Tutorial: Creating an LLVM Backend for the Cpu0 Architecture

Release 3.2.13

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

May 23, 2013

CONTENTS

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.

CHAPTER

ONE

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.

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 3.2.14, Not Released Yet 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.

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 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.

Adding arithmetic and local pointer support:

Over ten C operators and their corresponding LLVM IR instructions are introduced in this chapter. Around 345 lines of source code, mostly in .td Target Description files, are added. With these 345 lines, the backend can now translate the **+, -, *, /, &, |, ^, <<, >>, !** and **%** C operators into the appropriate Cpu0 assembly code. Use of the `llc` debug option and of **Graphviz** as a debug tool are introduced in this chapter.

Generating object files:

Object file generation support for the Cpu0 backend is added in this chapter, as the Target Registration structure is introduced. With 700 lines of additional code, the Cpu0 backend can now generate big and little endian object files.

Global variables, 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 area 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.

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

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

Appendix B: LLVM changes:

Introduces the difference of the LLVM APIs used by Cpu0 and Mips when updating this guide between LLVM different version.

Appendix C: instructions discuss:

Discuss the other backend instructions.

CHAPTER

TWO

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 knowledgable of compiler backends.

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

This 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

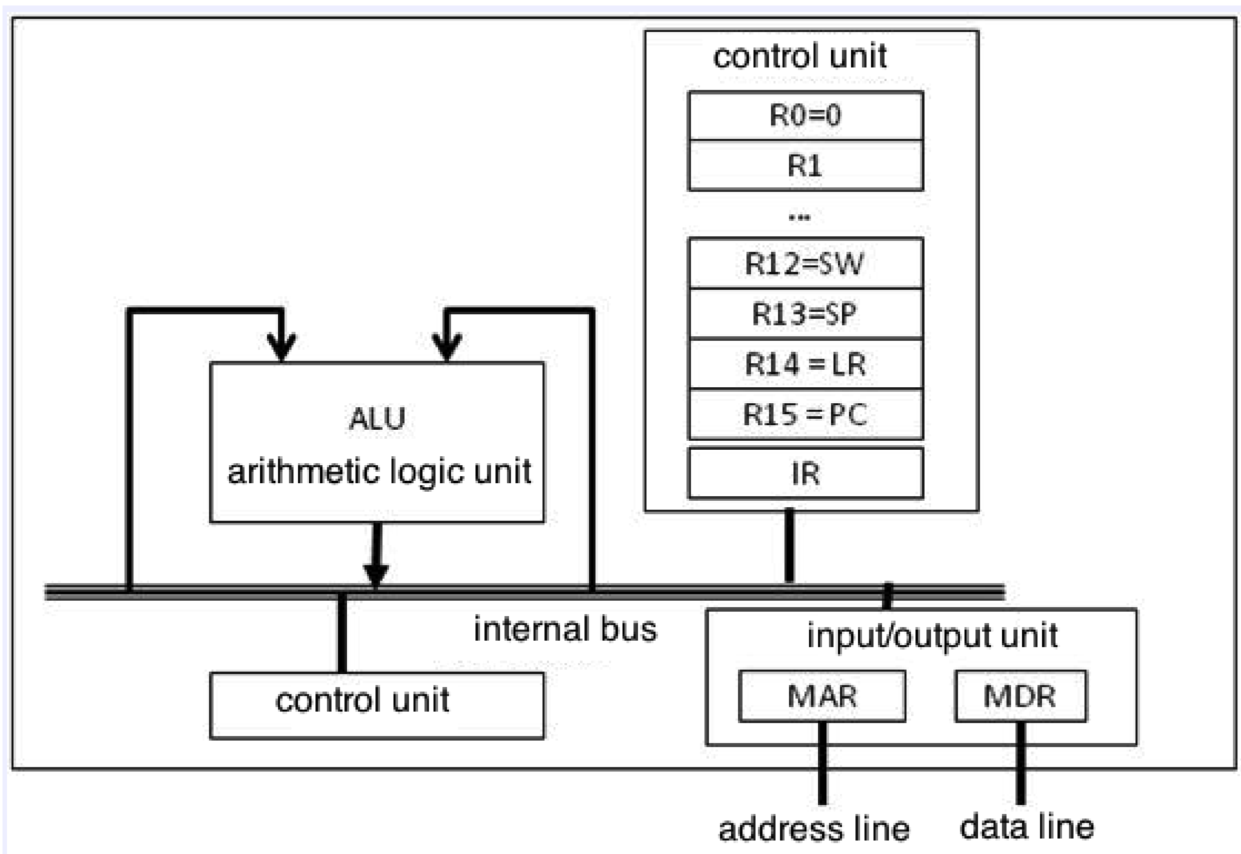


Figure 2.1: Architectural block diagram of the Cpu0 processor

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.

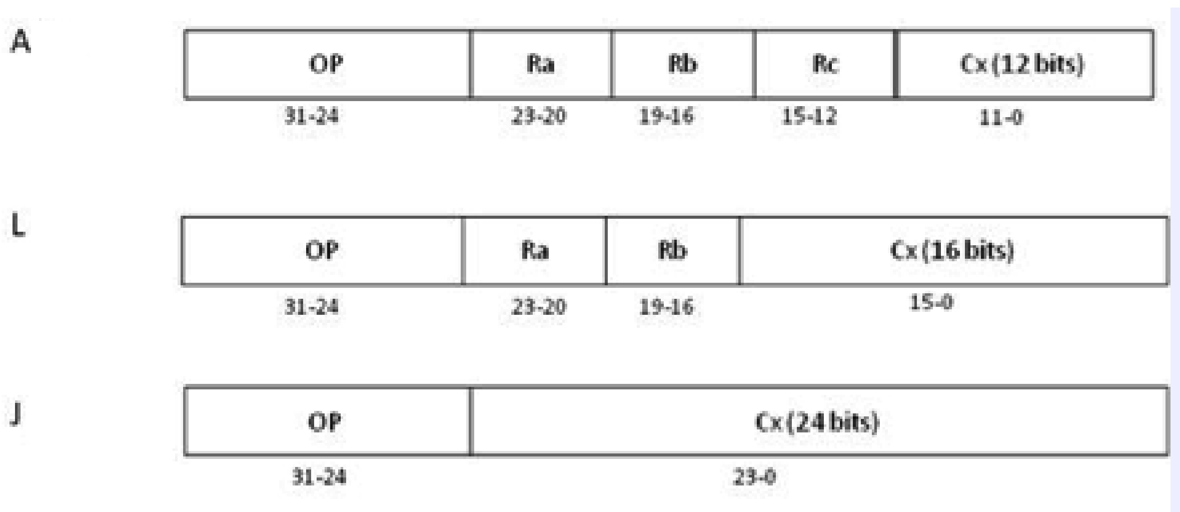


Figure 2.2: Cpu0's three instruction formats

The following table details the Cpu0 instruction set:

Table 2.1: Cpu0 Instruction Set

Format	Mnemonic	Opcode	Meaning	Syntax	Operation
L	LD	00	Load word	LD Ra, [Rb+Cx]	Ra <= [Rb+Cx]
L	ST	01	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

Continued on next page

Table 2.1 – continued from previous page

Format	Mnemonic	Opcode	Meaning	Syntax	Operation
L	ADDiu	09	Add immediate	ADDiu Ra, Rb, Cx	Ra <= (Rb + Cx)
A	CMP	10	Compare	CMP Ra, Rb	SW <= (Ra cond Rb) [4]
A	MOV	12	Move	MOV Ra, Rb	Ra <= Rb
A	ADD	13	Add	ADD Ra, Rb, Rc	Ra <= Rb + Rc
A	SUB	14	Subtract	SUB Ra, Rb, Rc	Ra <= Rb - Rc
A	MUL	15	Multiply	MUL Ra, Rb, Rc	Ra <= Rb * Rc
A	DIV	16	Divide	DIV Ra, Rb	HI<=Ra%Rb, LO<=Ra/Rb
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	SRA	1B	Shift right	SHR Ra, Rb, Cx	Ra <= (h80000000|Rb>>Cx)
A	ROL	1C	Rotate left	ROL Ra, Rb, Cx	Ra <= Rb rol Cx
A	ROR	1D	Rotate right	ROR Ra, Rb, Cx	Ra <= Rb ror Cx
A	SHL	1E	Shift left	SHL Ra, Rb, Cx	Ra <= Rb << Cx
A	SHR	1F	Shift right	SHR Ra, Rb, Cx	Ra <= Rb >> Cx
J	JEQ	20	Jump if equal (==)	JEQ Cx	if SW(==), PC <= PC + Cx
J	JNE	21	Jump if not equal (!=)	JNE Cx	if SW(!=), PC <= PC + Cx
J	JLT	22	Jump if less than (<)	JLT Cx	if SW(<), PC <= PC + Cx
J	JGT	23	Jump if greater than (>)	JGT Cx	if SW(>), PC <= PC + Cx
J	JLE	24	Jump if less than or equals (<=)	JLE Cx	if SW(<=), PC <= PC + Cx
J	JGE	25	Jump if greater than or equals (>=)	JGE Cx	if SW(>=), PC <= PC + Cx
J	JMP	26	Jump (unconditional)	JMP Cx	PC <= PC + Cx
J	SWI	2A	Software interrupt	SWI Cx	LR <= PC; PC <= Cx
J	JSUB	2B	Jump to subroutine	JSUB Cx	LR <= PC; PC <= PC + Cx
J	RET	2C	Return from subroutine	RET Cx	PC <= LR
J	IRET	2D	Return from interrupt handler	IRET	PC <= LR; INT 0
J	JR	2E	Jump to subroutine	JR Rb	LR <= PC; PC <= Rb
A	PUSH	30	Push word	PUSH Ra	[SP] <= Ra; SP -= 4
A	POP	31	Pop word	POP Ra	Ra <= [SP]; SP += 4
A	PUSHB	32	Push byte	PUSHB Ra	[SP] <= (byte)Ra; SP -= 4
A	POPB	33	Pop word	POP Ra	Ra <= (byte)[SP]; SP += 4
L	MFHI	40	Move HI to GPR	MFHI Ra	Ra <= HI
L	MFLO	41	Move LO to GPR	MFLO Ra	Ra <= LO
L	MTHI	42	Move GPR to HI	MTHI Ra	HI <= Ra
L	MTLO	43	Move GPR to LO	MTLO Ra	LO <= Ra
L	MULT	50	Multiply for 64 bits result	MULT Ra, Rb	(HI,LO) <= MULT(Ra,Rb)
L	MULTU	51	MULT for unsigned 64 bits	MULTU Ra, Rb	(HI,LO) <= MULTU(Ra,Rb)

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.

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

[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.

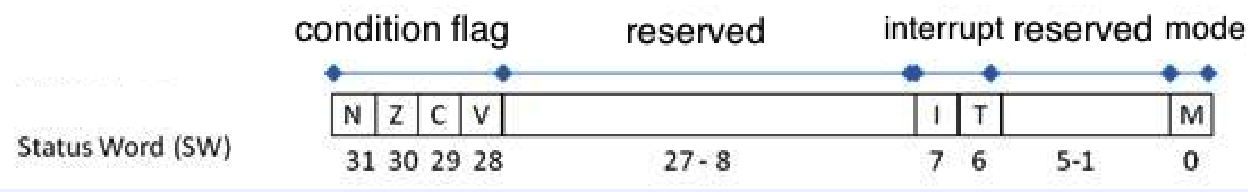


Figure 2.3: Cpu0 status word (SW) register

- 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.

2.2 LLVM Structure

The text in this and the following section comes from the AOSA chapter on LLVM written by Chris Lattner [4].

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

Figure 2.4: Three Major Components of a Three Phase Compiler

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

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

The most important win of this classical design comes when a compiler decides to support multiple source languages or target architectures. If the compiler uses a common code representation in its optimizer, then a front end can be written for any language that can compile to it, and a back end can be written for any target that can compile from it, as shown in Figure 2.5.

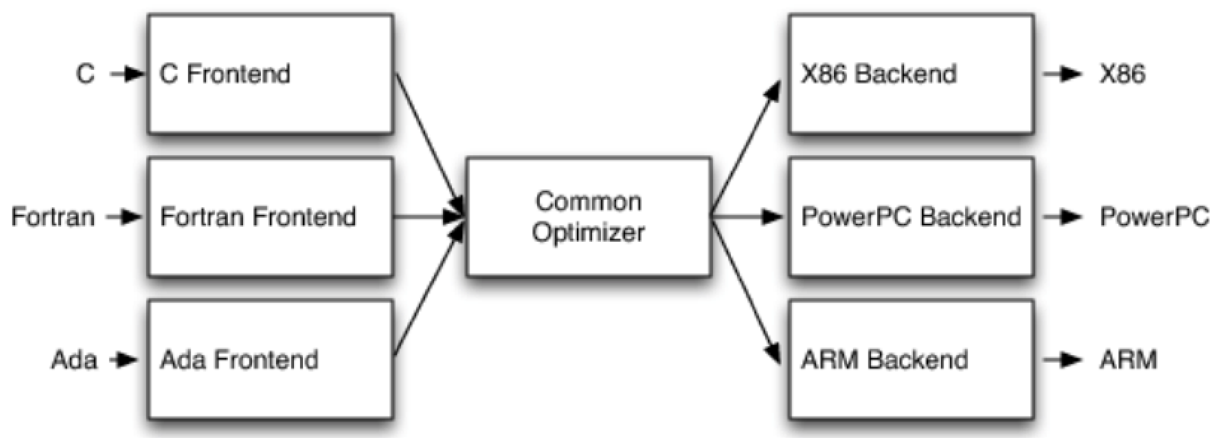


Figure 2.5: Retargetablity

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

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

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

The most important aspect of its design is the LLVM Intermediate Representation (IR), which is the form it uses to represent code in the compiler. LLVM IR is designed to host mid-level analyses and transformations that you find in the optimizer 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.

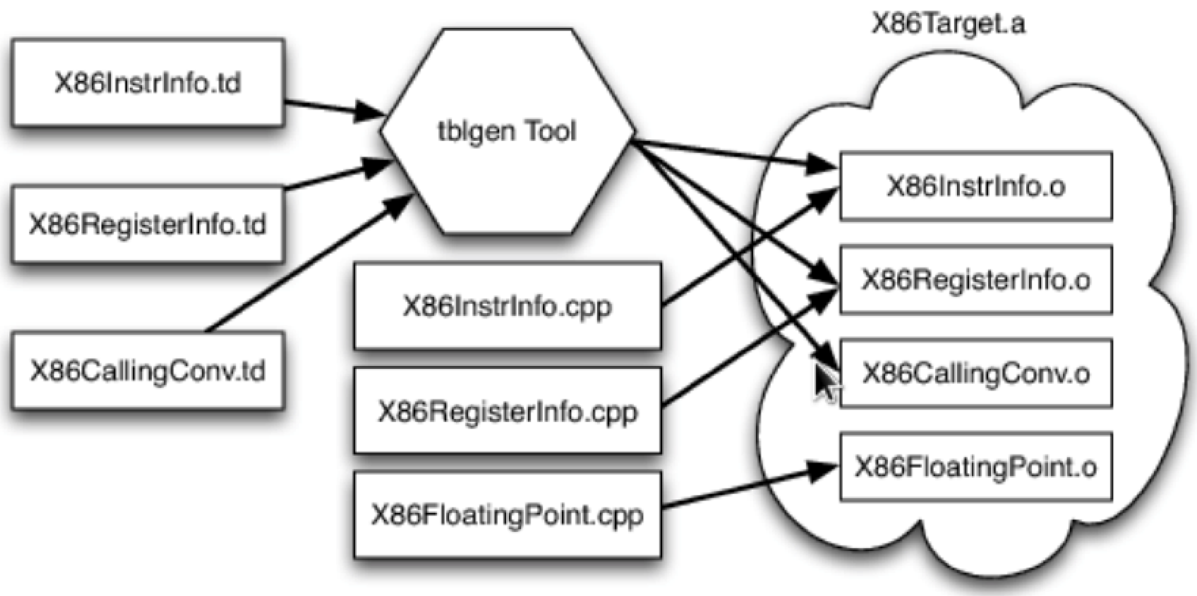


Figure 2.6: Simplified x86 Target Definition

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 [5] 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 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

[5] http://llvm.org/docs/TableGenFundamentals.html

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

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

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

LLVMBackendTutorialExampleCode/Chapter2/Cpu0InstrFormats.td

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:$imm16),
   !strconcat(instr_asm, "\t$ra, $rb, $imm16"),
   [(set RC:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {
  let isReMaterializable = 1;
}

So,
op = 0x09
instr_asm = “addiu”
OpNode = add
Od = simm16
imm_type = immSExt16
RC = CPURegs
```

Expand with FL further,

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

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

  let Opcode = op;

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

So,
op = 0x09
outs = CPURegs:$ra
ins = CPURegs:$rb,simm16:$imm16
asmstr = "addiu\t$ra, $rb, $imm16"
pattern = [(set CPURegs:$ra, (add RC:$rb, immSExt16:$imm16))]
itin = IIAlu

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

Expand with Cpu0Inst further,

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

class Cpu0Inst<dag outs, dag ins, string asmstr, list<dag> pattern,
         InstrItinClass itin, Format f>: Instruction
{
  field bits<32> Inst;
  Format Form = f;

  let Namespace = "Cpu0";

  let Size = 4;

  bits<8> Opcode = 0;

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

  let OutOperandList = outs;
  let InOperandList  = ins;

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

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

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

  let DecoderNamespace = "Cpu0";

  field bits<32> SoftFail = 0;
}

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

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

Summary with all members are,
// Inherited from parent like Instruction
Namespace = "Cpu0";
DecoderNamespace = "Cpu0";
```

```
Inst{31-24} = 0x08;
Inst{23-20} = CPURegs:$ra;
Inst{19-16} = CPURegs:$rb;
Inst{15-0}  = simm16:$imm16;
OutOperandList = CPURegs:$ra
InOperandList  = CPURegs:$rb,simm16:$imm16
AsmString = "addiu\t$ra, $rb, $imm16"
Pattern = [(set CPURegs:$ra, (add RC:$rb, immSExt16:$imm16))]
Itinerary = IIAlu
// From Cpu0Inst
Opcode = 0x09;
// From FL
ra = CPURegs:$ra
rb = CPURegs:$rb
imm16 = simm16:$imm16
```

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

2.5 Write cmake file

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

LLVMBackendTutorialExampleCode/Chapter2/CMakeLists.txt

LLVMBackendTutorialExampleCode/Chapter2/LLVMBuild.txt

CMakeLists.txt is the make information for cmake, # is comment. LLVMBuild.txt files are written in a simple variant of the INI or configuration file format. Comments are prefixed by `#` in both files. We explain the setting for these 2 files in comments. Please spend a little time to read it.

Both CMakeLists.txt and LLVMBuild.txt coexist in sub-directories `MCTargetDesc` and `TargetInfo`. Their contents indicate they will generate Cpu0Desc and Cpu0Info libraries. After building, you will find three libraries: `libLLVMCpu0CodeGen.a`, `libLLVMCpu0Desc.a` and `libLLVMCpu0Info.a` in lib/ of your build directory. For more details please see "Building LLVM with CMake" [6] and "LLVMBuild Guide" [7].

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.

[6] http://llvm.org/docs/CMake.html
[7] http://llvm.org/docs/LLVMBuild.html

LLVMBackendTutorialExampleCode/Chapter2/TargetInfo/Cpu0TargetInfo.cpp

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

LLVMBackendTutorialExampleCode/Chapter2/MCTargetDesc/Cpu0MCTargetDesc.h

LLVMBackendTutorialExampleCode/Chapter2/MCTargetDesc/Cpu0MCTargetDesc.cpp

Please see "Target Registration" [8] for reference.

2.7 Build libraries and td

The llvm source code is put in /Users/Jonathan/llvm/release/src and have llvm release-build in /Users/Jonathan/llvm/release/configure_release_build. About how to build llvm, please refer [9]. 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/ADT/Triple.h:    cpu064,
src/include/llvm/ADT/Triple.h:    cpu064el,
src/include/llvm/Support/ELF.h:  EF_CPU0_ARCH_32R2 = 0x70000000, // cpu032r2
src/include/llvm/Support/ELF.h:  EF_CPU0_ARCH_64R2 = 0x80000000, // cpu064r2
src/lib/Support/Triple.cpp:  case cpu0:    return "cpu0";
...
```

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

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

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

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

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

```
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 TargetInfo/Cpu0TargetInfo.cpp what in "section Target Registration" [10] we made. List them as follows again,

LLVMBackendTutorialExampleCode/Chapter2/TargetInfo/Cpu0TargetInfo.cpp

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

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
```

[10] http://jonathan2251.github.com/lbd/llvmstructure.html#target-registration

```
// ch3.ll
; ModuleID = 'ch3.bc'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f3
2:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:6
4-S128"
target triple = "x86_64-unknown-linux-gnu"

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

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

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

CHAPTER

THREE

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 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

include/llvm/Target/TargetInstInfo.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:
  ...
}
```

cmake_debug_build/lib/Target/Cpu0/Cpu0GenInstInfo.inc

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

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

```
  explicit Cpu0InstrInfo(Cpu0TargetMachine &TM);
};
```

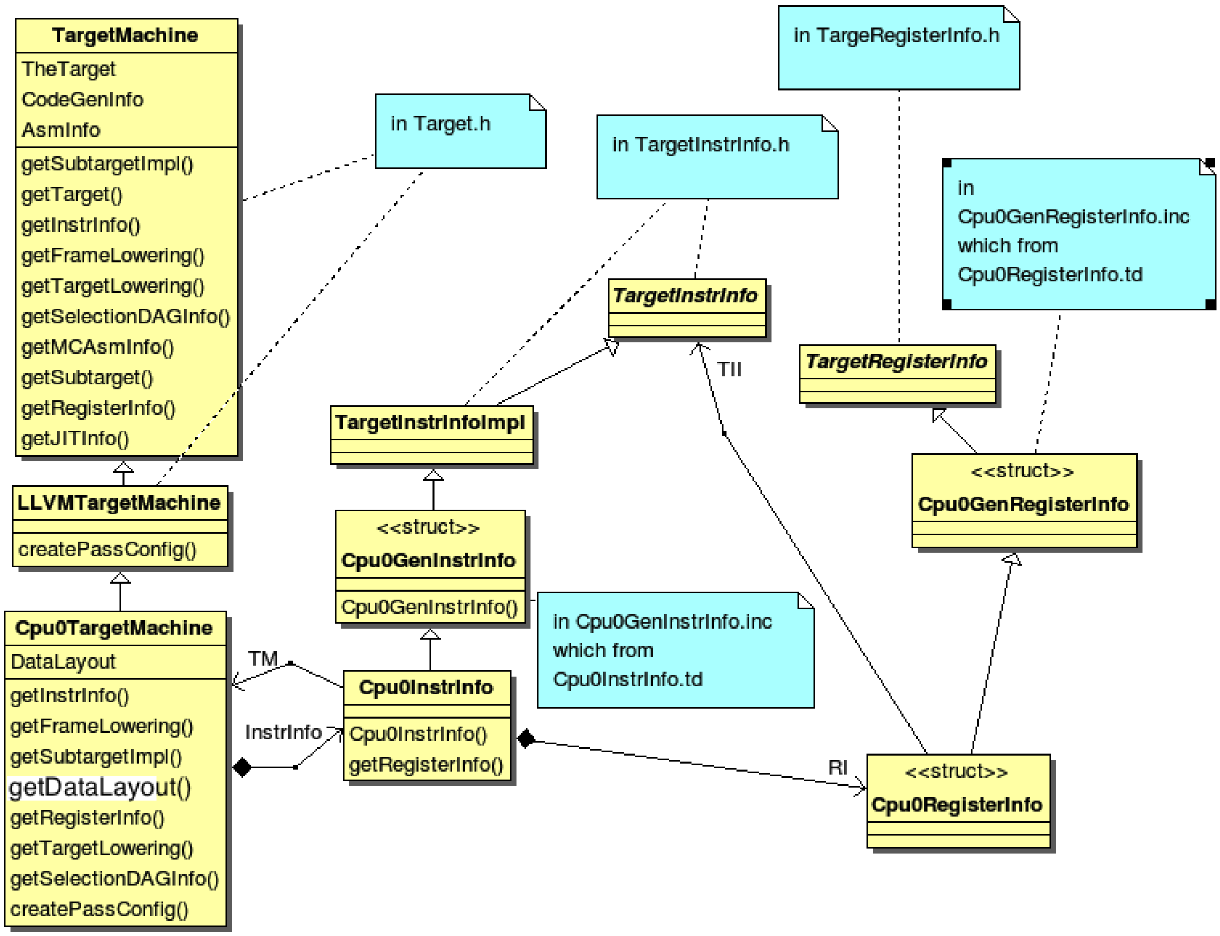


Figure 3.1: TargetMachine class diagram 1

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 3.3 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

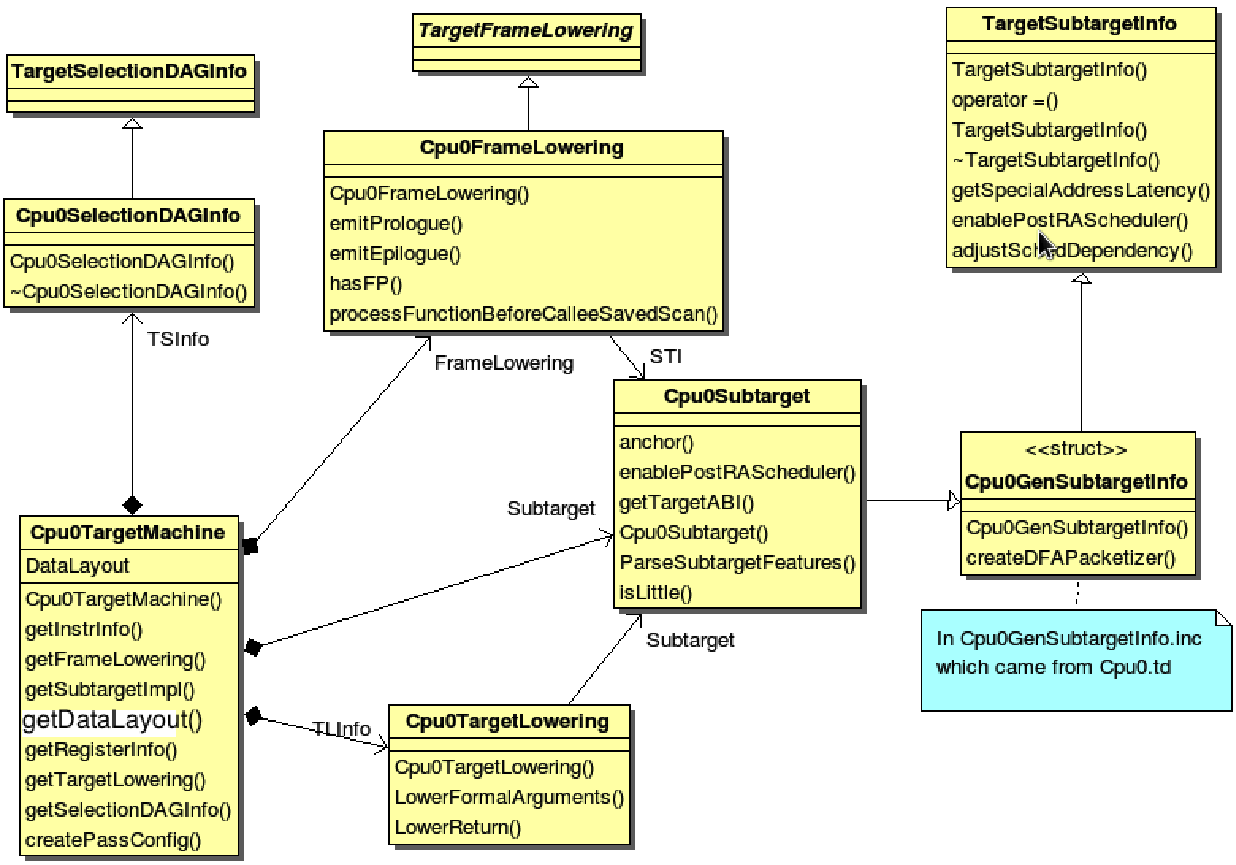


Figure 3.2: TargetMachine class diagram 2

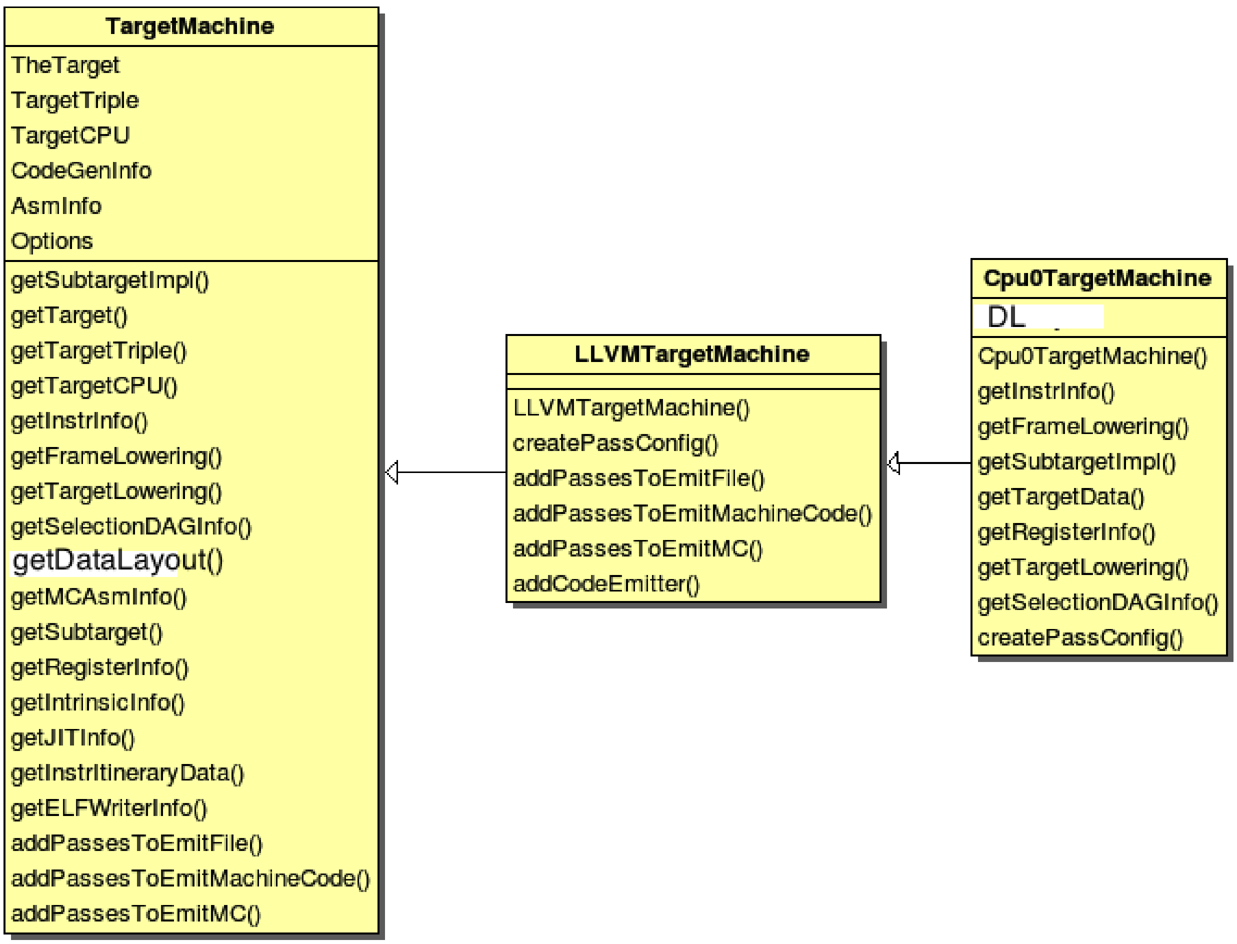


Figure 3.3: TargetMachine members and operators

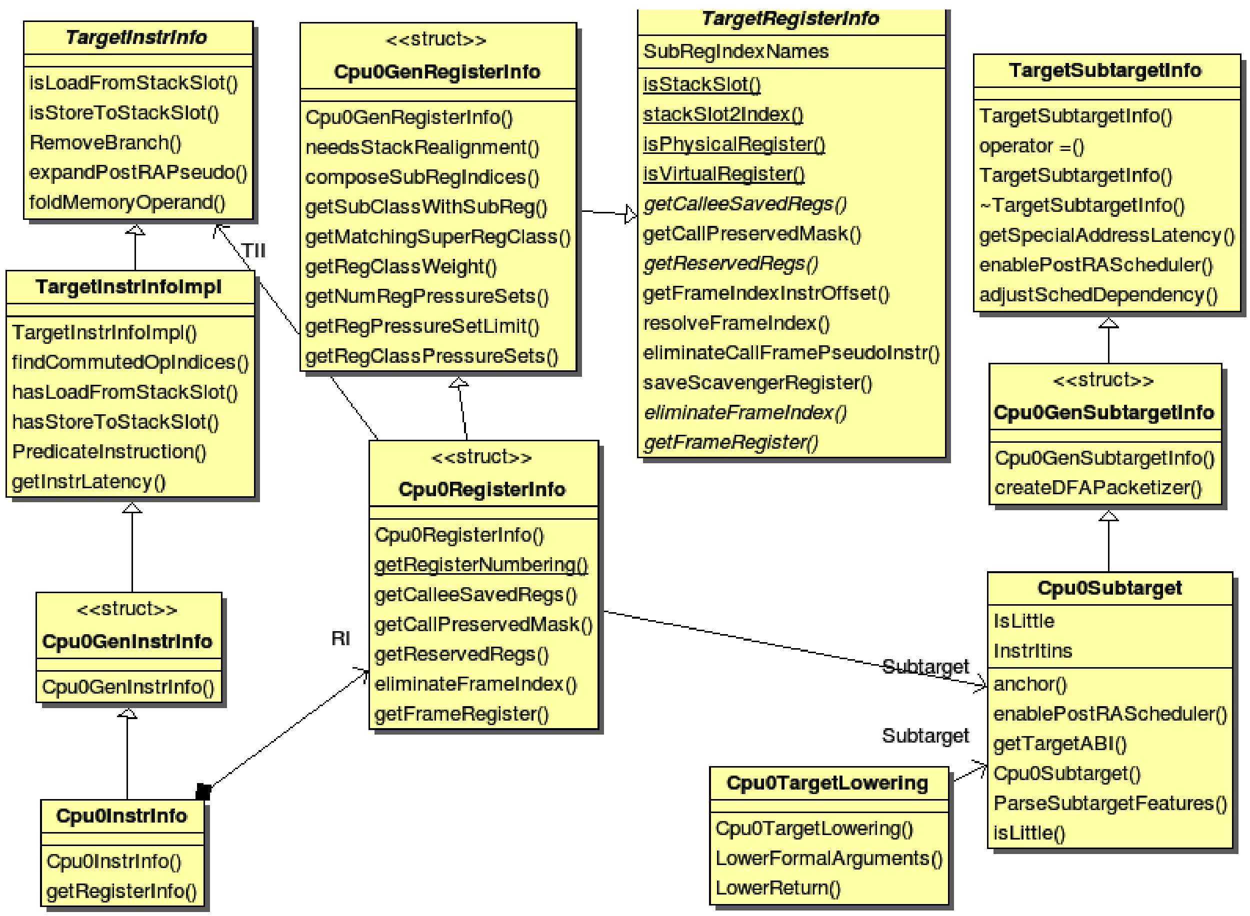


Figure 3.4: Other class members and operators

GET_INSTRINFO_HEADER". Following is the code fragment from Cpu0GenInstrInfo.inc. Code between "#if def GET_INSTRINFO_HEADER" and "#endif // GET_INSTRINFO_HEADER" will be extracted by Cpu0InstrInfo.h.

cmake_debug_build/lib/Target/Cpu0/Cpu0GenInstInfo.inc

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

Reference Write An LLVM Backend web site [1].

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

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 rebuild, then your Target library will regenerate. Command as follows,

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

3.2 Add RegisterInfo

As depicted in Figure 3.1, the Cpu0InstrInfo class should contains Cpu0RegisterInfo. So Chapter3_2/ add Cpu0RegisterInfo class (Cpu0RegisterInfo.h, Cpu0RegisterInfo.cpp), and Cpu0RegisterInfo class in files Cpu0InstrInfo.h, Cpu0InstrInfo.cpp, Cpu0TargetMachine.h, and modify CMakeLists.txt as follows,

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

LLVMBackendTutorialExampleCode/Chapter3_2/Cpu0RegisterInfo.h

LLVMBackendTutorialExampleCode/Chapter3_2/Cpu0RegisterInfo.cpp

LLVMBackendTutorialExampleCode/Chapter3_2/Cpu0InstrInfo.h

LLVMBackendTutorialExampleCode/Chapter3_2/Cpu0InstrInfo.cpp

LLVMBackendTutorialExampleCode/Chapter3_2/Cpu0TargetMachine.h

```
# CMakeLists.txt
...
add_llvm_target(Cpu0CodeGen
  ...
  Cpu0RegisterInfo.cpp
  ...
  )
```

Now, let's replace Chapter3_1/ with Chapter3_2/ of adding register class definition as command below and rebuild.

```
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_2/
* ../.
```

After that, let's try to run the `llc` compile command to see what happen,

```
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." "Make sure you includ
...
```

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

3.3 Add AsmPrinter

Chapter3_3/cpu0 contains the Cpu0AsmPrinter definition. First, we add definitions in Cpu0.td to support AssemblyWriter. Cpu0.td is added with the following fragment,

LLVMBackendTutorialExampleCode/Chapter3_3/Cpu0.td

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.

File Chapter3_3/Cpu0/InstPrinter/Cpu0InstPrinter.cpp include Cpu0GenAsmWrite.inc and call the auto-generated functions as follows,

LLVMBackendTutorialExampleCode/Chapter3_3/InstPrinter/Cpu0InstPrinter.h

LLVMBackendTutorialExampleCode/Chapter3_3/InstPrinter/Cpu0InstPrinter.cpp

Next, Cpu0MCInstLower (Cpu0MCInstLower.h, Cpu0MCInstLower.cpp), as well as Cpu0BaseInfo.h, Cpu0FixupKinds.h and Cpu0MCAsmInfo (Cpu0MCAsmInfo.h, Cpu0MCAsmInfo.cpp) in sub-directory MCTargetDesc as follows,

LLVMBackendTutorialExampleCode/Chapter3_3/Cpu0MCInstLower.h

LLVMBackendTutorialExampleCode/Chapter3_3/Cpu0MCInstLower.cpp

LLVMBackendTutorialExampleCode/Chapter3_3/MCTargetDesc/Cpu0BaseInfo.h

LLVMBackendTutorialExampleCode/Chapter3_3/MCTargetDesc/Cpu0FixupKinds.h

LLVMBackendTutorialExampleCode/Chapter3_3/MCTargetDesc/Cpu0MCAsmInfo.h

LLVMBackendTutorialExampleCode/Chapter3_3/MCTargetDesc/Cpu0MCAsmInfo.cpp

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

LLVMBackendTutorialExampleCode/MCTargetDesc/Cpu0MCTargetDesc.cpp

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_3/Cpu0AsmPrinter.h

LLVMBackendTutorialExampleCode/Chapter3_3/Cpu0AsmPrinter.cpp

The dynamic register mechanism is a good idea, right.

Except add the new .cpp files to CMakeLists.txt, please remember to add subdirectory InstPrinter, enable asmprinter, add libraries AsmPrinter and Cpu0AsmPrinter to LLVMBuild.txt as follows,

LLVMBackendTutorialExampleCode/Chapter3_3/LLVMBuild.txt

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

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

[2] http://jonathan2251.github.com/lbd/llvmstructure.html#target-registration

```
[component_1]
# Add AsmPrinter Cpu0AsmPrinter
required_libraries = AsmPrinter CodeGen Core MC Cpu0AsmPrinter Cpu0Desc
                     Cpu0Info SelectionDAG Support Target
```

Now, run Chapter3_3/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.

3.4 LLVM Code Generation Sequence

Following diagram came from tricore_llvm.pdf.

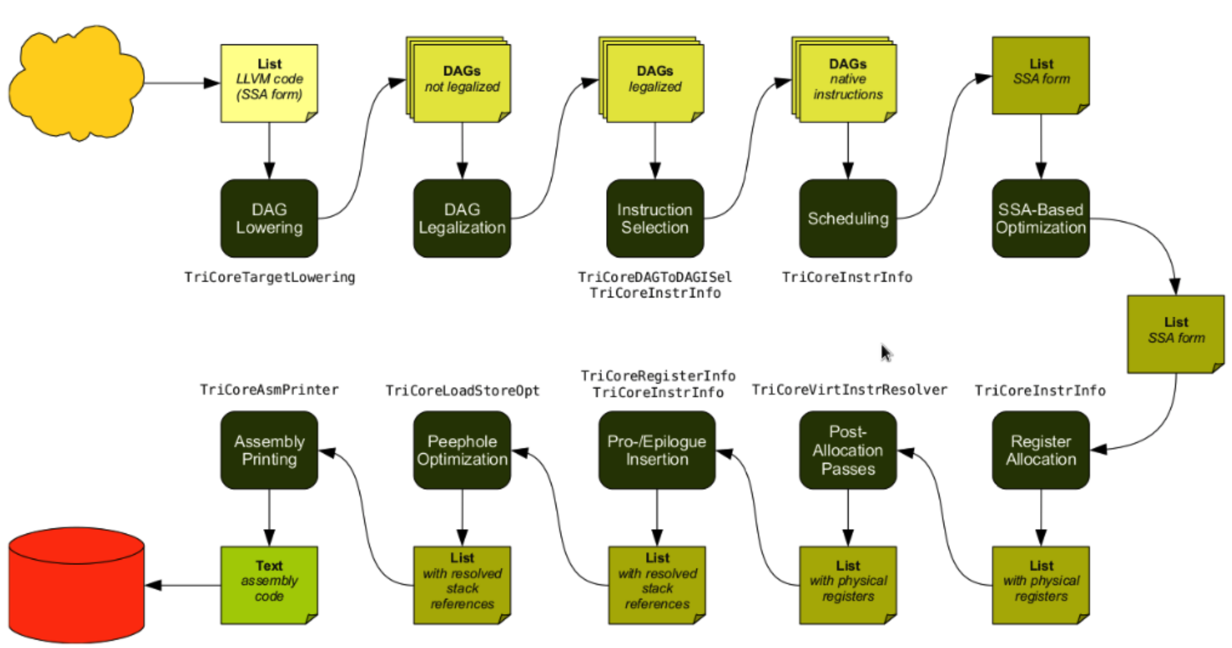


Figure 3.5: tricore_llvm.pdf: Code generation sequence. On the path from LLVM code to assembly code, numerous passes are run through and several data structures are used to represent the intermediate results.

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 isi32 type virtual
```

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

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 this section and next 2 sections article for DAG and Instruction Selection.

1. Instruction Selection

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

2. Scheduling and Formation

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

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

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

```
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.

3.5 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 it's corresponding DAG as Figure 3.6.

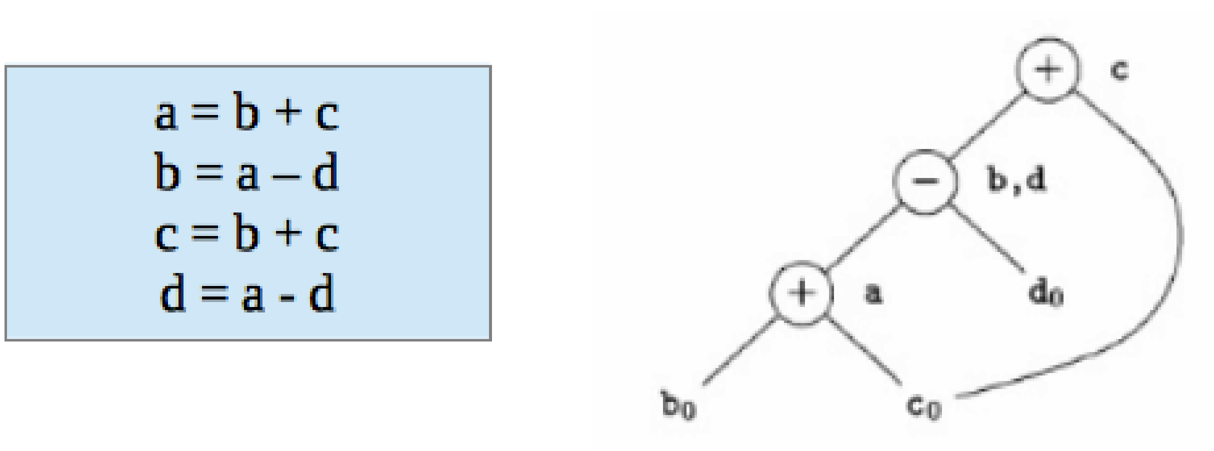


Figure 3.6: DAG example

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.

3.6 Instruction Selection

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

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

Figure 3.7: IR and it's corresponding machine instruction

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

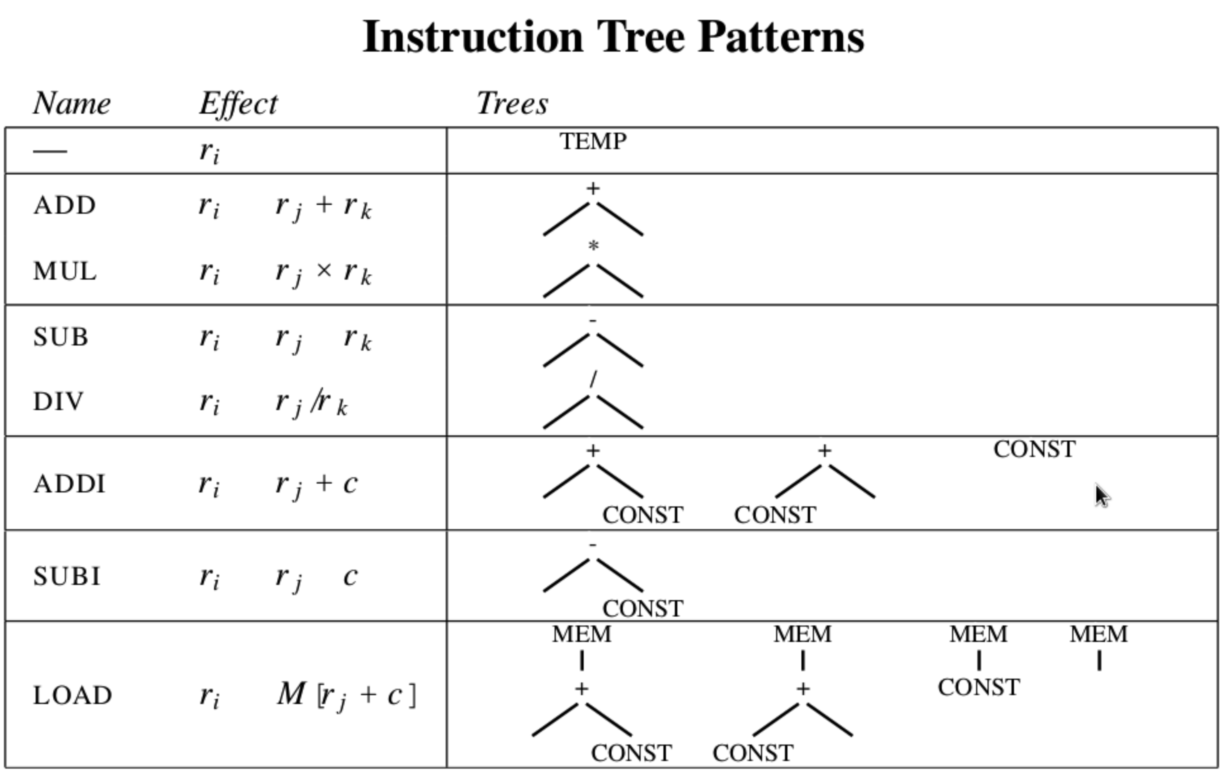


Figure 3.8: Instruction DAG representation

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

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

LLVMBackendTutorialExampleCode/Chapter3_3/Cpu0InstrFormats.td

LLVMBackendTutorialExampleCode/Chapter3_3/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 the pattern match which bind the IR node **add** and instruction ADDiu which defined in Cpu0InstrInfo.td. For the example IR node "add %a, 5", it 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**.

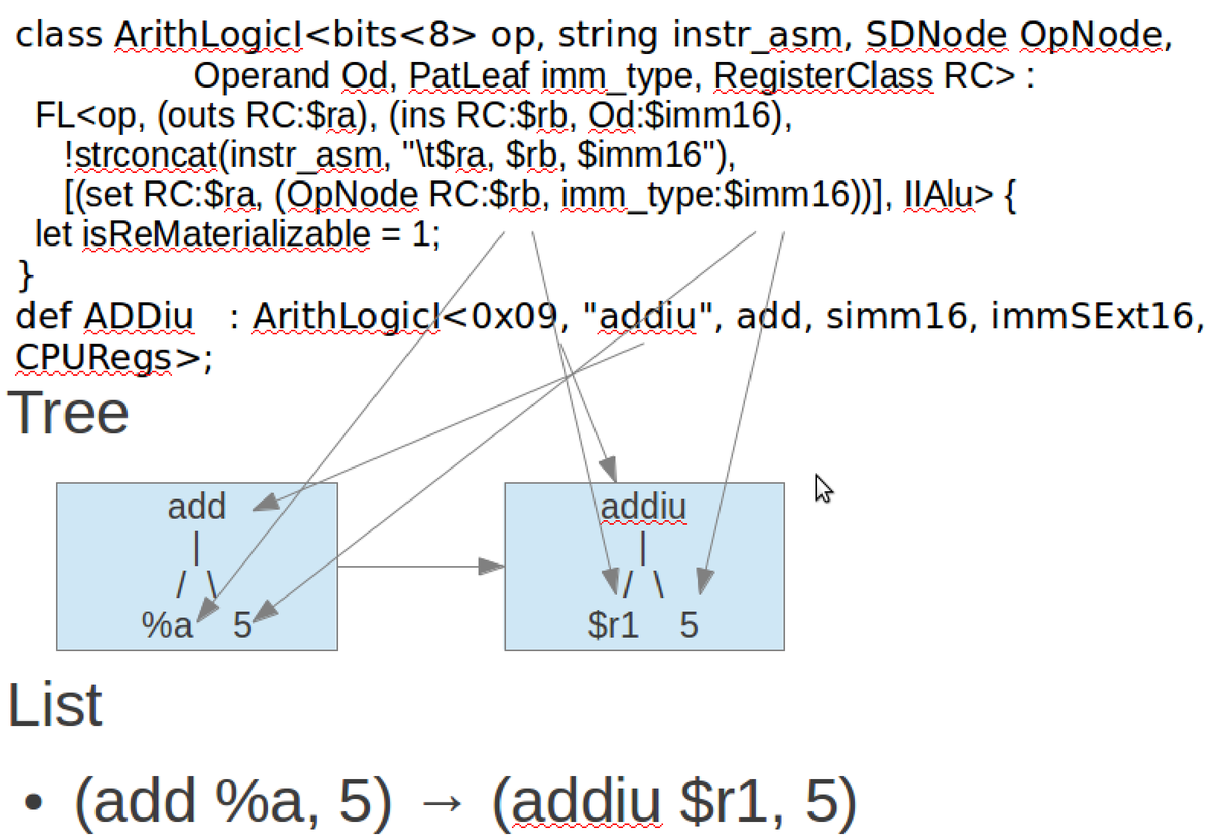


Figure 3.9: Pattern match for ADDiu instruction and IR node add

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;
```

3.7 Add Cpu0DAGToDAGISel class

The IR DAG to machine instruction DAG transformation is introduced in the previous section. Now, let's check what IR DAG node 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_3/Cpu0InstrInfo.td

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

LLVMBackendTutorialExampleCode/Chapter3_4/Cpu0TargetMachine.cpp

LLVMBackendTutorialExampleCode/Chapter3_4/Cpu0ISelDAGToDAG.cpp

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

LLVMBackendTutorialExampleCode/Chapter3_4/Cpu0InstrInfo.cpp

Build Chapter3_4, run it, we find the error message in Chapter3_3 is gone. The new error message for Chapter3_4 as follows,

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

3.8 Add Prologue/Epilogue functions

Following came from tricore_llvm.pdf section "4.4.2 Non-static Register Information ".

For some target architectures, some aspects of the target architecture's register set are dependent upon variable factors and have to be determined at runtime. As a consequence, they cannot be generated statically from a TableGen description – although that would be possible for the bulk of them in the case of the TriCore backend. Among them are the following points:

- Callee-saved registers. Normally, the ABI specifies a set of registers that a

function must save on entry and restore on return if their contents are possibly modified during execution.

- Reserved registers. Although the set of unavailable registers is already

defined in the TableGen file, TriCoreRegisterInfo contains a method that marks all non-allocatable register numbers in a bit vector.

The following methods are implemented:

- emitPrologue() inserts prologue code at the beginning of a function. Thanks to TriCore's context model, this is a trivial task as it is not required to save any registers manually. The only thing that has to be done is reserving space for the function's stack frame by decrementing the stack pointer. In addition, if the function needs a frame pointer, the frame register %a14 is set to the old value of the stack pointer beforehand.

- emitEpilogue() is intended to emit instructions to destroy the stack frame

and restore all previously saved registers before returning from a function. However, as %a10 (stack pointer), %a11 (return address), and %a14 (frame pointer, if any) are all part of the upper context, no epilogue code is needed at all. All cleanup operations are performed implicitly by the ret instruction.

- eliminateFrameIndex() is called for each instruction that references a word

of data in a stack slot. All previous passes of the code generator have been addressing stack slots through an abstract frame index and an immediate offset. The purpose of this function is to translate such a reference into a register–offset pair. Depending on whether the machine function that contains the instruction has a fixed or a variable stack frame, either the stack pointer %a10 or the frame pointer %a14 is used as the base register. The offset is computed accordingly. Figure 3.10 demonstrates for both cases how a stack slot is addressed.

If the addressing mode of the affected instruction cannot handle the address because the offset is too large (the offset field has 10 bits for the BO addressing mode and 16 bits for the BOL mode), a sequence of instructions is emitted that explicitly computes the effective address. Interim results are put into an unused address register. If none is available, an already occupied address register is scavenged. For this purpose, LLVM's framework offers a class named RegScavenger that takes care of all the details.

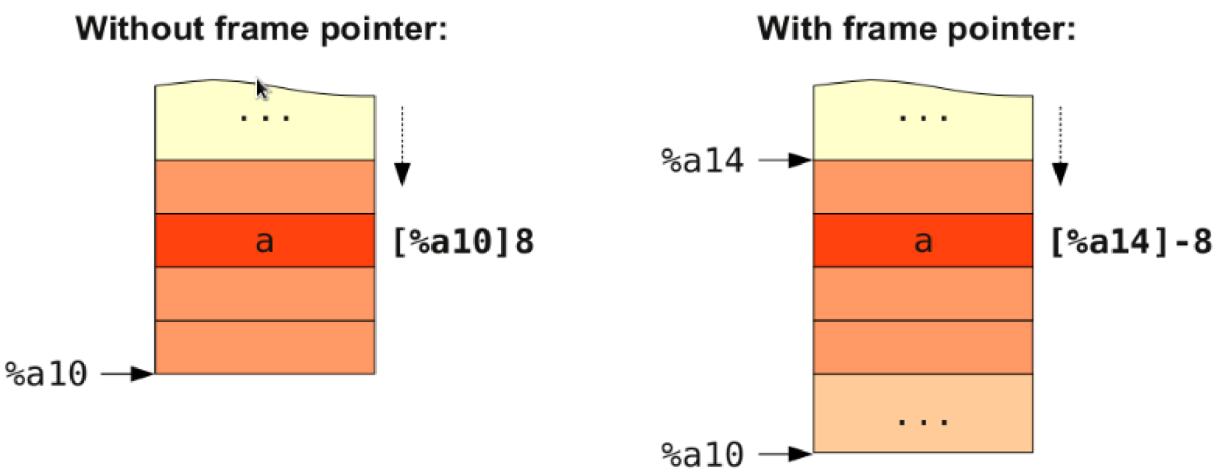


Figure 3.10: Addressing of a variable a located on the stack. If the stack frame has a variable size, slot must be addressed relative to the frame pointer

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

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

  .section .mdebug.abi32
  .previous
  .file "ch3.bc"
  .text
```

```
  .globl  main
  .align  2
  .type main,@function
  .ent  main                    # @main
main:
  .cfi_startproc
  .frame  $sp,8,$lr
  .mask   0x00000000,0
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -8
$tmp1:
  .cfi_def_cfa_offset 8
  addiu $2, $zero, 0
  st  $2, 4($sp)
  addiu $sp, $sp, 8
  ret $lr
  .set  macro
  .set  reorder
  .end  main
$tmp2:
  .size main, ($tmp2)-main
  .cfi_endproc
```

LLVM get the stack size by parsing IR and counting how many virtual registers is assigned to local variables. After that, it call emitPrologue(). This function will emit machine instructions to adjust sp (stack pointer register) for local variables since we don't use fp (frame pointer register). For our example, it will emit the instructions,

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

The emitEpilogue will emit "addiu $sp, $sp, 8", 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_5/Cpu0FrameLowering.h

LLVMBackendTutorialExampleCode/Chapter3_5/Cpu0FrameLowering.cpp

LLVMBackendTutorialExampleCode/Chapter3_5/Cpu0RegisterInfo.cpp

After add these Prologue and Epilogue functions, and build with Chapter3_5/Cpu0. Now we are ready to compile our example code ch3.bc into cpu0 assembly code. Following is the command and output file ch3.cpu0.s,

```
118-165-78-230: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
```

3.9 Summary of this Chapter

We have finished a simple assembler for cpu0 which only support **addiu**, **st** and **ret** 3 instructions.

We are satisfied with this result. But you may think "After so many codes we program, and just get the 3 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 around 3050 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.

CHAPTER

FOUR

ADDING ARITHMETIC AND LOCAL POINTER SUPPORT

This chapter add more cpu0 arithmetic instructions support first. The logic operation **"not"** support and translation in section Operator "not" !. The section Display llvm IR nodes with Graphviz will show you the DAG optimization steps and their corresponding `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. In section Adjust cpu0 instructions, we adjust cpu0 instructions to support some data type for C language. The section Local variable pointer introduce you the local variable pointer translation. Finally, section Operator mod, % take care the C operator %.

4.1 Support arithmetic instructions

Run the Chapter3_5/Cpu0 `llc` with input file ch4_1_1.bc will get the error as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch4_1_1.cpp

```
118-165-78-230:InputFiles Jonathan$ clang -c ch4_1_1.cpp -emit-llvm -o
ch4_1_1.bc
118-165-78-230:InputFiles Jonathan$ llvm-dis ch4_1_1.bc -o ch4_1_1.ll
118-165-78-230:InputFiles Jonathan$ cat ch4_1_1.ll
; ModuleID = 'ch4_1_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"

define i32 @main() nounwind uwtable ssp {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 0, i32* %1
  store i32 5, i32* %a, align 4
  store i32 2, i32* %b, align 4
  store i32 0, i32* %c, align 4
  %2 = load i32* %a, align 4
  %3 = load i32* %b, align 4
  %4 = add nsw i32 %2, %3
```

```
  store i32 %4, i32* %c, align 4
  %5 = load i32* %c, align 4
  ret i32 %5
}

118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch4_1_1.bc -o
ch4_1_1.cpu0.s
LLVM ERROR: Cannot select: 0x7ff02102b010: i32 = add 0x7ff02102ae10, ...
...
```

This error says we have not instructions to translate IR DAG node **add**. The ADDiu instruction is defined for node **add** with operands of 1 register and 1 immediate. This node **add** is for 2 registers. So, appending the following code to Cpu0InstrInfo.td and Cpu0Schedule.td in Chapter4_1/,

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;
}

class CmpInstr<bits<8> op, string instr_asm,
          InstrItinClass itin, RegisterClass RC, bit isComm = 0>:
  FA<op, (outs RC:$SW), (ins RC:$ra, RC:$rb),
   !strconcat(instr_asm, "\t$ra, $rb"), [], itin> {
  let rc = 0;
  let shamt = 0;
  let isCommutable = isComm;
}
...
// Shifts
class shift_rotate_imm<bits<8> op, bits<4> isRotate, string instr_asm,
             SDNode OpNode, PatFrag PF, Operand ImmOpnd,
             RegisterClass RC>:
  FA<op, (outs RC:$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 Imediate
class LoadUpper<bits<8> op, string instr_asm, RegisterClass RC, Operand Imm>:
  FL<op, (outs RC:$ra), (ins Imm:$imm16),
   !strconcat(instr_asm, "\t$ra, $imm16"), [], IIAlu> {
  let rb = 0;
  let neverHasSideEffects = 1;
  let isReMaterializable = 1;
}
...
/// Arithmetic Instructions (3-Operand, R-Type)
def CMP   : CmpInstr<0x10, "cmp", IIAlu, CPURegs, 1>;
def ADD     : ArithLogicR<0x13, "add", add, IIAlu, CPURegs, 1>;
def SUB     : ArithLogicR<0x14, "sub", sub, IIAlu, CPURegs, 1>;
def MUL     : ArithLogicR<0x15, "mul", mul, IIImul, CPURegs, 1>;
def DIV     : ArithLogicR<0x16, "div", sdiv, IIIdiv, CPURegs, 1>;
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>;

/// Shift Instructions
def ROL     : shift_rotate_imm32<0x1C, 0x01, "rol", rotl>;
def ROR     : shift_rotate_imm32<0x1D, 0x01, "ror", rotr>;
def SHL     : shift_rotate_imm32<0x1E, 0x00, "shl", shl>;
// work, it's for ashr llvm IR instruction
//def SHR     : shift_rotate_imm32<0x1F, 0x00, "sra", sra>;
// work, it's for lshr llvm IR instruction
def SHR     : shift_rotate_imm32<0x1F, 0x00, "shr", srl>;


// 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.

Now, let's build Chapter4_1/ and run with input file ch4_1_2.cpp. This version can process **+, -, *, /, &, |, ^, <<,** and **>>** operators in C language. The corresponding llvm IR instructions are **add, sub, mul, sdiv, and, or, xor, shl, ashr**. IR instruction **sdiv** stand for signed div while **udiv** is for unsigned div. 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

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

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 Micosoft 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). We assume Cpu0 shr instruction is "shift with zero filled", and define it with IR DAG node srl. But at that way, Cpu0 will fail to compile x >> 1 in case of x is signed integer because clang and most compilers translate it into ashr, which meaning "shift with sign extension fill". Similarly, Cpu0 div instruction, has the same problem. We assume Cpu0 div instruction is for sdiv which can take care both positive and negative integer, but it will fail for divide operation "/ "on unsigned integer operand in C.

If we consider the x >> 1 definition is x = x/2. In case of x is unsigned int, range x is 0 ~ 4G-1 (0 ~ 0xFFFFFFFF) in 32 bits register, implement shift >> 1 by "shift with zero filled" is correct and satisfy the definition x = x/2, but "shift with sign extension fill" is not correct for range 2G ~ 4G-1. In case of x is signed int, range x is -2G ~ 2G-1, implement x >> 1 by "shift with sign extension fill" is correct for the definition, but "shift with zero filled" is not correct for range x is -2G ~ -1. So, if x = x/2 is definition for x >> 1, in order to satisfy the definition in both unsigned and signed integer of x, we need those two instructions, "shift with zero filled" and "shift with sign extension fill".

Again, consider the x << 1 definition is x = x*2. We apply the x << 1 with "shift 1 bit to left and fill the least bit with 0". In case of x is unsigned int, x << 1 satisfy the definition in range 0 ~ 2G-1, and x is overflow when x > 2G-1 (no need to care what the register value is because overflow). In case of x is signed int, x << 1 is correct for -1G ~ 1G-1; and x is overflow for -2G ~ -1G-1 or 1G ~ 2G-1. So, implementation by "shift 1bit to left and fill the least bit with 0" satisfy the definition x = x*2 for x << 1, no matter operand x is signed or unsigned int.

Micorsoft implementation references as [2].

The sub-section "'ashr' Instruction" and sub-section "'lshr' Instruction" of [3].

The version Chapter4_1 just add 70 lines code in td files. With these 70 lines code, it process 9 operators more for C language and their corresponding llvm IR instructions. The arithmetic instructions are easy to implement by add the definition in td file only.

4.2 Operator "not" !

Files ch4_2.cpp and ch4_2.bc are the C source code for **"not"** boolean operator and it's corresponding llvm IR. List them as follows,

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

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

LLVMBackendTutorialExampleCode/InputFiles/ch4_2.cpp

```
; ModuleID = 'ch4_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"

define i32 @main() nounwind ssp {
entry:
  %retval = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  store i32 0, i32* %retval
  store i32 5, i32* %a, align 4
  store i32 0, i32* %b, align 4
  %0 = load i32* %a, align 4        // a = %0
  %tobool = icmp ne i32 %0, 0   // ne: stand for not egual
  %lnot = xor i1 %tobool, true
  %conv = zext i1 %lnot to i32
  store i32 %conv, i32* %b, align 4
  %1 = load i32* %b, align 4
  ret i32 %1
}
```

As above comment, b = !a, translate to (xor (icmp ne i32 %0, 0), true). The %0 is the virtual register of variable **a** and the result of (icmp ne i32 %0, 0) is 1 bit size. To prove the translation is correct. Let's assume %0 != 0 first, then the (icmp ne i32 %0, 0) = 1 (or true), and (xor 1, 1) = 0. When %0 = 0, (icmp ne i32 %0, 0) = 0 (or false), and (xor 0, 1) = 1. So, the translation is correct.

Now, let's run ch4_2.bc with Chapter4_1/ with `llc -debug` option to get result as follows,

```
118-165-16-22:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/llc -march=cpu0 -debug -relocation-model=pic
-filetype=asm ch4_3.bc -o ch4_3.cpu0.s
...

=== main
Initial selection DAG: BB#0 'main:entry'
SelectionDAG has 20 nodes:
...
  0x7ffb7982ab10: <multiple use>
        0x7ffb7982ab10: <multiple use>
        0x7ffb7982a210: <multiple use>
        0x7ffb7982ac10: ch = setne [ORD=5]

      0x7ffb7982ad10: i1 = setcc 0x7ffb7982ab10, 0x7ffb7982a210, 0x7ffb7982ac10
      [ORD=5]

      0x7ffb7982ae10: i1 = Constant<-1> [ORD=6]

    0x7ffb7982af10: i1 = xor 0x7ffb7982ad10, 0x7ffb7982ae10 [ORD=6]

  0x7ffb7982b010: i32 = zero_extend 0x7ffb7982af10 [ORD=7]
...
Replacing.3 0x7ffb7982af10: i1 = xor 0x7ffb7982ad10, 0x7ffb7982ae10 [ORD=6]

With: 0x7ffb7982d210: i1 = setcc 0x7ffb7982ab10, 0x7ffb7982a210, 0x7ffb7982cf10
```

```
Optimized lowered selection DAG: BB#0 'main:'
SelectionDAG has 17 nodes:
...
   0x7ffb7982ab10: <multiple use>
        0x7ffb7982ab10: <multiple use>
        0x7ffb7982a210: <multiple use>
        0x7ffb7982cf10: ch = seteq

      0x7ffb7982d210: i1 = setcc 0x7ffb7982ab10, 0x7ffb7982a210, 0x7ffb7982cf10

    0x7ffb7982b010: i32 = zero_extend 0x7ffb7982d210 [ORD=7]
...
Type-legalized selection DAG: BB#0 'main:entry'
SelectionDAG has 18 nodes:
...
    0x7ffb7982ab10: <multiple use>
        0x7ffb7982ab10: <multiple use>
        0x7ffb7982a210: <multiple use>
        0x7ffb7982cf10: ch = seteq [ID=-3]

      0x7ffb7982ac10: i32 = setcc 0x7ffb7982ab10, 0x7ffb7982a210, 0x7ffb7982cf10
       [ID=-3]

      0x7ffb7982ad10: i32 = Constant<1> [ID=-3]

    0x7ffb7982ae10: i32 = and 0x7ffb7982ac10, 0x7ffb7982ad10 [ID=-3]
...
ISEL: Starting pattern match on root node: 0x7ffb7982ac10: i32 = setcc
0x7ffb7982ab10, 0x7ffb7982a210, 0x7ffb7982cf10 [ID=14]

  Initial Opcode index to 0
  Match failed at index 0
LLVM ERROR: Cannot select: 0x7ffb7982ac10: i32 = setcc 0x7ffb7982ab10,
0x7ffb7982a210, 0x7ffb7982cf10 [ID=14]
  0x7ffb7982ab10: i32,ch = load 0x7ffb7982aa10, 0x7ffb7982a710,
  0x7ffb7982a410<LD4[%a]> [ORD=4] [ID=13]
  0x7ffb7982a710: i32 = FrameIndex<1> [ORD=2] [ID=5]
  0x7ffb7982a410: i32 = undef [ORD=1] [ID=3]
  0x7ffb7982a210: i32 = Constant<0> [ORD=1] [ID=1]
In function: main
```

The (setcc %1, %2, setne) and (xor %3, -1) in “Initial selection DAG” stage corresponding (icmp %1, %2, ne) and (xor %3, 1) in ch4_2.bc. The argument in xor is 1 bit size (1 and -1 are same, they are all represented by 1). The (zero_extend %4) of “Initial selection DAG” corresponding (zext i1 %lnot to i32) of ch4_2.bc. As above it translate 2 DAG nodes (setcc %1, %2, setne) and (xor %3, -1) into 1 DAG node (setcc %1, %2, seteq) in “Optimized lowered selection DAG” stage. This translation is right since for 1 bit size, (xor %3, 1) and (not %3) has same result, and (not (setcc %1, %2, setne)) is equal to (setcc %1, %2, seteq). In “Optimized lowered selection DAG” stage, it also translate (zero_extern i1 %lnot to 32) into (and %lnot, 1). (zero_extern i1 %lnot to 32) just expand the %lnot to i32 32 bits result, so translate into (and %lnot, 1) is correct. It fails at (setcc %1, %2, seteq).

Run it with Chapter4_2/ which added code as below, to get the following result.

LLVMBackendTutorialExampleCode/Chapter4_2/Cpu0InstrInfo.td

```
def : Pat<(not CPURegs:$in),
      (XOR CPURegs:$in, (LDI ZERO, 1))>;
```

```
// setcc patterns
multiclass SeteqPats<RegisterClass RC, Instruction XOROp,
                     Register ZEROReg> {
  def : Pat<(seteq RC:$lhs, RC:$rhs),
            (XOROp (XOROp RC:$lhs, RC:$rhs), (LDI ZERO, 1))>;
}

defm : SeteqPats<CPURegs, XOR, ZERO>;
```

```
118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -debug -filetype=asm ch4_2.bc
-o ch4_2.cpu0.s
...
ISEL: Starting pattern match on root node: 0x7fbc6902ac10: i32 = setcc
0x7fbc6902ab10, 0x7fbc6902a210, 0x7fbc6902cf10 [ID=14]

  Initial Opcode index to 365
  Created node: 0x7fbc6902af10: i32 = XOR 0x7fbc6902ab10, 0x7fbc6902a210

  Created node: 0x7fbc6902d510: i32 = LDI 0x7fbc6902d310, 0x7fbc6902d410

  Morphed node: 0x7fbc6902ac10: i32 = XOR 0x7fbc6902af10, 0x7fbc6902d510

ISEL: Match complete!
=> 0x7fbc6902ac10: i32 = XOR 0x7fbc6902af10, 0x7fbc6902d510
```

Chapter4_2/ defined seteq DAG pattern. It translate (setcc %1, %2, seteq) into (xor (xor %1, %2), (ldi $0, 1) in "Instruction selection" stage by the rule defined in Cpu0InstrInfo.td as above.

After xor, the (and %4, 1) is translated into (and $2, (ldi $3, 1)) which is defined before already. List the asm file ch4_2.cpu0.s code fragment as below, you can check it with the final result.

```
118-165-16-22:InputFiles Jonathan$ cat ch4_2.cpu0.s
...
# BB#0:                                   # %entry
    addiu   $sp, $sp, -16
tmp1:
    .cfi_def_cfa_offset 16
    addiu   $2, $zero, 0
    st  $2, 12($sp)
    addiu   $3, $zero, 5
    st  $3, 8($sp)
    st  $2, 4($sp)
    ld  $3, 8($sp)
    xor $2, $3, $2
    ldi $3, 1
    xor $2, $2, $3
    addiu   $3, $zero, 1
    and $2, $2, $3
    st  $2, 4($sp)
    addiu   $sp, $sp, 16
    ret $lr
...
```

4.3 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 it help 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
```

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

---

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

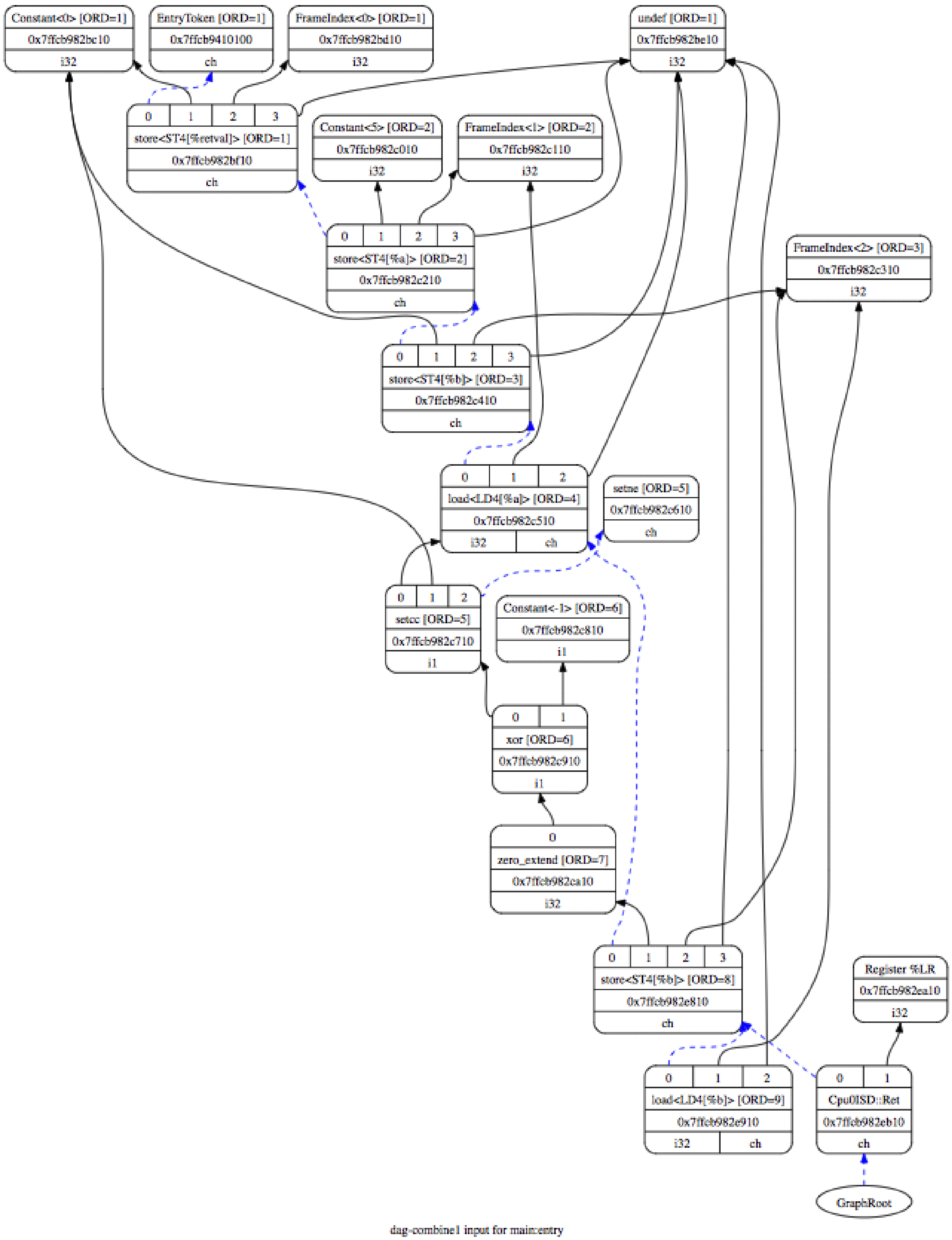


Figure 4.1: llc option -view-dag-combine1-dags graphic view

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.

4.4 Adjust cpu0 instructions

We decide add instructions udiv and sra to avoid compiler errors for C language operators **"/"** in unsigned int and **">>"** in signed int as section Support arithmetic instructions mentioned. To support these 2 operators, we only need to add these code in Cpu0InstrInfo.td as follows,

LLVMBackendTutorialExampleCode/Chapter4_4/Cpu0InstrInfo.td

```
//  Cpu0InstsInfo.td
...
def UDIV    : ArithLogicR<0x17, "udiv", udiv, IIIdiv, CPURegs, 1>;
...
/// Shift Instructions
// work, sra for ashr llvm IR instruction
def SRA     : shift_rotate_imm32<0x1B, 0x00, "sra", sra>;
```

To use addiu only instead of ldi, change Cpu0InstrInfo.td as follows,

LLVMBackendTutorialExampleCode/Chapter4_4/Cpu0InstrInfo.td

```
//def LDI     : MoveImm<0x08, "ldi", add, simm16, immSExt16, CPURegs>;
...
// setcc patterns
multiclass SeteqPats<RegisterClass RC, Instruction XOROp> {
  def : Pat<(seteq RC:$lhs, RC:$rhs),
      (XOROp (XOROp RC:$lhs, RC:$rhs), (ADDiu ZERO, 1))>;
}

defm : SeteqPats<CPURegs, XOR>;
```

Run ch4_4.cpp with code Chapter4_4/ which support udiv, sra, and use addiu only instead of ldi, will get the result as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch4_4.cpp

```
118-165-13-40:InputFiles Jonathan$ clang -c ch4_4.cpp -emit-llvm -o ch4_4.bc
118-165-13-40: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
118-165-13-40:InputFiles Jonathan$ cat ch4_4.cpu0.s
    ...
    addiu   $sp, $sp, -24
    addiu   $2, $zero, 0
    ...
    udiv    $2, $3, $2
    st  $2, 0($sp)
    ld  $2, 16($sp)
```

```
sra $2, $2, 2
...
```

4.5 Local variable pointer

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

LLVMBackendTutorialExampleCode/Chapter4_5/Cpu0InstrInfo.td

```
def mem_ea : Operand<i32> {
  let PrintMethod = "printMemOperandEA";
  let MIOperandInfo = (ops CPURegs, simm16);
  let EncoderMethod = "getMemEncoding";
}
...
class EffectiveAddress<string instr_asm, RegisterClass RC, Operand Mem> :
  FMem<0x09, (outs RC:$ra), (ins Mem:$addr),
     instr_asm, [(set RC:$ra, addr:$addr)], IIAlu>;
...
// FrameIndexes are legalized when they are operands from load/store
// instructions. The same not happens for stack address copies, so an
// add op with mem ComplexPattern is used and the stack address copy
// can be matched. It's similar to Sparc LEA_ADDRi
def LEA_ADDiu : EffectiveAddress<"addiu\t$ra, $addr", CPURegs, mem_ea> {
  let isCodeGenOnly = 1;
}
```

LLVMBackendTutorialExampleCode/Chapter4_5/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_5.cpp with code Chapter4_5/ which support pointer to local variable, will get result as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch4_5.cpp

```
118-165-66-82:InputFiles Jonathan$ clang -c ch4_5.cpp -emit-llvm -o ch4_5.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_5.bc -o ch4_5.cpu0.s
118-165-66-82:InputFiles Jonathan$ cat ch4_5.cpu0.s
  .section .mdebug.abi32
  .previous
  .file "ch4_5.bc"
```

```
  .text
  .globl  main
  .align  2
  .type main,@function
  .ent  main                    # @main
main:
  .cfi_startproc
  .frame  $sp,16,$lr
  .mask   0x00000000,0
  .set  noreorder
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -16
$tmp1:
  .cfi_def_cfa_offset 16
  addiu $2, $zero, 0
  st  $2, 12($sp)
  addiu $2, $zero, 3
  st  $2, 8($sp)
  addiu $2, $sp, 8
  st  $2, 0($sp)
  addiu $sp, $sp, 16
  ret $lr
  .set  macro
  .set  reorder
  .end  main
$tmp2:
  .size main, ($tmp2)-main
  .cfi_endproc
```

4.6 Operator mod, %

4.6.1 The DAG of %

Example input code ch4_6.cpp which contains the C operator **"%"** and it's corresponding llvm IR, as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch4_6.cpp

```
; ModuleID = 'ch4_6.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"

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,

---

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 Chapter4_5/ with input file ch4_6.bc and `llc` option –view-isel-dags as below, will get the error message as follows and the llvm DAG of Figure 4.2.

```
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_6.bc -o ch4_6.cpu0.s
...
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 by calculate the value in each node. The 0xC*0x2AAAAAAB=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.

4.6.2 Arm solution

Let's run Chapter4_6_1/ with ch4_6.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 Chapter4_6_1/ with ch4_6.bc.

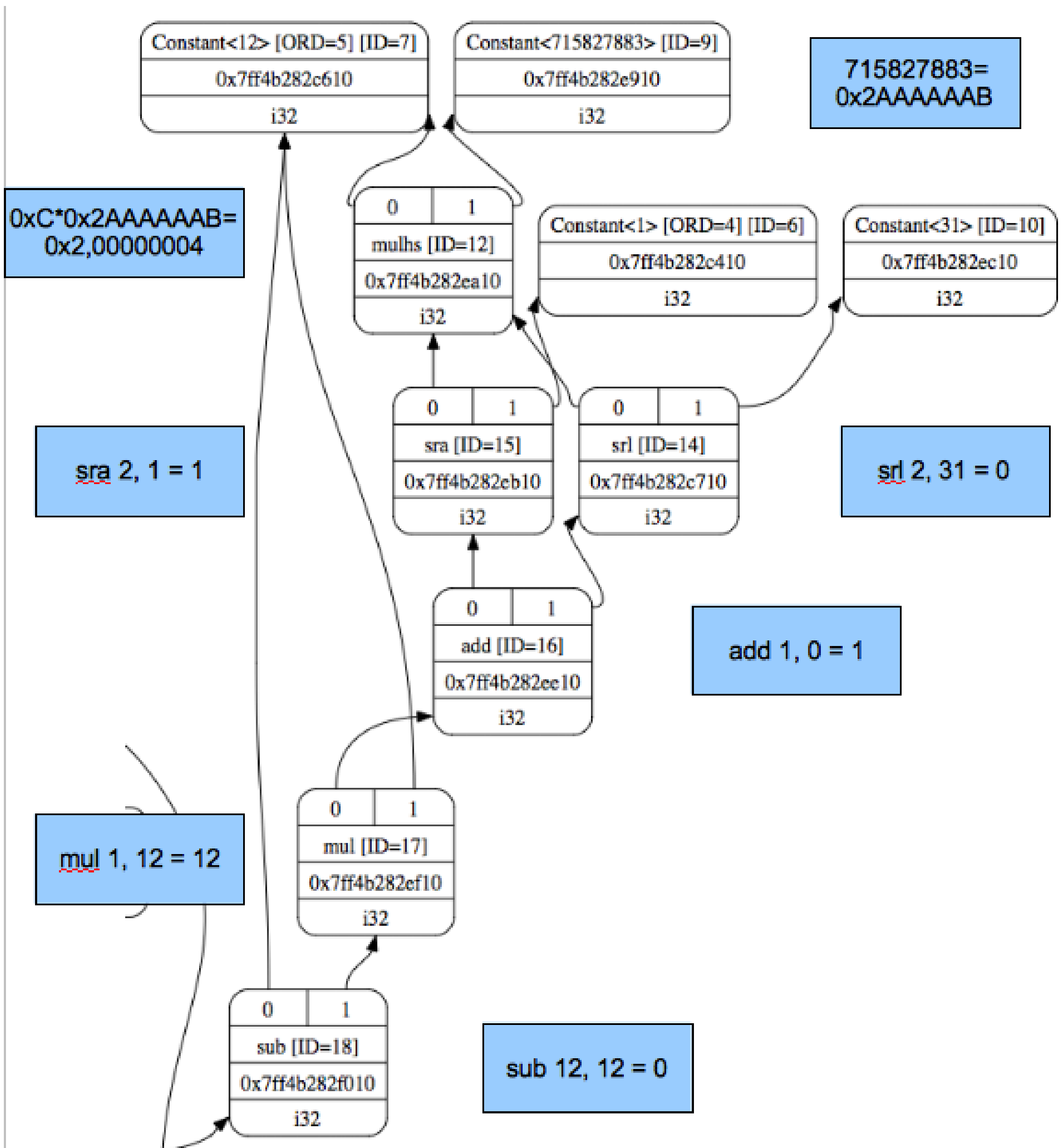


Figure 4.2: ch4_6.bc DAG

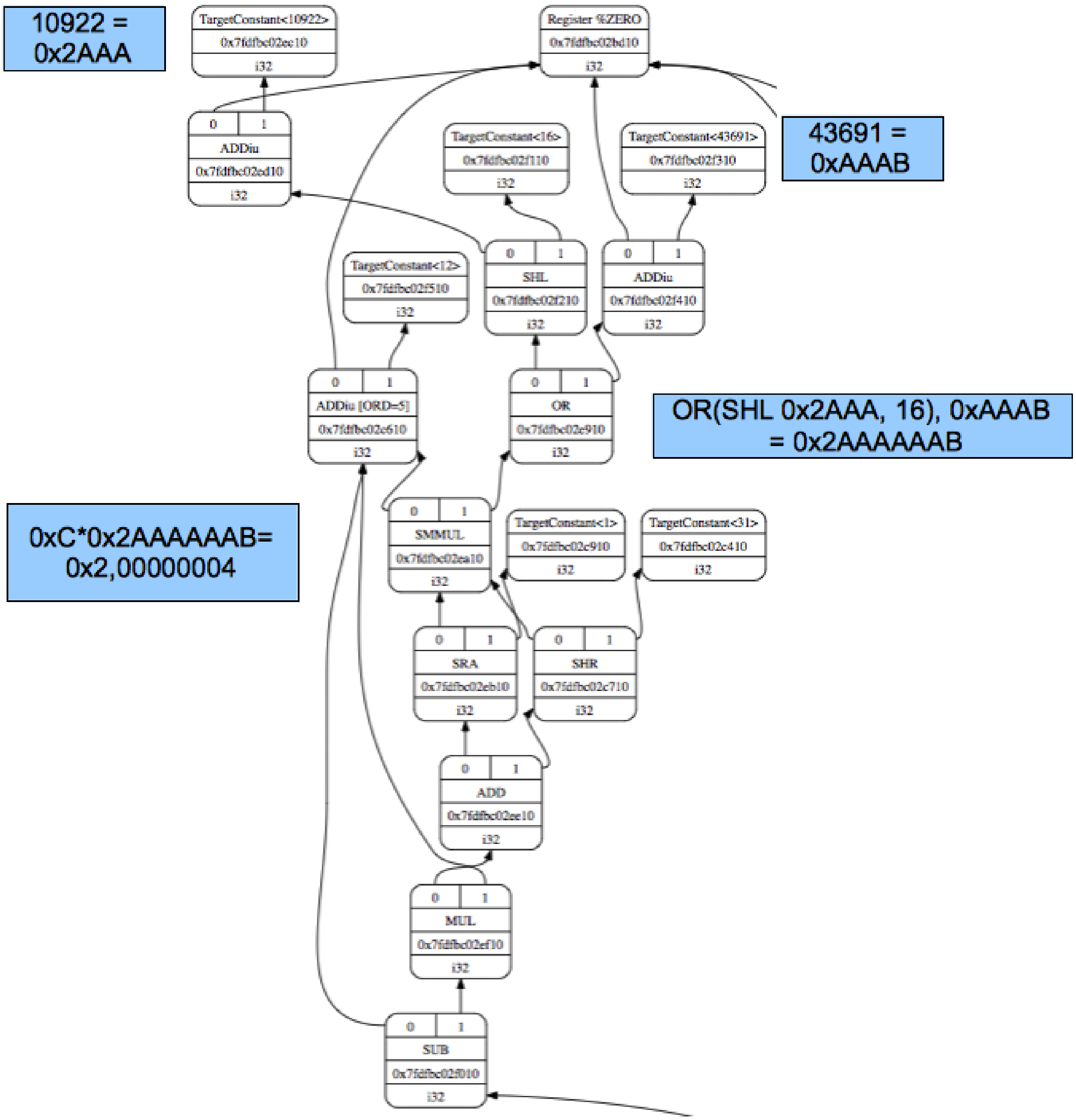


Figure 4.3: Translate ch4_6.bc into cpu0 backend DAG

```
118-165-66-82:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_
debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch4_6.bc -o ch4_6.cpu0.s
118-165-71-252:InputFiles Jonathan$ cat ch4_6.cpu0.s
    .section .mdebug.abi32
    .previous
    .file   "ch4_6.bc"
    .text
    .globl  main
    .align  2
    .type   main,@function
    .ent    main                    # @main
main:
    .frame  $sp,8,$lr
    .mask   0x00000000,0
    .set    noreorder
    .set    nomacro
# BB#0:                                 # %entry
    addiu   $sp, $sp, -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
    addiu   $3, $zero, 43691
    or  $3, $2, $3
    addiu   $2, $zero, 12
    smmul   $3, $2, $3
    shr $4, $3, 31
    sra $3, $3, 1
    add $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
$tmp1:
    .size   main, ($tmp1)-main
```

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_6.cpp.

Use SMMUL instruction to get the high word of multiplication result is adopted in ARM. The Chapter4_6_1/ use the ARM solution. With this solution, the following code is needed.

LLVMBackendTutorialExampleCode/Chapter4_6_1/Cpu0InstrInfo.td

```
// 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);
}]>;
...
def SMMUL   : ArithLogicR<0x50, "smmul", mulhs, IIImul, CPURegs, 1>;
def UMMUL   : ArithLogicR<0x51, "ummul", mulhu, IIImul, CPURegs, 1>;
...
// Arbitrary immediates
def : Pat<(i32 imm:$imm),
      (OR (SHL (ADDiu ZERO, (HI16 imm:$imm)), 16), (ADDiu ZERO, (LO16 imm:$imm)))>;
```

4.6.3 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). Meanwhile Mips is fast if you need both the HI and LO result. If you need the LO part of result, you can use Cpu0 MUL instruction which only get the LO part of result. Chapter4_6_2/ is implemented with Mips MULT style. We choose it as the implementation of this book. 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_6_2/ from TargetSelectionDAG.td.

LLVMBackendTutorialExampleCode/Chapter4_6_2/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_6_2/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;
}
...
def MULT    : Mult32<0x50, "mult", IIImul>;
def MULTu   : Mult32<0x51, "multu", IIImul>;

def MFHI : MoveFromLOHI<0x40, "mfhi", CPURegs, [HI]>;
def MFLO : MoveFromLOHI<0x41, "mflo", CPURegs, [LO]>;
```

LLVMBackendTutorialExampleCode/Chapter4_6_2/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_6.cpu0.s are the result of compile ch4_6.bc.

```
118-165-66-82:InputFiles Jonathan$ cat ch4_6.cpu0.s
  .section .mdebug.abi32
  .previous
  .file "ch4_6.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 $2, $zero, 11
  st  $2, 0($sp)
  addiu $2, $zero, 10922
  shl $2, $2, 16
  addiu $3, $zero, 43691
  or  $3, $2, $3
  addiu $2, $zero, 12
  mult  $2, $3
  mfhi  $3
  shr $4, $3, 31
  sra $3, $3, 1
  add $3, $3, $4
  mul $3, $3, $2
  sub $2, $2, $3
  st  $2, 0($sp)
  addiu $sp, $sp, 8
  ret $lr
  .set  macro
```

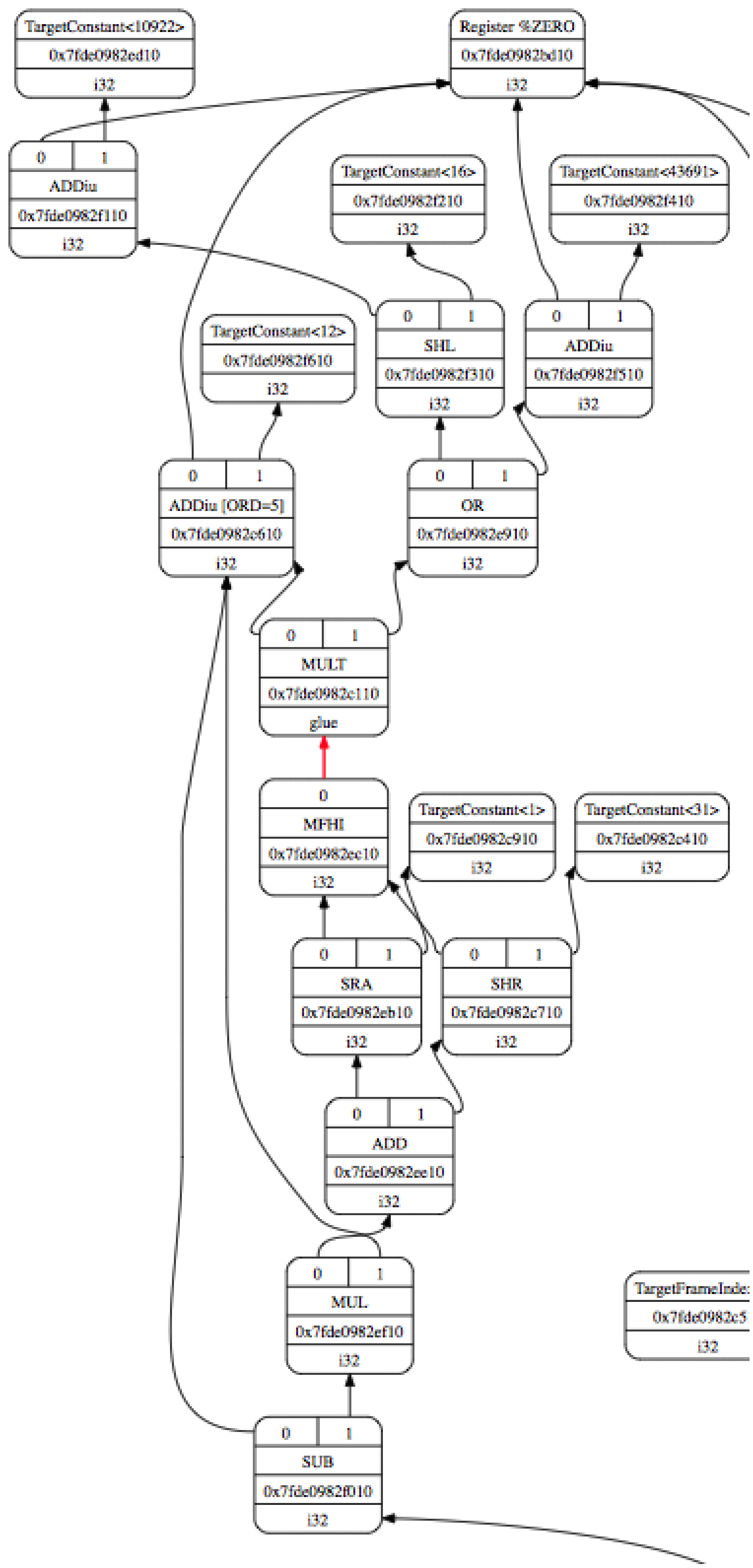


Figure 4.4: DAG for ch4_6.bc with Mips style MULT

```
  .set  reorder
  .end  main
$tmp2:
  .size main, ($tmp2)-main
  .cfi_endproc
```

4.7 Full support %

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 happens in our code. The answer is our code will have 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_6_4/ support operator **"%"** and **"/"**. The code added in Chapter4_6_4/ as follows,

LLVMBackendTutorialExampleCode/Chapter4_6_4/Cpu0InstrInfo.cpp

LLVMBackendTutorialExampleCode/Chapter4_6_4/Cpu0InstrInfo.h

LLVMBackendTutorialExampleCode/Chapter4_6_4/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, 0x16, "div", IIIdiv>;
def UDIV    : Div32<Cpu0DivRemU, 0x17, "divu", IIIdiv>;
...
def MTHI : MoveToLOHI<0x42, "mthi", CPURegs, [HI]>;
def MTLO : MoveToLOHI<0x43, "mtlo", CPURegs, [LO]>;
```

LLVMBackendTutorialExampleCode/Chapter4_6_4/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_6_4/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
  };
  }
...
```

Run with ch4_1_2.cpp can get the result for operator **"/"** as below. But run with ch4_6_1.cpp as below, cannot get the **"div"** for operator **"%"**. It still use **"multiplication"** instead of **"div"** because llvm do **"Constant Propagation Optimization"** in this. The ch4_6_2.cpp can get the **"div"** for **"%"** result since it make the llvm **"Constant Propagation Optimization"** useless in this. Unfortunately, we cannot run it now since it need the function call support. We will verify **"%"** with ch4_6_2.cpp at the end of chapter "Function Call". You can run with the end of Example Code of chapter "Function Call", if you like to verify it now.

LLVMBackendTutorialExampleCode/InputFiles/ch4_1_2.cpp

```
int main()
{
  ...
  f = a / b;
  ...
}
```

```
118-165-77-79:InputFiles Jonathan$ clang -c ch4_1_2.cpp -emit-llvm -o ch4_1_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_1_2.bc -o ch4_1_2.cpu0.s
118-165-77-79:InputFiles Jonathan$ cat ch4_1_2.cpu0.s
  div $zero, $3, $2
  mflo  $2
  ...
```

LLVMBackendTutorialExampleCode/InputFiles/ch4_6_1.cpp

LLVMBackendTutorialExampleCode/InputFiles/ch4_6_2.cpp

4.8 Summary

We support most of C operators in this chapter. Until now, we have around 3400 lines of source code with comments. With these 345 lines of source code added, it support the number of operators from three to over ten.

CHAPTER

FIVE

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_6_2/ to translate obj code will get the error message as follows,

```
[Gamma@localhost 3]$ /usr/local/llvm/test/cmake_debug_build/bin/
llc -march=cpu0 -relocation-model=pic -filetype=obj ch4_1_2.bc -o ch4_1_2.cpu0.o
/usr/local/llvm/test/cmake_debug_build/bin/llc: target does not
support generation of this file type!
```

The Chapter5/ 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_2.cpu0.s
...
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -72
  addiu $2, $zero, 0
  st  $2, 68($sp)
  addiu $3, $zero, 5
  st  $3, 64($sp)
...

[Gamma@localhost 3]$ /usr/local/llvm/test/cmake_debug_build/bin/
llc -march=cpu0 -relocation-model=pic -filetype=obj ch4_2.bc -o ch4_2.cpu0.o
[Gamma@localhost InputFiles]$ objdump -s ch4_2.cpu0.o

ch4_2.cpu0.o:     file format elf32-big

Contents of section .text:
 0000 09d0ffb8 09200000 012d0044 09300005  ..... ...-.D.0..
 0010 013d0040 09300002 013d003c 012d0038  .=.@.0...=.<.-.8
 0020 012d0034 012d0014 0930fffb 013d0010  .-.4.-...0...=..
 0030 012d000c 012d0008 002d003c 003d0040  .-...-...-.<.=.@
 0040 13232000 012d0038 002d003c 003d0040  .# ..-.8.-.<.=.@
```

```
 0050 14232000 012d0034 002d003c 003d0040  .# ..-.4.-.<.=.@
 0060 15232000 012d0030 002d003c 003d0040  .# ..-.0.-.<.=.@
 0070 16232000 012d002c 002d003c 003d0040  .# ..-.,.-.<.=.@
 0080 18232000 012d0028 002d003c 003d0040  .# ..-.(.-.<.=.@
 0090 19232000 012d0024 002d003c 003d0040  .# ..-.$.-.<.=.@
 00a0 1a232000 012d0020 002d0040 1e220002  .# ..-. .-.@."..
 00b0 012d001c 002d0010 1e220002 012d0004  .-...-..."...-..
 00c0 002d0010 1f220002 012d000c 09d00048  .-..."...-.....H
 00d0 2c00000e                             ,...
Contents of section .eh_frame:
 0000 00000010 00000000 017a5200 017c0e01  .........zR..|..
 0010 000c0d00 00000010 00000018 00000000  ................
 0020 000000d4 00440e48                    .....D.H
[Gamma@localhost InputFiles]$ /usr/local/llvm/test/
cmake_debug_build/bin/llc -march=cpu0el -relocation-model=pic -filetype=obj
ch4_2.bc -o ch4_2.cpu0el.o
[Gamma@localhost InputFiles]$ objdump -s ch4_2.cpu0el.o

ch4_2.cpu0el.o:     file format elf32-little

Contents of section .text:
 0000 b8ffd009 00002009 44002d01 05003009  ...... .D.-...0.
 0010 40003d01 02003009 3c003d01 38002d01  @.=...0.<.=.8.-.
 0020 34002d01 14002d01 fbff3009 10003d01  4.-...-...0...=.
 0030 0c002d01 08002d01 3c002d00 40003d00  ..-...-.<.-.@.=.
 0040 00202313 38002d01 3c002d00 40003d00  . #.8.-.<.-.@.=.
 0050 00202314 34002d01 3c002d00 40003d00  . #.4.-.<.-.@.=.
 0060 00202315 30002d01 3c002d00 40003d00  . #.0.-.<.-.@.=.
 0070 00202316 2c002d01 3c002d00 40003d00  . #.,.-.<.-.@.=.
 0080 00202318 28002d01 3c002d00 40003d00  . #.(.-.<.-.@.=.
 0090 00202319 24002d01 3c002d00 40003d00  . #.$.-.<.-.@.=.
 00a0 0020231a 20002d01 40002d00 0200221e  . #. .-.@.-...".
 00b0 1c002d01 10002d00 0200221e 04002d01  ..-...-..."...-.
 00c0 10002d00 0200221f 0c002d01 4800d009  ..-..."...-.H...
 00d0 0e00002c                             ...,
Contents of section .eh_frame:
 0000 10000000 00000000 017a5200 017c0e01  .........zR..|..
 0010 000c0d00 10000000 18000000 00000000  ................
 0020 d4000000 00440e48                    .....D.H
```

The first instruction is **"addiu $sp, -72"** and it's corresponding obj is 0x09d0ffb8. The addiu opcode is 0x09, 8 bits, $sp register number is 13(0xd), 4bits, second register is useless, so assign it to 0x0, and the immediate is 16 bits -72(=0xffb8), so it's correct. The third instruction **"st $2, 68($sp)"** and it's corresponding obj is 0x012d0044. The **st** opcode is **0x0a**, $2 is 0x2, $sp is 0xd and immediate is 68(0x0044). Thanks to cpu0 instruction format which opcode, register operand and offset(imediate value) size are multiple of 4 bits. The obj format is easy to check by eye. The big endian (B0, B1, B2, B3) = (09, d0, ff, b8), objdump from B0 to B3 as 0x09d0ffb8 and the little endian is (B3, B2, B1, B0) = (09, d0, ff, b8), objdump from B0 to B3 as 0xb8ffd009.

5.2 Backend Target Registration Structure

Now, let's examine Cpu0MCTargetDesc.cpp.

LLVMBackendTutorialExampleCode/Chapter5_1/MCTargetDesc/Cpu0MCTargetDesc.cpp

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.

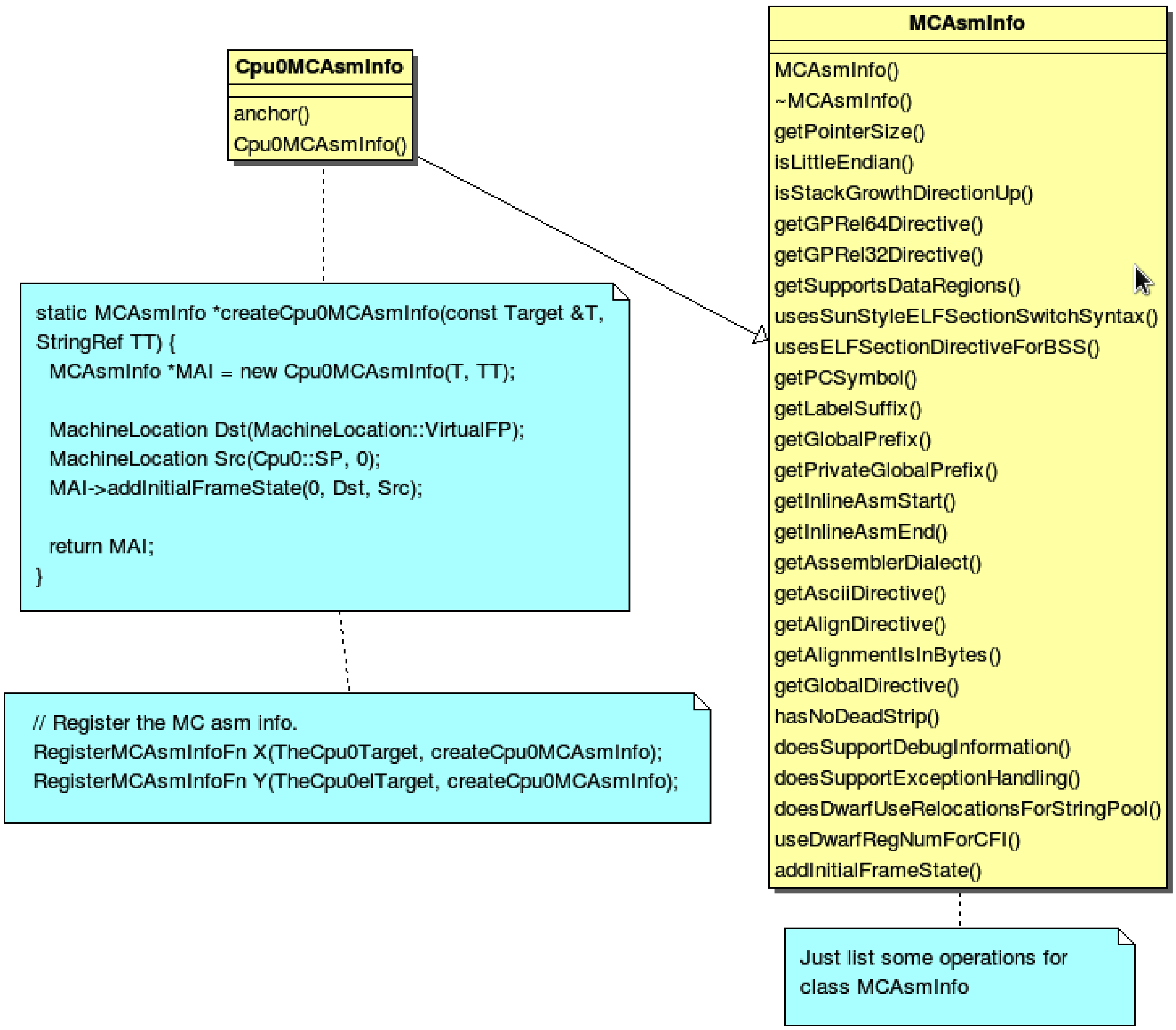


Figure 5.1: Register Cpu0MCAsmInfo

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 -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

[1] http://jonathan2251.github.com/lbd/llvmstructure.html#target-registration

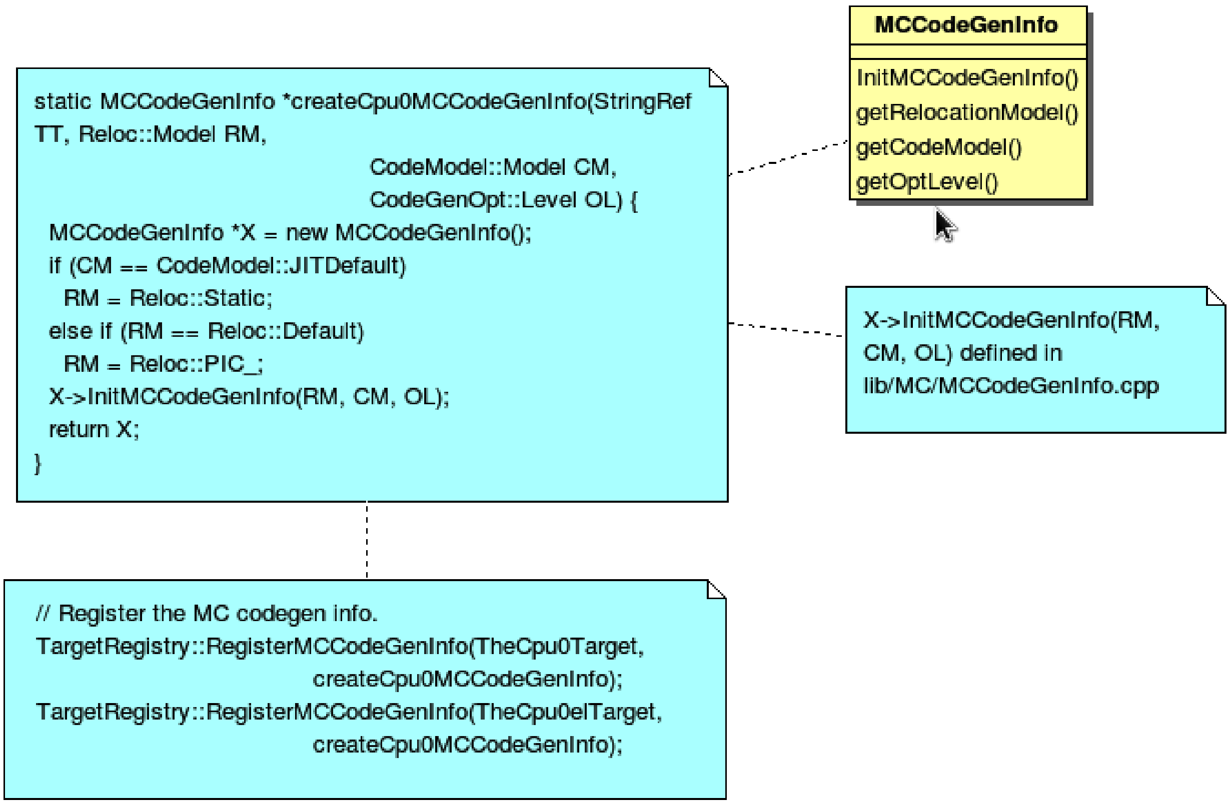


Figure 5.2: Register MCCodeGenInfo

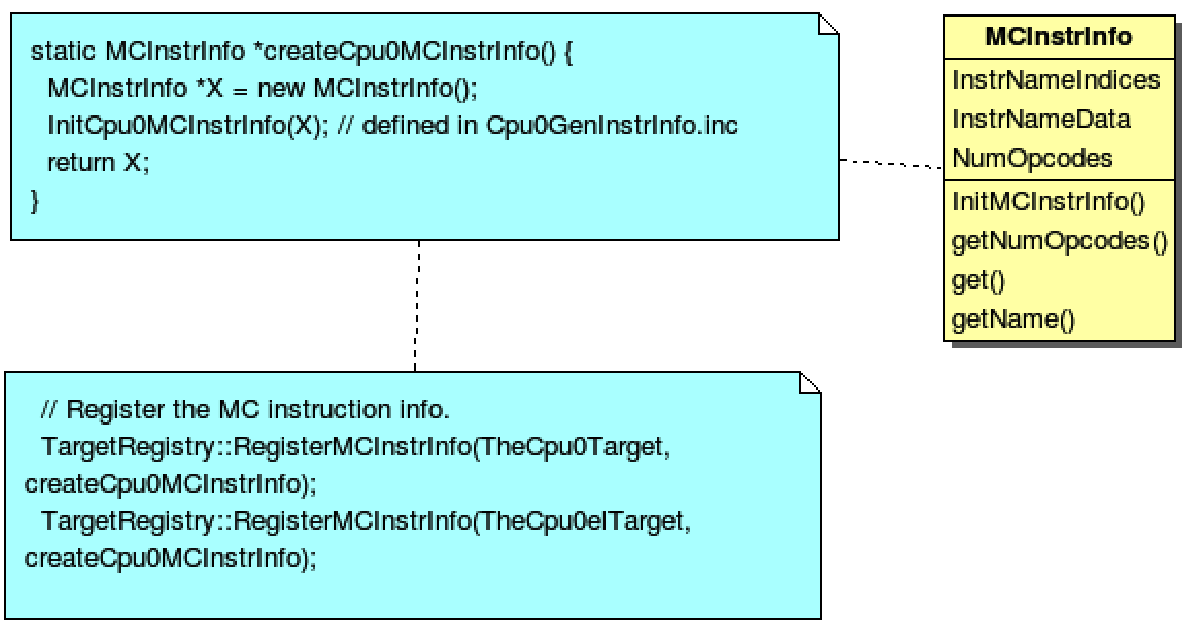


Figure 5.3: Register MCInstrInfo

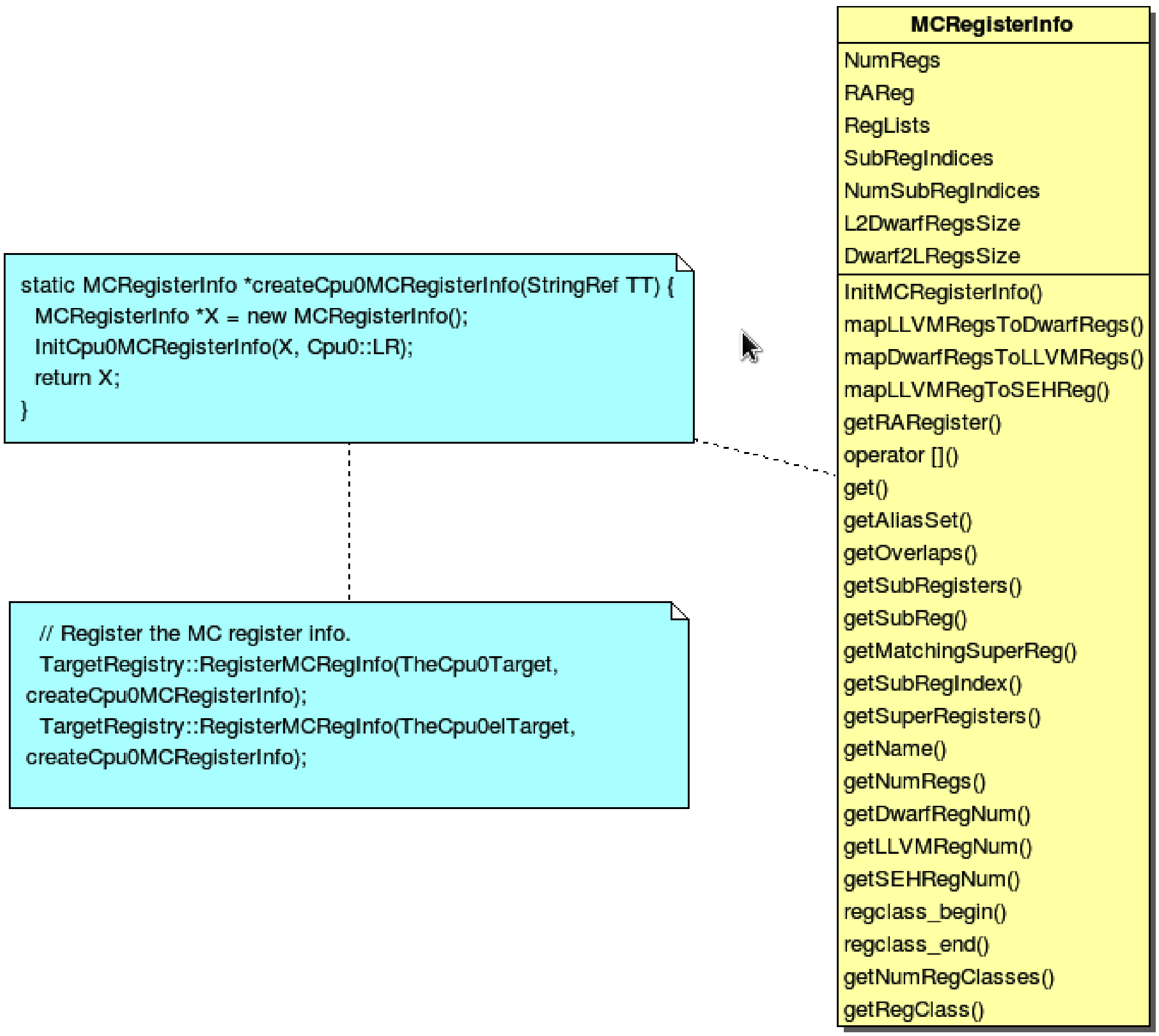


Figure 5.4: Register MCRegisterInfo

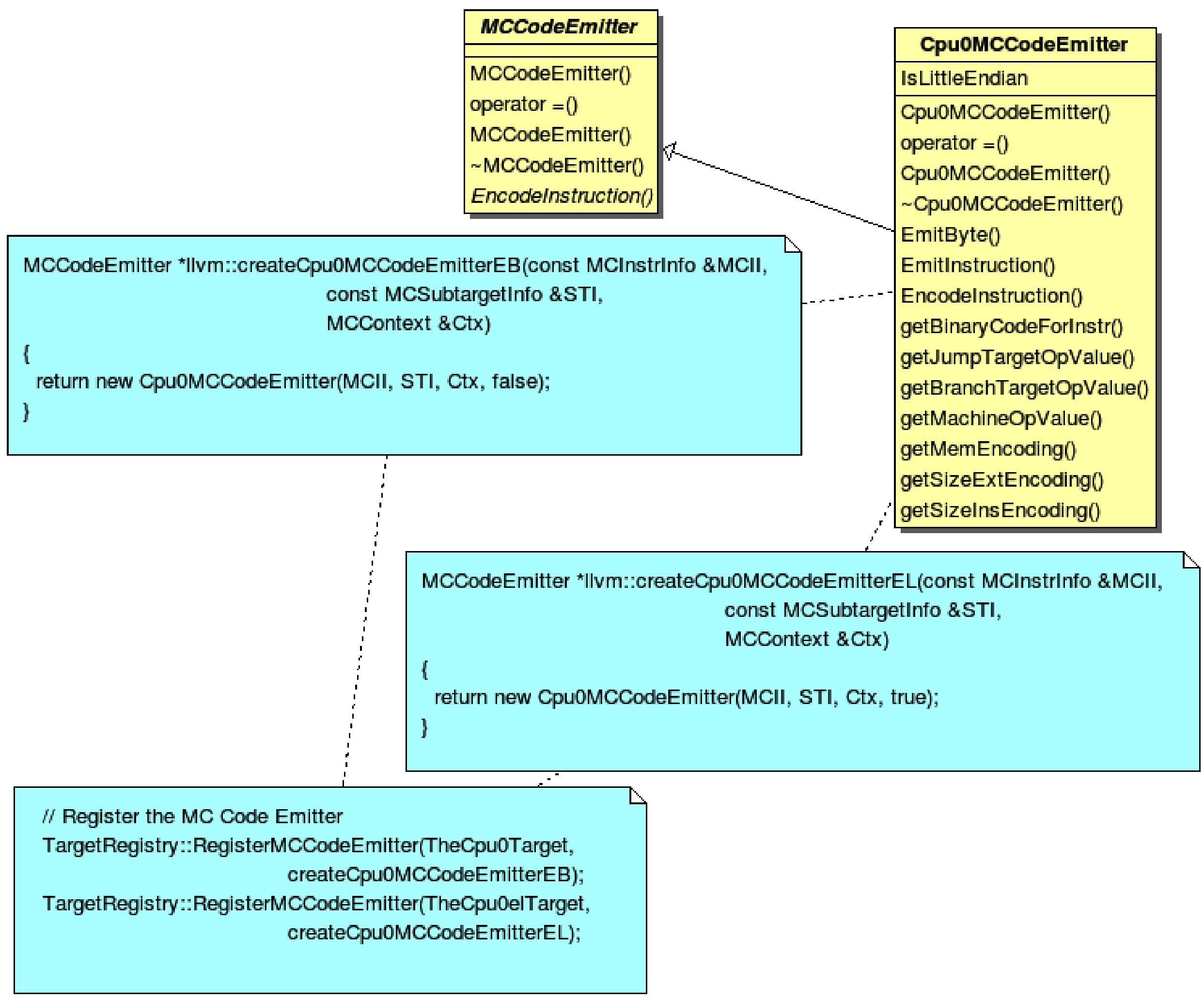


Figure 5.5: Register Cpu0MCCodeEmitter

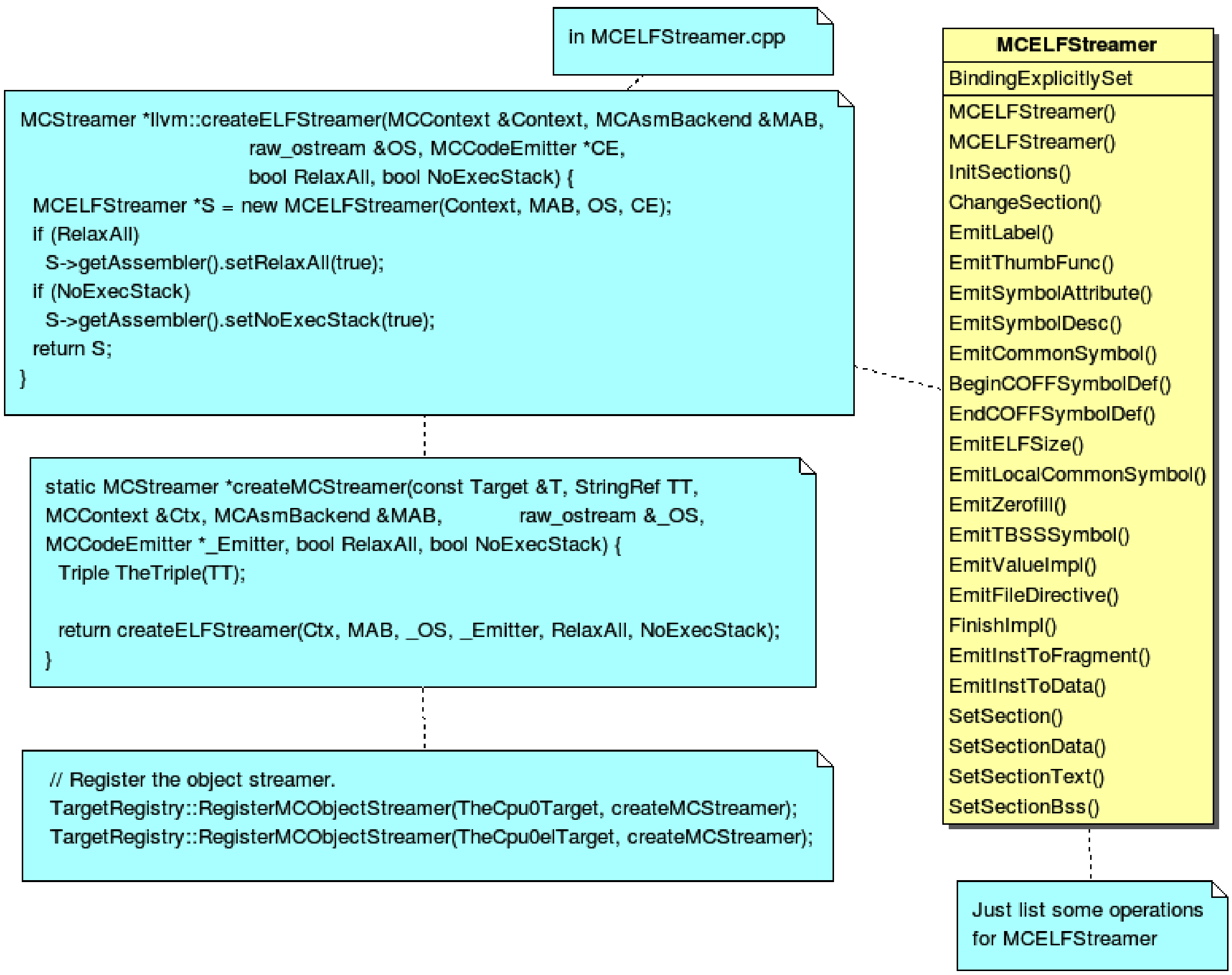


Figure 5.6: Register MCELFStreamer

Figure 5.7: Register Cpu0AsmBackend

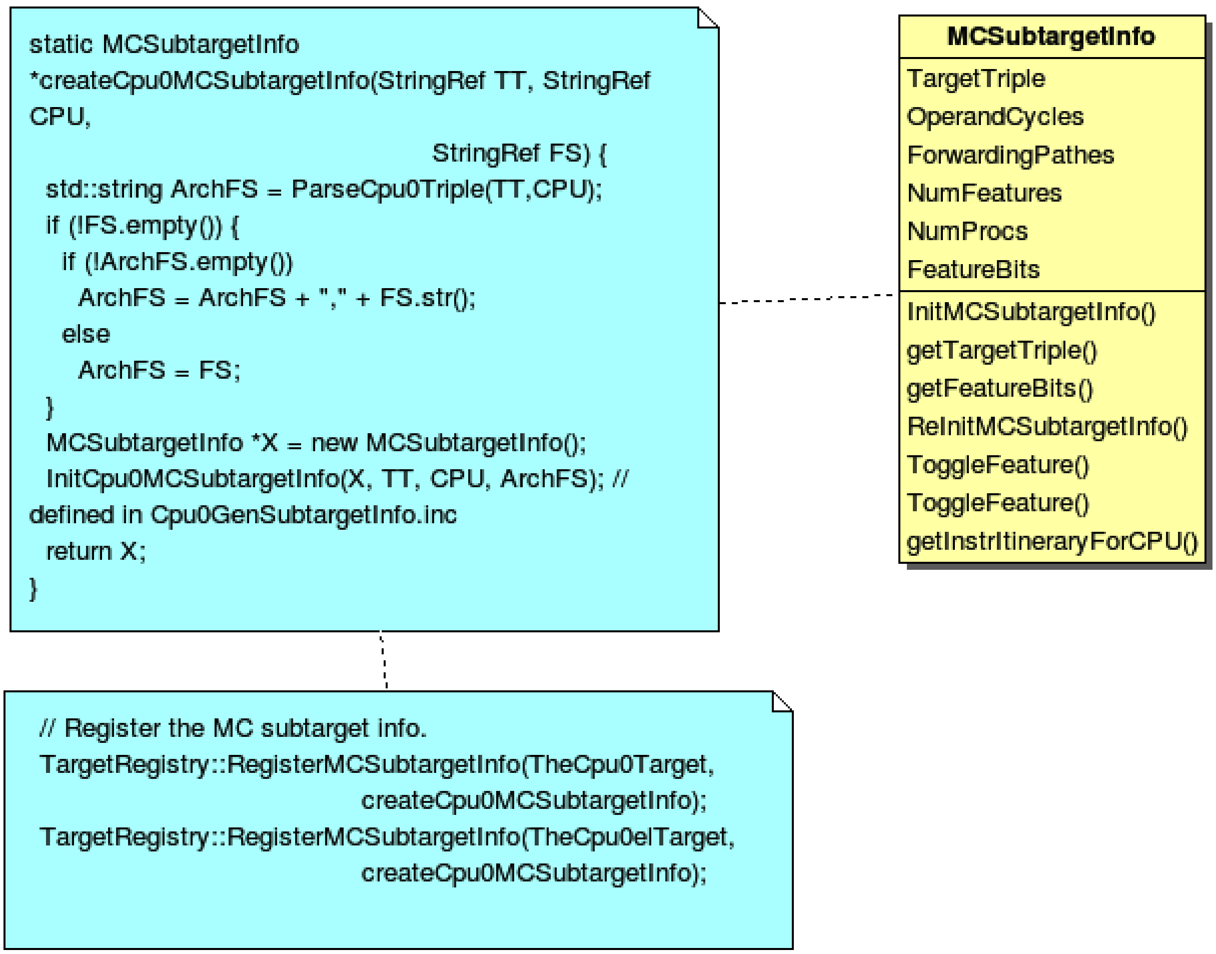


Figure 5.8: Register Cpu0MCSubtargetInfo

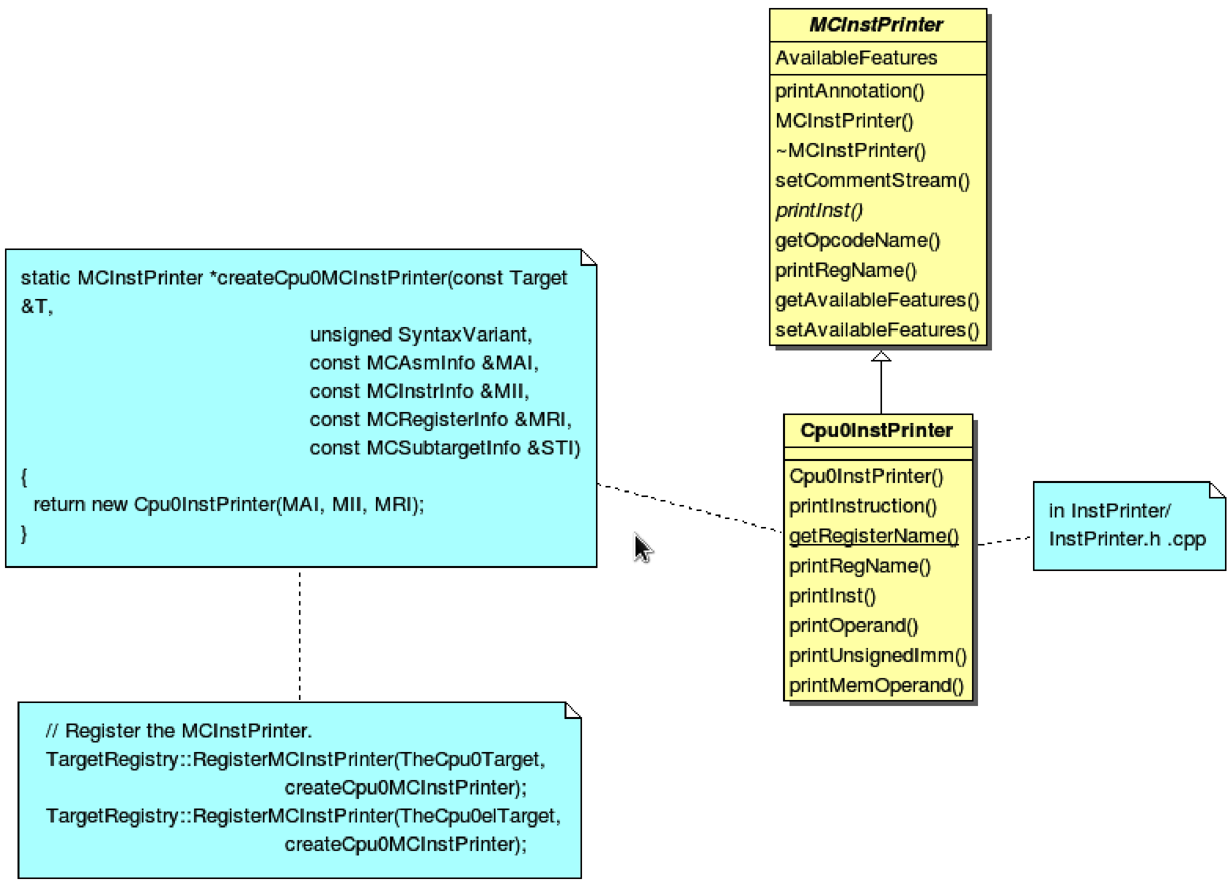


Figure 5.9: Register Cpu0InstPrinter

Figure 5.10: MCELFStreamer inherit tree

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.

CHAPTER

SIX

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 `llc -relocation-model` option while the others of DAG just do IR DAG to Machine DAG translation directly according the input file IR DAG.

6.1 Global variable

Chapter6_1/ support the global variable, let's compile ch6_1.cpp with this version first, and explain the code changes after that.

LLVMBackendTutorialExampleCode/InputFiles/ch6_1.cpp

```
118-165-66-82:InputFiles Jonathan$ llvm-dis ch6_1.bc -o ch6_1.ll
118-165-66-82:InputFiles Jonathan$ cat ch6_1.ll
; 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"

@gI = global i32 100, align 4

define i32 @main() 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
}
```

```
118-165-66-82:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_
debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch6_1.bc -o ch6_1.cpu0.s
118-165-66-82:InputFiles Jonathan$ 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
  .cpload $t9
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -8
$tmp1:
  .cfi_def_cfa_offset 8
  addiu $2, $zero, 0
  st  $2, 4($sp)
  st  $2, 0($sp)
  ld  $2, %got(gI)($gp)
  ld  $2, 0($2)
  st  $2, 0($sp)
  addiu $sp, $sp, 8
  ret $lr
  .set  macro
  .set  reorder
  .end  main
$tmp2:
  .size main, ($tmp2)-main
  .cfi_endproc

  .type gI,@object              # @gI
  .data
  .globl  gI
  .align  2
gI:
  .4byte  100                     # 0x64
  .size gI, 4
```

As above code, it translate **"load i32* @gI, align 4"** into **"ld $2, %got(gI)($gp)"** for `llc -march=cpu0 -relocation-model=pic`, position-independent mode. More specifically, it translate the global integer variable gI address into offset of register gp and load from $gp+(the offset) into register $2.

6.1.1 Static mode

We can also translate it with absolute address mode by following command,

```
118-165-66-82:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_
debug_build/bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=asm
ch6_1.bc -o ch6_1.cpu0.static.s
```

```
118-165-66-82:InputFiles Jonathan$ cat ch6_1.cpu0.static.s
  ...
  addiu $2, $zero, %hi(gI)
  shl $2, $2, 16
  addiu $2, $2, %lo(gI)
  ld  $2, 0($2)
```

Above code, it loads the high address part of gI absolute address (16 bits) to register $2 and shift 16 bits. Now, the register $2 got it's high part of gI absolute address. Next, it loads the low part of gI absolute address into register 3. Finally, add register $2 and $3 into $2, and loads the content of address $2+offset 0 into register $2. The `llc -relocation-model=static` is for static link mode which binding the address in static, compile/link time, not dynamic/run time. In this mode, you can also translate code with the following command,

```
118-165-66-82:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_
debug_build/bin/Debug/llc -march=cpu0 -relocation-model=static -cpu0-islinux-f
ormat=false -filetype=asm ch6_1.bc -o ch6_1.cpu0.islinux-format-false.s
118-165-66-82:InputFiles Jonathan$ cat ch6_1.cpu0.islinux-format-false.s
  ...
  st  $2, 0($sp)
  addiu $2, $gp, %gp_rel(gI)
  ld  $2, 0($2)
  ...
  .section  .sdata,"aw",@progbits
  .globl  gI
```

As above, it translate code with `llc -relocation-model=static -cpu0-islinux-format=false`. The -cpu0-islinux-format default is true which will allocate global variables in data section. With setting false, it will allocate global variables in sdata section. Section data and sdata are areas for global variable with initial value, int gI = 100 in this example. Section bss and sbss are areas for global variables without initial value (for example, int gI;). Allocate variables in sdata or sbss sections is addressable by 16 bits + $gp. The static mode with -cpu0-islinux-format=false is still static mode (variable is binding in compile/link time) even it's use $gp relative address. The $gp content is assigned at compile/link time, changed only at program be loaded, and is fixed during running the program; while the -relocation-model=pic the $gp can be changed during program running. For example, if $gp is assigned to start of .sdata like this example, then %gp_rel(gI) = (the relative address distance between gI and $gp) (is 0 in this case). When sdata is loaded into address x, then the gI variable can be got from address x+0 where x is the address stored in $gp, 0 is the value of $gp_rel(gI).

To support global variable, first add **IsLinuxOpt** command variable to Cpu0Subtarget.cpp. After that, user can run llc with argument `llc -cpu0-islinux-format=false` to specify **IsLinuxOpt** to false. The **IsLinuxOpt** is defaulted to true if without specify it. About the **cl** command variable, you can refer to [1] further.

LLVMBackendTutorialExampleCode/Chapter6_1/Cpu0Subtarget.cpp

```
static cl::opt<bool>
IsLinuxOpt("cpu0-islinux-format", cl::Hidden, cl::init(true),
                 cl::desc("Always use linux format."));
```

Next add the following code to Cpu0ISelLowering.cpp.

LLVMBackendTutorialExampleCode/Chapter6_1/Cpu0ISelLowering.cpp

```
//  Cpu0ISelLowering.cpp
Cpu0TargetLowering::
```

[1] http://llvm.org/docs/CommandLine.html

```
Cpu0TargetLowering(Cpu0TargetMachine &TM)
  : TargetLowering(TM, new Cpu0TargetObjectFile()),
    Subtarget(&TM.getSubtarget<Cpu0Subtarget>()) {
   ...
  // Cpu0 Custom Operations
  setOperationAction(ISD::GlobalAddress,      MVT::i32,   Custom);
  ...
}
...
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
  switch (Op.getOpcode())
  {
    case ISD::GlobalAddress:      return LowerGlobalAddress(Op, DAG);
  }
  return SDValue();
}

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

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

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

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

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

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

  EVT ValTy = Op.getValueType();
  bool HasGotOfst = (GV->hasInternalLinkage() ||
```

```
                     (GV->hasLocalLinkage() && !isa<Function>(GV)));
  unsigned GotFlag = (HasGotOfst ? Cpu0II::MO_GOT : Cpu0II::MO_GOT16);
  SDValue GA = DAG.getTargetGlobalAddress(GV, dl, ValTy, 0, GotFlag);
  GA = DAG.getNode(Cpu0ISD::Wrapper, dl, ValTy, GetGlobalReg(DAG, ValTy), GA);
  SDValue ResNode = DAG.getLoad(ValTy, dl, DAG.getEntryNode(), GA,
                                MachinePointerInfo(), false, false, false, 0);
  // On functions and global targets not internal linked only
  // a load from got/GP is necessary for PIC to work.
  if (!HasGotOfst)
    return ResNode;
  SDValue GALo = DAG.getTargetGlobalAddress(GV, dl, ValTy, 0,
                                                        Cpu0II::MO_ABS_LO);
  SDValue Lo = DAG.getNode(Cpu0ISD::Lo, dl, ValTy, GALo);
  return DAG.getNode(ISD::ADD, dl, ValTy, ResNode, Lo);
}
```

The setOperationAction(ISD::GlobalAddress, MVT::i32, Custom) tells `llc` that we implement global address operation in C++ function Cpu0TargetLowering::LowerOperation() and llvm will call this function only when llvm want to translate IR DAG of loading global variable into machine code. Since may have many Custom type of setOperationAction(ISD::XXX, MVT::XXX, Custom) in construction function Cpu0TargetLowering(), and llvm will call Cpu0TargetLowering::LowerOperation() for each ISD IR DAG node of Custom type translation. The global address access can be identified by check the DAG node of opcode is ISD::GlobalAddress. For static mode, LowerGlobalAddress() will check the translation is for IsGlobalInSmallSection() or not. When IsLinuxOpt is true and static mode, IsGlobalInSmallSection() always return false. LowerGlobalAddress() will translate global variable by create 2 DAG IR nodes ABS_HI and ABS_LO for high part and low part of address and one extra node ADD. List it again as follows.

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);
```

The DAG list form for these three DAG nodes as above code created can be represented as (ADD (Hi(h1, h2), Lo (l1, l2)). Since some DAG nodes are not with two arguments, we will define the list as (ADD (Hi (...), Lo (...)) or (ADD (Hi, Lo)) sometimes in this book. The corresponding machine instructions of these three IR nodes are defined in Cpu0InstrInfo.td as follows,

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)>;
```

Above code meaning translate ABS_HI into ADDiu and SHL two instructions. Remember the DAG and Instruction Selection introduced in chapter "Back end structure", DAG list (SHL (ADDiu ...), 16) meaning DAG node ADDiu and it's parent DAG node SHL two instructions nodes is for list IR DAG ABS_HI. The Pat<> has two list DAG representation. The left is IR DAG and the right is machine instruction DAG. So after Instruction Selection and Register Allocation, it translate ABS_HI to,

```
addiu $2, %hi(gI)
shl $2, $2, 16
```

According above code, we know llvm allocate register $2 for the output operand of ADDiu instruction and $2 for SHL instruction in this example. Since (SHL (ADDiu), 16), the ADDiu output result will be the SHL first register. The result is **"shl $2, 16"**. Above Pat<> also define DAG list (add $hi, (ABS_LO)) will be translated into (ADD $hi, (ADDiu ZERO, ...)) where ADD is machine instruction **add** and ADDiu is machine instruction **ldi** which defined in Cpu0InstrInfo.td too. Remember (add $hi, (ABS_LO)) meaning add DAG has two operands, the first is $hi and the second is the register which the ABS_LO output result register save to. So, the IR DAG pattern and it's corresponding machine instruction node as follows,

```
addiu $3, %lo(gI)  // def : Pat<(Cpu0Lo tglobaladdr:$in), (ADDiu ZERO,
                   // tglobaladdr:$in)>;

// def : Pat<(add CPURegs:$hi, (Cpu0Lo tglobaladdr:$lo)), (ADD CPURegs:$hi,
//  (LDI ZERO, tglobaladdr:$lo))>;
// So, the second register for add is the output register of ABS_LO IR DAG
//  translation result saved to;
// Since LowerGlobalAddress() create list (ADD (Hi, Lo)) with 3 DAG nodes,
//  the Hi output register $2 will be the first input register for add.
   add $2, $2, $3
```

After translated as above, the register $2 is the global variable address, so get the global variable by IR DAG load will translate into machine instruction as follows,

```
%2 = load i32* @gI, align 4
=>  ld  $2, 0($2)
```

When IsLinuxOpt is false and static mode, LowerGlobalAddress() will run the following code to create a DAG list (ADD GOT, GPRel).

LLVMBackendTutorialExampleCode/Chapter6_1/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);
}
```

As mentioned just before, all global variables allocated in sdata or sbss sections which is addressable by 16 bits + $gp in compile/link time (address binding in compile time). It's equal to offset+GOT where GOT is the base address for global variable and offset is 16 bits. Now, according the following Cpu0InstrInfo.td definition,

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))>;
```

It translate global variable address of list (ADD GOT, GPRel) into machine instructions as follows,

```
addiu $2, $gp, %gp_rel(gI)
```

6.1.2 PIC mode

When PIC mode, LowerGlobalAddress() will create the DAG list (load DAG.getEntryNode(), (Wrapper GetGlobalReg(), GA)) by the following code and the code in Cpu0ISeleDAGToDAG.cpp as follows,

LLVMBackendTutorialExampleCode/Chapter6_1/Cpu0ISeleDAGToDAG.cpp

```
  ...
  bool HasGotOfst = (GV->hasInternalLinkage() ||
                     (GV->hasLocalLinkage() && !isa<Function>(GV)));
  unsigned GotFlag = (HasGotOfst ? Cpu0II::MO_GOT : Cpu0II::MO_GOT16);
  SDValue GA = DAG.getTargetGlobalAddress(GV, dl, ValTy, 0, GotFlag);
  GA = DAG.getNode(Cpu0ISD::Wrapper, dl, ValTy, GetGlobalReg(DAG, ValTy), GA);
  SDValue ResNode = DAG.getLoad(ValTy, dl, DAG.getEntryNode(), GA,
                                MachinePointerInfo(), false, false, false, 0);
  // On functions and global targets not internal linked only
  // a load from got/GP is necessary for PIC to work.
  if (!HasGotOfst)
    return ResNode;
  ...

// 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;
  }
```

```
  ...
}
```

Then it translate into the following code,

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

Where DAG.getEntryNode() is the register $2 which decided by Register Allocator ; DAG.getNode(Cpu0ISD::Wrapper, dl, ValTy, GetGlobalReg(DAG, ValTy), GA) is translated into Base=$gp as well as the 16 bits Offset for $gp.

Apart from above code, add the following code to Cpu0AsmPrinter.cpp and it will emit .cpload asm pseudo instruction,

LLVMBackendTutorialExampleCode/Chapter6_1/Cpu0AsmPrinter.cpp

```
/// EmitFunctionBodyStart - Targets can override this to emit stuff before
/// the first basic block in the function.
void Cpu0AsmPrinter::EmitFunctionBodyStart() {
...
    // Emit .cpload directive if needed.
    if (EmitCPLoad)
    //- .cpload $t9
      OutStreamer.EmitRawText(StringRef("\t.cpload\t$t9"));
...
}

// ch6_1.cpu0.s
    .cpload $t9
    .set    nomacro
# BB#0:
    ldi $sp, -8
```

According Mips Application Binary Interface (ABI), $t9 ($25) is the register used in jalr $25 for long distance function pointer (far subroutine call). 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 is 32 bits. One example of PIC mode is used in share library. Share library is re-entry code which can be loaded in different memory address decided on run time. The static mode (absolute address mode) is usually designed to load in specific memory address decided on compile time. Since share library can be loaded in different memory address, the global variable address cannot be decided in compile time. As above, the global variable address is translated into the relative address of $gp. In example code ch6_1.ll, .cpload is a asm pseudo instruction just before the first instruction of main(), ldi. When the share library main() function be loaded, the loader will assign the $t9 value to $gp when it meet ".cpload $t9". After that, the $gp value is $9 which point to main(), and the global variable address is the relative address to main().

6.1.3 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-islinux-format=false (global var in .sdata)
  case Cpu0II::MO_GPREL:     Kind = MCSymbolRefExpr::VK_Cpu0_GPREL; break;

  case Cpu0II::MO_GOT16:     Kind = MCSymbolRefExpr::VK_Cpu0_GOT16; break;
  case Cpu0II::MO_GOT:       Kind = MCSymbolRefExpr::VK_Cpu0_GOT; break;
// ABS_HI and ABS_LO is for 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.

6.1.4 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 deal 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 corresponding 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 usually no need to coding. The section "Instruction Selector" of [2] is the references.

6.2 Array and struct support

LLVM use getelementptr to represent the array and struct type in C. Please reference section getelementptr of [3]. For ch6_2.cpp, the llvm IR as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch6_2.cpp

```
// 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 {
```

[2] http://llvm.org/docs/WritingAnLLVMBackend.html
[3] http://llvm.org/docs/LangRef.html

```
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>
  0x7f7f5b02de10: i32,ch = load 0x7f7f5b02d910, 0x7f7f5b030010, 0x7f7f5b02d210
  <LD4[getelementptr inbounds ([3 x i32]* @a, i32 0, i32 1)]> [ORD=6]
```

...

By `llc -debug`, you can see the DAG translation process. As above, the DAG list for date.day (add GlobalAddress<[3 x i32]* @a> 0, Constant<8>) with 3 nodes is replaced by 1 node GlobalAddress<%struct.Date* @date> + 8. The DAG list for a[1] is same. The replacement occurs since TargetLowering.cpp::isOffsetFoldingLegal(...) return true in `llc -static` static addressing mode as below. In Cpu0 the **ld** instruction format is **"ld $r1, offset($r2)"** which meaning load $r2 address+offset to $r1. So, we just replace the isOffsetFoldingLegal(...) function by override mechanism as below.

lib/CodeGen/SelectionDAG/TargetLowering.cpp

```
bool
TargetLowering::isOffsetFoldingLegal(const GlobalAddressSDNode *GA) const {
  // Assume that everything is safe in static mode.
  if (getTargetMachine().getRelocationModel() == Reloc::Static)
    return true;

  // In dynamic-no-pic mode, assume that known defined values are safe.
  if (getTargetMachine().getRelocationModel() == Reloc::DynamicNoPIC &&
     GA &&
     !GA->getGlobal()->isDeclaration() &&
     !GA->getGlobal()->isWeakForLinker())
  return true;

  // Otherwise assume nothing is safe.
  return false;
}
```

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 and short int

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

```
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
      .cpload $t9
      .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
```

CHAPTER

SEVEN

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

```
; ModuleID = 'ch7_1_1.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"

define i32 @main() nounwind ssp {
entry:
  %retval = alloca i32, align 4
  %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
  store i32 0, i32* %retval
  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
  %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* %a, align 4
  %inc35 = add i32 %21, 1
  store i32 %inc35, i32* %a, align 4
  br label %if.end36

if.end36:                                         ; preds = %if.then34, %if.end32
  %22 = load i32* %a, align 4
  ret i32 %22
}
```

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 (brcond (%1, setcc(%2, Constant<c>, setne)), BasicBlock_02), BasicBlock_01). Ignore %1, we get the form (br (brcond (setcc(%2, Constant<c>, setne)), BasicBlock_02), BasicBlock_01). For explanation, We list the IR

DAG as follows,

```
%cond=setcc(%2, Constant<c>, setne)
brcond %cond, BasicBlock_02
br BasicBlock_01
```

We want to translate them into cpu0 instructions DAG as follows,

```
addiu %3, ZERO, Constant<c>
cmp %2, %3
jne BasicBlock_02
jmp BasicBlock_01
```

For the first addiu instruction as above which move Constant<c> into register, we have defined it before by the following code,

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

Although the following definition in Cpu0RegisterInfo.td has no real effect in Reserved Registers, you should comment the Reserved Registers in it for readability. Setting SW into another register class to prevent the SW register allocated to the register used by other instruction. The copyPhysReg() is called when DestReg and SrcReg belong to different Register Class. 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)>;
...
```

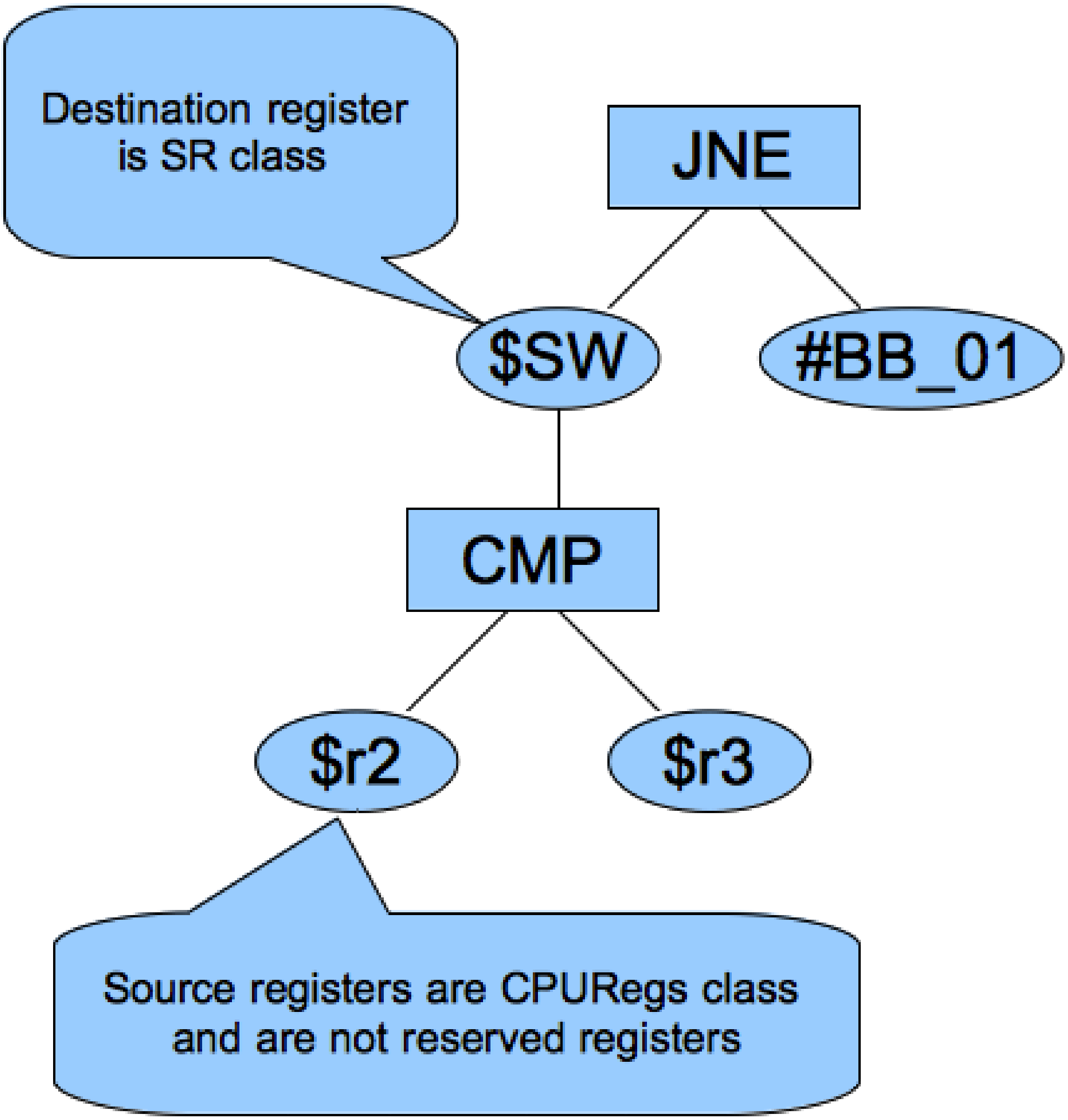


Figure 7.1: JNE (CMP $r2, $r3),

```
// 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  $3, 32($sp)
    cmp $3, $2
    jne $BB0_2
    jmp $BB0_1
$BB0_1:                                 # %if.then
    ld  $2, 32($sp)
    addiu   $2, $2, 1
    st  $2, 32($sp)
$BB0_2:                                 # %if.end
    ld  $2, 28($sp)
...

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 20 20 02 21 00 00 10

0000090 26 00 00 00 ..........................................
```

The immediate value of jne (op 0x21) 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  $2, 32($sp)
    addiu   $2, $2, 1
    st  $2, 32($sp)
$BB0_2:                                 # %if.end
    ld  $2, 28($sp)
```

If cpu0 do **"jne"** compare in execution stage, then we should set PC=PC+12, offset of ($BB0_2, jn e $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

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;
  ...
}

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, $rc
    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<0x20, "jeq", CPURegs>;
def JNE     : CBranch<0x21, "jne", CPURegs>;
def JLT     : CBranch<0x22, "jlt", CPURegs>;
def JGT     : CBranch<0x23, "jgt", CPURegs>;
def JLE     : CBranch<0x24, "jle", CPURegs>;
def JGE     : CBranch<0x25, "jge", CPURegs>;
```

```
def JMP     : UncondBranch<0x26, "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:$dst),
          (JLTOp (CMPOp RC:$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

```
; 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"

@_ZZ4mainE1a = 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]*
    @_ZZ4mainE1a 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
    .cpload $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  $3, %got($_ZZ4mainE1a)($gp)
    addiu   $3, $3, %lo($_ZZ4mainE1a)
    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   $_ZZ4mainE1a,@object    # @_ZZ4mainE1a
    .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

```
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:                         // a == 0
```

```
  ld  $3, 4($sp)
  cmp $3, $2
  jeq $BB0_3          // b == 2
  jmp $BB0_2
$BB0_2:
  ld  $3, 0($sp)
  cmp $3, $2          // c == 2
  jeq $BB0_4
  jmp $BB0_3
$BB0_3:                        // (a == 0 && b == 2) || (c != 2)
  ld  $2, 8($sp)
  addiu $2, $2, 1     // a++
  st  $2, 8($sp)
$BB0_4:
  addiu $sp, $sp, 16
  ret $lr
  .set  macro
  .set  reorder
  .end  main
$tmp2:
  .size main, ($tmp2)-main
  .cfi_endproc
```

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.

CHAPTER

EIGHT

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 upport 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

```
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
```

[1] http://jonathan2251.github.com/lbd/ctrlflow.html#risc-cpu-knowledge

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

Figure 8.1: Mips stack frame

```
  .align  2
  .type _Z5sum_iiiiii,@function
  .set  nomips16                # @_Z5sum_iiiiii
  .ent  _Z5sum_iiiiii
_Z5sum_iiiiii:
  .cfi_startproc
  .frame  $sp,32,$ra
  .mask   0x00000000,0
  .fmask  0x00000000,0
  .set  noreorder
  .set  nomacro
  .set  noat
# BB#0:
  addiu $sp, $sp, -32
$tmp1:
  .cfi_def_cfa_offset 32
  sw  $4, 28($sp)
  sw  $5, 24($sp)
  sw  $6, 20($sp)
  sw  $7, 16($sp)
  lw  $1, 48($sp) // load argument 5
  sw  $1, 12($sp)
  lw  $1, 52($sp) // load argument 6
  sw  $1, 8($sp)
  lw  $2, 24($sp)
  lw  $3, 28($sp)
  addu  $2, $3, $2
  lw  $3, 20($sp)
  addu  $2, $2, $3
  lw  $3, 16($sp)
  addu  $2, $2, $3
  lw  $3, 12($sp)
  addu  $2, $2, $3
  addu  $2, $2, $1
  sw  $2, 4($sp)
  jr  $ra
  addiu $sp, $sp, 32
  .set  at
  .set  macro
  .set  reorder
  .end  _Z5sum_iiiiii
$tmp2:
  .size _Z5sum_iiiiii, ($tmp2)-_Z5sum_iiiiii
  .cfi_endproc

  .globl  main
  .align  2
  .type main,@function
  .set  nomips16                # @main
  .ent  main
main:
  .cfi_startproc
  .frame  $sp,40,$ra
  .mask   0x80000000,-4
  .fmask  0x00000000,0
  .set  noreorder
  .set  nomacro
  .set  noat
```

```
# BB#0:
  lui $2, %hi(_gp_disp)
  addiu $2, $2, %lo(_gp_disp)
  addiu $sp, $sp, -40
$tmp5:
  .cfi_def_cfa_offset 40
  sw  $ra, 36($sp)            # 4-byte Folded Spill
$tmp6:
  .cfi_offset 31, -4
  addu  $gp, $2, $25
  sw  $zero, 32($sp)
  addiu $1, $zero, 6
  sw  $1, 20($sp) // Save argument 6 to 20($sp)
  addiu $1, $zero, 5
  sw  $1, 16($sp) // Save argument 5 to 16($sp)
  lw  $25, %call16(_Z5sum_iiiiiii)($gp)
  addiu $4, $zero, 1    // Pass argument 1 to $4 (=$a0)
  addiu $5, $zero, 2    // Pass argument 2 to $5 (=$a1)
  addiu $6, $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 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 happen.

```
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, ...
...
```

[2] https://www.dropbox.com/sh/2pkh1fewlq2zag9/OHnrYn2nOs/doc/MIPSproAssemblyLanguageProgrammerGuide

[3] http://www.linux-mips.org/pub/linux/mips/doc/ABI/mipsabi.pdf

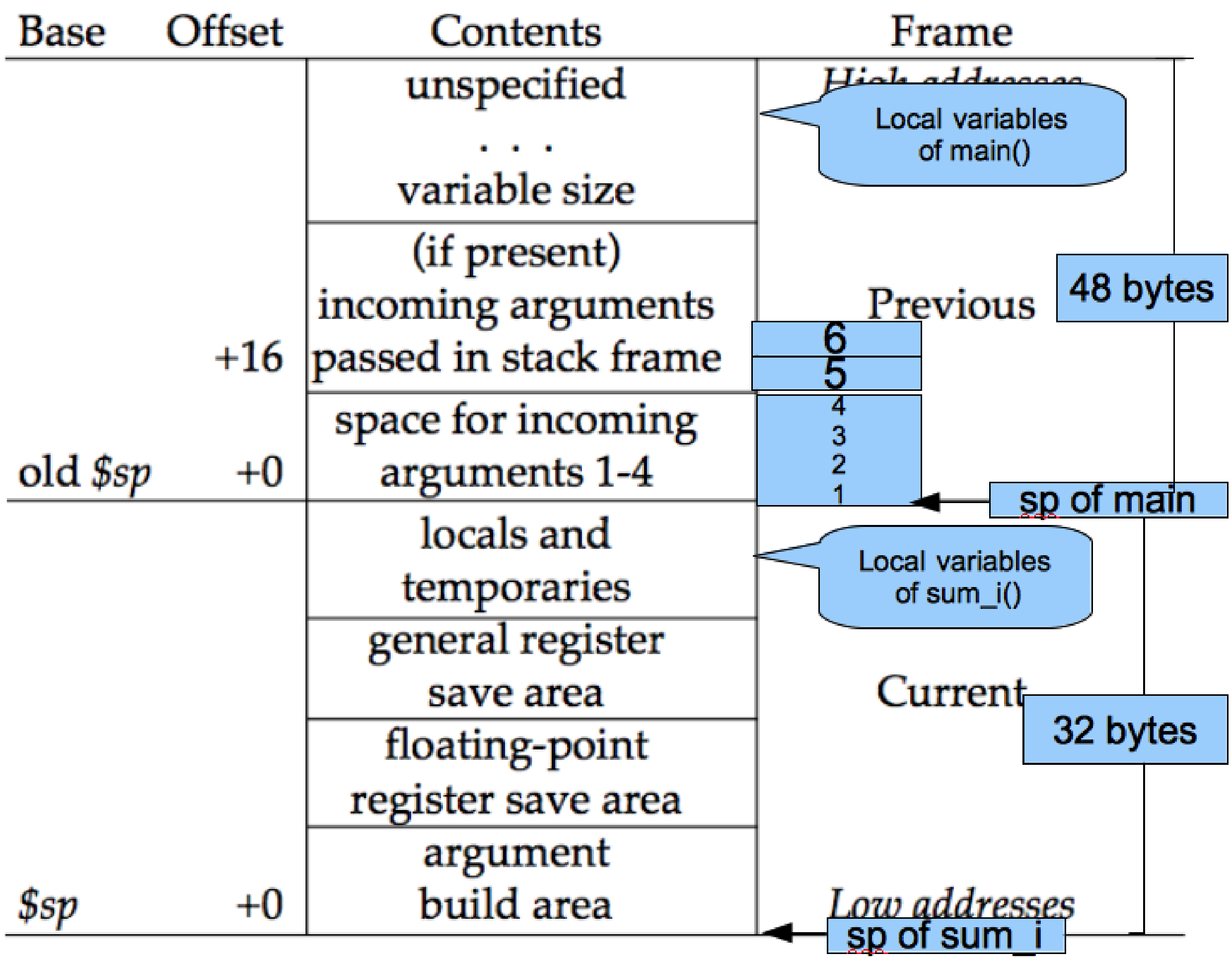


Figure 8.2: Mips arguments location in stack frame

```
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_2/Cpu0CallingConv.td

As above, CC_Cpu0 is the cpu0 Calling Convention which delegate to CC_Cpu0EABI and define the CC_Cpu0EABI. The reason we don't define the Calling Convention directly in CC_Cpu0 is that a real general CPU like Mips can have several Calling Convention. Combine with the mechanism of "section Target Registration" [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_2/Cpu0ISelLowering.cpp

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 `llc` 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.

LLVMBackendTutorialExampleCode/Chapter8_2/Cpu0InstrInfo.cpp

In addition to Calling Convention and LowerFormalArguments(), Chapter8_2/ add the following code for cpu0 instructions **swi** (Software Interrupt), **jsub** and **jalr** (function call) definition and printing.

[4] http://jonathan2251.github.com/lbd/llvmstructure.html#target-registration
[5] http://jonathan2251.github.com/lbd/globalvar.html#global-variable

LLVMBackendTutorialExampleCode/Chapter8_2/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_2/Cpu0InstrInfo.td

```
def SDT_Cpu0JmpLink      : SDTypeProfile<0, 1, [SDTCisVT<0, iPTR>]>;
...
// Call
def Cpu0JmpLink : SDNode<"Cpu0ISD::JmpLink",SDT_Cpu0JmpLink,
                         [SDNPHasChain, SDNPOutGlue, SDNPOptInGlue,
                          SDNPVariadic]>;
...
def jmptarget   : 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> {
       let DecoderMethod = "DecodeJumpTarget";
       }

  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)>;
...
```

LLVMBackendTutorialExampleCode/Chapter8_2/Cpu0InstPrinter.cpp

```
static void printExpr(const MCExpr *Expr, raw_ostream &OS) {
  switch (Kind) {
  ...
  case MCSymbolRefExpr::VK_Cpu0_GOT_CALL:  OS << "%call24("; break;
  ...
  }
...
}
```

LLVMBackendTutorialExampleCode/Chapter8_2/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_2/Cpu0MachineFucntion.h

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

  // Range of frame object indices.
  // InArgFIRange: Range of indices of all frame objects created during call to
  //               LowerFormalArguments.
  // OutArgFIRange: Range of indices of all frame objects created during call to
  //                LowerCall except for the frame object for restoring $gp.
  std::pair<int, int> InArgFIRange, OutArgFIRange;
  int 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 needGPSaveRestore() const { return getGPFI(); }
  bool isGPFI(int FI) const { return GPFI && GPFI == FI; }

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

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

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

After above changes, you can run Chapter8_2/ 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_3/Cpu0ISelLowering.cpp

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 $6, %call24(_Z5sum_iiiiii)($gp)"** and jalr $6 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_3/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_3/Cpu0InstrInfo.cpp

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

  unsigned Opc = 0;
```

```
  if (RC == Cpu0::CPURegsRegisterClass)
    Opc = Cpu0::ST;
  assert(Opc && "Register class not handled!");
  BuildMI(MBB, I, DL, get(Opc)).addReg(SrcReg, getKillRegState(isKill))
    .addFrameIndex(FI).addImm(0).addMemOperand(MMO);
}
```

Now, let's run Chapter8_3/ 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
  .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)
  ld  $3, 24($sp)
  ld  $4, 28($sp)
  add $3, $4, $3
  ld  $4, 20($sp)
  add $3, $3, $4
  ld  $4, 16($sp)
  add $3, $3, $4
  ld  $4, 12($sp)
  add $3, $3, $4
  add $2, $3, $2
  st  $2, 4($sp)
  addiu $sp, $sp, 32
  ret $lr
  .set  macro
  .set  reorder
```

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

  .globl  main
  .align  2
  .type main,@function
  .ent  main                    # @main
main:
  .cfi_startproc
  .frame  $sp,40,$lr
  .mask   0x00004000,-4
  .set  noreorder
  .cpload $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) // wrong offset
  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  $6, %call24(_Z5sum_iiiiiii)($gp)
  jalr  $6
  !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
```

It store the arguments to wrong offset. We will fix this issue and take care !ADJCALLSTACKUP and !ADJCALLSTACKDOWN in next two sections.

8.4 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_4/ to set the caller outgoing arguments into spOffset($sp) (Chapter8_3/ set them to pOffset+stackSize($sp).

LLVMBackendTutorialExampleCode/Chapter8_4/Cpu0RegisterInfo.cpp

```
void Cpu0RegisterInfo::
eliminateFrameIndex(MachineBasicBlock::iterator II, int SPAdj,
                    RegScavenger *RS) const {
  ...
  Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();
  ...
  if (Cpu0FI->isOutArgFI(FrameIndex) || Cpu0FI->isDynAllocFI(FrameIndex) ||
      (FrameIndex >= MinCSFI && FrameIndex <= MaxCSFI))
    FrameReg = Cpu0::SP;
  else
    FrameReg = getFrameRegister(MF);
  ...
  // Calculate final offset.
  // - There is no need to change the offset if the frame object is one of the
  //   following: an outgoing argument, pointer to a dynamically allocated
  //   stack space or a $gp restore location,
  // - If the frame object is any of the following, its offset must be adjusted
  //   by adding the size of the stack:
  //   incoming argument, callee-saved register location or local variable.
  if (Cpu0FI->isOutArgFI(FrameIndex) || Cpu0FI->isGPFI(FrameIndex) ||
      Cpu0FI->isDynAllocFI(FrameIndex))
    Offset = spOffset;
  else
    Offset = spOffset + (int64_t)stackSize;
  Offset    += MI.getOperand(i+1).getImm();
  ...
}
```

LLVMBackendTutorialExampleCode/Chapter8_4/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_4/ with ch8_1.cpp will get the following result. It correct arguements offset im main() from (0+40)$sp, (8+40)$sp, ..., to (0)$sp, (8)$sp, ..., where the stack size is 40 in main().

```
118-165-78-230:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm ch8_1.bc -o
ch8_1.cpu0.s
118-165-78-230:InputFiles Jonathan$ cat ch8_1.cpu0.s
  ...
  !ADJCALLSTACKDOWN 24
  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  $6, %call24(_Z5sum_iiiiii)($gp)
  jalr  $6
  !ADJCALLSTACKUP 24
  ...
```

The incoming arguments is the formal arguments defined in compiler and program language books. The outgoing arguments is the actual arguments. Summary callee incoming arguments and caller outgoing arguments as Figure 8.3.

* Arguments location is calculated in Cpu0RegisterInfo::eliminateFrameIndex().

	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

Figure 8.3: Callee incoming arguments and caller outgoing arguments

8.5 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_5/Cpu0InstrInfo.cpp

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

LLVMBackendTutorialExampleCode/Chapter8_5/Cpu0FrameLowering.cpp

```
// 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_5/ with ch8_1.cpp will get the following result.

```
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_iiiiii
  .align  2
  .type _Z5sum_iiiiii,@function
  .ent  _Z5sum_iiiiii          # @_Z5sum_iiiiii
_Z5sum_iiiiii:
  .cfi_startproc
  .frame  $sp,32,$lr
  .mask   0x00000000,0
  .set  noreorder
  .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)
  ld  $3, 24($sp)
  ld  $4, 28($sp)
  add $3, $4, $3
  ld  $4, 20($sp)
  add $3, $3, $4
  ld  $4, 16($sp)
  add $3, $3, $4
  ld  $4, 12($sp)
  add $3, $3, $4
  add $2, $3, $2
  st  $2, 4($sp)
  addiu $sp, $sp, 32
  ret $lr
  .set  macro
  .set  reorder
  .end  _Z5sum_iiiiiii
$tmp2:
  .size _Z5sum_iiiiiii, ($tmp2)-_Z5sum_iiiiiii
  .cfi_endproc

  .globl  main
  .align  2
  .type main,@function
  .ent  main                    # @main
main:
  .cfi_startproc
  .frame  $sp,64,$lr
  .mask   0x00004000,-4
  .set  noreorder
  .cpload $t9
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -64
$tmp5:
  .cfi_def_cfa_offset 64
  st  $lr, 60($sp)            # 4-byte Folded Spill
$tmp6:
  .cfi_offset 14, -4
  addiu $2, $zero, 0
  st  $2, 56($sp)
  addiu $2, $zero, 6
  st  $2, 20($sp)
  addiu $2, $zero, 5
  st  $2, 16($sp)
  addiu $2, $zero, 4
  st  $2, 12($sp)
  addiu $2, $zero, 3
  st  $2, 8($sp)
  addiu $2, $zero, 2
  st  $2, 4($sp)
  addiu $2, $zero, 1
  st  $2, 0($sp)
  ld  $6, %call24(_Z5sum_iiiiiii)($gp)
  jalr  $6
  st  $2, 52($sp)
  ld  $lr, 60($sp)            # 4-byte Folded Reload
```

```
  addiu $sp, $sp, 64
  ret $lr
  .set  macro
  .set  reorder
  .end  main
$tmp7:
  .size main, ($tmp7)-main
  .cfi_endproc
```

8.6 Handle $gp register in PIC addressing mode

In "section Global variable" [5], we mentioned two addressing mode, the static address mode and PIC (position-independent code) mode. We also mentioned, one example of PIC mode is used in share library. Share library usually can be loaded in different memory address decided at run time. The static mode (absolute address mode) is usually designed to load in specific memory address decided at compile time. Since share library can be loaded in different memory address, the global variable address cannot be decided at compile time. But, we can caculate the distance between the global variable address and shared library function if they will be loaded to the contiguous memory space together.

Let's run Chapter8_6/ with ch8_2.cpp to get the following result of we putting the comment in it for explanation.

```
118-165-78-230:InputFiles Jonathan$ cat ch8_2.cpu0.s
_Z5sum_iiiiiii:
...
    .cpload $t9 // assign $gp = $t9 by loader when loader load re-entry
                //  function (shared library) of _Z5sum_iiiiii
    .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_iiiiii)
...
    ret $lr
    .set    macro
    .set    reorder
    .end    _Z5sum_iiiiii
...
    .ent    main                    # @main
main:
    .cfi_startproc
...
    .cpload $t9
    .set    nomacro
...
    .cprestore  24    // save $gp to 24($sp)
    addiu   $2, $zero, 0
...
    ld  $6, %call24(_Z5sum_iiiiiii)($gp)
    jalr    $6        // $t9 register number is 6, meaning $6 and %t9 are the
                      //  same register
    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)**; Instruction **"ld $gp, 24($sp)"** will restore the $gp. In other word, $gp is 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 offset of (gI - _Z5sum_iiiiii) (we can write our cpu0 compiler to produce obj code by calculate the offset value).

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 is designed for teaching purpose, this book has the same purpose for llvm backend design. We reserve **"jalr"** as PIC mode for shared library or dynamic loading code to demonstrate the caller how to handle the caller saved register $gp in calling the shared library and the shared library how to use $gp to access global variable address. This solution is popular in reality and deserve change cpu0 official design as a compiler book.

Now, as the following code added in Chapter8_6/, 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_6/CMakeLists.txt

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

LLVMBackendTutorialExampleCode/Chapter8_6/Cpu0TargetMachine.cpp

```
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_6/Cpu0.h

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

LLVMBackendTutorialExampleCode/Chapter8_6/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_6/Cpu0InstrInfo.td

```
// When handling PIC code the assembler needs .cpload and .cprestore
// directives. If the real instructions corresponding these directives
// are used, we have the same behavior, but get also a bunch of warnings
// from the assembler.
let neverHasSideEffects = 1 in
def CPRESTORE : Cpu0Pseudo<(outs), (ins i32imm:$loc, CPURegs:$gp),
               ".cprestore\t$loc", []>;
```

LLVMBackendTutorialExampleCode/Chapter8_6/Cpu0ISelLowering.cpp

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

LLVMBackendTutorialExampleCode/Chapter8_6/Cpu0EmitGPRestore.cpp

LLVMBackendTutorialExampleCode/Chapter8_6/Cpu0MachineFunctionInfo.h

```
//===-- Cpu0MachineFunctionInfo.h - Private data used for Cpu0 ----*- C++ -*-=//
...
class Cpu0FunctionInfo : public MachineFunctionInfo {
  ...
  bool EmitNOAT;
```

```
public:
  Cpu0FunctionInfo(MachineFunction& MF)
  : ...
  MaxCallFrameSize(0), EmitNOAT(false)
  ...
  bool getEmitNOAT() const { return EmitNOAT; }
  void setEmitNOAT() { EmitNOAT = true; }

};

} // end of namespace llvm

#endif // CPU0_MACHINE_FUNCTION_INFO_H
```

LLVMBackendTutorialExampleCode/Chapter8_6/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);
}

void Cpu0AsmPrinter::EmitFunctionBodyStart() {
  ...
  if (OutStreamer.hasRawTextSupport()) {
    ...
    if (Cpu0FI->getEmitNOAT())
      OutStreamer.EmitRawText(StringRef("\t.set\tnoat"));
  } else if (EmitCPLoad) {
    SmallVector<MCInst, 4> MCInsts;
    MCInstLowering.LowerCPLOAD(MCInsts);
    for (SmallVector<MCInst, 4>::iterator I = MCInsts.begin();
       I != MCInsts.end(); ++I)
      OutStreamer.EmitInstruction(*I);
  }
}
```

LLVMBackendTutorialExampleCode/Chapter8_6/Cpu0MCInstLower.cpp

The above added code of Cpu0AsmPrinter.cpp will call the LowerCPLOAD() and LowerCPRESTORE() when user run with `llc -filetype=obj`. The above added code of Cpu0MCInstLower.cpp take care the .cpload and .cprestore machine instructions. It translate pseudo asm .cpload into four machine instructions, and .cprestore into one machine instruction as below. As mentioned in "section Global variable" [5]. When the share library main() function be loaded, the loader will set the $t9 value to $gp when meet **".cpload $t9"**. After that, the $gp value is $t9 which point to main(), and the global variable address is the relative address to main(). The _gp_disp is zero as the following reason from Mips ABI.

```
// Lower ".cpload $reg" to
//  "addiu $gp, $zero, %hi(_gp_disp)"
//  "shl   $gp, $gp, 16"
//  "addiu $gp, $gp, %lo(_gp_disp)"
//  "addu  $gp, $gp, $t9"

// Lower ".cprestore offset" to "st $gp, offset($sp)".
```

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.

By run with `llc -filetype=obj`, the .cpload and .cprestore are translated into machine code as follows,

```
118-165-76-131:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=
obj ch8_2.bc -o ch8_2.cpu0.o
118-165-76-131:InputFiles Jonathan$ hexdump  ch8_2.cpu0.o
...
// .cpload machine instructions "09 a0 00 00 to 13 aa 60 00"
0000030 00 0a 00 07 09 a0 00 00 1e aa 00 10 09 aa 00 00
0000040 13 aa 60 00 09 dd ff e0 00 2d 00 20 01 2d 00 1c
...

// .cpload machine instructions "09 a0 00 00 to 13 aa 60 00"
00000b0 09 dd 00 20 2c 00 00 00 09 a0 00 00 1e aa 00 10
00000c0 09 aa 00 00 13 aa 60 00 09 dd ff b8 01 ed 00 44
// .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_2.cpu0.s
...
  .ent  _Z5sum_iiiiiii          # @_Z5sum_iiiiiii
_Z5sum_iiiiiii:
...
  .cpload $t9 // assign $gp = $t9 by loader when loader load re-entry function
              // (shared library) of _Z5sum_iiiiiii
  .set  nomacro
# BB#0:
...
  .ent  main                    # @main
...
  .cpload $t9
  .set  nomacro
...
  .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_2.bc -o ch8_2.cpu0.s
118-165-76-131:InputFiles Jonathan$ cat ch8_2.cpu0.s
...
  jsub  _Z5sum_iiiiiii
...
```

Run with `llc -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_2.cpp, 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
```

8.7 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_6/ with ch8_3.cpp to get the following error,

LLVMBackendTutorialExampleCode/InputFiles/ch8_3.cpp

```
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
```

Run Chapter8_7/ 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
  .cpload $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  $6, %call24(_Z5sum_iiz)($gp)
  jalr  $6
  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_7/. The ch8_3_2.cpp is C++ template example code, it can be translated into cpu0 backend code too.

LLVMBackendTutorialExampleCode/Chapter8_7/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

Mips qemu reference [6], 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.

8.8 Correct the return of main()

Run Chapter8_7/ with ch6_2.cpp to get the incorrect main return (return register $2 is not 0) as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch6_2.cpp

```
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
```

[6] http://developer.mips.com/clang-llvm/

```
# 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
...
```

The LowerReturn() modified in Chapter8_8/ as below. It add the live out register $2 to function (main() as this example), and copy the OutVals[0] (0 as this example) to $2. Then call DAG.getNode(..., Flag) where Flag contains $2 and OutVals[0] information.

LLVMBackendTutorialExampleCode/Chapter8_8/Cpu0ISelLowering.cpp

Run Chapter8_8/ 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
```

8.9 Verify DIV for operator %

Now, let's run Chapter8_8/ with ch4_6_2.cpp to get the result as below. It translate **"(b+1)%c"** into **"div $zero, $3, $2"** and **"mfhi $2"**.

LLVMBackendTutorialExampleCode/InputFiles/ch4_6_2.cpp

```
118-165-70-242:InputFiles Jonathan$ clang -c ch4_6_2.cpp -I/Applications/
Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/
MacOSX10.8.sdk/usr/include/ -emit-llvm -o ch4_6_2.bc
118-165-70-242:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake
_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch4_6_2.bc -o ch4_6_2.cpu0.s
118-165-70-242:InputFiles Jonathan$ cat ch4_6_2.cpu0.s
  ...
  div $3, $2
  mfhi  $2
  ...
```

8.10 Structure type support

Run 8/8 with ch8_9_1.cpp will get the error message as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch8_9_1.cpp

```
JonathantekiiMac:InputFiles Jonathan$ clang -c ch8_9_1.cpp -emit-llvm -o
ch8_9_1.bc
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch8_9_1.bc -o ch8_9_1.cpu0.s
LLVM ERROR: Cannot select: 0x7fbe7c032210: ch = Cpu0ISD::Ret 0x7fbe7c032110 [ID=36]
In function: _Z7getDatev
...
```

Chapter8_9/ with the following code added to support the structure type in function call.

LLVMBackendTutorialExampleCode/Chapter8_9/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(ByValAlign, (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_9/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 size, cpu0 will save them with pointer. For explanation, let's run Chapter8_9/ with ch8_9_1.cpp and explain with this example.

```
JonathantekiiMac:InputFiles Jonathan$ cat ch8_9_1.cpu0.s
  .section .mdebug.abi32
  .previous
  .file "ch8_9_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
  .cpload $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)       // time1.minute, save time1.minute and
  st  $2, 36($sp)       //  time1.second to 36..32($sp)
  ld  $2, 16($sp)       // time1.second
  st  $2, 32($sp)
  ld  $2, 40($sp)       // $2 = 8($sp of main) = time1.hour
  st  $2, 56($sp)       // copy time1 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 time1 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 time1 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
  .cpload $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  $2, %got($_ZZ4mainE5time1)($gp)
  addiu $2, $2, %lo($_ZZ4mainE5time1)
  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  $6, %call24(_Z7getDatev)($gp) // copy gDate contents to date1, 212..192($sp)
  jalr  $6
  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  $6, %call24(_Z8copyDate4Date)($gp)
jalr  $6
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  $6, %call24(_Z8copyDateP4Date)($gp)
jalr  $6
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  $6, %call24(_Z8copyTime4Time)($gp)
jalr  $6
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  $6, %call24(_Z8copyTimeP4Time)($gp)
jalr  $6
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_9_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  $6, %call24(_Z8copyDate4Date)($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 [7]. 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,

LLVMBackendTutorialExampleCode/Chapter8_9/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  $6, %call24(_Z8copyDate4Date)($gp)
```

LLVMBackendTutorialExampleCode/Chapter8_9/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
```

[7] section "4.5.1 Calling Conventions" of tricore_llvm.pdf

```
  // 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_9_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
```

8.11 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.

LLVMBackendTutorialExampleCode/InputFiles/ch8_10.cpp

Run Chapter8_9 with ch8_10.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_10.cpp -emit-llvm -o ch8_10.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_10.bc -o
ch8_10.cpu0.s
LLVM ERROR: Cannot select: 0x7ffd8b02ff10: i32,ch = dynamic_stackalloc
0x7ffd8b02f910:1, 0x7ffd8b02fe10, 0x7ffd8b02c010 [ORD=12] [ID=48]
  0x7ffd8b02fe10: i32 = and 0x7ffd8b02fc10, 0x7ffd8b02fd10 [ORD=12] [ID=47]
    0x7ffd8b02fc10: i32 = add 0x7ffd8b02fa10, 0x7ffd8b02fb10 [ORD=12] [ID=46]
      0x7ffd8b02fa10: i32 = shl 0x7ffd8b02f910, 0x7ffd8b02f510 [ID=45]
        0x7ffd8b02f910: i32,ch = load 0x7ffd8b02ee10, 0x7ffd8b02e310,
        0x7ffd8b02b310<LD4[%1]> [ID=44]
          0x7ffd8b02e310: i32 = FrameIndex<1> [ORD=3] [ID=10]
```

```
          0x7ffd8b02b310: i32 = undef [ORD=1] [ID=2]
        0x7ffd8b02f510: i32 = Constant<2> [ID=25]
      0x7ffd8b02fb10: i32 = Constant<7> [ORD=12] [ID=16]
    0x7ffd8b02fd10: i32 = Constant<-8> [ORD=12] [ID=17]
  0x7ffd8b02c010: i32 = Constant<0> [ORD=12] [ID=8]
In function: _Z5sum_iiiiiii
```

Chapter8_10 support dynamic stack allocation with the following code added.

LLVMBackendTutorialExampleCode/Chapter8_10/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 {
  ...
  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_10/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_10/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_10 with ch8_10.cpp will get the following 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_10.cpp -emit-llvm -o ch8_10.bc
118-165-72-242:InputFiles Jonathan$ llvm-dis ch8_10.bc -o ch8_10.ll
118-165-72-242:InputFiles Jonathan$ cat ch8_10.ll
; ModuleID = 'ch8_10.bc'
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:
32:64-S128"
target triple = "x86_64-apple-macosx10.8.0"

define i32 @_Z5sum_iiiiiii(i32 %x1, i32 %x2, i32 %x3, i32 %x4, i32 %x5, i32 %x6)
 nounwind uwtable ssp {
  ...
  %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_10.bc -o
ch8_10.cpu0.s
118-165-72-242:InputFiles Jonathan$ cat ch8_10.cpu0.s
...
_Z10weight_sumiiiiii:
```

```
	.cfi_startproc
	.frame	$fp,80,$lr
	.mask	0x00004080,-4
	.set	noreorder
	.cpload	$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	$6, 48($fp)
	ld	$t0, 44($fp)
	st	$t0, 20($sp)
	shl	$6, $6, 1
	st	$6, 16($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	$6, %call24(_Z3sumiiiiii)($gp)
	jalr	$6
	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	$6, %call24(__stack_chk_fail)($gp)
	jalr	$6
	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 8.4, 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 8.5 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**.

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 8.6 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_10.cpu0.s, it access local variable (above of alloca()) by fp offset and outgoing arguments (below of alloca()) by sp offset.

8.12 Summary of this chapter

Until now, we have 5,900 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 3,000 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

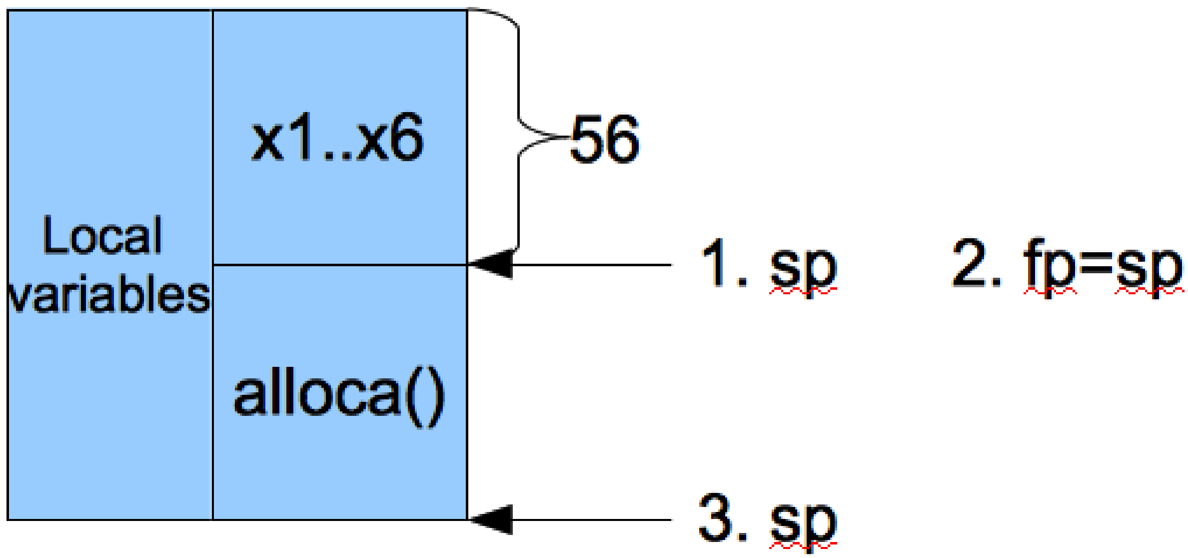


Figure 8.4: Frame pointer changes when enter function

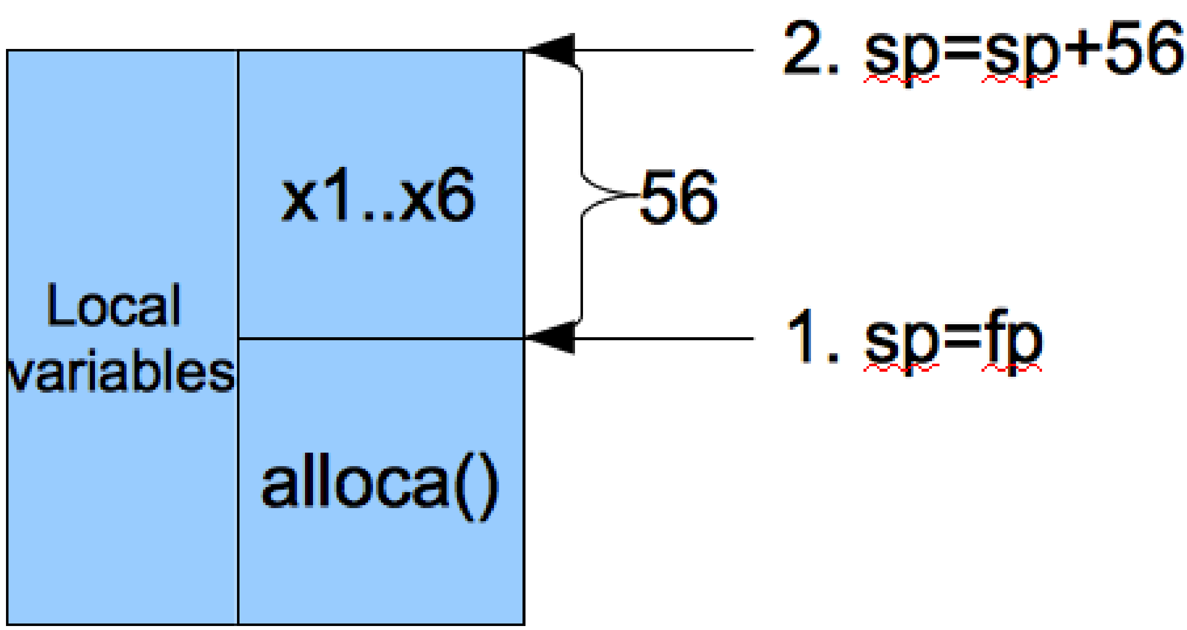


Figure 8.5: Stack pointer changes when exit function

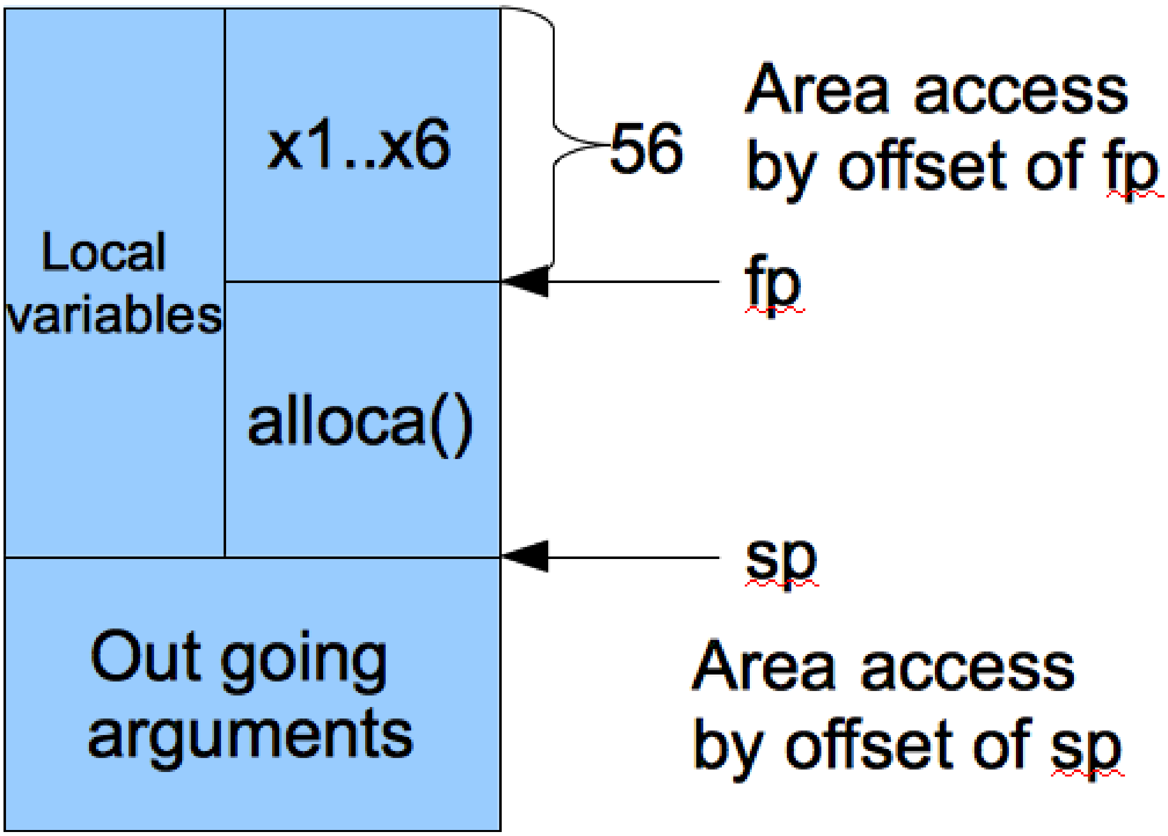


Figure 8.6: fp and sp access areas

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++ syntex, is handled by front end. LLVM is an real structure following the compiler theory, any backend of LLVM can benefit from this structure. A couple of thousands lines of code make OOP language translated into your backend. And your backend will grow up automatically through the front end support languages more and more.

CHAPTER

NINE

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"[#]_ 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.

[3], [4], [5] 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.

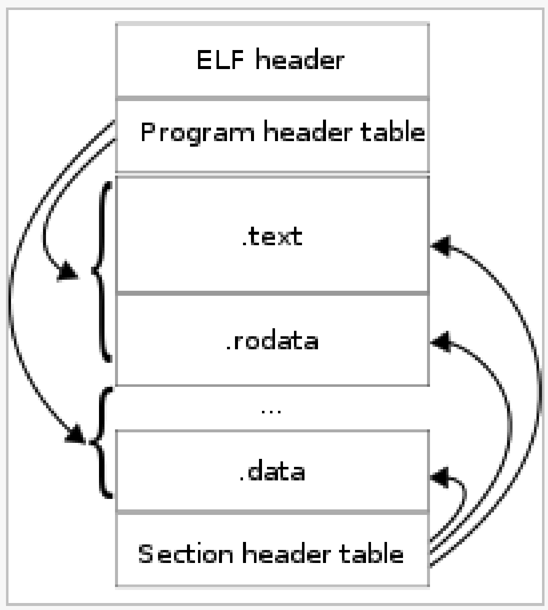
[1] http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[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



An ELF file has two views: The program header shows the *segments* used at run-time, whereas the section header lists the set of *sections* of the binary.

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 MIPSR3000. 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            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000034 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 000310 000018 08      8   1  4
  [ 3] .data             PROGBITS        00000000 000068 000004 00  WA  0   0  4
  [ 4] .bss              NOBITS          00000000 00006c 000000 00  WA  0   0  4
  [ 5] .eh_frame         PROGBITS        00000000 00006c 000028 00   A  0   0  4
  [ 6] .rel.eh_frame     REL             00000000 000328 000008 08      8   5  4
  [ 7] .shstrtab         STRTAB          00000000 000094 00003e 00      0   0  1
  [ 8] .symtab           SYMTAB          00000000 000264 000090 10      9   6  4
  [ 9] .strtab           STRTAB          00000000 0002f4 00001b 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
[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
  .cpload $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:
// .cpload 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 **".cpload %reg"** into the following.

```
// Lower ".cpload $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 **"ld $2, %got(gI)($gp)"** is same since we don't know what the address of .data section variable will load to. So, translate the address to 0 and made a relocation record on 0x00000020 of .text section. 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 [6].

[6] http://ccckmit.wikidot.com/lk:elf

9.6 llvm-objdump

9.6.1 llvm-objdump -t -r

In linux, `objdump -tr` can display the information of relocation records like `readelf -tr`. LLVM tool llvm-objdump is the same tool as objdump. Let's run the llvm-objdump command as follows to see the difference.

```
118-165-83-10: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
118-165-83-10:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build/
bin/Debug/llvm-objdump -t -r ch8_3_3.cpu0.o

118-165-83-10:InputFiles Jonathan$ llvm-objdump -t -r ch8_3_3.cpu0.o

ch8_3_3.cpu0.o: file format ELF32-unknown

RELOCATION RECORDS FOR [.text]:
0 Unknown Unknown
8 Unknown Unknown
28 Unknown Unknown
188 Unknown Unknown
224 Unknown Unknown
236 Unknown Unknown
244 Unknown Unknown
324 Unknown Unknown
344 Unknown Unknown
348 Unknown Unknown
356 Unknown Unknown

RELOCATION RECORDS FOR [.eh_frame]:
28 Unknown Unknown
52 Unknown Unknown

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 ch8_3_3.bc
00000000 l       .rodata.str1.1 00000008 $.str
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 l    d  .rodata.str1.1 00000000 .rodata.str1.1
00000000 l    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

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 l    df *ABS*  00000000 ch8_3_3.bc
00000000 l       .rodata.str1.1 00000008 $.str
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 l    d  .rodata.str1.1 00000000 .rodata.str1.1
00000000 l    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_RELOC_TYPE_NAME(R_CPU0_NONE);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_32);
```

```
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_REL32);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_24);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_HI16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_LO16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_GPREL16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_LITERAL);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_GOT16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_PC24);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_CALL24);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_GPREL32);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_SHIFT5);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_SHIFT6);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_64);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_GOT_DISP);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_GOT_PAGE);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_GOT_OFST);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_GOT_HI16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_GOT_LO16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_SUB);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_INSERT_A);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_INSERT_B);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_DELETE);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_HIGHER);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_HIGHEST);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_CALL_HI16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_CALL_LO16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_SCN_DISP);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_REL16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_ADD_IMMEDIATE);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_PJUMP);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_RELGOT);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_JALR);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_DTPMOD32);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_DTPREL32);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_DTPMOD64);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_DTPREL64);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_GD);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_LDM);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_DTPREL_HI16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_DTPREL_LO16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_GOTTPREL);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_TPREL32);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_TPREL64);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_TPREL_HI16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_TLS_TPREL_LO16);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_GLOB_DAT);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_COPY);
    LLVM_ELF_SWITCH_RELOC_TYPE_NAME(R_CPU0_JUMP_SLOT);
    LLVM_ELF_SWITCH_RELOC_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$addr"),
       [(brcond RC:$ra, bb:$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

LLVMBackendTutorialExampleCode/Chapter9_1/Disassembler/LLVMBuild.txt

LLVMBackendTutorialExampleCode/Chapter9_1/Disassembler/Cpu0Disassembler.cpp

As above code, it add directory Disassembler for handling the obj to assembly code reverse translation. So, add Disassembler/Cpu0Disassembler.cpp and modify the CMakeList.txt and LLVMBuild.txt to build with directory Disassembler and enable the disassembler table generated by "has_disassembler = 1". Most of code is handled by the table of *.td files defined. Not every instruction in *.td can be disassembled without trouble even though they can be translated into assembly and obj successfully. For those cannot be disassembled, LLVM supply the **"let DecoderMethod"** keyword to allow programmers implement their decode function. In Cpu0 example, we define function DecodeCMPInstruction(), DecodeBranchTarget() and DecodeJumpAbsoluteTarget() in Cpu0Disassembler.cpp and tell the LLVM table driven system by write **"let DecoderMethod = ..."** in the corresponding instruction definitions or ISD node of Cpu0InstrInfo.td. LLVM will call these DecodeMethod when user use Disassembler job in tools, like `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:
.text:
     0: 09 dd ff d8                                   addiu $sp, $sp, -40
     4: 09 30 00 00                                   addiu $3, $zero, 0
     8: 01 3d 00 24                                   st  $3, 36($sp)
     c: 01 3d 00 20                                   st  $3, 32($sp)
    10: 09 20 00 01                                   addiu $2, $zero, 1
    14: 01 2d 00 1c                                   st  $2, 28($sp)
    18: 09 40 00 02                                   addiu $4, $zero, 2
    1c: 01 4d 00 18                                   st  $4, 24($sp)
    20: 09 40 00 03                                   addiu $4, $zero, 3
    24: 01 4d 00 14                                   st  $4, 20($sp)
    28: 09 40 00 04                                   addiu $4, $zero, 4
    2c: 01 4d 00 10                                   st  $4, 16($sp)
    30: 09 40 00 05                                   addiu $4, $zero, 5
    34: 01 4d 00 0c                                   st  $4, 12($sp)
    38: 09 40 00 06                                   addiu $4, $zero, 6
    3c: 01 4d 00 08                                   st  $4, 8($sp)
    40: 09 40 00 07                                   addiu $4, $zero, 7
    44: 01 4d 00 04                                   st  $4, 4($sp)
    48: 09 40 00 08                                   addiu $4, $zero, 8
```

```
  4c: 01 4d 00 00                                   st   $4, 0($sp)
  50: 00 4d 00 20                                   ld   $4, 32($sp)
  54: 10 43 00 00                                   cmp $4, $3
  58: 21 00 00 10                                   jne 16
  5c: 26 00 00 00                                   jmp 0
  60: 00 4d 00 20                                   ld   $4, 32($sp)
  64: 09 44 00 01                                   addiu $4, $4, 1
  68: 01 4d 00 20                                   st   $4, 32($sp)
  6c: 00 4d 00 1c                                   ld   $4, 28($sp)
  70: 10 43 00 00                                   cmp $4, $3
  74: 20 00 00 10                                   jeq 16
  78: 26 00 00 00                                   jmp 0
  7c: 00 4d 00 1c                                   ld   $4, 28($sp)
  80: 09 44 00 01                                   addiu $4, $4, 1
  84: 01 4d 00 1c                                   st   $4, 28($sp)
  88: 00 4d 00 18                                   ld   $4, 24($sp)
  8c: 10 42 00 00                                   cmp $4, $2
  90: 22 00 00 10                                   jlt 16
  94: 26 00 00 00                                   jmp 0
  98: 00 4d 00 18                                   ld   $4, 24($sp)
  9c: 09 44 00 01                                   addiu $4, $4, 1
  a0: 01 4d 00 18                                   st   $4, 24($sp)
  a4: 00 4d 00 14                                   ld   $4, 20($sp)
  a8: 10 43 00 00                                   cmp $4, $3
  ac: 22 00 00 10                                   jlt 16
  b0: 26 00 00 00                                   jmp 0
  b4: 00 4d 00 14                                   ld   $4, 20($sp)
  b8: 09 44 00 01                                   addiu $4, $4, 1
  bc: 01 4d 00 14                                   st   $4, 20($sp)
  c0: 09 40 ff ff                                   addiu $4, $zero, -1
  c4: 00 5d 00 10                                   ld   $5, 16($sp)
  c8: 10 54 00 00                                   cmp $5, $4
  cc: 23 00 00 10                                   jgt 16
  d0: 26 00 00 00                                   jmp 0
  d4: 00 4d 00 10                                   ld   $4, 16($sp)
  d8: 09 44 00 01                                   addiu $4, $4, 1
  dc: 01 4d 00 10                                   st   $4, 16($sp)
  e0: 00 4d 00 0c                                   ld   $4, 12($sp)
  e4: 10 43 00 00                                   cmp $4, $3
  e8: 23 00 00 10                                   jgt 16
  ec: 26 00 00 00                                   jmp 0
  f0: 00 3d 00 0c                                   ld   $3, 12($sp)
  f4: 09 33 00 01                                   addiu $3, $3, 1
  f8: 01 3d 00 0c                                   st   $3, 12($sp)
  fc: 00 3d 00 08                                   ld   $3, 8($sp)
 100: 10 32 00 00                                   cmp $3, $2
 104: 23 00 00 10                                   jgt 16
 108: 26 00 00 00                                   jmp 0
 10c: 00 3d 00 08                                   ld   $3, 8($sp)
 110: 09 33 00 01                                   addiu $3, $3, 1
 114: 01 3d 00 08                                   st   $3, 8($sp)
 118: 00 3d 00 04                                   ld   $3, 4($sp)
 11c: 10 32 00 00                                   cmp $3, $2
 120: 22 00 00 10                                   jlt 16
 124: 26 00 00 00                                   jmp 0
 128: 00 2d 00 04                                   ld   $2, 4($sp)
 12c: 09 22 00 01                                   addiu $2, $2, 1
 130: 01 2d 00 04                                   st   $2, 4($sp)
```

```
134: 00 2d 00 04                                          ld  $2, 4($sp)
138: 00 3d 00 00                                          ld  $3, 0($sp)
13c: 10 32 00 00                                          cmp $3, $2
140: 25 00 00 10                                          jge 16
144: 26 00 00 00                                          jmp 0
148: 00 2d 00 00                                          ld  $2, 0($sp)
14c: 09 22 00 01                                          addiu $2, $2, 1
150: 01 2d 00 00                                          st  $2, 0($sp)
154: 00 2d 00 1c                                          ld  $2, 28($sp)
158: 00 3d 00 20                                          ld  $3, 32($sp)
15c: 10 32 00 00                                          cmp $3, $2
160: 20 00 00 10                                          jeq 16
164: 26 00 00 00                                          jmp 0
168: 00 2d 00 20                                          ld  $2, 32($sp)
16c: 09 22 00 01                                          addiu $2, $2, 1
170: 01 2d 00 20                                          st  $2, 32($sp)
174: 00 2d 00 20                                          ld  $2, 32($sp)
178: 09 dd 00 28                                          addiu $sp, $sp, 40
17c: 2c 00 00 00                                          ret $zero
```

CHAPTER TEN

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

```
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

LLVMBackendTutorialExampleCode/Chapter10_1/AsmParser/CMakeLists.txt

LLVMBackendTutorialExampleCode/Chapter10_1/AsmParser/LLVMBuild.txt

The Cpu0AsmParser.cpp contains one thousand of code which do the assembly language parsing. You can understand it with a little patient only. To let directory AsmParser be built, modify CMakeLists.txt and LLVMBuild.txt as follows,

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", shamt, CPURegs>;

class LoadAddress<string instr_asm, Operand MemOpnd, RegisterClass RC> :
  Cpu0AsmPseudoInst<(outs RC:$ra), (ins MemOpnd:$addr),
           !strconcat(instr_asm, "\t$ra, $addr")> ;
def LoadAddr32Reg : LoadAddress<"la", mem, CPURegs>;

class LoadAddressImm<string instr_asm, Operand Od, RegisterClass RC> :
  Cpu0AsmPseudoInst<(outs RC:$ra), (ins Od:$imm32),
```

```
          !strconcat(instr_asm, "\t$ra, $imm32")> ;
def LoadAddr32Imm : LoadAddressImm<"la", shamt, CPURegs>;
```

Above define the **ParserMethod = "parseMemOperand"** and implement the parseMemOperand() in Cpu0AsmParser.cpp to handle the **"mem"** operand which used in ld and st. For example, ld $2, 4($sp), the **mem** operand is 4($sp). Accompany with **"let ParserMatchClass = Cpu0MemAsmOperand;"**, LLVM will call parseMemOperand() of Cpu0AsmParser.cpp when it meets the assembly **mem** operand 4($sp). With above **"let"** assignment, TableGen will generate the following structure and functions in Cpu0GenAsmMatcher.inc.

cmake_debug_build/lib/Target/Cpu0/Cpu0GenAsmMatcher.inc

```
  enum OperandMatchResultTy {
    MatchOperand_Success,    // operand matched successfully
    MatchOperand_NoMatch,    // operand did not match
    MatchOperand_ParseFail   // operand matched but had errors
  };
  OperandMatchResultTy MatchOperandParserImpl(
    SmallVectorImpl<MCParsedAsmOperand*> &Operands,
    StringRef Mnemonic);
  OperandMatchResultTy tryCustomParseOperand(
    SmallVectorImpl<MCParsedAsmOperand*> &Operands,
    unsigned MCK);

Cpu0AsmParser::OperandMatchResultTy Cpu0AsmParser::
tryCustomParseOperand(SmallVectorImpl<MCParsedAsmOperand*> &Operands,
            unsigned MCK) {

  switch(MCK) {
  case MCK_Mem:
    return parseMemOperand(Operands);
  default:
    return MatchOperand_NoMatch;
  }
  return MatchOperand_NoMatch;
}

Cpu0AsmParser::OperandMatchResultTy Cpu0AsmParser::
MatchOperandParserImpl(SmallVectorImpl<MCParsedAsmOperand*> &Operands,
             StringRef Mnemonic) {
  ...
}

/// MatchClassKind - The kinds of classes which participate in
/// instruction matching.
enum MatchClassKind {
  ...
  MCK_Mem, // user defined class 'Cpu0MemAsmOperand'
  ...
};
```

Above 3 Pseudo Instruction definitions in Cpu0InstrInfo.td such as LoadImm32Reg are handled by Cpu0AsmParser.cpp as follows,

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, we change registers name to lower case as below since the assembly output and `llvm-objdump -d` using lower case. 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

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 site [1] has a pdf [2] 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. Alough it has not the pipeline features, we can assume the cpu0 backend code run on the pipeline machine because the pipeline version use the same machine instructions. Verilog is C like language in syntex 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,

```
ST :
  if (R[b]+c16 == 28672)
    $display("%4dns %8x : %8x OUTPUT=%-d", $stime, pc0, ir, R[a]);
```

LLVMBackendTutorialExampleCode/cpu0_verilog/raw/cpu0s.v

```
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 ch10_2.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.h

LLVMBackendTutorialExampleCode/InputFiles/ch10_2.cpp

```
JonathantekiiMac:InputFiles Jonathan$ pwd
/Users/Jonathan/test/2/lbd/LLVMBackendTutorialExampleCode/InputFiles
JonathantekiiMac:InputFiles Jonathan$ clang -c ch10_2.cpp -emit-llvm -o
ch10_2.bc
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=obj
ch10_2.bc -o ch10_2.cpu0.o
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llvm-objdump -d ch10_2.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    */
```

[1] http://www.ece.umd.edu/courses/enee359a/

[2] http://www.ece.umd.edu/courses/enee359a/verilog_tutorial.pdf

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/Cpu0MCCodeEmitter.cpp

```
unsigned Cpu0MCCodeEmitter::
getJumpTargetOpValue(const MCInst &MI, unsigned OpNo,
           SmallVectorImpl<MCFixup> &Fixups) const {

  unsigned Opcode = MI.getOpcode();
  ...
  if (Opcode == Cpu0::JSUB)
    Fixups.push_back(MCFixup::Create(0, Expr,
                     MCFixupKind(Cpu0::fixup_Cpu0_PC24)));
  else if (Opcode == Cpu0::JSUB)
    Fixups.push_back(MCFixup::Create(0, Expr,
                     MCFixupKind(Cpu0::fixup_Cpu0_24)));
  else
    llvm_unreachable("unexpect opcode in getJumpAbsoluteTargetOpValue()");

  return 0;
}
```

We change JSUB from Relocation Records fixup_Cpu0_24 to Non-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` again, will get the hex file as follows,

```
JonathantekiiMac:InputFiles Jonathan$ pwd
/Users/Jonathan/test/2/lbd/LLVMBackendTutorialExampleCode/InputFiles
JonathantekiiMac:InputFiles Jonathan$ clang -c ch10_2.cpp -emit-llvm -o
ch10_2.bc
```

```
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=obj
ch10_2.bc -o ch10_2.cpu0.o
JonathantekiiMac:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_
build/bin/Debug/llvm-objdump -d ch10_2.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

118-165-64-234:raw Jonathan$ cat cpu0s.hex
/* 0: */  09 10 00 00 /* addiu  $at, $zero, 0   */
/* 4: */  09 20 00 00 /* addiu  $2, $zero, 0  */
/* 8: */  09 30 00 00 /* addiu  $3, $zero, 0  */
/* c: */  09 40 00 00 /* addiu  $4, $zero, 0  */
/* 10: */ 09 50 00 00 /* addiu  $5, $zero, 0  */
/* 14: */ 09 60 00 00 /* addiu  $6, $zero, 0  */
/* 18: */ 09 70 00 00 /* addiu  $7, $zero, 0  */
/* 1c: */ 09 80 00 00 /* addiu  $8, $zero, 0  */
/* 20: */ 09 90 00 00 /* addiu  $9, $zero, 0  */
/* 24: */ 09 a0 00 00 /* addiu  $gp, $zero, 0   */
/* 28: */ 09 b0 00 00 /* addiu  $fp, $zero, 0   */
/* 2c: */ 09 c0 00 00 /* addiu  $sw, $zero, 0   */
/* 30: */ 09 e0 ff ff /* addiu  $lr, $zero, -1  */
/* 34: */ 09 d0 03 fc /* addiu  $sp, $zero, 1020  */
/* 38: */ 09 dd ff e0 /* addiu  $sp, $sp, -32   */
/* 3c: */ 01 ed 00 1c /* st $lr, 28($sp)    */
/* 40: */ 09 20 00 00 /* addiu  $2, $zero, 0  */
/* 44: */ 01 2d 00 18 /* st $2, 24($sp)   */
/* 48: */ 01 2d 00 14 /* st $2, 20($sp)   */
/* 4c: */ 2b 00 00 34 /* jsub 52    */
/* 50: */ 01 2d 00 14 /* st $2, 20($sp)   */
/* 54: */ 01 2d 00 00 /* st $2, 0($sp)    */
/* 58: */ 2b 00 01 74 /* jsub 372     */
/* 5c: */ 2b 00 01 94 /* jsub 404     */
/* 60: */ 00 3d 00 14 /* ld $3, 20($sp)   */
/* 64: */ 13 23 20 00 /* add  $2, $3, $2  */
/* 68: */ 01 2d 00 14 /* st $2, 20($sp)   */
/* 6c: */ 01 2d 00 00 /* st $2, 0($sp)    */
/* 70: */ 2b 00 01 5c /* jsub 348     */
/* 74: */ 00 2d 00 14 /* ld $2, 20($sp)   */
/* 78: */ 00 ed 00 1c /* ld $lr, 28($sp)    */
/* 7c: */ 09 dd 00 20 /* addiu  $sp, $sp, 32  */
/* 80: */ 2c 00 00 00 /* ret  $zero     */
/* 84: */ 09 dd ff a8 /* addiu  $sp, $sp, -88   */
/* 88: */ 01 ed 00 54 /* st $lr, 84($sp)    */
/* 8c: */ 01 7d 00 50 /* st $7, 80($sp)   */
/* 90: */ 09 20 00 0b /* addiu  $2, $zero, 11   */
/* 94: */ 01 2d 00 4c /* st $2, 76($sp)   */
/* 98: */ 09 20 00 02 /* addiu  $2, $zero, 2  */
/* 9c: */ 01 2d 00 48 /* st $2, 72($sp)   */
/* a0: */ 09 70 00 00 /* addiu  $7, $zero, 0  */
/* a4: */ 01 7d 00 44 /* st $7, 68($sp)   */
/* a8: */ 01 7d 00 40 /* st $7, 64($sp)   */
/* ac: */ 01 7d 00 20 /* st $7, 32($sp)   */
/* b0: */ 09 20 ff fb /* addiu  $2, $zero, -5   */
/* b4: */ 01 2d 00 1c /* st $2, 28($sp)   */
/* b8: */ 01 7d 00 18 /* st $7, 24($sp)   */
/* bc: */ 00 2d 00 48 /* ld $2, 72($sp)   */
/* c0: */ 00 3d 00 4c /* ld $3, 76($sp)   */
```

```
/* c4: */ 13 23 20 00 /* add  $2, $3, $2  */
/* c8: */ 01 2d 00 44 /* st $2, 68($sp)   */
/* cc: */ 00 2d 00 48 /* ld $2, 72($sp)   */
/* d0: */ 00 3d 00 4c /* ld $3, 76($sp)   */
/* d4: */ 14 23 20 00 /* sub  $2, $3, $2  */
/* d8: */ 01 2d 00 40 /* st $2, 64($sp)   */
/* dc: */ 00 2d 00 48 /* ld $2, 72($sp)   */
/* e0: */ 00 3d 00 4c /* ld $3, 76($sp)   */
/* e4: */ 15 23 20 00 /* mul  $2, $3, $2  */
/* e8: */ 01 2d 00 3c /* st $2, 60($sp)   */
/* ec: */ 00 2d 00 48 /* ld $2, 72($sp)   */
/* f0: */ 00 3d 00 4c /* ld $3, 76($sp)   */
/* f4: */ 16 32 00 00 /* div  $3, $2    */
/* f8: */ 41 20 00 00 /* mflo $2    */
/* fc: */ 09 30 2a aa /* addiu  $3, $zero, 10922  */
/* 100: */  1e 33 00 10 /* shl  $3, $3, 16  */
/* 104: */  09 40 aa ab /* addiu  $4, $zero, -21845   */
/* 108: */  19 33 40 00 /* or $3, $3, $4  */
/* 10c: */  01 2d 00 38 /* st $2, 56($sp)   */
/* 110: */  00 2d 00 4c /* ld $2, 76($sp)   */
/* 114: */  09 22 00 01 /* addiu  $2, $2, 1   */
/* 118: */  50 23 00 00 /* mult $2, $3    */
/* 11c: */  40 30 00 00 /* mfhi $3    */
/* 120: */  1f 43 00 1f /* shr  $4, $3, 31  */
/* 124: */  1b 33 00 01 /* sra  $3, $3, 1   */
/* 128: */  13 33 40 00 /* add  $3, $3, $4  */
/* 12c: */  09 40 00 0c /* addiu  $4, $zero, 12   */
/* 130: */  15 33 40 00 /* mul  $3, $3, $4  */
/* 134: */  14 22 30 00 /* sub  $2, $2, $3  */
/* 138: */  01 2d 00 48 /* st $2, 72($sp)   */
/* 13c: */  00 3d 00 4c /* ld $3, 76($sp)   */
/* 140: */  18 23 20 00 /* and  $2, $3, $2  */
/* 144: */  01 2d 00 34 /* st $2, 52($sp)   */
/* 148: */  00 2d 00 48 /* ld $2, 72($sp)   */
/* 14c: */  00 3d 00 4c /* ld $3, 76($sp)   */
/* 150: */  19 23 20 00 /* or $2, $3, $2  */
/* 154: */  01 2d 00 30 /* st $2, 48($sp)   */
/* 158: */  00 2d 00 48 /* ld $2, 72($sp)   */
/* 15c: */  00 3d 00 4c /* ld $3, 76($sp)   */
/* 160: */  1a 23 20 00 /* xor  $2, $3, $2  */
/* 164: */  01 2d 00 2c /* st $2, 44($sp)   */
/* 168: */  00 2d 00 4c /* ld $2, 76($sp)   */
/* 16c: */  1e 22 00 02 /* shl  $2, $2, 2   */
/* 170: */  01 2d 00 28 /* st $2, 40($sp)   */
/* 174: */  00 2d 00 4c /* ld $2, 76($sp)   */
/* 178: */  1b 22 00 02 /* sra  $2, $2, 2   */
/* 17c: */  01 2d 00 24 /* st $2, 36($sp)   */
/* 180: */  01 2d 00 00 /* st $2, 0($sp)    */
/* 184: */  2b 00 00 48 /* jsub 72    */
/* 188: */  00 2d 00 1c /* ld $2, 28($sp)   */
/* 18c: */  1f 22 00 02 /* shr  $2, $2, 2   */
/* 190: */  01 2d 00 18 /* st $2, 24($sp)   */
/* 194: */  01 2d 00 00 /* st $2, 0($sp)    */
/* 198: */  2b 00 00 34 /* jsub 52    */
/* 19c: */  00 2d 00 4c /* ld $2, 76($sp)   */
/* 1a0: */  1a 22 70 00 /* xor  $2, $2, $7  */
/* 1a4: */  09 30 00 01 /* addiu  $3, $zero, 1  */
/* 1a8: */  1a 22 30 00 /* xor  $2, $2, $3  */
```

```
/* 1ac: */  18 22 30 00 /* and  $2, $2, $3  */
/* 1b0: */  01 2d 00 48 /* st $2, 72($sp)   */
/* 1b4: */  09 2d 00 48 /* addiu  $2, $sp, 72   */
/* 1b8: */  01 2d 00 10 /* st $2, 16($sp)   */
/* 1bc: */  00 2d 00 44 /* ld $2, 68($sp)   */
/* 1c0: */  00 7d 00 50 /* ld $7, 80($sp)   */
/* 1c4: */  00 ed 00 54 /* ld $lr, 84($sp)    */
/* 1c8: */  09 dd 00 58 /* addiu  $sp, $sp, 88  */
/* 1cc: */  2c 00 00 00 /* ret  $zero     */
/* 1d0: */  09 dd ff f8 /* addiu  $sp, $sp, -8  */
/* 1d4: */  00 2d 00 08 /* ld $2, 8($sp)    */
/* 1d8: */  01 2d 00 04 /* st $2, 4($sp)    */
/* 1dc: */  09 20 70 00 /* addiu  $2, $zero, 28672  */
/* 1e0: */  01 2d 00 00 /* st $2, 0($sp)    */
/* 1e4: */  00 3d 00 04 /* ld $3, 4($sp)    */
/* 1e8: */  01 32 00 00 /* st $3, 0($2)   */
/* 1ec: */  09 dd 00 08 /* addiu  $sp, $sp, 8   */
/* 1f0: */  2c 00 00 00 /* ret  $zero     */
/* 1f4: */  09 dd ff e8 /* addiu  $sp, $sp, -24   */
/* 1f8: */  09 30 00 01 /* addiu  $3, $zero, 1  */
/* 1fc: */  01 3d 00 14 /* st $3, 20($sp)   */
/* 200: */  09 20 00 02 /* addiu  $2, $zero, 2  */
/* 204: */  01 2d 00 10 /* st $2, 16($sp)   */
/* 208: */  09 20 00 03 /* addiu  $2, $zero, 3  */
/* 20c: */  01 2d 00 0c /* st $2, 12($sp)   */
/* 210: */  09 20 00 04 /* addiu  $2, $zero, 4  */
/* 214: */  01 2d 00 08 /* st $2, 8($sp)    */
/* 218: */  09 20 00 05 /* addiu  $2, $zero, 5  */
/* 21c: */  01 2d 00 04 /* st $2, 4($sp)    */
/* 220: */  09 20 00 00 /* addiu  $2, $zero, 0  */
/* 224: */  00 4d 00 14 /* ld $4, 20($sp)   */
/* 228: */  10 42 00 00 /* cmp  $zero, $4, $2   */
/* 22c: */  20 00 00 10 /* jeq  $zero, 16   */
/* 230: */  26 00 00 00 /* jmp  0     */
/* 234: */  00 4d 00 14 /* ld $4, 20($sp)   */
/* 238: */  09 44 00 01 /* addiu  $4, $4, 1   */
/* 23c: */  01 4d 00 14 /* st $4, 20($sp)   */
/* 240: */  00 4d 00 10 /* ld $4, 16($sp)   */
/* 244: */  10 43 00 00 /* cmp  $zero, $4, $3   */
/* 248: */  22 00 00 10 /* jlt  $zero, 16   */
/* 24c: */  26 00 00 00 /* jmp  0     */
/* 250: */  00 3d 00 10 /* ld $3, 16($sp)   */
/* 254: */  09 33 00 01 /* addiu  $3, $3, 1   */
/* 258: */  01 3d 00 10 /* st $3, 16($sp)   */
/* 25c: */  00 3d 00 0c /* ld $3, 12($sp)   */
/* 260: */  10 32 00 00 /* cmp  $zero, $3, $2   */
/* 264: */  22 00 00 10 /* jlt  $zero, 16   */
/* 268: */  26 00 00 00 /* jmp  0     */
/* 26c: */  00 3d 00 0c /* ld $3, 12($sp)   */
/* 270: */  09 33 00 01 /* addiu  $3, $3, 1   */
/* 274: */  01 3d 00 0c /* st $3, 12($sp)   */
/* 278: */  09 30 ff ff /* addiu  $3, $zero, -1   */
/* 27c: */  00 4d 00 08 /* ld $4, 8($sp)    */
/* 280: */  10 43 00 00 /* cmp  $zero, $4, $3   */
/* 284: */  23 00 00 10 /* jgt  $zero, 16   */
/* 288: */  26 00 00 00 /* jmp  0     */
/* 28c: */  00 3d 00 08 /* ld $3, 8($sp)    */
/* 290: */  09 33 00 01 /* addiu  $3, $3, 1   */
```

```
/* 294: */  01 3d 00 08 /* st $3, 8($sp)    */
/* 298: */  00 3d 00 04 /* ld $3, 4($sp)    */
/* 29c: */  10 32 00 00 /* cmp  $zero, $3, $2   */
/* 2a0: */  23 00 00 10 /* jgt  $zero, 16   */
/* 2a4: */  26 00 00 00 /* jmp  0     */
/* 2a8: */  00 2d 00 04 /* ld $2, 4($sp)    */
/* 2ac: */  09 22 00 01 /* addiu  $2, $2, 1   */
/* 2b0: */  01 2d 00 04 /* st $2, 4($sp)    */
/* 2b4: */  00 2d 00 10 /* ld $2, 16($sp)   */
/* 2b8: */  00 3d 00 14 /* ld $3, 20($sp)   */
/* 2bc: */  13 23 20 00 /* add  $2, $3, $2  */
/* 2c0: */  00 3d 00 0c /* ld $3, 12($sp)   */
/* 2c4: */  13 22 30 00 /* add  $2, $2, $3  */
/* 2c8: */  00 3d 00 08 /* ld $3, 8($sp)    */
/* 2cc: */  13 22 30 00 /* add  $2, $2, $3  */
/* 2d0: */  00 3d 00 04 /* ld $3, 4($sp)    */
/* 2d4: */  13 22 30 00 /* add  $2, $2, $3  */
/* 2d8: */  09 dd 00 18 /* addiu  $sp, $sp, 24  */
/* 2dc: */  2c 00 00 00 /* ret  $zero     */
/* 2e0: */  09 dd ff f8 /* addiu  $sp, $sp, -8  */
/* 2e4: */  00 2d 00 08 /* ld $2, 8($sp)    */
/* 2e8: */  01 2d 00 04 /* st $2, 4($sp)    */
/* 2ec: */  00 1d 00 08 /* ld $at, 8($sp)   */
/* 2f0: */  01 10 70 00 /* st $at, 28672($zero)   */
/* 2f4: */  09 dd 00 08 /* addiu  $sp, $sp, 8   */
/* 2f8: */  2c 00 00 00 /* ret  $zero     */
```

From above result, you can find the print_integer() which implemented by C language has 8 instructions while the print1_integer() which implemented by assembly has 6 instructions. But the C version is better in portability since the assembly is binding with machine assembly language and make the assumption that the stack size of print1_integer() is 8. Now, run the cpu0 backend to get the result as follows,

```
118-165-64-234:raw Jonathan$ ./cpu0s
WARNING: cpu0s.v:219: $readmemh(cpu0s.hex): Not enough words in the file for
the requested range [0:1024].
00000000: 09100000
00000004: 09200000
00000008: 09300000
0000000c: 09400000
00000010: 09500000
00000014: 09600000
00000018: 09700000
0000001c: 09800000
00000020: 09900000
00000024: 09a00000
00000028: 09b00000
0000002c: 09c00000
00000030: 09e0ffff
00000034: 09d003fc
00000038: 09ddffe0
0000003c: 01ed001c
00000040: 09200000
00000044: 012d0018
00000048: 012d0014
0000004c: 2b000034
00000050: 012d0014
00000054: 012d0000
00000058: 2b000174
```

```
0000005c: 2b000194
00000060: 003d0014
00000064: 13232000
00000068: 012d0014
0000006c: 012d0000
00000070: 2b00015c
00000074: 002d0014
00000078: 00ed001c
0000007c: 09dd0020
00000080: 2c000000
00000084: 09ddffa8
00000088: 01ed0054
0000008c: 017d0050
00000090: 0920000b
00000094: 012d004c
00000098: 09200002
0000009c: 012d0048
000000a0: 09700000
000000a4: 017d0044
000000a8: 017d0040
000000ac: 017d0020
000000b0: 0920fffb
000000b4: 012d001c
000000b8: 017d0018
000000bc: 002d0048
000000c0: 003d004c
000000c4: 13232000
000000c8: 012d0044
000000cc: 002d0048
000000d0: 003d004c
000000d4: 14232000
000000d8: 012d0040
000000dc: 002d0048
000000e0: 003d004c
000000e4: 15232000
000000e8: 012d003c
000000ec: 002d0048
000000f0: 003d004c
000000f4: 16320000
000000f8: 41200000
000000fc: 09302aaa
00000100: 1e330010
00000104: 0940aaab
00000108: 19334000
0000010c: 012d0038
00000110: 002d004c
00000114: 09220001
00000118: 50230000
0000011c: 40300000
00000120: 1f43001f
00000124: 1b330001
00000128: 13334000
0000012c: 0940000c
00000130: 15334000
00000134: 14223000
00000138: 012d0048
0000013c: 003d004c
00000140: 18232000
```

```
00000144: 012d0034
00000148: 002d0048
0000014c: 003d004c
00000150: 19232000
00000154: 012d0030
00000158: 002d0048
0000015c: 003d004c
00000160: 1a232000
00000164: 012d002c
00000168: 002d004c
0000016c: 1e220002
00000170: 012d0028
00000174: 002d004c
00000178: 1b220002
0000017c: 012d0024
00000180: 012d0000
00000184: 2b000048
00000188: 002d001c
0000018c: 1f220002
00000190: 012d0018
00000194: 012d0000
00000198: 2b000034
0000019c: 002d004c
000001a0: 1a227000
000001a4: 09300001
000001a8: 1a223000
000001ac: 18223000
000001b0: 012d0048
000001b4: 092d0048
000001b8: 012d0010
000001bc: 002d0044
000001c0: 007d0050
000001c4: 00ed0054
000001c8: 09dd0058
000001cc: 2c000000
000001d0: 09ddfff8
000001d4: 002d0008
000001d8: 012d0004
000001dc: 09207000
000001e0: 012d0000
000001e4: 003d0004
000001e8: 01320000
000001ec: 09dd0008
000001f0: 2c000000
000001f4: 09ddffe8
000001f8: 09300001
000001fc: 013d0014
00000200: 09200002
00000204: 012d0010
00000208: 09200003
0000020c: 012d000c
00000210: 09200004
00000214: 012d0008
00000218: 09200005
0000021c: 012d0004
00000220: 09200000
00000224: 004d0014
00000228: 10420000
```

```
0000022c: 20000010
00000230: 26000000
00000234: 004d0014
00000238: 09440001
0000023c: 014d0014
00000240: 004d0010
00000244: 10430000
00000248: 22000010
0000024c: 26000000
00000250: 003d0010
00000254: 09330001
00000258: 013d0010
0000025c: 003d000c
00000260: 10320000
00000264: 22000010
00000268: 26000000
0000026c: 003d000c
00000270: 09330001
00000274: 013d000c
00000278: 0930ffff
0000027c: 004d0008
00000280: 10430000
00000284: 23000010
00000288: 26000000
0000028c: 003d0008
00000290: 09330001
00000294: 013d0008
00000298: 003d0004
0000029c: 10320000
000002a0: 23000010
000002a4: 26000000
000002a8: 002d0004
000002ac: 09220001
000002b0: 012d0004
000002b4: 002d0010
000002b8: 003d0014
000002bc: 13232000
000002c0: 003d000c
000002c4: 13223000
000002c8: 003d0008
000002cc: 13223000
000002d0: 003d0004
000002d4: 13223000
000002d8: 09dd0018
000002dc: 2c000000
000002e0: 09ddfff8
000002e4: 002d0008
000002e8: 012d0004
000002ec: 001d0008
000002f0: 01107000
000002f4: 09dd0008
000002f8: 2c000000
  90ns 00000000 : 09100000 R[01]=00000000=0          SW=00000000
 170ns 00000004 : 09200000 R[02]=00000000=0          SW=00000000
 250ns 00000008 : 09300000 R[03]=00000000=0          SW=00000000
 330ns 0000000c : 09400000 R[04]=00000000=0          SW=00000000
 410ns 00000010 : 09500000 R[05]=00000000=0          SW=00000000
 490ns 00000014 : 09600000 R[06]=00000000=0          SW=00000000
```

```
 570ns 00000018 : 09700000 R[07]=00000000=0          SW=00000000
 650ns 0000001c : 09800000 R[08]=00000000=0          SW=00000000
 730ns 00000020 : 09900000 R[09]=00000000=0          SW=00000000
 810ns 00000024 : 09a00000 R[10]=00000000=0          SW=00000000
 890ns 00000028 : 09b00000 R[11]=00000000=0          SW=00000000
 970ns 0000002c : 09c00000 R[12]=00000000=0          SW=00000000
1050ns 00000030 : 09e0ffff R[14]=ffffffff=-1         SW=00000000
1130ns 00000034 : 09d003fc R[13]=000003fc=1020       SW=00000000
1210ns 00000038 : 09ddffe0 R[13]=000003dc=988        SW=00000000
1370ns 00000040 : 09200000 R[02]=00000000=0          SW=00000000
1610ns 0000004c : 2b000034 R[00]=00000000=0          SW=00000000
1690ns 00000084 : 09ddffa8 R[13]=00000384=900        SW=00000000
1930ns 00000090 : 0920000b R[02]=0000000b=11         SW=00000000
2090ns 00000098 : 09200002 R[02]=00000002=2          SW=00000000
2250ns 000000a0 : 09700000 R[07]=00000000=0          SW=00000000
2570ns 000000b0 : 0920fffb R[02]=fffffffb=-5         SW=00000000
2810ns 000000bc : 002d0048 R[02]=00000002=2          SW=00000000
2890ns 000000c0 : 003d004c R[03]=0000000b=11         SW=00000000
2970ns 000000c4 : 13232000 R[02]=0000000d=13         SW=00000000
3130ns 000000cc : 002d0048 R[02]=00000002=2          SW=00000000
3210ns 000000d0 : 003d004c R[03]=0000000b=11         SW=00000000
3290ns 000000d4 : 14232000 R[02]=00000009=9          SW=00000000
3450ns 000000dc : 002d0048 R[02]=00000002=2          SW=00000000
3530ns 000000e0 : 003d004c R[03]=0000000b=11         SW=00000000
3610ns 000000e4 : 15232000 R[02]=00000016=22         SW=00000000
3770ns 000000ec : 002d0048 R[02]=00000002=2          SW=00000000
3850ns 000000f0 : 003d004c R[03]=0000000b=11         SW=00000000
3930ns 000000f4 : 16320000 HI=00000001 LO=00000005 SW=00000000
4010ns 000000f8 : 41200000 R[02]=00000005=5          SW=00000000
4090ns 000000fc : 09302aaa R[03]=00002aaa=10922      SW=00000000
4170ns 00000100 : 1e330010 R[03]=2aaa0000=715784192  SW=00000000
4250ns 00000104 : 0940aaab R[04]=ffffaaab=-21845     SW=00000000
4330ns 00000108 : 19334000 R[03]=ffffaaab=-21845     SW=00000000
4490ns 00000110 : 002d004c R[02]=0000000b=11         SW=00000000
4570ns 00000114 : 09220001 R[02]=0000000c=12         SW=00000000
4650ns 00000118 : 50230000 HI=ffffffff LO=fffc0004 SW=00000000
4730ns 0000011c : 40300000 R[03]=ffffffff=-1         SW=00000000
4810ns 00000120 : 1f43001f R[04]=00000001=1          SW=00000000
4890ns 00000124 : 1b330001 R[03]=ffffffff=-1         SW=00000000
4970ns 00000128 : 13334000 R[03]=00000000=0          SW=00000000
5050ns 0000012c : 0940000c R[04]=0000000c=12         SW=00000000
5130ns 00000130 : 15334000 R[03]=00000000=0          SW=00000000
5210ns 00000134 : 14223000 R[02]=0000000c=12         SW=00000000
5370ns 0000013c : 003d004c R[03]=0000000b=11         SW=00000000
5450ns 00000140 : 18232000 R[02]=00000008=8          SW=00000000
5610ns 00000148 : 002d0048 R[02]=0000000c=12         SW=00000000
5690ns 0000014c : 003d004c R[03]=0000000b=11         SW=00000000
5770ns 00000150 : 19232000 R[02]=0000000f=15         SW=00000000
5930ns 00000158 : 002d0048 R[02]=0000000c=12         SW=00000000
6010ns 0000015c : 003d004c R[03]=0000000b=11         SW=00000000
6090ns 00000160 : 1a232000 R[02]=00000007=7          SW=00000000
6250ns 00000168 : 002d004c R[02]=0000000b=11         SW=00000000
6330ns 0000016c : 1e220002 R[02]=0000002c=44         SW=00000000
6490ns 00000174 : 002d004c R[02]=0000000b=11         SW=00000000
6570ns 00000178 : 1b220002 R[02]=00000002=2          SW=00000000
6810ns 00000184 : 2b000048 R[00]=00000000=0          SW=00000000
6890ns 000001d0 : 09ddfff8 R[13]=0000037c=892        SW=00000000
6970ns 000001d4 : 002d0008 R[02]=00000002=2          SW=00000000
```

```
7130ns 000001dc : 09207000 R[02]=00007000=28672      SW=00000000
7290ns 000001e4 : 003d0004 R[03]=00000002=2          SW=00000000
7370ns 000001e8 : 01320000 OUTPUT=2
7450ns 000001ec : 09dd0008 R[13]=00000384=900        SW=00000000
7530ns 000001f0 : 2c000000 R[00]=00000000=0          SW=00000000
7610ns 00000188 : 002d001c R[02]=fffffffb=-5         SW=00000000
7690ns 0000018c : 1f220002 R[02]=3ffffffe=1073741822 SW=00000000
7930ns 00000198 : 2b000034 R[00]=00000000=0          SW=00000000
8010ns 000001d0 : 09ddfff8 R[13]=0000037c=892        SW=00000000
8090ns 000001d4 : 002d0008 R[02]=3ffffffe=1073741822 SW=00000000
8250ns 000001dc : 09207000 R[02]=00007000=28672      SW=00000000
8410ns 000001e4 : 003d0004 R[03]=3ffffffe=1073741822 SW=00000000
8490ns 000001e8 : 01320000 OUTPUT=1073741822
8570ns 000001ec : 09dd0008 R[13]=00000384=900        SW=00000000
8650ns 000001f0 : 2c000000 R[00]=00000000=0          SW=00000000
8730ns 0000019c : 002d004c R[02]=0000000b=11         SW=00000000
8810ns 000001a0 : 1a227000 R[02]=0000000b=11         SW=00000000
8890ns 000001a4 : 09300001 R[03]=00000001=1          SW=00000000
8970ns 000001a8 : 1a223000 R[02]=0000000a=10         SW=00000000
9050ns 000001ac : 18223000 R[02]=00000000=0          SW=00000000
9210ns 000001b4 : 092d0048 R[02]=000003cc=972        SW=00000000
9370ns 000001bc : 002d0044 R[02]=0000000d=13         SW=00000000
9450ns 000001c0 : 007d0050 R[07]=00000000=0          SW=00000000
9530ns 000001c4 : 00ed0054 R[14]=00000050=80         SW=00000000
9610ns 000001c8 : 09dd0058 R[13]=000003dc=988        SW=00000000
9690ns 000001cc : 2c000000 R[00]=00000000=0          SW=00000000
9930ns 00000058 : 2b000174 R[00]=00000000=0          SW=00000000
10010ns 000001d0 : 09ddfff8 R[13]=000003d4=980        SW=00000000
10090ns 000001d4 : 002d0008 R[02]=0000000d=13         SW=00000000
10250ns 000001dc : 09207000 R[02]=00007000=28672      SW=00000000
10410ns 000001e4 : 003d0004 R[03]=0000000d=13         SW=00000000
10490ns 000001e8 : 01320000 OUTPUT=13
10570ns 000001ec : 09dd0008 R[13]=000003dc=988        SW=00000000
10650ns 000001f0 : 2c000000 R[00]=00000000=0          SW=00000000
10730ns 0000005c : 2b000194 R[00]=00000000=0          SW=00000000
10810ns 000001f4 : 09ddffe8 R[13]=000003c4=964        SW=00000000
10890ns 000001f8 : 09300001 R[03]=00000001=1          SW=00000000
11050ns 00000200 : 09200002 R[02]=00000002=2          SW=00000000
11210ns 00000208 : 09200003 R[02]=00000003=3          SW=00000000
11370ns 00000210 : 09200004 R[02]=00000004=4          SW=00000000
11530ns 00000218 : 09200005 R[02]=00000005=5          SW=00000000
11690ns 00000220 : 09200000 R[02]=00000000=0          SW=00000000
11770ns 00000224 : 004d0014 R[04]=00000001=1          SW=00000000
11850ns 00000228 : 10420000 R[04]=00000001=1          SW=00000000
11930ns 0000022c : 20000010 R[00]=00000000=0          SW=00000000
12010ns 00000230 : 26000000 R[00]=00000000=0          SW=00000000
12090ns 00000234 : 004d0014 R[04]=00000001=1          SW=00000000
12170ns 00000238 : 09440001 R[04]=00000002=2          SW=00000000
12330ns 00000240 : 004d0010 R[04]=00000002=2          SW=00000000
12410ns 00000244 : 10430000 R[04]=00000002=2          SW=00000000
12490ns 00000248 : 22000010 R[00]=00000000=0          SW=00000000
12570ns 0000024c : 26000000 R[00]=00000000=0          SW=00000000
12650ns 00000250 : 003d0010 R[03]=00000002=2          SW=00000000
12730ns 00000254 : 09330001 R[03]=00000003=3          SW=00000000
12890ns 0000025c : 003d000c R[03]=00000003=3          SW=00000000
12970ns 00000260 : 10320000 R[03]=00000003=3          SW=00000000
13050ns 00000264 : 22000010 R[00]=00000000=0          SW=00000000
13130ns 00000268 : 26000000 R[00]=00000000=0          SW=00000000
```

```
13210ns 0000026c : 003d000c R[03]=00000003=3          SW=00000000
13290ns 00000270 : 09330001 R[03]=00000004=4          SW=00000000
13450ns 00000278 : 0930ffff R[03]=ffffffff=-1         SW=00000000
13530ns 0000027c : 004d0008 R[04]=00000004=4          SW=00000000
13610ns 00000280 : 10430000 R[04]=00000004=4          SW=00000000
13690ns 00000284 : 23000010 R[00]=00000000=0          SW=00000000
13770ns 00000298 : 003d0004 R[03]=00000005=5          SW=00000000
13850ns 0000029c : 10320000 R[03]=00000005=5          SW=00000000
13930ns 000002a0 : 23000010 R[00]=00000000=0          SW=00000000
14010ns 000002b4 : 002d0010 R[02]=00000003=3          SW=00000000
14090ns 000002b8 : 003d0014 R[03]=00000002=2          SW=00000000
14170ns 000002bc : 13232000 R[02]=00000005=5          SW=00000000
14250ns 000002c0 : 003d000c R[03]=00000004=4          SW=00000000
14330ns 000002c4 : 13223000 R[02]=00000009=9          SW=00000000
14410ns 000002c8 : 003d0008 R[03]=00000004=4          SW=00000000
14490ns 000002cc : 13223000 R[02]=0000000d=13         SW=00000000
14570ns 000002d0 : 003d0004 R[03]=00000005=5          SW=00000000
14650ns 000002d4 : 13223000 R[02]=00000012=18         SW=00000000
14730ns 000002d8 : 09dd0018 R[13]=000003dc=988        SW=00000000
14810ns 000002dc : 2c000000 R[00]=00000000=0          SW=00000000
14890ns 00000060 : 003d0014 R[03]=0000000d=13         SW=00000000
14970ns 00000064 : 13232000 R[02]=0000001f=31         SW=00000000
15210ns 00000070 : 2b00015c R[00]=00000000=0          SW=00000000
15290ns 000001d0 : 09ddfff8 R[13]=000003d4=980        SW=00000000
15370ns 000001d4 : 002d0008 R[02]=0000001f=31         SW=00000000
15530ns 000001dc : 09207000 R[02]=00007000=28672      SW=00000000
15690ns 000001e4 : 003d0004 R[03]=0000001f=31         SW=00000000
15770ns 000001e8 : 01320000 OUTPUT=31
15850ns 000001ec : 09dd0008 R[13]=000003dc=988        SW=00000000
15930ns 000001f0 : 2c000000 R[00]=00000000=0          SW=00000000
16010ns 00000074 : 002d0014 R[02]=0000001f=31         SW=00000000
16090ns 00000078 : 00ed001c R[14]=ffffffff=-1         SW=00000000
16170ns 0000007c : 09dd0020 R[13]=000003fc=1020       SW=00000000
16250ns 00000080 : 2c000000 R[00]=00000000=0          SW=00000000
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(). You can verify it with the **OUTPUT=xxx** in Verilog output.

Now, let’s run ch_10_3.cpp to verify the result as follows,

LLVMBackendTutorialExampleCode/InputFiles/ch10_3.cpp

```
118-165-75-175:InputFiles Jonathan$ clang -target `llvm-config --host-target`
-c ch10_3.cpp -emit-llvm -o ch10_3.bc
118-165-75-175:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build
/bin/Debug/llc -march=cpu0 -relocation-model=static -filetype=obj ch10_3.bc
-o ch10_3.cpu0.o
118-165-75-175:InputFiles Jonathan$ /Users/Jonathan/llvm/test/cmake_debug_build
/bin/Debug/llvm-objdump -d ch10_3.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

118-165-75-175:raw Jonathan$ ./cpu0s
```

```
...
12890ns 0000012c : 01320000 OUTPUT=15
...
```

We show Verilog PC output by display the I/O memory mapped address but we didn't implement the output hardware interface or port. The output hardware interface/port is dependent on hardware output device, 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.

CHAPTER

ELEVEN

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

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

```
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
 ...

118-165-78-10:InputFiles Jonathan$ cat ch11_1.cpu0.s
      .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   $3, $zero, 0
      st      $3, 12($sp)
      st      $3, 8($sp)
      addiu   $2, $zero, 1
      st      $2, 4($sp)
      addiu   $4, $zero, 2
      st      $4, 0($sp)
      ld      $4, 8($sp)
      cmp     $sw, $4, $3
      jne     $sw, $BB0_2
```

```
# BB#1:
	ld	$4, 8($sp)
	addiu	$4, $4, 1
	st	$4, 8($sp)
$BB0_2:
	ld	$4, 4($sp)
	cmp	$sw, $4, $3
	jne	$sw, $BB0_4
	jmp	$BB0_3
$BB0_4:
	addiu	$3, $zero, -1
	ld	$4, 4($sp)
	cmp	$sw, $4, $3
	jgt	$sw, $BB0_6
	jmp	$BB0_5
$BB0_3:
	ld	$3, 4($sp)
	ld	$4, 8($sp)
	add	$3, $4, $3
	st	$3, 8($sp)
	jmp	$BB0_6
$BB0_5:
	ld	$3, 8($sp)
	addiu	$4, $3, -1
	st	$4, 8($sp)
	st	$3, 8($sp)
$BB0_6:
	ld	$3, 0($sp)
	cmp	$sw, $3, $2
	jlt	$sw, $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),

Table 11.1: Cpu0 Instruction Set

Format	Mnemonic	Opcode	Meaning	Syntax	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	SLTi	0A	Set less Then	SLTi Ra, Rb, Cx	Ra <= (Rb < Cx)
L	SLTiu	0B	SLTi unsigned	SLTiu 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||0x0000)
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	15	Multiply	MUL Ra, Rb, Rc	Ra <= Rb * Rc
A	DIV	16	Divide	DIV Ra, Rb	HI<=Ra%Rb, LO<=Ra/Rb
A	DIVu	16	Div unsigned	DIVu Ra, Rb	HI<=Ra%Rb, LO<=Ra/Rb
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	1C	Rotate left	ROL Ra, Rb, Cx	Ra <= Rb rol Cx
A	ROR	1D	Rotate right	ROR Ra, Rb, Cx	Ra <= Rb ror Cx
A	SHL	1E	Shift left	SHL Ra, Rb, Cx	Ra <= Rb << Cx
A	SHR	1F	Shift right	SHR Ra, Rb, Cx	Ra <= Rb >> Cx
A	SLT	20	Set less Then	SLT Ra, Rb, Rc	Ra <= (Rb < Rc)
A	SLT	21	SLT unsigned	SLTu Ra, Rb, Rc	Ra <= (Rb < Rc)
L	MFHI	22	Move HI to GPR	MFHI Ra	Ra <= HI
L	MFLO	23	Move LO to GPR	MFLO Ra	Ra <= LO
L	MTHI	24	Move GPR to HI	MTHI Ra	HI <= Ra
L	MTLO	25	Move GPR to LO	MTLO Ra	LO <= Ra
L	MULT	26	Multiply for 64 bits result	MULT Ra, Rb	(HI,LO) <= MULT(Ra,Rb)

Continued on next page

Table 11.1 – continued from previous page

Format	Mnemonic	Opcode	Meaning	Syntax	Operation
L	MULTU	27	MULT for unsigned 64 bits	MULTU Ra, Rb	(HI,LO) <= MULTU(Ra,Rb)
J	JMP	26	Jump (unconditional)	JMP Cx	PC <= PC + Cx
L	BEQ	27	Jump if equal	BEQ Ra, Rb, Cx	if (Ra==Rb), PC <= PC + Cx
L	BNE	28	Jump if not equal	BNE Ra, Rb, Cx	if (Ra!=Rb), PC <= PC + Cx
J	SWI	2A	Software interrupt	SWI Cx	LR <= PC; PC <= Cx
J	JSUB	2B	Jump to subroutine	JSUB Cx	LR <= PC; PC <= PC + Cx
J	RET	2C	Return from subroutine	RET Cx	PC <= LR
J	IRET	2D	Return from interrupt handler	IRET	PC <= LR; INT 0
J	JR	2E	Jump to subroutine	JR Rb	LR <= PC; PC <= Rb

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
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);
  }
}

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;
}
```

LLVMBackendTutorialExampleCode/Chapter11_2/Cpu0InstrInfo.td

```
// 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>;
...
/// Load and Store Instructions
///  aligned
```

```
defm LD     : LoadM32<0x01,  "ld",  load_a>;
defm ST     : StoreM32<0x02, "st",  store_a>;

/// Arithmetic Instructions (ALU Immediate)
// add defined in include/llvm/Target/TargetSelectionDAG.td, line 315 (def add).
def ADDiu   : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;
def SLTi    : SetCC_I<0x0a, "slti", setlt, simm16, immSExt16, CPURegs>;
def SLTiu   : SetCC_I<0x0b, "sltiu", setult, simm16, immSExt16, CPURegs>;
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>;
def LUi     : LoadUpper<0x0f, "lui", CPURegs, uimm16>;

/// Arithmetic Instructions (3-Operand, R-Type)
def ADDu    : ArithLogicR<0x11, "addu", add, IIAlu, CPURegs, 1>;
def SUBu    : ArithLogicR<0x12, "subu", sub, IIAlu, CPURegs>;
def ADD     : ArithOverflowR<0x13, "add", IIAlu, CPURegs, 1>;
def SUB     : ArithOverflowR<0x14, "sub", IIAlu, CPURegs>;
def MUL     : ArithLogicR<0x15, "mul", mul, IIImul, CPURegs, 1>;
def DIV     : Div32<Cpu0DivRem, 0x16, "div", IIIdiv>;
def DIVu    : Div32<Cpu0DivRemU, 0x17, "divu", IIIdiv>;
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 SLT     : SetCC_R<0x20, "slt", setlt, CPURegs>;
def SLTu    : SetCC_R<0x21, "sltu", setult, CPURegs>;

def MFHI    : MoveFromLOHI<0x22, "mfhi", CPURegs, [HI]>;
def MFLO    : MoveFromLOHI<0x23, "mflo", CPURegs, [LO]>;
def MTHI    : MoveToLOHI<0x24, "mthi", CPURegs, [HI]>;
def MTLO    : MoveToLOHI<0x25, "mtlo", CPURegs, [LO]>;

def MULT    : Mult32<0x26, "mult", IIImul>;
def MULTu   : Mult32<0x27, "multu", IIImul>;
...

/// Jump and Branch Instructions
def BEQ     : CBranch<0x27, "beq", seteq, CPURegs>;
def BNE     : CBranch<0x28, "bne", setne, CPURegs>;

def JMP     : UncondBranch<0x26, "jmp">;

/// Jump and Branch Instructions
def SWI     : JumpLink<0x2A, "swi">;
def JSUB    : JumpLink<0x2B, "jsub">;
def JR      : JumpFR<0x2C, "ret", CPURegs>;

let isReturn=1, isTerminator=1, hasDelaySlot=1, isCodeGenOnly=1,
    isBarrier=1, hasCtrlDep=1, addr=0 in
  def RET   : FJ <0x2C, (outs), (ins CPURegs:$target),
                "ret\t$target", [(Cpu0Ret CPURegs:$target)], IIBranch>;

def IRET    : JumpFR<0x2D, "iret", CPURegs>;
def JALR    : JumpLinkReg<0x2E, "jalr", CPURegs>;

/// No operation
let addr=0 in
```

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

// FrameIndexes are legalized when they are operands from load/store
// instructions. The same not happens for stack address copies, so an
// add op with mem ComplexPattern is used and the stack address copy
// can be matched. It's similar to Sparc LEA_ADDRi
def LEA_ADDiu : EffectiveAddress<"addiu\t$ra, $addr", CPURegs, mem_ea> {
  let isCodeGenOnly = 1;
}

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

// 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))>;
...

// gp_rel relocs
...
def : Pat<(not CPURegs:$in),
          (XORi CPURegs:$in, 1)>;


// 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> {
  def : Pat<(seteq RC:$lhs, RC:$rhs),
              (SLTiuOp (XOROp RC:$lhs, RC:$rhs), 1)>;
  def : Pat<(setne RC:$lhs, RC:$rhs),
              (SLTuOp ZEROReg, (XOROp RC:$lhs, RC:$rhs))>;
}

multiclass SetlePats<RegisterClass RC, Instruction SLTOp, Instruction SLTuOp> {
  def : Pat<(setle RC:$lhs, RC:$rhs),
              (XORi (SLTOp RC:$rhs, RC:$lhs), 1)>;
  def : Pat<(setule RC:$lhs, RC:$rhs),
              (XORi (SLTuOp RC:$rhs, RC:$lhs), 1)>;
}

multiclass SetgtPats<RegisterClass RC, Instruction SLTOp, Instruction SLTuOp> {
  def : Pat<(setgt RC:$lhs, RC:$rhs),
              (SLTOp RC:$rhs, RC:$lhs)>;
  def : Pat<(setugt RC:$lhs, RC:$rhs),
              (SLTuOp RC:$rhs, RC:$lhs)>;
}

multiclass SetgePats<RegisterClass RC, Instruction SLTOp, Instruction SLTuOp> {
  def : Pat<(setge RC:$lhs, RC:$rhs),
              (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 SLTiuOp> {
  def : Pat<(setge RC:$lhs, immSExt16:$rhs),
              (XORi (SLTiOp RC:$lhs, immSExt16:$rhs), 1)>;
  def : Pat<(setuge RC:$lhs, immSExt16:$rhs),
              (XORi (SLTiuOp RC:$lhs, immSExt16:$rhs), 1)>;
}

defm : SeteqPats<CPURegs, SLTiu, XOR, SLTu, ZERO>;
defm : SetlePats<CPURegs, SLT, SLTu>;
defm : SetgtPats<CPURegs, SLT, SLTu>;
defm : SetgePats<CPURegs, SLT, SLTu>;
defm : SetgeImmPats<CPURegs, SLTi, SLTiu>;
```

LLVMBackendTutorialExampleCode/Chapter11_2/Cpu0MCInstLower.cpp

```
/ Lower ".cpload $reg" to
//  "lui   $gp, %hi(_gp_disp)"
//  "addiu $gp, $gp, %lo(_gp_disp)"
//  "addu  $gp, $gp, $t9"
void Cpu0MCInstLower::LowerCPLOAD(SmallVector<MCInst, 4>& MCInsts) {
  ...
  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);
  ...
}

// Lower ".cprestore offset" to "st $gp, offset($sp)".
void Cpu0MCInstLower::LowerCPRESTORE(int64_t Offset,
                                     SmallVector<MCInst, 4>& MCInsts) {
    ...
    // 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);
  }
```

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. And more, replace “ADDiu, SHL 16” with the efficient LUi instruction.

11.2.3 Cpu0 Verilog language changes

LLVMBackendTutorialExampleCode/cpu0_verilog/redesign/cpu0s.v

11.2.4 Run the redesigned Cpu0

Run Chapter11_2/ with ch11_2.cpp to get result as below. It match the expect value as comment in ch11_2.cpp.

LLVMBackendTutorialExampleCode/InputFiles/ch11_2.cpp

```
118-165-77-203:InputFiles Jonathan$ clang -target `llvm-config --host-target`
-c ch11_2.cpp -emit-llvm -o ch11_2.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
ch11_2.bc -o ch11_2.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 ch11_2.cpu0.o | tail -n +6| awk '{print "/* " $1
" */\t" $2 " " $3 " " $4 " " $5 "\t/* " $6"\t" $7" " $8" " $9" " $10 "\t

118-165-77-203:redesign Jonathan$ ./cpu0s
WARNING: cpu0s.v:227: $readmemh(cpu0s.hex): Not enough words in the file for
the requested range [0:1536].
00000000: 09100000
00000004: 09200000
00000008: 09300000
0000000c: 09400000
00000010: 09500000
00000014: 09600000
00000018: 09700000
0000001c: 09800000
00000020: 09900000
00000024: 09a00000
00000028: 09b00000
0000002c: 09c00000
00000030: 09e0ffff
00000034: 09d005fc
00000038: 09ddffe0
0000003c: 02ed001c
00000040: 09200000
00000044: 022d0018
00000048: 022d0014
0000004c: 2b000038
00000050: 022d0014
00000054: 022d0000
00000058: 2b000190
0000005c: 2b0001b0
00000060: 013d0014
00000064: 11232000
00000068: 022d0014
0000006c: 022d0000
00000070: 2b000178
00000074: 2b00026c
00000078: 012d0014
0000007c: 01ed001c
00000080: 09dd0020
00000084: 2c000000
00000088: 09ddffa0
0000008c: 02ed005c
00000090: 027d0058
00000094: 0920000b
00000098: 022d0054
0000009c: 09200002
000000a0: 022d0050
000000a4: 09700000
000000a8: 027d004c
000000ac: 027d0048
000000b0: 027d0028
```

```
000000b4: 0920fffb
000000b8: 022d0024
000000bc: 027d0020
000000c0: 0f20f000
000000c4: 0d220001
000000c8: 022d001c
000000cc: 0f20000f
000000d0: 0d22ffff
000000d4: 022d0018
000000d8: 013d001c
000000dc: 11232000
000000e0: 022d0024
000000e4: 012d0050
000000e8: 013d0054
000000ec: 11232000
000000f0: 022d004c
000000f4: 012d0050
000000f8: 013d0054
000000fc: 12232000
00000100: 022d0048
00000104: 012d0050
00000108: 013d0054
0000010c: 15232000
00000110: 022d0044
00000114: 012d0050
00000118: 013d0054
0000011c: 16320000
00000120: 23200000
00000124: 022d0040
00000128: 0f202aaa
0000012c: 0d32aaab
00000130: 012d0054
00000134: 09220001
00000138: 26230000
0000013c: 22300000
00000140: 1f43001f
00000144: 1b330001
00000148: 11334000
0000014c: 0940000c
00000150: 15334000
00000154: 12223000
00000158: 022d0050
0000015c: 013d0054
00000160: 18232000
00000164: 022d003c
00000168: 012d0050
0000016c: 013d0054
00000170: 19232000
00000174: 022d0038
00000178: 012d0050
0000017c: 013d0054
00000180: 1a232000
00000184: 022d0034
00000188: 012d0054
0000018c: 1e220002
00000190: 022d0030
00000194: 012d0054
00000198: 1b220002
```

```
0000019c: 022d002c
000001a0: 022d0000
000001a4: 2b000044
000001a8: 012d0024
000001ac: 1f220002
000001b0: 022d0020
000001b4: 022d0000
000001b8: 2b000030
000001bc: 012d0054
000001c0: 1a227000
000001c4: 0b220001
000001c8: 0c220001
000001cc: 022d0050
000001d0: 092d0050
000001d4: 022d0014
000001d8: 012d004c
000001dc: 017d0058
000001e0: 01ed005c
000001e4: 09dd0060
000001e8: 2c000000
000001ec: 09ddfff8
000001f0: 012d0008
000001f4: 022d0004
000001f8: 09207000
000001fc: 022d0000
00000200: 013d0004
00000204: 02320000
00000208: 09dd0008
0000020c: 2c000000
00000210: 09ddffe8
00000214: 09200001
00000218: 022d0014
0000021c: 09200002
00000220: 022d0010
00000224: 09200003
00000228: 022d000c
0000022c: 09200004
00000230: 022d0008
00000234: 09200005
00000238: 022d0004
0000023c: 012d0014
00000240: 2720000c
00000244: 012d0014
00000248: 09220001
0000024c: 022d0014
00000250: 012d0010
00000254: 0a220001
00000258: 2820000c
0000025c: 012d0010
00000260: 09220001
00000264: 022d0010
00000268: 012d000c
0000026c: 0a220000
00000270: 2820000c
00000274: 012d000c
00000278: 09220001
0000027c: 022d000c
00000280: 012d0008
```

```
00000284: 0930ffff
00000288: 20232000
0000028c: 2820000c
00000290: 012d0008
00000294: 09220001
00000298: 022d0008
0000029c: 012d0004
000002a0: 09300000
000002a4: 20232000
000002a8: 2820000c
000002ac: 012d0004
000002b0: 09220001
000002b4: 022d0004
000002b8: 012d0010
000002bc: 013d0014
000002c0: 11232000
000002c4: 013d000c
000002c8: 11223000
000002cc: 013d0008
000002d0: 11223000
000002d4: 013d0004
000002d8: 11223000
000002dc: 09dd0018
000002e0: 2c000000
000002e4: 09ddfff4
000002e8: 022d0008
000002ec: 023d0004
000002f0: 024d0000
000002f4: 0f207fff
000002f8: 0f301000
000002fc: 11423000
00000300: 0f207fff
00000304: 0f301000
00000308: 13423000
0000030c: 0f208fff
00000310: 0f307000
00000314: 14423000
00000318: 0f200000
0000031c: 0930ffff
00000320: 14423000
00000324: 0f20ffff
00000328: 0d22ffff
0000032c: 0c22ffff
00000330: 1e220010
00000334: 0e22ffff
00000338: 0930ffff
0000033c: 17230000
00000340: 16230000
00000344: 0e220001
00000348: 1c421004
0000034c: 1d421008
00000350: 012d0008
00000354: 013d0004
00000358: 014d0000
0000035c: 09dd000c
00000360: 2c000000
  90ns 00000000 : 09100000 R[01]=00000000=0          SW=00000000
 170ns 00000004 : 09200000 R[02]=00000000=0          SW=00000000
```

```
 250ns 00000008 : 09300000 R[03]=00000000=0          SW=00000000
 330ns 0000000c : 09400000 R[04]=00000000=0          SW=00000000
 410ns 00000010 : 09500000 R[05]=00000000=0          SW=00000000
 490ns 00000014 : 09600000 R[06]=00000000=0          SW=00000000
 570ns 00000018 : 09700000 R[07]=00000000=0          SW=00000000
 650ns 0000001c : 09800000 R[08]=00000000=0          SW=00000000
 730ns 00000020 : 09900000 R[09]=00000000=0          SW=00000000
 810ns 00000024 : 09a00000 R[10]=00000000=0          SW=00000000
 890ns 00000028 : 09b00000 R[11]=00000000=0          SW=00000000
 970ns 0000002c : 09c00000 R[12]=00000000=0          SW=00000000
1050ns 00000030 : 09e0ffff R[14]=ffffffff=-1         SW=00000000
1130ns 00000034 : 09d005fc R[13]=000005fc=1532       SW=00000000
1210ns 00000038 : 09ddffe0 R[13]=000005dc=1500       SW=00000000
1290ns 0000003c : 02ed001c m[1500+28  ]=-1           SW=00000000
1370ns 00000040 : 09200000 R[02]=00000000=0          SW=00000000
1450ns 00000044 : 022d0018 m[1500+24  ]=0            SW=00000000
1530ns 00000048 : 022d0014 m[1500+20  ]=0            SW=00000000
1610ns 0000004c : 2b000038 R[00]=00000000=0          SW=00000000
1690ns 00000088 : 09ddffa0 R[13]=0000057c=1404       SW=00000000
1770ns 0000008c : 02ed005c m[1404+92  ]=80           SW=00000000
1850ns 00000090 : 027d0058 m[1404+88  ]=0            SW=00000000
1930ns 00000094 : 0920000b R[02]=0000000b=11         SW=00000000
2010ns 00000098 : 022d0054 m[1404+84  ]=11           SW=00000000
2090ns 0000009c : 09200002 R[02]=00000002=2          SW=00000000
2170ns 000000a0 : 022d0050 m[1404+80  ]=2            SW=00000000
2250ns 000000a4 : 09700000 R[07]=00000000=0          SW=00000000
2330ns 000000a8 : 027d004c m[1404+76  ]=0            SW=00000000
2410ns 000000ac : 027d0048 m[1404+72  ]=0            SW=00000000
2490ns 000000b0 : 027d0028 m[1404+40  ]=0            SW=00000000
2570ns 000000b4 : 0920fffb R[02]=fffffffb=-5         SW=00000000
2650ns 000000b8 : 022d0024 m[1404+36  ]=-5           SW=00000000
2730ns 000000bc : 027d0020 m[1404+32  ]=0            SW=00000000
2810ns 000000c0 : 0f20f000 R[02]=f0000000=-268435456 SW=00000000
2890ns 000000c4 : 0d220001 R[02]=f0000001=-268435455 SW=00000000
2970ns 000000c8 : 022d001c m[1404+28  ]=-268435455   SW=00000000
3050ns 000000cc : 0f20000f R[02]=000f0000=983040     SW=00000000
3130ns 000000d0 : 0d22ffff R[02]=000fffff=1048575    SW=00000000
3210ns 000000d4 : 022d0018 m[1404+24  ]=1048575      SW=00000000
3290ns 000000d8 : 013d001c R[03]=f0000001=-268435455 SW=00000000
3370ns 000000dc : 11232000 R[02]=f0100000=-267386880 SW=00000000
3450ns 000000e0 : 022d0024 m[1404+36  ]=-267386880   SW=00000000
3530ns 000000e4 : 012d0050 R[02]=00000002=2          SW=00000000
3610ns 000000e8 : 013d0054 R[03]=0000000b=11         SW=00000000
3690ns 000000ec : 11232000 R[02]=0000000d=13         SW=00000000
3770ns 000000f0 : 022d004c m[1404+76  ]=13           SW=00000000
3850ns 000000f4 : 012d0050 R[02]=00000002=2          SW=00000000
3930ns 000000f8 : 013d0054 R[03]=0000000b=11         SW=00000000
4010ns 000000fc : 12232000 R[02]=00000009=9          SW=00000000
4090ns 00000100 : 022d0048 m[1404+72  ]=9            SW=00000000
4170ns 00000104 : 012d0050 R[02]=00000002=2          SW=00000000
4250ns 00000108 : 013d0054 R[03]=0000000b=11         SW=00000000
4330ns 0000010c : 15232000 R[02]=00000016=22         SW=00000000
4410ns 00000110 : 022d0044 m[1404+68  ]=22           SW=00000000
4490ns 00000114 : 012d0050 R[02]=00000002=2          SW=00000000
4570ns 00000118 : 013d0054 R[03]=0000000b=11         SW=00000000
4650ns 0000011c : 16320000 HI=00000001 LO=00000005 SW=00000000
4730ns 00000120 : 23200000 R[02]=00000005=5          SW=00000000
4810ns 00000124 : 022d0040 m[1404+64  ]=5            SW=00000000
```

```
4890ns 00000128 : 0f202aaa R[02]=2aaa0000=715784192  SW=00000000
4970ns 0000012c : 0d32aaab R[03]=2aaaaaab=715827883  SW=00000000
5050ns 00000130 : 012d0054 R[02]=0000000b=11         SW=00000000
5130ns 00000134 : 09220001 R[02]=0000000c=12         SW=00000000
5210ns 00000138 : 26230000 HI=00000002 LO=00000004 SW=00000000
5290ns 0000013c : 22300000 R[03]=00000002=2          SW=00000000
5370ns 00000140 : 1f43001f R[04]=00000000=0          SW=00000000
5450ns 00000144 : 1b330001 R[03]=00000001=1          SW=00000000
5530ns 00000148 : 11334000 R[03]=00000001=1          SW=00000000
5610ns 0000014c : 0940000c R[04]=0000000c=12         SW=00000000
5690ns 00000150 : 15334000 R[03]=0000000c=12         SW=00000000
5770ns 00000154 : 12223000 R[02]=00000000=0          SW=00000000
5850ns 00000158 : 022d0050 m[1404+80  ]=0            SW=00000000
5930ns 0000015c : 013d0054 R[03]=0000000b=11         SW=00000000
6010ns 00000160 : 18232000 R[02]=00000000=0          SW=00000000
6090ns 00000164 : 022d003c m[1404+60  ]=0            SW=00000000
6170ns 00000168 : 012d0050 R[02]=00000000=0          SW=00000000
6250ns 0000016c : 013d0054 R[03]=0000000b=11         SW=00000000
6330ns 00000170 : 19232000 R[02]=0000000b=11         SW=00000000
6410ns 00000174 : 022d0038 m[1404+56  ]=11           SW=00000000
6490ns 00000178 : 012d0050 R[02]=00000000=0          SW=00000000
6570ns 0000017c : 013d0054 R[03]=0000000b=11         SW=00000000
6650ns 00000180 : 1a232000 R[02]=0000000b=11         SW=00000000
6730ns 00000184 : 022d0034 m[1404+52  ]=11           SW=00000000
6810ns 00000188 : 012d0054 R[02]=0000000b=11         SW=00000000
6890ns 0000018c : 1e220002 R[02]=0000002c=44         SW=00000000
6970ns 00000190 : 022d0030 m[1404+48  ]=44           SW=00000000
7050ns 00000194 : 012d0054 R[02]=0000000b=11         SW=00000000
7130ns 00000198 : 1b220002 R[02]=00000002=2          SW=00000000
7210ns 0000019c : 022d002c m[1404+44  ]=2            SW=00000000
7290ns 000001a0 : 022d0000 m[1404+0   ]=2            SW=00000000
7370ns 000001a4 : 2b000044 R[00]=00000000=0          SW=00000000
7450ns 000001ec : 09ddfff8 R[13]=00000574=1396       SW=00000000
7530ns 000001f0 : 012d0008 R[02]=00000002=2          SW=00000000
7610ns 000001f4 : 022d0004 m[1396+4   ]=2            SW=00000000
7690ns 000001f8 : 09207000 R[02]=00007000=28672      SW=00000000
7770ns 000001fc : 022d0000 m[1396+0   ]=28672        SW=00000000
7850ns 00000200 : 013d0004 R[03]=00000002=2          SW=00000000
7930ns 00000204 : 02320000 OUTPUT=2
8010ns 00000208 : 09dd0008 R[13]=0000057c=1404       SW=00000000
8090ns 0000020c : 2c000000 R[00]=00000000=0          SW=00000000
8170ns 000001a8 : 012d0024 R[02]=f0100000=-267386880 SW=00000000
8250ns 000001ac : 1f220002 R[02]=3c040000=1006895104 SW=00000000
8330ns 000001b0 : 022d0020 m[1404+32  ]=1006895104   SW=00000000
8410ns 000001b4 : 022d0000 m[1404+0   ]=1006895104   SW=00000000
8490ns 000001b8 : 2b000030 R[00]=00000000=0          SW=00000000
8570ns 000001ec : 09ddfff8 R[13]=00000574=1396       SW=00000000
8650ns 000001f0 : 012d0008 R[02]=3c040000=1006895104 SW=00000000
8730ns 000001f4 : 022d0004 m[1396+4   ]=1006895104   SW=00000000
8810ns 000001f8 : 09207000 R[02]=00007000=28672      SW=00000000
8890ns 000001fc : 022d0000 m[1396+0   ]=28672        SW=00000000
8970ns 00000200 : 013d0004 R[03]=3c040000=1006895104 SW=00000000
9050ns 00000204 : 02320000 OUTPUT=1006895104
9130ns 00000208 : 09dd0008 R[13]=0000057c=1404       SW=00000000
9210ns 0000020c : 2c000000 R[00]=00000000=0          SW=00000000
9290ns 000001bc : 012d0054 R[02]=0000000b=11         SW=00000000
9370ns 000001c0 : 1a227000 R[02]=0000000b=11         SW=00000000
9450ns 000001c4 : 0b220001 R[02]=00000000=0          SW=00000000
```

```
9530ns 000001c8 : 0c220001 R[02]=00000000=0          SW=00000000
9610ns 000001cc : 022d0050 m[1404+80  ]=0            SW=00000000
9690ns 000001d0 : 092d0050 R[02]=000005cc=1484       SW=00000000
9770ns 000001d4 : 022d0014 m[1404+20  ]=1484         SW=00000000
9850ns 000001d8 : 012d004c R[02]=0000000d=13         SW=00000000
9930ns 000001dc : 017d0058 R[07]=00000000=0          SW=00000000
10010ns 000001e0 : 01ed005c R[14]=00000050=80         SW=00000000
10090ns 000001e4 : 09dd0060 R[13]=000005dc=1500       SW=00000000
10170ns 000001e8 : 2c000000 R[00]=00000000=0          SW=00000000
10250ns 00000050 : 022d0014 m[1500+20  ]=13           SW=00000000
10330ns 00000054 : 022d0000 m[1500+0   ]=13           SW=00000000
10410ns 00000058 : 2b000190 R[00]=00000000=0          SW=00000000
10490ns 000001ec : 09ddfff8 R[13]=000005d4=1492       SW=00000000
10570ns 000001f0 : 012d0008 R[02]=0000000d=13         SW=00000000
10650ns 000001f4 : 022d0004 m[1492+4   ]=13           SW=00000000
10730ns 000001f8 : 09207000 R[02]=00007000=28672      SW=00000000
10810ns 000001fc : 022d0000 m[1492+0   ]=28672        SW=00000000
10890ns 00000200 : 013d0004 R[03]=0000000d=13         SW=00000000
10970ns 00000204 : 02320000 OUTPUT=13
11050ns 00000208 : 09dd0008 R[13]=000005dc=1500       SW=00000000
11130ns 0000020c : 2c000000 R[00]=00000000=0          SW=00000000
11210ns 0000005c : 2b0001b0 R[00]=00000000=0          SW=00000000
11290ns 00000210 : 09ddffe8 R[13]=000005c4=1476       SW=00000000
11370ns 00000214 : 09200001 R[02]=00000001=1          SW=00000000
11450ns 00000218 : 022d0014 m[1476+20  ]=1            SW=00000000
11530ns 0000021c : 09200002 R[02]=00000002=2          SW=00000000
11610ns 00000220 : 022d0010 m[1476+16  ]=2            SW=00000000
11690ns 00000224 : 09200003 R[02]=00000003=3          SW=00000000
11770ns 00000228 : 022d000c m[1476+12  ]=3            SW=00000000
11850ns 0000022c : 09200004 R[02]=00000004=4          SW=00000000
11930ns 00000230 : 022d0008 m[1476+8   ]=4            SW=00000000
12010ns 00000234 : 09200005 R[02]=00000005=5          SW=00000000
12090ns 00000238 : 022d0004 m[1476+4   ]=5            SW=00000000
12170ns 0000023c : 012d0014 R[02]=00000001=1          SW=00000000
12250ns 00000240 : 2720000c HI=00000002 LO=00000004 SW=00000000
12330ns 00000244 : 012d0014 R[02]=00000001=1          SW=00000000
12410ns 00000248 : 09220001 R[02]=00000002=2          SW=00000000
12490ns 0000024c : 022d0014 m[1476+20  ]=2            SW=00000000
12570ns 00000250 : 012d0010 R[02]=00000002=2          SW=00000000
12650ns 00000254 : 0a220001 R[02]=00000000=0          SW=00000000
12730ns 00000258 : 2820000c R[02]=00000000=0          SW=00000000
12810ns 0000025c : 012d0010 R[02]=00000002=2          SW=00000000
12890ns 00000260 : 09220001 R[02]=00000003=3          SW=00000000
12970ns 00000264 : 022d0010 m[1476+16  ]=3            SW=00000000
13050ns 00000268 : 012d000c R[02]=00000003=3          SW=00000000
13130ns 0000026c : 0a220000 R[02]=00000000=0          SW=00000000
13210ns 00000270 : 2820000c R[02]=00000000=0          SW=00000000
13290ns 00000274 : 012d000c R[02]=00000003=3          SW=00000000
13370ns 00000278 : 09220001 R[02]=00000004=4          SW=00000000
13450ns 0000027c : 022d000c m[1476+12  ]=4            SW=00000000
13530ns 00000280 : 012d0008 R[02]=00000004=4          SW=00000000
13610ns 00000284 : 0930ffff R[03]=ffffffff=-1         SW=00000000
13690ns 00000288 : 20232000 R[02]=00000001=1          SW=00000000
13770ns 0000028c : 2820000c R[02]=00000001=1          SW=00000000
13850ns 0000029c : 012d0004 R[02]=00000005=5          SW=00000000
13930ns 000002a0 : 09300000 R[03]=00000000=0          SW=00000000
14010ns 000002a4 : 20232000 R[02]=00000001=1          SW=00000000
14090ns 000002a8 : 2820000c R[02]=00000001=1          SW=00000000
```

```
14170ns 000002b8 : 012d0010 R[02]=00000003=3          SW=00000000
14250ns 000002bc : 013d0014 R[03]=00000002=2          SW=00000000
14330ns 000002c0 : 11232000 R[02]=00000005=5          SW=00000000
14410ns 000002c4 : 013d000c R[03]=00000004=4          SW=00000000
14490ns 000002c8 : 11223000 R[02]=00000009=9          SW=00000000
14570ns 000002cc : 013d0008 R[03]=00000004=4          SW=00000000
14650ns 000002d0 : 11223000 R[02]=0000000d=13         SW=00000000
14730ns 000002d4 : 013d0004 R[03]=00000005=5          SW=00000000
14810ns 000002d8 : 11223000 R[02]=00000012=18         SW=00000000
14890ns 000002dc : 09dd0018 R[13]=000005dc=1500       SW=00000000
14970ns 000002e0 : 2c000000 R[00]=00000000=0          SW=00000000
15050ns 00000060 : 013d0014 R[03]=0000000d=13         SW=00000000
15130ns 00000064 : 11232000 R[02]=0000001f=31         SW=00000000
15210ns 00000068 : 022d0014 m[1500+20  ]=31           SW=00000000
15290ns 0000006c : 022d0000 m[1500+0   ]=31           SW=00000000
15370ns 00000070 : 2b000178 R[00]=00000000=0          SW=00000000
15450ns 000001ec : 09ddfff8 R[13]=000005d4=1492       SW=00000000
15530ns 000001f0 : 012d0008 R[02]=0000001f=31         SW=00000000
15610ns 000001f4 : 022d0004 m[1492+4   ]=31           SW=00000000
15690ns 000001f8 : 09207000 R[02]=00007000=28672      SW=00000000
15770ns 000001fc : 022d0000 m[1492+0   ]=28672        SW=00000000
15850ns 00000200 : 013d0004 R[03]=0000001f=31         SW=00000000
15930ns 00000204 : 02320000 OUTPUT=31
16010ns 00000208 : 09dd0008 R[13]=000005dc=1500       SW=00000000
16090ns 0000020c : 2c000000 R[00]=00000000=0          SW=00000000
16170ns 00000074 : 2b00026c R[00]=00000000=0          SW=00000000
16250ns 000002e4 : 09ddfff4 R[13]=000005d0=1488       SW=00000000
16330ns 000002e8 : 022d0008 m[1488+8   ]=28672        SW=00000000
16410ns 000002ec : 023d0004 m[1488+4   ]=31           SW=00000000
16490ns 000002f0 : 024d0000 m[1488+0   ]=12           SW=00000000
16570ns 000002f4 : 0f207fff R[02]=7fff0000=2147418112 SW=00000000
16650ns 000002f8 : 0f301000 R[03]=10000000=268435456  SW=00000000
16730ns 000002fc : 11423000 R[04]=8fff0000=-1879113728 SW=00000000
16810ns 00000300 : 0f207fff R[02]=7fff0000=2147418112 SW=00000000
16890ns 00000304 : 0f301000 R[03]=10000000=268435456  SW=00000000
16970ns 00000308 : 13423000 R[04]=8fff0000=-1879113728 SW=10000000
17050ns 0000030c : 0f208fff R[02]=8fff0000=-1879113728 SW=00000000
17130ns 00000310 : 0f307000 R[03]=70000000=1879048192 SW=00000000
17210ns 00000314 : 14423000 R[04]=1fff0000=536805376  SW=10000000
17290ns 00000318 : 0f200000 R[02]=00000000=0          SW=00000000
17370ns 0000031c : 0930ffff R[03]=ffffffff=-1         SW=00000000
17450ns 00000320 : 14423000 R[04]=00000001=1          SW=00000000
17530ns 00000324 : 0f20ffff R[02]=ffff0000=-65536     SW=00000000
17610ns 00000328 : 0d22ffff R[02]=ffffffff=-1         SW=00000000
17690ns 0000032c : 0c22ffff R[02]=0000ffff=65535      SW=00000000
17770ns 00000330 : 1e220010 R[02]=ffff0000=-65536     SW=00000000
17850ns 00000334 : 0e22ffff R[02]=ffffffff=-1         SW=00000000
17930ns 00000338 : 0930ffff R[03]=ffffffff=-1         SW=00000000
18010ns 0000033c : 17230000 HI=00000000 LO=00000001 SW=00000000
18090ns 00000340 : 16230000 HI=00000000 LO=00000001 SW=10000000
18170ns 00000344 : 0e220001 R[02]=fffffffe=-2         SW=00000000
18250ns 00000348 : 1c421004 R[04]=ffffffef=-17        SW=00000000
18330ns 0000034c : 1d421008 R[04]=feffffff=-16777217  SW=00000000
18410ns 00000350 : 012d0008 R[02]=00007000=28672      SW=00000000
18490ns 00000354 : 013d0004 R[03]=0000001f=31         SW=00000000
18570ns 00000358 : 014d0000 R[04]=0000000c=12         SW=00000000
18650ns 0000035c : 09dd000c R[13]=000005dc=1500       SW=00000000
18730ns 00000360 : 2c000000 R[00]=00000000=0          SW=00000000
```

```
18810ns 00000078 : 012d0014 R[02]=0000001f=31         SW=00000000
18890ns 0000007c : 01ed001c R[14]=ffffffff=-1         SW=00000000
18970ns 00000080 : 09dd0020 R[13]=000005fc=1532       SW=00000000
19050ns 00000084 : 2c000000 R[00]=00000000=0          SW=00000000
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:
      .frame  $sp,40,$lr
      .mask   0x00000000,0
      .set    noreorder
      .set    nomacro
# BB#0:
      addiu   $sp, $sp, -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
$tmp1:
      .size   main, ($tmp1)-main
```

The ch11_3.cpp is written in assembly for AsmParser test. You can check if it will generate the obj.

CHAPTER

TWELVE

APPENDIX A: GETTING STARTED: INSTALLING LLVM AND THE CPU0 EXAMPLE CODE

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 of merged date 2013/03/28. So, you have to get book example code and the based llvm trunk by git command as follows,

```
git clone https://github.com/Jonathan2251/lbd.git
```

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++.

Todo

Find information on debugging LLVM within Xcode for Macs.

Todo

Find information on building/debugging LLVM within Eclipse for Linux.

12.1 Setting Up Your Mac

12.1.1 Installing LLVM, Xcode and cmake

[1] http://llvm.org/docs/CMake.html?highlight=cmake

Todo

Fix centering for figure captions.

Please download LLVM latest release version 3.2 (llvm, clang, compiler-rf) from the "LLVM Download Page" [2]. Then extract them using `tar -zxvf {llvm-3.2.src.tar, clang-3.2.src.tar, compiler-rt-3.2.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/project/compiler-rt as shown as follows,

```
118-165-78-111:Downloads Jonathan$ tar -zxvf clang-3.2.src.tar.gz
118-165-78-111:Downloads Jonathan$ tar -zxvf compiler-rt-3.2.src.tar.gz
118-165-78-111:Downloads Jonathan$ tar -zxvf llvm-3.2.src.tar.gz
118-165-78-111:Downloads Jonathan$ mv llvm-3.2.src src
118-165-78-111:Downloads Jonathan$ mv clang-3.2.src src/tools/clang
118-165-78-111:Downloads Jonathan$ mv compiler-rt-3.2.src src/projects/compiler-rt
118-165-78-111:Downloads Jonathan$ pwd
/Users/Jonathan/Downloads
118-165-78-111:Downloads Jonathan$ ls
clang-3.2.src.tar.gz        llvm-3.2.src.tar.gz
compiler-rt-3.2.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 12.1 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.

12.1.2 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 12.2, then click **configure** button to enter Figure 12.3, and then click **Done** button to get Figure 12.4.

[2] http://llvm.org/releases/download.html#3.2

[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

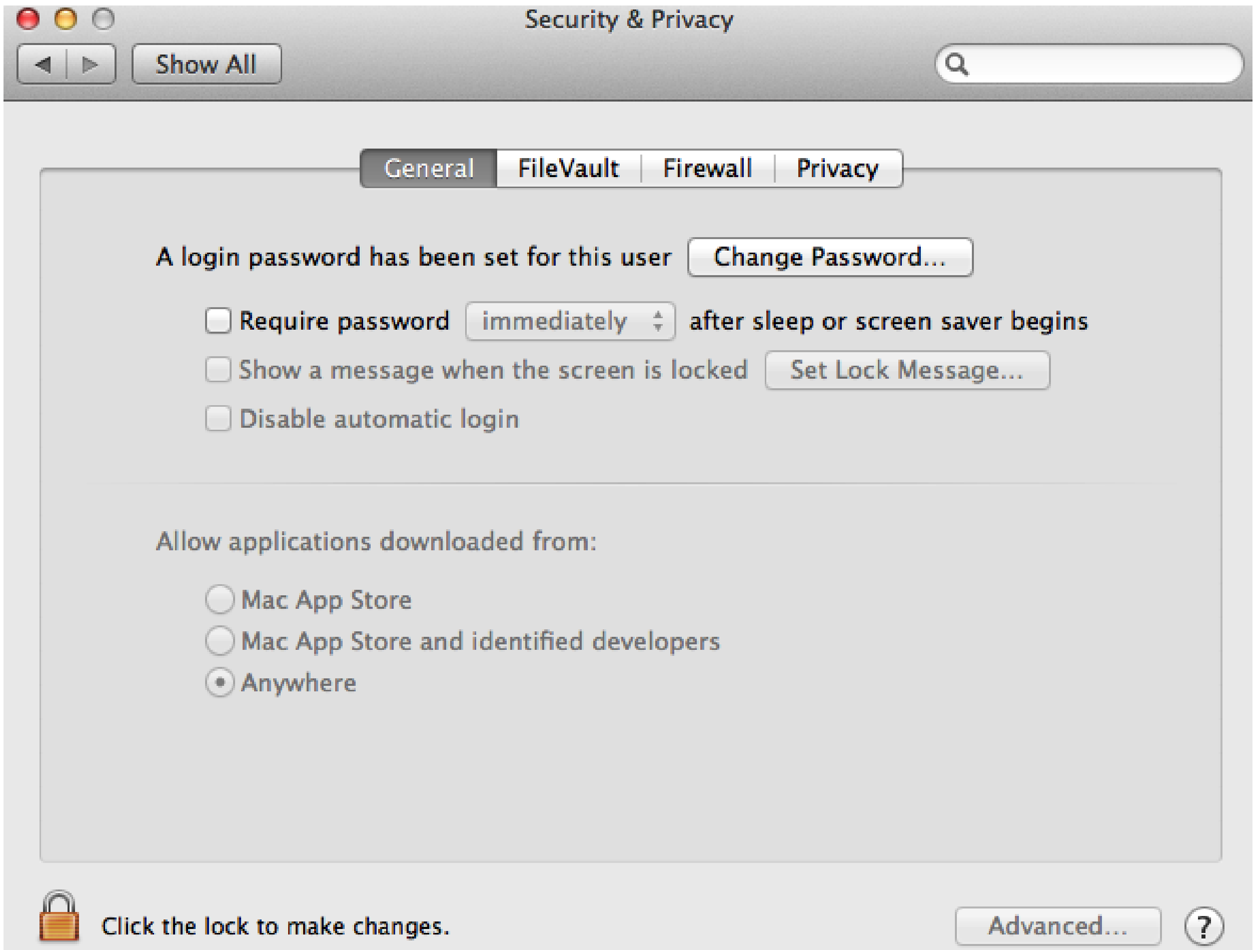


Figure 12.1: Adjusting Mac OS X security settings to allow cmake installation.

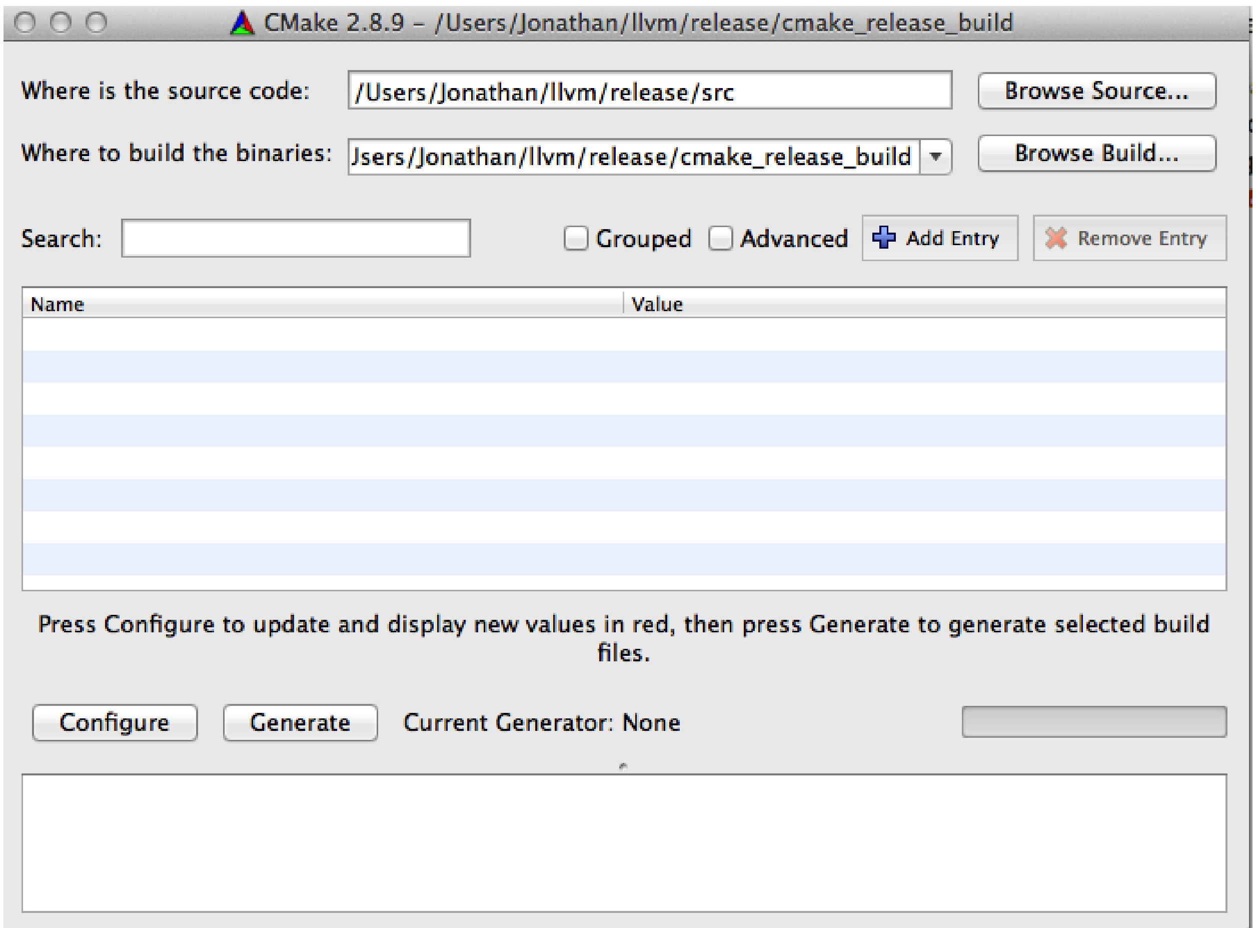


Figure 12.2: Start to create LLVM.xcodeproj by cmake

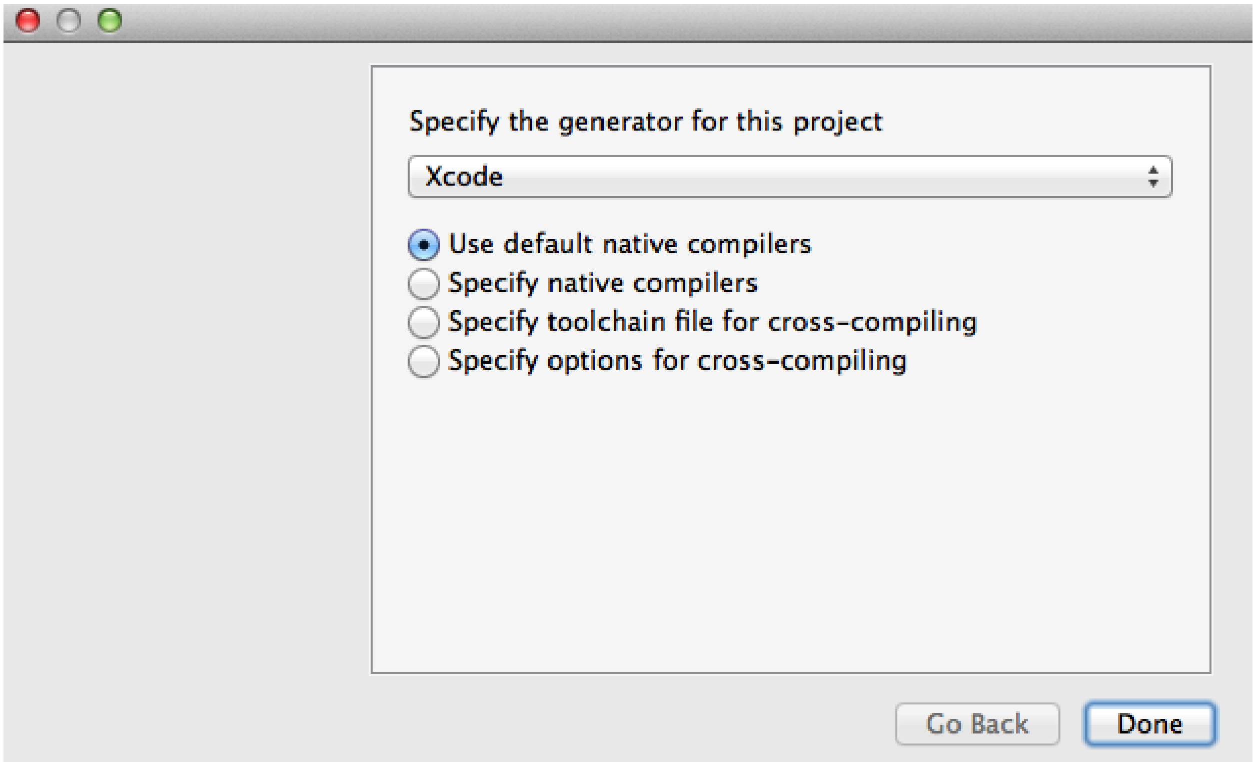


Figure 12.3: Create LLVM.xcodeproj by cmake – Set option to generate Xcode project

Click OK from Figure 12.4 and select Cmake 2.8-9.app for CMAKE_INSTALL_NAME_TOOL by click the right side button **"..."** of that row to get Figure 12.5.

Click Configure button to get Figure 12.6.

Check CLANG_BUILD_EXAMPLES, LLVM_BUILD_EXAMPLES, and uncheck LLVM_ENABLE_PIC as Figure 12.7.

Click Configure button again. If the output result message has no red color, then click Generate button to get Figure 12.8.

12.1.3 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 12.9.

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
```

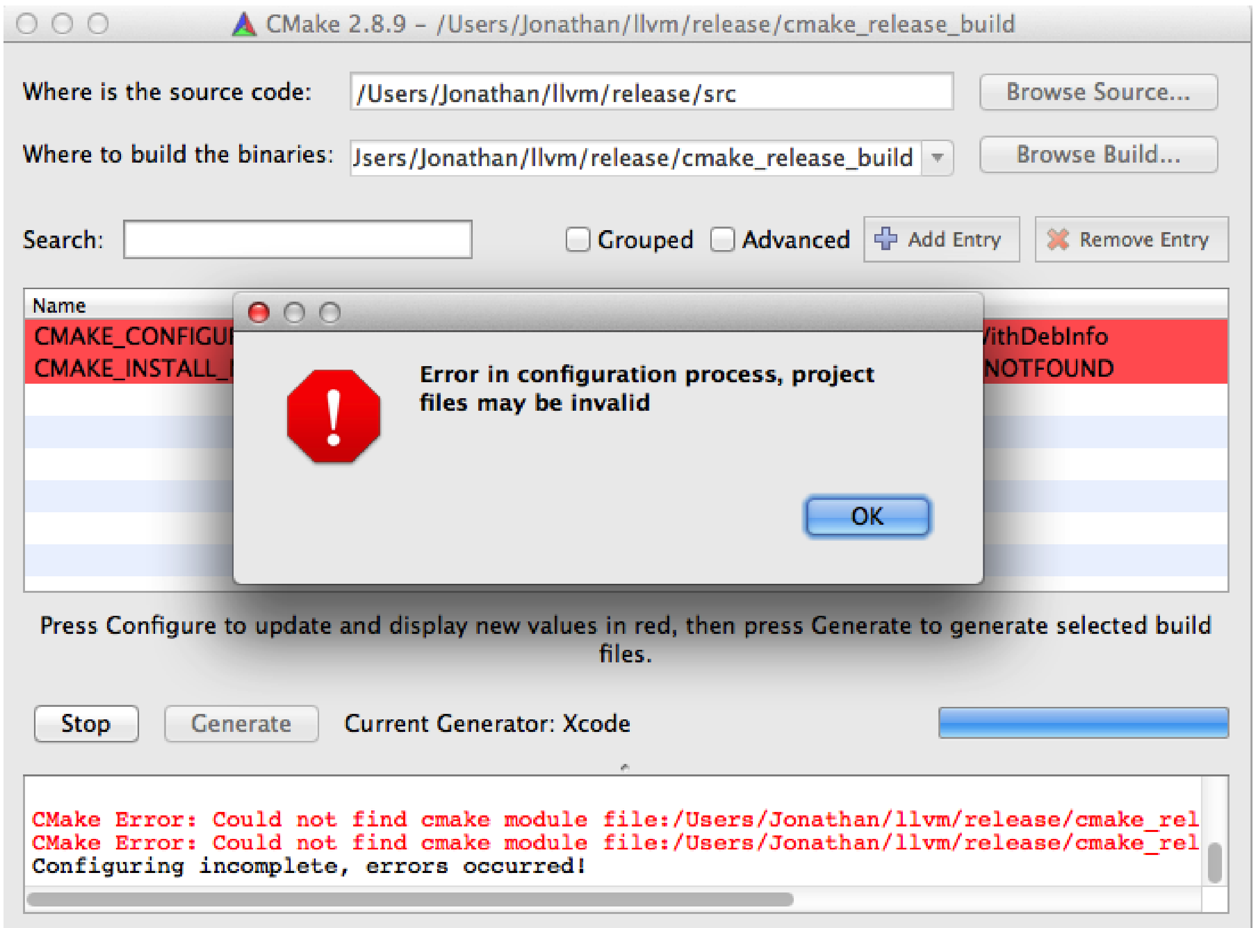


Figure 12.4: Create LLVM.xcodeproj by cmake – Before Adjust CMAKE_INSTALL_NAME_TOOL

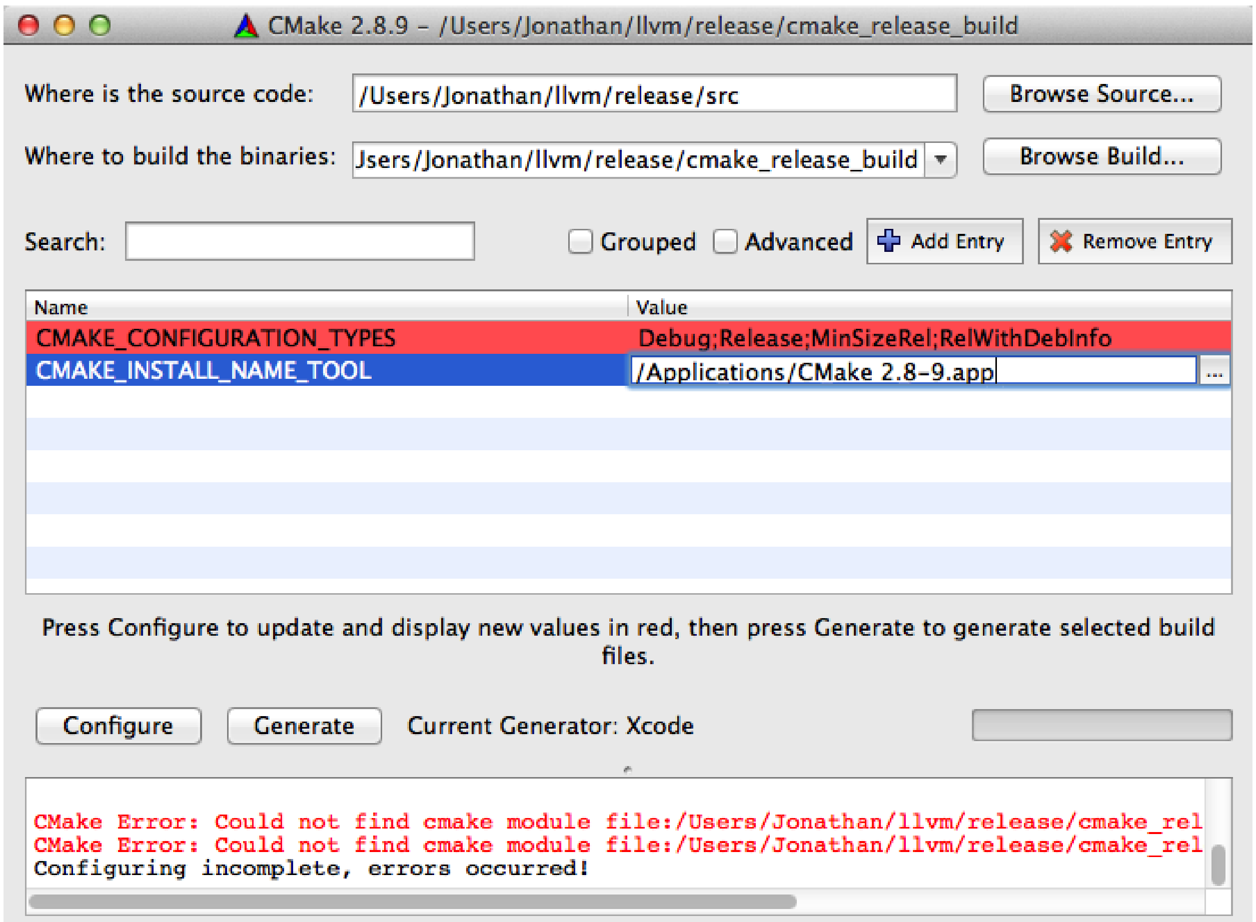


Figure 12.5: Select Cmake 2.8-9.app

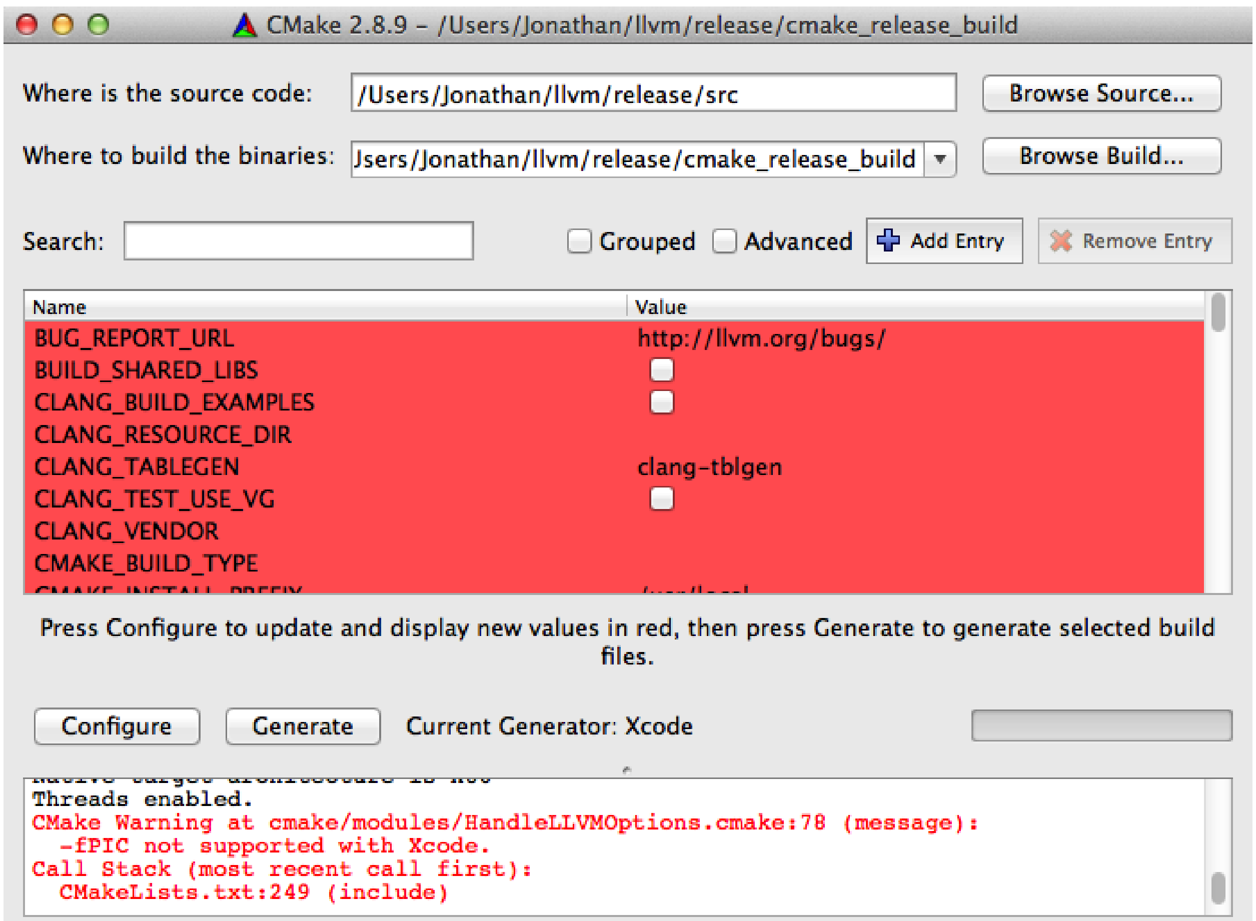


Figure 12.6: Click cmake Configure button first time

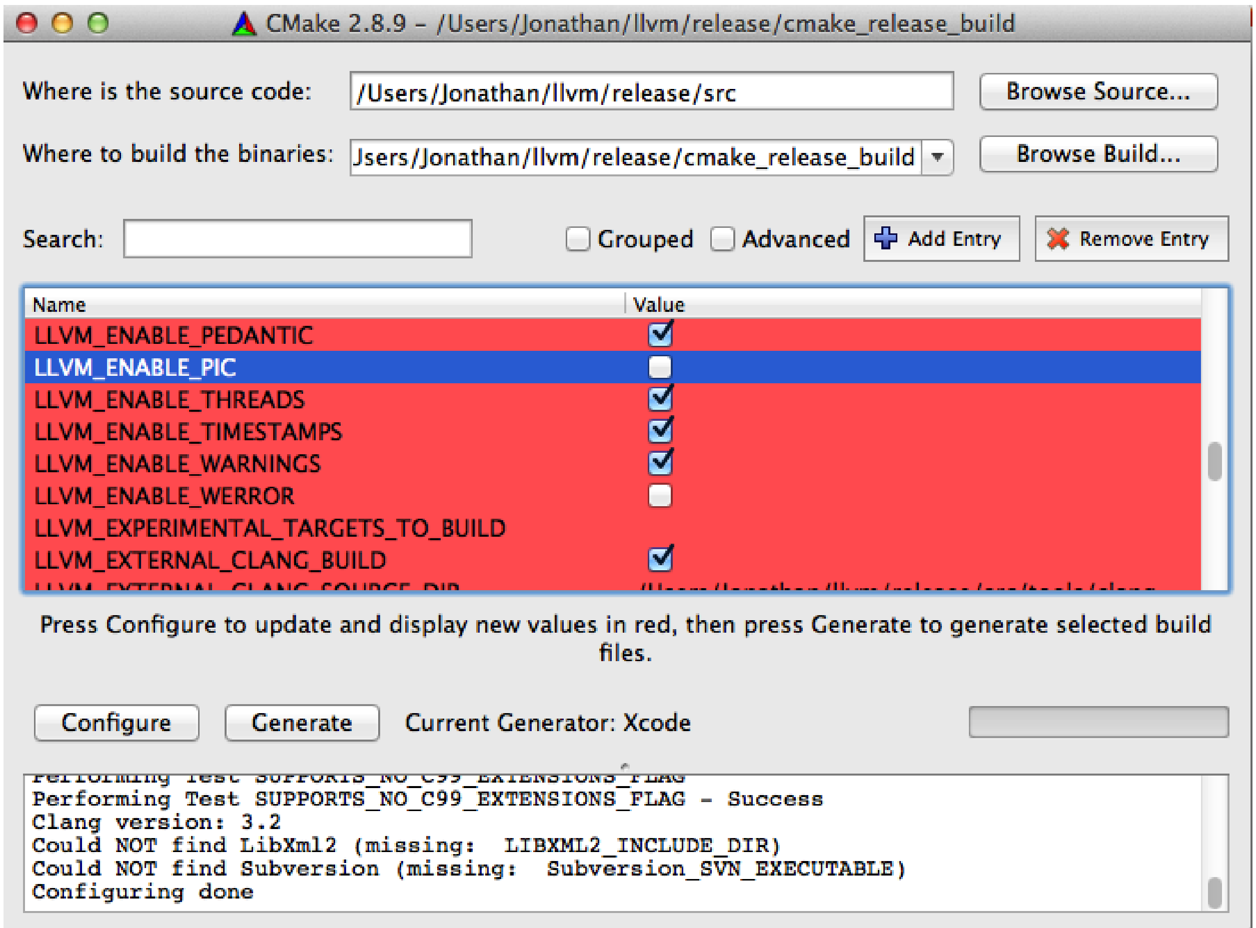


Figure 12.7: Check CLANG_BUILD_EXAMPLES, LLVM_BUILD_EXAMPLES, and uncheck LLVM_ENABLE_PIC in cmake

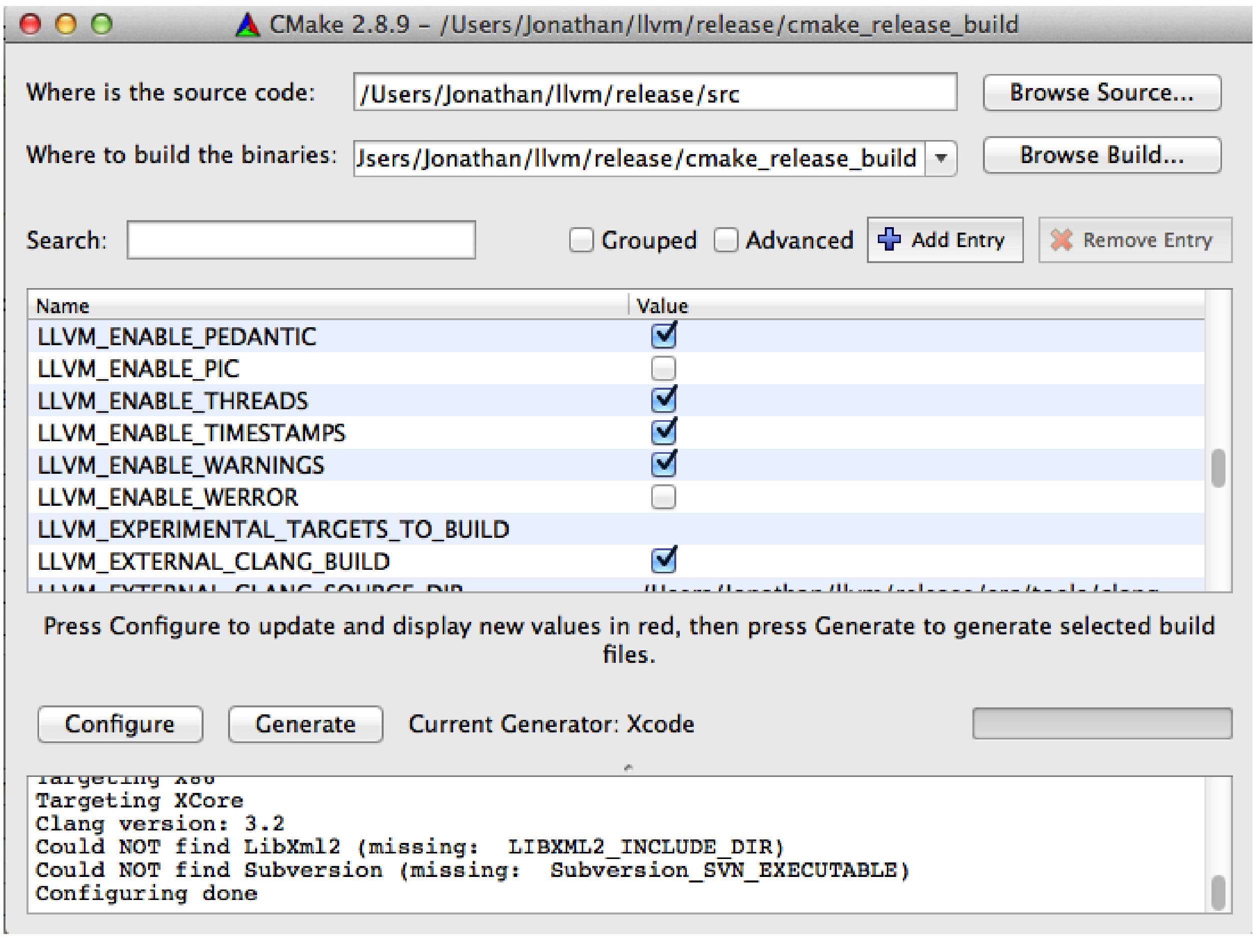


Figure 12.8: Click cmake Generate button second time

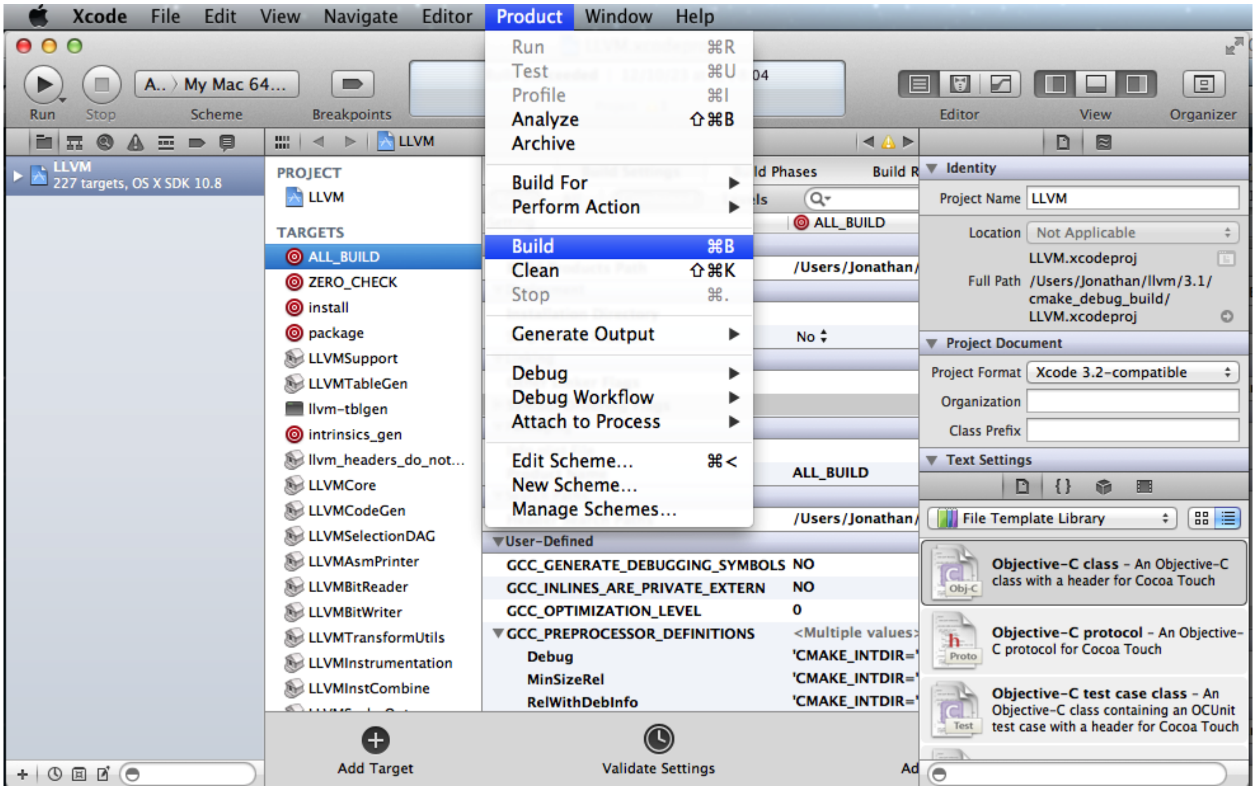


Figure 12.9: Click Build button to build LLVM.xcodeproj by Xcode

```
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
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=$PATH:/Applications/Xcode.app/Contents/Developer/usr/bin:/Applicatio
ns/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/:/Ap
plications/Graphviz.app/Contents/MacOS/:/Users/Jonathan/llvm/release/cmake_relea
se_build/bin/Debug
export WORKON_HOME=$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$
```

12.1.4 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 of merged date 2013/03/28. So, you have to get book example code and the based llvm trunk by git command as follows,

```
git clone https://github.com/Jonathan2251/lbd.git
```

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$ git clone https://github.com/Jonathan2251/lbd.git src
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/MCExpr.h:    VK_Cpu0_GPREL,
src/include/llvm/MC/MCExpr.h:    VK_Cpu0_GOT_CALL,
src/include/llvm/MC/MCExpr.h:    VK_Cpu0_GOT16,
src/include/llvm/MC/MCExpr.h:    VK_Cpu0_GOT,
src/include/llvm/MC/MCExpr.h:    VK_Cpu0_ABS_HI,
src/include/llvm/MC/MCExpr.h:    VK_Cpu0_ABS_LO,
...
src/lib/MC/MCExpr.cpp:  case VK_Cpu0_GOT_PAGE: return "GOT_PAGE";
src/lib/MC/MCExpr.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. Remind, currently, the `cmake` terminal command can work with lldb debug, but the "section Create LLVM.xcodeproj by cmake Graphic UI" [5] cannot.

[5] http://jonathan2251.github.com/lbd/install.html#create-llvm-xcodeproj-by-cmake-graphic-ui

```
118-165-78-111:Target Jonathan$ cd ../../../
118-165-78-111:test Jonathan$ ls
src
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/
llc' (x86_64)
Process 6058 stopped
* thread #1: tid = 0x1c03, 0x000000010077f231 llc`LLVMInitializeMipsTargetInfo
+ 33 at MipsTargetInfo.cpp:20, stop reason = breakpoint 1.1
  frame #0: 0x000000010077f231 llc`LLVMInitializeMipsTargetInfo + 33 at
  MipsTargetInfo.cpp:20
   17
   18   extern "C" void LLVMInitializeMipsTargetInfo() {
   19     RegisterTarget<Triple::mips,
-> 20           /*HasJIT=*/true> X(TheMipsTarget, "mips", "Mips");
   21
   22     RegisterTarget<Triple::mipsel,
   23           /*HasJIT=*/true> Y(TheMipselTarget, "mipsel", "Mipsel");
(lldb) n
Process 6058 stopped
* thread #1: tid = 0x1c03, 0x000000010077f24f llc`LLVMInitializeMipsTargetInfo
+ 63 at MipsTargetInfo.cpp:23, stop reason = step over
  frame #0: 0x000000010077f24f llc`LLVMInitializeMipsTargetInfo + 63 at
  MipsTargetInfo.cpp:23
   20           /*HasJIT=*/true> X(TheMipsTarget, "mips", "Mips");
   21
   22     RegisterTarget<Triple::mipsel,
-> 23           /*HasJIT=*/true> Y(TheMipselTarget, "mipsel", "Mipsel");
   24
   25     RegisterTarget<Triple::mips64,
   26           /*HasJIT=*/false> A(TheMips64Target, "mips64", "Mips64
   [experimental]");
(lldb) print X
(llvm::RegisterTarget<llvm::Triple::ArchType, true>) $0 = {}
(lldb) quit
118-165-65-128:InputFiles Jonathan$
```

About the lldb debug command, please reference [6] or lldb portal [7].

12.1.5 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

12.1.6 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
```

12.1.7 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 12.10 and drag it into the Dock as Figure 12.11.

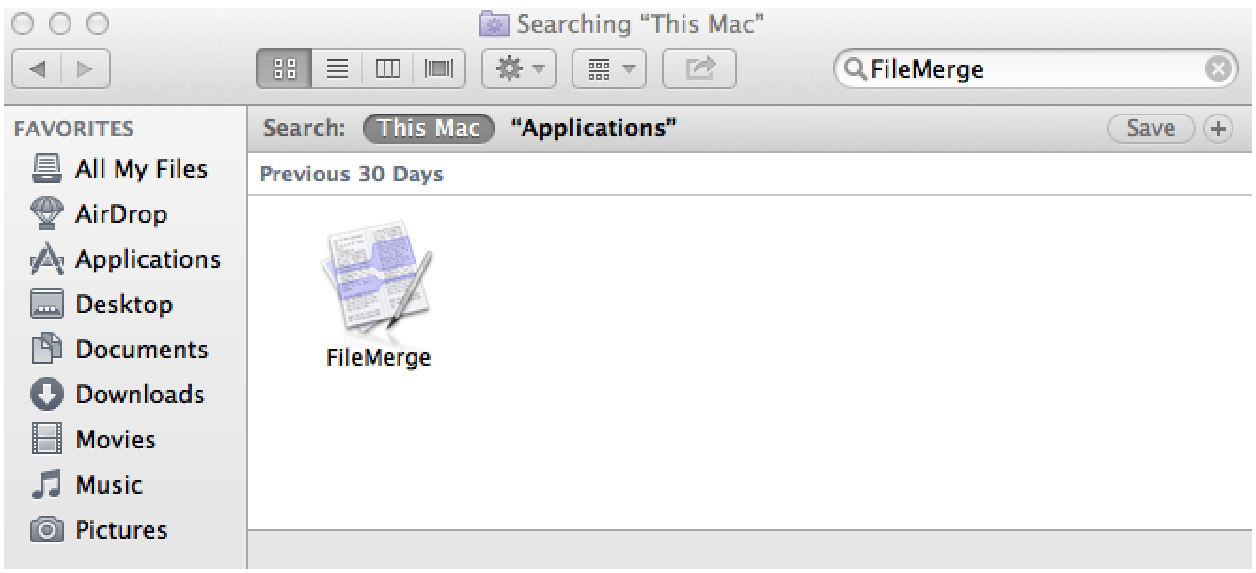


Figure 12.10: Type FileMerge in Finder – Applications

Figure 12.11: 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 12.12 since our iMac is Mountain Lion.

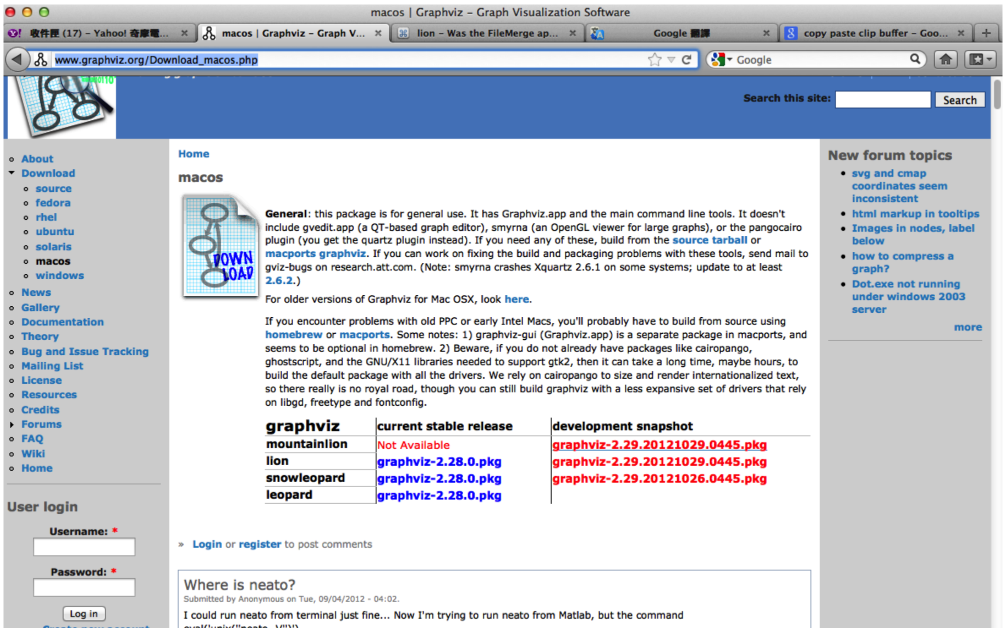


Figure 12.12: Download graphviz for llvm IR node display

After install Graphviz, please set the path to .profile. For example, we install the Graphviz in directory /Applications/Graphviz.app/Contents/MacOS/, so add this path to /User/Jonathan/.profile as follows,

```
118-165-12-177:InputFiles Jonathan$ cat /Users/Jonathan/.profile
export PATH=$PATH:/Applications/Xcode.app/Contents/Developer/usr/bin:
/Applications/Graphviz.app/Contents/MacOS/:/Users/Jonathan/llvm/release/
cmake_release_build/bin/Debug
```

The Graphviz information for llvm is in the section "SelectionDAG Instruction Selection Process" of [11] and the section "Viewing graphs while debugging code" of [12]. 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 12.13. Then select **"Front document's file"** as Figure 12.14.

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/loca
==> make
==> make install
/usr/local/Cellar/binutils/2.22: 90 files, 19M, built in 4.7 minutes
118-165-77-214:~ Jonathan$ ls /usr/local/Cellar/binutils/2.22
COPYING     README      lib
ChangeLog     bin       share
```

[10] http://www.graphviz.org/Download_macos.php
[11] http://llvm.org/docs/CodeGenerator.html
[12] http://llvm.org/docs/ProgrammersManual.html

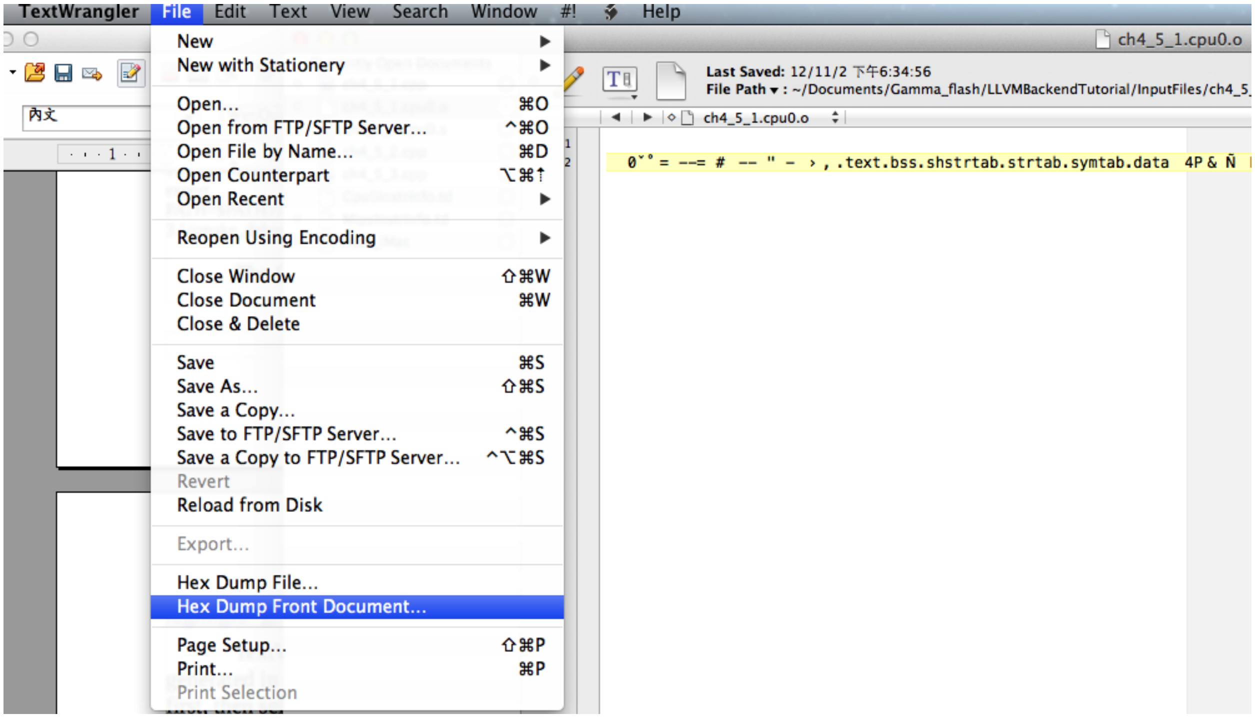


Figure 12.13: Select Hex Dump menu

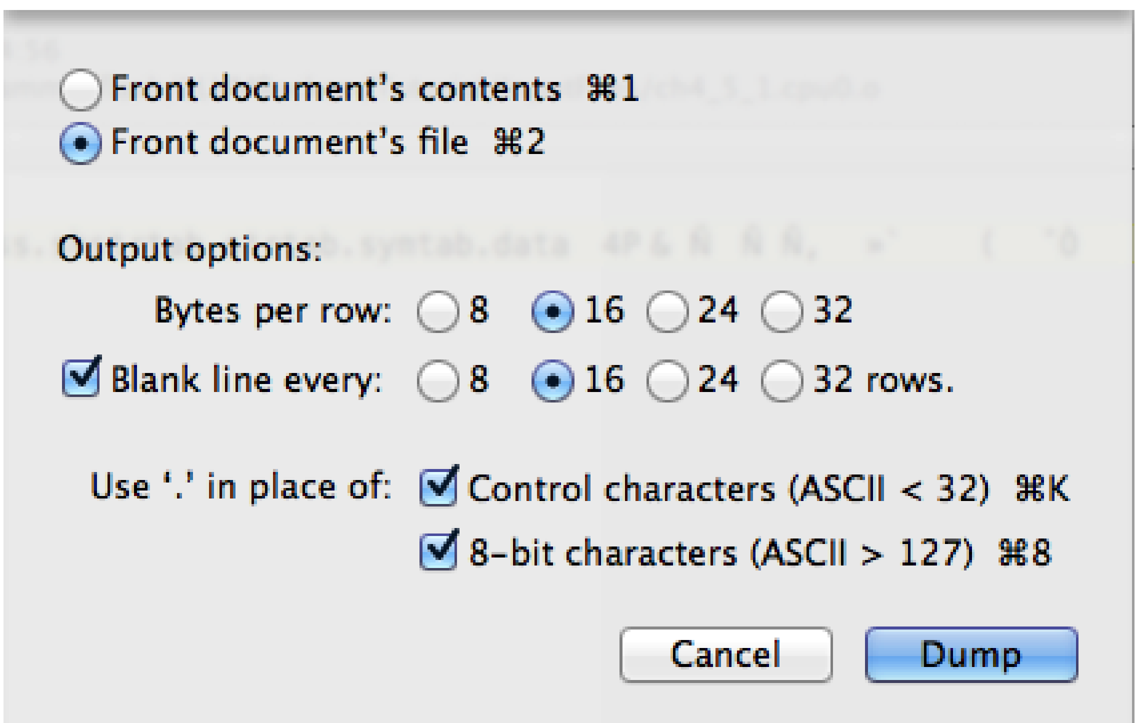


Figure 12.14: Select Front document's file in TextWrangler

```
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
```

12.2 Setting Up Your Linux Machine

12.2.1 Install LLVM 3.2 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 MBlaze
-- Targeting MSP430
-- Targeting PowerPC
-- Targeting PTX
-- Targeting Sparc
-- Targeting X86
-- Targeting XCore
-- Clang version: 3.2
-- 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
```

12.2.2 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 of merged date 2013/03/28. So, you have to get book example code and the based llvm trunk by git command as follows,

```
git clone https://github.com/Jonathan2251/lbd.git
```

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 trunk of Cpu0 based.

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 2 is effective by command `grep -R "Cpu0" .  | more`. I add the Cpu0 backend support, so check with grep.

5) Enter src/lib/Target/Cpu0/, generate LLVMBackendTutorialExampleCode, 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.

```
118-165-78-111:llvm Jonathan$ mkdir test
118-165-78-111:llvm Jonathan$ cd test
[Gamma@localhost test]$ pwd
/usr/local/llvm/test
[Gamma@localhost test]$ git clone https://github.com/Jonathan2251/lbd.git src
[Gamma@localhost test]$ cp -rf src/lib/Target/Cpu0/
LLVMBackendTutorialExampleCode/src_files_modify/modify/src/* src/.
[Gamma@localhost test]$ grep -R "Cpu0" src/inlcude
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 3.2.0
-- The CXX compiler identification is Clang 3.2.0
-- 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 MBlaze
-- 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]$
```

12.2.3 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.

12.2.4 Install other tools on Linux

Download Graphviz from [13] according your Linux distribution. Files compare tools Kdiff3 came from web site [8].

[13] http://www.graphviz.org/Download..php

CHAPTER THIRTEEN

APPENDIX B: LLVM CHANGES

This chapter show you the old version of LLVM API and structure those affect Cpu0 back end. Mips changes also mentioned in this chapter. If you work on the latest LLVM version only, please skip this chapter. LLVM version 3.2 released in 20 December, 2012. Version 3.1 released in 22 May, 2012. This book started from September, 2012. This chapter discuss the old version start from 3.1.

13.1 Difference between 3.2 and 3.1

13.1.1 API difference

Difference in API as follows,

1. In llvm 3.1, the parameters of call back function for Target Registration is different from 3.2. LLVM 3.2 add parameter "MCRegisterInfo" in the callback function for RegisterMCCodeEmitter() and "StringRef" in the callback function for RegisterMCAsmBackend. In other word, you can get more information of registers and CPU (type of StringRef) for your backend after this registration. Of course, these information came from TabGen which source is the Target Description .td you write.

```
extern "C" void LLVMInitializeCpu0TargetMC() {
  ...
  // Register the MC Code Emitter
  TargetRegistry::RegisterMCCodeEmitter(TheCpu0Target,
                    createCpu0MCCodeEmitterEB);
  TargetRegistry::RegisterMCCodeEmitter(TheCpu0elTarget,
                    createCpu0MCCodeEmitterEL);
  ...

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

Version 3.1 as follows,

```
MCCodeEmitter *createCpu0MCCodeEmitterEB(const MCInstrInfo &MCII,
                     const MCSubtargetInfo &STI,
                     MCContext &Ctx);
MCCodeEmitter *createCpu0MCCodeEmitterEL(const MCInstrInfo &MCII,
                     const MCSubtargetInfo &STI,
```

```
                     MCContext &Ctx);

MCAsmBackend *createCpu0AsmBackendEB32(const Target &T, StringRef TT);
MCAsmBackend *createCpu0AsmBackendEL32(const Target &T, StringRef TT);
```

Version 3.2 as follows,

```
MCCodeEmitter *createCpu0MCCodeEmitterEB(const MCInstrInfo &MCII,
                     const MCRegisterInfo &MRI,
                     const MCSubtargetInfo &STI,
                     MCContext &Ctx);
MCCodeEmitter *createCpu0MCCodeEmitterEL(const MCInstrInfo &MCII,
                     const MCRegisterInfo &MRI,
                     const MCSubtargetInfo &STI,
                     MCContext &Ctx);

MCAsmBackend *createCpu0AsmBackendEB32(const Target &T, StringRef TT,
                     StringRef CPU);
MCAsmBackend *createCpu0AsmBackendEL32(const Target &T, StringRef TT,
                     StringRef CPU);
```

2. Change LowerCall() parameters as follows,

Version 3.1 as follows,

```
SDValue
    LowerCall(SDValue Chain, SDValue Callee,
        CallingConv::ID CallConv, bool isVarArg,
        bool doesNotRet, bool &isTailCall,
        const SmallVectorImpl<ISD::OutputArg> &Outs,
        const SmallVectorImpl<SDValue> &OutVals,
        const SmallVectorImpl<ISD::InputArg> &Ins,
        DebugLoc dl, SelectionDAG &DAG,
        SmallVectorImpl<SDValue> &InVals) const;
```

Version 3.2 as follows,

```
LowerCall(TargetLowering::CallLoweringInfo &CLI,
        SmallVectorImpl<SDValue> &InVals) const;
```

The TargetLowering::CallLoweringInfo is type of structure/class which contains the old version 3.1 parameters. You can get the 3.1 same information by,

```
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;
  ...
}
```

As chapter “function call”, the role of LowerCall() is handling the outgoing arguments passing in function call.

3. The TargetData structure of LLVMTargetMachine has been renamed to DataLayout and the corresponding function name change as follows,

```
// 3.1
class Cpu0TargetMachine : public LLVMTargetMachine {
  ...
  virtual const TargetData      *getTargetData()    const
  { return &DataLayout;}
  ...
}

// 3.2
class Cpu0TargetMachine : public LLVMTargetMachine {
  ...
  virtual const DataLayout *getDataLayout()    const
  { return &DL;}
  ...
}
```

4. The "add a pass" API change as follows,

```
// 3.1
TargetPassConfig *Cpu0TargetMachine::createPassConfig(PassManagerBase &PM) {
  return new Cpu0PassConfig(this, PM);
}

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

// 3.2
// Install an instruction selector pass using
// the ISelDag to gen Cpu0 code.
bool Cpu0PassConfig::addInstSelector() {
  addPass(createCpu0ISelDag(getCpu0TargetMachine()));
  return false;
}
```

5. Above changes is mandatory. There are some changes are adviced to follow. Like the below. We comment the "Change Reason" in the following code. You can get the "Change Reason" by internet searching.

```
  MCObjectWriter *createObjectWriter(raw_ostream &OS) const {
  // Change Reason:
  // Reduce the exposure of Triple::OSType in the ELF object writer. This will
  //  avoid including ADT/Triple.h in many places when the target specific bits
  //  are moved.
  return createCpu0ELFObjectWriter(OS,
    MCELFObjectTargetWriter::getOSABI(OSType), IsLittle);
// Even though, the old function still work on LLVM version 3.2
//    return createCpu0ELFObjectWriter(OS, OSType, IsLittle);
  }

class Cpu0MCCodeEmitter : public MCCodeEmitter {
  // #define LLVM_DELETED_FUNCTION
  //  LLVM_DELETED_FUNCTION - Expands to = delete if the compiler supports it.
  //  Use to mark functions as uncallable. Member functions with this should be
```

```
  //  declared private so that some behavior is kept in C++03 mode.
  //  class DontCopy { private: DontCopy(const DontCopy&) LLVM_DELETED_FUNCTION;
  //  DontCopy &operator =(const DontCopy&) LLVM_DELETED_FUNCTION; public: ... };
  //  Definition at line 79 of file Compiler.h.

  Cpu0MCCodeEmitter(const Cpu0MCCodeEmitter &) LLVM_DELETED_FUNCTION;
  void operator=(const Cpu0MCCodeEmitter &) LLVM_DELETED_FUNCTION;
// Even though, the old function still work on LLVM version 3.2
//  Cpu0MCCodeEmitter(const Cpu0MCCodeEmitter &); // DO NOT IMPLEMENT
//  void operator=(const Cpu0MCCodeEmitter &); // DO NOT IMPLEMENT
...
```

13.1.2 Structure difference

1. Change the name from CPURegsRegisterClass (3.1) to CPURegsRegClass (3.2). The source of register class information came from your backend <register>.td. The new name CPURegsRegClass is **"call by reference"** type in C++ while the old CPURegsRegisterClass is **"pointer"** type. The "reference" type use **"."** while pointer type use **"->"** as follows,

```
// 3.2
unsigned CPURegSize = Cpu0::CPURegsRegClass.getSize();
// 3.1
unsigned CPURegSize = Cpu0::CPURegsRegisterClass->getSize();
```

2. The TargetData structure has been renamed to DataLayout and moved to VMCore to remove a dependency on Target [1].

```
// 3.1
#include "llvm/Target/TargetData.h"
class Cpu0TargetMachine : public LLVMTargetMachine {
  ...
  const TargetData    DataLayout; // Calculates type size & alignment
  ...
}

// 3.2
#include "llvm/DataLayout.h"
class Cpu0TargetMachine : public LLVMTargetMachine {
  ...
  const DataLayout    DL; // Calculates type size & alignment
  ...
}
```

3. DebugInfo.h is moved.

```
// 3.1
#include "llvm/Analysis/DebugInfo.h

// 3.2
#include "llvm/DebugInfo.h
```

[1] http://llvm.org/releases/3.2/docs/ReleaseNotes.html

13.1.3 Verify the Cpu0 for difference

3.1_src_files_modify include the LLVM 3.1 those files modified for Cpu0 backend support. Please copy 3.1_src_files_modify/src_files_modify/src to your LLVM 3.1 source directory. The llvm3.1/Cpu0 is the code for LLVM version 3.1. File ch_all.cpp include the all C/C++ operators, global variable, struct, array, control statement and function call test. Run llvm3.1/Cpu0 with ch_all.cpp will get the assembly code as below. By compare it with the output of 3.2 result, you can verify the correction as below. The difference came from 3.2 correcting the label number in order.

LLVMBackendTutorialExampleCode/InputFiles/ch_all.cpp

```
118-165-78-60:InputFiles Jonathan$ diff ch_all.3.1.cpu0.s ch_all.3.2.cpu0.s
262c262
<   jge $BB4_7
---
>   jge $BB4_6
285d284
< # BB#6:                                  #   in Loop: Header=BB4_1 Depth=1
290c289
< $BB4_7:
---
> $BB4_6:
295,297c294,296
<   jne $BB4_9
<   jmp $BB4_8
< $BB4_8:                                  # %SP_return
---
>   jne $BB4_8
>   jmp $BB4_7
> $BB4_7:                                  # %SP_return
301c300
< $BB4_9:                                  # %CallStackCheckFailBlk
---
> $BB4_8:                                  # %CallStackCheckFailBlk

// ch_all.3.2.cpu0.s
...
$BB4_5:                                  #   in Loop: Header=BB4_1 Depth=1
  ld  $3, 0($3)
  st  $3, 36($sp)
  ld  $4, 32($sp)
  add $3, $4, $3
  st  $3, 32($sp)
  ld  $3, 40($sp)
  addiu $3, $3, 1
  st  $3, 40($sp)
  jmp $BB4_1
$BB4_6:
  ld  $2, %got(__stack_chk_guard)($gp)
  ld  $2, 0($2)
  ld  $3, 48($sp)
  cmp $2, $3
  jne $BB4_8
  jmp $BB4_7
$BB4_7:                                  # %SP_return
...
```

```
// ch_all.3.1.cpu0.s
...
$BB4_5:                                 #   in Loop: Header=BB4_1 Depth=1
  ld  $3, 0($3)
  st  $3, 36($sp)
  ld  $4, 32($sp)
  add $3, $4, $3
  st  $3, 32($sp)
# BB#6:                                 #   in Loop: Header=BB4_1 Depth=1
  ld  $3, 40($sp)
  addiu $3, $3, 1
  st  $3, 40($sp)
  jmp $BB4_1
$BB4_7:
  ld  $2, %got(__stack_chk_guard)($gp)
  ld  $2, 0($2)
  ld  $3, 48($sp)
  cmp $2, $3
  jne $BB4_9
  jmp $BB4_8
$BB4_8:                                 # %SP_return
...
```

13.2 Difference in Mips backend

In 3.1, Mips use **".cpload"** and **".cprestore"** pseudo assembly code. It removes these pseudo assembly code in 3.2. This change is good for spim (mips assembly code simulator) which run for Mips assembly code. According the theory of "System Software", some pseudo assembly code (especially for those not in standard) cannot be translated by assembler. It will break down in assembly code simulator. Run ch_mips_llvm3.2_globalvar_changes.cpp with llvm 3.1 and 3.2 for mips, you will find the **".cprestore"** is removed directly since 3.2 use other register instead of $gp in the callee function (as example, it use $1 in f() and remove **.gprestore** in sum_i()). **".cpload"** is replaced with instructions as follows,

```
// llvm 3.1 mips
  .cpload $25

// llvm 3.2 mips
  lui $2, %hi(_gp_disp)
  addiu $2, $2, %lo(_gp_disp)
  ...
  addu  $gp, $2, $25
```

Reference [2] for **".cpload"**, **".cprestore"** and **"_gp_disp"**.

[2] http://jonathan2251.github.com/lbd/funccall.html#handle-gp-register-in-pic-addressing-mode

CHAPTER

FOURTEEN

APPENDIX C: INSTRUCTIONS DISCUSS

This chapter discuss other backend instructions.

14.1 Use cpu0 official LDI instead of ADDiu

According cpu0 web site instruction definition. There is no addiu instruction definition. We add **addiu** instruction because we find this instruction is more powerful and reasonable than **ldi** instruction. We highlight this change in section CPU0 processor architecture. Even with that, we show you how to replace our **addiu** with **ldi** according the cpu0 original design. Chapter4_2/ is the code changes for use **ldi** instruction. This changes replace **addiu** with **ldi** in Cpu0InstrInfo.td and modify Cpu0FrameLowering.cpp as follows,

LLVMBackendTutorialExampleCode/Chapter4_2/Cpu0InstrInfo.td

```
/// Arithmetic Instructions (ALU Immediate)
def LDI     : MoveImm<0x08, "ldi", add, simm16, immSExt16, CPURegs>;
// add defined in include/llvm/Target/TargetSelectionDAG.td, line 315 (def add).
//def ADDiu   : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;
...

// Small immediates

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

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

def : Pat<(add CPURegs:$hi, (Cpu0Lo tglobaladdr:$lo)),
          (ADD CPURegs:$hi, (LDI ZERO, tglobaladdr:$lo))>;

// gp_rel relocs
def : Pat<(add CPURegs:$gp, (Cpu0GPRel tglobaladdr:$in)),
          (ADD CPURegs:$gp, (LDI ZERO, tglobaladdr:$in))>;

def : Pat<(not CPURegs:$in),
           (XOR CPURegs:$in, (LDI ZERO, 1))>;
```

LLVMBackendTutorialExampleCode/Chapter4_2/Cpu0FrameLowering.cpp

```
void Cpu0FrameLowering::emitPrologue(MachineFunction &MF) const {
  ...
  // Adjust stack.
  if (isInt<16>(-StackSize)) {
    // ldi fp, (-stacksize)
    // add sp, sp, fp
    BuildMI(MBB, MBBI, dl, TII.get(Cpu0::LDI), Cpu0::FP).addReg(Cpu0::FP)
                                                         .addImm(-StackSize);
    BuildMI(MBB, MBBI, dl, TII.get(Cpu0::ADD), SP).addReg(SP).addReg(Cpu0::FP);
  }
  ...
}

void Cpu0FrameLowering::emitEpilogue(MachineFunction &MF,
                                 MachineBasicBlock &MBB) const {
  ...
  // Adjust stack.
  if (isInt<16>(-StackSize)) {
    // ldi fp, (-stacksize)
    // add sp, sp, fp
    BuildMI(MBB, MBBI, dl, TII.get(Cpu0::LDI), Cpu0::FP).addReg(Cpu0::FP)
                                                         .addImm(-StackSize);
    BuildMI(MBB, MBBI, dl, TII.get(Cpu0::ADD), SP).addReg(SP).addReg(Cpu0::FP);
  }
  ...
}
```

As above code, we use **add** IR binary instruction (1 register operand and 1 immediate operand, and the register operand is fixed with ZERO) in our solution since we didn't find the **move** IR unary instruction. This code is correct since all the immediate value is translated into **"ldi Zero, imm/address"**. And **(add CPURegs:$gp, $imm16)** is translated into **(ADD CPURegs:$gp, (LDI ZERO, $imm16))**. Let's run Chapter4_4_2/ with ch4_4.cpp to get the correct result below. As you will see, **"addiu $sp, $sp, -24"** will be replaced with the pair instructions of **"ldi $fp, -24"** and **"add $sp, $sp, $fp"**. Since the $sp pointer adjustment is so frequently occurs (it occurs in every function entry and exit point), we reserve the $fp to the pair of stack adjustment instructions **"ldi"** and **"add"**. If we didn't reserve the dedicate registers $fp and $sp, it need to save and restore them in the stack adjustment. It meaning more instructions running cost in this. Anyway, the pair of **"ldi"** and **"add"** to adjust stack pointer is double in cost compete to **"addiu"**, that's the benefit we mentioned in section CPU0 processor architecture.

```
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  main
  .align  2
  .type main,@function
  .ent  main                    # @main
main:
  .cfi_startproc
  .frame  $sp,24,$lr
  .mask   0x00000000,0
  .set  noreorder
```

```
  .set  nomacro
# BB#0:
  ldi $fp, -24
  add $sp, $sp, $fp
$tmp1:
  .cfi_def_cfa_offset 24
  ldi $2, 0
  st  $2, 20($sp)
  ldi $3, 1
  st  $3, 16($sp)
  ldi $3, 2
  st  $3, 12($sp)
  st  $2, 8($sp)
  ldi $3, -5
  st  $3, 4($sp)
  st  $2, 0($sp)
  ld  $2, 12($sp)
  ld  $3, 4($sp)
  udiv  $2, $3, $2
  st  $2, 0($sp)
  ld  $2, 16($sp)
  sra $2, $2, 2
  st  $2, 8($sp)
  ldi $fp, 24
  add $sp, $sp, $fp
  ret $lr
  .set  macro
  .set  reorder
  .end  main
$tmp2:
  .size main, ($tmp2)-main
  .cfi_endproc
```

14.2 Implicit operand

LLVM IR is a 3 address form (4 tuple <opcode, %1, %2, %3) which match the current RISC cpu0 (like Mips). So, it seems no "move" IR DAG. Because "move a, b" can be replaced by "lw a, b_offset($sp)" for local variable, or can be replaced by "addu $a, $0,$ b". The cpu0 is same as Mips. Base on this reason, the move instruction is useless even though it supplied by the cpu0 author.

For the old CPU or Micro Processor (MCU), like PIC, 8051 and old intel processor. These CPU/MCU need memory saving and not aim to high level of program (such as C) only (they aim to assembly code program too). These CPU/MCU need implicit operand, maybe use ACC (accumulate register).

It will translate,

```
c = a + b + d;
```

into,

```
mtacc   Addr(12) // Move b To Acc
add     Addr(16) // Add a To Acc
add     Addr(4)  // Add d To Acc
mfacc   Addr(8)  // Move Acc To c
```

Above code also can be coded by programmer who use assembly language directly in MCU or BIOS programm since maybe the code size is just 4KB or less.

Since cpu0 is a 32 bits (code size can be 4GB), it use Store and Load instructions for memory address access only. Other instructions (include add), use register to register style operation. We change the implicit operand support in this section. It's just a demonstration with this design, not fully support. The purpose is telling reader how to implement this style of CPU/MCU backend. Run Chapter8_8_2/ with ch_move.cpp will get the following result,

LLVMBackendTutorialExampleCode/InputFiles/ch_move.cpp

```
ld  $3, 12($sp) // $3 is a
ld  $4, 16($sp) // $4 is b
mtacc $4        // Move b To Acc
add $3          // Add a To Acc
ld  $4, 4($sp)  // $4 is d
add $4          // Add d To Acc
mfacc $3        // Move Acc to $3
addiu $3, $3, 5 // Add e(=5) to $3
st  $3, 8($sp)
```

To support this implicit operand, ACC. The following code is added to Chapter8_8_2.cpp.

LLVMBackendTutorialExampleCode/Chapter8_8_2/Cpu0RegisterInfo.td

```
let Namespace = "Cpu0" in {
  // General Purpose Registers
  def ZERO : Cpu0GPRReg< 0, "ZERO">, DwarfRegNum<[0]>;
  ...
  def ACC : Register<"acc">, DwarfRegNum<[20]>;
}
...
def RACC : RegisterClass<"Cpu0", [i32], 32, (add ACC)>;
```

LLVMBackendTutorialExampleCode/Chapter8_8_2/Cpu0InstrInfo.td

```
class MoveFromACC<bits<8> op, string instr_asm, RegisterClass RC,
           list<Register> UseRegs>:
  FL<op, (outs RC:$ra), (ins),
   !strconcat(instr_asm, "\t$ra"), [], IIAlu> {
  let rb = 0;
  let imm16 = 0;
  let Uses = UseRegs;
  let neverHasSideEffects = 1;
}

class MoveToACC<bits<8> op, string instr_asm, RegisterClass RC,
         list<Register> DefRegs>:
  FL<op, (outs), (ins RC:$ra),
   !strconcat(instr_asm, "\t$ra"), [], IIAlu> {
  let rb = 0;
  let imm16 = 0;
  let Defs = DefRegs;
  let neverHasSideEffects = 1;
}

class ArithLogicUniR2<bits<8> op, string instr_asm, RegisterClass RC1,
         RegisterClass RC2, list<Register> DefRegs>:
```

```
  FL<op, (outs), (ins RC1:$accum, RC2:$ra),
   !strconcat(instr_asm, "\t$ra"), [], IIAlu> {
  let rb = 0;
  let imm16 = 0;
  let Defs = DefRegs;
  let neverHasSideEffects = 1;
}
...
//def ADD     : ArithLogicR<0x13, "add", add, IIAlu, CPURegs, 1>;
...
def MFACC : MoveFromACC<0x44, "mfacc", CPURegs, [ACC]>;
def MTACC : MoveToACC<0x45, "mtacc", CPURegs, [ACC]>;
def ADD   : ArithLogicUniR2<0x46, "add", RACC, CPURegs, [ACC]>;
...
def : Pat<(add RACC:$lhs, CPURegs:$rhs),
      (ADD RACC:$lhs, CPURegs:$rhs)>;

def : Pat<(add CPURegs:$lhs, CPURegs:$rhs),
      (ADD (MTACC CPURegs:$lhs), CPURegs:$rhs)>;
```

LLVMBackendTutorialExampleCode/Chapter8_8_2/Cpu0InstrInfo.cpp

```
//- Called when DestReg and SrcReg belong to different Register Class.
void Cpu0InstrInfo::
copyPhysReg(MachineBasicBlock &MBB,
      MachineBasicBlock::iterator I, DebugLoc DL,
      unsigned DestReg, unsigned SrcReg,
      bool KillSrc) const {
  unsigned Opc = 0, ZeroReg = 0;

  if (Cpu0::CPURegsRegClass.contains(DestReg)) { // Copy to CPU Reg.
  ...
  else if (SrcReg == Cpu0::ACC)
    Opc = Cpu0::MFACC, SrcReg = 0;
  }
  else if (Cpu0::CPURegsRegClass.contains(SrcReg)) { // Copy from CPU Reg.
  ...
  else if (DestReg == Cpu0::ACC)
    Opc = Cpu0::MTACC, DestReg = 0;
  }
  ...
}
```

Explain the code as follows,

```
ld  $3, 12($sp) // $3 is a
ld  $4, 16($sp) // $4 is b

mtacc $4      // Move b To Acc
// After meet first a+b IR, it call this pattern,
//  def : Pat<(add CPURegs:$lhs, CPURegs:$rhs),
//        (ADD (MTACC CPURegs:$lhs), CPURegs:$rhs)>;
// After this pattern translation, the DestReg class change from CPU0Regs to
//  RACC according the following code of copyPhysReg(). copyPhysReg() is called
//  when DestReg and SrcReg belong to different Register Class.
//
//  if (DestReg)
```

```
//    MIB.addReg(DestReg, RegState::Define);
//
//  if (ZeroReg)
//    MIB.addReg(ZeroReg);
//
//  if (SrcReg)
//    MIB.addReg(SrcReg, getKillRegState(KillSrc));

add $3      // Add a To Acc
// Apply this pattern since the DestReg class is RACC
//  def : Pat<(add RACC:$lhs, CPURegs:$rhs),
//        (ADD RACC:$lhs, CPURegs:$rhs)>;

ld  $4, 4($sp)  // $4 is d
add $4      // Add d To Acc
// Apply the pattern as above since the DestReg class is RACC

mfacc $3    // Move Acc to $3
// Compiler/backend can use ADDiu since e is 5. But it add MFACC before ADDiu
//  since the DestReg class is RACC. Translate to CPU0Regs class by MFACC and
//  apply ADDiu since ADDiu use CPU0Regs as operands.
addiu $3, $3, 5 // Add e(=5) to $3
st  $3, 8($sp)
```

CHAPTER

FIFTEEN

TODO LIST

Todo

Add info about LLVM documentation licensing.

(The *original entry* is located in /home/cschen/test/1/lbd/source/about.rst, line 118.)

Todo

Find information on debugging LLVM within Xcode for Macs.

(The *original entry* is located in /home/cschen/test/1/lbd/source/install.rst, line 37.)

Todo

Find information on building/debugging LLVM within Eclipse for Linux.

(The *original entry* is located in /home/cschen/test/1/lbd/source/install.rst, line 38.)

Todo

Fix centering for figure captions.

(The *original entry* is located in /home/cschen/test/1/lbd/source/install.rst, line 47.)

Todo

I might want to re-edit the following paragraph

(The *original entry* is located in /home/cschen/test/1/lbd/source/llvmstructure.rst, line 677.)

CHAPTER

SIXTEEN

BOOK EXAMPLE CODE

The example code is available in:

http://jonathan2251.github.com/lbd/LLVMBackendTutorialExampleCode.tar.gz

CHAPTER

SEVENTEEN

ALTERNATE FORMATS

The book is also available in the following formats: