsmInfo(*MRI, TripleName));
if (!AsmInfo) {
    errs() << "error: no assembly info for target " << TripleName << "\n";
    return;
}

OwningPtr<const MCSubtargetInfo> STI(
    TheTarget->createMCSubtargetInfo(TripleName, "", FeaturesStr));
if (!STI) {
    errs() << "error: no subtarget info for target " << TripleName << "\n";
    return;
}

OwningPtr<const MCInstrInfo> MII (TheTarget->createMCInstrInfo());
if (!MII) {
    errs() << "error: no instruction info for target " << TripleName << "\n";
    return;
}

OwningPtr<MCDisassembler> DisAsm(TheTarget->createMCDisassembler(*STI));
if (!DisAsm) {
    errs() << "error: no disassembler for target " << TripleName << "\n";
    return;
}

OwningPtr<const MCObjectFileInfo> MOFI;
OwningPtr<MCContext> Ctx;

if (Symbolize) {
    MOFI.reset(new MCObjectFileInfo);
    Ctx.reset(new MCContext(AsmInfo.get(), MRI.get(), MOFI.get()));
    OwningPtr<MCRelocationInfo> RelInfo(
        TheTarget->createMCRelocationInfo(TripleName, *Ctx.get()));
    if (RelInfo) {
        OwningPtr<MCSSymbolizer> Symzer(
            MCObjectSymbolizer::createObjectSymbolizer(*Ctx.get(), RelInfo, Obj));
        if (Symzer)
            DisAsm->setSymbolizer(Symzer);
    }
}

OwningPtr<const MCInstrAnalysis>
MIA(TheTarget->createMCInstrAnalysis(MII.get()));

```

```

int AsmPrinterVariant = AsmInfo->getAssemblerDialect();
OwningPtr<MCInstPrinter> IP(TheTarget->createMCInstPrinter(
    AsmPrinterVariant, *AsmInfo, *MII, *MRI, *STI));
if (!IP) {
    errs() << "error: no instruction printer for target " << TripleName
    << '\n';
    return;
}

if (CFG) {
    OwningPtr<MCObjectDisassembler> OD(
        new MCObjectDisassembler(*Obj, *DisAsm, *MIA));
    OwningPtr<MCModule> Mod(OD->buildModule(/* withCFG */ true));
    for (MCModule::const_atom_iterator AI = Mod->atom_begin(),
        AE = Mod->atom_end();
        AI != AE; ++AI) {
        outs() << "Atom " << (*AI)->getName() << " : \n";
        if (const MCTextAtom *TA = dyn_cast<MCTextAtom>(*AI)) {
            for (MCTextAtom::const_iterator II = TA->begin(), IE = TA->end();
                II != IE;
                ++II) {
                IP->printInst(&II->Inst, outs(), "");
                outs() << "\n";
            }
        }
    }
    for (MCModule::const_func_iterator FI = Mod->func_begin(),
        FE = Mod->func_end();
        FI != FE; ++FI) {
        static int filenum = 0;
        emitDOTFile((Twine((*FI)->getName()) + "_" +
            utostr(filenum) + ".dot").str().c_str(),
            **FI, IP.get());
        ++filenum;
    }
}

error_code ec;
for (section_iterator i = Obj->begin_sections(),
    e = Obj->end_sections();
    i != e; i.increment(ec)) {
    if (error(ec)) break;
    bool text;
    if (error(i->isText(text))) break;
    if (!text) continue;

    uint64_t SectionAddr;
    if (error(i->getAddress(SectionAddr))) break;

    // Make a list of all the symbols in this section.
    std::vector<std::pair<uint64_t, StringRef> > Symbols;
    for (symbol_iterator si = Obj->begin_symbols(),
        se = Obj->end_symbols();
        si != se; si.increment(ec)) {
        bool contains;
        if (!error(i->containsSymbol(*si, contains)) && contains) {
            uint64_t Address;

```

```
    if (error(si->getAddress(Address))) break;
    if (Address == UnknownAddressOrSize) continue;
    Address -= SectionAddr;

    StringRef Name;
    if (error(si->getName(Name))) break;
    Symbols.push_back(std::make_pair(Address, Name));
}
}

// Sort the symbols by address, just in case they didn't come in that way.
array_pod_sort(Symbols.begin(), Symbols.end());

// Make a list of all the relocations for this section.
std::vector<RelocationRef> Rels;
/*  if (InlineRelocs) {
    for (relocation_iterator ri = i->begin_relocations(),
         re = i->end_relocations();
         ri != re; ri.increment(ec)) {
        if (error(ec)) break;
        Rels.push_back(*ri);
    }
} */

// Sort relocations by address.
std::sort(Rels.begin(), Rels.end(), RelocAddressLess);

StringRef SegmentName = "";
if (const MachOObjectFile *MachO =
    dyn_cast<const MachOObjectFile>(Obj)) {
    DataRefImpl DR = i->getRawDataRefImpl();
    SegmentName = MachO->getSectionFinalSegmentName(DR);
}
StringRef name;
if (error(i->getName(name))) break;
if (DumpSo && name == ".plt") continue;
outs() << /* */ << "Disassembly of section ";
if (!SegmentName.empty())
    outs() << SegmentName << ",";
outs() << name << ':' << /* */;

// If the section has no symbols just insert a dummy one and disassemble
// the whole section.
if (Symbols.empty())
    Symbols.push_back(std::make_pair(0, name));

SmallString<40> Comments;
raw_svector_ostream CommentStream(Comments);

StringRef Bytes;
if (error(i->getContents(Bytes))) break;
StringRefMemoryObject memoryObject(Bytes, SectionAddr);
uint64_t Size;
uint64_t Index;
uint64_t SectSize;
if (error(i->getSize(SectSize))) break;

std::vector<RelocationRef>::const_iterator rel_cur = Rels.begin();
```

```

std::vector<RelocationRef>::const_iterator rel_end = Rels.end();
// Disassemble symbol by symbol.
for (unsigned si = 0, se = Symbols.size(); si != se; ++si) {
    uint64_t Start = Symbols[si].first;
    uint64_t End;
    // The end is either the size of the section or the beginning of the next
    // symbol.
    if (si == se - 1)
        End = SectSize;
    // Make sure this symbol takes up space.
    else if (Symbols[si + 1].first != Start)
        End = Symbols[si + 1].first - 1;
    else {
        // This symbol has the same address as the next symbol. Skip it.
        if (DumpSo/* && Symbols[si].second != "__tls_get_addr"*/) {
            fprintf(fd_so_func_offset, "%02x ",
                    (uint8_t)(Symbols[si].first >> 24));
            fprintf(fd_so_func_offset, "%02x ",
                    (uint8_t)((Symbols[si].first >> 16) & 0xFF));
            fprintf(fd_so_func_offset, "%02x ",
                    (uint8_t)((Symbols[si].first >> 8) & 0xFF));
            fprintf(fd_so_func_offset, "%02x ",
                    (uint8_t)((Symbols[si].first) & 0xFF));
            std::string str = Symbols[si].second.str();
            std::size_t idx = 0;
            std::size_t strSize = 0;
            for (idx = 0, strSize = str.size(); idx < strSize; idx++) {
                fprintf(fd_so_func_offset, "%c%c ",
                    hexdigit((str[idx] >> 4) & 0xF, true),
                    hexdigit(str[idx] & 0xF, true));
            }
            for (idx = strSize; idx < 48; idx++) {
                fprintf(fd_so_func_offset, "%02x ", 0);
            }
            fprintf(fd_so_func_offset, "/* %s */\n", Symbols[si].second.begin());
            num_dyn_entry++;
        }
    }
    outs() << '\n' << "/* " << Symbols[si].second << " : */\n";
    continue;
}

if (DumpSo) {
    soLastPrintAddr = Symbols[si].first;
    fprintf(fd_so_func_offset, "%02x ", (uint8_t)(Symbols[si].first >> 24));
    fprintf(fd_so_func_offset, "%02x ",
            (uint8_t)((Symbols[si].first >> 16) & 0xFF));
    fprintf(fd_so_func_offset, "%02x ",
            (uint8_t)((Symbols[si].first >> 8) & 0xFF));
    fprintf(fd_so_func_offset, "%02x ",
            (uint8_t)((Symbols[si].first) & 0xFF));
    std::string str = Symbols[si].second.str();
    std::size_t idx = 0;
    std::size_t strSize = 0;
    for (idx = 0, strSize = str.size(); idx < strSize; idx++) {
        fprintf(fd_so_func_offset, "%c%c ",
            hexdigit((str[idx] >> 4) & 0xF, true),
            hexdigit(str[idx] & 0xF, true));
    }
}

```

```
        }
        for (idx = strSize; idx < 48; idx++) {
            fprintf(fd_so_func_offset, "%02x ", 0);
        }
        fprintf(fd_so_func_offset, /* %s */"\n", Symbols[si].second.begin());
        num_dyn_entry++;
    }

    outs() << '\n' << /* %s */"\n", Symbols[si].second << "/*\n";
    uint16_t funIndex = 0;
    if (LinkSo) {
        // correctDynFunIndex
        funIndex = cpu0DynFunIndex.correctDynFunIndex(Symbols[si].second.data());
    }

#ifndef NDEBUG
    raw_ostream &DebugOut = DebugFlag ? dbgs() : nulls();
#else
    raw_ostream &DebugOut = nulls();
#endif

for (Index = Start; Index < End; Index += Size) {
    MCInst Inst;

    if (LinkSo && funIndex && Index == Start) {
        outs() << format("/*%8" PRIx64 "/*\t", /*SectionAddr + */lastDumpAddr+Index);
        outs() << "09 60 " << format("%02" PRIx64, funIndex & 0xff00)
            << format(" %02" PRIx64, funIndex & 0x0ff);
        outs() << "                                /* addiu\t$t9, $zero, "
            << funIndex << "($gp)*\n";
    }
    else {
        if (DisAsm->getInstruction(Inst, Size, memoryObject,
            SectionAddr + Index,
            DebugOut, CommentStream)) {
            outs() << format("/*%8" PRIx64 "/*\t", /*SectionAddr + */lastDumpAddr+Index);
            if (!NoShowRawInsn) {
                outs() << "\t";
                DumpBytes(StringRef(Bytes.data() + Index, Size));
            }
            outs() << "/*";
            IP->printInst(&Inst, outs(), "");
            outs() << CommentStream.str();
            outs() << "*/";
            Comments.clear();
            outs() << "\n";
        } else {
            errs() << ToolName << ": warning: invalid instruction encoding\n";
            if (Size == 0)
                Size = 1; // skip illegible bytes
        }
    }
}

// outs() << "Size = " << Size << "Index = " << Index << "lastDumpAddr = "
//           << lastDumpAddr << "\n"; // debug
// Print relocation for instruction.
while (rel_cur != rel_end) {
    bool hidden = false;
```

```

        uint64_t addr;
        SmallString<16> name;
        SmallString<32> val;

        // If this relocation is hidden, skip it.
        if (error(rel_cur->getHidden(hidden))) goto skip_print_rel;
        if (hidden) goto skip_print_rel;

        if (error(rel_cur->getOffset(addr))) goto skip_print_rel;
        // Stop when rel_cur's address is past the current instruction.
        if (addr >= Index + Size) break;
        if (error(rel_cur->getTypeName(name))) goto skip_print_rel;
        if (error(rel_cur->getValueString(val))) goto skip_print_rel;

        outs() << format("\t\t\t/*%8" PRIx64 ": ", SectionAddr + addr) << name
            << "\t" << val << "*/\n";

        skip_print_rel:
        ++rel_cur;
    }
}
if (DumpSo)
    soLastPrintAddr = End;
}
lastDumpAddr += Index;
}
if (DumpSo) {
// Fix the issue that __tls_get_addr appear as file offset 0.
// Old lld version the __tls_get_addr appear at the last function name.
    std::pair<uint64_t, StringRef> dummy(soLastPrintAddr, "dummy");
    fprintf(fd_so_func_offset, "%02x ", (uint8_t)(dummy.first >> 24));
    fprintf(fd_so_func_offset, "%02x ", (uint8_t)((dummy.first >> 16) & 0xFF));
    fprintf(fd_so_func_offset, "%02x ", (uint8_t)((dummy.first >> 8) & 0xFF));
    fprintf(fd_so_func_offset, "%02x ", (uint8_t)(dummy.first) & 0xFF));
    std::string str = dummy.second.str();
    std::size_t idx = 0;
    std::size_t strSize = str.size();
    for (idx = 0, strSize = str.size(); idx < strSize; idx++) {
        fprintf(fd_so_func_offset, "%c%c ", hexdigit((str[idx] >> 4) & 0xF, true)
            , hexdigit(str[idx] & 0xF, true));
    }
    for (idx = strSize; idx < 48; idx++) {
        fprintf(fd_so_func_offset, "%02x ", 0);
    }
    fprintf(fd_so_func_offset, "/* %s */\n", dummy.second.begin());
    num_dyn_entry++;
    outs() << '\n' << "/*" << dummy.second << "*/\n";
}
if (DumpSo) {
    FILE *fd_num_dyn_entry;
    fd_num_dyn_entry = fopen("num_dyn_entry", "w");
    if (fd_num_dyn_entry != NULL) {
        fprintf(fd_num_dyn_entry, "%d\n", num_dyn_entry);
    }
    fclose(fd_num_dyn_entry);
}
}

```

```

#define DYNSYM_LIB_OFFSET 9

// Modified from PrintSectionContents()
static void PrintDataSections(const ObjectFile *o, uint64_t lastDumpAddr) {
    error_code ec;
    std::size_t addr, end;
    std::string Error;

    for (section_iterator si = o->begin_sections(),
         se = o->end_sections();
         si != se; si.increment(ec)) {
        if (error(ec)) return;
        StringRef Name;
        StringRef Contents;
        uint64_t BaseAddr;
        bool BSS;
        if (error(si->getName(Name))) continue;
        if (error(si->getContents(Contents))) continue;
        if (error(si->getAddress(BaseAddr))) continue;
        if (error(si->isBSS(BSS))) continue;

        if (Name == ".rodata" || Name == ".rodata1" || Name == ".data" ||
            Name == ".data1" || Name == ".sdata") {
            if (Contents.size() <= 0) {
                continue;
            }
            // Fill /*address*/ 00 00 00 00 between lastDumpAddr (= the address of last
            // end section + 1) and BaseAddr
            uint64_t ceilingLastAddr4 = ((lastDumpAddr + 3) / 4) * 4;
            assert((lastDumpAddr <= BaseAddr) && "lastDumpAddr must <= BaseAddr");
            // Fill /*address*/ bytes is odd for 4 by 00
            outs() << format("/*%04" PRIx64 " */", lastDumpAddr);
            if (ceilingLastAddr4 > BaseAddr) {
                for (std::size_t i = lastDumpAddr; i < BaseAddr; ++i) {
                    outs() << "00 ";
                }
                outs() << "\n";
                lastDumpAddr = BaseAddr;
            }
            else {
                for (std::size_t i = lastDumpAddr; i < ceilingLastAddr4; ++i) {
                    outs() << "00 ";
                }
                outs() << "\n";
                lastDumpAddr = ceilingLastAddr4;
            }
            // Fill /*address*/ 00 00 00 00 for 4 bytes (1 Cpu0 word size)
            for (addr = lastDumpAddr, end = BaseAddr; addr < end; addr += 4) {
                outs() << format("/*%04" PRIx64 " */", addr);
                outs() << format("%02" PRIx64 " ", 0) << format("%02" PRIx64 " ", 0) \
                    << format("%02" PRIx64 " ", 0) << format("%02" PRIx64 " ", 0) << '\n';
            }

            outs() << /*Contents of section */ << Name << ":\n";
            // Dump out the content as hex and printable ascii characters.
            for (std::size_t addr = 0, end = Contents.size(); addr < end; addr += 16) {
                outs() << format("/*%04" PRIx64 " */", BaseAddr + addr);
                // Dump line of hex.
            }
        }
    }
}

```

```

        for (std::size_t i = 0; i < 16; ++i) {
            if (i != 0 && i % 4 == 0)
                outs() << ' ';
            if (addr + i < end)
                outs() << hexdigit((Contents[addr + i] >> 4) & 0xF, true)
                << hexdigit(Contents[addr + i] & 0xF, true) << " ";
        }
        // Print ascii.
        outs() << "/*" << " ";
        for (std::size_t i = 0; i < 16 && addr + i < end; ++i) {
            if (std::isprint(static_cast<unsigned char>(Contents[addr + i]) & 0xFF))
                outs() << Contents[addr + i];
            else
                outs() << ".";
        }
        outs() << "*/" << "\n";
    }
    // save the end address of this section to lastDumpAddr
    lastDumpAddr = BaseAddr + Contents.size();
}
else if (Name == ".bss" || Name == ".sbss") {
    if (Contents.size() <= 0) {
        continue;
    }
    // Fill /*address*/ 00 00 00 00 between lastDumpAddr( = the address of last
    // end section + 1) and BaseAddr
    uint64_t ceilingLastAddr4 = ((lastDumpAddr + 3) / 4) * 4;
    assert((lastDumpAddr <= BaseAddr) && "lastDumpAddr must <= BaseAddr");
    // Fill /*address*/ bytes is odd for 4 by 00
    outs() << format("/*%04" PRIx64 " */", lastDumpAddr);
    if (ceilingLastAddr4 > BaseAddr) {
        for (std::size_t i = lastDumpAddr; i < BaseAddr; ++i) {
            outs() << "00 ";
        }
        outs() << "\n";
        lastDumpAddr = BaseAddr;
    }
    else {
        for (std::size_t i = lastDumpAddr; i < ceilingLastAddr4; ++i) {
            outs() << "00 ";
        }
        outs() << "\n";
        lastDumpAddr = ceilingLastAddr4;
    }
    // Fill /*address*/ 00 00 00 00 for 4 bytes (1 Cpu0 word size)
    for (addr = lastDumpAddr, end = BaseAddr; addr < end; addr += 4) {
        outs() << format("/*%04" PRIx64 " */", addr);
        outs() << format("%02" PRIx64 " ", 0) << format("%02" PRIx64 " ", 0) \
        << format("%02" PRIx64 " ", 0) << format("%02" PRIx64 " ", 0) << '\n';
    }
}

outs() << "/*Contents of section " << Name << " :*/\n";
// Dump out the content as hex and printable ascii characters.
for (std::size_t addr = 0, end = Contents.size(); addr < end; addr += 16) {
    outs() << format("/*%04" PRIx64 " */", BaseAddr + addr);
    // Dump line of hex.
    for (std::size_t i = 0; i < 16; ++i) {
        if (i != 0 && i % 4 == 0)

```

```

        outs() << ' ';
    if (addr + i < end)
        outs() << "00 ";
    }
    outs() << "\n";
}
// save the end address of this section to lastDumpAddr
lastDumpAddr = BaseAddr + Contents.size();
}

else if (DumpSo) {
    if (Name == ".dynsym") {
        int num_dyn_entry = 0;
        FILE *fd_num_dyn_entry;
        fd_num_dyn_entry = fopen("num_dyn_entry", "r");
        if (fd_num_dyn_entry != NULL) {
            fscanf(fd_num_dyn_entry, "%d", &num_dyn_entry);
        }
        fclose(fd_num_dyn_entry);
        raw_fd_ostream fd_dynsym("dynsym", Error);
        int count = 0;
        for (std::size_t addr = 0, end = Contents.size(); addr < end; addr += 16) {
            fd_dynsym << hexdigit((Contents[addr] >> 4) & 0xF, true)
                << hexdigit(Contents[addr] & 0xF, true) << " ";
            fd_dynsym << hexdigit((Contents[addr+1] >> 4) & 0xF, true)
                << hexdigit(Contents[addr+1] & 0xF, true) << " ";
            fd_dynsym << hexdigit((Contents[addr+2] >> 4) & 0xF, true)
                << hexdigit(Contents[addr+2] & 0xF, true) << " ";
            fd_dynsym << hexdigit((Contents[addr+3] >> 4) & 0xF, true)
                << hexdigit(Contents[addr+3] & 0xF, true) << " ";
            count++;
        }
        for (int i = count; i < num_dyn_entry; i++) {
            fd_dynsym << "00 00 00 00 ";
        }
    }
    else if (Name == ".dynstr") {
        raw_fd_ostream fd_dynstr("dynstr", Error);
        raw_fd_ostream fd_dynstrAscii("dynstrAscii", Error);
        for (std::size_t addr = 0, end = Contents.size(); addr < end; addr++) {
            fd_dynstr << hexdigit((Contents[addr] >> 4) & 0xF, true)
                << hexdigit(Contents[addr] & 0xF, true) << " ";
            if (addr == 0)
                continue;
            if (Contents[addr] == '\0')
                fd_dynstrAscii << "\n";
            else
                fd_dynstrAscii << Contents[addr];
        }
    }
    else if (!DumpSo) {
        if (Name == ".got.plt") {
            uint64_t BaseAddr;
            if (error(si->getAddress(BaseAddr)))
                assert(1 && "Cannot get BaseAddr of section .got.plt");
            raw_fd_ostream fd_global_offset("global_offset", Error);
            fd_global_offset << format("%02" PRIx64 " ", BaseAddr >> 24);
            fd_global_offset << format("%02" PRIx64 " ", (BaseAddr >> 16) & 0xFF);
        }
    }
}

```

```

        fd_global_offset << format("%02" PRIx64 " ", (BaseAddr >> 8) & 0xFF);
        fd_global_offset << format("%02" PRIx64 " ", BaseAddr & 0xFF);
    }
}
}
}

static void Elf2Hex(const ObjectFile *o) {
    uint64_t startAddr = GetSectionHeaderStartAddress(o, "_start");
// outs() << format("_start address:%08" PRIx64 "\n", startAddr);
    uint64_t lastDumpAddr = 0;
    if (LinkSo)
        cpu0DynFunIndex.createPltName(o);
    DisassembleObjectInHexFormat(o, lastDumpAddr);
// outs() << format("lastDumpAddr:%08" PRIx64 "\n", lastDumpAddr);
    PrintDataSections(o, lastDumpAddr);
}

```

### lbdex/llvm-objdump/llvm-objdump.cpp

```

#include "elf2hex.h"

static void DumpObject(const ObjectFile *o) {
    outs() << '\n';
    if (ConvertElf2Hex)
        outs() << /*/;
    outs() << o->getFileName()
        << ":\tfile format " << o->getFormatName();
    if (ConvertElf2Hex)
        outs() << /*/;
    outs() << "\n\n";

    if (Disassemble)
        DisassembleObject(o, Relocations);
    if (Relocations && !Disassemble)
        PrintRelocations(o);
    if (SectionHeaders)
        PrintSectionHeaders(o);
    if (SectionContents)
        PrintSectionContents(o);
    if (ConvertElf2Hex)
        Elf2Hex(o);
    if (SymbolTable)
        PrintSymbolTable(o);
    if (UnwindInfo)
        PrintUnwindInfo(o);
    if (PrivateHeaders)
        printPrivateFileHeader(o);
}

int main(int argc, char **argv) {
    // Print a stack trace if we signal out.
    sys::PrintStackTraceOnErrorSignal();
    PrettyStackTraceProgram X(argc, argv);
    llvm_shutdown_obj Y; // Call llvm_shutdown() on exit.
}

```

```
// Initialize targets and assembly printers/parsers.
llvm::InitializeAllTargetInfos();
llvm::InitializeAllTargetMCs();
llvm::InitializeAllAsmParsers();
llvm::InitializeAllDisassemblers();

// Register the target printer for --version.
cl::AddExtraVersionPrinter(TargetRegistry::printRegisteredTargetsForVersion);

cl::ParseCommandLineOptions(argc, argv, "llvm object file dumper\n");
TripleName = Triple::normalize(TripleName);

ToolName = argv[0];

// Defaults to a.out if no filenames specified.
if (InputFilenames.size() == 0)
    InputFilenames.push_back("a.out");

if (!Disassemble
    && !Relocations
    && !SectionHeaders
    && !SectionContents
    && !ConvertElf2Hex
    && !SymbolTable
    && !UnwindInfo
    && !PrivateHeaders) {
    cl::PrintHelpMessage();
    return 2;
}

std::for_each(InputFilenames.begin(), InputFilenames.end(),
              DumpInput);

return 0;
}
```

The “if (DumpSo)” and “if (LinkSo)” included code are for dynamic linker support. Others are used in both static and dynamic link execution file dump.

## 13.4 Static linker

Let's run the static linker first and explain it next.

### 13.4.1 Run

File printf-stdarg.c came from internet download which is GPL2 license. GPL2 is more restricted than LLVM license. File printf-stdarg-2.c is modified from printf-stdarg.c of printf() function supplied and add some test function for /demo/verification/debugpurpose on Cpu0 backend. File printf-stdarg-1.c is file for testing the printf() function implemented on PC OS platform. Let's run printf-stdarg-2.c on Cpu0 and compare with the result of printf() function which implemented by PC OS as below.

### lbdex/InputFiles/printf-stdarg-1.c

```
/*
Copyright 2001, 2002 Georges Menie (www.menie.org)
stdarg version contributed by Christian Ettinger

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU Lesser General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*/
/*
putchar is the only external dependency for this file,
if you have a working putchar, leave it commented out.
If not, uncomment the define below and
replace outbyte(c) by your own function call.

#define putchar(c) outbyte(c)
*/
// gcc printf-stdarg-1.c
// ./a.out

#include <stdio.h>

#define TEST_PRINTF

#ifdef TEST_PRINTF
int main(void)
{
    char *ptr = "Hello world!";
    char *np = 0;
    int i = 5;
    unsigned int bs = sizeof(int)*8;
    int mi;
    char buf[80];

    mi = (1 << (bs-1)) + 1;
    printf("%s\n", ptr);
    printf("printf test\n");
    printf("%s is null pointer\n", np);
    printf("%d = 5\n", i);
    printf("%d = - max int\n", mi);
    printf("char %c = 'a'\n", 'a');
    printf("hex %x = ff\n", 0xff);
    printf("hex %02x = 00\n", 0);
    printf("signed %d = unsigned %u = hex %x\n", -3, -3, -3);
    printf("%d %s(s)%", 0, "message");
}
```

```

printf("\n");
printf("%d %s(s) with %%\n", 0, "message");
sprintf(buf, "justif: \"%-10s\"\n", "left"); printf("%s", buf);
sprintf(buf, "justif: \"%10s\"\n", "right"); printf("%s", buf);
sprintf(buf, " 3: %04d zero padded\n", 3); printf("%s", buf);
sprintf(buf, " 3: %-4d left justif.\n", 3); printf("%s", buf);
sprintf(buf, " 3: %4d right justif.\n", 3); printf("%s", buf);
sprintf(buf, "-3: %04d zero padded\n", -3); printf("%s", buf);
sprintf(buf, "-3: %-4d left justif.\n", -3); printf("%s", buf);
sprintf(buf, "-3: %4d right justif.\n", -3); printf("%s", buf);

return 0;
}

/*
* if you compile this file with
*   gcc -Wall $(YOUR_C_OPTIONS) -DTEST_PRINTF -c printf.c
* you will get a normal warning:
*   printf.c:214: warning: spurious trailing '%' in format
* this line is testing an invalid % at the end of the format string.
*
* this should display (on 32bit int machine) :
*
* Hello world!
* printf test
* (null) is null pointer
* 5 = 5
* -2147483647 = - max int
* char a = 'a'
* hex ff = ff
* hex 00 = 00
* signed -3 = unsigned 4294967293 = hex ffffffff
* 0 message(s)
* 0 message(s) with %
* justif: "left      "
* justif: "      right"
* 3: 0003 zero padded
* 3: 3    left justif.
* 3:    3 right justif.
* -3: -003 zero padded
* -3: -3    left justif.
* -3:    -3 right justif.
*/
#endif

```

### Ibdex/InputFiles/printf-stdarg-2.c

```

#include "print.h"

extern int printf(const char *format, ...);
extern int sprintf(char *out, const char *format, ...);

struct Time
{
    int hour;
    int minute;

```

```
int second;
};

struct Date
{
    int year;
    int month;
    int day;
    int hour;
    int minute;
    int second;
};

int gI = 100;
struct Date gDate = {2012, 10, 12, 1, 2, 3};

int test_global()
{
    return gI;
}

struct Date copyDate(struct Date date)
{
    return date;
}

struct Time copyTime(struct Time time)
{
    return time;
}

// For memory IO
int putchar(const char c)
{
    char *p = (char*)OUT_MEM;
    *p = c;

    return 0;
}

int main(void)
{
    char *ptr = "Hello world!";
    char *np = 0;
    int i = 5;
    unsigned int bs = sizeof(int)*8;
    int mi;
    char buf[80];

    int a = 0;
    struct Time timel = {1, 10, 12};
    struct Time time2;
    struct Date date;

    a = test_global(); // gI = 100
    printf("global variable gI = %d\n", a);
    printf("timel = %d %d %d\n", timel.hour, timel.minute, timel.second);
    date = copyDate(gDate);
```

```

printf("date = %d %d %d %d %d\n", date.year, date.month, date.day, date.hour, date.minute, date
time2 = copyTime(time1); // test return V0, V1, A0
printf("time2 = %d %d %d\n", time2.hour, time2.minute, time2.second);

mi = (1 << (bs-1)) + 1;
printf("%s\n", ptr);
printf("printf test\n");
printf("%s is null pointer\n", np);
printf("%d = 5\n", i);
printf("%d = - max int\n", mi);
printf("char %c = 'a'\n", 'a');
printf("hex %x = ff\n", 0xff);
printf("hex %02x = 00\n", 0);
printf("signed %d = unsigned %u = hex %x\n", -3, -3, -3);
printf("%d %s(s)%", 0, "message");
printf("\n");
printf("%d %s(s) with %%\n", 0, "message");
sprintf(buf, "justif: \\"%-10s\\"", "left"); printf("%s", buf);
sprintf(buf, "justif: \\"%10s\\"", "right"); printf("%s", buf);
sprintf(buf, " 3: %04d zero padded\n", 3); printf("%s", buf);
sprintf(buf, " 3: %-4d left justif.\n", 3); printf("%s", buf);
sprintf(buf, " 3: %4d right justif.\n", 3); printf("%s", buf);
sprintf(buf, "-3: %04d zero padded\n", -3); printf("%s", buf);
sprintf(buf, "-3: %-4d left justif.\n", -3); printf("%s", buf);
sprintf(buf, "-3: %4d right justif.\n", -3); printf("%s", buf);

return 0;
}

```

### Index/InputFiles/printf-stdarg.c

```

/*
Copyright 2001, 2002 Georges Menie (www.menie.org)
stdarg version contributed by Christian Ettinger

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU Lesser General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
*/
/*
putchar is the only external dependency for this file,
if you have a working putchar, leave it commented out.
If not, uncomment the define below and
replace outbyte(c) by your own function call.

```

```

#define putchar(c) outbyte(c)
*/

#include <stdarg.h>

static void printchar(char **str, int c)
{
    extern int putchar(int c);

    if (*str) {
        **str = c;
        ++(*str);
    }
    else (void)putchar(c);
}

#define PAD_RIGHT 1
#define PAD_ZERO 2

static int prints(char **out, const char *string, int width, int pad)
{
    register int pc = 0, padchar = ' ';

    if (width > 0) {
        register int len = 0;
        register const char *ptr;
        for (ptr = string; *ptr; ++ptr) ++len;
        if (len >= width) width = 0;
        else width -= len;
        if (pad & PAD_ZERO) padchar = '0';
    }
    if (!(pad & PAD_RIGHT)) {
        for ( ; width > 0; --width) {
            printchar (out, padchar);
            ++pc;
        }
    }
    for ( ; *string ; ++string) {
        printchar (out, *string);
        ++pc;
    }
    for ( ; width > 0; --width) {
        printchar (out, padchar);
        ++pc;
    }
}

return pc;
}

/* the following should be enough for 32 bit int */
#define PRINT_BUF_LEN 12

static int printi(char **out, int i, int b, int sg, int width, int pad, int letbase)
{
    char print_buf[PRINT_BUF_LEN];
    register char *s;
    register int t, neg = 0, pc = 0;
    register unsigned int u = i;

```

```
if (i == 0) {
    print_buf[0] = '0';
    print_buf[1] = '\0';
    return prints (out, print_buf, width, pad);
}

if (sg && b == 10 && i < 0) {
    neg = 1;
    u = -i;
}

s = print_buf + PRINT_BUF_LEN-1;
*s = '\0';

while (u) {
    t = u % b;
    if( t >= 10 )
        t += letbase - '0' - 10;
    *--s = t + '0';
    u /= b;
}

if (neg) {
    if( width && (pad & PAD_ZERO) ) {
        printchar (out, '-');
        ++pc;
        --width;
    }
    else {
        *--s = '-';
    }
}

return pc + prints (out, s, width, pad);
}

static int print(char **out, const char *format, va_list args )
{
    register int width, pad;
    register int pc = 0;
    char scr[2];

    for ( ; *format != 0; ++format) {
        if (*format == '%') {
            ++format;
            width = pad = 0;
            if (*format == '\0') break;
            if (*format == '%') goto out;
            if (*format == '-') {
                ++format;
                pad = PAD_RIGHT;
            }
            while (*format == '0') {
                ++format;
                pad |= PAD_ZERO;
            }
            for ( ; *format >= '0' && *format <= '9'; ++format) {
                width *= 10;
                if (width >= 1000) {
                    if (pad & PAD_ZERO) {
                        if (*format == '\0') break;
                        if (*format == '%') goto out;
                    }
                    printchar (out, '0');
                    width -= 1000;
                }
            }
        }
    }
}
```

```

                width += *format - '0';
            }
            if( *format == 's' ) {
                register char *s = (char *)va_arg( args, int );
                pc += prints( out, s?s:"(null)", width, pad);
                continue;
            }
            if( *format == 'd' ) {
                pc += printi( out, va_arg( args, int ), 10, 1, width, pad, 'a');
                continue;
            }
            if( *format == 'x' ) {
                pc += printi( out, va_arg( args, int ), 16, 0, width, pad, 'a');
                continue;
            }
            if( *format == 'X' ) {
                pc += printi( out, va_arg( args, int ), 16, 0, width, pad, 'A');
                continue;
            }
            if( *format == 'u' ) {
                pc += printi( out, va_arg( args, int ), 10, 0, width, pad, 'a');
                continue;
            }
            if( *format == 'c' ) {
                /* char are converted to int then pushed on the stack */
                scr[0] = (char)va_arg( args, int );
                scr[1] = '\0';
                pc += prints( out, scr, width, pad);
                continue;
            }
        }
        else {
            out:
                printchar( out, *format );
                ++pc;
        }
    }
    if (out) **out = '\0';
    va_end( args );
    return pc;
}

int printf(const char *format, ...)
{
    va_list args;

    va_start( args, format );
    return print( 0, format, args );
}

int sprintf(char *out, const char *format, ...)
{
    va_list args;

    va_start( args, format );
    return print( &out, format, args );
}

```

```

#define TEST_PRINTF
int main(void)
{
    char *ptr = "Hello world!";
    char *np = 0;
    int i = 5;
    unsigned int bs = sizeof(int)*8;
    int mi;
    char buf[80];

    mi = (1 << (bs-1)) + 1;
    printf("%s\n", ptr);
    printf("printf test\n");
    printf("%s is null pointer\n", np);
    printf("%d = 5\n", i);
    printf("%d = - max int\n", mi);
    printf("char %c = 'a'\n", 'a');
    printf("hex %x = ff\n", 0xff);
    printf("hex %02x = 00\n", 0);
    printf("signed %d = unsigned %u = hex %x\n", -3, -3, -3);
    printf("%d %s(s)%", 0, "message");
    printf("\n");
    printf("%d %s(s) with %%\n", 0, "message");
    sprintf(buf, "justif: \"%-10s\"\n", "left"); printf("%s", buf);
    sprintf(buf, "justif: \"%10s\"\n", "right"); printf("%s", buf);
    sprintf(buf, " 3: %04d zero padded\n", 3); printf("%s", buf);
    sprintf(buf, " 3: %-4d left justif.\n", 3); printf("%s", buf);
    sprintf(buf, " 3: %4d right justif.\n", 3); printf("%s", buf);
    sprintf(buf, "-3: %04d zero padded\n", -3); printf("%s", buf);
    sprintf(buf, "-3: %-4d left justif.\n", -3); printf("%s", buf);
    sprintf(buf, "-3: %4d right justif.\n", -3); printf("%s", buf);

    return 0;
}

/*
* if you compile this file with
*   gcc -Wall $(YOUR_C_OPTIONS) -DTEST_PRINTF -c printf.c
* you will get a normal warning:
*   printf.c:214: warning: spurious trailing '%' in format
* this line is testing an invalid % at the end of the format string.
*
* this should display (on 32bit int machine) :
*
* Hello world!
* printf test
* (null) is null pointer
* 5 = 5
* -2147483647 = - max int
* char a = 'a'
* hex ff = ff
* hex 00 = 00
* signed -3 = unsigned 4294967293 = hex ffffffd
* 0 message(s)
* 0 message(s) with %
* justif: "left      "
* justif: "      right"
* 3: 0003 zero padded

```

```

* 3: 3    left justif.
* 3: 3 right justif.
* -3: -003 zero padded
* -3: -3 left justif.
* -3: -3 right justif.
*/
#endif

```

### lbdex/InputFiles/start.cpp

```

#include "dynamic_linker.h"

extern int main();

#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 $6,      $ZERO, 0"); \
asm("addiu $7,      $ZERO, 0"); \
asm("addiu $8,      $ZERO, 0"); \
asm("addiu $9,      $ZERO, 0"); \
asm("addiu $10,     $ZERO, 0"); \
asm("addiu $fp, $ZERO, 0");

void start() {
//  asm("boot:");
    asm("lui    $1, 0x7");
    asm("ori    $1, $1, 0xffff0");
    asm("ld     $gp, 0($1)"); // load $gp($10) value from 0x7fff0
    initRegs();
    asm("addiu $sp, $zero, 0x6ffc");
    main();
    asm("addiu $lr, $ZERO, -1");
    asm("ret $lr");
}

```

### lbdex/InputFiles/build-printf-stdarg-2.sh

```

#!/usr/bin/env bash
#TOOLDIR=/home/Gamma/test/lld/cmake_debug_build/bin
TOOLDIR=/home/cschen/test/lld/cmake_debug_build/bin

cpu=cpu032I

/usr/local/llvm/release/cmake_debug_build/bin/clang -target mips-unknown-linux-gnu -c start.cpp -emit-obj
/usr/local/llvm/release/cmake_debug_build/bin/clang -target mips-unknown-linux-gnu -c printf-stdarg.o
/usr/local/llvm/release/cmake_debug_build/bin/clang -target mips-unknown-linux-gnu -c printf-stdarg-2.o
${TOOLDIR}/llc -march=cpu0 -mcpu=${cpu} -relocation-model=static -filetype=obj start.bc -o start.cpu
${TOOLDIR}/llc -march=cpu0 -mcpu=${cpu} -relocation-model=static -filetype=obj printf-stdarg.bc -o printf-stdarg.cpu
${TOOLDIR}/llc -march=cpu0 -mcpu=${cpu} -relocation-model=static -filetype=obj printf-stdarg-2.bc -o printf-stdarg-2.cpu

```

```
 ${TOOLDIR}/lld -flavor gnu -target cpu0-unknown-linux-gnu start.cpu0.o printf-stdarg(cpu0.o printf-stdarg.c
 ${TOOLDIR}/llvm-objdump -elf2hex a.out > ../cpu0_verilog/cpu0.hex
```

The `cpu0_verilog/cpu0Is.v` support `cmp` instruction and static linker as follows,

#### **lbdex/cpu0\_verilog/cpu0Is.v**

```
// TRACE: Display the memory contents of the loaded program and data
//`define TRACE

`include "cpu0.v"
```

The `cpu0_verilog/cpu0IIs.v` support `slt` instruction and static linker as follows,

#### **lbdex/cpu0\_verilog/cpu0IIs.v**

```
`define CPU0II
// TRACE: Display the memory contents of the loaded program and data
//`define TRACE

`include "cpu0.v"
```

The `build-printf-stdarg-2.sh` is for my PC setting. Please change this script to your `lld` installed directory and run static linker example code as follows,

```
[Gamma@localhost cpu0_verilog]$ pwd
/home/Gamma/test/lbd/docs/BackendTutorial/source_ExampleCode/cpu0_verilog
[Gamma@localhost cpu0_verilog]$ bash clean.sh
[Gamma@localhost InputFiles]$ cd ../InputFiles/
[Gamma@localhost InputFiles]$ bash build-printf-stdarg-2.sh
printf-stdarg-2.c:85:19: warning: incomplete format specifier [-Wformat]
printf("%d %s(s)%", 0, "message");
^
1 warning generated.
[Gamma@localhost InputFiles]$ cd ../cpu0_verilog/
[Gamma@localhost cpu0_verilog]$ pwd
/home/Gamma/test/lbd/docs/BackendTutorial/source_ExampleCode/cpu0_verilog
[Gamma@localhost cpu0_verilog]$ iverilog -o cpu0IIs cpu0IIs.v
[Gamma@localhost cpu0_verilog]$ ls
clean.sh cpu0Id.v cpu0IId.v cpu0IIs.v cpu0IIs.v cpu0Is.v cpu0.v dynlinker.v
flashio.v
[Gamma@localhost cpu0_verilog]$ ./cpu0IIs
WARNING: cpu0.v:365: $readmemh(cpu0s.hex): Not enough words in the file for
the requested range [0:524287].
taskInterrupt(001)
global variable gI = 100
time1 = 1 10 12
date = 2012 10 12 1 2 3
time2 = 1 10 12
Hello world!
printf test
(null) is null pointer
5 = 5
-2147483647 = - max int
char a = 'a'
hex ff = ff
```

```
hex 00 = 00
signed -3 = unsigned 4294967293 = hex ffffffd
0 message(s)
0 message(s) with \%
justif: "left"
justif: "right"
3: 0003 zero padded
3: 3 left justif.
3: 3 right justif.
-3: -003 zero padded
```

Let's check the result with PC program printf-stdarg-1.c output as follows,

```
[Gamma@localhost InputFiles]$ gcc printf-stdarg-1.c
/usr/lib/gcc/x86_64-redhat-linux/4.7.2/../../../../lib64/crt1.o: In function
`_start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
[Gamma@localhost InputFiles]$ gcc printf-stdarg-1.c
[Gamma@localhost InputFiles]$ ./a.out
Hello world!
printf test
(null) is null pointer
5 = 5
-2147483647 = - max int
char a = 'a'
hex ff = ff
hex 00 = 00
signed -3 = unsigned 4294967293 = hex ffffffd
0 message(s)
0 message(s) with \%
justif: "left"
justif: "right"
3: 0003 zero padded
3: 3 left justif.
3: 3 right justif.
-3: -003 zero padded
-3: -3 left justif.
-3: -3 right justif.
```

They are same after the “Hello world!” of printf() function support. The cpu0I use cmp instruction. You can verify the slt instructions is work fine too by change cpu to cpu032II as follows,

### lbdx/InputFiles/build-printf-stdarg-2.sh

```
...
cpu=cpu032II
...

[Gamma@localhost cpu0_verilog]$ pwd
/home/Gamma/test/lbd/docs/BackendTutorial/source_ExampleCode/cpu0_verilog
[Gamma@localhost cpu0_verilog]$ bash clean.sh
[Gamma@localhost InputFiles]$ cd ../InputFiles/
[Gamma@localhost InputFiles]$ bash build-printf-stdarg-2.sh
printf-stdarg.c:102:19: warning: incomplete format specifier [-Wformat]
    printf("%d %s\b(s)\%\b", 0, "message");
           ^
```

```
1 warning generated.
[Gamma@localhost cpu0_verilog]$ ./cpu0IIs
```

The verilog machine cpu032IIs include cpu0I all instructions (cmp, jeq, ... are included also) and add Chapter12\_2 slt, beq, ..., instructions. Run build-printf-stdarg-2.sh with cpu=cpu032II will generate slt, beq and bne instructions instead cmp, jeq, ... instructions. Since cpu0IIs include both slt, cmp, ... instructions, the slt and cmp both code generated can be run on it without any problem.

### 13.4.2 Cpu0 lld structure

The Cpu0LinkingContext include the context information for those input obj files and output elf file you want to link. When do linking, the following code will create Cpu0LinkingContext.

[lbdex/Cpu0\\_Ild\\_1030/ELFLinkingContext.h](#)

```
class ELFLinkingContext : public LinkingContext {
public:
    ...
    static std::unique_ptr<ELFLinkingContext> create(llvm::Triple);
    ...
}
```

[lbdex/Cpu0\\_Ild\\_1030/ELFLinkingContext.cpp](#)

```
std::unique_ptr<ELFLinkingContext>
ELFLinkingContext::create(llvm::Triple triple) {
    switch (triple.getArch()) {
        ...
        case llvm::Triple::cpu0:
            return std::unique_ptr<ELFLinkingContext>(
                new lld::elf::Cpu0LinkingContext(triple));
        default:
            return nullptr;
    }
}
```

While Cpu0LinkingContext is created by lld ELF driver as above, the following code in Cpu0LinkingContext constructor will create Cpu0TargetHandler and passing the Cpu0LinkingContext object pointer to Cpu0TargetHandler.

[lbdex/Cpu0\\_Ild\\_1030/Cpu0/Cpu0LinkingContext.h](#)

```
class Cpu0LinkingContext LLVM_FINAL : public ELFLinkingContext {
public:
    Cpu0LinkingContext(llvm::Triple triple)
        : ELFLinkingContext(triple, std::unique_ptr<TargetHandlerBase>(
            new Cpu0TargetHandler(*this))) {}
    ...
}
```

Finally, the Cpu0TargetHandler constructor will create other related objects and set up the relation reference object pointers as [Figure 13.1](#) depicted by the following code.

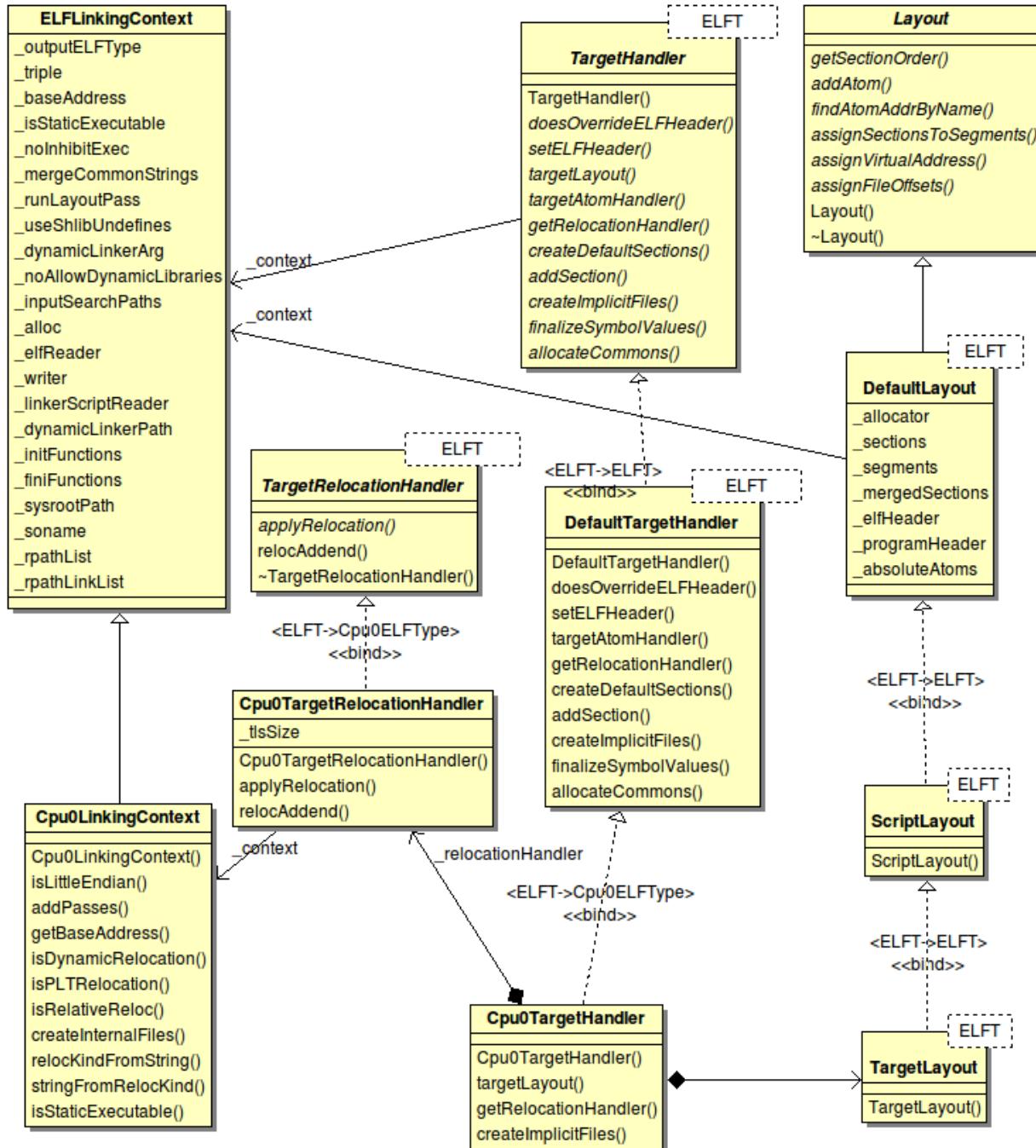


Figure 13.1: Cpu0 lld class relationship

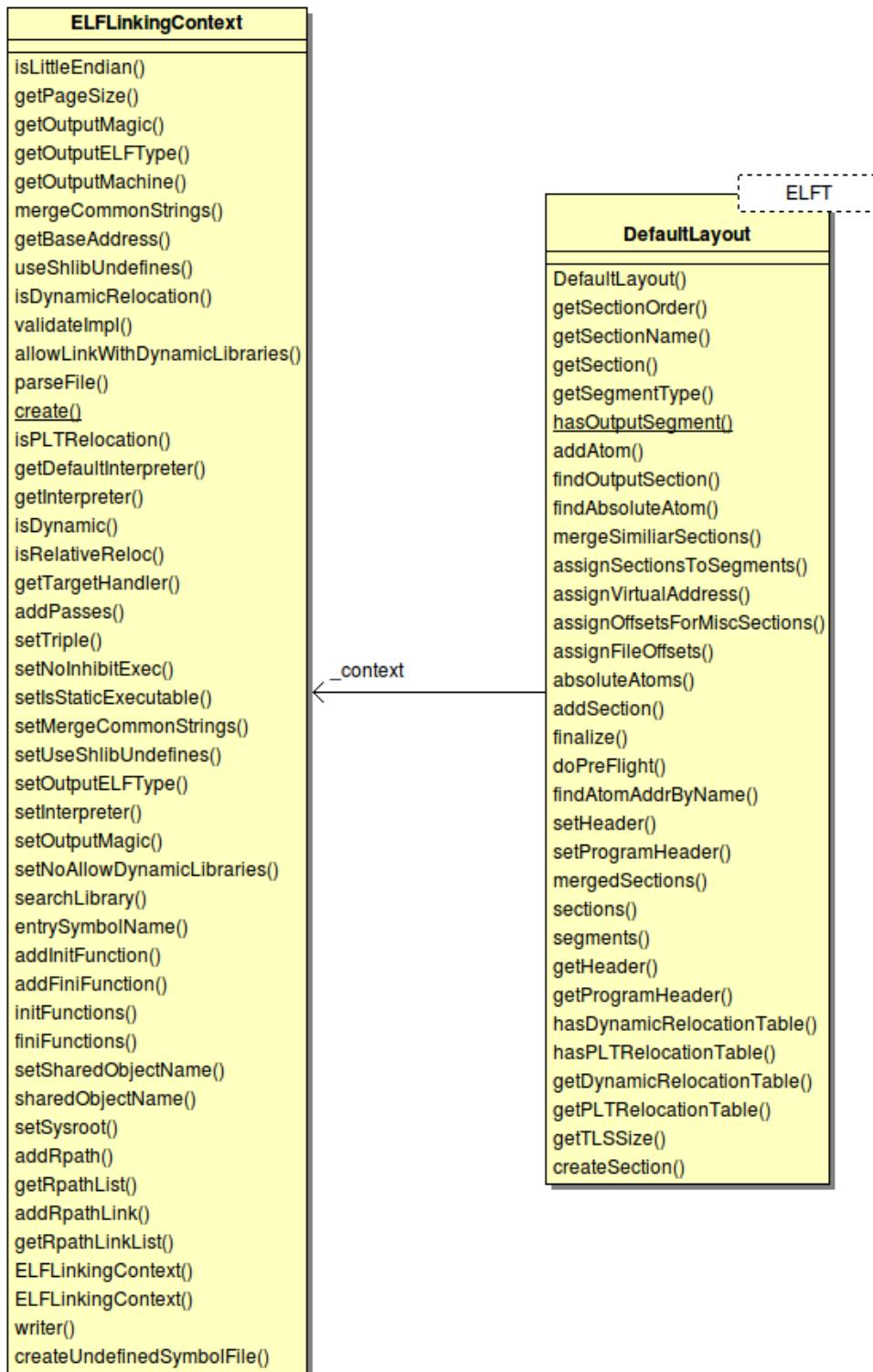


Figure 13.2: Cpu0 lld ELFLinkingContext and DefaultLayout member functions

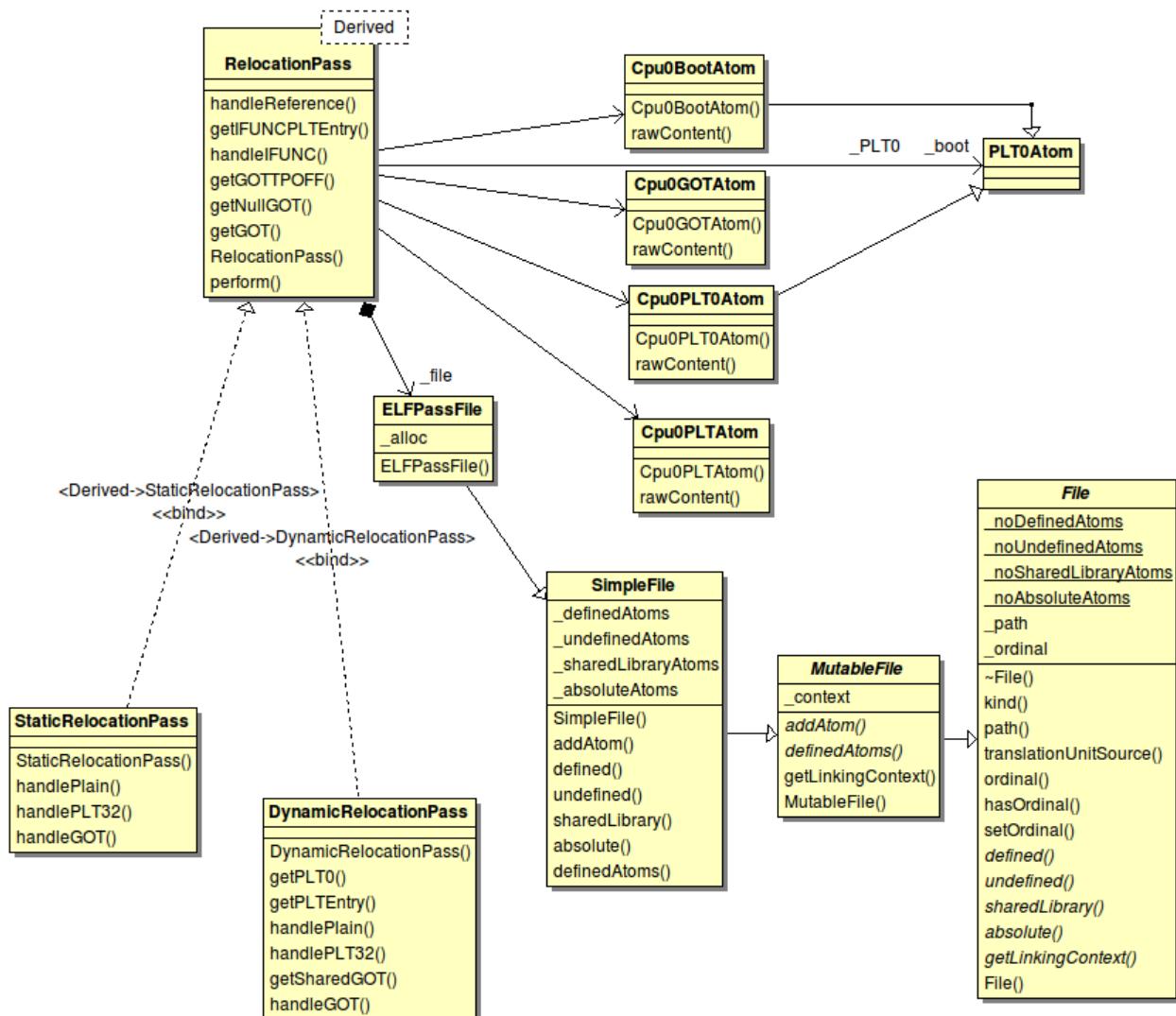


Figure 13.3: Cpu0 lld RelocationPass

### Ibdex/Cpu0\_Ild\_1030/Cpu0/Cpu0TargetHandler.cpp

```
Cpu0TargetHandler::Cpu0TargetHandler(Cpu0LinkingContext &context)
    : DefaultTargetHandler(context), _gotFile(new GOTFile(context)),
      _relocationHandler(context), _targetLayout(context) {}
```

According chapter ELF, the linker stands for resolve the relocation records. The following code give the chance to let lld system call our relocation function at proper time.

### Ibdex/Cpu0\_Ild\_1030/Cpu0/Cpu0RelocationPass.cpp

```
std::unique_ptr<Pass>
lld::createCpu0RelocationPass(const Cpu0LinkingContext &ctx) {
    switch (ctx.getOutputELFType()) {
        case llvm::ELF::ET_EXEC:
#ifdef DLINKER
            if (ctx.isDynamic())
                return std::unique_ptr<Pass> (new DynamicRelocationPass(ctx));
            else
#endif // DLINKER
            return std::unique_ptr<Pass> (new StaticRelocationPass(ctx));
#ifdef DLINKER
        case llvm::ELF::ET_DYN:
            return std::unique_ptr<Pass> (new DynamicRelocationPass(ctx));
#endif // DLINKER
        case llvm::ELF::ET_REL:
            return std::unique_ptr<Pass> ();
        default:
            llvm_unreachable("Unhandled output file type");
    }
}
```

The “#ifdef DLINKER” part is for dynamic linker will used in next section. For static linker, a StaticRelocationPass object is created and return.

Now the following code of Cpu0TargetRelocationHandler::applyRelocation() will be called through Cpu0TargetHandler by lld ELF driver when it meet each relocation record.

### Ibdex/Cpu0\_Ild\_1030/Cpu0/Cpu0RelocationHandler.cpp

```
ErrorOr<void> Cpu0TargetRelocationHandler::applyRelocation(
    ELFWriter &writer, llvm::FileOutputBuffer &buf, const lld::AtomLayout &atom,
    const Reference &ref) const {

    switch (ref.kind()) {
        case R_CPU0_NONE:
            break;
        case R_CPU0_HI16:
            relocHI16(location, relocVAddress, targetVAddress, ref.addend());
            break;
        case R_CPU0_LO16:
            relocLO16(location, relocVAddress, targetVAddress, ref.addend());
            break;
        ...
    }
}
```

```
    return error_code::success();  
}
```

### Ibdex/Cpu0\_Ild\_1030/Cpu0/Cpu0TargetHandler.h

```
class Cpu0TargetHandler LLVM_FINAL  
    : public DefaultTargetHandler<Cpu0ELFType> {  
public:  
    ...  
    virtual const Cpu0TargetRelocationHandler &getRelocationHandler() const {  
        return _relocationHandler;  
    }  
}
```

Summary as Figure 13.4.

Remind, static std::unique\_ptr<ELFLinkingContext> ELFLinkingContext::create(llvm::Triple) is called without an object of class ELFLinkingContext instance (because the static keyword). The Cpu0LinkingContext constructor will create it's ELFLinkingContext part. The std::unique\_ptr came from c++11 standard. The unique\_ptr objects automatically delete the object they manage (using a deleter) as soon as they themselves are destroyed. Just like the Singleton pattern in Design Pattern book or Smart Pointers in Effective C++ book.<sup>3</sup>

As Figure 13.1 depicted, the Cpu0TargetHandler include the members or pointers which can access to other object. The way to access Cpu0TargetHandler object from Cpu0LinkingContext or Cpu0RelocationHandler rely on LinkingContext::getTargetHandler() function. As Figure 13.5 depicted, the unique\_ptr point to Cpu0TargetHandler will be saved in LinkingContext constructor function.

---

**Note:** std::unique\_ptr::get()<sup>4</sup>

pointer get() const noexcept;

Get pointer Returns the stored pointer.

---

---

**Note:** std::move()<sup>5</sup>

**for example:** std::string bar = "bar-string"; std::move(bar);

bar is null after std::move(bar);

---

## 13.5 Dynamic linker

Except the lld code with #ifdef DLINKER. The following code in Verilog exist to support dynamic linker.

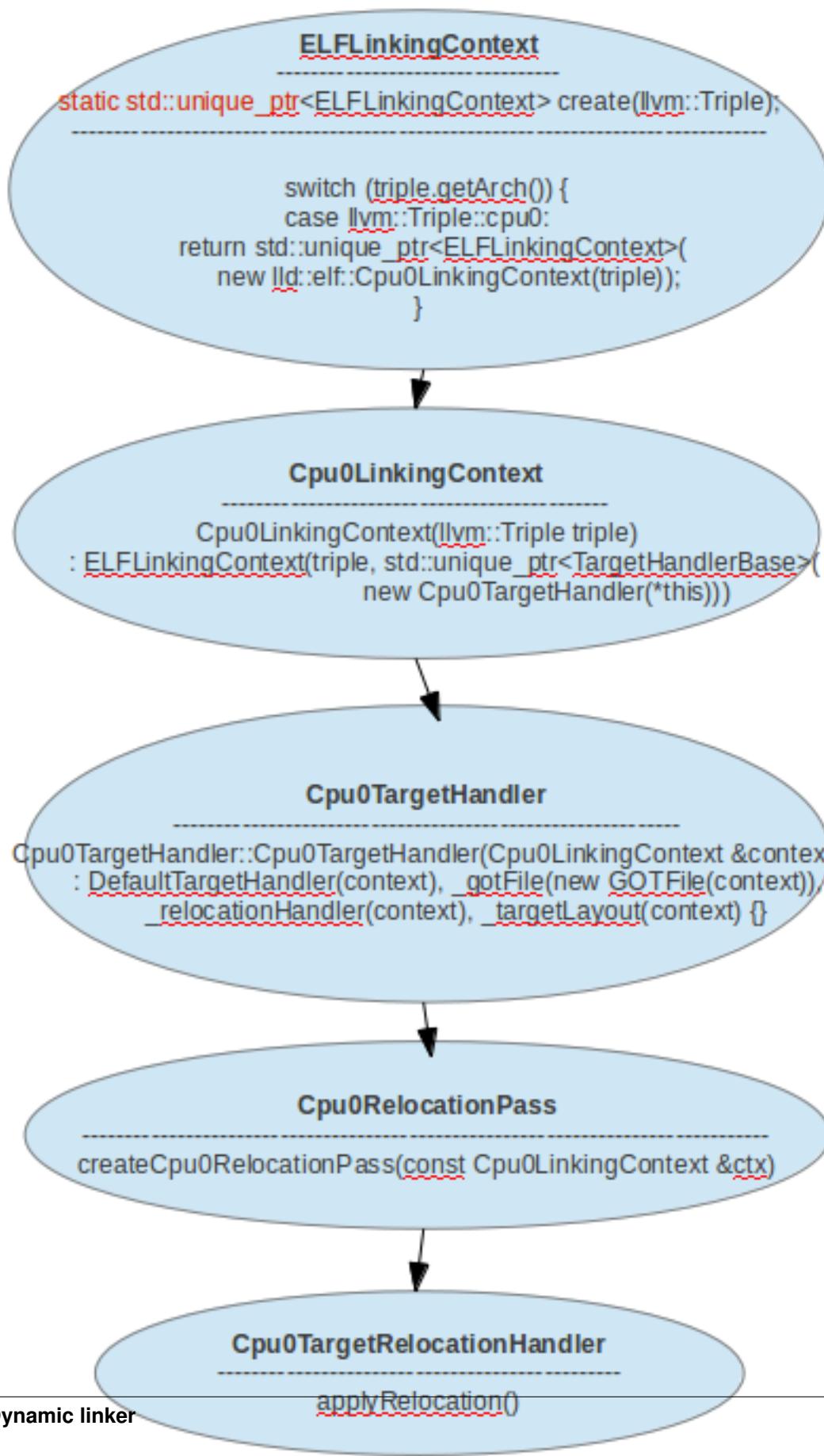
### Ibdex/cpu0\_verilog/dynlinker.v

```
'define DLINKER_INFO_ADDR  'h70000  
'define GPADDR      'h7FFF0  
  
'ifdef DLINKER
```

<sup>3</sup> [http://www.cplusplus.com/reference/memory/unique\\_ptr/](http://www.cplusplus.com/reference/memory/unique_ptr/)

<sup>4</sup> [http://www.cplusplus.com/reference/memory/unique\\_ptr/get/](http://www.cplusplus.com/reference/memory/unique_ptr/get/)

<sup>5</sup> <http://www.cplusplus.com/reference/utility/move/>



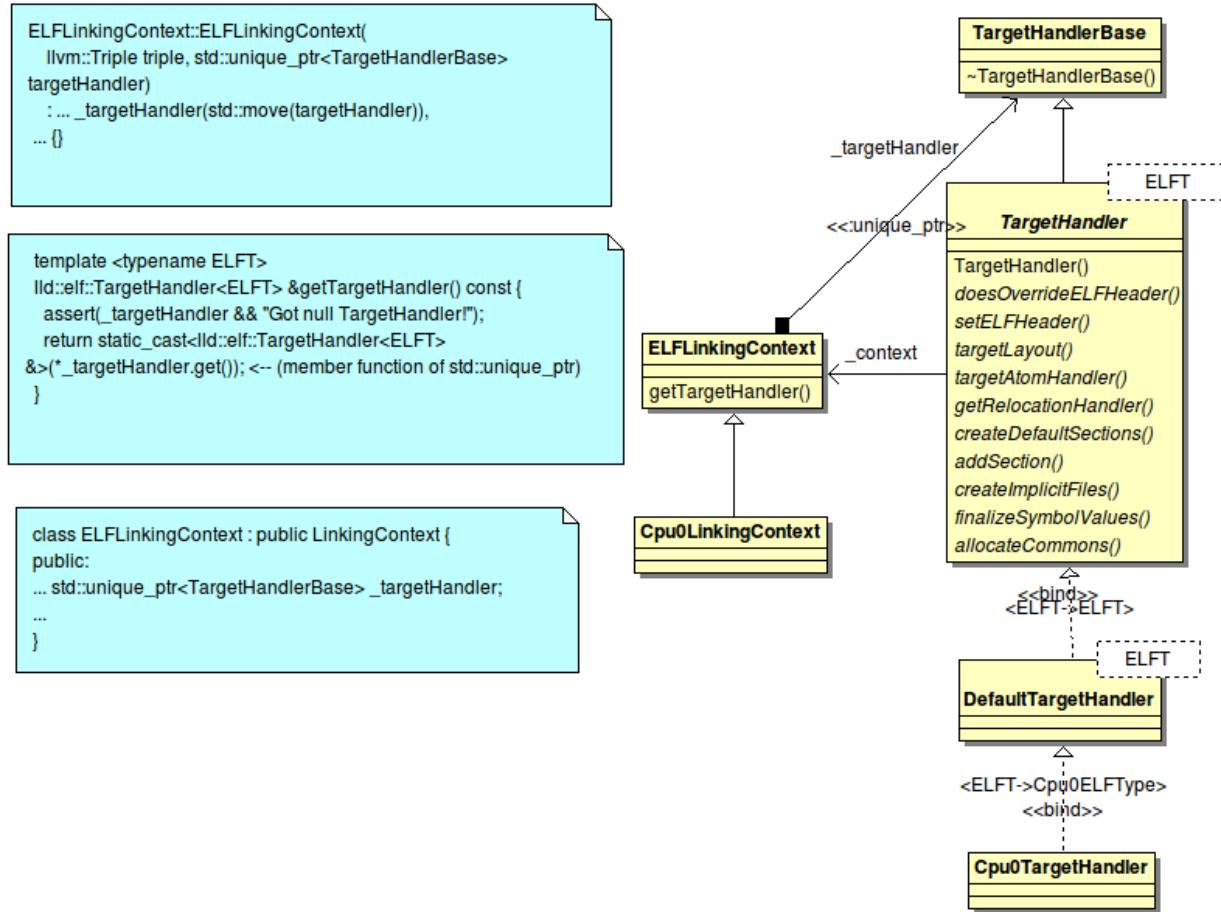


Figure 13.5: Cpu0LinkingContext get Cpu0TargetHandler through &getTargetHandler()

```

task setDynLinkerInfo; begin
// below code set memory as follows,
//                                     (4 bytes)
// -----
// DLINKER_INFO_ADDR -----> | numDynEntry | -----
// DLINKER_INFO_ADDR+4 -----> | index of dynsym (0st row) | -----
// above is the 1st word of section .dynsym of libfoobar.cpu0.so.
// DLINKER_INFO_ADDR+8 -----> | index of dynsym (1st row) | -----
// | ...
// DLINKER_INFO_ADDR+(numDynEntry-1)*4 -----> | index of dynsym (the last row) | -----
// DLINKER_INFO_ADDR+numDynEntry*4 -----> | 1st function (foo()) offset in lib | -----
// DLINKER_INFO_ADDR+numDynEntry*4+4 -----> | 1st function (foo()) name (48 bytes) | -----
// | ...
// DLINKER_INFO_ADDR+numDynEntry+(numDynEntry-1)*4 ---> | last function (foo()) offset in lib | -----
// DLINKER_INFO_ADDR+numDynEntry+(numDynEntry-1)*4+4 -> | last function (foo()) name | -----
// DLINKER_INFO_ADDR+4+numDynEntry*4+numDynEntry*52 --> | .dynstr of lib | -----
// | ...
// -----
// caculate number of dynamic entries
numDynEntry = 0;
j = 0;
for (i=0; i < 384 && j == 0; i=i+52) begin
  if (so_func_offset[i] == 'MEMEMPTY && so_func_offset[i+1] == 'MEMEMPTY &&
      so_func_offset[i+2] == 'MEMEMPTY && so_func_offset[i+3] == 'MEMEMPTY) begin
    numDynEntry = i/52;
    j = 1;
  'ifdef DEBUG_DLINKER
    $display("numDynEntry = %8x", numDynEntry);
  'endif
  end
end
// save number of dynamic entries to memory address 'DLINKER_INFO_ADDR
m['DLINKER_INFO_ADDR] = numDynEntry[31:24];
m['DLINKER_INFO_ADDR+1] = numDynEntry[23:16];
m['DLINKER_INFO_ADDR+2] = numDynEntry[15:8];
m['DLINKER_INFO_ADDR+3] = numDynEntry[7:0];
// copy section .dynsym of ELF to memory address 'DLINKER_INFO_ADDR+4
i = 'DLINKER_INFO_ADDR+4;
for (j=0; j < (4*numDynEntry); j=j+4) begin
  m[i] = dsym[j];
  m[i+1] = dsym[j+1];
  m[i+2] = dsym[j+2];
  m[i+3] = dsym[j+3];
  i = i + 4;
end
// copy the offset values of section .text of shared library .so of ELF to
// memory address 'DLINKER_INFO_ADDR+4+numDynEntry*4
i = 'DLINKER_INFO_ADDR+4+numDynEntry*4;
l = 0;
for (j=0; j < numDynEntry; j=j+1) begin
  for (k=0; k < 52; k=k+1) begin
    m[i] = so_func_offset[l];
    i = i + 1;
    l = l + 1;
  end

```

```

    end
`ifdef DEBUG_DLINKER
    i = `DLINKER_INFO_ADDR+4+numDynEntry*4;
    for (j=0; j < (8*numDynEntry); j=j+8) begin
        $display("%8x: %8x", i, {m[i], m[i+1], m[i+2], m[i+3]});
        i = i + 8;
    end
`endif
// copy section .dynstr of ELF to memory address
// `DLINKER_INFO_ADDR+4+numDynEntry*4+numDynEntry*52
i=`DLINKER_INFO_ADDR+4+numDynEntry*4+numDynEntry*52;
for (j=0; dstr[j] != 'MEMEMPTY; j=j+1) begin
    m[i] = dstr[j];
    i = i + 1;
end
`ifdef DEBUG_DLINKER
$display("In setDynLinkerInfo()");
for (i=`DLINKER_INFO_ADDR; i < `MEMSIZE; i=i+4) begin
    if (m[i] != 'MEMEMPTY || m[i+1] != 'MEMEMPTY ||
        m[i+2] != 'MEMEMPTY || m[i+3] != 'MEMEMPTY)
        $display("%8x: %8x", i, {m[i], m[i+1], m[i+2], m[i+3]});
end
$display("global address %8x", {m['GPADDR], m['GPADDR+1],
                                m['GPADDR+2], m['GPADDR+3]});
$display("gp = %8x", gp);
`endif
// below code set memory as follows,
// -----
// gp -----> | all 0 | (16 bytes)
// gp+16 -----> | 0 |
// gp+16+1*4 -----> | 1st plt entry address | (4 bytes)
// | ...
// gp+16+(numDynEntry-1)*4 -----> | the last plt entry address |
// -----
// gp -----> | all 0 | (16 bytes)
// gp+16+0*8'h10 -----> | 32'h10: pointer to plt0 |
// gp+16+1*8'h10 -----> | 1st plt entry |
// gp+16+2*8'h10 -----> | 2nd plt entry |
// | ...
// gp+16+(numDynEntry-1)*8'h10 --> | the last plt entry |
// -----
// note: gp point to the _GLOBAL_OFFSET_TABLE_,
//       numDynEntry = actual number of functions + 1.
//       gp+1*4..gp+numDynEntry*4 set to 8'h10 plt0 which will jump to dynamic
//       linker.
//       After dynamic linker load function to memory, it will set gp+index*4 to
//       function memory address. For example, if the function index is 2, then the
//       gp+2*4 is set to the memory address of this loaded function.
//       Then the the caller call
//       "ld $t9, 2*4($gp)" and "ret $t9" will jump to this loaded function directly.

gpPlt = gp+16+numDynEntry*4;
// set (gpPlt-16..gpPlt-1) to 0
for (j=16; j >= 1; j=j-1)
    m[gpPlt+j] = 8'h00;
// put plt in (gpPlt..gpPlt+numDynEntry*8'h10+1)
for (i=1; i < numDynEntry; i=i+1) begin
    j=i*4;

```

```

// (gp+'8h10..gp+numDynEntry*'8h10+15) set to plt entry
// addiu      $t9, $zero, dynsym_idx
m[gpPlt+i*8'h10] = 8'h09;
m[gpPlt+i*8'h10+1] = 8'h60;
m[gpPlt+i*8'h10+2] = i[15:8];
m[gpPlt+i*8'h10+3] = i[7:0];
// st      $t9, j($gp)
m[gpPlt+i*8'h10+4] = 8'h02;
m[gpPlt+i*8'h10+5] = 8'h6b;
m[gpPlt+i*8'h10+6] = 0;
m[gpPlt+i*8'h10+7] = 0;
// ld      $t9, ('16h0010)($gp)
m[gpPlt+i*8'h10+8] = 8'h01;
m[gpPlt+i*8'h10+9] = 8'h6b;
m[gpPlt+i*8'h10+10] = 0;
m[gpPlt+i*8'h10+11] = 8'h10;
// ret      $t9
m[gpPlt+i*8'h10+12] = 8'h3c;
m[gpPlt+i*8'h10+13] = 8'h60;
m[gpPlt+i*8'h10+14] = 0;
m[gpPlt+i*8'h10+15] = 0;
end

// .got.plt offset(0x00.0x03) has been set to 0 in elf already.
// Set .got.plt offset(8'h10..numDynEntry*'8h10) point to plt entry as above.
`ifdef DEBUG_DLINKER
    $display("numDynEntry = %8x", numDynEntry);
`endif
//      j32=32'h1fc0; // m[32'h1fc] = "something" will hang. Very tricky
m[gp+16] = 8'h0;
m[gp+16+1] = 8'h0;
m[gp+16+2] = 8'h0;
m[gp+16+3] = 8'h10;
j32=gpPlt+16;
for (i=1; i < numDynEntry; i=i+1) begin
    m[gp+16+i*4] = j32[31:24];
    m[gp+16+i*4+1] = j32[23:16];
    m[gp+16+i*4+2] = j32[15:8];
    m[gp+16+i*4+3] = j32[7:0];
    j32=j32+16;
end
`ifdef DEBUG_DLINKER
    // show (gp..gp+numDynEntry*4-1)
    for (i=0; i < numDynEntry; i=i+1) begin
        $display("%8x: %8x", gp+16+i*4, {m[gp+16+i*4], m[gp+16+i*4+1],
                                         m[gp+16+i*4+2], m[gp+16+i*4+3]});
    end
    // show (gpPlt..gpPlt+(numDynEntry+1)*8'h10-1)
    for (i=0; i < numDynEntry; i=i+1) begin
        for (j=0; j < 16; j=j+4)
            $display("%8x: %8x", gpPlt+i*8'h10+j,
                    {m[gpPlt+i*8'h10+j],
                     m[gpPlt+i*8'h10+j+1],
                     m[gpPlt+i*8'h10+j+2],
                     m[gpPlt+i*8'h10+j+3]});
    end
`endif
end endtask

```

```
'endif

`ifdef DLINKER
  task loadToFlash; begin
    // erase memory
    for (i=0; i < `MEMSIZE; i=i+1) begin
      flash[i] = 'MEMEMPTY;
    end
    $readmemh("libso.hex", flash);
`ifdef DEBUG_DLINKER
  for (i=0; i < `MEMSIZE && (flash[i] != 'MEMEMPTY ||
    flash[i+1] != 'MEMEMPTY || flash[i+2] != 'MEMEMPTY ||
    flash[i+3] != 'MEMEMPTY); i=i+4) begin
    $display("%8x: %8x", i, {flash[i], flash[i+1], flash[i+2], flash[i+3]});
  end
`endif
  end endtask
`endif

`ifdef DLINKER
  task createDynInfo; begin
    $readmemh("global_offset", globalAddr);
    m[`GPADDR] = globalAddr[0];
    m[`GPADDR+1] = globalAddr[1];
    m[`GPADDR+2] = globalAddr[2];
    m[`GPADDR+3] = globalAddr[3];
    gp[31:24] = globalAddr[0];
    gp[23:16] = globalAddr[1];
    gp[15:8] = globalAddr[2];
    gp[7:0] = globalAddr[3];
`ifdef DEBUG_DLINKER
    $display("global address %8x", {m[`GPADDR], m[`GPADDR+1],
      m[`GPADDR+2], m[`GPADDR+3]});
    $display("gp = %8x", gp);
`endif
  end
`endif
`ifdef DLINKER
  for (i=0; i < 192; i=i+1) begin
    dsym[i] = 'MEMEMPTY;
  end
  for (i=0; i < 96; i=i+1) begin
    dstr[i] = 'MEMEMPTY;
  end
  for (i=0; i < 384; i=i+1) begin
    so_func_offset[i] = 'MEMEMPTY;
  end
  $readmemh("dysym", dsym);
  $readmemh("dynstr", dstr);
  $readmemh("so_func_offset", so_func_offset);
  setDynLinkerInfo();
  end endtask
`endif
```

### **lbdex/cpu0\_verilog/flashio.v**

```

`define FLASHADDR 'hA0000

`ifdef DLINKER
    end else if (abus >= `FLASHADDR && abus <= `FLASHADDR+`MEMSIZE-4) begin
        fabus = abus-`FLASHADDR;
        if (en == 1 && rw == 0) begin // r_w==0:write
            data = dbus_in;
            case (m_size)
                'BYTE: {flash[fabus]} = dbus_in[7:0];
                'INT16: {flash[fabus], flash[fabus+1]} = dbus_in[15:0];
                'INT24: {flash[fabus], flash[fabus+1], flash[fabus+2]} = dbus_in[24:0];
                'INT32: {flash[fabus], flash[fabus+1], flash[fabus+2], flash[fabus+3]} = dbus_in;
            endcase
        end else if (en == 1 && rw == 1) begin// r_w==1:read
            case (m_size)
                'BYTE: data = {8'h00, 8'h00, 8'h00, flash[fabus]};
                'INT16: data = {8'h00, 8'h00, flash[fabus], flash[fabus+1]};
                'INT24: data = {8'h00, flash[fabus], flash[fabus+1], flash[fabus+2]};
                'INT32: data = {flash[fabus], flash[fabus+1], flash[fabus+2], flash[fabus+3]};
            endcase
        end else
            data = 32'hXXXXXXXX;
    `endif

```

### **lbdex/cpu0\_verilog/cpu0ld.v**

```

`define DLINKER // Dynamic Linker Support
//`define DEBUG_DLINKER // Dynamic Linker Debug
// TRACE: Display the memory contents of the loaded program and data
//`define TRACE

`include "cpu0.v"

```

### **lbdex/cpu0\_verilog/cpu0lld.v**

```

`define CPU0II
`define DLINKER // Dynamic Linker Support
//`define DEBUG_DLINKER // Dynamic Linker Debug
// TRACE: Display the memory contents of the loaded program and data
//`define TRACE

`include "cpu0.v"

```

The following code ch\_dynamiclinker.cpp and foobar.cpp is the example for dynamic linker demonstration. File dynamic\_linker.cpp is what our implementaion to execute the dynamic linker function on Cpu0 Verilog machine.

### lbdex/InputFiles/dynamic\_linker.h

```
#ifndef _DYNAMIC_LINKER_H_
#define _DYNAMIC_LINKER_H_

#define DYNLINKER_INFO_ADDR 0x70000
#define DYNENT_SIZE          4
#define DYNPROGSTART         0x40000
#define FLASHADDR            0xA0000
#define GPADDR               0x7FFF0

#define STOP \
asm("lui $t9, 0xffff"); \
asm("addiu $t9, $zero, 0xffff"); \
asm("ret $t9");

#define ENABLE_TRACE \
asm("mfsw $at"); \
asm("ori $at, $at, 0x0020"); \
asm("mtsw $at");

#define DISABLE_TRACE \
asm("mfsw $at"); \
asm("andi $at, $at, 0xffffdf"); \
asm("mtsw $at");

struct ProgAddr {
    int memAddr;
    int size;
};

extern void dynamic_linker_init();
extern void dynamic_linker();

#endif
```

### lbdex/InputFiles/dynamic\_linker.cpp

```
#include "dynamic_linker.h"

// #define DEBUG_DLINKER
#define PLTOADDR 0x10
#define REGADDR 0x7ff00

#define SAVE_REGISTERS \
asm("lui $at, 7"); \
asm("ori $at, $at, 0xff00"); \
asm("st $2, 0($at)"); \
asm("st $3, 4($at)"); \
asm("st $4, 8($at)"); \
asm("st $5, 12($at)"); \
asm("st $6, 16($at)"); \
asm("st $7, 20($at)"); \
asm("st $8, 24($at)"); \
asm("st $9, 28($at)"); \
asm("st $10, 32($at)");
```



```

dynsym = * (int *) ((DYNLINKER_INFO_ADDR+4) + (dynsym_idx*DYNENT_SIZE));
dynstr = (char *) (DYNLINKER_INFO_ADDR+4+numDynEntry*4+numDynEntry*52+dynsym);
libOffset = * ((int *) (DYNLINKER_INFO_ADDR+4+numDynEntry*4+(dynsym_idx-1)*52));
for (i = dynsym_idx; i < numDynEntry; i++) {
    nextFunLibOffset = * ((int *) (DYNLINKER_INFO_ADDR+4+numDynEntry*4+i*52));
    if (libOffset != nextFunLibOffset)
        break;
}
#endif DEBUG_DLINKER
printf("address of dstr = %x, dynsym = %d, dstr = %s\n",
       (int) dynstr, dynsym, dynstr);
printf("libOffset = %d, nextFunLibOffset = %d, progCounter = %d\n",
       libOffset, nextFunLibOffset, progCounter);
#endif
if (progCounter == 0)
    nextFreeAddr = DYNPROGSTART;
else
    nextFreeAddr = prog[progCounter-1].memAddr+prog[progCounter-1].size;
prog[progCounter].memAddr = nextFreeAddr;
prog[progCounter].size = (nextFunLibOffset - libOffset);

#endif DEBUG_DLINKER
printf("prog[progCounter].memAddr = %d, prog[progCounter].size = %d\n",
       prog[progCounter].memAddr, (unsigned int) (prog[progCounter].size));
#endif
// Load program from (FLASHADDR+libOffset..FLASHADDR+nextFunLibOffset-1) to
// (nextFreeAddr..nextFreeAddr+prog[progCounter].size-1)
src = (int *) (FLASHADDR+libOffset);
end = (int *) (src+prog[progCounter].size/4);
#endif DEBUG_DLINKER
printf("end = %x, src = %x, nextFreeAddr = %x\n",
       (unsigned int) end, (unsigned int) src, (unsigned int) nextFreeAddr);
printf("*src = %x\n", (unsigned int) (*src));
#endif
printf("loading %s...\n", dynstr);
for (dest = (int *) (prog[progCounter].memAddr); src < end; src++, dest++) {
    *dest = *src;
#endif DEBUG_DLINKER
printf("*dest = %08x\n", (unsigned int) (*dest));
#endif
}
progCounter++;

#endif DEBUG_DLINKER
printf("progCounter-1 = %x, prog[progCounter-1].memAddr = %x, \
*prog[progCounter-1].memAddr = %x\n",
       (unsigned int) (progCounter-1), (unsigned int) (prog[progCounter-1].memAddr),
       *(unsigned int *) (prog[progCounter-1].memAddr));
#endif
// Change .got.plt for "ld      $t9, idx($gp)"
*((int *) (gp+0x10+dynsym_idx*0x04)) = prog[progCounter-1].memAddr;
*(int *) (0x7FFE0) = prog[progCounter-1].memAddr;
#endif DEBUG_DLINKER
printf("*(int *) (gp+0x10+dynsym_idx*0x10)) = %x, *(int *) (0x7FFE0) = %x\n",
       *((int *) (gp+0x10+dynsym_idx*0x10)), (unsigned int) (*(int *) (0x7FFE0)));
printf("*(int *) (gp+0x04)) = %x, *((int *) (gp+0x08)) = %x, *((int *) (gp+0x0c)) = %x\n",
       *((int *) (gp+0x04)), *((int *) (gp+0x08)), *((int *) (gp+0x0c)));
#endif

```

```
printf("run %s...\n", dynstr);
RESTORE_REGISTERS;

// restore $lr. The next instruction of foo() of main.cpp for the main.cpp
// call foo() first time example.
// The $lr, $fp and $sp saved in cpu0P1t0AtomContent of Cpu0LinkingContext.cpp.
asm("ld $lr, 4($gp)"); // restore $lr
#ifndef DEBUG_DLINKER
ENABLE_TRACE;
#endif
asm("ld $fp, 8($gp)"); // restore $fp
asm("ld $sp, 12($gp)"); // restore $sp
#ifndef DEBUG_DLINKER
DISABLE_TRACE;
#endif
// jmp to the dynamic linked function. It's foo() for the
// caller, ch_dynamic_linker.cpp, call foo()
// first time example.
asm("lui $t9, 0x7");
asm("ori $t9, $t9, 0xFFE0");
asm("ld $t9, 0($t9)");
asm("ret $t9");

return;
}
```

#### lbdex/InputFiles/ch\_dynamiclinker.cpp

```
#include "dynamic_linker.h"
#include "print.h"

extern "C" int printf(const char *format, ...);

// For memory IO
extern "C" int putchar(const char c)
{
    char *p = (char*)OUT_MEM;
    *p = c;

    return 0;
}

extern int la(int x1, int x2);
extern int foo(int x1, int x2);
extern int bar();

int main()
{
//    ENABLE_TRACE;
    int a = 0;

#if 0
    a = la(1, 2);
    printf("la(1, 2) = %d\n", a);
#endif
#if 1
```

```
a = foo(1, 2);
printf("foo(1, 2) = %d\n", a);
#endif
#if 1
a = bar();
printf("bar() = %d\n", a);
#endif

return 0;
}
```

### Ibdex/InputFiles/foobar.cpp

```
#include "dynamic_linker.h"

int la(int x1, int x2)
{
    int sum = x1 + x1 + x2;

    return sum;
}

int foo(int x1, int x2)
{
    int sum = x1 + x2;

    return sum;
}

int bar()
{
    int a;
//  ENABLE_TRACE;
    a = foo(2, 2);
    a += la(2, 3); // 4+7=11

    return a;
}
```

### 13.5.1 Run

```
[Gamma@localhost cpu0_verilog]$ pwd
/home/Gamma/test/lbd/docs/BackendTutorial/source_ExampleCode/cpu0_verilog
[Gamma@localhost cpu0_verilog]$ bash clean.sh
[Gamma@localhost InputFiles]$ cd ../InputFiles/
[Gamma@localhost InputFiles]$ bash build-dlinker.sh
[Gamma@localhost InputFiles]$ cd ../cpu0_verilog/
[Gamma@localhost cpu0_verilog]$ pwd
/home/Gamma/test/lbd/docs/BackendTutorial/source_ExampleCode/cpu0_verilog
[Gamma@localhost cpu0_verilog]$ iverilog -o cpu0IID cpu0IID.v
[Gamma@localhost cpu0_verilog]$ ls
clean.sh  cpu0Id  cpu0Id.v  cpu0IID.v  cpu0IIs.v  cpu0Is.v  cpu0.v  dynlinker.v
flashio.v
[Gamma@localhost cpu0_verilog]$ ./cpu0Id
WARNING: ./cpu0.v:371: $readmemh(cpu0.hex): Not enough words in the file for
```

```
the requested range [0:524287].  
WARNING: ./dynlinker.v:185: $readmemh(libso.hex): Not enough words in the  
file for the requested range [0:524287].  
WARNING: ./dynlinker.v:223: $readmemh(dynsym): Not enough words in the file  
for the requested range [0:191].  
WARNING: ./dynlinker.v:224: $readmemh(dynstr): Not enough words in the file  
for the requested range [0:95].  
WARNING: ./dynlinker.v:225: $readmemh(so_func_offset): Not enough words in  
the file for the requested range [0:383].  
numDynEntry = 00000005  
taskInterrupt(001)  
loading _Z3fooii...  
run _Z3fooii...  
foo(1, 2) = 3  
loading _Z3barv...  
run _Z3barv...  
loading _Z2laii...  
run _Z2laii...  
bar() = 11  
RET to PC < 0, finished!
```

The “#ifdef DEBUG\_DLINKER” part of code in dynamic\_linker.cpp is for debugging purpose (since we coding it and take time to debug). After skip these debug code, the dynamic\_linker.cpp is short and not difficult to read.

The run result is under expectation. The main() call foo() function first. Function foo() is loaded by dynamic linker (dynamic\_linker.cpp) from memory address FLASHADDR (defined in dynamic\_linker.h) to memory. The flashio.v implement the simulation read from flash address. After loaded foo() body from flash, dynamic\_linker.cpp jump to this loaded address by “ret \$t9” instruction.

Same as static linker, you can generate slt instruction instead of cmp by change from cpu=cpu0I to cpu0=cpu0II in build-dlinker.sh and run it again to get the same result.

### 13.5.2 Cpu0 IId dynamic linker structure

## 13.6 Summary

Thanks the llvm open source project. To write a linker and ELF to Hex tools for the new CPU architecture is easy and reliable. Combine with the llvm compiler backend of support new architecture Cpu0 and Verilog language program in the previous Chapters, we design a software toolchain to compile C/C++ code, link and run it on Verilog Cpu0 simulated machine of PC without any real hardware to investment. If you like to pay money to buy the FPGA development hardware, we believe the code can run on FPGA CPU without problem even though we didn't do it. System program toolchain can be designed just like we show you at this point. School knowledge of system program, compiler, linker, loader, computer architecture and CPU design can be translated into a real work and see how it be run. Now, these school books knowledge is not limited on paper. We program it, design it and run it on real world.



# APPENDIX A: GETTING STARTED: INSTALLING LLVM AND THE CPU0 EXAMPLE CODE

This book is in the process of merging into llvm trunk but not finished yet. The merged llvm trunk version on my git hub is LLVM 3.3 released. The Cpu0 example code based on llvm 3.3.

In this chapter, we will run through how to set up LLVM using if you are using Mac OS X or Linux. When discussing Mac OS X, we are using Apple's Xcode IDE (version 4.5.1) running on Mac OS X Mountain Lion (version 10.8) to modify and build LLVM from source, and we will be debugging using lldb. We cannot debug our LLVM builds within Xcode at the moment, but if you have experience with this, please contact us and help us build documentation that covers this. For Linux machines, we are building and debugging (using gdb) our LLVM installations on a Fedora 17 system. We will not be using an IDE for Linux, but once again, if you have experience building/ debugging LLVM using Eclipse or other major IDEs, please contact the authors. For information on using `cmake` to build LLVM, please refer to the "Building LLVM with CMake" <sup>1</sup> documentation for further information. We are using `cmake` version 2.8.9.

We will install two llvm directories in this chapter. One is the directory `llvm/release/` which contains the clang, clang++ compiler we will use to translate the C/C++ input file into llvm IR. The other is the directory `llvm/test/` which contains our cpu0 backend program and without clang and clang++.

LLVM and this book use sphinx to generate html and pdf document. Sphinx install is included in this Chapter.

---

## Todo

Find information on debugging LLVM within Xcode for Macs.

---

---

## Todo

Find information on building/debugging LLVM within Eclipse for Linux.

---

## 14.1 Setting Up Your Mac

### 14.1.1 Installing LLVM, Xcode and `cmake`

---

## Todo

<sup>1</sup> <http://llvm.org/docs/CMake.html?highlight=cmake>

Fix centering for figure captions.

---

Please download LLVM latest release version 3.3 (llvm, clang, compiler-rt) from the “LLVM Download Page” <sup>2</sup>. Then extract them using `tar -zxvf {llvm-3.3.src.tar, clang-3.3.src.tar, compiler-rt-3.3.src.tar}`, and change the llvm source code root directory into src. After that, move the clang source code to src/tools/clang, and move the compiler-rt source to src/projects/compiler-rt as shown as follows,

```
118-165-78-111:Downloads Jonathan$ tar -zxvf clang-3.3.src.tar.gz
118-165-78-111:Downloads Jonathan$ tar -zxvf compiler-rt-3.3.src.tar.gz
118-165-78-111:Downloads Jonathan$ tar -zxvf llvm-3.3.src.tar.gz
118-165-78-111:Downloads Jonathan$ mv llvm-3.3.src src
118-165-78-111:Downloads Jonathan$ mv clang-3.3.src src/tools/clang
118-165-78-111:Downloads Jonathan$ mv compiler-rt-3.3.src src/projects/compiler-rt
118-165-78-111:Downloads Jonathan$ pwd
/Users/Jonathan/Downloads
118-165-78-111:Downloads Jonathan$ ls
clang-3.3.src.tar.gz      llvm-3.3.src.tar.gz
compiler-rt-3.3.src.tar.gz  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
llc            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
118-165-78-111:Downloads Jonathan$ ls src/projects/
CMakeLists.txt  LLVMBuild.txt Makefile  compiler-rt sample
```

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: <sup>3</sup>. Before installing cmake, make sure 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, right -click (or control-click) the cmake .pkg package file and click “Open.” Mac OS X will ask you if you are sure you want to install this package, and you can click “Open” to start the installer.

### 14.1.2 Create LLVM.xcodeproj by cmake Graphic UI

We install llvm source code with clang on directory `/Users/Jonathan/llvm/release/` in last section. Now, will generate the LLVM.xcodeproj in this chapter.

Currently, we cannot do debug by lldb with cmake graphic UI operations depicted in this section, but we can do debug by lldb with “section Create LLVM.xcodeproj of supporting cpu0 by terminal cmake command” <sup>4</sup>. Even with that, let’s build LLVM project with cmake graphic UI since this LLVM directory contains the release version for clang and clang++ execution file. First, create LLVM.xcodeproj as [Figure 14.2](#), then click **configure** button to enter [Figure 14.3](#), and then click **Done** button to get [Figure 14.4](#).

Click **OK** from [Figure 14.4](#) and select Cmake 2.8-9.app for CMAKE\_INSTALL\_NAME\_TOOL by click the right side button “...” of that row to get [Figure 14.5](#).

---

<sup>2</sup> <http://llvm.org/releases/download.html#3.3>

<sup>3</sup> <http://www.cmake.org/cmake/resources/software.html>

<sup>4</sup> <http://jonathan2251.github.com/lbd/install.html#create-llvm-xcodeproj-of-supporting-cpu0-by-terminal-cmake-command>

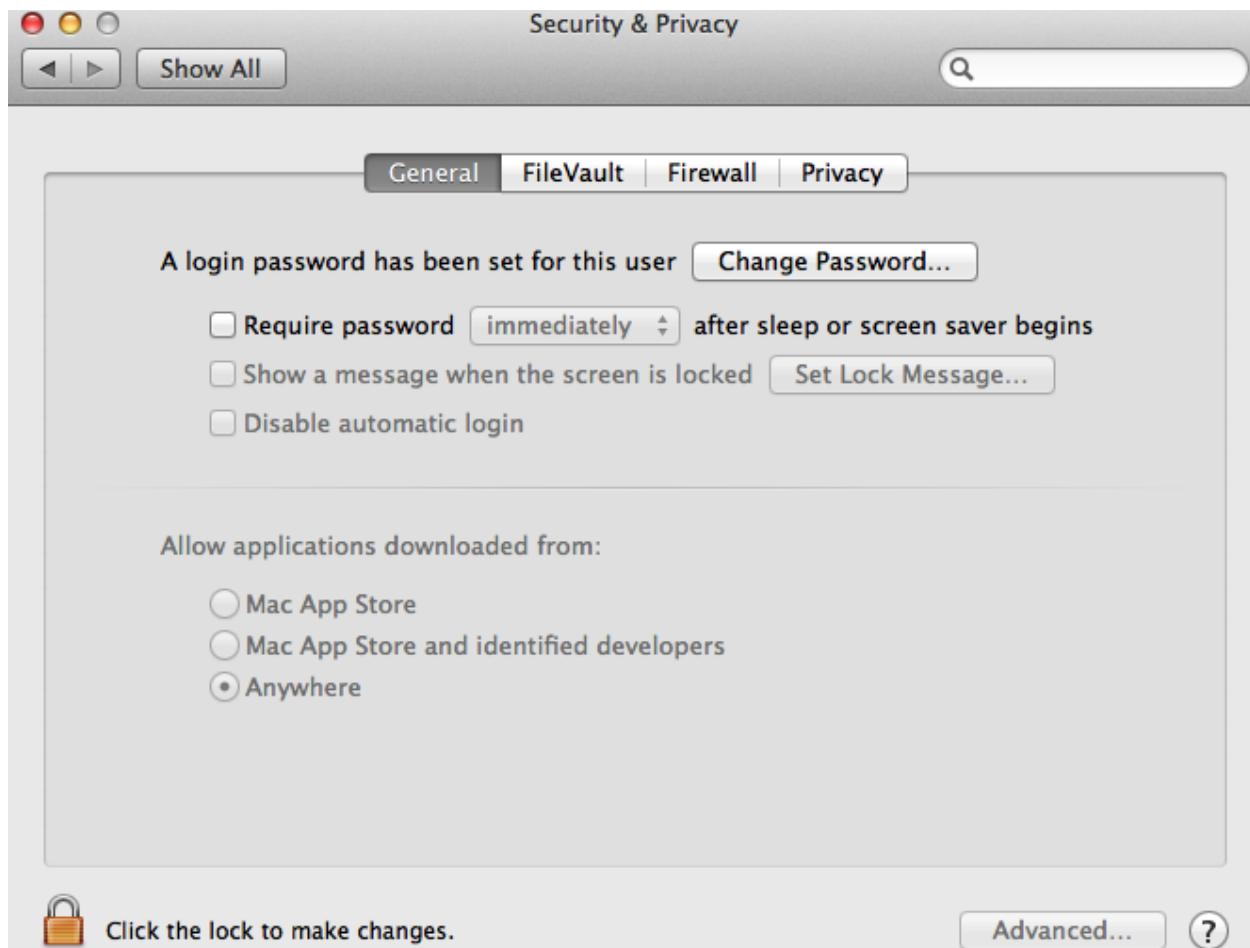


Figure 14.1: Adjusting Mac OS X security settings to allow cmake installation.

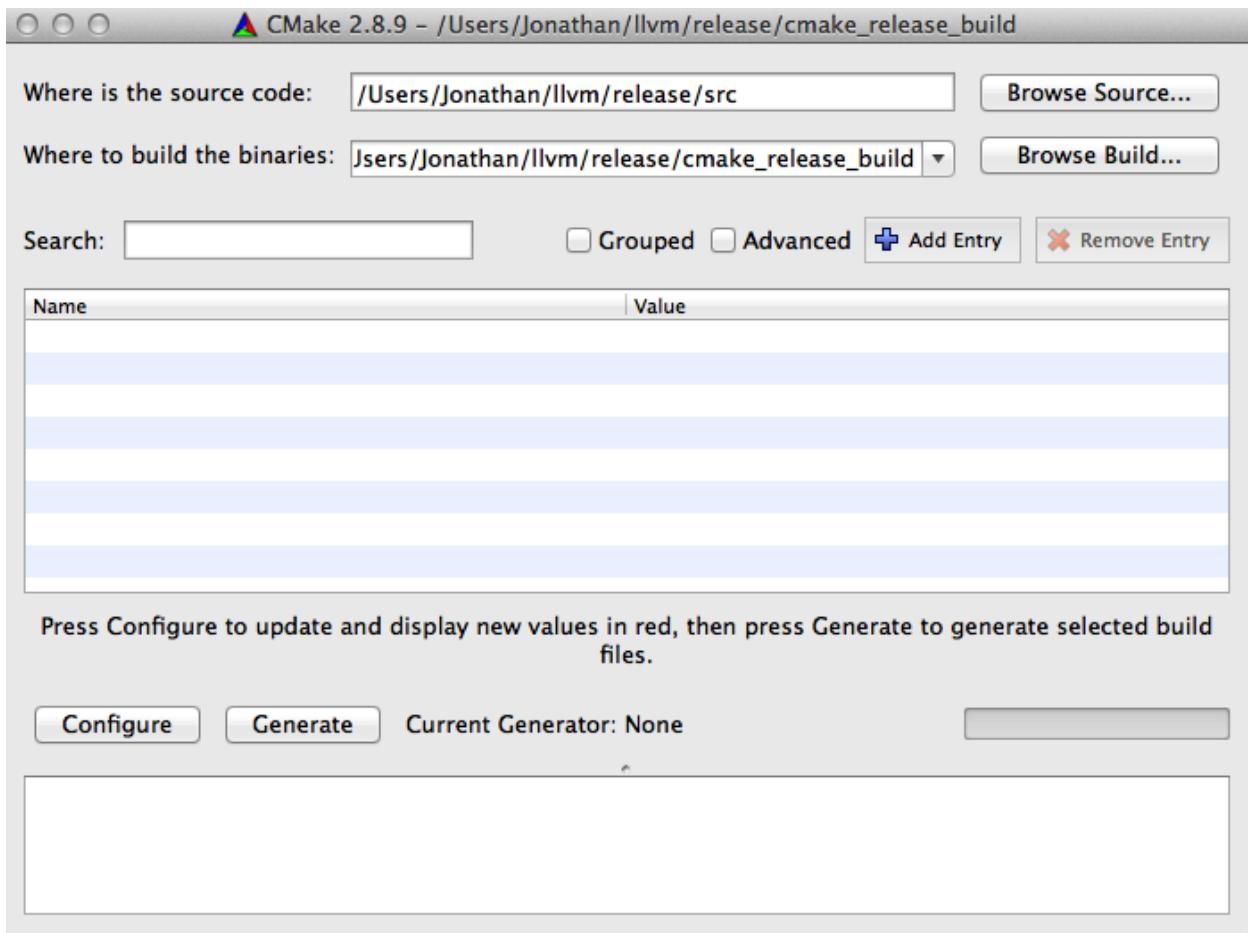


Figure 14.2: Start to create LLVM.xcodeproj by cmake

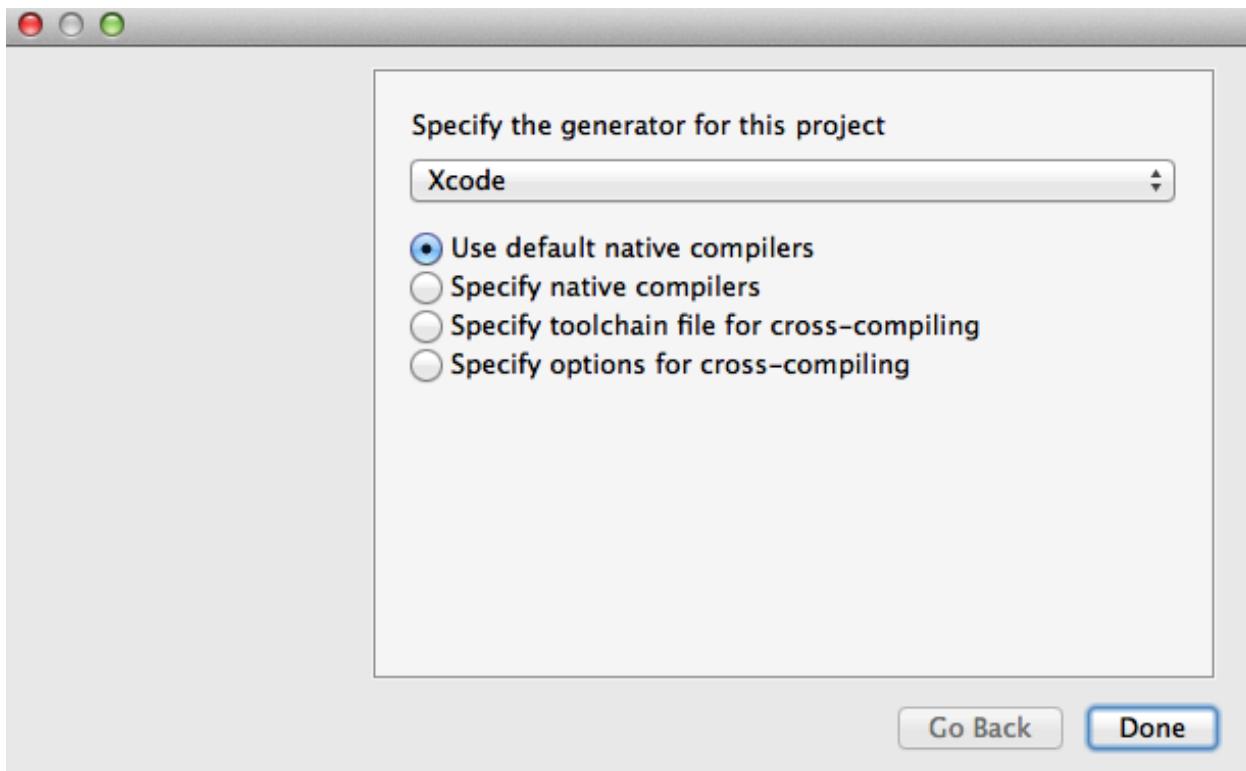


Figure 14.3: Create LLVM.xcodeproj by cmake – Set option to generate Xcode project

Click Configure button to get [Figure 14.6](#).

Check CLANG\_BUILD\_EXAMPLES, LLVM\_BUILD\_EXAMPLES, and uncheck LLVM\_ENABLE\_PIC as [Figure 14.7](#).

Click Configure button again. If the output result message has no red color, then click Generate button to get [Figure 14.8](#).

### 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 14.9](#).

After few minutes of build, the clang, llc, llvm-as, ..., can be found in cmake\_release\_build/bin/Debug/ as follows.

```
118-165-78-111:cmake_release_build Jonathan$ cd bin/Debug/
118-165-78-111:Debug Jonathan$ pwd
/Users/Jonathan/llvm/release/cmake_release_build/bin/Debug
118-165-78-111:Debug Jonathan$ ls
BrainF          Kaleidoscope-Ch7  clang-tblgen      llvm-dis      llvm-rtdyld
ExceptionDemo   ModuleMaker      count           llvm-dwarfdump  llvm-size
Fibonacci       ParallelJIT     diagtool        llvm-extract   llvm-stress
FileCheck        arcmt-test      llc            llvm-link     llvm-tblgen
FileUpdate       bugpoint        lli             llvm-mc       macho-dump
HowToUseJIT      c-arcmt-test   llvm-ar        llvm-mcmarkup  not
Kaleidoscope-Ch2 c-index-test   llvm-as        llvm-nm       obj2yaml
Kaleidoscope-Ch3 clang          llvm-bcanalyzer llvm-objdump  opt
Kaleidoscope-Ch4 clang++        llvm-config    llvm-prof    yaml-bench
```

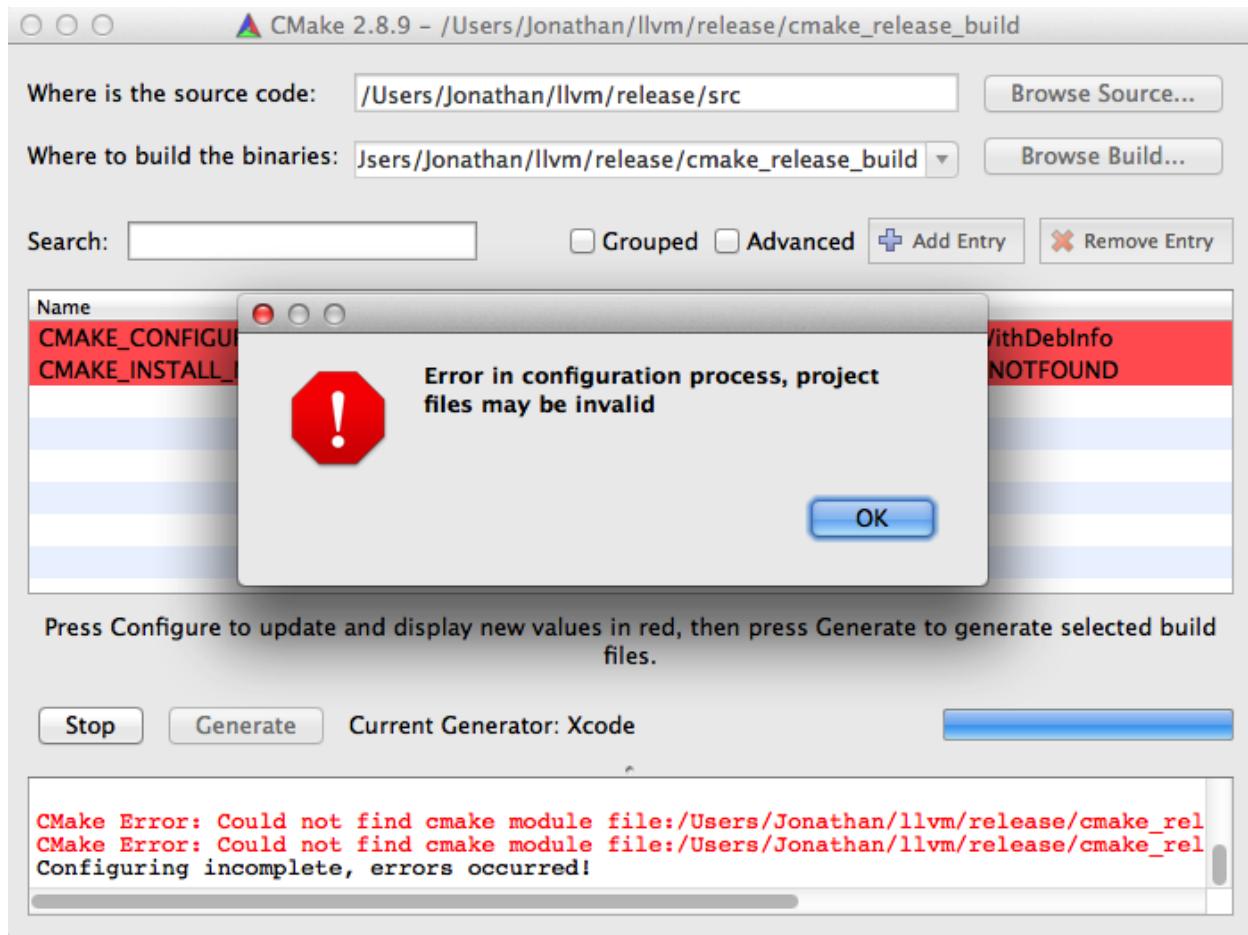


Figure 14.4: Create LLVM.xcodeproj by cmake – Before Adjust CMAKE\_INSTALL\_NAME\_TOOL

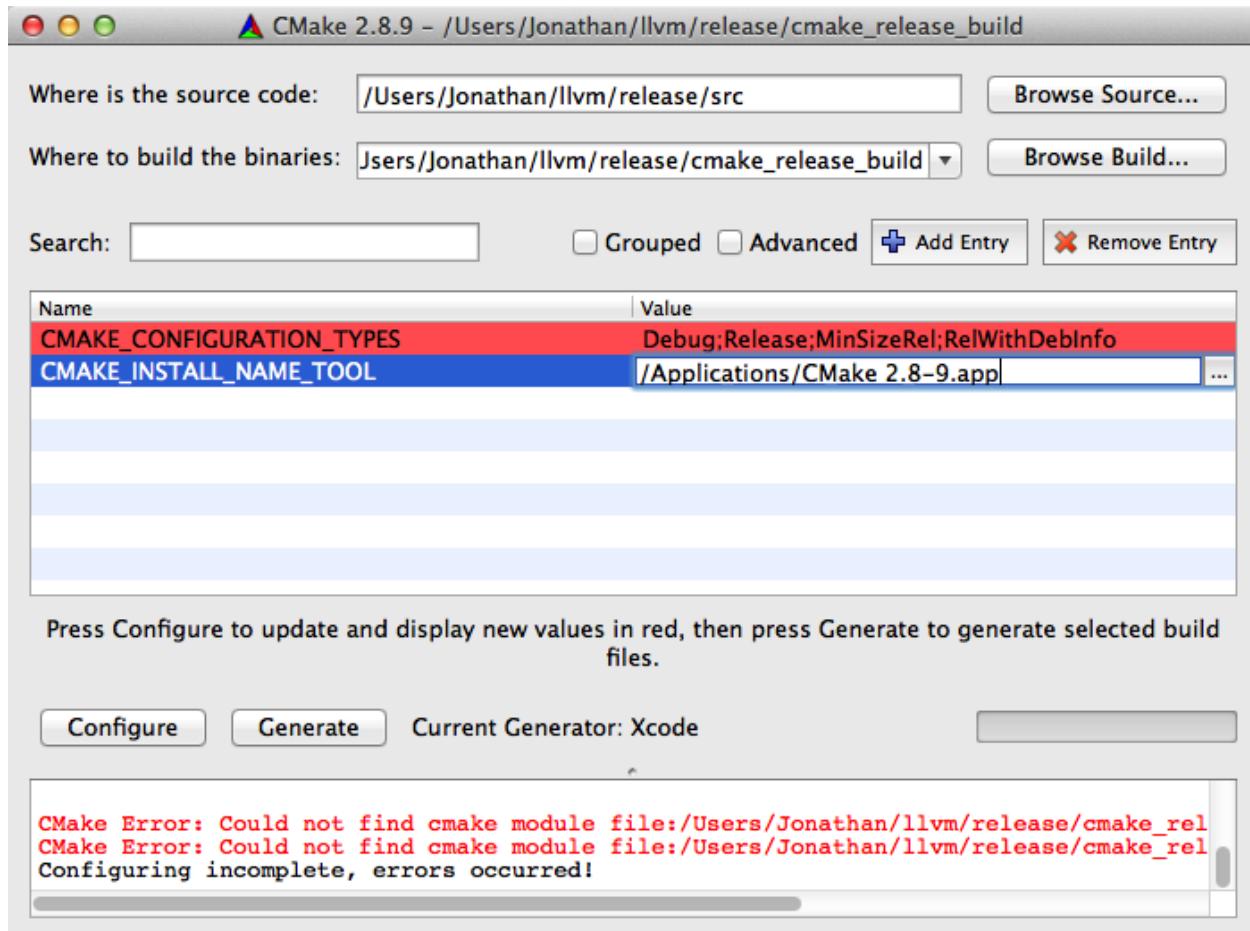


Figure 14.5: Select Cmake 2.8-9.app

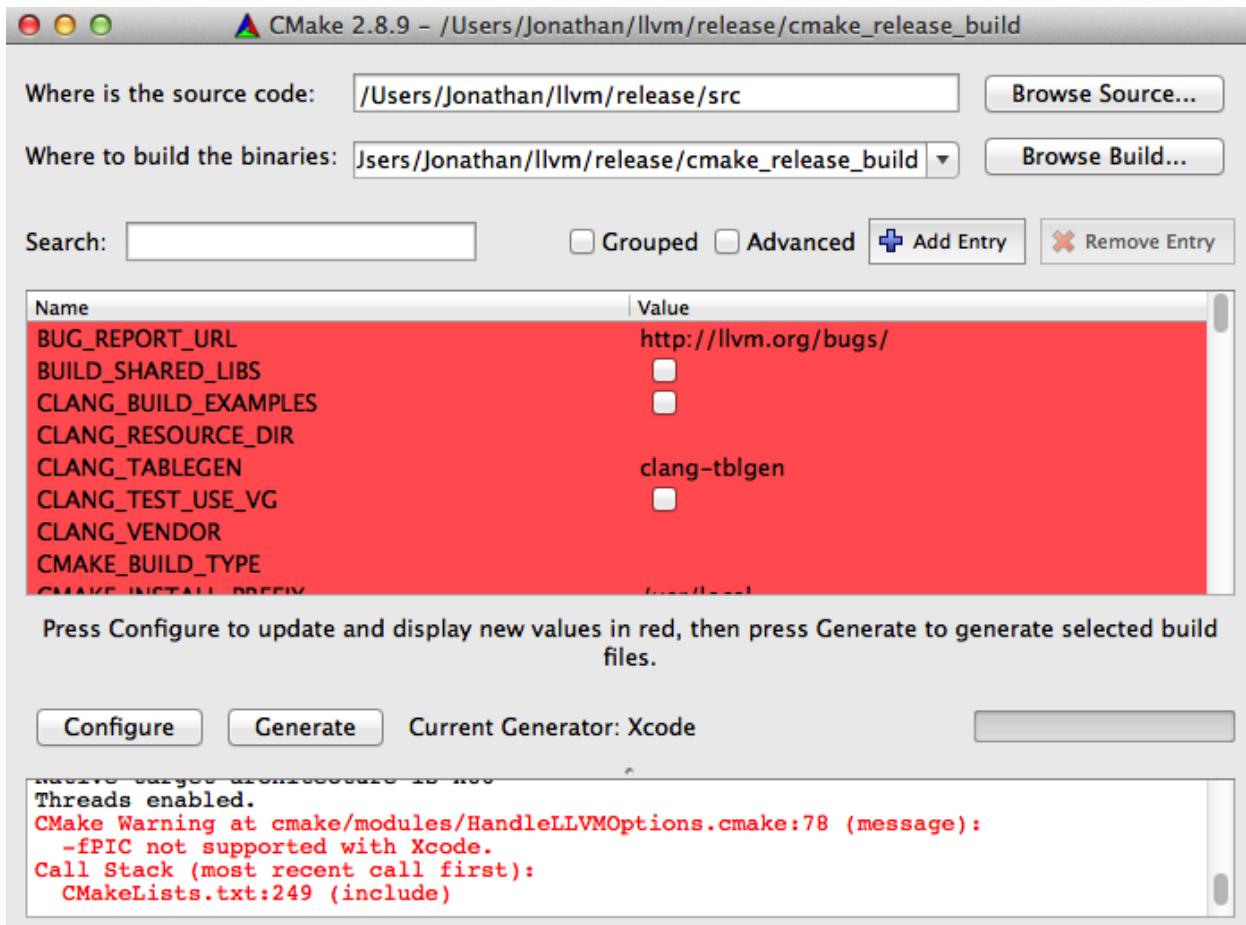


Figure 14.6: Click cmake Configure button first time

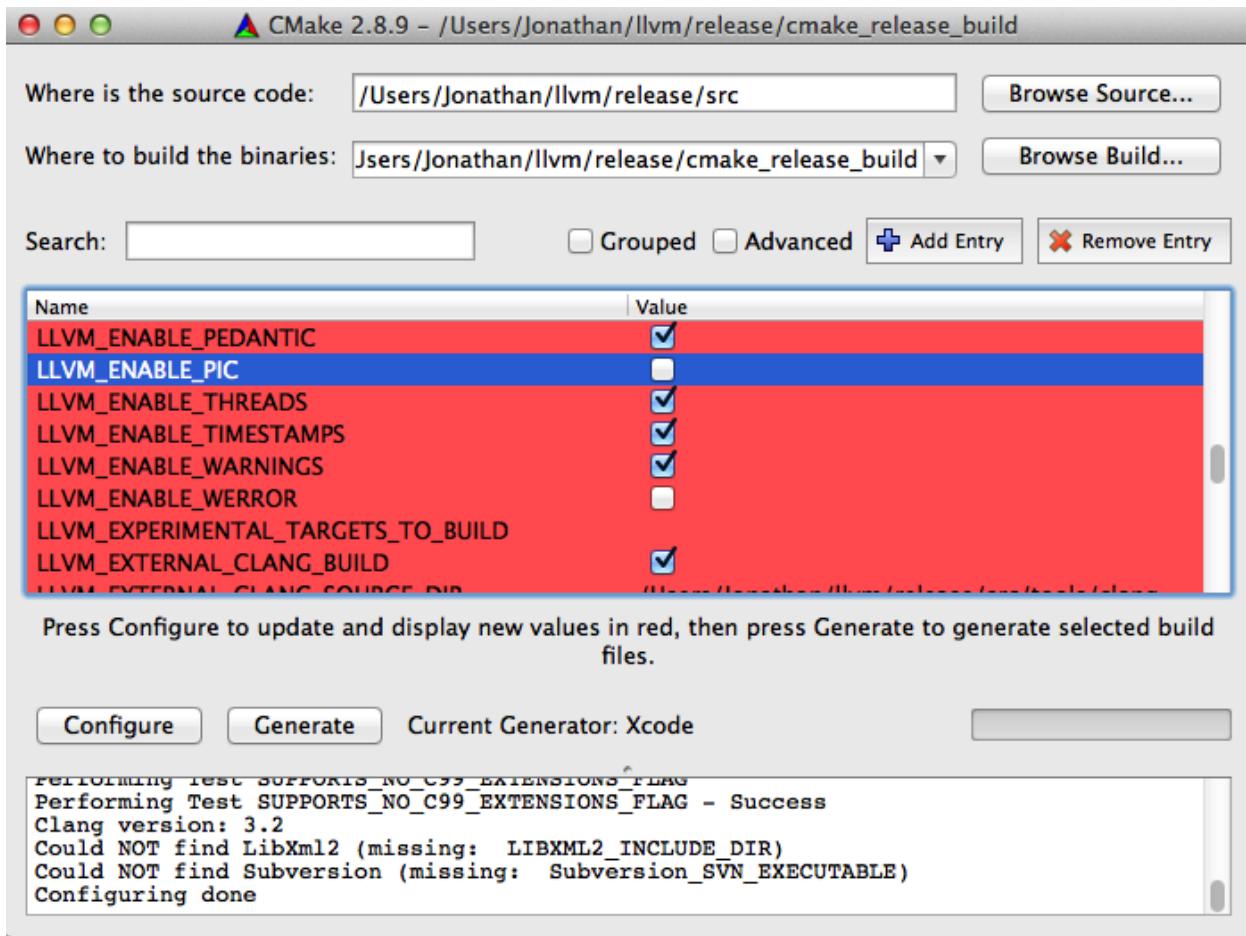


Figure 14.7: Check CLANG\_BUILD\_EXAMPLES, LLVM\_BUILD\_EXAMPLES, and uncheck LLVM\_ENABLE\_PIC in cmake

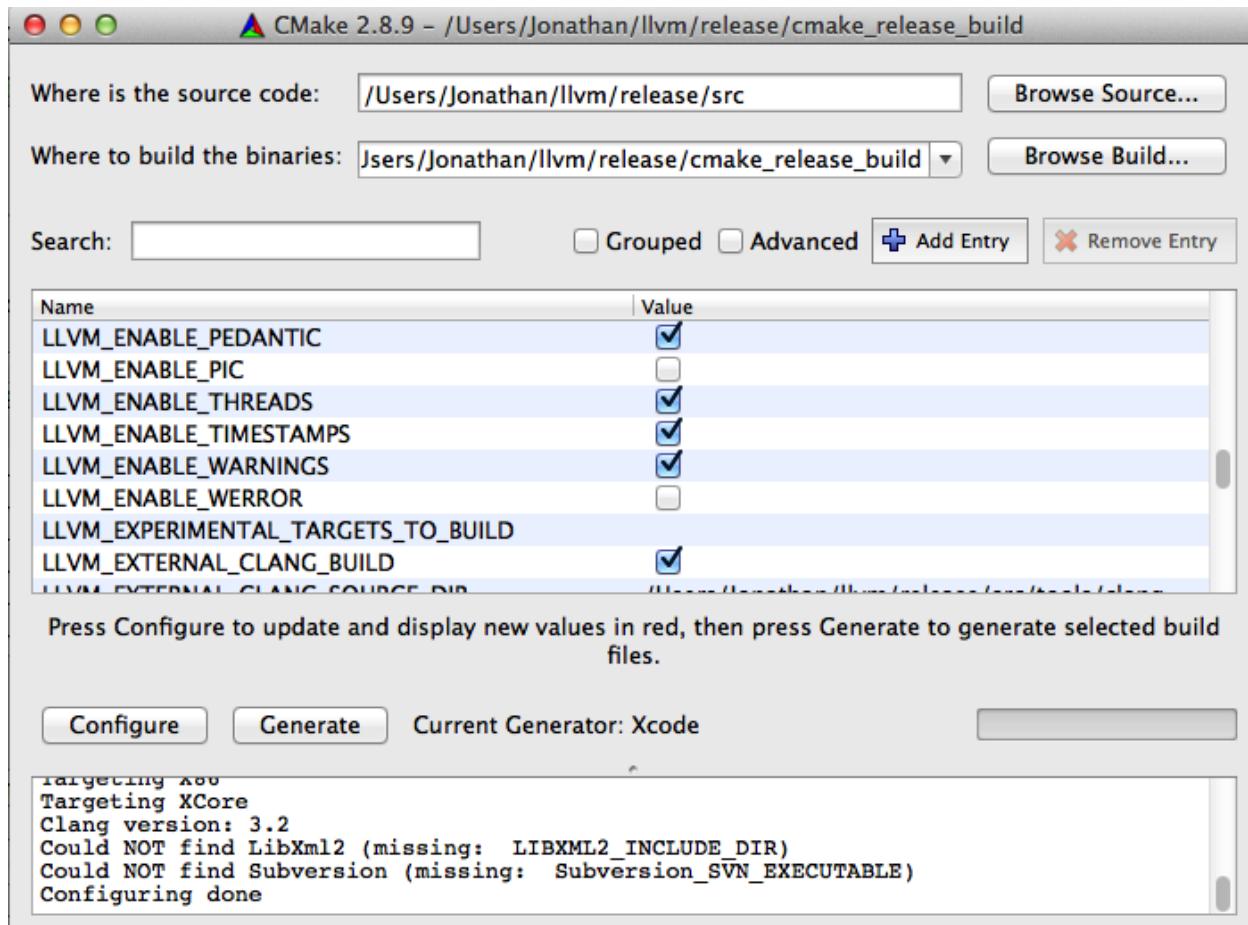


Figure 14.8: Click cmake Generate button second time

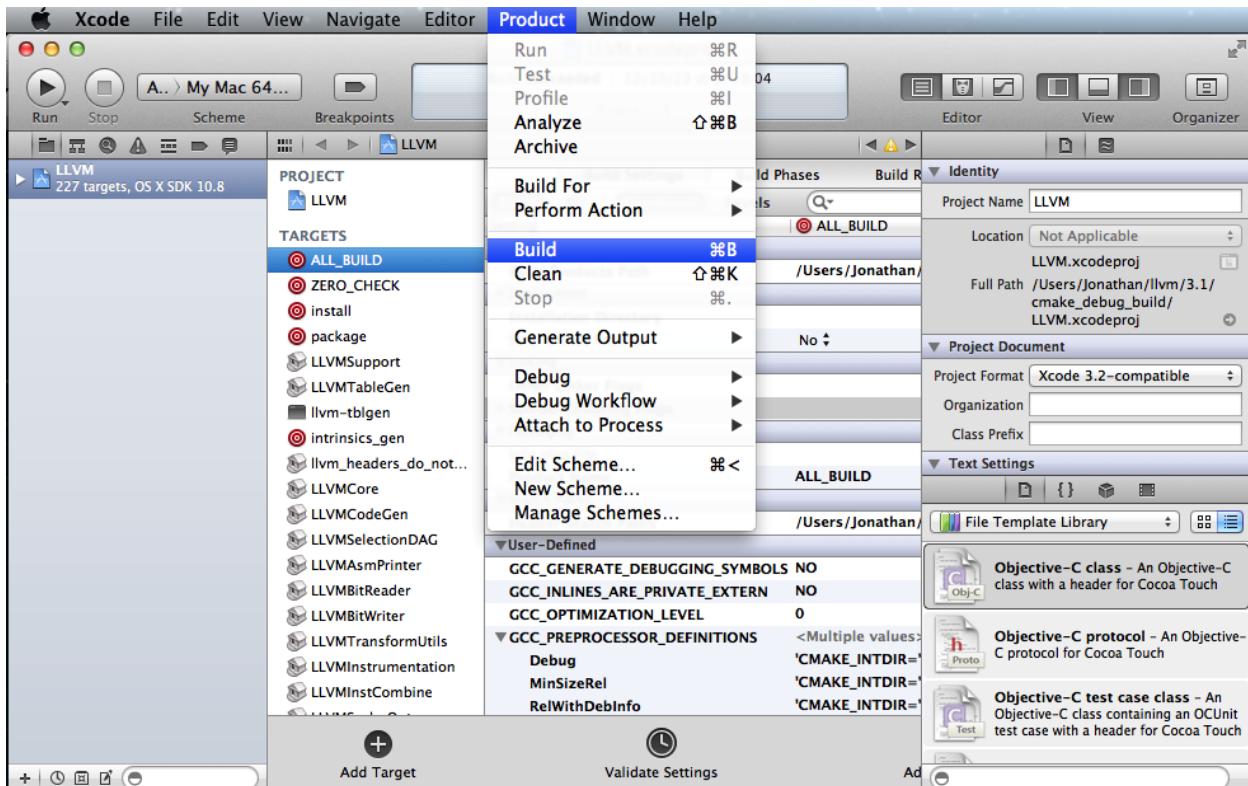


Figure 14.9: Click Build button to build LLVM.xcodeproj by Xcode

```

Kaleidoscope-Ch5  clang-check      llvm-cov      llvm-ranlib      yaml2obj
Kaleidoscope-Ch6  clang-interpreter  llvm-diff      llvm-readobj
118-165-78-111:Debug Jonathan$
```

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=$PATH:/Applications/Xcode.app/Contents/Developer/usr/bin:/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin:/Applications/Graphviz.app/Contents/MacOS:/Users/Jonathan/llvm/release/cmake_release_build/bin/Debug
export WORKON_HOME=$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.

In “section Create LLVM.xcodeproj by cmake Graphic UI”<sup>5</sup>, we create LLVM.xcodeproj by cmake graphic UI. We can create LLVM.xcodeproj by cmake command on terminal also. 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 3.3 released version. The lbd of Cpu0 example code is also based on llvm 3.3. So, please install the llvm 3.3 debug version as the llvm release 3.3 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-3.3.src.tar.gz .
118-165-78-111:test Jonathan$ tar -zxvf llvm-3.3.src.tar.gz
118-165-78-111:test Jonathan$ mv llvm-3.3.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$ mkdir src/lib/Target/Cpu0
118-165-78-111:test Jonathan$ mv lbdex
src/lib/Target/Cpu0/.
118-165-78-111:test Jonathan$ cp -rf src/lib/Target/Cpu0/
lbdex/src_files_modify/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 chapter 2 example code according the following commands,

```
118-165-80-55:test Jonathan$ cd src/lib/Target/Cpu0/lbdex/
118-165-80-55:lbdex Jonathan$ pwd
/Users/Jonathan/llvm/test/src/lib/Target/Cpu0/lbdex
118-165-80-55:lbdex Jonathan$ sh removecpu0.sh
118-165-80-55:lbdex Jonathan$ ls ..
lbdex
118-165-80-55:lbdex Jonathan$ cp -rf Chapter2/* ../.
118-165-80-55:lbdex Jonathan$ cd ..
118-165-80-55:Cpu0 Jonathan$ ls
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 -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_C_COMPILER=clang -DCMAKE_BUILD_TYPE =Debug -G "Xcode" ../src/` as follows.

---

<sup>5</sup> <http://jonathan2251.github.com/lbd/install.html#create-llvm-xcodeproj-by-cmake-graphic-ui>

Remind, currently, the `cmake` terminal command can work with `lldb` debug, but the “section Create LLVM.xcodeproj by `cmake` Graphic UI”<sup>5</sup> cannot.

```
118-165-78-111:Target Jonathan$ cd ../../../../
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 -G "Xcode" ../src/
CMake Error: The source directory "/Users/Jonathan/llvm/src" does not exist.
Specify --help for usage, or press the help button on the CMake GUI.
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 -G "Xcode" ../src/
-- The C compiler identification is Clang 4.1.0
-- The CXX compiler identification is Clang 4.1.0
-- Check for working C compiler using: Xcode
...
-- Targeting ARM
-- Targeting CellSPU
-- Targeting CppBackend
-- Targeting Hexagon
-- Targeting Mips
-- Targeting Cpu0
-- Targeting MBlaze
-- Targeting MSP430
-- Targeting NVPTX
-- Targeting PowerPC
-- Targeting Sparc
-- Targeting X86
-- Targeting XCore
-- 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 example code by Xcode as the last section indicated.

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/bin/Debug/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/bin/Debug/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/bin/Debug/
l1c' (x86_64)
Process 6058 stopped
* thread #1: tid = 0x1c03, 0x000000010077f231 l1c'LLVMInitializeMipsTargetInfo
+ 33 at MipsTargetInfo.cpp:20, stop reason = breakpoint 1.1
  frame #0: 0x000000010077f231 l1c'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 l1c'LLVMInitializeMipsTargetInfo
+ 63 at MipsTargetInfo.cpp:23, stop reason = step over
  frame #0: 0x000000010077f24f l1c'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
(l1vm::RegisterTarget<llvm::Triple::ArchType, true>) $0 = {}
(lldb) quit
118-165-65-128:InputFiles Jonathan$
```

About the lldb debug command, please reference <sup>6</sup> or lldb portal <sup>7</sup>.

### 14.1.5 Setup llvm-lit on iMac

The llvm-lit <sup>8</sup> is the llvm regression test tool. You don't need to set up it if you don't want to do regression test even though this book do the regression test. To set it up correctly in iMac, you need move it from directory bin/llvm-lit to bin/Debug/llvm-lit, and modify llvm-lit as follows,

```
118-165-69-59:bin Jonathan$ pwd
/Users/Jonathan/llvm/test/cmake_debug_build/bin
118-165-69-59:bin Jonathan$ ls
Debug          llvm-lit
118-165-69-59:bin Jonathan$ cp llvm-lit Debug/.
// edit llvm-lit as follows,
  'build_config' : ":" ,
  'build_mode'   : "Debug",
```

---

<sup>6</sup> <http://lldb.llvm.org/lldb-gdb.html>

<sup>7</sup> <http://lldb.llvm.org/>

<sup>8</sup> <http://llvm.org/docs/TestingGuide.html>

## 14.1.6 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
```

### 14.1.7 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 <sup>9</sup>. FileMerge is a part of Xcode, you can type FileMerge in Finder – Applications as Figure 14.10 and drag it into the Dock as Figure 14.11.

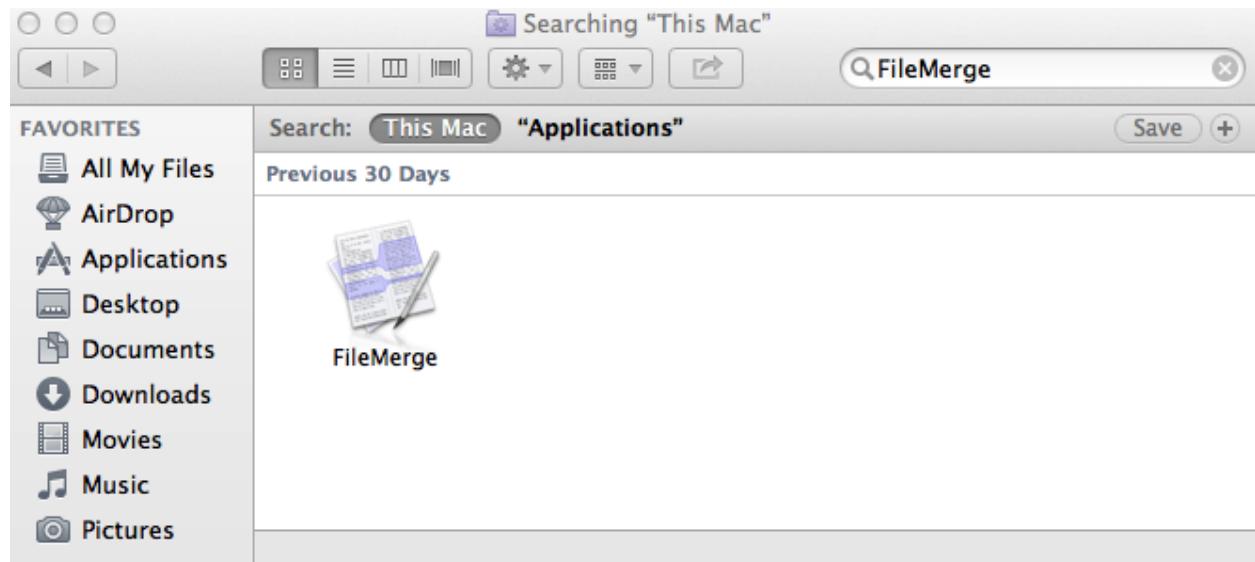


Figure 14.10: Type FileMerge in Finder – Applications



Figure 14.11: Drag FileMege into the Dock

<sup>9</sup> <http://kdiff3.sourceforge.net>

Download tool Graphviz for display llvm IR nodes in debugging,<sup>10</sup>. We choose mountainlion as Figure 14.12 since our iMac is Mountain Lion.

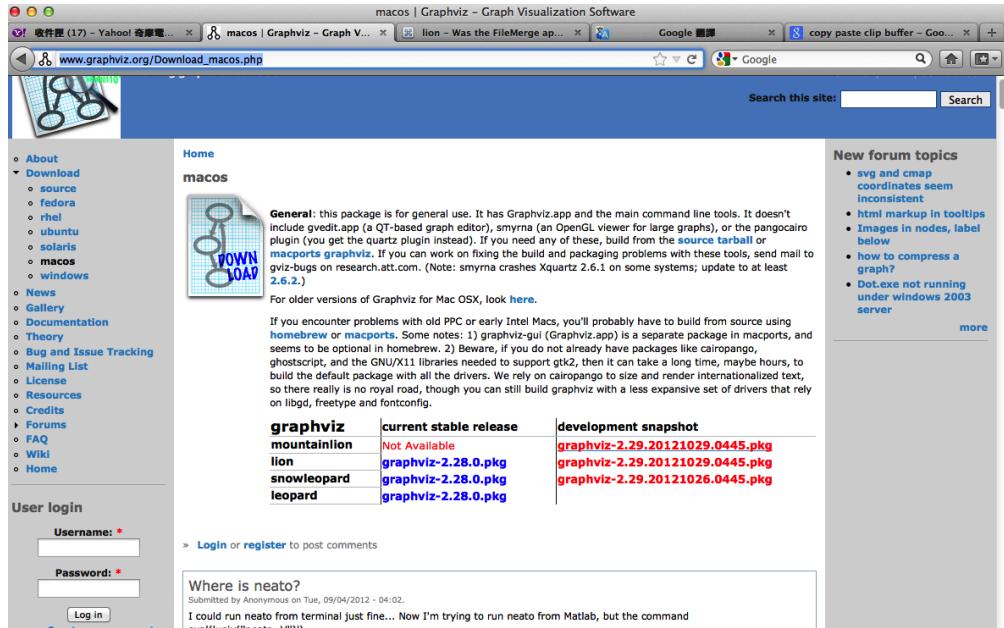


Figure 14.12: Download graphviz for llvm IR node display

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/bin/Debug
```

The Graphviz information for llvm is in the section “SelectionDAG Instruction Selection Process” of<sup>11</sup> and the section “Viewing graphs while debugging code” of<sup>12</sup>. 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 14.13. Then select “Front document’s file” as Figure 14.14.

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
ChangeLog    bin        share
```

<sup>10</sup> [http://www.graphviz.org/Download\\_macos.php](http://www.graphviz.org/Download_macos.php)

<sup>11</sup> <http://llvm.org/docs/CodeGenerator.html>

<sup>12</sup> <http://llvm.org/docs/ProgrammersManual.html>

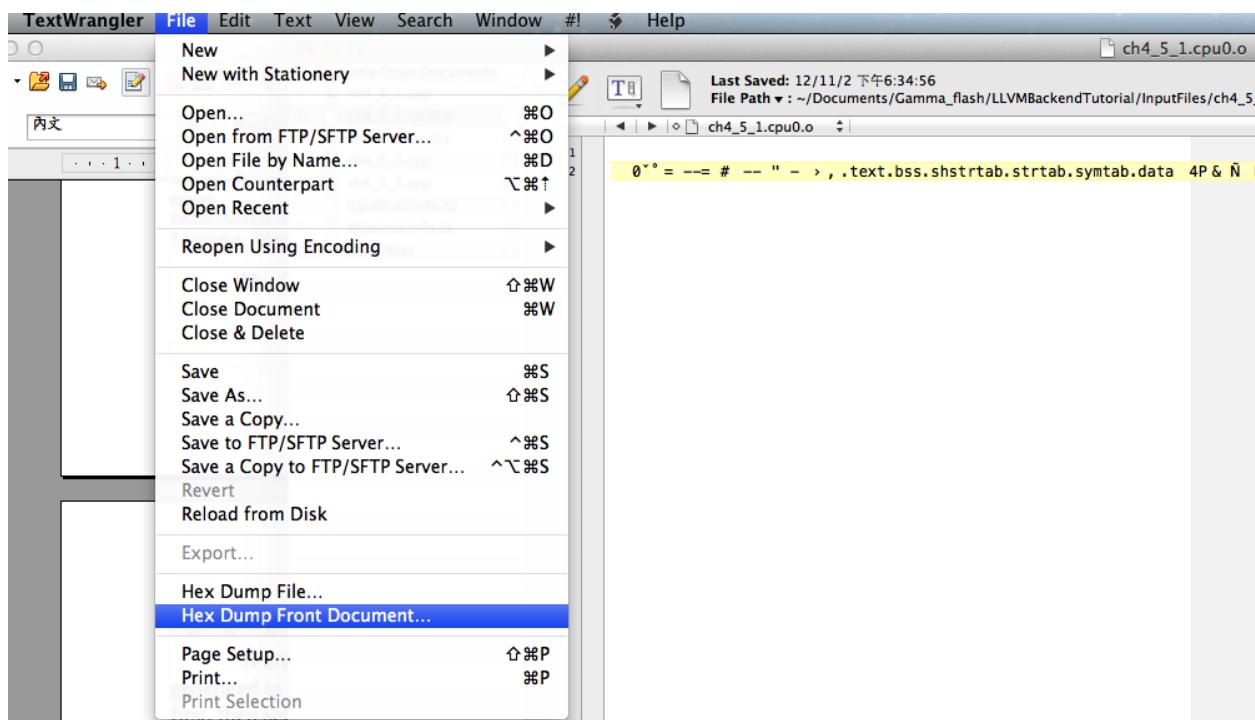


Figure 14.13: Select Hex Dump menu

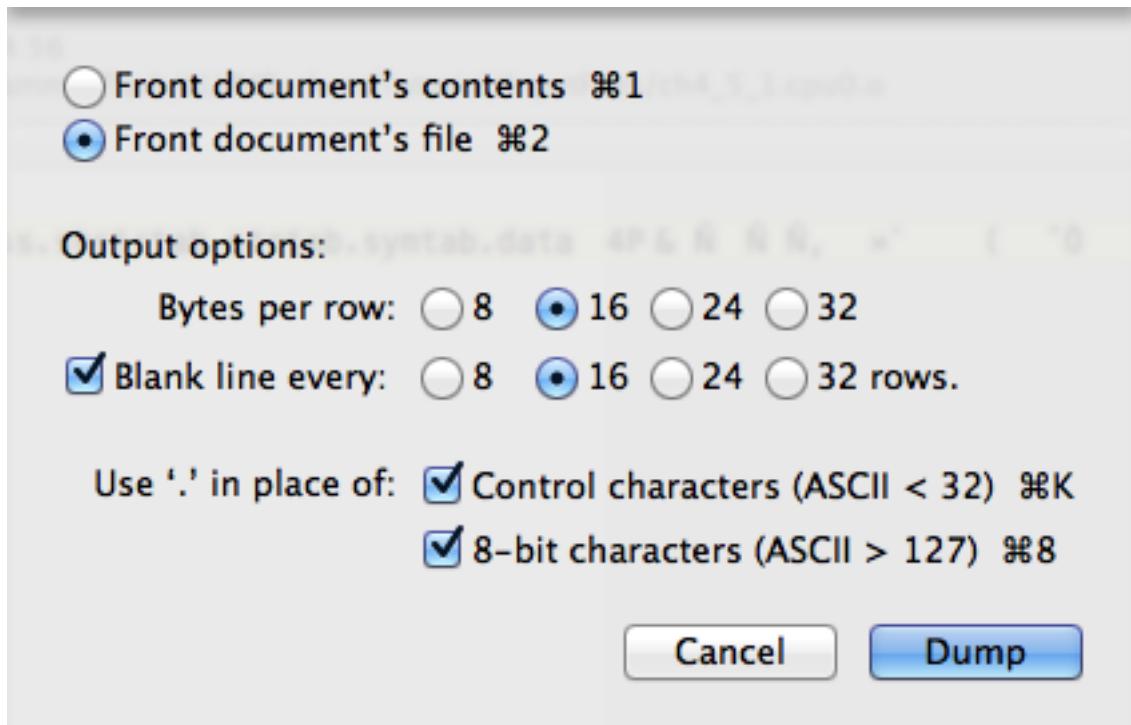


Figure 14.14: Select Front document's file in TextWrangler

```
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 3.3 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.
3. Untar compiler-rt and move it to src/project/compiler-rt.

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]$ cmake -DCMAKE_BUILD_TYPE=Release
-DCLANG_BUILD_EXAMPLES=ON -DLLVM_BUILD_EXAMPLES=ON -G "Unix Makefiles" ..../src/
-- The C compiler identification is GNU 4.7.0
...
-- Constructing LLVMBuild project information
-- Targeting ARM
-- Targeting CellSPU
-- Targeting CppBackend
-- Targeting Hexagon
-- Targeting Mips
-- Targeting MBBlaze
-- Targeting MSP430
-- Targeting PowerPC
-- Targeting PTX
-- Targeting Sparc
-- Targeting X86
-- Targeting XCore
-- Clang version: 3.3
-- Found Subversion: /usr/bin/svn (found version "1.7.6")
-- Configuring done
-- Generating done
-- Build files have been written to: /usr/local/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. Next, edit /home/Gamma/.bash\_profile with adding /usr/local/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=$PATH:/usr/local/sphinx/bin:/usr/local/llvm/release/cmake_release_build/bin:
/opt/mips_linux_toolchain_clang/mips_linux_toolchain/bin:$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:/opt/mips_linux_toolchain_clang/mips_linux_tool
chain/bin:/home/Gamma/.local/bin:/home/Gamma/bin:/usr/local/sphinx/bin:/usr/
local/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 3.3 released version. The Cpu0 example code is also based on llvm 3.3. So, please install the llvm 3.3 debug version as the llvm release 3.3 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 /usr/local/llvm/test/ and get Cpu0 example code as well as the llvm 3.3.
  2. Make dir Cpu0 in src/lib/Target and download example code.
- 3) Update my modified files to support cpu0 by command, cp -rf /usr/local/llvm/test/src/lib/Target/Cpu0/lbdex/src\_files\_modify/modify/src ..
- 4) Check step 3 is effective by command grep -R "Cpu0" . | more'. I add the Cpu0 backend support, so check with grep.
- 5) Enter src/lib/Target/Cpu0/ and copy example code lbdex/2/Cpu0 to the directory by commands cd src/lib/Target/Cpu0/ and cp -rf lbdex/Chapter2/\* ..../..
- 6) Remove clang from /usr/local/llvm/test/src/tools/clang, and mkdir test/cmake\_debug\_build. Without this you will waste extra time for command make in cpu0 example code build.

```
[Gamma@localhost llvm]$ mkdir test
[Gamma@localhost llvm]$ cd test
[Gamma@localhost test]$ pwd
/usr/local/llvm/test
[Gamma@localhost test]$ cp /home/Gamma/Downloads/llvm-3.3.src.tar.gz .
[Gamma@localhost test]$ tar -zvxf llvm-3.3.src.tar.gz
[Gamma@localhost test]$ mv llvm-3.3.src src
[Gamma@localhost test]$ cp /Users/Jonathan/Downloads/
lbdex.tar.gz .
[Gamma@localhost test]$ tar -zvxf lbdex.tar.gz
...
[Gamma@localhost test]$ mkdir src/lib/Target/Cpu0
118-165-78-111:test Jonathan$ mv lbdex src/lib/Target/Cpu0/.
[Gamma@localhost test]$ cp -rf lbdex/src_files_modify/
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/MCExpr.h:    VK_Cpu0_GPREL,
src/include//llvm/MC/MCExpr.h:    VK_Cpu0_GOT_CALL,
```

```
...
[Gamma@localhost test]$ cd src/lib/Target/Cpu0/lbdex/
[Gamma@localhost lbdex]$ sh removecpu0.sh
[Gamma@localhost lbdex]$ ls ../
lbdex
[Gamma@localhost lbdex]$ cp -rf Chapter2/* ../.
[Gamma@localhost lbdex]$ ls ..
CMakeLists.txt          Cpu0InstrInfo.td      Cpu0TargetMachine.cpp  TargetInfo
Cpu0.h                  Cpu0RegisterInfo.td  ExampleCode          readme
Cpu0.td                 Cpu0Schedule.td      LLVMBuild.txt
Cpu0InstrFormats.td     Cpu0Subtarget.h     MCTargetDesc
[Gamma@localhost Cpu0]$ cd ../../../../..
[Gamma@localhost test]$ pwd
/usr/local/llvm/test
```

Now, go into directory llvm/test/, create directory cmake\_debug\_build and do cmake like build the llvm/release, but we do Debug build and use clang as our compiler instead, as follows,

```
[Gamma@localhost test]$ pwd
/usr/local/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 -G "Unix Makefiles" ../src/
-- The C compiler identification is Clang 3.3.0
-- The CXX compiler identification is Clang 3.3.0
-- Check for working C compiler: /usr/local/llvm/release/cmake_release_build/bin/
clang
-- Check for working C compiler: /usr/local/llvm/release/cmake_release_build/bin/
clang
  -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/local/llvm/release/cmake_release_build/
bin/clang++
-- Check for working CXX compiler: /usr/local/llvm/release/cmake_release_build/
bin/clang++
  -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done ...
-- Targeting Mips
-- Targeting Cpu0
-- Targeting MBBlaze
-- Targeting MSP430
-- Targeting PowerPC
-- Targeting PTX
-- Targeting Sparc
-- Targeting X86
-- Targeting XCore
-- Configuring done
-- Generating done
-- Build files have been written to: /usr/local/llvm/test/cmake_debug
_build
[Gamma@localhost cmake_debug_build]$
```

Then do make as follows,

```
[Gamma@localhost cmake_debug_build]$ make
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/APInt.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]$
```

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
/usr/local/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 /usr/local/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 /usr/local/llvm/test/cmake_debug_build/bin/llc.
..done.
(gdb) break MipsTargetInfo.cpp:19
Breakpoint 1 at 0xd54441: file /usr/local/llvm/test/src/lib/Target/
Mips/TargetInfo/MipsTargetInfo.cpp, line 19.
(gdb) run
Starting program: /usr/local/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 /usr/local/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 <sup>13</sup> according your Linux distribution. Files compare tools Kdiff3 came from web site <sup>8</sup>.

## 14.3 Install sphinx

Sphinx install in <http://docs.geoserver.org/latest/en/docguide/install.html>.

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,

```
sudo apt-get install texlive texlive-latex-extra
```

or

```
sudo yum install texlive texlive-latex-extra
```

In Fedora 17, the texlive-latex-extra is missing. We install the package which include the pdflatex instead. For instance, we install pdfjam on Fedora 17 as follows,

```
[root@localhost BackendTutorial]# yum list pdfjam  
Loaded plugins: langpacks, presto, refresh-packagekit  
Installed Packages  
pdfjam.noarch           2.08-3.fc17                               @fedora  
[root@localhost BackendTutorial]#
```

Now, this book html/pdf can be generated by the following commands.

```
[Gamma@localhost BackendTutorial]# pwd  
/home/Gamma/test/lbd/docs/BackendTutorial  
[Gamma@localhost BackendTutorial]# make html  
...  
[Gamma@localhost BackendTutorial]# make latexpdf  
...
```

---

<sup>13</sup> <http://www.graphviz.org/Download.php>

# TODO LIST

---

**Todo**

Add info about LLVM documentation licensing.

---

(The *original entry* is located in /home/cschen/test/lbd/source/about.rst, line 197.)

---

**Todo**

Find information on debugging LLVM within Xcode for Macs.

---

(The *original entry* is located in /home/cschen/test/lbd/source/install.rst, line 33.)

---

**Todo**

Find information on building/debugging LLVM within Eclipse for Linux.

---

(The *original entry* is located in /home/cschen/test/lbd/source/install.rst, line 34.)

---

**Todo**

Fix centering for figure captions.

---

(The *original entry* is located in /home/cschen/test/lbd/source/install.rst, line 43.)

---

**Todo**

I might want to re-edit the following paragraph

---

(The *original entry* is located in /home/cschen/test/lbd/source/llvmstructure.rst, line 764.)

---



# BOOK EXAMPLE CODE

The example code `lbdex.tar.gz` is available in:

<http://jonathan2251.github.com/lbd/lbdex.tar.gz>

or

<https://www.dropbox.com/sh/2pkh1fewlq2zag9/r9n4gnqPm7/>



---

CHAPTER  
**SEVENTEEN**

---

## **ALTERNATE FORMATS**

The book is also available in the following formats: