8-2017

The Design of a Custom 32-bit RISC CPU and LLVM Compiler Backend

Connor Jan Goldberg
cjg3259@rit.edu

The Design of a Custom 32-bit RISC CPU and LLVM Compiler Backend

by

Connor Jan Goldberg

Graduate Paper

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Electrical Engineering

Approved by:

Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

---

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2017

To my family and friends, for all of their endless love, support, and encouragement throughout my career at Rochester Institute of Technology

Abstract

Compiler infrastructures are often an area of high interest for research. As the necessity for digital information and technology increases, so does the need for an increase in the performance of digital hardware. The main component in most complex digital systems is the central processing unit (CPU). Compilers are responsible for translating code written in a high-level programming language to a sequence of instructions that is then executed by the CPU. Most research in compiler technologies is focused on the design and optimization of the code written by the programmer; however, at some point in this process the code must be converted to instructions specific to the CPU. This paper presents the design of a simplified CPU architecture as well as the less understood side of compilers: the backend, which is responsible for the CPU instruction generation. The CPU design is a 32-bit reduced instruction set computer (RISC) and is written in Verilog. Unlike most embedded-style RISC architectures, which have a compiler port for GCC (The GNU Compiler Collection), this compiler backend was written for the LLVM compiler infrastructure project. Code generated from the LLVM backend is successfully simulated on the custom CPU with Cadence Incisive, and the CPU is synthesized using Synopsys Design Compiler.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This paper is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Connor Jan Goldberg

August 2017

Acknowledgements

I would like to thank my advisor, professor, and mentor, Mark A. Indovina, for all of his guidance and feedback throughout the entirety of this project. He is the reason for my love of digital hardware design and drove me to pursue it as a career path. He has been a tremendous help and a true friend during my graduate career at RIT.

Another professor I would like to thank is Dr. Dorin Patru. He led me to thoroughly enjoy computer architecture and always provided helpful knowledge and feedback for my random questions.

Additionally, I want to thank the Tight Squad, for giving me true friendship, endless laughs, and great company throughout the many, many long nights spent in the labs.

I would also like to thank my best friends, Lincoln and Matt. This project would not have been possible without their love, advice, and companionship throughout my entire career at RIT.

Finally I need to thank my amazing parents and brother. My family has been the inspiration for everything I strive to accomplish and my success would be nothing if not for their motivation, support, and love.

Forward

The paper describes a custom RISC CPU and associated LLVM compiler backend as a Graduate Research project undertaken by Connor Goldberg. Closing the loop between a new CPU architecture and companion compiler is no small feat; Mr.Goldberg took on the challenge with exemplary results. Without question I am extremely proud of the research work produced by this fine student.

Mark A. Indovina

Rochester, NY USA

August 2017

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Compiler infrastructures are a popular area of research in computer science. Almost every modern-day problem that arises yields a solution that makes use of software at some point in its implementation. This places an extreme importance on compilers as the tools to translate software from its written state, to a state that can be used by the central processing unit (CPU). The majority of compiler research is focused on functionality to efficiently read and optimize the input software. However, half of a compiler's functionality is to generate machine instructions for a specific CPU architecture. This area of compilers, the backend, is largely overlooked and undocumented.

With the goal to explore the backend design of compilers, a custom, embedded-style, 32-bit reduced instruction set computer (RISC) CPU was designed to be targeted by a C code compiler. Because designing such a compiler from scratch was not a feasible option for this project, two existing and mature compilers were considered as starting points: the GNU compiler collection (GCC) and LLVM. Although GCC has the capability of generating code for a wide variety of CPU architectures, the same is not true for LLVM. LLVM is a relatively new project; however, it has a very modern design and seemed to

be well documented. LLVM was chosen for these reasons, and additionally to explore the reason for its seeming lack of popularity within the embedded CPU community.

This project aims to provide a view into the process of taking a C function from source code to machine code, which can be executed on CPU hardware through the LLVM compiler infrastructure. Throughout Chapters 4 and 5, a simple C function is used as an example to detail the flow from C code to machine code execution. The machine code is simulated on the custom CPU using Cadence Incisive and synthesized with Synopsys Design Compiler.

1.1 Organization

Chapter 2 discusses the basic design of CPUs and compilers to provide some background information. Chapter 3 presents the design and implementation of the custom RISC CPU and architecture. Chapter 4 presents the design and implementation of the custom LLVM compiler backend. Chapter 5 shows tests and results from the implementation of LLVM compiler backend for the custom RISC CPU to show where this project succeeds and fails. Chapter 6 discusses possible future work and the concludes the paper.

Chapter 2

The Design of CPUs and Compilers

This chapter discusses relevant concepts and ideas pertaining to CPU architecture and compiler design.

2.1 CPU Design

The two prominent CPU design methodologies are reduced instruction set computer (RISC) and complex instruction set computer (CISC). While there is not a defined standard to separate specific CPU architectures into these two categories, it is common for most architectures to be easily classified into one or the other depending on their defining characteristics.

One key indicator as to whether an architecture is RISC or CISC is the number of CPU instructions along with the complexity of the instructions. RISC architectures are known for having a relatively small number of instructions that typically only perform one or two operations in a single clock cycle. However, CISC architectures are known for having a large number of instructions that typically perform multiple, complex operations

over multiple clock cycles [1]. For example, the ARM instruction set contains around 50 instructions [2], while the Intel x86-64 instruction set contains over 600 instructions [3]. This simple contrast highlights the main design objectives of the two categories; RISC architectures generally aim for lower complexity in the architecture and hardware design so as to shift the complexity into software, and CISC architectures aim to keep a bulk of the complexity in hardware with the goal of simplifying software implementations. While it might seem beneficial to shift complexity to hardware, it also causes hardware verification to increase in complexity. This can lead to errors in the hardware design, which are much more difficult to fix compared to bugs found in software [4].

Some of the other indicators for RISC or CISC are the number of addressing modes and format of the instruction words themselves. In general, using fewer addressing modes along with a consistent instruction format results in faster and less complex control signal logic [5]. Additionally, a study in [6] indicates that within the address calculation logic alone, there can be up to a 4× increase in structural complexity for CISC processors compared to RISC.

The reasoning behind CPU design choices have been changing throughout the past few decades. In the past, hardware complexity, chip area, and transistor count were some of the primary design considerations. In recent years, however, the focus has switched to minimizing energy and power while increasing speed. A study in [7] found that there is a similar overall performance between comparable RISC and CISC architectures, although the CISCs generally require more power.

There are many design choices involved in the development of a CPU aimed solely towards the hardware performance. However, for software to run on the CPU there are additional considerations to be made. Some of these considerations include the number of register classes, which types of addressing modes to implement, and the layout of the

memory space.

2.2 Compiler Design

In its simplest definition, a compiler accepts a program written in some source language, then translates it into a program with equivalent functionality in a target language [8]. While there are different variations of the compiling process (*e.g.* interpreters and just-in-time (JIT) compilers), this paper focuses on standard compilers, specifically ones that can accept an input program written in the C language, then output either the assembly or machine code of a target architecture. When considering C as the source language, two compiler suites are genuinely considered to be mature and optimized enough to handle modern software problems: GCC (the GNU Compiler Collection) and LLVM. Although similar in end-user functionality, GCC and LLVM each operate differently from each other both in their software architecture and even philosophy as organizations.

2.2.1 Application Binary Interface

Before considering the compiler, the application binary interface (ABI) must be defined for the target. This covers all of the details about how code and data interact with the CPU hardware. Some of the important design choices that need to be made include the alignment of different datatypes in memory, defining register classes (which registers can store which datatypes), and function calling conventions (whether function operands are placed on the stack, in registers, or a combination of both) [9]. The ABI must carefully consider the CPU architecture to be sure that each of the design choices are physically possible, and that they make efficient use of the CPU hardware when there are multiple solutions to a problem.

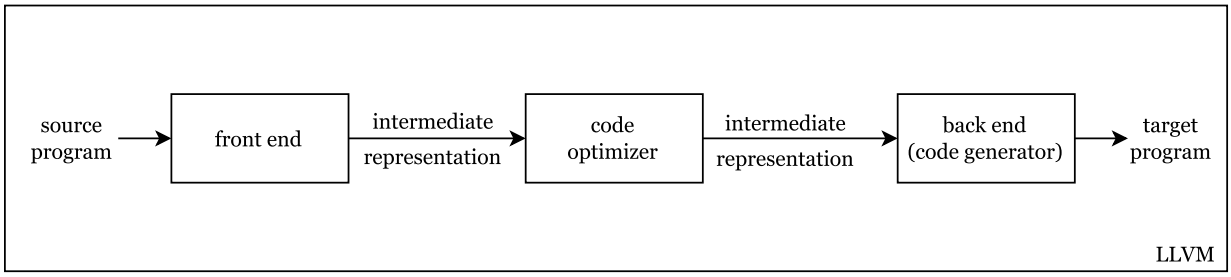

Figure 2.1: Aho Ullman Model

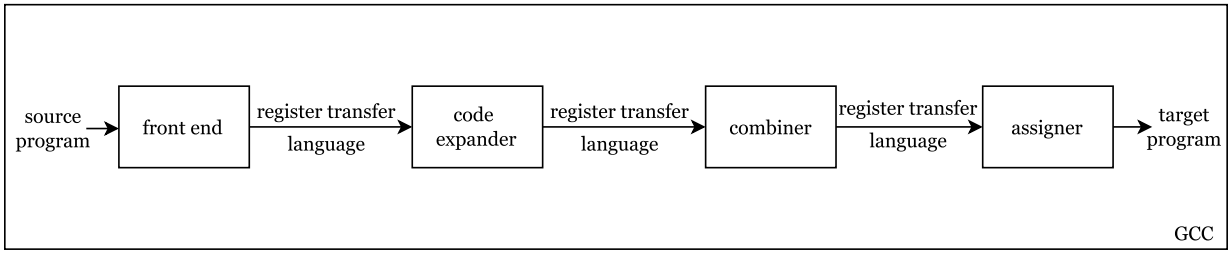

Figure 2.2: Davidson Fraser Model

2.2.2 Compiler Models

Modern compilers usually operate in three main phases: the front end, the optimizer, and the backend. Two approaches on how compilers should accomplish this task are the Aho Ullman approach [8] and the Davidson Fraser approach [10]. The block diagrams for each for each of these models are shown in Fig. 2.1 and Fig. 2.2. Although the function of the front end is similar between these models, there are some major differences in how they perform the process of optimization and code generation.

The Aho Ullman model places a large focus on having a target-independent intermediate representation (IR) language for a bulk of the optimization before the backend which allows the instruction selection process to use a cost-based approach. The Davidson Fraser model focuses on transforming the IR into a type of target-independent register transfer language (RTL).[1] The RTL then undergoes an expansion process followed by a recognizer which

[1] Register transfer language (RTL) is not to be confused with the register transfer level (RTL) design abstraction used in digital logic design

selects the instructions based on the expanded representation [9]. This paper will focus on the Aho Ullman model as LLVM is architected using this methodology.

Each phase of an Aho Ullman modeled compiler is responsible for translating the input program into a different representation, which brings the program closer to the target language. There is an extreme benefit of having a compiler architected using this model; because of the modularity and the defined boundaries of each stage, new source languages, target architectures, and optimization passes can be added or modified mostly independent of each other. A new source language implementation only needs to consider the design of the front end such that the output conforms to the IR, optimization passes are largely language-agnostic so long as they only operate on IR and preserve the program function, and lastly, generating code for a new target architecture only requires designing a backend that accepts IR and outputs the target code (typically assembly or machine code).

2.2.3 GCC

GCC was first released in 1984 by Richard M. Stallman [11]. GCC is written entirely in C and currently still maintains much of the same software architecture that existed in the initial release over 30 years ago. Regardless of this fact, almost every standard CPU has a port of GCC that is able to target it. Even architectures that do not have a backend in the GCC source tree typically have either a private release or custom build maintained by a third party; an example of one such architecture is the Texas Instruments MSP430 [12]. Although GCC is a popular compiler option, this paper focuses on LLVM instead for its significantly more modern code base.

2.2.4 LLVM

LLVM was originally released in 2003 by Chris Lattner [13] as a master's thesis project. The compiler has since grown tremendously into an fully complete and open-source compiler infrastructure. Written in C++ and embracing its object-oriented programming nature, LLVM has now become a rich set of compiler-based tools and libraries. While LLVM used to be an acronym for "low level virtual machine," representing its rich, virtual instruction set IR language, the project has grown to encompass a larger scope of projects and goals and LLVM no longer stands for anything [14]. There are a much fewer number of architectures that are supported in LLVM compared to GCC because it is so new. Despite this fact, there are still organizations choosing to use LLVM as the default compiler toolchain over GCC [15, 16]. The remainder of this section describes the three main phases of the LLVM compiler.

2.2.4.1 Front End

The front end is responsible for translating the input program from text written by a person. This stage is done through lexical, syntactical, and semantic analysis. The output format of the front end is the LLVM IR code. The IR is a fully complete virtual instruction set which has operations similar to RISC architectures; however, it is fully typed, uses Static Single Assignment (SSA) representation, and has an unlimited number of virtual registers. It is low-level enough such that it can be easily related to hardware operations, but it also includes enough high-level control-flow and data information to allow for sophisticated analysis and optimization [17]. All of these features of LLVM IR allow for a very efficient, machine-independent optimizer.

2.2.4.2 Optimization

The optimizer is responsible for translating the IR from the output of the front end, to an equivalent yet optimized program in IR. Although this phase is where the bulk of the optimizations are completed; optimizations can, and should be completed at each phase of the compilation. Users can optimize code when writing it before it even reaches the front end, and the backend can optimize code specifically for the target architecture and hardware.

In general, there are two main goals of the optimization phase: to increase the execution speed of the target program, and to reduce the code size of the target program. To achieve these goals, optimizations are usually performed in multiple passes over the IR where each pass has specific goal of smaller-scope. One simple way of organizing the IR to aid in optimization is through SSA form. This form guarantees that each variable is defined exactly once which simplifies many optimizations such as dead code elimination, edge elimination, loop construction, and many more [13].

2.2.4.3 Backend

The backend is responsible for translating a program from IR into target-specific code (usually assembly or machine code). For this reason, this phase is also commonly referred to as the code generator. The most difficult problems that are solved in this phase are instruction selection and register allocation.

Instruction selection is responsible for transforming the operations specified by the IR into instructions that are available on the target architecture. For a simple example, consider a program in IR containing a logical NOT operation. If the target architecture does not have a logical NOT instruction but it does contain a logical XOR function, the instruction selector would be responsible for converting the “NOT” operation into an “XOR

with -1" operation, as they are functionally equivalent.

Register allocation is an entirely different problem as the IR uses an unlimited number of variables, not a fixed number of registers. The register allocator assigns variables in the IR to registers in the target architecture. The compiler requires information about any special purpose registers along with different register classes that may exist in the target. Other issues such as instruction ordering, memory allocation, and relative address resolution are also solved in this phase. Once all of these problems are solved the backend can emit the final target-specific assembly or machine code.

Chapter 3

Custom RISC CPU Design

This chapter discusses the design and architecture of the custom CJG RISC CPU. Section 3.1 explains the design choices made, section 3.2 describes the implementation of the architecture, and section 3.3 describes all of the instructions in detail.

3.1 Instruction Set Architecture

The first stage in designing the CJG RISC was to specify its instruction set architecture (ISA). The ISA was designed to be simple enough to implement in hardware and describe for LLVM, while still including enough instructions and features such that it could execute sophisticated programs. The architecture is a 32-bit data path, register-register design. Each operand is 32-bits wide and all data manipulation instructions can only operate on operands that are located in the register file.

3.1.1 Register File

The register file is composed of 32 individual 32-bit registers denoted as `r0` through `r31`. All of the registers are general purpose with the exception of `r0-r2`, which are designated as special purpose registers.

The first special purpose register is the status register (`SR`), which is stored in `r0`. The status register contains the condition bits that are automatically set by the CPU following a manipulation instruction. The conditions bits set represent when an arithmetic operation results in any of the following: a carry, a negative result, an overflow, or a result that is zero. The status register bits can be seen in Fig. 3.1. A table describing the status register bits can be seen in Table 3.1.

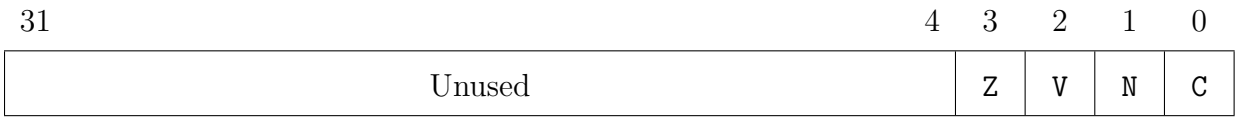

Figure 3.1: Status Register Bits

Bit	Description
`C`	The carry bit. This is set to 1 if the result of a manipulation instruction produced a carry and set to 0 otherwise
`N`	The negative bit. This is set to 1 when the result of a manipulation instruction produces a negative number (set to bit 31 of the result) and set to 0 otherwise
`V`	The overflow bit. This is set to 1 when a arithmetic operation results in an overflow (*e.g.* when a positive + positive results in a negative) and set to 0 otherwise
`Z`	The zero bit. This is set to 1 when the result of a manipulation instruction produces a result that is 0 and set to 0 otherwise

Table 3.1: Description of Status Register Bits

The next special purpose register is the program counter (`PC`) register, which is stored in `r1`. This register stores the current value of the program counter which is the address

of the current instruction word in memory. This register is write protected and cannot be overwritten by any manipulation instructions. The `PC` can only be changed by an increment during instruction fetch (see section 3.2.1.1) or a flow control instruction (see section 3.3.3). The `PC` bits can be seen in Fig. 3.2.

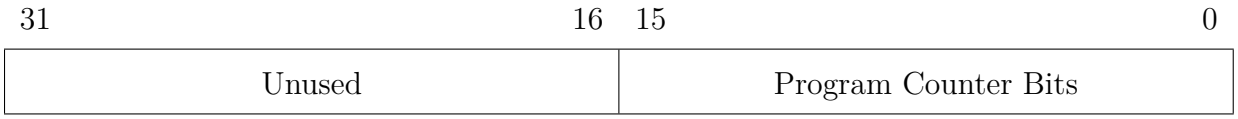

Figure 3.2: Program Counter Bits

The final special purpose register is the stack pointer (`SP`) register, which is stored in `r2`. This register stores the address pointing to the top of the data stack. The stack pointer is automatically incremented or decremented when values are pushed on or popped off the stack. The `SR` bits can be seen in Fig. 3.3.

Figure 3.3: Stack Pointer Register

3.1.2 Stack Design

There are two hardware stacks in the CJG RISC design. One stack is used for storing the `PC` and `SR` throughout calls and returns (the call stack). The other stack is used for storing variables (the data stack). Most CPUs utilize a data stack that is located within the data memory space, however, a hardware stack was used to simplify the implementation. Both stacks are 64 words deep, however they operate slightly differently. The call stack does not have an external stack pointer. The data is pushed on and popped off the stack using

internal control signals. The data stack, however, makes use of the `SP` register to access its contents acting similar to a memory structure.

During the call instruction the `PC` and then the `SR` are pushed onto the call stack. During the return instruction they are popped back into their respective registers.

The data stack is managed by push and pop instructions. The push instruction pushes a value onto the stack at the location of the `SP`, then automatically increments the stack pointer. The pop instruction first decrements the stack pointer, then pops the value at location of the decremented stack pointer into its destination register. These instructions are described further in Section 3.3.2.

3.1.3 Memory Architecture

There are two main memory design architectures used when designing CPUs: Harvard and von Neumann. Harvard makes use of two separate physical datapaths for accessing data and instruction memory. Von Neumann only utilizes a single datapath for accessing both data and instruction memory. Without the use of memory caching, traditional von Neumann architectures cannot access both instruction and data memory in parallel. The Harvard architecture was chosen to simplify implementation and avoid the need to stall the CPU during data memory accesses. Additionally, the Harvard architecture offers complete protection against conventional memory attacks (*e.g.* buffer/stack overflowing) as opposed to a more complex von Neumann architecture [18]. No data or instruction caches were implemented to keep memory complexity low.

Both memories are byte addressable with a 32-bit data bus and a 16-bit wide address bus. The upper 128 addresses of data memory are reserved for memory mapped input/output (I/O) peripherals.

3.2 Hardware Implementation

The CJG RISC is fully designed in the Verilog hardware description language (HDL) at the register transfer level (RTL). The CPU is implemented as a four-stage pipeline and the main components are the clock generator, register file, arithmetic logic unit (ALU), the shifter, and the two stacks. A simplified functional block diagram of the CPU can be seen in Fig. 3.4.

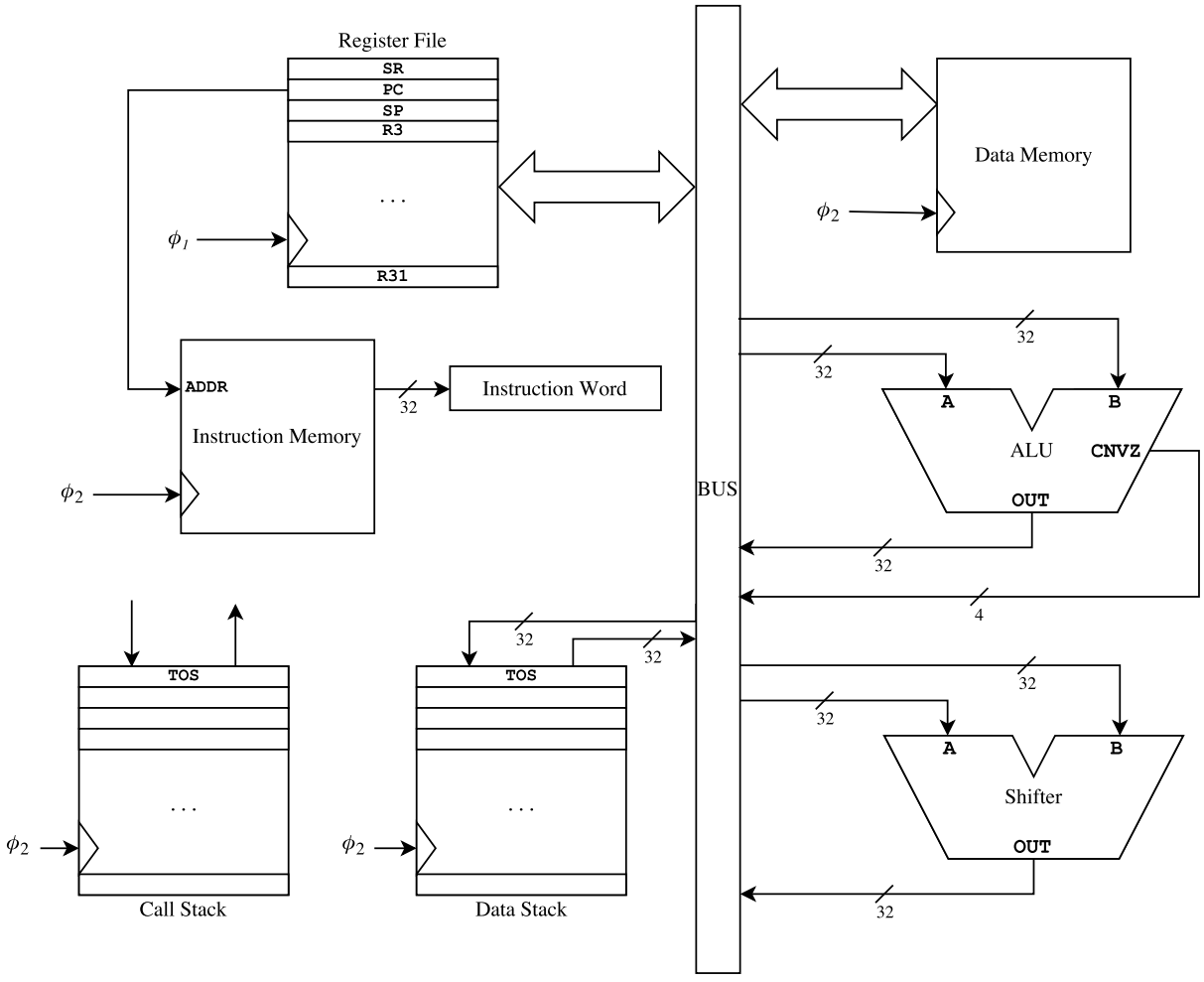

Figure 3.4: CJG RISC CPU Functional Block Diagram

Pipeline Stage	**Pipeline**						
IF	I_0	I_1	I_2	I_3	I_4	I_5	...
OF		I_0	I_1	I_2	I_3	I_4	...
EX			I_0	I_1	I_2	I_3	...
WB				I_0	I_1	I_2	...
Clock Cycle	**1**	**2**	**3**	**4**	**5**	**6**	**...**

Figure 3.5: Four-Stage Pipeline

Figure 3.6: Four-Stage Pipeline Block Diagram

3.2.1 Pipeline Design

The pipeline is a standard four-stage pipeline with instruction fetch (IF), operand fetch (OF), execute (EX), and write back (WB) stages. This pipeline structure can be seen in Fig. 3.5 where I_n represents a single instruction propagating through the pipeline. Additionally, a block diagram of the pipeline can be seen in Fig. 3.6. During clock cycles 1-3 the pipeline fills up with instructions and is not at maximum efficiency. For clock cycles 4 and onwards, the pipeline is fully filled and is effectively executing instructions at a rate of 1 *IPC* (instruction per clock cycle). The CPU will continue executing instructions at a rate of 1 *IPC* until a jump or a call instruction is encountered at which point the CPU will stall.

3.2.1.1 Instruction Fetch

Instruction fetch is the first machine cycle of the pipeline. Instruction fetch has the least logic of any stage and is the same for every instruction. This stage is responsible for loading the next instruction word from instruction memory, incrementing the program counter so it points at the next instruction word, and stalling the processor if a call or jump instruction

is encountered.

3.2.1.2 Operand Fetch

Operand fetch is the second machine cycle of the pipeline. This stage contains the most logic out of any of the pipeline stages due to the data forwarding logic implemented to resolve data dependency hazards. For example, consider an instruction, I_n, that modifies the $\texttt{R}_x$ register, followed by an instruction I_{n+1}, that uses $\texttt{R}_x$ as an operand.[1] Without any data forwarding logic, I_{n+1} would not fetch the correct value because I_n would still be in the execute stage of the pipeline, and $\texttt{R}_x$ would not be updated with the correct value until I_n completes write back. The data forwarding logic resolves this hazard by fetching the value at the output of the execute stage instead of from `Rx`. Data dependency hazards can also arise from less-common situations such as an instruction modifying the `SP` followed by a stack instruction. Because the stack instruction needs to modify the stack pointer, this would have to be forwarded as well.

An alternative approach to solving these data dependency hazards would be to stall CPU execution until the write back of the required operand has finished. This is a trade-off between an increase in stall cycles versus an increase in data forwarding logic complexity. Data forwarding logic was implemented to minimize the stall cycles, however, no in-depth efficiency analysis was calculated for this design choice.

3.2.1.3 Execute

Execution is the third machine cycle of the pipeline and is mainly responsible for three functions. The first is preparing any data in either the ALU or shifter module for the write back stage. The second is to handle reading the output of the memory for data. The third

[1] $\texttt{R}_x$ represents any modifiable general purpose register

function is to handle any data that was popped off of the stack, along with adjusting the stack pointer.

3.2.1.4 Write Back

The write back stage is the fourth and final machine cycle of the pipeline. This stage is responsible for writing any data from the execute stage back to the destination register. This stage additionally is responsible for handling the flow control logic for conditional jump instructions as well as calls and returns (as explained in Section 3.3.3).

3.2.2 Stalling

The CPU only stalls when a jump or call instruction is encountered. When the CPU stalls the pipeline is emptied of its current instructions and then the `PC` is set to the destination location of either the jump of the call. Once the CPU successfully jumps or calls to the new location the pipeline will begin filling again.

3.2.3 Clock Phases

The CPU contains a clock generator module which generates two clock phases, ϕ_1 and ϕ_2 (shown in Fig. 3.7), from the main system clock. The ϕ_1 clock is responsible for all of the pipeline logic while ϕ_2 acts as the memory clock for both the instruction and data memory. Additionally, the ϕ_2 clock is used for both the call and data stacks.

3.3 Instruction Details

This section lists all of the instructions, shows the significance of the instruction word bits, and describes other specific details pertaining to each instruction.

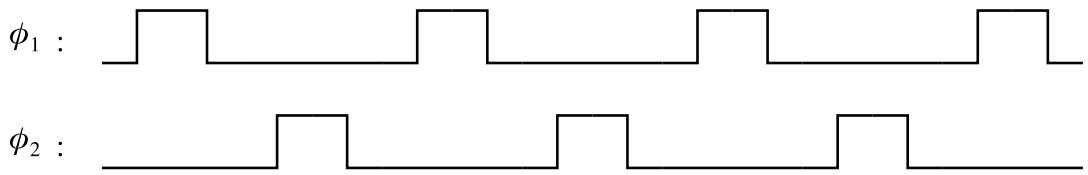

Figure 3.7: Clock Phases

3.3.1 Load and Store

Load and store instructions are responsible for transferring data between the data memory and the register file. The instruction word encoding is shown in Fig. 3.8.

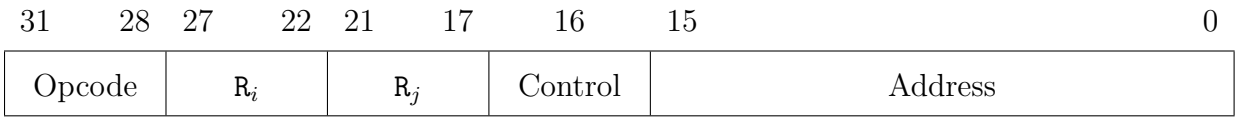

Figure 3.8: Load and Store Instruction Word

There are four different addressing modes that the CPU can utilize to access a particular memory location. These addressing modes along with how they are selected are described in Table 3.2 where R_x corresponds to the R_j register in the load and store instruction word. The load and store instruction details are described in Table 3.3.

Mode	R_x [2]	**Control**	**Effective Address Value**
Register Direct	Not 0	1	The value of the R_x register operand
Absolute	0	1	The value in the address field
Indexed	Not 0	0	The value of the R_x register operand + the value in the address field
PC Relative	0	0	The value of the PC register + the value in the address field

Table 3.2: Addressing Mode Descriptions

[2] R_x corresponds to R_j for load and store instructions, and to R_i for flow control instructions

Instruction	Mnemonic	Opcode	Function
Load	LD	0x0	Load the value in memory at the effective address or I/O peripheral into the R_i register
Store	ST	0x1	Store the value of the R_i register into memory at the effective address or I/O peripheral

Table 3.3: Load and Store Instruction Details

3.3.2 Data Transfer

Data instructions are responsible for moving data between the register file, instruction word field, and the stack. The instruction word encoding is shown in Fig. 3.9.

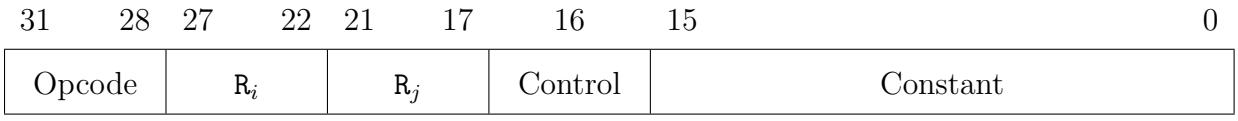

Figure 3.9: Data Transfer Instruction Word

The data transfer instruction details are described in Table 3.4. If the control bit is set high then the source operand for the copy and push instructions is taken from the 16-bit constant field and sign extended, otherwise the source operand is the register denoted by R_j.

Instruction	Mnemonic	Opcode	Function
Copy	CPY	0x2	Copy the value from the source operand into the R_i register
Push	PUSH	0x3	Push the value from the source operand onto the top of the stack and then increment the stack pointer
Pop	POP	0x4	Decrement the stack pointer and then pop the value from the top of the stack into the R_i register.

Table 3.4: Data Transfer Instruction Details

3.3.3 Flow Control

Flow control instructions are responsible for adjusting the sequence of instructions that are executed by the CPU. This allows a non-linear sequence of instructions that can be decided by the result of previous instructions. The purpose of the jump instruction is to conditionally move to different locations in the instruction memory. This allows for decision making in the program flow, which is one of the requirements for a computing machine to be Turing-complete [19]. The instruction word encoding is shown in Fig. 3.10.

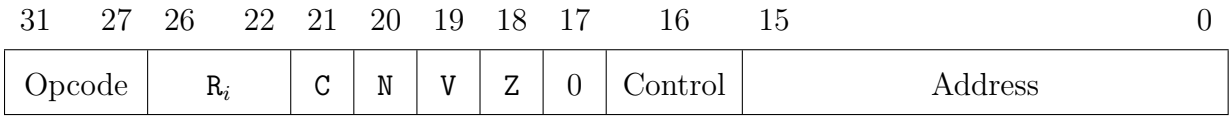

Figure 3.10: Flow Control Instruction Word

The CPU utilizes four distinct addressing modes to calculate the effective destination address similar to load and store instructions. These addressing modes along with how they are selected are described in Table 3.2, where R_x corresponds to the R_i register in the flow control instruction word. An additional layer of control is added in the C, N, V, and Z bit fields located at bits 21-18 in the instruction word. These bits only affect the jump instruction and are described in Table 3.5. The C, N, V, and Z columns in this table correspond to the value of the bits in the flow control instruction word and *not* the value of bits in the status register. However, in the logic to decide whether to jump (in the write back machine cycle), the actual value of the bit in the status register (corresponding to the one selected by the condition code) is used. The flow control instruction details are described in Table 3.6.

C	N	V	Z	Mnemonic	Description
0	0	0	0	JMP / JU	Jump unconditionally
1	0	0	0	JC	Jump if carry
0	1	0	0	JN	Jump if negative
0	0	1	0	JV	Jump if overflow
0	0	0	1	JZ / JEQ	Jump if zero / equal
0	1	1	1	JNC	Jump if not carry
1	0	1	1	JNN	Jump if not negative
1	1	0	1	JNV	Jump if not overflow
1	1	1	0	JNZ / JNE	Jump if not zero / not equal

Table 3.5: Jump Condition Code Description

Instruction	Mnemonic	Opcode	Function
Jump	J{CC}[3]	0x5	Conditionally set the PC to the effective address
Call	CALL	0x6	Push the PC followed by the SR onto the call stack, set the PC to the effective address
Return	RET	0x7	Pop the top of call stack into the SR, then pop the next value into the PC

Table 3.6: Flow Control Instruction Details

3.3.4 Manipulation Instructions

Manipulation instructions are responsible for the manipulation of data within the register file. Most of the manipulation instructions require three operands: one destination and two source operands. Any manipulation instruction that requires two source operands can either use the value in a register or an immediate value located in the instruction word as the second source operand. The instruction word encoding for these variants are shown in Fig. 3.11 and 3.12, respectively. All of the manipulation instructions have the possibility of changing the condition bits in the SR following their operation, and they all are calculated

[3] The value of {CC} depends on the condition code; see the Mnemonic column in Table 3.5

through the ALU.

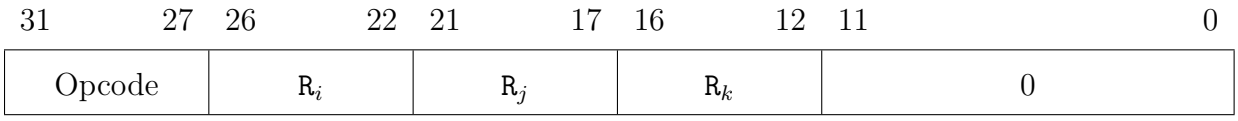

Figure 3.11: Register-Register Manipulation Instruction Word

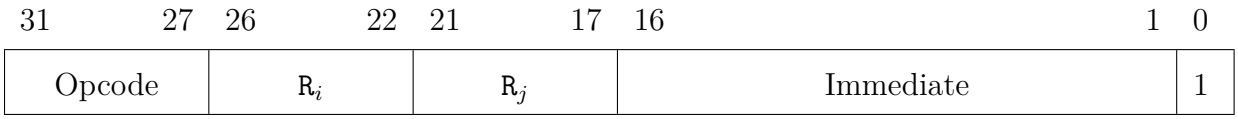

Figure 3.12: Register-Immediate Manipulation Instruction Word

Instruction	**Mnemonic**	**Opcode**	**Function**
Add	`ADD`	`0x8`	Store $R_j + SRC_2$ in R_i
Subtract	`SUB`	`0x9`	Store $R_j - SRC_2$ in R_i
Compare	`CMP`	`0xA`	Compute $R_j - SRC_2$ and discard result
Negate	`NOT`	`0xB`	Store $\sim R_j$ in R_i[4]
AND	`AND`	`0xC`	Store R_j `&` SRC_2 in R_i[5]
Bit Clear	`BIC`	`0xD`	Store R_j `&` $\sim SRC_2$ in R_i
OR	`OR`	`0xE`	Store R_j `|` SRC_2 in R_i[6]
Exclusive OR	`XOR`	`0xF`	Store R_j `^` SRC_2 in R_i[7]
Signed Multiplication	`MUL`	`0x1A`	Store $R_j \times SRC_2$ in R_i
Unsigned Division	`DIV`	`0x1B`	Store $R_j \div SRC_2$ in R_i

Table 3.7: Manipulation Instruction Details

The manipulation instruction details are described in Table 3.7. The value of SRC_2 either represents the R_k register for a register-register manipulation instruction or the immediate value (sign-extended to 32-bits) for a register-immediate manipulation instruction.

[4] The ~ symbol represents the unary logical negation operator
[5] The `&` symbol represents the logical `AND` operator
[6] The `|` symbol represents the logical inclusive `OR` operator
[7] The `^` symbol represents the logical exclusive `OR` (`XOR`) operator

3.3.4.1 Shift and Rotate

Shift and Rotate instructions are a specialized case of manipulation instructions. They are calculated through the shifter module, and the rotate-through-carry instructions have the possibility of changing the `C` bit within the `SR`. The logical shift shifts will always shift in bits with the value of 0 and discard the bits shifted out. Arithmetic shift will shift in bits with the same value as the most significant bit in the source operand as to preserve the correct sign of the data. As with the other manipulation instructions, these instructions can either use the contents of a register or an immediate value from the instruction word for the second source operand. The instruction word encoding for these variants are shown in Fig. 3.13 and 3.14, respectively.

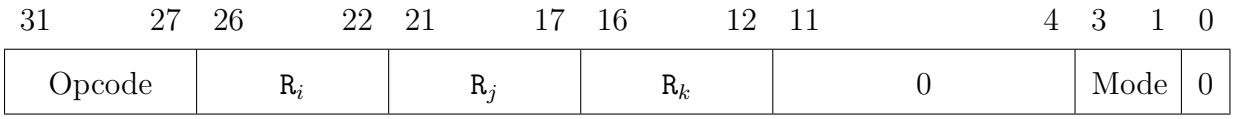

Figure 3.13: Register-Register Shift and Rotate Instruction Word

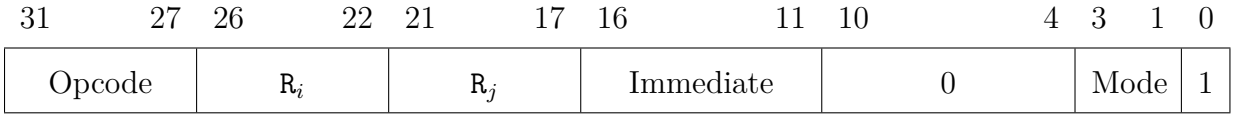

Figure 3.14: Register-Immediate Manipulation Instruction Word

The mode field in the shift and rotate instructions select which type of shift or rotate to perform. All instructions will perform the operation as defined by the mode field on the R_j register as the source data. The number of bits that the data will be shifter or rotated (SRC_2) is determined by either the value in the R_k register or the immediate value in the instruction word depending on if it is a register-register or register-immediate instruction word. The shift and rotate instruction details are described in Table 3.8.

Instruction	Mnemonic	Opcode	Mode	Function
Shift right logical	SRL	0x10	0x0	Shift R_j right logically by SRC_2 bits and store in R_i
Shift left logical	SLL	0x10	0x1	Shift R_j left logically by SRC_2 bits and store in R_i
Shift right arithmetic	SRA	0x10	0x2	Shift R_j right arithmetically by SRC_2 bits and store in R_i
Rotate right	RTR	0x10	0x4	Rotate R_j right by SRC_2 bits and store in R_i
Rotate left	RTL	0x10	0x5	Rotate R_j left by SRC_2 bits and store in R_i
Rotate right through carry	RRC	0x10	0x6	Rotate R_j right through carry by SRC_2 bits and store in R_i
Rotate left through carry	RLC	0x10	0x7	Rotate R_j left through carry by SRC_2 bits and store in R_i

Table 3.8: Shift and Rotate Instruction Details

Chapter 4

Custom LLVM Backend Design

This chapter discusses the structure and design of the custom target-specific LLVM backend. Section 4.1 discusses the high-level structure of LLVM and Section 4.2 describes the specific implementation of the custom backend.

4.1 Structure and Tools

LLVM is different from most traditional compiler projects because it is not just a collection of individual programs, but rather a collection of libraries. These libraries are all designed using object-oriented programming and are extendable and modular. This along with its three-phase approach (discussed in Section 2.2.4) and its modern code design makes it a very appealing compiler infrastructure to work with. This chapter presents a custom LLVM backend to target the custom CJG RISC CPU, which is explained in detail in Chapter 3.

4.1.1 Code Generator Design Overview

The code generator is one of the many large frameworks that is available within LLVM. This particular framework provides many classes, methods, and tools to help translate the LLVM IR code into target-specific assembly or machine code [20]. Most of the code base, classes, and algorithms are target-independent and can be used by all of the specific backends that are implemented. The two main target-specific components that comprise a custom backend are the abstract target description, and the abstract target description implementation. These target-specific components of the framework are necessary for every target-architecture in LLVM and the code generator uses them as needed throughout the code generation process.

The code generator is separated into several stages. Prior to the instruction scheduling stage, the code is organized into basic blocks, where each basic block is represented as a directed acyclic graph (DAG). A basic block is defined as a consecutive sequence of statements that are operated on, in order, from the beginning of the basic block to the end without having any possibility of branching, except for at the end [8]. DAGs can be very useful data structures for operating on basic blocks because they provide an easy means to determine which values used in a basic block are used in any subsequent operations. Any value that has the possibility of being used in a subsequent operation, even in a different basic block, is said to be a *live* value. Once a value no longer has a possibility of being used it is said to be a *killed* value.

The high-level descriptions of the stages which comprise the code generator are as follows:

1. **Instruction Selection** — Translates the LLVM IR into operations that can be performed in the target's instruction set. Virtual registers in SSA form are used to

represent the data assignments. The output of this stage are DAGs containing the target-specific instructions.

2. **Instruction Scheduling** — Determines the necessary order of the target machine instructions from the DAG. Once this order is determined the DAG is converted to a list of machine instructions and the DAG is destroyed.

3. **Machine Instruction Optimization** — Performs target-specific optimizations on the machine instructions list that can further improve code quality.

4. **Register Allocation** — Maps the current program, which can use any number of virtual registers, to one that only uses the registers available in the target-architecture. This stage also takes into account different register classes and the calling convention as defined in the ABI.

5. **Prolog and Epilog Code Insertion** — Typically inserts the code pertaining to setting up (prolog) and then destroying (epilog) the stack frame for each basic block.

6. **Final Machine Code Optimization** — Performs any final target-specific optimizations that are defined by the backend.

7. **Code Emission** — Lowers the code from the machine instruction abstractions provided by the code generator framework into target-specific assembly or machine code. The output of this stage is typically either an assembly text file or extendable and linkable format (ELF) object file.

4.1.2 TableGen

One of the LLVM tools that is necessary for writing the abstract target description is TableGen (`llvm-tblgen`). This tool translates a target description file (`.td`) into C++

code that is used in code generation. It's main goal is to reduce large, tedious descriptions into smaller and flexible definitions that are easier to manage and structure [21]. The core functionality of TableGen is located in the TableGen backends.[1] These backends are responsible for translating the target description files into a format that can be used by the code generator [22]. The code generator provides all of the TableGen backends that are necessary for most CPUs to complete their abstract target description, however, custom TableGen backends can be written for other purposes.

The same TableGen input code can typically produces a different output depending on the TableGen backend used. The TableGen code shown in Listing 4.1 is used to define each of the CPU registers that are in the CJG architecture. The AsmWriter TableGen backend, which is responsible for creating code to help with printing the target-specific assembly code, generates the C++ code seen in Listing 4.2. However, the RegisterInfo TableGen backend, which is responsible for creating code to help with describing the register file to the code generator, generates the C++ code seen in Listing 4.3.

There are many large tables (such as the one seen on line 7 of Listing 4.2) and functions that are generated from TableGen to help in the design of the custom LLVM backend. Although TableGen is currently responsible for a bulk of the target description, a large amount of C++ code still needs to be written to complete the abstract target description implementation. As the development of LLVM moves forward, the goal is to move as much of the target description as possible into TableGen form [20].

[1] Not to be confused with LLVM backends (target-specific code generators)

```
// Special purpose registers
def SR : CJGReg<0, "r0">;
def PC : CJGReg<1, "r1">;
def SP : CJGReg<2, "r2">;

// General purpose registers
foreach i = 3-31 in {
  def R#i : CJGReg< #i, "r"#i>;
}
```

Listing 4.1: TableGen Register Set Definitions

```
/// getRegisterName - This method is automatically generated by tblgen
/// from the register set description.  This returns the assembler name
/// for the specified register.
const char *CJGInstPrinter::getRegisterName(unsigned RegNo) {
  assert(RegNo && RegNo < 33 && "Invalid register number!");

  static const char AsmStrs[] = {
  /* 0 */ 'r', '1', '0', 0,
  /* 4 */ 'r', '2', '0', 0,
  ...
  };
...
}
```

Listing 4.2: TableGen AsmWriter Output

```
namespace CJG {
enum {
  NoRegister,
  PC = 1,
  SP = 2,
  SR = 3,
  R3 = 4,
  R4 = 5,
  ...
  };
} // end namespace CJG
```

Listing 4.3: TableGen RegisterInfo Output

4.1.3 Clang and llc

Clang is the front end for LLVM which supports C, C++, and Objective C/C++ [23]. Clang is responsible for the functionality discussed in Section 2.2.4.1. The llc tool is the LLVM static compiler which is responsible for the functionality discussed in Section 2.2.4.3. The custom backends written for LLVM are each linked into llc which then compiles LLVM IR code into the target-specific assembly or machine code.

4.2 Custom Target Implementation

The custom LLVM backend inherits from and extends many of the LLVM classes. To implement an LLVM backend, most of the files are placed within LLVM's `lib/Target/TargetName/` directory, where `TargetName` is the name of the target architecture as referenced by LLVM. This name is important and must stay consistent throughout the entirety of the backend development as it is used by LLVM internals to find the custom backend. The name for this target architecture was chosen as CJG, therefore, the custom backend is located in `lib/Target/CJG/`. The "entry point" for CJG LLVM backend is within the `CJGMCTargetDescription`. This is where the backend is registered with the LLVM `TargetRegistry` so that LLVM can find and use the backend. The graph shown in Fig. 4.1 gives a clear picture of the classes and files that are a part of the CJG backend.

In addition to the RISC backends that are currently in the LLVM source tree (namely ARM and MSP430), several out-of-tree, work-in-progress backends were used as resources during the implementation of the CJG backend: Cpu0 [24], LEG [25], and RISC-V [26]. The remainder of this section will discuss the details of the implementation of the custom CJG LLVM backend.

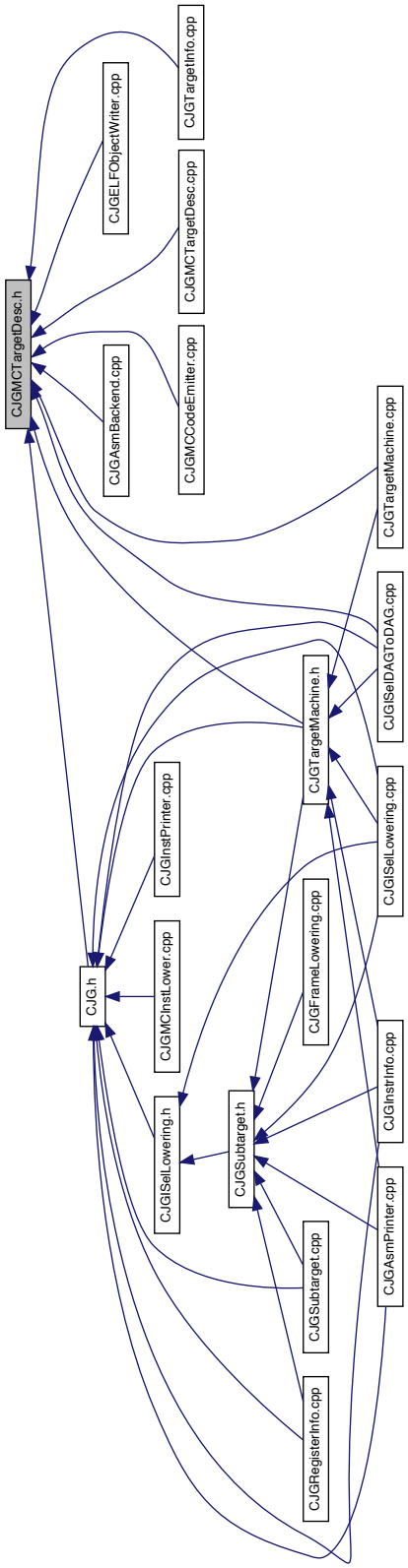

Figure 4.1: `CJGMCTargetDesc.h` Inclusion Graph

4.2.1 Abstract Target Description

As discussed in in Section 4.1.2, a majority of the abstract target description is written in TableGen format. The major components of the CJG backend written in TableGen form are the register information, calling convention, special operands, instruction formats, and the complete instruction definitions. In addition to the TableGen components, there are some details that must be written in C++. These components of the abstract target description are described in the following sections.

4.2.1.1 Register Information

The register information is found in `CJGRegisterInfo.td`. This file defines the register set of the CJG RISC as well as different register classes. This makes it easy to separate registers that may only be able to hold a specific datatype (*e.g.* integer vs. floating point register classes). Because the CJG architecture does not support floating point operations, the main register class is the general purpose register class. The definition of this class is shown in Listing 4.4. The definition of each individual register is also located in this file and is shown in Listing 4.1.

```
1 // General purpose registers class
2 def GPRegs : RegisterClass<"CJG", [i32], 32, (add
3         (sequence "R%u", 4, 31), SP, R3
4 )>;
```

Listing 4.4: General Purpose Registers Class Definition

4.2.1.2 Calling Conventions

The calling convention definitions describe the part of the ABI which controls how data moves between function calls. The calling convention definitions are defined in `CJGCallingConv.td` and the return calling convention definition is shown in Listing 4.5. This definition describes how values are returned from functions. Firstly, any 8-bit or 16-bit values must be converted to a 32-bit value. Then the first 8 return values are placed in registers `R24`–`R31`. Any remaining return values would be pushed onto the data stack.

```
1  //===----------------------------------------------------------------------===//
2  // CJG Return Value Calling Convention
3  //===----------------------------------------------------------------------===//
4  def RetCC_CJG : CallingConv<[
5    // Promote i8/i16 arguments to i32.
6    CCIfType<[i8, i16], CCPromoteToType<i32>>,
7
8    // i32 are returned in registers R24-R31
9    CCIfType<[i32], CCAssignToReg<[R24, R25, R26, R27, R28, R29, R30, R31]>>,
10
11   // Integer values get stored in stack slots that are 4 bytes in
12   // size and 4-byte aligned.
13   CCIfType<[i32], CCAssignToStack<4, 4>>
14 ]>;
```

Listing 4.5: Return Calling Convention Definition

4.2.1.3 Special Operands

There are several special types of operands that need to be defined as part of the target description. There are many operands that are pre-defined in TableGen such as `i16imm` and `i32imm` (defined in `include/llvm/Target/Target.td`), however, there are cases where

these are not sufficient. Two examples of special operands that need to be defined are the memory address operand and the jump condition code operand. Both of these operands need to be defined separately because they are not a standard datatype size both and need to have special methods for printing them in assembly. The custom `memsrc` operand holds both the register and immediate value for the indexed addressing mode (as shown in Table 3.2). These definitions are found in `CJGInstrInfo.td` and are shown in Listing 4.6. The `PrintMethod` and `EncoderMethod` define the names of custom C++ functions to be called when either *printing* the operand in assembly or *encoding* the operand in the machine code.

```
1  // Address operand for indexed addressing mode
2  def memsrc : Operand<i32> {
3    let PrintMethod = "printMemSrcOperand";
4    let EncoderMethod = "getMemSrcValue";
5    let MIOperandInfo = (ops GPRegs, CJGimm16);
6  }
7
8  // Operand for printing out a condition code.
9  def cc : Operand<i32> {
10   let PrintMethod = "printCCOperand";
11 }
```

Listing 4.6: Special Operand Definitions

4.2.1.4 Instruction Formats

The instruction formats describe the instruction word formats as per the formats described in Section 3.3 along with some other important properties. These formats are defined in `CJGInstrFormats.td`. The base class for all CJG instruction formats is shown in Listing 4.7. This is then expanded into several other classes for each type of instruction. For

example, the ALU instruction format definitions for both register-register and register-immediate modes are shown in Listing 4.8.

```
//===----------------------------------------------------------------------===//
// Instruction format superclass
//===----------------------------------------------------------------------===//
class InstCJG<dag outs, dag ins, string asmstr, list<dag> pattern>
    : Instruction {
  field bits<32> Inst;

  let Namespace = "CJG";
  dag OutOperandList = outs;
  dag InOperandList = ins;
  let AsmString   = asmstr;
  let Pattern = pattern;
  let Size = 4;

  // define Opcode in base class because all instrutions have the same
  // bit-size and bit-location for the Opcode
  bits<5> Opcode = 0;
  let Inst{31-27} = Opcode; // set upper 5 bits to opcode
}

// CJG pseudo instructions format
class CJGPseudoInst<dag outs, dag ins, string asmstr, list<dag> pattern>
    : InstCJG<outs, ins, asmstr, pattern> {
  let isPseudo = 1;
  let isCodeGenOnly = 1;
}
```

Listing 4.7: Base CJG Instruction Definition

4.2.1.5 Complete Instruction Definitions

The complete instruction definitions inherit from the instruction format classes to complete the TableGen `Instruction` base class. These complete instructions are defined in `CJGInstrInfo.td`. Some of the ALU instruction definitions are shown in Listing 4.9. The multiclass functionality makes it easier to define multiple instructions that are very similar

```
//===----------------------------------------------------------------------===//
// ALU Instructions
//===----------------------------------------------------------------------===//

// ALU register-register instruction
class ALU_Inst_RR<bits<5> opcode, dag outs, dag ins, string asmstr,
                  list<dag> pattern>
    : InstCJG<outs, ins, asmstr, pattern> {

  bits<5> ri; // destination register
  bits<5> rj; // source 1 register
  bits<5> rk; // source 2 register

  let Opcode = opcode;
  let Inst{26-22} = ri;
  let Inst{21-17} = rj;
  let Inst{16-12} = rk;
  let Inst{11-1} = 0;
  let Inst{0} = 0b0; // control-bit for immediate mode
}

// ALU register-immediate instruction
class ALU_Inst_RI<bits<5> opcode, dag outs, dag ins, string asmstr,
                  list<dag> pattern>
    : InstCJG<outs, ins, asmstr, pattern> {

  bits<5> ri; // destination register
  bits<5> rj; // source 1 register
  bits<16> const; // constant/immediate value

  let Opcode = opcode;
  let Inst{26-22} = ri;
  let Inst{21-17} = rj;
  let Inst{16-1} = const;
  let Inst{0} = 0b1; // control-bit for immediate mode
}
```

Listing 4.8: Base ALU Instruction Format Definitions

to each other. In this case the register-register (`rr`) and register-immediate (`ri`) ALU instructions are defined within the multiclass. When the `defm` keyword is used, all of the

classes within the multiclass are defined (*e.g.* the definition of the `ADD` instruction on line 23 of Listing 4.9 is expanded into an `ADDrr` and `ADDri` instruction definition).

```
1  //===----------------------------------------------------------------------===//
2  // ALU Instructions
3  //===----------------------------------------------------------------------===//
4
5  let Defs = [SR] in {
6  multiclass ALU<bits<5> opcode, string opstr, SDNode opnode> {
7
8    def rr : ALU_Inst_RR<opcode, (outs GPRegs:$ri),
9                         (ins GPRegs:$rj, GPRegs:$rk),
10                        !strconcat(opstr, "\t$ri, $rj, $rk"),
11                        [(set GPRegs:$ri, (opnode GPRegs:$rj, GPRegs:$rk)),
12                         (implicit SR)]> {
13   }
14
15   def ri : ALU_Inst_RI<opcode, (outs GPRegs:$ri),
16                        (ins GPRegs:$rj, CJGimm16:$const),
17                        !strconcat(opstr, "\t$ri, $rj, $const"),
18                        [(set GPRegs:$ri, (opnode GPRegs:$rj, CJGimm16:$const)),
19                         (implicit SR)]> {
20   }
21 }
22
23 defm ADD  : ALU<0b01000, "add", add>;
24 defm SUB  : ALU<0b01001, "sub", sub>;
25 defm AND  : ALU<0b01100, "and", and>;
26 defm OR   : ALU<0b01110, "or",  or>;
27 defm XOR  : ALU<0b01111, "xor", xor>;
28 defm MUL  : ALU<0b11010, "mul", mul>;
29 defm DIV  : ALU<0b11011, "div", udiv>;
30 ...
31 } // let Defs = [SR]
```

Listing 4.9: Completed ALU Instruction Definitions

In addition to the opcode, these definitions also contain some other extremely important information for LLVM. For example, consider the `ADDri` definition. The `outs` and `ins` fields on lines 15 and 16 of Listing 4.9 describe the source and destination of each instruction's

outputs and inputs. Line 15 describes that the instruction outputs one variable into the `GPRegs` register class and it is stored in the class's `ri` variable (defined on line 10 of Listing 4.8). Line 16 of Listing 4.9 describes that the instruction accepts two operands; the first operand comes from the `GPRegs` register class while the second is defined by the custom `CJGimm16` operand type. The first operand is stored in the class's `rj` variable and the second operand is stored in the class's `rk` variable. Line 17 shows the assembly string definition; the `opstr` variable is passed into the class as a parameter and the class variables are referenced by the '`$`' character. Lines 18 and 19 describe the instruction pattern. This is how the code generator eventually is able to select this instruction from the LLVM IR. The `opnode` parameter is passed in from the third parameter of the `defm` declaration shown on line 23. The `opnode` type is an `SDNode` class which represents a node in the DAG used for instruction selection (called the SelectionDAG). In this example the `SDNode` is `add`, which is already defined by LLVM. Some instructions, however, need a custom `SDNode` implementation. This pattern will be matched if there is an `add` node in the SelectionDAG with two operands, where one is a register in the `GPRegs` class and the other a constant. The destination of the node must also be a register in the `GPRegs` class.

One other detail that is expressed in the complete instruction definitions is the implicit use or definition of other physical registers in the CPU. Consider the simple assembly instruction

```
add r4, r5, r6
```

where `r5` is added to `r6` and the result is stored in `r4`. This instruction is said to *define* `r4` and *use* `r5` and `r6`. Because all `add` instructions can modify the status register, this instruction is also said to implicitly *define* `SR`. This is expressed in TableGen using the `Defs` and `implicit` keywords and can be seen on lines 5, 12, and 19 of Listing 4.9. The implicit use of a register can also be expressed in TableGen using the `Uses` keyword. This can be

seen in the definition of the jump conditional instruction. Because the jump conditional instruction is dependent on the status register, even though the status register is not an input to the instruction, it is said to implicitly *use* the `SR`. This definition is shown in Listing 4.10. This listing also shows the use of a custom `SDNode` class, `CJGbrcc`, along with the use of the custom `cc` operand (defined in Listing 4.6).

```
1  // Conditional jump
2  let isBranch = 1, isTerminator = 1, Uses=[SR] in {
3  def JCC : FC_Inst<0b00101,
4                    (outs), (ins jmptarget:$addr, cc:$condition),
5                    "j$condition\t$addr",
6                    [(CJGbrcc bb:$addr, imm:$condition)]> {
7    // set ri to 0 and control to 1 for absolute addressing mode
8    let ri = 0b00000;
9    let control = 0b1;
10 }
11 } // isBranch = 1, isTerminator = 1
```

Listing 4.10: Completed Jump Conditional Instruction Definition

4.2.1.6 Additional Descriptions

There are additional descriptions that have not yet been moved to TableGen and must be implemented in C++. One such example of this is the `CJGRegisterInfo` struct. The reserved registers of the CPU must be described by a function called `getReservedRegs`. This function is shown in Listing 4.11.

4.2.2 Instruction Selection

The instruction selection stage of the backend is responsible for translating the LLVM IR code into target-specific machine instructions [20]. This section describes the phases of the of the instruction selector.

```
BitVector CJGRegisterInfo::getReservedRegs(const MachineFunction &MF) const {
  BitVector Reserved(getNumRegs());

  Reserved.set(CJG::SR); // status regsiter
  Reserved.set(CJG::PC); // program counter
  Reserved.set(CJG::SP); // stack pointer

  return Reserved;
}
```

Listing 4.11: Reserved Registers Description Implementation

4.2.2.1 SelectionDAG Construction

The first step of this process is to build an illegal SelectionDAG from the input. A SelectionDAG is considered *illegal* if it contains instructions or operands that can not be represented on the target CPU. The conversion from LLVM IR to the initial SelectionDAG is mostly hard-coded and is completed by code generator framework. Consider an example function, `myDouble`, that accepts an integer as a parameter and returns the input, doubled. The C code implementation for this function, `myDouble`, is shown in Listing 4.12, and the equivalent LLVM IR code is shown in Listing 4.13.

```
int myDouble(int a) {
  if (a == 0) {
    return 0;
  }
  return a + a;
}
```

Listing 4.12: `myDouble` C Implementation

As discussed in Section 4.1.1, a separate SelectionDAG is constructed for each basic block of code. As denoted by the labels (`entry`, `if.then`, and `if.end`) in Listing 4.13, there are three basic blocks in this function. The initial SelectionDAGs constructed for each basic block in the `myDouble` LLVM IR code are shown in Figs. 4.2, 4.3 and 4.4. Each

```
define i32 @myDouble(i32 %a) #0 {
entry:
  %cmp = icmp eq i32 %a, 0
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  ret i32 0

if.end:                                           ; preds = %entry
  %add = add nsw i32 %a, %a
  ret i32 %add
}
```

Listing 4.13: `myDouble` LLVM IR Code

node of the graph represents an instance of an `SDNode` class. Each node typically contains an opcode to specify the specific function of the node. Some nodes only store values while other nodes operate on values from connecting nodes. In the SelectionDAG figures, inputs into nodes are enumerated at the top of the node and outputs are drawn at the bottom.

The SelectionDAG can represent both data flow and control flow dependencies. Consider the SelectionDAG shown in Fig. 4.2. The solid arrows (*e.g.* connecting node t1 and t2) represent a data flow dependency. However, the dashed arrows (*e.g.* connecting t0 and t2) represent a control flow dependency. Data flow dependencies preserve data that needs to be available for direct use in a future operation, and control flow dependencies preserve the order between nodes that have side effects (such as branching/jumping) [20]. The control flow dependencies are called *chain* edges and can be seen in the SelectionDAG figures as the dashed arrows connecting from a "ch" node output to the input of their dependent node. A custom dependency sometimes needs to be specified for target-specific operations. These can be specified through glue dependencies which can help to keep the nodes from being separated in scheduling. This can be seen in Fig. 4.3 by the arrow connecting the "glue" output of node t3 to input 2 of node t4. This is necessary because any return values

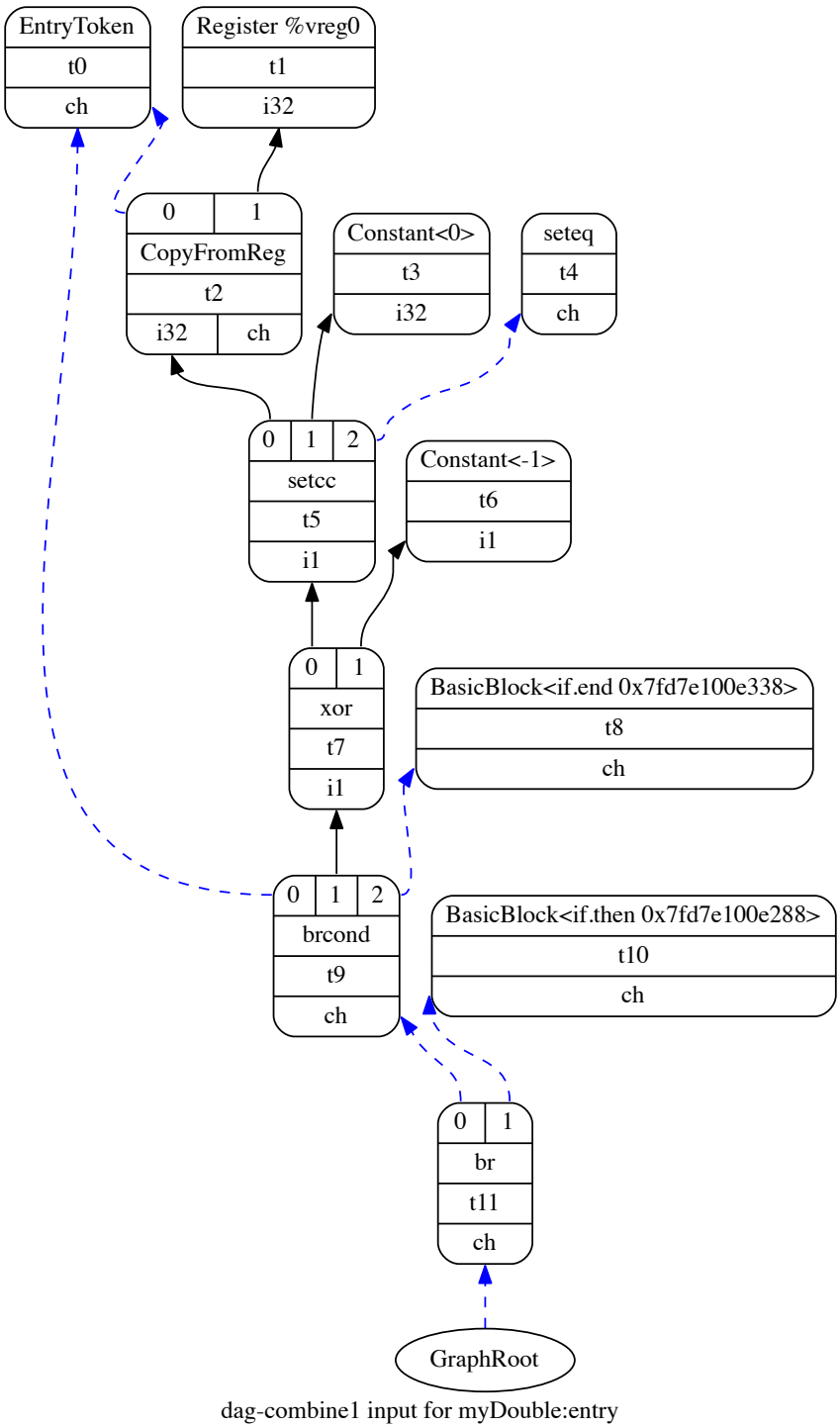

Figure 4.2: Initial `myDouble`:entry SelectionDAG

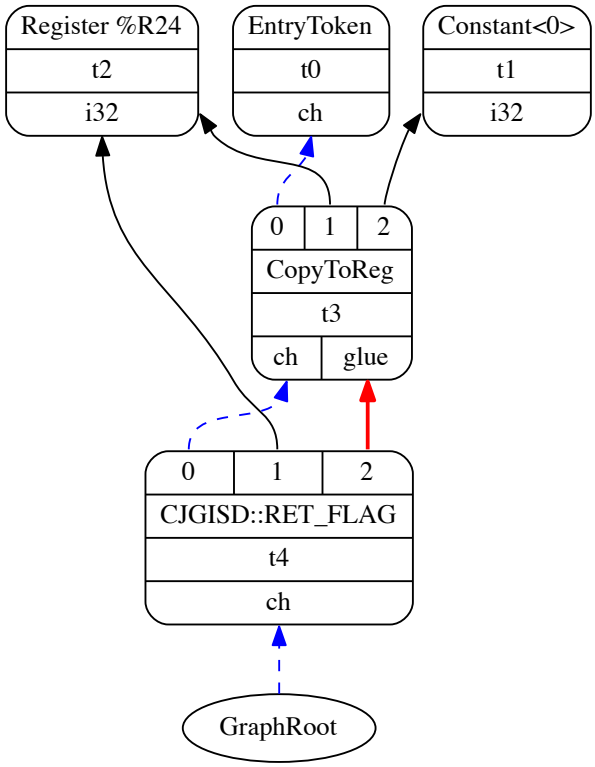

Figure 4.3: Initial `myDouble`:if.then SelectionDAG

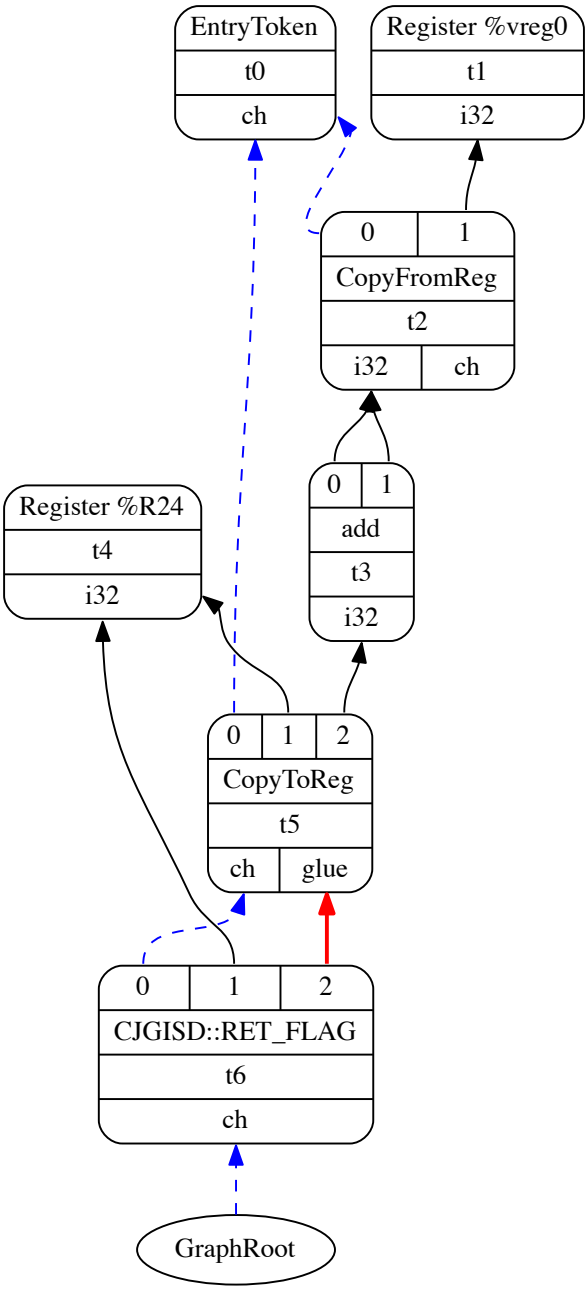

Figure 4.4: Initial `myDouble`:if.end SelectionDAG

must not be disturbed before the function returns.

4.2.2.2 Legalization

After the SelectionDAG is initially constructed, any LLVM instructions or datatypes that are not supported by the target CPU must be converted, or *legalized*, so that the entire DAG can be represented natively by the target. However, there are some initial optimization passes that occur before legalization. The SelectionDAG for the `myDouble`:entry basic block prior to legalization but following the initial optimization passes can be seen in Fig. 4.5. Comparing this to the SelectionDAG prior to the optimization (seen in Fig. 4.2) shows that nodes t4, t5, t6, t7, and t9 were combined into nodes t12 and t14.

The legalization passes run immediately following the optimization passes. The legalized SelectionDAG for the `myDouble`:entry basic block is shown in Fig. 4.6. As an example to show how legalization is implemented, consider the legalization of SelectionDAG nodes t12 and t14 (seen in Fig. 4.5), into nodes t15, t16, and t17 (seen in Fig. 4.6).

Implementing instruction legalization involves both TableGen descriptions and custom C++ code in the backend. Custom `SDNodes` are first defined in `CJGInstrInfo.td`. Two custom node definitions are shown in Listing 4.14. Although there are many target-independent SelectionDAG operations that are defined in the LLVM `ISDOpcodes.h` header file, the instructions for this example require the target-specific operations: `CJGISD::CMP` (compare) and `CJGISD::BR_CC` (conditional branch). These operations are defined in `CJGISelLowering.h` as seen in Listing 4.15. One other requirement is to describe the jump condition codes. This encodes the information described in Table 3.5 and is shown in Listing 4.16.

The implementation for the legalization is written in `CJGISelLowering.cpp` as part of the custom `CJGTargetLowering` class (inherited from LLVM's `TargetLowering` class).

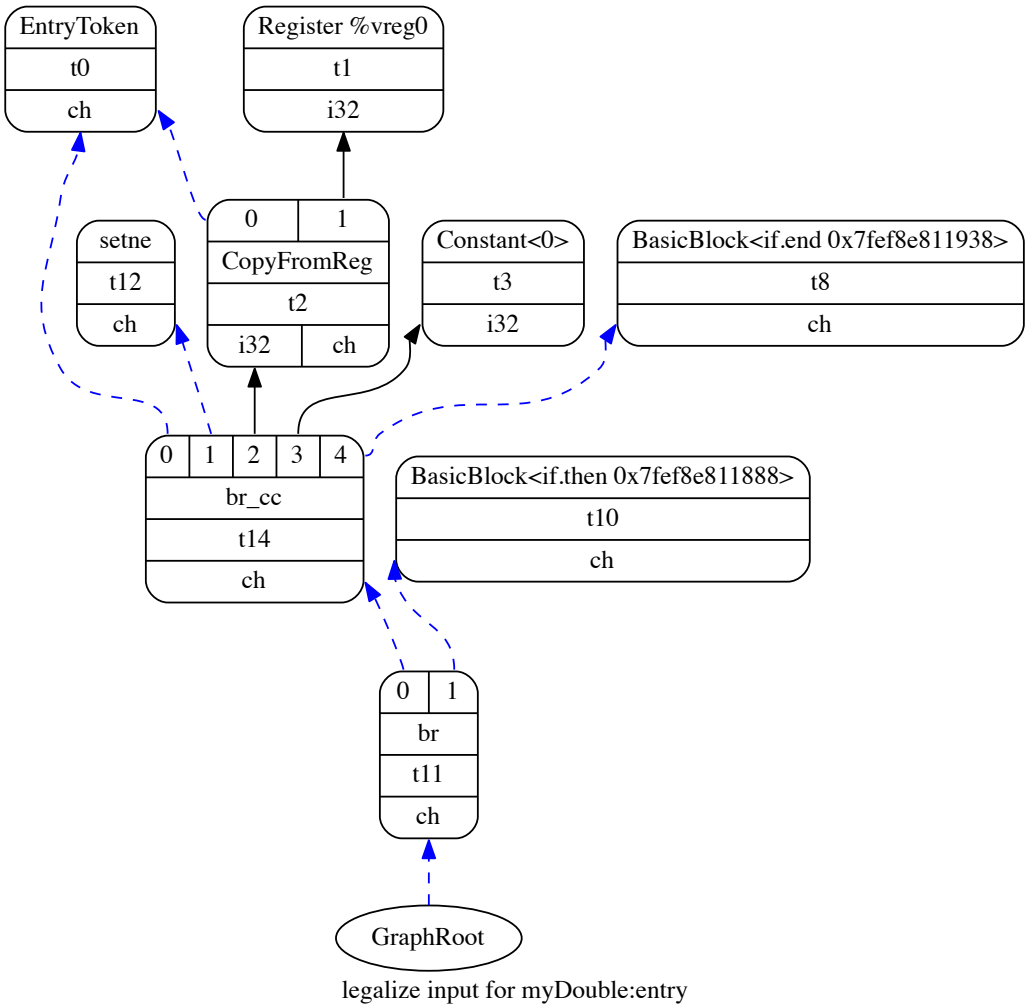

Figure 4.5: Optimized `myDouble`:entry SelectionDAG

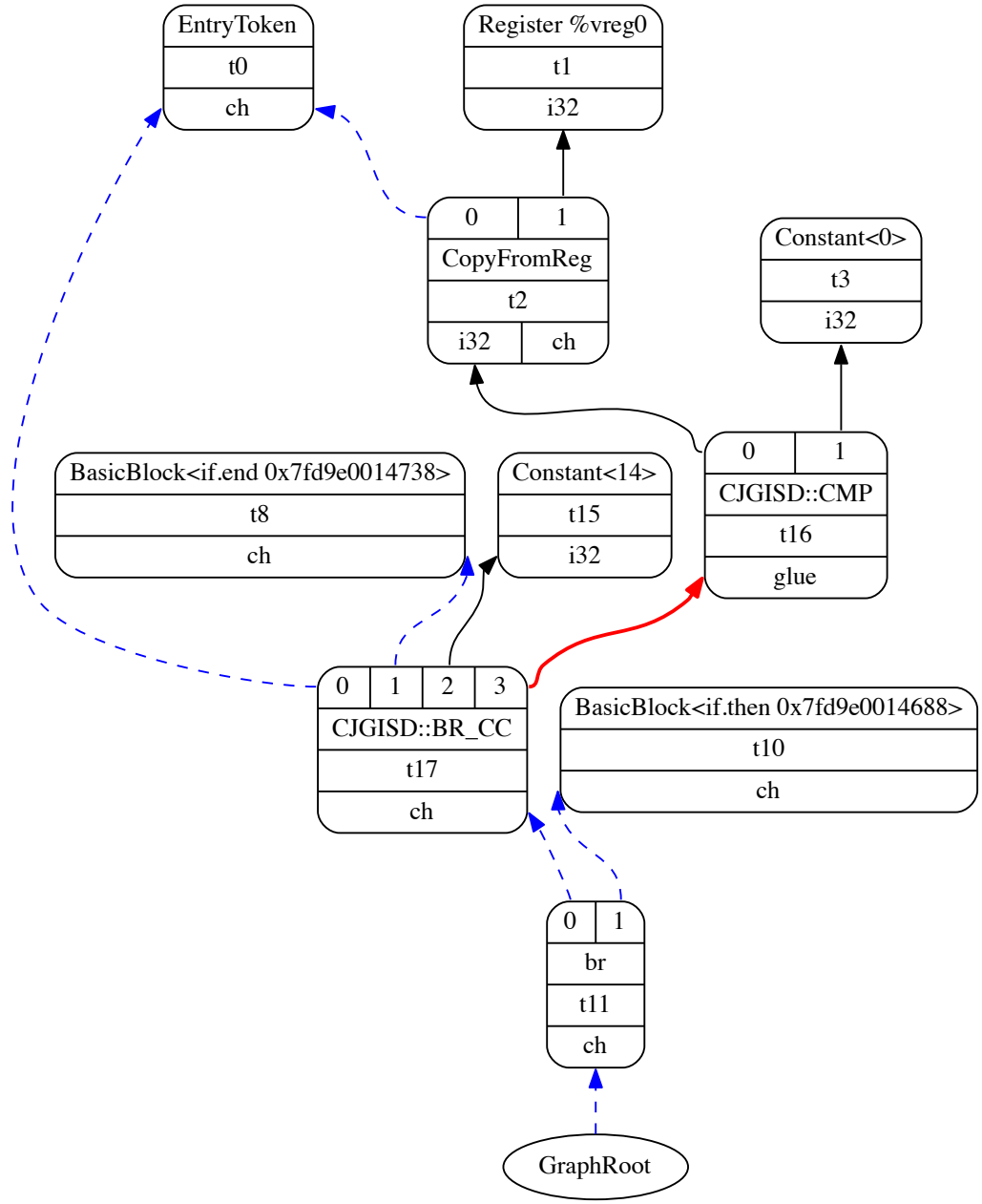

Figure 4.6: Legalized `myDouble`:entry SelectionDAG

```
def CJGcmp     : SDNode<"CJGISD::CMP", SDT_CJGCmp, [SDNPOutGlue]>;
def CJGbrcc    : SDNode<"CJGISD::BR_CC", SDT_CJGBrCC, [SDNPHasChain,
                                                       SDNPInGlue]>;
```

Listing 4.14: Custom `SDNode` TableGen Definitions

```
namespace CJGISD {
enum NodeType {
  FIRST_NUMBER = ISD::BUILTIN_OP_END,
...
  // The compare instruction
  CMP,

  // Branch conditional, condition-code
  BR_CC,
...
};
}
```

Listing 4.15: Target-Specific `SDNode` Operation Definitions

```
namespace CJGCC {
  // CJG specific condition codes
  enum CondCodes {
    COND_U      = 0,    // unconditional
    COND_C      = 8,    // carry
    COND_N      = 4,    // negative
    COND_V      = 2,    // overflow
    COND_Z      = 1,    // zero
    COND_NC     = 7,    // not carry
    COND_NN     = 11,   // not negative
    COND_NV     = 13,   // not overflow
    COND_NZ     = 14,   // not zero
    COND_GE     = 6,    // greater or equal
    COND_L      = 9,    // less than

    COND_INVALID = -1
  };
}
```

Listing 4.16: Jump Condition Code Encoding

The custom operations are first specified in the constructor for `CJGTargetLowering` which causes the method `LowerOperation` to be called when these custom operations are encountered. `LowerOperation` is responsible for choosing which class method to call for each custom operation. In this example, the method, `LowerBR_CC`, is called. This portion of the legalization implementation is shown in Listing 4.17.

```
1  SDValue CJGTargetLowering::LowerOperation(SDValue Op, SelectionDAG &DAG) const {
2    switch (Op.getOpcode()) {
3      case ISD::BR_CC:            return LowerBR_CC(Op, DAG);
4  ...
5      default:
6        llvm_unreachable("unimplemented operand");
7    }
8  }
9  ...
10 SDValue CJGTargetLowering::LowerBR_CC(SDValue Op, SelectionDAG &DAG) const {
11   SDValue Chain = Op.getOperand(0);
12   ISD::CondCode CC = cast<CondCodeSDNode>(Op.getOperand(1))->get();
13   SDValue LHS   = Op.getOperand(2);
14   SDValue RHS   = Op.getOperand(3);
15   SDValue Dest  = Op.getOperand(4);
16   SDLoc dl  (Op);
17
18   SDValue TargetCC;
19   SDValue Flag = EmitCMP(LHS, RHS, TargetCC, CC, dl, DAG);
20
21   return DAG.getNode(CJGISD::BR_CC, dl, Op.getValueType(),
22                      Chain, Dest, TargetCC, Flag);
23 }
```

Listing 4.17: Target-Specific `SDNode` Operation Implementation

The actual legalization occurs within the `LowerBR_CC` method. Lines 11–15 of Listing 4.17 show how the `SDNode` values (the inputs of node t14 of the SelectionDAG shown in Fig. 4.5) are stored into variables. The `EmitCMP` helper method (called on line 19) returns an `SDNode` for the `CJG::CMP` operation and also sets the `TargetCC` variable to the correct condition code. Once these values are set up, the new target-specific `SDNode` is created

using the `getNode` helper method defined in the `SelectionDAG` class. This node is then returned through the `LowerOperation` method and finally replaces the original nodes, t12 and t14, with nodes t15, t16, and t17 (as seen in Fig. 4.6).

4.2.2.3 Selection

The select phase is the largest phase within the instruction selection process [20]. This phase is responsible for transforming the legalized SelectionDAG comprised of LLVM and custom operations, into a DAG comprised of target operations. The selection phase is largely dependent on the patterns defined in the compete instruction descriptions (discussed in Section 4.2.1.5). For example, consider the ALU instruction patterns shown on lines 11 and 18 of Listing 4.9, as well as the jump conditional instruction pattern shown on line 6 of Listing 4.10. These patterns are used by the `SelectionDAGISel` class to select the target-specific instructions. The `myDouble` DAGs following the selection phase are shown in Figs. 4.7, 4.8, and 4.9.

The ALU patterns matched nodes t1 and t3, from the `myDouble`:if.then SelectionDAG shown in Fig. 4.3, into nodes t1 and t5, which are seen in the DAG shown in Fig. 4.8. Node t3 of the `myDouble`:if.end SelectionDAG shown in Fig. 4.4 was also matched by the ALU patterns. The target-independent "add" operation was replaced by the target-specific "ADDrr" operation, which is seen in node t3 of the DAG shown in Fig. 4.9. The custom "CJGISD::CMP" and "CJGISD::BR_CC" operations in nodes t16 and t17 of the SelectionDAG shown in Fig. 4.6 were also matched. The resulting, target-specific "CMPri" and "JCC" operations can be seen in nodes t16 and t17 of the DAG shown in Fig. 4.7. After the completion of this phase, all `SDNode` operations represent target instructions and the DAG is ready for scheduling.

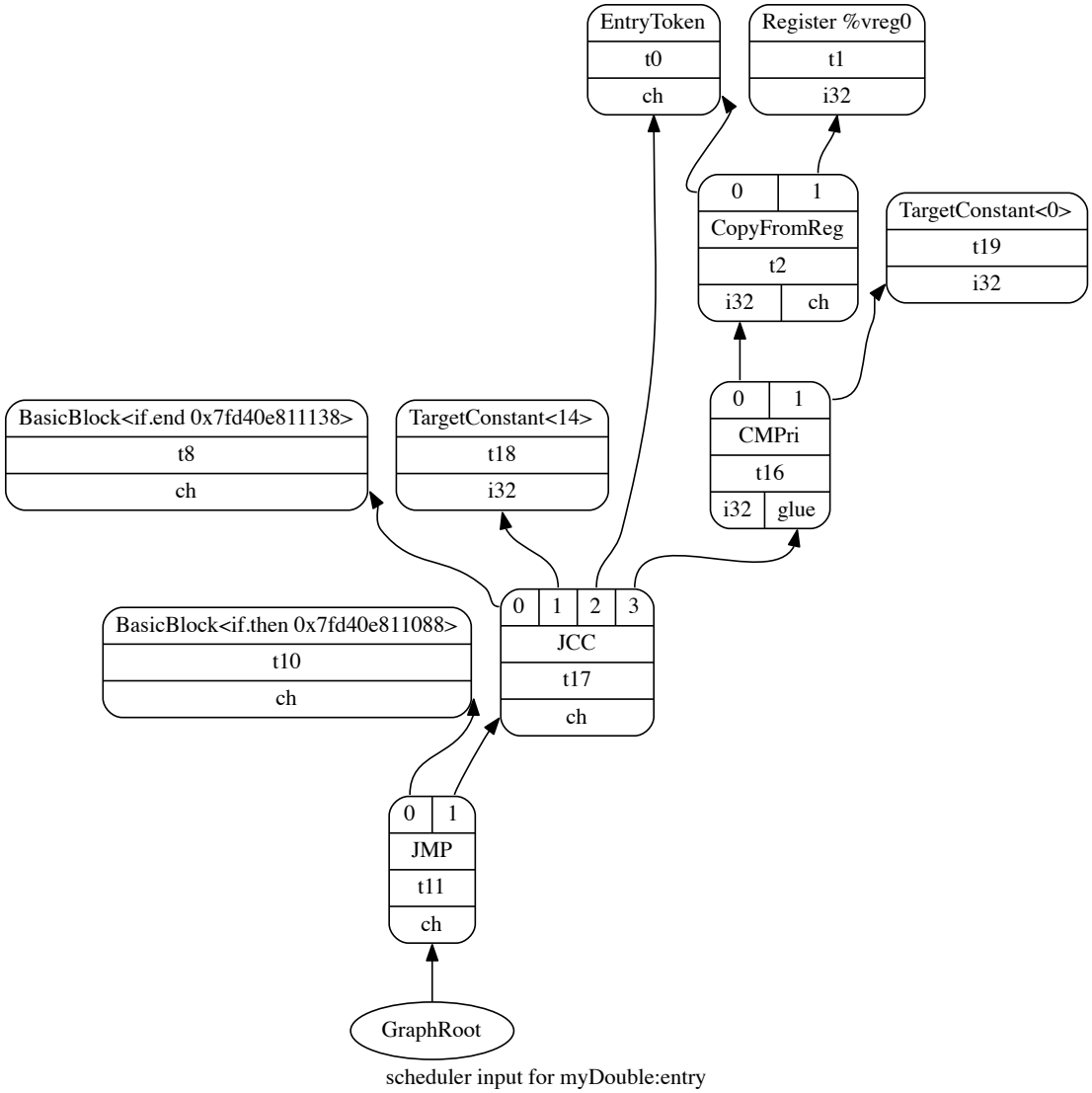

Figure 4.7: Selected `myDouble`:entry SelectionDAG

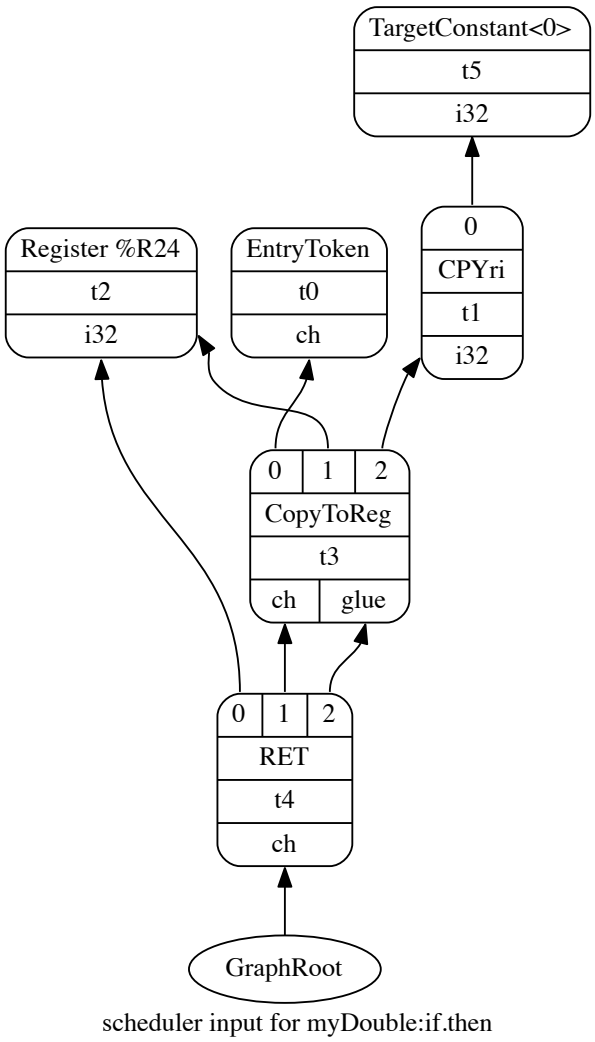

Figure 4.8: Selected `myDouble`:if.then SelectionDAG

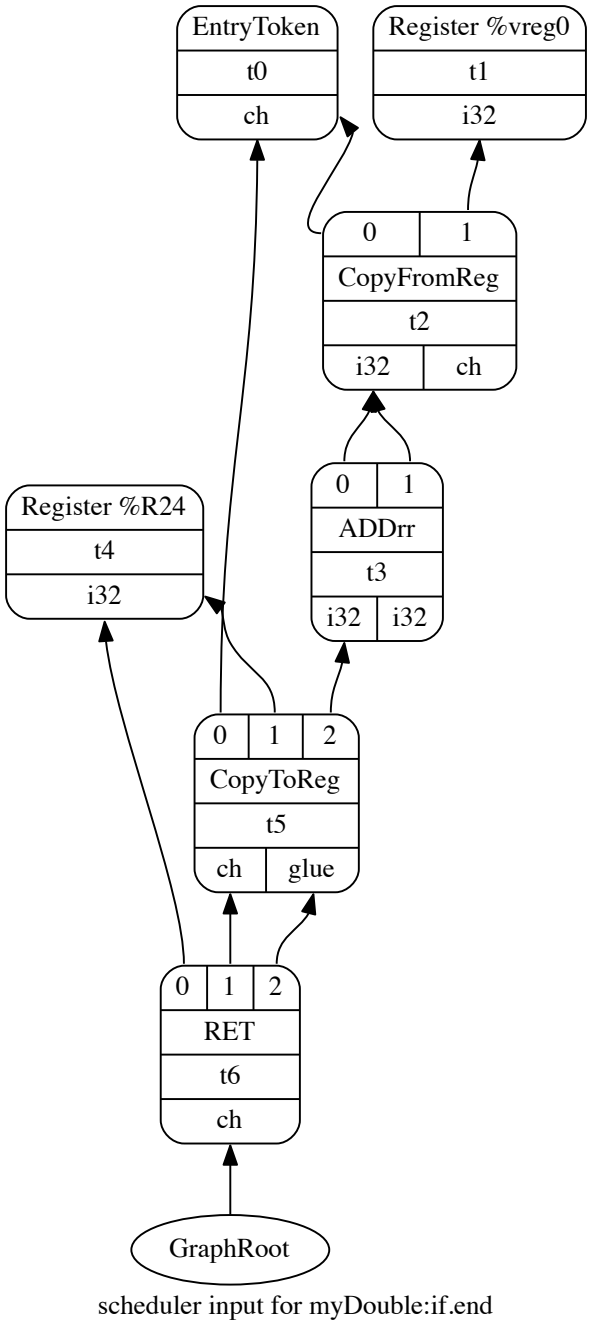

Figure 4.9: Selected `myDouble`:if.end SelectionDAG

4.2.2.4 Scheduling

The scheduling phase is responsible for transforming the DAG of target instructions into a list of machine instructions (represented by instances of the `MachineInstr` class). The scheduler can order the instructions depending on constraints such as minimizing register usage or reducing overall program latency [20]. Once the list of machine instructions has been finalized, the DAG is destroyed. The scheduled list of machine instructions for the `myDouble` function can be seen in Listing 4.18.

```
1  BB#0: derived from LLVM BB %entry
2      Live Ins: %R4
3      %vreg0<def> = COPY %R4; GPRegs:%vreg0
4      CMPri %vreg0, 0, %SR<imp-def>; GPRegs:%vreg0
5      JCC <BB#2>, 14, %SR<imp-use>
6      JMP <BB#1>
7
8  BB#1: derived from LLVM BB %if.then
9      Predecessors according to CFG: BB#0
10     %vreg2<def> = CPYri 0; GPRegs:%vreg2
11     %R24<def> = COPY %vreg2; GPRegs:%vreg2
12     RET %R24<imp-use>
13
14 BB#2: derived from LLVM BB %if.end
15     Predecessors according to CFG: BB#0
16     %vreg1<def> = ADDrr %vreg0, %vreg0, %SR<imp-def,dead>;
17                   GPRegs:%vreg1,%vreg0,%vreg0
18     %R24<def> = COPY %vreg1; GPRegs:%vreg1
19     RET %R24<imp-use>
```

Listing 4.18: Initial `myDouble` Machine Instruction List

4.2.3 Register Allocation

This phase of the backend is responsible for eliminating all of the virtual registers from the list of machine instructions and replacing them with physical registers. For a simple

RISC machine there is typically very little customization required for functional register allocation. The main algorithm used in this phase is called the "greedy register allocator." The main benefit to this algorithm is that it allocates the largest ranges of live variables first [27]. When there are live variables that cannot be assigned to a register because there are none available, they are *spilled* to memory. Then instead of using a physical register, load and store instructions are inserted into the list of machine instructions before and after the value is used. The final list of machine instructions for the `myDouble` function can be seen in Listing 4.19. The final register mapping is shown in Table 4.1. Once all of the virtual registers have been eliminated, the code can be emitted to the target language.

```
1  BB#0: derived from LLVM BB %entry
2      Live Ins: %R4 %SR
3      PUSH %SR<kill>, %SP<imp-def>
4      CMPri %R4, 0, %SR<imp-def>
5      JCC <BB#1>, 1, %SR<imp-use>
6
7  BB#2: derived from LLVM BB %if.end
8      Live Ins: %R4
9      Predecessors according to CFG: BB#0
10     %R24<def> = ADDrr %R4<kill>, %R4, %SR<imp-def,dead>
11     %SR<def> = POP %SP<imp-def>
12     RET %R24<imp-use>
13
14 BB#1: derived from LLVM BB %if.then
15     Predecessors according to CFG: BB#0
16     %R24<def> = CPYri 0
17     %SR<def> = POP %SP<imp-def>
18     RET %R24<imp-use>
```

Listing 4.19: Final `myDouble` Machine Instruction List

Virtual Register	Physical Register
%vreg0	%R4
%vreg1	%R24
%vreg2	%R24

Table 4.1: Register Map for `myDouble`

4.2.4 Code Emission

The final phase of the backend is to emit the machine instruction list as either target-specific assembly code (emitted by the assembly printer) or machine code (emitted by the object writer).

4.2.4.1 Assembly Printer

Printing assembly code requires the implementation of several custom classes. The `CJGAsmPrinter` class represents the pass that is run for printing the assembly code. The `CJGMCAsmInfo` class defines some basic static information to be used by the assembly printer, such as defining the string used for comments:

```
CommentString = "//";
```

The `CJGInstPrinter` class holds most of the important functions used when printing the assembly. It imports the C++ code that is automatically generated from the AsmWriter TableGen backend and specifies additional required methods. One such method is the `printMemSrcOperand` which is responsible for printing the custom `memsrc` operand defined in Listing 4.6. The implementation for this method is shown in Listing 4.20. The method operates on the `MCInst` class abstraction and outputs the correct string representation for the operand. The final assembly code for the `myDouble` function is shown in Listing 4.21. The assembly printer adds helpful comments and also comments out the label of any basic block that is not used as a jump location in the assembly code.

```
// Print a memsrc (defined in CJGInstrInfo.td)
// This is an operand which defines a location for loading or storing which
// is a register offset by an immediate value
void CJGInstPrinter::printMemSrcOperand(const MCInst *MI, unsigned OpNo,
                                        raw_ostream &O) {
  const MCOperand &BaseAddr = MI->getOperand(OpNo);
  const MCOperand &Offset = MI->getOperand(OpNo + 1);

  assert(Offset.isImm() && "Expected immediate in displacement field");

  O << "M[";
  printRegName(O, BaseAddr.getReg());
  unsigned OffsetVal = Offset.getImm();
  if (OffsetVal) {
    O << "+" << Offset.getImm();
  }
  O << "]";
}
```

Listing 4.20: Custom `printMemSrcOperand` Implementation

```
myDouble:                                 // @myDouble
// BB#0:                                  // %entry
    push r0
    cmp  r4, 0
    jeq  BB0_1
// BB#2:                                  // %if.end
    add r24, r4, r4
    pop r0
    ret
BB0_1:                                    // %if.then
    cpy r24, 0.
    pop r0
    ret
```

Listing 4.21: Final `myDouble` Assembly Code

4.2.4.2 ELF Object Writer

The custom machine code is emitted in the form of an ELF object file. As with the assembly printer, several custom classes need to be implemented for emitting machine code. The

`CJGELFObjectWriter` class mostly serves as a wrapper to its base class, the `MCELFObjectTargetWriter`, which is responsible for properly formatting the ELF file. The `CJGMCCodeEmitter` class contains most of the important functions for emitting the machine code. It imports the C++ code that is automatically generated from the CodeEmitter TableGen backend. This backend handles a majority of the bit-shifting and formatting required to encode the instructions as seen in Section 4.2.1.4. The `CJGMCCodeEmitter` class also is responsible for encoding custom operands, such as the `memsrc` operand defined in Listing 4.6. The implementation of the method responsible for encoding this custom operand, named `getMemSrcValue`, can be seen in Listing 4.22.

```
1  // Encode a memsrc (defined in CJGInstrInfo.td)
2  // This is an operand which defines a location for loading or storing which
3  // is a register offset by an immediate value
4  unsigned CJGMCCodeEmitter::getMemSrcValue(const MCInst &MI, unsigned OpIdx,
5                                            SmallVectorImpl<MCFixup> &Fixups,
6                                            const MCSubtargetInfo &STI) const {
7    unsigned Bits = 0;
8    const MCOperand &RegOp = MI.getOperand(OpIdx);
9    const MCOperand &ImmOp = MI.getOperand(OpIdx + 1);
10   Bits |= (getMachineOpValue(MI, RegOp, Fixups, STI) << 16);
11   Bits |= (unsigned)ImmOp.getImm() & 0xffff;
12   return Bits;
13 }
```

Listing 4.22: Custom `getMemSrcValue` Implementation

The custom `memsrc` operand represents 21 bits of data: 5 bits are required for the register encoding and another 16 bits for the immediate value. These are stored in a single value and then later separated by code automatically generated from TableGen. The usage of this custom operand can be seen in Listing 4.23, which shows instruction format definition for the load and store instructions (as specified in Section 3.3.1). Line 7 shows the declaration, line 11 shows the bits used for the register value, and line 13 shows the

bits used for the immediate value. The CodeEmitter TableGen backend for this definition produces the C++ code seen in Listing 4.24. This code is used when writing the machine code for the load instruction. Line 6 shows the usage of the custom `getMemSrcValue` method. Line 7 masks off everything except the register bits and shifts it into the proper place in the instruction word, and line 8 does the same but for the 16-bit immediate value instead.

```
1  class LS_Inst<bits<5> opcode, dag outs, dag ins, string asmstr,
2                list<dag> pattern>
3      : InstCJG<outs, ins, asmstr, pattern> {
4
5    bits<5> ri;
6    bits<1> control;
7    bits<21> addr;
8
9    let Opcode = opcode;
10   let Inst{26-22} = ri;
11   let Inst{21-17} = addr{20-16}; // rj
12   let Inst{16} = control;
13   let Inst{15-0} = addr{15-0};
14 }
```

Listing 4.23: Base Load and Store Instruction Format Definitions

```
1  case CJG::LD: {
2        // op: ri
3        op = getMachineOpValue(MI, MI.getOperand(0), Fixups, STI);
4        Value |= (op & UINT64_C(31)) << 22;
5        // op: addr
6        op = getMemSrcValue(MI, 1, Fixups, STI);
7        Value |= (op & UINT64_C(2031616)) << 1;
8        Value |= op & UINT64_C(65535);
9        break;
10     }
```

Listing 4.24: CodeEmitter TableGen Backend Output for Load

The target-specific machine instructions are placed into the ".text" section of the ELF

object file. Using a custom ELF parser and custom disassembler for the CJG architecture (described in Section 5.3), the resulting disassembly from the ELF object file can viewed. The disassembly and machine code (shown as a Verilog memory file) for the `myDouble` function is shown in Listings 4.25 and 4.26. This shows that the assembly code produced by the assembly printer (as shown in Listing 4.21) is equivalent to the machine code produced by the object writer.

```
1                push    r0                  // @00000000 00
2                cmp     r4, 0               // @00000004 01
3                jeq     label_0             // @00000008 18
4                add     r24, r4, r4         // @0000000C 00
5                pop     r0                  // @00000010 00
6                ret                         // @00000014 00
7   label_0:     cpy     r24, 0              // @00000018 00
8                pop     r0                  // @0000001C 00
9                ret                         // @00000020 00
```

Listing 4.25: Disassembled `myDouble` Machine Code

```
1   @00000000 18000000  //          push    r0
2   @00000004 50080001  //          cmp     r4, 0
3   @00000008 28050018  //          jeq     label_0
4   @0000000C 46084000  //          add     r24, r4, r4
5   @00000010 20000000  //          pop     r0
6   @00000014 38000000  //          ret
7   @00000018 16010000  // label_0: cpy     r24, 0
8   @0000001C 20000000  //          pop     r0
9   @00000020 38000000  //          ret
```

Listing 4.26: `myDouble` Machine Code

Chapter 5

Tests and Results

This chapter discusses the tests and results from the implementation of the custom CJG RISC CPU and LLVM backend and describes custom tools created to support the project.

5.1 LLVM Backend Validation

To test the functionality of the LLVM backend code generation, several programs written in either C or LLVM IR were compiled for the CJG RISC. Although there is a custom assembler that targets the CJG RISC and a majority of generated assembly code is correctly printed to a format that is compatible with the CJG assembler, there is some functionality that the CJG assembler does not support. This leads to some input code sequences that yield assembly code not supported by the CJG assembler. Because of this issue, most of the programs simulated on the CJG RISC CPU were taken from the compiled ELF object files which were then disassembled for easier debugging.

To simulate the CPU, a suite of tools from Cadence (Incisive) is used to simulate the CPU Verilog code for verification and viewing the simulation waveforms. The Synopsys

tools are then used to synthesize the CPU Verilog code. The resulting gate level netlist is then simulated and verified. A simple testbench instantiates the CJG RISC CPU and the two memory models (described in Section 3.1.3). The `$readmemh` Verilog function is used in the testbench to initialize the program memory with the machine code from the ELF object file. An intermediate tool, `elf2mem` (discussed in Section 5.3), is used to extract the machine code from the ELF file and write it to the format required by `$readmemh`. Additionally, the CJG disassembler is used to modify the generated code to make it more friendly to the simulation environment.

For example, consider the `myDouble` function that was discussed throughout Chapter 4. The code generated from the custom backend that is shown in Listing 4.25 was modified slightly, and the new code is shown in Listing 5.1. The first code modification made was inserting the instruction on line 2; this instruction loads `r4` from the CPU's GPIO input port, which is memory mapped to address `0xFFF0`. This allows different input values to be set from the testbench. The other modification made was to remove the return instructions and instead jump to the `done` label seen on line 10. This writes the result from `r24` to the CPU's GPIO output port so that the return value can be observed from the testbench. For this example, the testbench set the GPIO input as `0xC` (12). The simulation was run using NCSim and viewed in Cadence SimVision; the resulting waveform can be seen in Fig. 5.1. The simulation shows that the GPIO output is correctly set to `0x18` (24), which is double `0xC`, just before the 160,000 *ps* time mark.

Although this simple program successfully compiles and simulates successfully on the CPU, the backend is still not fully complete and has some errors when generating code for certain code sequences. One such example of this is the usage of datatypes that are not `int`, such as `short int`, and `char`, or more specifically, `i16`, `i8`, and `i1` as defined by LLVM [28]. These smaller datatypes need to be sign extended when loaded from memory, and

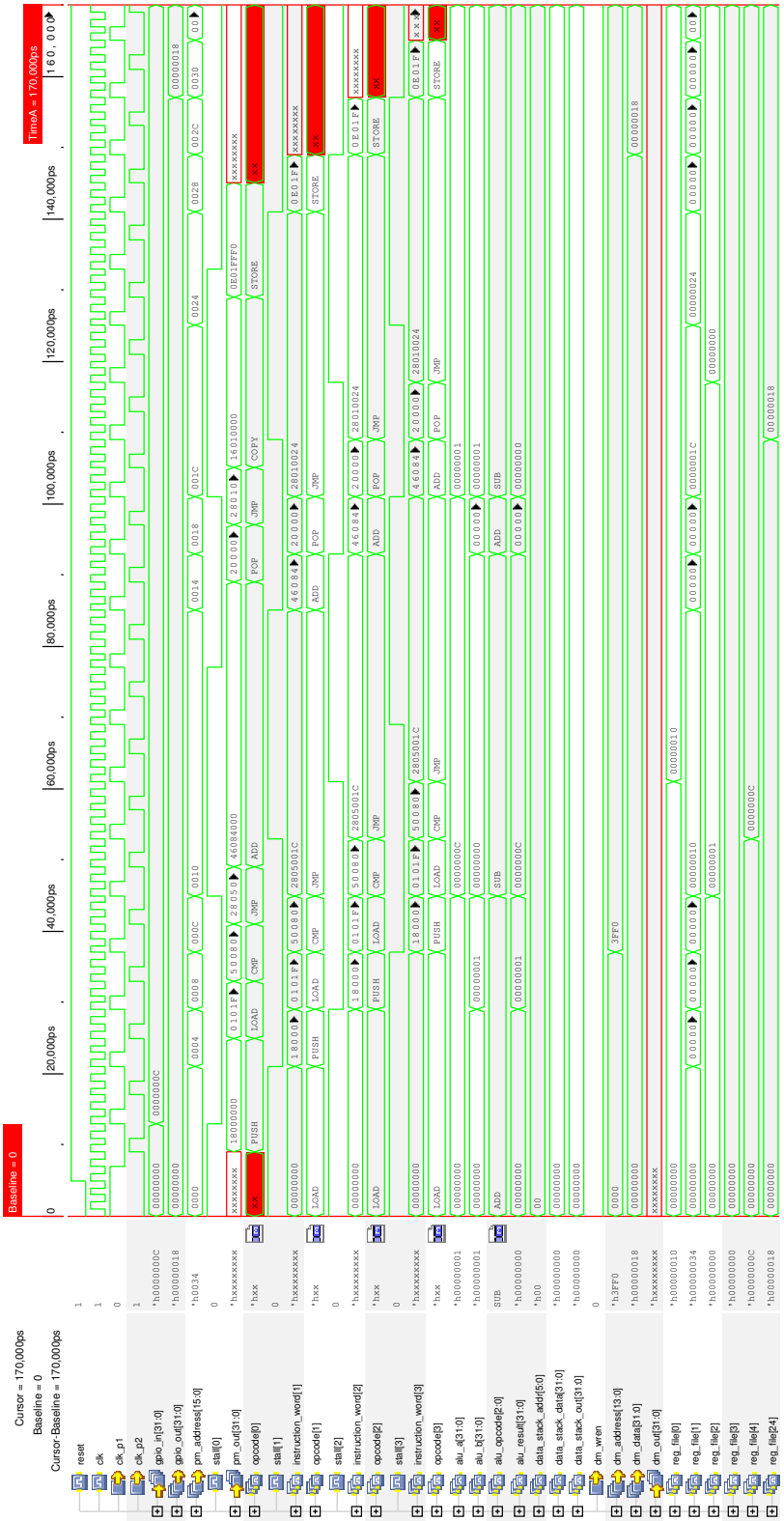

Figure 5.1: `myDouble` Simulation Waveform

```
1                push    r0                  // @00000000 00
2                ld      r4, M[0xFFF0]
3                cmp     r4, 0               // @00000004 01
4                jeq     label_0             // @00000008 18
5                add     r24, r4, r4         // @0000000C 00
6                pop     r0                  // @00000010 00
7                jmp     done
8   label_0:     cpy     r24, 0              // @00000018 00
9                pop     r0                  // @0000001C 00
10  done:        st      M[0xFFF0], r24
```

Listing 5.1: Modified `myDouble` Assembly Code

the CJG architecture does not implement any sign extension instructions. It is possible to describe how to perform sign extension in the instruction lowering process of the backend (discussed in Section 4.2.2.2), however, this is not fully implemented. Another example of code that is not supported involves stack operations. Even though the data stack within the CJG CPU is accessible by using a stack pointer, the stack data is not located within the memory space. This causes some complications in the backend involving stack operations, the stack pointer, and the stack frame, that are not completely resolved.

5.2 CPU Implementation

The CJG RISC CPU is designed using the Verilog HDL at the register transfer level (RTL) and synthesized using Synopsys Design Compiler with a 65 *nm* technology node from TSMC. The synthesis step is what transforms the RTL into a gate level *netlist*, which is a physical description of the hardware consisting of logic gates, standard cells, and their connections [29]. Two different synthesis options are used: RTL logic synthesis, and design for testability (DFT) synthesis using a full-scan methodology for test structure insertion, which inserts scan chains throughout the design. This section shows the results from each

Cell	**Global Cell Area**		**Local Cell Area (μm^2)**	
	Absolute Total (μm^2)	**Percent Total**	**Combinational**	**Noncombinational**
cjg_risc	94941.7219	100.0	11184.4803	15004.0802
cjg_alu	11650.3200	12.3	629.2800	0.0000
cjg_call_stack	24469.9207	25.8	6775.2000	17694.7207
cjg_clkgen	30.6000	0.0	14.7600	15.8400
cjg_data_stack	26258.4005	27.7	3886.5601	22371.8404
cjg_shifter	4805.6401	5.1	4805.6401	0.0000

Table 5.1: Pre-scan Netlist Area Results

Internal (mW)	**Switching (mW)**	**Leakage (μW)**	**Total (mW)**
7.6935	0.1759	3.1975	7.8729

Table 5.2: Pre-scan Netlist Power Results

of these synthesis passes. A system clock frequency of 1 *GHz* was used, resulting in an effective phase clock frequency of 250 *MHz*.

5.2.1 Pre-scan RTL Synthesis

The hierarchical area distribution report results for the pre-scan netlist are shown in Table 5.1. The total area of the design is the absolute total area of the cjg_risc module: 94941.7219 μm^2. The total gate count of a cell is calculated by dividing the cell's total area by the area of the NAND2 standard cell (1.44 μm^2). This synthesis pass yields about 65932 gates in the CPU. The results from the power report are shown in Table 5.2.

5.2.2 Post-scan DFT Synthesis

The post-scan synthesis pass and the pre-scan synthesis pass yield very similar results. The hierarchical area distribution report results for the post-scan netlist are shown in Table 5.3. The total gate count of the CPU in the post-scan netlist the same as the pre-scan netlist,

Cell	Global Cell Area		Local Cell Area (μm^2)	
	Absolute Total (μm^2)	Percent Total	Combinational	Noncombinational
cjg_risc	94912.9219	100.0	11180.1603	14996.1602
cjg_alu	11633.7600	12.3	629.2800	0.0000
cjg_call_stack	24469.9207	25.8	6775.2000	17694.7207
cjg_clkgen	30.6000	0.0	14.7600	15.8400
cjg_data_stack	26258.4005	27.7	3886.5601	22371.8404
cjg_shifter	4805.6401	5.1	4805.6401	0.0000

Table 5.3: Post-scan Netlist Area Results

Internal (mW)	Switching (mW)	Leakage (μW)	Total (mW)
7.6918	0.1759	3.1954	7.8711

Table 5.4: Post-scan Netlist Power Results

about 65932 gates. The results from the power report are shown in Table 5.2.

5.3 Additional Tools

This section discusses several other tools that were created or used throughout the design and implementation process of the CJG RISC and custom LLVM backend.

5.3.1 Clang

As discussed in Section 4.1.3, Clang is responsible for the front end actions of LLVM, however, a user compiling C code only needs to worry about clang because it links against the target-specific backends. Clang was used to output LLVM IR code from C code throughout the development of the backend. Even though most of code used for testing the backend throughout the development process was written in C, it was all manually converted into LLVM IR code by Clang before passing it into llc.

5.3.2 ELF to Memory

The ELF to memory (`elf2mem`) tool is a Python tool written to extract a binary section from an ELF file and output the binary in a format that is readable by the Verilog `$readmem` function. This tool was written so any ELF object files that are emitted from the custom LLVM backend can be read by the testbench and simulated on the CPU.

5.3.3 Assembler

The assembler was originally written for a different 32-bit RISC CPU; however, the architectures are similar and most of the assembler was re-used for this design. The assembler was heavily used during the implementation of the CJG RISC to verify that the CPU was functioning properly. Depending on the specific test, the assembly programs simulated on the CPU were either verified by visual inspection using SimVision or verified automatically using the testbench. Although there are frameworks within LLVM to create a target-specific assembler, the custom assembler was used instead because it was already mostly complete.

5.3.4 Disassembler

The disassembler was written when debugging the ELF object writer in the custom backend. This tool was fairly easy to write because it makes use of many of the classes found in the assembler. The disassembler reads in a memory file and outputs valid assembly code. When using this to debug ELF object files, the files were first converted to memory files using `elf2mem` and then disassembled using this tool.

Chapter 6

Conclusions

This chapter discusses future work that could be completed as well as the conclusions from this project.

6.1 Future Work

Compiler backends can always be improved upon and optimized. Even the LLVM backends currently located in the source tree (*e.g.* ARM and x86) that are considered completed are still receiving changes and improvements. To consider the LLVM backend for the CJG RISC CPU completed, the code generator would need to be able to support a majority of LLVM IR capabilities. In addition to making it possible to generate code from any valid LLVM IR input, target-specific optimization passes to increase machine code efficiency and quality should be implemented as well. The only optimization passes currently implemented are the target-independent optimizations included in the LLVM code generator framework. Lastly, the CJG backend should be fully integrated into Clang, eliminating the need to call llc and allowing C code to be compiled directly into CJG assembly or machine

code.

6.2 Project Conclusions

This paper describes the process of designing and implementing a custom 32-bit RISC CPU along with writing a custom LLVM compiler backend. Although compiler research is popular in computer science, the research generally is focused on the front end or optimization passes. Even when there is research focused on the backend of compilers, it typically is focused on the GCC project and not LLVM.

The custom RISC CPU was designed in Verilog and operates as a standard 4-stage pipeline. The goal was to create a simple enough RISC CPU that could be easily described for a compiler, while still retaining enough complexity to allow for sophisticated program execution. Although the custom CPU was fairly complete, there were still design choices that made the implementation of LLVM backend more complicated than needed, such as choosing a hardware data stack design instead of a memory based stack.

The custom compiler backend was written using the LLVM compiler infrastructure project. Although most CPU architectures are supported by the code generator in GCC, there are few that are supported by LLVM. The custom compiler backend was written using LLVM for its modern code design and to explore if there is a good reason for its lack of popularity in the embedded CPU community. Although implementing the custom LLVM backend to its current state was a difficult process, there does not seem to be a valid reason for its lack of popularity as a compiler. As more communities experiment with backends in LLVM and discover how modern and organized the project is, its popularity should rapidly increase, not only for the betterment of the embedded CPU community, but for everyone that relies on using a compiler.

References

[1] T. Jamil. RISC versus CISC. *IEEE Potentials*, 14(3):13–16, Aug 1995. doi:10.1109/45.464688.

[2] ARM. *ARM and Thumb-2 Instruction Set*, M edition, Sept 2008.

[3] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2 edition, July 2017.

[4] Xavier Leroy. How I Found a Bug in Intel Skylake Processors, July 2017. URL: http://gallium.inria.fr/blog/intel-skylake-bug/.

[5] R. P. Colwell, C. Y. I. Hitchcock, E. D. Jensen, H. M. Brinkley Sprunt, and C. P. Kollar. Instruction sets and beyond: Computers, complexity, and controversy. *Computer*, 18(9):8–19, Sept 1985. doi:10.1109/MC.1985.1663000.

[6] H. El-Aawar. An application of complexity measures in addressing modes for CISC- and RISC-architectures. In *2008 IEEE International Conference on Industrial Technology*, pages 1–7, April 2008. doi:10.1109/ICIT.2008.4608682.

[7] E. Blem, J. Menon, and K. Sankaralingam. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *2013 IEEE 19th In-*

ternational Symposium on High Performance Computer Architecture (HPCA), pages 1–12, Feb 2013. doi:10.1109/HPCA.2013.6522302.

[8] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*, volume 2. Addison-Wesley Publishing Company, 2007.

[9] L. Ghica and N. Tapus. Optimized retargetable compiler for embedded processors - GCC vs LLVM. In *2015 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 103–108, Sept 2015. doi:10.1109/ICCP.2015.7312613.

[10] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Trans. Program. Lang. Syst.*, 6(4):505–526, October 1984. URL: http://doi.acm.org/10.1145/1780.1783, doi:10.1145/1780.1783.

[11] William Von Hagen. *The Definitive Guide to GCC*. Apress, 2011.

[12] GCC - open source compiler for MSP microcontrollers. URL: http://www.ti.com/tool/msp430-gcc-opensource.

[13] Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[14] Chris Lattner. The name of LLVM, Dec 2011. URL: http://lists.llvm.org/pipermail/llvm-dev/2011-December/046445.html.

[15] Michael Larabel. Google now uses Clang as their production compiler for Chrome Linux builds, Nov 2014. URL: http://www.phoronix.com/scan.php?page=news_item&px=MTg0MTk.

[16] Brooks Davis. HEADS UP: Clang now the default on x86, Nov 2012. URL: https://lists.freebsd.org/pipermail/freebsd-current/2012-November/037610.html.

[17] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*, San Diego, California, Dec 2003.

[18] V. Iyer, A. Kanitkar, P. Dasgupta, and R. Srinivasan. Preventing overflow attacks by memory randomization. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 339–347, Nov 2010. doi:10.1109/ISSRE.2010.22.

[19] Alan M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[20] LLVM Project. The LLVM target-independent code generator, 2017. URL: http://llvm.org/docs/CodeGenerator.html.

[21] LLVM Project. TableGen, 2017. URL: http://llvm.org/docs/TableGen/index.html.

[22] LLVM Project. TableGen backends, 2017. URL: http://llvm.org/docs/TableGen/BackEnds.html.

[23] LLVM Project. Clang: a C language family frontend for LLVM, 2017. URL: http://clang.llvm.org.

[24] Chen Chung-Shu. Creating an LLVM backend for the Cpu0 architecture, 2016. URL: http://jonathan2251.github.io/lbd/index.html.

[25] Fraser Cormack and Pierre-André Saulais. Building an LLVM backend, Oct 2014. URL: http://llvm.org/devmtg/2014-10/Slides/Cormack-BuildingAnLLVMBackend.pdf.

[26] Alex Bradbury. RISC-V Backend, Aug 2016. URL: http://lists.llvm.org/pipermail/llvm-dev/2016-August/103748.html.

[27] Jakob Stoklund. Greedy register allocation in LLVM 3.0, Sept 2011. URL: http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html.

[28] LLVM Project. LLVM language reference manual, 2017. URL: https://llvm.org/docs/LangRef.html.

[29] Sarah L. Harris and David Money Harris. *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufmann, 2016.

Appendix I

Guides

I.1 Building LLVM-CJG

This guide will walk through downloading and building the LLVM tools from source. The paths are relative to the directory you decide to use when starting the guide, unless otherwise specified. At the time of this writing, the working repository for this backend can be found in the `llvm-cjg` repository hosted at https://github.com/connorjan/llvm-cjg, and additional information may be posted to http://connorgoldberg.com.

I.1.1 Downloading LLVM

Even though the working source tree is version controlled through SVN, an official mirror is hosted on GitHub which is what will be used for this guide.

1. Clone the repository into the `src` directory:
   ```
   $ git clone https://github.com/llvm-mirror/llvm.git src
   ```
2. Checkout the LLVM 4.0 branch:
   ```
   $ cd src
   $ git fetch
   $ git checkout release_40
   $ cd ..
   ```

I.1.2 Importing the CJG Source Files

Along with this paper should be a directory named `CJG`. This is the directory that contains all of code specific to the CJG backend. Copy this directory into the LLVM `lib/Target` directory:

```
$ cp -r CJG src/lib/Target/
```

I.1.3 Modifying Existing LLVM Files

Some files in the root of the LLVM tree need to be modified so that the CJG backend can be found and built correctly. Run

```
$ cd src
```

so the diff paths are relative to the root of the LLVM source repository.

1. Add `CJG` to the root cmake configuration:

CMakeLists.txt

```
diff --git a/CMakeLists.txt b/CMakeLists.txt
-- a/CMakeLists.txt
++ b/CMakeLists.txt
@@ -326,8 +326,9 @@ set(LLVM_ALL_TARGETS
  AMDGPU
  ARM
  BPF
+ CJG
  Hexagon
  Lanai
  Mips
  MSP430
  NVPTX
```

2. Add `cjg` to the `Triple::ArchType` enum:

include/llvm/ADT/Triple.h

```
diff --git a/include/llvm/ADT/Triple.h b/include/llvm/ADT/Triple.h
-- a/include/llvm/ADT/Triple.h
++ b/include/llvm/ADT/Triple.h
@@ -94,6 +94,7 @@ public:
    wasm64,         // WebAssembly with 64-bit pointers
    renderscript32, // 32-bit RenderScript
    renderscript64, // 64-bit RenderScript
+   cjg,            // CJG
    LastArchType = renderscript64
  };
  enum SubArchType {
```

3. Add `EM_CJG` to the ELF Machine enum:

include/llvm/Support/ELF.h

```
diff --git a/include/llvm/Support/ELF.h b/include/llvm/Support/ELF.h
-- a/include/llvm/Support/ELF.h
++ b/include/llvm/Support/ELF.h
@@ -310,7 +310,8 @@ enum {
  EM_RISCV = 243,         // RISC-V
  EM_LANAI = 244,         // Lanai 32-bit processor
  EM_BPF = 247,           // Linux kernel bpf virtual machine

+ EM_CJG = 327,           // CJG

  // A request has been made to the maintainer of the official registry for
  // such numbers for an official value for WebAssembly. As soon as one is
```

4. Add `cjg` to the `Triple` class:

lib/Support/Triple.cpp

```
diff --git a/lib/Support/Triple.cpp b/lib/Support/Triple.cpp
-- a/lib/Support/Triple.cpp
++ b/lib/Support/Triple.cpp
@@ -69,6 +69,7 @@ StringRef Triple::getArchTypeName(ArchType Kind) {
  case wasm64:         return "wasm64";
  case renderscript32: return "renderscript32";
  case renderscript64: return "renderscript64";
+ case cjg:            return "cjg";
  }

  llvm_unreachable("Invalid ArchType!");
@@ -140,6 +142,7 @@ StringRef Triple::getArchTypePrefix(ArchType Kind) {

  case riscv32:
  case riscv64:     return "riscv";
+ case cjg:         return "cjg";
  }
}

@@ -298,6 +302,7 @@ Triple::ArchType Triple::getArchTypeForLLVMName(StringRef
↪  Name) {
    .Case("wasm64", wasm64)
    .Case("renderscript32", renderscript32)
    .Case("renderscript64", renderscript64)
+   .Case("cjg", cjg)
```

```
    .Default(UnknownArch);
}

@@ -412,6 +418,7 @@ static Triple::ArchType parseArch(StringRef ArchName) {
    .Case("wasm64", Triple::wasm64)
    .Case("renderscript32", Triple::renderscript32)
    .Case("renderscript64", Triple::renderscript64)
+   .Case("cjg", Triple::cjg)
    .Default(Triple::UnknownArch);

  // Some architectures require special parsing logic just to compute the
@@ -640,6 +648,7 @@ static Triple::ObjectFormatType getDefaultFormat(const Triple
↪  &T) {
  case Triple::wasm32:
  case Triple::wasm64:
  case Triple::xcore:
+ case Triple::cjg:
    return Triple::ELF;

  case Triple::ppc:
@@ -1172,6 +1182,7 @@ static unsigned
↪  getArchPointerBitWidth(llvm::Triple::ArchType Arch) {
  case llvm::Triple::shave:
  case llvm::Triple::wasm32:
  case llvm::Triple::renderscript32:
+ case llvm::Triple::cjg:
    return 32;

  case llvm::Triple::aarch64:
@@ -1251,6 +1263,7 @@ Triple Triple::get32BitArchVariant() const {
  case Triple::shave:
  case Triple::wasm32:
  case Triple::renderscript32:
+ case Triple::cjg:
    // Already 32-bit.
    break;

@@ -1288,6 +1302,7 @@ Triple Triple::get64BitArchVariant() const {
  case Triple::xcore:
  case Triple::sparcel:
  case Triple::shave:
+ case Triple::cjg:
    T.setArch(UnknownArch);
    break;

@@ -1373,6 +1389,7 @@ Triple Triple::getBigEndianArchVariant() const {
  // drop any arch suffixes.
  case Triple::arm:
  case Triple::thumb:
```

```
+ case Triple::cjg:
    T.setArch(UnknownArch);
    break;

@@ -1458,6 +1476,7 @@ bool Triple::isLittleEndian() const {
  case Triple::tcele:
  case Triple::renderscript32:
  case Triple::renderscript64:
+ case Triple::cjg:
    return true;
  default:
    return false;
```

5. Add `CJG` to the cmake Target build configuration:

lib/Target/LLVMBuild.txt

```
diff --git a/lib/Target/LLVMBuild.txt b/lib/Target/LLVMBuild.txt
-- a/lib/Target/LLVMBuild.txt
++ b/lib/Target/LLVMBuild.txt
@@ -24,7 +24,8 @@ subdirectories =
  AArch64
  AVR
  BPF
+ CJG
  Lanai
  Hexagon
  MSP430
  NVPTX
```

Run

```
$ cd ..
```

to return to the root working directory of the guide.

I.1.4 Importing Clang

If you are only using LLVM IR then you can skip this step and go to Section I.1.5. If you want to be able to use C code:

1. Change your current directory into the LLVM tools directory:
   `$ cd src/tools`

2. Clone the Clang repository from GitHub:
   `$ git clone https://github.com/llvm-mirror/clang.git`

3. Checkout the Clang 4.0 branch:
   ```
   $ cd clang
   $ git fetch
   $ git checkout release_40
   ```

Now link the CJG backend into Clang (note: the diff paths are relative the root of the Clang repository):

1. Add the `CJGTargetInfo` class to `Targets.cpp`:

lib/Basic/Targets.cpp

```
diff --git a/lib/Basic/Targets.cpp b/lib/Basic/Targets.cpp
-- a/lib/Basic/Targets.cpp
++ b/lib/Basic/Targets.cpp
@@ -8587,6 +8587,59 @@ public:
      }
  };

+ class CJGTargetInfo : public TargetInfo {
+ public:
+     CJGTargetInfo(const llvm::Triple &Triple, const TargetOptions &):
+         TargetInfo(Triple) {
+             BigEndian = false;
+             NoAsmVariants = true;
+             LongLongAlign = 32;
+             SuitableAlign = 32;
+             DoubleAlign = LongDoubleAlign = 32;
+             SizeType = UnsignedInt;
+             PtrDiffType = SignedInt;
+             IntPtrType = SignedInt;
+             WCharType = UnsignedChar;
+             WIntType = UnsignedInt;
+             UseZeroLengthBitfieldAlignment = true;
+             resetDataLayout("e-m:e-p:32:32-i1:8:32-i8:8:32-i16:16:32-i64:32"
+               "-f64:32-a:0:32-n32");
+         }
+
+     void getTargetDefines(const LangOptions &Opts,
+                           MacroBuilder &Builder) const override {}
+
+     ArrayRef<Builtin::Info> getTargetBuiltins() const override {
+       return None;
+     }
+
+     BuiltinVaListKind getBuiltinVaListKind() const override {
+         return TargetInfo::VoidPtrBuiltinVaList;
+     }
+
```

```
+     const char *getClobbers() const override {
+         return "";
+     }
+
+     ArrayRef<const char *> getGCCRegNames() const override {
+       return None;
+     }
+
+     ArrayRef<TargetInfo::GCCRegAlias> getGCCRegAliases() const override {
+         return None;
+     }
+
+     bool validateAsmConstraint(const char *&Name,
+                                TargetInfo::ConstraintInfo &Info) const override {
+         return false;
+     }
+
+     int getEHDataRegisterNumber(unsigned RegNo) const override {
+         // R0=ExceptionPointerRegister R1=ExceptionSelectorRegister
+         return -1;
+     }
+ };
+
  } // end anonymous namespace

  //===----------------------------------------------------------------------===//
@@ -9044,4 +9097,7 @@ static TargetInfo *AllocateTarget(const llvm::Triple
↪  &Triple,
    case llvm::Triple::renderscript64:
      return new LinuxTargetInfo<RenderScript64TargetInfo>(Triple, Opts);
+
+   case llvm::Triple::cjg:
+     return new CJGTargetInfo(Triple, Opts);
    }
  }
```

2. Add the `CJGABIInfo` class to `TargetInfo.cpp`:

lib/CodeGen/TargetInfo.cpp

```
diff --git a/lib/CodeGen/TargetInfo.cpp b/lib/CodeGen/TargetInfo.cpp
index ec0aa16..1ec7455 100644
-- a/lib/CodeGen/TargetInfo.cpp
++ b/lib/CodeGen/TargetInfo.cpp
@@ -8349,8 +8349,25 @@ public:
    }
    return false;
  }
```

```
+
+
+ //===----------------------------------------------------------------------===//
+ // CJG ABI Implementation
+ //===----------------------------------------------------------------------===//
+ namespace {
+ class CJGABIInfo : public DefaultABIInfo {
+ public:
+     CJGABIInfo(CodeGen::CodeGenTypes &CGT) : DefaultABIInfo(CGT) {}
+ };
+
+ class CJGTargetCodeGenInfo : public TargetCodeGenInfo {
+ public:
+     CJGTargetCodeGenInfo(CodeGenTypes &CGT)
+     : TargetCodeGenInfo(new CJGABIInfo(CGT)) {}
+ };
+ } // end anonymous namespace


  //===----------------------------------------------------------------------===//
  // Driver code
  //===----------------------------------------------------------------------===//
@@ -8536,5 +8554,7 @@ const TargetCodeGenInfo
↪  &CodeGenModule::getTargetCodeGenInfo() {
    case llvm::Triple::spir:
    case llvm::Triple::spir64:
      return SetCGInfo(new SPIRTargetCodeGenInfo(Types));
+   case llvm::Triple::cjg:
+     return SetCGInfo(new CJGTargetCodeGenInfo(Types));
    }
  }
```

Run

```
$ cd ../../../
```

to return to the root working directory of the guide.

I.1.5 Building the Project

1. Make the build directory:
   ```
   $ mkdir build
   $ cd build
   ```

2. Set up the build files:
   Note: the following flags can be added to build the documentation:
   `-DLLVM_ENABLE_DOXYGEN=True -DLLVM_DOXYGEN_SVG=True`

(a) macOS only (for Xcode capabilities):
```
$ cmake -G "Xcode" -DCMAKE_BUILD_TYPE:STRING=DEBUG \
-DLLVM_TARGETS_TO_BUILD:STRING=CJG ../src
```

(b) Linux or macOS:
```
$ cmake -G "Unix Makefiles" -DCMAKE_BUILD_TYPE:STRING=DEBUG \
-DLLVM_TARGETS_TO_BUILD:STRING=CJG ../src
```

3. Build the project:

(a) If the `"Xcode"` cmake generator was used then the project can either be built two ways:

i. Opening the generated Xcode project: `LLVM.xcodeproj` and then running the build command

ii. Building the Xcode project from the command line with:
```
$ xcodebuild -project "LLVM.xcodeproj"
```

iii. View the compiled binaries in the `Debug/bin/` directory.

(b) If the `"Unix"` cmake generator was used then the project can be built by running make:
```
$ make
```
Note: make can be used with the `"-jn"` flag, where n is the number of cores on your build machine to parallelize the build process (*e.g.* `make -j4`).

(c) View the compiled binaries in the `bin/` directory.

I.1.6 Usage

First change your current directory to the directory where the compiled binaries are located (explained in step 3 of Section I.1.5).

I.1.6.1 Using llc

The input for each of the commands in this section is an example LLVM IR code file called `function.ll`.

1. **LLVM IR to CJG Assembly:**
```
$ ./llc -march cjg -o function.s function.ll
```

2. **LLVM IR to CJG Machine Code:**
```
$ ./llc -march cjg -filetype=obj -o function.o function.ll
```
Extracting the machine code from the object file is explained in Section I.1.6.3.

To enable all of the debug messages, use the

```
-debug
```

flag when running `llc`. To enable the printing of the code representation after every pass in the backend, use the

```
-print-after-all
```

flag when running `llc`.

I.1.6.2 Using Clang

Only available if the steps explained in Section I.1.4 were performed. The input for each of the Clang commands in this section is an example C file called `function.c` containing a single C function.

1. **C to LLVM IR:**
   ```
   $ ./clang -cc1 -triple cjg-unknown-unknown -o function.ll function.c -emit-llvm
   ```
2. **C to CJG Assembly:**
   ```
   $ ./clang -cc1 -triple cjg-unknown-unknown -S -o function.s function.c
   ```
3. **C to CJG Machine Code:**
   ```
   $ ./clang -cc1 -triple cjg-unknown-unknown -o function.o function.c
   ```
   Extracting the machine code from the object file is explained in Section I.1.6.3.
   Note: Trying to emit an object file from clang is currently unstable and may not work 100% of the time. Instead use `clang` to emit LLVM IR code and then use `llc` to write the object file.

I.1.6.3 Using ELF to Memory

To extract the machine code from an ELF object file using `elf2mem` as discussed in Section 5.3.2:

```
$ elf2mem -s .text -o function.mem function.o
```

I.2 LLVM Backend Directory Tree

This shows the directory tree for CJG LLVM backend:

```
lib/Target/CJG/
├── CJG.h
├── CJG.td
├── CJGAsmPrinter.cpp
├── CJGCallingConv.td
├── CJGFrameLowering.cpp
├── CJGFrameLowering.h
├── CJGISelDAGToDAG.cpp
├── CJGISelLowering.cpp
├── CJGISelLowering.h
├── CJGInstrFormats.td
├── CJGInstrInfo.cpp
├── CJGInstrInfo.h
├── CJGInstrInfo.td
├── CJGMCInstLower.cpp
├── CJGMCInstLower.h
├── CJGMachineFunctionInfo.cpp
├── CJGMachineFunctionInfo.h
├── CJGRegisterInfo.cpp
├── CJGRegisterInfo.h
├── CJGRegisterInfo.td
├── CJGSubtarget.cpp
├── CJGSubtarget.h
├── CJGTargetMachine.cpp
├── CJGTargetMachine.h
├── CMakeLists.txt
├── InstPrinter/
│   ├── CJGInstPrinter.cpp
│   ├── CJGInstPrinter.h
│   ├── CMakeLists.txt
│   └── LLVMBuild.txt
├── LLVMBuild.txt
├── MCTargetDesc/
│   ├── CJGAsmBackend.cpp
│   ├── CJGELFObjectWriter.cpp
```

```
    ├── CJGFixupKinds.h
    ├── CJGMCAsmInfo.cpp
    ├── CJGMCAsmInfo.h
    ├── CJGMCCodeEmitter.cpp
    ├── CJGMCTargetDesc.cpp
    ├── CJGMCTargetDesc.h
    ├── CMakeLists.txt
    └── LLVMBuild.txt
└── TargetInfo/
    ├── CJGTargetInfo.cpp
    ├── CMakeLists.txt
    └── LLVMBuild.txt
```

Appendix II

Source Code

II.1 CJG RISC CPU RTL

II.1.1 Opcodes Header

cjg_opcodes.vh

```
// Opcodes
`define LD_IC   5'h00   // Load
`define ST_IC   5'h01   // Store
`define CPY_IC  5'h02   // Copy
`define PUSH_IC 5'h03   // Push onto stack
`define POP_IC  5'h04   // Pop off of stack
`define JMP_IC  5'h05   // Jumps
`define CALL_IC 5'h06   // Call
`define RET_IC  5'h07   // Return and RETI
`define ADD_IC  5'h08   // Addition
`define SUB_IC  5'h09   // Subtract
`define CMP_IC  5'h0A   // Compare
`define NOT_IC  5'h0B   // Bitwise NOT
`define AND_IC  5'h0C   // Bitwise AND
`define BIC_IC  5'h0D   // Bit clear ~&=
`define OR_IC   5'h0E   // Bitwise OR
`define XOR_IC  5'h0F   // Bitwise XOR
`define RS_IC   5'h10   // Rotate/Shift

`define MUL_IC  5'h1A   // Signed multiplication
`define DIV_IC  5'h1B   // Unsigned division

```

```
23 `define INT_IC  5'h1F   // Interrupt
24
25 // ALU States
26 `define ADD_ALU 4'h0    // Signed Add
27 `define SUB_ALU 4'h1    // Signed Subtract
28 `define AND_ALU 4'h2    // Logical AND
29 `define BIC_ALU 4'h3    // Logical BIC
30 `define OR_ALU  4'h4    // Logical OR
31 `define NOT_ALU 4'h5    // Logical Invert
32 `define XOR_ALU 4'h6    // Logical XOR
33 `define NOP_ALU 4'h7    // No operation
34 `define MUL_ALU 4'h8    // Signed multiplication
35 `define DIV_ALU 4'h9    // Signed division
36
37 // Shifter states
38 `define SRL_SHIFT 3'h0  // shift right logical
39 `define SLL_SHIFT 3'h1  // shift left logical
40 `define SRA_SHIFT 3'h2  // shift right arithmetic
41 `define RTR_SHIFT 3'h4  // rotate right
42 `define RTL_SHIFT 3'h5  // rotate left
43 `define RRC_SHIFT 3'h6  // rotate right through carry
44 `define RLC_SHIFT 3'h7  // rotate left through carry
```

II.1.2 Definitions Header

cjg_definitions.vh

```
1  // Instruction word slices
2  `define OPCODE 31:27
3  `define REG_I 26:22
4  `define REG_J 21:17
5  `define REG_K 16:12
6  `define ALU_CONSTANT 16:1
7  `define ALU_CONSTANT_MSB 16
8  `define ALU_CONTROL 0
9  `define DT_CONTROL 16
10 `define DT_CONSTANT 15:0
11 `define DT_CONSTANT_MSB 15
12 `define JMP_CODE 21:18
13 `define JMP_ADDR 15:0
14 `define JMP_CONTROL 16
15 `define RS_CONTROL 0
16 `define RS_OPCODE 3:1
17 `define RS_CONSTANT 16:11
18
19 // Jump codes
20 `define JU  4'b0000
```

```
`define JC  4'b1000
`define JN  4'b0100
`define JV  4'b0010
`define JZ  4'b0001
`define JNC 4'b0111
`define JNN 4'b1011
`define JNV 4'b1101
`define JNZ 4'b1110
`define JGE 4'b0110
`define JL  4'b1001

// special register file registers
`define REG_SR 5'h0 // status register
`define REG_PC 5'h1 // program counter
`define REG_SP 5'h2 // stack pointer

// Status bit index in the status register / RF[0]
`define SR_C    5'd0
`define SR_N    5'd1
`define SR_V    5'd2
`define SR_Z    5'd3
`define SR_GE   5'd4
`define SR_L    5'd5

// MMIO
`define MMIO_START_ADDR 16'hFF00
`define MMIO_GPIO_OUT 16'hFFF0
`define MMIO_GPIO_IN 16'hFFF0
```

II.1.3 Pipeline

cjg_risc.v

```
/*
 * Title: cjg_risc
 * Author: Connor Goldberg
 *
 */

`include "src/cjg_definitions.vh"
`include "src/cjg_opcodes.vh"

// Any instruction with a writeback operation
```

```
`define WB_INSTRUCTION(mc) (opcode[mc] == `LD_IC || opcode[mc] == `CPY_IC || opcode[mc] == `POP_IC || opcode[mc] == `ADD_IC || opcode[mc] == `SUB_IC || opcode[mc] == `CMP_IC || opcode[mc] == `NOT_IC || opcode[mc] == `AND_IC || opcode[mc] == `BIC_IC || opcode[mc] == `OR_IC || opcode[mc] == `XOR_IC || opcode[mc] == `RS_IC || opcode[mc] == `MUL_IC || opcode[mc] == `DIV_IC)

// ALU instructions
`define ALU_INSTRUCTION(mc) (opcode[mc] == `CPY_IC || opcode[mc] == `ADD_IC || opcode[mc] == `SUB_IC || opcode[mc] == `CMP_IC || opcode[mc] == `NOT_IC || opcode[mc] == `AND_IC || opcode[mc] == `BIC_IC || opcode[mc] == `OR_IC || opcode[mc] == `XOR_IC || opcode[mc] == `MUL_IC || opcode[mc] == `DIV_IC)

// Stack instructions
`define STACK_INSTRUCTION(mc) (opcode[mc] == `PUSH_IC) || (opcode[mc] == `POP_IC)

`define LOAD_MMIO(dest,bits,expr) \
if (dm_address < `MMIO_START_ADDR) begin \
  dest <= dm_out[bits] expr; \
end \
else begin \
  case (dm_address) \
    `MMIO_GPIO_IN: begin \
      dest <= gpio_in[bits] expr; \
    end \
    default: begin \
      dest <= temp_wb[bits] expr; \
    end \
  endcase \
end

module cjg_risc (
  // system inputs
  input reset,                    // system reset
  input clk,                      // system clock
  input [31:0] gpio_in,           // gpio inputs
  input [3:0] ext_interrupt_bus,  //external interrupts

  // system outputs
  output reg [31:0] gpio_out,     // gpio outputs

  // program memory
  input [31:0] pm_out,            // program memory output data
  output [15:0] pm_address,       // program memory address

  // data memory
  input [31:0] dm_out,            // data memory output
  output reg [31:0] dm_data,      // data memory input data
  output reg dm_wren,             // data memory write enable
  output reg [15:0] dm_address,   // data memory address
```

```
53
54    // generated clock phases
55    output clk_p1,                    // clock phase 0
56    output clk_p2,                    // clock phase 1
57
58    // dft
59    input scan_in0,
60    input scan_en,
61    input test_mode,
62    output scan_out0
63  );
64
65  // integer for resetting arrays
66  integer i;
67
68  // register file
69  reg[31:0] reg_file[31:0];
70
71  // program counter regsiter (program memory address)
72  assign pm_address = reg_file[`REG_PC][15:0];
73
74  // temp address for jumps/calls
75  reg[15:0] temp_address;
76
77  // pipelined instruction registers
78  reg[31:0] instruction_word[3:1];
79
80  // address storage for each instruction
81  reg[13:0] instruction_addr[3:1];
82
83  // opcode slices
84  reg[4:0] opcode[3:0];
85
86  // TODO: is this even ok? 2d wires dont seem to work in simvision
87  always @(instruction_word[3] or instruction_word[2] or instruction_word[1] or pm_out)
    ↪  begin
88    opcode[0] = pm_out[`OPCODE];
89    opcode[1] = instruction_word[1][`OPCODE];
90    opcode[2] = instruction_word[2][`OPCODE];
91    opcode[3] = instruction_word[3][`OPCODE];
92  end
93
94  // stall signals
95  reg[3:0] stall_cycles;
96  reg stall[3:0];
97
98  // temp writeback register
99  reg[31:0] temp_wb; // general purpose
100 reg[31:0] temp_sp; // stack pointer
```

```
101
102 // data stack stuff
103 reg[31:0] data_stack_data;
104 reg[5:0] data_stack_addr;
105 reg data_stack_push;
106 reg data_stack_pop;
107 wire[31:0] data_stack_out;
108
109 // call stack stuff
110 reg[31:0] call_stack_data;
111 reg call_stack_push;
112 reg call_stack_pop;
113 wire[31:0] call_stack_out;
114
115 // ALU stuff
116 reg[31:0] alu_a, alu_b, temp_sr;
117 reg[3:0] alu_opcode;
118 wire[31:0] alu_result;
119 wire alu_c, alu_n, alu_v, alu_z;
120
121 // Shifter stuff
122 reg[31:0] shifter_operand;
123 reg[5:0] shifter_modifier;
124 reg shifter_carry_in;
125 reg[2:0] shifter_opcode;
126 wire[31:0] shifter_result;
127 wire shifter_carry_out;
128
129 // Clock phase generator
130 cjg_clkgen clkgen(
131   .reset(reset),
132   .clk(clk),
133   .clk_p1(clk_p1),
134   .clk_p2(clk_p2),
135
136   // dft
137   .scan_in0(scan_in0),
138   .scan_en(scan_en),
139   .test_mode(test_mode),
140   .scan_out0(scan_out0)
141 );
142
143 // Data Stack
144 cjg_mem_stack #(.DEPTH(64), .ADDRW(6)) data_stack (
145   // inputs
146   .clk(clk_p2),
147   .reset(reset),
148   .d(data_stack_data),
149   .addr(data_stack_addr),
```

```
150     .push(data_stack_push),
151     .pop(data_stack_pop),
152 
153     // output
154     .q(data_stack_out),
155 
156     // dft
157     .scan_in0(scan_in0),
158     .scan_en(scan_en),
159     .test_mode(test_mode),
160     .scan_out0(scan_out0)
161 );
162 
163 // Call Stack
164 cjg_stack #(.DEPTH(64)) call_stack (
165     // inputs
166     .clk(clk_p2),
167     .reset(reset),
168     .d(call_stack_data),
169     .push(call_stack_push),
170     .pop(call_stack_pop),
171 
172     // output
173     .q(call_stack_out),
174 
175     // dft
176     .scan_in0(scan_in0),
177     .scan_en(scan_en),
178     .test_mode(test_mode),
179     .scan_out0(scan_out0)
180 );
181 
182 // ALU
183 cjg_alu alu (
184     // dft
185     .reset(reset),
186     .clk(clk),
187     .scan_in0(scan_in0),
188     .scan_en(scan_en),
189     .test_mode(test_mode),
190     .scan_out0(scan_out0),
191 
192     // inputs
193     .a(alu_a),
194     .b(alu_b),
195     .opcode(alu_opcode),
196 
197     // outputs
198     .result(alu_result),
```

```
199   .c(alu_c),
200   .n(alu_n),
201   .v(alu_v),
202   .z(alu_z)
203 );
204
205 // Shifter and rotater
206 cjg_shifter shifter (
207   // dft
208   .reset(reset),
209   .clk(clk),
210   .scan_in0(scan_in0),
211   .scan_en(scan_en),
212   .test_mode(test_mode),
213   .scan_out0(scan_out0),
214
215   // inputs
216   .operand(shifter_operand),
217   .carry_in(shifter_carry_in),
218   .modifier(shifter_modifier),
219   .opcode(shifter_opcode),
220
221   // outputs
222   .result(shifter_result),
223   .carry_out(shifter_carry_out)
224 );
225
226
227 // Here we go
228
229 always @(posedge clk_p1 or negedge reset) begin
230   if (~reset) begin
231     // reset
232     reset_all;
233   end // if (~reset)
234   else begin
235     // Main code
236
237     // process stall signals
238     stall[3] <= stall[2];
239     stall[2] <= stall[1];
240     stall[1] <= stall[0];
241
242     if (stall_cycles != 0) begin
243       stall[0] <= 1'b1;
244       stall_cycles <= stall_cycles - 1'b1;
245     end
246     else begin
247       stall[0] <= 1'b0;
```

```
248     end
249
250     // Machine cycle 3
251     // writeback
252     if (stall[3] == 1'b0) begin
253
254       case (opcode[3])
255         `ADD_IC, `SUB_IC, `NOT_IC, `AND_IC, `BIC_IC, `OR_IC, `XOR_IC, `CPY_IC, `LD_IC,
          ↪  `RS_IC, `MUL_IC, `DIV_IC: begin
256           if (instruction_word[3][`REG_I] == `REG_PC) begin
257             // Do not allow writing to the program counter
258             reg_file[`REG_PC] <= reg_file[`REG_PC];
259           end
260           else begin
261             reg_file[instruction_word[3][`REG_I]] <= temp_wb;
262           end
263         end
264
265         `PUSH_IC: begin
266           reg_file[`REG_SP] <= temp_sp; // incremented stack pointer
267         end
268
269         `POP_IC: begin
270           reg_file[`REG_SP] <= temp_sp; // decremented stack pointer
271           reg_file[instruction_word[3][`REG_I]] <= temp_wb;
272           data_stack_pop <= 1'b0;
273         end
274
275         `ST_IC: begin
276           dm_wren <= 1'b0;
277         end
278
279         `JMP_IC: begin
280           // check the status register
281           case (instruction_word[3][`JMP_CODE])
282
283             `JU: begin
284               reg_file[`REG_PC] <= {16'h0, temp_address};
285             end
286
287             `JC: begin
288               if (reg_file[`REG_SR][`SR_C] == 1'b1) begin
289                 reg_file[`REG_PC] <= {16'h0, temp_address};
290               end
291               else begin
292                 reg_file[`REG_PC] <= reg_file[`REG_PC];
293               end
294             end
295
```

```
296             `JN: begin
297               if (reg_file[`REG_SR][`SR_N] == 1'b1) begin
298                 reg_file[`REG_PC] <= {16'h0, temp_address};
299               end
300               else begin
301                 reg_file[`REG_PC] <= reg_file[`REG_PC];
302               end
303             end
304
305             `JV: begin
306               if (reg_file[`REG_SR][`SR_V] == 1'b1) begin
307                 reg_file[`REG_PC] <= {16'h0, temp_address};
308               end
309               else begin
310                 reg_file[`REG_PC] <= reg_file[`REG_PC];
311               end
312             end
313
314             `JZ: begin
315               if (reg_file[`REG_SR][`SR_Z] == 1'b1) begin
316                 reg_file[`REG_PC] <= {16'h0, temp_address};
317               end
318               else begin
319                 reg_file[`REG_PC] <= reg_file[`REG_PC];
320               end
321             end
322
323             `JNC: begin
324               if (reg_file[`REG_SR][`SR_C] == 1'b0) begin
325                 reg_file[`REG_PC] <= {16'h0, temp_address};
326               end
327               else begin
328                 reg_file[`REG_PC] <= reg_file[`REG_PC];
329               end
330             end
331
332             `JNN: begin
333               if (reg_file[`REG_SR][`SR_N] == 1'b0) begin
334                 reg_file[`REG_PC] <= {16'h0, temp_address};
335               end
336               else begin
337                 reg_file[`REG_PC] <= reg_file[`REG_PC];
338               end
339             end
340
341             `JNV: begin
342               if (reg_file[`REG_SR][`SR_V] == 1'b0) begin
343                 reg_file[`REG_PC] <= {16'h0, temp_address};
344               end
```

```
345               else begin
346                 reg_file[`REG_PC] <= reg_file[`REG_PC];
347               end
348             end
349
350             `JNZ: begin
351               if (reg_file[`REG_SR][`SR_Z] == 1'b0) begin
352                 reg_file[`REG_PC] <= {16'h0, temp_address};
353               end
354               else begin
355                 reg_file[`REG_PC] <= reg_file[`REG_PC];
356               end
357             end
358
359             `JGE: begin
360               if (reg_file[`REG_SR][`SR_GE] == 1'b1) begin
361                 reg_file[`REG_PC] <= {16'h0, temp_address};
362               end
363               else begin
364                 reg_file[`REG_PC] <= reg_file[`REG_PC];
365               end
366             end
367
368             `JL: begin
369               if (reg_file[`REG_SR][`SR_L] == 1'b1) begin
370                 reg_file[`REG_PC] <= {16'h0, temp_address};
371               end
372               else begin
373                 reg_file[`REG_PC] <= reg_file[`REG_PC];
374               end
375             end
376
377             default: begin
378               reg_file[`REG_PC] <= reg_file[`REG_PC];
379             end
380           endcase // instruction_word[3][`JMP_CODE]
381
382         end // JMP_IC
383
384         `CALL_IC: begin
385           // jump to the routine address
386           call_stack_push <= 1'b0;
387           reg_file[`REG_PC] <= {16'h0, temp_address};
388         end
389
390         `RET_IC: begin
391           // pop the program counter
392           call_stack_pop <= 1'b0;
393           reg_file[`REG_PC] <= {16'h0, temp_address};
```

```
394         end
395
396         default: begin
397         end
398       endcase // opcode[3]
399
400       case (opcode[3])
401         `ADD_IC, `SUB_IC, `CMP_IC, `NOT_IC, `AND_IC, `BIC_IC, `OR_IC, `XOR_IC, `RS_IC,
          ↪  `MUL_IC, `DIV_IC: begin
402           // set the status register from the alu output
403           reg_file[`REG_SR] <= temp_sr;
404         end
405
406         default: begin
407           reg_file[`REG_SR] <= reg_file[`REG_SR];
408         end
409       endcase // opcode[3]
410
411     end // if (stall[3] == 1'b0)
412
413     // Machine cycle 2
414     // execution
415     if (stall[2] == 1'b0) begin
416
417       case (opcode[2])
418         `ADD_IC, `SUB_IC, `CMP_IC, `NOT_IC, `AND_IC, `BIC_IC, `OR_IC, `XOR_IC, `CPY_IC,
          ↪  `MUL_IC, `DIV_IC: begin
419           // set temp ALU out
420           temp_wb <= alu_result;
421
422           // Set status register
423           if (instruction_word[3][`REG_I] == `REG_SR && `WB_INSTRUCTION(3)) begin
424             // data forward from the status register
425             temp_sr <= {temp_wb[31:6], alu_n, ~alu_n, alu_z, alu_v, alu_n, alu_c};
426           end
427           else begin
428             // take the current status register
429             temp_sr <= {reg_file[`REG_SR][31:6], alu_n, ~alu_n, alu_z, alu_v, alu_n,
              ↪  alu_c};
430           end
431           // TODO: data forward from other sources in mc3
432         end
433
434         `RS_IC: begin
435           // grab the output from the shifter
436           temp_wb <= shifter_result;
437
438           // if rotating through carry, set the new carry value
```

```
439             if ((instruction_word[2][`RS_OPCODE] == `RRC_SHIFT) ||
                ↪  (instruction_word[2][`RS_OPCODE] == `RLC_SHIFT)) begin
440               // Set status register
441               if (instruction_word[3][`REG_I] == `REG_SR && `WB_INSTRUCTION(3)) begin
442                 // data forward from the status register
443                 temp_sr <= {temp_wb[31:1], shifter_carry_out};
444               end
445               else begin
446                 // take the current status register
447                 temp_sr <= {reg_file[`REG_SR][31:1], shifter_carry_out};
448               end
449             end
450             else begin
451               // dont change the status register
452               temp_sr <= reg_file[`REG_SR];
453             end
454           end
455
456           `PUSH_IC: begin
457             temp_sp <= alu_result; // incremented Stack Pointer
458             data_stack_push <= 1'b0;
459           end
460
461           `POP_IC: begin
462             // data_stack_pop <= 1'b1;
463             // data_stack_pop <= 1'b0;
464             temp_sp <= alu_result; // decremented Stack Pointer
465             temp_wb <= data_stack_out;
466           end
467
468           `LD_IC: begin
469             `LOAD_MMIO(temp_wb,31:0,)
470           end
471
472           `ST_IC: begin
473             if (dm_address < `MMIO_START_ADDR) begin
474               // enable write if not mmio
475               dm_wren <= 1'b1;
476             end
477             else begin
478               // write to mmio
479               dm_wren <= 1'b0;
480
481               case (dm_address)
482
483                 `MMIO_GPIO_OUT: begin
484                   gpio_out <= dm_data;
485                 end
486
```

```
487               default: begin
488               end
489
490             endcase // dm_address
491           end
492         end
493
494         `JMP_IC: begin
495           // Do nothing?
496         end
497
498         `CALL_IC: begin
499           // push the status register onto the stack
500           call_stack_push <= 1'b1;
501           call_stack_data <= reg_file[`REG_SR];
502         end
503
504         `RET_IC: begin
505           // pop the program counter
506           call_stack_pop <= 1'b1;
507           temp_address <= call_stack_out[15:0];
508         end
509
510         default: begin
511         end
512       endcase // opcode[2]
513
514       instruction_word[3] <= instruction_word[2];
515       instruction_addr[3] <= instruction_addr[2];
516     end // if (stall[2] == 1'b0)
517
518     // Machine cycle 1
519     // operand fetch
520     if (stall[1] == 1'b0) begin
521
522       case (opcode[1])
523         `ADD_IC, `SUB_IC, `CMP_IC, `NOT_IC, `AND_IC, `BIC_IC, `OR_IC, `XOR_IC, `MUL_IC,
          ↪  `DIV_IC: begin
524
525           // set alu_a
526           if ((instruction_word[1][`REG_J] == instruction_word[2][`REG_I]) &&
           ↪  `WB_INSTRUCTION(2) && !stall[2]) begin
527             // data forward from mc2
528             if (`ALU_INSTRUCTION(2)) begin
529               // data forward from alu output
530               alu_a <= alu_result;
531             end
532             else if (opcode[2] == `POP_IC) begin
533               alu_a <= data_stack_out;
```

```
534             end
535             else if (opcode[2] == `LD_IC) begin
536               `LOAD_MMIO(alu_a,31:0,)
537             end
538             else if (opcode[2] == `RS_IC) begin
539               alu_a <= shifter_result;
540             end
541             // TODO: data forward from other wb sources in mc2
542             else begin
543               // no data forwarding
544               alu_a <= reg_file[instruction_word[1][`REG_J]];
545             end
546           end
547           else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(2) &&
          ↪  !stall[2]) begin
548             // data forward from the increment/decrement of the stack pointer
549             alu_a <= alu_result;
550           end
551           else if ((instruction_word[1][`REG_J] == instruction_word[3][`REG_I]) &&
          ↪  `WB_INSTRUCTION(3) && !stall[3]) begin
552             // data forward from mc3
553             alu_a <= temp_wb;
554             // TODO: data forward from other wb sources in mc3
555           end
556           else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(3) &&
          ↪  !stall[3]) begin
557             // data forward from the increment/decrement of the stack pointer
558             alu_a <= temp_sp;
559           end
560           else begin
561             // no data forwarding
562             alu_a <= reg_file[instruction_word[1][`REG_J]];
563           end
564
565           // set alu_b
566           if (instruction_word[1][`ALU_CONTROL] == 1'b1) begin
567             // constant operand
568             alu_b <= {{16{instruction_word[1][`ALU_CONSTANT_MSB]}},
              ↪  instruction_word[1][`ALU_CONSTANT]}; // sign extend constant
569           end
570           else if ((instruction_word[1][`REG_K] == instruction_word[2][`REG_I]) &&
          ↪  `WB_INSTRUCTION(2) && !stall[2]) begin
571             //data forward from mc2
572             if (`ALU_INSTRUCTION(2)) begin
573               alu_b <= alu_result;
574             end
575             else if (opcode[2] == `POP_IC) begin
576               alu_b <= data_stack_out;
577             end
```

```
578             else if (opcode[2] == `LD_IC) begin
579               `LOAD_MMIO(alu_b,31:0,)
580             end
581             else if (opcode[2] == `RS_IC) begin
582               alu_b <= shifter_result;
583             end
584             // TODO: data forward from other wb sources in mc2
585             else begin
586               // no data forwarding
587               alu_b <= reg_file[instruction_word[1][`REG_K]];
588             end
589           end
590           else if (instruction_word[1][`REG_K] == `REG_SP && `STACK_INSTRUCTION(2) &&
              ↪  !stall[2]) begin
591             // data forward from the increment/decrement of the stack pointer
592             alu_b <= alu_result;
593           end
594           else if ((instruction_word[1][`REG_K] == instruction_word[3][`REG_I]) &&
              ↪  `WB_INSTRUCTION(3) && !stall[3]) begin
595             // data forward from mc3
596             alu_b <= temp_wb;
597             // TODO: data forward from other wb sources in mc3
598           end
599           else if (instruction_word[1][`REG_K] == `REG_SP && `STACK_INSTRUCTION(3) &&
              ↪  !stall[3]) begin
600             // data forward from the increment/decrement of the stack pointer
601             alu_b <= temp_sp;
602           end
603           else begin
604             // no data forwarding
605             alu_b <= reg_file[instruction_word[1][`REG_K]];
606           end
607         end // `ADD_IC, `SUB_IC, `CMP_IC, `NOT_IC, `AND_IC, `BIC_IC, `OR_IC, `XOR_IC
608
609         `CPY_IC: begin
610           // set source alu_a
611           if (instruction_word[1][`DT_CONTROL] == 1'b1) begin
612             // copy from constant
613             alu_a <= {{16{instruction_word[1][`DT_CONSTANT_MSB]}},
              ↪  instruction_word[1][`DT_CONSTANT]}; // sign extend constant
614           end
615           else if ((instruction_word[1][`REG_J] == instruction_word[2][`REG_I]) &&
              ↪  `WB_INSTRUCTION(2) && !stall[2]) begin
616             // data forward from mc2
617             if (`ALU_INSTRUCTION(2)) begin
618               alu_a <= alu_result;
619             end
620             else if (opcode[2] == `POP_IC) begin
621               alu_a <= data_stack_out;
```

```
622             end
623             else if (opcode[2] == `LD_IC) begin
624               `LOAD_MMIO(alu_a,31:0,)
625             end
626             else if (opcode[2] == `RS_IC) begin
627               alu_a <= shifter_result;
628             end
629             // TODO: data forward from other wb sources in mc2
630             else begin
631               // no data forwarding
632               alu_a <= reg_file[instruction_word[1][`REG_J]];
633             end
634           end
635           else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(2) &&
              ↪  !stall[2]) begin
636             // data forward from the increment/decrement of the stack pointer
637             alu_a <= alu_result;
638           end
639           else if ((instruction_word[1][`REG_J] == instruction_word[3][`REG_I]) &&
              ↪  `WB_INSTRUCTION(3) && !stall[3]) begin
640             // data forward from mc3
641             alu_a <= temp_wb;
642             // TODO: data forward from other wb sources in mc3
643           end
644           else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(3) &&
              ↪  !stall[3]) begin
645             // data forward from the increment/decrement of the stack pointer
646             alu_a <= temp_sp;
647           end
648           else begin
649             // no data forwarding
650             alu_a <= reg_file[instruction_word[1][`REG_J]];
651           end
652
653           // alu_b unused for cpy so just keep it the same
654           alu_b <= alu_b;
655         end // `CPY_IC
656
657         `RS_IC: begin
658           // set the opcode
659           shifter_opcode <= instruction_word[1][`RS_OPCODE];
660
661           // set the operand
662           if ((instruction_word[1][`REG_J] == instruction_word[2][`REG_I]) &&
              ↪  `WB_INSTRUCTION(2) && !stall[2]) begin
663             // data forward from mc2
664             if (`ALU_INSTRUCTION(2)) begin
665               shifter_operand <= alu_result;
666             end
```

```
667             else if (opcode[2] == `POP_IC) begin
668               shifter_operand <= data_stack_out;
669             end
670             else if (opcode[2] == `LD_IC) begin
671               `LOAD_MMIO(shifter_operand,31:0,)
672             end
673             else if (opcode[2] == `RS_IC) begin
674               shifter_operand <= shifter_result;
675             end
676             // TODO: data forward from other wb sources in mc2
677             else begin
678               // no data forwarding
679                shifter_operand <= reg_file[instruction_word[1][`REG_J]];
680             end
681           end
682           else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(2) &&
      ↪  !stall[2]) begin
683             // data forward from the increment/decrement of the stack pointer
684             shifter_operand <= alu_result;
685           end
686           else if ((instruction_word[1][`REG_J] == instruction_word[3][`REG_I]) &&
      ↪  `WB_INSTRUCTION(3) && !stall[3]) begin
687             // data forward from mc3
688             shifter_operand <= temp_wb;
689             // TODO: data forward from other wb sources in mc3
690           end
691           else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(3) &&
      ↪  !stall[3]) begin
692             // data forward from the increment/decrement of the stack pointer
693             shifter_operand <= temp_sp;
694           end
695           else begin
696             // no data forwarding
697             shifter_operand <= reg_file[instruction_word[1][`REG_J]];
698           end
699
700           // set the modifier
701           if (instruction_word[1][`RS_CONTROL] == 1'b1) begin
702             // copy from constant
703             shifter_modifier <= instruction_word[1][`RS_CONSTANT];
704           end
705           else if ((instruction_word[1][`REG_K] == instruction_word[2][`REG_I]) &&
      ↪  `WB_INSTRUCTION(2) && !stall[2])  begin
706             // data forward from mc2
707             if (`ALU_INSTRUCTION(2)) begin
708               shifter_modifier <= alu_result[5:0];
709             end
710             else if (opcode[2] == `POP_IC) begin
711               shifter_modifier <= data_stack_out[5:0];
```

```
712             end
713             else if (opcode[2] == `LD_IC) begin
714               `LOAD_MMIO(shifter_modifier,5:0,)
715             end
716             else if (opcode[2] == `RS_IC) begin
717               shifter_modifier <= shifter_result[5:0];
718             end
719             // TODO: data forward from other wb sources in mc2
720             else begin
721               // no data forwarding
722               shifter_modifier <= reg_file[instruction_word[1][`REG_K]][5:0];
723             end
724           end
725           else if (instruction_word[1][`REG_K] == `REG_SP && `STACK_INSTRUCTION(2) &&
          ↪  !stall[2]) begin
726             // data forward from the increment/decrement of the stack pointer
727             shifter_modifier <= alu_result[5:0];
728           end
729           else if ((instruction_word[1][`REG_K] == instruction_word[3][`REG_I]) &&
          ↪  `WB_INSTRUCTION(3) && !stall[3]) begin
730             // data forward from mc3
731             shifter_modifier <= temp_wb[5:0];
732             // TODO: data forward from other wb sources in mc3
733           end
734           else if (instruction_word[1][`REG_K] == `REG_SP && `STACK_INSTRUCTION(3) &&
          ↪  !stall[3]) begin
735             // data forward from the increment/decrement of the stack pointer
736             shifter_modifier <= temp_sp[5:0];
737           end
738           else begin
739             // no data forwarding
740             shifter_modifier <= reg_file[instruction_word[1][`REG_K]][5:0];
741           end
742
743           // set the carry in if rotating through carry
744           if ((instruction_word[1][`RS_OPCODE] == `RRC_SHIFT) ||
          ↪  (instruction_word[1][`RS_OPCODE] == `RLC_SHIFT)) begin
745             if ((instruction_word[2][`REG_I] == `REG_SR) && `WB_INSTRUCTION(2) &&
            ↪  !stall[2]) begin // if mc2 is writing to the REG_SR
746               // data forward from mc2
747               if (`ALU_INSTRUCTION(2)) begin
748                 shifter_carry_in <= alu_result[`SR_C];
749               end
750               else if (opcode[2] == `POP_IC) begin
751                 shifter_carry_in <= data_stack_out[`SR_C];
752               end
753               else if (opcode[2] == `LD_IC) begin
754                 `LOAD_MMIO(shifter_carry_in,`SR_C,)
755               end
```

```
756               else if (opcode[2] == `RS_IC) begin
757                 shifter_carry_in <= shifter_result[`SR_C];
758               end
759               // TODO: data forward from other wb sources in mc2
760               else begin
761                 // no data forwarding
762                 shifter_carry_in <= reg_file[`REG_SR][`SR_C];
763               end
764             end
765             else if ((instruction_word[3][`REG_I] == `REG_SR) && `WB_INSTRUCTION(3) &&
                ↪  !stall[3]) begin // if mc3 is writing to the REG_SR
766               // data forward from mc3
767               shifter_carry_in <= temp_wb[`SR_C];
768               // TODO: data forward from other wb sources in mc3
769             end
770             else if (`ALU_INSTRUCTION(2) && !stall[2]) begin // if the mc2 ALU
                ↪  instruction will change the REG_SR
771               // data forward from the alu output
772               shifter_carry_in <= alu_c;
773             end
774             else if (opcode[2] == `RS_IC && !stall[2]) begin // if the mc2 shift
                ↪  instruction will change the REG_SR
775               shifter_carry_in <= shifter_carry_out;
776             end
777             else if (`ALU_INSTRUCTION(3) || opcode[3] == `RS_IC && !stall[3]) begin //
                ↪  if the mc3 instruction will change the REG_SR
778               // data forward from the temp status register
779               shifter_carry_in <= temp_sr[`SR_C];
780             end
781             else begin
782               // no data forwarding
783               shifter_carry_in <= reg_file[`REG_SR][`SR_C];
784             end
785           end
786           else begin
787             shifter_carry_in <= reg_file[`REG_SR][`SR_C];
788           end
789
790         end // `RS_IC
791
792         `PUSH_IC: begin
793           // data forwarding for the data input
794           if (instruction_word[1][`DT_CONTROL] == 1'b1) begin
795             // push from constant
796             data_stack_data <= {{16{instruction_word[1][`DT_CONSTANT_MSB]}},
                ↪  instruction_word[1][`DT_CONSTANT]};
797           end
798           else if ((instruction_word[1][`REG_J] == instruction_word[2][`REG_I]) &&
                ↪  `WB_INSTRUCTION(2) && !stall[2]) begin
```

```
799             // data forward from mc2
800             if (`ALU_INSTRUCTION(2)) begin
801               data_stack_data <= alu_result;
802             end
803             else if (opcode[2] == `POP_IC) begin
804               data_stack_data <= data_stack_out;
805             end
806             else if (opcode[2] == `LD_IC) begin
807               `LOAD_MMIO(data_stack_data,31:0,)
808             end
809             else if (opcode[2] == `RS_IC) begin
810               data_stack_data <= shifter_result;
811             end
812             // TODO: data forward from other wb sources in mc2
813             else begin
814               // no data forwarding
815               data_stack_data <= reg_file[instruction_word[1][`REG_J]];
816             end
817           end
818           else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(2) &&
              ↪  !stall[2]) begin
819             // data forward from the increment/decrement of the stack pointer
820             data_stack_data <= alu_result;
821           end
822           else if ((instruction_word[1][`REG_J] == instruction_word[3][`REG_I]) &&
              ↪  `WB_INSTRUCTION(3) && !stall[3]) begin
823             // data forward from mc3
824             data_stack_data <= temp_wb;
825             // TODO: data forward from other wb sources in mc3
826           end
827           else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(3) &&
              ↪  !stall[3]) begin
828             // data forward from the increment/decrement of the stack pointer
829             data_stack_data <= temp_sp;
830           end
831           else begin
832             // no data forwarding
833             data_stack_data <= reg_file[instruction_word[1][`REG_J]];
834           end
835
836           // data foward stack pointer
837           // set alu_a to increment stack pointer
838           if ((`REG_SP == instruction_word[2][`REG_I]) && `WB_INSTRUCTION(2) &&
              ↪  !stall[2]) begin
839             // data forward from mc2
840             if (`ALU_INSTRUCTION(2)) begin
841               // data forward from alu output
842               alu_a <= alu_result;
843               data_stack_addr <= alu_result[5:0];
```

```
844             end
845             else if (opcode[2] == `POP_IC) begin
846               alu_a <= data_stack_out;
847               data_stack_addr <= data_stack_out[5:0];
848             end
849             else if (opcode[2] == `LD_IC) begin
850               `LOAD_MMIO(alu_a,31:0,)
851               `LOAD_MMIO(data_stack_addr,5:0,)
852             end
853             else if (opcode[2] == `RS_IC) begin
854               alu_a <= shifter_result;
855               data_stack_addr <= shifter_result[5:0];
856             end
857             // TODO: data forward from other wb sources in mc2
858             else begin
859               // no data forwarding
860               alu_a <= reg_file[`REG_SP];
861               data_stack_addr <= reg_file[`REG_SP][5:0];
862             end
863           end
864           else if ((opcode[2] == `PUSH_IC) || (opcode[2] == `POP_IC) && !stall[2])
          ↪  begin
865             // data forward from the output of the increment
866             alu_a <= alu_result;
867             data_stack_addr <= alu_result[5:0];
868           end
869           else if ((`REG_SP == instruction_word[3][`REG_I]) && `WB_INSTRUCTION(3) &&
          ↪  !stall[3]) begin
870             // data forward from mc3
871             alu_a <= temp_wb;
872             data_stack_addr <= temp_wb[5:0];
873             // TODO: data forward from other wb sources in mc3
874           end
875           else if ((opcode[3] == `PUSH_IC) || (opcode[3] == `POP_IC) && !stall[3])
          ↪  begin
876             // data forward from the output of the increment
877             alu_a <= temp_sp;
878             data_stack_addr <= temp_wb[5:0];
879           end
880           else begin
881             // no data forwarding
882             alu_a <= reg_file[`REG_SP];
883             data_stack_addr <= reg_file[`REG_SP][5:0];
884           end
885
886           alu_b <= 32'h00000001;
887
888           data_stack_push <= 1'b1;
889         end
```

```
890
891         `POP_IC: begin
892           // data foward stack pointer
893           // set alu_a to decrement stack pointer
894           if ((`REG_SP == instruction_word[2][`REG_I]) && `WB_INSTRUCTION(2) &&
              ↪  !stall[2]) begin
895             // data forward from mc2
896             if (`ALU_INSTRUCTION(2)) begin
897               // data forward from alu output
898               alu_a <= alu_result;
899               data_stack_addr <= alu_result[5:0] - 1'b1;
900             end
901             else if (opcode[2] == `POP_IC) begin
902               alu_a <= data_stack_out;
903               data_stack_addr <= data_stack_out[5:0] - 1'b1;
904             end
905             else if (opcode[2] == `LD_IC) begin
906               `LOAD_MMIO(alu_a,31:0,)
907               // data_stack_addr <= dm_out[5:0] - 1'b1;
908               `LOAD_MMIO(/*dest=*/data_stack_addr,/*bits=*/5:0,/*expr=*/-1'b1)
909             end
910             else if (opcode[2] == `RS_IC) begin
911               alu_a <= shifter_result;
912               data_stack_addr <= shifter_result[5:0] - 1'b1;
913             end
914             // TODO: data forward from other wb sources in mc2
915             else begin
916               // no data forwarding
917               alu_a <= reg_file[`REG_SP];
918               data_stack_addr <= reg_file[`REG_SP][5:0] - 1'b1;
919             end
920           end
921           else if ((opcode[2] == `PUSH_IC) || (opcode[2] == `POP_IC) && !stall[2])
              ↪  begin
922             // data forward from the output of the increment
923             alu_a <= alu_result;
924             data_stack_addr <= alu_result[5:0] - 1'b1;
925           end
926           else if ((`REG_SP == instruction_word[3][`REG_I]) && `WB_INSTRUCTION(3) &&
              ↪  !stall[3]) begin
927             // data forward from mc3
928             alu_a <= temp_wb;
929             data_stack_addr <= temp_wb[5:0] - 1'b1;
930             // TODO: data forward from other wb sources in mc3
931           end
932           else if ((opcode[3] == `PUSH_IC) || (opcode[3] == `POP_IC) && !stall[3])
              ↪  begin
933             // data forward from the output of the decrement
934             alu_a <= temp_sp;
```

```
935             data_stack_addr <= temp_sp[5:0] - 1'b1;
936           end
937           else begin
938             // no data forwarding
939             alu_a <= reg_file[`REG_SP];
940             data_stack_addr <= reg_file[`REG_SP][5:0] - 1'b1;
941           end
942
943           alu_b <= 32'h00000001;
944         end
945
946         `LD_IC, `ST_IC: begin
947           // Set the data memory address
948           if (instruction_word[1][`REG_J] != 5'b0 && instruction_word[1][`DT_CONTROL]
              ↪  == 1'b0) begin
949             // Indexed
950             if ((instruction_word[1][`REG_J] == instruction_word[2][`REG_I]) &&
                ↪  `WB_INSTRUCTION(2) && !stall[2]) begin
951               // data forward from mc2
952               if (`ALU_INSTRUCTION(2)) begin
953                 dm_address <= alu_result + instruction_word[1][`DT_CONSTANT];
954               end
955               else if (opcode[2] == `POP_IC) begin
956                 dm_address <= data_stack_out + instruction_word[1][`DT_CONSTANT];
957               end
958               else if (opcode[2] == `LD_IC) begin
959
                    ↪  `LOAD_MMIO(/*dest=*/dm_address,/*bits=*/31:0,/*expr=*/+instruction_word[1][`DT_CO
960               end
961               else if (opcode[2] == `RS_IC) begin
962                 dm_address <= shifter_result + instruction_word[1][`DT_CONSTANT];
963               end
964               // TODO: data forward from other wb sources in mc2
965               else begin
966                  // No data forwarding
967                 dm_address <= reg_file[instruction_word[1][`REG_J]] +
                    ↪  instruction_word[1][`DT_CONSTANT];
968               end
969             end
970             else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(2) &&
                ↪  !stall[2]) begin
971               // data forward from tne increment/decrement of the stack pointer
972               dm_address <= alu_result + instruction_word[1][`DT_CONSTANT];
973             end
974             else if ((instruction_word[1][`REG_J] == instruction_word[3][`REG_I]) &&
                ↪  `WB_INSTRUCTION(3) && !stall[3]) begin
975               // data forward from mc3
976               dm_address <= temp_wb + instruction_word[1][`DT_CONSTANT];
977               // TODO: data forward from other wb sources in mc3
```

```
978               end
979               else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(3) &&
                  ↪  !stall[3]) begin
980                 // data forward from the increment/decrement of the stack pointer
981                 dm_address <= temp_sp + instruction_word[1][`DT_CONSTANT];
982               end
983               else begin
984                 // No data forwarding
985                 dm_address <= reg_file[instruction_word[1][`REG_J]] +
                    ↪  instruction_word[1][`DT_CONSTANT];
986               end
987             end
988             else if (instruction_word[1][`REG_J] != 5'b0 &&
                ↪  instruction_word[1][`DT_CONTROL] == 1'b1) begin
989               // Register Direct
990               if ((instruction_word[1][`REG_J] == instruction_word[2][`REG_I]) &&
                  ↪  `WB_INSTRUCTION(2) && !stall[2]) begin
991                 // data forward from mc2
992                 if (`ALU_INSTRUCTION(2)) begin
993                   dm_address <= alu_result;
994                 end
995                 else if (opcode[2] == `POP_IC) begin
996                   dm_address <= data_stack_out;
997                 end
998                 else if (opcode[2] == `LD_IC) begin
999                   `LOAD_MMIO(dm_address,31:0,)
1000                end
1001                else if (opcode[2] == `RS_IC) begin
1002                  dm_address <= shifter_result;
1003                end
1004                // TODO: data forward from other wb sources in mc2
1005                else begin
1006                  // No data forwarding
1007                  dm_address <= reg_file[instruction_word[1][`REG_J]];
1008                end
1009              end
1010              else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(2) &&
                  ↪  !stall[2]) begin
1011                // data forward from tne increment/decrement of the stack pointer
1012                dm_address <= alu_result;
1013              end
1014              else if ((instruction_word[1][`REG_J] == instruction_word[3][`REG_I]) &&
                  ↪  `WB_INSTRUCTION(3) && !stall[3]) begin
1015                // data forward from mc3
1016                dm_address <= temp_wb;
1017                // TODO: data forward from other wb sources in mc3
1018              end
1019              else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(3) &&
                  ↪  !stall[3]) begin
```

```
1020              // data forward from the increment/decrement of the stack pointer
1021              dm_address <= temp_sp;
1022            end
1023            else begin
1024              // No data forwarding
1025              dm_address <= reg_file[instruction_word[1][`REG_J]];
1026            end
1027          end
1028          else if (instruction_word[1][`REG_J] == 5'b0 &&
          ↪  instruction_word[1][`DT_CONTROL] == 1'b0) begin
1029            // PC Relative
1030            dm_address <= instruction_addr[1] + instruction_word[1][`DT_CONSTANT];
1031          end
1032          else begin
1033            // Absolute
1034            dm_address <= instruction_word[1][`DT_CONSTANT];
1035          end
1036
1037
1038          // Set the data input
1039          if (opcode[1] == `ST_IC) begin
1040
1041            // set the data value
1042            if ((instruction_word[1][`REG_I] == instruction_word[2][`REG_I]) &&
            ↪  `WB_INSTRUCTION(2) && !stall[2]) begin
1043              // data forward from mc2
1044              if (`ALU_INSTRUCTION(2)) begin
1045                dm_data <= alu_result;
1046              end
1047              else if (opcode[2] == `POP_IC) begin
1048                dm_data <= data_stack_out;
1049              end
1050              else if (opcode[2] == `LD_IC) begin
1051                `LOAD_MMIO(dm_data,31:0,)
1052              end
1053              else if (opcode[2] == `RS_IC) begin
1054                dm_data <= shifter_result;
1055              end
1056              // TODO: data forward from other wb sources in mc2
1057              else begin
1058                // No data forwarding
1059                dm_data <= reg_file[instruction_word[1][`REG_I]];
1060              end
1061            end
1062            else if (instruction_word[1][`REG_I] == `REG_SP && `STACK_INSTRUCTION(2) &&
            ↪  !stall[2]) begin
1063              // data forward from tne increment/decrement of the stack pointer
1064              dm_data <= alu_result;
1065            end
```

```
1066              else if ((instruction_word[1][`REG_I] == instruction_word[3][`REG_I]) &&
                  ↪  `WB_INSTRUCTION(3) && !stall[3]) begin
1067                // data forward from mc3
1068                dm_data <= temp_wb;
1069                // TODO: data forward from other wb sources in mc3
1070              end
1071              else if (instruction_word[1][`REG_I] == `REG_SP && `STACK_INSTRUCTION(3) &&
                  ↪  !stall[3]) begin
1072                // data forward from the increment/decrement of the stack pointer
1073                dm_data <= temp_sp;
1074              end
1075              else begin
1076                // No data forwarding
1077                dm_data <= reg_file[instruction_word[1][`REG_I]];
1078              end
1079            end
1080
1081          end
1082
1083          `JMP_IC: begin
1084            // Set the temp program counter
1085            if (instruction_word[1][`REG_I] != 5'b0 && instruction_word[1][`JMP_CONTROL]
                  ↪  == 1'b0) begin
1086              // Indexed
1087              if ((instruction_word[1][`REG_I] == instruction_word[2][`REG_I]) &&
                  ↪  `WB_INSTRUCTION(2) && !stall[2]) begin
1088                // data forward from mc2
1089                if (`ALU_INSTRUCTION(2)) begin
1090                  temp_address <= alu_result + instruction_word[1][`JMP_ADDR];
1091                end
1092                else if (opcode[2] == `POP_IC) begin
1093                  temp_address <= data_stack_out + instruction_word[1][`JMP_ADDR];
1094                end
1095                else if (opcode[2] == `LD_IC) begin
1096
                    ↪  `LOAD_MMIO(/*dest=*/temp_address,/*bits=*/31:0,/*expr=*/+instruction_word[1][`JMP_
1097                end
1098                else if (opcode[2] == `RS_IC) begin
1099                  temp_address <= shifter_result + instruction_word[1][`JMP_ADDR];
1100                end
1101                // TODO: data forward from other wb sources in mc2
1102                else begin
1103                  // No data forwarding
1104                  temp_address <= reg_file[instruction_word[1][`REG_I]] +
                    ↪  instruction_word[1][`JMP_ADDR];
1105                end
1106              end
1107              else if (instruction_word[1][`REG_I] == `REG_SP && `STACK_INSTRUCTION(2) &&
                  ↪  !stall[2]) begin
```

```
1108              // data forward from tne increment/decrement of the stack pointer
1109              temp_address <= alu_result + instruction_word[1][`JMP_ADDR];
1110            end
1111            else if ((instruction_word[1][`REG_I] == instruction_word[3][`REG_I]) &&
                ↪  `WB_INSTRUCTION(3) && !stall[3]) begin
1112              // data forward from mc3
1113              temp_address <= temp_wb + instruction_word[1][`JMP_ADDR];
1114              // TODO: data forward from other wb sources in mc3
1115            end
1116            else if (instruction_word[1][`REG_I] == `REG_SP && `STACK_INSTRUCTION(3) &&
                ↪  !stall[3]) begin
1117              // data forward from the increment/decrement of the stack pointer
1118              temp_address <= temp_sp + instruction_word[1][`JMP_ADDR];
1119            end
1120            else begin
1121              // No data forwarding
1122              temp_address <= reg_file[instruction_word[1][`REG_I]] +
                  ↪  instruction_word[1][`JMP_ADDR];
1123            end
1124          end
1125          else if (instruction_word[1][`REG_I] != 5'b0 &&
              ↪  instruction_word[1][`JMP_CONTROL] == 1'b1) begin
1126            // Register Direct
1127            if ((instruction_word[1][`REG_I] == instruction_word[2][`REG_I]) &&
                ↪  `WB_INSTRUCTION(2) && !stall[2]) begin
1128              // data forward from mc2
1129              if (`ALU_INSTRUCTION(2)) begin
1130                temp_address <= alu_result;
1131              end
1132              else if (opcode[2] == `POP_IC) begin
1133                temp_address <= data_stack_out;
1134              end
1135              else if (opcode[2] == `LD_IC) begin
1136                `LOAD_MMIO(temp_address,31:0,)
1137              end
1138              else if (opcode[2] == `RS_IC) begin
1139                temp_address <= shifter_result;
1140              end
1141              // TODO: data forward from other wb sources in mc2
1142              else begin
1143                // No data forwarding
1144                temp_address <= reg_file[instruction_word[1][`REG_I]];
1145              end
1146            end
1147            else if (instruction_word[1][`REG_I] == `REG_SP && `STACK_INSTRUCTION(2) &&
                ↪  !stall[2]) begin
1148              // data forward from tne increment/decrement of the stack pointer
1149              temp_address <= alu_result;
1150            end
```

```
1151              else if ((instruction_word[1][`REG_I] == instruction_word[3][`REG_I]) &&
                  ↪  `WB_INSTRUCTION(3) && !stall[3]) begin
1152                // data forward from mc3
1153                temp_address <= temp_wb;
1154                // TODO: data forward from other wb sources in mc3
1155              end
1156              else if (instruction_word[1][`REG_I] == `REG_SP && `STACK_INSTRUCTION(3) &&
                  ↪  !stall[3]) begin
1157                // data forward from the increment/decrement of the stack pointer
1158                temp_address <= temp_sp;
1159              end
1160              else begin
1161                // No data forwarding
1162                temp_address <= reg_file[instruction_word[1][`REG_I]];
1163              end
1164            end
1165            else if (instruction_word[1][`REG_I] == 5'b0 &&
                  ↪  instruction_word[1][`JMP_CONTROL] == 1'b0) begin
1166              // PC Relative
1167              temp_address <= instruction_addr[1] + instruction_word[1][`JMP_ADDR];
1168            end
1169            else begin
1170              // Absolute
1171              temp_address <= instruction_word[1][`JMP_ADDR];
1172            end
1173          end // JMP_IC
1174
1175          `CALL_IC: begin
1176            // Set address
1177            // Always absolute mode for call (for now)
1178            temp_address <= instruction_word[1][`JMP_ADDR];
1179
1180            // push the program counter onto the stack for when we return
1181            call_stack_push <= 1'b1;
1182            call_stack_data <= reg_file[`REG_PC];
1183          end
1184
1185          `RET_IC: begin
1186            // pop the status register
1187            call_stack_pop <= 1'b1;
1188            reg_file[`REG_SR] <= call_stack_out;
1189          end
1190
1191          default: begin
1192          end
1193        endcase // opcode[1]
1194
1195        // set the alu opcode
1196        case (opcode[1])
```

```
1197          `ADD_IC, `PUSH_IC: begin
1198            alu_opcode <= `ADD_ALU;
1199          end
1200
1201          `SUB_IC, `CMP_IC, `POP_IC: begin
1202            alu_opcode <= `SUB_ALU;
1203          end
1204
1205          `NOT_IC: begin
1206            alu_opcode <= `NOT_ALU;
1207          end
1208
1209          `AND_IC: begin
1210            alu_opcode <= `AND_ALU;
1211          end
1212
1213          `BIC_IC: begin
1214            alu_opcode <= `BIC_ALU;
1215          end
1216
1217          `OR_IC: begin
1218            alu_opcode <= `OR_ALU;
1219          end
1220
1221          `XOR_IC: begin
1222            alu_opcode <= `XOR_ALU;
1223          end
1224
1225          `CPY_IC: begin
1226            alu_opcode <= `NOP_ALU;
1227          end
1228
1229          `MUL_IC: begin
1230            alu_opcode <= `MUL_ALU;
1231          end
1232
1233          `DIV_IC: begin
1234            alu_opcode <= `DIV_ALU;
1235          end
1236
1237          default: begin
1238            alu_opcode <= alu_opcode;
1239          end
1240
1241        endcase // opcode[1]
1242
1243        instruction_word[2] <= instruction_word[1];
1244        instruction_addr[2] <= instruction_addr[1];
1245      end // if (stall[1] == 1'b0)
```

```
1246
1247      // Machine cycle 0
1248      // instruction fetch
1249      if (stall[0] == 1'b0) begin
1250        reg_file[`REG_PC] <= reg_file[`REG_PC] + 3'h4;
1251        instruction_addr[1] <= reg_file[`REG_PC][13:0];
1252        instruction_word[1] <= pm_out;
1253
1254        // set stall cycles
1255        if ((opcode[0] == `JMP_IC) || (opcode[0] == `CALL_IC) || (opcode[0] == `RET_IC))
          ↪  begin
1256          stall_cycles <= 3'h3;
1257          stall[0] <= 1'b1;
1258        end
1259      end // if (stall[0] == 1'b0)
1260
1261    end // else begin
1262  end // always @(posedge clk)
1263
1264  task reset_all; begin
1265    gpio_out <= 32'b0;
1266    dm_data <= 32'b0;
1267    dm_wren <= 1'b0;
1268    dm_address <= 14'b0;
1269
1270    temp_address <= 16'b0;
1271
1272    instruction_word[3] <= 32'b0;
1273    instruction_word[2] <= 32'b0;
1274    instruction_word[1] <= 32'b0;
1275
1276    instruction_addr[3] <= 14'b0;
1277    instruction_addr[2] <= 14'b0;
1278    instruction_addr[1] <= 14'b0;
1279
1280    stall_cycles <= 4'b0;
1281    stall[3] <= 1'b1;
1282    stall[2] <= 1'b1;
1283    stall[1] <= 1'b1;
1284    stall[0] <= 1'b1;
1285
1286    data_stack_data <= 32'b0;
1287    data_stack_addr <= 6'b0;
1288    data_stack_push <= 1'b0;
1289    data_stack_pop <= 1'b0;
1290
1291    call_stack_data <= 32'b0;
1292    call_stack_push <= 1'b0;
1293    call_stack_pop <= 1'b0;
```

```
1294
1295    alu_a <= 32'b0;
1296    alu_b <= 32'b0;
1297    temp_sr <= 32'b0;
1298    temp_sp <= 32'b0;
1299    alu_opcode <= 4'b0;
1300
1301    shifter_operand <= 32'b0;
1302    shifter_carry_in <= 1'b0;
1303    shifter_modifier <= 6'b0;
1304    shifter_opcode <= 3'b0;
1305
1306    temp_wb <= 32'b0;
1307
1308    for (i=0; i<32; i=i+1) begin
1309      reg_file[i] <= 32'h0;
1310    end
1311
1312  end
1313  endtask // reset_all
1314
1315  endmodule // cjg_risc
```

II.1.4 Clock Generator

cjg_clkgen.v

```
1  module cjg_clkgen (
2      // system inputs
3      input reset,                    // system reset
4      input clk,                      // system clock
5
6      // system outputs
7      output clk_p1,                  // phase 0
8      output clk_p2,                  // phase 1
9
10     // dft
11     input scan_in0,
12     input scan_en,
13     input test_mode,
14     output scan_out0
15 );
16
17 // Clock counter
18 reg[1:0] clk_cnt;
19
20 // Signals for generating the clocks
```

```
wire pre_p1 = (~clk_cnt[1] & ~clk_cnt[0]);
wire pre_p2 = (clk_cnt[1] & ~clk_cnt[0]);

// Buffer output of phase 0 clock
CLKBUFX4 clk_p1_buf (
    .A(pre_p1),
    .Y(clk_p1)
);

// Buffer output of phase 1 clock
CLKBUFX4 clk_p2_buf (
    .A(pre_p2),
    .Y(clk_p2)
);

// Clock counter
always @ (posedge clk, negedge reset) begin
    if(~reset) begin
        clk_cnt <= 2'h0;
    end
    else begin
        clk_cnt <= clk_cnt + 1'b1;
    end
end

endmodule // cjg_clkgen
```

II.1.5 ALU

cjg_alu.v

```
// Dynamic width combinational logic ALU

`include "src/cjg_opcodes.vh"

module cjg_alu #(parameter WIDTH = 32) (
    // sys ports
    input reset,
    input clk,

    input [WIDTH-1:0] a,
    input [WIDTH-1:0] b,
    input [3:0] opcode,

    output [WIDTH-1:0] result,
    output c, n, v, z,

```

```
17      // dft
18      input scan_in0,
19      input scan_en,
20      input test_mode,
21      output scan_out0
22  );
23
24  reg[WIDTH:0] internal_result;
25  wire overflow, underflow;
26
27  assign result = internal_result[WIDTH-1:0];
28  assign c = internal_result[WIDTH];
29  assign n = internal_result[WIDTH-1];
30  assign z = (internal_result == 0 ? 1'b1 : 1'b0);
31
32  assign overflow = (internal_result[WIDTH:WIDTH-1] == 2'b01 ? 1'b1 : 1'b0);
33  assign underflow = (internal_result[WIDTH:WIDTH-1] == 2'b10 ? 1'b1 : 1'b0);
34
35  assign v = overflow | underflow;
36
37  always @(*) begin
38      internal_result = 0;
39
40      case (opcode)
41
42          `ADD_ALU: begin
43              // signed addition
44              internal_result = {a[WIDTH-1], a} + {b[WIDTH-1], b};
45          end
46
47          `SUB_ALU: begin
48              // signed subtraction
49              internal_result = ({a[WIDTH-1], a} + ~{b[WIDTH-1], b}) + 1'b1;
50          end
51
52          `AND_ALU: begin
53              // logical AND
54              internal_result = a & b;
55          end
56
57          `BIC_ALU: begin
58              // logical bit clear
59              internal_result = a & (~b);
60          end
61
62          `OR_ALU : begin
63              // logical OR
64              internal_result = a | b;
65          end
```

```
66
67          `NOT_ALU: begin
68              // logical invert
69              internal_result = ~a;
70          end
71
72          `XOR_ALU: begin
73              // logical XOR
74              internal_result = a ^ b;
75          end
76
77          `NOP_ALU: begin
78              // no operation
79              // sign extend a to prevent wrongful overflow flag by accident
80              internal_result = {a[WIDTH-1], a};
81          end
82
83          `MUL_ALU: begin
84              // signed multiplication
85              internal_result = a * b;
86          end
87
88          `DIV_ALU: begin
89              // unsigned division
90              internal_result = a / b;
91          end
92
93          default: begin
94              internal_result = internal_result;
95          end // default
96
97      endcase // opcode
98
99  end // always @(*)
100
101 endmodule // cjg_alu
```

II.1.6 Shifter

cjg_shifter.v

```
1  // Dynamic width combinational logic Shifter
2
3  `include "../cjg_risc/src/cjg_opcodes.vh"
4
5  // Whether or not to use the modifier shift logic
6  `define USE_MODIFIER
```

```
7
8  module cjg_shifter #(parameter WIDTH = 32, MOD_WIDTH = 6) (
9      input reset,
10     input clk,
11
12     input signed [WIDTH-1:0] operand,
13     input carry_in,
14     input [2:0] opcode,
15 `ifdef USE_MODIFIER
16     input [MOD_WIDTH-1:0] modifier,
17 `endif
18
19     output reg [WIDTH-1:0] result,
20     output reg carry_out,
21
22     // dft
23     input scan_in0,
24     input scan_en,
25     input test_mode,
26     output scan_out0
27 );
28
29 `ifdef USE_MODIFIER
30 wire[WIDTH+WIDTH-1:0] temp_rotate_right = {operand, operand} >>
   ↪  modifier[MOD_WIDTH-2:0];
31 wire[WIDTH+WIDTH-1:0] temp_rotate_left = {operand, operand} << modifier[MOD_WIDTH-2:0];
32
33 wire[WIDTH+WIDTH+1:0] temp_rotate_right_c = {carry_in, operand, carry_in, operand} >>
   ↪  modifier;
34 wire[WIDTH+WIDTH+1:0] temp_rotate_left_c = {carry_in, operand, carry_in, operand} <<
   ↪  modifier;
35 `endif
36
37 always @(*) begin
38
39     case (opcode)
40
41         `SRL_SHIFT: begin
42 `ifndef USE_MODIFIER
43             // shift right logical by 1
44             result <= {1'b0, operand[WIDTH-1:1]};
45 `else
46             // shift right by modifier
47             result <= operand >> modifier[MOD_WIDTH-2:0];
48 `endif
49             carry_out <= carry_in;
50         end
51
52         `SLL_SHIFT: begin
```

```
53  `ifndef USE_MODIFIER
54              // shift left logical by 1
55              result <= {operand[WIDTH-2:0], 1'b0};
56  `else
57              // shift left by modifier
58              result <= operand << modifier[MOD_WIDTH-2:0];
59  `endif
60              carry_out <= carry_in;
61          end
62
63          `SRA_SHIFT: begin
64  `ifndef USE_MODIFIER
65              // shift right arithmetic by 1
66              result <= {operand[WIDTH-1], operand[WIDTH-1:1]};
67  `else
68              // shift right arithmetic by modifier
69              result <= operand >>> modifier[MOD_WIDTH-2:0];
70  `endif
71              carry_out <= carry_in;
72          end
73
74          `RTR_SHIFT: begin
75  `ifndef USE_MODIFIER
76              // rotate right by 1
77              result <= {operand[0], operand[WIDTH-1:1]};
78  `else
79              // rotate right by modifier
80              result <= temp_rotate_right[WIDTH-1:0];
81  `endif
82              carry_out <= carry_in;
83          end
84
85          `RTL_SHIFT: begin
86  `ifndef USE_MODIFIER
87              // rotate left
88              result <= {operand[WIDTH-2:0], operand[WIDTH-1]};
89  `else
90              // rotate left by modifier
91              result <= temp_rotate_left[WIDTH+WIDTH-1:WIDTH];
92  `endif
93              carry_out <= carry_in;
94          end
95
96          `RRC_SHIFT: begin
97  `ifndef USE_MODIFIER
98              // rotate right through carry
99              result <= {carry_in, operand[WIDTH-1:1]};
100             carry_out <= operand[0];
101 `else
```

```
102
103             // rotate right through carry by modifier
104             result <= temp_rotate_right_c[WIDTH-1:0];
105             carry_out <= temp_rotate_right_c[WIDTH];
106 `endif
107         end
108
109         `RLC_SHIFT: begin
110 `ifndef USE_MODIFIER
111             // rotate left through carry
112             result <= {operand[WIDTH-2:0], carry_in};
113             carry_out <= operand[WIDTH-1];
114 `else
115             // rotate left through carry by modifier
116             result <= temp_rotate_left_c[WIDTH+WIDTH:WIDTH+1];
117             carry_out <= temp_rotate_left_c[WIDTH];
118 `endif
119         end
120
121         default: begin
122             result <= operand;
123             carry_out <= carry_in;
124         end // default
125
126     endcase // opcode
127
128 end // always @(*)
129
130 endmodule // cjg_alu
```

II.1.7 Data Stack

cjg_mem_stack.v

```
1  module cjg_mem_stack #(parameter WIDTH = 32, DEPTH = 32, ADDRW = 5) (
2
3      input clk,
4      input reset,
5      input [WIDTH-1:0] d,
6      input [ADDRW-1:0] addr,
7      input push,
8      input pop,
9
10     output reg [WIDTH-1:0] q,
11
12     // dft
13     input scan_in0,
```

```
    input scan_en,
    input test_mode,
    output scan_out0
);

reg [WIDTH-1:0] stack [DEPTH-1:0];
integer i;


always @(posedge clk or negedge reset) begin
    if (~reset) begin
        q <= {WIDTH{1'b0}};
        for (i=0; i < DEPTH; i=i+1) begin
            stack[i] <= {WIDTH{1'b0}};
        end
    end
    else begin
        if (push) begin
            stack[addr] <= d;
        end
        else begin
            stack[addr] <= stack[addr];
        end

        q <= stack[addr];
    end
end

endmodule // cjg_mem_stack
```

II.1.8 Call Stack

cjg_stack.v

```
module cjg_stack #(parameter WIDTH = 32, DEPTH = 16) (

    input clk,
    input reset,
    input [WIDTH-1:0] d,
    input push,
    input pop,

    output [WIDTH-1:0] q,

    // dft
    input scan_in0,
    input scan_en,
```

```
    input test_mode,
    output scan_out0
);

reg [WIDTH-1:0] stack [DEPTH-1:0];
integer i;
assign q = stack[0];

always @(posedge clk or negedge reset) begin
    if (~reset) begin
        for (i=0; i < DEPTH; i=i+1) begin
            stack[i] <= {WIDTH{1'b0}};
        end
    end
    else begin
        if (push) begin
            stack[0] <= d;
            for (i=1; i < DEPTH; i=i+1) begin
                stack[i] <= stack[i-1];
            end
        end
        else if (pop) begin
            for (i=0; i < DEPTH-1; i=i+1) begin
                stack[i] <= stack[i+1];
            end
            stack[DEPTH-1] <= 0;
        end
        else begin
            for (i=0; i < DEPTH; i=i+1) begin
                stack[i] <= stack[i];
            end
        end
    end
end

endmodule // cjg_stack
```

II.1.9 Testbench

cjg_risc_test.v

```
`include "src/cjg_opcodes.vh"

// must be in mif directory
`define MIF "myDouble"

//`define TEST_ALU
```

```
7
8  module test;
9
10 // tb stuff
11 integer i;
12
13 // system ports
14 reg  clk, reset;
15 wire clk_p1, clk_p2;
16
17 // dft ports
18 wire  scan_out0;
19 reg  scan_in0, scan_en, test_mode;
20
21 always begin
22     #0.5 clk = ~clk; // 1000 MHz clk
23 end
24
25 // program memory
26 reg [7:0] pm [0:65535];     // program memory
27 reg [31:0] pm_out;         // program memory output data
28 wire [15:0] pm_address;     // program memory address
29
30 // data memory
31 reg [7:0] dm [0:65535];     // data memory
32 reg [31:0] dm_out;         // data memory output
33 wire [31:0] dm_data;        // data memory input data
34 wire dm_wren;               // data memory write enable
35 wire [15:0] dm_address;     // data memory address
36
37 always @(posedge clk_p2) begin
38     if (dm_wren == 1'b1) begin
39         dm[dm_address+3] = dm_data[31:24];
40         dm[dm_address+2] = dm_data[23:16];
41         dm[dm_address+1] = dm_data[15:8];
42         dm[dm_address]   = dm_data[7:0];
43     end
44     pm_out = {pm[pm_address+3], pm[pm_address+2], pm[pm_address+1], pm[pm_address]};
45     dm_out = {dm[dm_address+3], dm[dm_address+2], dm[dm_address+1], dm[dm_address]};
46 end
47
48 // inputs
49 reg [31:0] gpio_in;         // button inputs
50 reg [3:0] ext_interrupt_bus;  //external interrupts
51
52 // outputs
53 wire [31:0] gpio_out;
54
55 `ifdef TEST_ALU
```

```
56
57  reg [31:0] alu_a, alu_b;
58  reg [3:0] alu_opcode;
59  wire [31:0] alu_result;
60  wire alu_c, alu_n, alu_v, alu_z;
61
62  reg [31:0] tb_alu_result;
63
64  cjg_alu alu(
65      .a(alu_a),
66      .b(alu_b),
67      .opcode(alu_opcode),
68
69      .result(alu_result),
70      .c(alu_c),
71      .n(alu_n),
72      .v(alu_v),
73      .z(alu_z)
74  );
75  `endif
76
77  cjg_risc top(
78      // system inputs
79      .reset(reset),
80      .clk(clk),
81      .gpio_in(gpio_in),
82      .ext_interrupt_bus(ext_interrupt_bus),
83
84      // generated clock phases
85      .clk_p1(clk_p1),
86      .clk_p2(clk_p2),
87
88      // system outputs
89      .gpio_out(gpio_out),
90
91      // program memory
92      .pm_out(pm_out),
93      .pm_address(pm_address),
94
95      // data memory
96      .dm_data(dm_data),
97      .dm_out(dm_out),
98      .dm_wren(dm_wren),
99      .dm_address(dm_address),
100
101     // dft
102     .scan_in0(scan_in0),
103     .scan_en(scan_en),
104     .test_mode(test_mode),
```

```
105     .scan_out0(scan_out0)
106 );
107 
108 initial begin
109     $timeformat(-9,2,"ns", 16);
110 `ifdef SDFSCAN
111     $sdf_annotate("sdf/cjg_risc_tsmc065_scan.sdf", test.top);
112 `endif
113 
114 `ifdef TEST_ALU
115     // ALU TEST
116     alu_a = 32'hfffffffc;
117     alu_b = 32'hfffffffe;
118     alu_opcode = `ADD_ALU;
119 
120     #10 tb_alu_result = alu_result;
121 
122     $display("alu_result = %x", tb_alu_result);
123     $display("internal_result = %x", alu.internal_result);
124     $display("alu_c = %x", alu_c);
125     $display("alu_n = %x", alu_n);
126     $display("alu_v = %x", alu_v);
127     $display("alu_z = %x", alu_z);
128 
129     $finish;
130 `endif
131 
132 // RISC TEST
133 
134     // init memories
135     $readmemh({"mif/", `MIF, ".mif"}, pm);
136     $readmemh({"mif/", `MIF, "_dm", ".mif"}, dm);
137     $display("Loaded %s", {"mif/", `MIF, ".mif"});
138     // reset for some cycles
139     assert_reset;
140     repeat (3) begin
141         @(posedge clk);
142     end
143 
144     // come out of reset a little before the edge
145     #0.25 deassert_reset;
146     @(posedge clk_p1);
147 
148     gpio_in = 12;
149 
150     // run until program reaches end of memory
151     while (!(^pm_out === 1'bX) && (pm_out != 32'hFFFFFFFF) && (gpio_out !=
    ↪  32'hDEADBEEF)) begin
152        @(posedge clk_p1);
```

```
153     end
154
155     $display("Trying to read from unknown program memory");
156
157     // run for a few more clock cycles to empty the pipeline
158     repeat (6) begin
159         @(posedge clk);
160     end
161
162     $display("gpio_out = %x", gpio_out);
163 `ifndef SDFSCAN
164     print_reg_file;
165 `endif
166     //print_stack;
167     $display("DONE");
168     $stop;
169 end // initial
170
171 `ifndef SDFSCAN
172 task print_reg_file; begin
173     $display("Register Contents:");
174     for (i=0; i<32; i=i+1) begin
175         $display("R%0d = 0x%X", i, top.reg_file[i]);
176     end
177     $display({30{"-"}});
178 end
179 endtask // print_reg_file
180
181 task print_stack; begin
182     $display("Stack Contents:");
183     for (i=0; i<32; i=i+1) begin
184         $display("S%0d = 0x%X", i, top.data_stack.stack[i]);
185     end
186     $display({30{"-"}});
187 end
188 endtask // print_stack
189 `endif
190
191 task assert_reset; begin
192     // reset dft ports
193     scan_in0 = 1'b0;
194     scan_en = 1'b0;
195     test_mode = 1'b0;
196
197     // reset system inputs
198     clk = 1'b0;
199     reset = 1'b0;
200     gpio_in = 32'b0;
201     ext_interrupt_bus = 4'b0;
```

```
202 end
203 endtask // assert_reset
204
205 task deassert_reset; begin
206     reset = 1'b1;
207 end
208 endtask // deassert_reset
209
210 endmodule // test
```

II.2 ELF to Memory

elf2mem.py

```
1  #!/usr/bin/env python
2
3  import argparse
4  import elffile
5  import os
6  import sys
7
8  def getData(section, wordLength):
9      data = []
10     buf = section.content
11
12     tmp = 0
13     for i in range(0, len(buf)):
14         byte = ord(buf[i]) # transform the character to binary
15         tmp |= byte << (8 * (i%wordLength)) # shift it into place in the word
16
17         if i%wordLength == wordLength-1: # if this is the last byte in the word
18             data.append(tmp)
19             tmp = 0
20
21     return data
22
23 def main(args):
24     if not os.path.isfile(args.elf):
25         print "error: cannot find file: {}".format(args.elf)
26         return 1
27     else:
28         with open(args.elf, 'rb') as f:
29             ef = elffile.open(fileobj=f)
30             section = None
31
32             if args.section is None:
```

```
33                  # if no section was provided in the arguments list all available
34                  sections = [section.name for section in ef.sectionHeaders if
                    ↪  section.name]
35                  print "list of sections: {}".format(" ".join(sections))
36                  return 0
37              else:
38                  sections = [section for section in ef.sectionHeaders if section.name ==
                    ↪  args.section][:1]
39                  if len(sections) == 1:
40                      section = sections[0]
41                  else:
42                      section = None
43
44              if not section:
45                  print "error: could not find section with name:
                    ↪  {}".format(args.section)
46                  return 0
47              elif elffile.SHT.bycode[section.type] !=
                ↪  elffile.SHT.byname["SHT_PROGBITS"]:
48                  print "error: section has invalid type:
                    ↪  {}".format(elffile.SHT.bycode[section.type])
49                  return 0
50              elif len(section.content) % args.length != 0:
51                  print "error: {} data ({} bytes) does not align with a word length of
                    ↪  {} bytes".format(section.name, len(section.content), args.length)
52                  return 0
53
54              # get the binary data from the section and align it to words
55              data = getData(section, args.length)
56
57          # write the data by word to a readmem formatted file
58          out = ""
59          out += "// Converted from the {} section in {}\n".format(section.name,
            ↪  args.elf)
60          out += "// $ {}\n".format(" ".join(sys.argv))
61          out += "\n"
62
63          counter = 0
64          for word in data:
65              out += "@{:08X} {:0{pad}X}\n".format(counter, word, pad=args.length*2)
66              counter += args.addresses
67
68          if args.output:
69              # write the output to a file
70              with open(args.output, "wb") as outputFile:
71                  outputFile.write(out)
72          else:
73              # write the output to stdout
74              sys.stdout.write(out)
```

```
75
76
77  if __name__ == "__main__":
78      parser = argparse.ArgumentParser(description="Extract a section from an ELF to
        ↪  readmem format")
79      parser.add_argument("-s", "--section", required=False, metavar="section", type=str,
        ↪  help="The name of the ELF section file to output")
80      parser.add_argument("-o", "--output", required=False, metavar="output", type=str,
        ↪  help="The path to the output readmem file (default: stdout)")
81      parser.add_argument("-l", "--length", required=False, metavar="length", type=int,
        ↪  help="The length of a memory word in number of bytes (default: 1)", default=1)
82      parser.add_argument("-a", "--addresses", required=False, metavar="address",
        ↪  type=int, help="The number of addresses to increment per word", default=1)
83      parser.add_argument("elf", metavar="elf-file", type=str, help="The input ELF file")
84      args = parser.parse_args()
85
86      main(args)
```