Cpu0 architecture and LLVM structure

Before you begin this tutorial, you should know that you can always try to develop your own backend by porting code from existing backends. The majority of the code you will want to investigate can be found in the /lib/Target directory of your root LLVM installation. As most major RISC instruction sets have some similarities, this may be the avenue you might try if you are an experienced programmer and knowledgable of compiler backends.

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

This chapter details the Cpu0 instruction set and the structure of LLVM. The LLVM structure information is adapted from Chris Lattner's LLVM chapter of the Architecture of Open Source Applications book [10]. You can read the original article from the AOSA website if you prefer.

At the end of this Chapter, you will begin to create a new LLVM backend by writing register and instruction definitions in the Target Description files which will be used in next chapter.

Finally, there are compiler knowledge like DAG (Directed-Acyclic-Graph) and instruction selection needed in llvm backend design, and they are explained here.

Cpu0 Processor Architecture Details

This section is based on materials available here [1] (Chinese) and here [2] (English). However, I changed some ISA from original Cpu0 for designing a simple integer operational CPU and llvm backend. This is my intention for writing this book that I want to know what a simple and robotic CPU ISA and llvm backend can be.

Brief introduction

Cpu0 is a 32-bit architecture. It has 16 general purpose registers (R0, …, R15), co-processor registers (like Mips), and other special registers. Its structure is illustrated in **Fig. 3** below.

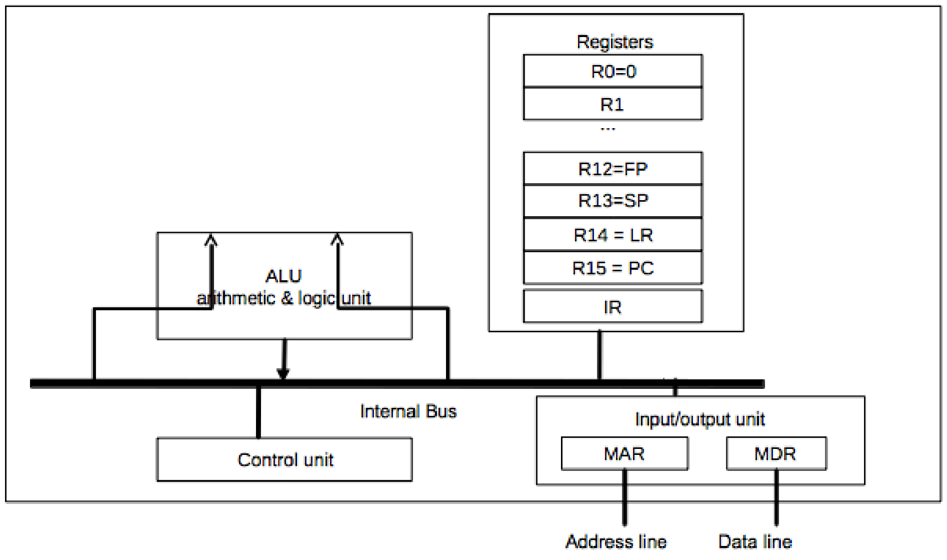


Fig. 3 Architectural block diagram of the Cpu0 processor

The registers are used for the following purposes:

Table 2 Cpu0 general purpose registers (GPR)

Register	Description
R0	Constant register, value is 0
R1-R10	General-purpose registers
R11	Global Pointer register (GP)
R12	Frame Pointer register (FP)
R13	Stack Pointer register (SP)
R14	Link Register (LR)
R15	Status Word Register (SW)

Table 3 Cpu0 co-processor 0 registers (C0R)

Register	Description
0	Program Counter (PC)
1	Error Program Counter (EPC)

Table 4 Cpu0 other registers

Register	Description
IR	Instruction register
MAR	Memory Address Register (MAR)
MDR	Memory Data Register (MDR)
HI	High part of MULT result
LO	Low part of MULT result

The Cpu0 Instruction Set

The Cpu0 instruction set can be divided into three types: L-type instructions, which are generally associated with memory operations, A-type instructions for arithmetic operations, and J-type instructions that are typically used when altering control flow (i.e. jumps). **Fig. 4** illustrates how the bitfields are broken down for each type of instruction.

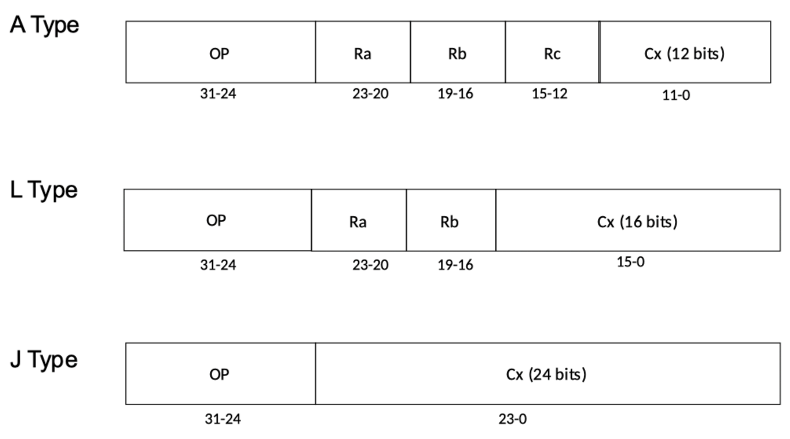


Fig. 4 Cpu0's three instruction formats

Table 5 C, llvm-ir [13] and Cpu0

C	llvm-ir	Cpu0	I or II	Comment
=	load/store	ld/lb/lbu/lh/lhu	I	
&, &&	and	and	I	
|, ||	or	or	I	
^	xor	xor/nor	I	! can be got from two ir
!	%tobool = icmp ne i32 %6, 0 %lnot = xor i1 %tobool, true	cmp xor		
==, !=, <, <=, >, >=	icmp/fcmp <cond> cond:eq/ne,…	cmp/ucmp … + floating-lib	I	
"	"	slt/sltu/slti/sltiu	II	slti/sltiu: ex. a == 3 reduce instructions
if (a <= b)	icmp/fcmp <cond> + br i1 <cond>, …	cmp/uccmp + jeq/jne/jlt/jgt/jle/jge	I	Conditional branch
if (bool)	br i1 <cond>, …	jeq/jne	I	
"	"	beq/bne	II	
goto	br <dest>	jmp	I	Uncondictional branch
call sub-function	call	jsub	I	Provide 24-bit address range of calling sub-function (the address from caller to callee is within 24-bit)
"	"	jalr	I	Add for 32-bit address range of calling sub-function
return	ret	ret	I	
+, -, *	add/fadd, sub/fsub, mul/fmul	add/addu/addiu, sub/subu, mul	I	
/, %	udiv/sdiv/fdiv, urem/srem/frem	div, mfhi/mflo/mthi/mtlo	I	
<<, >>	shl, lshr/ashr	shl/rol/rolv, srl/sra/ror/rorv	II	
float <-> int	fptoui, sitofp, …			Cpu0 uses SW for floating value, and these two IR are for HW floating instruction
__builtin_clz/clo	llvm.clz/llvm_clo	floating-lib + clz, clo	I	For SW floating-lib, uses __builtin_clz / __builtin_clo in clang and clang generates llvm.clz/llvm.clo intrinsic function
__builtin_eh_xxx	llvm.eh.xxx	st/ld	I	pass information to exception handler through $4, $5

Table 6 C++, llvm-ir [13] and Cpu0

C++	llvm-ir	Cpu0	I or II	Comment
try { }	invoke void @_Z15throw_exception	jsub _Z15throw_exception	I	
catch { }	landingpad…catch	st and ld	I	st/ld $4 & $5 to/from stack, $4:exception address, $5: exception typeid

Note

What and how the llvm-ir and the ISA of a RISC CPU be selected

The llvm-ir and the ISA of a RISC CPU emerged after C language. As table above, they can be selected based on C language.

Not listed in above table, LLVM-IR includes terminator instructions "switch, invoke, …", atomic and a lot of llvm-intrinsics to provide better performance to backend for their specific instructions such as llvm.vector.reduce.*.

For vector processing of CPU/GPU, they can use vector-type of math llvm-ir or llvm-intrinsic for implementation.

Note

What and how the ISA of Cpu0 be selected

The intention of orignal author of Cpu0: Design the ISA for teaching materials without considering performance.

My intention of goals: Adding a goal that what ISA is good to be selected or designed considering for both to be an llvm simple tutorial material and basic performance to be an ISA. I am not interested in a bad ISA.

As you can see from table above, "if (a <= b)" can be replaced with "t = (a <= b)" and "if (t)", so I designed the ISA II of Cpu0 "slt+beq" to replace "cmp+jeq" to reduce jeq/jne/jlt/jgt/jle/jge six intructions to two, beq/bne for the balance of the complexity in Cpu0 ISA and performance.

For the same reason, I hired **slt**,… from **Mips** instead of **cmp** from **ARM** as result that destination register can be in any GPR for avoiding the bottle neck on the same "status register".

Floating value can be implemented by software, so Cpu0 has integer instructions only. I add clz and clo to Cpu0 since the floating-lib such as compiler-rt/builtin is implemented on top of these two builtin-function. Normalization for Floating precsion can use clz and clo to speedup. Though Cpu0 can use a couple of instructions for handling the corresponding llvm.clz/llvm.clo, adding clz/clo can execute it in one single instruction.

I extend II of Cpu0 as reasons above for an better ISA in performace from Mips.

The following table details the cpu032I instruction set:

First column F.: meaning Format.

Table 7 cpu032I Instruction Set

F.	Mnemonic	Opcode	Meaning	Syntax	Operation
L	NOP	00	No Operation		
L	LD	01	Load word	LD Ra, [Rb+Cx]	Ra <= [Rb+Cx]
L	ST	02	Store word	ST Ra, [Rb+Cx]	[Rb+Cx] <= Ra
L	LB	03	Load byte	LB Ra, [Rb+Cx]	Ra <= (byte)[Rb+Cx] **[3]**
L	LBu	04	Load byte unsigned	LBu Ra, [Rb+Cx]	Ra <= (byte)[Rb+Cx] **[3]**
L	SB	05	Store byte	SB Ra, [Rb+Cx]	[Rb+Cx] <= (byte)Ra
L	LH	06	Load half word	LH Ra, [Rb+Cx]	Ra <= (2bytes)[Rb+Cx] **[3]**
L	LHu	07	Load half word unsigned	LHu Ra, [Rb+Cx]	Ra <= (2bytes)[Rb+Cx] **[3]**
L	SH	08	Store half word	SH Ra, [Rb+Cx]	[Rb+Cx] <= Ra
L	ADDiu	09	Add immediate	ADDiu Ra, Rb, Cx	Ra <= (Rb + Cx)
L	ANDi	0C	AND imm	ANDi Ra, Rb, Cx	Ra <= (Rb & Cx)
L	ORi	0D	OR	ORi Ra, Rb, Cx	Ra <= (Rb | Cx)
L	XORi	0E	XOR	XORi Ra, Rb, Cx	Ra <= (Rb ^ Cx)
L	LUi	0F	Load upper	LUi Ra, Cx	Ra <= (Cx << 16)
A	ADDu	11	Add unsigned	ADD Ra, Rb, Rc	Ra <= Rb + Rc **[4]**
A	SUBu	12	Sub unsigned	SUB Ra, Rb, Rc	Ra <= Rb - Rc **[4]**

F.	Mnemonic	Opcode	Meaning	Syntax	Operation
A	ADD	13	Add	ADD Ra, Rb, Rc	Ra <= Rb + Rc [4]
A	SUB	14	Subtract	SUB Ra, Rb, Rc	Ra <= Rb - Rc [4]
A	CLZ	15	Count Leading Zero	CLZ Ra, Rb	Ra <= bits of leading zero on Rb
A	CLO	16	Count Leading One	CLO Ra, Rb	Ra <= bits of leading one on Rb
A	MUL	17	Multiply	MUL Ra, Rb, Rc	Ra <= Rb * Rc
A	AND	18	Bitwise and	AND Ra, Rb, Rc	Ra <= Rb & Rc
A	OR	19	Bitwise or	OR Ra, Rb, Rc	Ra <= Rb | Rc
A	XOR	1A	Bitwise exclusive or	XOR Ra, Rb, Rc	Ra <= Rb ^ Rc
A	NOR	1B	Bitwise boolean nor	NOR Ra, Rb, Rc	Ra <= Rb nor Rc
A	ROL	1C	Rotate left	ROL Ra, Rb, Cx	Ra <= Rb rol Cx
A	ROR	1D	Rotate right	ROR Ra, Rb, Cx	Ra <= Rb ror Cx
A	SHL	1E	Shift left	SHL Ra, Rb, Cx	Ra <= Rb << Cx
A	SHR	1F	Shift right	SHR Ra, Rb, Cx	Ra <= Rb >> Cx
A	SRA	20	Shift right	SRA Ra, Rb, Cx	Ra <= Rb ‘>> Cx [6]
A	SRAV	21	Shift right	SRAV Ra, Rb, Rc	Ra <= Rb ‘>> Rc [6]
A	SHLV	22	Shift left	SHLV Ra, Rb, Rc	Ra <= Rb << Rc
A	SHRV	23	Shift right	SHRV Ra, Rb, Rc	Ra <= Rb >> Rc
A	ROL	24	Rotate left	ROL Ra, Rb, Rc	Ra <= Rb rol Rc
A	ROR	25	Rotate right	ROR Ra, Rb, Rc	Ra <= Rb ror Rc
A	CMP	2A	Compare	CMP Ra, Rb	SW <= (Ra cond Rb) [5]
A	CMPu	2B	Compare	CMPu Ra, Rb	SW <= (Ra cond Rb) [5]
J	JEQ	30	Jump if equal (==)	JEQ Cx	if SW(==), PC <= PC + Cx
J	JNE	31	Jump if not equal (!=)	JNE Cx	if SW(!=), PC <= PC + Cx
J	JLT	32	Jump if less than (<)	JLT Cx	if SW(<), PC <= PC + Cx
J	JGT	33	Jump if greater than (>)	JGT Cx	if SW(>), PC <= PC + Cx
J	JLE	34	Jump if less than or equals (<=)	JLE Cx	if SW(<=), PC <= PC + Cx
J	JGE	35	Jump if greater than or equals (>=)	JGE Cx	if SW(>=), PC <= PC + Cx
J	JMP	36	Jump (unconditional)	JMP Cx	PC <= PC + Cx
J	JALR	39	Indirect jump	JALR Rb	LR <= PC; PC <= Rb [7]
J	BAL	3A	Branch and link	BAL Cx	LR <= PC; PC <= PC + Cx
J	JSUB	3B	Jump to subroutine	JSUB Cx	LR <= PC; PC <= PC + Cx
J	JR/RET	3C	Return from subroutine	JR $1 or RET LR	PC <= LR [8]
A	MULT	41	Multiply for 64 bits result	MULT Ra, Rb	(HI,LO) <= MULT(Ra,Rb)
A	MULTU	42	MULT for unsigned 64 bits	MULTU Ra, Rb	(HI,LO) <= MULTU(Ra,Rb)
A	DIV	43	Divide	DIV Ra, Rb	HI<=Ra%Rb, LO<=Ra/Rb
A	DIVU	44	Divide unsigned	DIVU Ra, Rb	HI<=Ra%Rb, LO<=Ra/Rb
A	MFHI	46	Move HI to GPR	MFHI Ra	Ra <= HI
A	MFLO	47	Move LO to GPR	MFLO Ra	Ra <= LO
A	MTHI	48	Move GPR to HI	MTHI Ra	HI <= Ra
A	MTLO	49	Move GPR to LO	MTLO Ra	LO <= Ra
A	MFC0	50	Move C0R to GPR	MFC0 Ra, Rb	Ra <= Rb
A	MTC0	51	Move GPR to C0R	MTC0 Ra, Rb	Ra <= Rb
A	C0MOV	52	Move C0R to C0R	C0MOV Ra, Rb	Ra <= Rb

The following table details the cpu032II instruction set added:

Table 8 cpu032II Instruction Set

F.	Mnemonic	Opcode	Meaning	Syntax	Operation
L	SLTi	26	Set less Then	SLTi Ra, Rb, Cx	Ra <= (Rb < Cx)
L	SLTiu	27	SLTi unsigned	SLTiu Ra, Rb, Cx	Ra <= (Rb < Cx)

F.	Mnemonic	Opcode	Meaning	Syntax	Operation
A	SLT	28	Set less Then	SLT Ra, Rb, Rc	Ra <= (Rb < Rc)
A	SLTu	29	SLT unsigned	SLTu Ra, Rb, Rc	Ra <= (Rb < Rc)
L	BEQ	37	Branch if equal	BEQ Ra, Rb, Cx	if (Ra==Rb), PC <= PC + Cx
L	BNE	38	Branch if not equal	BNE Ra, Rb, Cx	if (Ra!=Rb), PC <= PC + Cx

Note

Cpu0 unsigned instructions

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

Why not using ADD instead of SUB?

From text book of computer introduction, we know SUB can be replaced by ADD as follows,

(A - B) = (A + (-B))

Since Mips uses 32 bits to represent int type of C language, if B is the value of -2G, then

(A - (-2G)) = (A + (2G))

But the problem is value -2G can be represented in 32 bits machine while 2G cannot, since the range of 2's complement representation for 32 bits is (-2G .. 2G-1). The 2's complement reprentation has the merit of fast computation in circuits design, it is widely used in real CPU implementation. That's why almost every CPU create SUB instruction, rather than using ADD instead of.

The Status Register

The Cpu0 status word register (SW) contains the state of the Negative (N), Zero (Z), Carry (C), Overflow (V), Debug (D), Mode (M), and Interrupt (I) flags. The bit layout of the SW register is shown in **Fig. 5** below.

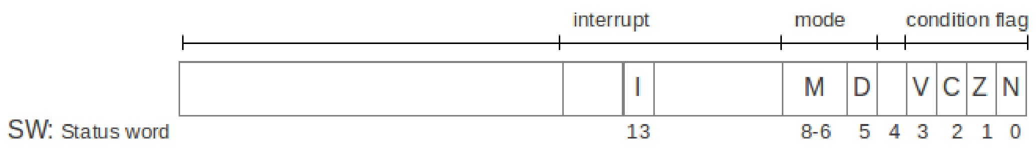


Fig. 5 Cpu0 status word (SW) register

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

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

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

Cpu0's Stages of Instruction Execution

The Cpu0 architecture has a five-stage pipeline. The stages are instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM) and write backe (WB). Here is a description of what happens in the processor for each stage:

1. Instruction fetch (IF)

The Cpu0 fetches the instruction pointed to by the Program Counter (PC) into the Instruction Register (IR): IR = [PC]. The PC is then updated to point to the next instruction: PC = PC + 4.

2. Instruction decode (ID)

    The control unit decodes the instruction stored in IR, which routes necessary data stored in registers to the ALU, and sets the ALU's operation mode based on the current instruction's opcode.

3. Execute (EX)

    The ALU executes the operation designated by the control unit upon data in registers. Except load and store instructions, the result is stored in the destination register after the ALU is done.

4. Memory access (MEM)

    Read data from data cache to pipeline register MEM/WB if it is load instruction; write data from register to data cache if it is strore instruction.

5. Write-back (WB)

    Move data from pipeline register MEM/WB to Register if it is load instruction.

Cpu0's Interrupt Vector

Table 9 Cpu0's Interrupt Vector

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

LLVM Structure

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

Three-phase design

This content and the following sub-section comes from the AOSA chapter on LLVM written by Chris Lattner [10].

The most popular design for a traditional static compiler (like most C compilers) is the three phase design whose major components are the front end, the optimizer and the back end, as seen in Fig. 6. 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.

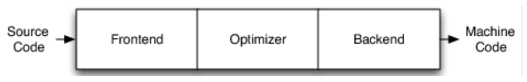


Fig. 6 Three Major Components of a Three Phase Compiler

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

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

The most important win of this classical design comes when a compiler decides to support multiple source languages or target architectures. If the compiler uses a common code representation in its optimizer, then a front end can be written for any language that can compile to it, and a back end can be written for any target that can compile from it, as shown in Fig. 7.

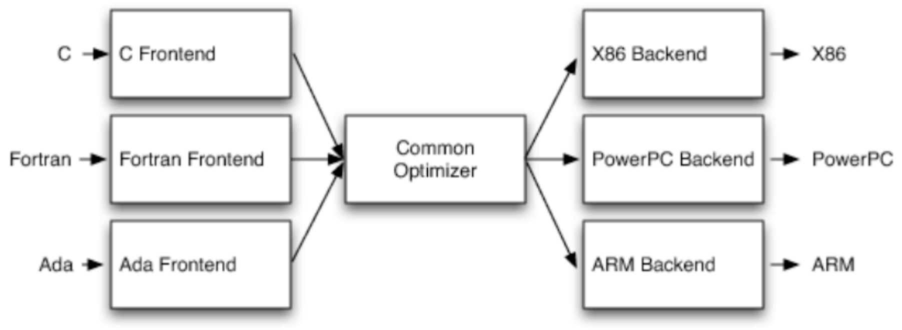


Fig. 7 Retargetablity

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 chapter of a compiler. It was designed with many specific goals in mind, including supporting lightweight runtime optimizations, cross-function/interprocedural optimizations, whole program analysis, and aggressive restructuring transformations, etc. The most important aspect of it, though, is that it is itself defined as a first class language with well-defined semantics. To make this concrete, here is a simple example of a .ll file:

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}
define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse
recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4
done:
  ret i32 %b
}
```

```
// Above 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.

LLVM's Target Description Files: .td

The "mix and match" approach allows target authors to choose what makes sense for their architecture and permits a large amount of code reuse across different targets. This brings up another challenge: each shared component needs to be able to reason about target specific properties in a generic way. For example, a shared register allocator needs to know the register file of each target and the constraints that exist between instructions and their register operands. LLVM's solution to this is for each target to provide a target description in a declarative domain-specific language (a set of .td files) processed by the tblgen tool. The (simplified) build process for the x86 target is shown in **Fig. 8**.

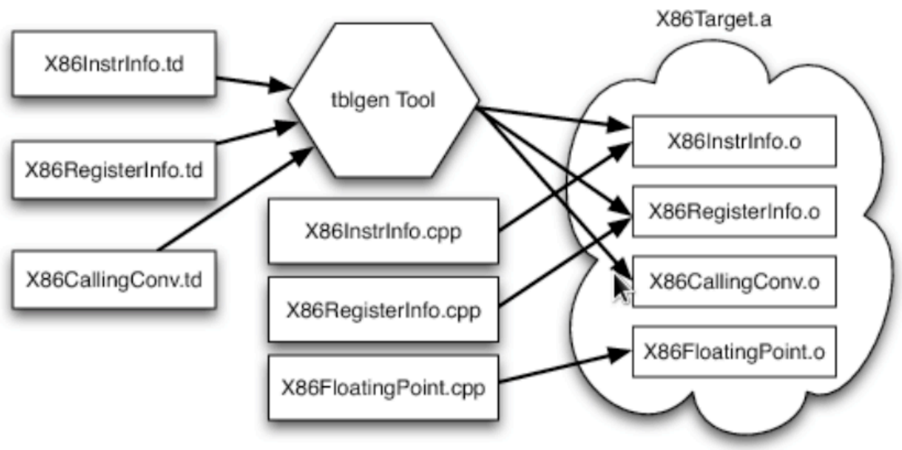


Fig. 8 Simplified x86 Target Definition

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]> { ... }
```

The language used in .td files are Target(Hardware) Description Language that let llvm backend compiler engineers to define the transformation for llvm IR and the machine instructions of their CPUs. In frontend, compiler development tools provide the "Parser Generator" for compiler development; in backend, they provide the "Machine Code Generator" for development, as **Fig. 9** and **Fig. 10**.

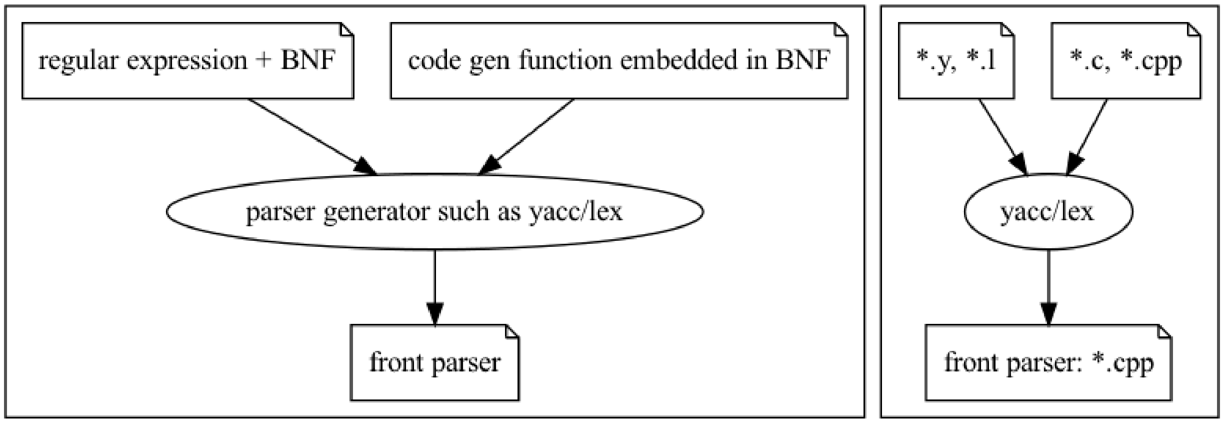


Fig. 9 Frontend TableGen Flow

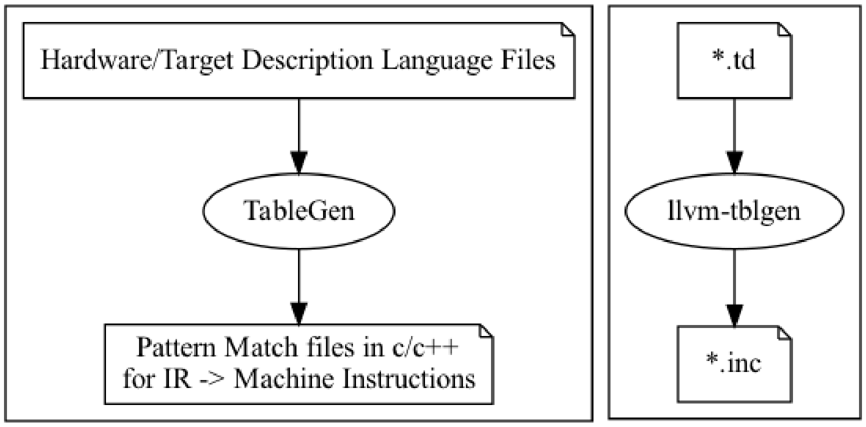

Hardware/Target Description Language Files
TableGen
Pattern Match files in c/c++
for IR -> Machine Instructions
*.td
llvm-tblgen
*.inc

Fig. 10 llvm TableGen Flow

Since the c++'s grammar is more context-sensitive than context-free, llvm frontend project clang uses handcode parser without BNF generator tools. In backend development, the IR to machine instructions transformation can get great benefits from TableGen tools. Though c++ compiler cannot get benefit from BNF generator tools, many computer languages and script languages are more context-free and can get benefit from the tools.

The following come from wiki:

Java syntax has a context-free grammar that can be parsed by a simple LALR parser. Parsing C++ is more complicated [9].

The gnu g++ compiler abandoned BNF tools since version 3.x. I think another reason beyond that c++ has more context-sensitive grammar is handcode parser can provide better error diagnosis than BNF tool since BNF tool always select the rules from BNF grammar if match.

LLVM Code Generation Sequence

Following diagram come from tricore_llvm.pdf.

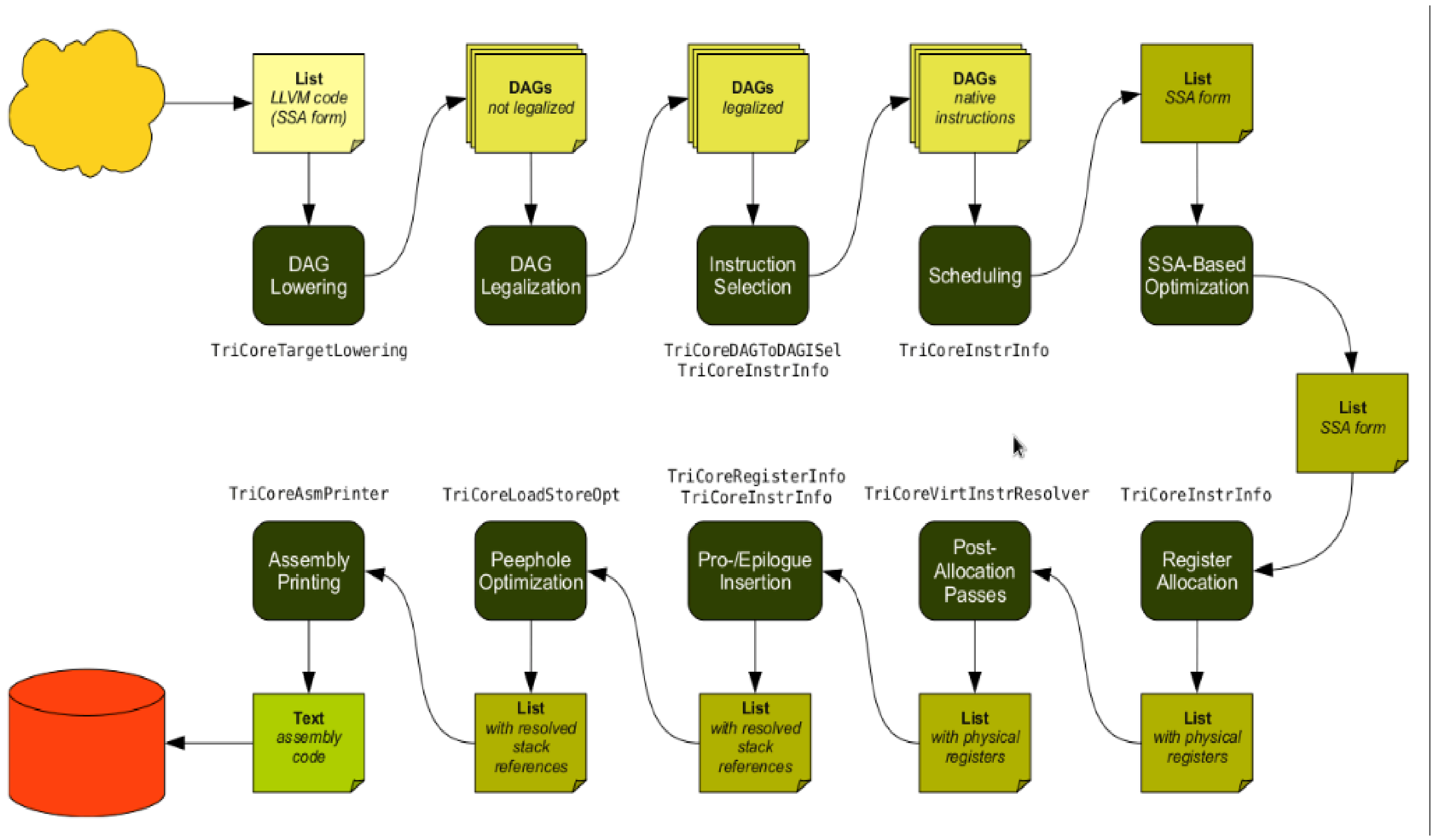


Fig. 11 tricore_llvm.pdf: Code generation sequence. On the path from LLVM code to assembly code, numerous passes are run through and several data structures are used to represent the intermediate results.

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

```
store i32 0, i32* %a  ; store i32 type of 0 to virtual register %a, %a is
            ;  pointer type which point to i32 value
store i32 %b, i32* %c ; store %b contents to %c point to, %b isi32 type virtual
            ;  register, %c is pointer type which point to i32 value.
%a1 = load i32* %a    ; load the memory value where %a point to and assign the
            ;  memory value to %a1
%a3 = add i32 %a2, 1  ; add %a2 and 1 and save to %a3
```

We explain the code generation process as below. If you don’t feel comfortable, please check tricore_llvm.pdf section 4.2 first. You can read “The LLVM Target-Independent Code Generator” from here [12] and “LLVM Language Reference Manual” from here [13] before go ahead, but we think the section 4.2 of tricore_llvm.pdf is enough and suggesting you read the web site documents as above only when you are still not quite understand, even if you have read the articles of this section and next two sections for DAG and Instruction Selection.

1. Instruction Selection

```
// In this stage, transfer the llvm opcode into machine opcode, but the operand
//  still is llvm virtual operand.
    store i16 0, i16* %a // store 0 of i16 type to where virtual register %a
                     //  point to.
=>  st i16 0, i32* %a    // Use Cpu0 backend instruction st instead of IR store.
```

2. Scheduling and Formation

```
// In this stage, reorder the instructions sequence for optimization in
//  instructions cycle or in register pressure.
    st i32 %a, i16* %b,  i16 5 // st %a to *(%b+5)
    st %b, i32* %c, i16 0
    %d = ld i32* %c
```

```
// Transfer above instructions order as follows. In RISC CPU of Mips, the ld
//  %c uses the result of the previous instruction st %c. So it must waits 1
//  cycle. Meaning the ld cannot follow st immediately.
=>  st %b, i32* %c, i16 0
    st i32 %a, i16* %b,  i16 5
    %d = ld i32* %c, i16 0
// If without reorder instructions, a instruction nop which do nothing must be
//  filled, contribute one instruction cycle more than optimization. (Actually,
//  Mips is scheduled with hardware dynamically and will insert nop between st
//  and ld instructions if compiler didn't insert nop.)
    st i32 %a, i16* %b,  i16 5
    st %b, i32* %c, i16 0
    nop
    %d = ld i32* %c, i16 0

// Minimum register pressure
//  Suppose %c is alive after the instructions basic block (meaning %c will be
//  used after the basic block), %a and %b are not alive after that.
// The following no-reorder-version need 3 registers at least
    %a = add i32 1, i32 0
    %b = add i32 2, i32 0
    st %a,  i32* %c, 1
    st %b,  i32* %c, 2

// The reorder version needs 2 registers only (by allocate %a and %b in the same
//  register)
=> %a = add i32 1, i32 0
    st %a,  i32* %c, 1
    %b = add i32 2, i32 0
    st %b,  i32* %c, 2
```

3. SSA-based Machine Code Optimization

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

4. Register Allocation

   Allocate real register for virtual register.

5. Prologue/Epilogue Code Insertion

   Explain in section Add Prologue/Epilogue functions

6. Late Machine Code Optimizations

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

7. Code Emission

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

The llvm code generation sequence also can be obtained by `llc -debug-pass=Structure` as the following. The first 4 code generation sequences from **Fig. 11** are in the **‘DAG->DAG Pattern Instruction Selection’** of the `llc -debug-pass=Structure` displayed. The order of Peephole Optimizations and Prologue/Epilogue Insertion is inconsistent between **Fig. 11** and `llc -debug-pass=Structure` (please check the * in the following). No need to be bothered with this since the the LLVM is under development and changed from time to time.

```
118-165-79-200:input Jonathan$ llc --help-hidden
OVERVIEW: llvm system compiler

USAGE: llc [options] <input bitcode>

OPTIONS:
...
  -debug-pass                             - Print PassManager debugging information
    =None                                 -   disable debug output
    =Arguments                            -   print pass arguments to pass to 'opt'
    =Structure                            -   print pass structure before run()
    =Executions                           -   print pass name before it is executed
    =Details                              -   print pass details when it is executed

118-165-79-200:input Jonathan$ llc -march=mips -debug-pass=Structure ch3.bc
...
Target Library Information
Target Transform Info
Data Layout
Target Pass Configuration
```

```
No Alias Analysis (always returns 'may' alias)
Type-Based Alias Analysis
Basic Alias Analysis (stateless AA impl)
Create Garbage Collector Module Metadata
Machine Module Information
Machine Branch Probability Analysis
  ModulePass Manager
    FunctionPass Manager
      Preliminary module verification
      Dominator Tree Construction
      Module Verifier
      Natural Loop Information
      Loop Pass Manager
        Canonicalize natural loops
      Scalar Evolution Analysis
      Loop Pass Manager
        Canonicalize natural loops
        Induction Variable Users
        Loop Strength Reduction
      Lower Garbage Collection Instructions
      Remove unreachable blocks from the CFG
      Exception handling preparation
      Optimize for code generation
      Insert stack protectors
      Preliminary module verification
      Dominator Tree Construction
      Module Verifier
      Machine Function Analysis
      Natural Loop Information
      Branch Probability Analysis
    * MIPS DAG->DAG Pattern Instruction Selection
      Expand ISel Pseudo-instructions
      Tail Duplication
      Optimize machine instruction PHIs
      MachineDominator Tree Construction
      Slot index numbering
      Merge disjoint stack slots
      Local Stack Slot Allocation
      Remove dead machine instructions
      MachineDominator Tree Construction
      Machine Natural Loop Construction
      Machine Loop Invariant Code Motion
      Machine Common Subexpression Elimination
      Machine code sinking
    * Peephole Optimizations
      Process Implicit Definitions
      Remove unreachable machine basic blocks
      Live Variable Analysis
      Eliminate PHI nodes for register allocation
      Two-Address instruction pass
      Slot index numbering
      Live Interval Analysis
      Debug Variable Analysis
      Simple Register Coalescing
      Live Stack Slot Analysis
      Calculate spill weights
      Virtual Register Map
      Live Register Matrix
      Bundle Machine CFG Edges
      Spill Code Placement Analysis
    * Greedy Register Allocator
      Virtual Register Rewriter
      Stack Slot Coloring
      Machine Loop Invariant Code Motion
    * Prologue/Epilogue Insertion & Frame Finalization
      Control Flow Optimizer
      Tail Duplication
      Machine Copy Propagation Pass
    * Post-RA pseudo instruction expansion pass
      MachineDominator Tree Construction
      Machine Natural Loop Construction
      Post RA top-down list latency scheduler
      Analyze Machine Code For Garbage Collection
      Machine Block Frequency Analysis
      Branch Probability Basic Block Placement
      Mips Delay Slot Filler
      Mips Long Branch
      MachineDominator Tree Construction
      Machine Natural Loop Construction
    * Mips Assembly Printer
      Delete Garbage Collector Information
```

Since Instructions Scheduling and Dead Code Removing will affect Register Allocation. However llvm does not go from later pass onto earlier pass again. The Register Allocation is after Instruction Scheduling. The passes from Live Variable Analysis to Greedy Register Allocator are passes for Register Allocation. About Register Allocation Passes are here [14] [15].

SSA form

SSA form says that each variable is assigned exactly once. One instruction in SSA form can have more than one destination virtual register. LLVM IR is SSA form which has unbounded virtual registers (each variable is assigned exactly once and is keeped in different virtual register). As the result, the optimization steps used in code generation sequence which include stages of **Instruction Selection**, **Scheduling and Formation** and **Register Allocation**, won't loss any optimization opportunity. For example, if using limited virtual registers instead of unlimited as the following code,

```
%a = add nsw i32 1, i32 0
store i32 %a, i32* %c, align 4
%a = add nsw i32 2, i32 0
store i32 %a, i32* %c, align 4
```

Above using limited virtual registers, so virtual register %a used twice. Compiler have to generate the following code since it assigns virtual register %a as output at two different statement.

=> %a = add i32 1, i32 0

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

Above code have to run in sequence. On the other hand, the SSA form as the following can be reodered and run in parallel with the following different version [16].

```
  %a = add nsw i32 1, i32 0
  store i32 %a, i32* %c, align 4
  %b = add nsw i32 2, i32 0
  store i32 %b, i32* %d, align 4

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

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

// version 3
=> %b = add i32 2, i32 0
    st %b,  i32* %d, 0
    %a = add i32 1, i32 0
    st %a,  i32* %c, 0
```

DSA form

```
for (int i = 0; i < 1000; i++) {
  b[i] = f(g(a[i]));
}
```

For the source program as above, the following are the SSA form in source code level and llvm IR level respectively.

```
for (int i = 0; i < 1000; i++) {
  t = g(a[i]);
  b[i] = f(t);
}
```

```
  %pi = alloca i32
  store i32 0, i32* %pi
  %i = load i32, i32* %pi
  %cmp = icmp slt i32 %i, 1000
  br i1 %cmp, label %true, label %end
true:
  %a_idx = add i32 %i, i32 %a_addr
  %val0 = load i32, i32* %a_idx
  %t = call i64 %g(i32 %val0)
  %val1 = call i64 %f(i32 %t)
  %b_idx = add i32 %i, i32 %b_addr
  store i32 %val1, i32* %b_idx
end:
```

The following is the DSA (Dynamic Single Assignment) form.

```
for (int i = 0; i < 1000; i++) {
  t[i] = g(a[i]);
  b[i] = f(t[i]);
}
```

```
  %pi = alloca i32
  store i32 0, i32* %pi
  %i = load i32, i32* %pi
  %cmp = icmp slt i32 %i, 1000
  br i1 %cmp, label %true, label %end
true:
  %a_idx = add i32 %i, i32 %a_addr
  %val0 = load i32, i32* %a_idx
  %t_idx = add i32 %i, i32 %t_addr
  %temp = call i64 %g(i32 %val0)
  store i32 %temp, i32* %t_idx
  %val1 = call i64 %f(i32 %temp)
  %b_idx = add i32 %i, i32 %b_addr
  store i32 %val1, i32* %b_idx
end:
```

In some internet video applications and muti-core (SMP) platforms, splitting g() and f() to two different loop have better perfomance. DSA can split as the following while SSA cannot. Of course, it's possible to do extra analysis on %temp of SSA and reverse it into %t_idx and %t_addr as the following DSA. But in compiler discussion, the translation is from high to low level of machine code. Besides, as you see, the llvm ir lose the for loop information already though it can be reconstructed by extra analysis. So, in this book and almost every paper in compiler discuss with this high-to-low premise, otherwise it's talking about reverse engineering in assembler or compiler.

```
for (int i = 0; i < 1000; i++) {
  t[i] = g(a[i]);
}

for (int i = 0; i < 1000; i++) {
  b[i] = f(t[i]);
}
```

```
  %pi = alloca i32
  store i32 0, i32* %pi
  %i = load i32, i32* %pi
  %cmp = icmp slt i32 %i, 1000
  br i1 %cmp, label %true, label %end
true:
  %a_idx = add i32 %i, i32 %a_addr
  %val0 = load i32, i32* %a_idx
  %t_idx = add i32 %i, i32 %t_addr
  %temp = call i32 %g(i32 %val0)
  store i32 %temp, i32* %t_idx
end:

  %pi = alloca i32
  store i32 0, i32* %pi
  %i = load i32, i32* %pi
  %cmp = icmp slt i32 %i, 1000
  br i1 %cmp, label %true, label %end
true:
  %t_idx = add i32 %i, i32 %t_addr
  %temp = load i32, i32* %t_idx
  %val1 = call i32 %f(i32 %temp)
  %b_idx = add i32 %i, i32 %b_addr
  store i32 %val1, i32* %b_idx
end:
```

Now, the data dependences only exist on t[i] between "t[i] = g(a[i])" and "b[i] = f(t[i])" for each i = (0..999). The program can be run on many different order, and it provides many parallel processing opportunities for multi-core (SMP) and heterogeneous processors. For instance, g(x) is run on GPU and f(x) is run on CPU.

LLVM vs GCC in structure

GCC document is here [17] .

Table 10 clang vs gcc-frontend

frontend	**clang**	**gcc-frontend [18]**
LANGUAGE	C/C++	C/C++
parsing	parsing	parsing
AST	clang-AST	GENERIC [19]
optimization & codgen	clang-backend	gimplifier
IR	LLVM IR	GIMPLE [20]

Table 11 llvm vs gcc (kernal and target/backend)

backend	**llvm**	**gcc**
IR	LLVM IR	GIMPLE
transfer	optimziation & pass	optimization & plugins
DAG	DAG	RTL [21]
codgen	tblgen for td	codgen for md [22]

Both LLVM IR and GIMPLE are SSA form. LLVM IR originally designed to be fully reusable across arbitrary tools besides compiler itself. GCC community never had desire to enable any tools besides compiler (Richard Stallman resisted attempts to make IR more reusable to prevent third-party commercial tools from reusing GCC's frontends). Thus GIMPLE (GCC's IR) was never considered to be more than an implementation detail, in particular it doesn't provide a full description of compiled program (e.g. it lacks program's call graph, type definitions, stack offsets and alias information) [23].

LLVM blog

User uses null pointer to guard code is correct. Undef is only happened in compiler optimization [24]. However when user forget to bind null pointer in guarding code directly or indirectly, compiler such as llvm and gcc may treat null pointer as undef and optimzation out [25].

CFG (Control Flow Graph)

The SSA form can be depicted in CFG and do optimization through the analysis on CFG. Each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block [26].

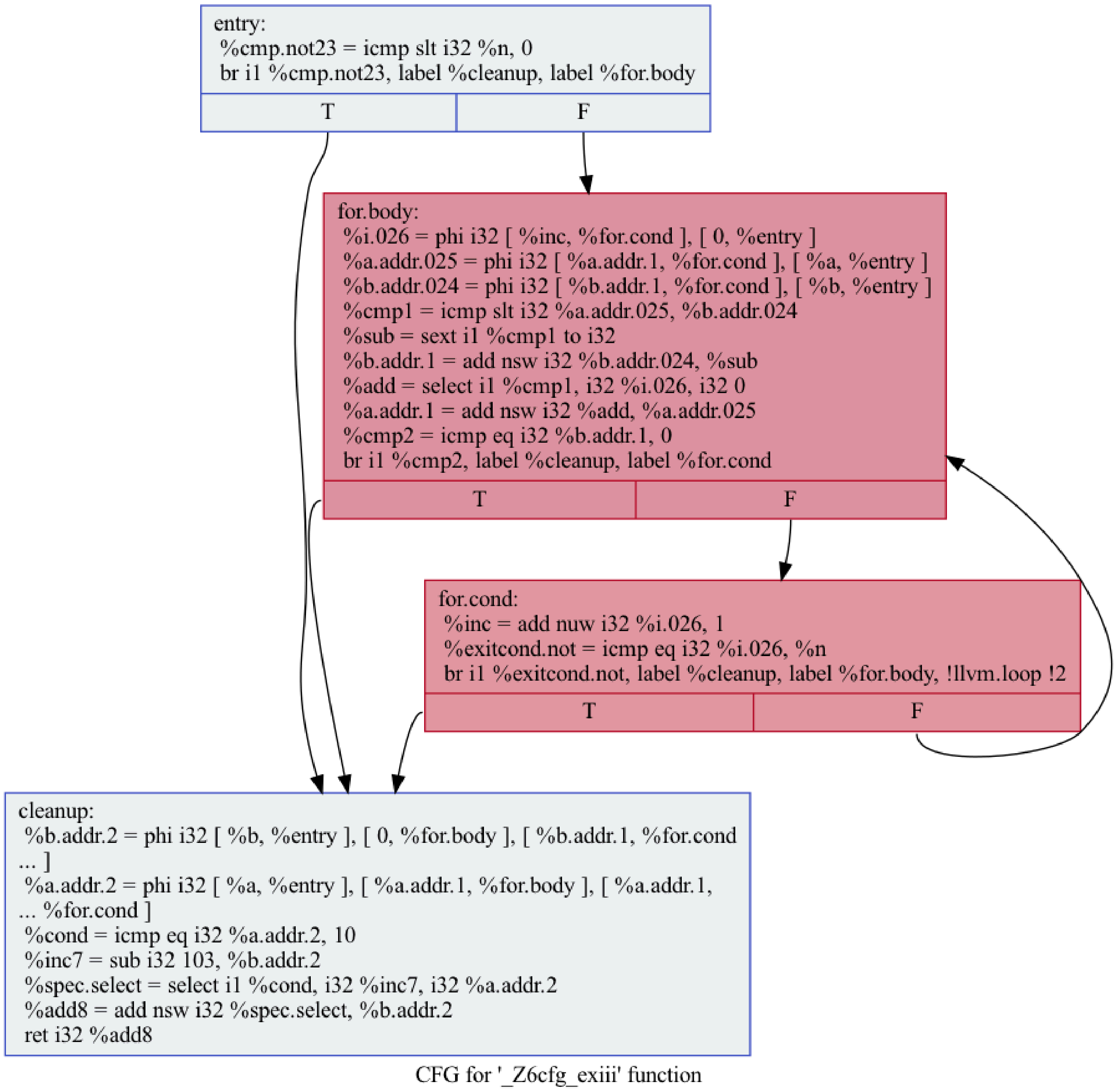
The following is an example of CFG. **The jump/branch always in the last statement of BBs (Basic Blocks)** in Fig. 12.

Fig/llvmstructure/cfg-ex.cpp

```
int cfg_ex(int a, int b, int n)
{
  for (int i = 0; i <= n; i++) {
    if (a < b) {
      a = a + i;
      b = b - 1;
    }
    if (b == 0) {
      goto label_1;
    }
  }

label_1:
  switch (a) {
  case 10:
    a = a*a-b+2;
    a++;
    break;
  }

  return (a+b);
}
```

Fig/llvmstructure/cfg-ex.ll

```
define dso_local i32 @_Z6cfg_exiii(i32 signext %a, i32 signext %b, i32 signext %n) local_unnamed_addr nounwind {
entry:
  %cmp.not23 = icmp slt i32 %n, 0
  br i1 %cmp.not23, label %cleanup, label %for.body

for.cond:                                         ; preds = %for.body
  %inc = add nuw i32 %i.026, 1
  %exitcond.not = icmp eq i32 %i.026, %n
  br i1 %exitcond.not, label %cleanup, label %for.body, !llvm.loop !2

for.body:                                         ; preds = %entry, %for.cond
  %i.026 = phi i32 [ %inc, %for.cond ], [ 0, %entry ]
  %a.addr.025 = phi i32 [ %a.addr.1, %for.cond ], [ %a, %entry ]
  %b.addr.024 = phi i32 [ %b.addr.1, %for.cond ], [ %b, %entry ]
  %cmp1 = icmp slt i32 %a.addr.025, %b.addr.024
  %sub = sext i1 %cmp1 to i32
  %b.addr.1 = add nsw i32 %b.addr.024, %sub
  %add = select i1 %cmp1, i32 %i.026, i32 0
  %a.addr.1 = add nsw i32 %add, %a.addr.025
  %cmp2 = icmp eq i32 %b.addr.1, 0
  br i1 %cmp2, label %cleanup, label %for.cond

cleanup:                                          ; preds = %for.cond, %for.body, %entry
  %b.addr.2 = phi i32 [ %b, %entry ], [ 0, %for.body ], [ %b.addr.1, %for.cond ]
  %a.addr.2 = phi i32 [ %a, %entry ], [ %a.addr.1, %for.body ], [ %a.addr.1, %for.cond ]
  %cond = icmp eq i32 %a.addr.2, 10
  %inc7 = sub i32 103, %b.addr.2
  %spec.select = select i1 %cond, i32 %inc7, i32 %a.addr.2
  %add8 = add nsw i32 %spec.select, %b.addr.2
  ret i32 %add8
}


!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 12.0.1"}
!2 = distinct !{!2, !3}
!3 = !{!"llvm.loop.mustprogress"}
```



```
entry:
 %cmp.not23 = icmp slt i32 %n, 0
 br i1 %cmp.not23, label %cleanup, label %for.body
```

```
for.body:
 %i.026 = phi i32 [ %inc, %for.cond ], [ 0, %entry ]
 %a.addr.025 = phi i32 [ %a.addr.1, %for.cond ], [ %a, %entry ]
 %b.addr.024 = phi i32 [ %b.addr.1, %for.cond ], [ %b, %entry ]
 %cmp1 = icmp slt i32 %a.addr.025, %b.addr.024
 %sub = sext i1 %cmp1 to i32
 %b.addr.1 = add nsw i32 %b.addr.024, %sub
 %add = select i1 %cmp1, i32 %i.026, i32 0
 %a.addr.1 = add nsw i32 %add, %a.addr.025
 %cmp2 = icmp eq i32 %b.addr.1, 0
 br i1 %cmp2, label %cleanup, label %for.cond
```

```
for.cond:
 %inc = add nuw i32 %i.026, 1
 %exitcond.not = icmp eq i32 %i.026, %n
 br i1 %exitcond.not, label %cleanup, label %for.body, !llvm.loop !2
```

```
cleanup:
 %b.addr.2 = phi i32 [ %b, %entry ], [ 0, %for.body ], [ %b.addr.1, %for.cond
... ]
 %a.addr.2 = phi i32 [ %a, %entry ], [ %a.addr.1, %for.body ], [ %a.addr.1,
... %for.cond ]
 %cond = icmp eq i32 %a.addr.2, 10
 %inc7 = sub i32 103, %b.addr.2
 %spec.select = select i1 %cond, i32 %inc7, i32 %a.addr.2
 %add8 = add nsw i32 %spec.select, %b.addr.2
 ret i32 %add8
```

Fig. 12 CFG for cfg-ex.ll

DAG (Directed Acyclic Graph)

The SSA in each BB (Basic Block) from CFG as the previous section can be represented in DAG.

Many important techniques for local optimization begin by transforming a basic block into DAG **[27]**. For example, the basic block code and it's corresponding DAG as **Fig. 13**.

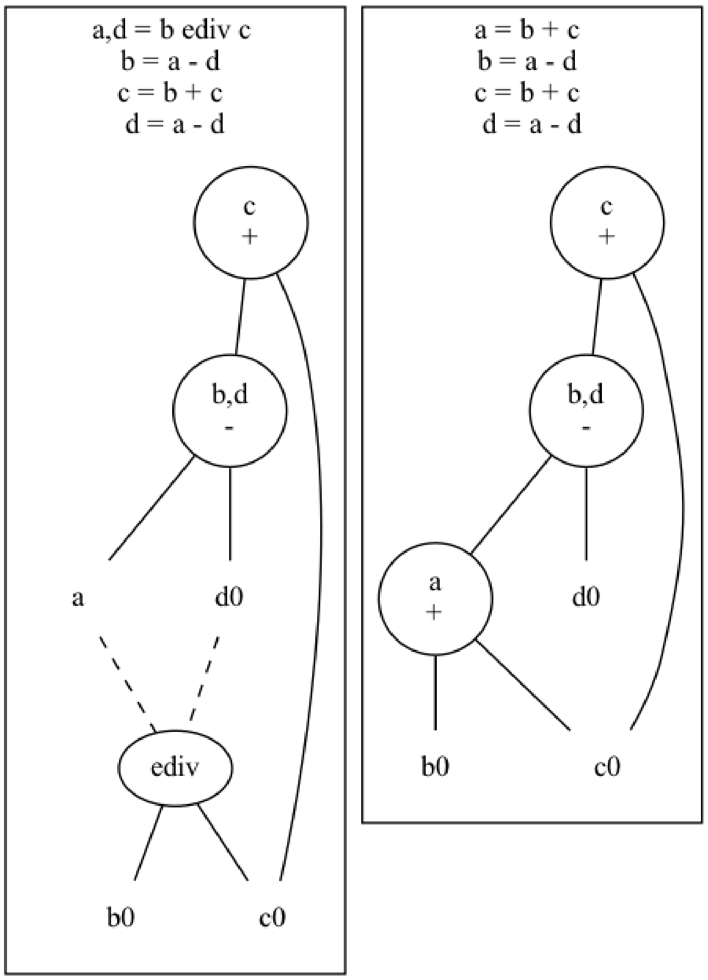


Fig. 13 The left includes two destination register of DAG example and the right has one destination only

DAG and SSA are allowed for two destination virtual registers. Assume ediv operation provides divide on interger which save quotient in “a” and remainder in “d”. For one destination register, DAG may simplify as the right of **Fig. 13**.

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

Table 12 common expression remove process

Replace node b with node d	Replace b_0, c_0, d_0 with b, c, d
a = b_0 + c_0	a = b + c
d = a – d_0	d = a – d
c = d + c	c = d + c

After removing b and traversing the DAGs from bottom to top (traverse binary tree by Depth-first In-order search) , the first column of above table will be gotten.

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

DAG is 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.

In addition to DAG optimization, the “kill” register has also mentioned in section 8.5.5 of the compiler book [27]. This optimization method also applied in llvm implementation.

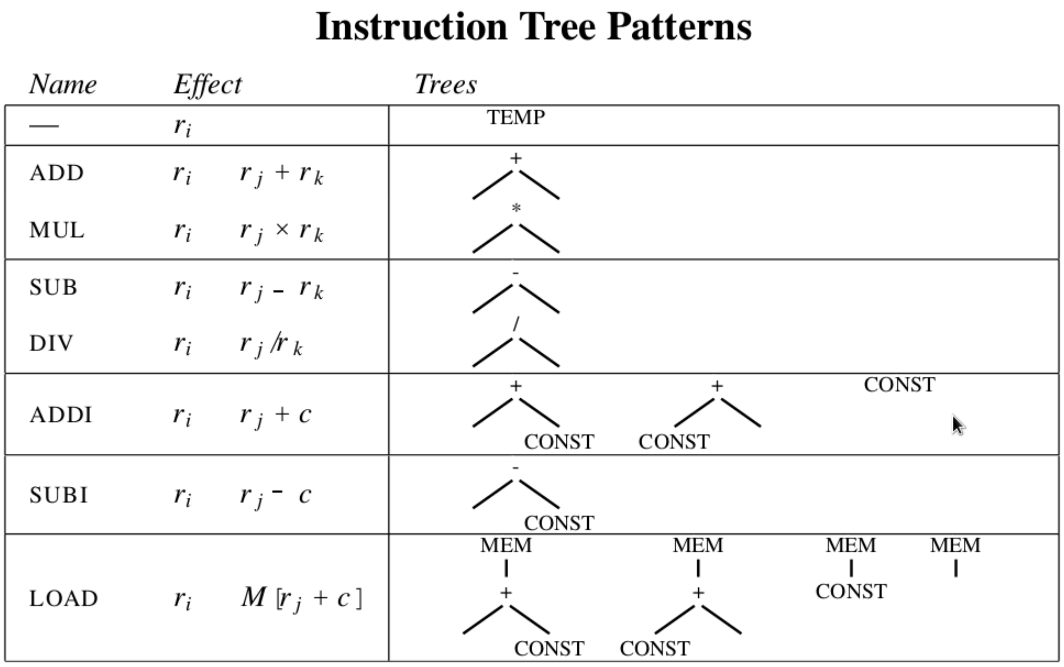
Instruction Selection

The major function of backend is that translate IR code into machine code at stage of Instruction Selection as **Fig. 14**.

$$\begin{array}{lll|ll}
\mathtt{MOV} & r_d = r_s & & \mathtt{ADDI} & r_d = r_s + 0 \\
\mathtt{MOV} & r_d = r_s & & \mathtt{ADD} & r_d = r_{s1} + r_0 \\
\mathtt{MOVI} & r_d = c & & \mathtt{ADDI} & r_d = r_0 + c
\end{array}$$

Fig. 14 IR and it’s corresponding machine instruction

For machine instruction selection, the best solution is representing IR and machine instruction by DAG. To simplify in view, the register leaf is skipped in **Fig. 15**. The $r_j + r_k$ is IR DAG representation (for symbol notation, not llvm SSA form). ADD is machine instruction.



Instruction Tree Patterns

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

Fig. 15 Instruction DAG representation

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

Now, let’s check the ADDiu instruction defined in Cpu0InstrInfo.td as follows,

lbdex/chapters/Chapter2/Cpu0InstrFormats.td

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

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

  let Opcode = op;

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

lbdex/chapters/Chapter2/Cpu0InstrInfo.td

```
// Node immediate fits as 16-bit sign extended on target immediate.
// e.g. addi, andi
def immSExt16  : PatLeaf<(imm), [{ return isInt<16>(N->getSExtValue()); }]>;
```

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

```
// IR "add" defined in include/llvm/Target/TargetSelectionDAG.td, line 315 (def add).
def ADDiu   : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;
```

Fig. 16 shows how the pattern match work in the IR node, **add**, and instruction node, **ADDiu**, which both defined in Cpu0InstrInfo.td. In this example, IR node “add %a, 5” will be translated to “addiu $r1, 5” after %a is allcated to register $r1 in regiter allocation stage since the IR pattern[(set RC:$ra, (OpNode RC:$rb, imm_type:$imm16))] is set in ADDiu and the 2nd operand is “signed immediate” which matched “%a, 5”. In addition to pattern match, the .td also set assembly string “addiu” and op code 0x09. With this information, the LLVM TableGen will generate instruction both in assembly and binary automatically (the binary instruction can be issued in obj file of ELF format which will be explained at later chapter). Similarly, the machine instruction DAG nodes LD and ST can be translated from IR DAG nodes **load** and **store**. Notice that the $rb in **Fig. 16** is virtual register name (not machine register). The detail for **Fig. 16** is depicted as **Fig. 17**.

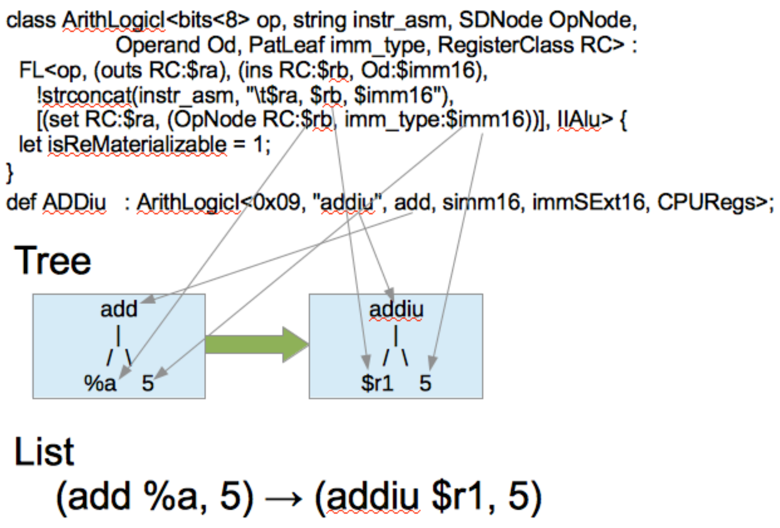

class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode,
Operand Od, PatLeaf imm_type, RegisterClass RC> :
FL<op, (outs RC:$ra), (ins RC:$rb, Od:$imm16),
!strconcat(instr_asm, "\t$ra, $rb, $imm16"),
[(set RC:$ra, (OpNode RC:$rb, imm_type:$imm16))], IIAlu> {
let isReMaterializable = 1;
}
def ADDiu   : ArithLogicI<0x09, "addiu", add, simm16, immSExt16, CPURegs>;
Tree
add
%a
5
addiu
$r1
5
List
(add %a, 5) → (addiu $r1, 5)

Fig. 16 Pattern match for ADDiu instruction and IR node add

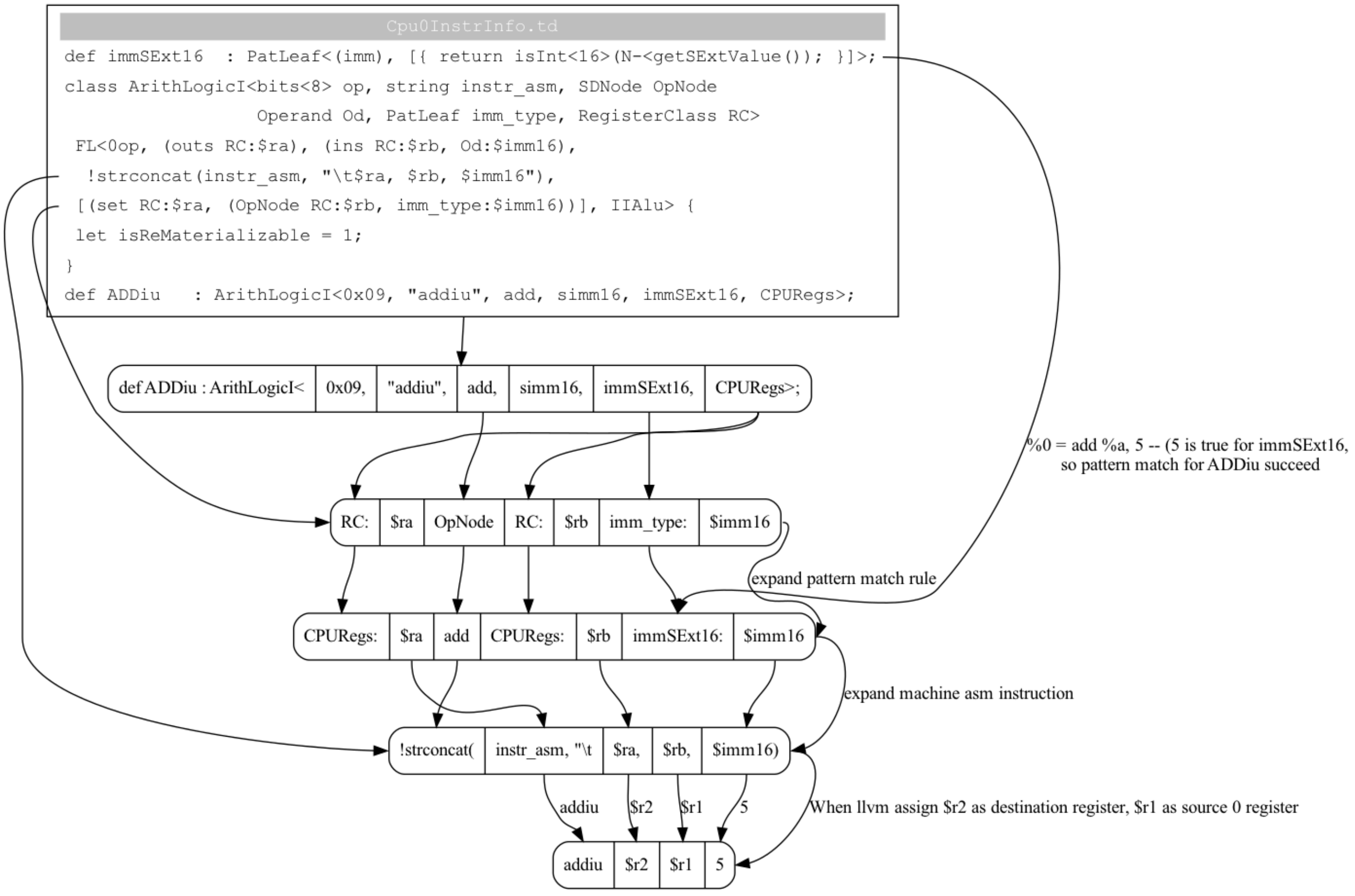

```
Cpu0InstrInfo.td
def immSExt16  : PatLeaf<(imm), [{ return isInt<16>(N-<getSExtValue()); }]>;
class ArithLogicI<bits<8> op, string instr_asm, SDNode OpNode
                  Operand Od, PatLeaf imm_type, RegisterClass RC>
 FL<0op, (outs RC:$ra), (ins RC:$rb, Od:$imm16),
  !strconcat(instr_asm, "\t$ra, $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>;
RC: | $ra | OpNode | RC: | $rb | imm_type: | $imm16
CPURegs: | $ra | add | CPURegs: | $rb | immSExt16: | $imm16
!strconcat(| instr_asm, "\t | $ra, | $rb, | $imm16)
addiu | $r2 | $r1 | 5
%0 = add %a, 5 -- (5 is true for immSExt16, so pattern match for ADDiu succeed
expand pattern match rule
expand machine asm instruction
addiu
$r2
$r1
5
When llvm assign $r2 as destination register, $r1 as source 0 register

Fig. 17 Pattern match for ADDiu instruction and IR node add in detail

From DAG instruction selection we mentioned, the leaf node must be a Data Node. ADDiu is format L type which the last operand must fits in 16 bits range. So, Cpu0InstrInfo.td define a PatLeaf type of immSExt16 to let llvm system know the PatLeaf range. If the imm16 value is out of this range, **"isInt<16>(N->getSExtValue())"** will return false and this pattern won't use ADDiu in instruction selection stage.

Some cpu/fpu (floating point processor) has multiply-and-add floating point instruction, fmadd. It can be represented by DAG list (fadd (fmul ra, rc), rb). For this implementation, we can assign fmadd DAG pattern to instruction td as follows,

```
def FMADDS : AForm_1<59, 29,
          (ops F4RC:$FRT, F4RC:$FRA, F4RC:$FRC, F4RC:$FRB),
          "fmadds $FRT, $FRA, $FRC, $FRB",
          [(set F4RC:$FRT, (fadd (fmul F4RC:$FRA, F4RC:$FRC),
                       F4RC:$FRB))]>;
```

Similar with ADDiu, [(set F4RC:$FRT, (fadd (fmul F4RC:$FRA, F4RC:$FRC), F4RC:$FRB))] is the pattern which include nodes **fmul** and **fadd**.

Now, for the following basic block notation IR and llvm SSA IR code,

```
d = a * c
e = d + b
...
```

```
%d = fmul %a, %c
%e = fadd %d, %b
...
```

the Instruction Selection Process will translate this two IR DAG node (fmul %a, %c) (fadd %d, %b) into one machine instruction DAG node (**fmadd** %a, %c, %b), rather than translate them into two machine instruction nodes **fmul** and **fadd** if the FMADDS is appear before FMUL and FADD in your td file.

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

As you can see, the IR notation representation is easier to read than llvm SSA IR form. So, this notation form is used in this book sometimes.

For the following basic block code,

```
a = b + c   // in notation IR form
d = a – d
%e = fmadd %a, %c, %b // in llvm SSA IR form
```

We can apply **Fig. 8** Instruction Tree Patterns to get the following machine code,

```
load  rb, M(sp+8); // assume b allocate in sp+8, sp is stack point register
load  rc, M(sp+16);
add ra, rb, rc;
load  rd, M(sp+24);
sub rd, ra, rd;
fmadd re, ra, rc, rb;
```

Caller and callee saved registers

lbdex/input/ch9_caller_callee_save_registers.cpp

```
extern int add1(int x);

int callee()
{
  int t1 = 3;
  int result = add1(t1);
  result = result - t1;

  return result;
}
```

Run Mips backend with above input will get the following result.

```
JonathantekiiMac:input Jonathan$ ~/llvm/debug/build/bin/llc
-O0 -march=mips -relocation-model=static -filetype=asm
ch9_caller_callee_save_registers.bc -o -
      .text
      .abicalls
      .option pic0
      .section        .mdebug.abi32,"",@progbits
      .nan    legacy
      .file   "ch9_caller_callee_save_registers.bc"
      .text
      .globl  _Z6calleev
      .align  2
      .type   _Z6calleev,@function
      .set    nomicromips
      .set    nomips16
      .ent    _Z6calleev
_Z6callerv:                             # @_Z6callerv
      .cfi_startproc
      .frame  $fp,32,$ra
      .mask   0xc0000000,-4
      .fmask  0x00000000,0
      .set    noreorder
      .set    nomacro
      .set    noat
# BB#0:
      addiu   $sp, $sp, -32
$tmp0:
      .cfi_def_cfa_offset 32
      sw      $ra, 28($sp)            # 4-byte Folded Spill
      sw      $fp, 24($sp)            # 4-byte Folded Spill
$tmp1:
      .cfi_offset 31, -4
$tmp2:
      .cfi_offset 30, -8
      move     $fp, $sp
$tmp3:
      .cfi_def_cfa_register 30
      addiu   $1, $zero, 3
      sw      $1, 20($fp)   # store t1 to 20($fp)
      move     $4, $1
      jal     _Z4add1i
      nop
      sw      $2, 16($fp)   # $2 : the return vaule for fuction add1()
      lw      $1, 20($fp)   # load t1 from 20($fp)
      subu    $1, $2, $1
      sw      $1, 16($fp)
      move     $2, $1     # move result to return register $2
      move     $sp, $fp
      lw      $fp, 24($sp)            # 4-byte Folded Reload
      lw      $ra, 28($sp)            # 4-byte Folded Reload
      addiu   $sp, $sp, 32
      jr      $ra
      nop
      .set    at
      .set    macro
      .set    reorder
      .end    _Z6calleev
$func_end0:
      .size   _Z6calleev, ($func_end0)-_Z6calleev
      .cfi_endproc
```

Caller and callee saved registers definition as follows,

> If the caller wants to use caller-saved registers after callee function, it must save caller-saved registers' content to memory for using and restore these registers from memory after function call.
> If the callee wants to use callee-saved registers, it must save its content to memory before using them and restore these registers from memory before return.

As above definition, if a register is not a callee-saved-registers, then it must be caller-saved-registers because the callee doesn't retore it and the value is changed after callee function. So, Mips only define the callee-saved registers in MipsCallingConv.td, and can be found in CSR_O32_SaveList of MipsGenRgisterInfo.inc for the default ABI.

As above assembly output, Mips allocates t1 variable to register $1 and no need to spill $1 since $1 is caller saved register. On the other hand, $ra is callee saved register, so it spills at beginning of the assembly output since jal uses $ra register. Cpu0 $lr is the same register as Mips $ra, so it calls setAliasRegs(MF, SavedRegs, Cpu0::LR) in determineCalleeSaves() of Cpu0SEFrameLowering.cpp when the function has called another function.

Live in and live out register

As the example of last sub-section. The $ra is "live in" register since the return address is decided by caller. The $2 is "live out" register since the return value of the function is saved in this register, and caller can get the result by read it directly as the comment in above example. Through mark "live in" and "live out" registers, backend provides llvm middle layer information to remove useless instructions in variables access. Of course, llvm applies the DAG analysis mentioned in the previous sub-section to finish it. Since C supports seperate compilation for different functions, the "live in" and "out" information from backend provides the optimization opportunity to llvm. LLVM provides function addLiveIn() to mark "live in" register but no function addLiveOut() provided. For the "live out" register, Mips backend marks it by DAG=DAG.getCopyToReg(…, $2, …) and return DAG instead, since all local varaiables are not exist after function exit.

Create Cpu0 backend

From now on, the Cpu0 backend will be created from scratch step by step. To make readers easily understanding the backend structure, Cpu0 example code can be generated chapter by chapter through command here [11]. Cpu0 example code, lbdex, can be found at near left bottom of this web site. Or here **http://jonathan2251.github.io/lbd/lbdex.tar.gz**.

Cpu0 backend machine ID and relocation records

To create a new backend, there are some files in <<llvm root dir>> need to be modified. The added information include both the ID and name of machine, and relocation records. Chapter "ELF Support" include the relocation records introduction. The following files are modified to add Cpu0 backend as follows,

lbdex/llvm/modify/llvm/config-ix.cmake

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

lbdex/llvm/modify/llvm/CMakeLists.txt

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

lbdex/llvm/modify/llvm/include/llvm/ADT/Triple.h

```
...
#undef mips
#undef cpu0
...
class Triple {
public:
  enum ArchType {
    ...
    cpu0,       // For Tutorial Backend Cpu0
    cpu0el,
    ...
  };
  ...
}
```

lbdex/llvm/modify/llvm/include/llvm/Object/ELFObjectFile.h

```
...
template <class ELFT>
StringRef ELFObjectFile<ELFT>::getFileFormatName() const {
  switch (EF.getHeader()->e_ident[ELF::EI_CLASS]) {
  case ELF::ELFCLASS32:
    switch (EF.getHeader()->e_machine) {
    ...
    case ELF::EM_CPU0:        // llvm-objdump -t -r
      return "ELF32-cpu0";
    ...
  }
  ...
}
...
template <class ELFT>
unsigned ELFObjectFile<ELFT>::getArch() const {
  bool IsLittleEndian = ELFT::TargetEndianness == support::little;
  switch (EF.getHeader()->e_machine) {
  ...
  case ELF::EM_CPU0:  // llvm-objdump -t -r
    switch (EF.getHeader()->e_ident[ELF::EI_CLASS]) {
    case ELF::ELFCLASS32:
    return IsLittleEndian ? Triple::cpu0el : Triple::cpu0;
    default:
      report_fatal_error("Invalid ELFCLASS!");
    }
  ...
  }
}
```

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

```
enum {
  ...
  EM_CPU0          = 999  // Document LLVM Backend Tutorial Cpu0
};
...
// Cpu0 Specific e_flags
enum {
  EF_CPU0_NOREORDER = 0x00000001, // Don't reorder instructions
  EF_CPU0_PIC       = 0x00000002, // Position independent code
  EF_CPU0_ARCH_32   = 0x50000000, // CPU032 instruction set per linux not elf.h
  EF_CPU0_ARCH      = 0xf0000000  // Mask for applying EF_CPU0_ARCH_ variant
};

// ELF Relocation types for Mips
enum {
#include "ELFRelocs/Cpu0.def"
};
...
```

lbdex/llvm/modify/llvm/lib/MC/MCSubtargetInfo.cpp

```
bool Cpu0DisableUnreconginizedMessage = false;

void MCSubtargetInfo::InitMCProcessorInfo(StringRef CPU, StringRef FS) {
  #if 1 // Disable reconginized processor message. For Cpu0
  if (TargetTriple.getArch() == llvm::Triple::cpu0 ||
      TargetTriple.getArch() == llvm::Triple::cpu0el)
    Cpu0DisableUnreconginizedMessage = true;
  #endif
  ...
}
...
const MCSchedModel &MCSubtargetInfo::getSchedModelForCPU(StringRef CPU) const {
  ...
    #if 1 // Disable reconginized processor message. For Cpu0
    if (TargetTriple.getArch() != llvm::Triple::cpu0 &&
        TargetTriple.getArch() != llvm::Triple::cpu0el)
    #endif
  ...
}
```

lbdex/llvm/modify/llvm/lib/MC/SubtargetFeature.cpp

```
extern bool Cpu0DisableUnreconginizedMessage; // For Cpu0
...
FeatureBitset
SubtargetFeatures::ToggleFeature(FeatureBitset Bits, StringRef Feature,
                                 ArrayRef<SubtargetFeatureKV> FeatureTable) {
  ...
    if (!Cpu0DisableUnreconginizedMessage) // For Cpu0
  ...
}

FeatureBitset
SubtargetFeatures::ApplyFeatureFlag(FeatureBitset Bits, StringRef Feature,
                                    ArrayRef<SubtargetFeatureKV> FeatureTable) {
  ...
    if (!Cpu0DisableUnreconginizedMessage) // For Cpu0
  ...
}

FeatureBitset
SubtargetFeatures::getFeatureBits(StringRef CPU,
                                  ArrayRef<SubtargetFeatureKV> CPUTable,
                                  ArrayRef<SubtargetFeatureKV> FeatureTable) {
  ...
    if (!Cpu0DisableUnreconginizedMessage) // For Cpu0
  ...
}
```

lib/object/ELF.cpp

```
...

StringRef getELFRelocationTypeName(uint32_t Machine, uint32_t Type) {
  switch (Machine) {
  ...
  case ELF::EM_CPU0:
    switch (Type) {
#include "llvm/Support/ELFRelocs/Cpu0.def"
    default:
      break;
    }
    break;
  ...
  }
```

include/llvm/Support/ELFRelocs/Cpu0.def

```
#ifndef ELF_RELOC
#error "ELF_RELOC must be defined"
#endif

ELF_RELOC(R_CPU0_NONE,                0)
ELF_RELOC(R_CPU0_32,                  2)
ELF_RELOC(R_CPU0_HI16,                5)
ELF_RELOC(R_CPU0_LO16,                6)
ELF_RELOC(R_CPU0_GPREL16,             7)
ELF_RELOC(R_CPU0_LITERAL,             8)
ELF_RELOC(R_CPU0_GOT16,               9)
ELF_RELOC(R_CPU0_PC16,               10)
ELF_RELOC(R_CPU0_CALL16,             11)
ELF_RELOC(R_CPU0_GPREL32,            12)
ELF_RELOC(R_CPU0_PC24,               13)
ELF_RELOC(R_CPU0_GOT_HI16,           22)
ELF_RELOC(R_CPU0_GOT_LO16,           23)
ELF_RELOC(R_CPU0_RELGOT,             36)
ELF_RELOC(R_CPU0_TLS_GD,             42)
ELF_RELOC(R_CPU0_TLS_LDM,            43)
ELF_RELOC(R_CPU0_TLS_DTP_HI16,       44)
ELF_RELOC(R_CPU0_TLS_DTP_LO16,       45)
ELF_RELOC(R_CPU0_TLS_GOTTPREL,       46)
ELF_RELOC(R_CPU0_TLS_TPREL32,        47)
ELF_RELOC(R_CPU0_TLS_TP_HI16,        49)
ELF_RELOC(R_CPU0_TLS_TP_LO16,        50)
ELF_RELOC(R_CPU0_GLOB_DAT,           51)
ELF_RELOC(R_CPU0_JUMP_SLOT,          127)
```

lbdex/llvm/modify/llvm/lib/Support/Triple.cpp

```
const char *Triple::getArchTypeName(ArchType Kind) {
  switch (Kind) {
  ...
  case cpu0:        return "cpu0";
  case cpu0el:      return "cpu0el";
  ...
  }
}
...
const char *Triple::getArchTypePrefix(ArchType Kind) {
  switch (Kind) {
  ...
  case cpu0:
  case cpu0el:      return "cpu0";
  ...
}
...
Triple::ArchType Triple::getArchTypeForLLVMName(StringRef Name) {
  return StringSwitch<Triple::ArchType>(Name)
    ...
    .Case("cpu0", cpu0)
    .Case("cpu0el", cpu0el)
    ...
}
...
static Triple::ArchType parseArch(StringRef ArchName) {
  return StringSwitch<Triple::ArchType>(ArchName)
    ...
    .Cases("cpu0", "cpu0eb", "cpu0allegrex", Triple::cpu0)
    .Cases("cpu0el", "cpu0allegrexel", Triple::cpu0el)
    ...
}
...
static Triple::ObjectFormatType getDefaultFormat(const Triple &T) {
  ...
  case Triple::cpu0:
  case Triple::cpu0el:
  ...
}
...
static unsigned getArchPointerBitWidth(llvm::Triple::ArchType Arch) {
  switch (Arch) {
```

```
  ...
  case llvm::Triple::cpu0:
  case llvm::Triple::cpu0el:
  ...
    return 32;
  }
}
...
Triple Triple::get32BitArchVariant() const {
  Triple T(*this);
  switch (getArch()) {
  ...
  case Triple::cpu0:
  case Triple::cpu0el:
  ...
    // Already 32-bit.
    break;
  }
  return T;
}
```

Creating the Initial Cpu0 .td Files

As it has been discussed in the previous section, LLVM uses target description files (which uses the .td file extension) to describe various components of a target's backend. For example, these .td files may describe a target's register set, instruction set, scheduling information for instructions, and calling conventions. When your backend is being compiled, the tablegen tool that ships with LLVM will translate these .td files into C++ source code written to files that have a .inc extension. Please refer to [32] for more information regarding how to use tablegen.

Every backend has its own .td to define some target information. These files have a similar syntax to C++. For Cpu0, the target description file is called Cpu0Other.td, which is shown below:

lbdex/chapters/Chapter2/Cpu0.td

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

// Without this will have error: 'cpu032I' is not a recognized processor for
//  this target (ignoring processor)
//===----------------------------------------------------------------------===//
// Cpu0 Subtarget features                                                    //
//===----------------------------------------------------------------------===//

def FeatureChapter3_1  : SubtargetFeature<"ch3_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter3_2  : SubtargetFeature<"ch3_2", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter3_3  : SubtargetFeature<"ch3_3", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter3_4  : SubtargetFeature<"ch3_4", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter3_5  : SubtargetFeature<"ch3_5", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter4_1  : SubtargetFeature<"ch4_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter4_2  : SubtargetFeature<"ch4_2", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter5_1  : SubtargetFeature<"ch5_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter6_1  : SubtargetFeature<"ch6_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter7_1  : SubtargetFeature<"ch7_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter8_1  : SubtargetFeature<"ch8_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter8_2  : SubtargetFeature<"ch8_2", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter9_1  : SubtargetFeature<"ch9_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter9_2  : SubtargetFeature<"ch9_2", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter9_3  : SubtargetFeature<"ch9_3", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter10_1 : SubtargetFeature<"ch10_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter11_1 : SubtargetFeature<"ch11_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter11_2 : SubtargetFeature<"ch11_2", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapter12_1 : SubtargetFeature<"ch12_1", "HasChapterDummy", "true",
                                "Enable Chapter instructions.">;
def FeatureChapterAll  : SubtargetFeature<"chall", "HasChapterDummy", "true",
                                "Enable Chapter instructions.",
                                [FeatureChapter3_1, FeatureChapter3_2,
                                 FeatureChapter3_3, FeatureChapter3_4,
                                 FeatureChapter3_5,
                                 FeatureChapter4_1, FeatureChapter4_2,
                                 FeatureChapter5_1, FeatureChapter6_1,
                                 FeatureChapter7_1, FeatureChapter8_1,
                                 FeatureChapter8_2, FeatureChapter9_1,
                                 FeatureChapter9_2, FeatureChapter9_3,
                                 FeatureChapter10_1,
                                 FeatureChapter11_1, FeatureChapter11_2,
                                 FeatureChapter12_1]>;

def FeatureCmp         : SubtargetFeature<"cmp", "HasCmp", "true",
                                "Enable 'cmp' instructions.">;
def FeatureSlt         : SubtargetFeature<"slt", "HasSlt", "true",
                                "Enable 'slt' instructions.">;
def FeatureCpu032I     : SubtargetFeature<"cpu032I", "Cpu0ArchVersion",
                                "Cpu032I", "Cpu032I ISA Support",
                                [FeatureCmp, FeatureChapterAll]>;
def FeatureCpu032II    : SubtargetFeature<"cpu032II", "Cpu0ArchVersion",
                               "Cpu032II", "Cpu032II ISA Support (slt)",
                                [FeatureCmp, FeatureSlt, FeatureChapterAll]>;

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

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

def Cpu0InstrInfo : InstrInfo;
//===----------------------------------------------------------------------===//
// Cpu0 processors supported.
//===----------------------------------------------------------------------===//

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

def : Proc<"cpu032I",  [FeatureCpu032I]>;
def : Proc<"cpu032II", [FeatureCpu032II]>;
// Above make Cpu0GenSubtargetInfo.inc set feature bit as the following order
// enum {
//   FeatureCmp =  1ULL << 0,
//   FeatureCpu032I =  1ULL << 1,
//   FeatureCpu032II =  1ULL << 2,
//   FeatureSlt =  1ULL << 3
// };

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

lbdex/chapters/Chapter2/Cpu0Other.td

```
//===-- Cpu0Other.td - Describe the Cpu0 Target Machine ----*- tablegen -*-===//
//
//                       The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
// This is the top level entry point for the Cpu0 target.
//===----------------------------------------------------------------------===//
```

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

include "llvm/Target/Target.td"

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

include "Cpu0RegisterInfo.td"
include "Cpu0RegisterInfoGPROutForOther.td" // except AsmParser
include "Cpu0.td"
```

Cpu0Other.td and Cpu0.td includes a few other .td files. Cpu0RegisterInfo.td (shown below) describes the Cpu0's set of registers. In this file, we see that each register has been given a name. For example, **"def PC"** indicates that there is a register name as PC. Beside of register information, it also define register class information. You may have multiple register classes such as CPURegs, SR, C0Regs and GPROut. GPROut defined in Cpu0RegisterInfoGPROutForOther.td which include CPURegs except SW, so SW won't be allocated as the output registers in register allocation stage.

lbdex/chapters/Chapter2/Cpu0RegisterInfo.td

```
//===-- Cpu0RegisterInfo.td - Cpu0 Register defs -----------*- tablegen -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

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

// We have banks of 16 registers each.
class Cpu0Reg<bits<16> Enc, string n> : Register<n> {
  // For tablegen(... -gen-emitter)  in CMakeLists.txt
  let HWEncoding = Enc;

  let Namespace = "Cpu0";
}

// Cpu0 CPU Registers
class Cpu0GPRReg<bits<16> Enc, string n> : Cpu0Reg<Enc, n>;

// Co-processor 0 Registers
class Cpu0C0Reg<bits<16> Enc, string n> : Cpu0Reg<Enc, n>;

//===----------------------------------------------------------------------===//
//@Registers
//===----------------------------------------------------------------------===//
// The register string, such as "9" or "gp" will show on "llvm-objdump -d"
//@ All registers definition
let Namespace = "Cpu0" in {
  //@ General Purpose Registers
  def ZERO : Cpu0GPRReg<0,  "zero">, DwarfRegNum<[0]>;
  def AT   : Cpu0GPRReg<1,  "1">,    DwarfRegNum<[1]>;
  def V0   : Cpu0GPRReg<2,  "2">,    DwarfRegNum<[2]>;
  def V1   : Cpu0GPRReg<3,  "3">,    DwarfRegNum<[3]>;
  def A0   : Cpu0GPRReg<4,  "4">,    DwarfRegNum<[4]>;
  def A1   : Cpu0GPRReg<5,  "5">,    DwarfRegNum<[5]>;
  def T9   : Cpu0GPRReg<6,  "t9">,   DwarfRegNum<[6]>;
  def T0   : Cpu0GPRReg<7,  "7">,    DwarfRegNum<[7]>;
  def T1   : Cpu0GPRReg<8,  "8">,    DwarfRegNum<[8]>;
  def S0   : Cpu0GPRReg<9,  "9">,    DwarfRegNum<[9]>;
  def S1   : Cpu0GPRReg<10, "10">,   DwarfRegNum<[10]>;
  def GP   : Cpu0GPRReg<11, "gp">,   DwarfRegNum<[11]>;
  def FP   : Cpu0GPRReg<12, "fp">,   DwarfRegNum<[12]>;
  def SP   : Cpu0GPRReg<13, "sp">,   DwarfRegNum<[13]>;
  def LR   : Cpu0GPRReg<14, "lr">,   DwarfRegNum<[14]>;
  def SW   : Cpu0GPRReg<15, "sw">,   DwarfRegNum<[15]>;
//  def MAR  : Register< 16, "mar">,  DwarfRegNum<[16]>;
//  def MDR  : Register< 17, "mdr">,  DwarfRegNum<[17]>;

  def PC   : Cpu0C0Reg<0, "pc">,  DwarfRegNum<[20]>;
  def EPC  : Cpu0C0Reg<1, "epc">, DwarfRegNum<[21]>;
}
```

```
//===----------------------------------------------------------------------===//
//@Register Classes
//===----------------------------------------------------------------------===//

def CPURegs : RegisterClass<"Cpu0", [i32], 32, (add
  // Reserved
  ZERO, AT,
  // Return Values and Arguments
  V0, V1, A0, A1,
  // Not preserved across procedure calls
  T9, T0, T1,
  // Callee save
  S0, S1,
  // Reserved
  GP, FP,
  SP, LR, SW)>;

//@Status Registers class
def SR     : RegisterClass<"Cpu0", [i32], 32, (add SW)>;

//@Co-processor 0 Registers class
def C0Regs : RegisterClass<"Cpu0", [i32], 32, (add PC, EPC)>;
```

lbdex/chapters/Chapter2/Cpu0RegisterInfoGPROutForOther.td

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

def GPROut : RegisterClass<"Cpu0", [i32], 32, (add (sub CPURegs, SW))>;
```

In C++, class typically provides a structure to lay out some data and functions, while definitions are used to allocate memory for specific instances of a class. For example:

```
class Date {  // declare Date
  int year, month, day;
};
Date birthday;  // define birthday, an instance of Date
```

The class **Date** has the members **year**, **month**, and **day**, but these do not yet belong to an actual object. By defining an instance of **Date** called **birthday**, you have allocated memory for a specific object, and can set the **year**, **month**, and **day** of this instance of the class.

In .td files, class describes the structure of how data is laid out, while definitions act as the specific instances of the class. If you look back at the Cpu0RegisterInfo.td file, you will see a class called **Cpu0Reg** which is derived from the **Register** class provided by LLVM. **Cpu0Reg** inherits all the fields that exist in the **Register** class. The “let HWEncoding = Enc” which means assign field HWEncoding from parameter Enc. Since Cpu0 reserve 4 bits for 16 registers in instruction format, the assigned value range is from 0 to 15. Once assigning the 0 to 15 to HWEncoding, the backend register number will be got from the function of llvm register class since TableGen will set this number automatically.

The **def** keyword is used to create instances of class. In the following line, the ZERO register is defined as a member of the **Cpu0GPRReg** class:

```
def ZERO : Cpu0GPRReg< 0, "ZERO">, DwarfRegNum<[0]>;
```

The **def ZERO** indicates the name of this register. **<0, “ZERO”>** are the parameters used when creating this specific instance of the **Cpu0GPRReg** class, thus the field **Enc** is set to 0, and the string **n** is set to **ZERO**.

As the register lives in the **Cpu0** namespace, you can refer to the ZERO register in backend C++ code by using **Cpu0::ZERO**.

Notice the use of the **let** expressions: these allow you to override values that are initially defined in a superclass. For example, **let Namespace = “Cpu0”** in the **Cpu0Reg** class will override the default namespace declared in **Register** class. The Cpu0RegisterInfo.td also defines that **CPURegs** is an instance of the class **RegisterClass**, which is an built-in LLVM class. A **RegisterClass** is a set of **Register** instances, thus **CPURegs** can be described as a set of registers.

The Cpu0 instructions td is named to Cpu0InstrInfo.td which contents as follows,

lbdex/chapters/Chapter2/Cpu0InstrInfo.td

```
//===- Cpu0InstrInfo.td - Target Description for Cpu0 Target -*- tablegen -*-=//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the Cpu0 implementation of the TargetInstrInfo class.
//
//===----------------------------------------------------------------------===//

//===----------------------------------------------------------------------===//
// Cpu0 profiles and nodes
//===----------------------------------------------------------------------===//

def SDT_Cpu0Ret          : SDTypeProfile<0, 1, [SDTCisInt<0>]>;

// Return
def Cpu0Ret : SDNode<"Cpu0ISD::Ret", SDTNone,
                     [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;

//===----------------------------------------------------------------------===//
// Instruction format superclass
//===----------------------------------------------------------------------===//

include "Cpu0InstrFormats.td"

//===----------------------------------------------------------------------===//
// Cpu0 Operand, Complex Patterns and Transformations Definitions.
//===----------------------------------------------------------------------===//
// Instruction operand types

// Signed Operand
def simm16      : Operand<i32> {
  let DecoderMethod= "DecodeSimm16";
}

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

// Node immediate fits as 16-bit sign extended on target immediate.
// e.g. addi, andi
def immSExt16  : PatLeaf<(imm), [{ return isInt<16>(N->getSExtValue()); }]>;

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

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

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

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

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

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.
class LoadM32<bits<8> op, string instr_asm, PatFrag OpNode,
                   bit Pseudo = 0>
  : LoadM<op, instr_asm, OpNode, GPROut, mem, Pseudo> {
}

// 32-bit store.
class StoreM32<bits<8> op, string instr_asm, PatFrag OpNode,
                    bit Pseudo = 0>
  : StoreM<op, instr_asm, OpNode, GPROut, mem, Pseudo> {
}

//@JumpFR {
let isBranch=1, isTerminator=1, isBarrier=1, imm16=0, hasDelaySlot = 1,
    isIndirectBranch = 1 in
class JumpFR<bits<8> op, string instr_asm, RegisterClass RC>:
  FL<op, (outs), (ins RC:$ra),
     !strconcat(instr_asm, "\t$ra"), [(brind RC:$ra)], IIBranch> {
  let rb = 0;
  let imm16 = 0;
}
//@JumpFR }

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


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

//===----------------------------------------------------------------------===//
// Cpu0 Instructions
//===----------------------------------------------------------------------===//

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

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

/// Arithmetic Instructions (3-Operand, R-Type)

/// Shift Instructions

def JR      : JumpFR<0x3c, "jr", GPROut>;
def RET     : RetBase<GPROut>;

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

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

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

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

lbdex/chapters/Chapter2/Cpu0InstrFormats.td

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

//===----------------------------------------------------------------------===//
//  Describe CPU0 instructions format
//
//  CPU INSTRUCTION FORMATS
//
//  opcode  - operation code.
//  ra      - dst reg, only used on 3 regs instr.
//  rb      - src reg.
//  rc      - src reg (on a 3 reg instr).
//  cx      - immediate
//
//===----------------------------------------------------------------------===//

// Format specifies the encoding used by the instruction.  This is part of the
// ad-hoc solution used to emit machine instruction encodings by our machine
// code emitter.
class Format<bits<4> val> {
  bits<4> Value = val;
}

def Pseudo    : Format<0>;
def FrmA      : Format<1>;
def FrmL      : Format<2>;
def FrmJ      : Format<3>;
def FrmOther  : Format<4>; // Instruction w/ a custom format

// Generic Cpu0 Format
class Cpu0Inst<dag outs, dag ins, string asmstr, list<dag> pattern,
               InstrItinClass itin, Format f>: Instruction
{
  // Inst and Size: for tablegen(... -gen-emitter) and
  // tablegen(... -gen-disassembler) in CMakeLists.txt
  field bits<32> Inst;
  Format Form = f;

  let Namespace = "Cpu0";

  let Size = 4;

  bits<8> Opcode = 0;

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

  let OutOperandList = outs;
  let InOperandList  = ins;

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

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

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

  let DecoderNamespace = "Cpu0";

  field bits<32> SoftFail = 0;
}

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

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

  let Opcode = op;

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

//@class FL {
//===----------------------------------------------------------------------===//
// Format L instruction class in Cpu0 : <|opcode|ra|rb|cx|>
//===----------------------------------------------------------------------===//

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

  let Opcode = op;

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

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

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

  let Opcode = op;

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

ADDiu is a instance of class ArithLogicI inherited from FL, it can be expanded and get member value further as follows,

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

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

So,

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

To expand the td, some principles are:

let: meaning override the existed field from parent class.

For instance: let isReMaterializable = 1; override the isReMaterializable from class instruction of Target.td.

declaration: meaning declare a new field for this class.

For instance: bits<4> ra; declare ra field for class FL.

The details of expanding as the following table:

Table 13 ADDiu expand part I

ADDiu	ArithLogicI	FL
0x09	op = 0x09	Opcode = 0x09;
addiu	instr_asm = “addiu”	(outs GPROut:$ra); !strconcat(“addiu”, “t$ra, $rb, $imm16”);
add	OpNode = add	[(set GPROut:$ra, (add CPURegs:$rb, immSExt16:$imm16))]
simm16	Od = simm16	(ins CPURegs:$rb, simm16:$imm16);
immSExt16	imm_type = immSExt16	Inst{15-0} = imm16;
CPURegs	RC = CPURegs isReMaterializable=1;	Inst{23-20} = ra; Inst{19-16} = rb;

Table 14 ADDiu expand part II

Cpu0Inst	instruction
Namespace = “Cpu0”	Uses = []; …
Inst{31-24} = 0x09;	Size = 0; …
OutOperandList = GPROut:$ra;	
InOperandList = CPURegs:$rb,simm16:$imm16;	
AsmString = “addiut$ra, $rb, $imm16”	
pattern = [(set GPROut:$ra, (add RC:$rb, immSExt16:$imm16))]	
Itinerary = IIAlu	
TSFlags{3-0} = FrmL.value	
DecoderNamespace = “Cpu0”	

The td expanding is a lousy process. Similarly, LD and ST instruction definition can be expanded in this way. Please notice the Pattern = [(set GPROut:$ra, (add RC:$rb, immSExt16:$imm16))] which include keyword **“add”**. The ADDiu with **“add”** is used in sub-section Instruction Selection of last section.

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

lbdex/chapters/Chapter2/Cpu0Schedule.td

```
//===-- Cpu0Schedule.td - Cpu0 Scheduling Definitions ------*- tablegen -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//

//===----------------------------------------------------------------------===//
// Functional units across Cpu0 chips sets. Based on GCC/Cpu0 backend files.
//===----------------------------------------------------------------------===//
def ALU     : FuncUnit;
def IMULDIV : FuncUnit;

//===----------------------------------------------------------------------===//
// Instruction Itinerary classes used for Cpu0
```

```
//===----------------------------------------------------------------------===//
def IIAlu              : InstrItinClass;
def II_CLO             : InstrItinClass;
def II_CLZ             : InstrItinClass;
def IILoad             : InstrItinClass;
def IIStore            : InstrItinClass;
def IIBranch           : InstrItinClass;

def IIPseudo           : InstrItinClass;

//===----------------------------------------------------------------------===//
// Cpu0 Generic instruction itineraries.
//===----------------------------------------------------------------------===//
//@ http://llvm.org/docs/doxygen/html/structllvm_1_1InstrStage.html
def Cpu0GenericItineraries : ProcessorItineraries<[ALU, IMULDIV], [], [
//@2
  InstrItinData<IIAlu              , [InstrStage<1,  [ALU]>]>,
  InstrItinData<II_CLO             , [InstrStage<1,  [ALU]>]>,
  InstrItinData<II_CLZ             , [InstrStage<1,  [ALU]>]>,
  InstrItinData<IILoad             , [InstrStage<3,  [ALU]>]>,
  InstrItinData<IIStore            , [InstrStage<1,  [ALU]>]>,
  InstrItinData<IIBranch           , [InstrStage<1,  [ALU]>]>
]>;
```

Write cmake file

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

lbdex/chapters/Chapter2/CMakeLists.txt

```
add_llvm_component_group(Cpu0)

set(LLVM_TARGET_DEFINITIONS Cpu0Other.td)

# Generate Cpu0GenRegisterInfo.inc and Cpu0GenInstrInfo.inc which included by
#  your hand code C++ files.
# Cpu0GenRegisterInfo.inc came from Cpu0RegisterInfo.td, Cpu0GenInstrInfo.inc
#  came from Cpu0InstrInfo.td.
tablegen(LLVM Cpu0GenRegisterInfo.inc -gen-register-info)
tablegen(LLVM Cpu0GenInstrInfo.inc -gen-instr-info)
tablegen(LLVM Cpu0GenSubtargetInfo.inc -gen-subtarget)
tablegen(LLVM Cpu0GenMCPseudoLowering.inc -gen-pseudo-lowering)

# Cpu0CommonTableGen must be defined
add_public_tablegen_target(Cpu0CommonTableGen)

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

  LINK_COMPONENTS
  Analysis
  AsmPrinter
  CodeGen
  Core
  MC
  Cpu0Desc
  Cpu0Info
  SelectionDAG
  Support
  Target
  GlobalISel

  ADD_TO_COMPONENT
  Cpu0
  )

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

CMakeLists.txt is the make information for cmake and # is comment. Comments are prefixed by **#** in both files. The “tablegen(” in above CMakeLists.txt is defined in cmake/modules/TableGen.cmake as below,

llvm/cmake/modules/TableGen.cmake

```
function(tablegen project ofn)
  ...
  add_custom_command(OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/${ofn}.tmp
    # Generate tablegen output in a temporary file.
    COMMAND ${${project}_TABLEGEN_EXE} ${ARGN} -I ${CMAKE_CURRENT_SOURCE_DIR}
  ...
endfunction()
...
macro(add_tablegen target project)
  ...
  if(LLVM_USE_HOST_TOOLS)
    if( ${${project}_TABLEGEN} STREQUAL "${target}" )
      if (NOT CMAKE_CONFIGURATION_TYPES)
        set(${project}_TABLEGEN_EXE "${LLVM_NATIVE_BUILD}/bin/${target}")
      else()
        set(${project}_TABLEGEN_EXE "${LLVM_NATIVE_BUILD}/Release/bin/${target}")
      endif()
  ...
endmacro()
```

llvm/utils/TableGen/CMakeLists.txt

```
add_tablegen(llvm-tblgen LLVM
  ...
)
```

Above "add_tablegen" in llvm/utils/TableGen/CMakeLists.txt makes the "tablegen(" written in Cpu0 CMakeLists.txt an alias of llvm-tblgen (${project} = LLVM and ${project}_TABLEGEN_EXE = llvm-tblgen). The "tablegen(", "add_public_tablegen_target(Cpu0CommonTableGen)" in lbdex/chapters/Chapter2/CMakeLists.txt and the following code define a target "Cpu0CommonTableGen" with it's output files "Cpu0Gen*.inc" as follows,

llvm/cmake/modules/TableGen.cmake

```
function(tablegen project ofn)
  ...
  set(TABLEGEN_OUTPUT ${TABLEGEN_OUTPUT} ${CMAKE_CURRENT_BINARY_DIR}/${ofn} PARENT_SCOPE)
  ...
endfunction()

# Creates a target for publicly exporting tablegen dependencies.
function(add_public_tablegen_target target)
  ...
  add_custom_target(${target}
    DEPENDS ${TABLEGEN_OUTPUT})
  ...
endfunction()
```

Since execution file llvm-tblgen is built before compiling any llvm backend source code during building llvm, the llvm-tblgen is always ready for backend's TableGen reguest.

This book breaks the whole backend source code by function, add code chapter by chapter. Don't try to understand everything in the text of book, the code added in each chapter is a reading material too. To understand the computer related knowledge in concept, you can ignore source code, but implementing based on an existed open software cannot. In programming, documentation cannot replace the source code totally. Reading source code is a big opportunity in the open source development.

CMakeLists.txt exists in sub-directories **MCTargetDesc** and **TargetInfo**. The contents of MakeLists.txt in these two directories instruct llvm generating Cpu0Desc and Cpu0Info libraries, repectively. 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" [28].

Target Registration

You must also register your target with the TargetRegistry. After registration, llvm tools are 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.

lbdex/chapters/Chapter2/Cpu0.h

```
//===-- Cpu0.h - Top-level interface for Cpu0 representation ----*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
//
// This file contains the entry points for global functions defined in
// the LLVM Cpu0 back-end.
//
//===----------------------------------------------------------------------===//

#ifndef LLVM_LIB_TARGET_CPU0_CPU0_H
#define LLVM_LIB_TARGET_CPU0_CPU0_H

#include "Cpu0Config.h"
#include "MCTargetDesc/Cpu0MCTargetDesc.h"
#include "llvm/Target/TargetMachine.h"

namespace llvm {
  class Cpu0TargetMachine;
  class FunctionPass;

} // end namespace llvm;

#endif
```

lbdex/chapters/Chapter2/TargetInfo/Cpu0TargetInfo.cpp

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

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

Target llvm::TheCpu0Target, llvm::TheCpu0elTarget;

extern "C" void LLVMInitializeCpu0TargetInfo() {
  RegisterTarget<Triple::cpu0,
        /*HasJIT=*/true> X(TheCpu0Target, "cpu0", "CPU0 (32-bit big endian)", "Cpu0");

  RegisterTarget<Triple::cpu0el,
        /*HasJIT=*/true> Y(TheCpu0elTarget, "cpu0el", "CPU0 (32-bit little endian)", "Cpu0");
}
```

lbdex/chapters/Chapter2/TargetInfo/CMakeLists.txt

```
# llvm 10.0.0 change from add_llvm_library to add_llvm_component_library
add_llvm_component_library(LLVMCpu0Info
  Cpu0TargetInfo.cpp

  LINK_COMPONENTS
  Support

  ADD_TO_COMPONENT
  Cpu0
  )
```

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

lbdex/chapters/Chapter2/Cpu0TargetMachine.cpp

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

#include "Cpu0TargetMachine.h"
#include "Cpu0.h"

#include "llvm/IR/Attributes.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/CodeGen.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/CodeGen/TargetPassConfig.h"
#include "llvm/Support/TargetRegistry.h"
#include "llvm/Target/TargetOptions.h"

using namespace llvm;

#define DEBUG_TYPE "cpu0"

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

lbdex/chapters/Chapter2/MCTargetDesc/Cpu0MCTargetDesc.h

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

#ifndef LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCTARGETDESC_H
#define LLVM_LIB_TARGET_CPU0_MCTARGETDESC_CPU0MCTARGETDESC_H

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

#include <memory>

namespace llvm {
class Target;
class Triple;

extern Target TheCpu0Target;
extern Target TheCpu0elTarget;

} // End llvm namespace

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

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

#define GET_SUBTARGETINFO_ENUM
#include "Cpu0GenSubtargetInfo.inc"

#endif
```

lbdex/chapters/Chapter2/MCTargetDesc/Cpu0MCTargetDesc.cpp

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

#include "Cpu0MCTargetDesc.h"
#include "llvm/MC/MachineLocation.h"
#include "llvm/MC/MCELFStreamer.h"
#include "llvm/MC/MCInstrAnalysis.h"
#include "llvm/MC/MCInstPrinter.h"
#include "llvm/MC/MCInstrInfo.h"
#include "llvm/MC/MCRegisterInfo.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/MC/MCSymbol.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/FormattedStream.h"
#include "llvm/Support/TargetRegistry.h"

using namespace llvm;

#define GET_INSTRINFO_MC_DESC
#include "Cpu0GenInstrInfo.inc"

#define GET_SUBTARGETINFO_MC_DESC
#include "Cpu0GenSubtargetInfo.inc"

#define GET_REGINFO_MC_DESC
#include "Cpu0GenRegisterInfo.inc"

//@2 {
extern "C" void LLVMInitializeCpu0TargetMC() {

}
//@2 }
```

lbdex/chapters/Chapter2/MCTargetDesc/CMakeLists.txt

```
# MCTargetDesc/CMakeLists.txt
add_llvm_component_library(LLVMCpu0Desc
  Cpu0MCTargetDesc.cpp

  LINK_COMPONENTS
  MC
  Cpu0Info
  Support

  ADD_TO_COMPONENT
  Cpu0
  )
```

Please see “Target Registration” [29] for reference.

Build libraries and td

Build steps **https://github.com/Jonathan2251/lbd/blob/master/README.md**. We set llvm source code in /Users/Jonathan/llvm/debug/llvm and have llvm debug-build in /Users/Jonathan/llvm/debug/build. About how to build llvm, please refer here [30]. In appendix A, we made a copy from /Users/Jonathan/llvm/debug/llvm to /Users/Jonathan/llvm/test/llvm for working with my Cpu0 target backend. Sub-directories llvm is for source code and build is for debug build directory.

Beside directory llvm/lib/Target/Cpu0, there are a couple of files modified to support cpu0 new Target, which includes both the ID and name of machine and relocation records listed in the early sub-section. You can update your llvm working copy and find the modified files by commands, cp -rf lbdex/llvm/modify/llvm/* <yourllvm/workingcopy/sourcedir>/.

```
118-165-78-230:lbd Jonathan$ pwd
/Users/Jonathan/git/lbd
118-165-78-230:lbd Jonathan$ cp -rf lbdex/llvm/modify/llvm/* ~/llvm/test/llvm/.
118-165-78-230:lbd Jonathan$ grep -R "cpu0" ~/llvm/test/llvm/include
llvm/cmake/config-ix.cmake:elseif (LLVM_NATIVE_ARCH MATCHES "cpu0")
llvm/include/llvm/ADT/Triple.h:#undef cpu0
llvm/include/llvm/ADT/Triple.h:    cpu0,       // For Tutorial Backend Cpu0
llvm/include/llvm/ADT/Triple.h:    cpu0el,
llvm/include/llvm/Support/ELF.h:  EF_CPU0_ARCH_32R2 = 0x70000000, // cpu032r2
llvm/include/llvm/Support/ELF.h:  EF_CPU0_ARCH_64R2 = 0x80000000, // cpu064r2
...
```

Next configure the Cpu0 example code to chapter2 as follows,

~/llvm/test/llvm/lib/Target/Cpu0/Cpu0SetChapter.h

```
#define CH       CH2
```

Beside configure chapter as above, I provide gen-chapters.sh that you can get each chapter code as follows,

```
118-165-78-230:lbdex Jonathan$ pwd
/Users/Jonathan/git/lbd/lbdex
118-165-78-230:lbdex Jonathan$ bash gen-chapters.sh
118-165-78-230:lbdex Jonathan$ ls chapters
Chapter10_1   Chapter11_2     Chapter2        Chapter3_2...
Chapter11_1   Chapter12_1     Chapter3_1      Chapter3_3...
```

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

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

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

118-165-78-230:build Jonathan$ make -j4

118-165-78-230:build Jonathan$
```

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

```
118-165-78-230:build Jonathan$ /Users/Jonathan/llvm/test/
build/bin/llc --version
LLVM (http://llvm.org/):
...
  Registered Targets:
  arm      - ARM
  ...
  cpp      - C++ backend
  cpu0     - Cpu0
  cpu0el   - Cpu0el
...
```

The `llc -version` can display Registered Targets **"cpu0"** and **"cpu0el"**, because the code in file TargetInfo/Cpu0TargetInfo.cpp we made in last sub-section "Target Registration" **[31]**.

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

```
118-165-75-57:test Jonathan$ pwd
/Users/Jonathan/test
118-165-75-57:test Jonathan$ cp -rf lbdex/Cpu0 ~/llvm/test/llvm/lib/Target/.

118-165-75-57:test Jonathan$ cd ~/llvm/test/build
118-165-75-57:build Jonathan$ pwd
/Users/Jonathan/llvm/test/build
118-165-75-57:build Jonathan$ rm -rf *
118-165-75-57:build Jonathan$ cmake -DCMAKE_CXX_COMPILER=clang++
-DCMAKE_C_COMPILER=clang -DCMAKE_BUILD_TYPE=Debug -DLLVM_TARGETS_TO_BUILD=Cpu0
```

```
-G "Unix Makefiles" ../llvm/
...
-- Targeting Cpu0
...
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/Jonathan/llvm/test/build
```

In order to save time, we build Cpu0 target only by option -DLLVM_TARGETS_TO_BUILD=Cpu0. After that, you can find the *.inc files in directory /Users/Jonathan/llvm/test/build/lib/Target/Cpu0 as follows,

build/lib/Target/Cpu0/Cpu0GenRegisterInfo.inc

```
namespace Cpu0 {
enum {
  NoRegister,
  AT = 1,
  EPC = 2,
  FP = 3,
  GP = 4,
  HI = 5,
  LO = 6,
  LR = 7,
  PC = 8,
  SP = 9,
  SW = 10,
  ZERO = 11,
  A0 = 12,
  A1 = 13,
  S0 = 14,
  S1 = 15,
  T0 = 16,
  T1 = 17,
  T9 = 18,
  V0 = 19,
  V1 = 20,
  NUM_TARGET_REGS     // 21
};
}
...
```

These *.inc are generated by llvm-tblgen at directory build/lib/Target/Cpu0 where their input files are the Cpu0 backend *.td files. The llvm-tblgen is invoked by **tablegen** of /Users/Jonathan/llvm/test/llvm/lib/Target/Cpu0/CMakeLists.txt. These *.inc files will be included by Cpu0 backend *.cpp or *.h files and compile into *.o further. TableGen is the important tool illustrated in the early sub-section ".td: LLVM's Target Description Files" of this chapter. List it again as follows,

"The "mix and match" approach allows target authors to choose what makes sense for their architecture and permits a large amount of code reuse across different targets".

Details about TableGen are here **[32] [33] [34]**.

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

lbdex/input/ch3.cpp

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

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

```
118-165-78-230:input Jonathan$ pwd
/Users/Jonathan/git/lbd/lbdex/input
118-165-78-230:input Jonathan$ clang -target mips-unknown-linux-gnu -c
ch3.cpp -emit-llvm -o ch3.bc
```

As above, compile C to .bc by `clang -target mips-unknown-linux-gnu` because Cpu0 borrows the ABI from Mips. Next step, transfer bitcode .bc to human readable text format as follows,

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

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

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

Now, when compiling ch3.bc will get the error message as follows,

```
118-165-78-230:input Jonathan$ /Users/Jonathan/llvm/test/build/
bin/llc -march=cpu0 -relocation-model=pic -filetype=asm ch3.bc -o
ch3.cpu0.s
...
... Assertion `target.get() && "Could not allocate target machine!"' failed
...
```

At this point, we finish the Target Registration for Cpu0 backend. The backend compiler command `llc` can recognize Cpu0 backend now. Currently we just define target td files (Cpu0.td, Cpu0Other.td, Cpu0RegisterInfo.td, …). According to LLVM structure, we need to define our target machine and include those td related files. The error message says we didn't define our target machine. This book is a step-by-step backend delvelopment. You can review the houndreds lines of Chapter2 example code to see how to do the Target Registration.

Options of llc for debug

llc –help-hidden

The following options for llc need to give a input .bc or .ll file.

-debug:
-debug-pass=Structure
-print-after-all, -print-before-all
-print-before="pass" and -print-after="pass", eg. -print-before="postra-machine-sink" and -print-after="postra-machine-sink". The pass name can be got as follows,

```
CodeGen % pwd
~/llvm/debug/llvm/lib/CodeGen
CodeGen % grep -R "INITIALIZE_PASS" |grep sink
./MachineSink.cpp:INITIALIZE_PASS(PostRAMachineSinking, "postra-machine-sink",
```

-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.
-march=<string>, eg. march=mips;
-relocation-model=static/pic
-filetype=asm/obj

Use F.dump() in code where F is class Function for passes in llvm/lib/Transformation.

Options of opt

Check from *opt –help-hidden* and LLVM passes **[35]**. Eg.

opt -dot-cfg input.ll: Print CFG of function to 'dot' file
-dot-cfg-only : Print CFG of function to 'dot' file (with no function bodies)

[1] Original Cpu0 architecture and ISA details (Chinese). **http://cccckmit.wikidot.com/ocs:cpu0**

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

[3](1,2,3,4) The difference between LB and LBu is signed and unsigned byte value expand to a word size. For example, After LB Ra, [Rb+Cx], Ra is 0xffffff80(= -128) if byte [Rb+Cx] is 0x80; Ra is 0x0000007f(= 127) if byte [Rb+Cx] is 0x7f. After LBu Ra, [Rb+Cx], Ra is 0x00000080(= 128) if byte [Rb+Cx] is 0x80; Ra is 0x0000007f(= 127) if byte [Rb+Cx] is 0x7f. Difference between LH and LHu is similar.

[4](1,2,3,4) The only difference between ADDu instruction and the ADD instruction is that the ADDU instruction never causes an Integer Overflow exception. SUBu and SUB is similar.

[5](1,2) CMP is signed-compare while CMPu is unsigned. Conditions include the following comparisons: >, >=, ==, !=, <=, <. SW is actually set by the subtraction of the two register operands, and the flags indicate which conditions are present.

[6](1,2) Rb '>> Cx, Rb '>> Rc: Shift with signed bit remain.

[7] jsub cx is direct call for 24 bits value of cx while jalr $rb is indirect call for 32 bits value of register $rb.

[8] Both JR and RET has same opcode (actually they are the same instruction for Cpu0 hardware). When user writes "jr $t9" meaning it jumps to address of register $t9; when user writes "jr $lr" meaning it jump back to the caller function (since $lr is the return address). For user read ability, Cpu0 prints "ret $lr" instead of "jr $lr".

[9] https://en.wikipedia.org/wiki/Comparison_of_Java_and_C%2B%2B

[10](1,2) Chris Lattner, **LLVM**. Published in The Architecture of Open Source Applications. http://www.aosabook.org/en/llvm.html

[11] http://jonathan2251.github.io/lbd/doc.html#generate-cpu0-document

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

[13](1,2,3) http://llvm.org/docs/LangRef.html

[14] https://www.cs.cmu.edu/afs/cs/academic/class/15745-s16/www/lectures/L23-Register-Coalescing.pdf

[15] https://en.wikipedia.org/wiki/Register_allocation

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

[17] https://en.wikipedia.org/wiki/GNU_Compiler_Collection

[18] https://en.wikipedia.org/wiki/GNU_Compiler_Collection#Front_ends

[19] https://gcc.gnu.org/onlinedocs/gccint/GENERIC.html

[20] https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html

[21] https://gcc.gnu.org/onlinedocs/gccint/RTL.html

[22] https://gcc.gnu.org/onlinedocs/gccint/Machine-Desc.html#Machine-Desc

[23] https://stackoverflow.com/questions/40799696/how-is-gcc-ir-different-from-llvm-ir/40802063

[24] https://github.com/Jonathan2251/lbd/tree/master/References/null_pointer.cpp is an example.

[25] Dereferencing a NULL Pointer: contrary to popular belief, dereferencing a null pointer in C is undefined. It is not defined to trap, and if you mmap a page at 0, it is not defined to access that page. This falls out of the rules that forbid dereferencing wild pointers and the use of NULL as a sentinel, from http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html. As link, https://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html. In this case, the developer forgot to call "set", did not crash with a null pointer dereference, and their code broke when someone else did a debug build.

[26] https://en.wikipedia.org/wiki/Control-flow_graph

[27](1,2) Refer section 8.5 of book Compilers: Principles, Techniques, and Tools (2nd Edition)

[28] http://llvm.org/docs/CMake.html

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

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

[31] http://jonathan2251.github.io/lbd/llvmstructure.html#target-registration

[32](1,2) http://llvm.org/docs/TableGen/index.html

[33] http://llvm.org/docs/TableGen/LangIntro.html

[34] http://llvm.org/docs/TableGen/LangRef.html

[35] https://llvm.org/docs/Passes.html