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

Connor Jan Goldberg
cjg3259@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Master's Project is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@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

Abstract ii  
Declaration iii  
Acknowledgements iv  
Forward v  
Contents vi  
List of Figures ix  
List of Listings x  
List of Tables xi  

1 Introduction 1  
1.1 Organization ................................................. 2  

2 The Design of CPUs and Compilers 3  
2.1 CPU Design .................................................. 3  
2.2 Compiler Design ........................................... 5  
2.2.1 Application Binary Interface ............................ 5  
2.2.2 Compiler Models ......................................... 6  
2.2.3 GCC ...................................................... 7  
2.2.4 LLVM ...................................................... 8  
2.2.4.1 Front End ............................................... 8  
2.2.4.2 Optimization ......................................... 9  
2.2.4.3 Backend ............................................... 9  

3 Custom RISC CPU Design 11  
3.1 Instruction Set Architecture .............................. 11  
3.1.1 Register File ............................................ 12
3.1.2 Stack Design ................................................. 13
3.1.3 Memory Architecture ..................................... 14
3.2 Hardware Implementation .................................. 15
  3.2.1 Pipeline Design ........................................ 16
    3.2.1.1 Instruction Fetch ............................... 16
    3.2.1.2 Operand Fetch ................................. 17
    3.2.1.3 Execute ....................................... 17
    3.2.1.4 Write Back ................................ 18
  3.2.2 Stalling ............................................... 18
3.2.3 Clock Phases ............................................ 18
3.3 Instruction Details ....................................... 18
  3.3.1 Load and Store ....................................... 19
  3.3.2 Data Transfer ........................................ 20
  3.3.3 Flow Control ........................................ 21
  3.3.4 Manipulation Instructions ......................... 22
    3.3.4.1 Shift and Rotate .............................. 24
4 Custom LLVM Backend Design ............................... 26
  4.1 Structure and Tools .................................... 26
    4.1.1 Code Generator Design Overview ................. 27
    4.1.2 TableGen ........................................ 28
    4.1.3 Clang and llc .................................. 31
  4.2 Custom Target Implementation ....................... 31
    4.2.1 Abstract Target Description .................... 33
      4.2.1.1 Register Information ....................... 33
      4.2.1.2 Calling Conventions ....................... 34
      4.2.1.3 Special Operands ......................... 34
      4.2.1.4 Instruction Formats ....................... 35
      4.2.1.5 Complete Instruction Definitions .......... 36
      4.2.1.6 Additional Descriptions ................... 40
    4.2.2 Instruction Selection ............................ 40
      4.2.2.1 SelectionDAG Construction ................. 41
      4.2.2.2 Legalization ................................ 46
      4.2.2.3 Selection .................................. 51
      4.2.2.4 Scheduling ................................ 55
  4.2.3 Register Allocation ................................ 55
  4.2.4 Code Emission ...................................... 57
    4.2.4.1 Assembly Printer ............................. 57
    4.2.4.2 ELF Object Writer ......................... 58
5 Tests and Results 62
5.1 LLVM Backend Validation ................................. 62
5.2 CPU Implementation .................................. 65
5.2.1 Pre-scan RTL Synthesis ............................ 66
5.2.2 Post-scan DFT Synthesis ............................ 66
5.3 Additional Tools ...................................... 67
5.3.1 Clang .............................................. 67
5.3.2 ELF to Memory ..................................... 68
5.3.3 Assembler .......................................... 68
5.3.4 Disassembler ......................................... 68

6 Conclusions 69
6.1 Future Work ............................................ 69
6.2 Project Conclusions .................................... 70

References 71

I Guides I-1
I.1 Building LLVM-CJG ..................................... I-1
I.1.1 Downloading LLVM .................................. I-1
I.1.2 Importing the CJG Source Files .................... I-2
I.1.3 Modifying Existing LLVM Files ..................... I-2
I.1.4 Importing Clang ..................................... I-5
I.1.5 Building the Project ................................ I-8
I.1.6 Usage ............................................... I-9
I.1.6.1 Using llc ........................................ I-9
I.1.6.2 Using Clang ..................................... I-10
I.1.6.3 Using ELF to Memory ............................ I-10
I.2 LLVM Backend Directory Tree ........................ I-11

II Source Code II-1
II.1 CJG RISC CPU RTL .................................... II-1
II.1.1 Opcodes Header .................................... II-1
II.1.2 Definitions Header ................................ II-2
II.1.3 Pipeline ........................................... II-3
II.1.4 Clock Generator ................................... II-32
II.1.5 ALU ............................................... II-33
II.1.6 Shifter ............................................. II-35
II.1.7 Data Stack ......................................... II-38
II.1.8 Call Stack .......................................... II-39
II.1.9 Testbench ........................................... II-40
II.2 ELF to Memory ......................................... II-45
List of Figures

2.1 Aho Ullman Model .................................................. 6
2.2 Davidson Fraser Model ............................................. 6

3.1 Status Register Bits ............................................... 12
3.2 Program Counter Bits ............................................... 13
3.3 Stack Pointer Register ............................................. 13
3.4 CJG RISC CPU Functional Block Diagram ......................... 15
3.5 Four-Stage Pipeline ............................................... 16
3.6 Four-Stage Pipeline Block Diagram ................................ 16
3.7 Clock Phases ....................................................... 19
3.8 Load and Store Instruction Word .................................. 19
3.9 Data Transfer Instruction Word .................................... 20
3.10 Flow Control Instruction Word .................................... 21
3.11 Register-Register Manipulation Instruction Word ............... 23
3.12 Register-Immediate Manipulation Instruction Word .............. 23
3.13 Register-Register Shift and Rotate Instruction Word ............ 24
3.14 Register-Immediate Manipulation Instruction Word .............. 24

4.1 CJGMCTargetDesc.h Inclusion Graph ............................. 32
4.2 Initial myDouble:entry SelectionDAG ............................ 43
4.3 Initial myDouble:if.then SelectionDAG ........................... 44
4.4 Initial myDouble:if.end SelectionDAG ............................ 45
4.5 Optimized myDouble:entry SelectionDAG ........................ 47
4.6 Legalized myDouble:entry SelectionDAG ........................ 48
4.7 Selected myDouble:entry SelectionDAG ........................... 52
4.8 Selected myDouble:if.then SelectionDAG ........................ 53
4.9 Selected myDouble:if.end SelectionDAG ........................ 54

5.1 myDouble Simulation Waveform .................................... 64
# List of Listings

4.1 TableGen Register Set Definitions .................................. 30
4.2 TableGen AsmWriter Output ........................................... 30
4.3 TableGen RegisterInfo Output ........................................ 30
4.4 General Purpose Registers Class Definition .......................... 33
4.5 Return Calling Convention Definition ................................ 34
4.6 Special Operand Definitions .......................................... 35
4.7 Base CJG Instruction Definition ...................................... 36
4.8 Base ALU Instruction Format Definitions ............................ 37
4.9 Completed ALU Instruction Definitions ............................... 38
4.10 Completed Jump Conditional Instruction Definition ................ 40
4.11 Reserved Registers Description Implementation ................... 41
4.12 `myDouble` C Implementation ....................................... 41
4.13 `myDouble` LLVM IR Code ............................................ 42
4.14 Custom `SDNode` TableGen Definitions ............................... 49
4.15 Target-Specific `SDNode` Operation Definitions .................... 49
4.16 Jump Condition Code Encoding ...................................... 49
4.17 Target-Specific `SDNode` Operation Implementation ............... 50
4.18 Initial `myDouble` Machine Instruction List ....................... 55
4.19 Final `myDouble` Machine Instruction List .......................... 56
4.20 Custom `printMemSrcOperand` Implementation ...................... 58
4.21 Final `myDouble` Assembly Code .................................... 58
4.22 Custom `getMemSrcValue` Implementation ............................ 59
4.23 Base Load and Store Instruction Format Definitions ................ 60
4.24 CodeEmitter TableGen Backend Output for Load .................... 60
4.25 Disassembled `myDouble` Machine Code .............................. 61
4.26 `myDouble` Machine Code ........................................... 61

5.1 Modified `myDouble` Assembly Code ................................ 65
List of Tables

3.1 Description of Status Register Bits ........................................ 12
3.2 Addressing Mode Descriptions ................................................ 19
3.3 Load and Store Instruction Details ......................................... 20
3.4 Data Transfer Instruction Details ............................................ 20
3.5 Jump Condition Code Description .......................................... 22
3.6 Flow Control Instruction Details ............................................ 22
3.7 Manipulation Instruction Details ............................................ 23
3.8 Shift and Rotate Instruction Details ....................................... 25
4.1 Register Map for myDouble .................................................. 57
5.1 Pre-scan Netlist Area Results ............................................... 66
5.2 Pre-scan Netlist Power Results ............................................. 66
5.3 Post-scan Netlist Area Results ............................................. 67
5.4 Post-scan Netlist Power Results ............................................. 67
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
to 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
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
2.1 CPU Design

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.
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 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).\textsuperscript{1} The RTL then undergoes an expansion process followed by a recognizer which

\textsuperscript{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 Compiler Design

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 Instruction Set Architecture

3.1.1 Register File

The register file is composed of 32 individual 32-bit registers denoted as $r_0$ through $r_{31}$. All of the registers are general purpose with the exception of $r_0$-$r_2$, which are designated as special purpose registers.

The first special purpose register is the status register (SR), which is stored in $r_0$. 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](image)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>The carry bit. This is set to 1 if the result of a manipulation instruction produced a carry and set to 0 otherwise</td>
</tr>
<tr>
<td>N</td>
<td>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</td>
</tr>
<tr>
<td>V</td>
<td>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</td>
</tr>
<tr>
<td>Z</td>
<td>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</td>
</tr>
</tbody>
</table>

Table 3.1: Description of Status Register Bits

The next special purpose register is the program counter (PC) register, which is stored in $r_1$. 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.

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.

### 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.

![CJG RISC CPU Functional Block Diagram](image-url)

Figure 3.4: CJG RISC CPU Functional Block Diagram
### 3.2 Hardware Implementation

<table>
<thead>
<tr>
<th>Pipeline Stage</th>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>$I_0$ $I_1$ $I_2$ $I_3$ $I_4$ $I_5$ ...</td>
</tr>
<tr>
<td>OF</td>
<td>$I_0$ $I_1$ $I_2$ $I_3$ $I_4$ ...</td>
</tr>
<tr>
<td>EX</td>
<td>$I_0$ $I_1$ $I_2$ $I_3$ ...</td>
</tr>
<tr>
<td>WB</td>
<td>$I_0$ $I_1$ $I_2$ ...</td>
</tr>
<tr>
<td>Clock Cycle</td>
<td>1 2 3 4 5 6 ...</td>
</tr>
</tbody>
</table>

![Figure 3.5: Four-Stage Pipeline](image)

![Instruction Fetch → Operand Fetch → Execute → Write Back](image)

#### 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
3.2 Hardware Implementation

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 $R_x$ register, followed by an instruction $I_{n+1}$, that uses $R_x$ 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 $R_x$ 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 $R_x$. 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

\[^{1}R_x\] 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, $\phi_1$ and $\phi_2$ (shown in Fig. 3.7), from the main system clock. The $\phi_1$ clock is responsible for all of the pipeline logic while $\phi_2$ acts as the memory clock for both the instruction and data memory. Additionally, the $\phi_2$ 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.
3.3 Instruction Details

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.

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.

<table>
<thead>
<tr>
<th>Mode</th>
<th>( R_x ) (^2)</th>
<th>Control</th>
<th>Effective Address Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Direct</td>
<td>Not 0</td>
<td>1</td>
<td>The value of the ( R_x ) register operand</td>
</tr>
<tr>
<td>Absolute</td>
<td>0</td>
<td>1</td>
<td>The value in the address field</td>
</tr>
<tr>
<td>Indexed</td>
<td>Not 0</td>
<td>0</td>
<td>The value of the ( R_x ) register operand + the value in the address field</td>
</tr>
<tr>
<td>PC Relative</td>
<td>0</td>
<td>0</td>
<td>The value of the PC register + the value in the address field</td>
</tr>
</tbody>
</table>

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
3.3 Instruction Details

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>LD</td>
<td>0x0</td>
<td>Load the value in memory at the effective address or I/O peripheral into the ( R_i ) register</td>
</tr>
<tr>
<td>Store</td>
<td>ST</td>
<td>0x1</td>
<td>Store the value of the ( R_i ) register into memory at the effective address or I/O peripheral</td>
</tr>
</tbody>
</table>

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.

```
31 28 27 22 21 17 16 15 0

<table>
<thead>
<tr>
<th>Opcode</th>
<th>( R_i )</th>
<th>( R_j )</th>
<th>Control</th>
<th>Constant</th>
</tr>
</thead>
</table>
```

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 \( R_j \).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>CPY</td>
<td>0x2</td>
<td>Copy the value from the source operand into the ( R_i ) register</td>
</tr>
<tr>
<td>Push</td>
<td>PUSH</td>
<td>0x3</td>
<td>Push the value from the source operand onto the top of the stack and then increment the stack pointer</td>
</tr>
<tr>
<td>Pop</td>
<td>POP</td>
<td>0x4</td>
<td>Decrement the stack pointer and then pop the value from the top of the stack into the ( R_i ) register.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>31</th>
<th>27</th>
<th>26</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>R_i</td>
<td>C</td>
<td>N</td>
<td>V</td>
<td>Z</td>
<td>0</td>
<td>Control</td>
<td>Address</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
3.3 Instruction Details

### Jump Condition Code Description

<table>
<thead>
<tr>
<th>C</th>
<th>N</th>
<th>V</th>
<th>Z</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>JMP / JU</td>
<td>Jump unconditionally</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>JC</td>
<td>Jump if carry</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>JN</td>
<td>Jump if negative</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>JV</td>
<td>Jump if overflow</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>JZ / JEQ</td>
<td>Jump if zero / equal</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>JNC</td>
<td>Jump if not carry</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>JNN</td>
<td>Jump if not negative</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>JNV</td>
<td>Jump if not overflow</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>JNZ / JNE</td>
<td>Jump if not zero / not equal</td>
</tr>
</tbody>
</table>

Table 3.5: Jump Condition Code Description

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jump</td>
<td>J{CC}(^3)</td>
<td>0x5</td>
<td>Conditionally set the PC to the effective address</td>
</tr>
<tr>
<td>Call</td>
<td>CALL</td>
<td>0x6</td>
<td>Push the PC followed by the SR onto the call stack, set the PC to the effective address</td>
</tr>
<tr>
<td>Return</td>
<td>RET</td>
<td>0x7</td>
<td>Pop the top of call stack into the SR, then pop the next value into the PC</td>
</tr>
</tbody>
</table>

Table 3.6: Flow Control Instruction Details

### 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\(^3\)

\(^3\)The value of \{CC\} depends on the condition code; see the Mnemonic column in Table 3.5
through the ALU.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>R_i</th>
<th>R_j</th>
<th>R_k</th>
<th>0</th>
</tr>
</thead>
</table>

Figure 3.11: Register-Register Manipulation Instruction Word

<table>
<thead>
<tr>
<th>Opcode</th>
<th>R_i</th>
<th>R_j</th>
<th>Immediate</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Figure 3.12: Register-Immediate Manipulation Instruction Word

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>ADD</td>
<td>0x8</td>
<td>Store R_j + SRC_2 in R_i</td>
</tr>
<tr>
<td>Subtract</td>
<td>SUB</td>
<td>0x9</td>
<td>Store R_j − SRC_2 in R_i</td>
</tr>
<tr>
<td>Compare</td>
<td>CMP</td>
<td>0xA</td>
<td>Compute R_j − SRC_2 and discard result</td>
</tr>
<tr>
<td>Negate</td>
<td>NOT</td>
<td>0xB</td>
<td>Store −R_j in R_i</td>
</tr>
<tr>
<td>AND</td>
<td>AND</td>
<td>0xC</td>
<td>Store R_j &amp; SRC_2 in R_i</td>
</tr>
<tr>
<td>Bit Clear</td>
<td>BIC</td>
<td>0xD</td>
<td>Store R_j &amp; −SRC_2 in R_i</td>
</tr>
<tr>
<td>OR</td>
<td>OR</td>
<td>0xE</td>
<td>Store R_j</td>
</tr>
<tr>
<td>Exclusive OR</td>
<td>XOR</td>
<td>0xF</td>
<td>Store R_j ^ SRC_2 in R_i</td>
</tr>
<tr>
<td>Signed Multiplication</td>
<td>MUL</td>
<td>0x1A</td>
<td>Store R_j × SRC_2 in R_i</td>
</tr>
<tr>
<td>Unsigned Division</td>
<td>DIV</td>
<td>0x1B</td>
<td>Store R_j ÷ SRC_2 in R_i</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>31</th>
<th>27</th>
<th>26</th>
<th>22</th>
<th>21</th>
<th>17</th>
<th>16</th>
<th>12</th>
<th>11</th>
<th>4</th>
<th>3</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Ri</td>
<td>Rj</td>
<td>Rk</td>
<td>0</td>
<td>Mode</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.13: Register-Register Shift and Rotate Instruction Word

<table>
<thead>
<tr>
<th>31</th>
<th>27</th>
<th>26</th>
<th>22</th>
<th>21</th>
<th>17</th>
<th>16</th>
<th>11</th>
<th>10</th>
<th>4</th>
<th>3</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>Ri</td>
<td>Rj</td>
<td>Immediate</td>
<td>0</td>
<td>Mode</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
### 3.3 Instruction Details

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Mnemonic</th>
<th>Opcode</th>
<th>Mode</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift right logical</td>
<td>SRL</td>
<td>0x10</td>
<td>0x0</td>
<td>Shift R&lt;sub&gt;j&lt;/sub&gt; right logically by SRC&lt;sub&gt;2&lt;/sub&gt; bits and store in R&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>Shift left logical</td>
<td>SLL</td>
<td>0x10</td>
<td>0x1</td>
<td>Shift R&lt;sub&gt;j&lt;/sub&gt; left logically by SRC&lt;sub&gt;2&lt;/sub&gt; bits and store in R&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>Shift right arithmetic</td>
<td>SRA</td>
<td>0x10</td>
<td>0x2</td>
<td>Shift R&lt;sub&gt;j&lt;/sub&gt; right arithmetically by SRC&lt;sub&gt;2&lt;/sub&gt; bits and store in R&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>Rotate right</td>
<td>RTR</td>
<td>0x10</td>
<td>0x4</td>
<td>Rotate R&lt;sub&gt;j&lt;/sub&gt; right by SRC&lt;sub&gt;2&lt;/sub&gt; bits and store in R&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>Rotate left</td>
<td>RTL</td>
<td>0x10</td>
<td>0x5</td>
<td>Rotate R&lt;sub&gt;j&lt;/sub&gt; left by SRC&lt;sub&gt;2&lt;/sub&gt; bits and store in R&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>Rotate right through carry</td>
<td>RRC</td>
<td>0x10</td>
<td>0x6</td>
<td>Rotate R&lt;sub&gt;j&lt;/sub&gt; right through carry by SRC&lt;sub&gt;2&lt;/sub&gt; bits and store in R&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
<tr>
<td>Rotate left through carry</td>
<td>RLC</td>
<td>0x10</td>
<td>0x7</td>
<td>Rotate R&lt;sub&gt;j&lt;/sub&gt; left through carry by SRC&lt;sub&gt;2&lt;/sub&gt; bits and store in R&lt;sub&gt;i&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

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 Structure and Tools

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. 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)
4.1 Structure and Tools

```cpp
// 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

```cpp
// 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

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

Listing 4.3: TableGen RegisterInfo Output
4.2 Custom Target Implementation

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.
4.2 Custom Target Implementation

Figure 4.1: CJGMCTargetDesc.h Inclusion Graph
4.2 Custom Target Implementation

4.2.1 Abstract Target Description

As discussed 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.

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

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 CJG-CallingConv.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.

Listing 4.5: Return Calling Convention Definition

```cpp
//===----------------------------------------------------------------------===//
// CJG Return Value Calling Convention
//===----------------------------------------------------------------------===//
def RetCC_CJG : CallingConv<[
  // Promote i8/i16 arguments to i32.
  CCIfType<[i8, i16], CCPromoteToType<i32>>,

  // i32 are returned in registers R24-R31
  CCIfType<[i32], CCAssignToReg<[R24, R25, R26, R27, R28, R29, R30, R31]>>,

  // Integer values get stored in stack slots that are 4 bytes in
  // size and 4-byte aligned.
  CCIfType<[i32], CCAssignToStack<4, 4>>
]>;
```

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 \texttt{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 \texttt{CJGInstrInfo.td} and are shown in Listing 4.6. The \texttt{PrintMethod} and \texttt{EncoderMethod} define the names of custom C++ functions to be called when either \textit{printing} the operand in assembly or \textit{encoding} the operand in the machine code.

```
// Address operand for indexed addressing mode
def memsrc : Operand<i32> {
    let PrintMethod = "printMemSrcOperand";
    let EncoderMethod = "getMemSrcValue";
    let MIOperandInfo = (ops GPRegs, CJGimm16);
}

// Operand for printing out a condition code.
def cc : Operand<i32> {
    let PrintMethod = "printCCOperand";
}
```

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 \texttt{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
4.2 Custom Target Implementation

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

```cpp
//===----------------------------------------------------------------------===//
//=== 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 instructions 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 CJG-InstrInfo.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
4.2 Custom Target Implementation

```cpp
//===----------------------------------------------------------------------===//
// 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 \texttt{ADD} instruction on line 23 of Listing 4.9 is expanded into an \texttt{ADD\texttt{rr}} and \texttt{ADD\texttt{ri}} instruction definition).

```cpp
let Defs = [SR] in {
  multiclass ALU<bits<5> opcode, string opstr, SDNode opnode> {
    def rr : ALU_Inst_RR<opcode, (outs GPRegs:$ri),
            (ins GPRegs:$rj, GPRegs:$rk),
            `strconcat(opstr, "$ri, $rj, $rk"),
            [(set GPRegs:$ri, (opnode GPRegs:$rj, GPRegs:$rk)),
             (implicit SR)]> {
      
    }
    def ri : ALU_Inst_RI<opcode, (outs GPRegs:$ri),
            (ins GPRegs:$rj, CJSImm16:$const),
            `strconcat(opstr, "$ri, $rj, $const"),
            [(set GPRegs:$ri, (opnode GPRegs:$rj, CJSImm16:$const)),
             (implicit SR)]> {
      
    }
    defm ADD : ALU<0b01000, "add", add>;
    defm SUB : ALU<0b01001, "sub", sub>;
    defm AND : ALU<0b01100, "and", and>;
    defm OR  : ALU<0b01110, "or", or>;
    defm XOR : ALU<0b01111, "xor", xor>;
    defm MUL : ALU<0b11010, "mul", mul>;
    defm DIV : ALU<0b11011, "div", udiv>;
    ...
  } // 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 \texttt{ADDri} definition. The \texttt{outs} and \texttt{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 \texttt{GPRegs} register class and it is stored in the class’s \texttt{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 \texttt{GPRegs} register class while the second is defined by the custom \texttt{CJGimm16} operand type. The first operand is stored in the class’s \texttt{rj} variable and the second operand is stored in the class’s \texttt{rk} variable. Line 17 shows the assembly string definition; the \texttt{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 \texttt{opnode} parameter is passed in from the third parameter of the \texttt{defm} declaration shown on line 23. The \texttt{opnode} type is an \texttt{SDNode} class which represents a node in the DAG used for instruction selection (called the SelectionDAG). In this example the \texttt{SDNode} is \texttt{add}, which is already defined by LLVM. Some instructions, however, need a custom \texttt{SDNode} implementation. This pattern will be matched if there is an \texttt{add} node in the SelectionDAG with two operands, where one is a register in the \texttt{GPRegs} class and the other a constant. The destination of the node must also be a register in the \texttt{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

\begin{verbatim}
add r4, r5, r6
\end{verbatim}

where \texttt{r5} is added to \texttt{r6} and the result is stored in \texttt{r4}. This instruction is said to \textit{define} \texttt{r4} and \textit{use} \texttt{r5} and \texttt{r6}. Because all \texttt{add} instructions can modify the status register, this instruction is also said to implicitly \textit{define} \texttt{SR}. This is expressed in TableGen using the \texttt{Defs} and \texttt{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 \texttt{Uses} keyword. This can be
4.2 Custom Target Implementation

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).

```plaintext
// Conditional jump
let isBranch = 1, isTerminator = 1, Uses=[SR] in {
  def JCC : FC_Inst<0b00101,
    (outs), (ins jmptarget:$addr, cc:$condition),
    "j$condition\t$addr",
    [(CJGbrcc bb:$addr, imm:$condition)]> {
    // set ri to 0 and control to 1 for absolute addressing mode
    let ri = 0b00000;
    let control = 0b1;
  }
} // 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 instruction selector.
4.2 Custom Target Implementation

```cpp
BitVector CJGRegisterInfo::getReservedRegs(const MachineFunction &MF) const {
    BitVector Reserved(getNumRegs());
    Reserved.set(CJG::SR); // status register
    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 cannot 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.

```c
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
4.2 Custom Target Implementation

```
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
4.2 Custom Target Implementation

Figure 4.2: Initial myDouble:entry SelectionDAG
Figure 4.3: Initial myDouble::if.then SelectionDAG
4.2 Custom Target Implementation

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 CJG-ISelLowering.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
4.2 Custom Target Implementation

Figure 4.6: Legalized myDouble:entry SelectionDAG
4.2 Custom Target Implementation

Listing 4.14: Custom SDNode TableGen Definitions

```cpp
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

```cpp
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.

```cpp
SDValue CJGTargetLowering::LowerOperation(SDValue Op, SelectionDAG &DAG) const {
    switch (Op.getOpcode()) {
    case ISD::BR_CC:        return LowerBR_CC(Op, DAG);
    ...
    default:                 llvm_unreachable("unimplemented operand");
    }
}
```

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.
4.2 Custom Target Implementation

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 “CJG1SD::CMP” and “CJG1SD::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
4.2 Custom Target Implementation

Figure 4.8: Selected `myDouble:if.then` SelectionDAG
4.2 Custom Target Implementation

scheduler input for myDouble:if.end

Figure 4.9: Selected myDouble:if.end SelectionDAG
4.2 Custom Target Implementation

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.

```
BB#0: derived from LLVM BB %entry
  Live Ins: %R4
  %vreg0<def> = COPY %R4; GPRegs:%vreg0
  CMPri %vreg0, 0, %SR<imp-def>; GPRegs:%vreg0
  JCC <BB#2>, 14, %SR<imp-use>
  JMP <BB#1>

BB#1: derived from LLVM BB %if.then
  Predecessors according to CFG: BB#0
  %vreg2<def> = CPYri 0; GPRegs:%vreg2
  %R24<def> = COPY %vreg2; GPRegs:%vreg2
  RET %R24<imp-use>

BB#2: derived from LLVM BB %if.end
  Predecessors according to CFG: BB#0
  %vreg1<def> = ADDrr %vreg0, %vreg0, %SR<imp-def,dead>;
                 GPRegs:%vreg1,%vreg0,%vreg0
  %R24<def> = COPY %vreg1; GPRegs:%vreg1
  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
4.2 Custom Target Implementation

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.

```
BB#0: derived from LLVM BB %entry
  Live Ins: %R4 %SR
  PUSH %SR<kill>, %SP<imp-def>
  CMPri %R4, 0, %SR<imp-def>
  JCC <BB#1>, 1, %SR<imp-use>

BB#2: derived from LLVM BB %if.end
  Live Ins: %R4
  Predecessors according to CFG: BB#0
  %R24<def> = ADDrr %R4<kill>, %R4, %SR<imp-def,dead>
  %SR<def> = POP %SP<imp-def>
  RET %R24<imp-use>

BB#1: derived from LLVM BB %if.then
  Predecessors according to CFG: BB#0
  %R24<def> = CPYri 0
  %SR<def> = POP %SP<imp-def>
  RET %R24<imp-use>
```

Listing 4.19: Final myDouble Machine Instruction List
4.2 Custom Target Implementation

<table>
<thead>
<tr>
<th>Virtual Register</th>
<th>Physical Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>%vreg0</td>
<td>%R4</td>
</tr>
<tr>
<td>%vreg1</td>
<td>%R24</td>
</tr>
<tr>
<td>%vreg2</td>
<td>%R24</td>
</tr>
</tbody>
</table>

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 CJG-AsmPrinter 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
   // %entry
   push r0
   cmp r4, 0
   jeq BB0_1
   // %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.

```cpp
// Encode 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
unsigned CJGMCCodeEmitter::getMemSrcValue(const MCInst &MI, unsigned OpIdx,
                                           SmallVectorImpl<MCFixup> &Fixups,
                                           const MCSubtargetInfo &STI) const {
  unsigned Bits = 0;
  const MCOperand &RegOp = MI.getOperand(OpIdx);
  const MCOperand &ImmOp = MI.getOperand(OpIdx + 1);
  Bits |= (getMachineOpValue(MI, RegOp, Fixups, STI) << 16);
  Bits |= (unsigned)ImmOp.getImm() & 0xffff;
  return Bits;
}
```

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.

```cpp
class LS_Inst<bits<5> opcode, dag outs, dag ins, string asmstr,
              list<dag> pattern>
    : InstCJG<outs, ins, asmstr, pattern> {

    bits<5> ri;
    bits<1> control;
    bits<21> addr;

    let Opcode = opcode;
    let Inst{26-22} = ri;
    let Inst{21-17} = addr{20-16}; // rj
    let Inst{16} = control;
    let Inst{15-0} = addr{15-0};
}
```

Listing 4.23: Base Load and Store Instruction Format Definitions

```cpp
case CJG::LD: {
    // op: ri
    op = getMachineOpValue(MI, MI.getOperand(0), Fixups, STI);
    Value |= (op & UINT64_C(31)) << 22;
    // op: addr
    op = getMemSrcValue(MI, 1, Fixups, STI);
    Value |= (op & UINT64_C(2031616)) << 1;
    Value |= op & UINT64_C(65535);
    break;
}
```

Listing 4.24: CodeEmitter TableGen Backend Output for Load

The target-specific machine instructions are placed into the “.text” section of the ELF
4.2 Custom Target Implementation

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.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>push r0</td>
<td>// @00000000 00</td>
</tr>
<tr>
<td>2</td>
<td>cmp r4, 0</td>
<td>// @00000004 01</td>
</tr>
<tr>
<td>3</td>
<td>jeq label_0</td>
<td>// @00000008 18</td>
</tr>
<tr>
<td>4</td>
<td>add r24, r4, r4</td>
<td>// @0000000C 00</td>
</tr>
<tr>
<td>5</td>
<td>pop r0</td>
<td>// @00000010 00</td>
</tr>
<tr>
<td>6</td>
<td>ret</td>
<td>// @00000014 00</td>
</tr>
<tr>
<td>7</td>
<td>label_0: cpy r24, 0</td>
<td>// @00000018 00</td>
</tr>
<tr>
<td>8</td>
<td>pop r0</td>
<td>// @0000001C 00</td>
</tr>
<tr>
<td>9</td>
<td>ret</td>
<td>// @00000020 00</td>
</tr>
</tbody>
</table>

Listing 4.25: Disassembled myDouble Machine Code

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>@00000000 18000000 // push r0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>@00000004 50080001 // cmp r4, 0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>@00000008 28050018 // jeq label_0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>@0000000C 46084000 // add r24, r4, r4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>@00000010 20000000 // pop r0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>@00000014 38000000 // ret</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>@00000018 16010000 // label_0: cpy r24, 0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>@0000001C 20000000 // pop r0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>@00000020 38000000 // ret</td>
<td></td>
</tr>
</tbody>
</table>

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 $\texttt{readmemh}$ Verilog function is used in the testbench to initialize the program memory with the machine code from the ELF object file. An intermediate tool, \texttt{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 $\texttt{readmemh}$. Additionally, the CJG disassembler is used to modify the generated code to make it more friendly to the simulation environment.

For example, consider the \texttt{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 \texttt{r4} from the CPU’s GPIO input port, which is memory mapped to address \texttt{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 \texttt{done} label seen on line 10. This writes the result from \texttt{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 \texttt{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 \texttt{0x18} (24), which is double \texttt{0xC}, just before the 160,000 \texttt{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 \texttt{int}, such as \texttt{short int}, and \texttt{char}, or more specifically, \texttt{i16}, \texttt{i8}, and \texttt{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
5.2 CPU Implementation

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
5.2 CPU Implementation

<table>
<thead>
<tr>
<th>Cell</th>
<th>Global Cell Area</th>
<th></th>
<th>Local Cell Area (µm²)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Absolute Total (µm²)</td>
<td>Percent Total</td>
<td>Combinational</td>
<td>Noncombinational</td>
</tr>
<tr>
<td>cjg_risc</td>
<td>94941.7219</td>
<td>100.0</td>
<td>11184.4803</td>
<td>15004.0802</td>
</tr>
<tr>
<td>cjg_alu</td>
<td>11650.3200</td>
<td>12.3</td>
<td>629.2800</td>
<td>0.0000</td>
</tr>
<tr>
<td>cjg_call_stack</td>
<td>24469.9207</td>
<td>25.8</td>
<td>6775.2000</td>
<td>17694.7207</td>
</tr>
<tr>
<td>cjg_clkgen</td>
<td>30.6000</td>
<td>0.0</td>
<td>14.7600</td>
<td>15.8400</td>
</tr>
<tr>
<td>cjg_data_stack</td>
<td>26258.4005</td>
<td>27.7</td>
<td>3886.5601</td>
<td>22371.8404</td>
</tr>
<tr>
<td>cjg_shifter</td>
<td>4805.6401</td>
<td>5.1</td>
<td>4805.6401</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

Table 5.1: Pre-scan Netlist Area Results

<table>
<thead>
<tr>
<th>Internal (mW)</th>
<th>Switching (mW)</th>
<th>Leakage (µW)</th>
<th>Total (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.6935</td>
<td>0.1759</td>
<td>3.1975</td>
<td>7.8729</td>
</tr>
</tbody>
</table>

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 µm². 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 µm²). 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,
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.

<table>
<thead>
<tr>
<th>Cell</th>
<th>Global Cell Area</th>
<th>Local Cell Area (µm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Absolute Total (µm²)</td>
<td>Percent Total</td>
</tr>
<tr>
<td>cjg_risc</td>
<td>94912.9219</td>
<td>100.0</td>
</tr>
<tr>
<td>cjg_alu</td>
<td>11633.7600</td>
<td>12.3</td>
</tr>
<tr>
<td>cjg_call_stack</td>
<td>24469.9207</td>
<td>25.8</td>
</tr>
<tr>
<td>cjg_clkgen</td>
<td>30.6000</td>
<td>0.0</td>
</tr>
<tr>
<td>cjg_data_stack</td>
<td>26258.4005</td>
<td>27.7</td>
</tr>
<tr>
<td>cjg_shifter</td>
<td>4805.6401</td>
<td>5.1</td>
</tr>
</tbody>
</table>

Table 5.3: Post-scan Netlist Area Results

<table>
<thead>
<tr>
<th>Internal (mW)</th>
<th>Switching (mW)</th>
<th>Leakage (µW)</th>
<th>Total (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.6918</td>
<td>0.1759</td>
<td>3.1954</td>
<td>7.8711</td>
</tr>
</tbody>
</table>

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
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


[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-


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


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 Building LLVM-CJG

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:

```
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:

```
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:

```c
#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:

```c
#include/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";

    }
```

```c
@@ -298,6 +302,7 @@ Triple::ArchType Triple::getArchTypeForLLVMName(StringRef
        .Case("wasm64", wasm64)
        .Case("renderscript32", renderscript32)
        .Case("renderscript64", renderscript64)
+       .Case("cjg", cjg)
```
.Default(UnknownArch);
}

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
  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:
    case Triple::wasm32:
    case Triple::wasm64:
    case Triple::xcore:
      case Triple::cjg:
    return 32;

    case llvm::Triple::aarch64:
  
  case Triple::shave:
  case Triple::wasm32:
  case Triple::renderscript32:
  case Triple::cjg:
    // Already 32-bit.
    break;

  case Triple::xcore:
  case Triple::sparcel:
  case Triple::shave:
    case Triple::cjg:
    T.setArch(UnknownArch);
    break;

  case Triple::arm:
  case Triple::thumb:
  }
I.1 Building LLVM-CJG

5. Add CJG to the cmake Target build configuration:

```
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
    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 = SignedInt;
    +         PtrDiffType = SignedInt;
    +         IntPtrType = SignedInt;
    +         WCharType = unsignedChar;
    +         WIntType = unsignedInt;
    +         UseZeroLengthBitfieldAlignment = true;
    +                 "-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::VoidPtr builtinVaList;
    +     }
    ```
I.1 Building LLVM-CJG

```cpp
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;
}
}
```

```cpp
//===----------------------------------------------------------------------===//
@@ -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`:

```diff
--- lib/CodeGen/TargetInfo.cpp
+++ b/lib/CodeGen/TargetInfo.cpp
@@ -8349,8 +8349,25 @@ public:
    }
    
    // end anonymous namespace

-/+9044,4 +9097,7 @@ static TargetInfo *AllocateTarget(const llvm::Triple &Triple,
+    case llvm::Triple::cjg:
+        return new CJGTargetInfo(Triple, Opts);
```

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.
I.1 Building LLVM-CJG

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
    │                           └── CJGISelFormats.td
    │                               └── CJGISelInfo.cpp
    │                                   └── CJGISelInfo.h
    │                                       └── CJGISelInfo.td
    │                                           └── CJGMCInstLower.cpp
    │                                               └── CJGMCInstLower.h
    │                                                   └── CJGMCInstLowering.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

├── LLVMBuild.txt
├── MCTargetDesc/
│    └── CJGAsmBackend.cpp
│        └── CJGELFObjectWriter.cpp
```
I.2 LLVM Backend Directory Tree

```
    CJGFixupKinds.h
    CJGMCAsmInfo.cpp
    CJGMCAsmInfo.h
    CJGMCCodeEmitter.cpp
    CJGMCCTargetDesc.cpp
    CJGMCCTargetDesc.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

```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
```
II.1 CJG RISC CPU RTL

II.1.2 Definitions Header

// Instruction word slices
#define OPCODE 31:27
#define REG_I 26:22
#define REG_J 21:17
#define REG_K 16:12
#define ALU_CONSTANT 16:1
#define ALU_CONSTANT_MSB 16
#define ALU_CONTROL 0
#define DT_CONTROL 16
#define DT_CONSTANT 15:0
#define DT_CONSTANT_MSB 15
#define JMP_CODE 21:18
#define JMP_ADDR 15:0
#define JMP_CONTROL 16
#define RS_CONTROL 0
#define RS_OPCODE 3:1
#define RS_CONSTANT 16:11

// Jump codes
#define JU 4'b0000
II.1 CJG RISC CPU RTL

II.1.3 Pipeline

---

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

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

// Any instruction with a writeback operation

// ALU instructions

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

\`define LOAD_MMIO(dest,bits,expr) \nif (dm_address < `MMIO_START_ADDR) begin \ndest <= dm_out[bits] expr; \nend \nelse begin \ncase (dm_address) \n`MMIO_GPIO_IN: begin \ndest <= gpio_in[bits] expr; \nend \ndefault: begin \ndest <= temp_wb[bits] expr; \nend \nendcase \nend

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
// generated clock phases
output clk_p1, \quad \text{\textit{clock phase 0}}
output clk_p2, \quad \text{\textit{clock phase 1}}

// dft
input scan_in0,
input scan_en,
input test_mode,
output scan_out0
);

// integer for resetting arrays
integer i;

// register file
reg[31:0] reg_file[31:0];

// program counter register (program memory address)
assign pm_address = reg_file[\texttt{\text{REG_PC}}][15:0];

// temp address for jumps/calls
reg[15:0] temp_address;

// pipelined instruction registers
reg[31:0] instruction_word[3:1];

// address storage for each instruction
reg[13:0] instruction_addr[3:1];

// opcode slices
reg[4:0] opcode[3:0];

// TODO: is this even ok? 2d wires dont seem to work in simvision
always @(instruction_word[3] or instruction_word[2] or instruction_word[1] or pm_out)
begin
  opcode[0] = pm_out[\texttt{OPCODE}];
  opcode[1] = instruction_word[1][\texttt{OPCODE}];
  opcode[2] = instruction_word[2][\texttt{OPCODE}];
  opcode[3] = instruction_word[3][\texttt{OPCODE}];
end

// stall signals
reg[3:0] stall_cycles;
reg stall[3:0];

// temp writeback register
reg[31:0] temp_wb; // general purpose
reg[31:0] temp_sp; // stack pointer
II.1 CJG RISC CPU RTL

// data stack stuff
reg[31:0] data_stack_data;
reg[5:0] data_stack_addr;
reg data_stack_push;
reg data_stack_pop;
wire[31:0] data_stack_out;

// call stack stuff
reg[31:0] call_stack_data;
reg call_stack_push;
reg call_stack_pop;
wire[31:0] call_stack_out;

// ALU stuff
reg[31:0] alu_a, alu_b, temp_sr;
reg[3:0] alu_opcode;
wire[31:0] alu_result;
wire alu_c, alu_n, alu_v, alu_z;

// Shifter stuff
reg[31:0] shifter_operand;
reg[5:0] shifter_modifier;
reg shifter_carry_in;
reg[2:0] shifter_opcode;
wire[31:0] shifter_result;
wire shifter_carry_out;

// Clock phase generator
cjg_clkgen clkgen(
  .reset(reset),
  .clk(clk),
  .clk_p1(clk_p1),
  .clk_p2(clk_p2),
  // dft
  .scan_in0(scan_in0),
  .scan_en(scan_en),
  .test_mode(test_mode),
  .scan_out0(scan_out0)
);

// Data Stack
cjg_mem_stack #(DEPTH(64), ADDRW(6)) data_stack(
  // inputs
  .clk(clk_p2),
  .reset(reset),
  .d(data_stack_data),
  .addr(data_stack_addr),
II.1 CJG RISC CPU RTL

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),
II.1 CJG RISC CPU RTL

```verilog
  .c(alu_c),
  .n(alu_n),
  .v(alu_v),
  .z(alu_z)
);

  // Shifter and rotater
  cjg_shifter shifter (  
    // dft  
    .reset(reset),
    .clk(clk),
    .scan_in0(scan_in0),
    .scan_en(scan_en),
    .test_mode(test_mode),
    .scan_out0(scan_out0),
    .operand(shifter_operand),
    .carry_in(shifter_carry_in),
    .modifier(shifter_modifier),
    .opcode(shifter_opcode),
    .result(shifter_result),
    .carry_out(shifter_carry_out)
  );

  // Here we go

  always @(posedge clk_p1 or negedge reset) begin
    if (~reset) begin  
      // reset
      reset_all;  
    end // if (~reset)
    else begin  
      // Main code

        // process stall signals
        stall[3] <= stall[2];
        stall[2] <= stall[1];
        stall[1] <= stall[0];

        if (stall_cycles != 0) begin
          stall[0] <= 1'b1;
          stall_cycles <= stall_cycles - 1'b1;
        end
        else begin
          stall[0] <= 1'b0;
    ```
II.1 CJG RISC CPU RTL

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,
256       `RS_IC, `MUL_IC, `DIV_IC: begin
257        if (instruction_word[3][`REG_I] == `REG_PC) begin
258          // Do not allow writing to the program counter
259          reg_file[`REG_PC] <= reg_file[`REG_PC];
260        end
261        else begin
262          reg_file[instruction_word[3][`REG_I]] <= temp_wb;
263        end
264      end
265
266      `PUSH_IC: begin
267        reg_file[`REG_SP] <= temp_sp; // incremented stack pointer
268      end
269
270      `POP_IC: begin
271        reg_file[`REG_SP] <= temp_sp; // decremented stack pointer
272        reg_file[instruction_word[3][`REG_I]] <= temp_wb;
273        data_stack_pop <= 1'b0;
274      end
275
276      `ST_IC: begin
277        dm_wren <= 1'b0;
278      end
279
280      `JMP_IC: begin
281        // check the status register
282        case (instruction_word[3][`JMP_CODE])
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  end
```
\`JN: begin
  if (reg_file[REG_SR][SR_N] == 1'b1) begin
    reg_file[REG_PC] <= {16'h0, temp_address};
  end
  else begin
    reg_file[REG_PC] <= reg_file[REG_PC];
  end
end

\`JV: begin
  if (reg_file[REG_SR][SR_V] == 1'b1) begin
    reg_file[REG_PC] <= {16'h0, temp_address};
  end
  else begin
    reg_file[REG_PC] <= reg_file[REG_PC];
  end
end

\`JZ: begin
  if (reg_file[REG_SR][SR_Z] == 1'b1) begin
    reg_file[REG_PC] <= {16'h0, temp_address};
  end
  else begin
    reg_file[REG_PC] <= reg_file[REG_PC];
  end
end

\`JNC: begin
  if (reg_file[REG_SR][SR_C] == 1'b0) begin
    reg_file[REG_PC] <= {16'h0, temp_address};
  end
  else begin
    reg_file[REG_PC] <= reg_file[REG_PC];
  end
end

\`JNN: begin
  if (reg_file[REG_SR][SR_N] == 1'b0) begin
    reg_file[REG_PC] <= {16'h0, temp_address};
  end
  else begin
    reg_file[REG_PC] <= reg_file[REG_PC];
  end
end

\`JNV: begin
  if (reg_file[REG_SR][SR_V] == 1'b0) begin
    reg_file[REG_PC] <= {16'h0, temp_address};
  end
end
```
else begin
    reg_file['REG_PC] <= reg_file['REG_PC];
end

`JNZ: begin
    if (reg_file['REG_SR][SR_Z] == 1'b0) begin
        reg_file['REG_PC] <= {16'h0, temp_address};
    end
    else begin
        reg_file['REG_PC] <= reg_file['REG_PC];
    end
end

`JGE: begin
    if (reg_file['REG_SR][SR_GE] == 1'b1) begin
        reg_file['REG_PC] <= {16'h0, temp_address};
    end
    else begin
        reg_file['REG_PC] <= reg_file['REG_PC];
    end
end

`JL: begin
    if (reg_file['REG_SR][SR_L] == 1'b1) begin
        reg_file['REG_PC] <= {16'h0, temp_address};
    end
    else begin
        reg_file['REG_PC] <= reg_file['REG_PC];
    end
end

default: begin
    reg_file['REG_PC] <= reg_file['REG_PC];
end
endcase // instruction_word[3][JMP_CODE]

end // JMP_IC

`CALL_IC: begin
    // jump to the routine address
    call_stack_push <= 1'b0;
    reg_file['REG_PC] <= {16'h0, temp_address};
end

`RET_IC: begin
    // pop the program counter
    call_stack_pop <= 1'b0;
    reg_file['REG_PC] <= {16'h0, temp_address};
end

default: begin
end
case // opcode[3]
case (opcode[3])
ADD_IC, SUB_IC, CMP_IC, NOT_IC, AND_IC, BIC_IC, OR_IC, XOR_IC, RS_IC,
MUL_IC, DIV_IC: begin
// set the status register from the alu output
reg_file[\`REG_SR] <= temp_sr;
end
default: begin
reg_file[\`REG_SR] <= reg_file[\`REG_SR];
end
case // opcode[3]
end // if (stall[3] == 1'b0)

// Machine cycle 2
// execution
if (stall[2] == 1'b0) begin
case (opcode[2])
ADD_IC, SUB_IC, CMP_IC, NOT_IC, AND_IC, BIC_IC, OR_IC, XOR_IC, CPY_IC,
MUL_IC, DIV_IC: begin
// set temp ALU out
temp_wb <= alu_result;

// Set status register
if (instruction_word[3][\`REG_I] == \`REG_SR &\`WB_INSTRUCTION(3)) begin
// data forward from the status register
temp_sr <= {temp_wb[31:6], alu_n, ~alu_n, alu_z, alu_v, alu_n, alu_c};
end
else begin
// take the current status register
temp_sr <= {reg_file[\`REG_SR][31:6], alu_n, ~alu_n, alu_z, alu_v, alu_n,
~alu_c};
end
// TODO: data forward from other sources in mc3
end
RS_IC: begin
// grab the output from the shifter
temp_wb <= shifter_result;

// if rotating through carry, set the new carry value
if ((instruction_word[2][`RS_OPCODE] == `RRC_SHIFT) ||
    (instruction_word[2][`RS_OPCODE] == `RLC_SHIFT)) begin
  // Set status register
  if (instruction_word[3][`REG_I] == `REG_SR & & `WB_INSTRUCTION(3)) begin
    // data forward from the status register
    temp_sr <= {temp_wb[31:1], shifter_carry_out};
    end
  else begin
    // take the current status register
    temp_sr <= {reg_file[`REG_SR][31:1], shifter_carry_out};
    end
  end
else begin
  // dont change the status register
  temp_sr <= reg_file[`REG_SR];
end
end

`PUSH_IC: begin
  temp_sp <= alu_result; // incremented Stack Pointer
  data_stack_push <= 1'b0;
end

`POP_IC: begin
  // data_stack_pop <= 1'b1;
  // data_stack_pop <= 1'b0;
  temp_sp <= alu_result; // decremented Stack Pointer
  temp_wb <= data_stack_out;
end

`LD_IC: begin
  `LOAD_MMIO(temp_wb,31:0,)
end

`ST_IC: begin
  if (dm_address < `MMIO_START_ADDR) begin
    // enable write if not mmio
    dm_wren <= 1'b1;
  end
  else begin
    // write to mmio
    dm_wren <= 1'b0;
  case (dm_address)
  `MMIO_GPIO_OUT: begin
    gpio_out <= dm_data;
  end
default: begin
    end

endcase // dm_address
end

\`JMP_IC: begin
// Do nothing?
end

\`CALL_IC: begin
// push the status register onto the stack
call_stack_push <= 1'b1;
call_stack_data <= reg_file['REG_SR];
end

\`RET_IC: begin
// pop the program counter
call_stack_pop <= 1'b1;
temp_address <= call_stack_out[15:0];
end

default: begin
end
endcase // opcode[2]

instruction_word[3] <= instruction_word[2];
end // if (stall[2] == 1'b0)

// Machine cycle 1
// operand fetch
if (stall[1] == 1'b0) begin
    case (opcode[1])
        \`ADD_IC, \`SUB_IC, \`CMP_IC, \`NOT_IC, \`AND_IC, \`BIC_IC, \`OR_IC, \`XOR_IC, \`MUL_IC,
        \`DIV_IC: begin
            // set alu_a
                 \`WB_INSTRUCTION(2) && !stall[2]) begin
                // data forward from mc2
                if (\`ALU_INSTRUCTION(2)) begin
                    // data forward from alu output
                    alu_a <= alu_result;
                end
            end
        endcase
        \`POP_IC: begin
            alu_a <= data_stack_out;
        end
else if (opcode[2] == `LD_IC) begin
    `LOAD_MMIO(alu_a,31:0,)
end
else if (opcode[2] == `RS_IC) begin
    alu_a <= shifter_result;
end
// TODO: data forward from other wb sources in mc2
else begin
    // no data forwarding
    alu_a <= reg_file[instruction_word[1][`REG_J]];
end
end

else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(2) && !stall[2]) begin
    // data forward from the increment/decrement of the stack pointer
    alu_a <= alu_result;
end
else if ((instruction_word[1][`REG_J] == instruction_word[3][`REG_I]) &&
    `WB_INSTRUCTION(3) && !stall[3]) begin
    // data forward from mc3
    alu_a <= temp_wb;
    // TODO: data forward from other wb sources in mc3
end
else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(3) &&
    !stall[3]) begin
    // data forward from the increment/decrement of the stack pointer
    alu_a <= temp_sp;
end
else begin
    // no data forwarding
    alu_a <= reg_file[instruction_word[1][`REG_J]];
end

// set alu_b
if (instruction_word[1][`ALU_CONTROL] == 1'b1) begin
    // constant operand
    alu_b <= {{16{instruction_word[1][`ALU_CONSTANT_MSB]}},
               instruction_word[1][`ALU_CONSTANT]}; // sign extend constant
end
else if ((instruction_word[1][`REG_K] == instruction_word[2][`REG_I]) &&
    `WB_INSTRUCTION(2) && !stall[2]) begin
    // data forward from mc2
    if (`ALU_INSTRUCTION(2)) begin
        alu_b <= alu_result;
    end
else if (opcode[2] == `POP_IC) begin
    alu_b <= data_stack_out;
end
else if (opcode[2] == `LD_IC) begin
    `LOAD_MMIO(alu_b,31:0),
end
else if (opcode[2] == `RS_IC) begin
    alu_b <= shifter_result;
end
// TODO: data forward from other wb sources in mc2
else begin
    // no data forwarding
    alu_b <= reg_file[instruction_word[1][`REG_K];
end
else if (instruction_word[1][`REG_K] == `REG_SP && `STACK_INSTRUCTION(2) && !stall[2]) begin
    // data forward from the increment/decrement of the stack pointer
    alu_b <= alu_result;
end
else if ((instruction_word[1][`REG_K] == instruction_word[3][`REG_I]) &&
    `WB_INSTRUCTION(3) && !stall[3]) begin
    // data forward from mc3
    alu_b <= temp_wb;
    // TODO: data forward from other wb sources in mc3
end
else if (instruction_word[1][`REG_K] == `REG_SP && `STACK_INSTRUCTION(3) &&
    !stall[3]) begin
    // data forward from the increment/decrement of the stack pointer
    alu_b <= temp_sp;
end
else begin
    // no data forwarding
    alu_b <= reg_file[instruction_word[1][`REG_K];
end

`CPY_IC: begin
    // set source alu_a
    if (instruction_word[1][`DT_CONTROL] == 1'b1) begin
        // copy from constant
        alu_a <= {{16[instruction_word[1][`DT_CONSTANT_MSB]}},
            instruction_word[1][`DT_CONSTANT]; // sign extend constant
    end
else if ((instruction_word[1][`REG_J] == instruction_word[2][`REG_I]) &&
            `WE_INSTRUCTION(2) && !stall[2]) begin
    // data forward from mc2
    if (`ALU_INSTRUCTION(2)) begin
        alu_a <= alu_result;
    end
else if (opcode[2] == `POP_IC) begin
    alu_a <= data_stack_out;
II.1 CJG RISC CPU RTL

622
623 else if (opcode[2] == `LD_IC) begin
624    LOAD_MMIO(alu_a,31:0,)
625 end
626
627 else if (opcode[2] == `RS_IC) begin
628    alu_a <= shifter_result;
629 end
630
631 // TODO: data forward from other wb sources in mc2
632 else begin
633    // no data forwarding
634    alu_a <= reg_file[instruction_word[1][`REG_J]];
635 end
636
637 else if (instruction_word[1][`REG_J] == `REG_SP &&
638          !stall[2]) begin
639    // data forward from the increment/decrement of the stack pointer
640    alu_a <= alu_result;
641 end
642
643 else if ((instruction_word[1][`REG_J] == instruction_word[3][`REG_I]) &&
644          `WB_INSTRUCTION(3) && !stall[3]) begin
645    // data forward from mc3
646    alu_a <= temp_wb;
647    // TODO: data forward from other wb sources in mc3
648 end
649
650 else if (instruction_word[1][`REG_J] == `REG_SP &&
651          `STACK_INSTRUCTION(3) &&
652          !stall[3]) begin
653    // data forward from the increment/decrement of the stack pointer
654    alu_a <= temp_sp;
655 end
656
657 else begin
658    // no data forwarding
659    alu_a <= reg_file[instruction_word[1][`REG_J]];
660 end
661
662 // alu_b unused for cpy so just keep it the same
663 alu_b <= alu_b;
664 end
665
666 `RS_IC: begin
667    // set the opcode
668    shifter_opcode <= instruction_word[1][`RS_OPCODE];
669
670    // set the operand
671 if ((instruction_word[1][`REG_J] == instruction_word[2][`REG_I]) &&
672      `WB_INSTRUCTION(2) && !stall[2]) begin
673      // data forward from mc2
674      if (`ALU_INSTRUCTION(2)) begin
675          shifter_operand <= alu_result;
676      end
677  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) &&
683               !stall[2]) begin
684       // data forward from the increment/decrement of the stack pointer
685       shifter_operand <= alu_result;
686    end
687    else if ((instruction_word[1][`REG_J] == instruction_word[3][`REG_I]) &&
688              `WB_INSTRUCTION(3) && !stall[3]) begin
689       // data forward from mc3
690       shifter_operand <= temp_wb;
691       // TODO: data forward from other wb sources in mc3
692    end
693    else if (instruction_word[1][`REG_J] == `REG_SP && `STACK_INSTRUCTION(3) &&
694               !stall[3]) begin
695       // data forward from the increment/decrement of the stack pointer
696       shifter_operand <= temp_sp;
697    end
698    else begin
699       // no data forwarding
700       shifter_operand <= reg_file[instruction_word[1][`REG_J]];
701    end
702    end
703    // set the modifier
704    if (instruction_word[1][`RS_CONTROL] == 1'b1) begin
705       // copy from constant
706       shifter_modifier <= instruction_word[1][`RS_CONSTANT];
707    end
708    else if ((instruction_word[1][`REG_K] == instruction_word[2][`REG_I]) &&
709               `WE_INSTRUCTION(2) && !stall[2]) begin
710       // data forward from mc2
711       if (`ALU_INSTRUCTION(2)) begin
712          shifter_modifier <= alu_result[5:0];
713       end
714    else if (opcode[2] == `POP_IC) begin
715       shifter_modifier <= data_stack_out[5:0];
II.1 CJG RISC CPU RTL

```
end
else if (opcode[2] == `LD_IC) begin
  `LOAD_MMIO(shifter_modifer,5:0,)
end
else if (opcode[2] == `RS_IC) begin
  shifter_modifier <= shifter_result[5:0];
end
// TODO: data forward from other wb sources in mc2
else begin
  // no data forwarding
  shifter_modifier <= reg_file[instruction_word[1][`REG_K]][5:0];
end
else if (instruction_word[1][`REG_K] == `REG_SP && `STACK_INSTRUCTION(2) &&
  `!stall[2]) begin
  // data forward from the increment/decrement of the stack pointer
  shifter_modifier <= alu_result[5:0];
end
else if ((instruction_word[1][`REG_K] == instruction_word[3][`REG_I]) &&
  `!WB_INSTRUCTION(3) && `!stall[3]) begin
  // data forward from mc3
  shifter_modifier <= temp_wb[5:0];
  // TODO: data forward from other wb sources in mc3
end
else if (instruction_word[1][`REG_K] == `REG_SP && `STACK_INSTRUCTION(3) &&
  `!stall[3]) begin
  // data forward from the increment/decrement of the stack pointer
  shifter_modifier <= temp_sp[5:0];
end
else begin
  // no data forwarding
  shifter_modifier <= reg_file[instruction_word[1][`REG_K]][5:0];
end

// set the carry in if rotating through carry
if ((instruction_word[1][`RS_OPCODE] == `RRC_SHIFT) ||
  (instruction_word[1][`RS_OPCODE] == `RLC_SHIFT)) begin
  if ((instruction_word[2][`REG_I] == `REG_SR) && `!WB_INSTRUCTION(2) &&
    `!stall[2]) begin // if mc2 is writing to the REG_SR
    // data forward from mc2
    if (`ALU_INSTRUCTION(2)) begin
      shifter_carry_in <= alu_result[`SR_C];
    end
  end if (opcode[2] == `POP_IC) begin
    shifter_carry_in <= data_stack_out[`SR_C];
  end
else if (opcode[2] == `LD_IC) begin
  `LOAD_MMIO(shifter_carry_in,`SR_C),
end
```
else if (opcode[2] == `RS_IC) begin
    shifter_carry_in <= shifter_result[`SR_C];
end

// TODO: data forward from other wb sources in mc2
else begin
    // no data forwarding
    shifter_carry_in <= reg_file[`REG_SR][`SR_C];
end
end

else if ((instruction_word[3][`REG_I] == `REG_SR) && `WB_INSTRUCTION(3) && 
          !stall[3]) begin // if mc3 is writing to the REG_SR
    // data forward from mc3
    shifter_carry_in <= temp_wb[`SR_C];
    // TODO: data forward from other wb sources in mc3
end

else if (`ALU_INSTRUCTION(2) && !stall[2]) begin // if the mc2 ALU
    // instruction will change the REG_SR
    // data forward from the alu output
    shifter_carry_in <= alu_c;
end

else if (opcode[2] == `RS_IC && !stall[2]) begin // if the mc2 shift
    // instruction will change the REG_SR
    shifter_carry_in <= shifter_carry_out;
end

else if (`ALU_INSTRUCTION(3) || opcode[3] == `RS_IC && !stall[3]) begin //
    // if the mc3 instruction will change the REG_SR
    // data forward from the temp status register
    shifter_carry_in <= temp_sr[`SR_C];
end

else begin
    // no data forwarding
    shifter_carry_in <= reg_file[`REG_SR][`SR_C];
end
end

else begin
    shifter_carry_in <= reg_file[`REG_SR][`SR_C];
end
end

end // `RS_IC

`PUSH_IC: begin
    // data forwarding for the data input
    if (instruction_word[1][`DT_CONTROL] == 1'b1) begin
        // push from constant
        data_stack_data <= {{16{instruction_word[1][`DT_CONSTANT_MSB]}},
                               instruction_word[1][`DT_CONSTANT]};
    end
    else if ((instruction_word[1][`REG_I] == instruction_word[2][`REG_I]) &&
             `WB_INSTRUCTION(2) && !stall[2]) begin
if (`ALU_INSTRUCTION(2)) begin
    data_stack_data <= alu_result;
end
else if (opcode[2] == `POP_IC) begin
    data_stack_data <= data_stack_out;
end
else if (opcode[2] == `LD_IC) begin
    LOAD_MMIO(data_stack_data,31:0);
end
else if (opcode[2] == `RS_IC) begin
    data_stack_data <= shifter_result;
end
// TODO: data forward from other wb sources in mc2
else begin
    // no data forwarding
    data_stack_data <= reg_file[instruction_word[1][`REG_J]];
end
else if (`REG_J == `REG_SP && `STACK_INSTRUCTION(2) && !stall[2]) begin
    // data forward from the increment/decrement of the stack pointer
    data_stack_data <= alu_result;
end
else if ((instruction_word[1][`REG_J] == instruction_word[3][`REG_I]) &&
    `WB_INSTRUCTION(3) && !stall[3]) begin
    // data forward from mc3
    data_stack_data <= temp_wb;
    // TODO: data forward from other wb sources in mc3
end
else if (`REG_J == `REG_SP && `STACK_INSTRUCTION(3) &&
    !stall[3]) begin
    // data forward from the increment/decrement of the stack pointer
    data_stack_data <= temp_sp;
end
else begin
    // no data forwarding
    data_stack_data <= reg_file[instruction_word[1][`REG_J]];
end

// data forward stack pointer
// set alu_a to increment stack pointer
if ((`REG_SP == instruction_word[2][`REG_I]) && `WB_INSTRUCTION(2) &&
    !stall[2]) begin
    // data forward from mc2
    if (`ALU_INSTRUCTION(2)) begin
        // data forward from alu output
        alu_a <= alu_result;
        data_stack_addr <= alu_result[5:0];
else if (opcode[2] == `POP_IC) begin
  alu_a <= data_stack_out;
  data_stack_addr <= data_stack_out[5:0];
end
else if (opcode[2] == `LD_IC) begin
  `LOAD_MMIO(alu_a,31:0,)
  `LOAD_MMIO(data_stack_addr,5:0,)
end
else if (opcode[2] == `RS_IC) begin
  alu_a <= shifter_result;
  data_stack_addr <= shifter_result[5:0];
end
// TODO: data forward from other wb sources in mc2
else begin
  // no data forwarding
  alu_a <= reg_file[`REG_SP];
  data_stack_addr <= reg_file[`REG_SP][5:0];
end
  begin
  // data forward from the output of the increment
  alu_a <= alu_result;
  data_stack_addr <= alu_result[5:0];
end
else if (`REG_SP == instruction_word[3][`REG_I]) && `WB_INSTRUCTION(3) &&
  !stall[3]) begin
  // data forward from mc3
  alu_a <= temp_wb;
  data_stack_addr <= temp_wb[5:0];
  // TODO: data forward from other wb sources in mc3
end
  begin
  // data forward from the output of the increment
  alu_a <= temp_sp;
  data_stack_addr <= temp_sp[5:0];
end
else begin
  // no data forwarding
  alu_a <= reg_file[`REG_SP];
  data_stack_addr <= reg_file[`REG_SP][5:0];
end

alu_b <= 32'h00000001;
data_stack_push <= 1'b1;
end
`POP_IC: begin
  // data forward stack pointer
  // set alu_a to decrement stack pointer
  if (`REG_SP == instruction_word[2][`REG_I] && `WB_INSTRUCTION(2) && !stall[2]) begin
    // data forward from mc2
    if (`ALU_INSTRUCTION(2)) begin
      // data forward from alu output
      alu_a <= alu_result;
      data_stack_addr <= alu_result[5:0] - 1'b1;
    end
    else if (opcode[2] == `POP_IC) begin
      alu_a <= data_stack_out;
      data_stack_addr <= data_stack_out[5:0] - 1'b1;
    end
    else if (opcode[2] == `LD_IC) begin
      `LOAD_MMIO(alu_a,31:0,)
      // data_stack_addr <= dm_out[5:0] - 1'b1;
      `LOAD_MMIO(`/dest=*/data_stack_addr,/*bits=*/5:0,/*expr=*/-1'b1)
    end
    else if (opcode[2] == `RS_IC) begin
      alu_a <= shifter_result;
      data_stack_addr <= shifter_result[5:0] - 1'b1;
    end
    // TODO: data forward from other wb sources in mc2
    else begin
      // no data forwarding
      alu_a <= reg_file[`REG_SP];
      data_stack_addr <= reg_file[`REG_SP][5:0] - 1'b1;
    end
  end
  // data forward from the output of the increment
  alu_a <= alu_result;
  data_stack_addr <= alu_result[5:0] - 1'b1;
end
else if (`REG_SP == instruction_word[3][`REG_I] && `WB_INSTRUCTION(3) && !stall[3]) begin
  // data forward from mc3
  alu_a <= temp_wb;
  data_stack_addr <= temp_wb[5:0] - 1'b1;
  // TODO: data forward from other wb sources in mc3
end
  // data forward from the output of the decrement
  alu_a <= temp_sp;
```verilog
data_stack_addr <= temp_sp[5:0] - 1'b1;
end
else begin
  // no data forwarding
  alu_a <= reg_file['REG_SP];
  data_stack_addr <= reg_file['REG_SP][5:0] - 1'b1;
end

alu_b <= 32'h00000001;
end

'LD_IC, 'ST_IC: begin
  // Set the data memory address
  if (instruction_word[1]['REG_J] != 5'b0 && instruction_word[1]['DT_CONTROL]
  ~== 1'b0) begin
    // Indexed
    if ((instruction_word[1]['REG_J] == instruction_word[2]['REG_I']) &&
    ~'WB_INSTRUCTION(2) && !stall[2]) begin
      // data forward from mc2
      if ('alu_instruction(2)) begin
        dm_address <= alu_result + instruction_word[1]['DT_CONSTANT];
      end
      else if (opcode[2] == 'POP_IC) begin
        dm_address <= data_stack_out + instruction_word[1]['DT_CONSTANT];
      end
      else if (opcode[2] == 'LD_IC) begin
        LOAD_MMIO(/*dest=*/dm_address,/*bits=*/31:0,/*expr=*/+instruction_word[1]['DT_CONSTANT]);
      end
    end
    else if (opcode[2] == 'RS_IC) begin
      dm_address <= shifter_result + instruction_word[1]['DT_CONSTANT];
    end
    // TODO: data forward from other wb sources in mc2
    else begin
      // No data forwarding
      dm_address <= reg_file[instruction_word[1]['REG_J]] +
      ~ instruction_word[1]['DT_CONSTANT];
    end
  end
else if (instruction_word[1]['REG_J] == 'REG_SP && 'STACK_INSTRUCTION(2) &&
~'stall[2]) begin
  // data forward from the increment/decrement of the stack pointer
  dm_address <= alu_result + instruction_word[1]['DT_CONSTANT];
end
else if ((instruction_word[1]['REG_J] == instruction_word[3]['REG_I]) &&
~'WB_INSTRUCTION(3) && !stall[3]) begin
  // data forward from mc3
  dm_address <= temp_wb + instruction_word[1]['DT_CONSTANT];
  // TODO: data forward from other wb sources in mc3
end
```
 else if (instruction_word[1]['REG_J'] == 'REG_SP && STACK_INSTRUCTION(3) && !stall[3]) begin
  // data forward from the increment/decrement of the stack pointer
  dm_address <= temp_sp + instruction_word[1]['DT_CONSTANT'];
 end
 else begin
  // No data forwarding
  dm_address <= reg_file[instruction_word[1]['REG_J']] +
              instruction_word[1]['DT_CONSTANT'];
 end
 end

 else if (instruction_word[1]['REG_J'] != 5'b0 &&
          instruction_word[1]['DT_CONTROL'] == 1'b1) begin
  // Register Direct
  if ((instruction_word[1]['REG_J'] == instruction_word[2]['REG_I']) &&
       WB_INSTRUCTION(2) && !stall[2]) begin
    // data forward from mc2
    if (ALU_INSTRUCTION(2)) begin
      dm_address <= alu_result;
    end
    else if (opcode[2] == 'POP_IC) begin
      dm_address <= data_stack_out;
    end
    else if (opcode[2] == 'LD_IC) begin
      LOAD_MMIO(dm_address,31:0,)
    end
    else if (opcode[2] == 'RS_IC) begin
      dm_address <= shifter_result;
    end
    // TODO: data forward from other wb sources in mc2
    end
  else begin
    // No data forwarding
    dm_address <= reg_file[instruction_word[1]['REG_J']];
  end
 end
 end

 else if (instruction_word[1]['REG_J'] == 'REG_SP && STACK_INSTRUCTION(2) &&
          !stall[2]) begin
  // data forward from the increment/decrement of the stack pointer
  dm_address <= alu_result;
 end
 else if ((instruction_word[1]['REG_J'] == instruction_word[3]['REG_I']) &&
          WB_INSTRUCTION(3) && !stall[3]) begin
  // data forward from mc3
  dm_address <= temp_wb;
  // TODO: data forward from other wb sources in mc3
 end
 else if (instruction_word[1]['REG_J'] == 'REG_SP && STACK_INSTRUCTION(3) &&
          !stall[3]) begin

II.1 CJG RISC CPU RTL

1020 // data forward from the increment/decrement of the stack pointer
dm_address <= temp_sp;
end
else begin
  // No data forwarding
  dm_address <= reg_file[instruction_word[1][`REG_J]];
end
end

else if (instruction_word[1][`REG_J] == 5'b0 &&
  instruction_word[1][`DT_CONTROL] == 1'b0) begin
  // PC Relative
  dm_address <= instruction_addr[1] + instruction_word[1][`DT_CONSTANT];
end
else begin
  // Absolute
  dm_address <= instruction_word[1][`DT_CONSTANT];
end

// Set the data input
if (opcode[1] == `ST_IC) begin

  // set the data value
  if ((instruction_word[1][`REG_I] == instruction_word[2][`REG_I]) &&
    `WB_INSTRUCTION(2) && !stall[2]) begin
    // data forward from mc2
    if (`ALU_INSTRUCTION(2)) begin
      dm_data <= alu_result;
    end
    else if (opcode[2] == `POP_IC) begin
      dm_data <= data_stack_out;
    end
    else if (opcode[2] == `LD_IC) begin
      LOAD_MMIO(dm_data,31:0,)
    end
    else if (opcode[2] == `RS_IC) begin
      dm_data <= shifter_result;
    end
    // TODO: data forward from other wb sources in mc2
    else begin
      // No data forwarding
      dm_data <= reg_file[instruction_word[1][`REG_I]];
    end
  end
else if (instruction_word[1][`REG_J] == `REG_SP &&
  `STACK_INSTRUCTION(2) &&
  !stall[2]) begin
  // data forward from the increment/decrement of the stack pointer
  dm_data <= alu_result;
end
else if ((instruction_word[1][`REG_I] == instruction_word[3][`REG_I]) &&
    `WB_INSTRUCTION(3) && !stall[3]) begin
    // data forward from mc3
    dm_data <= temp_wb;
    // TODO: data forward from other wb sources in mc3
end
else if (instruction_word[1][`REG_I] == `REG_SP &&
    `STACK_INSTRUCTION(3) && !stall[3]) begin
    // data forward from the increment/decrement of the stack pointer
    dm_data <= temp_sp;
end
else begin
    // No data forwarding
    dm_data <= reg_file[instruction_word[1][`REG_I]];
end
end

`JMP_IC: begin
    // Set the temp program counter
    if (instruction_word[1][`REG_I] != 5'b0 && instruction_word[1][`JMP_CONTROL] == 1'b0) begin
        // Indexed
        if ((instruction_word[1][`REG_I] == instruction_word[2][`REG_I]) &&
            `WB_INSTRUCTION(2) && !stall[2]) begin
            // data forward from mc2
            temp_address <= alu_result + instruction_word[1][`JMP_ADDR];
        end
        else if (opcode[2] == `POP_IC) begin
            temp_address <= data_stack_out + instruction_word[1][`JMP_ADDR];
        end
        else if (opcode[2] == `LD_IC) begin
            --> `LOAD_MMI0(*dest=+/temp_address,*bits=+/31:0,*expr=+/instruction_word[1][`JMP_ADDR],*
        end
else if (opcode[2] == `RS_IC) begin
            temp_address <= shifter_result + instruction_word[1][`JMP_ADDR];
        end
    end
    // TODO: data forward from other wb sources in mc2
    else begin
        // No data forwarding
        temp_address <= reg_file[instruction_word[1][`REG_I]] +
            instruction_word[1][`JMP_ADDR];
    end
else if (instruction_word[1][`REG_I] == `REG_SP && `STACK_INSTRUCTION(2) &&
    !stall[2]) begin

II.1 CJG RISC CPU RTL

```verbatim
// data forward from the increment/decrement of the stack pointer
temp_address <= alu_result + instruction_word[1][`JMP_ADDR];
end
else if ((instruction_word[1][`REG_I] == instruction_word[3][`REG_I]) &&
          `WB_INSTRUCTION(3) && !stall[3]) begin
  // data forward from mc3
  temp_address <= temp_wb + instruction_word[1][`JMP_ADDR];
  // TODO: data forward from other wb sources in mc3
end
else if (instruction_word[1][`REG_I] == `REG_SP &&
          `STACK_INSTRUCTION(3) &&
          !stall[3]) begin
  // data forward from the increment/decrement of the stack pointer
  temp_address <= temp_sp + instruction_word[1][`JMP_ADDR];
end
else
  // No data forwarding
  temp_address <= reg_file[instruction_word[1][`REG_I]] +
                  instruction_word[1][`JMP_ADDR];
end
else if (instruction_word[1][`REG_I] != 5'b0 &&
         instruction_word[1][`JMP_CONTROL] == 1'b1) begin
  // Register Direct
  if (`ALU_INSTRUCTION(2)) begin
    temp_address <= alu_result;
  end
  else if (opcode[2] == `POP_IC) begin
    temp_address <= data_stack_out;
  end
  else if (opcode[2] == `LD_IC) begin
    `LOAD_MMIO(temp_address,31:0,)
  end
  else if (opcode[2] == `RS_IC) begin
    temp_address <= shifter_result;
  end
  // TODO: data forward from other wb sources in mc2
  else begin
    // No data forwarding
    temp_address <= reg_file[instruction_word[1][`REG_I]];
  end
end
else if (instruction_word[1][`REG_I] == `REG_SP &&
         `STACK_INSTRUCTION(2) &&
         !stall[2]) begin
  // data forward from the increment/decrement of the stack pointer
  temp_address <= alu_result;
end
```
else if ((instruction_word[1][`REG_I] == instruction_word[3][`REG_I]) &&
          `WB_INSTRUCTION(3) && !stall[3]) begin
  // data forward from mc3
  temp_address <= temp_wb;
  // TODO: data forward from other wb sources in mc3
  end
else if (instruction_word[1][`REG_I] == `REG_SP &&
          `STACK_INSTRUCTION(3) &&
          !stall[3]) begin
  // data forward from the increment/decrement of the stack pointer
  temp_address <= temp_sp;
  end
else begin
  // No data forwarding
  temp_address <= reg_file[instruction_word[1][`REG_I]];
  end
end
else if (instruction_word[1][`REG_I] == 5'b0 &&
          instruction_word[1][`JMP_CONTROL] == 1'b0) begin
  // PC Relative
  temp_address <= instruction_addr[1] + instruction_word[1][`JMP_ADDR];
end
else begin
  // Absolute
  temp_address <= instruction_word[1][`JMP_ADDR];
end
end // JMP_IC
`CALL_IC: begin
  // Set address
  // Always absolute mode for call (for now)
  temp_address <= instruction_word[1][`JMP_ADDR];
  // push the program counter onto the stack for when we return
  call_stack_push <= 1'b1;
  call_stack_data <= reg_file[`REG_PC];
end
`RET_IC: begin
  // pop the status register
  call_stack_pop <= 1'b1;
  reg_file[`REG_SR] <= call_stack_out;
end
default: begin
  end
endcase // opcode[1]
// set the alu opcode
case (opcode[1])
ADD_IC, `PUSH_IC: begin
  alu_opcode <= `ADD_ALU;
end

SUB_IC, `CMP_IC, `POP_IC: begin
  alu_opcode <= `SUB_ALU;
end

NOT_IC: begin
  alu_opcode <= `NOT_ALU;
end

AND_IC: begin
  alu_opcode <= `AND_ALU;
end

BIC_IC: begin
  alu_opcode <= `BIC_ALU;
end

OR_IC: begin
  alu_opcode <= `OR_ALU;
end

XOR_IC: begin
  alu_opcode <= `XOR_ALU;
end

CPY_IC: begin
  alu_opcode <= `NOP_ALU;
end

MUL_IC: begin
  alu_opcode <= `MUL_ALU;
end

DIV_IC: begin
  alu_opcode <= `DIV_ALU;
end

default: begin
  alu_opcode <= alu_opcode;
end
endcase // opcode[1]

instruction_word[2] <= instruction_word[1];
instruction_addr[2] <= instruction_addr[1];
end // if (stall[1] == 1'b0)
// Machine cycle 0
// instruction fetch
if (stall[0] == 1'b0) begin
    reg_file['REG_PC] <= reg_file['REG_PC] + 3'h4;
    instruction_addr[1] <= reg_file['REG_PC'][13:0];
    instruction_word[1] <= pm_out;
end

// set stall cycles
if (opcode[0] == 'JMP_IC || opcode[0] == 'CALL_IC || opcode[0] == 'RET_IC))
    begin
        stall_cycles <= 3'h3;
        stall[0] <= 1'b1;
    end
end // if (stall[0] == 1'b0)
end // else begin
end // always @(posedge clk)

task reset_all; begin
    gpio_out <= 32'b0;
    dm_data <= 32'b0;
    dm_wren <= 1'b0;
    dm_address <= 14'b0;
    temp_address <= 16'b0;
    instruction_word[3] <= 32'b0;
    instruction_word[2] <= 32'b0;
    instruction_word[1] <= 32'b0;
    instruction_addr[3] <= 14'b0;
    instruction_addr[2] <= 14'b0;
    instruction_addr[1] <= 14'b0;
    stall_cycles <= 4'b0;
    stall[3] <= 1'b1;
    stall[2] <= 1'b1;
    stall[1] <= 1'b1;
    stall[0] <= 1'b1;
    data_stack_data <= 32'b0;
    data_stack_addr <= 6'b0;
    data_stack_push <= 1'b0;
    data_stack_pop <= 1'b0;
    call_stack_data <= 32'b0;
    call_stack_push <= 1'b0;
    call_stack_pop <= 1'b0;
end
II.1 CJG RISC CPU RTL

II.1.4 Clock Generator

```verilog
module cjg_clkgen (
    input reset,          // system reset
    input clk,            // system clock
    output clk_p1,        // phase 0
    output clk_p2,        // phase 1
    input scan_in0,
    input scan_en,
    input test_mode,
    output scan_out0
)
);

// Clock counter
reg[1:0] clk_cnt;

// Signals for generating the clocks
```
II.1 CJG RISC CPU RTL

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,
// dft
input scan_in0,
input scan_en,
input test_mode,
output scan_out0
);

reg[WIDTH:0] internal_result;
wire overflow, underflow;

assign result = internal_result[WIDTH-1:0];
assign c = internal_result[WIDTH];
assign n = internal_result[WIDTH-1];
assign z = (internal_result == 0 ? 1'b1 : 1'b0);

assign overflow = (internal_result[WIDTH:WIDTH-1] == 2'b01 ? 1'b1 : 1'b0);
assign underflow = (internal_result[WIDTH:WIDTH-1] == 2'b10 ? 1'b1 : 1'b0);

assign v = overflow | underflow;

always @(*) begin
    internal_result = 0;
    case (opcode)
        `ADD_ALU: begin
            // signed addition
            internal_result = {a[WIDTH-1], a} + {b[WIDTH-1], b};
        end
        `SUB_ALU: begin
            // signed subtraction
            internal_result = ({a[WIDTH-1], a} + ~{b[WIDTH-1], b}) + 1'b1;
        end
        `AND_ALU: begin
            // logical AND
            internal_result = a & b;
        end
        `BIC_ALU: begin
            // logical bit clear
            internal_result = a & (~b);
        end
        `OR_ALU: begin
            // logical OR
            internal_result = a | b;
        end
    endcase
end
II.1 CJG RISC CPU RTL

II.1.6 Shifter

cjg_shifter.v

// Dynamic width combinational logic Shifter

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

// Whether or not to use the modifier shift logic
`define USE_MODIFIER

```verilog
```

`NOT_ALU: begin
    // logical invert
    internal_result = ~a;
end

`XOR_ALU: begin
    // logical XOR
    internal_result = a ^ b;
end

`NOP_ALU: begin
    // no operation
    // sign extend a to prevent wrongful overflow flag by accident
    internal_result = {a[WIDTH-1], a};
end

`MUL_ALU: begin
    // signed multiplication
    internal_result = a * b;
end

`DIV_ALU: begin
    // unsigned division
    internal_result = a / b;
end

default: begin
    internal_result = internal_result;
end // default

default: begin
    internal_result = internal_result;
end // default
endcase // opcode
```
module cgj_shifter #(parameter WIDTH = 32, MOD_WIDTH = 6) (
  input reset,
  input clk,
  input signed [WIDTH-1:0] operand,
  input carry_in,
  input [2:0] opcode,
  `ifdef USE_MODIFIER
    input [MOD_WIDTH-1:0] modifier,
  `endif

  output reg [WIDTH-1:0] result,
  output reg carry_out,
  // dft
  input scan_in0,
  input scan_en,
  input test_mode,
  output scan_out0);

`ifdef USE_MODIFIER
  wire[WIDTH+WIDTH-1:0] temp_rotate_right = {operand, operand} >>
    modifier[MOD_WIDTH-2:0];
  wire[WIDTH+WIDTH-1:0] temp_rotate_left = {operand, operand} << modifier[MOD_WIDTH-2:0];
  wire[WIDTH+WIDTH+1:0] temp_rotate_right_c = {carry_in, operand, carry_in, operand} >>
    modifier;
  wire[WIDTH+WIDTH+1:0] temp_rotate_left_c = {carry_in, operand, carry_in, operand} <<
    modifier;
`endif

always @(*) begin

  case (opcode)

    `SRL_SHIFT: begin
      `ifndef USE_MODIFIER
        // shift right logical by 1
        result <= {1'b0, operand[WIDTH-1:1]};
      `else
        // shift right by modifier
        result <= operand >> modifier[MOD_WIDTH-2:0];
      `endif
      carry_out <= carry_in;
    end

    `SLL_SHIFT: begin

  endcase
end
```verilog
`ifndef USE_MODIFIER
  // shift left logical by 1
  result <= {operand[WIDTH-2:0], 1'b0};
`else
  // shift left by modifier
  result <= operand << modifier[MOD_WIDTH-2:0];
`endif

carry_out <= carry_in;
end

'SRA_SHIFT: begin
`ifndef USE_MODIFIER
  // shift right arithmetic by 1
  result <= {operand[WIDTH-1], operand[WIDTH-1:1]};
`else
  // shift right arithmetic by modifier
  result <= operand >>> modifier[MOD_WIDTH-2:0];
`endif

carry_out <= carry_in;
end

'RTR_SHIFT: begin
`ifndef USE_MODIFIER
  // rotate right by 1
  result <= {operand[0], operand[WIDTH-1:1]};
`else
  // rotate right by modifier
  result <= temp_rotate_right[WIDTH-1:0];
`endif

carry_out <= carry_in;
end

'RTL_SHIFT: begin
`ifndef USE_MODIFIER
  // rotate left
  result <= {operand[WIDTH-2:0], operand[WIDTH-1]};
`else
  // rotate left by modifier
  result <= temp_rotate_left[WIDTH+WIDTH-1:WIDTH];
`endif

carry_out <= carry_in;
end

'RRC_SHIFT: begin
`ifndef USE_MODIFIER
  // rotate right through carry
  result <= {carry_in, operand[WIDTH-1:1]};
  carry_out <= operand[0];
`else
```
II.1 CJG RISC CPU RTL

II.1.7 Data Stack

cjg_mem_stack.v

module cjg_mem_stack #(parameter WIDTH = 32, DEPTH = 32, ADDRW = 5) (}

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

output reg [WIDTH-1:0] q,

// dft
input scan_in0,
II.1 CJG RISC CPU RTL

14   input scan_en,
15   input test_mode,
16   output scan_out0
17);
18
19   reg [WIDTH-1:0] stack [DEPTH-1:0];
20   integer i;
21
22   always @(posedge clk or negedge reset) begin
23      if (~reset) begin
24         q <= {WIDTH{1'b0}};
25         for (i=0; i < DEPTH; i=i+1) begin
26             stack[i] <= {WIDTH{1'b0}};
27         end
28         end
29      else begin
30         if (push) begin
31             stack[addr] <= d;
32         end
33         else begin
34             stack[addr] <= stack[addr];
35         end
36     end
37
38     q <= stack[addr];
39     end
40     endmodule // cjg_mem_stack

II.1.8 Call Stack

---

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,
II.1 CJG RISC CPU RTL

II-40

II.1.9 Testbench

cjg_risc_test.v

`include "src/cjg_opcodes.vh"

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

//`define TEST_ALU
module test;

// tb stuff
integer i;

// system ports
reg clk, reset;
wire clk_p1, clk_p2;

// dft ports
wire scan_out0;
reg scan_in0, scan_en, test_mode;

always begin
  #0.5 clk = ~clk; // 1000 MHz clk
end

// program memory
reg [7:0] pm [0:65535]; // program memory
reg [31:0] pm_out; // program memory output data
wire [15:0] pm_address; // program memory address

// data memory
reg [7:0] dm [0:65535]; // data memory
reg [31:0] dm_out; // data memory output
wire [31:0] dm_data; // data memory input data
wire dm_wren; // data memory write enable
wire [15:0] dm_address; // data memory address

always @(posedge clk_p2) begin
  if (dm_wren == 1'b1) begin
    dm[dm_address+3] = dm_data[31:24];
    dm[dm_address+2] = dm_data[23:16];
    dm[dm_address+1] = dm_data[15:8];
    dm[dm_address] = dm_data[7:0];
  end
  pm_out = {pm[pm_address+3], pm[pm_address+2], pm[pm_address+1], pm[pm_address]};
  dm_out = {dm[dm_address+3], dm[dm_address+2], dm[dm_address+1], dm[dm_address]};
end

// inputs
reg [31:0] gpio_in; // button inputs
reg [3:0] ext_interrupt_bus; // external interrupts

// outputs
wire [31:0] gpio_out;

`ifdef TEST_ALU
reg [31:0] alu_a, alu_b;
reg [3:0] alu_opcode;
wire [31:0] alu_result;
wire alu_c, alu_n, alu_v, alu_z;
reg [31:0] tb_alu_result;
cjg_alu alu(
    .a(alu_a),
    .b(alu_b),
    .opcode(alu_opcode),
    .result(alu_result),
    .c(alu_c),
    .n(alu_n),
    .v(alu_v),
    .z(alu_z)
);
`endif
cjg_risc top(
    // system inputs
    .reset(reset),
    .clk(clk),
    .gpio_in(gpio_in),
    .ext_interrupt_bus(ext_interrupt_bus),

    // generated clock phases
    .clk_p1(clk_p1),
    .clk_p2(clk_p2),

    // system outputs
    .gpio_out(gpio_out),

    // program memory
    .pm_out(pm_out),
    .pm_address(pm_address),

    // data memory
    .dm_data(dm_data),
    .dm_out(dm_out),
    .dm_wren(dm_wren),
    .dm_address(dm_address),

    // dft
    .scan_in0(scan_in0),
    .scan_en(scan_en),
    .test_mode(test_mode),
```

initial begin
    #10  tb_alu_result = alu_result;
    $finish;
end

// RISC TEST

    // init memories
    $readmemh("mif/", MIF, ".mif", pm);
    $readmemh("mif/", MIF, ".dm", ".mif", dm);
    $display("Loaded %s", pm);
    assert_reset;
    repeat (3) begin
        @(posedge clk);
    end

    // come out of reset a little before the edge
    #0.25 deassert_reset;
    @(posedge clk_p1);

    gpio_in = 12;

    // run until program reaches end of memory
    while (!((pm_out == 1'hX) && (pm_out != 32'hFFFFFFFF) && (gpio_out != 32'hDEADBEEF)) begin
        @(posedge clk_p1);
```
$display("Trying to read from unknown program memory");

// run for a few more clock cycles to empty the pipeline
repeat (6) begin
@ (posedge clk);
end

$display("gpio_out = %x", gpio_out);

ifndef SDFSCAN
print_reg_file;
endif

ifndef SDFSCAN
task print_reg_file; begin
$display("Register Contents:");
for (i=0; i<32; i=i+1) begin
$display("R%0d = 0x%X", i, top.reg_file[i]);
end
$display({30{"-"}});
endtask // print_reg_file

task print_stack; begin
$display("Stack Contents:");
for (i=0; i<32; i=i+1) begin
$display("S%0d = 0x%X", i, top.data_stack.stack[i]);
end
$display({30{"-"}});
endtask // print_stack
'endif

task assert_reset; begin
// reset dft ports
scan_in0 = 1'b0;
scan_en = 1'b0;
test_mode = 1'b0;

// reset system inputs
clock = 1'b0;
reset = 1'b0;
gpio_in = 32'b0;
ext_interrupt_bus = 4'b0;
II.2 ELF to Memory

```python
#!usr/bin/env python
import argparse
import elffile
import os
import sys

def getData(section, wordLength):
    data = []
    buf = section.content
    tmp = 0
    for i in range(0, len(buf)):
        byte = ord(buf[i])  # transform the character to binary
        tmp |= byte << (8 * (i % wordLength))  # shift it into place in the word
        if i % wordLength == wordLength-1:  # if this is the last byte in the word
            data.append(tmp)
            tmp = 0
    return data

def main(args):
    if not os.path.isfile(args.elf):
        print "error: cannot find file: {}".format(args.elf)
        return 1
    else:
        with open(args.elf, 'rb') as f:
            ef = elffile.open(fileobj=f)
            section = None
            if args.section is None:
```

# if no section was provided in the arguments list all available
sections = [section.name for section in ef.sectionHeaders if 
                       section.name]
print "list of sections: {}".format(" ".join(sections))
return 0
else:
    sections = [section for section in ef.sectionHeaders if section.name == 
                       args.section][1]
    if len(sections) == 1:
        section = sections[0]
    else:
        section = None

if not section:
    print "error: could not find section with name: 
                      {}".format(args.section)
    return 0
elif elffile.SHT.bycode[section.type] !=
                       elffile.SHT.byname["SHT_PROGBITS"]:  
    print "error: section has invalid type: 
                       {}".format(elffile.SHT.bycode[section.type])
    return 0
elif len(section.content) % args.length != 0:
    print "error: {} data ({} bytes) does not align with a word length of 
                      {} bytes".format(section.name, len(section.content), args.length)
    return 0

# get the binary data from the section and align it to words
data = getData(section, args.length)

# write the data by word to a readmem formatted file
out = ""
out += "// Converted from the {} section in {}\n".format(section.name, 
                      args.elf)
out += "// $ {}\n".format(" ".join(sys.argv))
out += "\n"

counter = 0
for word in data:
    out += "{0:08X} {0:{pad}X}\n".format(counter, word, pad=args.length*2)
    counter += args.addresses

if args.output:
    # write the output to a file
    with open(args.output, "wb") as outputFile:
        outputFile.write(out)
else:
    # write the output to stdout
    sys.stdout.write(out)
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Extract a section from an ELF to
    readmem format")
    parser.add_argument("-s", "--section", required=False, metavar="section", type=str,
                       help="The name of the ELF section file to output")
    parser.add_argument("-o", "--output", required=False, metavar="output", type=str,
                       help="The path to the output readmem file (default: stdout")
    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)
    parser.add_argument("-a", "--addresses", required=False, metavar="address",
                       type=int, help="The number of addresses to increment per word", default=1)
    parser.add_argument("elf", metavar="elf-file", type=str, help="The input ELF file")
    args = parser.parse_args()
    main(args)