
Tutorial: Creating an LLVM Backend for the Cpu0 Architecture

Release 3.3.2

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

September 17, 2013

CONTENTS

1	About	3
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	5
1.9	Outline of Chapters	6
2	Cpu0 Instruction Set and LLVM Target Description	9
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	17
2.5	Write cmake file	27
2.6	Target Registration	29
2.7	Build libraries and td	31
3	Backend structure	35
3.1	TargetMachine structure	35
3.2	Add AsmPrinter	61
3.3	LLVM Code Generation Sequence	77
3.4	DAG (Directed Acyclic Graph)	81
3.5	Instruction Selection	81
3.6	Add Cpu0DAGToDAGISel class	84
3.7	Add Prologue/Epilogue functions	90
3.8	Summary of this Chapter	106
4	Arithmetic, local pointer and logic lsupport	109
4.1	Arithmetic	109
4.2	Logic	138
4.3	Summary	146
5	Generating object files	147
5.1	Translate into obj file	147
5.2	Backend Target Registration Structure	148
6	Global variables, structs and arrays, other type	161

6.1	Global variable	161
6.2	Array and struct support	187
6.3	Type of char, short int and bool	193
7	Control flow statements	199
7.1	Control flow statement	199
7.2	RISC CPU knowledge	214
8	Function call	215
8.1	Mips stack frame	215
8.2	Load incoming arguments from stack frame	220
8.3	Store outgoing arguments to stack frame	228
8.4	Fix issues	237
8.5	Support features	253
8.6	Summary of this chapter	279
9	ELF Support	281
9.1	ELF format	281
9.2	ELF header and Section header table	283
9.3	Relocation Record	284
9.4	Cpu0 ELF related files	289
9.5	lld	289
9.6	llvm-objdump	290
9.7	Dynamic link	302
10	Run backend	315
10.1	AsmParser support	315
10.2	Verilog of CPU0	340
10.3	Run program on CPU0 machine	347
11	Backend Optimization	355
11.1	Cpu0 backend Optimization: Remove useless JMP	355
11.2	Cpu0 Optimization: Redesign instruction sets	359
12	LLD for Cpu0	379
12.1	Install lld	379
12.2	Cpu0 lld	382
12.3	ELF to Hex	405
12.4	Run	416
12.5	Summary	425
13	Appendix A: Getting Started: Installing LLVM and the Cpu0 example code	427
13.1	Setting Up Your Mac	427
13.2	Setting Up Your Linux Machine	444
13.3	Install sphinx	448
14	Todo List	449
15	Book example code	451
16	Alternate formats	453

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

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

Anoushe Jamshidi ajamshidi@gmail.com

1.2 Contributors

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

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

1.3 Acknowledgments

We would like to thank Sean Silva, 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, 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> in va_arg question answer.

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

1.5 Revision history

Version 3.3.2, Not release yet

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 `externalsym` 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 by 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 backends 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, local pointer 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 `+`, `-`, `*`, `/`, `&`, `l`, `^`, `<<`, `>>`, `!` 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, structs and arrays, other type:

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.

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 ¹. 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 ² (Chinese) and ³ (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:

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

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

³ English translation of Cpu0 description. http://translate.google.com.tw/translate?js=n&prev=_t&hl=zh-TW&ie=UTF-8&layout=2&eotf=1&sl=zh-CN&tl=en&u=http://ccckmit.wikidot.com/ocs:cpu0

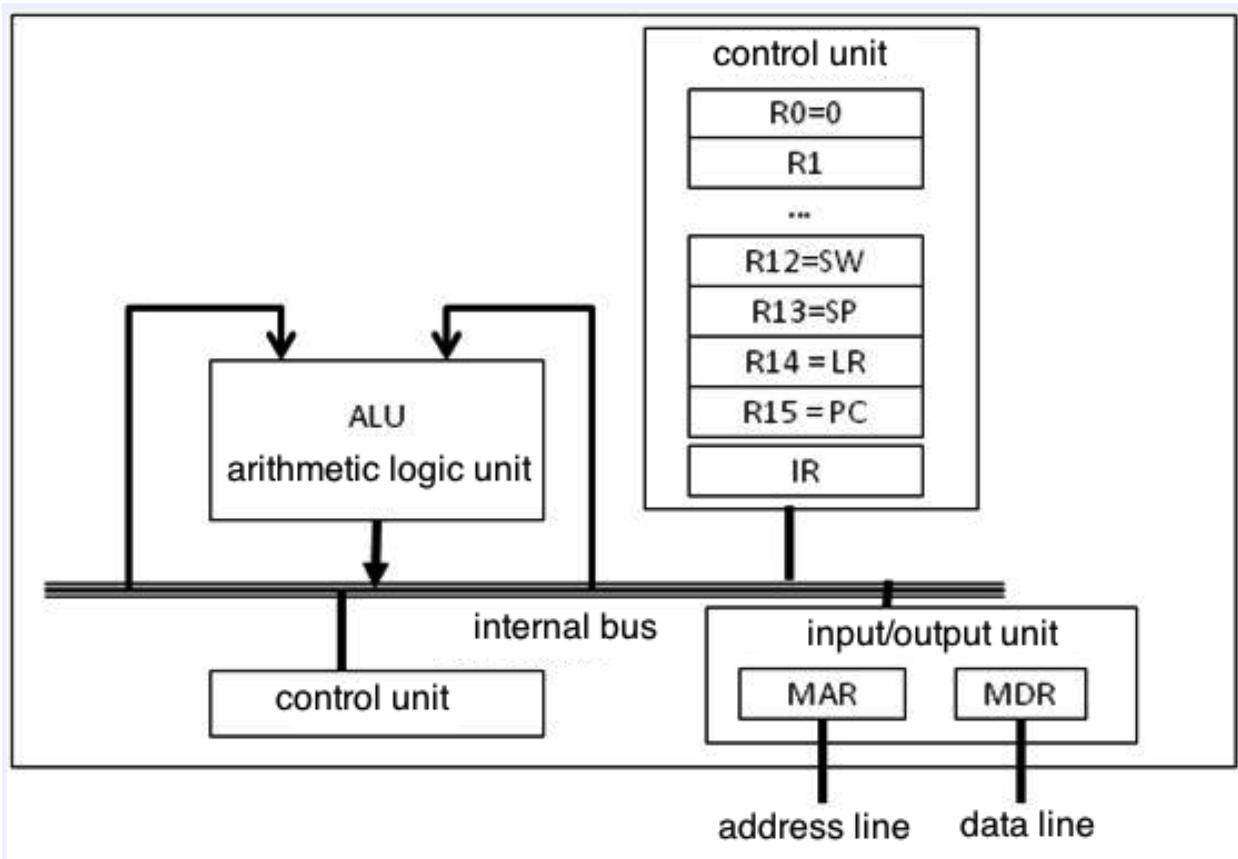


Figure 2.1: Architectural block diagram of the Cpu0 processor

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

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.

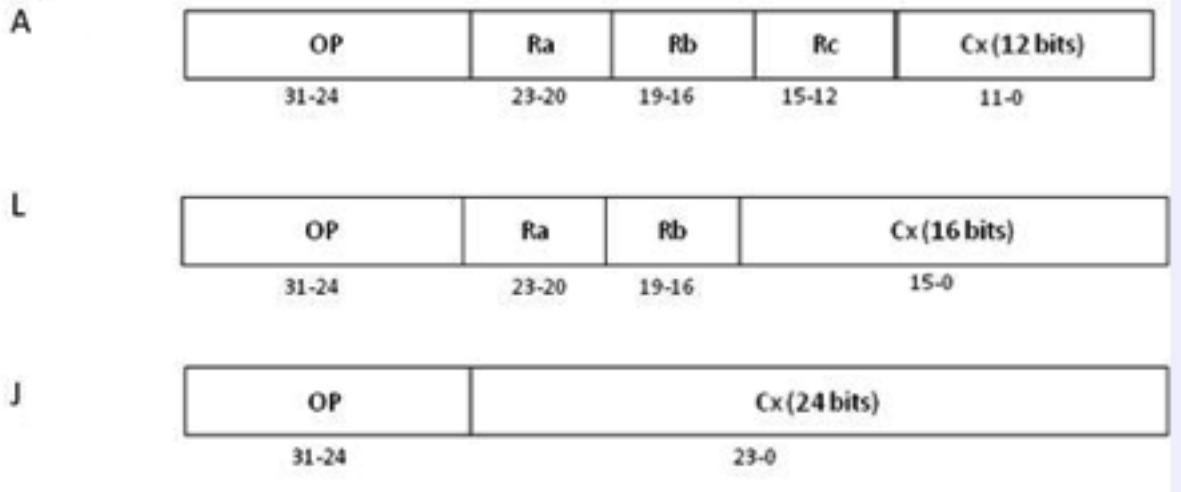


Figure 2.2: Cpu0's three instruction formats

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
L	LB	03	Load byte	LB Ra, [Rb+Cx]	Ra <= (byte)[Rb+Cx]

Continued on next page

Table 2.2 – continued from previous page

F.	Mnemonic	Opcode	Meaning	Syntax	Operation
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) ⁴
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 ⁵
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 ⁴
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	Jump to subroutine	JR Rb	LR <= PC; PC <= Rb
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
L	MFLO	47	Move LO to GPR	MFLO Ra	Ra <= LO
L	MTHI	48	Move GPR to HI	MTHI Ra	HI <= Ra

Continued on next page

⁴ 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.

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

Table 2.2 – continued from previous page

F.	Mnemonic	Opcode	Meaning	Syntax	Operation
L	MTLO	49	Move GPR to LO	MTLO Ra	LO <= Ra

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.

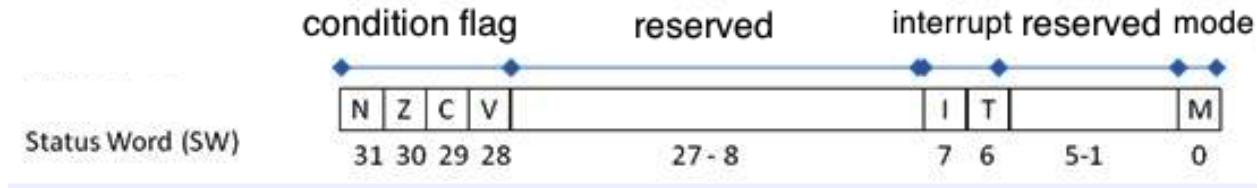


Figure 2.3: Cpu0 status word (SW) register

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 Lattner⁵.

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.

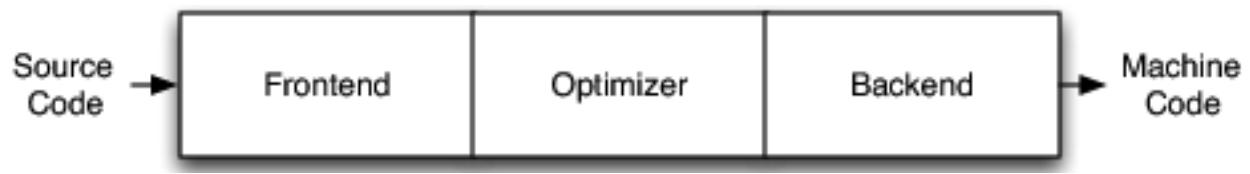


Figure 2.4: Three Major Components of a Three Phase Compiler

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.

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.

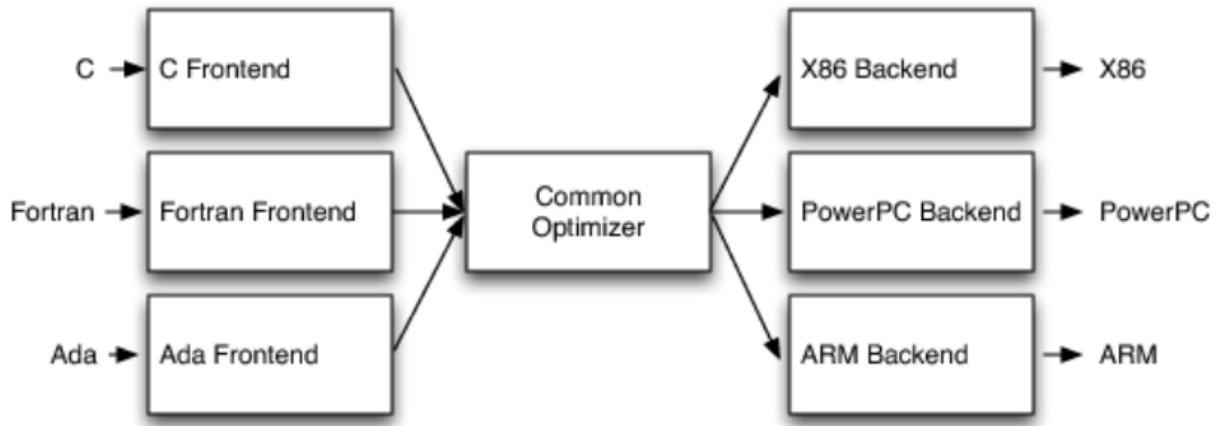


Figure 2.5: Retargetability

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.

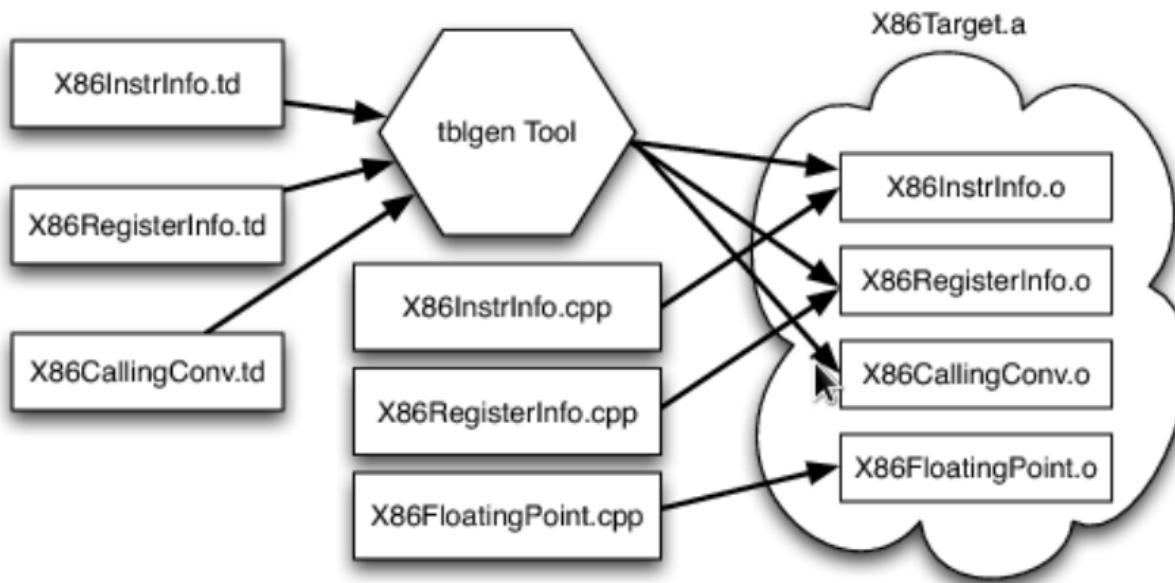


Figure 2.6: Simplified x86 Target Definition

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

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

2.4 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 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 ⁶ 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:

LLVMBackendTutorialExampleCode/Chapter2/Cpu0.td

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

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

include "llvm/Target/Target.td"

//-----=====
// Register File, Calling Conv, Instruction Descriptions
//-----=====

include "Cpu0RegisterInfo.td"
include "Cpu0Schedule.td"
include "Cpu0InstrInfo.td"

def Cpu0InstrInfo : InstrInfo;

def Cpu0 : Target {
// def Cpu0InstrInfo : InstrInfo as before.
  let InstructionSet = Cpu0InstrInfo;
}
```

⁶ <http://llvm.org/docs/TableGenFundamentals.html>

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.

LLVMBackendTutorialExampleCode/Chapter2/Cpu0RegisterInfo.td

```
===== Cpu0RegisterInfo.td - Cpu0 Register defs -----*- tablegen -*=====//
//  
//  
//          The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====-----=====//  
//-----=====//  
// Declarations that describe the CPU0 register file  
//=====-----=====//  
  
// We have banks of 16 registers each.  
class Cpu0Reg<string n> : Register<n> {  
    field bits<4> Num;  
    let Namespace = "Cpu0";  
}  
  
// Cpu0 CPU Registers  
class Cpu0GPRReg<bits<4> num, string n> : Cpu0Reg<n> {  
    let Num = num;  
}  
  
//-----=====//  
// Registers  
//-----=====//  
// The register string, such as "9" or "gp" will show on "llvm-objdump -d"  
let Namespace = "Cpu0" in {  
    // General Purpose Registers  
    def ZERO : Cpu0GPRReg< 0, "zero">, DwarfRegNum<[0]>;  
    def AT : Cpu0GPRReg< 1, "1">, DwarfRegNum<[1]>;  
    def V0 : Cpu0GPRReg< 2, "2">, DwarfRegNum<[2]>;  
    def V1 : Cpu0GPRReg< 3, "3">, DwarfRegNum<[3]>;  
    def A0 : Cpu0GPRReg< 4, "4">, DwarfRegNum<[6]>;  
    def A1 : Cpu0GPRReg< 5, "5">, DwarfRegNum<[7]>;  
    def T9 : Cpu0GPRReg< 6, "t9">, DwarfRegNum<[6]>;  
    def S0 : Cpu0GPRReg< 7, "7">, DwarfRegNum<[7]>;  
    def S1 : Cpu0GPRReg< 8, "8">, DwarfRegNum<[8]>;  
    def S2 : Cpu0GPRReg< 9, "9">, DwarfRegNum<[9]>;  
    def GP : Cpu0GPRReg< 10, "gp">, DwarfRegNum<[10]>;  
    def FP : Cpu0GPRReg< 11, "fp">, DwarfRegNum<[11]>;  
    def SW : Cpu0GPRReg< 12, "sw">, DwarfRegNum<[12]>;  
    def SP : Cpu0GPRReg< 13, "sp">, DwarfRegNum<[13]>;  
    def LR : Cpu0GPRReg< 14, "lr">, DwarfRegNum<[14]>;  
    def PC : Cpu0GPRReg< 15, "pc">, DwarfRegNum<[15]>;  
// def MAR : Register< 16, "mar">, DwarfRegNum<[16]>;
```

```

//  def MDR  : Register< 17, "mdr">,  DwarfRegNum<[17]>;
}

//=====
// 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,
    // Callee save
    S0, S1, S2,
    // Reserved
    GP, FP,
    // Not preserved across procedure calls
    SW,
    // Reserved
    SP, LR, PC)>;

```

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.

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,

LLVMBackendTutorialExampleCode/Chapter2/Cpu0InstrInfo.td

```
===== Cpu0InstrInfo.td - Target Description for Cpu0 Target -*- tablegen -*-//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====//  
//  
// This file contains the Cpu0 implementation of the TargetInstrInfo class.  
//  
//=====//  
//=====//  
// Instruction format superclass  
//=====//  
  
include "Cpu0InstrFormats.td"  
  
//=====//  
// Cpu0 profiles and nodes  
//=====//  
  
def SDT_Cpu0Ret : SDTypeProfile<0, 1, [SDTCisInt<0>]>;  
  
// Return  
def Cpu0Ret : SDNode<"Cpu0ISD::Ret", SDT_Cpu0Ret, [SDNPHasChain,  
SDNPOptInGlue]>;  
  
//=====//  
// Cpu0 Operand, Complex Patterns and Transformations Definitions.  
//=====//  
  
// Signed Operand  
def simm16 : Operand<i32> {  
    let DecoderMethod= "DecodeSimm16";  
}  
  
// Address operand  
def mem : Operand<i32> {  
    let PrintMethod = "printMemOperand";  
    let MIOOperandInfo = (ops CPUREgs, simm16);  
    let EncoderMethod = "getMemEncoding";  
}  
  
// Node immediate fits as 16-bit sign extended on target immediate.  
// e.g. addi, andi  
def immSExt16 : PatLeaf<(imm), [{ return isInt<16>(N->getSExtValue()); }]>;  
  
// Cpu0 Address Mode! SDNode frameindex could possibily be a match
```

```

// since load and store instructions from stack used it.
def addr : ComplexPattern<iPTR, 2, "SelectAddr", [frameindex], [SDNPWantParent]>;

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

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

class AlignedStore<PatFrag Node> :
    PatFrag<(ops node:$val, node:$ptr), (Node node:$val, node:$ptr), [{{
        StoreSDNode *SD = cast<StoreSDNode>(N);
        return SD->getMemoryVT().getSizeInBits()/8 <= SD->getAlignment();
    }}>;

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

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

// Arithmetic and logical instructions with 2 register operands.
class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,
    Operand Od, PatLeaf imm_type, RegisterClass RC> :
    FL<op, (outs 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;
}

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

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

class StoreM<bits<8> op, string instr_asm, PatFrag OpNode, RegisterClass RC,
    Operand MemOpnd, bit Pseudo>:
    FMem<op, (outs), (ins RC:$ra, MemOpnd:$addr),
        !strconcat(instr_asm, "\t$ra, $addr"),

```

```

        [(OpNode RC:$ra, addr:$addr)], IIStore> {
    let isPseudo = Pseudo;
}

// 32-bit load.
multiclass LoadM32<bits<8> op, string instr_asm, PatFrag OpNode,
    bit Pseudo = 0> {
    def #NAME# : LoadM<op, instr_asm, OpNode, CPURegs, mem, Pseudo>;
}

// 32-bit store.
multiclass StoreM32<bits<8> op, string instr_asm, PatFrag OpNode,
    bit Pseudo = 0> {
    def #NAME# : StoreM<op, instr_asm, OpNode, CPURegs, mem, Pseudo>;
}

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

//=====
// Cpu0I Instructions
//=====

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

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

let isReturn=1, isTerminator=1, hasDelaySlot=1, isCodeGenOnly=1,
    isBarrier=1, hasCtrlDep=1 in
    def RET : FJ <0x2c, (outs), (ins CPURegs:$target),
        "ret\t$target", [(Cpu0Ret CPURegs:$target)], IIBranch>;

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

// Small immediates

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

```

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

LLVMBackendTutorialExampleCode/Chapter2/Cpu0InstrFormats.td

```

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

```

```
//=====//  
//=====//  
// Describe CPU0 instructions format  
//  
// CPU INSTRUCTION FORMATS  
//  
// opcode - operation code.  
// ra - dst reg, only used on 3 reg instr.  
// rb - src reg.  
// rc - src reg (on a 3 reg instr).  
// cx - immediate  
//  
//=====//  
// Format specifies the encoding used by the instruction. This is part of the  
// ad-hoc solution used to emit machine instruction encodings by our machine  
// code emitter.  
class Format<bits<4> val> {  
    bits<4> Value = val;  
}  
  
def Pseudo      : Format<0>;  
def FrmA        : Format<1>;  
def FrmL        : Format<2>;  
def FrmJ        : Format<3>;  
def FrmOther    : Format<4>; // Instruction w/ a custom format  
  
// Generic Cpu0 Format  
class Cpu0Inst<dag outs, dag ins, string asmstr, list<dag> pattern,  
                  InstrItinClass itin, Format f>: Instruction  
{  
    field bits<32> Inst;  
    Format Form = f;  
  
    let Namespace = "Cpu0";  
  
    let Size = 4;  
  
    bits<8> Opcode = 0;  
  
    // Top 8 bits are the 'opcode' field  
    let Inst{31-24} = Opcode;  
  
    let OutOperandList = outs;  
    let InOperandList = ins;  
  
    let AsmString    = asmstr;  
    let Pattern      = pattern;  
    let Itinerary    = itin;  
  
    //  
    // Attributes specific to Cpu0 instructions...  
    //  
    bits<4> FormBits = Form.Value;  
  
    // TSFlags layout should be kept in sync with Cpu0InstrInfo.h.
```

```

let TSFlags{3-0}      = FormBits;

let DecoderNamespace = "Cpu0";

field bits<32> SoftFail = 0;
}

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

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

    let Opcode = op;

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

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

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

    let Opcode = op;

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

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

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

    let Opcode = op;

    let Inst{23-0} = addr;

```

}

ADDiu is class ArithLogicI inherited from FL, can expand 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:$rb, imm_type:$simm16))], 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:$simm16),
!strconcat(instr_asm, "\t$ra, $rb, $simm16"),
[(set RC:$ra, (OpNode RC:$rb, imm_type:$simm16))], 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,simm16:$simm16
asmstr = "addiu\t$ra, $rb, $simm16"
pattern = [(set CPURegs:$ra, (add RC:$rb, immSExt16:$simm16)) ]
itin = IIAlu

Members are,
ra = CPURegs:$ra
rb = CPURegs:$rb
imm16 = simm16:$simm16
Opcode = 0x09;
Inst{23-20} = CPURegs:$ra;
Inst{19-16} = CPURegs:$rb;
Inst{15-0} = simm16:$simm16;
```

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.

2.5 Write cmake file

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

LLVMBackendTutorialExampleCode/Chapter2/CMakeLists.txt

```

# CMakeLists.txt
# Our td all in Cpu0.td, Cpu0RegisterInfo.td and Cpu0InstrInfo.td included in
# Cpu0.td.
set(LLVM_TARGET_DEFINITIONS Cpu0.td)

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

# Cpu0CommonTableGen must be defined
add_public_tablegen_target(Cpu0CommonTableGen)

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

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

```

LLVMBackendTutorialExampleCode/Chapter2/LLVMBuild.txt

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

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

[common]
subdirectories = MCTargetDesc TargetInfo

[component_0]
# TargetGroup components are an extension of LibraryGroups, specifically for
# defining LLVM targets (which are handled specially in a few places).
type = TargetGroup
# The name of the component should always be the name of the target. (should
# match "def Cpu0 : Target" in Cpu0.td)
name = Cpu0
# Cpu0 component is located in directory Target/
parent = Target
# Whether this target defines an assembly parser, assembly printer, disassembler
# , and supports JIT compilation. They are optional.
#has_asmparser = 1
#has_asmprinter = 1
#has_disassembler = 1
#has_jit = 1

[component_1]
# component_1 is a Library type and name is Cpu0CodeGen. After build it will
# in lib/libLLVMCpu0CodeGen.a of your build command directory.
type = Library
name = Cpu0CodeGen
# Cpu0CodeGen component (Library) is located in directory Cpu0/
parent = Cpu0
# If given, a list of the names of Library or LibraryGroup components which
# must also be linked in whenever this library is used. That is, the link time
# dependencies for this component. When tools are built, the build system will
# include the transitive closure of all required_libraries for the components
# the tool needs.
required_libraries = CodeGen Core MC Cpu0Desc Cpu0Info SelectionDAG Support
                    Target
# All LLVMBuild.txt in Target/Cpu0 and subdirectory use 'add_to_library_groups
# = Cpu0'
add_to_library_groups = Cpu0
```

CMakeLists.txt is the make information for cmake, # 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” ⁷ and “LLVMBuild Guide” ⁸.

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.

LLVMBackendTutorialExampleCode/Chapter2/TargetInfo/Cpu0TargetInfo.cpp

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

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

Target llvm::TheCpu0Target, llvm::TheCpu0elTarget;

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

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

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

⁷ <http://llvm.org/docs/CMake.html>

⁸ <http://llvm.org/docs/LLVMBuild.html>

LLVMBackendTutorialExampleCode/Chapter2/MCTargetDesc/Cpu0MCTargetDesc.h

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

#ifndef CPU0MCTARGETDESC_H
#define CPU0MCTARGETDESC_H

namespace llvm {
class Target;

extern Target TheCpu0Target;
extern Target TheCpu0elTarget;
} // End llvm namespace

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

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

#define GET_SUBTARGETINFO_ENUM
#include "Cpu0GenSubtargetInfo.inc"
#endif
```

LLVMBackendTutorialExampleCode/Chapter2/MCTargetDesc/Cpu0MCTargetDesc.cpp

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

#include "Cpu0MCTargetDesc.h"
#include "llvm/MC/MachineLocation.h"
#include "llvm/MC/MCCodeGenInfo.h"
```

```

#include "llvm/MC/MCInstrInfo.h"
#include "llvm/MC/MCRegisterInfo.h"
#include "llvm/MC/MCStreamer.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/TargetRegistry.h"

#define GET_INSTRINFO_MC_DESC
#include "Cpu0GenInstrInfo.inc"

#define GET_SUBTARGETINFO_MC_DESC
#include "Cpu0GenSubtargetInfo.inc"

#define GET_REGINFO_MC_DESC
#include "Cpu0GenRegisterInfo.inc"

using namespace llvm;

extern "C" void LLVMInitializeCpu0TargetMC() {
}

```

Please see “Target Registration”⁹ 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 refer¹⁰. 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_modify/src_files_modified/src/.

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

```
cp -rf LLVMBackendTutorialExampleCode/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),

⁹ <http://llvm.org/docs/WritingAnLLVMBackend.html#target-registration>

¹⁰ http://clang.llvm.org/get_started.html

```
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$
```

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 following code from file Target-Info/Cpu0TargetInfo.cpp what in “section Target Registration” ¹¹ we made. List them as follows again,

LLVMBackendTutorialExampleCode/Chapter2/TargetInfo/Cpu0TargetInfo.cpp

```
===== Cpu0TargetInfo.cpp - Cpu0 Target Implementation =====//  
//  
//           The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
=====//  
  
#include "Cpu0.h"  
#include "llvm/IR/Module.h"  
#include "llvm/Support/TargetRegistry.h"  
using namespace llvm;  
  
Target llvm::TheCpu0Target, llvm::TheCpu0elTarget;  
  
extern "C" void LLVMInitializeCpu0TargetInfo() {  
    RegisterTarget<Triple::cpu0,  
        /*HasJIT=*/true> X(TheCpu0Target, "cpu0", "Cpu0");  
  
    RegisterTarget<Triple::cpu0el,  
        /*HasJIT=*/true> Y(TheCpu0elTarget, "cpu0el", "Cpu0el");  
}
```

¹¹ <http://jonathan2251.github.com/lbd/llvmstructure.html#target-registration>

Let's build LLVMBackendTutorialExampleCode/Chapter2 code as follows,

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

LLVMBackendTutorialExampleCode/InputFiles/ch3.cpp

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

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

```
[Gamma@localhost InputFiles]$ clang -c ch3.cpp -emit-llvm -o ch3.bc
```

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
// 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,
```

```
line 271.
```

```
...
```

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/Cpu0TargetMachine.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);
...
};

```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0TargetMachine.h

```

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

#ifndef CPU0TARGETMACHINE_H
#define CPU0TARGETMACHINE_H

#include "Cpu0FrameLowering.h"
#include "Cpu0InstrInfo.h"
#include "Cpu0ISelLowering.h"
#include "Cpu0SelectionDAGInfo.h"
#include "Cpu0Subtarget.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/Target/TargetFrameLowering.h"

namespace llvm {
    class formated_raw_ostream;

    class Cpu0TargetMachine : public LLVMTargetMachine {
        Cpu0Subtarget      Subtarget;
        const DataLayout    DL; // Calculates type size & alignment
        Cpu0InstrInfo      InstrInfo; //-- Instructions
        Cpu0FrameLowering   FrameLowering; //-- Stack(Frame) and Stack direction
        Cpu0TargetLowering  TLInfo; //-- Stack(Frame) and Stack direction
        Cpu0SelectionDAGInfo TSInfo; //-- Map .bc DAG to backend DAG

    public:
        Cpu0TargetMachine(const Target &T, StringRef TT,
                          StringRef CPU, StringRef FS, const TargetOptions &Options,

```

```

        Reloc::Model RM, CodeModel::Model CM,
        CodeGenOpt::Level OL,
        bool isLittle);

virtual const Cpu0InstrInfo *getInstrInfo() const
{ return &InstrInfo; }

virtual const TargetFrameLowering *getFrameLowering() const
{ return &FrameLowering; }

virtual const Cpu0Subtarget *getSubtargetImpl() const
{ return &Subtarget; }

virtual const DataLayout *getDataLayout() const
{ return &DL; }

virtual const Cpu0RegisterInfo *getRegisterInfo() const {
    return &InstrInfo.getRegisterInfo();
}

virtual const Cpu0TargetLowering *getTargetLowering() const {
    return &TLInfo;
}

virtual const Cpu0SelectionDAGInfo* getSelectionDAGInfo() const {
    return &TSInfo;
}

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

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

/// Cpu0elTargetMachine - Cpu032 little endian target machine.
///
class Cpu0elTargetMachine : public Cpu0TargetMachine {
    virtual void anchor();
public:
    Cpu0elTargetMachine(const Target &T, StringRef TT,
                        StringRef CPU, StringRef FS, const TargetOptions &Options,
                        Reloc::Model RM, CodeModel::Model CM,
                        CodeGenOpt::Level OL);
};
} // End llvm namespace

#endif

```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0TargetMachine.cpp

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

#include "Cpu0TargetMachine.h"
#include "Cpu0.h"
#include "llvm/PassManager.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/Support/TargetRegistry.h"
using namespace llvm;

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

// DataLayout --> Big-endian, 32-bit pointer/ABI/alignment
// The stack is always 8 byte aligned
// On function prologue, the stack is created by decrementing
// its pointer. Once decremented, all references are done with positive
// offset from the stack/frame pointer, using StackGrowsUp enables
// an easier handling.
// Using CodeModel::Large enables different CALL behavior.
Cpu0TargetMachine::
Cpu0TargetMachine(const Target &T, StringRef TT,
                  StringRef CPU, StringRef FS, const TargetOptions &Options,
                  Reloc::Model RM, CodeModel::Model CM,
                  CodeGenOpt::Level OL,
                  bool isLittle)
// Default is big endian
: LLVMTargetMachine(T, TT, CPU, FS, Options, RM, CM, OL),
  Subtarget(TT, CPU, FS, isLittle),
  DL(isLittle ?
      ("e-p:32:32:32-i8:8:32-i16:16:32-i64:64:64-n32") :
      ("E-p:32:32:32-i8:8:32-i16:16:32-i64:64:64-n32")),
  InstrInfo(*this),
  FrameLowering(Subtarget),
  TLInfo(*this), TSInfo(*this) {
}

void Cpu0ebTargetMachine::anchor() { }

Cpu0ebTargetMachine::
Cpu0ebTargetMachine(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, false) {}

void Cpu0elTargetMachine::anchor() { }

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

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

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

} // namespace

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

```

include/llvm/Target/TargetInstrInfo.h

```

class TargetInstrInfo : public MCInstrInfo {
    TargetInstrInfo(const TargetInstrInfo &) 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:
    ...
}

```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0CallingConv.td

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

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

def CSR_O32 : CalleeSavedRegs<(add LR, FP,
                                sequence "S%u", 2, 0)>;
```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0FrameLowering.h

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

#ifndef CPU0_FRAMEINFO_H
#define CPU0_FRAMEINFO_H

#include "Cpu0.h"
#include "Cpu0Subtarget.h"
#include "llvm/Target/TargetFrameLowering.h"

namespace llvm {
    class Cpu0Subtarget;

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

public:
    explicit Cpu0FrameLowering(const Cpu0Subtarget &sti)
        : TargetFrameLowering(StackGrowsDown, 8, 0),
          STI(sti) {
    }

    /// emitProlog/emitEpilog - These methods insert prolog and epilog code into
    /// the function.
}
```

```

void emitPrologue(MachineFunction &MF) const;
void emitEpilogue(MachineFunction &MF, MachineBasicBlock &MBB) const;
bool hasFP(const MachineFunction &MF) const;
};

} // End llvm namespace

#endif

```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0FrameLowering.cpp

```

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

#include "Cpu0FrameLowering.h"
#include "Cpu0InstrInfo.h"
#include "Cpu0MachineFunction.h"
#include "llvm/IR/Function.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineModuleInfo.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/Target/TargetOptions.h"
#include "llvm/Support/CommandLine.h"

using namespace llvm;

// emitPrologue() and emitEpilogue must exist for main().

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


```

```

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

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

void Cpu0FrameLowering::emitPrologue(MachineFunction &MF) const {

void Cpu0FrameLowering::emitEpilogue(MachineFunction &MF,
                                     MachineBasicBlock &MBB) const {
}

```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0InstrInfo.h

```

//===== Cpu0InstrInfo.h - Cpu0 Instruction Information -----*- C++ -*==//
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====//
//
// This file contains the Cpu0 implementation of the TargetInstrInfo class.
//
//=====//
#ifndef CPU0INSTRUCTIONINFO_H
#define CPU0INSTRUCTIONINFO_H

#include "Cpu0.h"
#include "Cpu0RegisterInfo.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Target/TargetInstrInfo.h"

#define GET_INSTRINFO_HEADER
#include "Cpu0GenInstrInfo.inc"

namespace llvm {

class Cpu0InstrInfo : public Cpu0GenInstrInfo {
    Cpu0TargetMachine &TM;
    const Cpu0RegisterInfo RI;
public:
    explicit Cpu0InstrInfo(Cpu0TargetMachine &TM);

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

public:
};

}

#endif

```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0InstrInfo.cpp

```

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

```

```
//  
// This file contains the Cpu0 implementation of the TargetInstrInfo class.  
//  
//=====//  
  
#include "Cpu0InstrInfo.h"  
#include "Cpu0TargetMachine.h"  
#define GET_INSTRINFOCTOR  
#include "Cpu0GenInstrInfo.inc"  
  
using namespace llvm;  
  
Cpu0InstrInfo::Cpu0InstrInfo(Cpu0TargetMachine &tm)  
:  
    TM(tm),  
    RI(*TM.getSubtargetImpl(), *this) {}  
  
const Cpu0RegisterInfo &Cpu0InstrInfo::getRegisterInfo() const {  
    return RI;  
}
```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0ISelLowering.h

```
//===== Cpu0ISelLowering.h - Cpu0 DAG Lowering Interface -----*- C++ -*=====//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====//  
//  
// This file defines the interfaces that Cpu0 uses to lower LLVM code into a  
// selection DAG.  
//  
//=====//  
  
#ifndef Cpu0ISELLOWERING_H  
#define Cpu0ISELLOWERING_H  
  
#include "Cpu0.h"  
#include "Cpu0Subtarget.h"  
#include "llvm/CodeGen/SelectionDAG.h"  
#include "llvm/Target/TargetLowering.h"  
  
namespace llvm {  
    namespace Cpu0ISD {  
        enum NodeType {  
            // Start the numbering from where ISD NodeType finishes.  
            FIRST_NUMBER = ISD::BUILTIN_OP_END,  
            Ret  
        };  
    }  
}  
  
//=====//  
// TargetLowering Implementation
```

```

//=====

class Cpu0TargetLowering : public TargetLowering {
public:
    explicit Cpu0TargetLowering(Cpu0TargetMachine &TM) ;

private:
    // Subtarget Info
    const Cpu0Subtarget *Subtarget;

    // must be exist without function all
    virtual SDValue
    LowerFormalArguments(SDValue Chain,
                         CallingConv::ID CallConv, bool isVarArg,
                         const SmallVectorImpl<ISD::InputArg> &Ins,
                         DebugLoc dl, SelectionDAG &DAG,
                         SmallVectorImpl<SDValue> &InVals) const;

    // must be exist without function all
    virtual SDValue
    LowerReturn(SDValue Chain,
                CallingConv::ID CallConv, bool isVarArg,
                const SmallVectorImpl<ISD::OutputArg> &Outs,
                const SmallVectorImpl<SDValue> &OutVals,
                DebugLoc dl, SelectionDAG &DAG) const;
};

#endif // Cpu0ISELLOWERING_H

```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0ISellowering.cpp

```

//===== Cpu0ISellowering.cpp - Cpu0 DAG Lowering Implementation =====
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====

//
// This file defines the interfaces that Cpu0 uses to lower LLVM code into a
// selection DAG.
//
//=====

#define DEBUG_TYPE "cpu0-lower"
#include "Cpu0ISellowering.h"
#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/TargetLoweringObjectFileImpl.h"
#include "Cpu0Subtarget.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/GlobalVariable.h"
#include "llvm/IR/Intrinsics.h"
#include "llvm/IR/CallingConv.h"

```

```

#include "llvm/CodeGen/CallingConvLower.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/SelectionDAGISel.h"
#include "llvm/CodeGen/ValueTypes.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

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

#include "Cpu0GenCallingConv.inc"

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

//=====//
//          Return Value Calling Convention Implementation
//=====//

SDValue
Cpu0TargetLowering::LowerReturn(SDValue Chain,
                               CallingConv::ID CallConv, bool isVarArg,
                               const SmallVectorImpl<ISD::OutputArg> &Outs,
                               const SmallVectorImpl<SDValue> &OutVals,
                               DebugLoc dl, SelectionDAG &DAG) const {

    return DAG.getNode(Cpu0ISD::Ret, dl, MVT::Other,
                       Chain, DAG.getRegister(Cpu0::LR, MVT::i32));
}

```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0MachineFunction.h

```

//===== Cpu0MachineFunctionInfo.h - Private data used for Cpu0 -----*-- C++ -*--//
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source

```

```

// License. See LICENSE.TXT for details.
//
//=====
// This file declares the Cpu0 specific subclass of MachineFunctionInfo.
//
//=====

#ifndef CPU0_MACHINE_FUNCTION_INFO_H
#define CPU0_MACHINE_FUNCTION_INFO_H

#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include <utility>

namespace llvm {

/// Cpu0FunctionInfo - This class is derived from MachineFunction private
/// Cpu0 target-specific information for each MachineFunction.
class Cpu0FunctionInfo : public MachineFunctionInfo {
    MachineFunction& MF;
    unsigned MaxCallFrameSize;

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

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

} // end of namespace llvm

#endif // CPU0_MACHINE_FUNCTION_INFO_H

```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0SelectionDAGInfo.h

```

//===== Cpu0SelectionDAGInfo.h - Cpu0 SelectionDAG Info -----*-- C++ -*==//
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====

// This file defines the Cpu0 subclass for TargetSelectionDAGInfo.
//
//=====

#ifndef CPU0SELECTIONDAGINFO_H
#define CPU0SELECTIONDAGINFO_H

#include "llvm/Target/TargetSelectionDAGInfo.h"

```

```
namespace llvm {

class Cpu0TargetMachine;

class Cpu0SelectionDAGInfo : public TargetSelectionDAGInfo {
public:
    explicit Cpu0SelectionDAGInfo(const Cpu0TargetMachine &TM);
    ~Cpu0SelectionDAGInfo();
};

}

#endif
```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0SelectionDAGInfo.cpp

```
===== Cpu0SelectionDAGInfo.cpp - Cpu0 SelectionDAG Info =====
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
=====

// This file implements the Cpu0SelectionDAGInfo class.
//
=====

#define DEBUG_TYPE "cpu0-selectiondag-info"
#include "Cpu0TargetMachine.h"
using namespace llvm;

Cpu0SelectionDAGInfo::Cpu0SelectionDAGInfo(const Cpu0TargetMachine &TM)
    : TargetSelectionDAGInfo(TM) {
}

Cpu0SelectionDAGInfo::~Cpu0SelectionDAGInfo() {
```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0Subtarget.h

```
#define GET_SUBTARGETINFO_HEADER
#include "Cpu0GenSubtargetInfo.inc"
...
class Cpu0Subtarget : public Cpu0GenSubtargetInfo {
...
// Virtual function, must have
/// ParseSubtargetFeatures - Parses features string setting specified
/// subtarget options. Definition of function is auto generated by tblgen.
void ParseSubtargetFeatures(StringRef CPU, StringRef FS);
...
}
```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0Subtarget.cpp

```

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

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

using namespace llvm;

void Cpu0Subtarget::anchor() { }

Cpu0Subtarget::Cpu0Subtarget(const std::string &TT, const std::string &CPU,
                           const std::string &FS, bool little) :
    Cpu0GenSubtargetInfo(TT, CPU, FS),
    Cpu0ABI(UnknownABI), IsLittle(little)
{
    std::string CPUName = CPU;
    if (CPUName.empty())
        CPUName = "cpu032";

    // Parse features string.
    ParseSubtargetFeatures(CPUName, FS);

    // Initialize scheduling itinerary for the specified CPU.
    InstrItins = getInstrItineraryForCPU(CPUName);

    // Set Cpu0ABI if it hasn't been set yet.
    if (Cpu0ABI == UnknownABI)
        Cpu0ABI = O32;
}

```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0RegisterInfo.h

```

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

```

```

//=====//
// This file contains the Cpu0 implementation of the TargetRegisterInfo class.
//=====//

#ifndef CPU0REGISTERINFO_H
#define CPU0REGISTERINFO_H

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

#define GET_REGINFO_HEADER
#include "Cpu0GenRegisterInfo.inc"

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

struct Cpu0RegisterInfo : public Cpu0GenRegisterInfo {
    const Cpu0Subtarget &Subtarget;
    const TargetInstrInfo &TII;

    Cpu0RegisterInfo(const Cpu0Subtarget &Subtarget, const TargetInstrInfo &TII);

    /// getRegisterNumbering - Given the enum value for some register, e.g.
    /// Cpu0::RA, return the number that it corresponds to (e.g. 31).
    static unsigned getRegisterNumbering(unsigned RegEnum);

    /// Code Generation virtual methods...
    const uint16_t *getCalleeSavedRegs(const MachineFunction* MF = 0) const;
    const uint32_t *getCallPreservedMask(CallingConv::ID) const;

    // pure virtual method
    BitVector getReservedRegs(const MachineFunction &MF) const;

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

    // pure virtual method
    /// Debug information queries.
    unsigned getFrameRegister(const MachineFunction &MF) const;
};

} // end namespace llvm

#endif

```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0RegisterInfo.cpp

```

===== Cpu0RegisterInfo.cpp - CPU0 Register Information === -----
// The LLVM Compiler Infrastructure

```

```

// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
// -----
// This file contains the CPU0 implementation of the TargetRegisterInfo class.
// -----
#define DEBUG_TYPE "cpu0-reg-info"

#include "Cpu0RegisterInfo.h"
#include "Cpu0.h"
#include "Cpu0Subtarget.h"
#include "Cpu0MachineFunction.h"
#include "llvm/IR/Constants.h"
#include "llvm/DebugInfo.h"
#include "llvm/IR/Type.h"
#include "llvm/IR/Function.h"
#include "llvm/CodeGen/ValueTypes.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/Target/TargetFrameLowering.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/TargetOptions.h"
#include "llvm/Target/TargetInstrInfo.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/ADT/BitVector.h"
#include "llvm/ADT/STLExtras.h"

#define GET_REGINFO_TARGET_DESC
#include "Cpu0GenRegisterInfo.inc"

using namespace llvm;

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

// -----
// Callee Saved Registers methods
// -----
/// Cpu0 Callee Saved Registers
// In Cpu0CallConv.td,
// def CSR_032 : CalleeSavedRegs<(add LR, FP,
//                                     (sequence "S%u", 2, 0))>;
// llc create CSR_032_SaveList and CSR_032_RegMask from above defined.
const uint16_t* Cpu0RegisterInfo::
getCalleeSavedRegs(const MachineFunction *MF) const
{
    return CSR_032_SaveList;
}

```

```

const uint32_t*
Cpu0RegisterInfo::getCallPreservedMask(CallingConv::ID) const
{
    return CSR_O32_RegMask;
}

// 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]);

    return Reserved;
}

// pure virtual method
// FrameIndex represent objects inside a abstract stack.
// We must replace FrameIndex with an stack/frame pointer
// direct reference.
void Cpu0RegisterInfo::
eliminateFrameIndex(MachineBasicBlock::iterator II, int SPAdj,
                    unsigned FIOperandNum, RegScavenger *RS) const {
}

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

```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0TargetMachine.h

```

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

#ifndef CPU0TARGETMACHINE_H
#define CPU0TARGETMACHINE_H

```

```

#include "Cpu0FrameLowering.h"
#include "Cpu0InstrInfo.h"
#include "Cpu0ISelLowering.h"
#include "Cpu0SelectionDAGInfo.h"
#include "Cpu0Subtarget.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/Target/TargetFrameLowering.h"

namespace llvm {
    class formatted_raw_ostream;

    class Cpu0TargetMachine : public LLVMTargetMachine {
        Cpu0Subtarget Subtarget;
        const DataLayout DL; // Calculates type size & alignment
        Cpu0InstrInfo InstrInfo; // Instructions
        Cpu0FrameLowering FrameLowering; // Stack(Frame) and Stack direction
        Cpu0TargetLowering TLInfo; // Stack(Frame) and Stack direction
        Cpu0SelectionDAGInfo TSInfo; // Map .bc DAG to backend DAG

    public:
        Cpu0TargetMachine(const Target &T, StringRef TT,
                          StringRef CPU, StringRef FS, const TargetOptions &Options,
                          Reloc::Model RM, CodeModel::Model CM,
                          CodeGenOpt::Level OL,
                          bool isLittle);

        virtual const Cpu0InstrInfo *getInstrInfo() const
        { return &InstrInfo; }
        virtual const TargetFrameLowering *getFrameLowering() const
        { return &FrameLowering; }
        virtual const Cpu0Subtarget *getSubtargetImpl() const
        { return &Subtarget; }
        virtual const DataLayout *getDataLayout() const
        { return &DL; }

        virtual const Cpu0RegisterInfo *getRegisterInfo() const {
            return &InstrInfo.getRegisterInfo();
        }

        virtual const Cpu0TargetLowering *getTargetLowering() const {
            return &TLInfo;
        }

        virtual const Cpu0SelectionDAGInfo* getSelectionDAGInfo() const {
            return &TSInfo;
        }

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

    /// Cpu0ebTargetMachine - Cpu032 big endian target machine.
    ///
    class Cpu0ebTargetMachine : public Cpu0TargetMachine {
        virtual void anchor();
    public:
        Cpu0ebTargetMachine(const Target &T, StringRef TT,

```

```
StringRef CPU, StringRef FS, const TargetOptions &Options,
Reloc::Model RM, CodeModel::Model CM,
CodeGenOpt::Level OL);
};

/// Cpu0elTargetMachine - Cpu032 little endian target machine.
///
class Cpu0elTargetMachine : public Cpu0TargetMachine {
    virtual void anchor();
public:
    Cpu0elTargetMachine(const Target &T, StringRef TT,
                        StringRef CPU, StringRef FS, const TargetOptions &Options,
                        Reloc::Model RM, CodeModel::Model CM,
                        CodeGenOpt::Level OL);
};
} // End llvm namespace

#endif
```

LLVMBackendTutorialExampleCode/Chapter3_1/Cpu0TargetMachine.cpp

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

// Implements the info about Cpu0 target spec.
//
=====

#include "Cpu0TargetMachine.h"
#include "Cpu0.h"
#include "llvm/PassManager.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/Support/TargetRegistry.h"
using namespace llvm;

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

// DataLayout --> Big-endian, 32-bit pointer/ABI/alignment
// The stack is always 8 byte aligned
// On function prologue, the stack is created by decrementing
// its pointer. Once decremented, all references are done with positive
// offset from the stack/frame pointer, using StackGrowsUp enables
// an easier handling.
// Using CodeModel::Large enables different CALL behavior.
Cpu0TargetMachine::
```

```

Cpu0TargetMachine(const Target &T, StringRef TT,
                  StringRef CPU, StringRef FS, const TargetOptions &Options,
                  Reloc::Model RM, CodeModel::Model CM,
                  CodeGenOpt::Level OL,
                  bool isLittle)
//- Default is big endian
: LLVMTargetMachine(T, TT, CPU, FS, Options, RM, CM, OL),
  Subtarget(TT, CPU, FS, isLittle),
  DL(isLittle ?
      ("e-p:32:32:32-i8:8:32-i16:16:32-i64:64:64-n32") :
      ("E-p:32:32:32-i8:8:32-i16:16:32-i64:64:64-n32")),
  InstrInfo(*this),
  FrameLowering(Subtarget),
  TLInfo(*this), TSInfo(*this) {
}

void Cpu0ebTargetMachine::anchor() { }

Cpu0ebTargetMachine::
Cpu0ebTargetMachine(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, false) {}

void Cpu0elTargetMachine::anchor() { }

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

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

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

} // namespace

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

```

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 {
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"
//- Cpu0InstrInfo.h
class Cpu0InstrInfo : public Cpu0GenInstrInfo {
    Cpu0TargetMachine &TM;
public:
    explicit Cpu0InstrInfo(Cpu0TargetMachine &TM);
};


```

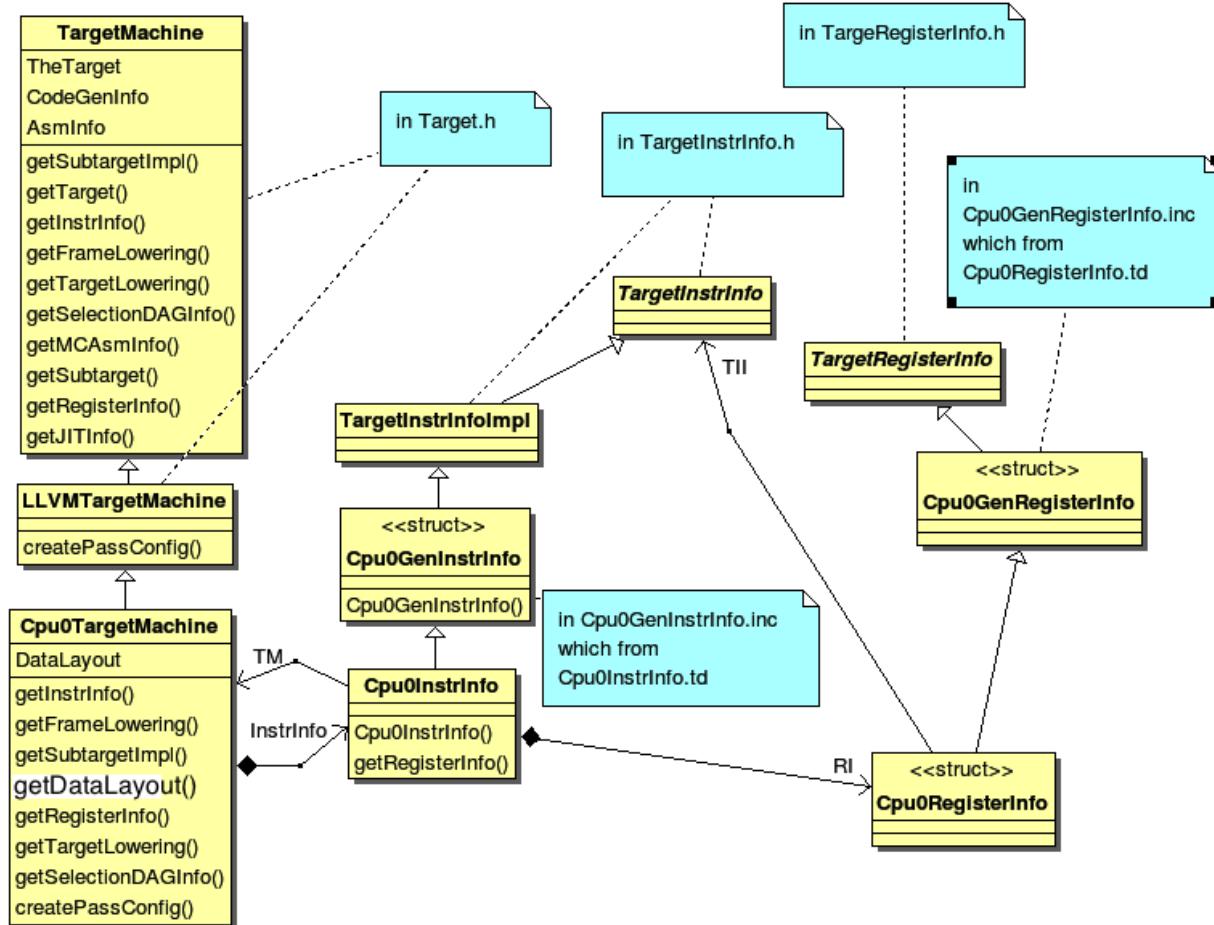


Figure 3.1: TargetMachine class diagram 1

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 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.

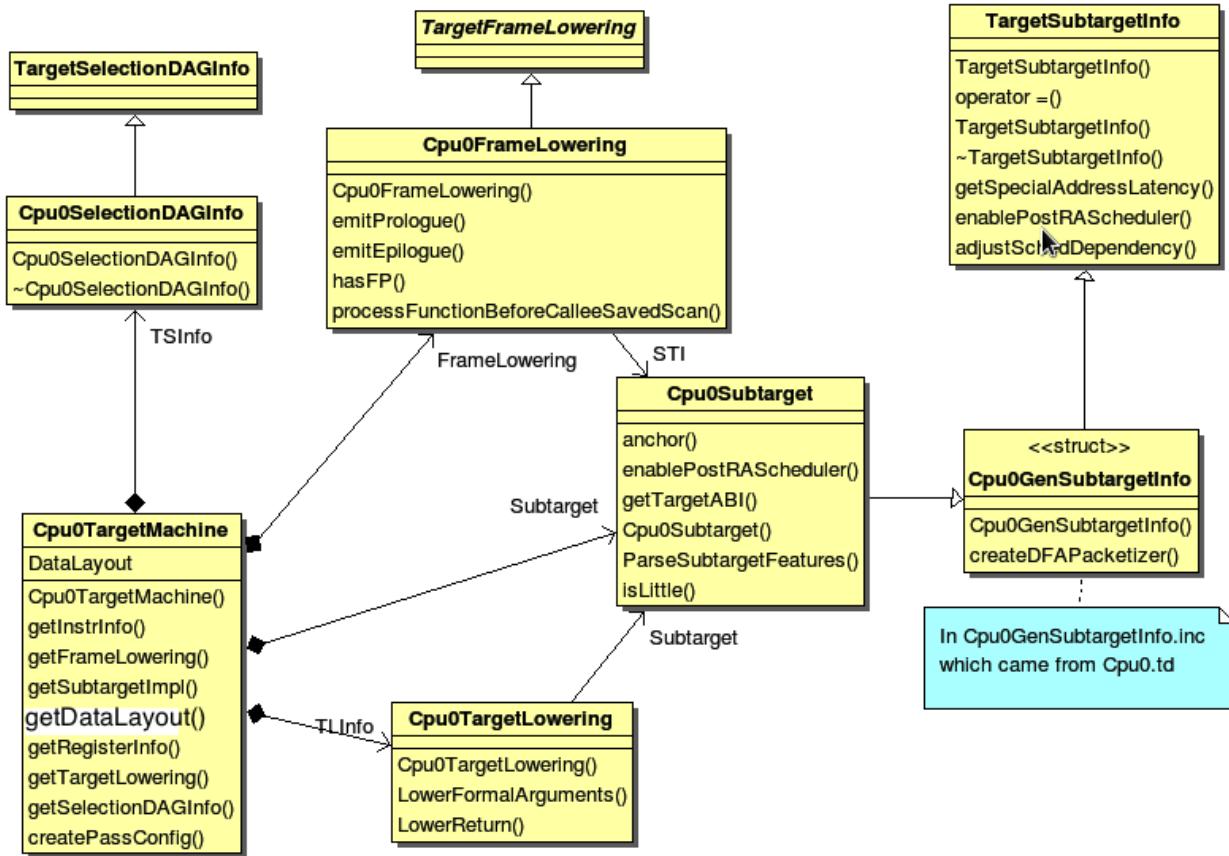


Figure 3.2: TargetMachine class diagram 2

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.

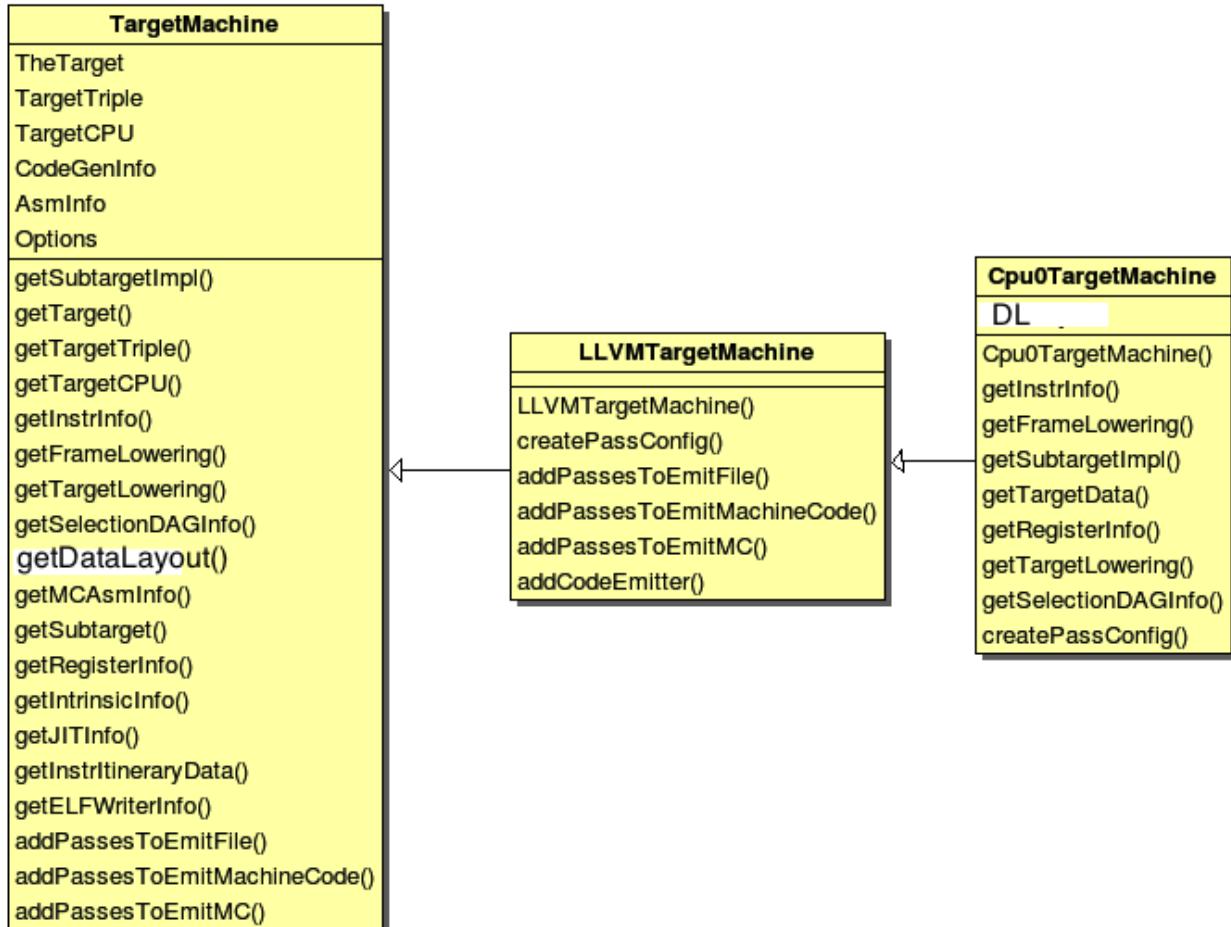


Figure 3.3: TargetMachine members and operators

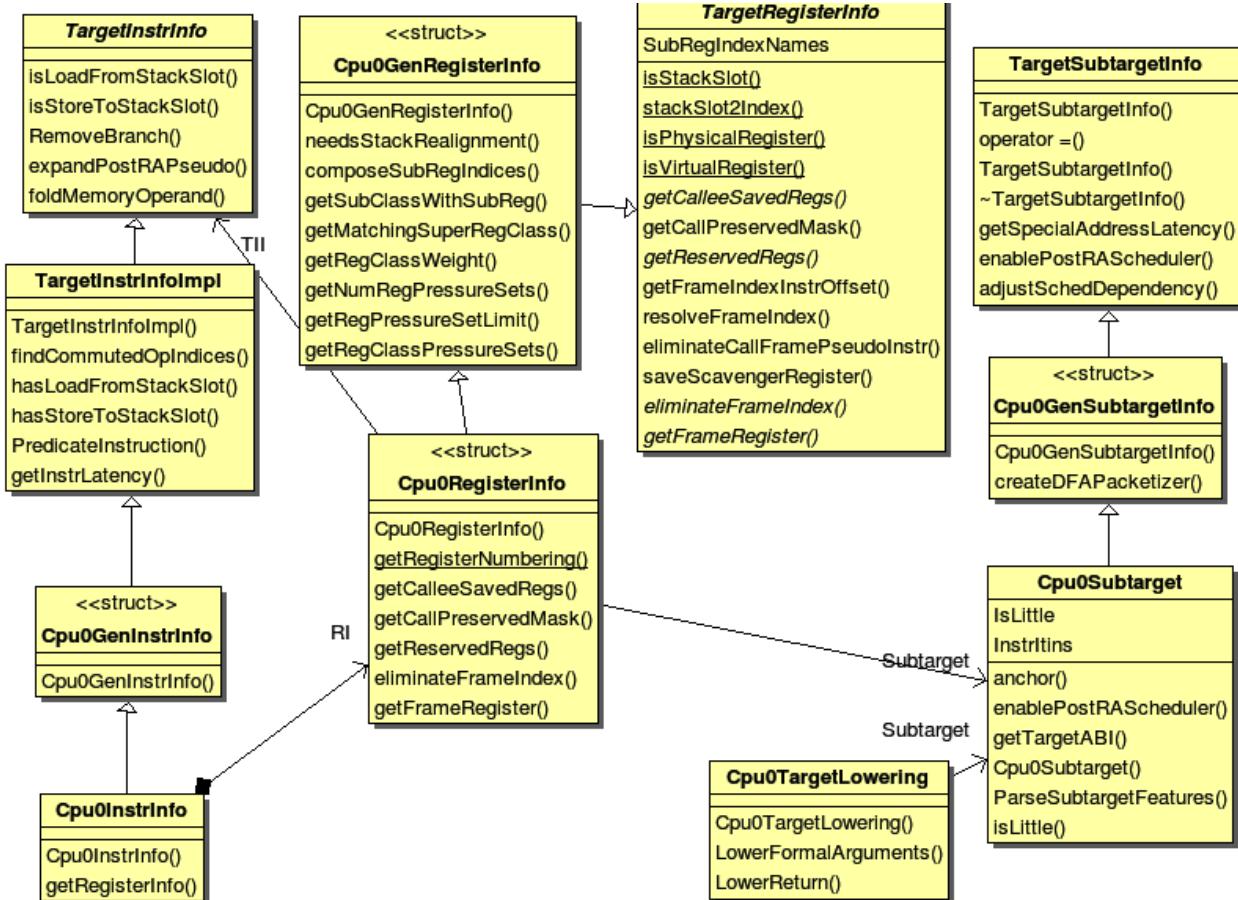


Figure 3.4: Other class members and operators

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
```

Reference Write An LLVM Backend web site¹.

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,

LLVMBackendTutorialExampleCode/Chapter3_1/CMakeLists.txt

```
# Cpu0CodeGen should match with LLVMBuild.txt Cpu0CodeGen
add_llvm_target(Cpu0CodeGen
    Cpu0ISelLowering.cpp
    Cpu0InstrInfo.cpp
    Cpu0FrameLowering.cpp
    Cpu0RegisterInfo.cpp
    Cpu0Subtarget.cpp
    Cpu0TargetMachine.cpp
    Cpu0SelectionDAGInfo.cpp
)
```

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 with src/lib/Target/Cpu0/LLVMBackendTutorialExampleCode/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 rebuilt, then your Target library will regenerate. 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/LLVMBackendTutorialExampleCode
118-165-75-57:LLVMBackendTutorialExampleCode Jonathan$ sh removecpu0.sh
118-165-75-57:LLVMBackendTutorialExampleCode 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.

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

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,

LLVMBackendTutorialExampleCode/Chapter3_2/Cpu0.td

```
// Without this will have error: 'cpu032' is not a recognized processor for
// this target (ignoring processor)
//=====
// Cpu0 Subtarget features
//=====

def FeatureCpu032      : SubtargetFeature<"cpu032", "Cpu0ArchVersion", "Cpu032",
                           "Cpu032 ISA Support">;

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

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

def : Proc<"cpu032", [FeatureCpu032]>;

def Cpu0AsmWriter : AsmWriter {
  string AsmWriterClassName  = "InstPrinter";
  bit isMCAsmWriter = 1;
}

// Will generate Cpu0GenAsmWrite.inc included by Cpu0InstPrinter.cpp, contents
// as follows,
// void Cpu0InstPrinter::printInstruction(const MCInst *MI, raw_ostream &O) {...}
// const char *Cpu0InstPrinter::getRegisterName(unsigned RegNo) {...}
def Cpu0 : Target {
// def Cpu0InstrInfo : InstrInfo as before.
  let InstructionSet = Cpu0InstrInfo;
  let AssemblyWriters = [Cpu0AsmWriter];
}
```

As comments indicate, it will generate Cpu0GenAsmWrite.inc which is included by Cpu0InstPrinter.cpp. 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 as shown in last section.

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,

LLVMBackendTutorialExampleCode/Chapter3_2/Cpu0MCInstLower.h

```
===== Cpu0MCInstLower.h - Lower MachineInstr to MCInst -----* C++ *=====//
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
=====//
```

```
#ifndef CPU0MCINSTLOWER_H
#define CPU0MCINSTLOWER_H
#include "llvm/ADT/SmallVector.h"
#include "llvm/CodeGen/MachineOperand.h"
#include "llvm/Support/Compiler.h"

namespace llvm {
    class MCContext;
    class MCInst;
    class MCOperand;
    class MachineInstr;
    class MachineFunction;
    class Mangler;
    class Cpu0AsmPrinter;

    /// Cpu0MCInstLower - This class is used to lower an MachineInstr into an
    // MCInst.
    class LLVM_LIBRARY_VISIBILITY Cpu0MCInstLower {
        typedef MachineOperand::MachineOperandType MachineOperandType;
        MCContext *Ctx;
        Mangler *Mang;
        Cpu0AsmPrinter &AsmPrinter;
    public:
        Cpu0MCInstLower(Cpu0AsmPrinter &asmprinter);
        void Initialize(Mangler *mang, MCContext* C);
        void Lower(const MachineInstr *MI, MCInst &OutMI) const;
    private:
        MCOperand LowerSymbolOperand(const MachineOperand &MO,
                                     MachineOperandType MOTy, unsigned Offset) const;
        MCOperand LowerOperand(const MachineOperand& MO, unsigned offset = 0) const;
    };
}

#endif
```

LLVMBackendTutorialExampleCode/Chapter3_2/Cpu0MCInstLower.cpp

```
===== Cpu0MCInstLower.cpp - Convert Cpu0 MachineInstr to MCInst =====//
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
=====//
```

```
//  
// This file contains code to lower Cpu0 MachineInstrs to their corresponding  
// MCInst records.  
//  
//=====-----=====//  
  
#include "Cpu0MCInstLower.h"  
#include "Cpu0AsmPrinter.h"  
#include "Cpu0InstrInfo.h"  
#include "MCTargetDesc/Cpu0BaseInfo.h"  
#include "llvm/CodeGen/MachineFunction.h"  
#include "llvm/CodeGen/MachineInstr.h"  
#include "llvm/CodeGen/MachineOperand.h"  
#include "llvm/MC/MCContext.h"  
#include "llvm/MC/MCE Expr.h"  
#include "llvm/MC/MCInst.h"  
#include "llvm/Target/Mangler.h"  
  
using namespace llvm;  
  
Cpu0MCInstLower::Cpu0MCInstLower(Cpu0AsmPrinter &asmprinter)  
: AsmPrinter(asmprinter) {}  
  
void Cpu0MCInstLower::Initialize(Mangler *M, MCContext* C) {  
    Mang = M;  
    Ctx = C;  
}  
  
MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,  
                                              MachineOperandType MOTy,  
                                              unsigned Offset) const {  
    MCSymbolRefExpr::VariantKind Kind;  
    const MCSymbol *Symbol;  
  
    switch (MO.getTargetFlags()) {  
    default: llvm_unreachable("Invalid target flag!");  
    }  
  
    switch (MOTy) {  
    case MachineOperand::MO_GlobalAddress:  
        Symbol = Mang->getSymbol(MO.getGlobal());  
        break;  
  
    default:  
        llvm_unreachable("<unknown operand type>");  
    }  
  
    const MCSymbolRefExpr *MCSym = MCSymbolRefExpr::Create(Symbol, Kind, *Ctx);  
  
    if (!Offset)  
        return MCOperand::CreateExpr(MCSym);  
  
    // Assume offset is never negative.  
    assert(Offset > 0);  
  
    const MCConstantExpr *OffsetExpr = MCConstantExpr::Create(Offset, *Ctx);  
    const MCBinaryExpr *AddExpr = MCBinaryExpr::CreateAdd(MCSym, OffsetExpr, *Ctx);  
    return MCOperand::CreateExpr(AddExpr);
```

```

}

MCOperand Cpu0MCInstLower::LowerOperand(const MachineOperand& MO,
                                         unsigned offset) const {
    MachineOperandType MOTy = MO.getType();

    switch (MOTy) {
        default: llvm_unreachable("unknown operand type");
        case MachineOperand::MO_Register:
            // Ignore all implicit register operands.
            if (MO.isImplicit()) break;
            return MCOperand::CreateReg(MO.getReg());
        case MachineOperand::MO_Immediate:
            return MCOperand::CreateImm(MO.getImm() + offset);
        case MachineOperand::MO_RegisterMask:
            break;
    }

    return MCOperand();
}

void Cpu0MCInstLower::Lower(const MachineInstr *MI, MCInst &OutMI) const {
    OutMI.setOpcode(MI->getOpcode());

    for (unsigned i = 0, e = MI->getNumOperands(); i != e; ++i) {
        const MachineOperand &MO = MI->getOperand(i);
        MCOperand MCOp = LowerOperand(MO);

        if (MCOp.isValid())
            OutMI.addOperand(MCOp);
    }
}

```

LLVMBackendTutorialExampleCode/Chapter3_2/MCTargetDesc/Cpu0BaseInfo.h

```

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

//  

//          The LLVM Compiler Infrastructure  

//  

// This file is distributed under the University of Illinois Open Source  

// License. See LICENSE.TXT for details.  

//  

//=====//  

//  

// This file contains small standalone helper functions and enum definitions for  

// the Cpu0 target useful for the compiler back-end and the MC libraries.  

//  

//=====//  

#ifndef CPU0BASEINFO_H  

#define CPU0BASEINFO_H  

#include "Cpu0FixupKinds.h"  

#include "Cpu0MCTargetDesc.h"  

#include "llvm/MC/MCE Expr.h"  

#include "llvm/Support/DataTypes.h"  

#include "llvm/Support/ErrorHandling.h"

```

```
namespace llvm {

/// Cpu0II - This namespace holds all of the target specific flags that
/// instruction info tracks.
///
namespace Cpu0II {
    /// Target Operand Flag enum.
    enum {
        //=====
        // Instruction encodings. These are the standard/most common forms for
        // Cpu0 instructions.
        //

        // Pseudo - This represents an instruction that is a pseudo instruction
        // or one that has not been implemented yet. It is illegal to code generate
        // it, but tolerated for intermediate implementation stages.
        Pseudo = 0,

        /// FrmR - This form is for instructions of the format R.
        FrmR = 1,
        /// FrmI - This form is for instructions of the format I.
        FrmI = 2,
        /// FrmJ - This form is for instructions of the format J.
        FrmJ = 3,
        /// FrmOther - This form is for instructions that have no specific format.
        FrmOther = 4,

        FormMask = 15
    };
}

/// getCpu0RegisterNumbering - Given the enum value for some register,
/// return the number that it corresponds to.
inline static unsigned getCpu0RegisterNumbering(unsigned RegEnum)
{
    switch (RegEnum) {
    case Cpu0::ZERO:
        return 0;
    case Cpu0::AT:
        return 1;
    case Cpu0::V0:
        return 2;
    case Cpu0::V1:
        return 3;
    case Cpu0::A0:
        return 4;
    case Cpu0::A1:
        return 5;
    case Cpu0::T9:
        return 6;
    case Cpu0::S0:
        return 7;
    case Cpu0::S1:
        return 8;
    case Cpu0::S2:
        return 9;
    case Cpu0::GP:
        return 10;
    }
}
```

```

    case Cpu0::FP:
        return 11;
    case Cpu0::SW:
        return 12;
    case Cpu0::SP:
        return 13;
    case Cpu0::LR:
        return 14;
    case Cpu0::PC:
        return 15;
    default: llvm_unreachable("Unknown register number!");
}
}

#endif

```

LLVMBackendTutorialExampleCode/Chapter3_2/MCTargetDesc/Cpu0FixupKinds.h

```

//===== Cpu0FixupKinds.h - Cpu0 Specific Fixup Entries -----*- C++ -*==//  

//  

//          The LLVM Compiler Infrastructure  

//  

// This file is distributed under the University of Illinois Open Source  

// License. See LICENSE.TXT for details.  

//  

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

#ifndef LLVM_CPU0_CPU0FIXUPKINDS_H
#define LLVM_CPU0_CPU0FIXUPKINDS_H  

#include "llvm/MC/MCFixup.h"  

namespace llvm {
namespace Cpu0 {
    // Although most of the current fixup types reflect a unique relocation
    // one can have multiple fixup types for a given relocation and thus need
    // to be uniquely named.
    //
    // This table *must* be in the same order of
    // MCFixupKindInfo Infos[Cpu0::NumTargetFixupKinds]
    // in Cpu0AsmBackend.cpp.
    //
    enum Fixups {
        // Branch fixups resulting in R_CPU0_16.
        fixup_Cpu0_16 = FirstTargetFixupKind,
  

        // Marker
        LastTargetFixupKind,
        NumTargetFixupKinds = LastTargetFixupKind - FirstTargetFixupKind
    };
} // namespace Cpu0
} // namespace llvm
  

#endif // LLVM_CPU0_CPU0FIXUPKINDS_H

```

LLVMBackendTutorialExampleCode/Chapter3_2/MCTargetDesc/Cpu0MCAsmInfo.h

```
===== Cpu0MCAsmInfo.h - Cpu0 Asm Info -----*-- C++ -*=====//
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//-----=====
//
// This file contains the declaration of the Cpu0MCAsmInfo class.
//
//-----=====
```

```
#ifndef CPU0TARGETASMINFO_H
#define CPU0TARGETASMINFO_H

#include "llvm/MC/MCAsmInfo.h"

namespace llvm {
    class StringRef;
    class Target;

    class Cpu0MCAsmInfo : public MCAsmInfo {
        virtual void anchor();
    public:
        explicit Cpu0MCAsmInfo(const Target &T, StringRef TT);
    };

} // namespace llvm

#endif
```

LLVMBackendTutorialExampleCode/Chapter3_2/MCTargetDesc/Cpu0MCAsmInfo.cpp

```
===== Cpu0MCAsmInfo.cpp - Cpu0 Asm Properties -----=====
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//-----=====
//
// This file contains the declarations of the Cpu0MCAsmInfo properties.
//
//-----=====

#include "Cpu0MCAsmInfo.h"
#include "llvm/ADT/Triple.h"

using namespace llvm;
```

```

void Cpu0MCAsmInfo::anchor() { }

Cpu0MCAsmInfo::Cpu0MCAsmInfo(const Target &T, StringRef TT) {
    Triple TheTriple(TT);
    if ((TheTriple.getArch() == Triple::cpu0))
        IsLittleEndian = false;

    AlignmentIsInBytes = false;
    Data16bitsDirective = "\t.2byte\t";
    Data32bitsDirective = "\t.4byte\t";
    Data64bitsDirective = "\t.8byte\t";
    PrivateGlobalPrefix = "$";
    CommentString = "#";
    ZeroDirective = "\t.space\t";
    GPRel32Directive = "\t.gpword\t";
    GPRel64Directive = "\t.gpdword\t";
    WeakRefDirective = "\t.weak\t";

    SupportsDebugInformation = true;
    ExceptionsType = ExceptionHandling::DwarfCFI;
    HasLEB128 = true;
    DwarfRegNumForCFI = true;
}

```

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

LLVMBackendTutorialExampleCode/MCTargetDesc/Cpu0MCTargetDesc.cpp

```

static std::string ParseCpu0Triple(StringRef TT, StringRef CPU) {
    std::string Cpu0ArchFeature;
    size_t DashPosition = 0;
    StringRef TheTriple;

    // Let's see if there is a dash, like cpu0-unknown-linux.
    DashPosition = TT.find('-');

    if (DashPosition ==StringRef::npos) {
        // No dash, we check the string size.
        TheTriple = TT.substr(0);
    } else {
        // We are only interested in substring before dash.
        TheTriple = TT.substr(0, DashPosition);
    }

    if (TheTriple == "cpu0" || TheTriple == "cpu0el") {
        if (CPU.empty() || CPU == "cpu032") {
            Cpu0ArchFeature = "+cpu032";
        }
    }
    return Cpu0ArchFeature;
}

static MCInstrInfo *createCpu0MCInstrInfo() {
    MCInstrInfo *X = new MCInstrInfo();
    InitCpu0MCInstrInfo(X); // defined in Cpu0GenInstrInfo.inc
    return X;
}

```

```

}

static MCRegisterInfo *createCpu0MCRegisterInfo(StringRef TT) {
    MCRegisterInfo *X = new MCRegisterInfo();
    InitCpu0MCRegisterInfo(X, Cpu0::LR); // defined in Cpu0GenRegisterInfo.inc
    return X;
}

static MCSubtargetInfo *createCpu0MCSubtargetInfo(StringRef TT, StringRef CPU,
                                                 StringRef FS) {
    std::string ArchFS = ParseCpu0Triple(TT, CPU);
    if (!FS.empty()) {
        if (!ArchFS.empty())
            ArchFS = ArchFS + "," + FS.str();
        else
            ArchFS = FS;
    }
    MCSubtargetInfo *X = new MCSubtargetInfo();
    InitCpu0MCSubtargetInfo(X, TT, CPU, ArchFS); // defined in Cpu0GenSubtargetInfo.inc
    return X;
}

static MCAsmInfo *createCpu0MCAsmInfo(const Target &T, StringRef TT) {
    MCAsmInfo *MAI = new Cpu0MCAsmInfo(T, TT);

    MachineLocation Dst(MachineLocation::VirtualFP);
    MachineLocation Src(Cpu0::SP, 0);
    MAI->addInitialFrameState(0, Dst, Src);

    return MAI;
}

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

static MCInstPrinter *createCpu0MCInstPrinter(const Target &T,
                                                unsigned SyntaxVariant,
                                                const MCAsmInfo &MAI,
                                                const MCInstrInfo &MII,
                                                const MCRegisterInfo &MRI,
                                                const MCSubtargetInfo &STI) {
    return new Cpu0InstPrinter(MAI, MII, MRI);
}

extern "C" void LLVMInitializeCpu0TargetMC() {
    // Register the MC asm info.
    RegisterMCAsmInfoFn X(TheCpu0Target, createCpu0MCAsmInfo);
    RegisterMCAsmInfoFn Y(TheCpu0elTarget, createCpu0MCAsmInfo);
}

```

```
// Register the MC codegen info.
TargetRegistry::RegisterMCCodeGenInfo(TheCpu0Target,
                                         createCpu0MCCodeGenInfo);
TargetRegistry::RegisterMCCodeGenInfo(TheCpu0elTarget,
                                         createCpu0MCCodeGenInfo);

// Register the MC instruction info.
TargetRegistry::RegisterMCInstrInfo(TheCpu0Target, createCpu0MCInstrInfo);
TargetRegistry::RegisterMCInstrInfo(TheCpu0elTarget, createCpu0MCInstrInfo);

// Register the MC register info.
TargetRegistry::RegisterMCRegInfo(TheCpu0Target, createCpu0MCRegisterInfo);
TargetRegistry::RegisterMCRegInfo(TheCpu0elTarget, createCpu0MCRegisterInfo);

// Register the MC subtarget info.
TargetRegistry::RegisterMCSubtargetInfo(TheCpu0Target,
                                         createCpu0MCSubtargetInfo);
TargetRegistry::RegisterMCSubtargetInfo(TheCpu0elTarget,
                                         createCpu0MCSubtargetInfo);

// Register the MCInstPrinter.
TargetRegistry::RegisterMCInstPrinter(TheCpu0Target,
                                         createCpu0MCInstPrinter);
TargetRegistry::RegisterMCInstPrinter(TheCpu0elTarget,
                                         createCpu0MCInstPrinter);
}

}
```

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

LLVMBackendTutorialExampleCode/Chapter3_2/Cpu0AsmPrinter.h

```
===== Cpu0AsmPrinter.h - Cpu0 LLVM Assembly Printer -----*-- C++ --=====//
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// Cpu0 Assembly printer class.
//
//=====

#ifndef CPU0ASMPRINTER_H
#define CPU0ASMPRINTER_H

#include "Cpu0MachineFunction.h"
#include "Cpu0MCInstLower.h"
#include "Cpu0Subtarget.h"
#include "llvm/CodeGen/AsmPrinter.h"
#include "llvm/Support/Compiler.h"
#include "llvm/Target/TargetMachine.h"

namespace llvm {
class MCStreamer;
```

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

```

class MachineInstr;
class MachineBasicBlock;
class Module;
class raw_ostream;

class LLVM_LIBRARY_VISIBILITY Cpu0AsmPrinter : public AsmPrinter {

    void EmitInstrWithMacroNoAT(const MachineInstr *MI);

public:

    const Cpu0Subtarget *Subtarget;
    const Cpu0FunctionInfo *Cpu0FI;
    Cpu0MCInstLower MCInstLowering;

    explicit Cpu0AsmPrinter(TargetMachine &TM, MCStreamer &Streamer)
        : AsmPrinter(TM, Streamer), MCInstLowering(*this) {
        Subtarget = &TM.getSubtarget<Cpu0Subtarget>();
    }

    virtual const char *getPassName() const {
        return "Cpu0 Assembly Printer";
    }

    virtual bool runOnMachineFunction(MachineFunction &MF);

    // - EmitInstruction() must exists or will have run time error.
    void EmitInstruction(const MachineInstr *MI);
    void printSavedRegsBitmask(raw_ostream &O);
    void printHex32(unsigned int Value, raw_ostream &O);
    void emitFrameDirective();
    const char *getCurrentABIString() const;
    virtual void EmitFunctionEntryLabel();
    virtual void EmitFunctionBodyStart();
    virtual void EmitFunctionBodyEnd();
    void EmitStartOfAsmFile(Module &M);
    virtual MachineLocation getDebugValueLocation(const MachineInstr *MI) const;
    void PrintDebugValueComment(const MachineInstr *MI, raw_ostream &OS);
};

#endif

```

LLVMBackendTutorialExampleCode/Chapter3_2/Cpu0AsmPrinter.cpp

```

===== Cpu0AsmPrinter.cpp - Cpu0 LLVM Assembly Printer =====
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
===== ===== =====
//
// This file contains a printer that converts from our internal representation
// of machine-dependent LLVM code to GAS-format CPU0 assembly language.
//

```

```

//=====

#define DEBUG_TYPE "cpu0-asm-printer"
#include "Cpu0AsmPrinter.h"
#include "Cpu0.h"
#include "Cpu0InstrInfo.h"
#include "InstPrinter/Cpu0InstPrinter.h"
#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "llvm/ADT/SmallString.h"
#include "llvm/ADT/StringExtras.h"
#include "llvm/ADT/Twine.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Instructions.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/MachineConstantPool.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineInstr.h"
#include "llvm/CodeGen/MachineMemOperand.h"
#include "llvm/MC/MCStreamer.h"
#include "llvm/MC/MCAsmInfo.h"
#include "llvm/MC/MCInst.h"
#include "llvm/MC/MCSymbol.h"
#include "llvm/Support/TargetRegistry.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Target/Mangler.h"
#include "llvm/Target/TargetLoweringObjectFile.h"
#include "llvm/Target/TargetOptions.h"

using namespace llvm;

bool Cpu0AsmPrinter::runOnMachineFunction(MachineFunction &MF) {
    Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
    AsmPrinter::runOnMachineFunction(MF);
    return true;
}

// EmitInstruction() must exists or will have run time error.
void Cpu0AsmPrinter::EmitInstruction(const MachineInstr *MI) {
    if (MI->isDebugValue()) {
        SmallString<128> Str;
        raw_svector_ostream OS(Str);

        PrintDebugValueComment(MI, OS);
        return;
    }

    MCInst TmpInst0;
    MCInstLowering.Lower(MI, TmpInst0);
    OutStreamer.EmitInstruction(TmpInst0);
}

//=====
// Cpu0 Asm Directives
// -- Frame directive "frame Stackpointer, Stacksize, RARegister"
// Describe the stack frame.
//

```

```

// -- Mask directives "(f)mask bitmask, offset"
// Tells the assembler which registers are saved and where.
// bitmask - contain a little endian bitset indicating which registers are
//           saved on function prologue (e.g. with a 0x80000000 mask, the
//           assembler knows the register 31 (RA) is saved at prologue.
// offset - the position before stack pointer subtraction indicating where
//           the first saved register on prologue is located. (e.g. with a
//
// Consider the following function prologue:
//
//     .frame $fp,48,$ra
//     .mask 0xc0000000,-8
//     addiu $sp, $sp, -48
//     st $ra, 40($sp)
//     st $fp, 36($sp)
//
//     With a 0xc0000000 mask, the assembler knows the register 31 (RA) and
//     30 (FP) are saved at prologue. As the save order on prologue is from
//     left to right, RA is saved first. A -8 offset means that after the
//     stack pointer subtraction, the first register in the mask (RA) will be
//     saved at address 48-8=40.
//
//=====//=====
// Mask directives
//=====
//     .frame      $sp,8,$lr
//->     .mask      0x00000000,0
//     .set       noreorder
//     .set       nomacro

// Create a bitmask with all callee saved registers for CPU or Floating Point
// registers. For CPU registers consider RA, GP and FP for saving if necessary.
void Cpu0AsmPrinter::printSavedRegsBitmask(raw_ostream &O) {
    // CPU and FPU Saved Registers Bitmasks
    unsigned CPUBitmask = 0;
    int CPUTopSavedRegOff;

    // Set the CPU and FPU Bitmasks
    const MachineFrameInfo *MFI = MF->getFrameInfo();
    const std::vector<CalleeSavedInfo> &CSI = MFI->getCalleeSavedInfo();
    // size of stack area to which FP callee-saved regs are saved.
    unsigned CPURegSize = Cpu0::CPURegsRegClass.getSize();
    unsigned i = 0, e = CSI.size();

    // Set CPU Bitmask.
    for (i != e; ++i) {
        unsigned Reg = CSI[i].getReg();
        unsigned RegNum = getCPURegisterNumbering(Reg);
        CPUBitmask |= (1 << RegNum);
    }

    CPUTopSavedRegOff = CPUBitmask ? -CPURegSize : 0;

    // Print CPUBitmask
    O << "\t.mask \t"; printHex32(CPUBitmask, 0);
    O << ',' << CPUTopSavedRegOff << '\n';
}

```

```

}

// Print a 32 bit hex number with all numbers.
void Cpu0AsmPrinter::printHex32(unsigned Value, raw_ostream &O) {
    O << "0x";
    for (int i = 7; i >= 0; i--)
        O.write_hex((Value & (0xF << (i*4))) >> (i*4));
}

//=====//
// Frame and Set directives
//=====//
//-> .frame    $sp,8,$lr
// .mask      0x00000000,0
// .set       noreorder
// .set       nomacro
/// Frame Directive
void Cpu0AsmPrinter::emitFrameDirective() {
    const TargetRegisterInfo &RI = *TM.getRegisterInfo();

    unsigned stackReg = RI.getFrameRegister(*MF);
    unsigned returnReg = RI.getRARegister();
    unsigned stackSize = MF->getFrameInfo()->getStackSize();

    if (OutStreamer.hasRawTextSupport())
        OutStreamer.EmitRawText("\t.frame\t" +
                              StringRef(Cpu0InstPrinter::getRegisterName(stackReg)).lower() +
                               "," + Twine(stackSize) + ",$" +
                              StringRef(Cpu0InstPrinter::getRegisterName(returnReg)).lower());
}

/// Emit Set directives.
const char *Cpu0AsmPrinter::getCurrentABIString() const {
    switch (Subtarget->getTargetABI()) {
        case Cpu0Subtarget::O32: return "abi32";
        default: llvm_unreachable("Unknown Cpu0 ABI");
    }
}

// .type      main,@function
//-> .ent      main                      # @main
// main:
void Cpu0AsmPrinter::EmitFunctionEntryLabel() {
    if (OutStreamer.hasRawTextSupport())
        OutStreamer.EmitRawText("\t.ent\t" + Twine(CurrentFnSym->getName()));
    OutStreamer.EmitLabel(CurrentFnSym);
}

// .frame    $sp,8,$pc
// .mask      0x00000000,0
//-> .set       noreorder
//-> .set       nomacro
/// EmitFunctionBodyStart - Targets can override this to emit stuff before
/// the first basic block in the function.
void Cpu0AsmPrinter::EmitFunctionBodyStart() {
    MCInstLowering.Initialize(Mang, &MF->getContext());
}

```

```

emitFrameDirective();

if (OutStreamer.hasRawTextSupport()) {
    SmallString<128> Str;
    raw_svector_ostream OS(Str);
    printSavedRegsBitmask(OS);
    OutStreamer.EmitRawText(OS.str());
    OutStreamer.EmitRawText(StringRef("\t.set\tnoreorder"));
    OutStreamer.EmitRawText(StringRef("\t.set\tmacro"));
    if (Cpu0FI->get.EmitNOAT())
        OutStreamer.EmitRawText(StringRef("\t.set\tat"));
}
}

//-->      .set      macro
//-->      .set      reorder
//-->      .end      main
/// EmitFunctionBodyEnd - Targets can override this to emit stuff after
/// the last basic block in the function.
void Cpu0AsmPrinter::EmitFunctionBodyEnd() {
    // There are instruction for this macros, but they must
    // always be at the function end, and we can't emit and
    // break with BB logic.
    if (OutStreamer.hasRawTextSupport()) {
        if (Cpu0FI->get.EmitNOAT())
            OutStreamer.EmitRawText(StringRef("\t.set\tat"));
        OutStreamer.EmitRawText(StringRef("\t.set\tmacro"));
        OutStreamer.EmitRawText(StringRef("\t.set\treorder"));
        OutStreamer.EmitRawText("\t.end\t" + Twine(CurrentFnSym->getName()));
    }
}

//      .section .mdebug.abi32
//      .previous
void Cpu0AsmPrinter::EmitStartOfAsmFile(Module &M) {
    // FIXME: Use SwitchSection.

    // Tell the assembler which ABI we are using
    if (OutStreamer.hasRawTextSupport())
        OutStreamer.EmitRawText("\t.section .mdebug." +
                               Twine(getCurrentABIString()));

    // return to previous section
    if (OutStreamer.hasRawTextSupport())
        OutStreamer.EmitRawText(StringRef("\t.previous"));
}

MachineLocation
Cpu0AsmPrinter::getDebugValueLocation(const MachineInstr *MI) const {
    // Handles frame addresses emitted in Cpu0InstrInfo::emitFrameIndexDebugValue.
    assert(MI->getNumOperands() == 4 && "Invalid no. of machine operands!");
    assert(MI->getOperand(0).isReg() && MI->getOperand(1).isImm() &&
           "Unexpected MachineOperand types");
    return MachineLocation(MI->getOperand(0).getReg(),
                           MI->getOperand(1).getImm());
}

void Cpu0AsmPrinter::PrintDebugValueComment(const MachineInstr *MI,

```

```
    raw_ostream &OS) {  
    // TODO: implement  
    OS << "PrintDebugValueComment()";  
}  
  
// Force static initialization.  
extern "C" void LLVMInitializeCpu0AsmPrinter() {  
    RegisterAsmPrinter<Cpu0AsmPrinter> X(TheCpu0Target);  
    RegisterAsmPrinter<Cpu0AsmPrinter> Y(TheCpu0elTarget);  
}
```

The dynamic register mechanism is a good idea, right.

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,

LLVMBackendTutorialExampleCode/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)  
...
```

LLVMBackendTutorialExampleCode/Chapter3_2/LLVMBuild.txt

```
// LLVMBuild.txt  
[common]  
subdirectories = InstPrinter MCTargetDesc TargetInfo  
  
[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`.

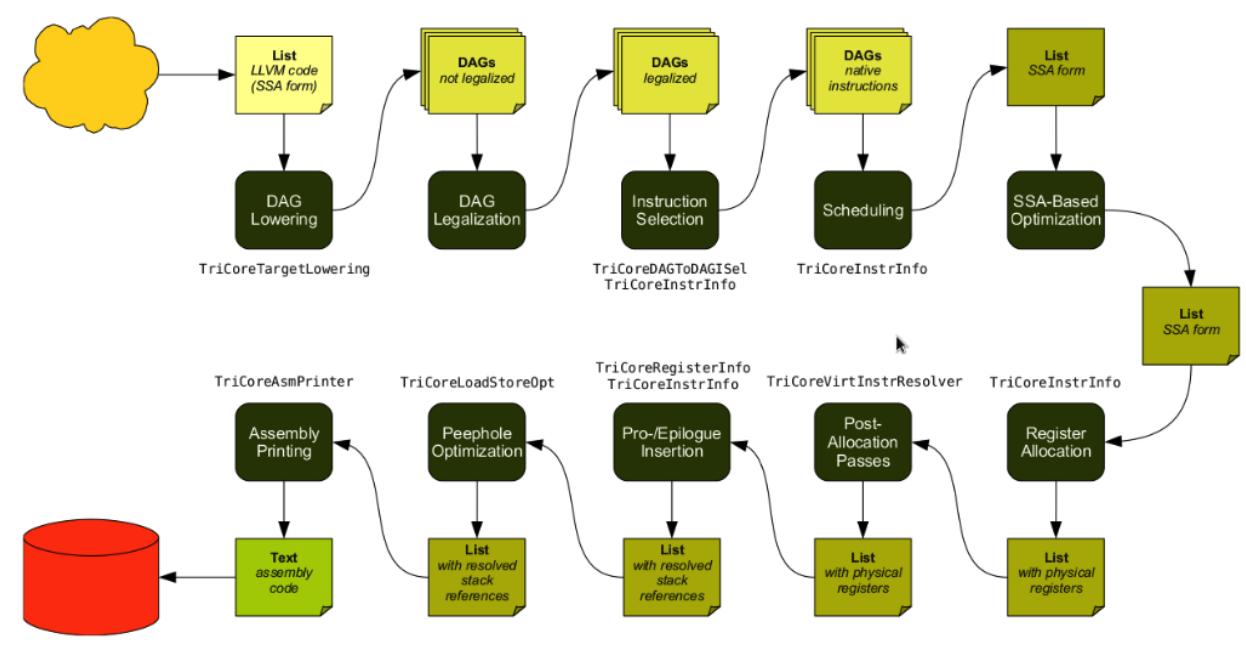


Figure 3.5: `tricore_llvm.pdf`: Code generation sequence. On the path from LLVM code to assembly code, numerous passes are run through and several data structures are used to represent the intermediate results.

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

```

store i32 0, i32* %a ; store i32 type of 0 to virtual register %a, %a is
; pointer type which point to i32 value
store i32 %b, i32* %c ; store %b contents to %c point to, %b is i32 type virtual
; register, %c is pointer type which point to i32 value.
%a1 = load i32* %a ; load the memory value where %a point to and assign the
; 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 ³ and "LLVM Language Reference Manual" from ⁴ 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.

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

⁴ <http://llvm.org/docs/LangRef.html>

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)
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 llv_m 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.

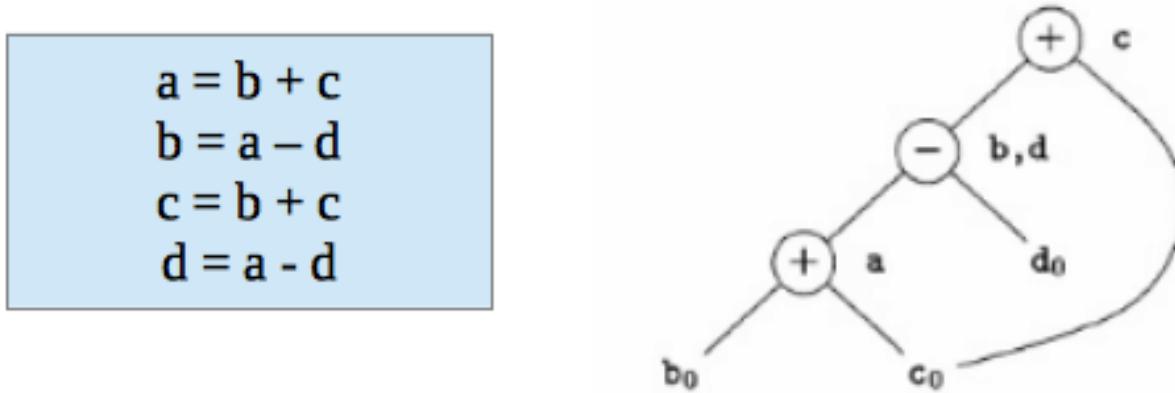


Figure 3.6: DAG example

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

```

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

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

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

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

Figure 3.7: IR and its 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.

Instruction Tree Patterns

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

Figure 3.8: Instruction DAG representation

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

LLVMBackendTutorialExampleCode/Chapter3_2/Cpu0InstrFormats.td

```

}

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

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

    let Opcode = op;

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

```

```
//=====//
```

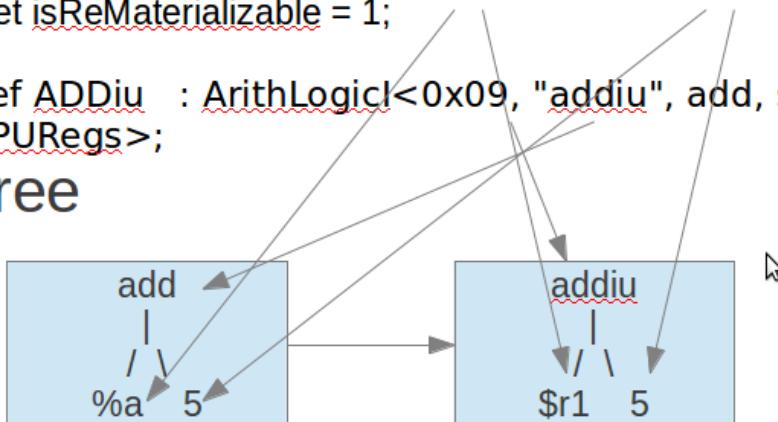
LLVMBackendTutorialExampleCode/Chapter3_2/Cpu0InstrInfo.td

```
// 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. For the 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.

LLVMBackendTutorialExampleCode/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 <0x2c, (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,

LLVMBackendTutorialExampleCode/Chapter3_3/CMakeLists.txt

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

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

LLVMBackendTutorialExampleCode/Chapter3_3/Cpu0TargetMachine.cpp

```

}
virtual bool addInstSelector();

bool Cpu0PassConfig::addInstSelector() {
  addPass(createCpu0ISelDag(getCpu0TargetMachine()));
  return false;
}
```

LLVMBackendTutorialExampleCode/Chapter3_3/Cpu0ISelDAGToDAG.cpp

```
===== Cpu0ISelDAGToDAG.cpp - A Dag to Dag Inst Selector for Cpu0 =====
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
//
// This file defines an instruction selector for the CPU0 target.
//
//=====

#define DEBUG_TYPE "cpu0-isel"
#include "Cpu0.h"
#include "Cpu0RegisterInfo.h"
#include "Cpu0Subtarget.h"
#include "Cpu0TargetMachine.h"
#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "llvm/IR/GlobalValue.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/Intrinsics.h"
#include "llvm/Support/CFG.h"
#include "llvm/IR/Type.h"
#include "llvm/CodeGen/MachineConstantPool.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineFrameInfo.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/CodeGen/SelectionDAGISel.h"
#include "llvm/CodeGen/SelectionDAGNodes.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

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

//=====
// Cpu0DAGToDAGISel - CPU0 specific code to select CPU0 machine
// instructions for SelectionDAG operations.
//=====
namespace {

class Cpu0DAGToDAGISel : public SelectionDAGISel {

    /// TM - Keep a reference to Cpu0TargetMachine.
    Cpu0TargetMachine &TM;

    /// Subtarget - Keep a pointer to the Cpu0Subtarget around so that we can
    /// make the right decision when generating code for different targets.
    const Cpu0Subtarget &Subtarget;
```

```

public:
  explicit Cpu0DAGToDAGISel(Cpu0TargetMachine &tm) :
    SelectionDAGISel(tm),
    TM(tm), Subtarget(tm.getSubtarget<Cpu0Subtarget>()) {}

  // Pass Name
  virtual const char *getPassName() const {
    return "CPU0 DAG->DAG Pattern Instruction Selection";
  }

  virtual bool runOnMachineFunction(MachineFunction &MF);

private:
  // Include the pieces autogenerated from the target description.
  #include "Cpu0GenDAGISel.inc"

  /// getTargetMachine - Return a reference to the TargetMachine, casted
  /// to the target-specific type.
  const Cpu0TargetMachine &getTargetMachine() {
    return static_cast<const Cpu0TargetMachine &>(TM);
  }

  /// getInstrInfo - Return a reference to the TargetInstrInfo, casted
  /// to the target-specific type.
  const Cpu0InstrInfo *getInstrInfo() {
    return getTargetMachine().getInstrInfo();
  }

  SDNode *getGlobalBaseReg();

  SDNode *Select(SDNode *N);
  // Complex Pattern.
  bool SelectAddr(SDNode *Parent, SDValue N, SDValue &Base, SDValue &Offset);
  // getImm - Return a target constant with the specified value.
  inline SDValue getImm(const SDNode *Node, unsigned Imm) {
    return CurDAG->getTargetConstant(Imm, Node->getValueType(0));
  }
};

}

bool Cpu0DAGToDAGISel::runOnMachineFunction(MachineFunction &MF) {
  bool Ret = SelectionDAGISel::runOnMachineFunction(MF);

  return Ret;
}

/// ComplexPattern used on Cpu0InstrInfo
/// Used on Cpu0 Load/Store instructions
bool Cpu0DAGToDAGISel::
SelectAddr(SDNode *Parent, SDValue Addr, SDValue &Base, SDValue &Offset) {
  EVT ValTy = Addr.getValueType();

  // If Parent is an unaligned f32 load or store, select a (base + index)
  // floating point load/store instruction (luxcl or luxcl).
  const LSBaseSDNode* LS = 0;

  if (Parent && (LS = dyn_cast<LSBaseSDNode>(Parent))) {
    EVT VT = LS->getMemoryVT();

```

```

if (VT.getSizeInBits() / 8 > LS->getAlignment()) {
    assert(TLI.allowsUnalignedMemoryAccesses(VT) &&
        "Unaligned loads/stores not supported for this type.");
    if (VT == MVT::f32)
        return false;
}
}

// if Address is FI, get the TargetFrameIndex.
if (FrameIndexSDNode *FIN = dyn_cast<FrameIndexSDNode>(Addr)) {
    Base = CurDAG->getTargetFrameIndex(FIN->getIndex(), ValTy);
    Offset = CurDAG->getTargetConstant(0, ValTy);
    return true;
}

Base = Addr;
Offset = CurDAG->getTargetConstant(0, ValTy);
return true;
}

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

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

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

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

    switch(Opcode) {
    default: break;

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

        if (Size == 32)
            break;
    }
}

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

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

```

```

    DEBUG(ResNode->dump(CurDAG));
    DEBUG(errs() << "\n");
    return ResNode;
}

/// createCpu0ISelDag - This pass converts a legalized DAG into a
/// CPU0-specific DAG, ready for instruction scheduling.
FunctionPass *llvm::createCpu0ISelDag(Cpu0TargetMachine &TM) {
    return new Cpu0DAGToDAGISel(TM);
}

```

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

LLVMBackendTutorialExampleCode/Chapter3_3/Cpu0InstrInfo.h

```

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

```

LLVMBackendTutorialExampleCode/Chapter3_3/Cpu0InstrInfo.cpp

```

#include "llvm/CodeGen/MachineInstrBuilder.h"
}

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

```

Build Chapter3_3, 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
...
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'
...

```

3.7 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.

We will explain the Prologue and Epilogue further by example code. So for the following llvm IR code, Cpu0 backend will emit the corresponding machine instructions as follows,

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

.section .mdebug.abi32
.previous
.file "ch3.bc"
.text
.globl main
.align 2
.type main,@function
.ent main          # @main
main:
.cfi_startproc
```

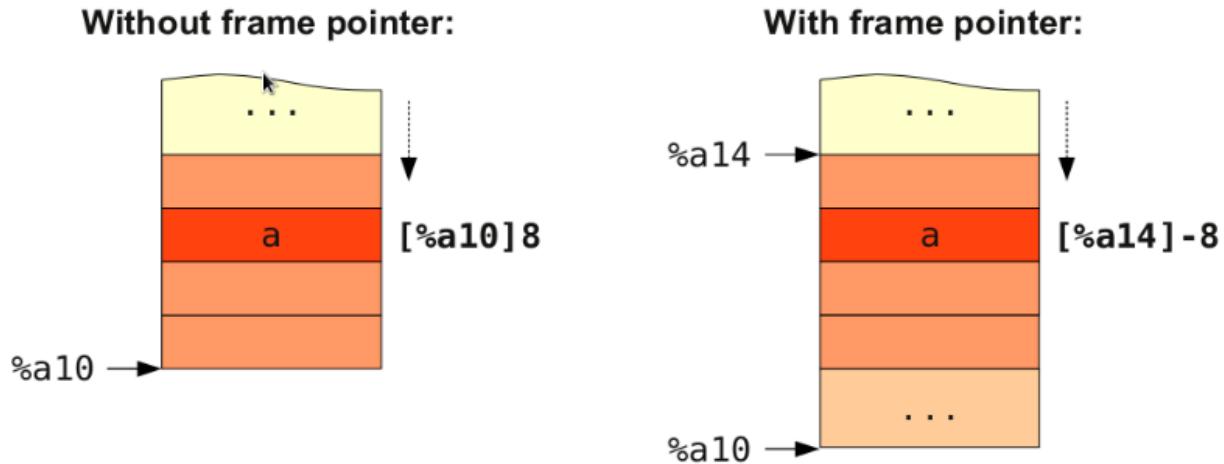


Figure 3.10: Addressing of a variable a located on the stack. If the stack frame has a variable size, slot must be addressed relative to the frame pointer

```

.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`”, 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,

LLVMBackendTutorialExampleCode/Chapter3_4/Cpu0FrameLowering.h

```

void emitPrologue(MachineFunction &MF) const;
void emitEpilogue(MachineFunction &MF, MachineBasicBlock &MBB) const;

```

LLVMBackendTutorialExampleCode/Chapter3_4/Cpu0FrameLowering.cpp

```

static void expandLargeImm(unsigned Reg, int64_t Imm,
                           const Cpu0InstrInfo &TII, MachineBasicBlock &MBB,
                           MachineBasicBlock::iterator II, DebugLoc DL) {
    unsigned LUI = Cpu0::LUI;
    unsigned ADDu = Cpu0::ADDu;
    unsigned ZEROReg = Cpu0::ZERO;
    unsigned ATReg = Cpu0::AT;
    Cpu0AnalyzeImmediate AnalyzeImm;
    const Cpu0AnalyzeImmediate::InstSeq &Seq =
        AnalyzeImm.Analyze(Imm, 32, false /* LastInstrIsADDiu */);
    Cpu0AnalyzeImmediate::InstSeq::const_iterator Inst = Seq.begin();

    // The first instruction can be a LUI, which is different from other
    // instructions (ADDiu, ORI and SLL) in that it does not have a register
    // operand.
    if (Inst->Opc == LUI)
        BuildMI(MBB, II, DL, TII.get(LUI), ATReg)
            .addImm(SignExtend64<16>(Inst->ImmOpnd));
    else
        BuildMI(MBB, II, DL, TII.get(Inst->Opc), ATReg).addReg(ZEROReg)
            .addImm(SignExtend64<16>(Inst->ImmOpnd));

    // Build the remaining instructions in Seq.
    for (++Inst; Inst != Seq.end(); ++Inst)
        BuildMI(MBB, II, DL, TII.get(Inst->Opc), ATReg).addReg(ATReg)
            .addImm(SignExtend64<16>(Inst->ImmOpnd));

    BuildMI(MBB, II, DL, TII.get(ADDu), Reg).addReg(Reg).addReg(ATReg);
}

```

```

void Cpu0FrameLowering::emitPrologue(MachineFunction &MF) const {
    MachineBasicBlock &MBB = MF.front();
    MachineFrameInfo *MFI = MF.getFrameInfo();
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
    const Cpu0InstrInfo &TII =
        *static_cast<const Cpu0InstrInfo*>(MF.getTarget().getInstrInfo());
    MachineBasicBlock::iterator MBBI = MBB.begin();
    DebugLoc dl = MBBI != MBB.end() ? MBBI->getDebugLoc() : DebugLoc();
    unsigned SP = Cpu0::SP;
    unsigned ADDiu = Cpu0::ADDiu;
    // First, compute final stack size.
    unsigned StackAlign = getStackAlignment();
    unsigned LocalVarAreaOffset = Cpu0FI->getMaxCallFrameSize();
    uint64_t StackSize = RoundUpToAlignment(LocalVarAreaOffset, StackAlign) +
        RoundUpToAlignment(MFI->getStackSize(), StackAlign);

    // Update stack size
    MFI->setStackSize(StackSize);

    // No need to allocate space on the stack.
    if (StackSize == 0 && !MFI->adjustsStack()) return;

    MachineModuleInfo &MMI = MF.getMMI();
    std::vector<MachineMove> &Moves = MMI.getFrameMoves();
    MachineLocation DstML, SrcML;

    // Adjust stack.
    if (isInt<16>(-StackSize)) // addiu sp, sp, (-stacksize)
        BuildMI(MBB, MBBI, dl, TII.get(ADDiu), SP).addReg(SP).addImm(-StackSize);
    else { // Expand immediate that doesn't fit in 16-bit.
        Cpu0FI->setEmitNOAT();
        expandLargeImm(SP, -StackSize, TII, MBB, MBBI, dl);
    }

    // emit ".cfi_def_cfa_offset StackSize"
    MCSymbol *AdjustSPLabel = MMI.getContext().CreateTempSymbol();
    BuildMI(MBB, MBBI, dl,
            TII.get(TargetOpcode::PROLOG_LABEL)).addSym(AdjustSPLabel);
    DstML = MachineLocation(MachineLocation::VirtualFP);
    SrcML = MachineLocation(MachineLocation::VirtualFP, -StackSize);
    Moves.push_back(MachineMove(AdjustSPLabel, DstML, SrcML));

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

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

        // Iterate over list of callee-saved registers and emit .cfi_offset
        // directives.
        MCSymbol *CSLabel = MMI.getContext().CreateTempSymbol();
        BuildMI(MBB, MBBI, dl,
                TII.get(TargetOpcode::PROLOG_LABEL)).addSym(CSLabel);

        for (std::vector<CalleeSavedInfo>::const_iterator I = CSI.begin(),
              E = CSI.end(); I != E; ++I) {

```

```

int64_t Offset = MFI->getObjectOffset(I->getFrameIdx());
unsigned Reg = I->getReg();
{
    // Reg is either in CPURegs or FGR32.
    DstML = MachineLocation(MachineLocation::VirtualFP, Offset);
    SrcML = MachineLocation(Reg);
    Moves.push_back(MachineMove(CSLabel, DstML, SrcML));
}
}
}

void Cpu0FrameLowering::emitEpilogue(MachineFunction &MF,
                                      MachineBasicBlock &MBB) const {
    MachineBasicBlock::iterator MBBI = MBB.getLastNonDebugInstr();
    MachineFrameInfo *MFI = MF.getFrameInfo();
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
    const Cpu0InstrInfo &TII =
        *static_cast<const Cpu0InstrInfo*>(MF.getTarget().getInstrInfo());
    DebugLoc dl = MBBI->getDebugLoc();
    unsigned SP = Cpu0::SP;
    unsigned ADDiu = Cpu0::ADDiu;

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

    if (!StackSize)
        return;

    // Adjust stack.
    if (isInt<16>(StackSize)) // addiu sp, sp, (stacksize)
        BuildMI(MBB, MBBI, dl, TII.get(ADDiu), SP).addReg(SP).addImm(StackSize);
    else { // Expand immediate that doesn't fit in 16-bit.
        Cpu0FI->setEmitNOAT();
        expandLargeImm(SP, StackSize, TII, MBB, MBBI, dl);
    }
}

// This method is called immediately before PrologEpilogInserter scans the
// physical registers used to determine what callee saved registers should be
// spilled. This method is optional.
// Without this will have following errors,
// Target didn't implement TargetInstrInfo::storeRegToStackSlot!
// UNREACHABLE executed at /usr/local/llvm/3.1.test/cpu0/1/src/include/llvm/
// Target/TargetInstrInfo.h:390!
// Stack dump:
// 0.      Program arguments: /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/
// bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch0.bc -o
// ch0.cpu0.s
// 1.      Running pass 'Function Pass Manager' on module 'ch0.bc'.
// 2.      Running pass 'Prologue/Epilogue Insertion & Frame Finalization' on
//         function '@main'
// Aborted (core dumped)

// Must exist
//      addiu      $sp, $sp, 8
//->      ret       $lr
//      .set      macro

```

```

//      .set      reorder
//      .end      main
void Cpu0FrameLowering::
processFunctionBeforeCalleeSavedScan(MachineFunction &MF,
                                      RegScavenger *RS) const {
    MachineRegisterInfo& MRI = MF.getRegInfo();

    // FIXME: remove this code if register allocator can correctly mark
    // $fp and $ra used or unused.

    // The register allocator might determine $ra is used after seeing
    // instruction "jr $ra", but we do not want PrologEpilogInserter to insert
    // instructions to save/restore $ra unless there is a function call.
    // To correct this, $ra is explicitly marked unused if there is no
    // function call.
    if (MF.getFrameInfo()->hasCalls())
        MRI.setPhysRegUsed(Cpu0::LR);
    else {
        MRI.setPhysRegUnused(Cpu0::LR);
    }
}

```

LLVMBackendTutorialExampleCode/Chapter3_4/Cpu0AnalyzeImmediate.h

```

//===== Cpu0AnalyzeImmediate.h - Analyze Immediates -----*-- C++ --=====//
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====-----=====//
#ifndef CPU0_ANALYZE_IMMEDIATE_H
#define CPU0_ANALYZE_IMMEDIATE_H

#include "llvm/ADT/SmallVector.h"
#include "llvm/Support/DataTypes.h"

namespace llvm {

    class Cpu0AnalyzeImmediate {
    public:
        struct Inst {
            unsigned Opc, ImmOpnd;
            Inst(unsigned Opc, unsigned ImmOpnd);
        };
        typedef SmallVector<Inst, 7 > InstSeq;

        /// Analyze - Get an instruction sequence to load immediate Imm. The last
        /// instruction in the sequence must be an ADDiu if LastInstrIsADDiu is
        /// true;
        const InstSeq &Analyze(uint64_t Imm, unsigned Size, bool LastInstrIsADDiu);
    private:
        typedef SmallVector<InstSeq, 5> InstSeqLs;

        /// AddInstr - Add I to all instruction sequences in SeqLs.
    }
}

```

```

void AddInstr(InstSeqLs &SeqLs, const Inst &I);

/// GetInstSeqLsADDiu - Get instrucion sequences which end with an ADDiu to
/// load immediate Imm
void GetInstSeqLsADDiu(uint64_t Imm, unsigned RemSize, InstSeqLs &SeqLs);

/// GetInstSeqLsORi - Get instrucion sequences which end with an ORi to
/// load immediate Imm
void GetInstSeqLsORi(uint64_t Imm, unsigned RemSize, InstSeqLs &SeqLs);

/// GetInstSeqLsSHL - Get instrucion sequences which end with a SHL to
/// load immediate Imm
void GetInstSeqLsSHL(uint64_t Imm, unsigned RemSize, InstSeqLs &SeqLs);

/// GetInstSeqLs - Get instrucion sequences to load immediate Imm.
void GetInstSeqLs(uint64_t Imm, unsigned RemSize, InstSeqLs &SeqLs);

/// ReplaceADDiuSHLWithLUi - Replace an ADDiu & SHL pair with a LUi.
void ReplaceADDiuSHLWithLUi(InstSeq &Seq);

/// GetShortestSeq - Find the shortest instruction sequence in SeqLs and
/// return it in Insts.
void GetShortestSeq(InstSeqLs &SeqLs, InstSeq &Insts);

unsigned Size;
unsigned ADDiu, ORi, SHL, LUi;
InstSeq Insts;
};

}

#endif

```

LLVMBackendTutorialExampleCode/Chapter3_4/Cpu0AnalyzeImmediate.cpp

```

===== Cpu0AnalyzeImmediate.cpp - Analyze Immediates =====
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====

#include "Cpu0AnalyzeImmediate.h"
#include "Cpu0.h"
#include "llvm/Support/MathExtras.h"

using namespace llvm;

Cpu0AnalyzeImmediate::Inst::Inst(unsigned O, unsigned I) : Opc(O), ImmOpnd(I) {}

// Add I to the instruction sequences.
void Cpu0AnalyzeImmediate::AddInstr(InstSeqLs &SeqLs, const Inst &I) {
    // Add an instruction seqeunce consisting of just I.
    if (SeqLs.empty()) {
        SeqLs.push_back(InstSeq(1, I));
        return;
    }
}

```

```

for (InstSeqLs::iterator Iter = SeqLs.begin(); Iter != SeqLs.end(); ++Iter)
    Iter->push_back(I);
}

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

void Cpu0AnalyzeImmediate::GetInstSeqLsORi(uint64_t Imm, unsigned RemSize,
                                             InstSeqLs &SeqLs) {
    GetInstSeqLs(Imm & 0xfffffffffffff0000ULL, RemSize, SeqLs);
    AddInstr(SeqLs, Inst(ORi, Imm & 0xffffULL));
}

void Cpu0AnalyzeImmediate::GetInstSeqLsSHL(uint64_t Imm, unsigned RemSize,
                                             InstSeqLs &SeqLs) {
    unsigned Shamt = CountTrailingZeros_64(Imm);
    GetInstSeqLs(Imm >> Shamt, RemSize - Shamt, SeqLs);
    AddInstr(SeqLs, Inst(SHL, Shamt));
}

void Cpu0AnalyzeImmediate::GetInstSeqLs(uint64_t Imm, unsigned RemSize,
                                         InstSeqLs &SeqLs) {
    uint64_t MaskedImm = Imm & (0xfffffffffffff000ULL >> (64 - Size));

    // Do nothing if Imm is 0.
    if (!MaskedImm)
        return;

    // A single ADDiu will do if RemSize <= 16.
    if (RemSize <= 16) {
        AddInstr(SeqLs, Inst(ADDiu, MaskedImm));
        return;
    }

    // Shift if the lower 16-bit is cleared.
    if (!(Imm & 0xffff)) {
        GetInstSeqLsSHL(Imm, RemSize, SeqLs);
        return;
    }

    GetInstSeqLsADDiu(Imm, RemSize, SeqLs);

    // If bit 15 is cleared, it doesn't make a difference whether the last
    // instruction is an ADDiu or ORi. In that case, do not call GetInstSeqLsORi.
    if (Imm & 0x8000) {
        InstSeqLs SeqLsORi;
        GetInstSeqLsORi(Imm, RemSize, SeqLsORi);
        SeqLs.insert(SeqLs.end(), SeqLsORi.begin(), SeqLsORi.end());
    }
}

// Replace a ADDiu & SHL pair with a LUi.
// e.g. the following two instructions
// ADDiu 0x0111
// SHL 18

```

```

// are replaced with
// LUI 0x444
void Cpu0AnalyzeImmediate::ReplaceADDiuSHLWithLUI(InstSeq &Seq) {
    // Check if the first two instructions are ADDiu and SHL and the shift amount
    // is at least 16.
    if ((Seq.size() < 2) || (Seq[0].Opc != ADDiu) ||
        (Seq[1].Opc != SHL) || (Seq[1].ImmOpnd < 16))
        return;

    // Sign-extend and shift operand of ADDiu and see if it still fits in 16-bit.
    int64_t Imm = SignExtend64<16>(Seq[0].ImmOpnd);
    int64_t ShiftedImm = (uint64_t)Imm << (Seq[1].ImmOpnd - 16);

    if (!isInt<16>(ShiftedImm))
        return;

    // Replace the first instruction and erase the second.
    Seq[0].Opc = LUI;
    Seq[0].ImmOpnd = (unsigned) (ShiftedImm & 0xffff);
    Seq.erase(Seq.begin() + 1);
}

void Cpu0AnalyzeImmediate::GetShortestSeq(InstSeqLs &SeqLs, InstSeq &Insts) {
    InstSeqLs::iterator ShortestSeq = SeqLs.end();
    // The length of an instruction sequence is at most 7.
    unsigned ShortestLength = 8;

    for (InstSeqLs::iterator S = SeqLs.begin(); S != SeqLs.end(); ++S) {
        ReplaceADDiuSHLWithLUI(*S);
        assert(S->size() <= 7);

        if (S->size() < ShortestLength) {
            ShortestSeq = S;
            ShortestLength = S->size();
        }
    }

    Insts.clear();
    Insts.append(ShortestSeq->begin(), ShortestSeq->end());
}

const Cpu0AnalyzeImmediate::InstSeq
&Cpu0AnalyzeImmediate::Analyze(uint64_t Imm, unsigned Size,
                                bool LastInstrIsADDiu) {
    this->Size = Size;

    ADDiu = Cpu0::ADDiu;
    ORi = Cpu0::ORi;
    SHL = Cpu0::SHL;
    LUI = Cpu0::LUI;

    InstSeqLs SeqLs;

    // Get the list of instruction sequences.
    if (LastInstrIsADDiu & !Imm)
        GetInstSeqLsADDiu(Imm, Size, SeqLs);
    else
        GetInstSeqLs(Imm, Size, SeqLs);
}

```

```

    // Set Insts to the shortest instruction sequence.
    GetShortestSeq(SeqLs, Insts);

    return Insts;
}

```

LLVMBackendTutorialExampleCode/Chapter3_4/Cpu0RegisterInfo.cpp

```

}

//-- If eliminateFrameIndex() is empty, it will hang on run.
// pure virtual method
// FrameIndex represent objects inside a abstract stack.
// We must replace FrameIndex with an stack/frame pointer
// direct reference.
void Cpu0RegisterInfo::
eliminateFrameIndex(MachineBasicBlock::iterator III, int SPAdj,
                     unsigned FIOperandNum, RegScavenger *RS) const {
    MachineInstr &MI = *III;
    MachineFunction &MF = *MI.getParent()->getParent();
    MachineFrameInfo *MFI = MF.getFrameInfo();

    unsigned i = 0;
    while (!MI.getOperand(i).isFI()) {
        ++i;
        assert(i < MI.getNumOperands() &&
               "Instr doesn't have FrameIndex operand!");
    }

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

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

    DEBUG(errs() << "FrameIndex : " << FrameIndex << "\n"
          << "spOffset : " << spOffset << "\n"
          << "stackSize : " << stackSize << "\n");

    const std::vector<CalleeSavedInfo> &CSI = MFI->getCalleeSavedInfo();
    int MinCSI = 0;
    int MaxCSI = -1;

    if (CSI.size()) {
        MinCSI = CSI[0].getFrameIdx();
        MaxCSI = CSI[CSI.size() - 1].getFrameIdx();
    }

    // The following stack frame objects are always referenced relative to $sp:
    // 1. Outgoing arguments.
    // 2. Pointer to dynamically allocated stack space.
    // 3. Locations for callee-saved registers.
    // Everything else is referenced relative to whatever register
    // getFrameRegister() returns.
    unsigned FrameReg;

```

```

FrameReg = 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.
int64_t Offset;
Offset = spOffset + (int64_t)stackSize;

Offset += MI.getOperand(i+1).getImm();

DEBUG(errs() << "Offset : " << Offset << "\n" << "-----\n");

// If MI is not a debug value, make sure Offset fits in the 16-bit immediate
// field.
if (!MI.isDebugValue() && !isInt<16>(Offset)) {
    assert("(!MI.isDebugValue() && !isInt<16>(Offset))");
}

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

// pure virtual method

```

LLVMBackendTutorialExampleCode/Chapter3_4/CMakeLists.txt

```

add_llvm_target(...

...
Cpu0AnalyzeImmediate.cpp
...
)

```

After add these Prologue and Epilogue functions, and build with Chapter3_4/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 3.1: 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.

LLVMBackendTutorialExampleCode/Chapter3_4/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 translate the very simple main() function with return 0 single instruction. The Cpu0AnalyzeImmediate.cpp defined as above and the Cpu0InstrInfo.td instructions add as below, takes care the 32 bits stack size adjustments.

LLVMBackendTutorialExampleCode/Chapter3_4/Cpu0InstrInfo.td

```
def shamt      : Operand<i32>;
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:$ra), (ins Imm:$imml6),
        !strconcat(instr_asm, "\t$ra, $imml6"), [], 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. You can skip these recursive code and think these code in last chapter 12. Since in Chapter 12 Optimization, it replace addiu and shl with lui single instruction, you have chance to think this thing in details. 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 3.2: 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 ~ 0xffff	• 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 3.3: Cpu0 stack adjustment instructions after replace addiu and shl with lui instruction

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

Assume sp = 0xa0008000 and stack size = 0x90008000, then (0xa0008000 - 0x90008000) => 0x10000000. Verify with the Cpu0 Prologue instructions as follows,

1. “lui \$1, 28671” => \$1 = 0x6fff0000.
2. “ori \$1, \$1, 32768” => \$1 = (0x6fff0000 + 0x00008000) => \$1 = 0x6fff8000.
3. “addu \$sp, \$sp, \$1” => \$sp = (0xa0008000 + 0x6fff8000) => \$sp = 0x10000000.

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

1. “lui \$1, 36865” => \$1 = 0x90010000.
2. “addiu \$1, \$1, -32768” => \$1 = (0x90010000 + 0xffff8000) => \$1 = 0x90008000.
3. “addu \$sp, \$sp, \$1” => \$sp = (0x10000000 + 0x90008000) => \$sp = 0xa0008000.

3.8 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 3.4: functions for llvm backend stage

Stage	Function
Before CPU0 DAG->DAG Pattern Instruction Selection	<ul style="list-style-type: none"> • Cpu0TargetLowering::LowerFormalArguments • Cpu0TargetLowering::LowerReturn
Instruction selection	<ul style="list-style-type: none"> • Cpu0DAGToDAGISel::Select
Prologue/Epilogue Insertion & Frame Finalization	<ul style="list-style-type: none"> • Cpu0FrameLowering.cpp • Cpu0RegisterInfo::eliminateFrameIndex()
Cpu0 Assembly Printer	<ul style="list-style-type: none"> • Cpu0AsmPrinter.cpp -> Cpu0MCInstLower.cpp • Cpu0InstPrinter.cpp

We add a pass in Instruction Section stage in section “Add Cpu0DAGToDAGISel class”. You can embed your code into other pass like that. Please check CodeGen/Passes.h for the information. Remember the pass is called according the function unit as the `llc -debug-pass=Structure` indicated.

We have finished a simple assembler for cpu0 which only support **ld, st, addiu, ori, lui, addu, shl** and **ret** 8 instructions.

We are satisfied with this result. But you may think “After so many codes we program, and just get these 8 instructions”. The point is we have created a frame work for cpu0 target machine (please look back the llvm back end structure class 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 LLVMBuild.txt. It can be counted by command `wc 'find dir -name *.cpp'` for files *.cpp, *.h, *.td, *.txt. LLVM front end tutorial have 700 lines of source code without comments 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, LOCAL POINTER 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 Ilvm IR nodes with Graphviz](#) will show you the DAG optimization steps and their corresponding llc 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

4.1.1 +, -, *, <<, and >>

Appending the following code to Cpu0InstrInfo.td and Cpu0Schedule.td in Chapter4_1/ to support operators +, -, *, <<, and >>.

[LLVMBackendTutorialExampleCode/Chapter4_1/Cpu0InstrInfo.td](#)

```
def shamt      : Operand<i32>;
...
// 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>;

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;
}
...
/// Arithmetic Instructions (3-Operand, R-Type)
...
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 ashtr 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>;
def SHL     : shift_rotate_imm32<0x1e, 0x00, "shl", shl>;
// 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", CPURegs>;
def SHLV    : shift_rotate_reg<0x21, 0x00, "shlv", shl, CPURegs>;
def SHRV    : shift_rotate_reg<0x22, 0x00, "shrv", srl, CPURegs>;

```

LLVMBackendTutorialExampleCode/Chapter4_1/Cpu0Schedule.td

```

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

```

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 ¹ 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: ashr

Example: `<result> = ashr i32 4, 1 ; yields {i32}:result = 2`

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

`<result> = ashr i32 1, 32 ; undefined`

The C operator **>>** for negative operand is dependent on implementation. Most compiler translate it into "shift with sign extension fill", for example, Mips **sra** is the instruction. Following is the Microsoft web site explanation,

Note: >>, Microsoft Specific

The result of a right shift of a signed negative quantity is implementation dependent. Although Microsoft C++ propagates the most-significant bit to fill vacated bit positions, there is no guarantee that other implementations will do likewise.

In addition to **ashr**, the other instruction "shift with zero filled" **lshr** in LLVM (Mips implement lshr with instruction **srl**) has the following meaning.

Note: lshr

Example: `<result> = lshr i8 -2, 1 ; yields {i8}:result = 0x7FFFFFFF`

In LLVM, IR node **sra** is defined for ashr IR instruction, 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 **>>** implementation.

Table 4.1: C operator **>>** implementation

Description	Shift with zero filled	Shift with signed extension filled
symbol in .bc	lshr	ashr
symbol in IR node	srl	sra
Mips instruction	srl	sra
Cpu0 instruction	shr	sra
signed example before $x >> 1$	0xfffffffffe i.e. -2	0xfffffffffe i.e. -2
signed example after $x >> 1$	0x7fffffff i.e. 2G-1	0xffffffff i.e. -1
unsigned example before $x >> 1$	0xfffffffffe i.e. 4G-2	0xfffffffffe i.e. 4G-2
unsigned example after $x >> 1$	0x7fffffff i.e. 2G-1	0xffffffff i.e. 4G-1

lshr: Logical SHift Right

ashr: Arithmetic SHift right

srl: Shift Right Logically

sra: Shift Right Arithmetically

shr: SHift Right

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

If we consider the $x \gg 1$ definition is $x = x/2$ for compiler implementation. 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 ².

The sub-section ““ashr‘ Instruction” and sub-section ““lshr‘ Instruction” of ³.

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,

LLVMBackendTutorialExampleCode/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);
}
```

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

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

```
// 7+3+10+20-10+1+3+32+4+4+0 = 74
}

118-165-78-12:InputFiles Jonathan$ clang -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
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:
.cfi_startproc
.frame $fp,64,$lr
.mask 0x00000800,-4
.set noreorder
.set nomacro
# BB#0:                                # %entry
addiu $sp, $sp, -64
$tmp3:
.cfi_def_cfa_offset 64
st $fp, 60($sp)           # 4-byte Folded Spill
$tmp4:
.cfi_offset 11, -4
addu $fp, $sp, $zero
$tmp5:
.cfi_def_cfa_register 11
addiu $2, $zero, 5
st $2, 56($fp)
addiu $2, $zero, 2

```

```
st    $2, 52($fp)
addiu $2, $zero, -5
st    $2, 48($fp)
ld    $2, 52($fp)
ld    $3, 56($fp)
addu $2, $3, $2
st    $2, 44($fp)
ld    $2, 52($fp)
ld    $3, 56($fp)
subu $2, $3, $2
st    $2, 40($fp)
ld    $2, 52($fp)
ld    $3, 56($fp)
mul   $2, $3, $2
st    $2, 36($fp)
ld    $2, 56($fp)
shl   $2, $2, 2
st    $2, 32($fp)
ld    $2, 48($fp)
shl   $2, $2, 1
st    $2, 16($fp)
ld    $2, 56($fp)
sra   $2, $2, 2
st    $2, 28($fp)
ld    $2, 48($fp)
shr   $2, $2, 30
st    $2, 12($fp)
addiu $2, $zero, 1
ld    $3, 56($fp)
shlv  $3, $2, $3
st    $3, 24($fp)
ld    $3, 52($fp)
shlv  $2, $2, $3
st    $2, 8($fp)
addiu $2, $zero, 128
ld    $3, 56($fp)
shrv  $2, $2, $3
st    $2, 20($fp)
ld    $2, 56($fp)
ld    $3, 52($fp)
srav  $2, $3, $2
st    $2, 4($fp)
ld    $3, 40($fp)
ld    $4, 44($fp)
addu $3, $4, $3
ld    $4, 36($fp)
addu $3, $3, $4
ld    $4, 32($fp)
addu $3, $3, $4
ld    $4, 16($fp)
addu $3, $3, $4
ld    $4, 28($fp)
addu $3, $3, $4
ld    $4, 12($fp)
addu $3, $3, $4
ld    $4, 24($fp)
addu $3, $3, $4
ld    $4, 8($fp)
```

```

addu $3, $3, $4
ld $4, 20($fp)
addu $3, $3, $4
addu $2, $3, $2
addu $sp, $fp, $zero
ld $fp, 60($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 64
ret $lr
.set macro
.set reorder
.end _Z9test_mathv
$tmp6:
.size _Z9test_mathv, ($tmp6)-_Z9test_mathv
.cfi_endproc

```

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 ⁴ 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,

```

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

```

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

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,

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

LLVM `srem` is the IR corresponding “`%`”, reference sub-section “`srem instruction`” of ³. Copy the reference as follows,

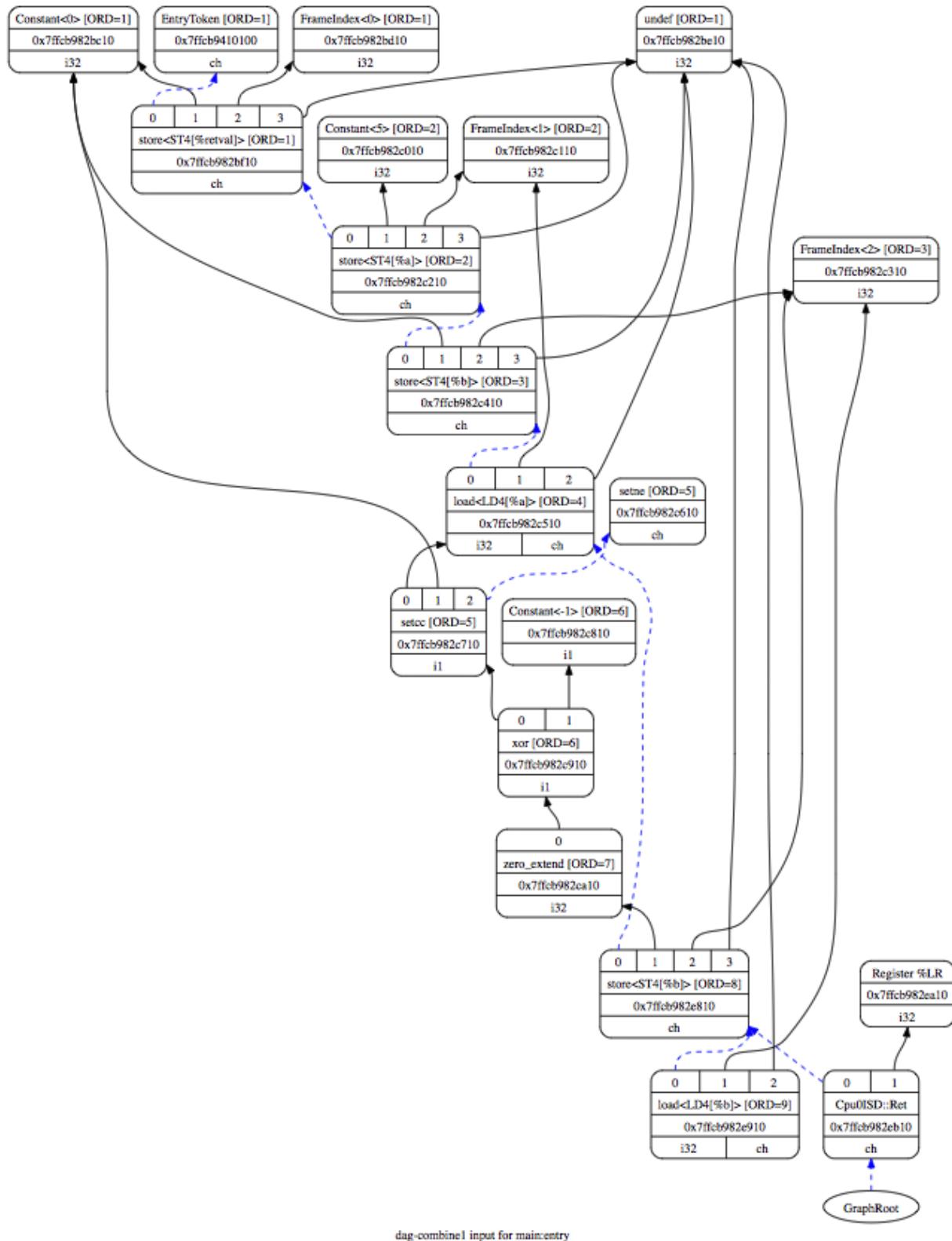


Figure 4.1: llc option -view-dag-combine1-dags graphic view

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,

LLVMBackendTutorialExampleCode/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>;
```

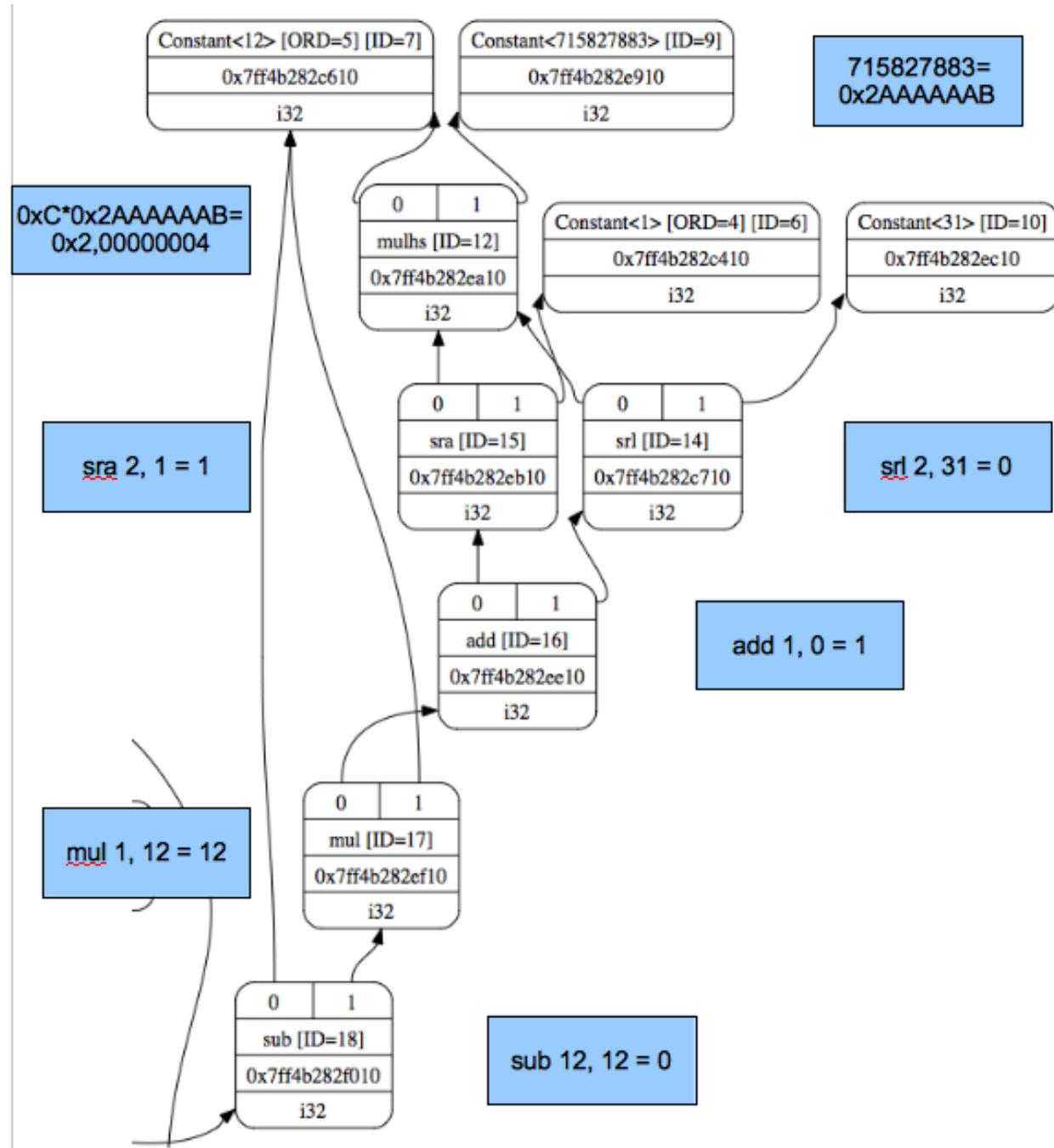


Figure 4.2: ch4_2.bc DAG

LLVMBackendTutorialExampleCode/Chapter4_1/Cpu0ISelDAGToDAG.cpp

```

#ifndef 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;
#ifndef 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 -
    .section .mdebug.abi32
    .previous
    .file "ch4_2.bc"
    .text
    .globl      main
    .align      2
    .type main,@function
    .ent main           # @main
main:
    .cfi_startproc

```

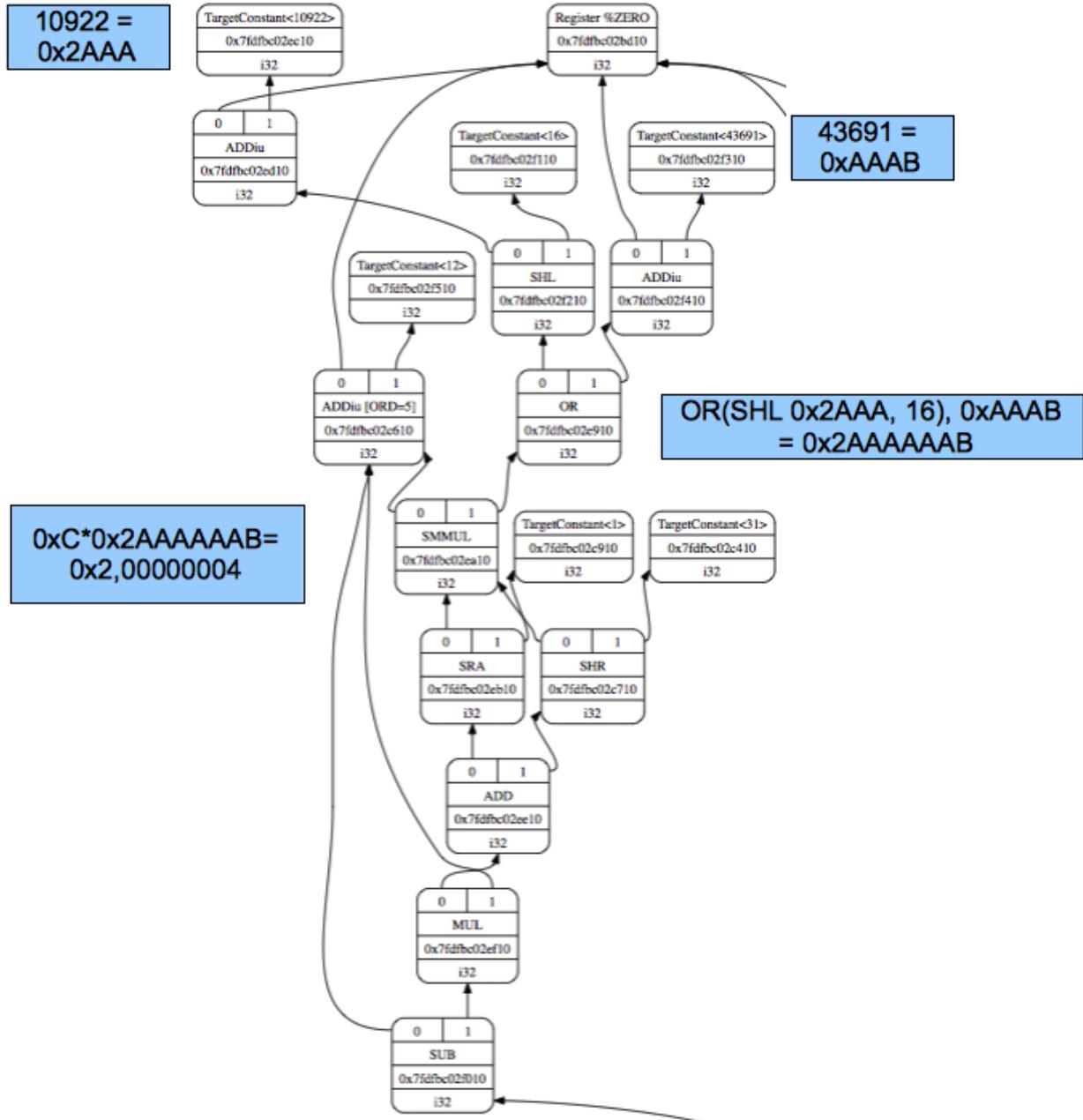


Figure 4.3: Translate ch4_2.bc into cpu0 backend DAG

```

.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. For Mips style implementation, we add the following code in Cpu0RegisterInfo.td, Cpu0InstrInfo.td and Cpu0ISelDAGToDAG.cpp. And list the related DAG nodes mulhs and mulhu which are used in Chapter4_1/ from TargetSelectionDAG.td.

LLVMBackendTutorialExampleCode/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)>

// Cpu0Schedule.td
...
def IIHiLo      : InstrItinClass;
...
def Cpu0GenericItineraries : ProcessorItineraries<[ALU, IMULDIV], [], [
...
InstrItinData<IIHiLo      , [InstrStage<1, [IMULDIV]>]>,
...
]>;
```

LLVMBackendTutorialExampleCode/Chapter4_1/Cpu0InstrInfo.td

```
// 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]>;

// 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;
}
...

/// Multiply and Divide Instructions.
def MULT      : Mult32<0x41, "mult", IIImul>;
def MULTu     : Mult32<0x42, "multu", IIImul>;

def MFHI      : MoveFromLOHI<0x46, "mfhi", CPURegs, [HI]>;
def MFLO      : MoveFromLOHI<0x47, "mflo", CPURegs, [LO]>;
```

LLVMBackendTutorialExampleCode/Chapter4_1/Cpu0ISelDAGToDAG.cpp

```
/// 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;
}
...
}

```

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 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 main
.align 2
.type main,@function
.ent main           # @main
main:
.cfi_startproc

```

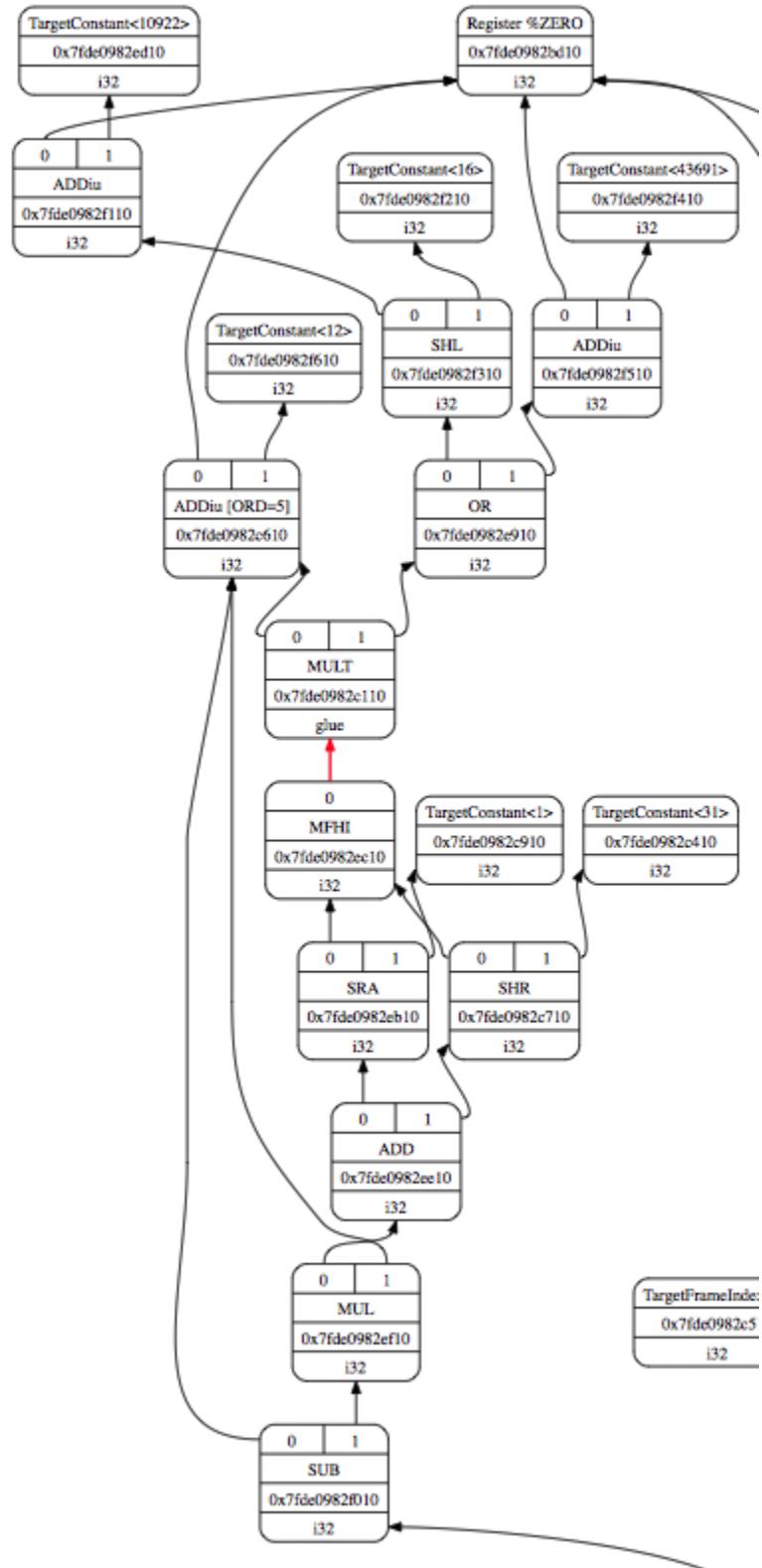


Figure 4.4: DAG for ch4_2.bc with Mips style MULT

```

.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 $2, $zero, 11
st $2, 0($sp)
addiu $2, $zero, 10922
shl $2, $2, 16
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
sub $2, $2, $3
st $2, 0($sp)
addiu $sp, $sp, 8
ret $lr
.set macro
.set reorder
.end main
$tmp2:
.size main, ($tmp2)-main
.cfi_endproc

```

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. In [section Support arithmetic instructions](#), we use “**div a, b**” to hold the quotient part in register. The multiplication operator “*****” need 64 bits of register to hold the result for two 32 bits of operands multiplication. We modify cpu0 to use the pair of registers LO and HI which just like Mips to solve this issue in last section. Now, it’s time to modify cpu0 for integer “**divide**” operator again. We 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**”.

Chapter4_1/ support operator “%” and “/”. The code added in Chapter4_1/ as follows,

LLVMBackendTutorialExampleCode/Chapter4_1/Cpu0InstrInfo.cpp

```

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 (Cpu0::CPURegsRegClass.contains(SrcReg))
        Opc = Cpu0::ADD, ZeroReg = Cpu0::ZERO;
    else if (SrcReg == Cpu0::HI)
        Opc = Cpu0::MFHI, SrcReg = 0;
    else if (SrcReg == Cpu0::LO)
        Opc = Cpu0::MFLO, SrcReg = 0;
}
else if (Cpu0::CPURegsRegClass.contains(SrcReg)) { // Copy from CPU Reg.
    if (DestReg == Cpu0::HI)
        Opc = Cpu0::MTHI, DestReg = 0;
    else if (DestReg == Cpu0::LO)
        Opc = Cpu0::MTLO, DestReg = 0;
}

assert(Opc && "Cannot copy registers");

MachineInstrBuilder MIB = BuildMI(MBB, I, DL, get(Opc));

if (DestReg)
    MIB.addReg(DestReg, RegState::Define);

if (ZeroReg)
    MIB.addReg(ZeroReg);

if (SrcReg)
    MIB.addReg(SrcReg, getKillRegState(KillSrc));
}

```

LLVMBackendTutorialExampleCode/Chapter4_1/Cpu0InstrInfo.h

```

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

```

LLVMBackendTutorialExampleCode/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 Div<SDNode opNode, bits<8> op, string instr_asm, InstrItinClass itin,
    RegisterClass RC, list<Register> DefRegs>:
    FL<op, (outs), (ins RC:$rb, RC:$rc),
    !strconcat(instr_asm, "\t$$zero, $rb, $rc"),
    [(opNode RC:$rb, RC:$rc)], 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]>;
...
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 SDIV      : Div32<Cpu0DivRem, 0x43, "div", IIIdiv>;
def UDIV      : Div32<Cpu0DivRemU, 0x44, "divu", IIIdiv>;
...
def MTHI      : MoveToLOHI<0x48, "mthi", CPURegs, [HI]>;
def MTLO      : MoveToLOHI<0x49, "mtlo", CPURegs, [LO]>;

```

LLVMBackendTutorialExampleCode/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)
const {
  SelectionDAG &DAG = DCI.DAG;
  unsigned opc = N->getOpcode();

  switch (opc) {
  default: break;
  case ISD::SDIVREM:
  case ISD::UDIVREM:
  return PerformDivRemCombine(N, DAG, DCI, Subtarget);
  }

  return SDValue();
}

```

LLVMBackendTutorialExampleCode/Chapter4_1/Cpu0ISelLowering.h

```

namespace llvm {
namespace Cpu0ISD {
enum NodeType {
  // Start the numbering from where ISD NodeType finishes.
  FIRST_NUMBER = ISD::BUILTIN_OP_END,
  Ret,
  // DivRem(u)
  DivRem,
  DivRemU
};
}
...

```

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.

LLVMBackendTutorialExampleCode/InputFiles/ch4_2_1.cpp

```

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

    b = (b+1)%a;

    return b;
}

```

LLVMBackendTutorialExampleCode/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 -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 -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. ⁵ and ⁶. 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.

4.1.4 Local variable pointer

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

LLVMBBackendTutorialExampleCode/Chapter4_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> {
```

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

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

```

    let isCodeGenOnly = 1;
}

```

LLVMBackendTutorialExampleCode/Chapter4_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 ch4_4.cpp with code Chapter4_1/ which support pointer to local variable, will get result as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch4_4.cpp

```

int test_local_pointer()  

{  

    int b = 3;  

    int* p = &b;  

    return *p;
}

```

```

118-165-66-82:InputFiles Jonathan$ clang -c ch4_4.cpp -emit-llvm -o ch4_4.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_4.bc -o ch4_4.cpu0.s
118-165-66-82:InputFiles Jonathan$ cat ch4_4.cpu0.s
    .section .mdebug.abi32
    .previous
    .file "ch4_4.bc"
    .text
    .globl      _Z18test_local_pointerv
    .align      2
    .type      _Z18test_local_pointerv,@function
    .ent      _Z18test_local_pointerv # @_Z18test_local_pointerv
_Z18test_local_pointerv:
    .cfi_startproc
    .frame      $fp,16,$lr
    .mask       0x00000800,-4
    .set      noreorder
    .set      nomacro
# BB#0:                      # %entry
    addiu $sp, $sp, -16
$tmp3:
    .cfi_def_cfa_offset 16
    st      $fp, 12($sp)          # 4-byte Folded Spill
$tmp4:
    .cfi_offset 11, -4
    addu $fp, $sp, $zero

```

```

$tmp5:
    .cfi_def_cfa_register 11
    addiu $2, $zero, 3
    st    $2, 8($fp)
    addiu $2, $fp, 8
    st    $2, 0($fp)
    ld    $2, 8($fp)
    addu $sp, $fp, $zero
    ld    $fp, 12($sp)           # 4-byte Folded Reload
    addiu $sp, $sp, 16
    ret   $lr
    .set  macro
    .set  reorder
    .end  _Z18test_local_pointerv

$tmp6:
    .size _Z18test_local_pointerv, ($tmp5)-_Z18test_local_pointerv
    .cfi_endproc

```

4.2 Logic

Chapter4_2 support logic operators **&**, **l**, **^**, **!**, **==**, **!=**, **<**, **<=**, **>** 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.

LLVMBackendTutorialExampleCode/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;
}

...
/// Arithmetic Instructions (ALU Immediate)
...
def ANDi      : ArithLogicI<0x0c, "andi", and, uimm16, immZExt16, CPURegs>;
def ORi       : ArithLogicI<0x0d, "ori", or, 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 : Pat<(not CPURegs:$in),
// 1: in == 0; 0: in != 0
  (XORi CPURegs:$in, 1)>;
// Sign Extend in Register.

```

```

// Get the least 7bits from register $in and signed bit (the 31th bit) from the
// 8th bit of register $in.
def : Pat<(sext_inreg CPUREgs:$in, i8),
       (OR (SHL (ANDi CPUREgs:$in, 0x0080), 16), (ANDi CPUREgs:$in, 0x007f))>;
// Get the least 15bits from register $in and signed bit (the 31th bit) from the
// 16th bit of register $in.
def : Pat<(sext_inreg CPUREgs:$in, i8),
       (OR (SHL (ANDi CPUREgs:$in, 0x8000), 16), (ANDi CPUREgs:$in, 0x7fff))>

// 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>;
```

LLVMBackendTutorialExampleCode/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 -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 -
    .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:
    .cfi_startproc
    .frame      $fp,24,$lr
    .mask       0x00000800,-4
    .set  noreorder
    .set  nomacro
# BB#0:                      # %entry
    addiu $sp, $sp, -24
$tmp3:
    .cfi_def_cfa_offset 24
    st   $fp, 20($sp)          # 4-byte Folded Spill
$tmp4:
    .cfi_offset 11, -4
    addu $fp, $sp, $zero
$tmp5:
    .cfi_def_cfa_register 11
    addiu $2, $zero, 5
    st   $2, 16($fp)
    addiu $2, $zero, 3
    st   $2, 12($fp)
    addiu $2, $zero, 0
    st   $2, 8($fp)
    st   $2, 4($fp)
    st   $2, 0($fp)
    ld   $3, 12($fp)

```

```

ld    $4, 16($fp)
and  $3, $4, $3
st    $3, 8($fp)
ld    $3, 12($fp)
ld    $4, 16($fp)
or    $3, $4, $3
st    $3, 4($fp)
ld    $3, 12($fp)
ld    $4, 16($fp)
xor  $3, $4, $3
st    $3, 0($fp)
ld    $3, 16($fp)
cmp   $sw, $3, $2
andi  $2, $sw, 2
shr   $2, $2, 1
st    $2, 12($fp)
ld    $3, 4($fp)
ld    $4, 8($fp)
addu $3, $4, $3
ld    $4, 0($fp)
addu $3, $3, $4
addu $2, $3, $2
addu $sp, $fp, $zero
ld    $fp, 20($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 24
ret   $lr
.set  macro
.set  reorder
.end  _Z16test_andorxornotv

$tmp6:
.size _Z16test_andorxornotv, ($tmp6)-_Z16test_andorxornotv
.cfi_endproc

.globl      _Z10test_setxxv
.align      2
.type      _Z10test_setxxv, @function
.ent       _Z10test_setxxv      # @_Z10test_setxxv
_Z10test_setxxv:
.cfi_startproc
.frame     $fp, 40, $lr
.mask      0x00000800, -4
.set      noreorder
.set      nomacro
# BB#0:                      # %entry
addiu $sp, $sp, -40
$tmp10:
.cfi_def_cfa_offset 40
st    $fp, 36($sp)           # 4-byte Folded Spill
$tmp11:
.cfi_offset 11, -4
addu $fp, $sp, $zero
$tmp12:
.cfi_def_cfa_register 11
addiu $2, $zero, 5
st    $2, 32($fp)
addiu $2, $zero, 3
st    $2, 28($fp)
ld    $3, 32($fp)

```

```

        cmp    $sw, $3, $2
        andi   $2, $sw, 2
        shr    $2, $2, 1
        st     $2, 24($fp)
        ld     $2, 28($fp)
        ld     $3, 32($fp)
        cmp    $sw, $3, $2
        andi   $2, $sw, 2
        shr    $2, $2, 1
        xorri $2, $2, 1
        st     $2, 20($fp)
        ld     $2, 28($fp)
        ld     $3, 32($fp)
        cmp    $sw, $3, $2
        andi   $2, $sw, 1
        st     $2, 16($fp)
        ld     $2, 32($fp)
        ld     $3, 28($fp)
        cmp    $sw, $3, $2
        andi   $2, $sw, 1
        xorri $2, $2, 1
        st     $2, 12($fp)
        ld     $2, 32($fp)
        ld     $3, 28($fp)
        cmp    $sw, $3, $2
        andi   $2, $sw, 1
        st     $2, 8($fp)
        ld     $2, 28($fp)
        ld     $3, 32($fp)
        cmp    $sw, $3, $2
        andi   $2, $sw, 1
        xorri $2, $2, 1
        st     $2, 4($fp)
        ld     $3, 20($fp)
        ld     $4, 24($fp)
        addu  $3, $4, $3
        ld     $4, 16($fp)
        addu  $3, $3, $4
        ld     $4, 12($fp)
        addu  $3, $3, $4
        ld     $4, 8($fp)
        addu  $3, $3, $4
        addu  $2, $3, $2
        addu  $sp, $fp, $zero
        ld     $fp, 36($sp)          # 4-byte Folded Reload
        addiu $sp, $sp, 40
        ret    $lr
        .set   macro
        .set   reorder
        .end   _Z10test_setxxv
$tmp13:
.size _Z10test_setxxv, ($tmp13)-_Z10test_setxxv
.cfi_endproc
    
```

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"> %tobool = icmp ne i32 %6, 0 %lnot = xor i1 %tobool, true %conv = zext i1 %lnot to i32 	<ul style="list-style-type: none"> %lnot = (setcc %tobool, 0, seteq) %conv = (and %lnot, 1) • 	<ul style="list-style-type: none"> xor \$3, \$4, \$3
==	<ul style="list-style-type: none"> %cmp = icmp eq i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, seteq) 	<ul style="list-style-type: none"> cmp \$sw, \$3, \$2 andi \$2, \$sw, 2 shr \$2, \$2, 1
!=	<ul style="list-style-type: none"> %cmp = icmp ne i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setne) 	<ul style="list-style-type: none"> cmp \$sw, \$3, \$2 andi \$2, \$sw, 2 shr \$2, \$2, 1
<	<ul style="list-style-type: none"> %cmp = icmp lt i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setlt) 	<ul style="list-style-type: none"> cmp \$sw, \$3, \$2 andi \$2, \$sw, 2
<=	<ul style="list-style-type: none"> %cmp = icmp le i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, settle) 	<ul style="list-style-type: none"> cmp \$sw, \$2, \$3 andi \$2, \$sw, 1 xori \$2, \$2, 1
>	<ul style="list-style-type: none"> %cmp = icmp gt i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, setgt) 	<ul style="list-style-type: none"> cmp \$sw, \$2, \$3 andi \$2, \$sw, 2
>=	<ul style="list-style-type: none"> %cmp = icmp le i32 %0, %1 %conv = zext i1 %cmp to i32 	<ul style="list-style-type: none"> (setcc %0, %1, settle) 	<ul style="list-style-type: none"> cmp \$sw, \$3, \$2 andi \$2, \$sw, 1 xori \$2, \$2, 1

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

The “Optimized legalized selection DAG” is the last DAG stage just before the “instruction selection” as the section mentioned in this chapter. You can see the whole DAG stages by `llc -debug` option.

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

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_1/ 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 09ddffd8 09200005 022b0024 09200002  ..... .+.$. .
```

```

0010 022b0020 09200000 022b001c 022b0018 .+. . . . .+...+..
0020 022b0010 0930ffffb 023b000c 022b0008 .+...0...;...+..
0030 022b0004 012b0020 013b0024 11232000 .+...+. .;.$.# .
0040 022b001c 012b0020 013b0024 12232000 .+...+. .;.$.# .
0050 022b0018 012b0020 013b0024 17232000 .+...+. .;.$.# .
0060 022b0018 012b0024 1e220002 022b0014 .+...+.$.#"...+..
0070 012b000c 1e220002 022b0008 012b0024 .+...#"...+...+$
0080 1d220002 022b0010 012b000c 1f220002 ."...+...+...".
0090 022b0004 09200001 013b0024 21323000 .+... . .;.$!20.
00a0 023b0014 013b000c 21223000 022b0008 .;...;...!"0..+..
00b0 0f208000 013b0024 22223000 022b0010 . . .;.$""0..+..
00c0 012b0024 013b0020 20232000 022b0004 .+.$.; # .+..
00d0 09dd0028 2c00000e ... (, ...

Contents of section .eh_frame:
0000 00000010 00000000 017a5200 017c0e01 .....zR...|...
0010 1b0c0d00 00000010 00000018 00000000 .....D..(.
0020 000000d8 00440e28 .....D..(.

```

```

[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 d8fffd09 05002009 24002b02 02002009 .....$.+... .
0010 20002b02 00002009 1c002b02 18002b02 .+... . .+...+.
0020 10002b02 fbf3009 0c003b02 08002b02 ..+...0...;...+.
0030 04002b02 20002b01 24003b01 00202311 ..+.. .+.$.;... #.
0040 1c002b02 20002b01 24003b01 00202312 ..+.. .+.$.;... #.
0050 18002b02 20002b01 24003b01 00202317 ..+.. .+.$.;... #.
0060 18002b02 24002b01 0200221e 14002b02 ..+..$.+...#"...+.
0070 0c002b01 0200221e 08002b02 24002b01 ..+...#"...+.$.+.
0080 0200221d 10002b02 0c002b01 0200221f ."...+...+...".
0090 04002b02 01002009 24003b01 00303221 ..+... .$.;...02!
00a0 14003b02 0c003b01 00302221 08002b02 .;...;...0!"...+.
00b0 0080200f 24003b01 00302222 10002b02 .. .$.;...0""...+.
00c0 24002b01 20003b01 00202320 04002b02 $.. . .;... # .+.
00d0 2800dd09 0e00002c .....,
Contents of section .eh_frame:
0000 10000000 00000000 017a5200 017c0e01 .....zR...|...
0010 1b0c0d00 10000000 18000000 00000000 .....D..(.
0020 d8000000 00440e28 .....D..(.

```

The first instruction is “**addiu \$sp, -40**” and it’s corresponding obj is 0x09ddffd8. The addiu opcode is 0x09, 8 bits, \$sp register number is 13(0xd), 4bits, and the immediate is 16 bits -40(=0xffd8), so it’s correct. The third instruction “**st \$2, 36(\$fp)**” and it’s corresponding obj is 0x022b0024. The **st** opcode is **0x02**, \$2 is 0x2, \$fp is 0xb and immediate is 36(0x0024). 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, d8), objdump from B0 to B3 as 0x09ddffd8 and the little endian is (B3, B2, B1, B0) = (09, dd, ff, d8), objdump from B0 to B3 as 0xd8fffd09.

5.2 Backend Target Registration Structure

Now, let’s examine Cpu0MCTargetDesc.cpp.

LLVMBackendTutorialExampleCode/Chapter5_1/MCTargetDesc/Cpu0MCTargetDesc.cpp

```

}

extern "C" void LLVMInitializeCpu0TargetMC() {
    // Register the MC asm info.
    RegisterMCAsmInfoFn X(TheCpu0Target, createCpu0MCAsmInfo);
    RegisterMCAsmInfoFn Y(TheCpu0elTarget, createCpu0MCAsmInfo);

    // Register the MC codegen info.
    TargetRegistry::RegisterMCCodeGenInfo(TheCpu0Target,
                                           createCpu0MCCodeGenInfo);
    TargetRegistry::RegisterMCCodeGenInfo(TheCpu0elTarget,
                                           createCpu0MCCodeGenInfo);
    // Register the MC instruction info.
    TargetRegistry::RegisterMCInstrInfo(TheCpu0Target, createCpu0MCInstrInfo);
    TargetRegistry::RegisterMCInstrInfo(TheCpu0elTarget, createCpu0MCInstrInfo);

    // Register the MC register info.
    TargetRegistry::RegisterMCRegInfo(TheCpu0Target, createCpu0MCRegisterInfo);
    TargetRegistry::RegisterMCRegInfo(TheCpu0elTarget, createCpu0MCRegisterInfo);

    // Register the MC Code Emitter
    TargetRegistry::RegisterMCCodeEmitter(TheCpu0Target,
                                           createCpu0MCCodeEmitterEB);
    TargetRegistry::RegisterMCCodeEmitter(TheCpu0elTarget,
                                           createCpu0MCCodeEmitterEL);

    // Register the object streamer.
    TargetRegistry::RegisterMCObjectStreamer(TheCpu0Target, createMCStreamer);
    TargetRegistry::RegisterMCObjectStreamer(TheCpu0elTarget, createMCStreamer);

    // Register the asm backend.
    TargetRegistry::RegisterMCAsmBackend(TheCpu0Target,
                                         createCpu0AsmBackendEB32);
    TargetRegistry::RegisterMCAsmBackend(TheCpu0elTarget,
                                         createCpu0AsmBackendEL32);
    // Register the MC subtarget info.
    TargetRegistry::RegisterMCSubtargetInfo(TheCpu0Target,
                                           createCpu0MCSubtargetInfo);
    TargetRegistry::RegisterMCSubtargetInfo(TheCpu0elTarget,
                                           createCpu0MCSubtargetInfo);
    // Register the MCInstPrinter.
    TargetRegistry::RegisterMCInstPrinter(TheCpu0Target,
                                         createCpu0MCInstPrinter);
    TargetRegistry::RegisterMCInstPrinter(TheCpu0elTarget,
                                         createCpu0MCInstPrinter);
}

```

Cpu0MCTargetDesc.cpp do the target registration as mentioned in “section Target Registration”¹ of the last chapter. Drawing the register function and those class it registered in Figure 5.1 to Figure 5.9 for explanation.

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 MCCodeGenInfo, and initialize it by pass Roloc::PIC because we use command llc

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

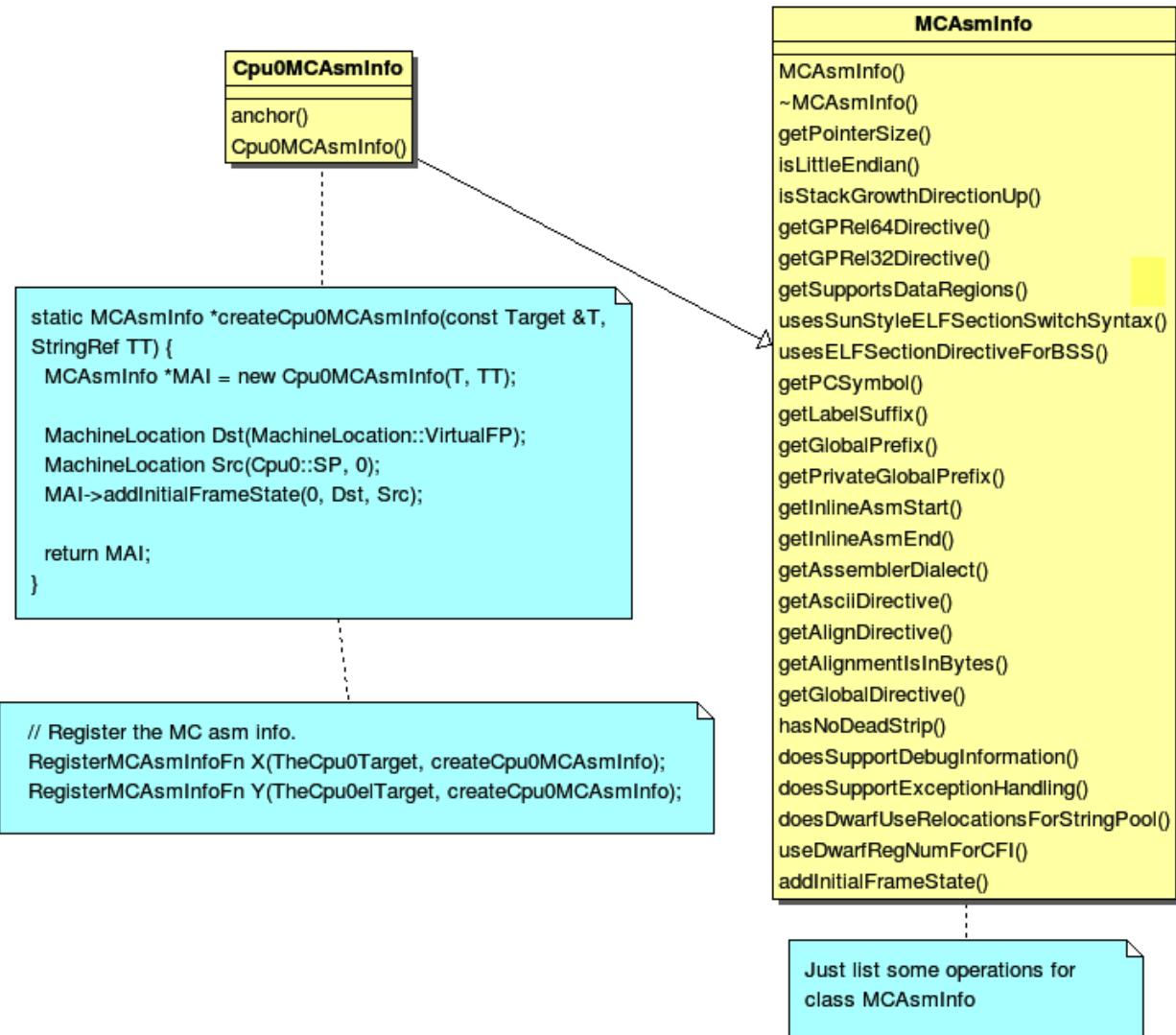


Figure 5.1: Register Cpu0MCAsmInfo

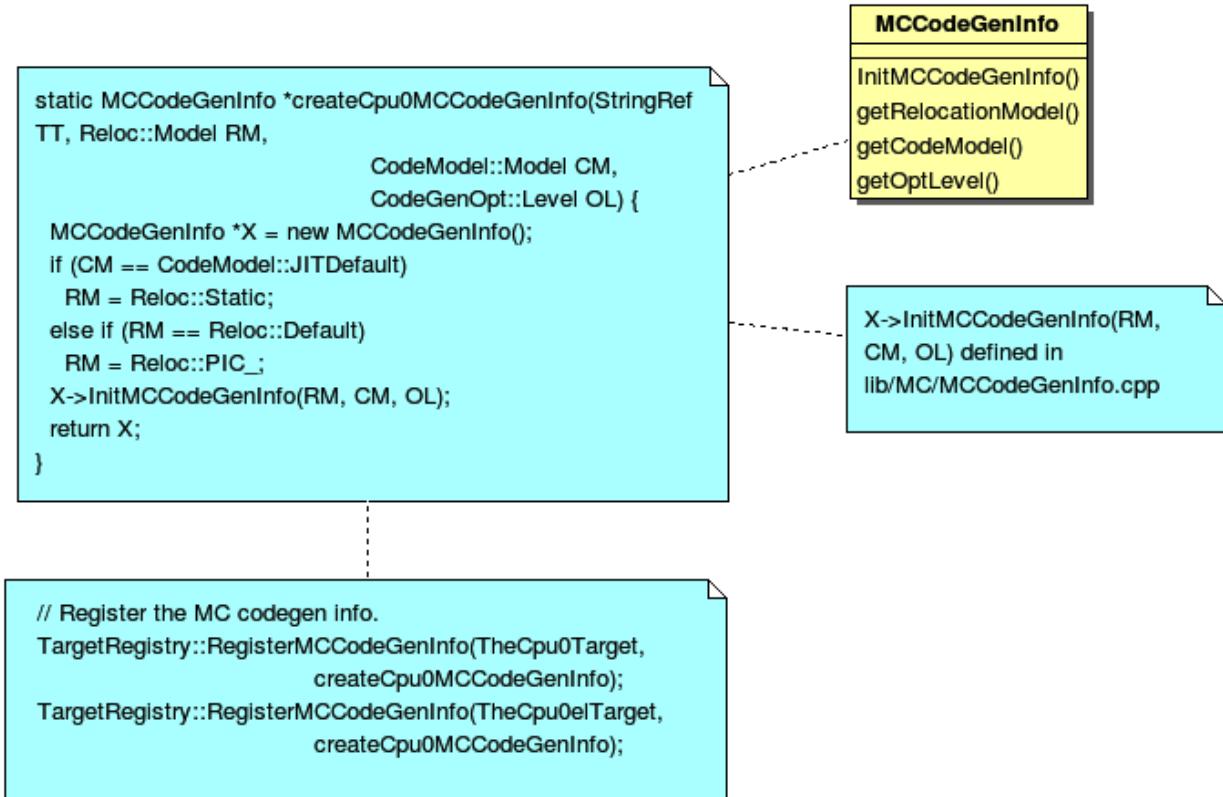


Figure 5.2: Register MCCCodeGenInfo

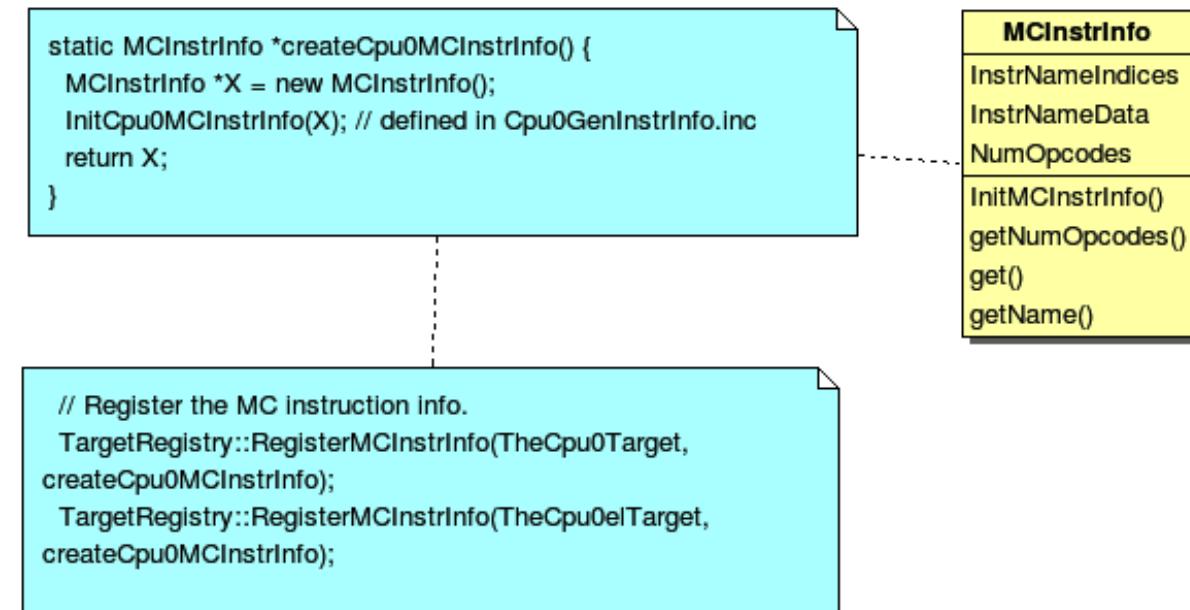


Figure 5.3: Register MCInstrInfo

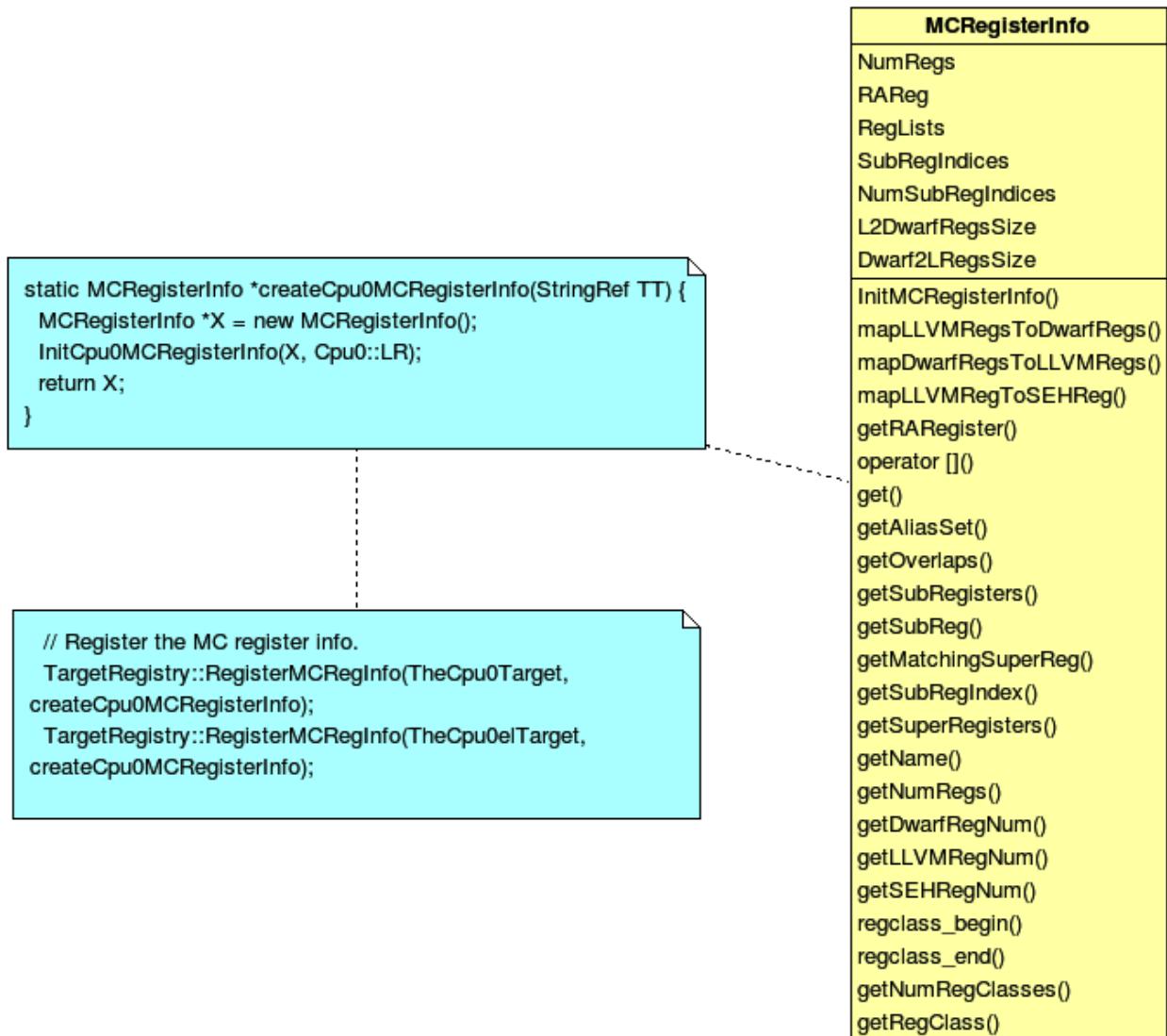


Figure 5.4: Register MCRegisterInfo

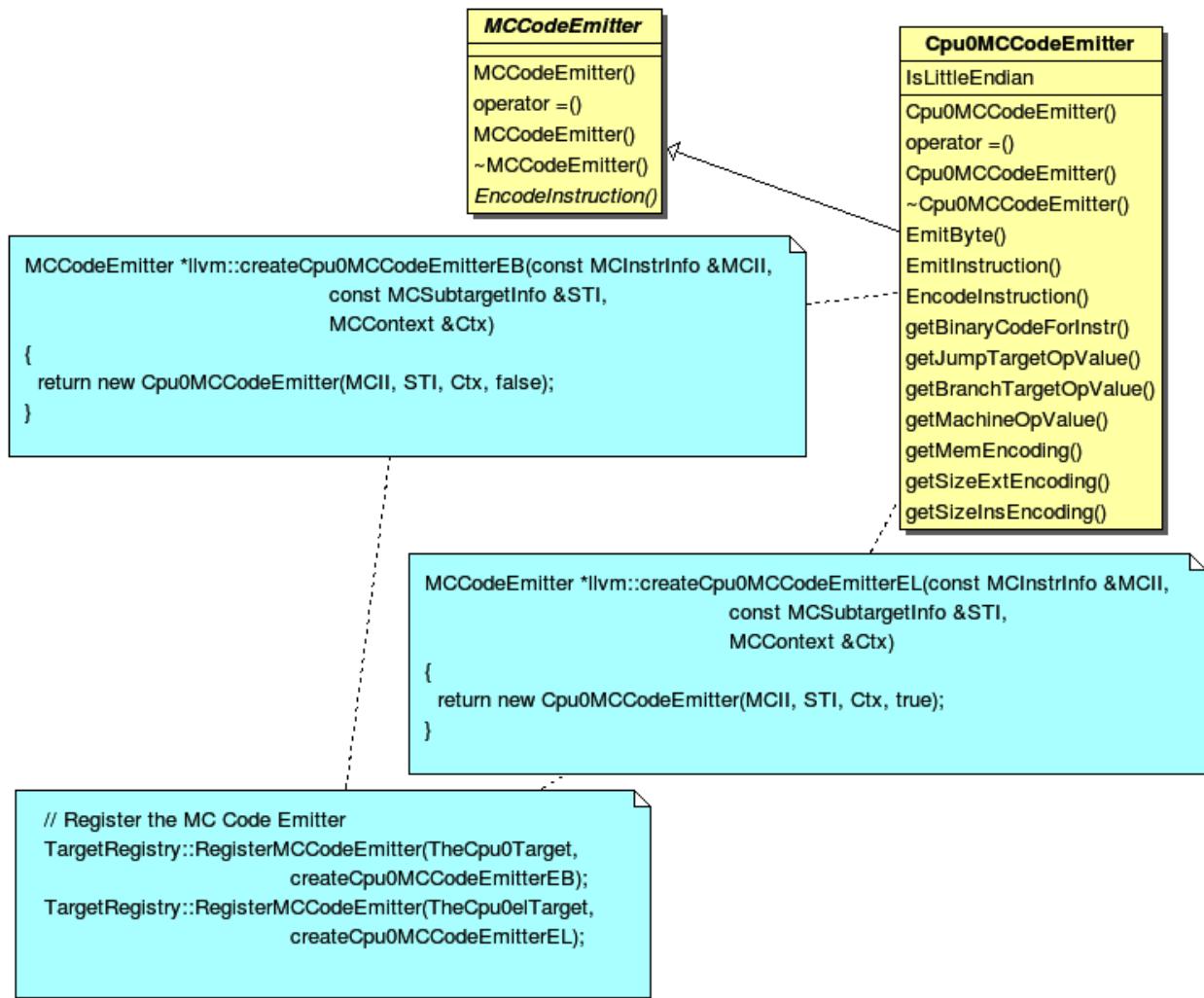


Figure 5.5: Register Cpu0MCCodeEmitter

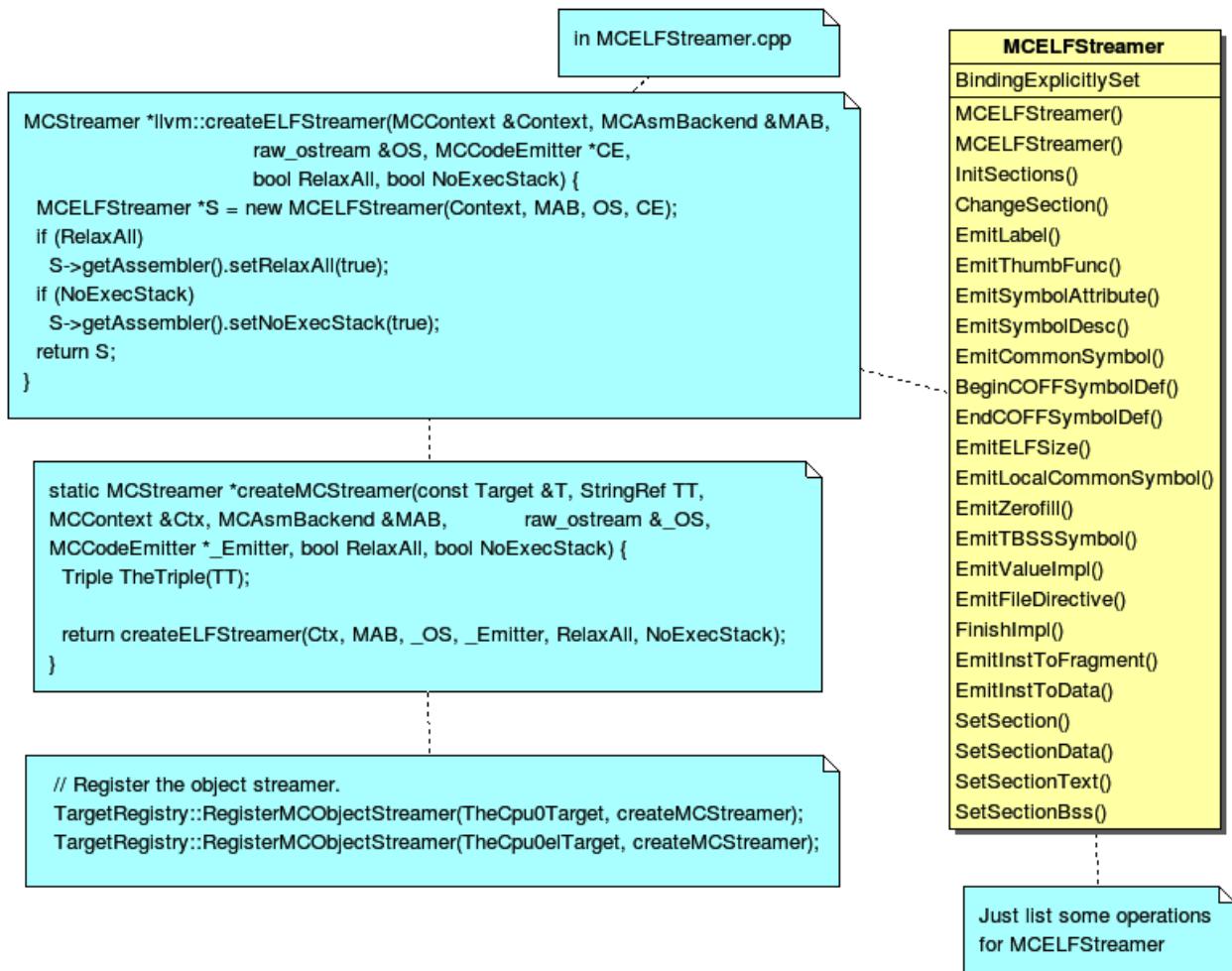


Figure 5.6: Register MCELFStreamer

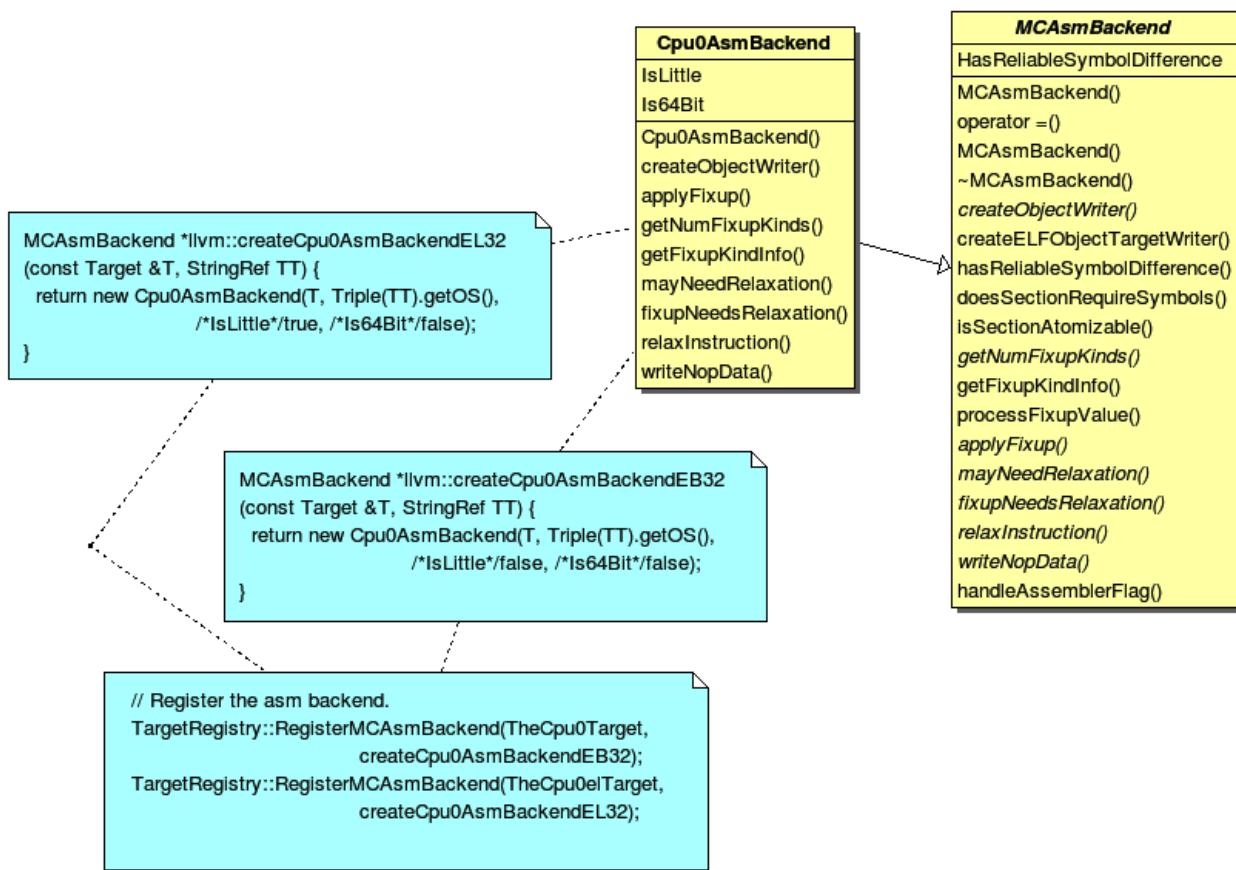


Figure 5.7: Register Cpu0AsmBackend

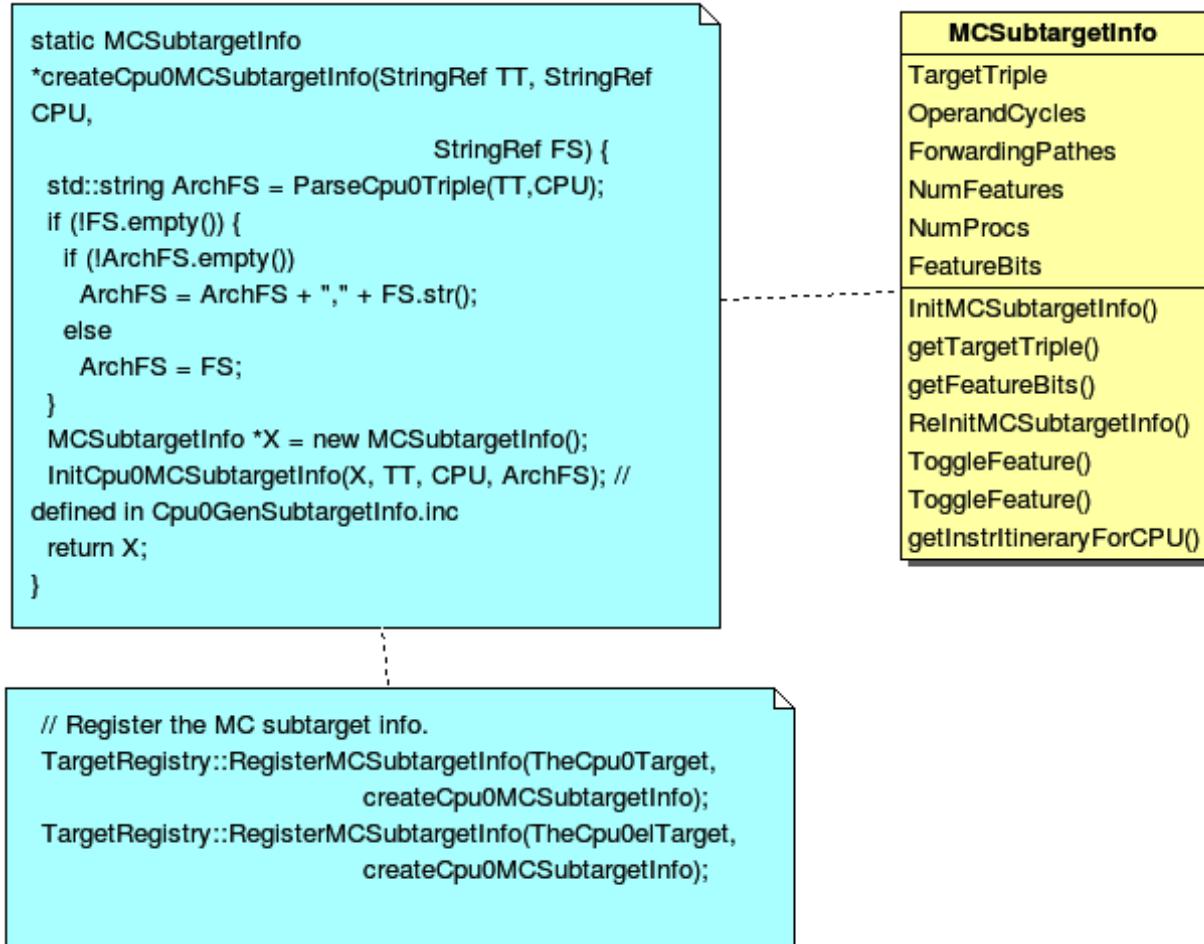


Figure 5.8: Register Cpu0MCSUBTARGETINFO

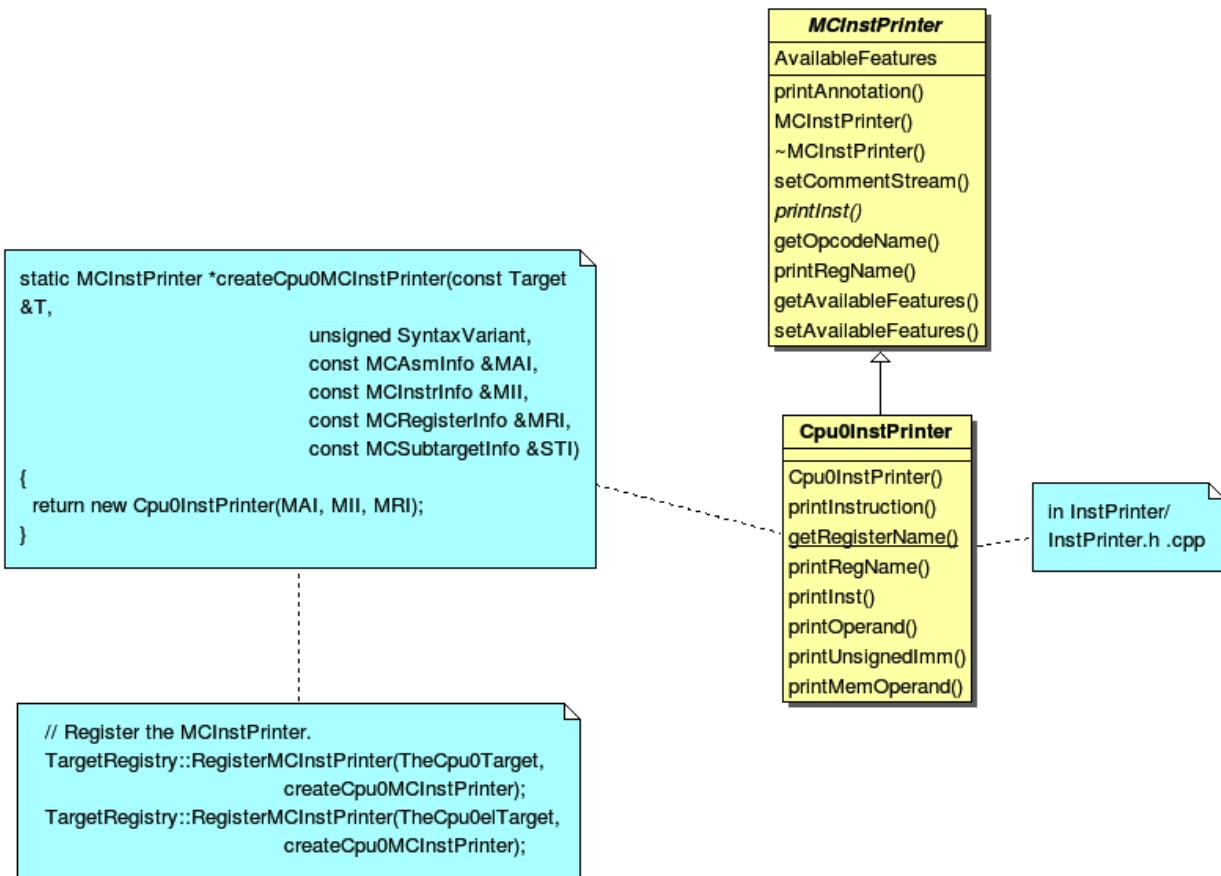


Figure 5.9: Register Cpu0InstPrinter



Figure 5.10: MCELFStreamer inherit tree

`-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_6_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_6_2/](#) code for assembly file generated support.

GLOBAL VARIABLES, STRUCTS AND ARRAYS, OTHER TYPE

In the previous two chapters, we only access the local variables. This chapter will deal global variable access translation. After that, introducing the types of struct and array as well as their corresponding llvm IR statement, and how the cpu0 translate these llvm IR statements in [section Array and struct support](#). Finally, we deal the other types such as “**short int**” and **char** in the last section.

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 `l1c -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.

[LLVMBackendTutorialExampleCode/InputFiles/ch6_1.cpp](#)

```
int gStart;
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 -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]
...
addiu  $2, $zero, %hi(gI)
shl   $2, $2, 16
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"> • pic: Position Independent Address • static: Absolute Address
-cpu0-use-small-section	false	true	<ul style="list-style-type: none"> • false: .data or .bss, 16 bits addressable • true: .sdata or .sbss, 32 bits addressable

Table 6.2: Cpu0 DAGs and instructions for -relocation-model=static

option: cpu0-use-small-section	false	true
addressing mode	absolute	\$gp relative
addressing	absolute	\$gp+offset
Legalized selection DAG	(add Cpu0ISD::Hi<gI offset Hi16> Cpu0ISD::Lo<gI offset Lo16>)	(add 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 ¹ further.

LLVMBackendTutorialExampleCode/Chapter6_1/Cpu0Subtarget.h

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

LLVMBackendTutorialExampleCode/Chapter6_1/Cpu0Subtarget.cpp

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

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

Next add file Cpu0TargetObjectFile.h, Cpu0TargetObjectFile.cpp and the following code to Cpu0RegisterInfo.cpp and Cpu0ISelLowering.cpp.

LLVMBackendTutorialExampleCode/Chapter6_1/Cpu0TargetObjectFile.h

```
===== llvm/Target/Cpu0TargetObjectFile.h - Cpu0 Object Info ----- C++ -----//  
//  
//          The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
=====//  
  
#ifndef LLVM_TARGET_CPU0_TARGETOBJECTFILE_H  
#define LLVM_TARGET_CPU0_TARGETOBJECTFILE_H  
  
#include "llvm/CodeGen/TargetLoweringObjectFileImpl.h"  
  
namespace llvm {  
  
    class Cpu0TargetObjectFile : public TargetLoweringObjectFileELF {  
        const MCSection *SmallDataSection;  
        const MCSection *SmallBSSSection;  
    public:  
  
        void Initialize(MCContext &Ctx, const TargetMachine &TM);  
  
        /// IsGlobalInSmallSection - Return true if this global address should be  
        /// placed into small data/bss section.  
        bool IsGlobalInSmallSection(const GlobalValue *GV,  
                                  const TargetMachine &TM, SectionKind Kind) const;  
        bool IsGlobalInSmallSection(const GlobalValue *GV,  
                                  const TargetMachine &TM) const;  
  
        const MCSection *SelectSectionForGlobal(const GlobalValue *GV,  
                                              SectionKind Kind,  
                                              Mangler *Mang,  
                                              const TargetMachine &TM) const;  
  
        // TODO: Classify globals as cpu0 wishes.  
    };  
} // end namespace llvm  
  
#endif
```

LLVMBackendTutorialExampleCode/Chapter6_1/Cpu0TargetObjectFile.cpp

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

```

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

#include "Cpu0TargetObjectFile.h"
#include "Cpu0Subtarget.h"
#include "llvm/IR/DerivedTypes.h"
#include "llvm/IR/GlobalVariable.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCSectionELF.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/ELF.h"
using namespace llvm;

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

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

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

    SmallBSSSection =
        getContext().getELFSection(".sbss", ELF::SHT_NOBITS,
                                  ELF::SHF_WRITE | ELF::SHF_ALLOC,
                                  SectionKind::getBSS());
}

// A address must be loaded from a small section if its size is less than the
// small section size threshold. Data in this section must be addressed using
// gp_rel operator.
static bool IsInSmallSection(uint64_t Size) {
    return Size > 0 && Size <= SSThreshold;
}

bool Cpu0TargetObjectFile::IsGlobalInSmallSection(const GlobalValue *GV,
                                                const TargetMachine &TM) const {
    if (GV->isDeclaration() || GV->hasAvailableExternallyLinkage())
        return false;

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

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

    // Only use small section for non linux targets.
    const Cpu0Subtarget &Subtarget = TM.getSubtarget<Cpu0Subtarget>();

```

```

// Return if small section is not available.
if (!Subtarget.useSmallSection())
    return false;

// Only global variables, not functions.
const GlobalVariable *GVA = dyn_cast<GlobalVariable>(GV);
if (!GVA)
    return false;

// We can only do this for datarel or BSS objects for now.
if (!Kind.isBSS() && !Kind.isDataRel())
    return false;

// If this is a internal constant string, there is a special
// section for it, but not in small data/bss.
if (Kind.isMergeable1ByteCString())
    return false;

Type *Ty = GV->getType()->getElementType();
return IsInSmallSection(TM.getDataLayout()->getTypeAllocSize(Ty));
}

const MCSection *Cpu0TargetObjectFile::
SelectSectionForGlobal(const GlobalValue *GV, SectionKind Kind,
                     Mangler *Mang, const TargetMachine &TM) const {
    // TODO: Could also support "weak" symbols as well with ".gnu.linkonce.s.*"
    // sections?

    // Handle Small Section classification here.
    if (Kind.isBSS() && IsGlobalInSmallSection(GV, TM, Kind))
        return SmallBSSSection;
    if (Kind.isDataNoRel() && IsGlobalInSmallSection(GV, TM, Kind))
        return SmallDataSection;

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

```

LLVMBackendTutorialExampleCode/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);
    }
    ...
}

```

LLVMBackendTutorialExampleCode/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_) {
    SDVList VTs = DAG.getVList(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.

LLVMBackendTutorialExampleCode/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 variable 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.

```

...
    addiu  $2, $zero, %hi(gI)
    shl    $2, $2, 16
    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 its high part of `gI` absolute address. Next, it add register `$2` and low part of `gI` absolute address into `$2`. At this point, it gets the `gI` memory address. Finally, it gets the `gI` content by instruction “`ld $2, 0($2)`”. The `l1c -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 calculated. Since use PC relative address coding, this program can be loaded to any address and run well there. If this program uses 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 uses 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.

LLVMBackendTutorialExampleCode/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 -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.

LLVMBackendTutorialExampleCode/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.

LLVMBackendTutorialExampleCode/Chapter6_1/Cpu0ISelLowering.cpp

```
// Cpu0ISelLowering.cpp
...
// %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);
}

...
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.

LLVMBackendTutorialExampleCode/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
    // ld    $2, %got(gI)($gp)
    case ISD::GLOBAL_OFFSET_TABLE:
        return getGlobalBaseReg();
    ...
}

```

LLVMBackendTutorialExampleCode/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 llvmm never allocate \$gp for variables.

LLVMBackendTutorialExampleCode/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;
    if (RM == Reloc::Static && !UseSmallSection)
        FixGlobalBaseReg = false;
    else
        FixGlobalBaseReg = true;
}
```

LLVMBackendTutorialExampleCode/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.

LLVMBackendTutorialExampleCode/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
```

LLVMBackendTutorialExampleCode/Chapter6_1/Cpu0MachineFunction.cpp

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

#include "Cpu0MachineFunction.h"
#include "Cpu0InstrInfo.h"
#include "Cpu0Subtarget.h"
#include "MCTargetDesc/Cpu0BaseInfo.h"
#include "llvm/IR/Function.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"

using namespace llvm;

bool FixGlobalBaseReg = true;

bool Cpu0FunctionInfo::globalBaseRegFixed() const {
```

```

    return FixGlobalBaseReg;
}

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

unsigned Cpu0FunctionInfo::getGlobalBaseReg() {
    // Return if it has already been initialized.
    if (GlobalBaseReg)
        return GlobalBaseReg;

    if (FixGlobalBaseReg) // $gp is the global base register.
        return GlobalBaseReg = Cpu0::GP;

    const TargetRegisterClass *RC;
    RC = (const TargetRegisterClass*)&Cpu0::CPURegsRegClass;

    return GlobalBaseReg = MF.getRegInfo().createVirtualRegister(RC);
}

void Cpu0FunctionInfo::anchor() { }

```

LLVMBackendTutorialExampleCode/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.LowerCPOLOAD(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 ABI².

Following code will expand .cupload into machine instructions as below. “09a00000 1eaa0010 09aa0000 13aa6000” is the **.cupload** machine instructions displayed in comments of Cpu0MCInstLower.cpp.

LLVMBackendTutorialExampleCode/Chapter6_1/Cpu0MCInstLower.cpp

```

}

static void CreateMCInst(MCInst& Inst, unsigned Opc, const MCOperand& Opnd0,
                        const MCOperand& Opnd1,
                        const MCOperand& Opnd2 = MCOperand()) {
    Inst.setOpcode(Opc);
    Inst.addOperand(Opnd0);
    Inst.addOperand(Opnd1);
    if (Opnd2.isValid())
        Inst.addOperand(Opnd2);
}

// Lower ".cupload $reg" to
// "lui $gp, %hi(_gp_disp)"
// "addiu $gp, $gp, %lo(_gp_disp)"
// "addu $gp, $gp, $t9"
void Cpu0MCInstLower::LowerCPLOAD(SmallVector<MCInst, 4>& MCInsts) {
    MCOperand GPReg = MCOperand::CreateReg(Cpu0::GP);
    MCOperand T9Reg = MCOperand::CreateReg(Cpu0::T9);
    MCOperand ZEROReg = MCOperand::CreateReg(Cpu0::ZERO);
    StringRef SymName("_gp_disp");
    const MCSymbol *Sym = Ctx->GetOrCreateSymbol(SymName);
    const MCSymbolRefExpr *MCSym;

    MCSym = MCSymbolRefExpr::Create(Sym, MCSymbolRefExpr::VK_Cpu0_ABS_HI, *Ctx);
    MCOperand SymHi = MCOperand::CreateExpr(MCSym);
    MCSym = MCSymbolRefExpr::Create(Sym, MCSymbolRefExpr::VK_Cpu0_ABS_LO, *Ctx);
    MCOperand SymLo = MCOperand::CreateExpr(MCSym);

    MCInsts.resize(3);

    CreateMCInst(MCInsts[0], Cpu0::LUI, GPReg, ZEROReg, SymHi);
    CreateMCInst(MCInsts[1], Cpu0::ADDIU, GPReg, GPReg, SymLo);
    CreateMCInst(MCInsts[2], Cpu0::ADD, GPReg, GPReg, T9Reg);
}

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 09a00000 1eaa0010 09aa0000 13aa6000  .....`.
0010 09ddfff8 09200000 022d0004 022d0000  .....-...-..
...

```

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

```
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 relocation 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.

LLVMBackendTutorialExampleCode/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);
    ...
}
```

LLVMBackendTutorialExampleCode/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=3]

    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.

LLVMBackendTutorialExampleCode/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.

LLVMBackendTutorialExampleCode/Chapter6_1/Cpu0MCInstLower.cpp

```

MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,
                                                MachineOperandType MOTY,
                                                unsigned Offset) const {
    MCSymbolRefExpr::VariantKind Kind;
    const MCSymbol *Symbol;

```

```

switch (MO.getTargetFlags()) {
    default:                                         llvm_unreachable("Invalid target flag!");
// Cpu0_GPREL is for llc -march=cpu0 -relocation-model=static
// -cpu0-use-small-section=false (global var in .sdata)
    case Cpu0II::MO_GPREL:      Kind = MCSymbolRefExpr::VK_Cpu0_GPREL; break;

    case Cpu0II::MO_GOT16:      Kind = MCSymbolRefExpr::VK_Cpu0_GOT16; break;
    case Cpu0II::MO_GOT:        Kind = MCSymbolRefExpr::VK_Cpu0_GOT; break;
// ABS_HI and ABS_LO is for llc -march=cpu0 -relocation-model=static
// (global var in .data)
    case Cpu0II::MO_ABS_HI:    Kind = MCSymbolRefExpr::VK_Cpu0_ABS_HI; break;
    case Cpu0II::MO_ABS_LO:    Kind = MCSymbolRefExpr::VK_Cpu0_ABS_LO; break;
}

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

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

MCOperand Cpu0MCInstLower::LowerOperand(const MachineOperand& MO,
                                         unsigned offset) const {
    MachineOperandType MOTy = MO.getType();

    switch (MOTy) {
    ...
    case MachineOperand::MO_GlobalAddress:
        return LowerSymbolOperand(MO, MOTy, offset);
    ...
}

```

[LLVMBackendTutorialExampleCode/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.

LLVMBackendTutorialExampleCode/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 ³ 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".

6.2 Array and struct support

LLVM use getelementptr to represent the array and struct type in C. Please reference section getelementptr of ⁴. For ch6_2.cpp, the llvm IR as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch6_2.cpp

```

struct Date
{
    int year;
    int month;
    int day;
};

```

³ <http://llvm.org/docs/WritingAnLLVMBBackend.html>

⁴ <http://llvm.org/docs/LangRef.html>

```

Date date = {2012, 10, 12};
int a[3] = {2012, 10, 12};

int main()
{
    int day = date.day;
    int i = a[1];

    return 0;
}

// ch6_2.ll
; ModuleID = 'ch6_2.bc'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-
f32:32:32-f64:32:64-v64:64:64-v128:128:128-a0:0:64-f80:128:128-n8:16:32-S128"
target triple = "i386-apple-macosx10.8.0"

%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 ch6_2.bc on static mode will get the incorrect asm file as follows,

```

118-165-66-82:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_
debug_build/bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=asm
ch6_2.bc -o ch6_2.cpu0.static.s
118-165-66-82:InputFiles Jonathan$ cat ch6_2.cpu0.static.s
.section .mdebug.abi32
.previous
.file "ch6_2.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 $2, $zero, %hi(date)
shl $2, $2, 16
addiu $2, $2, %lo(date)
ld $2, 0($2) // the correct one is ld $2, 8($2)
st $2, 8($sp)
addiu $2, $zero, %hi(a)
shl $2, $2, 16
addiu $2, $2, %lo(a)
ld $2, 0($2)
st $2, 4($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

```

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>
0x7f7f5b02d910: i32, ch = load 0x7f7f5b02d910, 0x7f7f5b030010, 0x7f7f5b02d210
<LD4[getelementptr inbounds ([3 x i32]* @a, i32 0, i32 1)]> [ORD=6]

...

```

By `l1c -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 `l1c -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;
}

```

LLVMBackendTutorialExampleCode/Chapter6_2/Cpu0ISelLowering.cpp

```

bool
Cpu0TargetLowering::isOffsetFoldingLegal (const GlobalAddressSDNode *GA) const {
    // The Cpu0 target isn't yet aware of offsets.
    return false;
}

```

Beyond that, we need to add the following code fragment to Cpu0ISelDAGToDAG.cpp,

LLVMBackendTutorialExampleCode/Chapter6_2/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.

LLVMBackendTutorialExampleCode/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.

Chapter6_2/ include these changes as above, you can run it with `ch6_2.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)
```

6.3 Type of char, short int and bool

To support signed/unsigned char and short int, we add the following code to Chapter6_3/.

LLVMBackendTutorialExampleCode/Chapter6_3/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 Chapter6_3/ with `ch6_3.cpp` will get the following result.

LLVMBackendTutorialExampleCode/InputFiles/ch6_3.cpp

```
struct Date
{
    short year;
    char month;
    char day;
    char hour;
    char minute;
    char second;
};
```

```

unsigned char b[4] = {'a', 'b', 'c', '\0'};

int main()
{
    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 -c ch6_3.cpp -emit-llvm -o ch6_3.bc
118-165-64-245:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch6_3.bc -o
ch6_3.cpu0.s
118-165-64-245:InputFiles Jonathan$ cat ch6_3.cpu0.s
    .section .mdebug.abi32
    .previous
    .file   "ch6_3.bc"
    .text
    .globl  main
    .align  2
    .type   main,@function
    .ent    main          # @_main
main:
    .cfi_startproc
    .frame  $sp,32,$lr
    .mask   0x00000000,0
    .set    noreorder
    .cupload $6
    .set    nomacro
# BB#0:
    addiu  $sp, $sp, -32
$tmp1:
    .cfi_def_cfa_offset 32
    addiu  $2, $zero, 0
    st     $2, 28($sp)
    ld     $3, %got(b)($gp)
    lbu   $4, 1($3)
    sb     $4, 24($sp)
    lbu   $3, 1($3)
    sb     $3, 20($sp)
    ld     $3, %got($_ZZ4mainE5date1)($gp)
    addiu $3, $3, %lo($_ZZ4mainE5date1)
    lhu   $4, 4($3)
    shl   $4, $4, 16
    lhu   $5, 6($3)
    or    $4, $4, $5
    st     $4, 12($sp)           // store hour, minute and second on 12($sp)
    lhu   $4, 2($3)
    lhu   $3, 0($3)
    shl   $3, $3, 16
    or    $3, $3, $4
    st     $3, 8($sp)           // store year, month and day on 8($sp)
    lbu   $3, 10($sp)          // m = date1.month;
    sb     $3, 4($sp)

```

```

        lbu    $3, 14($sp)           // s = date1.second;
        sb    $3, 0($sp)
        addiu $sp, $sp, 32
        ret   $lr
        .set  macro
        .set  reorder
        .end  main

$tmp2:
        .size  main, ($tmp2)-main
        .cfi_endproc

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

        .type  $_ZZ4mainE5date1,@object # @_ZZ4mainE5date1
        .section .rodata.cst8,"aM",@progbits,8
        .align 1
$_ZZ4mainE5date1:
        .2byte 2012                # 0x7dc
        .byte 11                   # 0xb
        .byte 25                   # 0x19
        .byte 9                    # 0x9
        .byte 40                   # 0x28
        .byte 15                   # 0xf
        .space 1
        .size  $_ZZ4mainE5date1, 8
    
```

To support load bool type, the following code added.

LLVMBackendTutorialExampleCode/Chapter6_3/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 ch6_3_2.ll still works as below. The IR input file ch6_3_2.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;
    }
```

LLVMBackendTutorialExampleCode/InputFiles/ch6_3_2.ll

```
define zeroext i1 @verify_load_bool() #0 {
entry:
    %retval = alloca i1, 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 ch6_3_2.ll -o -

    .section .mdebug.abi32
    .previous
    .file "ch6_4.ll"
    .text
    .globl verify_load_bool
    .align 2
    .type verify_load_bool,@function
    .ent verify_load_bool      # @verify_load_bool
verify_load_bool:
    .cfi_startproc
    .frame $sp,8,$lr
    .mask 0x00000000,0
    .set noreorder
    .set nomacro
# BB#0:                                # %entry
    addiu $sp, $sp, -8
$tmp1:
    .cfi_def_cfa_offset 8
    lbu $2, 7($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
```

LLVMBackendTutorialExampleCode/InputFiles/ch_run_backend.cpp

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

    if (a < 0)
        return false;

    return true;
}
```


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.

7.1 Control flow statement

Run ch7_1_1.cpp with clang will get result as follows,

[LLVMBackendTutorialExampleCode/InputFiles/ch7_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 `(br cond (%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,

LLVMBackendTutorialExampleCode/Chapter7_1/Cpu0InstrInfo.td

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

// Arbitrary immediates
def : Pat<(i32 imm:$imm),
      (OR (SHL (ADDiu ZERO, (HI16 imm:$imm)), 16),
       (ADDiu ZERO, (LO16 imm:$imm)))>;
```

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

LLVMBackendTutorialExampleCode/Chapter7_1/Cpu0InstrInfo.td

```
def brtarget : Operand<OtherVT> {
    let EncoderMethod = "getBranchTargetOpValue";
    let OperandType = "OPERAND_PCREL";
    let DecoderMethod = "DecodeBranchTarget";
}

...
// 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,

LLVMBackendTutorialExampleCode/Chapter7_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 7.1. 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,

LLVMBackendTutorialExampleCode/Chapter7_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

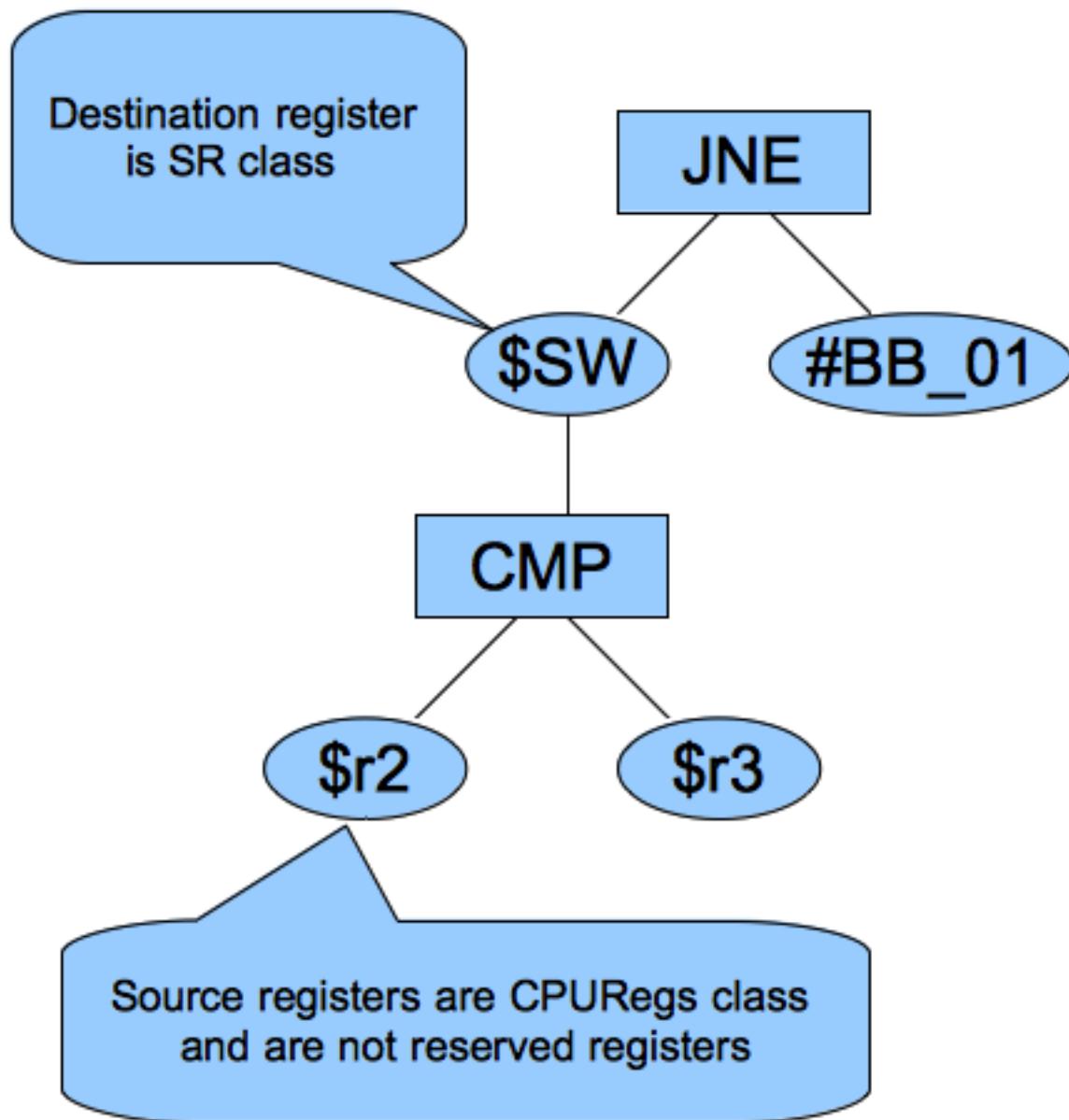


Figure 7.1: JNE (CMP \$r2, \$r3),

Register Class. As comment, the only possibility in (DestReg==SW, SrcReg==CPU0Regs) is “cmp \$SW, \$ZERO, \$rc”.

LLVMBackendTutorialExampleCode/Chapter7_1/Cpu0RegisterInfo.td

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

def CPUREgs : RegisterClass<"Cpu0", [i32], 32, (add
    // Return Values and Arguments
    V0, V1, A0, A1,
    // Not preserved across procedure calls
    T9,
    // Callee save
    S0, S1, S2,
    // Reserved
    ZERO, AT, GP, FP, SP, LR, PC)>;
...
// Status Registers
def SR : RegisterClass<"Cpu0", [i32], 32, (add SW)>;
```

LLVMBackendTutorialExampleCode/Chapter7_1/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.
        ...
        else if (SrcReg == Cpu0::SW) // add $ra, $ZERO, $SW
            Opc = Cpu0::ADD, ZeroReg = Cpu0::ZERO;
    }
    else if (Cpu0::CPUREgsRegClass.contains(SrcReg)) { // Copy from CPU Reg.
        ...
        // Only possibility in (DestReg==SW, SrcReg==CPU0Regs) is
        // cmp $SW, $ZERO, $rc
        else if (DestReg == Cpu0::SW)
            Opc = Cpu0::CMP, ZeroReg = Cpu0::ZERO;
    }
}
```

Chapter7_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 hexdump as follows,

```
118-165-79-206:InputFiles Jonathan$ cat ch7_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
ch7_1_1.bc -o ch7_1_1.cpu0.o

118-165-79-206:InputFiles Jonathan$ hexdump ch7_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,

LLVMBackendTutorialExampleCode/Chapter7_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```

}

/// getBranchTargetOpValue - Return binary encoding of the branch
/// target operand. If the machine operand requires relocation,
/// record the relocation and return zero.
unsigned Cpu0MCCodeEmitter::
getBranchTargetOpValue(const MCInst &MI, unsigned OpNo,
    SmallVectorImpl<MCFixup> &Fixups) const {

```

```

const MCOperand &MO = MI.getOperand(OpNo);
assert(MO.isExpr() && "getBranchTargetOpValue expects only expressions");

const MCExpr *Expr = MO.getExpr();
Fixups.push_back(MCFixup::Create(0, Expr,
                                  MCFixupKind(Cpu0::fixup_Cpu0_PC24)));
return 0;
}

```

LLVMBackendTutorialExampleCode/Chapter7_1/Cpu0MCInstLower.cpp

```

MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,
                                              MachineOperandType MOTy,
                                              unsigned Offset) const {
    ...
    switch (MO.getTargetFlags()) {
    default: llvm_unreachable("Invalid target flag!");
    case Cpu0II::MO_NO_FLAG: Kind = MCSymbolRefExpr::VK_None; break;
    ...
    ...
    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) {
    default: llvm_unreachable("unknown operand type");
    case MachineOperand::MO_Register:
    ...
    case MachineOperand::MO_MachineBasicBlock:
    case MachineOperand::MO_GlobalAddress:
    case MachineOperand::MO_BlockAddress:
    ...
    }
}

```

LLVMBackendTutorialExampleCode/Chapter7_1/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 {
  if (Cpu0::CPURegsRegClass.contains(DestReg)) { // Copy to CPU Reg.
  ...
  else if (SrcReg == Cpu0::SW) // add $ra, $ZERO, $SW
    Opc = Cpu0::ADD, ZeroReg = Cpu0::ZERO;
  }
  else if (Cpu0::CPURegsRegClass.contains(SrcReg)) { // Copy from CPU Reg.
  ...
  // Only possibility in (DestReg==SW, SrcReg==CPU0Regs) is
  // cmp $SW, $ZERO, $src
  else if (DestReg == Cpu0::SW)
    Opc = Cpu0::CMP, ZeroReg = Cpu0::ZERO;
  }
  ...
}

```

LLVMBackendTutorialExampleCode/Chapter7_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);
  ...
}

```

LLVMBackendTutorialExampleCode/Chapter7_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;
}

```

LLVMBackendTutorialExampleCode/Chapter7_1/Cpu0InstrInfo.td

```

// Cpu0InstrInfo.td
// Instruction operand types
def brtarget : Operand<OtherVT> {
    let EncoderMethod = "getBranchTargetOpValue";
    let OperandType = "OPERAND_PCREL";
    let DecoderMethod = "DecodeBranchTarget";
}
...

/// Conditional Branch
class CBranch<bits<8> op, string instr_asm, RegisterClass RC,
              list<Register> UseRegs>:
    FJ<op, (outs), (ins RC:$ra, brtarget:$addr),
        !strconcat(instr_asm, "\t$addr"),
        [(brcond RC:$ra, bb:$addr)], IIBranch> {
        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;
            let DecoderMethod = "DecodeJumpRelativeTarget";
        }
    ...

    /// 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 (setult RC:$lhs, RC:$rhs)), bb:$dst),
                  (JGTOp (CMPOp RC:$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 ch7_1_2.cpp is for “**nest if**” test. The ch7_1_3.cpp is the “**for loop**” as well as “**while loop**”, “**continue**”, “**break**”, “**goto**” test. The ch7_1_6.cpp is for “**goto**” test. You can run with them if you like to test more.

Finally, Chapter7_1/ support the local array definition by add the LowerCall() empty function in Cpu0ISelLowering.cpp as follows,

LLVMBackendTutorialExampleCode/Chapter7_1/Cpu0ISelLowering.cpp

```

// Cpu0ISelLowering.cpp
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                               SmallVectorImpl<SDValue> &InVals) const {
    return CLI.Chain;
}

```

With this LowerCall(), it can translate ch7_1_4.cpp, ch7_1_4.bc to ch7_1_4.cpu0.s as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch7_1_4.cpp

```

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

    return 0;
}

; ModuleID = 'ch7_1_4 .bc'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:32:64-
f32:32:32-f64:32:64-v64:64:64-v128:128:128-a0:0:64-f80:128:128-n8:16:32-S128"
target triple = "i386-apple-macosx10.8.0"

 @_ZZ4mainEla = private unnamed_addr constant [3 x i32] [i32 0, i32 1, i32 2],
align 4

define i32 @main() nounwind ssp {

```

```

entry:
%retval = alloca i32, align 4
%a = alloca [3 x i32], align 4
store i32 0, i32* %retval
%0 = bitcast [3 x i32]* %a to i8*
call void @llvm.memcpy.p0i8.p0i8.i32(i8* %0, i8* bitcast ([3 x i32]*
 @_ZZ4mainEla to i8*), i32 12, i32 4, i1 false)
ret i32 0
}

118-165-79-206:InputFiles Jonathan$ cat ch7_1_4.cpu0.s
.section .mdebug.abi32
.previous
.file "ch7_1_4.bc"
.text
.globl main
.align 2
.type main,@function
.ent main # @main
main:
.frame $sp,24,$lr
.mask 0x00000000,0
.set noreorder
.cupload $t9
.set nomacro
# BB#0: # %entry
addiu $sp, $sp, -24
ld $2, %got(__stack_chk_guard) ($gp)
ld $3, 0($2)
st $3, 20($sp)
addiu $3, $zero, 0
st $3, 16($sp)
ld $3, %got($_ZZ4mainEla) ($gp)
addiu $3, $3, %lo($_ZZ4mainEla)
ld $4, 8($3)
st $4, 12($sp)
ld $4, 4($3)
st $4, 8($sp)
ld $3, 0($3)
st $3, 4($sp)
ld $2, 0($2)
ld $3, 20($sp)
cmp $2, $3
jne $BB0_2
jmp $BB0_1
$BB0_1: # %SP_return
addiu $sp, $sp, 24
ret $lr
$BB0_2: # %CallStackCheckFailBlk
.set macro
.set reorder
.end main
$tmp1:
.size main, ($tmp1)-main
.type $_ZZ4mainEla,@object # @_ZZ4mainEla
.section .rodata,"a",@progbits
.align 2

```

```

$_ZZ4mainE1a:
    .4byte 0          # 0x0
    .4byte 1          # 0x1
    .4byte 2          # 0x2
    .size  $_ZZ4mainE1a, 12

```

The ch7_1_5.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,

[LLVMBackendTutorialExampleCode/InputFiles/ch7_1_5.cpp](#)

```

int main()
{
    unsigned int a = 0;
    int b = 1;
    int c = 2;

    if ((a == 0 && b == 2) || (c != 2)) {
        a++;
    }

    return 0;
}

```

```

118-165-78-230:InputFiles Jonathan$ clang -c ch7_1_5.cpp -emit-llvm -o ch7_1_5.bc
118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch7_1_5.bc -o
ch7_1_5.cpu0.s
118-165-78-230:InputFiles Jonathan$ cat ch7_1_5.cpu0.s
.section .mdebug.abi32
.previous
.file "ch7_1_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 $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:
    ld $3, 4($sp)           // a == 0
    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:
    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
```

7.2 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”¹, “Run Time Memory” of compiler book, or “Function Call Sequence” and “Stack Frame” of Mips ABI.

8.1 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 8.1 is the Mips stack frame.

Run `llc -march=mips` for `ch8_1.bc`, you will get the following result. See comment “//”.

LLVMBackendTutorialExampleCode/InputFiles/ch8_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()
```

¹ <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	
\$sp	+0	argument build area	Current
			<i>Low addresses</i>

Figure 8.1: 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 -c ch8_1.cpp -emit-llvm -o ch8_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 ch8_1.bc -o  
ch8_1.mips.s  
118-165-78-230:InputFiles Jonathan$ cat ch8_1.mips.s  
.section .mdebug.abi32  
.previous  
.file "ch8_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  

```

```

$tmp2:
.size _Z5sum_iiiiiii, ($tmp2)-_Z5sum_iiiiiii
.cfi_endproc

.globl main
.align 2
.type main,@function
.set nomips16           # @main
.ent main
main:
.cfi_startproc
.frame $sp,40,$ra
.mask 0x80000000,-4
.fmask 0x00000000,0
.set noreorder
.set nomacro
.set noat
# BB#0:
lui $2, %hi(_gp_disp)
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 $25, %call16(_Z5sum_iiiiiii)($gp)
addiu $4, $zero, 1 // Pass argument 1 to $4 (=a0)
addiu $5, $zero, 2 // Pass argument 2 to $5 (=a1)
addiu $t9, $zero, 3
jalr $25
addiu $7, $zero, 4
sw $2, 28($sp)
lw $ra, 36($sp)          # 4-byte Folded Reload
jr $ra
addiu $sp, $sp, 40
.set at
.set macro
.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 8.2 is the arguments location for example code ch8_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.

The 007-2418-003.pdf in ² is the Mips assembly language manual. ³ is Mips Application Binary Interface which

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

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

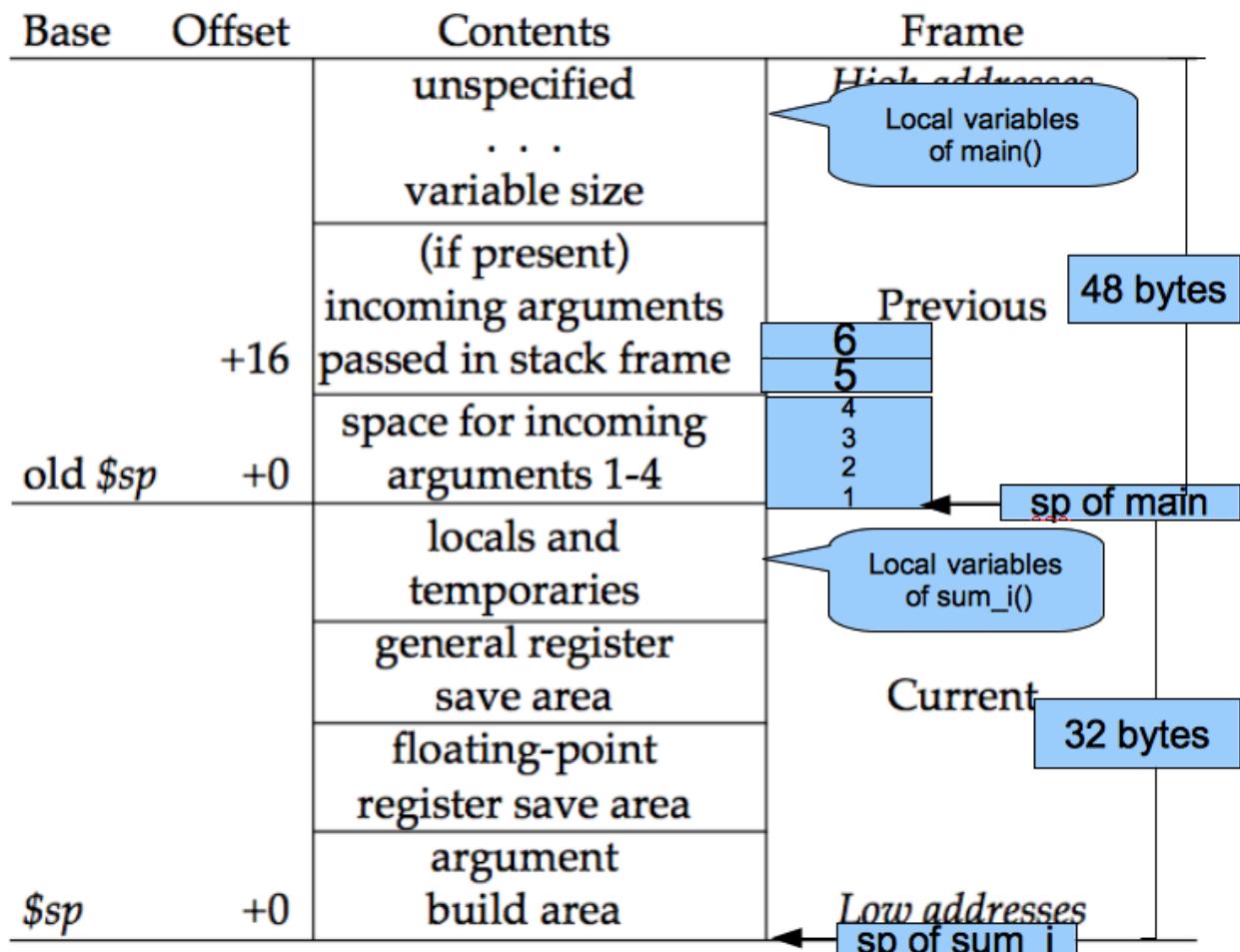


Figure 8.2: Mips arguments location in stack frame

include the Figure 8.1.

8.2 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 Chapter7_1/ with ch8_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
ch8_1.bc -o ch8_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 ch8_1.bc -o
ch8_1.cpu0.s
1. Running pass 'Function Pass Manager' on module 'ch8_1.bc'.
2. Running pass 'CPU0 DAG->DAG Pattern Instruction Selection' on function
'@_Z5sum_iiiiiii'
Illegal instruction: 4
```

Since Chapter7_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,

LLVMBackendTutorialExampleCode/Chapter8_1/Cpu0CallingConv.td

```
def RetCC_Cpu0EABI : CallingConv<[
  // i32 are returned in registers V0, V1
  CCIfType<i32>, CCAssignToReg<[V0, V1]>>
]>;
//=====//
// Cpu0 EABI Calling Convention
//=====//

def CC_Cpu0EABI : CallingConv<[
  // Promote i8/i16 arguments to i32.
  CCIfType<i8, i16>, CCPromoteToType<i32>>,
  // Integer values get stored in stack slots that are 4 bytes in
  // size and 4-byte aligned.
  CCIfType<i32>, CCAssignToStack<4, 4>>
]>;
//=====//
// Cpu0 Calling Convention Dispatch
//=====//

def CC_Cpu0 : CallingConv<[
  CCDelegateTo<CC_Cpu0EABI>
]>;
```

```

def RetCC_Cpu0 : CallingConv<[
    CCDelegateTo<RetCC_Cpu0EABI>
]>;
def CSR_O32 : CalleeSavedRegs<(add LR, FP,
                               (sequence "S%u", 2, 0))>;

```

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"⁴ 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,

LLVMBackendTutorialExampleCode/Chapter8_1/Cpu0ISelLowering.cpp

```

}

/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool isVarArg,
                                         const SmallVectorImpl<ISD::InputArg> &Ins,
                                         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 acumulate 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;

```

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

```

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;
}

//=====//

```

Refresh “section Global variable”⁵, 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 `ch8_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 `ch8_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.

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

LLVMBackendTutorialExampleCode/Chapter8_1/Cpu0InstrInfo.cpp

```

}

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

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

void Cpu0InstrInfo::loadRegFromStackSlot(MachineBasicBlock &MBB, MachineBasicBlock::iterator I,
                                         unsigned DestReg, int FI,
                                         const TargetRegisterClass *RC,
                                         const TargetRegisterInfo *TRI) const
{
    DebugLoc DL;
    if (I != MBB.end()) DL = I->getDebugLoc();
    MachineMemOperand *MMO = GetMemOperand(MBB, FI, MachineMemOperand::MOLoad);
    unsigned Opc = 0;

    if (Cpu0::CPURegsRegClass.hasSubClassEq(RC))
        Opc = Cpu0::LD;
    assert(Opc && "Register class not handled!");
    BuildMI(MBB, I, DL, get(Opc), DestReg).addFrameIndex(FI).addImm(0)
        .addMemOperand(MMO);
}

```

In addition to Calling Convention and LowerFormalArguments(), Chapter8_1/ add the following code for cpu0 instructions **swi** (Software Interrupt), **jsub** and **jalr** (function call) definition and printing.

LLVMBackendTutorialExampleCode/Chapter8_1/Cpu0InstrFormats.td

```

// Cpu0 Pseudo Instructions Format
class Cpu0Pseudo<dag outs, dag ins, string asmstr, list<dag> pattern>:
    Cpu0Inst<outs, ins, asmstr, pattern, IIPseudo, Pseudo> {
    let isCodeGenOnly = 1;
    let isPseudo = 1;
}

```

LLVMBackendTutorialExampleCode/Chapter8_1/Cpu0InstrInfo.td

```

def SDT_Cpu0JmpLink      : SDTypeProfile<0, 1, [SDTCisVT<0, iPTR]>;
...
// Call
def Cpu0JmpLink : SDNode<"Cpu0ISD::JmpLink", SDT_Cpu0JmpLink,
    [SDNPHasChain, SDNPOutGlue, SDNPOptInGlue,
     SDNPVariadic]>;
...
def jumptarget : Operand<OtherVT> {
    let EncoderMethod = "getJumpTargetOpValue";
}

```

```
}

...
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 IRET : JumpFR<0x2d, "iret", CPURegs>;
def JALR : JumpLinkReg<0x2e, "jalr", CPURegs>;
...
def : Pat<(Cpu0JmpLink (i32 tglobaladdr:$dst)),
       (JSUB tglobaladdr:$dst)>;
def : Pat<(Cpu0JmpLink (i32 texternalsym:$dst)),
       (JSUB texternalsym:$dst)>;
...
...
```

LLVMBackendTutorialExampleCode/Chapter8_1/Cpu0InstPrinter.cpp

```
static void printExpr(const MCExpr *Expr, raw_ostream &OS) {
    switch (Kind) {
    ...
    case MCSymbolRefExpr::VK_Cpu0_GOT_CALL: OS << "%call124("; break;
    ...
    }
}

...
```

LLVMBackendTutorialExampleCode/Chapter8_1/Cpu0MCInstLower.cpp

```
MCOperand Cpu0MCInstLower::LowerSymbolOperand(const MachineOperand &MO,
                                              MachineOperandType MOTy,
                                              unsigned Offset) const {
    ...
    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);
        ...
    }
    ...
}

```

LLVMBackendTutorialExampleCode/Chapter8_1/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```

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_CALL24;
            break;
        ...
    }
    ...
}

```

LLVMBackendTutorialExampleCode/Chapter8_1/Cpu0MachineFunction.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, intint GPFI; // Index of the frame object for restoring $gp
    mutable int DynAllocFI; // Frame index of dynamically allocated stack area.
    unsigned MaxCallFrameSize;

```

```

public:
  Cpu0FunctionInfo(MachineFunction& MF)
  : MF(MF), GlobalBaseReg(0),
    VarArgsFrameIndex(0), InArgFIRange(std::make_pair(-1, 0)),
    OutArgFIRange(std::make_pair(-1, 0)), GPFI(0), DynAllocFI(0),
    MaxCallFrameSize(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 needGPSSaveRestore() 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; }

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

```

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.

LLVMBackendTutorialExampleCode/Chapter8_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 ch8_1_2.cpp is an example to generate “memcpy” function call. It will be shown in next section of Chapter8_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):$dst)
/*710*/
    /*SwitchOpcode*/ 11, TARGET_VAL(ISD::TargetExternalSymbol), // ->721
    OPC_CheckType, MVT::i32,
/*712*/
    OPC_MoveParent,
/*713*/
    OPC_EmitMergeInputChains1_0,
/*714*/
    OPC_MorphNodeTo, TARGET_VAL(Cpu0::JSUB), 0|OPFL_Chain|
OPFL_GlueInput|OPFL_GlueOutput|OPFL_Variadic1,
    0/*#VTS*/, 1/*#Ops*/, 1,
    // Src: (Cpu0JmpLink (texternalsym:i32):$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 Chapter8_1/ with ch8_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
ch8_1.bc -o ch8_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 ch8_1.bc -o
ch8_1.cpu0.s
1. Running pass 'Function Pass Manager' on module 'ch8_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!

8.3 Store outgoing arguments to stack frame

Figure 8.2 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,

LLVMBackendTutorialExampleCode/Chapter8_2/Cpu0ISelLowering.cpp

```
SDValue
Cpu0TargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
                               SmallVectorImpl<SDValue> &InVals) const {
    SelectionDAG &DAG = CLI.DAG;
    DebugLoc &dl = CLI.DL;
    SmallVector<ISD::OutputArg, 32> &Outs = CLI.Outs;
    SmallVector<SDValue, 32> &OutVals = CLI.OutVals;
    SmallVector<ISD::InputArg, 32> &Ins = CLI.Ins;
    SDValue InChain = CLI.Chain;
    SDValue Callee = CLI.Callee;
    bool &isTailCall = CLI.IsTailCall;
    CallingConv::ID CallConv = CLI.CallConv;
    bool isVarArg = CLI.IsVarArg;

    MachineFunction &MF = DAG.getMachineFunction();
    MachineFrameInfo *MFI = MF.getFrameInfo();
    const TargetFrameLowering *TFL = MF.getTarget().getFrameLowering();
    bool IsPIC = getTargetMachine().getRelocationModel() == Reloc::PIC_;
    Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

    // Analyze operands of the call, assigning locations to each operand.
    SmallVector<CCValAssign, 16> ArgLocs;
    CCState CCInfo(CallConv, isVarArg, DAG.getMachineFunction(),
                   getTargetMachine(), ArgLocs, *DAG.getContext());
```

```

CCInfo.AnalyzeCallOperands(Outs, CC_Cpu0);

// Get a count of how many bytes are to be pushed on the stack.
unsigned NextStackOffset = CCInfo.getNextStackOffset();

// If this is the first call, create a stack frame object that points to
// a location to which .cprestore saves $gp.
if (IsPIC && Cpu0FI->globalBaseRegFixed() && !Cpu0FI->getGPFI())
    Cpu0FI->setGPF(MFI->CreateFixedObject(4, 0, true));
// Get the frame index of the stack frame object that points to the location
// of dynamically allocated area on the stack.
int DynAllocFI = Cpu0FI->getDynAllocFI();
unsigned MaxCallFrameSize = Cpu0FI->getMaxCallFrameSize();

if (MaxCallFrameSize < NextStackOffset) {
    Cpu0FI->setMaxCallFrameSize(NextStackOffset);

    // Set the offsets relative to $sp of the $gp restore slot and dynamically
    // allocated stack space. These offsets must be aligned to a boundary
    // determined by the stack alignment of the ABI.
    unsigned StackAlignment = TFL->getStackAlignment();
    NextStackOffset = (NextStackOffset + StackAlignment - 1) /
        StackAlignment * StackAlignment;

    MFI->setObjectOffset(DynAllocFI, NextStackOffset);
}
// Chain is the output chain of the last Load/Store or CopyToReg node.
// ByValChain is the output chain of the last Memcpy node created for copying
// byval arguments to the stack.
SDValue Chain, CallSeqStart, ByValChain;
SDValue NextStackOffsetVal = DAG.getIntPtrConstant(NextStackOffset, true);
Chain = CallSeqStart = DAG.getCALLSEQ_START(InChain, NextStackOffsetVal);
ByValChain = InChain;

// With EABI is it possible to have 16 args on registers.
SmallVector<std::pair<unsigned, SDValue>, 16> RegsToPass;
SmallVector<SDValue, 8> MemOpChains;

int FirstFI = -MFI->getNumFixedObjects() - 1, LastFI = 0;

// Walk the register/memloc assignments, inserting copies/loads.
for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
    SDValue Arg = OutVals[i];
    CCValAssign &VA = ArgLocs[i];
    MVT ValVT = VA.getValVT(), LocVT = VA.getLocVT();
    ISD::ArgFlagsTy Flags = Outs[i].Flags;

    // ByVal Arg.
    if (Flags.isByVal()) {
        assert("!!!Error!!!, Flags.isByVal()==true");
        assert(Flags.getByValSize() &&
               "ByVal args of size 0 should have been ignored by front-end.");
        continue;
    }

    // Promote the value if needed.
    switch (VA.getLocInfo()) {
        default: llvm_unreachable("Unknown loc info!");
    }
}

```

```

case CCValAssign::Full:
break;
case CCValAssign::SExt:
    Arg = DAG.getNode(ISD::SIGN_EXTEND, dl, LocVT, Arg);
break;
case CCValAssign::ZExt:
    Arg = DAG.getNode(ISD::ZERO_EXTEND, dl, LocVT, Arg);
break;
case CCValAssign::AExt:
    Arg = DAG.getNode(ISD::ANY_EXTEND, dl, LocVT, Arg);
break;
}
// Arguments that can be passed on register must be kept at
// RegsToPass vector
if (VA.isRegLoc()) {
    RegsToPass.push_back(std::make_pair(VA.getLocReg(), Arg));
    continue;
}

// Register can't get to this point...
assert(VA.isMemLoc());

// Create the frame index object for this incoming parameter
LastFI = MFI->CreateFixedObject(ValVT.getSizeInBits()/8,
                                  VA.getLocMemOffset(), true);
SDValue PtrOff = DAG.getFrameIndex(LastFI, getPointerTy());

// emit ISD::STORE whichs stores the
// parameter value to a stack Location
MemOpChains.push_back(DAG.getStore(Chain, dl, Arg, PtrOff,
                                    MachinePointerInfo(), false, false, 0));
}

// Extend range of indices of frame objects for outgoing arguments that were
// created during this function call. Skip this step if no such objects were
// created.
if (LastFI)
    Cpu0FI->extendOutArgFIRange(FirstFI, LastFI);

// If a memcpy has been created to copy a byval arg to a stack, replace the
// chain input of CallSeqStart with ByValChain.
if (InChain != ByValChain)
    DAG.UpdateNodeOperands(CallSeqStart.getNode(), ByValChain,
                           NextStackOffsetVal);

// Transform all store nodes into one single node because all store
// nodes are independent of each other.
if (!MemOpChains.empty())
    Chain = DAG.getNode(ISD::TokenFactor, dl, MVT::Other,
                        &MemOpChains[0], MemOpChains.size());

// If the callee is a GlobalAddress/ExternalSymbol node (quite common, every
// direct call is) turn it into a TargetGlobalAddress/TargetExternalSymbol
// node so that legalize doesn't hack it.
unsigned char OpFlag;
bool IsPICCall = IsPIC; // true if calls are translated to jalr $25
bool GlobalOrExternal = false;
SDValue CalleeLo;

```

```

if (GlobalAddressSDNode *G = dyn_cast<GlobalAddressSDNode>(Callee)) {
    OpFlag = IsPICCall ? Cpu0II::MO_GOT_CALL : Cpu0II::MO_NO_FLAG;
    Callee = DAG.getTargetGlobalAddress(G->getGlobal(), dl,
                                         getPointerTy(), 0, OpFlag);
    GlobalOrExternal = true;
}
else if (ExternalSymbolSDNode *S = dyn_cast<ExternalSymbolSDNode>(Callee)) {
    if (!IsPIC) // static
        OpFlag = Cpu0II::MO_NO_FLAG;
    else // O32 & PIC
        OpFlag = Cpu0II::MO_GOT_CALL;
    Callee = DAG.getTargetExternalSymbol(S->getSymbol(), getPointerTy(),
                                         OpFlag);
    GlobalOrExternal = true;
}

SDValue InFlag;

// Create nodes that load address of callee and copy it to T9
if (IsPICCall) {
    if (GlobalOrExternal) {
        // Load callee address
        Callee = DAG.getNode(Cpu0ISD::Wrapper, dl, getPointerTy(),
                             getGlobalReg(DAG, getPointerTy()), Callee);
        SDValue LoadValue = DAG.getLoad(getPointerTy(), dl, DAG.getEntryNode(),
                                         Callee, MachinePointerInfo::getGOT(),
                                         false, false, false, 0);

        // Use GOT+LO if callee has internal linkage.
        if (CalleeLo.getNode()) {
            SDValue Lo = DAG.getNode(Cpu0ISD::Lo, dl, getPointerTy(), CalleeLo);
            Callee = DAG.getNode(ISD::ADD, dl, getPointerTy(), LoadValue, Lo);
        } else
            Callee = LoadValue;
    }
}

// T9 should contain the address of the callee function if
// -relocation-model=pic or it is an indirect call.
if (IsPICCall || !GlobalOrExternal) {
    // copy to T9
    unsigned T9Reg = Cpu0::T9;
    Chain = DAG.getCopyToReg(Chain, dl, T9Reg, Callee, SDValue(0, 0));
    InFlag = Chain.getValue(1);
    Callee = DAG.getRegister(T9Reg, getPointerTy());
}

// Cpu0JmpLink = #chain, #target_address, #opt_in_flags...
//
// Returns a chain & a flag for retval copy to use.
SDVTList NodeTys = DAG.getVTList(MVT::Other, MVT::Glue);
SmallVector<SDValue, 8> Ops;
Ops.push_back(Chain);
Ops.push_back(Callee);

// Add argument registers to the end of the list so that they are
// known live into the call.

```

```

for (unsigned i = 0, e = RegsToPass.size(); i != e; ++i)
    Ops.push_back(DAG.getRegister(RegsToPass[i].first,
                                RegsToPass[i].second.getValueType()));

    // Add a register mask operand representing the call-preserved registers.
const TargetRegisterInfo *TRI = getTargetMachine().getRegisterInfo();
const uint32_t *Mask = TRI->getCallPreservedMask(CallConv);
assert(Mask && "Missing call preserved mask for calling convention");
Ops.push_back(DAG.getRegisterMask(Mask));

if (InFlag.getNode())
    Ops.push_back(InFlag);

Chain = DAG.getNode(Cpu0ISD::JmpLink, dl, NodeTys, &Ops[0], Ops.size());
InFlag = Chain.getValue(1);

    // Create the CALLSEQ_END node.
Chain = DAG.getCALLSEQ_END(Chain,
                           DAG.getIntPtrConstant(NextStackOffset, true),
                           DAG.getIntPtrConstant(0, true), InFlag);
InFlag = Chain.getValue(1);

    // Handle result values, copying them out of physregs into vregs that we
    // return.
return LowerCallResult(Chain, InFlag, CallConv, isVarArg,
                      Ins, dl, DAG, InVals);
}

/// LowerCallResult - Lower the result values of a call into the
/// appropriate copies out of appropriate physical registers.
SDValue
Cpu0TargetLowering::LowerCallResult(SDValue Chain, SDValue InFlag,
                                    CallingConv::ID CallConv, bool isVarArg,
                                    const SmallVectorImpl<ISD::InputArg> &Ins,
                                    DebugLoc dl, SelectionDAG &DAG,
                                    SmallVectorImpl<SDValue> &InVals) const {
    // Assign locations to each value returned by this call.
    SmallVector<CCValAssign, 16> RVLocs;
    CCState CCInfo(CallConv, isVarArg, DAG.getMachineFunction(),
                   getTargetMachine(), RVLocs, *DAG.getContext());

    CCInfo.AnalyzeCallResult(Ins, RetCC_Cpu0);

    // Copy all of the result registers out of their specified physreg.
    for (unsigned i = 0; i != RVLocs.size(); ++i) {
        Chain = DAG.getCopyFromReg(Chain, dl, RVLocs[i].getLocReg(),
                                   RVLocs[i].getValVT(), InFlag).getValue(1);
        InFlag = Chain.getValue(2);
        InVals.push_back(Chain.getValue(0));
    }

    return Chain;
}

```

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, %call24(_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.

LLVMBackendTutorialExampleCode/Chapter8_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.

LLVMBackendTutorialExampleCode/Chapter8_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::MOSStore);

    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);
}

```

Now, let's run Chapter8_2/ with ch8_1.cpp to get result as follows (see comment //),

```
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.bc -o
ch8_1.cpu0.s
118-165-78-230:InputFiles Jonathan$ cat ch8_1.cpu0.s
    .section .mdebug.abi32
    .previous
    .file   "ch8_1.bc"
    .text
    .globl  _Z5sum_iiiiiii
    .align  2
    .type   _Z5sum_iiiiiii,@function
    .ent    _Z5sum_iiiiiii          # @_Z5sum_iiiiiii
_Z5sum_iiiiiii:
    .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_iiiiiii
$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    $t9, %call24(_Z5sum_iuiuii)($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 Chapter8_2 with ch8_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.

LLVMBackendTutorialExampleCode/InputFiles/ch8_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 content” as follows,

```
Jonathan@tekiiMac:~/InputFiles$ /Users/Jonathan/llvm/test/cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=asm ch8_1_2.bc -debug -o -
```

```
Initial selection DAG: BB#0 'main:entry'
SelectionDAG has 35 nodes:
...
0x7fd909030810: <multiple use>
0x7fd909030c10: i32 = Constant<1214606444> // 1214606444=0x48656c6c="Hell"
0x7fd909030910: <multiple use>
0x7fd90902d810: <multiple use>
0x7fd909030d10: ch = store 0x7fd909030810, 0x7fd909030c10, 0x7fd909030910,
0x7fd90902d810<ST4[%1]>
0x7fd909030810: <multiple use>
0x7fd909030e10: i16 = Constant<28416> // 28416=0x6f00="o\0"
...
0x7fd90902d810: <multiple use>
0x7fd909031210: ch = store 0x7fd909030810, 0x7fd909030e10, 0x7fd909031010,
0x7fd90902d810<ST2[%1+4] (align=4)>
...

```

8.4 Fix issues

Run Chapter8_2/ with ch6_2.cpp to get the incorrect main return (return register \$2 is not 0) as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch6_2.cpp

```
struct Date
{
    int year;
    int month;
    int day;
};
```

```
Date date = {2012, 10, 12};
int a[3] = {2012, 10, 12};

int main()
{
    int day = date.day;
    int i = a[1];

    return 0;
}

118-165-78-31:InputFiles Jonathan$ clang -c ch6_2.cpp -emit-llvm -o ch6_2.bc
118-165-78-31:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=asm ch6_2.bc -o
ch6_2.cpu0.static.s
118-165-78-31:InputFiles Jonathan$ cat ch6_2.cpu0.static.s
.section .mdebug.abi32
.previous
.file "ch6_2.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 $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
.set reorder
.end main
...
```

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.

8.4.1 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 Chapter8_3/ to set the caller outgoing arguments into spOffset(\$sp) (Chapter8_2/ set them to pOffset+stackSize(\$sp)).

[LLVMBackendTutorialExampleCode/Chapter8_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->isGPFIFI(FrameIndex) ||  
        Cpu0FI->isDynAllocFI(FrameIndex))  
        Offset = spOffset;  
    else  
        Offset = spOffset + (int64_t)stackSize;  
    Offset += MI.getOperand(i+1).getImm();  
    ...  
}
```

[LLVMBackendTutorialExampleCode/Chapter8_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)  
: MF(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 Chapter8_3/ with ch8_1.cpp will get the following result. It correct arguments offset in 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 ch8_1.bc -o
ch8_1.cpu0.s
118-165-78-230:InputFiles Jonathan$ cat ch8_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 $t9, %call24(_Z5sum_iiiiii)($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 8.1: 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

8.4.2 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,

LLVMBackendTutorialExampleCode/Chapter8_3/Cpu0InstrInfo.cpp

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

LLVMBackendTutorialExampleCode/Chapter8_3/Cpu0FrameLowering.h

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

LLVMBackendTutorialExampleCode/Chapter8_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 `Chapter8_3/` with `ch8_1.cpp` will these two Pseudo hook instructions.

8.4.3 Handle \$gp register in PIC addressing mode

In “section Global variable”⁵, 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 `Chapter8_3/` with `ch8_2.cpp` to get the following correct result. We putting the comments in the result for explanation.

```
118-165-78-230:InputFiles Jonathan$ cat ch8_1.cpu0.s
_Z5sum_iiiiiii:
...
.cupload $t9 // assign $gp = $t9 by loader when loader load re-entry
               // function (shared library) of _Z5sum_iiiiiii
.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_iiiiiii)
...
.ret    $lr
.set      macro
.set      reorder
.end     _Z5sum_iiiiiii
...
.ent    main                  # @main
main:
```

```

.cfi_startproc
...
.cupload $t9
.set nomacro
...
.cprestore 24 // save $gp to 24($sp)
addiu $2, $zero, 0
...
ld $t9, %call24(_Z5sum_iiiiii)($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 Chapter8_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.

LLVMBackendTutorialExampleCode/Chapter8_3/CMakeLists.txt

```

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

```

LLVMBackendTutorialExampleCode/Chapter8_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;
}
```

LLVMBackendTutorialExampleCode/Chapter8_3/Cpu0.h

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

LLVMBackendTutorialExampleCode/Chapter8_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);
}
}
```

LLVMBackendTutorialExampleCode/Chapter8_3/Cpu0InstrInfo.td

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

LLVMBackendTutorialExampleCode/Chapter8_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->getGPF() )
    ...
    if (MaxCallFrameSize < NextStackOffset) {
        ...
        if (Cpu0FI->needGPSaveRestore())
            MFI->setObjectOffset(Cpu0FI->getGPF(), NextStackOffset);
        }
        ...
    }
    ...
}
```

LLVMBackendTutorialExampleCode/Chapter8_3/Cpu0EmitGPRestore.cpp

```
===== Cpu0EmitGPRestore.cpp - Emit GP Restore Instruction =====
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// This pass emits instructions that restore $gp right
// after jalr instructions.
//
//=====

#define DEBUG_TYPE "emit-gp-restore"

#include "Cpu0.h"
#include "Cpu0TargetMachine.h"
#include "Cpu0MachineFunction.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/Target/TargetInstrInfo.h"
#include "llvm/ADT/Statistic.h"

using namespace llvm;

namespace {
    struct Inserter : public MachineFunctionPass {
        TargetMachine &TM;
        const TargetInstrInfo *TII;

        static char ID;
        Inserter(TargetMachine &tm)
```

```

        : MachineFunctionPass(ID), TM(tm), TII(tm.getInstrInfo()) { }

    virtual const char *getPassName() const {
        return "Cpu0 Emit GP Restore";
    }

    bool runOnMachineFunction(MachineFunction &F);
};

char Inserter::ID = 0;
} // end of anonymous namespace

bool Inserter::runOnMachineFunction(MachineFunction &F) {
    Cpu0FunctionInfo *Cpu0FI = F.getInfo<Cpu0FunctionInfo>();

    if ((TM.getRelocationModel() != Reloc::PIC_) ||
        (!Cpu0FI->globalBaseRegFixed()))
        return false;

    bool Changed = false;
    int FI = Cpu0FI->getGPFIndex();

    for (MachineFunction::iterator MFI = F.begin(), MFE = F.end();
        MFI != MFE; ++MFI) {
        MachineBasicBlock& MBB = *MFI;
        MachineBasicBlock::iterator I = MFI->begin();

        /// IsLandingPad - Indicate that this basic block is entered via an
/// exception handler.
/// If MBB is a landing pad, insert instruction that restores $gp after
/// EH_LABEL.
        if (MBB.isLandingPad()) {
            // Find EH_LABEL first.
            for (; I->getOpcode() != TargetOpcode::EH_LABEL; ++I) {

                // Insert ld.
                ++I;
                DebugLoc dl = I != MBB.end() ? I->getDebugLoc() : DebugLoc();
                BuildMI(MBB, I, dl, TII->get(Cpu0::LD), Cpu0::GP).addFrameIndex(FI)
                    .addImm(0);
                Changed = true;
            }

            while (I != MFI->end()) {
                if (I->getOpcode() != Cpu0::JALR) {
                    ++I;
                    continue;
                }

                DebugLoc dl = I->getDebugLoc();
                // emit lw $gp, ($gp save slot on stack) after jalr
                BuildMI(MBB, ++I, dl, TII->get(Cpu0::LD), Cpu0::GP).addFrameIndex(FI)
                    .addImm(0);
                Changed = true;
            }
        }

        return Changed;
    }
}

```

```
/// createCpu0EmitGPRestorePass - Returns a pass that emits instructions that
/// restores $gp clobbered by jalr instructions.
FunctionPass *llvm::createCpu0EmitGPRestorePass(Cpu0TargetMachine &tm) {
    return new Inserter(tm);
}
```

LLVMBackendTutorialExampleCode/Chapter8_3/Cpu0AsmPrinter.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();
    MCInst TmpInst0;
    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;
}

MCInstLowering.Lower(MI, TmpInst0);
OutStreamer.EmitInstruction(TmpInst0);
```

}

LLVMBackendTutorialExampleCode/Chapter8_3/Cpu0MCInstLower.cpp

```

void Cpu0MCInstLower::LowerCPRESTORE(int64_t Offset,
                                      SmallVector<MCInst, 4>& MCInsts) {
    assert(isInt<32>(Offset) && (Offset >= 0) &&
           "Imm operand of .cprestore must be a non-negative 32-bit value.");

    MCOperand SPReg = MCOperand::CreateReg(Cpu0::SP), BaseReg = SPReg;
    MCOperand GPReg = MCOperand::CreateReg(Cpu0::GP);
    MCOperand ZEROReg = MCOperand::CreateReg(Cpu0::ZERO);

    if (!isInt<16>(Offset)) {
        unsigned Hi = ((Offset + 0x8000) >> 16) & 0xffff;
        Offset &= 0xffff;
        MCOperand ATReg = MCOperand::CreateReg(Cpu0::AT);
        BaseReg = ATReg;

        // lui at,hi
        // add at,at,sp
        MCInsts.resize(2);
        CreateMCInst(MCInsts[0], Cpu0::LUI, ATReg, ZEROReg, MCOperand::CreateImm(Hi));
        CreateMCInst(MCInsts[1], Cpu0::ADD, ATReg, ATReg, SPReg);
    }

    MCInst St;
    CreateMCInst(St, Cpu0::ST, GPReg, BaseReg, MCOperand::CreateImm(Offset));
    MCInsts.push_back(St);
}

```

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 ch8_1.bc -o ch8_1.cpu0.o
118-165-76-131:InputFiles Jonathan$ hexdump ch8_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 ch8_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 ch8_1.bc -o ch8_1.cpu0.s
118-165-76-131:InputFiles Jonathan$ cat ch8_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 ch8_1_3.cpp and ch8_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
```

8.4.4 Correct the return of main()

The LowerReturn() modified in Chapter8_3/ as follows,

LLVMBackendTutorialExampleCode/Chapter8_3/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 {

    // CCValAssign - represent the assignment of
    // the return value to a location
    SmallVector<CCValAssign, 16> RVLocs;

    // CCState - Info about the registers and stack slot.
    CCState CCInfo(CallConv, isVarArg, DAG.getMachineFunction(),
                   getTargetMachine(), RVLocs, *DAG.getContext());

    // Analyze return values.
    CCInfo.AnalyzeReturn(Outs, RetCC_Cpu0);

    SDValue Flag;
    SmallVector<SDValue, 4> RetOps(1, Chain);

    // Copy the result values into the output registers.
    for (unsigned i = 0; i != RVLocs.size(); ++i) {
        CCValAssign &VA = RVLocs[i];
        assert(VA.isRegLoc() && "Can only return in registers!");

        Chain = DAG.getCopyToReg(Chain, dl, VA.getLocReg(), OutVals[i], Flag);

        // Guarantee that all emitted copies are stuck together with flags.
    }
}
```

```

    Flag = Chain.getValue(1);
    RetOps.push_back(DAG.getRegister(VA.getLocReg(), VA.getLocVT()));
}

RetOps[0] = Chain; // Update chain.

// Add the flag if we have it.
if (Flag.getNode())
    RetOps.push_back(Flag);

// Return on Cpu0 is always a "ret $lr"
return DAG.getNode(Cpu0ISD::Ret, dl, MVT::Other, &RetOps[0], RetOps.size());
}

bool

```

LLVMBackendTutorialExampleCode/Chapter8_3/Cpu0InstrInfo.h

```

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

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

```

LLVMBackendTutorialExampleCode/Chapter8_3/Cpu0InstrInfo.cpp

```

/// Expand Pseudo instructions into real backend instructions
bool Cpu0InstrInfo::expandPostRAPpseudo(MachineBasicBlock::iterator MI) const {
    MachineBasicBlock &MBB = *MI->getParent();

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

    MBB.erase(MI);
    return true;
}

void Cpu0InstrInfo::ExpandRetLR(MachineBasicBlock &MBB,
                                  MachineBasicBlock::iterator I,
                                  unsigned Opc) const {
    BuildMI(MBB, I, I->getDebugLoc(), get(Opc)).addReg(Cpu0::LR);
}

```

LLVMBackendTutorialExampleCode/Chapter8_3/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,

LLVMBackendTutorialExampleCode/Chapter8_3/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 Chapter8_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 Chapter8_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>
0x1ea1c50: i32 = FrameIndex<2> [ORD=7]

0x1e9f120: <multiple use>
0x1ea1d50: ch = store 0x1e9fd20:1, 0x1e9fd20, 0x1ea1c50,
```

```

0x1e9f120<ST4[%i]> [ORD=7]

0x1ea1e50: <multiple use>
0x1e9ef20: <multiple use>
0x1ea1f50: ch,glue = CopyToReg 0x1ea1d50, 0x1ea1e50, 0x1e9ef20

0x1ea1f50: <multiple use>
0x1ea1e50: <multiple use>
0x1ea1f50: <multiple use>
0x1ea2050: ch = Cpu0ISD::Ret 0x1ea1f50, 0x1ea1e50, 0x1ea1f50:1

```

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 Chapter8_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.

LLVMBackendTutorialExampleCode/Chapter8_3/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<0x2C, "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 ch6_2.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 8.2: 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 Chapter8_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 ch6_2.bc -o
ch6_2.cpu0.static.s
118-165-78-31:InputFiles Jonathan$ cat ch6_2.cpu0.static.s
.section .mdebug.abi32
.previous
.file "ch6_2.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

```

8.5 Support features

This section support features of struct type, variable number of arguments and dynamic stack allocation.

Run Chapter8_3 with ch8_2_1.cpp will get the error message as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch8_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 -c ch8_2_1.cpp -emit-llvm -o
ch8_2_1.bc
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch8_2_1.bc -o ch8_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 Chapter8_3/ with ch8_3.cpp to get the following error,

LLVMBackendTutorialExampleCode/InputFiles/ch8_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 `llvm-config --host-target` -c
ch8_3.cpp -emit-llvm -o ch8_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 ch8_3.bc -o
ch8_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
```

LLVMBackendTutorialExampleCode/InputFiles/ch8_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 main()
{
    int a = weight_sum(1, 2, 3, 4, 5, 6);

    return a;
}
```

Run Chapter8_3 with ch8_4.cpp will get the following error.

```
118-165-72-242:InputFiles Jonathan$ clang -I/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.8.sdk/usr/include/-c ch8_4.cpp -emit-llvm -o ch8_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 ch8_4.bc -o ch8_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_iiiiii
```

8.5.1 Structure type support

Chapter8_4/ with the following code added to support the structure type in function call.

LLVMBackendTutorialExampleCode/Chapter8_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(ByVersionAlign, (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);
    }
}
...
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,

LLVMBackendTutorialExampleCode/Chapter8_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 Chapter8_4/ with ch8_2_1.cpp and explain with this example.

```
JonathantekiiMac:InputFiles Jonathan$ cat ch8_2_1.cpu0.s  
.section .mdebug.abi32  
.previous  
.file "ch8_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 $t9, %call24(_Z7getDatev)($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 $t9, %call24(_Z8copyDate4Date)($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 $t9, %call24(_Z8copyDateP4Date)($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  $t9, %call124(_Z8copyTime4Time) ($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  $t9, %call124(_Z8copyTimeP4Time) ($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 ch8_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 $t9, %call124(_Z8copyDate4Date) ($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 ⁶. 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,

⁶ <http://developer.mips.com/clang-llvm/>

LLVMBackendTutorialExampleCode/Chapter8_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 $t9, %call24(_Z8copyDate4Date)($gp)

```

LLVMBackendTutorialExampleCode/Chapter8_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 ch8_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

```

8.5.2 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 Chapter8_4/ with ch8_3.cpp as well as clang option, `clang -target 'llvm-config --host-target'`, to get the following result,

```

118-165-76-131:InputFiles Jonathan$ clang -target 'llvm-config --host-target' -c
ch8_3.cpp -emit-llvm -o ch8_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
ch8_3.bc -o ch8_3.cpu0.s
118-165-76-131:InputFiles Jonathan$ cat ch8_3.cpu0.s
.section .mdebug.abi32
.previous
.file "ch8_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  $t9, %call24(_Z5sum_iiz)($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 ch8_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 Chapter8_4/. The ch8_3_2.cpp is C++ template example code, it can be translated into cpu0 backend code too.

LLVMBackendTutorialExampleCode/Chapter8_4/Cpu0TargetLowering.h

```

class Cpu0TargetLowering : public TargetLowering {
...
private:
...
SDValue LowerVASTART(SDValue Op, SelectionDAG &DAG) const;
...
}

```

LLVMBackendTutorialExampleCode/Chapter8_4/Cpu0TargetLowering.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());

    // vastart just stores the address of the VarArgsFrameIndex slot into the
    // memory location argument.
const Value *SV = cast<SrcValueSDNode>(Op.getOperand(2))->getValue();
return DAG.getStore(Op.getOperand(0), dl, FI, Op.getOperand(1),
                    MachinePointerInfo(SV), false, false, 0);
}

...
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 RegSize = Cpu0::CPURegsRegClass.getSize();
    // Offset of the first variable argument from stack pointer.
    int FirstVaArgOffset = RegSize;

    // 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);
}
...
}
}

```

LLVMBackendTutorialExampleCode/InputFiles/ch8_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 ⁷, 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.

8.5.3 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.

Chapter8_4 support dynamic stack allocation with the following code added.

LLVMBackendTutorialExampleCode/Chapter8_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));
    }
    ...
}

void Cpu0FrameLowering::emitEpilogue(MachineFunction &MF,
                                      MachineBasicBlock &MBB) const {
    ...

```

⁷ section “4.5.1 Calling Conventions” of tricore_llvm.pdf

```
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);
}
...
}
```

LLVMBackendTutorialExampleCode/Chapter8_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);
...
}
```

LLVMBackendTutorialExampleCode/Chapter8_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 Chapter8_4 with ch8_4.cpp will get the following correct result.

```
118-165-72-242:InputFiles Jonathan$ clang -I/Applications/Xcode.app/Contents/
Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.8.sdk/usr/include/
-c ch8_4.cpp -emit-llvm -o ch8_4.bc
118-165-72-242:InputFiles Jonathan$ llvm-dis ch8_4.bc -o ch8_4.ll
118-165-72-242:InputFiles Jonathan$ cat ch8_4.ll
; ModuleID = 'ch8_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-a0:0:64:64-f80:128:128-n8:16:
```

```
32:64-S128"
target triple = "x86_64-apple-macosx10.8.0"

define i32 @_Z5sum_iiiiiii(i32 %x1, i32 %x2, i32 %x3, i32 %x4, i32 %x5, i32 %x6)
nounwind uwtable ssp {
    ...
    %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 ch8_4.bc -o
ch8_4.cpu0.s
118-165-72-242:InputFiles Jonathan$ cat ch8_4.cpu0.s
...
_Z10weight_sumiiiiii:
.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    $t9, %call24(_Z3sumiiiiii)($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    $t9, %call24(__stack_chk_fail)($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 8.3, 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 8.4 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**.

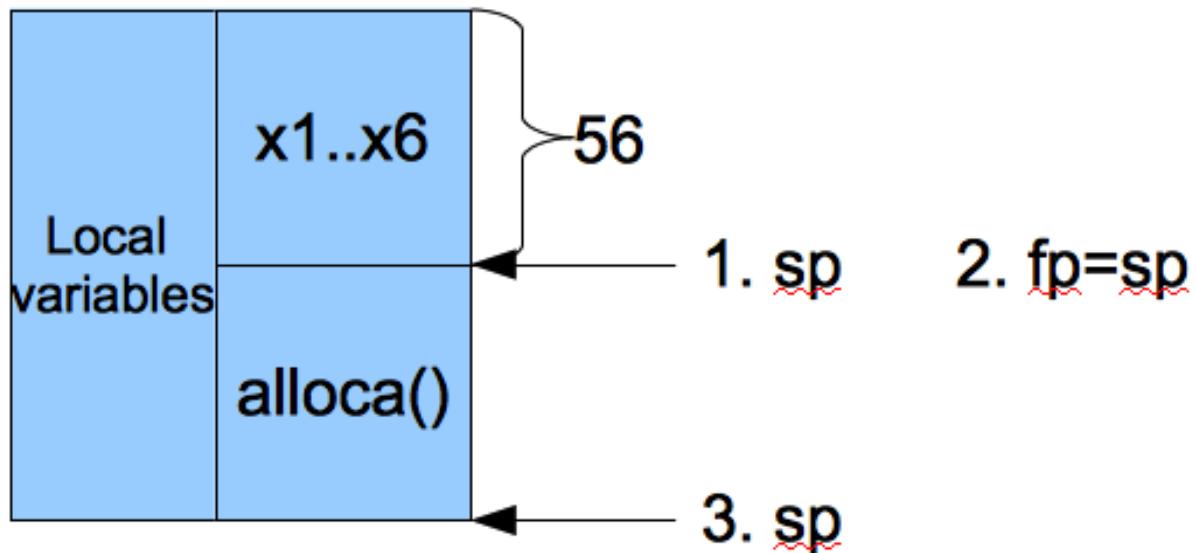


Figure 8.3: Frame pointer changes when enter function

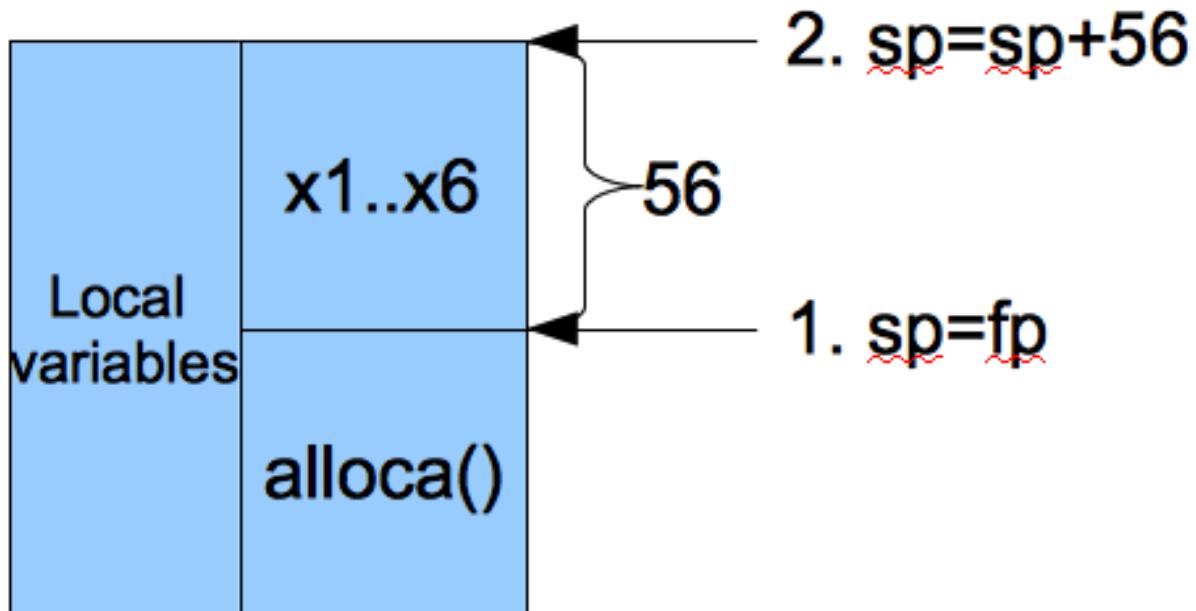


Figure 8.4: Stack pointer changes when exit function

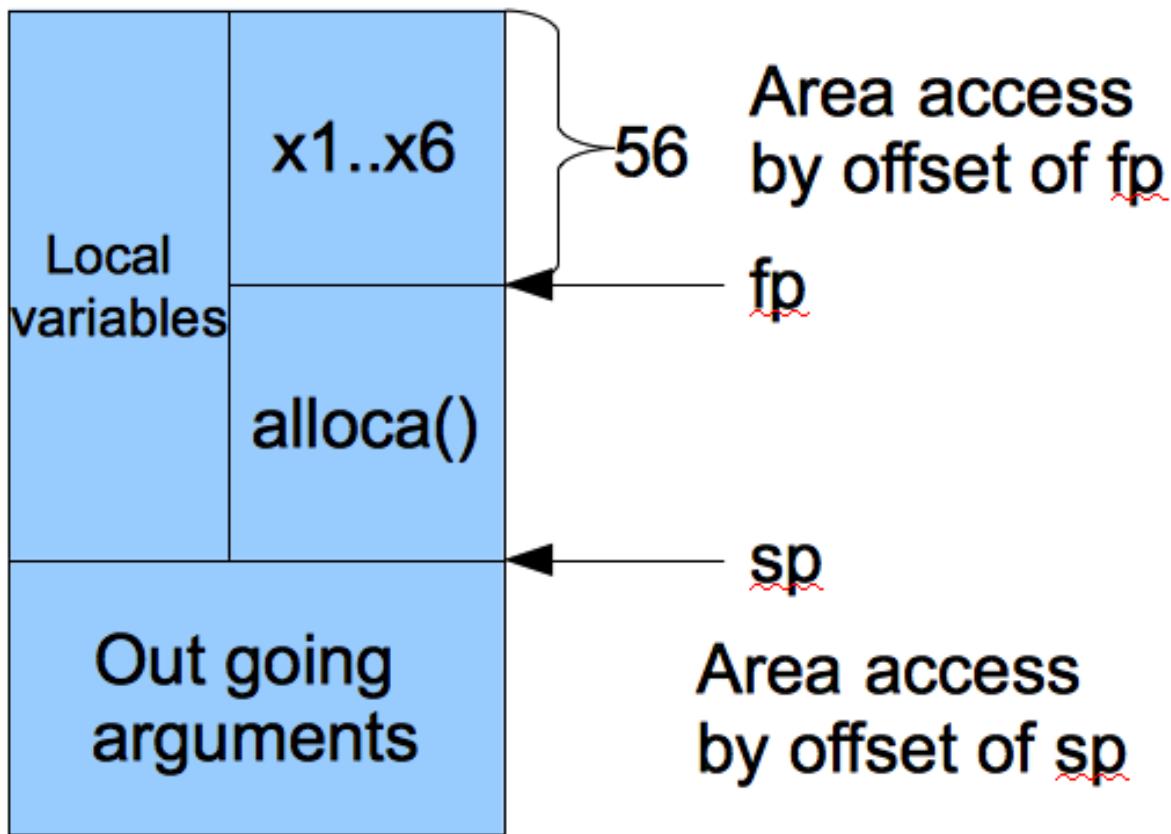


Figure 8.5: fp and sp access areas

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 8.5](#) 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 ch8_4.cpu0.s, it access local variable (above of alloca()) by fp offset and outgoing arguments (below of alloca()) by sp offset.

8.6 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 a 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.

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 ¹.

The binary encode of cpu0 instruction set in obj has been checked in the previous chapters. But we didn't dig into the ELF file format like elf header and relocation record at that time. This chapter will use the binutils which has been installed in “sub-section Install other tools on iMac” of Appendix A: “Installing LLVM” ² to 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 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”³ written by Beck is a famous book in concept of telling readers what is the compiler output, what is the linker output, what is the loader output, and how they work together. But it covers the concept only. You can reference it to understand how the “**Relocation Record**” works if you need to refresh or learning this knowledge for this chapter.

⁴, ⁵, ⁶ are the Chinese documents available from the cpu0 author on web site.

9.1 ELF format

ELF is a format used both in obj and executable file. So, there are two views in it as [Figure 9.1](#).

¹ http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

² <http://jonathan2251.github.com/lbd/install.html#install-other-tools-on-imac>

³ Leland Beck, System Software: An Introduction to Systems Programming.

⁴ <http://ccckmit.wikidot.com/lk:aout>

⁵ <http://ccckmit.wikidot.com/lk:objfile>

⁶ <http://ccckmit.wikidot.com/lk:elf>

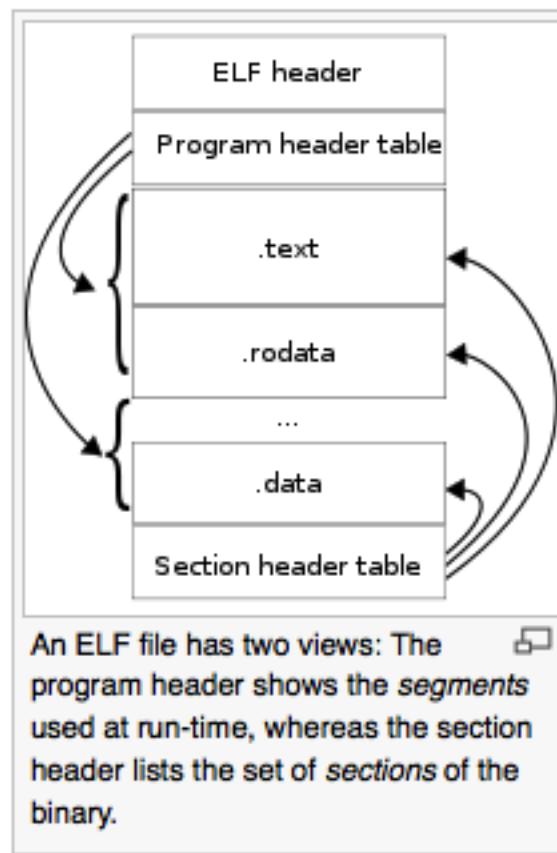


Figure 9.1: ELF file format overview

As Figure 9.1, 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, and sections is link time layout for code and data.

9.2 ELF header and Section header table

Let's run Chapter7_7/ 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
ELF Header:
  Magic: 7f 45 4c 46 01 02 01 08 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, big endian
  Version: 1 (current)
  OS/ABI: UNIX - IRIX
  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: 212 (bytes into file)
  Flags: 0x70000001
  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: 10
  Section header string table index: 7
```

```
[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 08 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, big endian
  Version: 1 (current)
  OS/ABI: UNIX - IRIX
  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: 212 (bytes into file)
  Flags: 0x70000001
  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: 11
Section header string table index: 8
[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] .null NULL 00000000 000000 000000 00 0 0 0 0
[ 1] .text PROGBITS 00000000 000034 000034 00 AX 0 0 4
[ 2] .rel.text REL 00000000 000310 000018 08 8 1 4
[ 3] .data PROGBITS 00000000 000068 000004 00 WA 0 0 4
[ 4] .bss NOBITS 00000000 00006c 000000 00 WA 0 0 4
[ 5] .eh_frame PROGBITS 00000000 00006c 000028 00 A 0 0 4
[ 6] .rel.eh_frame REL 00000000 000328 000008 08 8 5 4
[ 7] .shstrtab STRTAB 00000000 000094 00003e 00 0 0 1
[ 8] .symtab SYMTAB 00000000 000264 000090 10 9 6 4
[ 9] .strtab STRTAB 00000000 0002f4 00001b 00 0 0 1
```

Key to Flags:

```
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
```

```
[Gamma@localhost InputFiles]$
```

9.3 Relocation Record

The cpu0 backend translate global variable as follows,

```
[Gamma@localhost InputFiles]$ clang -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
.globl main
.align 2
.type main,@function
.ent main # @main
main:
.cfi_startproc
```

```

.frame $sp,8,$lr
.mask 0x00000000,0
.set noreorder
.cupload $t9
...
    ld $2, %got(gI)($gp)
...
.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 09a00000 1eaa0010 09aa0000 13aa6000  .....
...
0020 002a0000 00220000 012d0000 09dd0008  .*...."-.....
...
[Gamma@localhost InputFiles]$ Jonathan$


[Gamma@localhost InputFiles]$ readelf -tr ch6_1.cpu0.o
There are 10 section headers, starting at offset 0xd4:

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 000034 00      0      0      4
    [00000006]: ALLOC, EXEC
[ 2] .rel.text
    REL        00000000 000310 000018 08      8      1      4
    [00000000]:
[ 3] .data
    PROGBITS    00000000 000068 000004 00      0      0      4
    [00000003]: WRITE, ALLOC
[ 4] .bss
    NOBITS    00000000 00006c 000000 00      0      0      4
    [00000003]: WRITE, ALLOC
[ 5] .eh_frame
    PROGBITS    00000000 00006c 000028 00      0      0      4
    [00000002]: ALLOC
[ 6] .rel.eh_frame
    REL        00000000 000328 000008 08      8      5      4
    [00000000]:

```

```

[ 7] .shstrtab
    STRTAB      00000000 000094 00003e 00    0    0    1
    [00000000]:
[ 8] .symtab
    SYMTAB      00000000 000264 000090 10    9    6    4
    [00000000]:
[ 9] .strtab
    STRTAB      00000000 0002f4 00001b 00    0    0    1
    [00000000]:


Relocation section '.rel.text' at offset 0x310 contains 3 entries:
  Offset      Info      Type          Sym.Value  Sym. Name
00000000  00000805 unrecognized: 5      00000000  _gp_disp
00000008  00000806 unrecognized: 6      00000000  _gp_disp
00000020  00000609 unrecognized: 9      00000000  gI

Relocation section '.rel.eh_frame' at offset 0x328 contains 1 entries:
  Offset      Info      Type          Sym.Value  Sym. Name
0000001c  00000202 unrecognized: 2      00000000  .text
[Gamma@localhost InputFiles]$ readelf -tr ch6_1.mips.o
There are 10 section headers, starting at offset 0xd0:


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 000030 00    0    0    4
    [00000006]: ALLOC, EXEC
[ 2] .rel.text
    REL      00000000 00030c 000018 08    8    1    4
    [00000000]:
[ 3] .data
    PROGBITS  00000000 000064 000004 00    0    0    4
    [00000003]: WRITE, ALLOC
[ 4] .bss
    NOBITS   00000000 000068 000000 00    0    0    4
    [00000003]: WRITE, ALLOC
[ 5] .eh_frame
    PROGBITS  00000000 000068 000028 00    0    0    4
    [00000002]: ALLOC
[ 6] .rel.eh_frame
    REL      00000000 000324 000008 08    8    5    4
    [00000000]:
[ 7] .shstrtab
    STRTAB      00000000 000090 00003e 00    0    0    1
    [00000000]:
[ 8] .symtab
    SYMTAB      00000000 000260 000090 10    9    6    4
    [00000000]:
[ 9] .strtab
    STRTAB      00000000 0002f0 00001b 00    0    0    1
    [00000000]:


Relocation section '.rel.text' at offset 0x30c contains 3 entries:

```

```

Offset      Info      Type          Sym.Value  Sym. Name
00000000  00000805 R_MIPS_HI16    00000000  _gp_disp
00000004  00000806 R_MIPS_LO16    00000000  _gp_disp
00000018  00000609 R_MIPS_GOT16   00000000  gI

Relocation section '.rel.eh_frame' at offset 0x324 contains 1 entries:
Offset      Info      Type          Sym.Value  Sym. Name
0000001c  00000202 R_MIPS_32      00000000  .text

```

As depicted in section Handle \$gp register in PIC addressing mode, it translate “**.cupload %reg**” into the following.

```

// Lower ".cupload $reg" to
// "addiu $gp, $zero, %hi(_gp_disp)"
// "shl $gp, $gp, 16"
// "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 8 of .text section referred to `_gp_disp` value. The offset 0 and 8 of .text section are instructions “`addiu $gp, $zero, %hi(_gp_disp)`” and “`addiu $gp, $gp, %lo(_gp_disp)`” and their corresponding obj encode are 09a00000 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 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 greadelf utilitly. The instruction “**Id \$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. Loader will change this address too.

Run with ch8_3_3.cpp will get the unknown result in `_Z5sum_iiz` and other symbol reference as below. Loader or linker will take care them according the relocation records compiler generated.

```

[Gamma@localhost InputFiles]$ /usr/local/llvm/test/cmake_debug_build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=obj ch8_3_3.bc -o ch8_3_3.
cpu0.o
[Gamma@localhost InputFiles]$ readelf -tr ch8_3_3.cpu0.o
There are 11 section headers, starting at offset 0x248:

```

```

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  000178  00      0      0      4
  [00000006]: ALLOC, EXEC
[ 2] .rel.text
  REL        00000000  000538  000058  08      9      1      4
  [00000000]:
[ 3] .data
  PROGBITS    00000000  0001ac  000000  00      0      0      4
  [00000003]: WRITE, ALLOC
[ 4] .bss
  NOBITS     00000000  0001ac  000000  00      0      0      4
  [00000003]: WRITE, ALLOC
[ 5] .rodata.str1.1
  PROGBITS    00000000  0001ac  000008  01      0      0      1

```

```

[00000032]: ALLOC, MERGE, STRINGS
[ 6] .eh_frame
  PROGBITS        00000000 0001b4 000044 00  0  0  4
  [00000002]: ALLOC
[ 7] .rel.eh_frame
  REL            00000000 000590 000010 08  9  6  4
  [00000000]:
[ 8] .shstrtab
  STRTAB         00000000 0001f8 00004d 00  0  0  1
  [00000000]:
[ 9] .symtab
  SYMTAB         00000000 000400 0000e0 10  10  8  4
  [00000000]:
[10] .strtab
  STRTAB         00000000 0004e0 000055 00  0  0  1
  [00000000]:

```

Relocation section '.rel.text' at offset 0x538 contains 11 entries:

Offset	Info	Type	Sym.	Value	Sym.	Name
00000000	00000c05	unrecognized:	5	00000000	_gp_disp	
00000008	00000c06	unrecognized:	6	00000000	_gp_disp	
0000001c	00000b09	unrecognized:	9	00000000	_stack_chk_guard	
000000b8	00000b09	unrecognized:	9	00000000	_stack_chk_guard	
000000dc	00000a0b	unrecognized:	b	00000000	_stack_chk_fail	
000000e8	00000c05	unrecognized:	5	00000000	_gp_disp	
000000f0	00000c06	unrecognized:	6	00000000	_gp_disp	
00000140	0000080b	unrecognized:	b	00000000	_Z5sum_iiz	
00000154	00000209	unrecognized:	9	00000000	\$._str	
00000158	00000206	unrecognized:	6	00000000	\$._str	
00000160	00000d0b	unrecognized:	b	00000000	printf	

Relocation section '.rel.eh_frame' at offset 0x590 contains 2 entries:

Offset	Info	Type	Sym.	Value	Sym.	Name
0000001c	00000302	unrecognized:	2	00000000	.text	
00000034	00000302	unrecognized:	2	00000000	.text	

[Gamma@localhost InputFiles]\$ /usr/local/llvm/test/cmake_debug_build/bin/llc -march=mips -relocation-model=pic -filetype=obj ch8_3_3.bc -o ch8_3_3.mips.o

[Gamma@localhost InputFiles]\$ readelf -tr ch8_3_3.mips.o

There are 11 section headers, starting at offset 0x254:

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	000184	00	0	0	4
	[00000006]: ALLOC, EXEC								
[2]	.rel.text	REL	00000000	000544	000058	08	9	1	4
	[00000000]:								
[3]	.data	PROGBITS	00000000	0001b8	000000	00	0	0	4
	[00000003]: WRITE, ALLOC								
[4]	.bss								

```

NOBITS          00000000 0001b8 000000 00    0    0    4
[00000003]: WRITE, ALLOC
[ 5] .rodata.str1.1
PROGBITS       00000000 0001b8 000008 01    0    0    1
[00000032]: ALLOC, MERGE, STRINGS
[ 6] .eh_frame
PROGBITS       00000000 0001c0 000044 00    0    0    4
[00000002]: ALLOC
[ 7] .rel.eh_frame
REL            00000000 00059c 000010 08    9    6    4
[00000000]:
[ 8] .shstrtab
STRTAB         00000000 000204 00004d 00    0    0    1
[00000000]:
[ 9] .symtab
SYMTAB         00000000 00040c 0000e0 10   10    8    4
[00000000]:
[10] .strtab
STRTAB         00000000 0004ec 000055 00    0    0    1
[00000000]:


Relocation section '.rel.text' at offset 0x544 contains 11 entries:
Offset  Info  Type      Sym.Value  Sym. Name
00000000 00000c05 R_MIPS_HI16    00000000  __gp_disp
00000004 00000c06 R_MIPS_LO16    00000000  __gp_disp
00000024 00000b09 R_MIPS_GOT16   00000000  __stack_chk_guard
000000c8 00000b09 R_MIPS_GOT16   00000000  __stack_chk_guard
000000f0 00000a0b R_MIPS_CALL16  00000000  __stack_chk_fail
00000100 00000c05 R_MIPS_HI16    00000000  __gp_disp
00000104 00000c06 R_MIPS_LO16    00000000  __gp_disp
00000134 0000080b R_MIPS_CALL16  00000000  __Z5sum_iiz
00000154 00000209 R_MIPS_GOT16   00000000  $.str
00000158 00000206 R_MIPS_LO16    00000000  $.str
0000015c 00000d0b R_MIPS_CALL16  00000000  printf


Relocation section '.rel.eh_frame' at offset 0x59c contains 2 entries:
Offset  Info  Type      Sym.Value  Sym. Name
0000001c 00000302 R_MIPS_32     00000000  .text
00000034 00000302 R_MIPS_32     00000000  .text
[Gamma@localhost InputFiles]$
```

9.4 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 the instruction into obj format automatically.

9.5 lld

The lld is a project of LLVM linker. It's under development and we cannot finish the installation by following the web site direction. Even with this, it's really make sense to develop a new linker according lld web site information. Please visit the web site ⁷.

⁷ <http://lld.llvm.org/>

9.6 llvm-objdump

9.6.1 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 -c ch8_3_3.cpp -emit-llvm -I/
Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/
SDKs/MacOSX10.8.sdk/usr/include/ -o ch8_3_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 ch8_3_3.bc -o
ch8_3_3.cpu0.o

118-165-78-12:InputFiles Jonathan$ gobjdump -t -r ch8_3_3.cpu0.o

ch8_3_3.cpu0.o: file format elf32-big
```

SYMBOL TABLE:

00000000 1	df	*ABS*	00000000 ch8_3_3.bc
00000000 1	o	.rodata.str1.1	00000008 \$.str
00000000 1	d	.text	00000000 .text
00000000 1	d	.data	00000000 .data
00000000 1	d	.bss	00000000 .bss
00000000 1	d	.rodata.str1.1	00000000 .rodata.str1.1
00000000 1	d	.eh_frame	00000000 .eh_frame
00000000 g	F	.text	00000d4 _Z5sum_iiz
000000d4 g	F	.text	00000074 main
00000000	*	UND*	00000000 __stack_chk_fail
00000000	*	UND*	00000000 __stack_chk_guard
00000000	*	UND*	00000000 printf

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
00000008	UNKNOWN	__stack_chk_guard
00000010	UNKNOWN	__stack_chk_guard
000000d0	UNKNOWN	__stack_chk_fail
00000118	UNKNOWN	_Z5sum_iiz
00000124	UNKNOWN	\$.str
0000012c	UNKNOWN	\$.str
00000134	UNKNOWN	printf

RELOCATION RECORDS FOR [.eh_frame]:

OFFSET	TYPE	VALUE
0000001c	UNKNOWN	.text
00000034	UNKNOWN	.text

```
118-165-83-10:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llvm-objdump -t -r ch8_3_3.cpu0.o
```

```
ch8_3_3.cpu0.o: file format ELF32-CPU0
```

RELOCATION RECORDS FOR [.text]:

```

0 R_CPU0_HI16 _gp_disp
8 R_CPU0_LO16 _gp_disp
28 R_CPU0_GOT16 __stack_chk_guard
188 R_CPU0_GOT16 __stack_chk_guard
224 R_CPU0_CALL24 __stack_chk_fail
236 R_CPU0_HI16 _gp_disp
244 R_CPU0_LO16 _gp_disp
324 R_CPU0_CALL24 _Z5sum_iiz
344 R_CPU0_GOT16 $.str
348 R_CPU0_LO16 $.str
356 R_CPU0_CALL24 printf

```

RELOCATION RECORDS FOR [.eh_frame]:

```

28 R_CPU0_32 .text
52 R_CPU0_32 .text

```

SYMBOL TABLE:

```

00000000 1 df *ABS* 00000000 ch8_3_3.bc
00000000 1 .rodata.str1.1 00000008 $.str
00000000 1 d .text 00000000 .text
00000000 1 d .data 00000000 .data
00000000 1 d .bss 00000000 .bss
00000000 1 d .rodata.str1.1 00000000 .rodata.str1.1
00000000 1 d .eh_frame 00000000 .eh_frame
00000000 g F .text 000000ec _Z5sum_iiz
000000ec g F .text 00000094 main
00000000 *UND* 00000000 __stack_chk_fail
00000000 *UND* 00000000 __stack_chk_guard
00000000 *UND* 00000000 _gp_disp
00000000 *UND* 00000000 printf

```

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_CALL24);
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 -r
        res = symname;
        break;
    ...
}

template<support::endianness target_endianness, bool is64Bits>
StringRef ELFObjectFile<target_endianness, is64Bits>
    ::getFileFormatName() const {
    switch(Header->e_ident[ELF::EI_CLASS]) {
    case ELF::ELFCLASS32:
        switch(Header->e_machine) {
        ...
        case ELF::EM_CPU0: // llvm-objdump -t -r
            return "ELF32-CPU0";
        ...
    }
}

template<support::endianness target_endianness, bool is64Bits>
unsigned ELFObjectFile<target_endianness, is64Bits>::getArch() const {
    switch(Header->e_machine) {
    ...
    case ELF::EM_CPU0: // llvm-objdump -t -r
        return (target_endianness == support::little) ?
            Triple::cpu0el : Triple::cpu0;
    ...
}

```

9.6.2 llvm-objdump -d

Run Chapter8_9/ and command `llvm-objdump -d` for dump file from elf to hex as follows,

```

JonathanTekiiMac:InputFiles Jonathan$ clang -c ch7_1_1.cpp -emit-llvm -o
ch7_1_1.bc
JonathanTekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=obj ch7_1_1.bc
-o ch7_1_1.cpu0.o
JonathanTekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llvm-objdump -d ch7_1_1.cpu0.o

ch7_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 Chapter9_1/.

LLVMBackendTutorialExampleCode/Chapter9_1/CMakeLists.txt

```

tablegen(LLVM Cpu0GenDisassemblerTables.inc -gen-disassembler)
...

```

LLVMBackendTutorialExampleCode/Chapter9_1/LLVMBuild.txt

```
[common]
subdirectories = Disassembler ...
...
has_disassembler = 1
...
```

LLVMBackendTutorialExampleCode/Chapter9_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";
}

class CBranch<bits<8> op, string instr_asm, RegisterClass RC,
  list<Register> UseRegs>:
  FJ<op, (outs), (ins RC:$ra, brtarget:$addr),
  !strconcat(instr_asm, "\t$addr"),
  [], IIBranch> {
  ...
  let DecoderMethod = "DecodeBranchTarget";
}

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>;
```

LLVMBackendTutorialExampleCode/Chapter9_1/Disassembler/CMakeLists.txt

```
include_directories( ${CMAKE_CURRENT_BINARY_DIR}/.. ${CMAKE_CURRENT_SOURCE_DIR}/.. )

add_llvm_library(LLVMCpu0Disassembler
  Cpu0Disassembler.cpp
)
```

```
# workaround for hanging compilation on MSVC9 and 10
if( MSVC_VERSION EQUAL 1400 OR MSVC_VERSION EQUAL 1500 OR MSVC_VERSION EQUAL 1600 )
set_property(
  SOURCE Cpu0Disassembler.cpp
  PROPERTY COMPILE_FLAGS "/Od"
)
endif()

add_dependencies(LLVMCpu0Disassembler Cpu0CommonTableGen)
```

LLVMBackendTutorialExampleCode/Chapter9_1/Disassembler/LLVMBuild.txt

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

[component_0]
type = Library
name = Cpu0Disassembler
parent = Cpu0
required_libraries = MC Support Cpu0Info
add_to_library_groups = Cpu0

LLVMBackendTutorialExampleCode/Chapter9_1/Disassembler/Cpu0Disassembler.cpp

```
===== Cpu0Disassembler.cpp - Disassembler for Cpu0 -----*-- C++ -*====//  
//  
// The LLVM Compiler Infrastructure  
//  
// This file is distributed under the University of Illinois Open Source  
// License. See LICENSE.TXT for details.  
//  
//=====-----=====//  
//  
// This file is part of the Cpu0 Disassembler.  
//  
//=====-----=====//  
  
#include "Cpu0.h"  
#include "Cpu0Subtarget.h"  
#include "Cpu0RegisterInfo.h"  
#include "llvm/MC/MCDisassembler.h"
```

```

#include "llvm/MC/MCFixedLenDisassembler.h"
#include "llvm/Support/MemoryObject.h"
#include "llvm/Support/TargetRegistry.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/MC/MCInst.h"
#include "llvm/Support/MathExtras.h"

using namespace llvm;

typedef MCDisassembler::DecodeStatus DecodeStatus;

/// Cpu0Disassembler - a disassembler class for Cpu032.
class Cpu0Disassembler : public MCDisassembler {
public:
    /// Constructor      - Initializes the disassembler.
    ///
    Cpu0Disassembler(const MCSubtargetInfo &STI, bool bigEndian) :
        MCDisassembler(STI), isBigEndian(bigEndian) {
    }

    ~Cpu0Disassembler() {
    }

    /// getInstruction - See MCDisassembler.
    DecodeStatus getInstruction(MCInst &instr,
                                uint64_t &size,
                                const MemoryObject &region,
                                uint64_t address,
                                raw_ostream &vStream,
                                raw_ostream &cStream) const;

private:
    bool isBigEndian;
};

// Decoder tables for Cpu0 register
static const unsigned CPUREgsTable[] = {
    Cpu0::ZERO, Cpu0::AT, Cpu0::V0, Cpu0::V1,
    Cpu0::A0, Cpu0::A1, Cpu0::T9, Cpu0::S0,
    Cpu0::S1, Cpu0::S2, Cpu0::GP, Cpu0::FP,
    Cpu0::SW, Cpu0::SP, Cpu0::LR, Cpu0::PC
};

static DecodeStatus DecodeCPUREgsRegisterClass(MCInst &Inst,
                                              unsigned RegNo,
                                              uint64_t Address,
                                              const void *Decoder);

static DecodeStatus DecodeCMPInstruction(MCInst &Inst,
                                         unsigned Insn,
                                         uint64_t Address,
                                         const void *Decoder);

static DecodeStatus DecodeBranchTarget(MCInst &Inst,
                                       unsigned Insn,
                                       uint64_t Address,
                                       const void *Decoder);

static DecodeStatus DecodeJumpRelativeTarget(MCInst &Inst,
                                             unsigned Insn,
                                             uint64_t Address,
                                             const void *Decoder);

```

```

        const void *Decoder);
static DecodeStatus DecodeJumpAbsoluteTarget(MCInst &Inst,
                                              unsigned Insn,
                                              uint64_t Address,
                                              const void *Decoder);

static DecodeStatus DecodeMem(MCInst &Inst,
                               unsigned Insn,
                               uint64_t Address,
                               const void *Decoder);
static DecodeStatus DecodeSimm16(MCInst &Inst,
                                 unsigned Insn,
                                 uint64_t Address,
                                 const void *Decoder);

namespace llvm {
extern Target TheCpu0elTarget, TheCpu0Target, TheCpu064Target,
               TheCpu064elTarget;
}

static MCDisassembler *createCpu0Disassembler(
    const Target &T,
    const MCSubtargetInfo &STI) {
    return new Cpu0Disassembler(STI, true);
}

static MCDisassembler *createCpu0elDisassembler(
    const Target &T,
    const MCSubtargetInfo &STI) {
    return new Cpu0Disassembler(STI, false);
}

extern "C" void LLVMInitializeCpu0Disassembler() {
    // Register the disassembler.
    TargetRegistry::RegisterMCDisassembler(TheCpu0Target,
                                            createCpu0Disassembler);
    TargetRegistry::RegisterMCDisassembler(TheCpu0elTarget,
                                            createCpu0elDisassembler);
}

#include "Cpu0GenDisassemblerTables.inc"

/// readInstruction - read four bytes from the MemoryObject
/// and return 32 bit word sorted according to the given endianess
static DecodeStatus readInstruction32(const MemoryObject &region,
                                       uint64_t address,
                                       uint64_t &size,
                                       uint32_t &insn,
                                       bool isBigEndian) {
    uint8_t Bytes[4];

    // We want to read exactly 4 Bytes of data.
    if (region.readBytes(address, 4, (uint8_t*)Bytes, NULL) == -1) {
        size = 0;
        return MCDisassembler::Fail;
    }
}

```

```

if (isBigEndian) {
    // Encoded as a big-endian 32-bit word in the stream.
    insn = (Bytes[3] << 0) |
        (Bytes[2] << 8) |
        (Bytes[1] << 16) |
        (Bytes[0] << 24);
}
else {
    // Encoded as a small-endian 32-bit word in the stream.
    insn = (Bytes[0] << 0) |
        (Bytes[1] << 8) |
        (Bytes[2] << 16) |
        (Bytes[3] << 24);
}

return MCDisassembler::Success;
}

DecodeStatus
Cpu0Disassembler::getInstruction(MCInst &instr,
                                  uint64_t &Size,
                                  const MemoryObject &Region,
                                  uint64_t Address,
                                  raw_ostream &vStream,
                                  raw_ostream &cStream) const {
    uint32_t Insn;

    DecodeStatus Result = readInstruction32(Region, Address, Size,
                                             Insn, isBigEndian);
    if (Result == MCDisassembler::Fail)
        return MCDisassembler::Fail;

    // Calling the auto-generated decoder function.
    Result = decodeInstruction(DecoderTableCpu032, instr, Insn, Address,
                               this, STI);
    if (Result != MCDisassembler::Fail) {
        Size = 4;
        return Result;
    }

    return MCDisassembler::Fail;
}

static DecodeStatus DecodeCPURegsRegisterClass(MCInst &Inst,
                                                unsigned RegNo,
                                                uint64_t Address,
                                                const void *Decoder) {
    if (RegNo > 16)
        return MCDisassembler::Fail;

    Inst.addOperand(MCOperand::CreateReg(CPURegsTable[RegNo]));
    return MCDisassembler::Success;
}

static DecodeStatus DecodeMem(MCInst &Inst,
                               unsigned Insn,
                               uint64_t Address,
                               const void *Decoder) {

```

```

int Offset = SignExtend32<16>(Insn & 0xffff);
int Reg = (int)fieldFromInstruction(Insn, 20, 4);
int Base = (int)fieldFromInstruction(Insn, 16, 4);

Inst.addOperand(MCOperand::CreateReg(CPURegsTable[Reg]));
Inst.addOperand(MCOperand::CreateReg(CPURegsTable[Base]));
Inst.addOperand(MCOperand::CreateImm(Offset));

return MCDisassembler::Success;
}

/* CMP instruction define $rc and then $ra, $rb; The printOperand() print
operand 1 and operand 2 (operand 0 is $rc and operand 1 is $ra), so we Create
register $rc first and create $ra next, as follows,

// Cpu0InstrInfo.td
class CmpInstr<bits<8> op, string instr_asm,
          InstrItinClass itin, RegisterClass RC, RegisterClass RD, bit isComm = 0>:
FA<op, (outs RD:$rc), (ins RC:$ra, RC:$rb),
          !strconcat(instr_asm, "\t$ra, $rb"), [], itin> {

// Cpu0AsmWriter.inc
void Cpu0InstPrinter::printInstruction(const MCInst *MI, raw_ostream &O) {
...
    case 3:
        // CMP, JEQ, JGE, JGT, JLE, JLT, JNE
        printOperand(MI, 1, O);
        break;
...
    case 1:
        // CMP
        printOperand(MI, 2, O);
        return;
        break;
*/
static DecodeStatus DecodeCMPInstruction(MCInst &Inst,
                                         unsigned Insn,
                                         uint64_t Address,
                                         const void *Decoder) {
    int Reg_a = (int)fieldFromInstruction(Insn, 20, 4);
    int Reg_b = (int)fieldFromInstruction(Insn, 16, 4);
    int Reg_c = (int)fieldFromInstruction(Insn, 12, 4);

    Inst.addOperand(MCOperand::CreateReg(CPURegsTable[Reg_c]));
    Inst.addOperand(MCOperand::CreateReg(CPURegsTable[Reg_a]));
    Inst.addOperand(MCOperand::CreateReg(CPURegsTable[Reg_b]));
    return MCDisassembler::Success;
}

/* CBranch instruction define $ra and then imm24; The printOperand() print
operand 1 (operand 0 is $ra and operand 1 is imm24), so we Create register
operand first and create imm24 next, as follows,

// Cpu0InstrInfo.td
class CBranch<bits<8> op, string instr_asm, RegisterClass RC,
          list<Register> UseRegs>:
FJ<op, (outs), (ins RC:$ra, brtarget:$addr),
          !strconcat(instr_asm, "\t$addr"),

```

```

[(brcond RC:$ra, bb:$addr)], IIBranch> {

// Cpu0AsmWriter.inc
void Cpu0InstPrinter::printInstruction(const MCInst *MI, raw_ostream &O) {
...
case 3:
    // CMP, JEQ, JGE, JGT, JLE, JLT, JNE
    printOperand(MI, 1, O);
    break;
*/
static DecodeStatus DecodeBranchTarget (MCInst &Inst,
                                       unsigned Insn,
                                       uint64_t Address,
                                       const void *Decoder) {
int BranchOffset = fieldFromInstruction(Insn, 0, 24);
if (BranchOffset > 0x8fffff)
    BranchOffset = -1*(0x1000000 - BranchOffset);
Inst.addOperand(MCOperand::CreateReg(CPURegsTable[0]));
Inst.addOperand(MCOperand::CreateImm(BranchOffset));
return MCDisassembler::Success;
}

static DecodeStatus DecodeJumpRelativeTarget (MCInst &Inst,
                                             unsigned Insn,
                                             uint64_t Address,
                                             const void *Decoder) {

int JumpOffset = fieldFromInstruction(Insn, 0, 24);
if (JumpOffset > 0x8fffff)
    JumpOffset = -1*(0x1000000 - JumpOffset);
Inst.addOperand(MCOperand::CreateImm(JumpOffset));
return MCDisassembler::Success;
}

static DecodeStatus DecodeJumpAbsoluteTarget (MCInst &Inst,
                                             unsigned Insn,
                                             uint64_t Address,
                                             const void *Decoder) {

unsigned JumpOffset = fieldFromInstruction(Insn, 0, 24);
Inst.addOperand(MCOperand::CreateImm(JumpOffset));
return MCDisassembler::Success;
}

static DecodeStatus DecodeSimm16(MCInst &Inst,
                               unsigned Insn,
                               uint64_t Address,
                               const void *Decoder) {
Inst.addOperand(MCOperand::CreateImm(SignExtend32<16>(Insn)));
return MCDisassembler::Success;
}

```

As above code, it 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 Decoder-Method**” keyword to allow programmers implement their decode function. In Cpu0 example, we define function De-

codeCMPInstruction(), 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 `llvm-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 Chapter9_1/ with command `llvm-objdump -d ch7_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
ch7_1_1.bc -o ch7_1_1.cpu0.o
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llvm-objdump -d ch7_1_1.cpu0.o
```

```
ch7_1_1.cpu0.o:          file format ELF32-CPU0
```

Disassembly of section .text:

main:

0:	09 dd ff d8	addiu \$sp, \$sp, -40
4:	09 30 00 00	addiu \$3, \$zero, 0
8:	02 3d 00 24	st \$3, 36(\$sp)
c:	02 3d 00 20	st \$3, 32(\$sp)
10:	09 20 00 01	addiu \$2, \$zero, 1
14:	02 2d 00 1c	st \$2, 28(\$sp)
18:	09 40 00 02	addiu \$4, \$zero, 2
1c:	02 4d 00 18	st \$4, 24(\$sp)
20:	09 40 00 03	addiu \$4, \$zero, 3
24:	02 4d 00 14	st \$4, 20(\$sp)
28:	09 40 00 04	addiu \$4, \$zero, 4
2c:	02 4d 00 10	st \$4, 16(\$sp)
30:	09 40 00 05	addiu \$4, \$zero, 5
34:	02 4d 00 0c	st \$4, 12(\$sp)
38:	09 40 00 06	addiu \$4, \$zero, 6
3c:	02 4d 00 08	st \$4, 8(\$sp)
40:	09 40 00 07	addiu \$4, \$zero, 7
44:	02 4d 00 04	st \$4, 4(\$sp)
48:	09 40 00 08	addiu \$4, \$zero, 8
4c:	02 4d 00 00	st \$4, 0(\$sp)
50:	01 4d 00 20	ld \$4, 32(\$sp)
54:	28 40 00 0c	bne \$4, \$zero, 12
58:	01 4d 00 20	ld \$4, 32(\$sp)
5c:	09 44 00 01	addiu \$4, \$4, 1
60:	02 4d 00 20	st \$4, 32(\$sp)
64:	01 4d 00 1c	ld \$4, 28(\$sp)
68:	27 40 00 0c	beq \$4, \$zero, 12
6c:	01 4d 00 1c	ld \$4, 28(\$sp)
70:	09 44 00 01	addiu \$4, \$4, 1
74:	02 4d 00 1c	st \$4, 28(\$sp)
78:	01 4d 00 18	ld \$4, 24(\$sp)
7c:	0a 44 00 01	slti \$4, \$4, 1
80:	28 40 00 0c	bne \$4, \$zero, 12

84: 01 4d 00 18	ld \$4, 24(\$sp)
88: 09 44 00 01	addiu \$4, \$4, 1
8c: 02 4d 00 18	st \$4, 24(\$sp)
90: 01 4d 00 14	ld \$4, 20(\$sp)
94: 0a 44 00 00	slti \$4, \$4, 0
98: 28 40 00 0c	bne \$4, \$zero, 12
9c: 01 4d 00 14	ld \$4, 20(\$sp)
a0: 09 44 00 01	addiu \$4, \$4, 1
a4: 02 4d 00 14	st \$4, 20(\$sp)
a8: 01 4d 00 10	ld \$4, 16(\$sp)
ac: 09 50 ff ff	addiu \$5, \$zero, -1
b0: 20 45 40 00	slt \$4, \$5, \$4
b4: 28 40 00 0c	bne \$4, \$zero, 12
b8: 01 4d 00 10	ld \$4, 16(\$sp)
bc: 09 44 00 01	addiu \$4, \$4, 1
c0: 02 4d 00 10	st \$4, 16(\$sp)
c4: 01 4d 00 0c	ld \$4, 12(\$sp)
c8: 20 33 40 00	slt \$3, \$3, \$4
cc: 28 30 00 0c	bne \$3, \$zero, 12
d0: 01 3d 00 0c	ld \$3, 12(\$sp)
d4: 09 33 00 01	addiu \$3, \$3, 1
d8: 02 3d 00 0c	st \$3, 12(\$sp)
dc: 01 3d 00 08	ld \$3, 8(\$sp)
e0: 20 22 30 00	slt \$2, \$2, \$3
e4: 28 20 00 0c	bne \$2, \$zero, 12
e8: 01 2d 00 08	ld \$2, 8(\$sp)
ec: 09 22 00 01	addiu \$2, \$2, 1
f0: 02 2d 00 08	st \$2, 8(\$sp)
f4: 01 2d 00 04	ld \$2, 4(\$sp)
f8: 0a 22 00 01	slti \$2, \$2, 1
fc: 28 20 00 0c	bne \$2, \$zero, 12
100: 01 2d 00 04	ld \$2, 4(\$sp)
104: 09 22 00 01	addiu \$2, \$2, 1
108: 02 2d 00 04	st \$2, 4(\$sp)
10c: 01 2d 00 04	ld \$2, 4(\$sp)
110: 01 3d 00 00	ld \$3, 0(\$sp)
114: 20 23 20 00	slt \$2, \$3, \$2
118: 27 20 00 0c	beq \$2, \$zero, 12
11c: 01 2d 00 00	ld \$2, 0(\$sp)
120: 09 22 00 01	addiu \$2, \$2, 1
124: 02 2d 00 00	st \$2, 0(\$sp)
128: 01 2d 00 1c	ld \$2, 28(\$sp)
12c: 01 3d 00 20	ld \$3, 32(\$sp)
130: 27 32 00 0c	beq \$3, \$2, 12
134: 01 2d 00 20	ld \$2, 32(\$sp)
138: 09 22 00 01	addiu \$2, \$2, 1
13c: 02 2d 00 20	st \$2, 32(\$sp)
140: 01 2d 00 20	ld \$2, 32(\$sp)
144: 09 dd 00 28	addiu \$sp, \$sp, 40
148: 2c 00 00 00	ret \$zero

9.7 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.

9.7.1 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,

LLVMBackendTutorialExampleCode/InputFiles/main.cpp

```
extern int foo(int x1, int x2);
extern int bar();

int main()
{
    int a = foo(1, 2);
    a += foo(3, 4);
    a += bar();

    return a;
}

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 $sp,40,$lr
    .mask 0x00004080,-4
    .set noreorder
    .cupload $t9
    .set nomacro
# BB#0:
    addiu $sp, $sp, -40
$tmp2:
    .cfi_def_cfa_offset 40
    st $lr, 36($sp)          # 4-byte Folded Spill
    st $7, 32($sp)           # 4-byte Folded Spill
$tmp3:
    .cfi_offset 14, -4
$tmp4:
    .cfi_offset 7, -8
    .cprestore 8
    addiu $2, $zero, 0
    st $2, 28($sp)
    addiu $2, $zero, 2
    st $2, 4($sp)
    addiu $2, $zero, 1
    st $2, 0($sp)
    ld $7, %call124(_Z3fooii)($gp)
    add $6, $zero, $7
```

```

jalr $6
ld $gp, 8($sp)
st $2, 24($sp)
addiu $2, $zero, 4
st $2, 4($sp)
addiu $2, $zero, 3
st $2, 0($sp)
add $6, $zero, $7
jalr $6
ld $gp, 8($sp)
ld $3, 24($sp)
add $2, $3, $2
st $2, 24($sp)
ld $6, %call24(_Z3barv)($gp)
jalr $6
ld $gp, 8($sp)
ld $3, 24($sp)
add $2, $3, $2
st $2, 24($sp)
ld $7, 32($sp)           # 4-byte Folded Reload
ld $lr, 36($sp)           # 4-byte Folded Reload
addiu $sp, $sp, 40
ret $2
.set macro
.set reorder
.end main

$tmp5:
.size main, ($tmp5)-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_CALL24 _Z3fooii
112 R_CPU0_CALL24 _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 _Z5fooii 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      $6,0x0acc($8)
400738:    addiu  $9,$zero,0x18
40073c:    jr      $6

00400730 <_Z3fooii@plt>:
400730:    lui     $8,0x41
400734:    ld      $6,0x0ado($8)
400738:    addiu  $9,$zero,0x08
40073c:    jr      $6
...
004009f0 <.CPU0.stubs>:
4009F0: 8f998010 ld      $6,-32752(gp)
4009F4: 03e07821 add    $8,$zero, lr
4009F8: 0320f809 jalr  $6

```

9.7.2 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 9.2](#)

Table 9.1: 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
\$6 (at the end of <code>_Z3fooii@plt</code>)	point to CPU0.stubs0	point to <code>_Z3fooii</code>
\$6 (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 $7, %call24(_Z3fooii@plt)($gp)`”, “`add $6, $zero, $7`” and “`jalr $6`”. Remember, `.dynsym+0x08` contains information (libfoobar.so, offset, length) which is set by linker at link to dynamic shared library. After “`jalr $6`”, 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 `$6 = 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 `$6, $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.

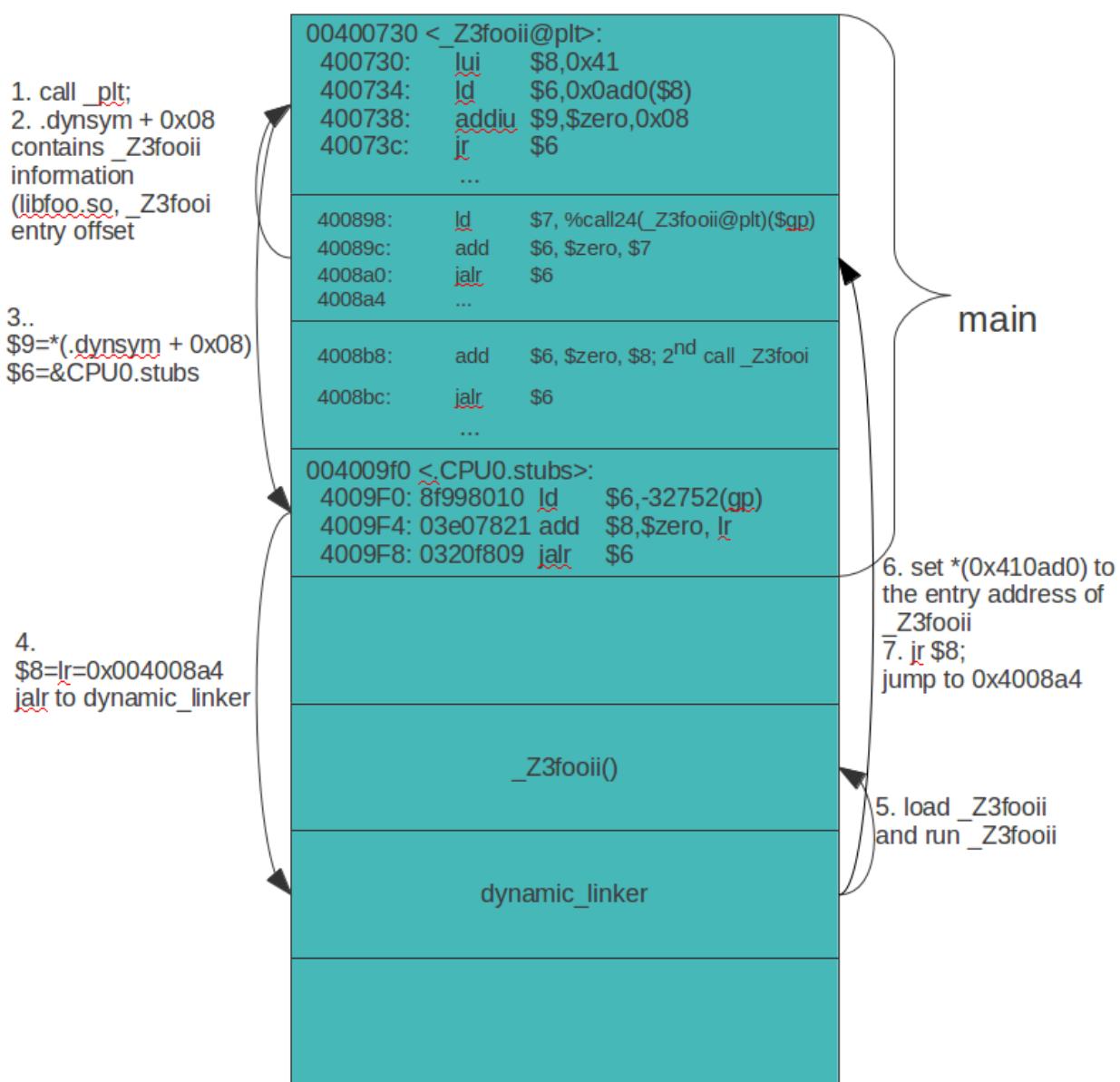


Figure 9.2: Call dynamic function `_Z3fooii()` first time

After the `_Z3fooii()` is called at second time, it looks like Figure 9.3. 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 9.2. 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 `$6` as the `$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.

9.7.3 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
```

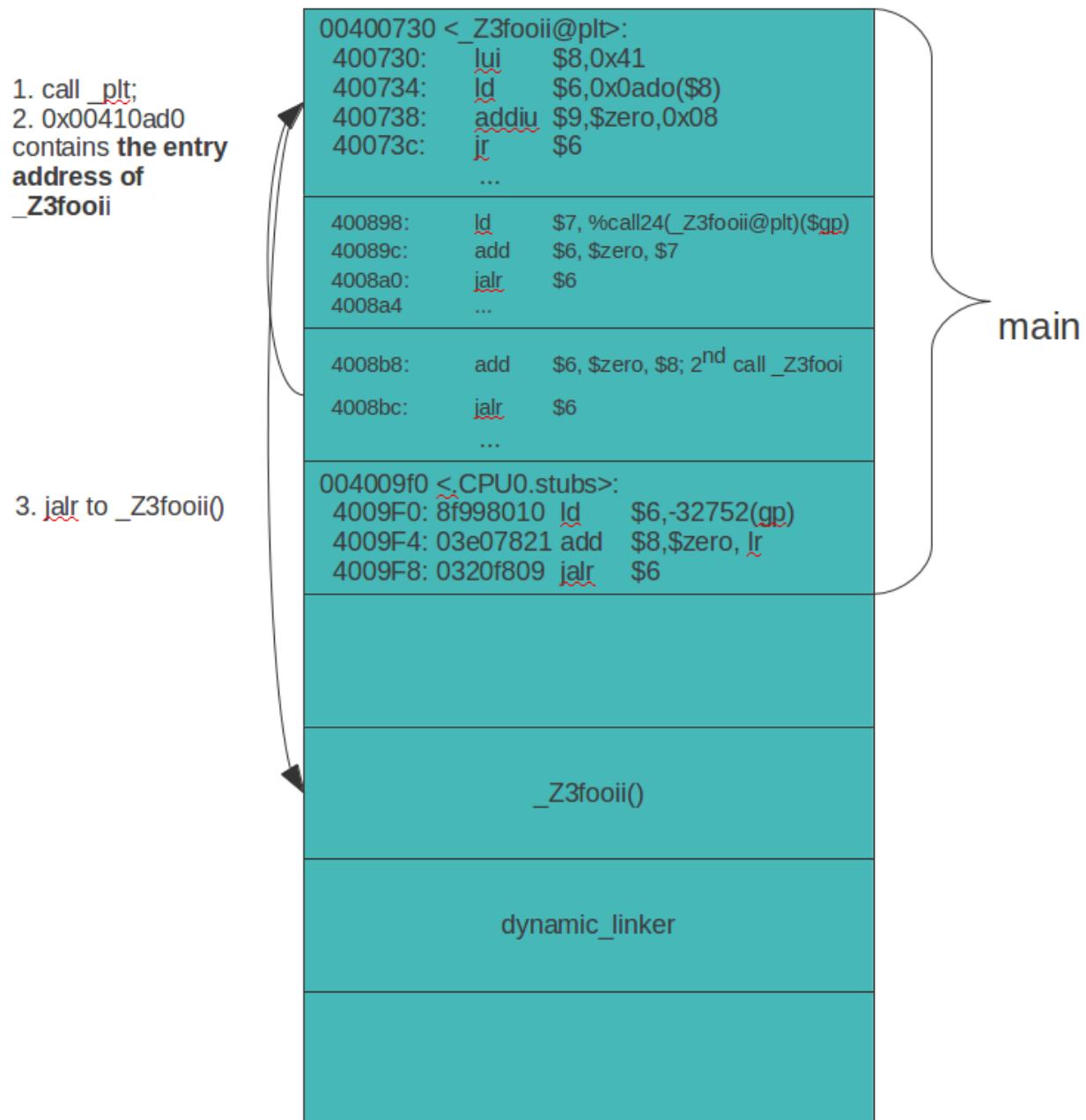
```
118-165-77-200:InputFiles Jonathan$ gobjdump -d a.out
```

```
...
```

```
main.o:      fileformat mach-o-x86-64
```

Disassembly of section .text:

```
0000000100000ef0 <_main>:
```

Figure 9.3: Call dynamic function `_Z3fooii()` second time

```

1000000ef0: 55          push    %rbp
1000000ef1: 48 89 e5    mov     %rsp,%rbp
1000000ef4: 48 83 ec 10 sub    $0x10,%rsp
1000000ef8: bf 01 00 00 00 mov    $0x1,%edi
1000000efd: be 02 00 00 00 mov    $0x2,%esi
1000000f02: c7 45 fc 00 00 00 00 movl   $0x0,-0x4(%rbp)
1000000f09: e8 36 00 00 00 callq  100000f44 <__Z3fooii$stub>
1000000f0e: bf 03 00 00 00 mov    $0x3,%edi
1000000f13: be 04 00 00 00 mov    $0x4,%esi
1000000f18: 89 45 f8      mov     %eax,-0x8(%rbp)
1000000f1b: e8 24 00 00 00 callq  100000f44 <__Z3fooii$stub>
1000000f20: 8b 75 f8      mov     -0x8(%rbp),%esi
1000000f23: 01 c6          add    %eax,%esi
1000000f25: 89 75 f8      mov     %esi,-0x8(%rbp)
1000000f28: e8 11 00 00 00 callq  100000f3e <__Z3barv$stub>
1000000f2d: 8b 75 f8      mov     -0x8(%rbp),%esi
1000000f30: 01 c6          add    %eax,%esi
1000000f32: 89 75 f8      mov     %esi,-0x8(%rbp)
1000000f35: 8b 45 f8      mov     -0x8(%rbp),%eax
1000000f38: 48 83 c4 10      add    $0x10,%rsp
1000000f3c: 5d          pop    %rbp
1000000f3d: 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/
LLVMBackendTutorialExampleCode/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/
LLVMBackendTutorialExampleCode/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 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:
-> 0xffff978da878: pushq %rbp
  0xffff978da879: 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.

10.1 AsmParser support

Run Chapter9_1/ with ch10_1.cpp will get the following error message.

[LLVMBackendTutorialExampleCode/InputFiles/ch10_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 ch10_1.cpp -emit-llvm -o ch10_1.bc
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=obj ch10_1.bc
-o ch10_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 Chapter10_1/ include AsmParser implementation as follows,

LLVMBackendTutorialExampleCode/Chapter10_1/AsmParser/Cpu0AsmParser.cpp

```
===== Cpu0AsmParser.cpp - Parse Cpu0 assembly to MCInst instructions =====/
//
// The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
=====

#include "MCTargetDesc/Cpu0MCTargetDesc.h"
#include "Cpu0RegisterInfo.h"
#include "llvm/ADT/StringSwitch.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCEExpr.h"
#include "llvm/MC/MCInst.h"
#include "llvm/MC/MCStreamer.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/MC/MCSymbol.h"
#include "llvm/MC/MCParser/MCAsmLexer.h"
#include "llvm/MC/MCParser/MCParsedAsmOperand.h"
#include "llvm/MC/MCTargetAsmParser.h"
#include "llvm/Support/TargetRegistry.h"

using namespace llvm;

namespace {
class Cpu0AssemblerOptions {
public:
    Cpu0AssemblerOptions():
        aTReg(1), reorder(true), macro(true) {
    }
}
```

```

bool isReorder() {return reorder; }
void setReorder() {reorder = true; }
void setNoreorder() {reorder = false; }

bool isMacro() {return macro; }
void setMacro() {macro = true; }
void setNomacro() {macro = false; }

private:
    unsigned aTReg;
    bool reorder;
    bool macro;
};

}

namespace {
class Cpu0AsmParser : public MCTargetAsmParser {
    MCSubtargetInfo &STI;
    MCAsmParser &Parser;
    Cpu0AssemblerOptions Options;

#define GET_ASSEMBLER_HEADER
#include "Cpu0GenAsmMatcher.inc"

    bool MatchAndEmitInstruction(SMLoc IDLoc, unsigned &Opcode,
                                SmallVectorImpl<MCParsedAsmOperand*> &Operands,
                                MCStreamer &Out, unsigned &ErrorInfo,
                                bool MatchingInlineAsm);

    bool ParseRegister(unsigned &RegNo, SMLoc &StartLoc, SMLoc &EndLoc);

    bool ParseInstruction(ParseInstructionInfo &Info, StringRef Name,
                           SMLoc NameLoc,
                           SmallVectorImpl<MCParsedAsmOperand*> &Operands);

    bool parseMathOperation(StringRef Name, SMLoc NameLoc,
                           SmallVectorImpl<MCParsedAsmOperand*> &Operands);

    bool ParseDirective(AsmToken DirectiveID);

    Cpu0AsmParser::OperandMatchResultTy
    parseMemOperand(SmallVectorImpl<MCParsedAsmOperand*>&);

    bool ParseOperand(SmallVectorImpl<MCParsedAsmOperand*> &,
                      StringRef Mnemonic);

    int tryParseRegister(StringRef Mnemonic);

    bool tryParseRegisterOperand(SmallVectorImpl<MCParsedAsmOperand*> &Operands,
                                StringRef Mnemonic);

    bool needsExpansion(MCInst &Inst);

    void expandInstruction(MCInst &Inst, SMLoc IDLoc,
                           SmallVectorImpl<MCInst> &Instructions);
    void expandLoadImm(MCInst &Inst, SMLoc IDLoc,
                        SmallVectorImpl<MCInst> &Instructions);

```

```

void expandLoadAddressImm(MCInst &Inst, SMLoc IDLoc,
                           SmallVectorImpl<MCInst> &Instructions);
void expandLoadAddressReg(MCInst &Inst, SMLoc IDLoc,
                           SmallVectorImpl<MCInst> &Instructions);
bool reportParseError(StringRef ErrorMsg);

bool parseMemOffset(const MCEexpr *&Res);
bool parseRelocOperand(const MCEexpr *&Res);

bool parseDirectiveSet();

bool parseSetAtDirective();
bool parseSetNoAtDirective();
bool parseSetMacroDirective();
bool parseSetNoMacroDirective();
bool parseSetReorderDirective();
bool parseSetNoReorderDirective();

MCSymbolRefExpr::VariantKind getVariantKind(StringRef Symbol);

int matchRegisterName(StringRef Symbol);

int matchRegisterByNumber(unsigned RegNum, StringRef Mnemonic);

unsigned getReg(int RC, int RegNo);

public:
    Cpu0AsmParser(MCSubtargetInfo &sti, MCAsmParser &parser)
        : MCTargetAsmParser(), STI(sti), Parser(parser) {
        // Initialize the set of available features.
        setAvailableFeatures(ComputeAvailableFeatures(STI.getFeatureBits()));
    }

    MCAsmParser &getParser() const { return Parser; }
    MCAsmLexer &getLexer() const { return Parser.getLexer(); }

};

namespace {

/// Cpu0Operand - Instances of this class represent a parsed Cpu0 machine
/// instruction.
class Cpu0Operand : public MCParsedAsmOperand {

    enum KindTy {
        k_CondCode,
        k_CoprocNum,
        k_Immediate,
        k_Memory,
        k_PostIndexRegister,
        k_Register,
        k_Token
    } Kind;

    Cpu0Operand(KindTy K) : MCParsedAsmOperand(), Kind(K) {}

    union {

```

```

struct {
  const char *Data;
  unsigned Length;
} Tok;

struct {
  unsigned RegNum;
} Reg;

struct {
  const MCExpr *Val;
} Imm;

struct {
  unsigned Base;
  const MCExpr *Off;
} Mem;
};

SMLoc StartLoc, EndLoc;

public:
  void addRegOperands(MCInst &Inst, unsigned N) const {
    assert(N == 1 && "Invalid number of operands!");
    Inst.addOperand(MCOperand::CreateReg(getReg()));
  }

  void addExpr(MCInst &Inst, const MCExpr *Expr) const{
    // Add as immediate when possible. Null MCExpr = 0.
    if (Expr == 0)
      Inst.addOperand(MCOperand::CreateImm(0));
    else if (const MCConstantExpr *CE = dyn_cast<MCConstantExpr>(Expr))
      Inst.addOperand(MCOperand::CreateImm(CE->getValue()));
    else
      Inst.addOperand(MCOperand::CreateExpr(Expr));
  }

  void addImmOperands(MCInst &Inst, unsigned N) const {
    assert(N == 1 && "Invalid number of operands!");
    const MCExpr *Expr = getImm();
    addExpr(Inst,Expr);
  }

  void addMemOperands(MCInst &Inst, unsigned N) const {
    assert(N == 2 && "Invalid number of operands!");

    Inst.addOperand(MCOperand::CreateReg(getMemBase()));

    const MCExpr *Expr = getMemOff();
    addExpr(Inst,Expr);
  }

  bool isReg() const { return Kind == k_Register; }
  bool isImm() const { return Kind == k_Immediate; }
  bool isToken() const { return Kind == k_Token; }
  bool isMem() const { return Kind == k_Memory; }

 StringRef getToken() const {

```

```

    assert(Kind == k_Token && "Invalid access!");
    return StringRef(Tok.Data, Tok.Length);
}

unsigned getReg() const {
    assert((Kind == k_Register) && "Invalid access!");
    return Reg.RegNum;
}

const MCExpr *getImm() const {
    assert((Kind == k_Immediate) && "Invalid access!");
    return Imm.Val;
}

unsigned getMemBase() const {
    assert((Kind == k_Memory) && "Invalid access!");
    return Mem.Base;
}

const MCExpr *getMemOff() const {
    assert((Kind == k_Memory) && "Invalid access!");
    return Mem.Off;
}

static Cpu0Operand *CreateToken(StringRef Str, SMLoc S) {
    Cpu0Operand *Op = new Cpu0Operand(k_Token);
    Op->Tok.Data = Str.data();
    Op->Tok.Length = Str.size();
    Op->StartLoc = S;
    Op->EndLoc = S;
    return Op;
}

static Cpu0Operand *CreateReg(unsigned RegNum, SMLoc S, SMLoc E) {
    Cpu0Operand *Op = new Cpu0Operand(k_Register);
    Op->Reg.RegNum = RegNum;
    Op->StartLoc = S;
    Op->EndLoc = E;
    return Op;
}

static Cpu0Operand *CreateImm(const MCExpr *Val, SMLoc S, SMLoc E) {
    Cpu0Operand *Op = new Cpu0Operand(k_Immediate);
    Op->Imm.Val = Val;
    Op->StartLoc = S;
    Op->EndLoc = E;
    return Op;
}

static Cpu0Operand *CreateMem(unsigned Base, const MCExpr *Off,
                           SMLoc S, SMLoc E) {
    Cpu0Operand *Op = new Cpu0Operand(k_Memory);
    Op->Mem.Base = Base;
    Op->Mem.Off = Off;
    Op->StartLoc = S;
    Op->EndLoc = E;
    return Op;
}

```

```
/// getStartLoc - Get the location of the first token of this operand.
SMLoc getStartLoc() const { return StartLoc; }
/// getEndLoc - Get the location of the last token of this operand.
SMLoc getEndLoc() const { return EndLoc; }

virtual void print(raw_ostream &OS) const {
    llvm_unreachable("unimplemented!");
}
};

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);
}
}

void Cpu0AsmParser::expandLoadImm(MCInst &Inst, SMLoc IDLoc,
                                  SmallVectorImpl<MCInst> &Instructions) {
MCInst tmpInst;
const MCOperand &ImmOp = Inst.getOperand(1);
assert(ImmOp.isImm() && "expected immediate operand kind");
const MCOperand &RegOp = Inst.getOperand(0);
assert(RegOp.isReg() && "expected register operand kind");

int ImmValue = ImmOp.getImm();
tmpInst.setLoc(IDLoc);
if (0 <= ImmValue && ImmValue <= 65535) {
    // for 0 <= j <= 65535.
    // li d,j => ori d,$zero,j
    tmpInst.setOpcode(Cpu0::ORI);
    tmpInst.addOperand(MCOperand::CreateReg(RegOp.getReg()));
    tmpInst.addOperand(
        MCOperand::CreateReg(Cpu0::ZERO));
    tmpInst.addOperand(MCOperand::CreateImm(ImmValue));
    Instructions.push_back(tmpInst);
} else if (ImmValue < 0 && ImmValue >= -32768) {
    // for -32768 <= j < 0.
    // li d,j => addiu d,$zero,j
    tmpInst.setOpcode(Cpu0::ADDIU); //TODO: no ADDIU64 in td files?
}
```

```

tmpInst.addOperand(MCOperand::CreateReg(RegOp.getReg()));
tmpInst.addOperand(
    MCOperand::CreateReg(Cpu0::ZERO));
tmpInst.addOperand(MCOperand::CreateImm(ImmValue));
Instructions.push_back(tmpInst);
} else {
    // for any other value of j that is representable as a 32-bit integer.
    // li d,j => lui d,hi16(j)
    // ori d,d,lo16(j)
tmpInst.setOpcode(Cpu0::LUI);
tmpInst.addOperand(MCOperand::CreateReg(RegOp.getReg()));
tmpInst.addOperand(MCOperand::CreateImm((ImmValue & 0xffff0000) >> 16));
Instructions.push_back(tmpInst);
tmpInst.clear();
tmpInst.setOpcode(Cpu0::ORI);
tmpInst.addOperand(MCOperand::CreateReg(RegOp.getReg()));
tmpInst.addOperand(MCOperand::CreateReg(RegOp.getReg()));
tmpInst.addOperand(MCOperand::CreateImm(ImmValue & 0xffff));
tmpInst.setLoc(IDLoc);
Instructions.push_back(tmpInst);
}
}

void Cpu0AsmParser::expandLoadAddressReg(MCInst &Inst, SMLoc IDLoc,
                                         SmallVectorImpl<MCInst> &Instructions) {
MCInst tmpInst;
const MCOperand &ImmOp = Inst.getOperand(2);
assert(ImmOp.isImm() && "expected immediate operand kind");
const MCOperand &SrcRegOp = Inst.getOperand(1);
assert(SrcRegOp.isReg() && "expected register operand kind");
const MCOperand &DstRegOp = Inst.getOperand(0);
assert(DstRegOp.isReg() && "expected register operand kind");
int ImmValue = ImmOp.getImm();
if (-32768 <= ImmValue && ImmValue <= 32767) {
    // for -32768 <= j < 32767.
    //la d,j(s) => addiu d,s,j
tmpInst.setOpcode(Cpu0::ADDiu); //TODO: no ADDiu64 in td files?
tmpInst.addOperand(MCOperand::CreateReg(DstRegOp.getReg()));
tmpInst.addOperand(MCOperand::CreateReg(SrcRegOp.getReg()));
tmpInst.addOperand(MCOperand::CreateImm(ImmValue));
Instructions.push_back(tmpInst);
} else {
    // for any other value of j that is representable as a 32-bit integer.
    // la d,j(s) => lui d,hi16(j)
    // ori d,d,lo16(j)
    // add d,d,s
tmpInst.setOpcode(Cpu0::LUI);
tmpInst.addOperand(MCOperand::CreateReg(DstRegOp.getReg()));
tmpInst.addOperand(MCOperand::CreateImm((ImmValue & 0xffff0000) >> 16));
Instructions.push_back(tmpInst);
tmpInst.clear();
tmpInst.setOpcode(Cpu0::ORI);
tmpInst.addOperand(MCOperand::CreateReg(DstRegOp.getReg()));
tmpInst.addOperand(MCOperand::CreateReg(DstRegOp.getReg()));
tmpInst.addOperand(MCOperand::CreateImm(ImmValue & 0xffff));
Instructions.push_back(tmpInst);
tmpInst.clear();
tmpInst.setOpcode(Cpu0::ADD);

```

```

        tmpInst.addOperand(MCOperand::CreateReg(DstRegOp.getReg()));
        tmpInst.addOperand(MCOperand::CreateReg(DstRegOp.getReg()));
        tmpInst.addOperand(MCOperand::CreateReg(SrcRegOp.getReg()));
        Instructions.push_back(tmpInst);
    }
}

void Cpu0AsmParser::expandLoadAddressImm(MCInst &Inst, SMLoc IDLoc,
                                         SmallVectorImpl<MCInst> &Instructions) {
    MCInst tmpInst;
    const MCOperand &ImmOp = Inst.getOperand(1);
    assert(ImmOp.isImm() && "expected immediate operand kind");
    const MCOperand &RegOp = Inst.getOperand(0);
    assert(RegOp.isReg() && "expected register operand kind");
    int ImmValue = ImmOp.getImm();
    if ( -32768 <= ImmValue && ImmValue <= 32767) {
        // for -32768 <= j < 32767.
        // la d,j => addiu d,$zero,j
        tmpInst.setOpcode(Cpu0::ADDiu);
        tmpInst.addOperand(MCOperand::CreateReg(RegOp.getReg()));
        tmpInst.addOperand(
            MCOperand::CreateReg(Cpu0::ZERO));
        tmpInst.addOperand(MCOperand::CreateImm(ImmValue));
        Instructions.push_back(tmpInst);
    } else {
        // for any other value of j that is representable as a 32-bit integer.
        // la d,j => lui d,hi16(j)
        //          ori d,d,lo16(j)
        tmpInst.setOpcode(Cpu0::LUI);
        tmpInst.addOperand(MCOperand::CreateReg(RegOp.getReg()));
        tmpInst.addOperand(MCOperand::CreateImm((ImmValue & 0xffff0000) >> 16));
        Instructions.push_back(tmpInst);
        tmpInst.clear();
        tmpInst.setOpcode(Cpu0::ORi);
        tmpInst.addOperand(MCOperand::CreateReg(RegOp.getReg()));
        tmpInst.addOperand(MCOperand::CreateReg(RegOp.getReg()));
        tmpInst.addOperand(MCOperand::CreateImm(ImmValue & 0xffff));
        Instructions.push_back(tmpInst);
    }
}

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);
            for (unsigned i = 0; i < Instructions.size(); i++) {
                Out.EmitInstruction(Instructions[i]);
        }
    }
}

```

```

        }
    } else {
    Inst.setLoc(IDLoc);
    Out.EmitInstruction(Inst);
}
return false;
}
case Match_MissingFeature:
Error(IDLoc, "instruction requires a CPU feature not currently enabled");
return true;
case Match_InvalidOperand: {
SMLoc ErrorLoc = IDLoc;
if (ErrorInfo != ~0U) {
    if (ErrorInfo >= Operands.size())
        return Error(IDLoc, "too few operands for instruction");

    ErrorLoc = ((Cpu0Operand*)Operands[ErrorInfo])->getStartLoc();
    if (ErrorLoc == SMLoc()) ErrorLoc = IDLoc;
}

return Error(ErrorLoc, "invalid operand for instruction");
}
case Match_MnemonicFail:
return Error(IDLoc, "invalid instruction");
}
return true;
}

int Cpu0AsmParser::matchRegisterName(StringRef Name) {

int CC;
CC = StringSwitch<unsigned>(Name)
.Case("zero", Cpu0::ZERO)
.Case("at", Cpu0::AT)
.Case("v0", Cpu0::V0)
.Case("v1", Cpu0::V1)
.Case("a0", Cpu0::A0)
.Case("a1", Cpu0::A1)
.Case("t9", Cpu0::T9)
.Case("s0", Cpu0::S0)
.Case("s1", Cpu0::S1)
.Case("s2", Cpu0::S2)
.Case("gp", Cpu0::GP)
.Case("fp", Cpu0::FP)
.Case("sw", Cpu0::SW)
.Case("sp", Cpu0::SP)
.Case("lr", Cpu0::LR)
.Case("pc", Cpu0::PC)
.Default(-1);

if (CC != -1)
    return CC;

return -1;
}

unsigned Cpu0AsmParser::getReg(int RC, int RegNo) {
    return *(getContext().getRegisterInfo().getRegClass(RC).begin() + RegNo);
}

```

```

}

int Cpu0AsmParser::matchRegisterByNumber(unsigned RegNum, StringRef Mnemonic) {
    if (RegNum > 15)
        return -1;

    return getReg(Cpu0::CPUREgsRegClassID, RegNum);
}

int Cpu0AsmParser::tryParseRegister(StringRef Mnemonic) {
    const AsmToken &Tok = Parser.getTok();
    int RegNum = -1;

    if (Tok.is(AsmToken::Identifier)) {
        std::string lowerCase = Tok.getString().lower();
        RegNum = matchRegisterName(lowerCase);
    } else if (Tok.is(AsmToken::Integer))
        RegNum = matchRegisterByNumber(static_cast<unsigned>(Tok.getIntVal()),
                                      Mnemonic.lower());
    else
        return RegNum; //error
    return RegNum;
}

bool Cpu0AsmParser::tryParseRegisterOperand(SmallVectorImpl<MCParsedAsmOperand*> &Operands,
                                             StringRef Mnemonic) {

    SMLoc S = Parser.getTok().getLoc();
    int RegNo = -1;

    RegNo = tryParseRegister(Mnemonic);
    if (RegNo == -1)
        return true;

    Operands.push_back(Cpu0Operand::CreateReg(RegNo, S,
                                              Parser.getTok().getLoc()));
    Parser.Lex(); // Eat register token.
    return false;
}

bool Cpu0AsmParser::ParseOperand(SmallVectorImpl<MCParsedAsmOperand*> &Operands,
                                  StringRef Mnemonic) {
    // Check if the current operand has a custom associated parser, if so, try to
    // custom parse the operand, or fallback to the general approach.
    OperandMatchResultTy Resty = MatchOperandParserImpl(Operands, Mnemonic);
    if (Resty == MatchOperand_Success)
        return false;
    // If there wasn't a custom match, try the generic matcher below. Otherwise,
    // there was a match, but an error occurred, in which case, just return that
    // the operand parsing failed.
    if (Resty == MatchOperand_ParseFail)
        return true;

    switch (getLexer().getKind()) {
    default:
        Error(Parser.getTok().getLoc(), "unexpected token in operand");
        return true;
    }
}

```

```

case AsmToken::Dollar: {
    // parse register
    SMLoc S = Parser.getTok().getLoc();
    Parser.Lex(); // Eat dollar token.
    // parse register operand
    if (!tryParseRegisterOperand(Operands, Mnemonic)) {
        if (getLexer().is(AsmToken::LParen)) {
            // check if it is indexed addressing operand
            Operands.push_back(Cpu0Operand::CreateToken("(", S));
            Parser.Lex(); // eat parenthesis
            if (getLexer().isNot(AsmToken::Dollar))
                return true;
        }

        Parser.Lex(); // eat dollar
        if (tryParseRegisterOperand(Operands, Mnemonic))
            return true;

        if (!getLexer().is(AsmToken::RParen))
            return true;
    }

    S = Parser.getTok().getLoc();
    Operands.push_back(Cpu0Operand::CreateToken(")", S));
    Parser.Lex();
}
return false;
}
// maybe it is a symbol reference
StringRef Identifier;
if (Parser.parseIdentifier(Identifier))
    return true;

SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);

MCSymbol *Sym = getContext().GetOrCreateSymbol("$" + Identifier);

// Otherwise create a symbol ref.
const MCEexpr *Res = MCSymbolRefExpr::Create(Sym, MCSymbolRefExpr::VK_None,
                                                getContext());

Operands.push_back(Cpu0Operand::CreateImm(Res, S, E));
return false;
}
case AsmToken::Identifier:
case AsmToken::LParen:
case AsmToken::Minus:
case AsmToken::Plus:
case AsmToken::Integer:
case AsmToken::String: {
    // quoted label names
    const MCEexpr *IdVal;
    SMLoc S = Parser.getTok().getLoc();
    if (getParser().parseExpression(IdVal))
        return true;
    SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);
    Operands.push_back(Cpu0Operand::CreateImm(IdVal, S, E));
    return false;
}
case AsmToken::Percent: {

```

```

// it is a symbol reference or constant expression
const MCExpr *IdVal;
SMLoc S = Parser.getTok().getLoc(); // start location of the operand
if (parseRelocOperand(IdVal))
    return true;

SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);

Operands.push_back(Cpu0Operand::CreateImm(IdVal, S, E));
return false;
} // case AsmToken::Percent
} // switch(getLexer().getKind())
return true;
}

bool Cpu0AsmParser::parseRelocOperand(const MCExpr *&Res) {
    Parser.Lex(); // eat % token
    const AsmToken &Tok = Parser.getTok(); // get next token, operation
    if (Tok.isNot(AsmToken::Identifier))
        return true;

    std::string Str = Tok.getIdentifier().str();

    Parser.Lex(); // eat identifier
    // now make expression from the rest of the operand
    const MCExpr *IdVal;
    SMLoc EndLoc;

    if (getLexer().getKind() == AsmToken::LParen) {
        while (1) {
            Parser.Lex(); // eat '(' token
            if (getLexer().getKind() == AsmToken::Percent) {
                Parser.Lex(); // eat % token
                const AsmToken &nextTok = Parser.getTok();
                if (nextTok.isNot(AsmToken::Identifier))
                    return true;
                Str += "(%";
                Str += nextTok.getIdentifier();
                Parser.Lex(); // eat identifier
                if (getLexer().getKind() != AsmToken::LParen)
                    return true;
            } else
                break;
        }
        if (getParser().parseParenExpression(IdVal, EndLoc))
            return true;
    }

    while (getLexer().getKind() == AsmToken::RParen)
        Parser.Lex(); // eat ')' token

    } else
    return true; // parenthesis must follow reloc operand

    // Check the type of the expression
    if (const MCConstantExpr *MCE = dyn_cast<MCConstantExpr>(IdVal)) {
        // it's a constant, evaluate lo or hi value
        int Val = MCE->getValue();

```

```

if (Str == "lo") {
    Val = Val & 0xffff;
} else if (Str == "hi") {
    Val = (Val & 0xffff0000) >> 16;
}
Res = MCConstantExpr::Create(Val, getContext());
return false;
}

if (const MCSymbolRefExpr *MSRE = dyn_cast<MCSymbolRefExpr>(IdVal)) {
    // it's a symbol, create symbolic expression from symbol
    StringRef Symbol = MSRE->getSymbol().getName();
    MCSymbolRefExpr::VariantKind VK = getVariantKind(Str);
    Res = MCSymbolRefExpr::Create(Symbol, VK, getContext());
    return false;
}
return true;
}

bool Cpu0AsmParser::ParseRegister(unsigned &RegNo, SMLoc &StartLoc,
                                    SMLoc &EndLoc) {

    StartLoc = Parser.getTok().getLoc();
    RegNo = tryParseRegister("");
    EndLoc = Parser.getTok().getLoc();
    return (RegNo == (unsigned)-1);
}

bool Cpu0AsmParser::parseMemOffset(const MCExpr *&Res) {

    SMLoc S;

    switch(getLexer().getKind()) {
    default:
        return true;
    case AsmToken::Integer:
    case AsmToken::Minus:
    case AsmToken::Plus:
        return (getParser().parseExpression(Res));
    case AsmToken::Percent:
        return parseRelocOperand(Res);
    case AsmToken::LParen:
        return false; // it's probably assuming 0
    }
    return true;
}

// eg, 12($sp) or 12($a)
Cpu0AsmParser::OperandMatchResultTy Cpu0AsmParser::parseMemOperand(
    SmallVectorImpl<MCParsedAsmOperand*> &Operands) {

    const MCExpr *IdVal = 0;
    SMLoc S;
    // first operand is the offset
    S = Parser.getTok().getLoc();

    if (parseMemOffset(IdVal))
        return MatchOperand_ParseFail;
}

```

```

const AsmToken &Tok = Parser.getTok(); // get next token
if (Tok.isNot(AsmToken::LParen)) {
    Cpu0Operand *Mnemonic = static_cast<Cpu0Operand*>(Operands[0]);
    if (Mnemonic->getToken() == "la") {
        SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);
        Operands.push_back(Cpu0Operand::CreateImm(IdVal, S, E));
        return MatchOperand_Success;
    }
    Error(Parser.getTok().getLoc(), "'(' expected");
    return MatchOperand_ParseFail;
}

Parser.Lex(); // Eat '(' token.

const AsmToken &Tok1 = Parser.getTok(); // get next token
if (Tok1.is(AsmToken::Dollar)) {
    Parser.Lex(); // Eat '$' token.
    if (tryParseRegisterOperand(Operands, "")) {
        Error(Parser.getTok().getLoc(), "unexpected token in operand");
        return MatchOperand_ParseFail;
    }
}

else {
    Error(Parser.getTok().getLoc(), "unexpected token in operand");
    return MatchOperand_ParseFail;
}

const AsmToken &Tok2 = Parser.getTok(); // get next token
if (Tok2.isNot(AsmToken::RParen)) {
    Error(Parser.getTok().getLoc(), "')' expected");
    return MatchOperand_ParseFail;
}

SMLoc E = SMLoc::getFromPointer(Parser.getTok().getLoc().getPointer() - 1);

Parser.Lex(); // Eat ')' token.

if (IdVal == 0)
    IdVal = MCConstantExpr::Create(0, getContext());

    // now replace register operand with the mem operand
    Cpu0Operand* op = static_cast<Cpu0Operand*>(Operands.back());
    int RegNo = op->getReg();
    // remove register from operands
    Operands.pop_back();
    // and add memory operand
    Operands.push_back(Cpu0Operand::CreateMem(RegNo, IdVal, S, E));
    delete op;
    return MatchOperand_Success;
}

MCSymbolRefExpr::VariantKind Cpu0AsmParser::getVariantKind(StringRef Symbol) {

    MCSymbolRefExpr::VariantKind VK
        = StringSwitch<MCSymbolRefExpr::VariantKind>(Symbol)
            .Case("hi", MCSymbolRefExpr::VK_Cpu0_ABS_HI)
            .Case("lo", MCSymbolRefExpr::VK_Cpu0_ABS_LO)
            .Case("gp_rel", MCSymbolRefExpr::VK_Cpu0_GPREL)
}

```

```

.Case("call24",      MCSymbolRefExpr::VK_Cpu0_GOT_CALL)
.Case("got",         MCSymbolRefExpr::VK_Cpu0_GOT)
.Case("tlsldm",     MCSymbolRefExpr::VK_Cpu0_TLSGD)
.Case("tlsldm",     MCSymbolRefExpr::VK_Cpu0_TLSLD)
.Case("dtprel_hi",  MCSymbolRefExpr::VK_Cpu0_DTPREL_HI)
.Case("dtprel_lo",  MCSymbolRefExpr::VK_Cpu0_DTPREL_LO)
.Case("gottprel",   MCSymbolRefExpr::VK_Cpu0_GOTTPREL)
.Case("tprel_hi",   MCSymbolRefExpr::VK_Cpu0_TPREL_HI)
.Case("tprel_lo",   MCSymbolRefExpr::VK_Cpu0_TPREL_LO)
.Case("got_disp",   MCSymbolRefExpr::VK_Cpu0_GOT_DISP)
.Case("got_page",   MCSymbolRefExpr::VK_Cpu0_GOT_PAGE)
.Case("got_ofst",   MCSymbolRefExpr::VK_Cpu0_GOT_OFST)
.Case("hi(%neg(%gp_rel)", MCSymbolRefExpr::VK_Cpu0_GPOFF_HI)
.Case("lo(%neg(%gp_rel)", MCSymbolRefExpr::VK_Cpu0_GPOFF_LO)
.Default(MCSymbolRefExpr::VK_None);

return VK;
}

bool Cpu0AsmParser::
parseMathOperation(StringRef Name, SMLoc NameLoc,
                  SmallVectorImpl<MCParsedAsmOperand*> &Operands) {
    // split the format
    size_t Start = Name.find('.'), Next = Name.rfind('.');
    StringRef Format1 = Name.slice(Start, Next);
    // and add the first format to the operands
    Operands.push_back(Cpu0Operand::CreateToken(Format1, NameLoc));
    // now for the second format
    StringRef Format2 = Name.slice(Next,StringRef::npos);
    Operands.push_back(Cpu0Operand::CreateToken(Format2, NameLoc));

    // set the format for the first register
    // setFpFormat(Format1);

    // Read the remaining operands.
    if (getLexer().isNot(AsmToken::EndOfStatement)) {
        // Read the first operand.
        if (ParseOperand(Operands, Name)) {
            SMLoc Loc = getLexer().getLoc();
            Parser.eatToEndOfStatement();
            return Error(Loc, "unexpected token in argument list");
        }

        if (getLexer().isNot(AsmToken::Comma)) {
            SMLoc Loc = getLexer().getLoc();
            Parser.eatToEndOfStatement();
            return Error(Loc, "unexpected token in argument list");

        }
        Parser.Lex(); // Eat the comma.

        // Parse and remember the operand.
        if (ParseOperand(Operands, Name)) {
            SMLoc Loc = getLexer().getLoc();
            Parser.eatToEndOfStatement();
            return Error(Loc, "unexpected token in argument list");
        }
    }
}

```

```
if (getLexer().isNot(AsmToken::EndOfStatement)) {
    SMLoc Loc = getLexer().getLoc();
    Parser.eatToEndOfStatement();
    return Error(Loc, "unexpected token in argument list");
}

Parser.Lex(); // Consume the EndOfStatement
return false;
}

bool Cpu0AsmParser::ParseInstruction(ParseInstructionInfo &Info, StringRef Name, SMLoc NameLoc,
                                      SmallVectorImpl<MCParsedAsmOperand*> &Operands) {

    // Create the leading tokens for the mnemonic, split by '.' characters.
    size_t Start = 0, Next = Name.find('.');
    StringRef Mnemonic = Name.slice(Start, Next);

    Operands.push_back(Cpu0Operand::CreateToken(Mnemonic, NameLoc));

    // Read the remaining operands.
    if (getLexer().isNot(AsmToken::EndOfStatement)) {
        // Read the first operand.
        if (ParseOperand(Operands, Name)) {
            SMLoc Loc = getLexer().getLoc();
            Parser.eatToEndOfStatement();
            return Error(Loc, "unexpected token in argument list");
        }
    }

    while (getLexer().is(AsmToken::Comma) ) {
        Parser.Lex(); // Eat the comma.

        // Parse and remember the operand.
        if (ParseOperand(Operands, Name)) {
            SMLoc Loc = getLexer().getLoc();
            Parser.eatToEndOfStatement();
            return Error(Loc, "unexpected token in argument list");
        }
    }
}

if (getLexer().isNot(AsmToken::EndOfStatement)) {
    SMLoc Loc = getLexer().getLoc();
    Parser.eatToEndOfStatement();
    return Error(Loc, "unexpected token in argument list");
}

Parser.Lex(); // Consume the EndOfStatement
return false;
}

bool Cpu0AsmParser::reportParseError(StringRef ErrorMsg) {
    SMLoc Loc = getLexer().getLoc();
    Parser.eatToEndOfStatement();
    return Error(Loc, ErrorMsg);
}

bool Cpu0AsmParser::parseSetReorderDirective() {
```

```
Parser.Lex();
// if this is not the end of the statement, report error
if (getLexer().isNot(AsmToken::EndOfStatement)) {
    reportParseError("unexpected token in statement");
    return false;
}
Options.setReorder();
Parser.Lex(); // Consume the EndOfStatement
return false;
}

bool Cpu0AsmParser::parseSetNoReorderDirective() {
    Parser.Lex();
    // if this is not the end of the statement, report error
    if (getLexer().isNot(AsmToken::EndOfStatement)) {
        reportParseError("unexpected token in statement");
        return false;
    }
    Options.setNoreorder();
    Parser.Lex(); // Consume the EndOfStatement
    return false;
}

bool Cpu0AsmParser::parseSetMacroDirective() {
    Parser.Lex();
    // if this is not the end of the statement, report error
    if (getLexer().isNot(AsmToken::EndOfStatement)) {
        reportParseError("unexpected token in statement");
        return false;
    }
    Options.setMacro();
    Parser.Lex(); // Consume the EndOfStatement
    return false;
}

bool Cpu0AsmParser::parseSetNoMacroDirective() {
    Parser.Lex();
    // if this is not the end of the statement, report error
    if (getLexer().isNot(AsmToken::EndOfStatement)) {
        reportParseError("`noreorder` must be set before `nomacro`");
        return false;
    }
    if (Options.isReorder()) {
        reportParseError("`noreorder` must be set before `nomacro`");
        return false;
    }
    Options.setNomacro();
    Parser.Lex(); // Consume the EndOfStatement
    return false;
}

bool Cpu0AsmParser::parseDirectiveSet() {

    // get next token
    const AsmToken &Tok = Parser.getTok();

    if (Tok.getString() == "reorder") {
        return parseSetReorderDirective();
    } else if (Tok.getString() == "noreorder") {
```

```
    return parseSetNoReorderDirective();
} else if (Tok.getString() == "macro") {
    return parseSetMacroDirective();
} else if (Tok.getString() == "nomacro") {
    return parseSetNoMacroDirective();
}
return true;
}

bool Cpu0AsmParser::ParseDirective(AsmToken DirectiveID) {

    if (DirectiveID.getString() == ".ent") {
        // ignore this directive for now
        Parser.Lex();
        return false;
    }

    if (DirectiveID.getString() == ".end") {
        // ignore this directive for now
        Parser.Lex();
        return false;
    }

    if (DirectiveID.getString() == ".frame") {
        // ignore this directive for now
        Parser.eatToEndOfStatement();
        return false;
    }

    if (DirectiveID.getString() == ".set") {
        return parseDirectiveSet();
    }

    if (DirectiveID.getString() == ".fmask") {
        // ignore this directive for now
        Parser.eatToEndOfStatement();
        return false;
    }

    if (DirectiveID.getString() == ".mask") {
        // ignore this directive for now
        Parser.eatToEndOfStatement();
        return false;
    }

    if (DirectiveID.getString() == ".gpword") {
        // ignore this directive for now
        Parser.eatToEndOfStatement();
        return false;
    }

    return true;
}

extern "C" void LLVMInitializeCpu0AsmParser() {
    RegisterMCAsmParser<Cpu0AsmParser> X(TheCpu0Target);
    RegisterMCAsmParser<Cpu0AsmParser> Y(TheCpu0elTarget);
}
```

```
#define GET_REGISTER_MATCHER
#define GET_MATCHER_IMPLEMENTATION
#include "Cpu0GenAsmMatcher.inc"
```

LLVMBackendTutorialExampleCode/Chapter10_1/AsmParser/CMakeLists.txt

```
include_directories( ${CMAKE_CURRENT_BINARY_DIR}/.. ${CMAKE_CURRENT_SOURCE_DIR}/.. )
add_llvm_library(LLVMCpu0AsmParser
  Cpu0AsmParser.cpp
)
add_dependencies(LLVMCpu0AsmParser Cpu0CommonTableGen)
```

LLVMBackendTutorialExampleCode/Chapter10_1/AsmParser/LLVMBuild.txt

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

[component_0]
type = Library
name = Cpu0AsmParser
parent = Mips
required_libraries = MC MCParse Support MipsDesc MipsInfo
add_to_library_groups = Cpu0
```

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,

LLVMBackendTutorialExampleCode/Chapter10_1/CMakeLists.txt

```
tablegen (LLVM_Cpu0GenAsmMatcher.inc -gen-asm=matcher)
...
add_subdirectory(AsmParser)
```

LLVMBackendTutorialExampleCode/Chapter10_1/LLVMBuild.txt

```
subdirectories = AsmParser ...
...
has_asmparser = 1
```

The other files change as follows,

LLVMBackendTutorialExampleCode/Chapter10_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();
    ...
}
```

LLVMBackendTutorialExampleCode/Chapter10_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];
}
```

LLVMBackendTutorialExampleCode/Chapter10_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 = [];
}
```

LLVMBackendTutorialExampleCode/Chapter10_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", sham, 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", sham, 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,

LLVMBackendTutorialExampleCode/Chapter10_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.

LLVMBackendTutorialExampleCode/Chapter10_1/Cpu0RegisterInfo.td

```

//=====
// The register string, such as "9" or "gp" will show on "llvm-objdump -d"
let Namespace = "Cpu0" in {
    // General Purpose Registers
    def ZERO : Cpu0GPRReg< 0, "zero">, DwarfRegNum<[0]>;
    def AT : Cpu0GPRReg< 1, "1">, DwarfRegNum<[1]>;
    def V0 : Cpu0GPRReg< 2, "2">, DwarfRegNum<[2]>;
    def V1 : Cpu0GPRReg< 3, "3">, DwarfRegNum<[3]>;
    def A0 : Cpu0GPRReg< 4, "4">, DwarfRegNum<[6]>;
    def A1 : Cpu0GPRReg< 5, "5">, DwarfRegNum<[7]>;
    def T9 : Cpu0GPRReg< 6, "t9">, DwarfRegNum<[6]>;
    def S0 : Cpu0GPRReg< 7, "7">, DwarfRegNum<[7]>;
    def S1 : Cpu0GPRReg< 8, "8">, DwarfRegNum<[8]>;
    def S2 : Cpu0GPRReg< 9, "9">, DwarfRegNum<[9]>;
}

```

```

def GP    : Cpu0GPRReg< 10, "gp">, DwarfRegNum<[10]>;
def FP    : Cpu0GPRReg< 11, "fp">, DwarfRegNum<[11]>;
def SW    : Cpu0GPRReg< 12, "sw">, DwarfRegNum<[12]>;
def SP    : Cpu0GPRReg< 13, "sp">, DwarfRegNum<[13]>;
def LR    : Cpu0GPRReg< 14, "lr">, DwarfRegNum<[14]>;
def PC    : Cpu0GPRReg< 15, "pc">, DwarfRegNum<[15]>;
// def MAR : Register< 16, "mar">, DwarfRegNum<[16]>;
// def MDR : Register< 17, "mdr">, DwarfRegNum<[17]>;

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

=====//
// 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,
    // Callee save
    S0, S1, S2,
    // Reserved
    GP, FP,
    // Not preserved across procedure calls
    SW,
    // Reserved
    SP, LR, PC)>;

```

Run Chapter10_1/ with ch10_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 ch10_1.bc -o
ch10_1.cpu0.o
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llvm-objdump -d ch10_1.cpu0.o

```

```
ch10_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.

10.2 Verilog of CPU0

Verilog language is an IEEE standard in IC design. There are a lot of book and documents for this language. Web site¹ has a pdf² in this. Example code LLVMBackendTutorialExampleCode/cpu0s_verilog/raw/cpu0s.v is the cpu0 design in Verilog. In Appendix A, we have downloaded and installed Icarus Verilog tool both on iMac and Linux. The cpu0s.v is a simple design with only 280 lines of code. Although 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 cpu0s.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,

¹ <http://www.ece.umd.edu/courses/enee359a/>

² http://www.ece.umd.edu/courses/enee359a/verilog_tutorial.pdf

```
ST :
if (R[b]+c16 == 28672)
    $display("%4dns %8x : %8x OUTPUT=%-d", $stime, pc0, ir, R[a]);
```

LLVMBackendTutorialExampleCode/cpu0_verilog/raw/cpu0s.v

```
//`define TRACE

`define MEMSIZE 'h80000
`define MEMEMPTY 8'hFF
`define NULL     8'h00
`define IOADDR   'h80000

// 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;

    // register name
    `define PC   R[15]    // Program Counter
    `define LR   R[14]    // Link Register
    `define SP   R[13]    // Stack Pointer
    `define SW   R[12]    // Status Word
    // SW Flage
    `define C    'SW[29]   // Carry
    `define V    'SW[28]   // Overflow
    `define MODE 'SW[25:23] // itype
    `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[8]    // Hardware Interrupt 1, IO1 interrupt, Enable
    `define I1E  'SW[7]    // Hardware Interrupt 0, timer interrupt, Enable
    `define IOE  'SW[6]    // Software Interrupt Enable
    `define IE   'SW[5]    // Interrupt Enable
    `define M    'SW[4]    // Mode bit
    `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,
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;

reg [0:0] inInt = 0;
reg [2:0] state, next_state;
parameter Reset=3'h0, Fetch=3'h1, Decode=3'h2, Execute=3'h3, WriteBack=3'h4;
integer i;

task memReadStart(input [31:0] addr, input [1:0] size); begin // Read Memory Word
    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

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

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; 'I0 = 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:    memReadStart(Rb+c16, 'BYTE);           // LB Ra, [Rb+Cx]; Ra<=(byte) [Rb+Cx]
LBu:   memReadStart(Rb+c16, 'BYTE);           // LBu Ra, [Rb+Cx]; Ra<=(byte) [Rb+Cx]
SB:    memWriteStart(Rb+c16, Ra, 'BYTE); // SB Ra, [Rb+Cx]; Ra>=(byte) [Rb+Cx]
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:    memWriteStart(Rb+c16, Ra, 'INT16); // SH Ra, [Rb+Cx]; Ra>=(2bytes) [Rb+Cx]
// 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
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
// 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 `LR='PC; `PC=Ra; end // JALR Ra,Rb; Ra<=PC; PC<=Rb
RET:    begin `PC='LR; end           // RET; PC <= LR
IRET:   begin
        `PC=Ra; `I = 1'b0; `MODE = '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)
        `ifdef TRACE
        MULT, MULTu, DIV, DIVu, MTHI, MTLO :
            $display("%4dns %8x : %8x HI=%8x LO=%8x SW=%8x", $stime, pc0, ir, HI,
            LO, `SW);
        `endif
        ST : begin
            `ifdef TRACE
            $display("%4dns %8x : %8x m[%-04d+%-04d]=-d SW=%8x", $stime, pc0, ir,
            R[b], c16, R[a], `SW);
            `endif
            if (R[b]+c16 == `IOADDR) begin
                outw(R[a]);
            end
        end
        SB : begin
            `ifdef TRACE
            $display("%4dns %8x : %8x m[%-04d+%-04d]=%c SW=%8x", $stime, pc0, ir,
            R[b], c16, R[a][7:0], `SW);
            `endif
            if (R[b]+c16 == `IOADDR) begin
                outc(R[a][7:0]);
            end
        end
        `ifdef TRACE
        default :
            $display("%4dns %8x : %8x R[%02d]=-8x=%-d SW=%8x", $stime, pc0, ir, a,
            R[a], R[a], `SW);
        `endif
    endcase
    if (op==RET && `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);
        'MODE = 'RESET;
        state = Fetch;
    end else if (inInt == 0 && (state == Fetch) && ('IE && 'I) && ((`IOE && `IO) || (`IIE && `II) || `IRQ)) begin
        'MODE = 'IRQ;
        taskInterrupt('IRQ);
        state = Fetch;
    end else begin
        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];
    reg [31:0] data;

    integer i;
    initial begin
        // erase memory
        for (i=0; i < 'MEMSIZE; i=i+1) begin
            m[i] = 'MEMEMPTY;
        end
        // display memory contents
        $readmemh("cpu0s.hex", m);
        `ifdef TRACE
        for (i=0; i < 'MEMSIZE && (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
        `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;
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;
    #3000000 $finish;
end

always #10 clock=clock+1;

endmodule

JonathantekiiMac:raw Jonathan$ pwd
/Users/Jonathan/test/2/lbd/LLVMBackendTutorialExampleCode/cpu0_verilog/raw
JonathantekiiMac:raw Jonathan$ iverilog -o cpu0s cpu0s.v

```

10.3 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 cpu0s.v use 0x7000 bytes of memory.

LLVMBackendTutorialExampleCode/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 $11,     $ZERO, 0");
asm("addiu $12,     $ZERO, 0");
asm("addiu $14,     $ZERO, -1");
```

LLVMBackendTutorialExampleCode/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
```

LLVMBackendTutorialExampleCode/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;
}
```

LLVMBackendTutorialExampleCode/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();
int test_operators(int x);
int test_control1();

int main()
{
    int a = 0;
    a = test_math(); // a = 74
    print_integer(a); // a = 253
    a = test_div();
    print_integer(a); // a = 3
    a = test_local_pointer();
    print_integer(a); // a = 1
    a = (int)test_load_bool();
    print_integer(a); // a = 14
    print_integer(a);
    a = test_andorxornot(); // a = 14
    print_integer(a);
    a = test_setxx(); // a = 3
    print_integer(a);
    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

    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 "ch4_1.cpp"
#include "ch4_3.cpp"
#include "ch4_4.cpp"
#include "ch4_5.cpp"
#include "ch7_1_1.cpp"

JonathantekiiMac:InputFiles Jonathan$ pwd
/Users/Jonathan/test/2/lbd/LLVMBackendTutorialExampleCode/InputFiles
JonathantekiiMac:InputFiles Jonathan$ clang -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/cpu0s.hex

118-165-81-39:raw Jonathan$ cat cpu0s.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 Chapter10_2/ as follow,

LLVMBackendTutorialExampleCode/Chapter10_2/MCTargetDesc/Cpu0MCCCodeEmitter.cpp

```

    else
        llvm_unreachable("unexpect opcode in getJumpAbsoluteTargetOpValue()");

    return 0;
}

```

We change JSUB from Relocation Records fixup_Cpu0_24 to Non-Relocaton 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-Relocaton 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.

LLVMBackendTutorialExampleCode/Chapter10_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 Chapter10_2/ with `llvm-objdump -d` for input files `ch_run_backend.cpp` and `ch_run_sum_i.cpp` to generate the hex file and input to cpu0s Verilog simulator to get the output result as below. You can unmark the `$display()` in `cpu0s.v` to trace the memory binary code and destination register change at every instruction execution. Remind `ch_run_sum_i.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.

LLVMBackendTutorialExampleCode/InputFiles/ch_run_sum_i.cpp

```

#include "boot.cpp"

#include "print.h"

int sum_i(int amount, ...);

int main()
{
    int a = 0;
    a = sum_i(6, 0, 1, 2, 3, 4, 5);
    print_integer(a);    // a = 15

    return a;
}

#include "print.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;
}

JonathantekiiMac:InputFiles Jonathan$ clang -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/cpu0s.hex

JonathantekiiMac:raw Jonathan$ ./cpu0s
WARNING: cpu0s.v:386: $readmemh(cpu0s.hex): Not enough words in the file for the
requested range [0:28671].
taskInterrupt(001)
74
253
3
1
14
3
51
2147483647
-2147483648
RET to PC < 0, finished!

JonathantekiiMac:raw Jonathan$ cd ../../InputFiles/
JonathantekiiMac:InputFiles Jonathan$ ~/llvm/release/cmake_debug_build/bin/Debug/
clang -target mips-unknown-linux-gnu -c ch_run_sum_i.cpp -emit-llvm -o
ch_run_sum_i.bc
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=obj
ch_run_sum_i.bc -o ch_run_sum_i.cpu0.o
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llvm-objdump -d ch_run_sum_i.cpu0.o | tail -n +6 | awk '{print
"/* " $1 " */\t" $2 " " $3 " " $4 " " $5 "\t/* " $6"\t" $7" " $8" " $9" " $10
"\t*/"}' > ../cpu0_verilog/raw/cpu0s.hex
JonathantekiiMac:InputFiles Jonathan$ cd ../cpu0_verilog/raw/
JonathantekiiMac:raw Jonathan$ ./cpu0s
WARNING: cpu0s.v:386: $readmemh(cpu0s.hex): Not enough words in the file for the
```

```

requested range [0:28671].
taskInterrupt(001)
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.

```

JonathantekiiMac:raw Jonathan$ cd ../../InputFiles/
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build
/bin/Debug/llvm-objdump -d ch_run_backend.cpu0.o
...
_Z13print_integeri:
 100: 09 dd ff e8          addiu $sp, $sp, -24
 104: 02 ed 00 14          st $lr, 20($sp)
 108: 02 bd 00 10          st $fp, 16($sp)
 10c: 13 bd 00 00          add $fp, $sp, $zero
 110: 01 2b 00 18          ld $2, 24($fp)
 114: 02 2b 00 0c          st $2, 12($fp)
 118: 02 2d 00 00          st $2, 0($sp)
 11c: 2b 00 02 cc          jsub 716
 120: 02 2b 00 08          st $2, 8($fp)
 124: 02 2d 00 00          st $2, 0($sp)
 128: 2b 00 04 3c          jsub 1084
 12c: 13 db 00 00          add $sp, $fp, $zero
 130: 01 bd 00 10          ld $fp, 16($sp)
 134: 01 ed 00 14          ld $lr, 20($sp)
 138: 09 dd 00 18          addiu $sp, $sp, 24
 13c: 2c e0 00 00          ret $lr
...
_Z14print1_integeri:
 5e4: 09 dd ff f8          addiu $sp, $sp, -8
 5e8: 02 bd 00 04          st $fp, 4($sp)
 5ec: 13 bd 00 00          add $fp, $sp, $zero
 5f0: 01 2b 00 08          ld $2, 8($fp)
 5f4: 02 2b 00 00          st $2, 0($fp)
 5f8: 01 1d 00 08          ld $1, 8($sp)
 5fc: 02 10 70 00          st $1, 28672($zero)
 600: 13 db 00 00          add $sp, $fp, $zero
 604: 01 bd 00 04          ld $fp, 4($sp)
 608: 09 dd 00 08          addiu $sp, $sp, 8
 60c: 2c e0 00 00          ret $lr

```

Unmark the \$display() in cpu0s.v to trace the memory binary code and destination register change at every instruction execution as follows,

```

JonathantekiiMac:raw Jonathan$ ./cpu0s
WARNING: cpu0s.v:386: $readmemh(cpu0s.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
1850ns 00000064 : 022b0024 m[28620+36] =0           SW=10000000
1930ns 00000068 : 022b0020 m[28620+32] =0           SW=10000000
...
29370ns 000002ec : 022b0004 m[28548+4] =28672       SW=50000000
29530ns 000002f4 : 05320000 m[28672+0] = SW=50000000
15
RET to PC < 0, finished!
```

As above result, cpu0s.v dump the memory first after read input cpu0s.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 the “>>” is correct on both signed and unsigned int type , and tracking the variable **a** value by print_integer(). The output value **a** is 15 in this program.

We 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.

11.1 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.

Chapter11_1/ support this optimization algorithm include the added codes as follows,

LLVMBackendTutorialExampleCode/Chapter11_1/CMakeLists.txt

```
add_llvm_target(Cpu0CodeGen
...
Cpu0DelUselessJMP.cpp
...
)
```

LLVMBackendTutorialExampleCode/Chapter11_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;
}
```

LLVMBackendTutorialExampleCode/Chapter11_1/Cpu0DelUselessJMP.cpp

```
===== Cpu0DelUselessJMP.cpp - Cpu0 DelJmp =====
//
//          The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//=====
// Simple pass to fills delay slots with useful instructions.
//
//=====

#define DEBUG_TYPE "del-jmp"

#include "Cpu0.h"
#include "Cpu0TargetMachine.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/TargetInstrInfo.h"
#include "llvm/ADT/SmallSet.h"
#include "llvm/ADT/Statistic.h"

using namespace llvm;

STATISTIC(NumDelJmp, "Number of useless jmp deleted");

static cl::opt<bool> EnableDelJmp(
    "enable-cpu0-del-useless-jmp",
    cl::init(true),
    cl::desc("Delete useless jmp instructions: jmp 0."),
    cl::Hidden);

namespace {
    struct DelJmp : public MachineFunctionPass {

        TargetMachine &TM;
        const TargetInstrInfo *TII;

        static char ID;
        DelJmp(TargetMachine &tm)
            : MachineFunctionPass(ID), TM(tm), TII(tm.getInstrInfo()) { }

        virtual const char *getPassName() const {
            return "Cpu0 Del Useless jmp";
        }
    };
}
```

```

bool runOnMachineBasicBlock(MachineBasicBlock &MBB, MachineBasicBlock &MBBN);
bool runOnMachineFunction(MachineFunction &F) {
    bool Changed = false;
    if (EnableDelJmp) {
        MachineFunction::iterator FJ = F.begin();
        if (FJ != F.end())
            FJ++;
        if (FJ == F.end())
            return Changed;
        for (MachineFunction::iterator FI = F.begin(), FE = F.end();
              FJ != FE; ++FI, ++FJ)
            // In STL style, F.end() is the dummy BasicBlock() like '\0' in
            // C string.
            // FJ is the next BasicBlock of FI; When FI range from F.begin() to
            // the PreviousBasicBlock of F.end() call runOnMachineBasicBlock().
            Changed |= runOnMachineBasicBlock(*FI, *FJ);
    }
    return Changed;
}

};

char DelJmp::ID = 0;
} // end of anonymous namespace

bool DelJmp::
runOnMachineBasicBlock(MachineBasicBlock &MBB, MachineBasicBlock &MBBN) {
    bool Changed = false;

    MachineBasicBlock::iterator I = MBB.end();
    if (I != MBB.begin())
        I--;
    // set I to the last instruction
    else
        return Changed;

    if (I->getOpcode() == Cpu0::JMP && I->getOperand(0).getMBB() == &MBBN) {
        // I is the instruction of "jmp #offset=0", as follows,
        //     jmp      $BB0_3
        // $BB0_3:
        //     ld      $4, 28($sp)
        ++NumDelJmp;
        MBB.erase(I);
        Changed = true; // Notify LLVM kernel Changed
    }
    return Changed;
}

/// createCpu0DelJmpPass - Returns a pass that DelJmp in Cpu0 MachineFunctions
FunctionPass *llvm::createCpu0DelJmpPass(Cpu0TargetMachine &tm) {
    return new DelJmp(tm);
}

```

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 Chapter11_1/ with ch11_1.cpp to explain it easier.

LLVMBackendTutorialExampleCode/InputFiles/ch11_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 -c ch11_1.cpp -emit-llvm -o ch11_1.bc
118-165-78-10:InputFiles Jonathan$ clang -target `llvm-config --host-target` -c ch11_1.cpp -emit-llvm -o ch11_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 ch11_1.bc -o ch11_1.cpu0.s
=====
          ... Statistics Collected ...
=====
...
2 del-jmp      - Number of useless jmp deleted
...

.section .mdebug.abi32
.previous
.file "ch11_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 `ch7_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.

11.2 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.

11.2.1 Cpu0 new instruction sets table

Redesign Cpu0 instruction set and remap OP code as follows (OP code 0x00 is reserved for NOP operation in pipeline architecture),

- First column F.: meaning Format.

Table 11.1: Cpu0 Instruction Set :widths: 1 4 3 11 7 10 :header-rows: 1

F.	Mnemonic	Opcode	Meaning	Syntax	Operation
L	NOP	00	No Operation		
L	LD	01	Load word	LD Ra, [Rb+Cx]	Ra <= [Rb+Cx]
L	ST	02	Store word	ST Ra, [Rb+Cx]	[Rb+Cx] <= Ra
L	LB	03	Load byte	LB Ra, [Rb+Cx]	Ra <= (byte)[Rb+Cx]
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	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 ¹
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 ¹
A	SHLV	21	Shift left	SHLV Ra, Rb, Rc	Ra <= Rb << Rc
A	SHRV	22	Shift right	SHRV Ra, Rb, Rc	Ra <= Rb >> Rc
L	BEQ	30	Jump if equal	BEQ Ra, Rb, Cx	if (Ra==Rb), PC <= PC + Cx
L	BNE	31	Jump if not equal	BNE Ra, Rb, Cx	if (Ra!=Rb), PC <= PC + Cx
J	JMP	36	Jump (unconditional)	JMP Cx	PC <= PC + Cx

Continued on next page

¹ Rb ' >> Cx, Rb ' >> Rc: Shift with signed bit remain. It's equal to ((Rb&'h80000000)|Rb>>Cx) or ((Rb&'h80000000)|Rb>>Rc).

Table 11.1 – continued from previous page

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	Jump to subroutine	JR Rb	LR <= PC; PC <= Rb
L	SLTi	26	Set less Then	SLTi Ra, Rb, Cx	Ra <= (Rb < Cx)
L	SLTi _u	27	SLTi unsigned	SLTi _u Ra, Rb, Cx	Ra <= (Rb < Cx)
A	SLT	28	Set less Then	SLT Ra, Rb, Rc	Ra <= (Rb < Rc)
A	SLTu	29	SLT unsigned	SLTu Ra, Rb, Rc	Ra <= (Rb < Rc)
L	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
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

As above, the OPu, such as ADDu is for unsigned integer or No Trigger Exception. The LUi for example, “LUi \$2, 0x7000”, load 0x700 to high 16 bits of \$2 and fill the low 16 bits of \$2 to 0x0000.

11.2.2 Cpu0 code changes

Chapter11_2/ include the changes for new instruction sets as follows,

LLVMBackendTutorialExampleCode/Chapter11_2/AsmParser/Cpu0AsmParser.cpp

```
// Cpu0AsmParser.cpp
...
int Cpu0AsmParser::matchRegisterName(StringRef Name) {
    ...
    .Case("t0", Cpu0::T0)
    ...
}
```

LLVMBackendTutorialExampleCode/Chapter11_2/Disassembler/Cpu0Disassembler.cpp

```
// Decoder tables for Cpu0 register
static const unsigned CPUREgsTable[] = {
// Change SW to T0 which is a caller saved
    Cpu0::T0, ...
};

// DecodeCMPInstruction() function is removed since No CMP instruction.
...

// Change DecodeBranchTarget() to following for 16 bit offset
static DecodeStatus DecodeBranchTarget(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;
}

```

LLVMBackendTutorialExampleCode/Chapter11_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 },
...
}

```

LLVMBackendTutorialExampleCode/Chapter11_2/MCTargetDesc/Cpu0BaseInfo.cpp

```

inline static unsigned getCpu0RegisterNumbering(unsigned RegEnum)
{
    switch (RegEnum) {
...
    case Cpu0::T0:
...
}

```

LLVMBackendTutorialExampleCode/Chapter11_2/MCTargetDesc/Cpu0FixupKinds.cpp

```

enum Fixups {
...
// PC relative branch fixup resulting in - R_CPU0_PC16.
// cpu0 PC16, e.g. beq
fixup_Cpu0_PC16,

```

```
}; ...
```

LLVMBackendTutorialExampleCode/Chapter11_2/MCTargetDesc/Cpu0MCCodeEmitter.cpp

```
unsigned Cpu0MCCodeEmitter::  
getBranchTargetOpValue(const MCInst &MI, unsigned OpNo,  
                      SmallVectorImpl<MCFixup> &Fixups) const {  
    ...  
    Fixups.push_back(MCFixup::Create(0, Expr,  
                                     MCFixupKind(Cpu0::fixup_Cpu0_PC16)));  
    return 0;  
}  
...  
unsigned Cpu0MCCodeEmitter::  
getJumpTargetOpValue(const MCInst &MI, unsigned OpNo,  
                     SmallVectorImpl<MCFixup> &Fixups) const {  
    ...  
    if (Opcode == Cpu0::JSUB || Opcode == Cpu0::JMP)  
    ...  
}
```

LLVMBackendTutorialExampleCode/Chapter11_2/Cpu0InstrInfo.cpp

```
// Cpu0InstrInfo::copyPhysReg()  
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 (Cpu0::CPURegsRegClass.contains(SrcReg))  
            Opc = Cpu0::ADD, ZeroReg = Cpu0::ZERO;  
        else if (SrcReg == Cpu0::HI)  
            Opc = Cpu0::MFHI, SrcReg = 0;  
        else if (SrcReg == Cpu0::LO)  
            Opc = Cpu0::MFLO, SrcReg = 0;  
    }  
    else if (Cpu0::CPURegsRegClass.contains(SrcReg)) { // Copy from CPU Reg.  
        if (DestReg == Cpu0::HI)  
            Opc = Cpu0::MTHI, DestReg = 0;  
        else if (DestReg == Cpu0::LO)  
            Opc = Cpu0::MTLO, DestReg = 0;  
    }  
  
    assert(Opc && "Cannot copy registers");  
  
    MachineInstrBuilder MIB = BuildMI(MBB, I, DL, get(Opc));  
  
    if (DestReg)  
        MIB.addReg(DestReg, RegState::Define);  
  
    if (ZeroReg)
```

```

    MIB.addReg(ZeroReg);

    if (SrcReg)
        MIB.addReg(SrcReg, getKillRegState(KillSrc));
}

```

LLVMBackendTutorialExampleCode/Chapter11_2/Cpu0InstrInfo.td

```

def jmptarget : Operand<OtherVT> {
    let EncoderMethod = "getJumpTargetOpValue";
    let OperandType = "OPERAND_PCREL";
    let DecoderMethod = "DecodeJumpRelativeTarget";
}

...
// 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);
]>;
...

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;
}

// Conditional Branch
class CBranch<bits<8> op, string instr_asm, PatFrag cond_op, RegisterClass RC>:
    FL<op, (outs), (ins RC:$ra, RC:$rb, brtarget:$imm16),
        !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
        [(brcond (i32 (cond_op RC:$ra, RC:$rb)), bb:$imm16)], IIBranch> {
    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> {
    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>;
// Unconditional branch, such as JMP
class UncondBranch<bits<8> op, string instr_asm>:

```

```

FJ<op, (outs), (ins jmptarget:$addr),
    !strconcat(instr_asm, "\t$addr"), [(br bb:$addr)], IIBranch> {
let isBranch = 1;
let isTerminator = 1;
let isBarrier = 1;
let hasDelaySlot = 0;
}
...
def 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, "beq", seteq, CPURegs>;
def BNE      : CBranch<0x31, "bne", setne, CPURegs>;

// brcond patterns
multiclass BrcondPats<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:$dst),
            (BEQ (SLTiuOp RC:$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)>;
}

defm : BrcondPats<CPURegs, BEQ, BNE, SLT, SLTu, SLTi, SLTiu, ZERO>;

// setcc patterns
multiclass SeteqPats<RegisterClass RC, Instruction SLTiuOp, Instruction XOROp,
                      Instruction SLTuOp, Register ZEROReg> {
// a == b
def : Pat<(seteq RC:$lhs, RC:$rhs),
            (SLTiuOp (XOROp RC:$lhs, RC:$rhs), 1)>;
// a != b
def : Pat<(setne RC:$lhs, RC:$rhs),
            (SLTuOp ZEROReg, (XOROp RC:$lhs, RC:$rhs))>;
}

```

```

// a <= b
multiclass SetlePats<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 SetgtPats<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 SetgePats<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 SetgeImmPats<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)>;
}

defm : SeteqPats<CPUREgs, SLTi, XOR, SLTu, ZERO>;
defm : SetlePats<CPUREgs, SLT, SLTu>;
defm : SetgtPats<CPUREgs, SLT, SLTu>;
defm : SetgePats<CPUREgs, SLT, SLTu>;
defm : SetgeImmPats<CPUREgs, SLTi, SLTi>;

```

LLVMBackendTutorialExampleCode/Chapter11_2/Cpu0RegisterInfo.td

```

let Namespace = "Cpu0" in {
    ...
    def T0      : Cpu0GPRReg< 12, "t0">,    DwarfRegNum<[12]>;
    ...

def CPUREgs : RegisterClass<"Cpu0", [i32], 32, (add
    T0,
    // Reserved
    SP, LR, PC)>;

// Remove SR RegisterClass since no SW in General register
// Status Registers

```

```
/* def SR  : RegisterClass<"Cpu0", [i32], 32, (add SW)>; */
```

As modified from above, it remove the CMP instruction, SW register and related code from Chapter11_1/, and change from JEQ 24bits offset to BEQ 16 bits offset.

11.2.3 Cpu0 Verilog language changes

LLVMBackendTutorialExampleCode/cpu0_verilog/redesign/cpu0s.v

```
//`define TRACE

`define MEMSIZE 'h80000
`define MEMEMPTY 8'hFF
`define NULL     8'h00
`define IOADDR   'h80000

// 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], SW;
    // HI, LO: High and Low part of 64 bit result
    // SW: Status Word
    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;

    // register name
    `define PC  R[15]    // Program Counter
    `define LR  R[14]    // Link Register
    `define SP  R[13]    // Stack Pointer
    // SW Flage
    `define C   SW[29]   // Carry
    `define V   SW[28]   // Overflow
    `define MODE SW[25:23] // itype
    `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[8]    // Hardware Interrupt 1, IO1 interrupt, Enable
    `define I1E SW[7]    // Hardware Interrupt 0, timer interrupt, Enable
    `define IOE SW[6]    // Software Interrupt Enable
```

```

`define IE    SW[5]  // Interrupt Enable
`define M     SW[4]  // Mode bit
`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,
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,
SLTi=8'h26,SLTiU=8'h27, SLT=8'h28,SLTu=8'h29,
BEQ=8'h30,BNE=8'h31,
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;

reg [0:0] inInt = 0;
reg [2:0] state, next_state;
parameter Reset=3'h0, Fetch=3'h1, Decode=3'h2, Execute=3'h3, WriteBack=3'h4;
integer i;

task memReadStart(input [31:0] addr, input [1:0] size); begin // Read Memory Word
    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

```

```

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

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
    endcase
end

```

```

end
Execute: begin // Tick 3 : instruction execution
  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:   memReadStart(Rb+c16, 'BYTE);             // LB Ra, [Rb+Cx]; Ra<=(byte) [Rb+Cx]
    LBu:  memReadStart(Rb+c16, 'BYTE);             // LBu Ra, [Rb+Cx]; Ra<=(byte) [Rb+Cx]
    SB:   memWriteStart(Rb+c16, Ra, 'BYTE);          // SB Ra, [Rb+Cx]; Ra=>(byte) [Rb+Cx]
    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:   memWriteStart(Rb+c16, Ra, 'INT16);          // SH Ra, [Rb+Cx]; Ra=>(2bytes) [Rb+Cx]
    // Mathematic
    ADDiu: R[a] = Rb+c16;                         // ADDiu Ra, Rb+Cx; Ra<=Rb+Cx
    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;
    // set
    SLT:   if (Rb < Rc) R[a]=1; else R[a]=0;
    SLTu:  if (Rb < Rc) R[a]=1; else R[a]=0;
    SLTi:  if (Rb < c16) R[a]=1; else R[a]=0;
    SLTi:  if (Rb < c16) R[a]=1; else R[a]=0;
    // Branch Instructions
    BEQ:   if (Ra==Rb) 'PC='PC+c16;
    BNE:   if (Ra!=Rb) 'PC='PC+c16;
    MFLO:  regSet(a, LO);                      // MFLO Ra; Ra<=LO
    MFHI:  regSet(a, HI);                      // MFHI Ra; Ra<=HI
    MTLO:  LO = Ra;                          // MTLO Ra; LO<=Ra

```

```

MTHI:  HI = Ra;                                // MTHI Ra; HI<=Ra
MULT:  {HI, LO}=Ra*Rb;                          // MULT Ra,Rb; HI<=((Ra*Rb)>>32);
                                                // LO<=((Ra*Rb) and 0x00000000ffffffffff);
                                                // with exception overflow
MULTU: {HI, LO}=URa*URb;                      // MULT URa,URb; HI<=((URa*URb)>>32);
                                                // LO<=((URa*URb) and 0x00000000ffffffffff);
                                                // without exception overflow

// Jump Instructions
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 'LR='PC; 'PC=Ra; end // JALR Ra,Rb; Ra<=PC; PC<=Rb
RET:   begin 'PC='LR; end                      // RET; PC <= LR
IRET:  begin
        'PC=Ra; 'I = 1'b0; 'MODE = '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)
        'ifdef TRACE
        MULT, MULTu, DIV, DIVu, MTHI, MTLO :
            $display("%4dns %8x : %8x HI=%8x LO=%8x SW=%8x", $stime, pc0, ir, HI,
                    LO, SW);
        'endif
        ST : begin
            'ifdef TRACE
                $display("%4dns %8x : %8x m[%-04d+%-04d]=%-d SW=%8x", $stime, pc0, ir,
                    R[b], c16, R[a], SW);
            'endif
            if (R[b]+c16 == 'IOADDR) begin
                outw(R[a]);
            end
        end
        SB : begin
            'ifdef TRACE
                $display("%4dns %8x : %8x m[%-04d+%-04d]=%c SW=%8x", $stime, pc0, ir,
                    R[b], c16, R[a][7:0], SW);
            'endif
            if (R[b]+c16 == 'IOADDR) begin
                outc(R[a][7:0]);
            end
        end
        'ifdef TRACE
    default :
        $display("%4dns %8x : %8x R[%02d]=-8x=%-d SW=%8x", $stime, pc0, ir, a,
            R[a], R[a], SW);
    end

```

```

`endif
endcase
if (op==RET && `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);
        'MODE = 'RESET;
        state = Fetch;
    end else if (inInt == 0 && (state == Fetch) && ('IE && 'I) && ((`IOE && `IO) || ('IIE && 'II) || 'IRQ;
        taskInterrupt('IRQ);
        state = Fetch;
    end else begin
        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];
    reg [31:0] data;

    integer i;
    initial begin
        // erase memory
        for (i=0; i < 'MEMSIZE; i=i+1) begin
            m[i] = 'MEMEMPTY;
        end
        // display memory contents
        $readmemh("cpu0s.hex", m);
        `ifdef TRACE
        for (i=0; i < 'MEMSIZE && (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
        `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;
                end
            end
        end
    end
endmodule

```

```

        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;
    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;
        #3000000 $finish;
    end

    always #10 clock=clock+1;

endmodule

```

11.2.4 Run the redesigned Cpu0

Run Chapter11_2/ with ch_run_backend.cpp to get result as below. It match the expect value as comment in ch_run_backend.cpp.

LLVMBackendTutorialExampleCode/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();
int test_operators(int x);
int test_control1();

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);
    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

    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 "ch4_1.cpp"
#include "ch4_3.cpp"
#include "ch4_4.cpp"
#include "ch4_5.cpp"
#include "ch7_1_1.cpp"

118-165-77-203:InputFiles Jonathan$ clang -target `llvm-config --host-target` \
-c ch_run_backend.cpp -emit-llvm -o ch_run_backend.bc
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/cpu0s.hex

JonathantekiiMac:InputFiles Jonathan$ cd ../cpu0_verilog/redesign/
JonathantekiiMac:redesign Jonathan$ iverilog -o cpu0s cpu0s.v
JonathantekiiMac:redesign Jonathan$ ./cpu0s
WARNING: cpu0s.v:396: $readmemh(cpu0s.hex): Not enough words in the file for the
requested range [0:28671].
taskInterrupt(001)
1
2
1073741821
0
128
146
RET to PC < 0, finished!

```

Run with ch7_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 -relocation-model=static -filetype=asm ch7_1_1.bc -o
ch7_1_1.cpu0.s
118-165-77-203:InputFiles Jonathan$ cat ch7_1_1.cpu0.s
.section .mdebug.abi32
.previous
.file "ch7_1_1.bc"
.text
.globl main
.align 2
.type main,@function
.ent main          # @main
main:
.cfi_startproc
.frame $sp,40,$lr
.mask 0x00000000,0
.set noreorder
.set nomacro

```

```
# BB#0:
addiu $sp, $sp, -40
$tmp1:
.cfi_def_cfa_offset 40
addiu $3, $zero, 0
st $3, 36($sp)
st $3, 32($sp)
addiu $2, $zero, 1
st $2, 28($sp)
addiu $4, $zero, 2
st $4, 24($sp)
addiu $4, $zero, 3
st $4, 20($sp)
addiu $4, $zero, 4
st $4, 16($sp)
addiu $4, $zero, 5
st $4, 12($sp)
addiu $4, $zero, 6
st $4, 8($sp)
addiu $4, $zero, 7
st $4, 4($sp)
addiu $4, $zero, 8
st $4, 0($sp)
ld $4, 32($sp)
bne $4, $zero, $BB0_2
# BB#1:
ld $4, 32($sp)
addiu $4, $4, 1
st $4, 32($sp)
$BB0_2:
ld $4, 28($sp)
beq $4, $zero, $BB0_4
# BB#3:
ld $4, 28($sp)
addiu $4, $4, 1
st $4, 28($sp)
$BB0_4:
ld $4, 24($sp)
slti $4, $4, 1
bne $4, $zero, $BB0_6
# BB#5:
ld $4, 24($sp)
addiu $4, $4, 1
st $4, 24($sp)
$BB0_6:
ld $4, 20($sp)
slti $4, $4, 0
bne $4, $zero, $BB0_8
# BB#7:
ld $4, 20($sp)
addiu $4, $4, 1
st $4, 20($sp)
$BB0_8:
ld $4, 16($sp)
addiu $5, $zero, -1
slt $4, $5, $4
bne $4, $zero, $BB0_10
# BB#9:
```

```

ld  $4, 16($sp)
addiu $4, $4, 1
st  $4, 16($sp)
$BB0_10:
ld  $4, 12($sp)
slt $3, $3, $4
bne $3, $zero, $BB0_12
# BB#11:
ld  $3, 12($sp)
addiu $3, $3, 1
st  $3, 12($sp)
$BB0_12:
ld  $3, 8($sp)
slt $2, $2, $3
bne $2, $zero, $BB0_14
# BB#13:
ld  $2, 8($sp)
addiu $2, $2, 1
st  $2, 8($sp)
$BB0_14:
ld  $2, 4($sp)
slti $2, $2, 1
bne $2, $zero, $BB0_16
# BB#15:
ld  $2, 4($sp)
addiu $2, $2, 1
st  $2, 4($sp)
$BB0_16:
ld  $2, 4($sp)
ld  $3, 0($sp)
slt $2, $3, $2
beq $2, $zero, $BB0_18
# BB#17:
ld  $2, 0($sp)
addiu $2, $2, 1
st  $2, 0($sp)
$BB0_18:
ld  $2, 28($sp)
ld  $3, 32($sp)
beq $3, $2, $BB0_20
# BB#19:
ld  $2, 32($sp)
addiu $2, $2, 1
st  $2, 32($sp)
$BB0_20:
ld  $2, 32($sp)
addiu $sp, $sp, 40
ret $lr
.set macro
.set reorder
.end main
$tmp2:
.size main, ($tmp2)-main
.cfi_endproc

```

The ch11_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 ¹. LLD install requirement on Linux ². 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.

12.1 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 2013/08/16, so please checkout the last commit of 2013/08/15 of llvm and lld or the commit id are da44b4f68bcf2adcb74214670a266b43a1a6888f(llvm) 014d684d27a0f520a30285051a5c8194c87e0194(lld) 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 log
...
Date: Fri Aug 16 00:15:20 2013 +0000
```

InstCombine: Simplify if(x!=0 && x!=-1).

When both constants are positive or both constants are negative, InstCombine already simplifies comparisons like this, but when it's exactly zero and -1, the operand sorting ends up reversed

¹ <http://lld.llvm.org/>

² http://lld.llvm.org/getting_started.html#on-unix-like-systems

and the pattern fails to match. Handle that special case.

Follow up for rdar://14689217

```
git-svn-id: https://llvm.org/svn/llvm-project/llvm/trunk@188512 91177308-0d3
```

```
commit da44b4f68bcf2adcb74214670a266b43a1a6888f
```

```
Author: Hans Wennborg <hans@hanshq.net>
```

```
Date: Thu Aug 15 23:44:31 2013 +0000
```

```
[Gamma@localhost src]$ git checkout da44b4f68bcf2adcb74214670a266b43a1a6888f
Note: checking out 'da44b4f68bcf2adcb74214670a266b43a1a6888f'.
```

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 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 src]$ git log
```

```
...
```

```
Date: Wed Aug 21 22:57:10 2013 +0000
```

```
add InputGraph functionality
```

```
git-svn-id: https://llvm.org/svn/llvm-project/lld/trunk@188958 91177308-0d34
```

```
commit 014d684d27a0f520a30285051a5c8194c87e0194
```

```
Author: Hans Wennborg <hans@hanshq.net>
```

```
Date: Tue Aug 13 21:44:44 2013 +0000
```

```
[Gamma@localhost lld]$ git checkout 014d684d27a0f520a30285051a5c8194c87e0194
```

```
Note: checking out '014d684d27a0f520a30285051a5c8194c87e0194'.
```

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 2013/08/16 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/
```

```

LLVMBackendTutorialExampleCode/3.4_20130816_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          NVPTX           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/
LLVMBackendTutorialExampleCode/3.4_20130816_Chapter11_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/
LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/CMakeLists.txt ~/test/lbd/
docs/BackendTutorial/LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/
ELFLinkingContext.cpp ~/test/lbd/docs/BackendTutorial/
LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/Targets.h .
[Gamma@localhost ELF]$ cp -rf ~/test/lbd/docs/BackendTutorial/
LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/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/
LLVMBackendTutorialExampleCode/llvm-objdump/* .

```

Now, build llvm/lld 2013/08/16 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
```

12.2 Cpu0 lld

The code added on lld to support Cpu0 ELF as follows,

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/CMakeLists.txt

```
target_link_libraries(lldELF  
...  
lldCpu0ELFTarget  
)
```

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/ELFLinkingContext.cpp

```
uint16_t ELFLinkingContext::getOutputMachine() const {  
    switch (getTriple().getArch()) {  
        ...  
        case llvm::Triple::cpu0:  
            return llvm::ELF::EM_CPU0;  
        ...  
    }  
}
```

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/Targets.h

```
#include "Cpu0/Cpu0Target.h"
```

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/Resolver.cpp

```
bool Resolver::checkUndefines(bool final) {  
    ...  
    if (_context.printRemainingUndefines()) {  
        if (undefAtom->name() == "_start") { // cschen debug  
            foundUndefines = false;  
            continue;  
        }  
        ...  
    }
```

```

        }
        ...
    }
}
```

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/Cpu0/CMakeLists.txt

```

add_lld_library(lldCpu0ELFTarget
    Cpu0LinkingContext.cpp
    Cpu0TargetHandler.cpp
    Cpu0RelocationHandler.cpp
)

target_link_libraries(lldCpu0ELFTarget
    lldCore
)
```

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/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 {

#if 1
/// \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,
};

#endif

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 &) const;

// 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_RELATIVE:
        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:
#ifndef O
            case llvm::ELF::R_CPU0_RELATIVE:
                return true;
#endif
        default:
            return false;
    }
}

virtual ErrorOr<Reference::Kind> relocKindFromString(StringRef str) const;
virtual ErrorOr<std::string> stringFromRelocKind(Reference::Kind kind) const;

};

} // end namespace elf
} // end namespace lld

#endif

```

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/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 "Atoms.h"
#include "Cpu0LinkingContext.h"

#include "lld/Core/File.h"
#include "lld/Core/Instrumentation.h"
#include "lld/Core/Parallel.h"
#include "lld/Core/Pass.h"
#include "lld/Core/PassManager.h"
#include "lld/ReaderWriter/Simple.h"

#include "llvm/ADT/ArrayRef.h"
#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/StringSwitch.h"

using namespace lld;
#if 1
using namespace lld::elf;

namespace {
using namespace llvm::ELF;

// .got values
const uint8_t cpu0GotAtomContent[8] = { 0 };

// .plt value (entry 0)
const uint8_t cpu0Plt0AtomContent[16] = {
    0xff, 0x35, 0x00, 0x00, 0x00, 0x00, // pushq GOT+8(%rip)
    0xff, 0x25, 0x00, 0x00, 0x00, 0x00, // jmp *GOT+16(%rip)
    0x90, 0x90, 0x90, 0x90           // nopnopnop
};

// .plt values (other entries)
const uint8_t cpu0PltAtomContent[16] = {
    0xff, 0x25, 0x00, 0x00, 0x00, 0x00, // jmpq *gotatom(%rip)
    0x68, 0x00, 0x00, 0x00, 0x00,      // pushq reloc-index
    0xe9, 0x00, 0x00, 0x00, 0x00       // jmpq plt[-1]
};

/// \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, 8);
    }
};

=====
```

```

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);
    }
};

class ELFPassFile : public SimpleFile {
public:
    ELFPassFile(const ELFLinkingContext &eti) : SimpleFile(eti, "ELFPassFile") {}

    llvm::BumpPtrAllocator _alloc;
};

/// \brief Create GOT and PLT entries for relocations. Handles standard GOT/PLT
/// along with IFUNC and TLS.
template <class Derived> class GOTPLTPass : public Pass {
    /// \brief Handle a specific reference.
    void handleReference(const DefinedAtom &atom, const Reference &ref) {
        switch (ref.kind()) {
#ifndef 0
        case R_CPU0_PLT32:
            static_cast<Derived *>(this)->handlePLT32(ref);
            break;
#endif
        case R_CPU0_PC24:
            static_cast<Derived *>(this)->handlePC24(ref);
            break;
#ifndef 0
        case R_CPU0_GOTPOFF: // GOT Thread Pointer Offset
            static_cast<Derived *>(this)->handleGOTPOFF(ref);
            break;
        case R_CPU0_GOTPCREL:
            static_cast<Derived *>(this)->handleGOTPCREL(ref);
            break;
#endif
    }
};

protected:
    /// \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);
#endif 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;
}

/// \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 RELGOT relocation to the original target resolver.
ErrorOr<void> handleIFUNC(const Reference &ref) {
    auto target = dyn_cast_or_null<const DefinedAtom>(ref.target());
    if (target && target->contentType() == DefinedAtom::typeResolver)
        const_cast<Reference &>(ref).setTarget(getIFUNCPLTEntry(target));
    return error_code::success();
}

/// \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);
#endif 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 TLS_TPREL32 GOT entry and change the relocation to a PC24 to
/// the GOT.
void handleGOTPOFF(const Reference &ref) {
    const_cast<Reference &>(ref).setTarget(getGOTPOFF(ref.target()));
    const_cast<Reference &>(ref).setKind(R_CPU0_PC24);
}

/// \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;
}

/// \brief Handle a GOTPCREL relocation to an undefined weak atom by using a
/// null GOT entry.
void handleGOTPCREL(const Reference &ref) {
    const_cast<Reference &>(ref).setKind(R_CPU0_PC24);
    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));
}

public:
GOTPLTPass(const ELFLinkingContext &ti)
: _file(ti), _null(nullptr), _PLT0(nullptr), _got0(nullptr),
_got1(nullptr) {}

/// \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(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 (_PLT0) {
        _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++);
        _got1->setOrdinal(ordinal++);
        mf.addAtom(*_got0);
        mf.addAtom(*_got1);
    }
    for (auto &got : _gotVector) {
        got->setOrdinal(ordinal++);
        mf.addAtom(*got);
    }
}

protected:
/// \brief Owner of all the Atoms created by this pass.
ELFPassFile _file;

/// \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;

/// \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;
GOTAtom *_got1;
/// @}
};

/// 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 StaticGOTPLTPass LLVM_FINAL : public GOTPLTPass<StaticGOTPLTPass> {
public:
    StaticGOTPLTPass(const elf::Cpu0LinkingContext &ti) : GOTPLTPass(ti) {}
}

```

```

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> handlePC24(const Reference &ref) { return handleIFUNC(ref); }

class DynamicGOTPLTPass LLVM_FINAL : public GOTPLTPass<DynamicGOTPLTPass> {
public:
    DynamicGOTPLTPass(const elf::Cpu0LinkingContext &ti) : GOTPLTPass(ti) {}

    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");
        _got1 = new (_file._alloc) Cpu0GOTAtom(_file, ".got.plt");
        _PLT0->addReference(R_CPU0_PC24, 2, _got0, -4);
        _PLT0->addReference(R_CPU0_PC24, 8, _got1, -4);
#ifndef NDEBUG
        _got0->_name = "__got0";
        _got1->_name = "__got1";
#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");
        pa->addReference(R_CPU0_PC24, 2, ga, -4);
        pa->addReference(LLD_R_CPU0_GOTRELINDEX, 7, ga, 0);
        pa->addReference(R_CPU0_PC24, 12, getPLT0(), -4);
        // Set the starting address of the got entry to the second instruction in
        // the plt entry.
        ga->addReference(R_CPU0_32, 0, pa, 6);
#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> handlePLT32(const Reference &ref) {
// Turn this into a PC24 to the PLT entry.
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);
if (isa<const SharedLibraryAtom>(ref.target()))
    const_cast<Reference &>(ref).setTarget(getPLTEEntry(ref.target()));
return error_code::success();
}

ErrorOr<void> handlePC24(const Reference &ref) {
if (ref.target() && isa<SharedLibraryAtom>(ref.target()))
    return handlePLT32(ref);
return handleIFUNC(ref);
}

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;
}

void handleGOTPCREL(const Reference &ref) {
const_cast<Reference &>(ref).setKind(R_CPU0_PC24);
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));
}
};

} // end anon namespace

void elf::Cpu0LinkingContext::addPasses(PassManager &pm) const {
switch (_outputFileType) {
case llvm::ELF::ET_EXEC:

```

```

if (_isStaticExecutable)
    pm.add(std::unique_ptr<Pass> (new StaticGOTPLTPass(*this)));
else
    pm.add(std::unique_ptr<Pass> (new DynamicGOTPLTPass(*this)));
break;
case llvm:::ELF::ET_DYN:
    pm.add(std::unique_ptr<Pass> (new DynamicGOTPLTPass(*this)));
    break;
case llvm:::ELF::ET_REL:
    break;
default:
    llvm_unreachable("Unhandled output file type");
}
ELFLinkingContext::addPasses(pm);
}
#endif

#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_16)
    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_CALL24)
    .Case("LLD_R_CPU0_GOTRELINDEX", LLD_R_CPU0_GOTRELINDEX)
    .Default(-1);

    if (ret == -1)
        return make_error_code(yaml_reader_error::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_16)
    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_CALL24)
    case LLD_R_CPU0_GOTRELINDEX:

```

```
    return std::string("LLD_R_CPU0_GOTRELINDEX");
}

return make_error_code(yaml_reader_error::illegal_value);
}
```

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/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
```

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/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"

using namespace lld;
using namespace elf;

namespace {
/// \brief R_CPU0_HI16 - word64: (S + A) >> 16
void relocHI16(uint8_t *location, uint64_t P, uint64_t S, int64_t A) {
// uint32_t result = (uint32_t)((S + A) >> 16); // Don't know why 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) {
// uint32_t result = (uint32_t)((S + A) & 0x0000ffff);
    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_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 + A) - P);
    uint32_t result = (uint32_t)(S - P);
    *reinterpret_cast<llvm::support::ubig32_t *>(location) =
        result +
        (uint32_t) * reinterpret_cast<llvm::support::ubig32_t *>(location);
}

/// \brief R_CPU0_32 - word32: S + A
void reloc32(uint8_t *location, uint64_t P, uint64_t S, int64_t A) {
// int32_t result = (int32_t)(S + A);
    int32_t result = (int32_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;
}

ErrorOr<void> Cpu0TargetRelocationHandler::applyRelocation(
    ELFWriter &writer, llvm::FileOutputBuffer &buf, const lld::AtomLayout &atom,
    const Reference &ref) const {
    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();

    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;
        case R_CPU0_PC24:
            relocPC24(location, relocVAddress, targetVAddress, ref.addend());
            break;
        case R_CPU0_32:
            reloc32(location, relocVAddress, targetVAddress, ref.addend());
            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();
}
```

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/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 "Atoms.h"
#include "Cpu0LinkingContext.h"

#include "lld/Core/File.h"
#include "lld/Core/Instrumentation.h"
#include "lld/Core/Parallel.h"
#include "lld/Core/Pass.h"
#include "lld/Core/PassManager.h"
#include "lld/ReaderWriter/Simple.h"

#include "llvm/ADT/ArrayRef.h"
#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/StringSwitch.h"

using namespace lld;
#if 1
using namespace lld::elf;

namespace {
using namespace llvm::ELF;

// .got values
const uint8_t cpu0GotAtomContent[8] = { 0 };

// .plt value (entry 0)
const uint8_t cpu0Plt0AtomContent[16] = {
    0xff, 0x35, 0x00, 0x00, 0x00, 0x00, // pushq GOT+8(%rip)
    0xff, 0x25, 0x00, 0x00, 0x00, 0x00, // jmp *GOT+16(%rip)
    0x90, 0x90, 0x90, 0x90           // nopnopnop
};

// .plt values (other entries)
const uint8_t cpu0PltAtomContent[16] = {
    0xff, 0x25, 0x00, 0x00, 0x00, 0x00, // jmpq *gotatom(%rip)
    0x68, 0x00, 0x00, 0x00,           // pushq reloc-index
    0xe9, 0x00, 0x00, 0x00           // jmpq plt[-1]
};

/// \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, 8);
    }
};

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);
    }
}

class ELFPassFile : public SimpleFile {
public:
    ELFPassFile(const ELFLinkingContext &eti) : SimpleFile(eti, "ELFPassFile") {}

    llvm::BumpPtrAllocator _alloc;
};

/// \brief Create GOT and PLT entries for relocations. Handles standard GOT/PLT
/// along with IFUNC and TLS.
template <class Derived> class GOTPLTPass : public Pass {
    /// \brief Handle a specific reference.
    void handleReference(const DefinedAtom &atom, const Reference &ref) {
        switch (ref.kind()) {
#ifndef 0
        case R_CPU0_PLT32:
            static_cast<Derived *>(this)->handlePLT32(ref);
            break;
#endif
        case R_CPU0_PC24:
            static_cast<Derived *>(this)->handlePC24(ref);
            break;
#ifndef 0
        case R_CPU0_GOTPOFF: // GOT Thread Pointer Offset
            static_cast<Derived *>(this)->handleGOTPOFF(ref);
            break;
        case R_CPU0_GOTPCREL:
            static_cast<Derived *>(this)->handleGOTPCREL(ref);
            break;
#endif
#endif
    }
}

protected:
    /// \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;
}

/// \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 RELGOT relocation to the original target resolver.
ErrorOr<void> handleIFUNC(const Reference &ref) {
    auto target = dyn_cast_or_null<const DefinedAtom>(ref.target());
    if (target && target->contentType() == DefinedAtom::typeResolver)
        const_cast<Reference &>(ref).setTarget(getIFUNCPLTEntry(target));
    return error_code::success();
}

/// \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 TLS_TPREL32 GOT entry and change the relocation to a PC24 to
/// the GOT.
void handleGOTPOFF(const Reference &ref) {
    const_cast<Reference &>(ref).setTarget(getGOTPOFF(ref.target()));
    const_cast<Reference &>(ref).setKind(R_CPU0_PC24);
}

/// \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;
}

/// \brief Handle a GOTPCREL relocation to an undefined weak atom by using a
/// null GOT entry.
void handleGOTPCREL(const Reference &ref) {
    const cast<Reference &>(ref).setKind(R_CPU0_PC24);
    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));
}

public:
GOTPLTPass(const ELFLinkingContext &ti)
    : _file(ti), _null(nullptr), _PLT0(nullptr), _got0(nullptr),
    _got1(nullptr) {}

/// \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(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 (_PLT0) {
        _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++);
        _got1->setOrdinal(ordinal++);
        mf.addAtom(*_got0);
        mf.addAtom(*_got1);
    }
    for (auto &got : _gotVector) {
        got->setOrdinal(ordinal++);
        mf.addAtom(*got);
    }
}

protected:
/// \brief Owner of all the Atoms created by this pass.
ELFPassFile _file;

/// \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;

/// \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;
GOTAtom *_got1;
/// @@
};

/// 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 StaticGOTPLTPass LLVM_FINAL : public GOTPLTPass<StaticGOTPLTPass> {
public:
    StaticGOTPLTPass(const elf::Cpu0LinkingContext &ti) : GOTPLTPass(ti) {}

    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> handlePC24(const Reference &ref) { return handleIFUNC(ref); }

class DynamicGOTPLTPass LLVM_FINAL : public GOTPLTPass<DynamicGOTPLTPass> {
public:
    DynamicGOTPLTPass(const elf::Cpu0LinkingContext &ti) : GOTPLTPass(ti) {}

    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");
        _got1 = new (_file._alloc) Cpu0GOTAtom(_file, ".got.plt");
        _PLT0->addReference(R_CPU0_PC24, 2, _got0, -4);
        _PLT0->addReference(R_CPU0_PC24, 8, _got1, -4);
#ifdef NDEBUG
        _got0->_name = "__got0";
        _got1->_name = "__got1";
#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");
        pa->addReference(R_CPU0_PC24, 2, ga, -4);
        pa->addReference(LLD_R_CPU0_GOTRELINDEX, 7, ga, 0);
        pa->addReference(R_CPU0_PC24, 12, getPLT0(), -4);
        // Set the starting address of the got entry to the second instruction in
        // the plt entry.
        ga->addReference(R_CPU0_32, 0, pa, 6);
#ifdef 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> handlePLT32(const Reference &ref) {
    // Turn this into a PC24 to the PLT entry.
    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);
    if (isa<const SharedLibraryAtom>(ref.target()))
        const_cast<Reference &>(ref).setTarget(getPLTEEntry(ref.target()));
    return error_code::success();
}

ErrorOr<void> handlePC24(const Reference &ref) {
    if (ref.target() && isa<SharedLibraryAtom>(ref.target()))
        return handlePLT32(ref);
    return handleIFUNC(ref);
}

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;
}

void handleGOTPCREL(const Reference &ref) {
    const_cast<Reference &>(ref).setKind(R_CPU0_PC24);
    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));
}
};

} // end anon namespace

void elf::Cpu0LinkingContext::addPasses(PassManager &pm) const {
    switch (_outputFileType) {
    case llvm::ELF::ET_EXEC:
        if (_isStaticExecutable)
            pm.add(std::unique_ptr<Pass>(<new> StaticGOTPLTPass(*this)));
        else
            pm.add(std::unique_ptr<Pass>(<new> DynamicGOTPLTPass(*this)));
        break;
    case llvm::ELF::ET_DYN:
        pm.add(std::unique_ptr<Pass>(<new> DynamicGOTPLTPass(*this)));
        break;
    }
}

```

```
case llvm::ELF::ET_REL:
    break;
default:
    llvm_unreachable("Unhandled output file type");
}
ELFLinkingContext::addPasses(pm);
#endif

#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_16)
        .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_CALL24)
        .Case("LLD_R_CPU0_GOTRELINDEX", LLD_R_CPU0_GOTRELINDEX)
        .Default(-1);

    if (ret == -1)
        return make_error_code(yaml_reader_error::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) {
        .Case(R_CPU0_NONE)
        .Case(R_CPU0_16)
        .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_CALL24)
        .Case LLD_R_CPU0_GOTRELINDEX:
            return std::string("LLD_R_CPU0_GOTRELINDEX");
    }

    return make_error_code(yaml_reader_error::illegal_value);
}
```

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/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"
```

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/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 "Cpu0RelocationHandler.h"
#include "TargetLayout.h"

#include "lld/ReaderWriter/Simple.h"

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 void addFiles(InputFiles &f);

private:
    class GOTFile : public SimpleFile {
public:
```

```

        GOTFile(const ELFLinkingContext &eti) : SimpleFile(eti, "GOTFile") {}
        llvm::BumpPtrAllocator _alloc;
    } _gotFile;

    Cpu0TargetRelocationHandler _relocationHandler;
    TargetLayout<Cpu0ELFType> _targetLayout;
};

} // end namespace elf
} // end namespace lld

#endif

```

LLVMBackendTutorialExampleCode/Cpu0_lld_20130816/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;

Cpu0TargetHandler::Cpu0TargetHandler(Cpu0LinkingContext &context)
    : DefaultTargetHandler(context), _gotFile(context),
    _relocationHandler(context), _targetLayout(context) {}

void Cpu0TargetHandler::addFiles(InputFiles &f) {
    _gotFile.addAtom(*new (_gotFile._alloc) GLOBAL_OFFSET_TABLEAtom(_gotFile));
    _gotFile.addAtom(*new (_gotFile._alloc) TLSGETADDRAtom(_gotFile));
    f.appendFile(_gotFile);
}

```

12.3 ELF to Hex

Update llvm-objdump driver to support ELF to Hex for Cpu0 backend as follows,

LLVMBackendTutorialExampleCode/llvm-objdump/llvm-objdump.cpp

```

// llvm-objdump -elf2hex code added begin:
static cl::opt<bool>
ConvertElf2Hex("elf2hex", cl::desc("Display the hex content of verilog cpu0 needed sections"));

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;
}

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';
}
}

// Modified from DisassembleObject()
static void DisassembleObjectForHex(const ObjectFile *Obj/*, bool InlineRelocs*/, uint64_t& lastAddr)
{
    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<MCSymbolizer> 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;
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.
            continue;
```

```

        outs() << '\n' << "/*" << Symbols[si].second << "*/\n";
```

```

#ifndef NDEBUG
    raw_ostream &DebugOut = DebugFlag ? dbgs() : nulls();
#else
    raw_ostream &DebugOut = nulls();
#endif
```

```

for (Index = Start; Index < End; Index += Size) {
    MCInst Inst;
```

```

        if (DisAsm->getInstruction(Inst, Size, memoryObject,
                                     SectionAddr + Index,
                                     DebugOut, CommentStream)) {
            outs() << format("/*%8" PRIx64 "*/", /*SectionAddr + */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
        }
```

```

        // 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;
}
}
lastAddr = Index;
}
}
}

// Modified from PrintSectionContents()
static void PrintDataSections(const ObjectFile *o, uint64_t lastAddr) {
    error_code ec;
    std::size_t addr, end;
    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 == ".data") {
            if (Contents.size() <= 0) {
                continue;
            }
            // Fill /*address*/ 00 00 00 00 between lastAddr and BaseAddr
            for (addr = lastAddr, 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) << " ";
                    else
                        outs() << " ";
                }
                // save lastAddr
                if ((BaseAddr + addr + 16) > end)

```

```

        lastAddr = BaseAddr + end;
    else
        lastAddr = BaseAddr + addr + 16;
    // 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";
}
}
}

static void Elf2Hex(const ObjectFile *o) {
    uint64_t startAddr = GetSectionHeaderStartAddress(o, "_start");
    // outs() << format("_start address:%08" PRIx64 "\n", startAddr);
    uint64_t lastAddr;
    DisassembleObjectForHex(o, lastAddr);
    // outs() << format("lastAddr:%08" PRIx64 "\n", lastAddr);
    PrintDataSections(o, lastAddr);
}
// llvm-objdump -elf2hex code added end:
// Code added fo cpu0 -elf2hex end:

static void DumpObject(const ObjectFile *o) {
    outs() << '\n';
    if (ConvertElf2Hex)
        outs() << "/*";
    outs() << o->getFileName()
        << ":\tfile format " << o->getFileFormatName();
    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 && o->isELF())
        printELFFileHeader(o);
}

/// @brief Dump each object file in \a a;
static void DumpArchive(const Archive *a) {

```

```

for (Archive::child_iterator i = a->begin_children(),
        e = a->end_children(); i != e; ++i) {
    OwningPtr<Binary> child;
    if (error_code ec = i->getAsBinary(child)) {
        // Ignore non-object files.
        if (ec != object_error::invalid_file_type)
            errs() << ToolName << ":" << a->getFileName() << "' : " << ec.message()
                << ".\n";
        continue;
    }
    if (ObjectFile *o = dyn_cast<ObjectFile>(child.get()))
        DumpObject(o);
    else
        errs() << ToolName << ":" << a->getFileName() << "' : "
            << "Unrecognized file type.\n";
}
}

/// @brief Open file and figure out how to dump it.
static void DumpInput(StringRef file) {
    // If file isn't stdin, check that it exists.
    if (file != "-" && !sys::fs::exists(file)) {
        errs() << ToolName << ":" << file << "' : " << "No such file\n";
        return;
    }

    if (MachoOpt && Disassemble) {
        DisassembleInputMachO(file);
        return;
    }

    // Attempt to open the binary.
    OwningPtr<Binary> binary;
    if (error_code ec = createBinary(file, binary)) {
        errs() << ToolName << ":" << file << "' : " << ec.message() << ".\n";
        return;
    }

    if (Archive *a = dyn_cast<Archive>(binary.get()))
        DumpArchive(a);
    else if (ObjectFile *o = dyn_cast<ObjectFile>(binary.get()))
        DumpObject(o);
    else
        errs() << ToolName << ":" << file << "' : " << "Unrecognized file type.\n";
}

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;
}

static void DumpObject(const ObjectFile *o) {
    outs() << '\n';
    if (ConvertElf2Hex)
        outs() << "/*";
    outs() << o->getFileName()
        << ":\tfile format " << o->getFileName();
    if (ConvertElf2Hex)
        outs() << "*/";
    ...
    if (ConvertElf2Hex)
        Elf2Hex(o);
    ...
}

int main(int argc, char **argv) {
    ...
    if (!Disassemble
        ...
        && !ConvertElf2Hex
        ... ) {
        ...
    }
    ...
}
```

12.4 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 follows,

LLVMBackendTutorialExampleCode/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 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

```

LLVMBackendTutorialExampleCode/InputFiles/printf-stdarg-2.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>
#include "print.h"

#define TEST_PRINTF

#include "boot.cpp"

struct Time
{
    int hour;
    int minute;
    int second;
};

struct Date
{
    int year;
    int month;
    int day;
    int hour;
    int minute;
    int second;
};

int test_global();
int printf(const char *format, ...);
int sprintf(char *out, const char *format, ...);
struct Date copyDate(struct Date date);
```

```

struct Time copyTime(struct Time time);

int gI = 100;
struct Date gDate = {2012, 10, 12, 1, 2, 3};

#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];

    int a = 0;
    struct Time time1 = {1, 10, 12};
    struct Time time2;
    struct Date date;

    a = test_global(); // gI = 100
    printf("global variable gI = %d\n", a);
    printf("time1 = %d %d %d\n", time1.hour, time1.minute, time1.second);
    date = copyDate(gDate);
    printf("date = %d %d %d %d %d\n", date.year, date.month, date.day, date.hour, date.minute, date.minute);
    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\\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

// For memory IO
void putchar(const char c)
{
    char *p = (char*)OUT_MEM;
    *p = c;

    return;
}

static void printchar(char **str, int c)
{
    if (*str) {
        **str = c;
        ++(*str);
    }
    else 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)) {
    // pad left
    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;
                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 );
}

int test_global()
{
    return gI;
}

struct Date copyDate(struct Date date)
{
    return date;
}

struct Time copyTime(struct Time time)
{
    return time;
}

[Gamma@localhost InputFiles]$ /usr/local/llvm/release/cmake_debug_build/bin/
clang -target mips-unknown-linux-gnu -c printf-stdarg-2.c -emit-llvm -o
printf-stdarg-2.bc
printf-stdarg-2.c:75:19: warning: incomplete format specifier [-Wformat]
    printf("%d %s(s)%", 0, "message");
                           ^
1 warning generated.
[Gamma@localhost InputFiles]$ /home/Gamma/test/lld/cmake_debug_build/bin/llc
-march=cpu0 -relocation-model=static -filetype=obj printf-stdarg-2.bc -o
printf-stdarg-2.cpu0.o
[Gamma@localhost InputFiles]$ /home/Gamma/test/lld/cmake_debug_build/bin/lld
-flavor gnu -target cpu0-unknown-linux-gnu printf-stdarg-2.cpu0.o -o a.out
```

```
[Gamma@localhost InputFiles]$ /home/Gamma/test/lld/cmake_debug_build/bin/
11vm-objdump -elf2hex a.out > ../cpu0_verilog/redesign/cpu0s.hex
[Gamma@localhost InputFiles]$ cd ../cpu0_verilog/redesign/
[Gamma@localhost redesign]$ iverilog -o cpu0s cpu0s.v
[Gamma@localhost redesign]$ ls
cpu0s  cpu0s.hex  cpu0s.v
[Gamma@localhost redesign]$ ./cpu0s
WARNING: cpu0s.v:317: $readmemh(cpu0s.hex): Not enough words in the file for
the requested range [0:65535].
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

[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.

12.5 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 translate 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 relased. My Cpu0 example code is also 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" ¹ 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.

13.1 Setting Up Your Mac

13.1.1 Installing LLVM, Xcode and `cmake`

Todo

¹ <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” ². 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: ³. 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 13.1](#) 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.

13.1.2 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” ⁴. 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 13.2](#), then click **configure** button to enter [Figure 13.3](#), and then click **Done** button to get [Figure 13.4](#).

Click OK from [Figure 13.4](#) and select Cmake 2.8-9.app for CMAKE_INSTALL_NAME_TOOL by click the right side button “...” of that row to get [Figure 13.5](#).

² <http://llvm.org/releases/download.html#3.3>

³ <http://www.cmake.org/cmake/resources/software.html>

⁴ <http://jonathan2251.github.com/lbd/install.html#create-llvm-xcodeproj-of-supporting-cpu0-by-terminal-cmake-command>

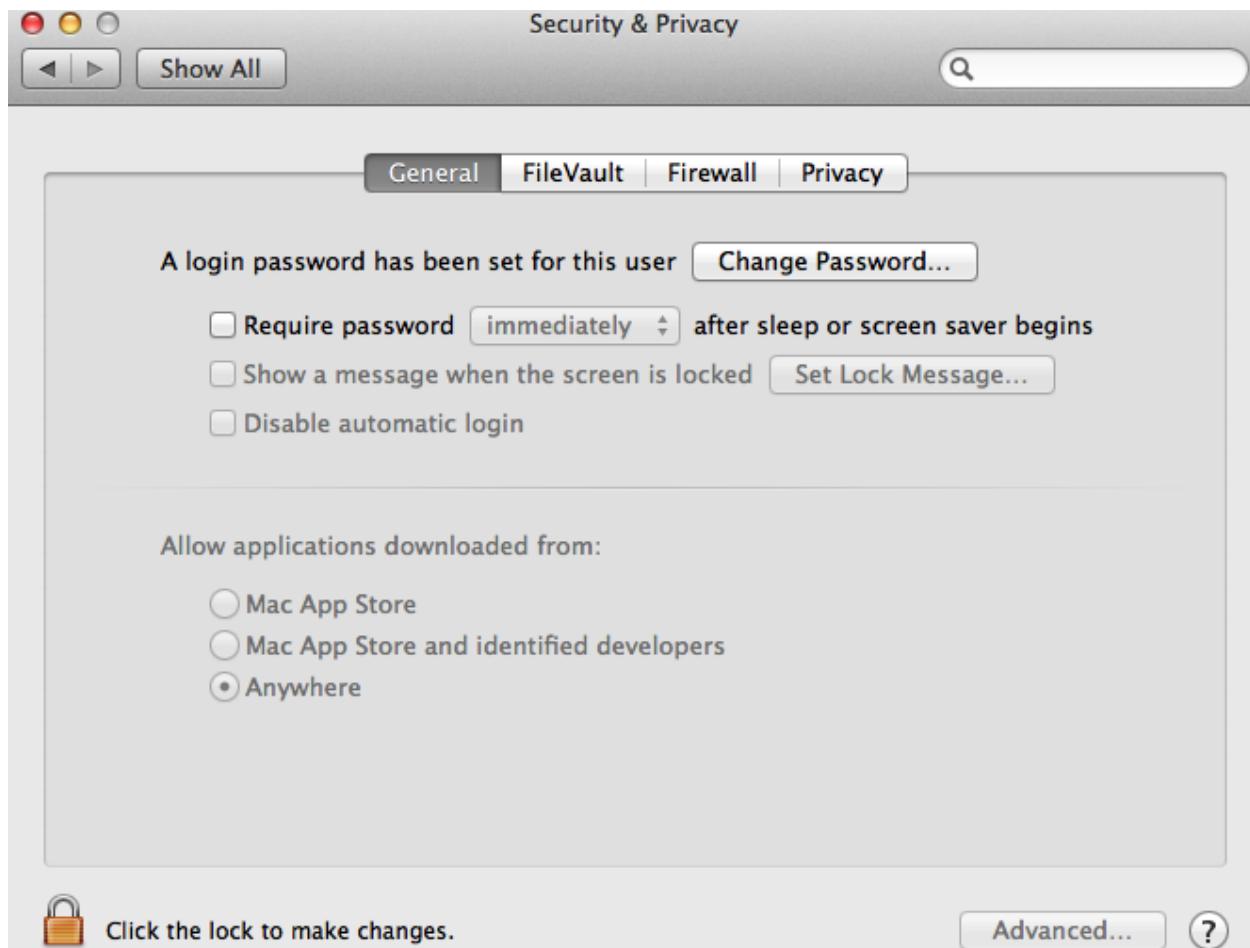


Figure 13.1: Adjusting Mac OS X security settings to allow cmake installation.

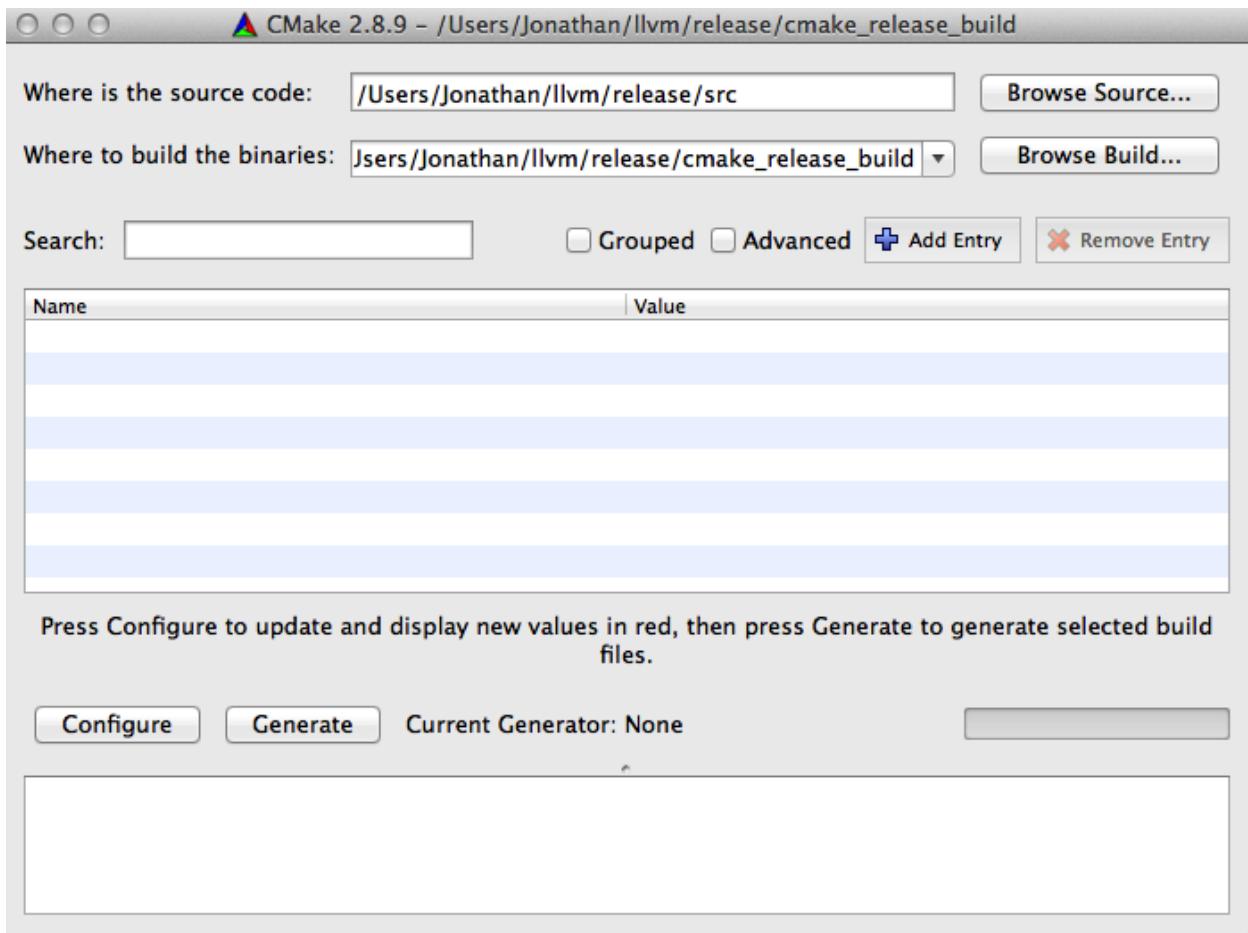


Figure 13.2: Start to create LLVM.xcodeproj by cmake

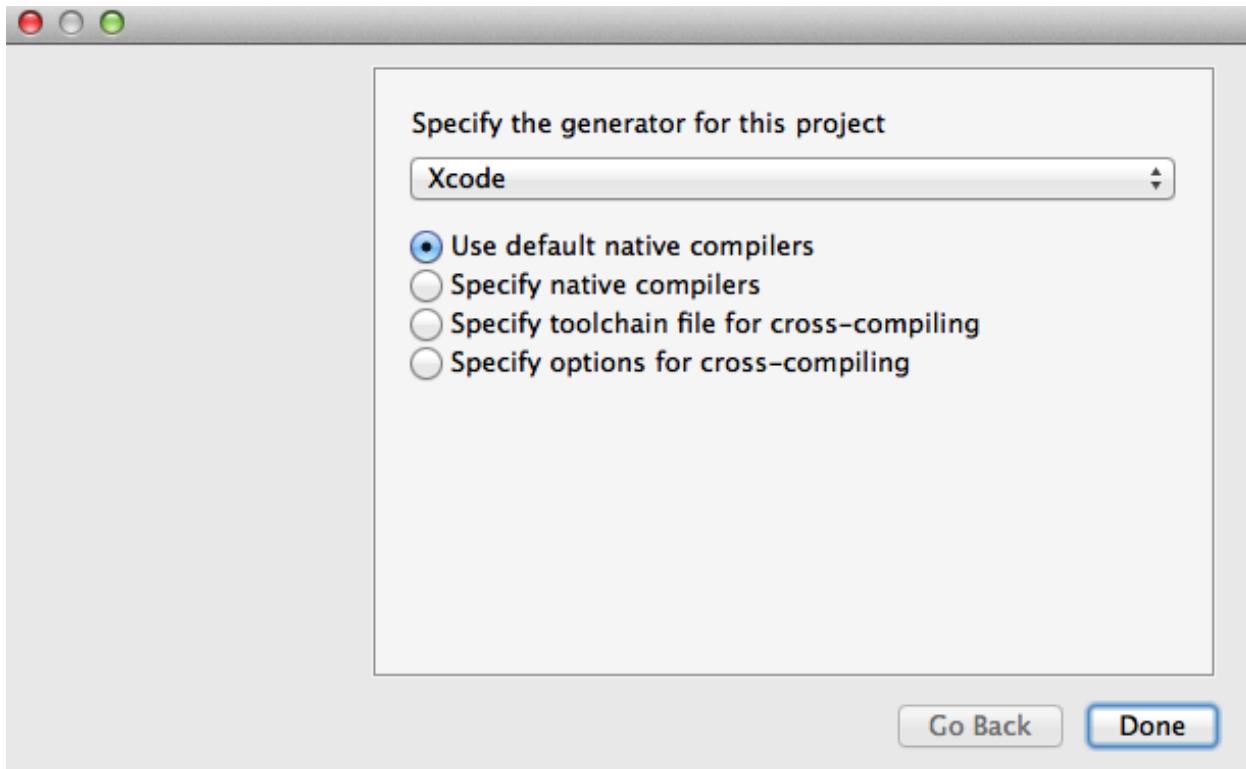


Figure 13.3: Create LLVM.xcodeproj by cmake – Set option to generate Xcode project

Click Configure button to get [Figure 13.6](#).

Check CLANG_BUILD_EXAMPLES, LLVM_BUILD_EXAMPLES, and uncheck LLVM_ENABLE_PIC as [Figure 13.7](#).

Click Configure button again. If the output result message has no red color, then click Generate button to get [Figure 13.8](#).

13.1.3 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 13.9](#).

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
```

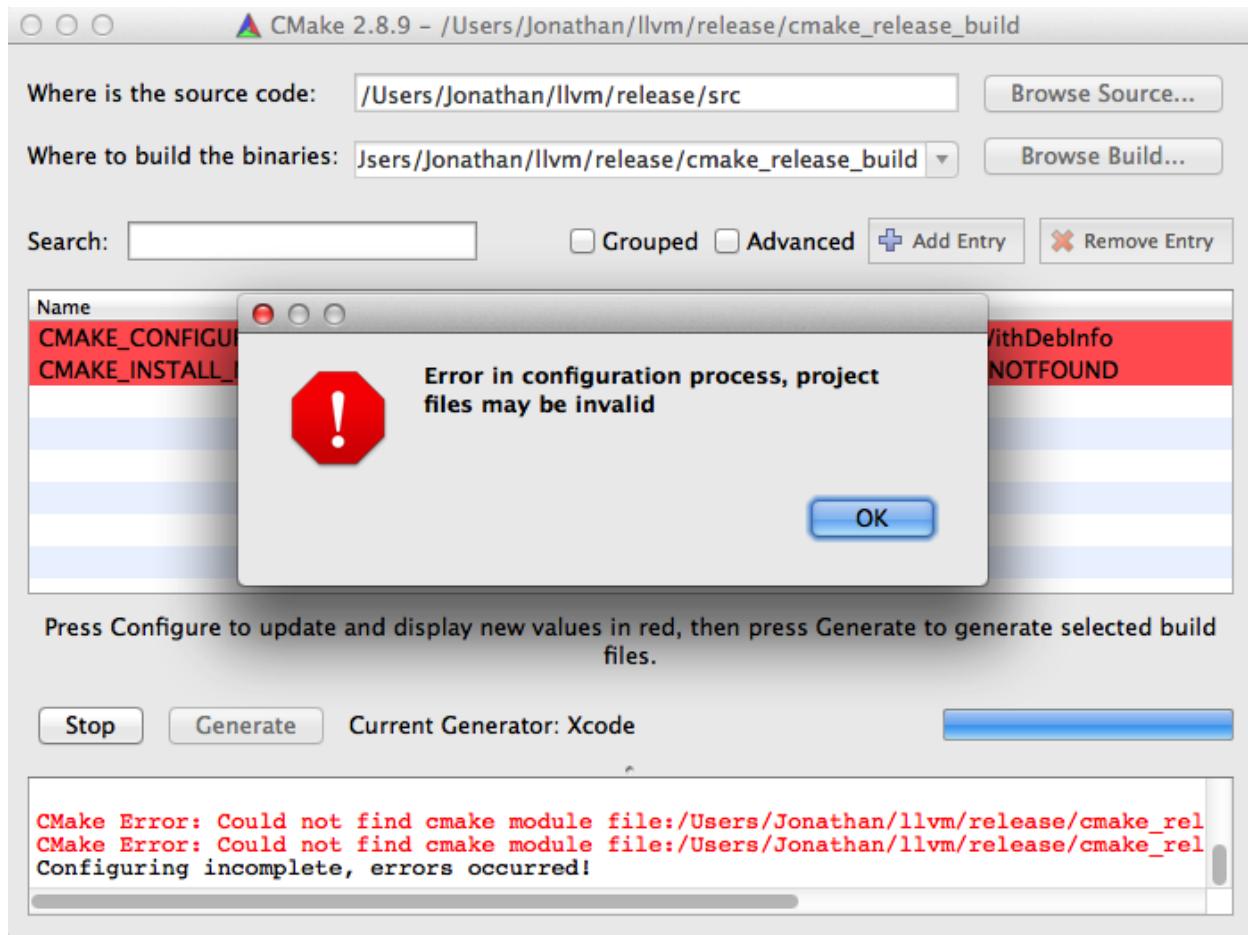


Figure 13.4: Create LLVM.xcodeproj by cmake – Before Adjust CMAKE_INSTALL_NAME_TOOL

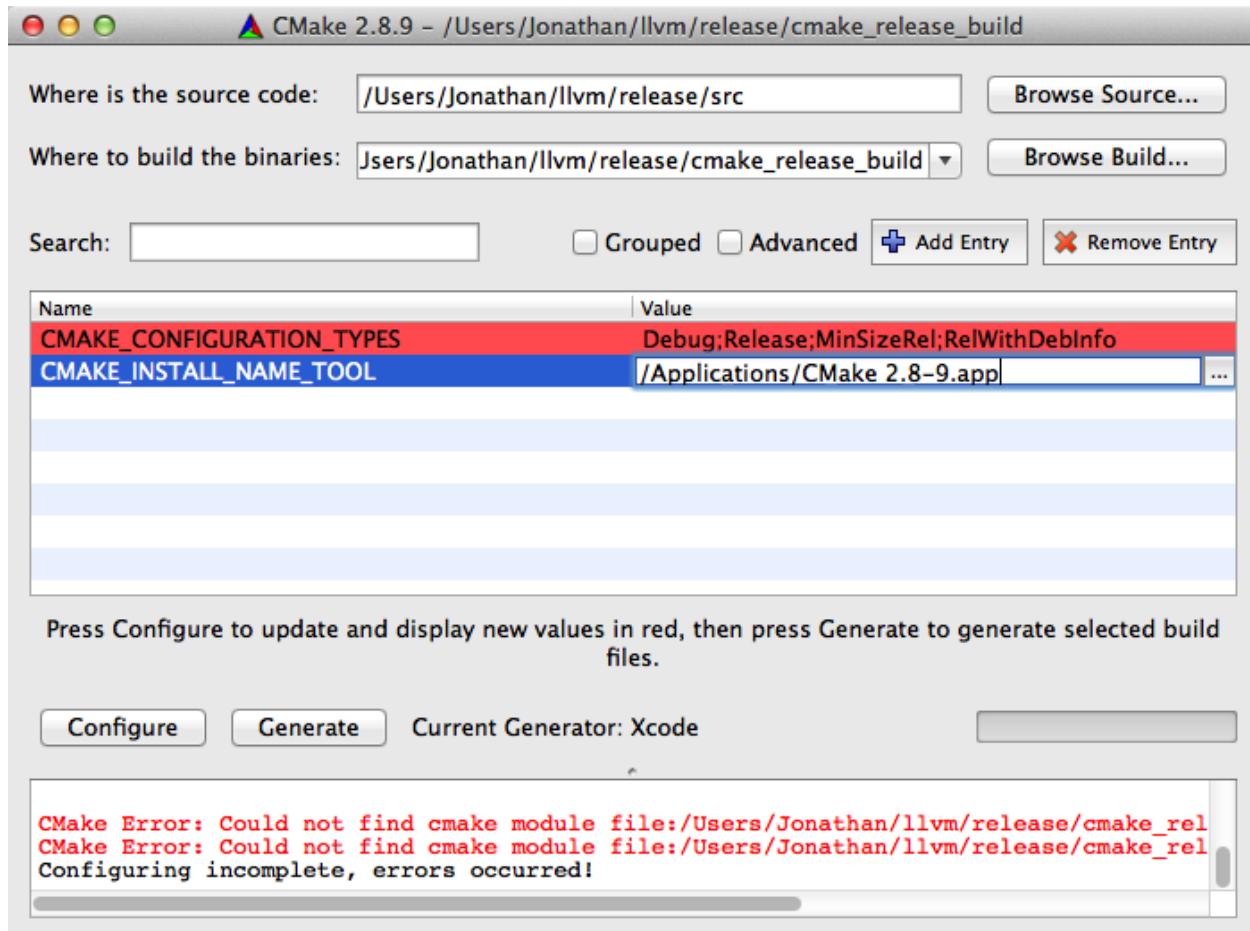


Figure 13.5: Select Cmake 2.8-9.app

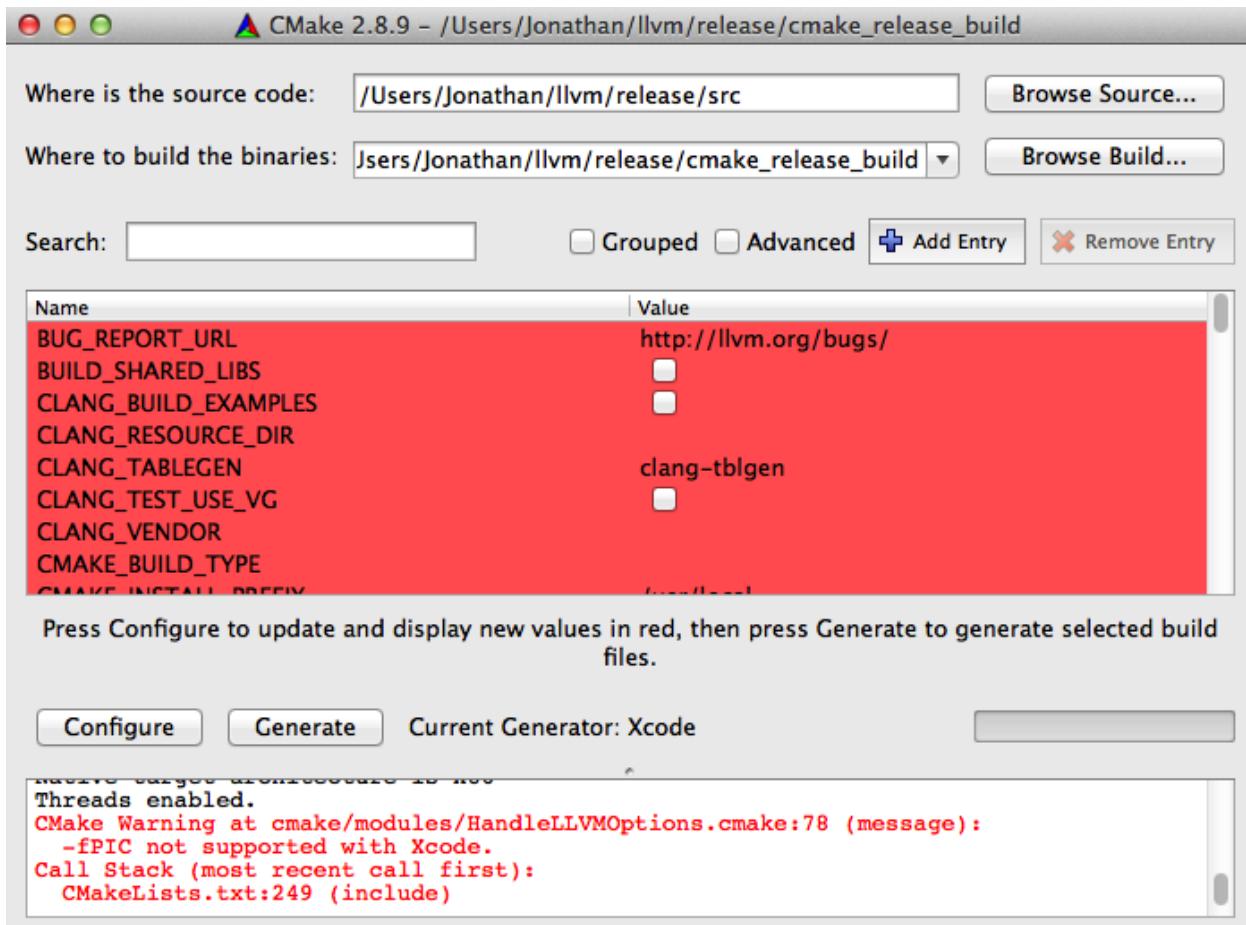


Figure 13.6: Click cmake Configure button first time

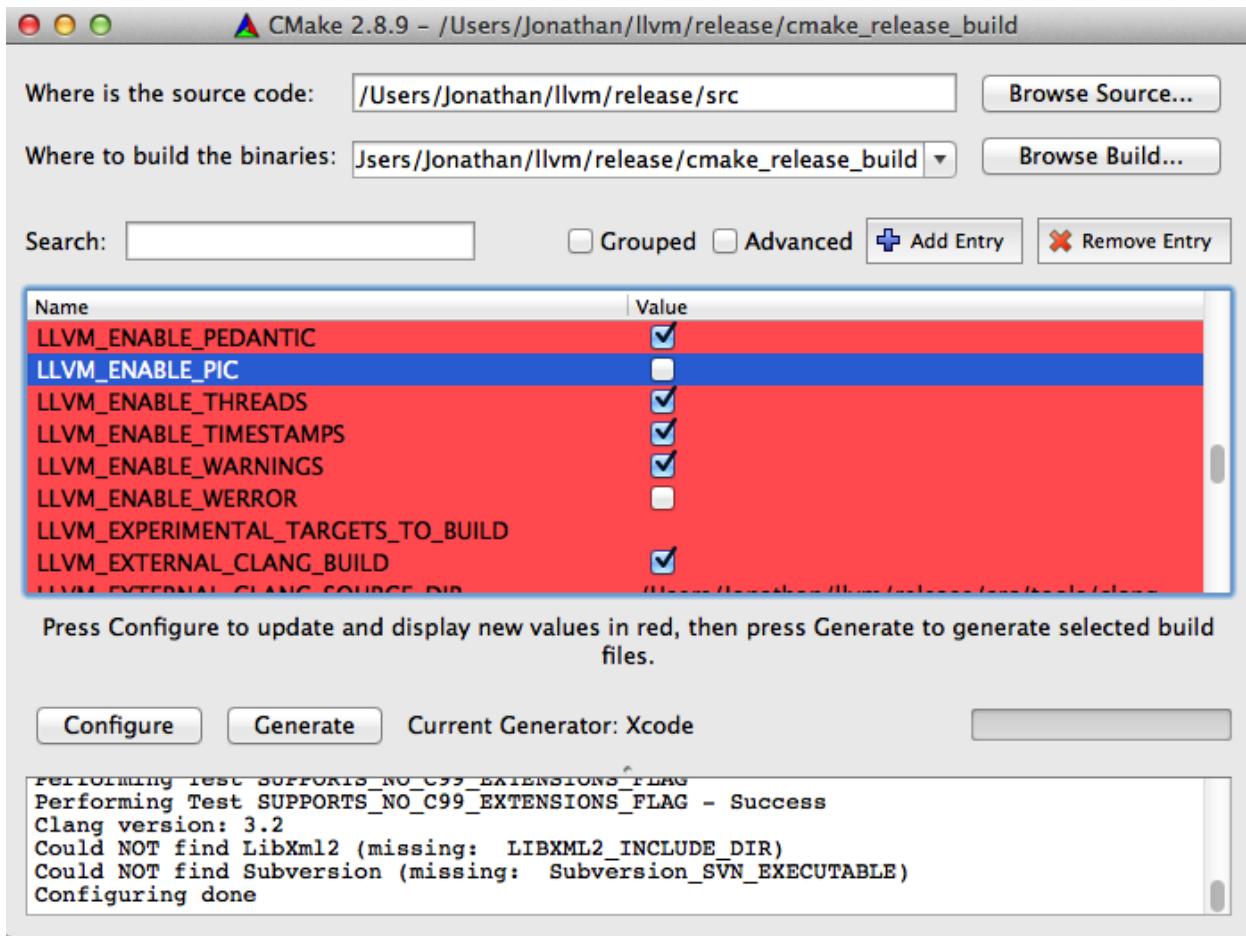


Figure 13.7: Check CLANG_BUILD_EXAMPLES, LLVM_BUILD_EXAMPLES, and uncheck LLVM_ENABLE_PIC in cmake

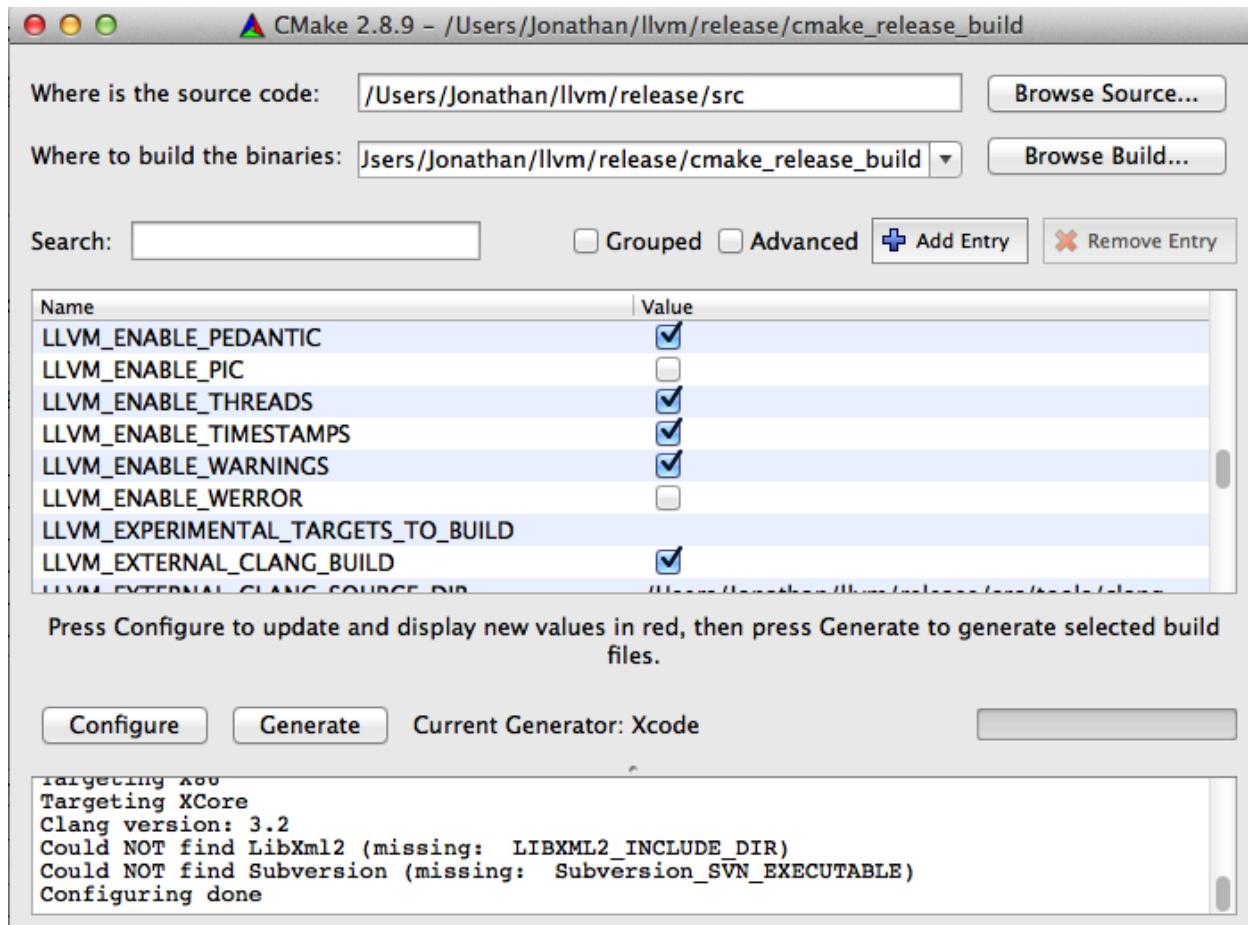


Figure 13.8: Click cmake Generate button second time

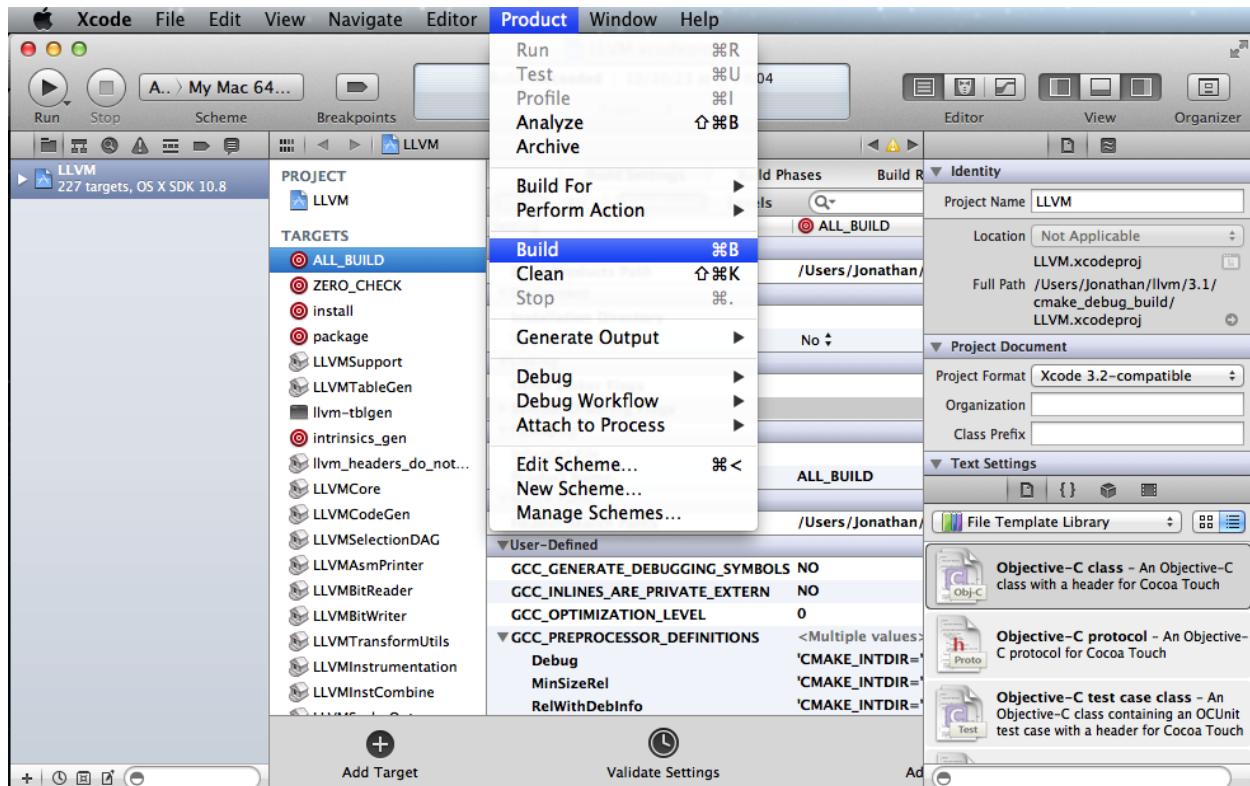


Figure 13.9: Click Build button to build LLVM.xcodeproj by Xcode

```
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$
```

13.1.4 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”⁵, 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 my git hub is LLVM 3.3 released version. My 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 -zxfv 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/
LLVMBackendTutorialExampleCode.tar.gz .
118-165-78-111:test Jonathan$ tar -zxfv LLVMBackendTutorialExampleCode.tar.gz
118-165-78-111:test Jonathan$ mkdir src/lib/Target/Cpu0
118-165-78-111:test Jonathan$ mv LLVMBackendTutorialExampleCode
src/lib/Target/Cpu0/.
118-165-78-111:test Jonathan$ cp -rf src/lib/Target/Cpu0/
LLVMBackendTutorialExampleCode/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/LLVMBackendTutorialExampleCode/
118-165-80-55:LLVMBackendTutorialExampleCode Jonathan$ pwd
/Users/Jonathan/llvm/test/src/lib/Target/Cpu0/LLVMBackendTutorialExampleCode
118-165-80-55:LLVMBackendTutorialExampleCode Jonathan$ sh removecpu0.sh
118-165-80-55:LLVMBackendTutorialExampleCode Jonathan$ ls ..
LLVMBackendTutorialExampleCode
118-165-80-55:LLVMBackendTutorialExampleCode Jonathan$ cp -rf Chapter2/* ../..
118-165-80-55:LLVMBackendTutorialExampleCode 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.

⁵ <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”⁵ 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/LLVMBackendTutorialExampleCode/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 ⁶ or lldb portal ⁷.

13.1.5 Setup llvm-lit on iMac

The llvm-lit ⁸ 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",
```

⁶ <http://lldb.llvm.org/lldb-gdb.html>

⁷ <http://lldb.llvm.org/>

⁸ <http://llvm.org/docs/TestingGuide.html>

13.1.6 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
```

13.1.7 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 ⁹. FileMerge is a part of Xcode, you can type FileMerge in Finder – Applications as Figure 13.10 and drag it into the Dock as Figure 13.11.

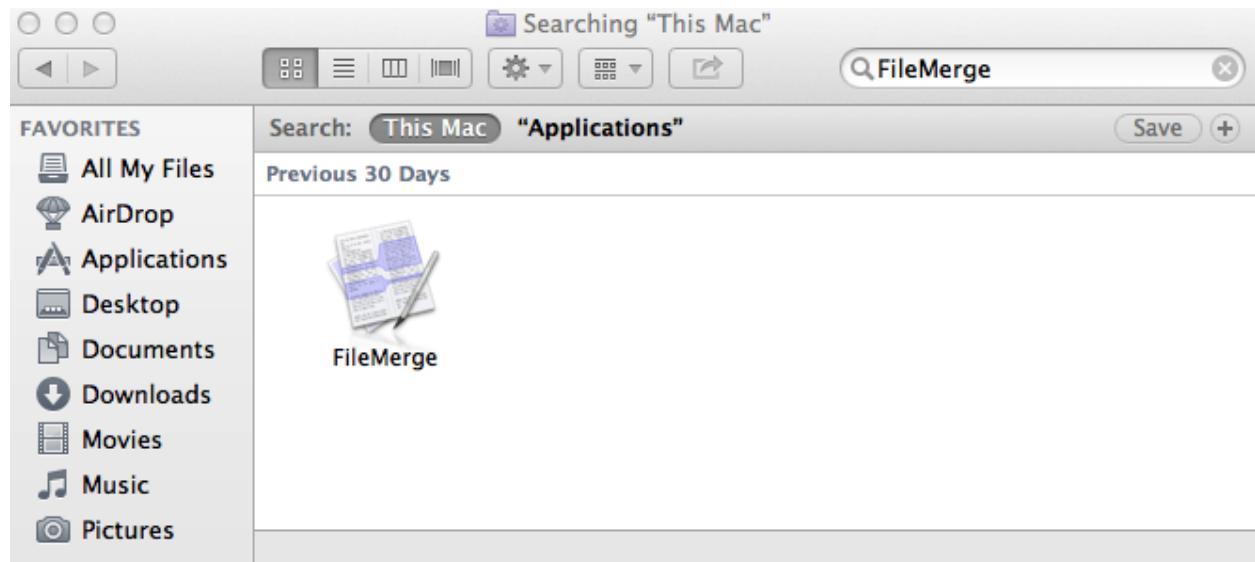


Figure 13.10: Type FileMerge in Finder – Applications



Figure 13.11: Drag FileMege into the Dock

⁹ <http://kdiff3.sourceforge.net>

Download tool Graphviz for display llvm IR nodes in debugging,¹⁰. We choose mountainlion as Figure 13.12 since our iMac is Mountain Lion.

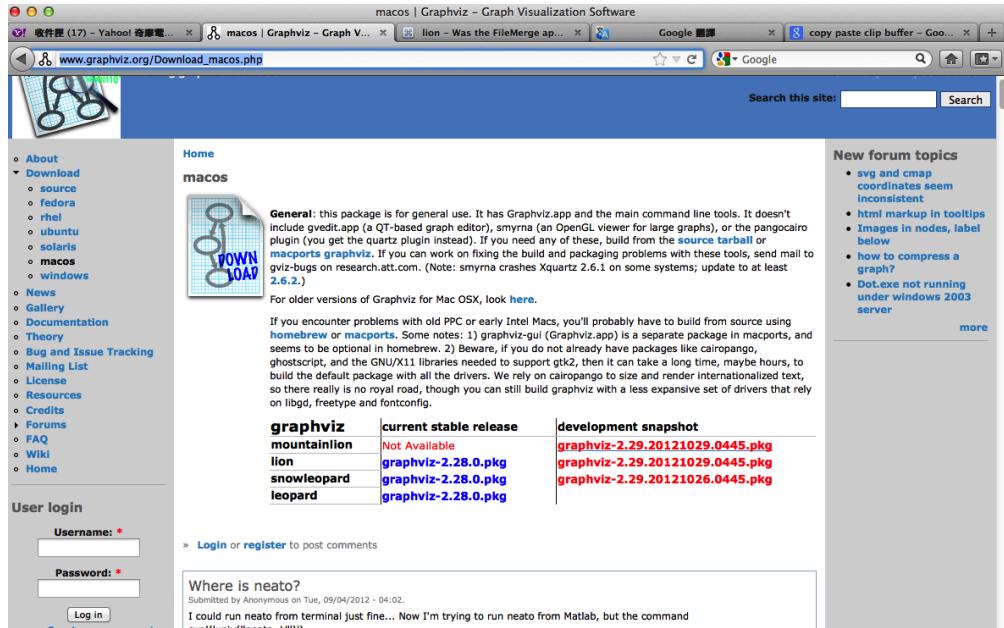


Figure 13.12: Download graphviz for llvm IR node display

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” of¹¹ and the section “Viewing graphs while debugging code” of¹². 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 Other instructions. You can download from App Store. To dump binary file, first, open the binary file, next, select menu “File – Hex Front Document” as Figure 13.13. Then select “Front document’s file” as Figure 13.14.

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
```

¹⁰ http://www.graphviz.org/Download_macos.php

¹¹ <http://llvm.org/docs/CodeGenerator.html>

¹² <http://llvm.org/docs/ProgrammersManual.html>

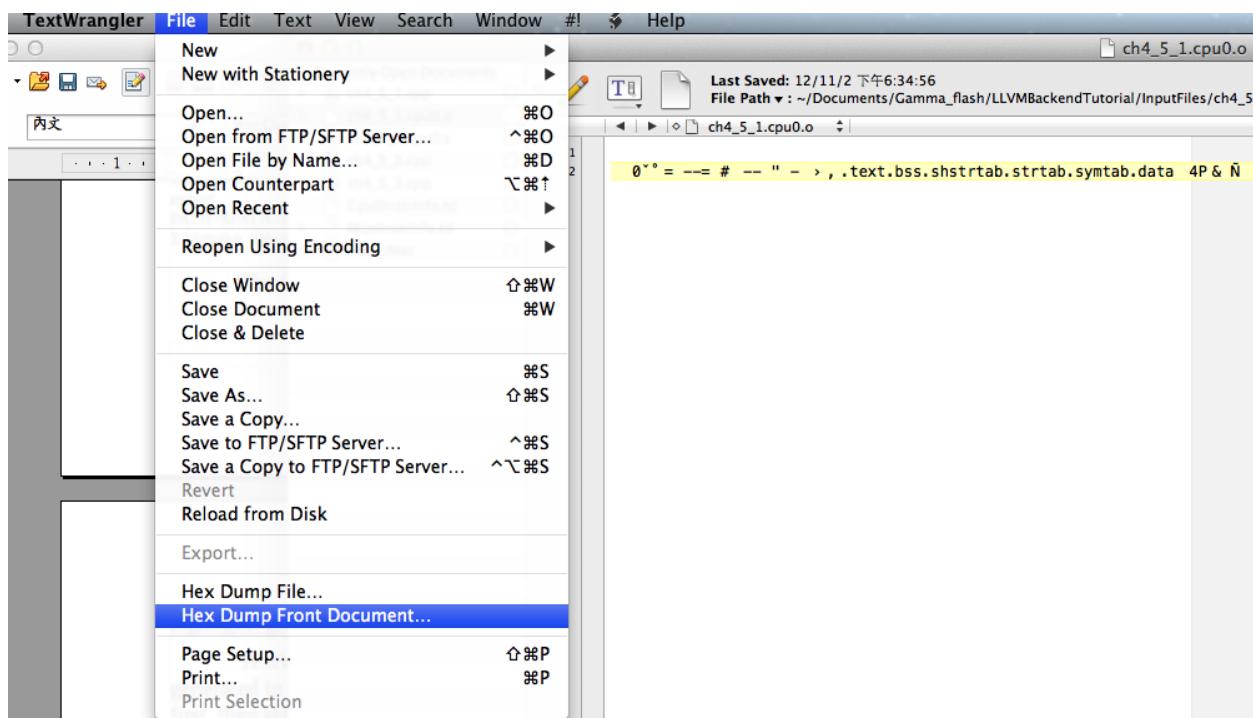


Figure 13.13: Select Hex Dump menu

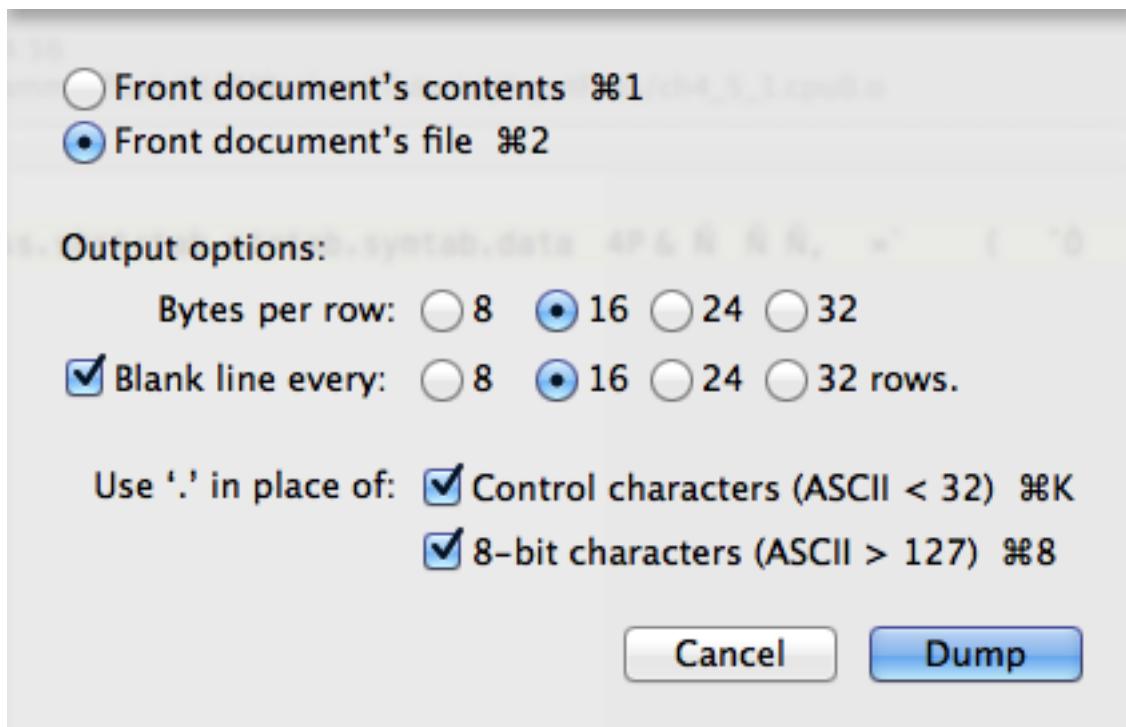


Figure 13.14: Select Front document's file in TextWrangler

```
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
```

13.2 Setting Up Your Linux Machine

13.2.1 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
```

13.2.2 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 my git hub is LLVM 3.3 released version. My 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/LLVMBackendTutorialExampleCode/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 LLVMBackendTutorialExampleCode/2/Cpu0 to the directory by commands cd src/lib/Target/Cpu0/ and cp -rf LLVMBackendTutorialExample/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 -zxvf llvm-3.3.src.tar.gz
[Gamma@localhost test]$ mv llvm-3.3.src src
[Gamma@localhost test]$ cp /Users/Jonathan/Downloads/
LLVMBackendTutorialExampleCode.tar.gz .
[Gamma@localhost test]$ tar -zxvf LLVMBackendTutorialExampleCode.tar.gz
...
[Gamma@localhost test]$ mkdir src/lib/Target/Cpu0
118-165-78-111:test Jonathan$ mv LLVMBackendTutorialExampleCode src/lib/Target/Cpu0/ .
[Gamma@localhost test]$ cp -rf LLVMBackendTutorialExampleCode/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/LLVMBackendTutorialExampleCode/
[Gamma@localhost LLVMBackendTutorialExampleCode]$ sh removecpu0.sh
[Gamma@localhost LLVMBackendTutorialExampleCode]$ ls ../
LLVMBackendTutorialExampleCode
[Gamma@localhost LLVMBackendTutorialExampleCode]$ cp -rf Chapter2/* ../.
[Gamma@localhost LLVMBackendTutorialExampleCode]$ 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/
LLVMBackendTutorialExampleCode/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]$
```

13.2.3 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.

13.2.4 Install other tools on Linux

Download Graphviz from ¹³ according your Linux distribution. Files compare tools Kdiff3 came from web site ⁸.

13.3 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
...
```

¹³ <http://www.graphviz.org/Download..php>

TODO LIST

Todo

Add info about LLVM documentation licensing.

(The *original entry* is located in /Users/Jonathan/test/lbd/source/about.rst, line 167.)

Todo

Find information on debugging LLVM within Xcode for Macs.

(The *original entry* is located in /Users/Jonathan/test/lbd/source/install.rst, line 33.)

Todo

Find information on building/debugging LLVM within Eclipse for Linux.

(The *original entry* is located in /Users/Jonathan/test/lbd/source/install.rst, line 34.)

Todo

Fix centering for figure captions.

(The *original entry* is located in /Users/Jonathan/test/lbd/source/install.rst, line 43.)

Todo

I might want to re-edit the following paragraph

(The *original entry* is located in /Users/Jonathan/test/lbd/source/llvmstructure.rst, line 727.)

BOOK EXAMPLE CODE

The example code is available in:

<http://jonathan2251.github.com/lbd/LLVMBackendTutorialExampleCode.tar.gz>

CHAPTER
SIXTEEN

ALTERNATE FORMATS

The book is also available in the following formats: