Write An LLVM Backend Tutorial For Cpu0

Release 3.2.2

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

January 11, 2013

CONTENTS

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 Revision history

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.

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.

1.3 Licensing

Todo

Add info about LLVM documentation licensing.

1.4 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.

1.5 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

1.6 Outline of chapters

Chapter of Cpu0 Instruction and LLVM Target Description:

Introduce Cpu0 and LLVM architecture, Target Description td, cmake and LLVMBuild files, Target Registration. And shows how to start a backend and build it through our example code Cpu0. Around 750 lines of source code are added by the end of this chapter.

Chapter of Back end Structure:

Introduce the backend structure by UML graph, and continue to add code in each section. Around 2300 lines of source code added in this chapter. These 2300 lines of source code is common in backend design. Many code are same from backend to backend except the backend name. At end of this chapter, an LLVM backend structure for cpu0 is created with three instructions support to generate the assembly output file.

Chapter of Other instructions:

Over ten of C operators and their corresponding LLVM IR instructions introduced in this chapter. Around 345 lines of source code (most are in .td Target Description file) are added. With these 345 lines, it can translate **+, -, *, /, &, |, ^, <<, >>, !** and **%** into assembly code. The `llc` debug option and debug tool **Graphviz** which supported by LLVM introduced in this chapter.

Chapter of Generate obj file:

Add the obj file format generated support in cpu0 backend, and introduce the cpu0 backend Target Registration structure. The Registration structure is common in backend design. With these 700 lines of code added, cpu0 can handle big and little endian obj file generated.

Chapter of Global variable, struct and array:

Global variable, struct and array support added in this chapter. Over 300 lines of source code needed in global variable access while 345 lines needed for ten of operators in chapter of Other instructions. The cpu0 support PIC and static addressing mode. These two modes are explained in this chapter.

Chapter of Control flow statement:

The **if, else, while, for, goto** flow control statements support is added in this chapter. Around 150 lines of source code added in this chapter.

Chapter of Function call:

The stack frame, incoming & outgoing arguments, and LLVM corresponding handle functions introduced in this chapter. Over 700 lines of source code added in this chapter.

Chapter of ELF:

Like many backend. Cpu0 support ELF obj format generated. The ELF format and binutils tools (not a part of LLVM) introduced and demonstrate backend designers how to use this ELF tool to verify and analysis the backend obj.

Chapter of Appendix A:

Include the LLVM source code, development tools installation, and environment setting on iMac and Linux platform.

CHAPTER

TWO

GETTING STARTED: INSTALLING LLVM AND THE CPU0 EXAMPLE CODE

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/3.1/ 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/3.1.test/cpu0/1 which contains our cpu0 backend program and without clang and clang++.

2.1 Install LLVM 3.1 release on Linux

Our tutorial is based on LLVM 3.1 release. We need LLVM, clang and compiler-rt. In this section, we will show the command line steps to build and install LLVM on your Linux machine. We use the following names to denote paths specific to your enviroment.

`LLVM_SRC`

This is the top level directory of the LLVM source tree.

`Example_SRC`

This is the top level directory where our example code lives. In our tutorial, the example is a git submodule stored in `lbd-example` directory.

`Debug_OBJ`

This is the top level directory of the LLVM object tree (i.e. the tree where object files and compiled programs will be placed). We build LLVM in debug mode here.

`INSTALL`

The resulting binary will be installed in this directory.

If you are using IDE, please see *appendix* for more information.

1. Download the source code.

[1] http://www.llvm.org/docs/CMake.html

```
$ wget http://llvm.org/releases/3.1/llvm-3.1.src.tar.gz
$ wget http://llvm.org/releases/3.1/clang-3.1.src.tar.gz
$ wget http://llvm.org/releases/3.1/compiler-rt-3.1.src.tar.gz
$ tar xvf llvm-3.1.src.tar.gz
$ tar xvf clang-3.1.src.tar.gz -C llvm-3.1.src/tools/
$ tar xvf compiler-rt-3.1.src.tar.gz -C llvm-3.1.src/projects/
$ mv llvm-3.1.src/tools/clang-3.1.src llvm-3.1.src/tools/clang
$ mv llvm-3.1.src/projects/compiler-rt-3.1.src llvm-3.1.src/projects/compiler-rt
```

2. Build LLVM

```
$ mkdir obj; cd obj
$ ../llvm-3.1.src/configure --prefix=$INSTALL
  --enable-targets=host
$ make -j; make install
```

3. Set $PATH

```
$ echo "PATH=$INSTALL/bin:$PATH" > $HOME/.bash_profile
```

2.2 Install cpu0 debug build on Linux

1. Download the example code.

```
$ git clone git://github.com/azru0512/lbd-example.git example
```

2. Make a new directory.

```
$ mkdir -p cpu0/2/src; cd cpu0/2/src
$ cp -rf $LLVM_SRC/* .
```

3. Update top-level source code.

```
$ cp -rf $Example_SRC/top-level/* .
```

4. Update subdirecotry `lib/Target`.

```
$ cp -rf $Example_SRC/2/Cpu0/ lib/Target/
```

5. Remove clang source tree since we don't need it at this moment. Build LLVM in debug mode.

```
$ cd ..
$ mkdir debug; cd debug
$ rm -rf ../src/tools/clang/
$ cmake -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_C_COMPILER=clang \
  -DLLVM_TARGETS_TO_BUILD="Mips;Cpu0" \
  -DCMAKE_BUILD_TYPE=Debug -G "Unix Makefiles" ../src/
$ make
```

6. Debug

```
$ gdb bin/llc
```

CHAPTER

THREE

INTRODUCE CPU0 AND LLVM

Before you start, you should know that you can always examine existing LLVM backend code and attempt to port what you find for your own target architecture . The majority of this code can be found in the /lib/Target directory of your root LLVM directory. As most major RISC instruction set architectures have some similarities, this may be the avenue you might try if you are both an experienced programmer and knowledgable of compiler backends. However, there is a steep learning curve and you may easily get held up 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, this can easily become a headache. 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 does not produce desirable output using the output provided by LLVM.

This chapter shows you the cpu0 instruction format first. Next, the llvm structure is introduced to you by copy and paste the related article from llvm web site. The llvm structure introduced here is extracted from the asop web site. You can read the whole article from the asop web site. After that we will show you how to write register and instruction definitions (Target Description File) which will be used in next chapter.

3.1 Cpu0 Architecture

We copy and redraw figures in english in this section. This web site [1] is chinese version and here [2] is english version.

3.1.1 Introduction

CPU0 is a 32-bit processor which has registers R0 .. R15, IR, MAR, MDR, etc., and its structure is shown below.

Uses of each register as follows:

3.1.2 Instruction Set

The CPU0 instruction divided into three types, L-type usually load the saved instruction, A-type arithmetic instruction-based J-type usually jump instruction, the following figure shows the three types of instruction encoding format.

The following is the CPU0 processor's instruction table format

In the second edition of CPU0_v2 we fill the following command:

[1] http://ccckmit.wikidot.com/ocs:cpu0
[2] 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

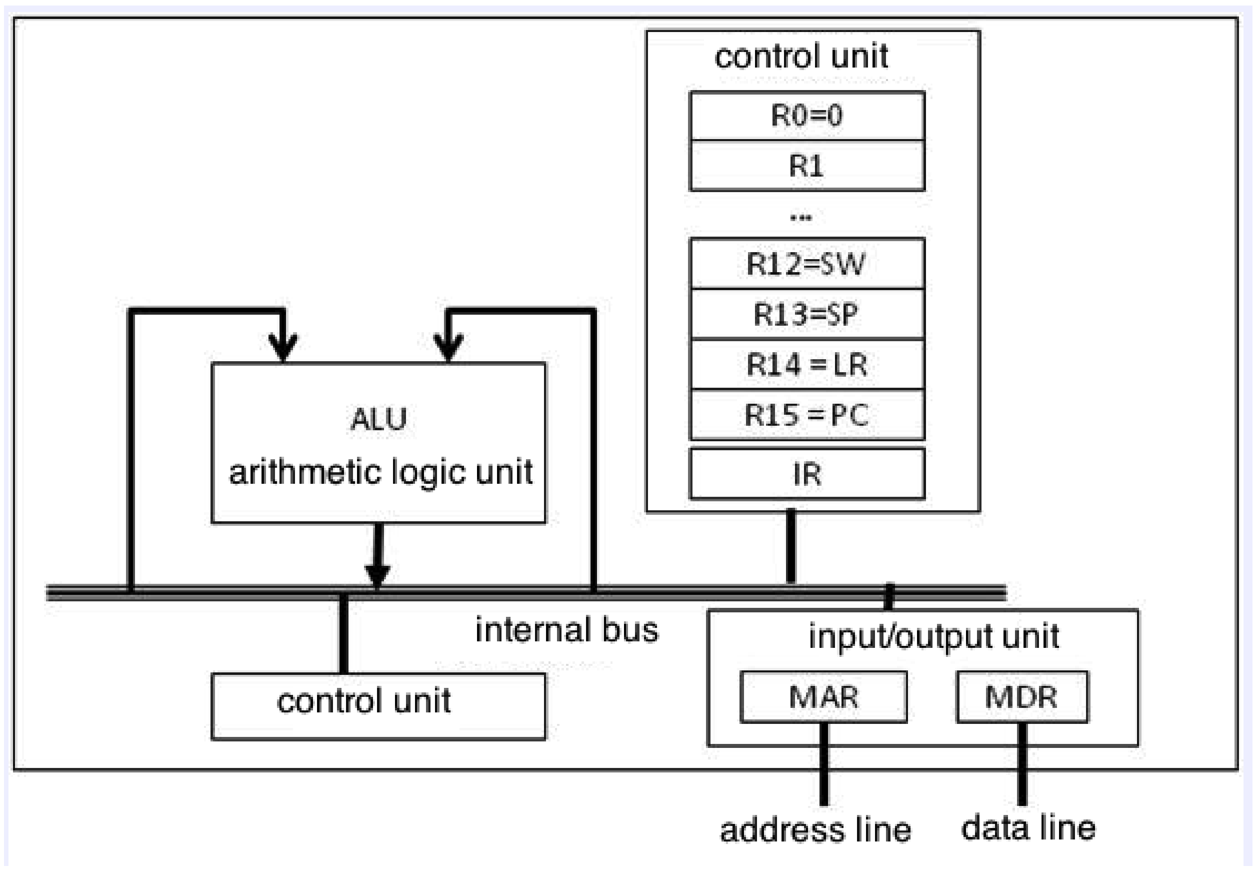
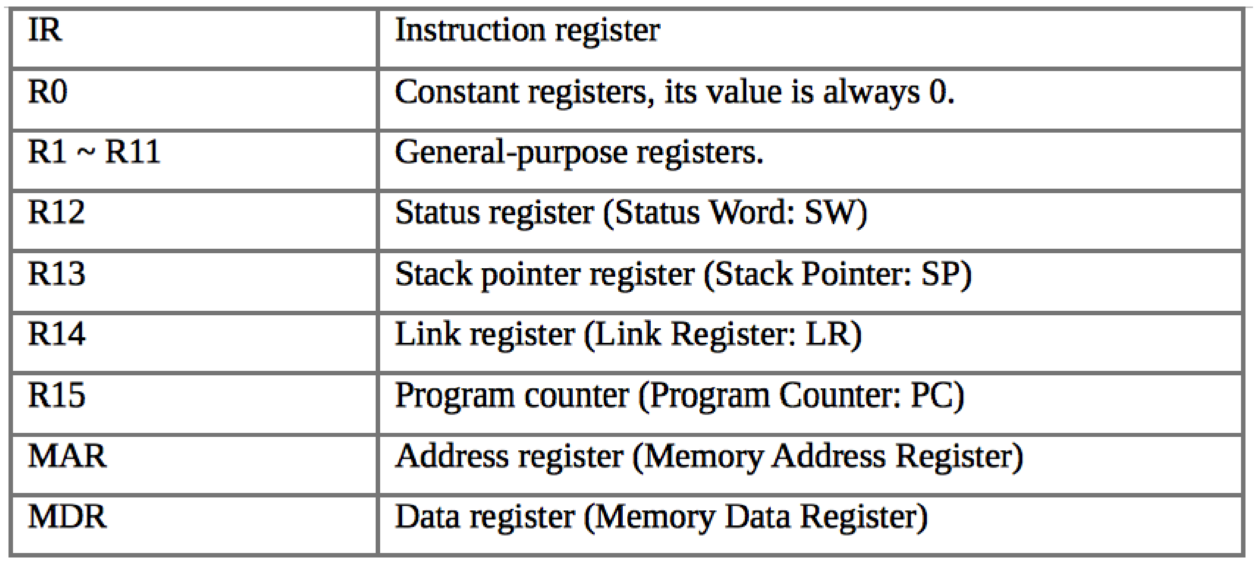


Figure 3.1: The structure of the processor of CPU0



IR	Instruction register
R0	Constant registers, its value is always 0.
R1 ~ R11	General-purpose registers.
R12	Status register (Status Word: SW)
R13	Stack pointer register (Stack Pointer: SP)
R14	Link register (Link Register: LR)
R15	Program counter (Program Counter: PC)
MAR	Address register (Memory Address Register)
MDR	Data register (Memory Data Register)

Figure 3.2: Cpu0 registers table

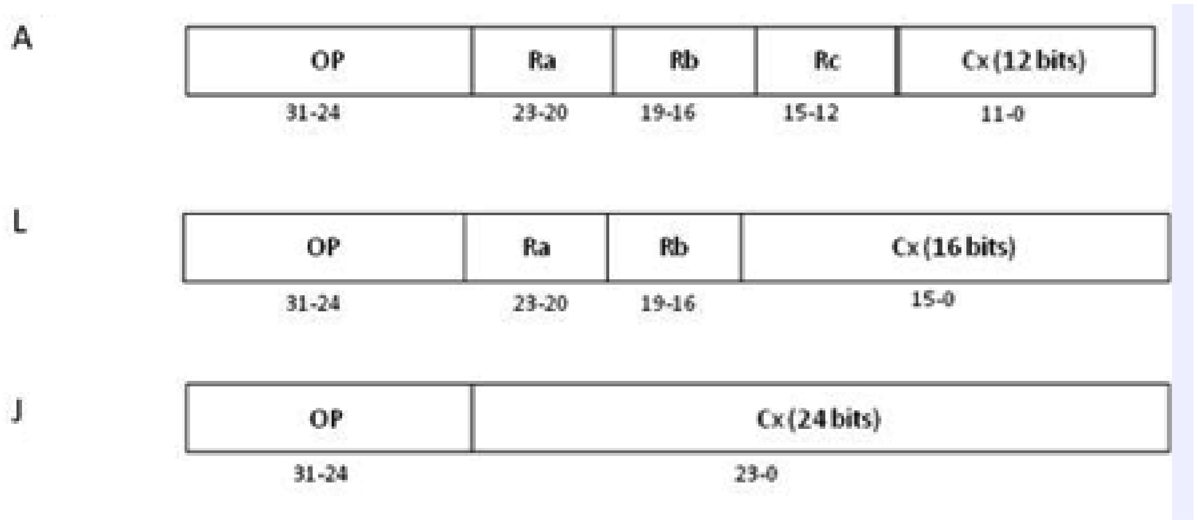


Figure 3.3: CPU0 three instruction formats

3.1.3 Status Register

CPU0 status register contains the state of the N, Z, C, V, and I, T and other interrupt mode bit. Its structure is shown below.

When CMP Ra, Rb instruction execution, the state flag will thus change.

If Ra> Rb, then the setting state of N = 0, Z = 0. If Ra <Rb, it will set the state of N = 1, Z = 0. If Ra = Rb, then the setting state of N = 0, Z = 1.

So conditional jump the JGT, JLT, JGE, JLE, JEQ, JNE instruction jumps N, Z flag in the status register.

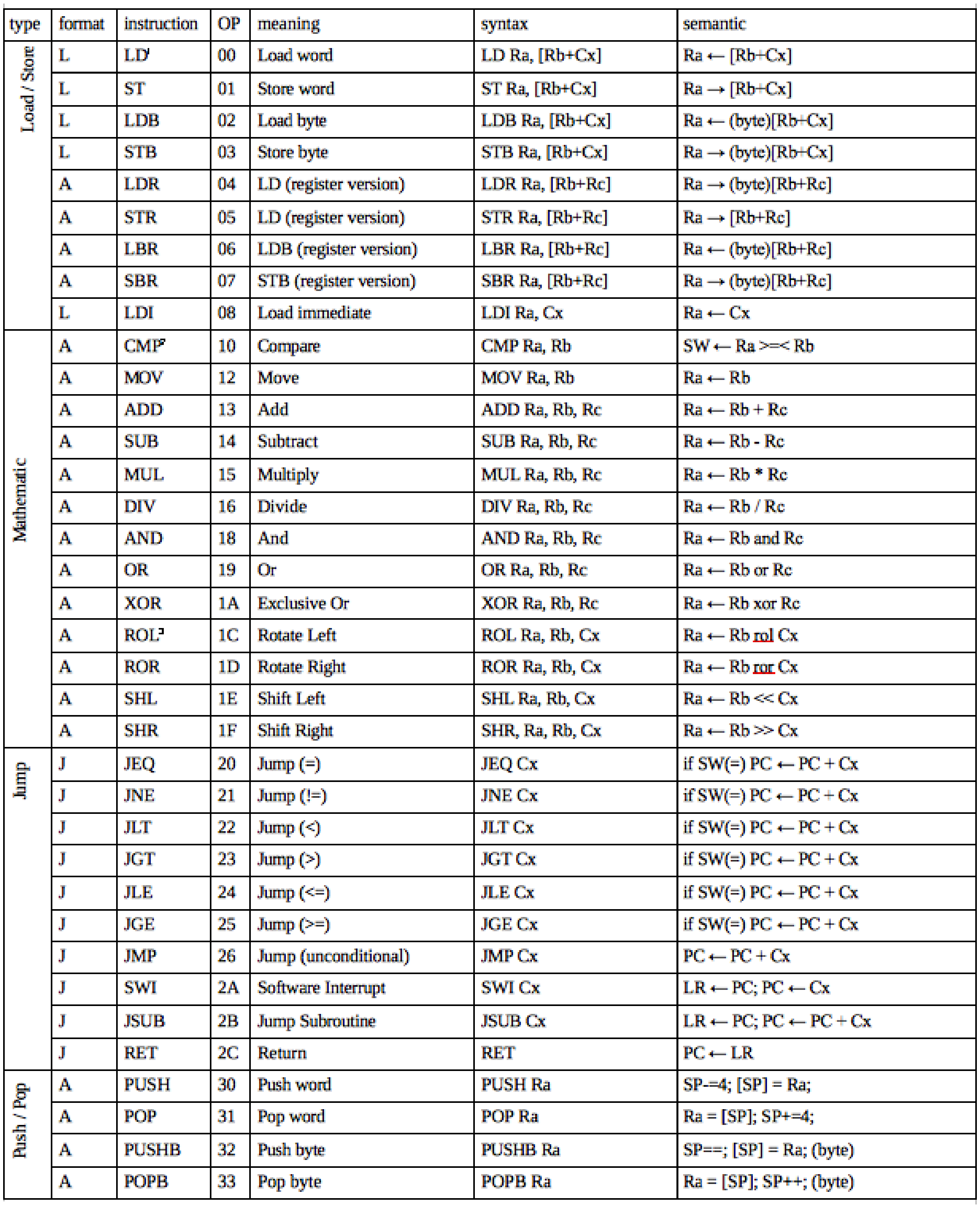
3.1.4 Pipeline

CPU0 has three stage pipeline: Instruction fetch, Decode and Execution.

1. Instruction fetch
   - Action 1. The instruction fetch: IR = [PC]
   - Action 2. Update program counter: PC = PC + 4
2. Decode
   - **Action 3. Decode: Control unit decodes IR, then set data flow switch** and ALU operation mode.
3. Execute
   - **Action 4. Execute: Data flow into ALU. After ALU done the operation,** the result stored back into destination register.

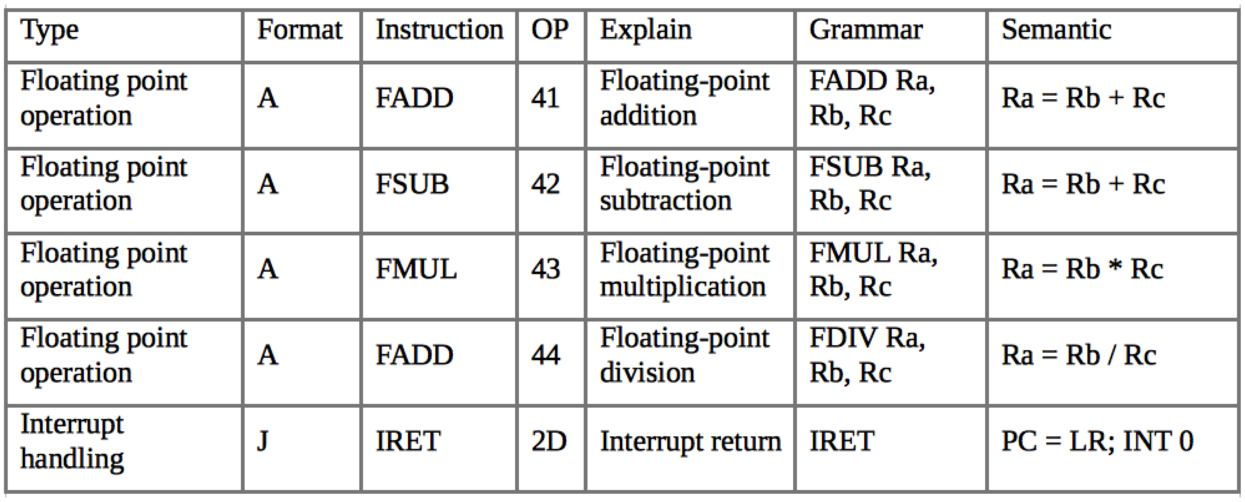
3.1.5 Instruction Change

We have recognized the ldi instruction is a bad design and replace it with mips instruction addiu. The reason we replace ldi with addiu is that ldi use only one register even though ldi is L type format and has two registers, as 3.7. Mips addiu which allow programmer to do load constant to register like ldi, and add constant to a register. So, it's powerful and fully contains the ldi ability.



type	format	instruction	OP	meaning	syntax	semantic
Load / Store	L	LD[1]	00	Load word	LD Ra, [Rb+Cx]	Ra ← [Rb+Cx]
	L	ST	01	Store word	ST Ra, [Rb+Cx]	Ra → [Rb+Cx]
	L	LDB	02	Load byte	LDB Ra, [Rb+Cx]	Ra ← (byte)[Rb+Cx]
	L	STB	03	Store byte	STB Ra, [Rb+Cx]	Ra → (byte)[Rb+Cx]
	A	LDR	04	LD (register version)	LDR Ra, [Rb+Rc]	Ra → (byte)[Rb+Rc]
	A	STR	05	LD (register version)	STR Ra, [Rb+Rc]	Ra → [Rb+Rc]
	A	LBR	06	LDB (register version)	LBR Ra, [Rb+Rc]	Ra ← (byte)[Rb+Rc]
	A	SBR	07	STB (register version)	SBR Ra, [Rb+Rc]	Ra → (byte)[Rb+Rc]
	L	LDI	08	Load immediate	LDI Ra, Cx	Ra ← Cx
Mathematic	A	CMP[2]	10	Compare	CMP Ra, Rb	SW ← Ra >=< Rb
	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, Rc	Ra ← Rb / Rc
	A	AND	18	And	AND Ra, Rb, Rc	Ra ← Rb and Rc
	A	OR	19	Or	OR Ra, Rb, Rc	Ra ← Rb or Rc
	A	XOR	1A	Exclusive Or	XOR Ra, Rb, Rc	Ra ← Rb xor Rc
	A	ROL[3]	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
Jump	J	JEQ	20	Jump (=)	JEQ Cx	if SW(=) PC ← PC + Cx
	J	JNE	21	Jump (!=)	JNE Cx	if SW(=) PC ← PC + Cx
	J	JLT	22	Jump (<)	JLT Cx	if SW(=) PC ← PC + Cx
	J	JGT	23	Jump (>)	JGT Cx	if SW(=) PC ← PC + Cx
	J	JLE	24	Jump (<=)	JLE Cx	if SW(=) PC ← PC + Cx
	J	JGE	25	Jump (>=)	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 Subroutine	JSUB Cx	LR ← PC; PC ← PC + Cx
	J	RET	2C	Return	RET	PC ← LR
Push / Pop	A	PUSH	30	Push word	PUSH Ra	SP-=4; [SP] = Ra;
	A	POP	31	Pop word	POP Ra	Ra = [SP]; SP+=4;
	A	PUSHB	32	Push byte	PUSHB Ra	SP==; [SP] = Ra; (byte)
	A	POPB	33	Pop byte	POPB Ra	Ra = [SP]; SP++; (byte)

Figure 3.4: CPU0 instruction table



Type	Format	Instruction	OP	Explain	Grammar	Semantic
Floating point operation	A	FADD	41	Floating-point addition	FADD Ra, Rb, Rc	Ra = Rb + Rc
Floating point operation	A	FSUB	42	Floating-point subtraction	FSUB Ra, Rb, Rc	Ra = Rb + Rc
Floating point operation	A	FMUL	43	Floating-point multiplication	FMUL Ra, Rb, Rc	Ra = Rb * Rc
Floating point operation	A	FADD	44	Floating-point division	FDIV Ra, Rb, Rc	Ra = Rb / Rc
Interrupt handling	J	IRET	2D	Interrupt return	IRET	PC = LR; INT 0

Figure 3.5: CPU0_v2 instruction table

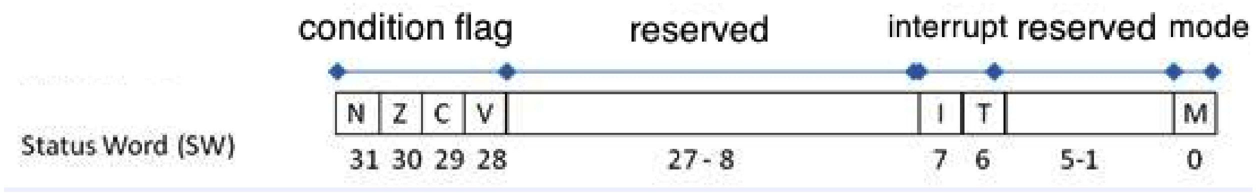
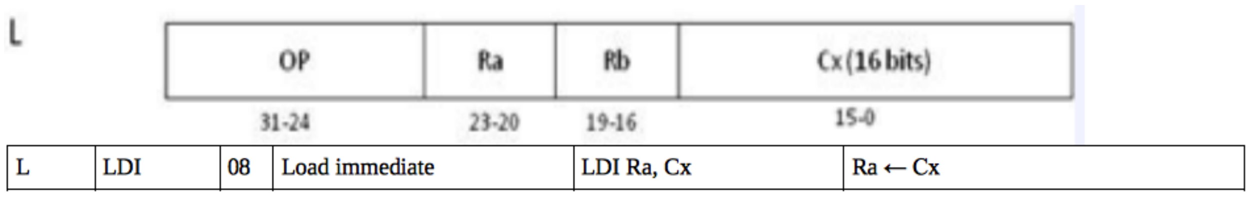


Figure 3.6: CPU0 status register

The two instruction formats are shown in figure 3.7 and figure 3.8.



L	LDI	08	Load immediate	LDI Ra, Cx	Ra ← Cx

Figure 3.7: Cpu0 ldi instruction

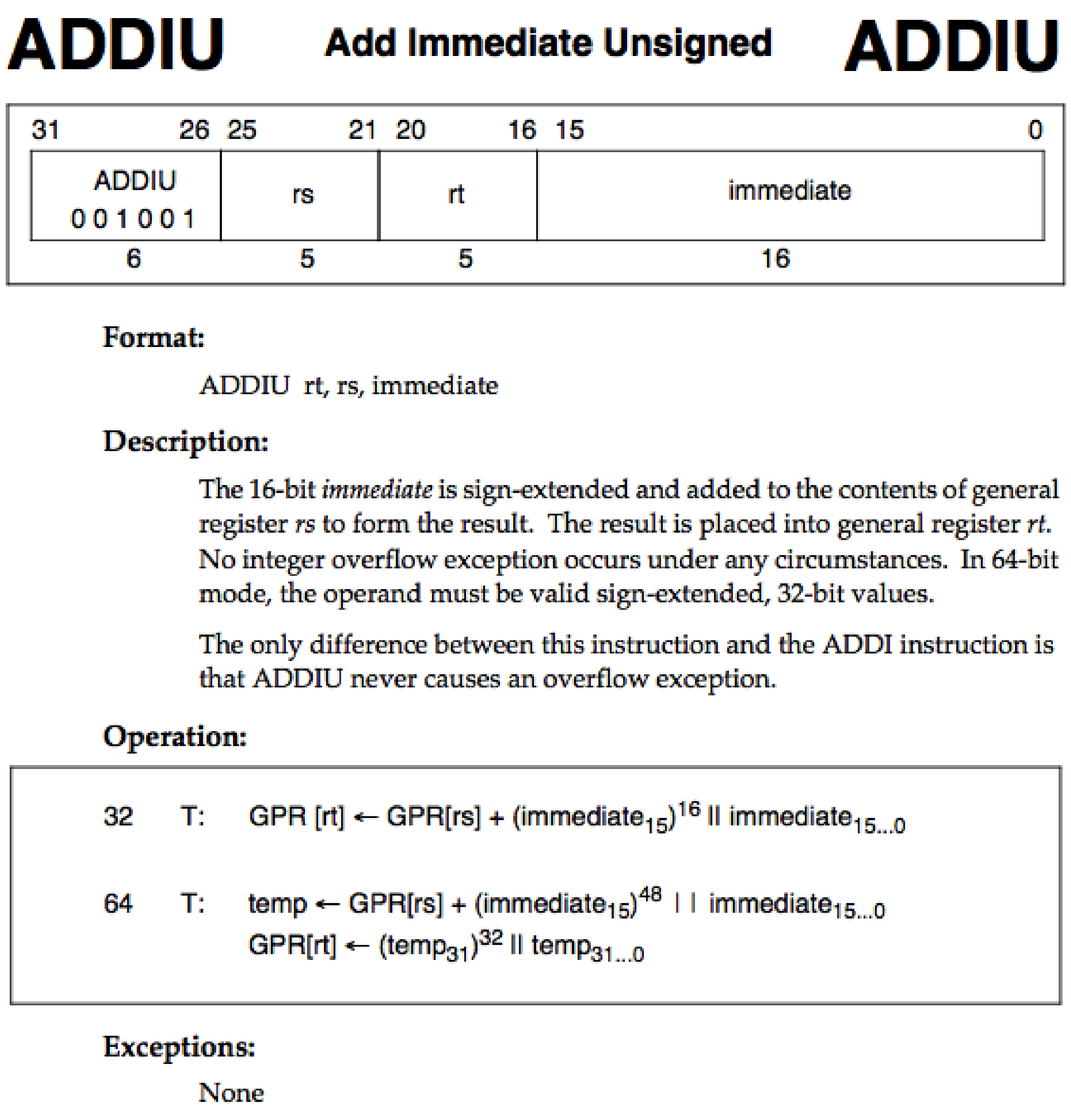
From figure 3.7 and figure 3.8, you can find `ldi $Ra, 5` can be replaced by `addiu $Ra, $zero, 5`. And more, `addiu` can do `addiu $Ra, $Rb, 5` which add $Rb and 5 then save to $Ra, but ldi cannot.

As a CPU design, it's common to redesign CPU instruction when find a better solution during design the compiler backend for that CPU. So, we add addiu instruction to cpu0. The cpu0 is my brother's work, I will find time to talk with him.

3.2 LLVM Structure

3.2.1 Overview

首先介紹編譯的基本知識，編譯流程可以簡化成 figure 3.9 。 源語言 (代碼) 經過編譯器前端解析成編譯器內部的中間表示，並在此階段優化代碼，最後 產生目標平台語言。figure 3.10 展示了更進一步的編譯流程。



ADDIU **Add Immediate Unsigned** **ADDIU**

31 26	25 21	20 16	15 0
ADDIU 001001	rs	rt	immediate
6	5	5	16

Format:

ADDIU rt, rs, immediate

Description:

The 16-bit *immediate* is sign-extended and added to the contents of general register *rs* to form the result. The result is placed into general register *rt*. No integer overflow exception occurs under any circumstances. In 64-bit mode, the operand must be valid sign-extended, 32-bit values.

The only difference between this instruction and the ADDI instruction is that ADDIU never causes an overflow exception.

Operation:

```
32   T:   GPR [rt] ← GPR[rs] + $(immediate_{15})^{16}$ || $immediate_{15...0}$

64   T:   temp ← GPR[rs] + $(immediate_{15})^{48}$ || $immediate_{15...0}$
          GPR[rt] ← $(temp_{31})^{32}$ || $temp_{31...0}$
```

Exceptions:

None

Figure 3.8: Mips addiu instruction format

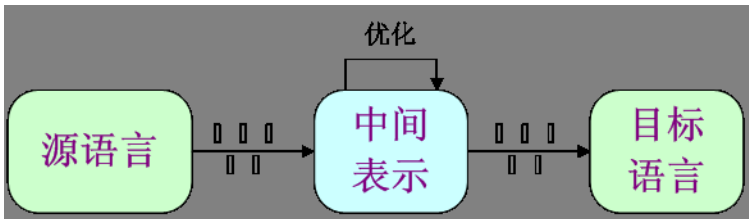


Figure 3.9: Compilation Pipeline

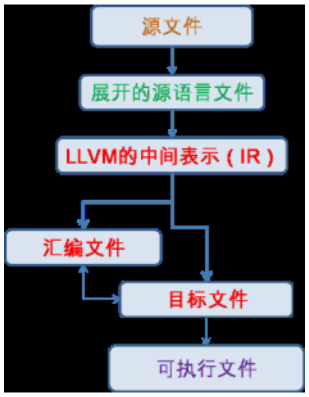


Figure 3.10: Compilation Flow

LLVM 後端的四大功能標明在 figure 3.11 。

1. 將 LLVM IR 編譯成匯編文本文件: 傳統的靜態編譯。
2. 將 LLVM IR 編譯成目標二進制文件: .o file writer。
3. 將匯編文本文件轉譯為目標二進製文件: 匯編。
4. 將目標二進制文件還原為匯編文本文件: 反匯編。

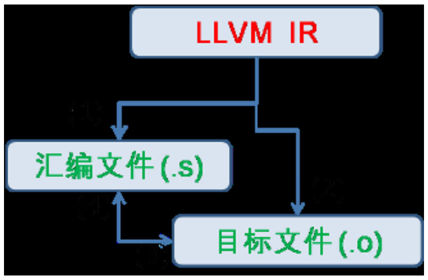


Figure 3.11: LLVM 後端

3.2.2 Detail

Following came from AOSA [3].

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 (figure 3.12). 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.

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.

[3] http://www.aosabook.org/en/llvm.html

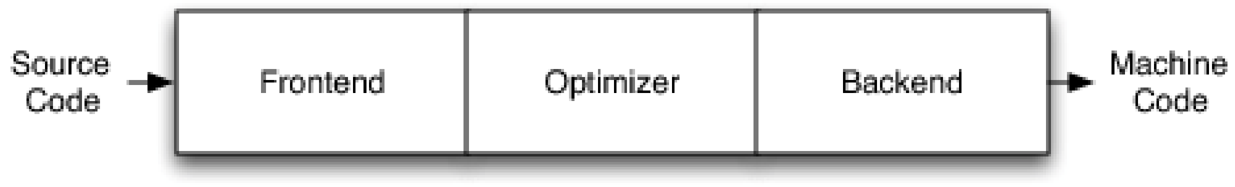


Figure 3.12: Three major components of a Three Phase Compiler

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 3.13.

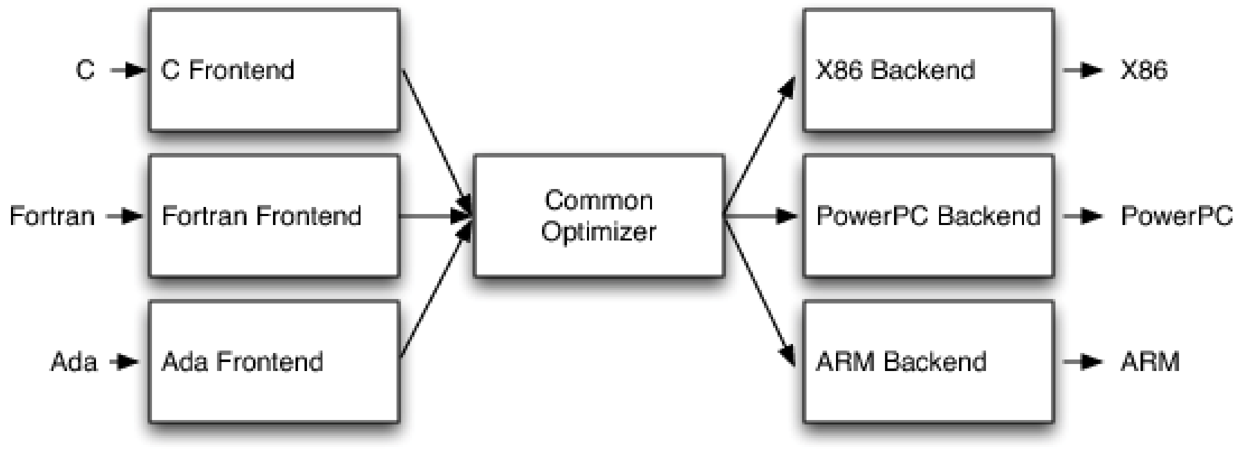


Figure 3.13: Retargetablity

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.

3.2.3 Introduce Target Description File

Note: This section comes from AOSA. http://www.ituring.com.cn/article/4206

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 3.14 .

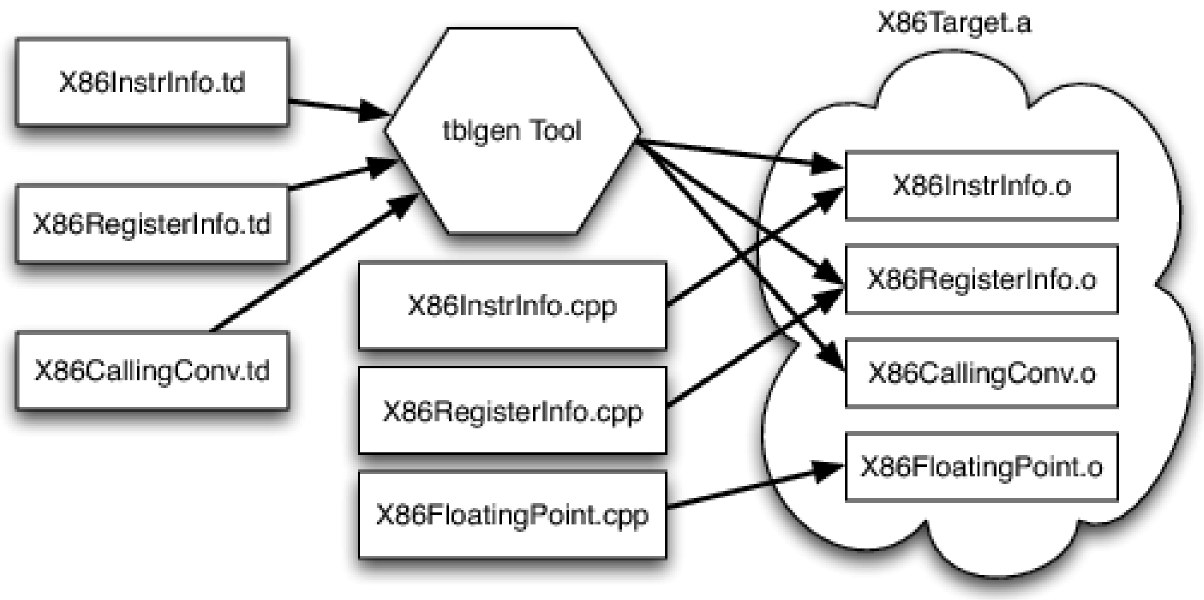


Figure 3.14: Simplified x86 Target Definition

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]> { ... }
```

3.2.4 Writing Target Description File

LLVM use target description files (.td) to describe target register file and instruction format. After you writing up those .td files, LLVM generates C++ files (*.inc) by `llvm-tblgen`. Those *.inc files are C++ files with table driven in concept. [4] is the web site.

You should write one `XX.td` under your target subdirectory, it then includes other target description files defining various target information. The syntax of target description file is pretty similar to C++. Below is `Cpu0.td`.

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

include "llvm/Target/Target.td"
```

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

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

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

def Cpu0InstrInfo : InstrInfo;

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

As shown above, `Cpu0.td` includes `Cpu0RegisterInfo.td` which defines the Cpu0 register file.

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

// We have banks of 16 registers each.
class Cpu0Reg<string n> : Register<n> {
  field bits<4> Num;
  let Namespace = "Cpu0";
}

// Cpu0 CPU Registers
class Cpu0GPRReg<bits<4> num, string n> : Cpu0Reg<n> {
  let Num = num;
}

//===----------------------------------------------------------------------===//
//  Registers
//===----------------------------------------------------------------------===//

let Namespace = "Cpu0" in {
  // General Purpose Registers
  def ZERO : Cpu0GPRReg< 0, "ZERO">, DwarfRegNum<[0]>;
  def AT   : Cpu0GPRReg< 1, "AT">,   DwarfRegNum<[1]>;
  def V0   : Cpu0GPRReg< 2, "2">,    DwarfRegNum<[2]>;
  def V1   : Cpu0GPRReg< 3, "3">,    DwarfRegNum<[3]>;
  def A0   : Cpu0GPRReg< 4, "4">,    DwarfRegNum<[6]>;
  def A1   : Cpu0GPRReg< 5, "5">,    DwarfRegNum<[7]>;
  def T9   : Cpu0GPRReg< 6, "6">,    DwarfRegNum<[6]>;
  def S0   : Cpu0GPRReg< 7, "7">,    DwarfRegNum<[7]>;
  def S1   : Cpu0GPRReg< 8, "8">,    DwarfRegNum<[8]>;
  def S2   : Cpu0GPRReg< 9, "9">,    DwarfRegNum<[9]>;
  def GP   : Cpu0GPRReg< 10, "GP">,  DwarfRegNum<[10]>;
  def FP   : Cpu0GPRReg< 11, "FP">,  DwarfRegNum<[11]>;
  def SW   : Cpu0GPRReg< 12, "SW">,   DwarfRegNum<[12]>;
  def SP   : Cpu0GPRReg< 13, "SP">,   DwarfRegNum<[13]>;
  def LR   : Cpu0GPRReg< 14, "LR">,   DwarfRegNum<[14]>;
  def PC   : Cpu0GPRReg< 15, "PC">,   DwarfRegNum<[15]>;
}

//===----------------------------------------------------------------------===//
// 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, SW, SP, LR, PC)>;
```

In C++ the data layout is declared by class. Declaration tells the variable layout; definition allocates memory for the variable. For example,

```
class Date {          // declare Date
    int year, month, day;
};
Date date;          // define(instance) date
```

Just like C++ class, the keyword **"class"** is used for declaring data structure layout. `Cpu0Reg<string n>` declare a derived class from `Register<n>` which is declared by llvm already, where n is the argument of type string. In addition to inherited from all the fields of Register class, Cpu0Reg add a new field "Num" of type 4 bits. Namespace is same with C++ namespace. **"Def"** is used by define(instance) a concrete variable.

As above, we define a ZERO register which type is Cpu0GPRReg, it's field Num is 0 (4 bits) and field n is **"ZERO"** (declared in Register class). Note the use of **"let"** expressions to override values that are initially defined in a super-class. For example, let Namespace = **"Cpu0"** in class Cpu0Reg, will override Namespace declared in Register class. The Cpu0RegisterInfo.td also define that CPURegs is a variable for type of RegisterClass, where the RegisterClass is a llvm built-in class. The type of RegisterClass is a set/group of Register, so CPURegs variable is defined with a set of Register.

`Cpu0InstrFormats.td` defines Cpu0 instruction format. We define a base class `Cpu0Inst`, then define class `FA`, `FI` and `FJ` for each Cpu0 instruction types.

```
// Generic Cpu0 Format
class Cpu0Inst<dag outs, dag ins, string asmstr, list<dag> pattern,
               InstrItinClass itin, Format f>: Instruction
{
  field bits<32> Inst;
  Format Form = f;

  let Namespace = "Cpu0";

  let Size = 4;

  bits<8> Opcode = 0;

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

  let OutOperandList = outs;
  let InOperandList  = ins;

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

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

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

  let DecoderNamespace = "Cpu0";

  field bits<32> SoftFail = 0;
}

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

class FA<bits<8> op, dag outs, dag ins, string asmstr,
         list<dag> pattern, InstrItinClass itin>:
      Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmA>
{
  bits<4>  ra;
  bits<4>  rb;
  bits<4>  rc;
  bits<12> imm12;

  let Opcode = op;

  let Inst{23-20} = ra;
  let Inst{19-16} = rb;
  let Inst{15-12} = rc;
  let Inst{11-0}  = imm12;
}

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

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

  let Opcode = op;

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

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

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

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

The cpu0 instructions td is named to Cpu0InstrInfo.td which contents as follows,

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

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

//===----------------------------------------------------------------------===//
// Pattern fragment for load/store
//===----------------------------------------------------------------------===//
class AlignedLoad<PatFrag Node> :
  PatFrag<(ops node:$ptr), (Node node:$ptr), [{
  LoadSDNode *LD = cast<LoadSDNode>(N);
  return LD->getMemoryVT().getSizeInBits()/8 <= LD->getAlignment();
}]>;
class AlignedStore<PatFrag Node> :
  PatFrag<(ops node:$val, node:$ptr), (Node node:$val, node:$ptr), [{
  StoreSDNode *SD = cast<StoreSDNode>(N);
  return SD->getMemoryVT().getSizeInBits()/8 <= SD->getAlignment();
}]>;
// Load/Store PatFrags.
def load_a          : AlignedLoad<load>;
def store_a         : AlignedStore<store>;
//===----------------------------------------------------------------------===//
// Instructions specific format
//===----------------------------------------------------------------------===//
// Arithmetic and logical instructions with 2 register operands.
class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,
                  Operand Od, PatLeaf imm_type, RegisterClass RC> :
  FL<op, (outs 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;
}

// Move immediate imm16 to register ra.
class MoveImm<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, $imm16"),
     [(set RC:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {
  let rb = 0;
  let isReMaterializable = 1;
}

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

// Memory Load/Store
let canFoldAsLoad = 1 in
class LoadM<bits<8> op, string instr_asm, PatFrag OpNode, RegisterClass RC,
            Operand MemOpnd, bit Pseudo>:
  FMem<op, (outs RC:$ra), (ins MemOpnd:$addr),
     !strconcat(instr_asm, "\t$ra, $addr"),
     [(set RC:$ra, (OpNode addr:$addr))], IILoad> {
  let isPseudo = Pseudo;
}
class StoreM<bits<8> op, string instr_asm, PatFrag OpNode, RegisterClass RC,
             Operand MemOpnd, bit Pseudo>:
  FMem<op, (outs), (ins RC:$ra, MemOpnd:$addr),
     !strconcat(instr_asm, "\t$ra, $addr"),
     [(OpNode RC:$ra, addr:$addr)], IIStore> {
  let isPseudo = Pseudo;
}
// 32-bit load.
multiclass LoadM32<bits<8> op, string instr_asm, PatFrag OpNode,
                   bit Pseudo = 0> {
```

```
  def #NAME# : LoadM<op, instr_asm, OpNode, CPURegs, mem, Pseudo>;
}
// 32-bit store.
multiclass StoreM32<bits<8> op, string instr_asm, PatFrag OpNode,
                    bit Pseudo = 0> {
  def #NAME# : StoreM<op, instr_asm, OpNode, CPURegs, mem, Pseudo>;
}
//===----------------------------------------------------------------------===//
// Instruction definition
//===----------------------------------------------------------------------===//
//===----------------------------------------------------------------------===//
// Cpu0I Instructions
//===----------------------------------------------------------------------===//
/// Load and Store Instructions
///  aligned
defm LD      : LoadM32<0x00,  "ld",  load_a>;
defm ST      : StoreM32<0x01, "st",  store_a>;

/// 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>;

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

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

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

Here we take `ADDiu` as an example. `ADDiu` is an instance of class `ArithLogicI` which is a subclass of class `FL`.

```
include "Cpu0InstrFormats.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>;
```

```
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.

3.2.5 Writing configuration files

You must have to add one LLVMBuild.txt under `Target/Cpu0` directory. If you want to support cmake, then CMakeLists.txt needed to be added.

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" [5] and "LLVMBuild Guide" [6].

3.3 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.

```
// TargetInfo/Cpu0TargetInfo.cpp
...
Target llvm::TheCpu0Target, llvm::TheCpu0elTarget;
extern "C" void LLVMInitializeCpu0TargetInfo() {
  RegisterTarget<Triple::cpu0,
        /*HasJIT=*/true> X(TheCpu0Target, "cpu0", "Cpu0");

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

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

```
//===-- Cpu0TargetMachine.cpp - Define TargetMachine for Cpu0 -------------===//
...

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

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

```
}
...

//===-- Cpu0MCTargetDesc.cpp - Cpu0 Target Descriptions --------------------===//
...
extern "C" void LLVMInitializeCpu0TargetMC() {
}
```

Please see “Target Registration” [7] for reference.

3.4 Build libraries and td

The llvm3.1 source code is put in /usr/local/llvm/3.1/src and have llvm3.1 release-build in /usr/local/llvm/3.1/configure_release_build. About how to build llvm, please refer [8]. We made a copy from /usr/local/llvm/3.1/src to /usr/local/llvm/3.1.test/cpu0/1/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/. You can search cpu0 without case sensitive to find the modified files by command,

```
[Gamma@localhost cmake_debug_build]$ grep -R -i "cpu0" ../src/
../src/CMakeLists.txt:  Cpu0
../src/lib/Target/LLVMBuild.txt:subdirectories = ARM CellSPU CppBackend Hexagon
MBlaze MSP430 Mips Cpu0 PTX PowerPC Sparc X86 XCore ../src/lib/MC/MCExpr.cpp:
case VK_Cpu0_GPREL: return "GPREL";
...
../src/lib/MC/MCELFStreamer.cpp:    case MCSymbolRefExpr::VK_Cpu0_TLSGD:
...
../src/lib/MC/MCDwarf.cpp:  // AT_language, a 4 byte value.  We use DW_LANG_Cpu0
_Assembler as the dwarf2
../src/lib/MC/MCDwarf.cpp: // MCOS->EmitIntValue(dwarf::DW_LANG_Cpu0_Assembler,
2);
../src/lib/Support/Triple.cpp:  case cpu0:    return "cpu0";
 ...
../src/include/llvm/Support/ELF.h:  EM_LATTICEMICO32 = 138, // RISC processor fo
r Lattice CPU0 architecture
...
```

You can update your llvm working copy by,

```
cp -rf LLVMBackendTutorial/src_files_modified/src/*   yourllvm/workingcopy/sourc
edir/.
```

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

```
[Gamma@localhost cmake_debug_build]$ 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: /usr/local/llvm/3.1.test/cpu0/1/cmake_debug
_build
```

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

```
[Gamma@localhost cmake_debug_build]$ make
...
[100%] Built target gtest_main
```

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

```
[Gamma@localhost cmake_debug_build]$ /usr/local/llvm/3.1.test/cpu0/1/cmake_debug
_build/bin/llc --version
LLVM (http://llvm.org/):
  LLVM version 3.1svn
  DEBUG build with assertions.
  Built Sep 21 2012 (18:27:58).
  Default target: x86_64-unknown-linux-gnu
  Host CPU: penryn

  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" [9] we made. List them as follows again,

```
// Cpu0TargetInfo.cpp
Target llvm::TheCpu0Target, llvm::TheCpu0elTarget;

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

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

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

```
// ch3.cpp
int main()
{
    return 0;
}
```

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

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

Next step, transfer bitcode .bc to human readable text format as follows,

```
[Gamma@localhost InputFiles]$ llvm-dis ch3.bc -o ch3.ll

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

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

```
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,

```
[Gamma@localhost InputFiles]$ /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o ch3.cpu0.s
llc: /usr/local/llvm/3.1.test/cpu0/1/src/tools/llc/llc.cpp:456: int main(int, ch
ar **): Assertion `target.get() && "Could not allocate target machine!"' failed.
Stack dump:
0.       Program arguments: /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc
 -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o ch3.cpu0.s
Aborted (core dumped)
```

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

FOUR

LLVM BACKEND STRUCTURE

This chapter introduces the LLVM backend class inheritance tree and class members first. Next, we follow the LLVM backend structure and add class implementation one by one in each section. In this chapter, we need some compiler knowledge, like DAG and instruction selection, we will introduce them when needed. At the end of this chapter, we will have a backend being able to compile LLVM IR into cpu0 assembly code.

The code added in this chapter are common in every LLVM backend, the only difference are their name. In fact, since Cpu0 is similar to Mips we copy most of the code from the Mips backend. In this chapter, we will focus on the relationship between classes involved in writing a LLVM backend. Once knowing the overall structure, you can quickly create a simple backend from scratch.

4.1 Overview

Note: Following content came from "Design and Implementation of a TriCore Backend for the LLVM Compiler Framework" [1] Chapter 4. I already got author's approve on the Chinese translation work. :-)

As mentioned in Chapter 2, LLVM provides developers with an extensive generic frame work for code generation. It prescribes a fixed directory layout and class hierarchy template. Each backend is located in its own subdirectory within lib/Target, where most of the code to be implemented goes. Apart from this, only a handful of the original files in the LLVM source tree have to be modified in order to integrate the new backend with the existing LLVM codebase.

如同第二章所提到，LLVM 提供開發者一套相當通用的框架用來開發後端。LLVM 提供一個 固定的目錄布局，和類別階層。每一個後端都位在各自位於 `lib/Target` 的子目錄， 大部分的實作代碼都在該子目錄底下。除此之外，只有少部分 LLVM 代碼需要修改以將 新增的後端整合進 LLVM 代碼庫。

The target code generator is divided into several components, each of which is discussed in a separate section. For a better overview, each section contains a list of the source files that comprise the respective component

目標後端被分為數個元件，底下各節將分別加以討論。為了加强讀者整體印象，各節會列出 相應的檔案。

The given class hierarchy specifies a number of abstract base classes with virtual functions and requires the backend developer to implement subclasses for each of them. Most of these classes do not provide any immediate functionality for code generation, but merely give selected information about the characteristics and properties of the target machine. This makes it possible to keep the bulk of the actual algorithms and procedures fully target-independent by accessing all the required target-specific information through the specified interfaces.

LLVM 的類別階層指定數個抽象基礎類別，其中定義數個虛擬函式。後端開發者必須實做前述 抽象類別的子類別，並依據需要實做虛擬函式。大部分的類別僅提供目標的各項資訊和特性， 與後端代碼生成並不相關。這使得演算法和其它程序與底層目標無關，僅透過指定的界面存取 目標資訊。

[1] http://www.opus.ub.uni-erlangen.de/opus/volltexte/2010/1659/pdf/tricore_llvm.pdf

4.2 Code Generation Process

為了瞭解 LLVM 後端的結構，先理解 LLVM 後端是如何生成目標代碼是必要的。 將 LLVM IR 轉成目標代碼，後端需要經過數個步驟 [2] 。

4.2.1 Instruction Selection

目標代碼生成的第一步，是將 LLVM IR 轉成一組 `SelectionDAG` ， 這是一個有向無環圖，其中每一個節點代表一條 LLVM 指令。任兩條指令之間的 definition-use 關係由一條從 using node 指向 defining node 的邊描述，邊上伴隨受影響變數的型別其資訊。 如果指令之間有 control flow dependency ，會以額外的邊加以表示。由於目標並不保證支持 所有 LLVM 提供的型別和操作，前述的 DAG 必須先加以合法化，也就是要先將其轉成目標支持 的型別和操作。合法化的 DAG 再交由 instruction selector 以樣式匹配的方式，創建以目標 指令為內容的 DAG 節點。對於每一個匹配到的 LLVM DAG 節點，相應目標指令的 DAG 節點 會被建立。instruction selector 的輸入和輸出都是 DAG，只是其內容由 LLVM 指令轉成目標指令。

4.2.2 Scheduling and Formation

前述由目標指令所構成的的 DAG 將會在此階段被解構成目標指令序列 (list)。每一個函式以 一個 MachineFunction 加以表示，其中包含 MachineBlock 串列。 MachineBlock 中又包含數條 MachineInstr 。Scheduler 必須決定以何種順序將目標指令寫出，這會受 到許多因素影響，例如: 最小 register pressure，亦即減少因暫存器數量不足，而需要將 暫存器內容暫時搬移至內存的次數。

此時，目標指令序列仍舊保持 SSA 形式，它並非合法的匯編。除了少數例外 (像是將返回值 搬移至 ABI 所規定的暫存器)，所有的指令仍舊操作 LLVM 提供的無限量虛擬暫存器，且所有 存取棧的指令均是存取虛擬的棧空間，而非真正的偏移量。

4.2.3 SSA-based Machine Code Optimization

在分配物理暫存器之前，後端有機會針對保有 SSA 形式的目標指令序列進行 SSA 相關的優化。

4.2.4 Register Allocation

虛擬暫存器將在此階段被消除，改以物理暫存器對應之。LLVM 暫存器分配器將會為每一個虛擬 暫存器分配一個物理暫存器。如果虛擬暫存器數量超過實際可用的物理暫存器，暫存器分配器 會產生 spill code 將某個物理暫存器搬移至內存以供使用。因為暫存器有可能重疊，例如: `%e2` 暫存器分別由 `%d3` 和 `%d2` 構成其上下半部，暫存器分配器必須考慮到這種 情況。虛擬暫存器的消除同時伴隨著 SSA 形式的解構。為建立 SSA 形式所插入的 phi 指令 將被 copy 指令所取代。

4.2.5 Prologue/Epilogue Code Insertion

在暫存器分配器運行之後，我們可以計算每一個函式將會需要多少棧空間。進而在函式的出入口 寫入相對應的 prologue 和 epilogue。之前存取虛擬棧空間的指令，現在可以存取到相對應 棧頂 (sp) 或是棧底 (fp) 指針的偏移量。

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

4.2.6 Late Machine Code Optimizations

在將目標指令寫出之前，可以做窺孔優化 (peephole optimization)，對目標指令做細部調整。

4.2.7 Code Emission

最後，完整的目標代碼被寫出。對於靜態編譯，其結果可以是匯編文本; 對於 JIT 編譯，目標指 令的編碼將被寫入內存。

4.3 General Target Information

每一個 LLVM 後端都必須要提供一個介面供上層原件調用。後端必須實現 LLVMTargetMachine 的子類，提供目標平台必要資訊，並向 LLVM 代碼生成器註冊。

4.3.1 Target Machine Characteristics

類別 `XXTargetMachine` 扮演著中心的角色，它是目標後端和 LLVM 其它原件之間的界面。 它創建並擁有數個物件描述目標平台資訊，LLVM 代碼生成器透過 `XXTargetMachine` 取 得相應資訊。

- DataLayout
- TargetFrameLowering
- TargetInstrInfo 、 TargetLowering 和 TargetRegisterInfo

4.4 Build Example Code

We build our example code first, then add components step by step.

1. Prepare the source code.

```
$ cd cpu0
$ mkdir -p 3/src; cd 3/src
$ cp -rf ../../2/src/* .
$ cp -rf $Example_SRC/3/1/Cpu0/ lib/Target/
```

2. Build

```
$ mkdir debug; cd debug
$ cmake -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_C_COMPILER=clang \
  -DLLVM_TARGETS_TO_BUILD="Mips;Cpu0" \
  -DCMAKE_BUILD_TYPE=Debug -G "Unix Makefiles" ../src/
```

4.4.1 Step 1. Add XXTargetMachine

The role of TargetMachine in the LLVM backend structure is shown in figure 4.1. llc is LLVM static compiler. When you invoke `llc` command with option `-march=XX`, it will trigger the corresponding callback functions.

Target Machine is a base class for targets implemented with the LLVM target-independent code generator. You should define a `XXTargetMachine` class which inherit class `LLVMTargetMachine`, where XX is your target name. In

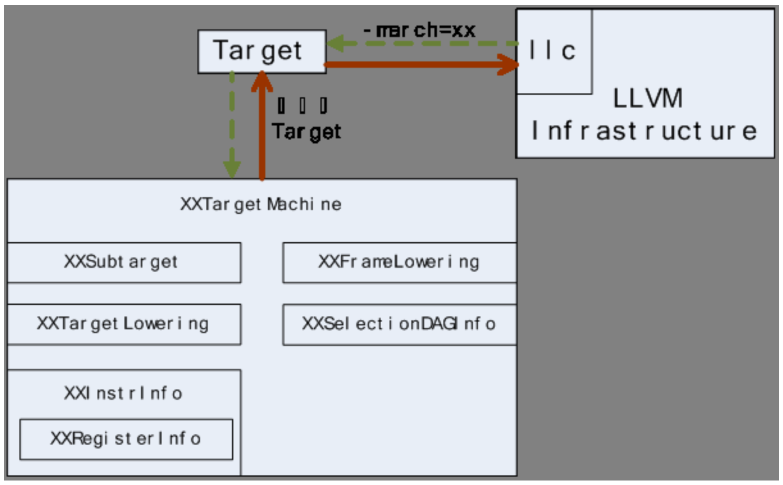


Figure 4.1: The role of TargetMachine

our tutorial we name it as `Cpu0TargetMachine`. Class Cpu0TargetMachine has data members which define target layout, instruction information, frame/stack ...etc. The Cpu0TargetMachine contents as follows,

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

#ifndef CPU0TARGETMACHINE_H
#define CPU0TARGETMACHINE_H

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

namespace llvm {
  class formatted_raw_ostream;

  class Cpu0TargetMachine : public LLVMTargetMachine {
    Cpu0Subtarget       Subtarget;
    const TargetData    DataLayout; // Calculates type size & alignment
    Cpu0InstrInfo       InstrInfo;        //- Instructions
    Cpu0FrameLowering   FrameLowering;        //- Stack(Frame) and Stack direction
    Cpu0TargetLowering  TLInfo;        //- Stack(Frame) and Stack direction
    Cpu0SelectionDAGInfo TSInfo;        //- Map .bc DAG to backend DAG
  public:
    Cpu0TargetMachine(const Target &T, StringRef TT,
                      StringRef CPU, StringRef FS, const TargetOptions &Options,
                      Reloc::Model RM, CodeModel::Model CM,
                      CodeGenOpt::Level OL,
                      bool isLittle);

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

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

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

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

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

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

#endif
```

Due to the OO design of LLVM, most member functions are already implemented by base classes. We only need to override some member functions for our target machine.

After writing up our target description files, `llvm-tblgen` generates C++ files included by others. Take `Cpu0InstrInfo.td` as an example, `llvm-tblgen` generates `Cpu0GenInstrInfo.inc` from

`Cpu0InstrInfo.td`. Then `Cpu0InstrInfo.h` defines `GET_INSTRINFO_HEADER` marco so that it can get code it needs from `Cpu0GenInstrInfo.inc` Below 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`.

```
//- 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

//- Cpu0InstrInfo.h extracts above code by defining GET_INSTRINFO_HEADER
#define GET_INSTRINFO_HEADER
#include "Cpu0GenInstrInfo.inc"

namespace llvm {

class Cpu0InstrInfo : public Cpu0GenInstrInfo {
  Cpu0TargetMachine &TM;
  public:
    explicit Cpu0InstrInfo(Cpu0TargetMachine &TM);
  };
}
```

Reference Write An LLVM Backend web site [3].

```
$ cp -rf $Example_SRC/3/1/Cpu0 lib/Target
```

We add the following files in this section, and we have to modify CMakefile.txt to include thhose new added C++ files.

1. Cpu0TargetMachine.[h,cpp]
2. Cpu0Subtarget.[h,cpp]
3. Cpu0InstrInfo.[h,cpp]
4. Cpu0FrameLowering.[h,cpp]
5. Cpu0TargetLowering.[h,cpp]
6. Cpu0SelectionDAGInfo.[h,cpp]

```
# CMakeLists.txt
...
add_llvm_target(Cpu0CodeGen
  Cpu0ISelLowering.cpp
  Cpu0InstrInfo.cpp
  Cpu0FrameLowering.cpp
  Cpu0Subtarget.cpp
  Cpu0TargetMachine.cpp
  Cpu0SelectionDAGInfo.cpp
  )
```

Please follow commands below to build example code for this section.

1. Prepare the source code.

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

```
$ cd cpu0
$ mkdir -p 3/1/src; cd 3/1/src
$ cp -rf ../../../2/src/* .
$ cp -rf $Example_SRC/3/1/Cpu0/ lib/Target/
```

2. Build

```
$ mkdir debug; cd debug
$ cmake -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_C_COMPILER=clang \
  -DLLVM_TARGETS_TO_BUILD="Mips;Cpu0" \
  -DCMAKE_BUILD_TYPE=Debug -G "Unix Makefiles" ../src/
```

4.4.2 Step 2. Add RegisterInfo

class *Cpu0InstrInfo* has a data member of class *Cpu0RegisterInfo*. This section adds class *Cpu0RegisterInfo*, then includes it in *Cpu0InstrInfo.h* and *Cpu0TargetMachine.h*. Finally, we modify CMakefile.txt to add our new added file *Cpu0RegisterInfo.cpp*.

```
//- Cpu0InstrInfo.h
class Cpu0InstrInfo : public Cpu0GenInstrInfo {
  Cpu0TargetMachine &TM;
  const Cpu0RegisterInfo RI;
public:
  explicit Cpu0InstrInfo(Cpu0TargetMachine &TM);

  virtual const Cpu0RegisterInfo &getRegisterInfo() const;

public:
};

//- Cpu0InstrInfo.cpp
Cpu0InstrInfo::Cpu0InstrInfo(Cpu0TargetMachine &tm)
  :
    TM(tm),
    RI(*TM.getSubtargetImpl(), *this) {}

const Cpu0RegisterInfo &Cpu0InstrInfo::getRegisterInfo() const {
  return RI;
}

//  Cpu0TargetMachine.h
    virtual const Cpu0RegisterInfo *getRegisterInfo()  const {
      return &InstrInfo.getRegisterInfo();
    }

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

Now, let's replace 3/1/Cpu0 with 3/2/Cpu0 of adding register class definition and rebuild.

1. Prepare the source code.

```
$ cd cpu0/3
$ mkdir -p 2/src; cd 2/src
$ cp -rf ../../1/src/* .
$ cp -rf $Example_SRC/3/2/Cpu0/ lib/Target/
```

2. Build

```
$ mkdir debug; cd debug
$ cmake -DCMAKE_CXX_COMPILER=clang++ -DCMAKE_C_COMPILER=clang \
  -DLLVM_TARGETS_TO_BUILD="Mips;Cpu0" \
  -DCMAKE_BUILD_TYPE=Debug -G "Unix Makefiles" ../src/
```

Now, let's run `llc` to compile a LLVM bitcode to see what happen.

1. Prepare LLVM bitcode.

```
$ clang -c example/input/ch3.cpp -emit-llvm -o ch3.bc
```

2. Compile

```
$ llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o ch3.cpu0.s
```

```
llc: /cpu0/3/2/src/lib/CodeGen/LLVMTargetMachine.cpp:78: llvm::LLVMTargetMachine::LLVMTargetMachine(c
Stack dump:
0.      Program arguments: ./bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o ch3.cp
Aborted
```

Look into the assertion in above error message, it tell us that we don't have a ASMPrinter for our target to print assembly. We will add it in next section.

4.4.3 Step 3. Add AsmPrinter

First, wee modify `Cpu0.td` to add support for ASMPrinter. LLVM now has a newer ASMPrinter called MCAsmWriter, we explictly say we will support MCAsmWriter.

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

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

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

def Cpu0 : Target {
  let InstructionSet = Cpu0InstrInfo;
  let AssemblyWriters = [Cpu0AsmWriter];
}
```

`llvm-tblgen` will generate `Cpu0GenAsmWrite.inc` included by `Cpu0InstPrinter.cpp` which will be added soon. `Cpu0GenAsmWrite.inc` implements two member functions: `printInstruction` and `getRegisterName` for class `Cpu0InstPrinter`. Both of them are auto-generated from target description files we added before, i.e., `Cpu0InstrInfo.td` and `Cpu0RegisterInfo.td`. The last thing we have to do is adding class `Cpu0InstPrinter`, then include `Cpu0GenAsmWrite.inc` in it.

```
//-  Cpu0InstPrinter.cpp
#include "Cpu0GenAsmWriter.inc"
```

```
void Cpu0InstPrinter::printRegName(raw_ostream &OS, unsigned RegNo) const {
//- getRegisterName(RegNo) defined in Cpu0GenAsmWriter.inc which came from
//-  Cpu0.td indicate.
  OS << '$' << StringRef(getRegisterName(RegNo)).lower();
}

void Cpu0InstPrinter::printInst(const MCInst *MI, raw_ostream &O,
                                StringRef Annot) {
//- printInstruction(MI, O) defined in Cpu0GenAsmWriter.inc which came from
//-  Cpu0.td indicate.
  printInstruction(MI, O);
  printAnnotation(O, Annot);
}
```

Next, add files below into new subdirectory `MCTargetDesc`.

1. Cpu0AsmPrinter.[h,cpp]
2. Cpu0MCInstLower.[h,cpp]
3. Cpu0BaseInfo.[h,cpp]
4. Cpu0FixupKinds.[h,cpp]
5. Cpu0MCAsmInfo.[h,cpp]

We register our target MCAsmInfo and InstPrinter in `Cpu0MCTargetDesc.cpp`.

```
//-  Cpu0MCTargetDesc.cpp
static MCAsmInfo *createCpu0MCAsmInfo(const Target &T, StringRef TT) {
  MCAsmInfo *MAI = new Cpu0MCAsmInfo(T, TT);

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

  return MAI;
}

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

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

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

Then we register AsmPrinter in `Cpu0AsmPrinter.cpp`.

```
//- Cpu0AsmPrinter.cpp

// Force static initialization.
extern "C" void LLVMInitializeCpu0AsmPrinter() {
  RegisterAsmPrinter<Cpu0AsmPrinter> X(TheCpu0Target);
  RegisterAsmPrinter<Cpu0AsmPrinter> Y(TheCpu0elTarget);
}
```

This sections adds two subdirectories: `InstPrinter` and `MCTargetDesc`. You have to modify top-level LLVM-Build.txt and CMakefile.txt accordingly, and add LLVMBuild.txt and CMakefile.txt into those subdirectories also.

```
//-  top-level LLVMBuild.txt

[common]
subdirectories = InstPrinter MCTargetDesc TargetInfo

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

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

O.K., let’s build example code, then run `llc` as before. You will see error below:

```
$ ./bin/llc -march=cpu0 ../../2/debug/ch3.bc
./bin/llc: target does not support generation of this file type!
```

`llc` fails to compile LLVM IR into target machine code since we don’t implement class `Cpu0DAGToDAGISel` yet. Before moving on, we will introduce LLVM code generation sequence, DAG, and instruction selection in next three sections.

4.5 LLVM Code Generation Sequence

Following content came from “Design and Implementation of a TriCore Backend for the LLVM Compiler Framework” [6] section 4.2 Code Generation Process.

figure 4.2 gives an overview of LLVM code generation sequence.

LLVM IR is a Static Single Assignment (SSA) [4] based representation. LLVM provides infinite virtual registers which can hold value of primitive type (integral, floating point, or pointer). So, every operand can save in different virtual register in LLVM SSA representation. Comment is prefixed with `;` in LLVM representation.

Following is a LLVM IR example.

```
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
```

[4] http://en.wikipedia.org/wiki/Static_single_assignment_form

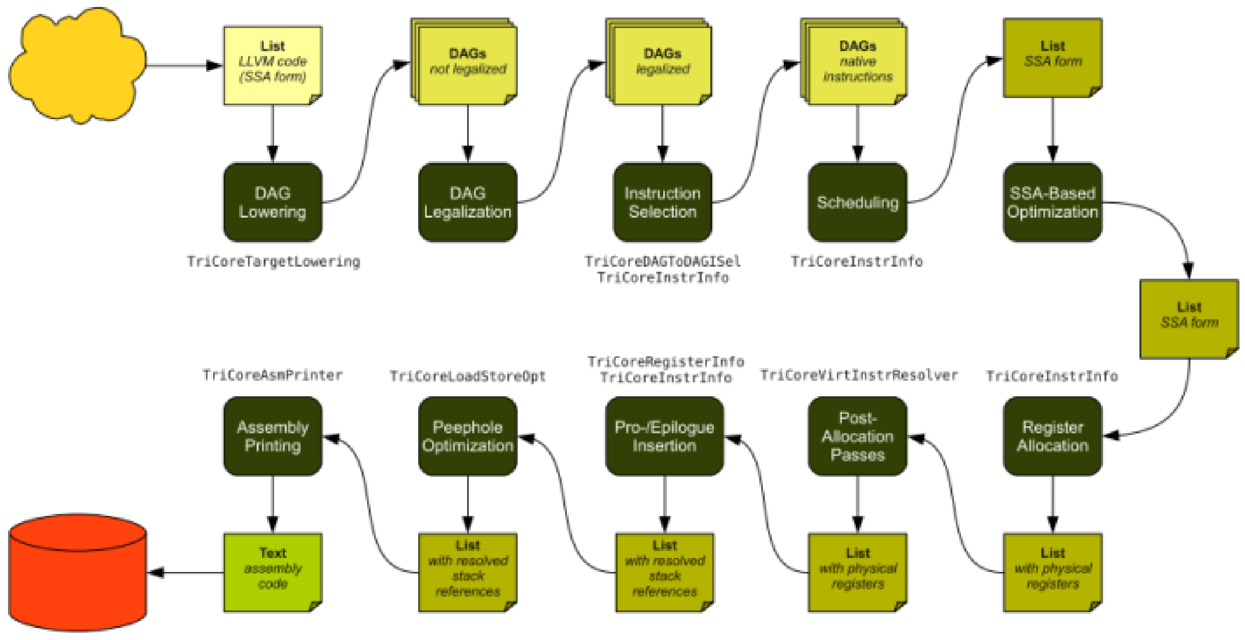


Figure 4.2: LLVM code generation sequence

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 [5] and "LLVM Language Reference Manual" from [6] 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.

"Design and Implementation of a TriCore Backend for the LLVM Compiler Framework" [6] section 4.2.

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 architecture like
    // Mips, since the "ld %c" depends on previous instruction "st %c", it must
    // wait more than 1 cycles. Therefore, 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
```

[5] http://llvm.org/docs/CodeGenerator.html
[6] http://llvm.org/docs/LangRef.html

```
        // If without reordering instructions, a instruction nop which does nothing
        // must be inserted as a bubble, 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 this basic block (meaning %c will be
        // used after this basic block), %a and %b are not dead after this
        // one. The following non-reordering 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 reordering version need 2 registers only (by allocate %a and %b in the same
        // register)
    =>  %a = add i32 1, i32 0
        st %a,  i32* %c, 1
        %b = add i32 2, i32 0
        st %b,  i32* %c, 2
```

3. SSA-based Machine Code Optimization

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

4. Register Allocation

   Allocate real register for virtual register.

5. Prologue/Epilogue Code Insertion

   Explain in section Add Prologue/Epilogue functions

6. Late Machine Code Optimizations

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

7. Code Emission

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

4.5.1 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 *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
```

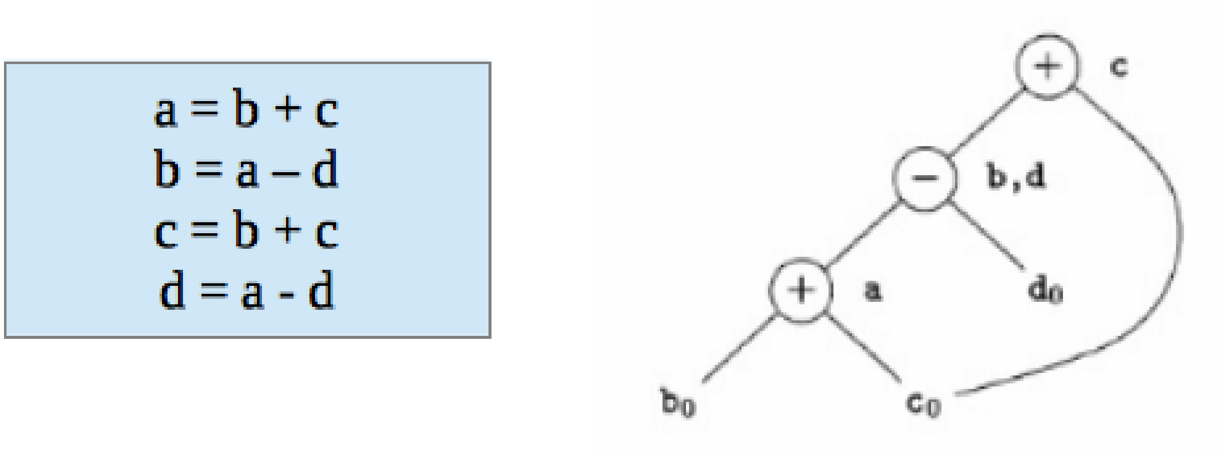

$a = b + c$
$b = a - d$
$c = b + c$
$d = a - d$

Figure 4.3: DAG example

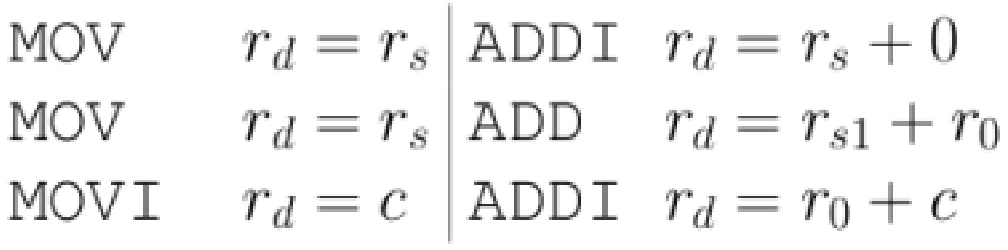
As you can imagine, the common expression remove can apply in IR or machine code.

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.

4.5.2 Instruction Selection

In the compiler backend, we need to translate compiler intermediate representation , i.e., IR code into machine code at instruction selection stage [7].

figure 4.4 shows a simple example to illustrate the before and after instruction selection.



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 4.4: IR and it's corresponding machine instruction

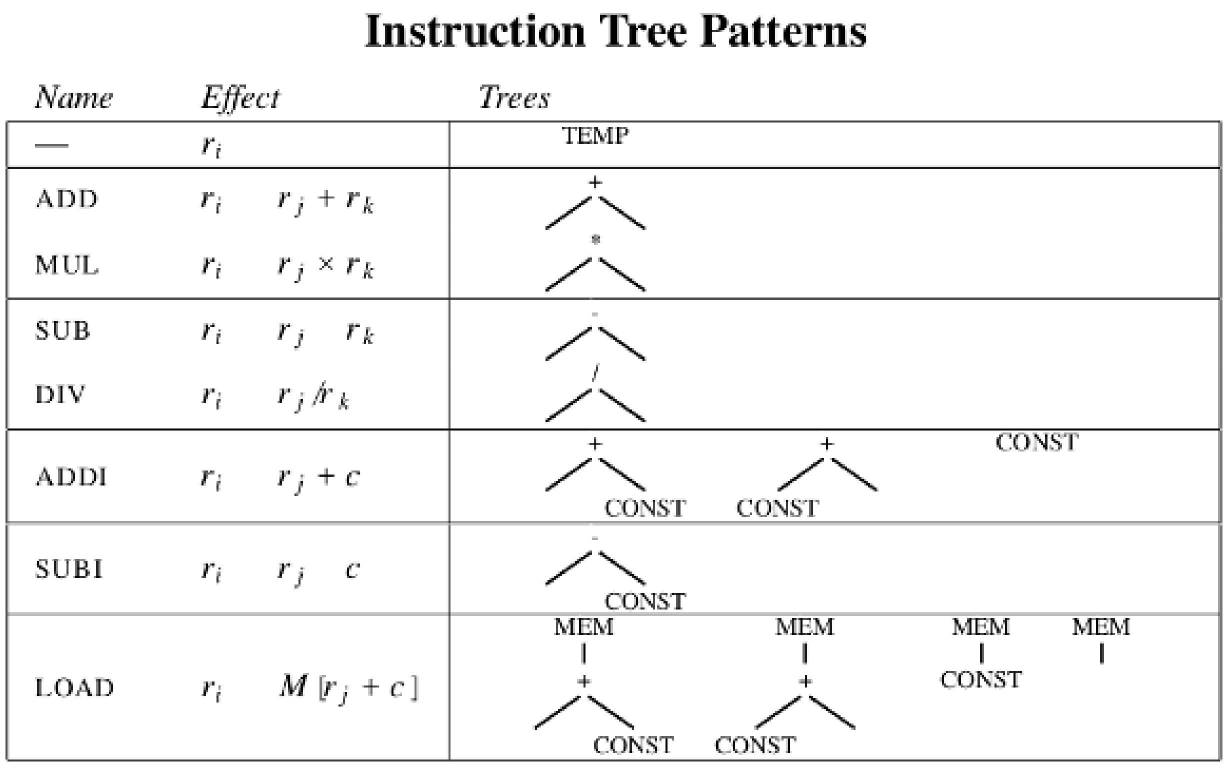
For machine instruction selection, the better solution is represent IR and machine instruction by DAG. In figure 4.5, we skip the register leaf. The rj + rk is IR DAG representation (for symbol notation, not llvm SSA form). ADD is machine instruction.

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

Now, let's recall the ADDiu instruction defined on Cpu0InstrInfo.td in the previous chapter. And It will expand to the following Pattern as mentioned in section Write td (Target Description) of the previous chapter as follows,

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

Pattern = [(set CPURegs:$ra, (add RC:$rb, immSExt16:$imm16))]
```

[7] http://en.wikipedia.org/wiki/Instruction_selection



Instruction Tree Patterns

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

Figure 4.5: Instruction DAG representation

This pattern meaning the IR DAG node **add** can translate into machine instruction DAG node ADDiu by pattern match mechanism. Similarly, the machine instruction DAG node LD and ST can be got from IR DAG node **load** and **store**.

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 *backendstructure_f7* 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;
```

4.5.3 Step 4. Add Cpu0DAGToDAGISel

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 Cpu0InstInfo.td as follows is enough. IR DAG is defined in file include/llvm/Target/TargetSelectionDAG.td.

```
/// Load and Store Instructions
///  aligned
defm LD      : LoadM32<0x00,  "ld",  load_a>;
defm ST      : StoreM32<0x01, "st",  store_a>;

/// 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>;

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

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

```
//  Cpu0TargetMachine.cpp
...
// Install an instruction selector pass using
// the ISelDag to gen Cpu0 code.
bool Cpu0PassConfig::addInstSelector() {
  PM->add(createCpu0ISelDag(getCpu0TargetMachine()));
  return false;
}

//  Cpu0ISelDAGToDAG.cpp
/// createCpu0ISelDag - This pass converts a legalized DAG into a
/// CPU0-specific DAG, ready for instruction scheduling.
FunctionPass *llvm::createCpu0ISelDag(Cpu0TargetMachine &TM) {
  return new Cpu0DAGToDAGISel(TM);
}
```

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

```
// Cpu0InstInfo.cpp
...
MachineInstr*
Cpu0InstrInfo::emitFrameIndexDebugValue(MachineFunction &MF, int FrameIx,
                                        uint64_t Offset, const MDNode *MDPtr,
                                        DebugLoc DL) const {
  MachineInstrBuilder MIB = BuildMI(MF, DL, get(Cpu0::DBG_VALUE))
    .addFrameIndex(FrameIx).addImm(0).addImm(Offset).addMetadata(MDPtr);
  return &*MIB;
}
```

Build 3/4, run it, we find the error message in 3/3 is gone. The new error message for 3/4 as follows,

```
[Gamma@localhost InputFiles]$ /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o ch3.cpu0.s
Target didn't implement TargetInstrInfo::storeRegToStackSlot!
UNREACHABLE executed at /usr/local/llvm/3.1.test/cpu0/1/src/include/llvm/Target/
TargetInstrInfo.h:390!
Stack dump:
0.        Program arguments: /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/llc
 -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o ch3.cpu0.s
1.        Running pass 'Function Pass Manager' on module 'ch3.bc'.
2.        Running pass 'Prologue/Epilogue Insertion & Frame Finalization' on function
'@main'
Aborted (core dumped)
```

4.5.4 Step 5. Add Prologue/Epilogue

Following came from "Design and Implementation of a TriCore Backend for the LLVM Compiler Framework" [5] section 4.4.2.

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 3.9 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.

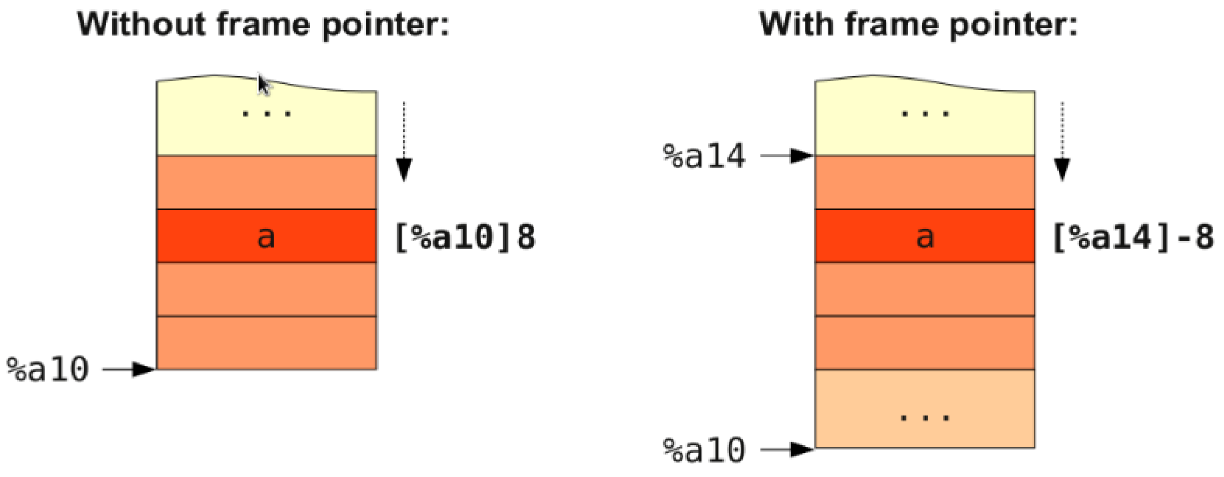


Figure 4.6: Addressing of a variable a located on the stack. If the stack frame has a variable size, slot must be addressed relative to the frame pointer

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

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
```

After add these Prologue and Epilogue functions, and build with 3/5/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,

```
[Gamma@localhost InputFiles]$ /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o ch3.cpu0.s
[Gamma@localhost InputFiles]$ cat ch3.cpu0.s
        .section .mdebug.abi32
        .previous
        .file        "ch3.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        $sp, $sp, 8
        ret        $lr
        .set        macro
        .set        reorder
        .end        main
$tmp1:
        .size        main, ($tmp1)-main
```

4.6 Summary

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

FIVE

OTHER INSTRUCTIONS

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 %.

5.1 Support arithmetic instructions

Run the 3/5/Cpu0 llc with input file ch4_1_1.bc will get the error as follows,

```
// ch4_1_1.cpp
int main()
{
        int a = 5;
        int b = 2;
        int c = 0;

        c = a + b;

        return c;
}
```

```
[Gamma@localhost 3]$ clang -c ch4_1_1.cpp -emit-llvm -o ch4_1_1.bc
[Gamma@localhost 3]$ llvm-dis ch4_1_1.bc -o ch4_1_1.ll
[Gamma@localhost 3]$ cat ch4_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-unknown-linux-gnu"

define i32 @main() nounwind uwtable {
  %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
}
[Gamma@localhost 3]$ /usr/local/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/
llc -march=cpu0 -relocation-model=pic -filetype=asm ch4_1_1.bc -o ch4_1_1.
cpu0.s
LLVM ERROR: Cannot select: 0x30da480: i32 = add 0x30da280, 0x30da380
[ORD=7] [ID=17]
  0x30da280: i32,ch = load 0x30da180, 0x30d9b80, 0x30d9880<LD4[%a]> [ORD=5]
  [ID=15]
        0x30d9b80: i32 = FrameIndex<1> [ORD=2] [ID=5]
        0x30d9880: i32 = undef [ORD=1] [ID=3]
  0x30da380: i32,ch = load 0x30da180, 0x30d9e80, 0x30d9880<LD4[%b]> [ORD=6]
  [ID=14]
        0x30d9e80: i32 = FrameIndex<2> [ORD=3] [ID=7]
        0x30d9880: i32 = undef [ORD=1] [ID=3]
```

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 4/1/Cpu0,

```
// Cpu0InstrInfo.td
/// 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     : ArithLogicR<0x1C, "rol", rotl, IIAlu, CPURegs, 1>;
def ROR     : ArithLogicR<0x1D, "ror", rotr, IIAlu, CPURegs, 1>;
def SHL     : ArithLogicR<0x1E, "shl", shl, IIAlu, CPURegs, 1>;
def SHR     : ArithLogicR<0x1F, "shr", sra, IIAlu, CPURegs, 1>;

// Cpu0Schedule.td
...
def ALU     : FuncUnit;
def IMULDIV : FuncUnit;

//===----------------------------------------------------------------------===//
// Instruction Itinerary classes used for Cpu0
//===----------------------------------------------------------------------===//
...
def IIImul              : InstrItinClass;
def IIIdiv              : InstrItinClass;

def IIPseudo            : InstrItinClass;

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

```
// Cpu0 Generic instruction itineraries.
//===----------------------------------------------------------------------===//
// 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 4/1/Cpu0 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".

```
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.

```
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

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

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 4/1 version just add 40 lines code in td files. With these 40 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.

5.2 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,

```
// ch4_2.cpp
int main()
{
  int a = 5;
  int b = 0;

  b = !a;

  return b;
}
```

```
; 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.

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

Now, let's run ch4_2.bc with 4/2/Cpu0 with llc -debug option to get result as follows,

```
118-165-16-22:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/
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:
...
    0x7fbfc282c510: <multiple use>
          0x7fbfc282c510: <multiple use>
          0x7fbfc282bc10: <multiple use>
          0x7fbfc282c610: ch = setne [ORD=5]

        0x7fbfc282c710: i1 = setcc 0x7fbfc282c510, 0x7fbfc282bc10,
        0x7fbfc282c610 [ORD=5]

        0x7fbfc282c810: i1 = Constant<-1> [ORD=6]

      0x7fbfc282c910: i1 = xor 0x7fbfc282c710, 0x7fbfc282c810 [ORD=6]

    0x7fbfc282ca10: i32 = zero_extend 0x7fbfc282c910 [ORD=7]

...


Replacing.3 0x7fbfc282c910: i1 = xor 0x7fbfc282c710, 0x7fbfc282c810 [ORD=6]

With: 0x7fbfc282ec10: i1 = setcc 0x7fbfc282c510, 0x7fbfc282bc10,
0x7fbfc282e910

Optimized lowered selection DAG: BB#0 'main:entry'
SelectionDAG has 17 nodes:
...
      0x7fbfc282c510: <multiple use>
          0x7fbfc282c510: <multiple use>
          0x7fbfc282bc10: <multiple use>
          0x7fbfc282e910: ch = seteq

        0x7fbfc282ec10: i1 = setcc 0x7fbfc282c510, 0x7fbfc282bc10,
        0x7fbfc282e910

      0x7fbfc282ca10: i32 = zero_extend 0x7fbfc282ec10 [ORD=7]
...
Type-legalized selection DAG: BB#0 'main:entry'
SelectionDAG has 18 nodes:
...
      0x7fbfc282c510: <multiple use>
          0x7fbfc282c510: <multiple use>
          0x7fbfc282bc10: <multiple use>
          0x7fbfc282e910: ch = seteq [ID=-3]

        0x7fbfc282c610: i32 = setcc 0x7fbfc282c510, 0x7fbfc282bc10,
        0x7fbfc282e910 [ID=-3]

        0x7fbfc282c710: i32 = Constant<1> [ID=-3]
      0x7fbfc282c810: i32 = and 0x7fbfc282c610, 0x7fbfc282c710 [ID=-3]

 ...
```

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. Finally, 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 follows,

```
//  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>;
```

After xor, the (and %4, 1) is translated into (and $2, (ldi $3, 1)) which is defined before already. List the asm file ch4_3.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
...
```

5.3 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/3.1.test/cpu0/1/
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 *llc option -view-dag-combine1-dags graphic view*.

From *llc option -view-dag-combine1-dags graphic view*, 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,

```
-view-dag-combine1-dags: Initial selection DAG
-view-legalize-dags: Optimized type-legalized selection DAG
-view-dag-combine2-dags: Legalized selection DAG
-view-isel-dags: Optimized legalized selection DAG
-view-sched-dags: Selected selection DAG
```

The -view-isel-dags is important and often used by an llvm backend writer because it is the DAG before instruction selection. The backend programmer need to know what is the DAG for writing the pattern match instruction in target description file .td.

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

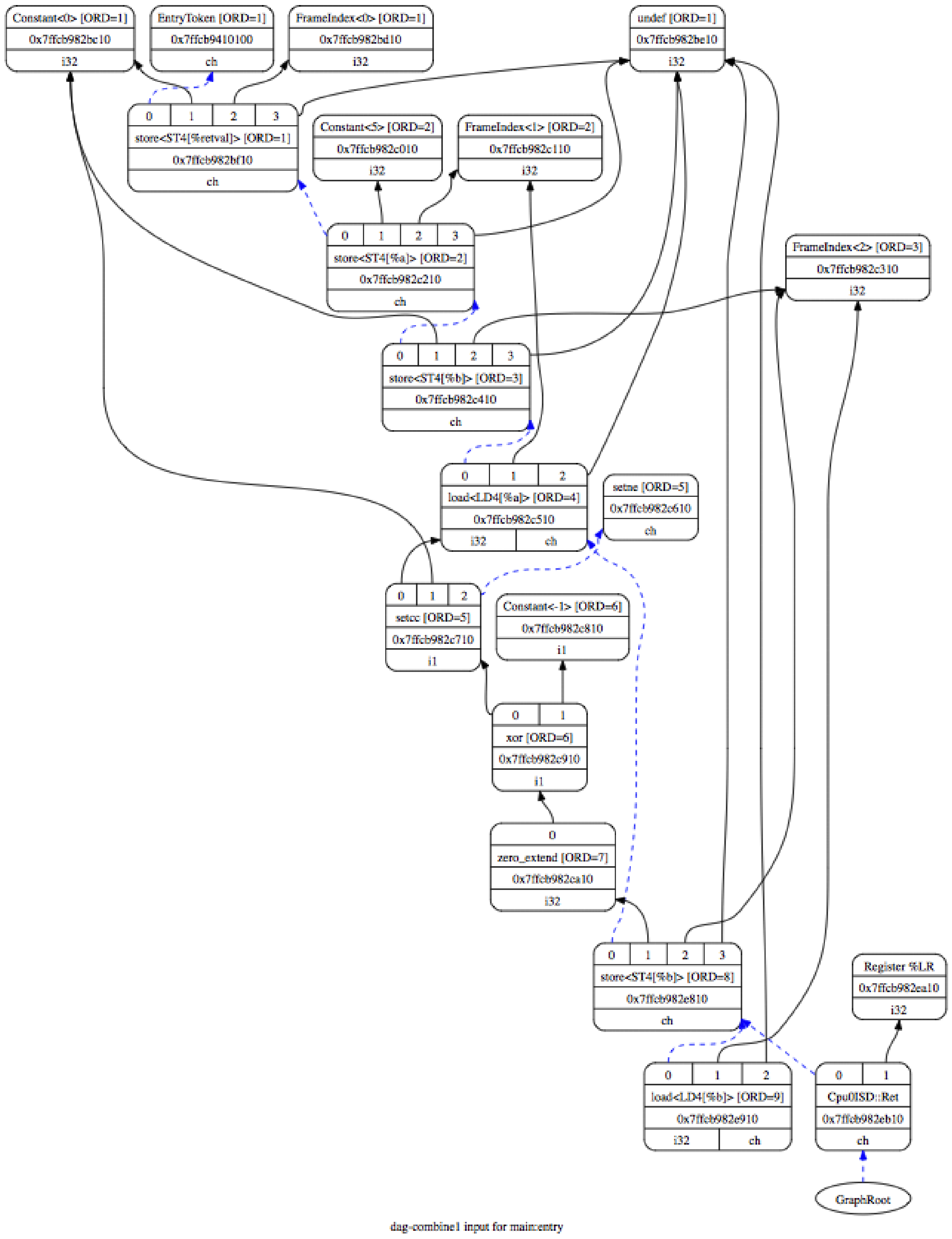


Figure 5.1: llc option -view-dag-combine1-dags graphic view

5.4 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,

```
//  Cpu0InstsInfo.td
...
def UDIV    : ArithLogicR<0x17, "udiv", udiv, IIIdiv, CPURegs, 1>;
...
/// Shift Instructions
// work, it's 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,

```
//  Cpu0InstsInfo.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 4/4/Cpu0 which support udiv, sra, and use addiu only instead of ldi, will get the result as follows,

```
// ch4_4.cpp
int main()
{
    int a = 1;
    int b = 2;
    int k = 0;
    unsigned int a1 = -5, f1 = 0;

    f1 = a1 / b;
    k = (a >> 2);

    return k;
}
```

```
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/3.1.test/cpu0/1/
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
    ...
```

5.4.1 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. 4/4_2/Cpu0 is the code changes for use **ldi** instruction. This changes replace **addiu** with **ldi** in Cpu0InstrInfo.td and modify Cpu0FrameLowering.cpp as follows,

```
// 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))>;

// 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 4/4_2/Cpu0 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/3.1.test/cpu0/1/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
```

5.5 Local variable pointer

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

```
// 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;
}

// Cpu0InstPrinter.cpp
...
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 4/5/Cpu0 which support pointer to local variable, will get result as follows,

```
// ch4_5.cpp
int main()
{
  int b = 3;

  int* p = &b;

  return *p;
}

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/3.1.test/cpu0/1/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
```

5.6 Operator mod, %

5.6.1 The DAG of %

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

```
// ch4_6.cpp
int main()
{
  int b = 11;
  //  unsigned int b = 11;

  b = (b+1)%12;

  return b;
}
```

```
; 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 4/5/Cpu0 with input file ch4_6.bc and `llc` option –view-isel-dags as follows, will get the error message as follows and the llvm DAG of *ch4_6.bc DAG*.

```
118-165-79-37:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/
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]
```

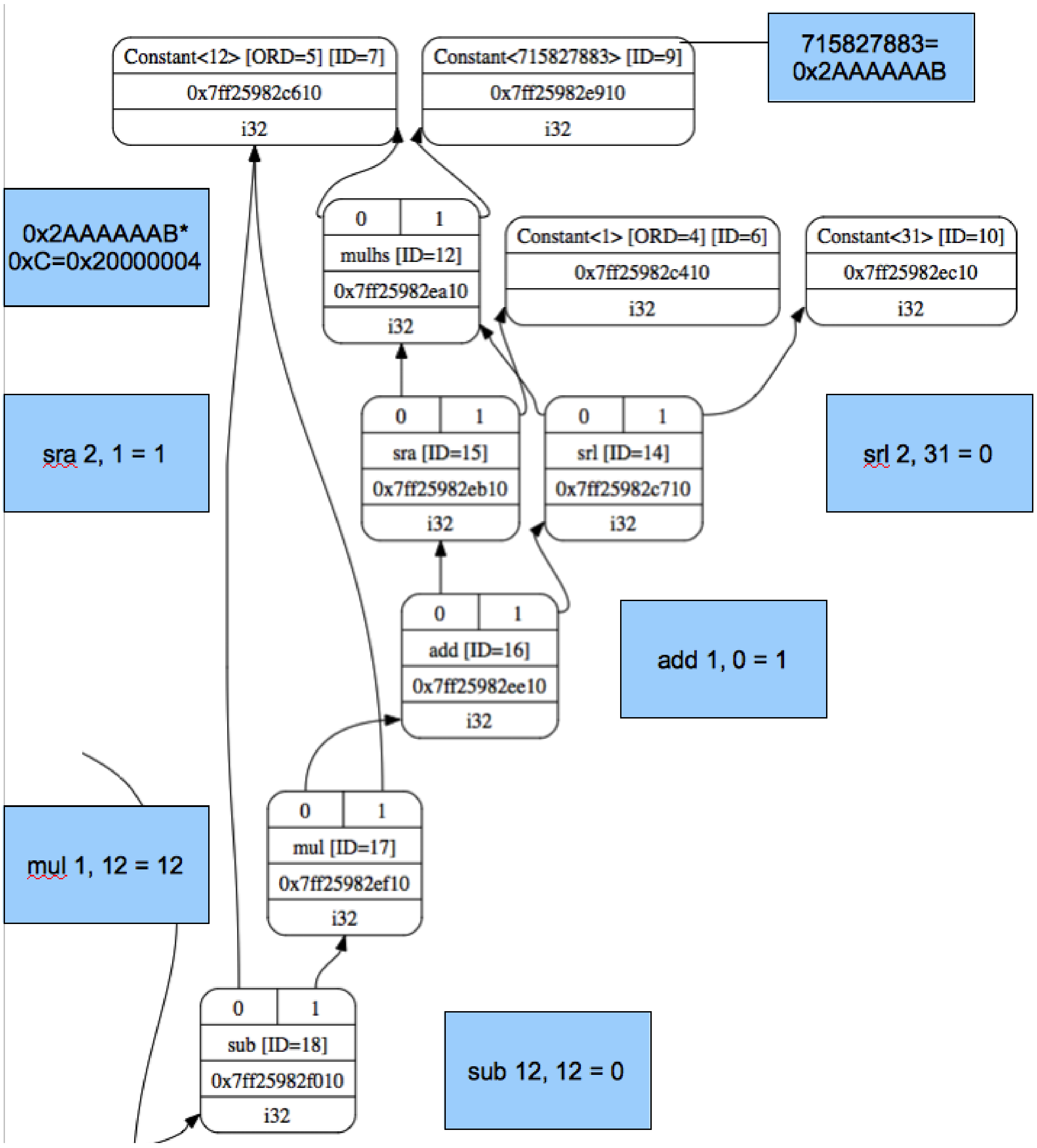

Constant<12> [ORD=5] [ID=7]
0x7ff25982c610
i32
Constant<715827883> [ID=9]
0x7ff25982e910
i32
715827883= 0x2AAAAAAB
0x2AAAAAAB* 0xC=0x20000004
0 1 mulhs [ID=12] 0x7ff25982ea10 i32
Constant<1> [ORD=4] [ID=6] 0x7ff25982c410 i32
Constant<31> [ID=10] 0x7ff25982ec10 i32
sra 2, 1 = 1
0 1 sra [ID=15] 0x7ff25982eb10 i32
0 1 srl [ID=14] 0x7ff25982c710 i32
srl 2, 31 = 0
0 1 add [ID=16] 0x7ff25982ee10 i32
add 1, 0 = 1
mul 1, 12 = 12
0 1 mul [ID=17] 0x7ff25982ef10 i32
0 1 sub [ID=18] 0x7ff25982f010 i32
sub 12, 12 = 0

Figure 5.2: ch4_6.bc DAG

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 *ch4_6.bc DAG*. We verify the result and explain by calculate the value in each node. The 0xC*0x2AAAAAAB=0x200000004, (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.

5.6.2 Arm solution

Let's run 4/6_1/Cpu0 with ch4_6.cpp as well as llc option -view-sched-dags to get *Translate ch4_6.bc into cpu0 backend DAG*. Similarly, SMMUL get the high word of multiply result.

Follows is the result of run 4/6_1/Cpu0 with ch4_6.bc.

```
118-165-66-82:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/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 4/6_1/Cpu0 use the ARM solution. With this solution, the following code is needed.

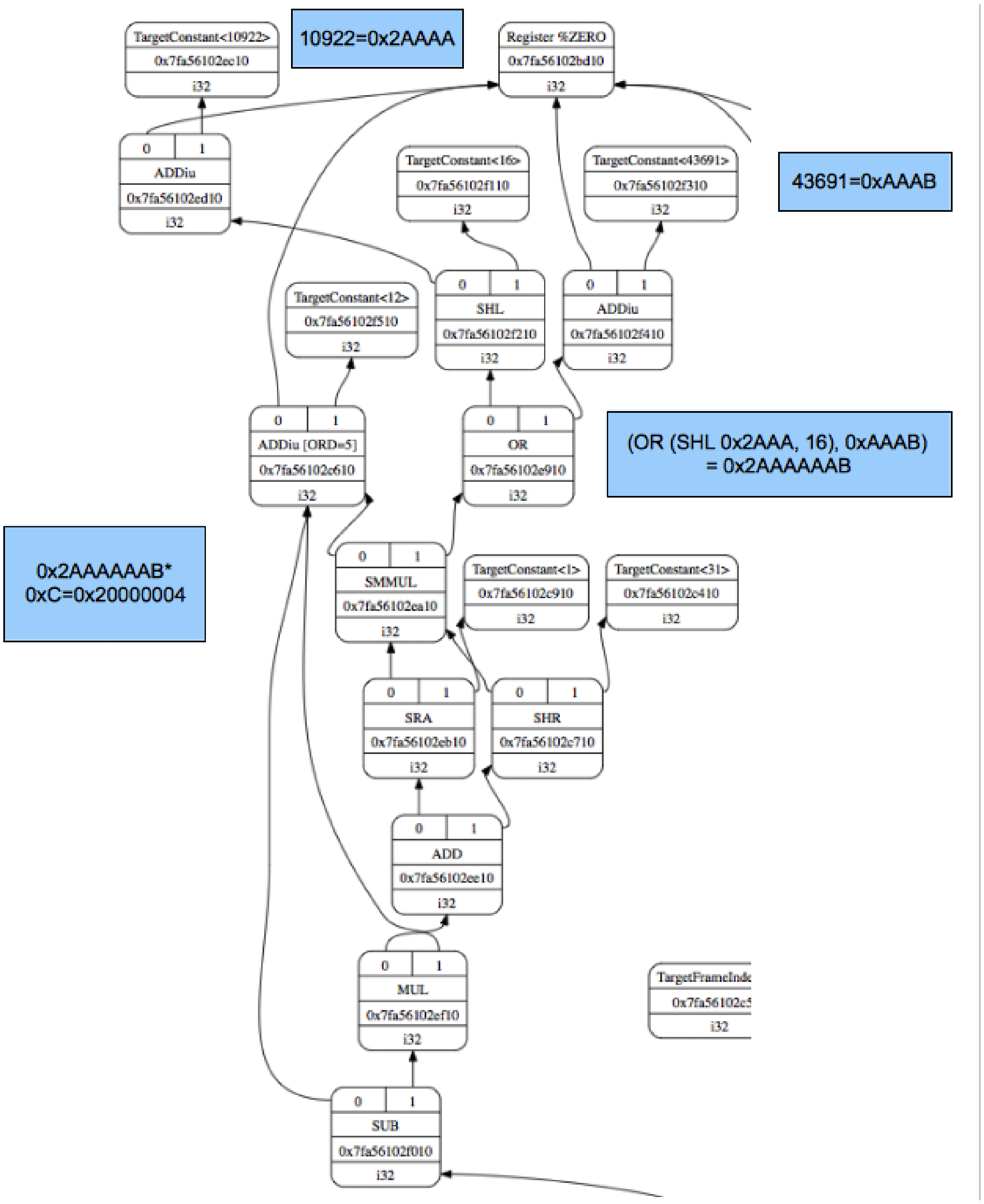


Figure 5.3: Translate ch4_6.bc into cpu0 backend DAG

```
// 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)))>;
```

5.6.3 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. 4/6_2/Cpu0 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 4/6_2/Cpu0 from TargetSelectionDAG.td.

```
// Cpu0RegisterInfo.td
  ...
  // Hi/Lo registers
  def HI  : Register<"hi">, DwarfRegNum<[18]>;
    def LO  : Register<"lo">, DwarfRegNum<[19]>;

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

// 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]>;

// 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();
  ...
  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;
  }
  ...
}

// 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. *DAG for ch4_6.bc with Mips style MULT* 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
  .set  reorder
  .end  main
$tmp2:
  .size main, ($tmp2)-main
  .cfi_endproc
```

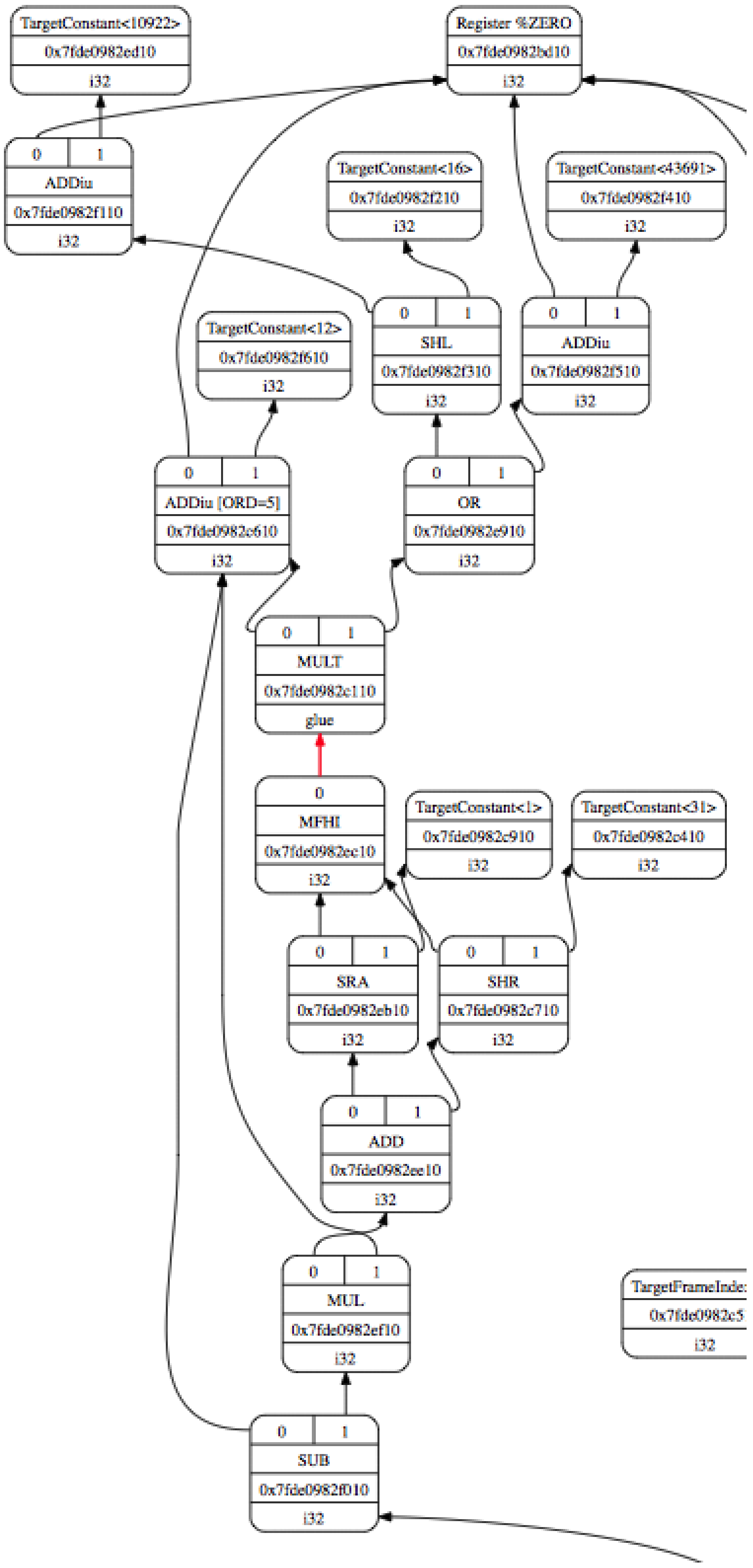


Figure 5.4: DAG for ch4_6.bc with Mips style MULT

5.7 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"**.

4/6_4/Cpu0 support operator **"%"** and **"/"**. The code added in 4/6_4/Cpu0 as follows,

```
// Cpu0InstrInfo.cpp
...
void Cpu0InstrInfo::
copyPhysReg(MachineBasicBlock &MBB,
      MachineBasicBlock::iterator I, DebugLoc DL,
      unsigned DestReg, unsigned SrcReg,
      bool KillSrc) const {
  unsigned Opc = 0, ZeroReg = 0;

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

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

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

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

  if (ZeroReg)
  MIB.addReg(ZeroReg);

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

// Cpu0InstrInfo.h
...
  virtual void copyPhysReg(MachineBasicBlock &MBB,
               MachineBasicBlock::iterator MI, DebugLoc DL,
               unsigned DestReg, unsigned SrcReg,
               bool KillSrc) const;
```

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

// 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();
}

// 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
  };
  }
...

// Cpu0RegisterInfo.td
...
// Hi/Lo Registers
def HILO : RegisterClass<"Cpu0", [i32], 32, (add HI, LO)>;
```

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.

```
// 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/3.1.test/cpu0/1/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
  ...

// ch4_6_1.cpp
int main()
{
  int b = 11;
  int a = 12;

  b = (b+1)%a;

  return b;
}

// ch4_6_2.cpp
#include <stdlib.h>

int main()
{
  int b = 11;
//  unsigned int b = 11;
  int c = rand();

  b = (b+1)%c;
  return b;
}
```

5.8 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

SIX

GENERATE OBJ FILE

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.

6.1 Translate into obj file

Currently, we only support translate llvm IR code into assembly code. If you try to run 4/6_2/Cpu0 to translate obj code will get the error message as follows,

```
[Gamma@localhost 3]$ /usr/local/llvm/3.1.test/cpu0/1/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/3.1.test/cpu0/1/cmake_debug_build/bin/llc: target does not
support generation of this file type!
```

The 5/Cpu0 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/3.1.test/cpu0/1/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/3.1.test/cpu0/1/
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.

6.2 Backend Target Registration Structure

Now, let's examine Cpu0MCTargetDesc.cpp.

```
// Cpu0MCTargetDesc.cpp
...
extern "C" void LLVMInitializeCpu0TargetMC() {
  // Register the MC asm info.
  RegisterMCAsmInfoFn X(TheCpu0Target, createCpu0MCAsmInfo);
  RegisterMCAsmInfoFn Y(TheCpu0elTarget, createCpu0MCAsmInfo);

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

  // Register the MC register info.
  TargetRegistry::RegisterMCRegInfo(TheCpu0Target, createCpu0MCRegisterInfo);
  TargetRegistry::RegisterMCRegInfo(TheCpu0elTarget, createCpu0MCRegisterInfo);
  // Register the MC Code Emitter
  TargetRegistry::RegisterMCCodeEmitter(TheCpu0Target,
                                        createCpu0MCCodeEmitterEB);
  TargetRegistry::RegisterMCCodeEmitter(TheCpu0elTarget,
                                        createCpu0MCCodeEmitterEL);

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

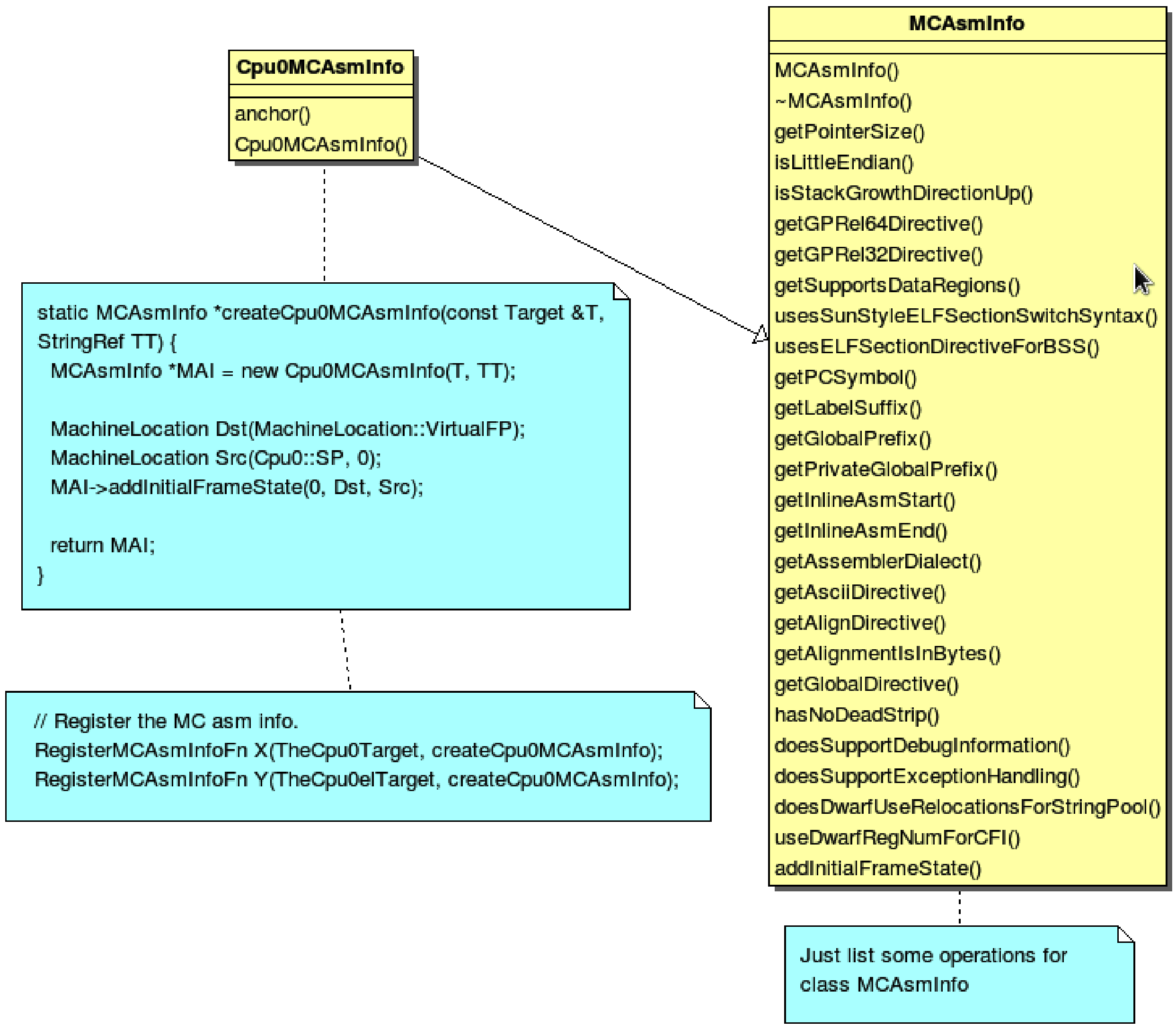
Cpu0MCTargetDesc.cpp do the target registration as mentioned in “section Target Registration” [1] of the last chapter. Drawing the register function and those class it registered in *Register Cpu0MCAsmInfo* to *Register Cpu0InstPrinter* for explanation.

In *Register Cpu0MCAsmInfo*, 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 *Register MCCodeGenInfo*, 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 *Register MCInstrInfo*, instancing MCInstrInfo object X, and initialize it by InitCpu0MCInstrInfo(X). Since InitCpu0MCInstrInfo(X) is defined in Cpu0GenInstrInfo.inc, it will add the information fromCpu0InstrInfo.td we

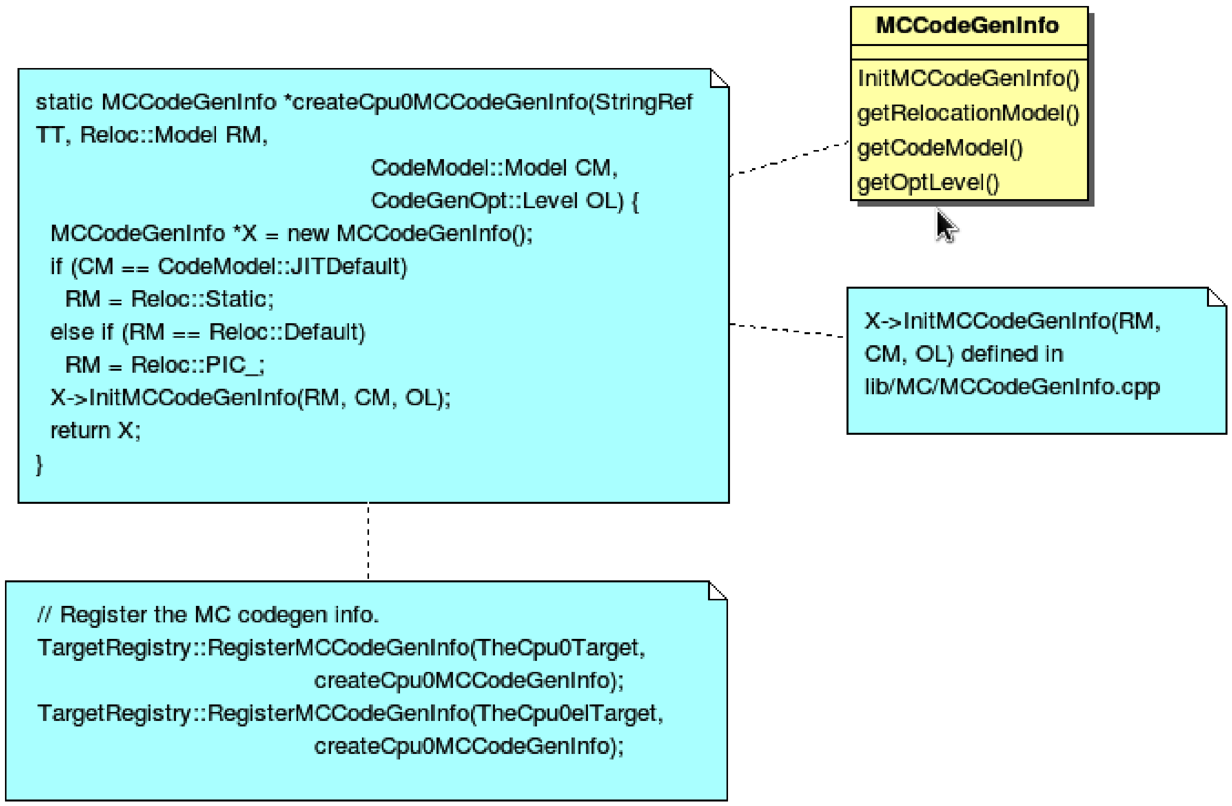
[1] http://jonathan2251.github.com/lbd/llvmstructure.html#target-registration



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

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

  return MAI;
}
```

```
// Register the MC asm info.
RegisterMCAsmInfoFn X(TheCpu0Target, createCpu0MCAsmInfo);
RegisterMCAsmInfoFn Y(TheCpu0elTarget, createCpu0MCAsmInfo);
```

Figure 6.1: Register Cpu0MCAsmInfo

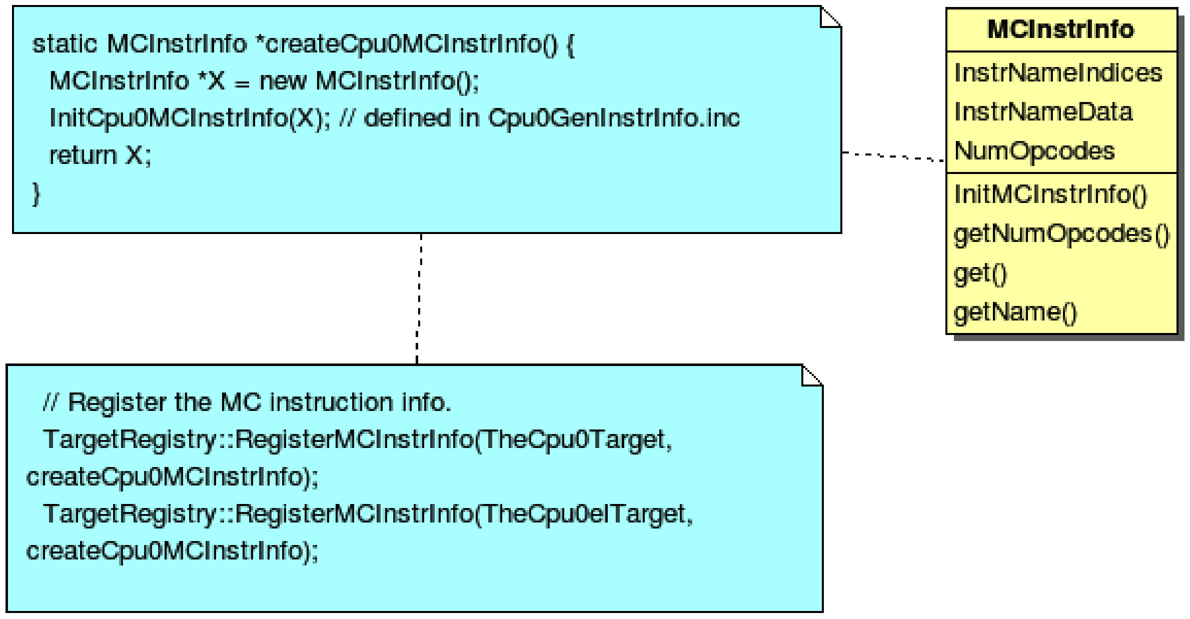


```
static MCCodeGenInfo *createCpu0MCCodeGenInfo(StringRef
TT, Reloc::Model RM,
                                              CodeModel::Model CM,
                                              CodeGenOpt::Level OL) {
  MCCodeGenInfo *X = new MCCodeGenInfo();
  if (CM == CodeModel::JITDefault)
    RM = Reloc::Static;
  else if (RM == Reloc::Default)
    RM = Reloc::PIC_;
  X->InitMCCodeGenInfo(RM, CM, OL);
  return X;
}
```

X->InitMCCodeGenInfo(RM, CM, OL) defined in lib/MC/MCCodeGenInfo.cpp

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

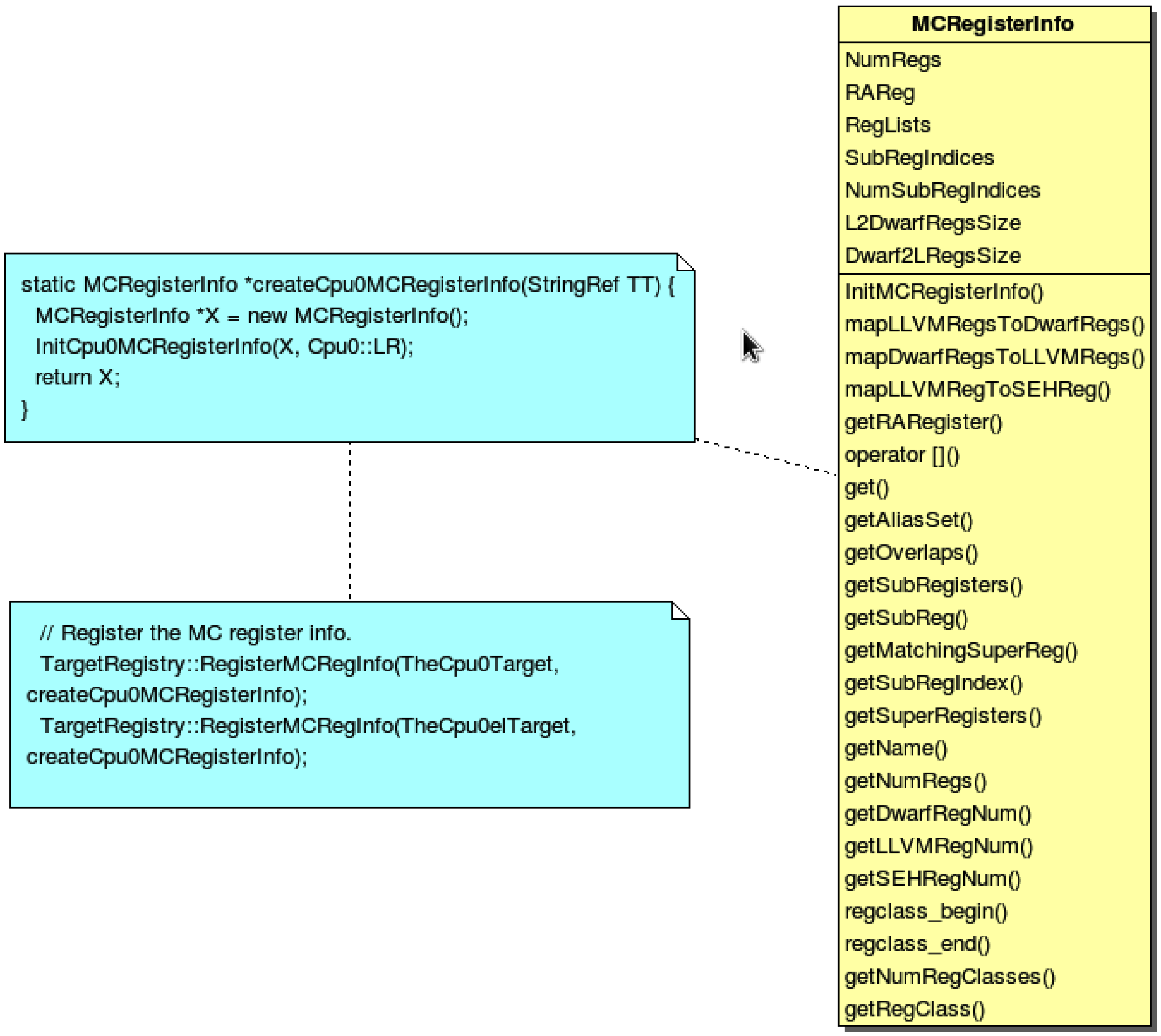
Figure 6.2: Register MCCodeGenInfo



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

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

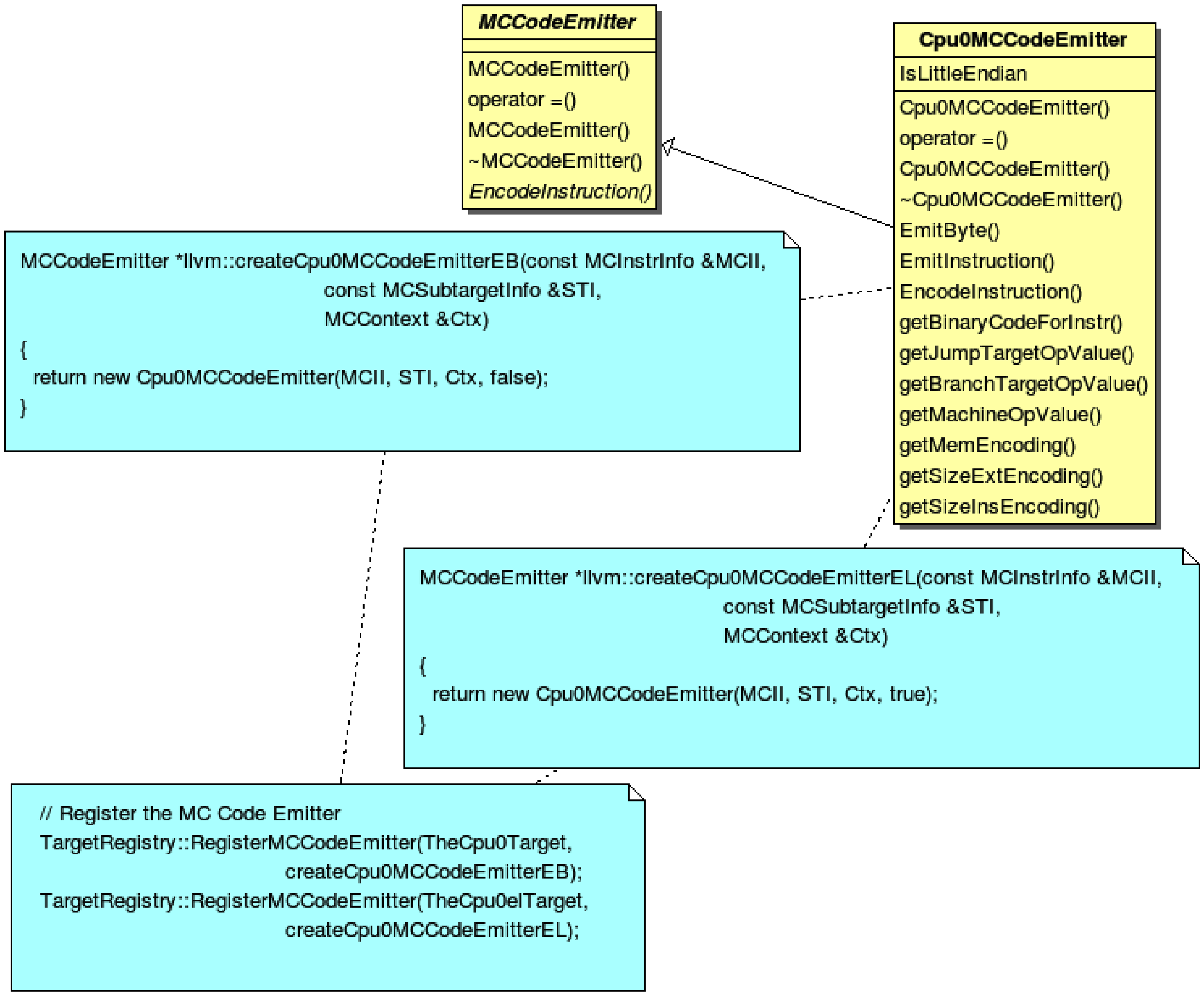
Figure 6.3: Register MCInstrInfo



```
static MCRegisterInfo *createCpu0MCRegisterInfo(StringRef TT) {
  MCRegisterInfo *X = new MCRegisterInfo();
  InitCpu0MCRegisterInfo(X, Cpu0::LR);
  return X;
}
```

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

Figure 6.4: Register MCRegisterInfo

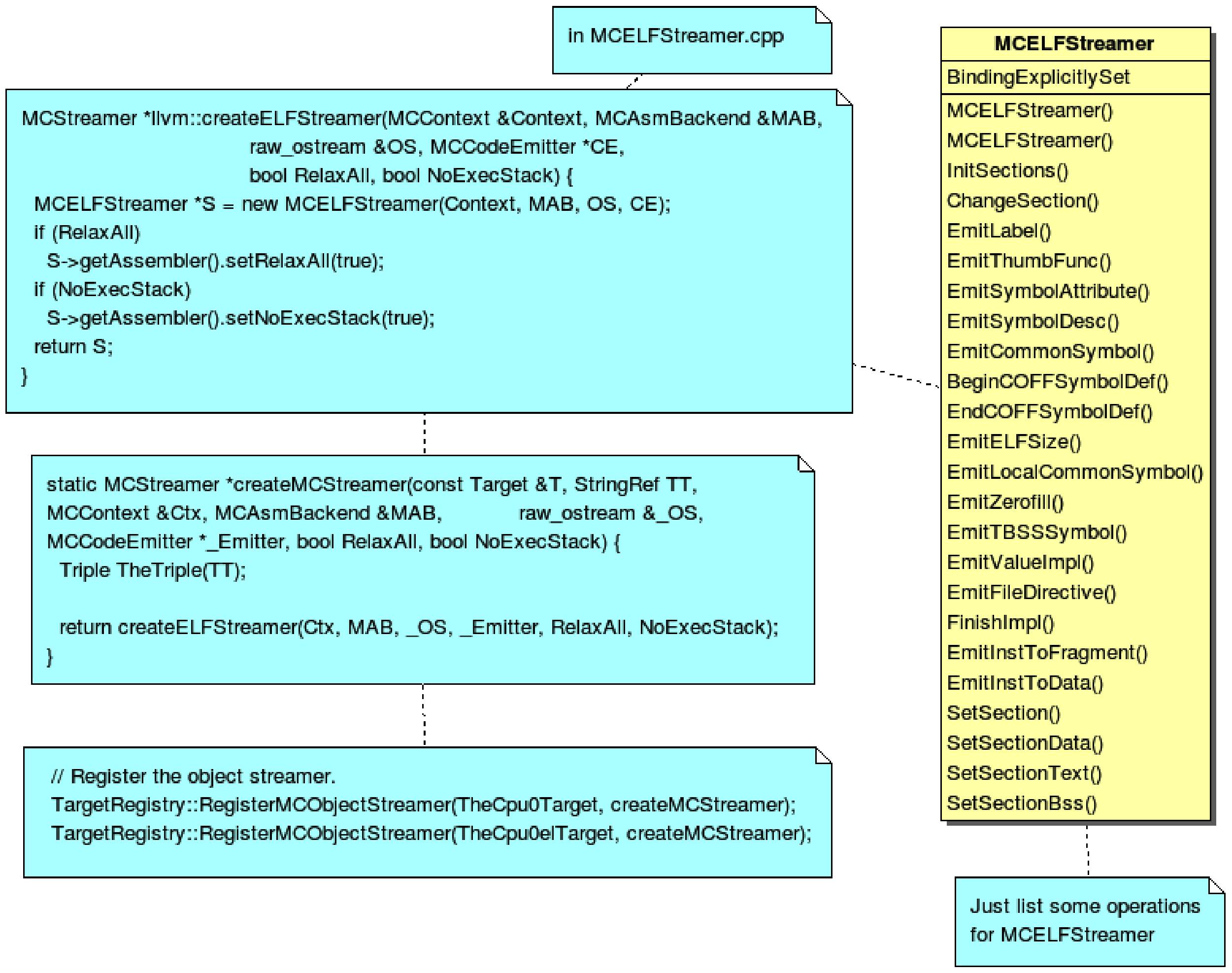


```
MCCodeEmitter *llvm::createCpu0MCCodeEmitterEB(const MCInstrInfo &MCII,
                                               const MCSubtargetInfo &STI,
                                               MCContext &Ctx)
{
  return new Cpu0MCCodeEmitter(MCII, STI, Ctx, false);
}
```

```
MCCodeEmitter *llvm::createCpu0MCCodeEmitterEL(const MCInstrInfo &MCII,
                                               const MCSubtargetInfo &STI,
                                               MCContext &Ctx)
{
  return new Cpu0MCCodeEmitter(MCII, STI, Ctx, true);
}
```

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

Figure 6.5: Register Cpu0MCCodeEmitter

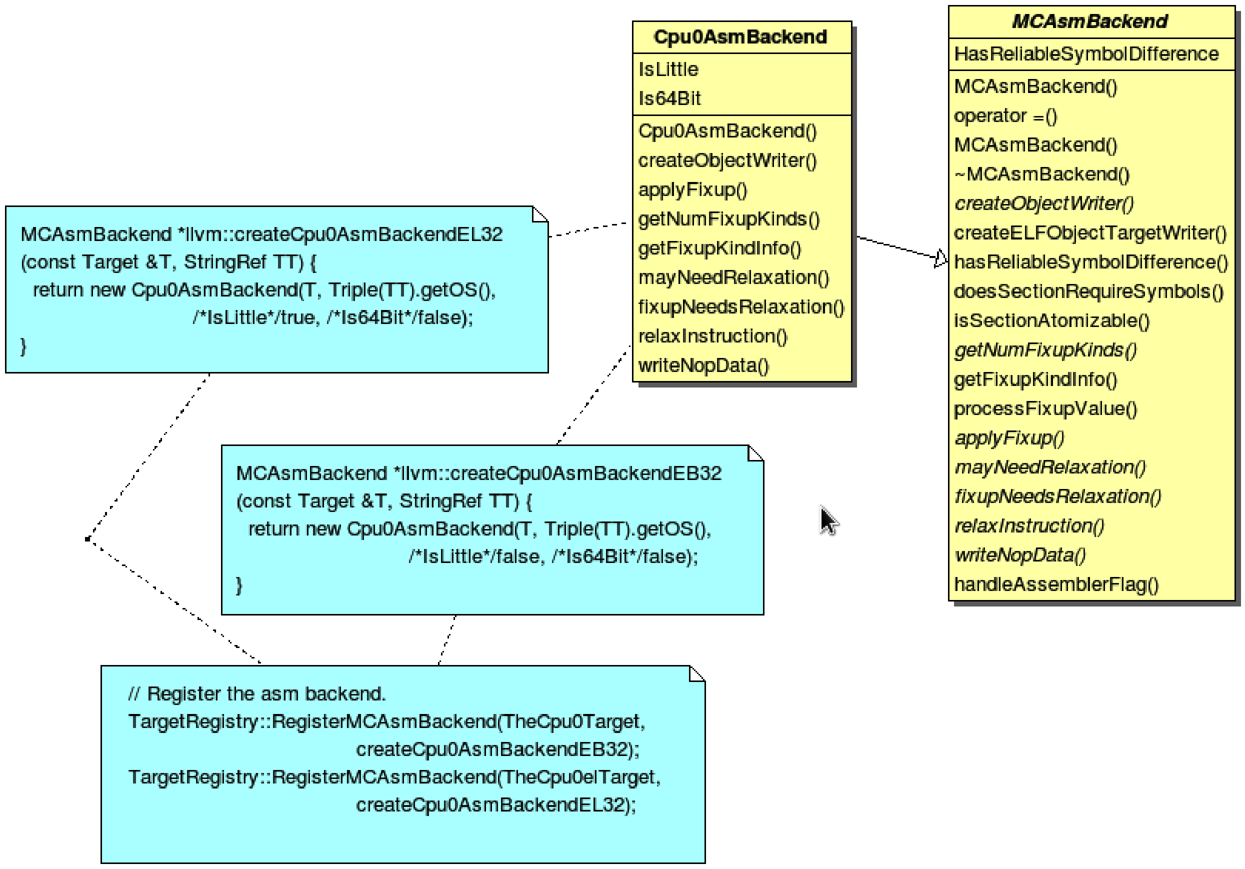


```
MCStreamer *llvm::createELFStreamer(MCContext &Context, MCAsmBackend &MAB,
                                    raw_ostream &OS, MCCodeEmitter *CE,
                                    bool RelaxAll, bool NoExecStack) {
  MCELFStreamer *S = new MCELFStreamer(Context, MAB, OS, CE);
  if (RelaxAll)
    S->getAssembler().setRelaxAll(true);
  if (NoExecStack)
    S->getAssembler().setNoExecStack(true);
  return S;
}
```

```
static MCStreamer *createMCStreamer(const Target &T, StringRef TT,
MCContext &Ctx, MCAsmBackend &MAB,            raw_ostream &_OS,
MCCodeEmitter *_Emitter, bool RelaxAll, bool NoExecStack) {
  Triple TheTriple(TT);

  return createELFStreamer(Ctx, MAB, _OS, _Emitter, RelaxAll, NoExecStack);
}
```

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

Figure 6.6: Register MCELFStreamer

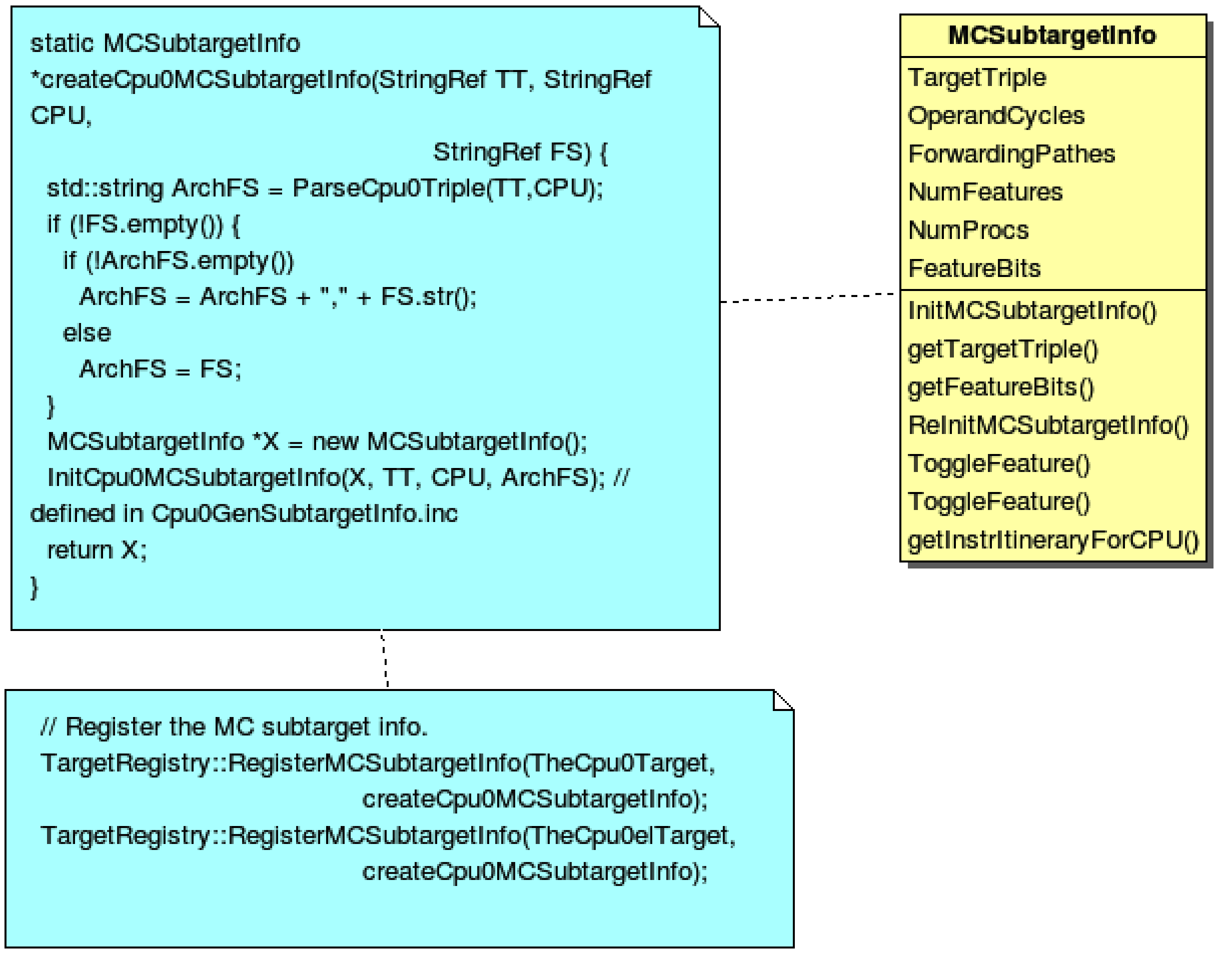


```
MCAsmBackend *llvm::createCpu0AsmBackendEL32
(const Target &T, StringRef TT) {
  return new Cpu0AsmBackend(T, Triple(TT).getOS(),
                     /*IsLittle*/true, /*Is64Bit*/false);
}
```

```
MCAsmBackend *llvm::createCpu0AsmBackendEB32
(const Target &T, StringRef TT) {
  return new Cpu0AsmBackend(T, Triple(TT).getOS(),
                     /*IsLittle*/false, /*Is64Bit*/false);
}
```

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

Figure 6.7: Register Cpu0AsmBackend

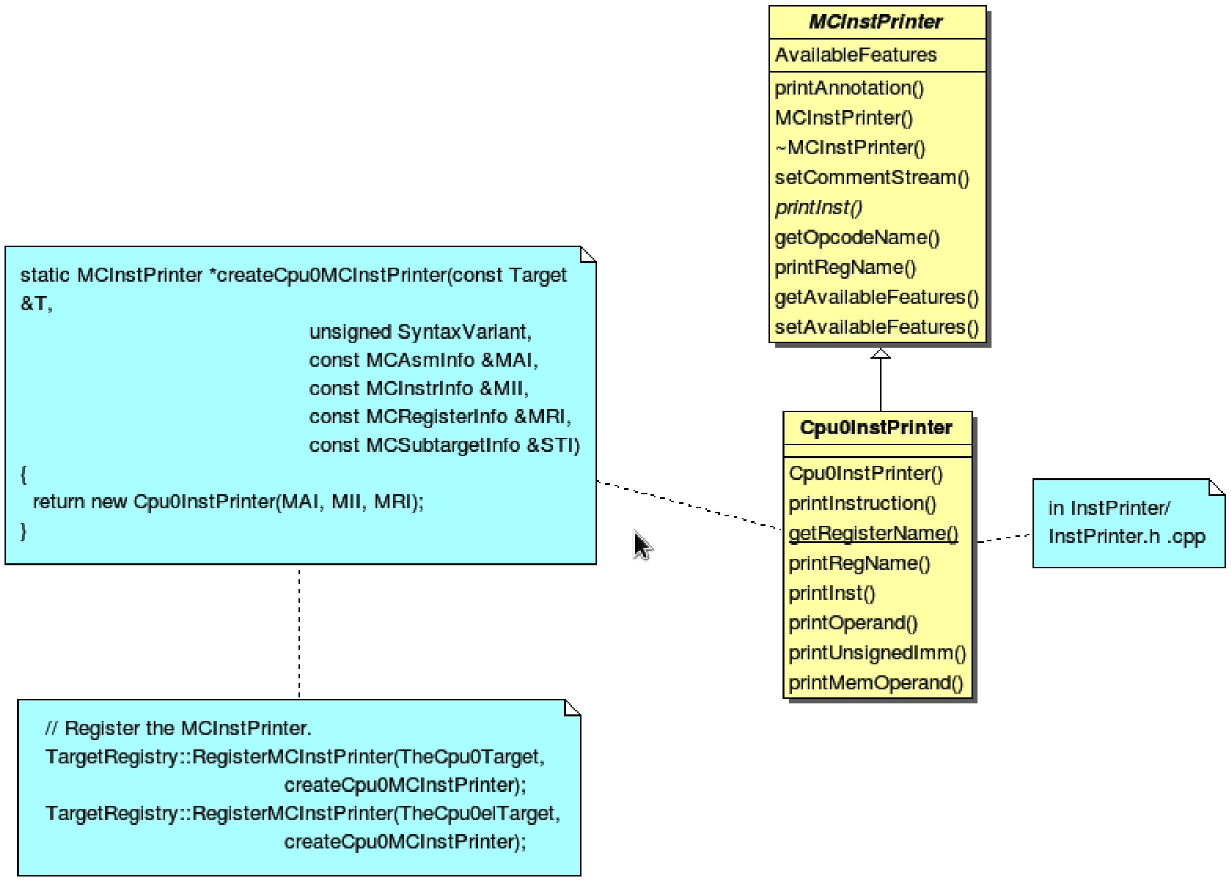


```
static MCSubtargetInfo
*createCpu0MCSubtargetInfo(StringRef TT, StringRef
CPU,
                                                  StringRef FS) {
  std::string ArchFS = ParseCpu0Triple(TT,CPU);
  if (!FS.empty()) {
    if (!ArchFS.empty())
      ArchFS = ArchFS + "," + FS.str();
    else
      ArchFS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

```
MCSubtargetInfo
  TargetTriple
  OperandCycles
  ForwardingPathes
  NumFeatures
  NumProcs
  FeatureBits
  InitMCSubtargetInfo()
  getTargetTriple()
  getFeatureBits()
  ReInitMCSubtargetInfo()
  ToggleFeature()
  ToggleFeature()
  getInstrItineraryForCPU()
```

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

Figure 6.8: Register Cpu0MCSubtargetInfo



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

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

Figure 6.9: Register Cpu0InstPrinter

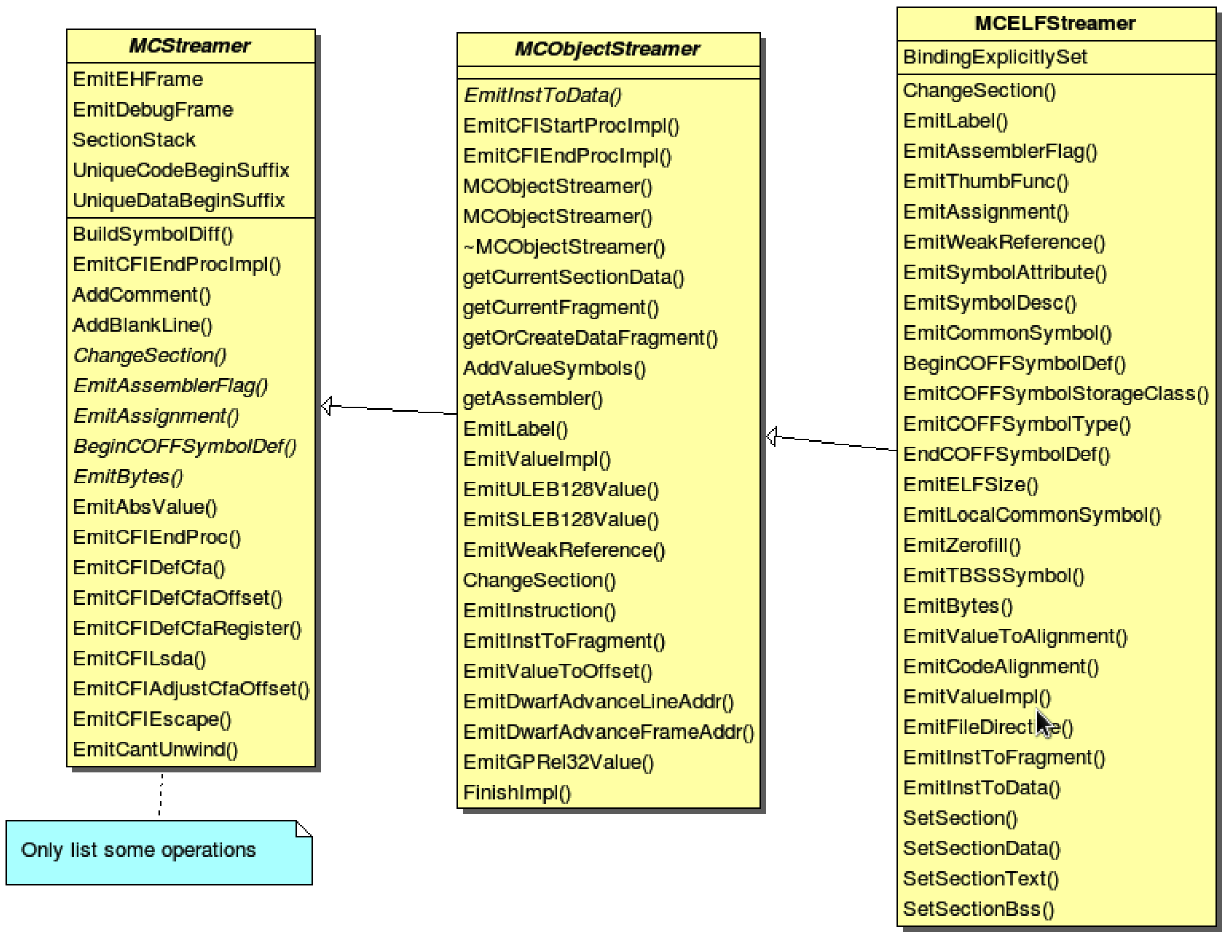


Figure 6.10: MCELFStreamer inherit tree

specified. *Register MCRegisterInfo* is similar to *Register MCInstrInfo*, but it initialize the register information specified in Cpu0RegisterInfo.td. They share a lot of code with instruction/register td description.

Register Cpu0MCCodeEmitter, 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 4/6_2/Cpu0 which support assembly code only.

Register MCELFStreamer, MCELFStreamer take care the obj format also. *Register Cpu0MCCodeEmitter* Cpu0MCCodeEmitter take care code emitter while MCELFStreamer take care the obj output streamer. *MCELFStreamer inherit tree* 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().

Register Cpu0AsmBackend, 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.

Register Cpu0MCSubtargetInfo, instancing MCSubtargetInfo object and initialize with Cpu0.td information. *Register Cpu0InstPrinter*, instancing Cpu0InstPrinter to take care printing function for instructions. Like *Register Cpu0MCAsmInfo* to *Register MCRegisterInfo*, it has been defined in 4/6_2/Cpu0 code for assembly file generated support.

CHAPTER

SEVEN

GLOBAL VARIABLE, STRUCT AND ARRAY

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.

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.

7.1 Global variable

6/1/Cpu0 support the global variable, let's compile ch6_1.cpp with this version first, and explain the code changes after that.

```
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/3.1.test/cpu0/1/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.

7.1.1 Static mode

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

```
118-165-66-82:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/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 load 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 load the low part of gI absolute address into register 3. Finally, add register $2 and $3 into $2, and load 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 following command,

```
118-165-66-82:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/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 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 in compile/link time, change only in 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 $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 default to true if without specify it. About the cl command, you can refer to http://llvm.org/docs/CommandLine.html further.

```
//  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.
//  Cpu0ISelLowering.cpp
Cpu0TargetLowering::
Cpu0TargetLowering(Cpu0TargetMachine &TM)
  : TargetLowering(TM, new Cpu0TargetObjectFile()),
    Subtarget(&TM.getSubtarget<Cpu0Subtarget>()) {
   ...
  // Cpu0 Custom Operations
  setOperationAction(ISD::GlobalAddress,       MVT::i32,   Custom);
   ...
}
SDValue Cpu0TargetLowering::
LowerOperation(SDValue Op, SelectionDAG &DAG) const
{
  switch (Op.getOpcode())
  {
    case ISD::GlobalAddress:      return LowerGlobalAddress(Op, DAG);
  }
  return SDValue();
}

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

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

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

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

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

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

  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) tell 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.

```
//  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,

```
  //  Cpu0InstrInfo.td
  // Hi and Lo nodes are used to handle global addresses. Used on
  // Cpu0ISelLowering to lower stuff like GlobalAddress, ExternalSymbol
  // static model. (nothing to do with Cpu0 Registers Hi and Lo)
  def Cpu0Hi    : SDNode<"Cpu0ISD::Hi", SDTIntUnaryOp>;
  def Cpu0Lo    : SDNode<"Cpu0ISD::Lo", SDTIntUnaryOp>;
  ...
// hi/lo relocs
def : Pat<(Cpu0Hi tglobaladdr:$in), (SHL (ADDiu ZERO, tglobaladdr:$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).

```
// %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,

```
//  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)
```

7.1.2 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,

```
  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 $2 which decide by Register Allocator, and (Wrapper GetGlobalReg(), GA) translate into Base=$gp and 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,

```
// 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().

7.1.3 Global variable print support

Above code is for global address DAG translation. Next, add the following code to Cpu0MCInstLower.cpp and Cpu0InstPrinter.cpp for global variable printing operand function.

```
// 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);
  ...
 }

// 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;
  }
  ...
}
```

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

7.1.4 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 http://llvm.org/docs/WritingAnLLVMBackend.html is the references.

7.2 Array and struct support

Shifting our work to iMac at this point. The Linux platform is fine. The reason we do the shift is for new platform using experience.

LLVM use getelementptr to represent the array and struct type in C. Please reference section getelementptr of [1]. For ch6_2.cpp, the llvm IR as follows,

```
// ch6_2.cpp
struct Date
{
    int year;
    int month;
    int day;
};

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

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

    return 0;
}
```

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

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

%struct.Date = type { i32, i32, i32 }

@date = global %struct.Date { i32 2012, i32 10, i32 12 }, align 4
@a = global [3 x i32] [i32 2012, i32 10, i32 12], align 4

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

Run 6/1/Cpu0 with ch6_2.bc on static mode will get the incorrect asm file as follows,

```
118-165-66-82:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/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/3.1.test/cpu0/1/
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.

```
// 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;
}

// Cpu0TargetLowering.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,

```
//  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.

```
// 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.

6/2/Cpu0 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)
```

CHAPTER

EIGHT

CONTROL FLOW STATEMENT

This chapter illustrates the corresponding IR for control flow statements, like **"if else"**, **"while"** and **"for"** loop statements in C, and how to translate these control flow statements of llvm IR into cpu0 instructions.

8.1 Control flow statement

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

```
// ch7_1_1.cpp
int main()
{
    unsigned int a = 0;
    int b = 1;
    int c = 2;
    int d = 3;
    int e = 4;
    int f = 5;
    int g = 6;
    int h = 7;
    int i = 8;

    if (a == 0) {
        a++;
    }
    if (b != 0) {
        b++;
    }
    if (c > 0) {
        c++;
    }
    if (d >= 0) {
        d++;
    }
    if (e < 0) {
        e++;
    }
    if (f <= 0) {
        f++;
    }
    if (g <= 1) {
        g++;
    }
    if (h >= 1) {
        h++;
    }
    if (i < h) {
        i++;
    }
    if (a != b) {
        a++;
    }

    return a;
}
```

```
; 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"** stand for Set Less Than, **"sle"** stand for Set Less Equal. Run version 6/2/Cpu0 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,

```
// 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,

```
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,

```
// 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. Since SW is a reserved register, it will be 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,

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

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

  // If GP is dedicated as a global base register, reserve it.
  if (MF.getInfo<Cpu0FunctionInfo>()->globalBaseRegFixed()) {
    Reserved.set(Cpu0::GP);
  }

  return Reserved;
}
```

Although the following definition in Cpu0RegisterInfo.td has no real effect in Reserved Registers, you should comment the Reserved Registers in it for readability.

```
// 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, SW, SP, LR, PC)>;
```

By 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/3.1.test/cpu0/1/
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,

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

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

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

// 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:
  ...
  }
  ...
}

// 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::Other, Expand);
  ...
}

// Cpu0InstrFormats.td
```

```
class BranchBase<bits<8> op, dag outs, dag ins, string asmstr,
                  list<dag> pattern, InstrItinClass itin>:
  Cpu0Inst<outs, ins, asmstr, pattern, itin, FrmL>
{
  bits<24> imm24;

  let Opcode = op;

  let Inst{23-0}  = imm24;
}

// 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>:
  BranchBase<op, (outs), (ins RC:$cond, brtarget:$imm24),
             !strconcat(instr_asm, "\t$imm24"),
             [], IIBranch> {
  let isBranch = 1;
  let isTerminator = 1;
  let hasDelaySlot = 0;
}

// Unconditional branch
class UncondBranch<bits<8> op, string instr_asm>:
  BranchBase<op, (outs), (ins brtarget:$imm24),
             !strconcat(instr_asm, "\t$imm24"), [(br bb:$imm24)], IIBranch> {
  let isBranch = 1;
  let isTerminator = 1;
  let isBarrier = 1;
  let hasDelaySlot = 0;
}
...
/// Jump and Branch Instructions
def JEQ     : CBranch<0x20, "jeq", CPURegs>;
def JNE     : CBranch<0x21, "jne", CPURegs>;
def JLT     : CBranch<0x22, "jlt", CPURegs>;
def JGT     : CBranch<0x23, "jgt", CPURegs>;
def JLE     : CBranch<0x24, "jle", CPURegs>;
def JGE     : CBranch<0x25, "jge", CPURegs>;
def JMP     : UncondBranch<0x26, "jmp">;
...
// brcond patterns
multiclass BrcondPats<RegisterClass RC, Instruction JEQOp,
  Instruction JNEOp, Instruction JLTOp, Instruction JGTOp,
  Instruction JLEOp, Instruction JGEOp, Instruction CMPOp,
  Register ZEROReg> {
def : Pat<(brcond (i32 (seteq RC:$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. You can run with them if you like to test more.

Finally, 7/1/Cpu0 support the local array definition by add the LowerCall() empty function in Cpu0ISelLowering.cpp as follows,

```
// Cpu0ISelLowering.cpp
SDValue
Cpu0TargetLowering::LowerCall(SDValue InChain, 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 {
  return InChain;
}
```

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

```
// ch7_1_4.cpp
int main()
{
    int a[3]={0, 1, 2};

    return 0;
}
```

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

```
@_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
```

8.2 RISC CPU knowledge

As mentioned in the previous section, cpu0 is a RISC (Reduced Instruction Set Computer) CPU with 3 stages of pipeline. RISC CPU is full in world. Even the X86 of CISC (Complex Instruction Set Computer) is RISC inside. (It translate CISC instruction into micro-instruction which do pipeline as RISC). Knowledge with RISC will make you satisfied in compiler design. List these two excellent books we have read which include the real RISC CPU knowledge needed for reference. Sure, there are many books in Computer Architecture, and some of them contain real RISC CPU knowledge needed, but these two are what we read.

Computer Organization and Design: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)

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

The book of "Computer Organization and Design: The Hardware/Software Interface" (there are 4 editions until the book is written) is for the introduction (simple). "Computer Architecture: A Quantitative Approach" (there are 5 editions until the book is written) is more complicate and deep in CPU architecture.

CHAPTER
NINE

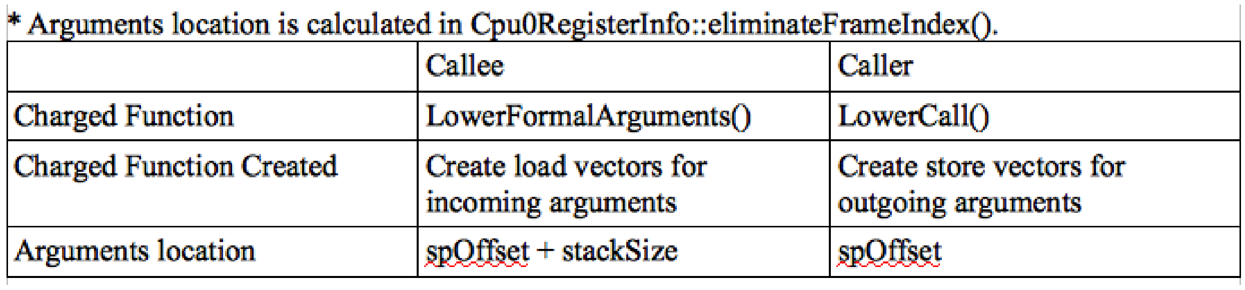
FUNCTION CALL

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.

9.1 Mips stack frame

The first thing for design the cpu0 function call is deciding how to pass arguments in function call. There are two options. The first is pass arguments all in stack. Second is pass arguments in the registers which are reserved for function arguments, and put the other arguments in stack if it over the number of registers reserved for function call. For example, Mips pass the first 4 arguments in register $a0, $a1, $a2, $a3, and the other arguments in stack if it over 4 arguments. *Mips stack frame* is the Mips stack frame.



* 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 9.1: Mips stack frame

Run `llc -march=mips` for ch8_1.bc, you will get the following result.

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

```
// ch8_1.cpp
int sum_i(int x1, int x2, int x3, int x4, int x5, int x6)
{
    int sum = x1 + x2 + x3 + x4 + x5 + x6;

    return sum;
}

int main()
{
    int a = sum_i(1, 2, 3, 4, 5, 6);

    return a;
}

118-165-79-31:InputFiles Jonathan$ clang -c ch8_1.cpp -emit-llvm -o ch8_1.bc
118-165-79-31:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/
cmake_debug_build/bin/Debug/llc -march=mips -relocation-model=pic -filetype=asm
ch8_1.bc -o ch8_1.mips.s
118-165-79-31:InputFiles Jonathan$ cat ch8_1.mips.s
    .section .mdebug.abi32
    .previous
    .file   "ch8_1.bc"
    .text
    .globl  _Z5sum_iiiiiii
    .align  2
    .type   _Z5sum_iiiiiii,@function
    .ent    _Z5sum_iiiiiii          # @_Z5sum_iiiiiii
_Z5sum_iiiiiii:
    .frame  $sp,32,$ra
    .mask   0x00000000,0
    .fmask  0x00000000,0
    .set    noreorder
    .set    nomacro
# BB#0:                                 # %entry
    addiu   $sp, $sp, -32
    st  $4, 28($sp)
    st  $5, 24($sp)
    st  $6, 20($sp)
    st  $7, 16($sp)
    ld  $2, 48($sp) // load argument 5
    st  $2, 12($sp)
    ld  $2, 52($sp) // load argument 6
    st  $2, 8($sp)
    ld  $3, 24($sp)
    ld  $4, 28($sp)
    addu    $3, $4, $3
    ld  $4, 20($sp)
    addu    $3, $3, $4
    ld  $4, 16($sp)
    addu    $3, $3, $4
    ld  $4, 12($sp)
    addu    $3, $3, $4
    addu    $2, $3, $2
    st  $2, 4($sp)
    addiu   $sp, $sp, 32
    jr  $ra
    nop
    .set    macro
    .set    reorder
    .end    _Z5sum_iiiiii
$tmp1:
    .size   _Z5sum_iiiiii, ($tmp1)-_Z5sum_iiiiii

    .globl  main
    .align  2
    .type   main,@function
    .ent    main                    # @main
main:
    .frame  $sp,48,$ra
    .mask   0x80000000,-4
    .fmask  0x00000000,0
    .set    noreorder
    .cpload $25
    .set    nomacro
# BB#0:                                 # %entry
    addiu   $sp, $sp, -48
    st  $ra, 44($sp)            # 4-byte Folded Spill
    .cprestore  24
    st  $zero, 40($sp)
    addiu   $2, $zero, 6
    st  $2, 20($sp) // Save argument 6 to 20($sp)
    addiu   $2, $zero, 5
    st  $2, 16($sp) // Save argument 5 to 16($sp)
    ld  $25, %call16(_Z5sum_iiiiii)($gp)
    addiu   $4, $zero, 1    // Pass argument 1 to $4 (=$a0)
    addiu   $5, $zero, 2    // Pass argument 2 to $5 (=$a1)
    addiu   $6, $zero, 3
    addiu   $7, $zero, 4
    jalr    $25
    nop
    ld  $gp, 24($sp)
    st  $2, 36($sp)
    ld  $ra, 44($sp)            # 4-byte Folded Reload
    addiu   $sp, $sp, 48
    jr  $ra
    nop
    .set    macro
    .set    reorder
    .end    main
$tmp4:
    .size   main, ($tmp4)-main
```

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). *Mips arguments location in stack frame* 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 *Mips stack frame*.

[2] https://www.dropbox.com/sh/2pkh1fewlq2zag9/OHnrYn2nOs/doc/MIPSproAssemblyLanguageProgrammerGuide
[3] http://www.linux-mips.org/pub/linux/mips/doc/ABI/mipsabi.pdf

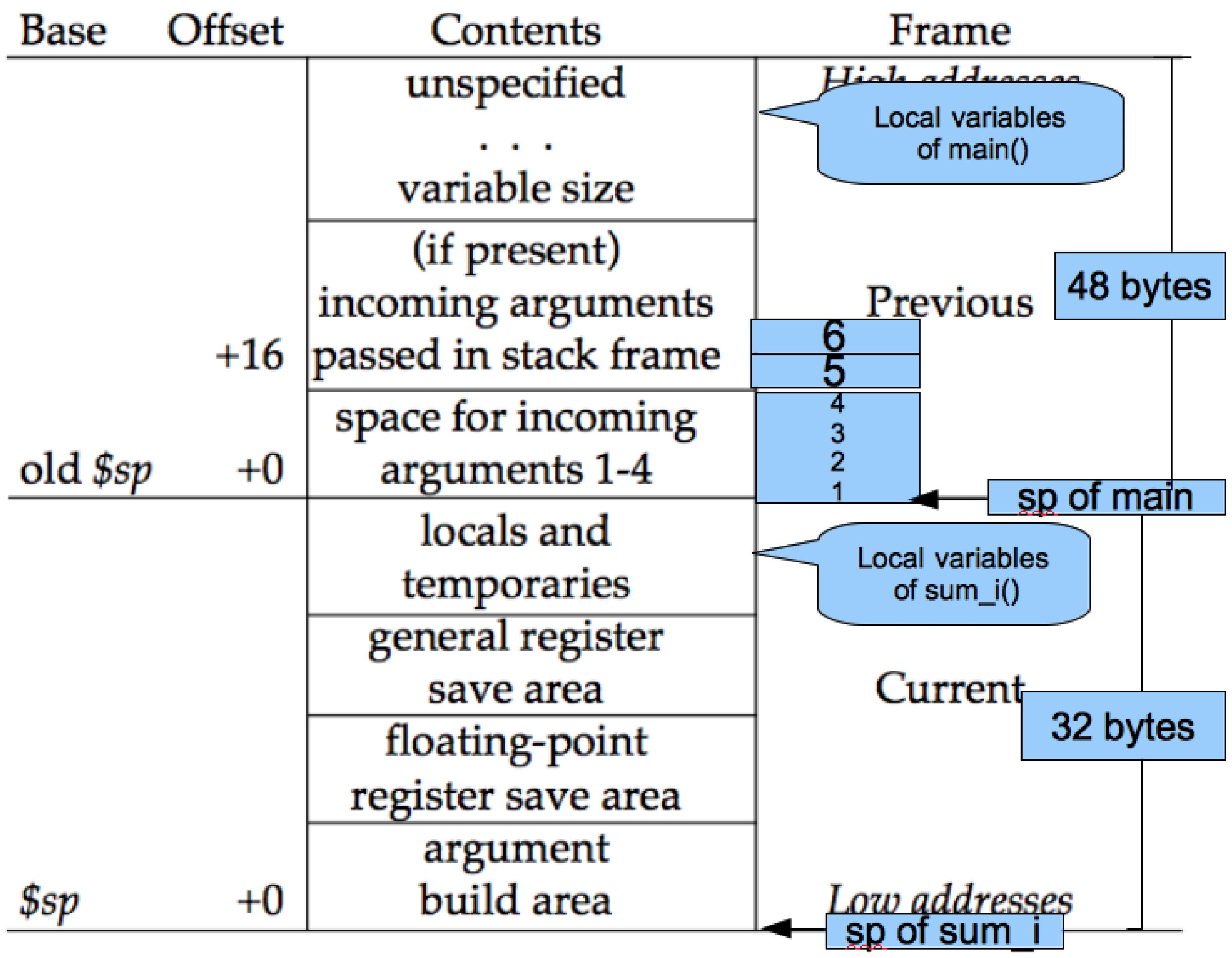


Figure 9.2: Mips arguments location in stack frame

9.2 Load incoming arguments from stack frame

From last section, to support function call, we need implementing the arguments pass mechanism with stack frame. Before do that, let's run the old version of code 7/1/Cpu0 with ch8_1.cpp and see what happen.

```
118-165-79-31:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/
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/3.1.test/cpu0/1/src/lib/CodeGen/SelectionDAG/
SelectionDAGBuilder.cpp, line 6671.
Stack dump:
0.  Program arguments: /Users/Jonathan/llvm/3.1.test/cpu0/1/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 7/1/Cpu0 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,

```
// Cpu0CallingConv.td
...
def RetCC_Cpu0EABI : CallingConv<[
  // i32 are returned in registers V0, V1
  CCIfType<[i32], CCAssignToReg<[V0, V1]>>
]>;

//===----------------------------------------------------------------------===//
// Cpu0 EABI Calling Convention
//===----------------------------------------------------------------------===//

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


//===----------------------------------------------------------------------===//
// Cpu0 Calling Convention Dispatch
//===----------------------------------------------------------------------===//

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


def RetCC_Cpu0 : CallingConv<[
  CCDelegateTo<RetCC_Cpu0EABI>
]>;
```

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

As above, CC_Cpu0 is the cpu0 Calling Convention which delegate to CC_Cpu0EABI and define the CC_Cpu0EABI. The reason we don't define the Calling Convention directly in CC_Cpu0 is that a real general CPU like Mips can have several Calling Convention. Combine with the mechanism of "section Target Registration" [4] which llvm supplied, we can use different Calling Convention in different target. Although cpu0 only have a Calling Convention right now, define with a dedicate Call Convention name (CC_Cpu0EABI in this example) is a better solution for system expand, and naming your Calling Convention. CC_Cpu0EABI as above, say it pass arguments in stack frame.

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

```
// Cpu0ISelLowering.cpp
...
/// LowerFormalArguments - transform physical registers into virtual registers
/// and generate load operations for arguments places on the stack.
SDValue
Cpu0TargetLowering::LowerFormalArguments(SDValue Chain,
                                         CallingConv::ID CallConv,
                                         bool isVarArg,
                                      const SmallVectorImpl<ISD::InputArg> &Ins,
                                         DebugLoc dl, SelectionDAG &DAG,
                                         SmallVectorImpl<SDValue> &InVals)
                                          const {
  MachineFunction &MF = DAG.getMachineFunction();
  MachineFrameInfo *MFI = MF.getFrameInfo();
  Cpu0FunctionInfo *Cpu0FI = MF.getInfo<Cpu0FunctionInfo>();

  Cpu0FI->setVarArgsFrameIndex(0);

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

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

  CCInfo.AnalyzeFormalArguments(Ins, CC_Cpu0);

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

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

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

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

```
    assert(VA.isMemLoc());

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

    // Create load nodes to retrieve arguments from the stack
    SDValue FIN = DAG.getFrameIndex(LastFI, getPointerTy());
    InVals.push_back(DAG.getLoad(ValVT, dl, Chain, FIN,
                                 MachinePointerInfo::getFixedStack(LastFI),
                                 false, false, false, 0));
  }
  Cpu0FI->setLastInArgFI(LastFI);
  // All stores are grouped in one node to allow the matching between
  // the size of Ins and InVals. This only happens when on varg functions
  if (!OutChains.empty()) {
    OutChains.push_back(Chain);
    Chain = DAG.getNode(ISD::TokenFactor, dl, MVT::Other,
                        &OutChains[0], OutChains.size());
  }
  return Chain;
}
```

Refresh “section Global variable” [5], we handled global variable translation by create the IR DAG in LowerGlobalAddress() first, and then do the Instruction Selection by their corresponding machine instruction DAG in Cpu0InstrInfo.td. LowerGlobalAddress() is called when `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.

```
// Cpu0InstrInfo.cpp
...
static MachineMemOperand* GetMemOperand(MachineBasicBlock &MBB, int FI,
                                        unsigned Flag) {
  MachineFunction &MF = *MBB.getParent();
  MachineFrameInfo &MFI = *MF.getFrameInfo();
  unsigned Align = MFI.getObjectAlignment(FI);

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

void Cpu0InstrInfo::
loadRegFromStackSlot(MachineBasicBlock &MBB, MachineBasicBlock::iterator I,
```

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

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

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

In addition to Calling Convention and LowerFormalArguments(), 8/2/Cpu0 add the following code for cpu0 instructions swi (Software Interrupt), jsub and jalr (function call) definition and printing.

```
// 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;
}

// 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 JALR : JumpLinkReg<0x2D, "jalr", CPURegs>;
...
def : Pat<(Cpu0JmpLink (i32 tglobaladdr:$dst)),
          (JSUB tglobaladdr:$dst)>;
...

// Cpu0InstPrinter.cpp
...
static void printExpr(const MCExpr *Expr, raw_ostream &OS) {
  switch (Kind) {
  ...
  case MCSymbolRefExpr::VK_Cpu0_GOT_CALL:  OS << "%call24("; break;
  ...
  }
...
}

// 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;
  ...
  }
...
}

// 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 8/2/Cpu0 with ch8_1.cpp and see what happens in the following,

```
118-165-79-83:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch8_1.bc -o ch8_1.cpu0.s
Assertion failed: ((isTailCall || InVals.size() == Ins.size()) &&
"LowerCall didn't emit the correct number of values!"), function LowerCallTo,
file /Users/Jonathan/llvm/3.1.test/cpu0/1/src/lib/CodeGen/SelectionDAG/
SelectionDAGBuilder.cpp, line 6482.
Stack dump:
0.  Program arguments: /Users/Jonathan/llvm/3.1.test/cpu0/1/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
```

9.3 Store outgoing arguments to stack frame

Mips arguments location in stack frame 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,

```
// Cpu0ISelLowering.cpp
...
SDValue
Cpu0TargetLowering::LowerCall(SDValue InChain, 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 {
  // Cpu0 target does not yet support tail call optimization.
  isTailCall = false;

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

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

  CCInfo.AnalyzeCallOperands(Outs, CC_Cpu0);

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

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

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

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

```
                      StackAlignment * StackAlignment;

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

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

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

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

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

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

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

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

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

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

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

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

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

SDValue InFlag;

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

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

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

```
    Callee = DAG.getRegister(T9Reg, getPointerTy());
  }

  // Cpu0JmpLink = #chain, #target_address, #opt_in_flags...
  //             = Chain, Callee, Reg#1, Reg#2, ...
  //
  // Returns a chain & a flag for retval copy to use.
  SDVTList NodeTys = DAG.getVTList(MVT::Other, MVT::Glue);
  SmallVector<SDValue, 8> Ops;
  Ops.push_back(Chain);
  Ops.push_back(Callee);

  // Add argument registers to the end of the list so that they are
  // known live into the call.
  for (unsigned i = 0, e = RegsToPass.size(); i != e; ++i)
    Ops.push_back(DAG.getRegister(RegsToPass[i].first,
                                  RegsToPass[i].second.getValueType()));

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

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

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

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

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

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

  CCInfo.AnalyzeCallResult(Ins, RetCC_Cpu0);

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

  return Chain;
}
```

Just like load incoming arguments from stack frame, we call CCInfo(CallConv,..., ArgLocs, …) to get outgoing arguments information before enter **“for loop”** and set stack alignment with 8 bytes. They’re almost same in **“for loop”** with LowerFormalArguments(), except LowerCall() create store DAG vector instead of load DAG vector. After the **“for loop”**, it create **“ld $6, %call24(_Z5sum_iiiiii)($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.

```
// 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.

.. code-block:: c++

// 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 8/3/Cpu0 with ch8_1.cpp to get result as follows,

```
118-165-79-83:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=asm
ch8_1.bc -o ch8_1.cpu0.s
118-165-79-83: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:
    .frame  $sp,32,$lr
    .mask   0x00000000,0
    .set    noreorder
    .set    nomacro
# BB#0:                                     # %entry
    addiu   $sp, $sp, -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_iiiiii
$tmp1:
    .size   _Z5sum_iiiiii, ($tmp1)-_Z5sum_iiiiii

    .globl  main
    .align  2
    .type   main,@function
    .ent    main                    # @main
main:
    .frame  $sp,40,$lr
    .mask   0x00004000,-4
    .set    noreorder
    .cpload $t9
    .set    nomacro
# BB#0:                                 # %entry
    addiu   $sp, $sp, -40
    st  $lr, 36($sp)            # 4-byte Folded Spill
    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  $6, %call24(_Z5sum_iiiiii)($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
$tmp4:
    .size   main, ($tmp4)-main
```

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

9.4 Fix the wrong offset in storing arguments to stack frame

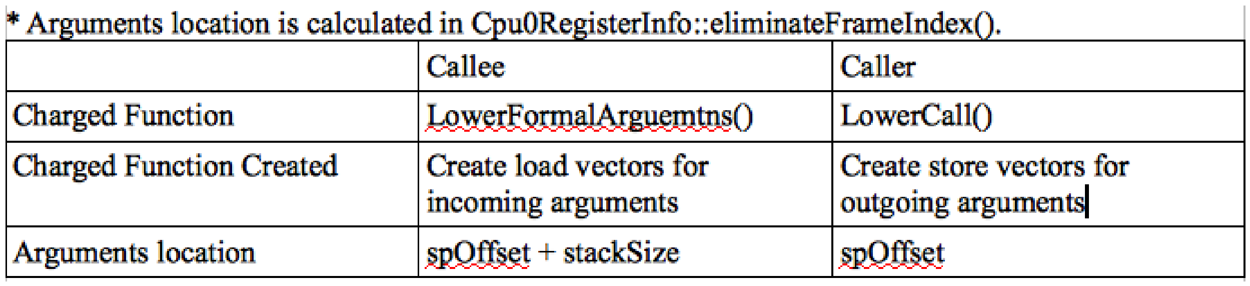
To fix the wrong offset in storing arguments, we modify the following code in eliminateFrameIndex() as follows. The bold text as below is added in 8/4/Cpu0 to set the caller outgoing arguments into spOffset($sp) (8/3/Cpu0 set them to pOffset+stackSize($sp).

```
// 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();

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

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

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

}

// Cpu0MachineFunction.h
  ...
  bool isOutArgFI(int FI) const {
    return FI <= OutArgFIRange.first && FI >= OutArgFIRange.second;
  }
```

Run 8/4/Cpu0 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().

```
118-165-76-131:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype=
asm ch8_1.bc -o ch8_1.cpu0.s
118-165-76-131: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_iiiiii
$tmp2:
  .size _Z5sum_iiiiii, ($tmp2)-_Z5sum_iiiiii
  .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  $2, 20($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  $6, %call24(_Z5sum_iiiiii)($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
```

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 *Callee incoming arguments and caller outgoing arguments*.



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

	Callee	Caller
Charged Function	LowerFormalArguemtns()	LowerCall()
Charged Function Created	Create load vectors for incoming arguments	Create store vectors for outgoing arguments
Arguments location	spOffset + stackSize	spOffset

Figure 9.3: Callee incoming arguments and caller outgoing arguments

9.5 Pseudo hook instruction ADJCALLSTACKDOWN and ADJCALLSTACKUP

To fix the !ADJSTACKDOWN and !1ADJSTACKUP, we call Cpu0GenInstrInfo(Cpu0:: ADJCALLSTACKDOWN, Cpu0::ADJCALLSTACKUP) in Cpu0InstrInfo() constructor function and define eliminateCallFramePseudoInstr() as follows,

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

// Cpu0RegisterInfo.cpp
...
// Cpu0
// This function eliminate ADJCALLSTACKDOWN,
// ADJCALLSTACKUP pseudo instructions
void Cpu0RegisterInfo::
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 8/5/Cpu0 with ch8_1.cpp will get the following result.

```
118-165-76-131:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/
cmake_debug_build/bin/Debug/llc -march=cpu0 -relocation-model=pic -filetype
=asm ch8_1.bc -o ch8_1.cpu0.s
118-165-76-131: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_iiiiii
$tmp2:
  .size _Z5sum_iiiiii, ($tmp2)-_Z5sum_iiiiii
  .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
```

9.6 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 8/6/Cpu0 with ch8_2.cpp to get the following result of we putting the comment in it for explanation.

```
118-165-67-25:InputFiles Jonathan$ cat ch8_2.cpu0.s
    .section .mdebug.abi32
    .previous
    .file   "ch8_2.bc"
    .text
    .globl  _Z5sum_iiiiiii
    .align  2
    .type   _Z5sum_iiiiiii,@function
    .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:
    addiu   $sp, $sp, -32
$tmp1:
    .cfi_def_cfa_offset 32
...
    ld  $3, %got(gI)($gp)   // %got(gI) is offset of (gI - _Z5sum_iiiiiii)
...
    ret $lr
    .set    macro
    .set    reorder
    .end    _Z5sum_iiiiiii
...
    .ent    main                    # @main
main:
    .cfi_startproc
    .frame  $sp,72,$lr
    .mask   0x00004000,-4
    .set    noreorder
    .cpload $t9
    .set    nomacro
# BB#0:
    addiu   $sp, $sp, -72
$tmp5:
    .cfi_def_cfa_offset 72
    st  $lr, 68($sp)            # 4-byte Folded Spill
$tmp6:
    .cfi_offset 14, -4
    .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)
...
    addiu   $sp, $sp, 72
    ret $lr
    .set    macro
    .set    reorder
    .end    main
$tmp7:
    .size   main, ($tmp7)-main
    .cfi_endproc

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

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_iiiiiii() function. In _Z5sum_iiiiiii() function, we translate global variable gI address by **"ld $3, %got(gI)($gp)"** where %got(gI) is offset of (gI - _Z5sum_iiiiiii) (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. Mips use the same solution in 32 bits Mips32 CPU.

Now, as the following code added in 8/6/Cpu0, 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.

```
// # CMakeLists.txt
...
add_llvm_target(Cpu0CodeGen
  ...
  Cpu0EmitGPRestore.cpp
...
```

```
// Cpu0TargetMachine.cpp
...
bool Cpu0PassConfig::addPreRegAlloc() {
  // Do not restore $gp if target is Cpu064.
  // In N32/64, $gp is a callee-saved register.

  PM->add(createCpu0EmitGPRestorePass(getCpu0TargetMachine()));
  return true;
}

// Cpu0.h
  ...
  FunctionPass *createCpu0EmitGPRestorePass(Cpu0TargetMachine &TM);

// 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);
  }
}

// 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", []>;


// Cpu0SelLowering.cpp
...
SDValue
Cpu0TargetLowering::LowerCall(SDValue InChain, 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 {
  ...
  // 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);
  ...
}

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

#define DEBUG_TYPE "emit-gp-restore"

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

using namespace llvm;

namespace {
  struct Inserter : public MachineFunctionPass {

  TargetMachine &TM;
  const TargetInstrInfo *TII;

  static char ID;
  Inserter(TargetMachine &tm)
    : MachineFunctionPass(ID), TM(tm), TII(tm.getInstrInfo()) { }

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

  bool runOnMachineFunction(MachineFunction &F);
  };
  char Inserter::ID = 0;
} // end of anonymous namespace

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

  if ((TM.getRelocationModel() != Reloc::PIC_) ||
    (!Cpu0FI->globalBaseRegFixed()))
  return false;
  bool Changed = false;
  int FI = Cpu0FI->getGPFI();

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

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

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

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

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

  return Changed;
}

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

//===-- 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

//  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);
  }
}

// Cpu0MCInstLower.cpp
...
sstatic void CreateMCInst(MCInst& Inst, unsigned Opc, const MCOperand& Opnd0,
             const MCOperand& Opnd1,
             const MCOperand& Opnd2 = MCOperand()) {
  Inst.setOpcode(Opc);
  Inst.addOperand(Opnd0);
  Inst.addOperand(Opnd1);
  if (Opnd2.isValid())
  Inst.addOperand(Opnd2);
}

// Lower ".cpload $reg" to
//  "addiu $gp, $zero, %hi(_gp_disp)"
//  "shl   $gp, $gp, 16"
//  "addiu $gp, $gp, %lo(_gp_disp)"
//  "addu  $gp, $gp, $t9"
void Cpu0MCInstLower::LowerCPLOAD(SmallVector<MCInst, 4>& MCInsts) {
  MCOperand GPReg = MCOperand::CreateReg(Cpu0::GP);
  MCOperand T9Reg = MCOperand::CreateReg(Cpu0::T9);
  MCOperand ZEROReg = MCOperand::CreateReg(Cpu0::ZERO);
  StringRef SymName("_gp_disp");
  const MCSymbol *Sym = Ctx->GetOrCreateSymbol(SymName);
  const MCSymbolRefExpr *MCSym;

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

  MCInsts.resize(4);

  CreateMCInst(MCInsts[0], Cpu0::ADDiu, GPReg, ZEROReg, SymHi);
  CreateMCInst(MCInsts[1], Cpu0::SHL, GPReg, GPReg, MCOperand::CreateImm(16));
  CreateMCInst(MCInsts[2], Cpu0::ADDiu, GPReg, GPReg, SymLo);
  CreateMCInst(MCInsts[3], Cpu0::ADD, GPReg, GPReg, T9Reg);
}

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

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

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

    // addiu   at,zero,hi
    // shl     at,at,16
    // add     at,at,sp
    MCInsts.resize(3);
    CreateMCInst(MCInsts[0], Cpu0::ADDiu, ATReg, ZEROReg, MCOperand::CreateImm(Hi));
    CreateMCInst(MCInsts[1], Cpu0::SHL, ATReg, ATReg, MCOperand::CreateImm(16));
    CreateMCInst(MCInsts[2], Cpu0::ADD, ATReg, ATReg, SPReg);
  }

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

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/3.1.test/cpu0/1/
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/3.1.test/cpu0/1/
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
```

9.7 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 8/7/Cpu0 with ch8_3.cpp to get the following,

```
// ch8_3.cpp
//#include <stdio.h>
#include <stdarg.h>

int sum_i(int amount, ...)
{
  int i = 0;
  int val = 0;
  int sum = 0;

  va_list vl;
  va_start(vl, amount);
  for (i = 0; i < amount; i++)
  {
  val = va_arg(vl, int);
  sum += val;
  }
  va_end(vl);

  return sum;
}

int main()
{
  int a = sum_i(6, 1, 2, 3, 4, 5, 6);
//  printf("a = %d\n", a);

  return a;
}

118-165-76-131:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/
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:
  .cfi_startproc
  .frame  $sp,56,$lr
  .mask   0x00004000,-4
  .set  noreorder
  .cpload $t9
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -56
$tmp2:
  .cfi_def_cfa_offset 56
  st  $lr, 52($sp)              # 4-byte Folded Spill
$tmp3:
  .cfi_offset 14, -4
  .cprestore  0
  ld  $2, %got(__stack_chk_guard)($gp)
  ld  $2, 0($2)
  st  $2, 48($sp) // 48($sp) = 0
  ld  $2, 56($sp) // amount
  st  $2, 44($sp) // amount
  addiu $2, $zero, 0
  st  $2, 40($sp) // i   = 0
  st  $2, 36($sp) // val = 0
  st  $2, 32($sp) // sum = 0
  addiu $3, $sp, 48 // $3 = 48($sp)
  st  $3, 8($sp)  // 8($sp) = 48($sp) = arg_ptr
  st  $2, 40($sp) // i = 0
  addiu $2, $zero, 40 // $2 = 40
$BB0_1:                                 # =>This Inner Loop Header: Depth=1
  ld  $3, 44($sp) // $3 = amount
  ld  $4, 40($sp) // $4 = i
  cmp $4, $3
  jge $BB0_7    // i >= amount
  jmp $BB0_2
$BB0_2:                                 #   in Loop: Header=BB0_1 Depth=1
                // i < amount
  ld  $3, 8($sp)  // $3 = arg_ptr
  cmp $3, $2
  jgt $BB0_4    // arg_ptr > 40
  jmp $BB0_3
$BB0_3:                                 #   in Loop: Header=BB0_1 Depth=1
                // arg_ptr <= 40
  addiu $4, $3, 8
  ld  $5, 20($sp) // *(20($sp)) = arg_offset = 12
  st  $4, 8($sp)  // arg_ptr += 8
  add $3, $5, $3  // $3 = (arg_ptr + arg_offset)
  jmp $BB0_5
$BB0_4:                                 #   in Loop: Header=BB0_1 Depth=1
  ld  $3, 16($sp)
  addiu $4, $3, 8
  st  $4, 16($sp)
$BB0_5:                                 #   in Loop: Header=BB0_1 Depth=1
  ld  $3, 0($3)   // $3 = val = *(arg_ptr + arg_offset)
  st  $3, 36($sp)
  ld  $4, 32($sp) // $4 = sum
  add $3, $4, $3
  st  $3, 32($sp) // sum += val
# BB#6:                                 #   in Loop: Header=BB0_1 Depth=1
  ld  $3, 40($sp) // $3 = i
  addiu $3, $3, 1
  st  $3, 40($sp) // i = i + 1
  jmp $BB0_1
$BB0_7:
  ld  $2, %got(__stack_chk_guard)($gp)
  ld  $2, 0($2)
  ld  $3, 48($sp)
  cmp $2, $3
  jne $BB0_9
  jmp $BB0_8
$BB0_8:                                 # %SP_return
  ld  $lr, 52($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 56
  ret $lr
$BB0_9:                                 # %CallStackCheckFailBlk
```

```
  ld  $6, %call24(__stack_chk_fail)($gp)
  jalr  $6
  ld  $gp, 0($sp)
  .set  macro
  .set  reorder
  .end  _Z5sum_iiz
$tmp4:
  .size _Z5sum_iiz, ($tmp4)-_Z5sum_iiz
  .cfi_endproc

  .globl  main
  .align  2
  .type main,@function
  .ent  main                    # @main
main:
  .cfi_startproc
  .frame  $sp,88,$lr
  .mask   0x00004000,-4
  .set  noreorder
  .cpload $t9
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -88
$tmp7:
  .cfi_def_cfa_offset 88
  st  $lr, 84($sp)            # 4-byte Folded Spill
$tmp8:
  .cfi_offset 14, -4
  .cprestore  32
  addiu $2, $zero, 0
  st  $2, 80($sp)
  addiu $2, $zero, 5
  st  $2, 20($sp)
  addiu $2, $zero, 4
  st  $2, 16($sp)
  addiu $2, $zero, 3
  st  $2, 12($sp)
  addiu $2, $zero, 2
  st  $2, 8($sp)
  addiu $2, $zero, 1
  st  $2, 4($sp)
  addiu $2, $zero, 6
  st  $2, 24($sp)
  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
$tmp9:
  .size main, ($tmp9)-main
    .cfi_endproc
```

We have problem in analysis of the output ch8_3.cpu0.s. We guess and try to analysis as follows. As above code, we get the first argument **"amount"** from **"ld $2, 56($sp)"** since the stack size of the callee function **"_Z5sum_iiz()"** is 56. Next, check i < amount in block $BB0_1. If i < amount, than enter into $BB0_2. We assume arg_ptr < 40 and the content of address 8($sp) is the arg_ptr. When it exits $BB0_2 and enter into $BB0_3, the register ($3 + $5) = (arg_ptr + arg_offset=12) is point to the second argument and it do the sum += val in $BB0_5. It do i += 1 in $BB0_6 and jumb to $BB0_1 enter into second round. The second round do as above again, it will get the third argument and add to sum in $BB0_5 since the ptr_arg (16($sp)) is added 8 in the previous run. We assume the arg_prt < 40 but actually according the analysis the arg_prt is 48($sp) which > 40, so the above analysis is not satisfied. The compare arg_prt with 40 is exist in llvm IR, and mips has the same translated output. So, we don't know what's wrong. We believe the arg < 40 is satisfied because the native Intel CPU has the arg_ptr < 40 in it's assembly code and the Intel CPU native execution file can print correct result. You will see it soon in the bellow code. If the arg_ptr < 40 is satisfied and *(20($sp)) = arg_offset = 12, then the assembly output is correct. The llvm IR and mips assembly output as follows,

```
118-165-78-221:InputFiles Jonathan$ llvm-dis ch8_3.bc -o ch8_3.ll
118-165-78-221:InputFiles Jonathan$ cat ch8_3.ll
; ModuleID = 'ch8_3.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"

%struct.__va_list_tag = type { i32, i32, i8*, i8* }

define i32 @_Z5sum_iiz(i32 %amount, ...) nounwind uwtable ssp {
...
; <label>:8                                       ; preds = %4
...
  %12 = icmp ule i32 %11, 40
  br i1 %12, label %13, label %19

118-165-67-185:InputFiles Jonathan$ cat ch8_3.mips.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:
  .cfi_startproc
  .frame  $sp,72,$ra
  .mask   0x80000000,-4
  .fmask  0x00000000,0
  .set  noreorder
  .cpload $25
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -72
$tmp2:
  .cfi_def_cfa_offset 72
  sw  $ra, 68($sp)            # 4-byte Folded Spill
$tmp3:
  .cfi_offset 31, -4
  .cprestore  16
  sw  $7, 84($sp)
  sw  $6, 80($sp)
  sw  $5, 76($sp) // 76($sp) = arg[1]
  lw  $2, %got(__stack_chk_guard)($gp)
  lw  $2, 0($2)
  sw  $2, 64($sp)
  sw  $4, 60($sp) // 60($sp) = amount = arg[0]
  sw  $zero, 56($sp)  // i
  sw  $zero, 52($sp)  // val
  sw  $zero, 48($sp)  // sum
  addiu $2, $sp, 76
  sw  $2, 24($sp) // 24($sp) = arg_ptr
  sw  $zero, 56($sp)
  addiu $2, $zero, 40 // $2 = 40
  b $BB0_1
  nop
$BB0_5:                                 #   in Loop: Header=BB0_1 Depth=1
  lw  $3, 0($3)   // $3 = *arg_ptr
  sw  $3, 52($sp) // val
  lw  $4, 48($sp) // sum
  addu  $3, $4, $3  //
  sw  $3, 48($sp) // sum += val
  lw  $3, 56($sp)
  addiu $3, $3, 1
  sw  $3, 56($sp) // i += 1
$BB0_1:                                 # =>This Inner Loop Header: Depth=1
  lw  $3, 60($sp)
  lw  $4, 56($sp)
  slt $3, $4, $3  // set if i < amount
  beq $3, $zero, $BB0_6 // i >= amount
  nop
# BB#2:                                 #   in Loop: Header=BB0_1 Depth=1
  lw  $3, 24($sp) // $3 = arg_ptr
  sltu  $4, $2, $3  // set if 40 < arg_ptr
  bne $4, $zero, $BB0_4
  nop
# BB#3:                                 #   in Loop: Header=BB0_1 Depth=1
                    // arg_ptr <= 40
  addiu $4, $3, 8
  lw  $5, 36($sp) // 36($sp) = 0, assume even though we didn't find the
                // 36($sp) is 0
  sw  $4, 24($sp) // arg_ptr += 8
  addu  $3, $5, $3  // arg_ptr + 0
  b $BB0_5
  nop
$BB0_4:                                 #   in Loop: Header=BB0_1 Depth=1
                   // 40 < arg_ptr
  lw  $3, 32($sp)
  addiu $4, $3, 8
  sw  $4, 32($sp)
  b $BB0_5
  nop
$BB0_6:
  lw  $2, %got(__stack_chk_guard)($gp)
  lw  $2, 0($2)
  lw  $3, 64($sp)
  bne $2, $3, $BB0_8
  nop
# BB#7:                                 # %SP_return
  lw  $2, 48($sp)
  lw  $ra, 68($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 72
  jr  $ra
  nop
$BB0_8:                               # %CallStackCheckFailBlk
  lw  $25, %call16(__stack_chk_fail)($gp)
  jalr  $25
  nop
  lw  $gp, 16($sp)
  .set  macro
  .set  reorder
  .end  _Z5sum_iiz
$tmp4:
  .size _Z5sum_iiz, ($tmp4)-_Z5sum_iiz
  .cfi_endproc

  .globl  main
  .align  2
  .type main,@function
  .ent  main                    # @main
main:
  .cfi_startproc
  .frame  $sp,64,$ra
  .mask   0x80000000,-4
  .fmask  0x00000000,0
  .set  noreorder
  .cpload $25
  .set  nomacro
# BB#0:
  addiu $sp, $sp, -64
$tmp7:
  .cfi_def_cfa_offset 64
  sw  $ra, 60($sp)            # 4-byte Folded Spill
$tmp8:
  .cfi_offset 31, -4
  .cprestore  40
  sw  $zero, 56($sp)
  addiu $2, $zero, 7
  sw  $2, 28($sp)
  addiu $2, $zero, 6
  sw  $2, 24($sp)
  addiu $2, $zero, 5
  sw  $2, 20($sp)
  addiu $2, $zero, 4
  sw  $2, 16($sp)
  addiu $4, $zero, 8
  sw  $4, 32($sp)
  lw  $25, %call16(_Z5sum_iiz)($gp)
  addiu $5, $zero, 1
  addiu $6, $zero, 2
  addiu $7, $zero, 3
  jalr  $25
  nop
  lw  $gp, 40($sp)
  sw  $2, 52($sp)
  addu  $2, $zero, $zero
  lw  $ra, 60($sp)            # 4-byte Folded Reload
  addiu $sp, $sp, 64
  jr  $ra
  nop
  .set  macro
  .set  reorder
  .end  main
$tmp9:
  .size main, ($tmp9)-main
  .cfi_endproc
```

We have verified the translation of ch8_3.cpp is correct by add printf in ch8_3.cpp to get ch8_3_3.cpp and run with `lli` llvm interpreter. We also translate it into native Intel CPU code and get the correct print result. Following are the ch8_3_3.cpp, and lli, Intel native code run result.

```
// ch8_3_3.cpp
// 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
// /Users/Jonathan/llvm/3.1.test/cpu0/1/cmake_debug_build/bin/Debug/llc
ch8_3_3.bc -o ch8_3_3.s
// clang++ ch8_3_3.s -o ch8_3_3.native
// ./ch8_3_3.native
// lldb -- ch8_3_3.native
// b main
// s
// ...
// print $rsp   ; print %rsp, choose $ instead of % in assembly code

// mips-linux-gnu-g++ -g ch8_3_3.cpp -o ch8_3_3 -static
// qemu-mips ch8_3_3
// mips-linux-gnu-g++ -S ch8_3_3.cpp
// cat ch8_3_3.s

#include <stdio.h>
#include <stdarg.h>

int sum_i(int amount, ...)
{
  int i = 0;
  int val = 0;
  int sum = 0;

  va_list vl;
  va_start(vl, amount);
  for (i = 0; i < amount; i++)
  {
  val = va_arg(vl, int);
  sum += val;
  }
  va_end(vl);

  return sum;
}

int main()
{
  int a = sum_i(6, 1, 2, 3, 4, 5, 6);
  printf("a = %d\n", a);

  return a;
}

118-165-78-221:InputFiles Jonathan$ lli ch8_3_3.bc
a = 21

118-165-67-185: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
118-165-67-185:InputFiles Jonathan$ /Users/Jonathan/llvm/3.1.test/cpu0/1/
cmake_debug_build/bin/Debug/llc ch8_3_3.bc -o ch8_3_3.s
118-165-67-185:InputFiles Jonathan$ clang++ ch8_3_3.s -o ch8_3_3.native
118-165-67-185:InputFiles Jonathan$ ./ch8_3_3.native
a = 21

118-165-67-185:InputFiles Jonathan$ cat ch8_3_3.s
...
LBB0_3:                                 ## =>This Inner Loop Header: Depth=1
  movl  216(%rsp), %eax
  cmpl  220(%rsp), %eax
  jge LBB0_8    // i >= amount
## BB#4:                                ##   in Loop: Header=BB0_3 Depth=1
  movl  176(%rsp), %eax // i < amount
  cmpl  $40, %eax // arg_ptr < 40
  ja  LBB0_6
```

We have run mips qemu on Linux by gcc. It get the correct print result, and the mips code has no analysis problem since it hasn't the arg_ptr < 40 in assembly output. The qemu mips gcc result as follows,

```
[Gamma@localhost InputFiles]$ qemu-mips ch8_3_3
a = 21
[Gamma@localhost InputFiles]$ mips-linux-gnu-g++ -g ch8_3_3.cpp -o ch8_3_3 -static
[Gamma@localhost InputFiles]$ qemu-mips ch8_3_3
a = 21
[Gamma@localhost InputFiles]$ mips-linux-gnu-g++ -S ch8_3_3.cpp
[Gamma@localhost InputFiles]$ cat ch8_3_3.s
  .file 1 "ch8_3_3.cpp"
  .section .mdebug.abi32
  .previous
  .gnu_attribute 4, 1
  .abicalls
  .option pic0
  .text
  .align  2
  .globl  _Z5sum_iiz
$LFB0 = .
  .set  nomips16
  .ent  _Z5sum_iiz
  .type _Z5sum_iiz, @function
_Z5sum_iiz:
  .frame  $fp,32,$31    # vars= 16, regs= 1/0, args= 0, gp= 8
  .mask 0x40000000,-4
  .fmask  0x00000000,0
  .set  noreorder
  .set  nomacro

  addiu $sp,$sp,-32
$LCFI0:
  sw  $fp,28($sp)
```

```
$LCFI1:
  move  $fp,$sp
$LCFI2:
  sw  $5,36($fp)  // arg[1]
  sw  $6,40($fp)
  sw  $7,44($fp)
  sw  $4,32($fp)  // amount = arg[0]
  sw  $0,16($fp)  // i = 0
  sw  $0,12($fp)  // val = 0
  sw  $0,8($fp)   // sum = 0
  addiu $2,$fp,36
  sw  $2,20($fp)  // arg_ptr = &arg[1]
  sw  $0,16($fp)
  j $L2
  nop

$L3:              // i < amount
  lw  $2,20($fp)  // arg_ptr
  addiu $3,$2,4
  sw  $3,20($fp)  // arg_ptr += 4
  lw  $2,0($2)    // $2 = *arg_ptr
  sw  $2,12($fp)  // val = *arg_ptr
  lw  $3,8($fp)
  lw  $2,12($fp)
  addu  $2,$3,$2
  sw  $2,8($fp)   // sum += val
  lw  $2,16($fp)
  addiu $2,$2,1
  sw  $2,16($fp)  // i += 1
$L2:
  lw  $3,16($fp)
  lw  $2,32($fp)
  slt $2,$3,$2  // set if i < amount
  andi  $2,$2,0x00ff
  bne $2,$0,$L3
  nop

  lw  $2,8($fp)  // i >= amount
  move  $sp,$fp
  lw  $fp,28($sp)
  addiu $sp,$sp,32
  j $31
  nop

  .set  macro
  .set  reorder
  .end  _Z5sum_iiz
$LFE0:
  .size _Z5sum_iiz, .-_Z5sum_iiz
  .rdata
  .align  2
$LC0:
  .ascii  "a = %d\012\000"
  .text
  .align  2
  .globl  main
$LFB1 = .
  .set  nomips16
  .ent  main
  .type main, @function
main:
  .frame  $fp,56,$31    # vars= 8, regs= 2/0, args= 32, gp= 8
  .mask 0xc0000000,-4
  .fmask  0x00000000,0
  .set  noreorder
  .set  nomacro

  addiu $sp,$sp,-56
$LCFI3:
  sw  $31,52($sp)
$LCFI4:
  sw  $fp,48($sp)
$LCFI5:
  move  $fp,$sp
$LCFI6:
  li  $2,4      # 0x4
  sw  $2,16($sp)
  li  $2,5      # 0x5
  sw  $2,20($sp)
  li  $2,6      # 0x6
  sw  $2,24($sp)
  li  $4,6      # 0x6
  li  $5,1      # 0x1
  li  $6,2      # 0x2
  li  $7,3      # 0x3
  jal _Z5sum_iiz
  nop

  sw  $2,40($fp)
  lui $2,%hi($LC0)
  addiu $4,$2,%lo($LC0)
  lw  $5,40($fp)
  jal printf
  nop

  lw  $2,40($fp)
  move  $sp,$fp
  lw  $31,52($sp)
  lw  $fp,48($sp)
  addiu $sp,$sp,56
  j $31
  nop

  .set  macro
  .set  reorder
  .end  main
$LFE1:
  .size main, .-main
  .section  .eh_frame,"a",@progbits
$Lframe1:
  .4byte  $LECIE1-$LSCIE1
$LSCIE1:
  .4byte  0x0
  .byte 0x1
  .globl  __gxx_personality_v0
  .ascii  "zP\000"
  .uleb128 0x1
  .sleb128 -4
  .byte 0x1f
  .uleb128 0x5
  .byte 0x0
  .4byte  __gxx_personality_v0
  .byte 0xc
  .uleb128 0x1d
  .uleb128 0x0
  .align  2
$LECIE1:
$LSFDE3:
  .4byte  $LEFDE3-$LASFDE3
$LASFDE3:
  .4byte  $LASFDE3-$Lframe1
  .4byte  $LFB1
  .4byte  $LFE1-$LFB1
  .uleb128 0x0
  .byte 0x4
  .4byte  $LCFI3-$LFB1
  .byte 0xe
  .uleb128 0x38
  .byte 0x4
  .4byte  $LCFI5-$LCFI3
  .byte 0x11
  .uleb128 0x1e
  .sleb128 2
  .byte 0x11
  .uleb128 0x1f
  .sleb128 1
  .byte 0x4
  .4byte  $LCFI6-$LCFI5
  .byte 0xd
  .uleb128 0x1e
  .align  2
$LEFDE3:
  .ident  "GCC: (GNU) 4.4.6"
[Gamma@localhost InputFiles]$
```

To support variable number of arguments, the following code needed to add in 8/7/Cpu0. The ch8_3_2.cpp is C++ template example code, it can be translated into cpu0 backend code too.

```
// 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);
}
```

```
// ch8_3_2.cpp
...
//#include <stdio.h>
#include <stdarg.h>

template<class T>
T sum(T amount, ...)
{
  T i = 0;
  T val = 0;
  T sum = 0;

  va_list vl;
  va_start(vl, amount);
  for (i = 0; i < amount; i++)
  {
  val = va_arg(vl, T);
  sum += val;
  }
  va_end(vl);

  return sum;
}

int main()
{
  int a = sum<int>(6, 1, 2, 3, 4, 5, 6);
//  printf("a = %d\n", a);

  return a;
}
```

Mips qemu reference .

9.8 Verify DIV for operator %

Now, let’s run 8/7/Cpu0 with 4_6_2.cpp to get the result as below. It translate **“(b+1)%c”** into **“div $zero, $3, $2”** and **“mfhi $2”**.

```
// ch4_6_2.cpp
#include <stdlib.h>

int main()
{
  int b = 11;
//  unsigned int b = 11;
  int c = rand();

  b = (b+1)%c;

  return b;
}
```

```
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/3.1.test/cpu0/1/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 $zero, $3, $2
  mfhi  $2
  ...
```

9.9 Summary of this chapter

Until now, we have 5,700 lines of source code around in 8/7/Cpu0. 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 87% more of code, it can translate tens of instructions, global variable, control flow statement and function call.

CHAPTER

TEN

ELF

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 but it's only support the native CPU. The binutils support other CPU ELF dump as a cross compiler tool chains. Linux platform has binutils already and no need to install it further. We use Linux binutils in this chapter just because iMac will display Chinese text. The iMac corresponding binutils have no problem except it use add g in command, for example, use gobjdump instead of objdump, and display your area language instead of pure English.

The binutils tool we use is not a part of llvm tools, but it's a powerful tool in ELF analysis. This chapter introduce the tool to readers since we think it is a valuable knowledge in this popular ELF format and the ELF binutils analysis tool. An LLVM compiler engineer has the responsibility to analyze the ELF since the obj is need to be handled by linker or loader later. With this tool, you can verify your generated ELF format.

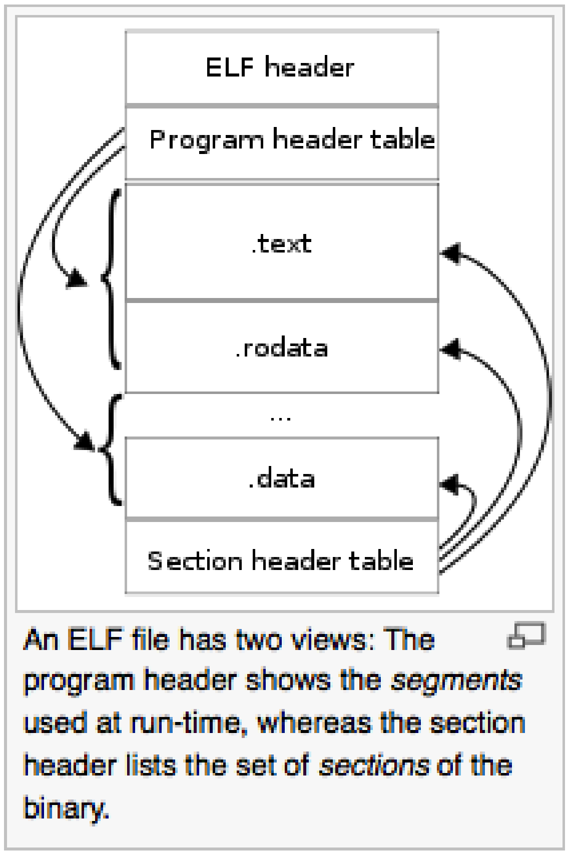
The cpu0 author has published a "System Software" book which introduce the topics of assembler, linker, loader, compiler and OS in concept, and at same time demonstrate how to use binutils and gcc to analysis ELF through the example code in his book. It's a Chinese book of "System Software" in concept and practice. This book does the real analysis through binutils. The "System Software"[3] written by Beck is a famous book in concept of telling readers what is the compiler output, what is the linker output, what is the loader output, and how they work together. But it covers the concept only. You can reference it to understand how the **"Relocation Record"** works if you need to refresh or learning this knowledge for this chapter.

[4], [5], [6] are the Chinese documents available from the cpu0 author on web site.

10.1 ELF format

ELF is a format used both in obj and executable file. So, there are two views in it as *ELF file format overview*.

[1] http://en.wikipedia.org/wiki/Executable_and_Linkable_Format
[2] http://jonathan2251.github.com/lbd/install.html#install-other-tools-on-imac
[3] Leland Beck, System Software: An Introduction to Systems Programming.
[4] http://ccckmit.wikidot.com/lk:aout
[5] http://ccckmit.wikidot.com/lk:objfile
[6] http://ccckmit.wikidot.com/lk:elf



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 10.1: ELF file format overview

As *ELF file format overview*, 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.

10.2 ELF header and Section header table

Let's run 7/7/Cpu0 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/3.1.test/cpu0/1/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/3.1.test/cpu0/1/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, . . . , . 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 execute. 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]$
```

10.3 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/3.1.test/cpu0/1/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/3.1.test/cpu0/1/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/3.1.test/cpu0/1/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/3.1.test/cpu0/1/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]$
```

10.4 Cpu0 ELF related files

Files Cpu0ELFObjectWrite.cpp and Cpu0MC*.cpp are the files take care the obj format. Most obj code translation are defined by Cpu0InstrInfo.td and Cpu0RegisterInfo.td. With these td description, LLVM translate the instruction into obj format automatically.

10.5 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 it's web site information. Please visit the web site [7].

[7] http://lld.llvm.org/

CHAPTER

ELEVEN

TODO LIST

Todo

Add info about LLVM documentation licensing.

(The *original entry* is located in /z/tmp/chenwj/lbd/source/about.rst, line 52.)

Todo

Fix centering for figure captions.

(The *original entry* is located in /z/tmp/chenwj/lbd/source/appendix.rst, line 11.)

Todo

Should we just write out commands in a terminal for people to execute?

(The *original entry* is located in /z/tmp/chenwj/lbd/source/appendix.rst, line 20.)

Todo

The html will follow the appear order in *.rst source context but latexpdf didn't. For example, the *Create LLVM.xcodeproj by cmake – Set option to generate Xcode project* Figure 2.4 and *Create LLVM.xcodeproj by cmake – Before Adjust CMAKE_INSTALL_NAME_TOOL* Figure 2.5 appear after the below text "Click OK from ..." in pdf. If find the **NoReorder** or **newpage** directive, maybe can solve this problem.

(The *original entry* is located in /z/tmp/chenwj/lbd/source/appendix.rst, line 89.)

CHAPTER

TWELVE

BOOK EXAMPLE CODE

The example code is available in:

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

CHAPTER

THIRTEEN

ALTERNATE FORMATS

The book is also available in: