5-2020

The Design of a Custom 32-Bit RISC CPU and Port to GCC Compiler Backend

Danielle Megan Fischer
dmf6080@rit.edu

THE DESIGN OF A CUSTOM 32-BIT RISC CPU AND PORT TO GCC COMPILER BACKEND

by
DANIELLE MEGAN FISCHER

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, Senior 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
MAY 2020

To everyone who has supported me throughout my college career and will continue to in the future; specifically, my mom, sister, brother, and my closest friends. Thank you to everyone for their love and support during these years.

Abstract

This paper presents the design of a 32-bit RISC processor, which is then mapped to the backend of GCC so basic C code can be compiled successfully to the processor. There are many design decisions that go into the construction of a processor. The instruction set architecture gives away a lot of information regarding the individual instructions that the processor will have, the memory architecture, as well as how I/O peripherals will be handled. Additionally, the hardware implementation of the processor needs to be kept in mind when crating the design. Pipelining can often help with processor speed, while cache implementation can assist in memory speed. After designing the processor, GCC's backend needs to be analyzed to port it to function with the processors individual opcodes. Once GCC can compile its C code to an assembly language which is able to assemble into machine code that matches up with the opcodes the processor was created for, the machine code can be written into the processor's program memory and executed successfully. This paper also talks about different design decisions that are made during the process of creating a processor, as well as the general makeup of the GCC compilation process.

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.

Danielle Megan Fischer

May, 2020

Acknowledgements

I would like to first thank all of my professors at RIT, specifically my EE professors, for supporting me through my college career. First, I'd like to thank Mark Indovina, a mentor and my advisor for this paper, for always having more confidence in me than I did, and for always being ready with a spit of sarcasm. I would also like to thank Dr. Moon for instilling in me my initial interest in digital electronics, Dr. Patru for teaching me everything I know about processor design, Dr. Brown for pushing me to be the best student I can be (and teaching me Circuits 1, a very important fundamental, of course), Dr. Amuso for teaching me that engineering courses can judge you off more than just examinations, and Carlos for making the embedded final humble me and my GPA (and employing me as a TA). I would also like to thank Patti, an amazing BS/MS advisor who always seems to have more on her plate than she can fit, but always finding a way to prioritize the students, and Ken, the EE facilities manager, and my favorite guitar-playing, story-telling guy on floor. Thank you to everyone on the EE floor for an amazing 5 years, I'm so disappointed I didn't get to spend the last few weeks of my senior year with you all.

Of course, I couldn't have done a lot of what I have accomplished without the support of my family. Thank you so much to my mom, for being my best friend and #1 supporter since birth, to my sister, Cassie, for being my other best friend, my #2 supporter since her birth, and for being the coolest sister I've ever had, and to my brother, Billy, for always making me laugh when you call. I love you all so much, and I can't wait to see what your futures bring.

I would also like to thank pretty much everyone that every sat in the TA office for always brightening my day. Special thanks to my main group of EEs, DarDar, DrewBB, Ry Ry, Kev Kev, Andrew, Dania, Dakota, Connor, Dan, Danny, Tommy, John (my mans was always pullin through with the free food), and Adam. Without these guys, my everyday struggle would've been so much less enjoyable.

Last but not least, I would like to thank all my closest friends I've made (or kept) over my years here: Justine, Becca, Dania, Claire, Mandy, Ricky (whom made my college career a much deeper learning experience from having to explain everything to), Ajnur, Alessandro, and Rowan. A special thanks to my boyfriend, Darian, for always making me laugh, even at my lowest and most stressful times, and for being my favorite (and probably only possible) quarantine buddy during coronavirus. I am so fortunate to have such a large support group that makes my acknowledgments section so long. You guys are always there when I need it, and I couldn't be where I am today without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The functionality of processors makes them a tool that is used universally. Processors are seen in every piece of technology used on a day-to-day basis. Computers, phones, tablets, robots, cars, anything that "thinks" has a processor at its core, making the most basic of decisions. For every new processor that is designed, there is an assembly language to match. Having a different programming language to learn for every unique processor would be tedious and difficult to manage. Processors would be less ubiquitous. For this reason, it is common for compilers of commonly-used coding languages, like C, C++, Java, etc. to be ported to these processors, so higher level code can be compiled and run on unique processors. This process is explored in this paper. For this project, a 32-bit RISC pipelined processor with cache memory was designed and GCC (GNU C Compiler) was ported to the custom processor to run basic C code.

1.1 Organization

This paper will be organized into 6 chapters. This chapter, Chapter 1, is the introduction. Chapter 2 will talk about the background of processors and compilers, specifically RISC processors

and GNU Compiler Collection. Other processors and compilers will be explored to find their advantages and disadvantages. Next, Chapter 3 will talk about the design of the custom RISC processor used in this project. Chapter 4 will talk about the back end alterations made to GCC, and then Chapter 5 will follow with testing and results of the GCC porting to the custom RISC processor. Finally, Chapter 6 will talk about future work and conclusions of the project.

Chapter 2

Background on Processors and Compilers

This chapter discusses background of processors, specifically RISC and CISC, as well as decisions made during the design of processors. This chapter also talks about the background of C compilers, specifically GCC and its structure, with some comparison to the operation of LLVM.

2.1 Processor Design

There are many processor architectures to bear in mind when designing a processor. Many articles and studies have debated over which processor structure is the best, comparing speed to size on a chip to overall performance, like power and energy consumption, to rate these processor designs. One of the most common comparisons is done between the reduced instruction set computer (RISC) and the complex instruction set computer (CISC) architectures. This comparison will be talked about more in section 2.1.1. Even after deciding on a general instruction set architecture (ISA) to follow, there are many other decisions that go into the process of designing the processor, talked more about in section 2.1.2.

Hand-in-hand with processor design comes the compiler that allows code to be written to

and executed on the processor. GCC, which stands for "GNU Compiler Collection", supports over 30 different processor architectures and 7 programming languages, and is one of the most used compilers for processor execution [11]. Another rising compiler is LLVM, which stands for low-level virtual machine. These compilers will be compared in section 2.2.4. The general design flow of a compiler will be talked about in section 2.2.1. Finally, GCC will be discussed in further detail in section 2.2.2, since GCC is the compiler used in this project.

2.1.1 RISC vs CISC

The debate of whether RISC or CISC is a higher performer in the playing field of computer and processor architectures is one that was sparked around the time of the mid-80s. The general consensus for a while was that the RISC architecture was the superior performer, but this was during a time when the main constraints were chip size and processor design complexity [12]. Nowadays, architect engineers are more concerned with the energy and power consumption of a processor, which largely changes the debate. Arguably, when comparing RISC and CISC architectures based on today's more relevant constraints, it is irrelevant whether the processor is a RISC or a CISC. Rather, it is other components of the processor that aren't categorized under the RISC/CISC ISA that make the processor execute more efficiently, with respect to its energy and power consumption [12, 13].

2.1.1.1 CISC

The idea of a reduced instruction set computer (RISC) came around to specifically compete with the presently (in the early 80s) used complex instruction set computer (CISC). CISC was the architecture currently used on nearly all computers and computing machines at the time. Some of the more common ones were the Intel x86 and Motorola 68000 [14]. Some basic characteristics that make a processor have a CISC architecture are:

- Large instructions sets: One of the goals of a CISC architecture is to cover many programming scenarios through complicated instructions
- Complex instructions: Some instructions are more complex than the basic add and subtract, and might contain multiple basic instructions in one.
- Instructions that operate over different numbers of clock cycles: Since there are so many instructions, many of them have different structures and take a different number of clock cycles to complete, which can make pipelining difficult [12, 14].
- Multiple addressing modes: CISC processors typically have a number of ways to reference memory locations.
- Register limitations: CISC architectures typically only have 16 registers, which is not always enough to support all functionality going on at one time. Often, values have to transferred from registers into memory to make space so certain manipulations can be done. These values are then transferred back into the corresponding register when the manipulations are over. This can be a waste of time, as transferring data in and out of memory is not always a quick task [14].
- Multiple instruction formats: Some of the instructions will do their operations from register to register, some memory to register, and others memory to memory [14]. This can greatly affect the number of clock cycles it takes for the instruction to be executed.

To this day, CISC is largely used on most large-scale electronic devices with processors, such as laptops, desktop computers, and servers.

2.1.1.2 RISC

RISC was designed with the goal of simplifying the complicated instruction set known to the CISC architecture. Many of the CISC instructions were uneven and took too long to execute. RISC takes the burden of the complicated instructions off the architecture and places it in the compiler [15]. For a RISC processor, the compiler needs to be much heftier, being able to break down the same code the is executed by CISC processors into much more basic instructions that can execute on a RISC implementation. Studies on program behavior showed that 25% of instructions in CISC instruction sets make up 95% of the program execution time, meaning about 75% of the instruction set are hardly used or not used at all [14]. Studies such as these proved that a RISC architecture could be implemented realistically and would be quite useful in reducing chip size. Some characteristics of a RISC architecture are:

- Small instruction sets: Of course, a main goal of RISC was to reduce the instruction set from CISC. Usually RISC architectures only have about 20 instructions or less.
- Simple instructions: Instructions are stripped down to only the most basic operations, like add, subtract, shift, etc.
- Instructions all take the same number of clock cycles to execute: Instructions usually take 4-5 clock cycles, depending on the design, and all instructions take the same amount of time to execute, simplifying the pipelining process.
- Few addressing modes: Typically, RISC processors will only have direct addressing mode, register-direct addressing mode, and only maybe a couple others, such as PC-relative.
- Large number of registers: RISC processors usually have at least 32 general-purpose registers. Since the processor takes up less space on the chip, some of the extra space can be used for more registers [13].

Table 2.1: Characteristics of RISC vs. CISC

Architecture Characteristic	RISC	CISC
Instruction Set Size	small	large
Instruction Complexity	simple	complex
Instruction Execution Times	all the same	different
Number of Addressing Modes	few, 3-4	many
Number of Registers	at least 32	very few, generally only 16
Number of Instruction Formats	two: load-store, reg-reg	three: load-store, reg-reg, reg-mem, mem-mem

- One instruction format: RISC processors only have register-to-register instructions, with only the load and store instructions being capable of accessing memory.

Today, RISC processors are mostly used in smaller electronics, such as tablets, phones, and smart watches.

Table 2.1 displays the characteristics of RISC and CISC listed above in a way that makes them easy to compare.

The most basic RISC processor needs nothing more than an instruction execution unit, an arithmetic unit, memory, I/O peripherals, and a bus to connect all these units. A RISC processor can be extremely simple in structure, and still be able to run the same code as a CISC or more complex-structured processor [1]. Figure 2.1 represents a block diagram of the most simple RISC processor. When compared to Figure 3.8, it can be seen that the DMF RISC processor has a RISC architecture, but is not of the simplest design.

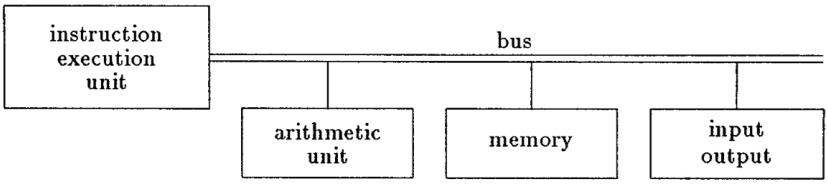

Figure 2.1: Basic RISC block diagram [1]

2.1.1.3 Comparison: Which is Better?

The question of which processor is better really depended on the time that question was asked. Back in the 80s and 90s, this question could be answered by which fit onto a smaller chip size and was still able to execute quickly. RISC programs were longer than CISC programs, but only by about 30% [14]. This is one of the few shortcomings of the RISC processor. Some of the more complex instructions in the CISC architecture took multiple instructions to replace the equivalent functionality in a RISC processor. Other than this, there were only good things when talking about switching to a RISC processor. Despite the slightly longer programs, however, studies have actually found that RISC execution is faster than CISC, largely because much of the run-time complexity is resolved at compile-time for the RISC processor [16].

So then, why are all processors not a RISC architecture? First, companies did not necessarily need to switch to a different processor to get a smaller chip size, until the phone and tablet age came about. Secondly, as technology advanced, the question was no longer of chip size, but rather the energy and power consumption. With these new variables in play, the processor architectures were re-compared, and found to be very similar. In fact, it was found that any performance gains were not due to characteristics that had to do with the RISC/CISC architecture, they were instead other design adjustments made to the processor architecture, such as cache memory, branch prediction, out-of-order execution, fetch prediction methods, along with other instruction organization methods [13, 17]. These added characteristics make the processor

"smarter", as they allow it to predict what will happen next based on previous execution. When these methods are executed in a way that they are correct a high percentage of the time, this saves much processing power and execution time. These methods will be talked more about in section 2.1.2.

2.1.2 Design Decisions

When designing a processor, more goes into the decision process than just what ISA should be used. Things like memory architecture, I/O peripheral access, cache memory, and pipeline structure are just some of the basic decisions that need to be made. These are talked more about in Chapter 3. Some of the more interesting decisions come in when optimizing the processor. Aspects like branch prediction and out-of-order execution make the complexity of the processor much greater, but can also largely increase the performance.

2.1.2.1 Branch Prediction

Branch prediction is a method used to guess what the outcome of a conditional jump or branch statement will be. Referring to Figure 3.9, it can be seen that different stages of different instructions are executed at the same time. So, when a conditional jump or branch statement is brought into the instruction fetch phase, it will be another few clock cycles (2 in the case of the DMF RISC processor) before the processor knows if the jump will occur or not, since the outcome of the conditional is decided in the execute stage [18]. To avoid the stalling of the pipeline that would normally have to take place until the conditional branch statement was in the execution phase, some processors implement branch predictors. When a branch predictor predicts correctly, some time is saved since the pipeline does not have to be stalled, and instructions continue executing as they would. If the prediction is incorrect, then the processor has to ignore the instructions that have been brought into the pipeline incorrectly, and carry on from where the next

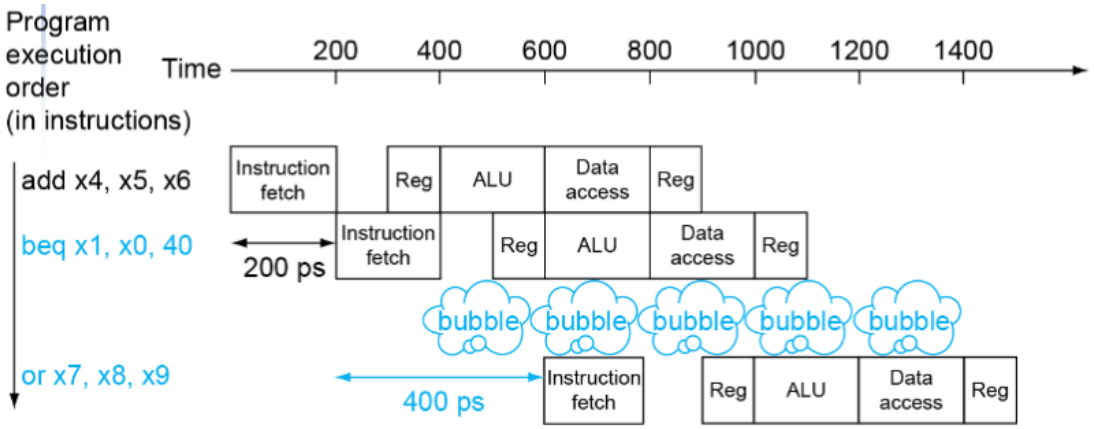

Figure 2.2: Pipeline with no branch prediction [2]

instruction will be. An incorrect branch prediction looks a lot like no branch prediction at all in terms of time it takes to execute in the pipeline. This can be seen by comparing Figures 2.2 and 2.3.

Often, a branch prediction can be as simple as one bit. If this bit it a 0, then it is predicting that the branch will be false. If the bit is a 1, it is predicting the branch will be true. Every time the bit predicts incorrectly, it changes to match what actually happened. In the case of loops, which is how most conditional branches are used, this can be a very effective method, as the branch predictor is usually only incorrect on the first and last iteration of the loop. Depending how many times the loop executes, having only two incorrect predictions can make for a very small percentage of wrong guesses. Since the worst case scenario is that upon a false prediction, the pipeline is delayed only as much as it would be without any branch prediction at all, having a branch predictor is only advantageous to the efficiency of the processor.

An interesting method some processors use to eliminate buffer time, even with a false branch prediction is having a small pipeline in parallel with the main pipeline. This small pipeline will start executing instructions starting where the processor thinks the branch will NOT bring the processor to. For example, if a branch predictor thinks the branch will not be executed, the

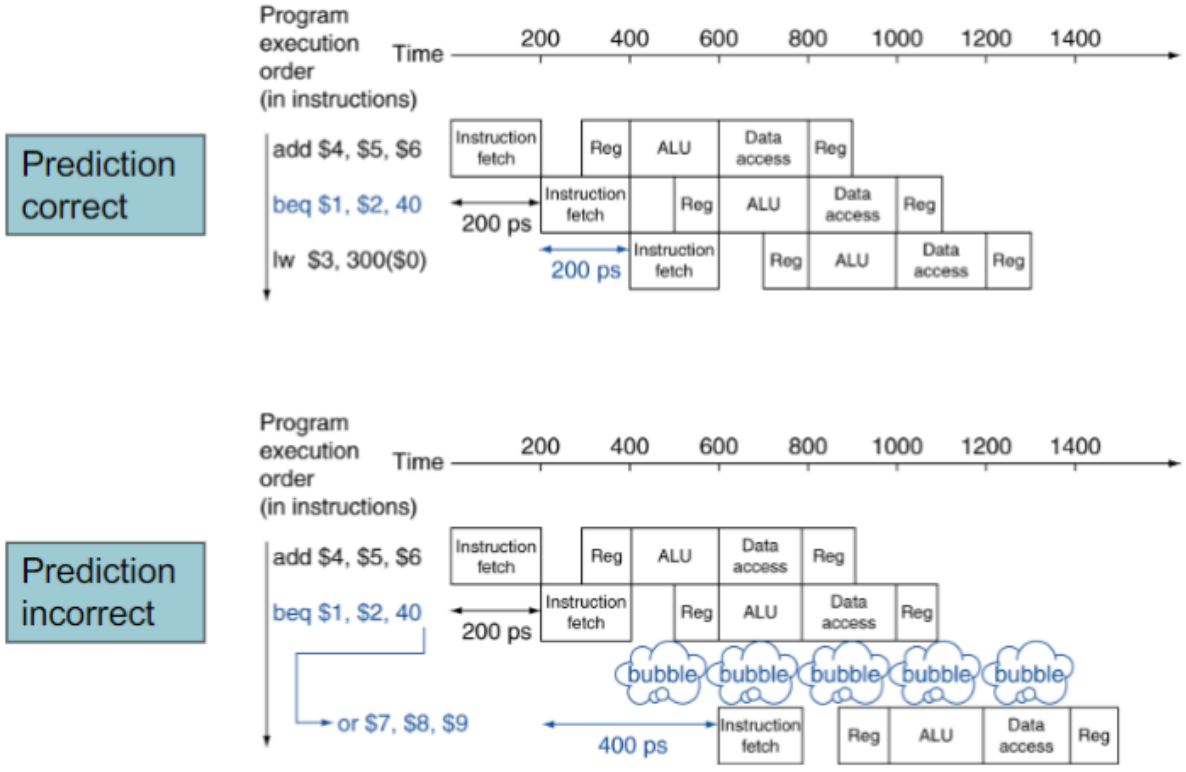

Figure 2.3: Pipeline with correct and incorrect branch prediction [2]

main pipeline will continue executing the instructions as usual, and the side pipeline will execute instructions starting from where the branch would bring the pipeline to if the condition for the branch was met. This way, even if the branch prediction is incorrect, those instructions have been being executed on the side all along, and the transition is much smoother, still with some delay, though [18].

2.1.2.2 Out-of-Order Execution

Out-of-order execution is a method used to maximize hardware usage in a processor. At any given point in a typical pipelined processor design, only one instruction will be in the execute phase. If this instruction is, for example, a load, store, data transfer, or flow control instruction, the ALU will not generally be used, and is sitting idle. Meanwhile, there may be other instructions that are manipulation instructions that will need to use the ALU. Out-of-order execution takes advantage of all hardware items at once, and orders instructions in a way that optimizes the use of hardware, rather than ordering them in the order they are written in the program memory. Out-of-order execution can be very effective in optimizing the processor, as seen from Figure 2.4 but can be a very complicated design component for a processor architect to take on.

2.1.2.3 Reducing Processor Size

Other options that were explored to reduce chip size of a processor were to move from 32-bit processors to 16-bit. This usually presents a problem because 16-bit processors would not have the same functionality and capabilities that a 32-bit would. At the 16-bit level, things like code size and instruction cache efficiency are greatly improved. On the other hand, however, due to the small bit size, 16-bit processors lack support for certain data types and three-address mode [19]. The problem with a processor not being able to handle a three-address mode is that a simple instruction, such as $r0 = r1 + r2$ has to be implemented using two instructions, a move

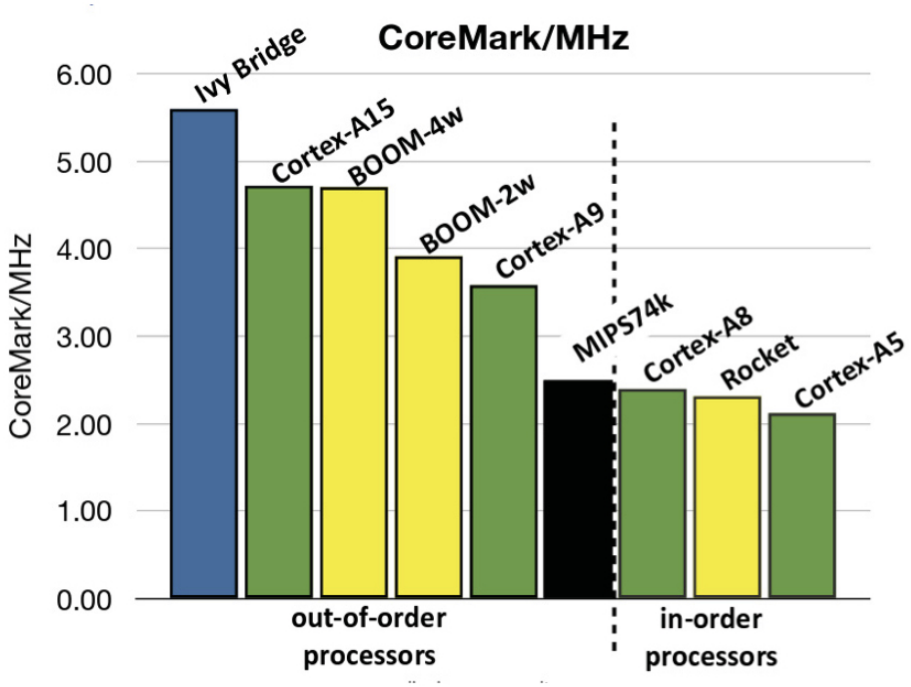

Figure 2.4: Performance of processors using out-of-order vs in-order execution [3]

and then an add: $mov\ r0, r1$ followed by $add\ r0, r2$ [19]. In [19], a process called move-folding is introduced to try and fix this problem to make 16-bit processors more appealing. The process of move folding is an interesting technique where an extra register (MR) is used to hold the value of the extra source register. When it comes time to execute, instead of selecting the destination register as the second source, the MR is used as the second source. This interesting solution could increase the chances of a 16-bit processor being more attractive than before, however there is still extra hardware used to properly implement this kind of instruction.

2.2 Compilers

The quality of a compiler can make a huge difference when designing code for an embedded system. Many compilers are used to decompose basic code languages, such as C, Java, or C++

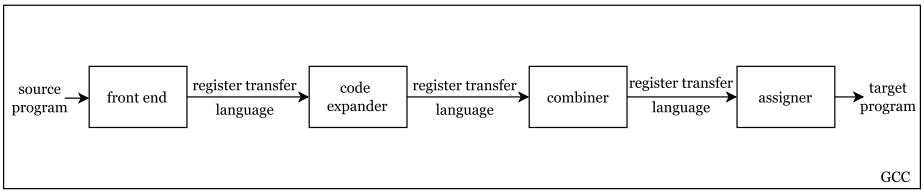

Figure 2.5: Davidson Fraser Model [4]

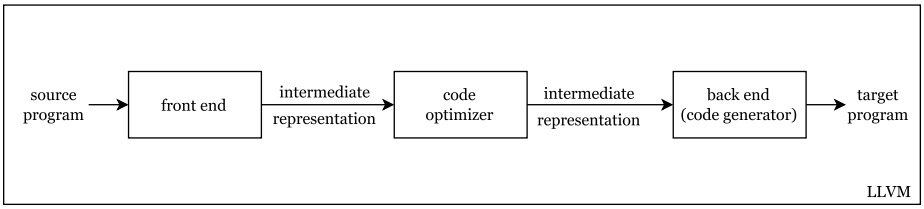

Figure 2.6: Aho Ullman Model [4]

to run on CPUs. How efficient a compiler is can often be determined by how much they condense the code and the dynamic instruction count in the output executable file. Especially in embedded systems, these are indicators of power consumption and execution time [20]. Two main compilers used are GNU Compiler Collection (GCC) and Low Level Virtual Machine (LLVM). These are both often used to compile C code to standard RISC-V or ARM processors, and are often compared using bench-marking programs to determine their abilities to create short code and optimize different aspects like register allocation and dynamic instruction count.

2.2.1 Compiler Design

Compiler's have three main parts to them: a front end, a middle end, and a back end. Many compilers typically do the same general functions in each phase, but how they go about it differs for each compiler. Two main models of compilers are the Davidson Fraser Model, shown in Figure 2.5, which GCC is modeled off of, and the Aho Ullman Model, shown in Figure 2.6, which LLVM is modeled off of [21].

Generally, the front-end of a compiler parses the program and checks for syntax errors and

convert the file into some intermediate representation file format [5]. For Davidson Fraser, this file format is register transfer language (RTL), whereas the Aho Ullman Model uses an intermediate representation specific to the compiler. The middle-end is responsible for most of the optimization. This portion is where a compiler can shine, with the right optimization techniques. Typically, how well a compiler optimizes is based on how well it can condense the code and order the instructions in a way that makes the output as short and as quickly executed as possible. Usually, this portion can be done target-independently. The back end is essentially the code generator, and is responsible for converting the code from the optimized intermediate representation (or the RTL) to the binary the target processor will accept to create the executable. Sometimes, some optimization is left to the back end, depending on the target. Since GCC is the compiler used in this project, and it is based on the Davidson Fraser Model, this will be the model most closely discussed in the rest of this paper.

2.2.2 GCC

Launched in 1984 by Richard Stallman, the GNU Compiler Collection is a portable compiler designed to optimize and compile code for a wide array of processor targets [11]. GCC is one of the most widely used compilers for C and supports more processor architectures, and offers some coder-friendly features making any code compiled by GCC very portable [22]. GCC is compatible with multiple languages in the front end, and uses a common language-independent middle end and back end to compile code into target-specific machine code. No other compiler suite can do quite what GCC can, all while being free, open-source software [23]. Being older than LLVM, the optimization process has been worked on for longer, and is much more efficient and better at its process. An overall block diagram and more specific compilation overview can be seen in Figures 2.7 and 2.8, respectively. The next subsections will break down the three stages of the GCC compilation process: front end, middle end, and back end.

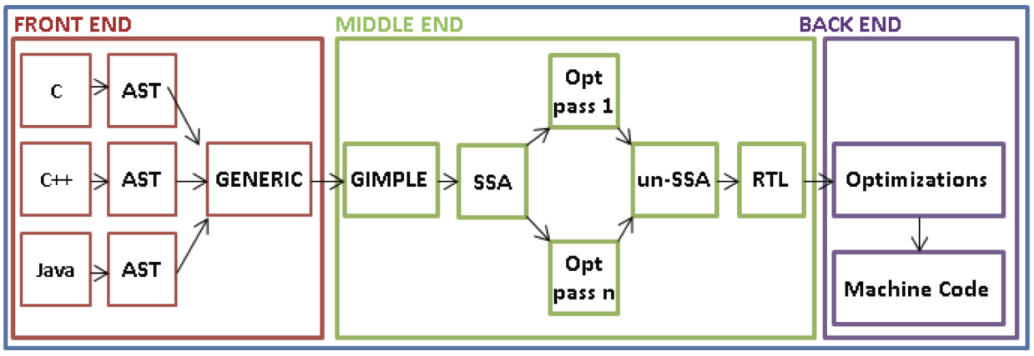

Figure 2.7: GCC compilation process block diagram [5]

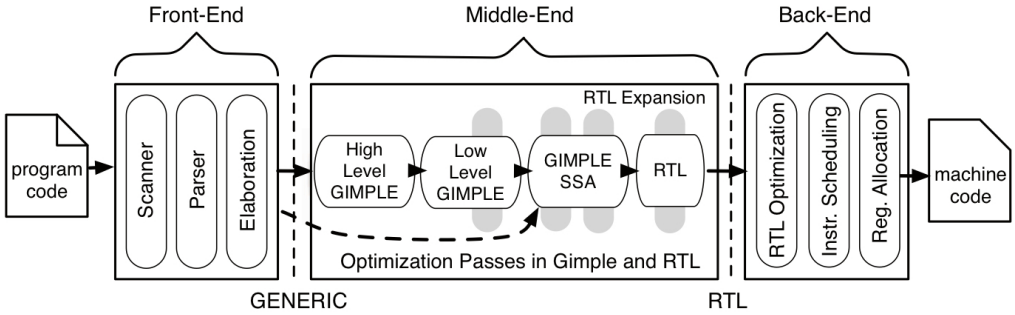

Figure 2.8: GCC compiler overview [6]

2.2.2.1 Front End

As mentioned previously, the front end of any compiler is responsible first for checking for any syntactical errors in the code. The GCC front end specifically will then create the Intermediate Representation (IR) of the code in a tree-like format. The IR is language-independent, and convert many different languages into IR, such as Java, C, C++, and FORTRAN [5]. The tree structure the IR takes the form of is called GENERIC. GENERIC is the language the code is in when it goes from the front end stage of compilation to the middle end stage of the process, as seen in Figures 2.7 and 2.8. Since GENERIC is language-independent, the compiler process used for the middle end can be more universal, used for any language.

2.2.2.2 Middle End (Optimization)

The middle end is responsible for the first part of optimizing code to reduce execution time and output program size. The GENERIC trees are converted to GIMPLE, another language independent IR. In the transformation from GENERIC to GIMPLE, complicated processes and instructions are broken down into simpler statements [5]. This transformation process is called *gimplification*. After the gimplification process, Static Single Assignment (SSA) information is given to GIMPLE to incorporate the data flow. As seen in Figure 2.7, a series of optimization passes are then performed on the GIMPLE with SSA. The SSA is then removed and this is converted into Register Transfer Language (RTL). RTL is the language the middle end finishes in and passes along to the back end [5].

2.2.2.3 Back End

The back end converts the inputted RTL into machine language that is target-dependent. Different target-dependent optimizations can be done to reduce code size and execution time even

more. These optimizations include processes such as instruction scheduling and register allocation. Instruction scheduling reorganizes the instruction stream, only if the target processor has pipeline units [24]. It often attempts to reorganize in a way that causes as few dependencies as possible while still having the correct program results. The register allocator must take the pseudo registers in the RTL and map them each to an actual register in the target processor. Registers are an invaluable and high speed portion of the processor, so register allocation can be considered one of the most important optimization processes in the entire compilation [25]. There are two types of register allocation, intraprocedural and interprocedural. Intraprocedural allocates registers within a procedure, making sure there are enough registers, and making intelligent decisions about how to rearrange when there are not. Interprocedural allocates registers across multiple procedures. This mostly takes care of global variables, keeping them in registers permanently throughout program execution [25]. Of course, after these optimizations are complete, the machine code is generated to be run on the target machine or CPU.

Since the back end is responsible for generating the target-dependent machine code from the given RTL, the back end of GCC is where most processor architects would also retarget GCC to port the compilation process to their individual processor. The retargeting process is not a simple one, but GCC makes it very doable by being open-source. After understanding certain target processor components, such as the register file, pipeline, and ISA, the architect must redefine GCC's application binary interface (ABI) [21]. The ABI contains information regarding alignment of data types as well as defining how the stack will be used in a call, how registers will be used and data transferred, etc. The third step is to define and write three machine description files that will tell GCC about the processor's environment and setup [21]. The fact that very little GCC back end code needs to be altered to retarget the entire compilation process to a new target platform makes GCC very attractive for retargeting purposes.

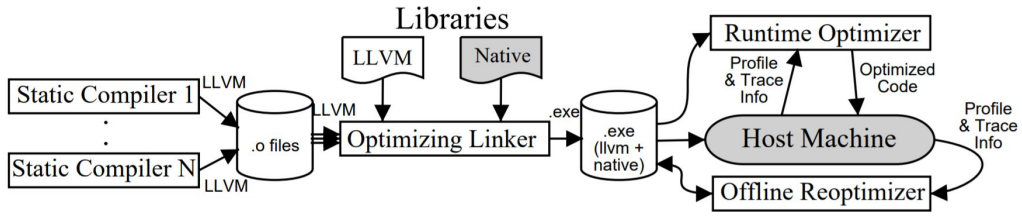

Figure 2.9: LLVM architecture [7]

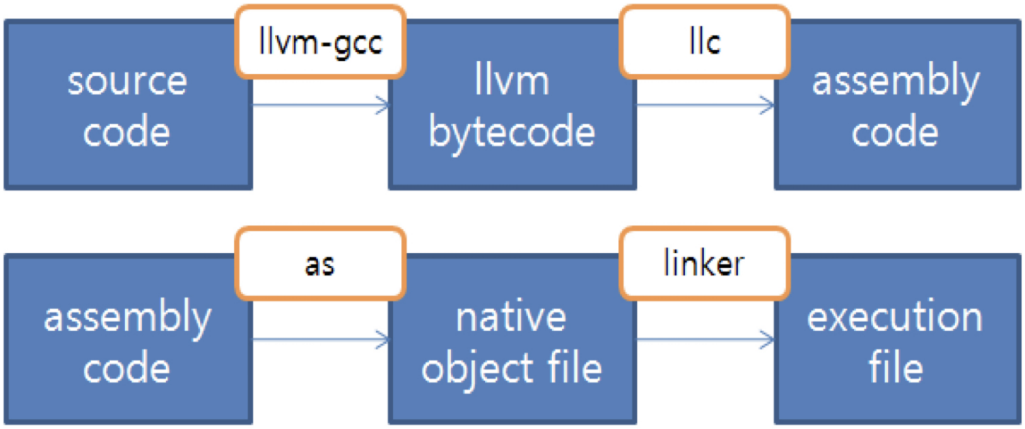

Figure 2.10: LLVM compilation process block diagram [8]

2.2.3 LLVM

Low Level Virtual Machine (LLVM) was created in 2003 by Chris Lattner for a Master's Thesis [7]. LLVM is largely based around optimizing during compile-time, link-time, and run time [8]. Unlike GCC, which compiles into an IR to optimize, and then overlays SSA, LLVM compiles directly to an IR that is in SSA form to perform optimizations. LLVM's architecture can be found in Figure 2.9. The LLVM compilation process can be broken down into a block diagram, as shown in Figure 2.10.

The LLVM compilation process starts with source code that goes through llvm-gcc. Llvm-gcc is a front end compatible with C code based of GCC. It compiles C programs into LLVM bitcode, which is usually an object or executable file [8]. Next, the llvm bitcode goes through llc,

Table 2.2: GCC vs LLVM Compiler Characteristics

Characteristic	GCC	LLVM
Compiler Model	Davison-Frasser	Aho Ullman
Machine Description	RTL	SSA IR
Main optimization points	instruction scheduling and register allocation during back end	link-time
Register Allocation	Inter and intra (local and global)	Linear scan

which compiles the bitcode into assembly of the host machine, or another machine if specified. This is then assembled into the native object file, which is put through the llvm linker, which can merge multiple llvm bitcodes stemming from multiple C files into one execution file. The linker also provides most of the optimizations performed in the LLVM compilation process.

2.2.4 GCC vs. LLVM

GCC and LLVM have been comparing using many different variables and benchmarking programs. Table 2.2 compares some of the basic compiler features between GCC and LLVM.

It was found in most results that GCC is overall a better compiler due to its robustness and better optimization techniques. There are, however, pluses and minuses to both. In the end, however, code size determines memory length, and memory is the most expensive unit in a CPU, so whichever compiler makes the shortest programs takes the cake [20]. In a study done to compare GCC and LLVM on the EISC Processor, it was found that although LLVM was good with calculation intensive programs, GCC had the leg up on register allocation and jump optimization. As mentioned previously, register allocation can be one of the most crucial optimizations for shortening execution time of a program [20]. LLVM was able to get through calculation intensive functions due to its superior loop-unrolling techniques. LLVM is known to aggressively unroll

loops, which can end up shortening programs quite a bit. The GCC compiler, however, uses registers more often and more efficiently, and rearranges blocks in order to eliminate unconditional jumps, a factor that sets the bar pretty high for competitor optimizers [20]. In another study done to compare LLVM and GCC on an ARM Platform, it was found that while LLVM falls close to GCC in some benchmarks, in others it pales greatly in comparison. Specifically, code with heavy memory access was handled inefficiently by LLVM [8]. However, it was deemed that since most programs have many files and many function calls, and LLVM has a strong suit in linker optimization, LLVM's performance with these types of programs would shine, and be nearly equivalent to GCC [8].

Chapter 3

Custom 32-bit RISC Processor Design

This chapter discusses the design and relevant aspects of the 32-bit processor, including why certain design choices were made, and what alternatives were available.

3.1 Instruction Set Architecture

The instruction set architecture (ISA) of a processor is key to its design. The ISA contains information regarding how many bits the processor will be, the number of general purpose registers, the load-store technique, memory structure, input/output peripheral structure, addressing modes, and more. Most of the general functionality of the processor is defined in the ISA. For the DMF RISC processor, the processor is 32 bits to make it easier to match up with C compiled code. This means that each register will be 32 bits long, and memory would be capable of containing 2^{32} memory locations, more than enough, even if a Von Neumann memory architecture was chosen.

3.1.1 Register File

The register file contains the set of all registers used in this design. There are 32 general purpose registers, each of them capable of storing numbers up to 32 bits, which is the equivalent of a float in C. Other important registers include the status register, the stack pointer register, and the program counter. The status register is a 32-bit register where only 4 bits are used to keep track of the carry, negative, overflow, and zero (CNVZ) bits from any computation done by the processor. The status register is critical in making a decision for conditional jumps. The layout of the status register can be seen in Figure 3.1.

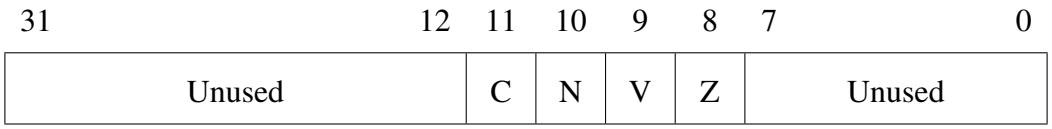

Figure 3.1: Status Register Bits

The status register bits (CNVZ) can be explained as follows:

- C: C is the carry bit. This bit is set to 1 when an executed manipulation instruction has a high carry out. This mostly applies for addition and rotate instructions. This bit is otherwise a 0.
- N: N is the negative bit. If any manipulation instruction results in a negative number (1 in the most significant bit place), this bit will be set to 1. Otherwise this bit will remain at 0.
- V: V is the overflow bit. If a manipulation instruction has operands that lead to a result that is too large to fit in a 32-bit register, the overflow bit will be set to 1. Otherwise this bit will remain at 0.
- Z: Z is the zero bit. If a manipulation instruction results in zero, this bit will be set to 1. Otherwise this bit will remain at 0.

Figure 3.2: Program Counter Register

The program counter is important for keeping track of which address in memory holds the next instruction to be executed. The program counter (PC) needs to have enough bits to contain every possible address in the program memory, usually the ROM (read-only memory). Since this processor is 32-bits, the PC is 32 bits long, and can therefore point to 2^{32}memory locations, if needed. The layout of the program counter register can be seen in Figure 3.2.

Lastly, the stack pointer is a register used to keep track of what spot in the stack the processor will use to store information in next. In this processor, the stack is mostly used to store information, such as the SR and the PC, for maintaining the state of the processor when returning from a call. When items are added to the stack, the stack pointer will decrement to point to the next location in stack. The stack pointer in this processor starts at the top and stores in decreasing address orders. This does not change the standard behavior of the stack, where items are pushed and popped into and out of stack. Like most stacks, the stack is also still last in first out, or LIFO. The layout of the stack pointer register can be found in Figure 3.3.

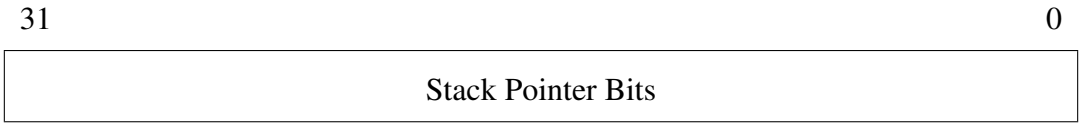

Figure 3.3: Stack Pointer Register

3.1.2 Peripheral Layout/Stack Design

For the DMF RISC processor, the peripherals are memory mapped, meaning that the peripherals are accessed by using a designated spot in the data memory RAM. The alternative to memory-mapped peripherals is separate-mapped peripherals, which are accessed via a memory block that

is separate from the data memory. In this memory-mapped peripheral design, the stack is treated as a peripheral. Since the stack is only used for storing the PC and SR during a call-return sequence, not much space needs to be designated to stack. From memory spaces 0x3ff0 to 0x3fff are designated to the stack and other peripheral uses.

3.1.3 Memory Architecture

Two main memory architectures that are commonly used in processors are Harvard and Von Neumann. Von Neumann is a structure where the program and data memory both share one block of memory. Typically, the data memory will start at address 0 (or some other arbitrary address value) and fill upwards, while the program memory will start at the last address and fill down.

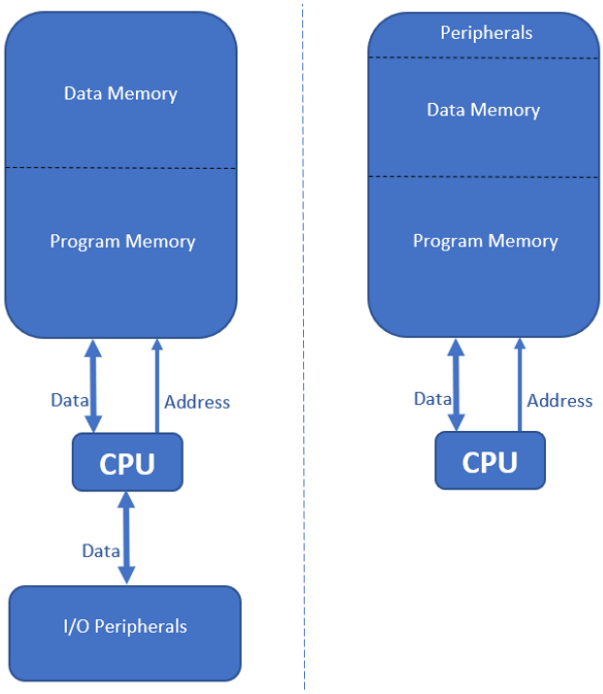

Figure 3.4: Von Neumann memory structure with separate-mapped (left) vs. memory-mapped (right) I/O peripherals

For this processor, the Harvard memory architecture was implemented. This means that a different memory block was used for the program memory and the data memory [26]. For the DMF RISC processor, a read-only memory (ROM) block was used for the program memory, since the program memory is never written to after the initial programming. A random-access memory (RAM) block was used for the data memory, where the peripherals and stack are also located. The memory and stack layout of this processor can be seen in Figure 3.5.

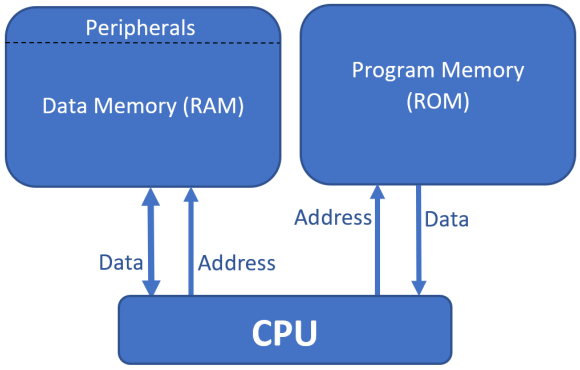

Figure 3.5: Harvard memory structure with memory-mapped I/O peripherals

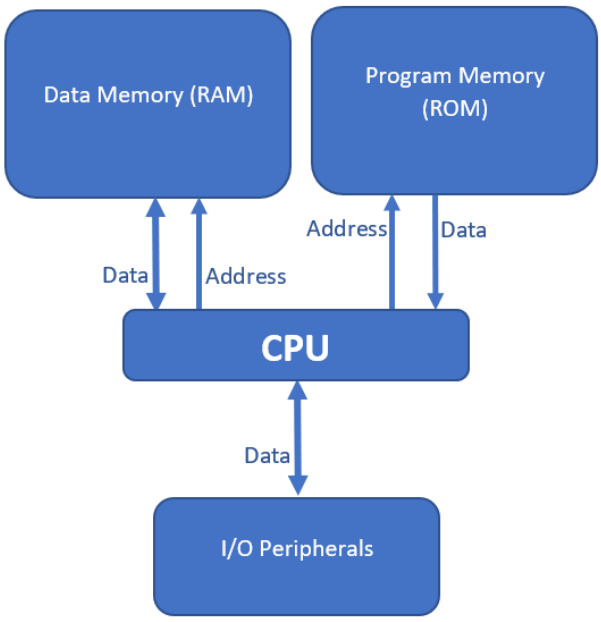

Figure 3.6: Harvard memory structure with separate-mapped I/O peripherals

3.1.4 Addressing Modes

Addressing modes allow the program writer to reference addresses in memory in different methods. Available in the DMF RISC is direct addressing mode, PC-relative addressing mode, and register direct addressing mode using the general purpose registers. The program can also use the SP as the address. How these addressing modes are implemented is talked about later, in section 3.3.1, specifically the load and store subsection. Figure 3.7 also visually explains the different forms of addressing modes used in most basic RISC processors.

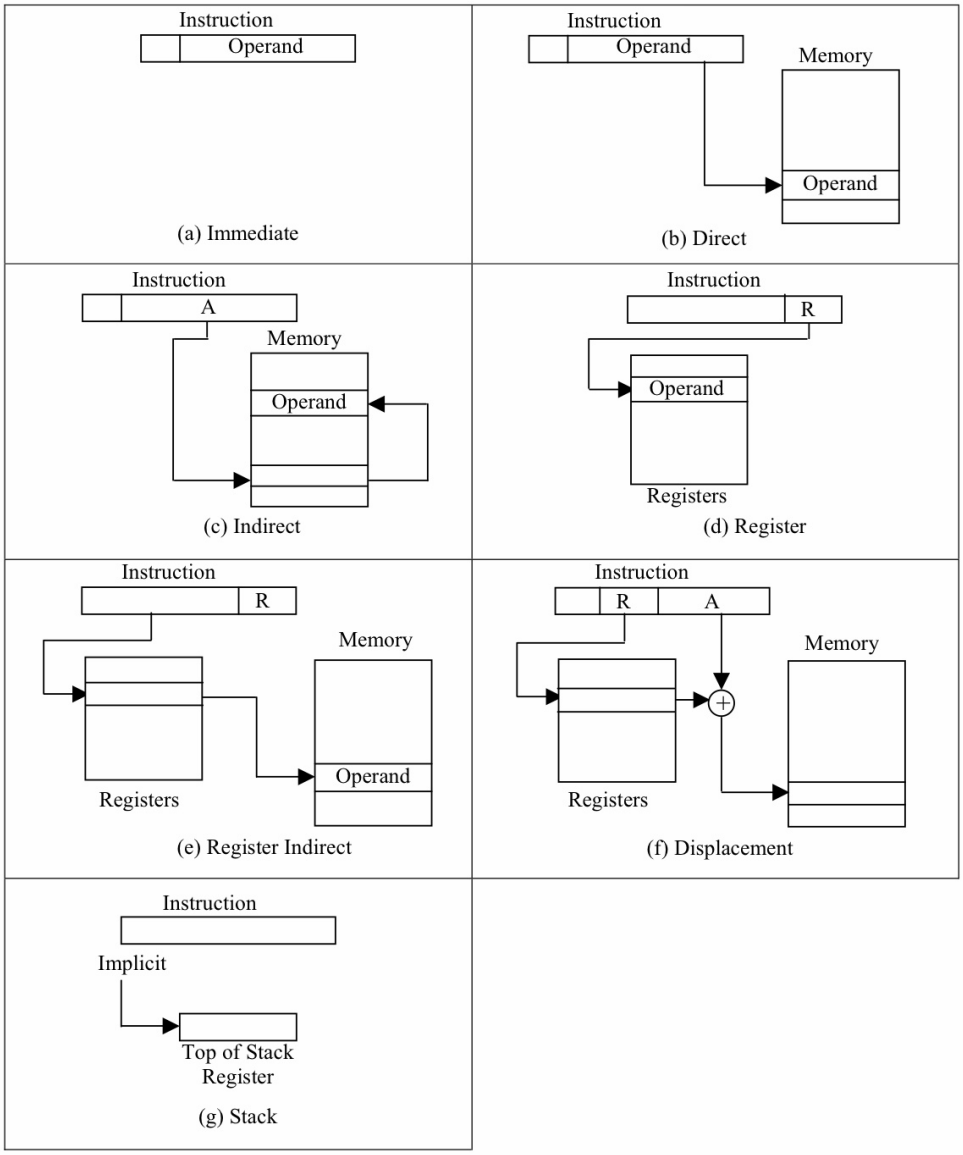

Figure 3.7: Addressing mode types [9]

3.2 Hardware Implementation

There are many functional pieces that make up the hardware implementation of the DMF RISC processor designed for this project. The DMF RISC processor was designed using Verilog hard-

ware description language (HDL) at the register transfer level (RTL). Some features of the hardware implementation are the four-stage pipeline, the arithmetic logic unit (ALU), the shifter, the multiplier/divider unit, the register file, the cache memories, as well as the main memories (program and data), and the phase-locked loop (PLL) clock generator (not pictured in Figure 3.8).

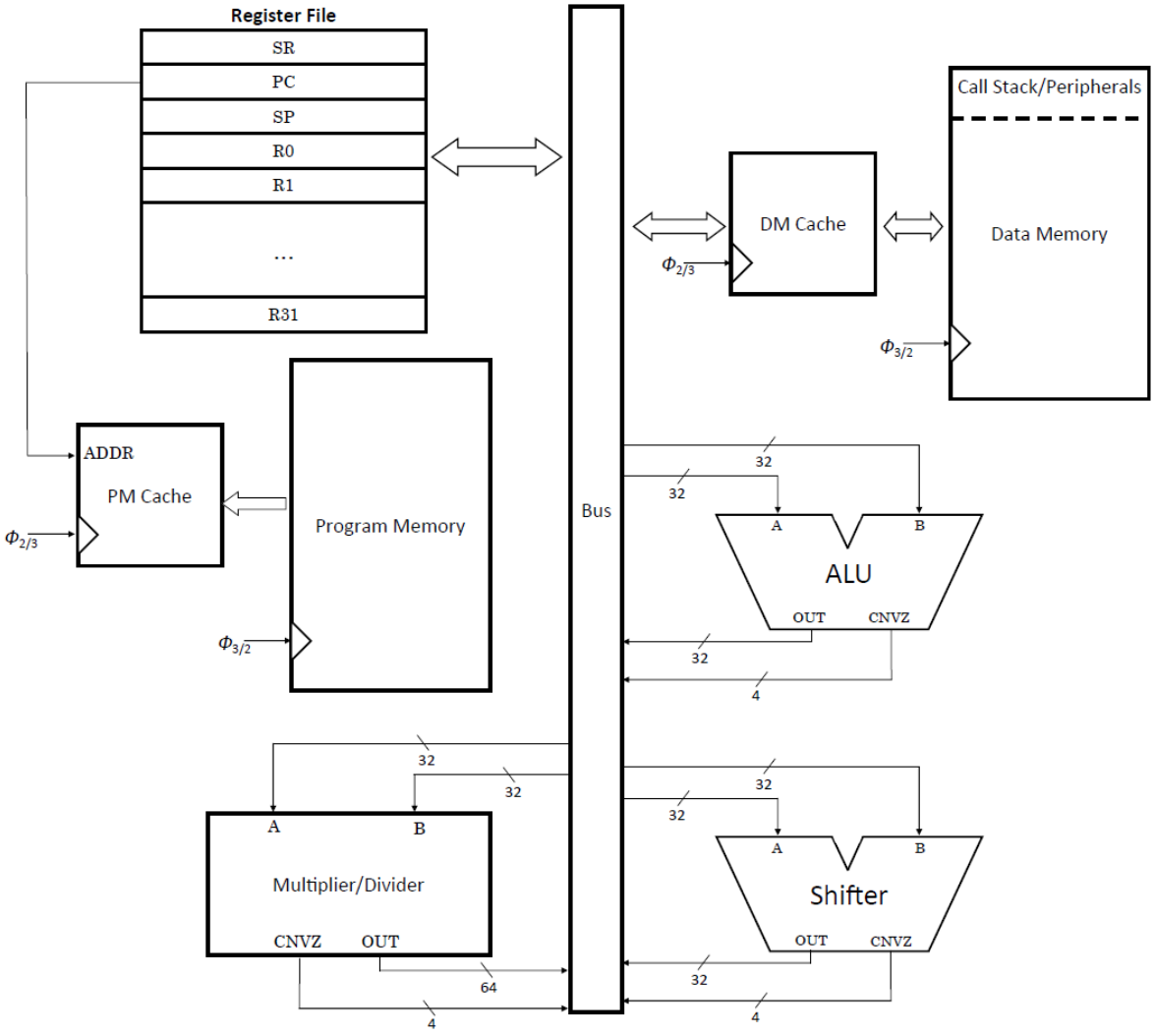

Figure 3.8: DMF RISC CPU Functional Block Diagram

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

Figure 3.9: Four-Stage Pipeline [4]

Figure 3.10: Four-Stage Pipeline Block Diagram[4]

3.2.1 Pipeline Design

The use of a pipeline is key for increasing the throughput rate of instructions being executed in a CPU [27]. The four-stage pipeline in the DMF RISC Processor consists of the following four stages: instruction fetch, operand fetch, instruction execution, and write-back. After the first instruction is finished with the instruction fetch stage, the next instruction may enter the pipeline, and so on. In the CPU implementation, it's important to note the order that the instructions are executed during any single clock cycle. Looking at Figure 3.9, if the processor is in clock cycle 4, there is an instruction in each of the pipeline stages. For the purpose of fixing any dependency issues, the write-back stage with instruction I_0 will be executed first, then the execution with I_1, then the operand fetch with I_2 and finally, the instruction fetch with I_3. The implications of dependencies will be talked about more later in section 3.2.1.2.

3.2.1.1 Instruction Fetch

The instruction fetch (IF) stage is fairly straightforward. This stage consists of loading the next instruction word from the program memory cache and transferring it into internal registers in

the main CPU. The instruction word is generally broken up into the op-code and other relevant pieces of the instruction word, such as operand values. The program counter is then incremented to point to the next instruction word. The instruction fetch stage is also responsible for stalling the processor if the next instruction will be a jump, call, return, load, or store. Any instruction requiring a change in program counter or a second instruction word requires some processor stalling to execute properly.

3.2.1.2 Operand Fetch

The operand fetch (OF) stage largely depends on the type of instruction being looked at. For manipulation instructions, the original instruction word had two operands, two source registers, and a destination register that the result is stored in. This data must be used to find which general purpose registers are being used in this instruction, and obtain the data from within these registers. The OF stage is also responsible for working out all dependency issues. In pipelining, multiple dependency issues can be encountered, such as read after write (RAW), write after read (WAR), and write after write (WAW). The one that is focused on being handled in this pipeline structure is read after write. If two consecutive instructions use the same operand as the destination operand, it is critical that the value from the first instruction be placed in the operand before the next instruction is executed. To solve this problem, a technique called data forwarding is called. The operand fetch stage is responsible for executing all data forwarding. Data forwarding is the process of using internal registers to transfer the value of an operand after an instruction is executed to the next instruction for execution with the proper number values. For example, if an instruction that is the equivalent of R2 = R2 + R1 is followed by an instruction that uses the value of R2 to multiply with R3, data forwarding will be needed because while the addition instruction is in the execution phase, the multiplication instruction will be in the operand fetch stage, where the values of the registers will be obtained. In this case, when the

multiplication instruction is in the OF stage, instead of obtaining the value of R2 by looking in the register file, the processor will use the result of the addition instruction as the value of R2. Because there are many cases that could occur that would require data forwarding, the OF stage is one of the most logic-heavy stages, with many conditional statements to make sure all bases are covered.

3.2.1.3 Instruction Execution

Instruction execution (EX) is exactly what it sounds like, the actual execution of the instruction. If the instruction is a manipulation instruction, the ALU, shifter, or multiplier/divider is used to perform logic on the operands and store the result in an internal register. If the instruction is a load or store, in this phase, the data will be stored to or loaded from memory. If the instruction is a jump, call, or return, during this stage, the program counter will be set to the new location for it to continue keeping track of instruction words in the program memory.

3.2.1.4 Write Back

In the write back phase, the result of the instruction execution is written back into the general purpose destination register, if this applies. If the instruction was a flow control instruction, such as a jump, call, or return, this phase will also be responsible for telling the processor there is no longer a need to stall.

3.2.2 Cache Memory

Cache memory is an important feature when a processor would otherwise be reading/writing from large, monolithic blocks of memory. In hardware, the blocks of memory being used for program and data memory can be very large and far away from the CPU, making interactions with these blocks take a long time. To solve this issue, often cache memories will be used. A

cache memory is a smaller block of memory that temporarily stores relevant blocks of memory to be used with the CPU. It is up to the architect to decide what makes a block of memory relevant, and how much memory will be placed in cache at a time. For the DMF RISC processor, the cache memories each contain 8 blocks of memory, where each block has 16 words.

The way this processor decides which blocks of memory are placed in cache (placement strategy) are simply based on which addresses are used by the CPU. When an address is called for by the CPU, the corresponding block will be moved into cache so the CPU can access that address. Once the cache is full, a replacement strategy must be implemented to decide which blocks will be overwritten to accommodate for the new block needed by the CPU. In this processor, the replacement strategy is the block of memory that was used the longest time ago is the one that will be replaced by the new block of memory.

There are a few ways to map the main memory blocks to blocks in cache.

- Associative: For a cache that is fully associative, any block in memory can be placed anywhere in the cache. This means for a replacement strategy, all blocks in cache will have to be compared to see which was the least recently used. A visual of this cache organization can be seen in Figure 3.11.

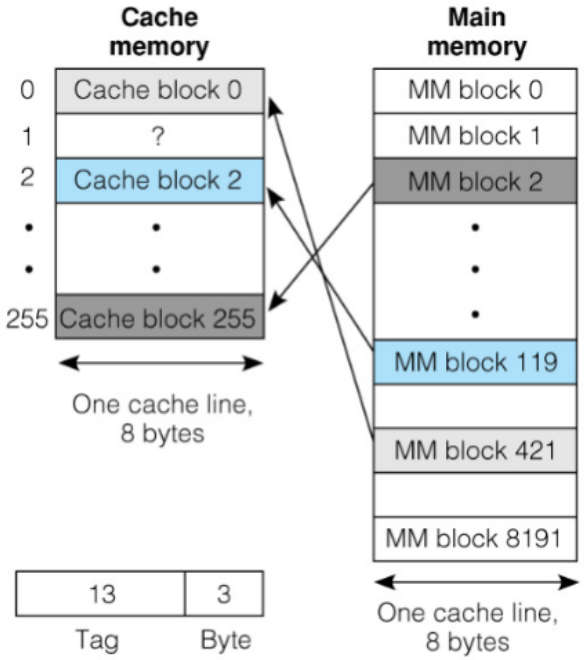

Figure 3.11: Fully associative cache organization[10]

- Direct: For a cache that is direct mapped, each block in main memory has a designated block in cache, depending on the group bits in the address. Each address in main memory is broken into parts that help for organizing where each block belongs. This organization is broken down in a visual in Figure 3.12.

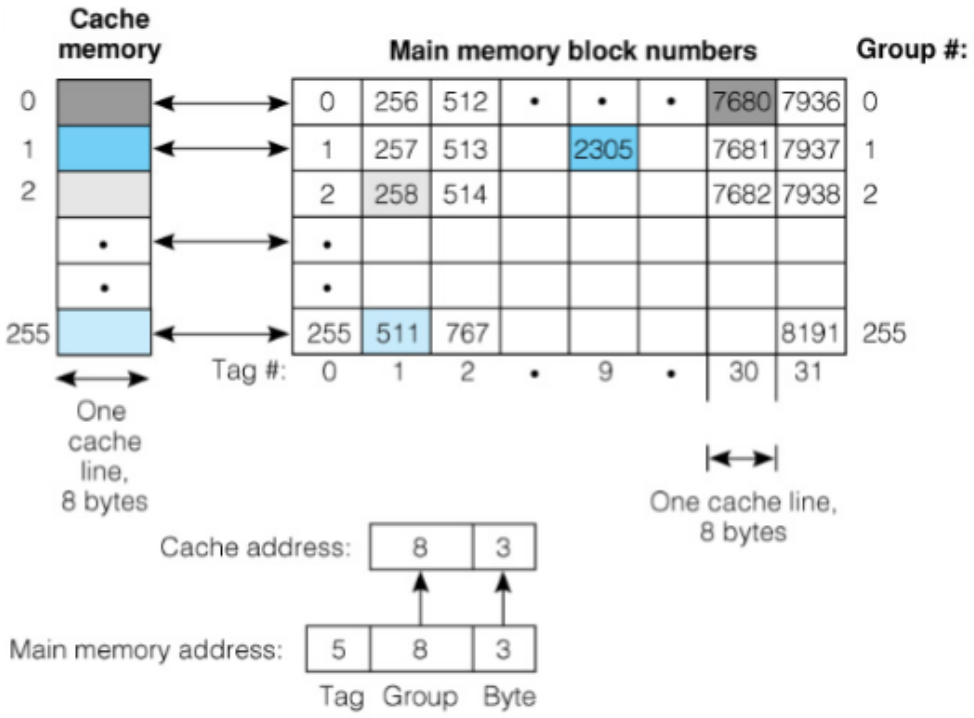

Figure 3.12: Direct-mapped cache organization[10]

- Block-set Associative: Block-set associative can have any number of blocks, for example 4-way set associative or 2-way set associative. This organization method is in between fully associative and direct mapped. Instead of only having one spot each memory block can map to, like in direct-mapped, there are a set number of spots each block in memory can map to. If the organization is 2-way set associative, like in Figure 3.13, there are 2 spots in cache each block in main memory will be able to map to.

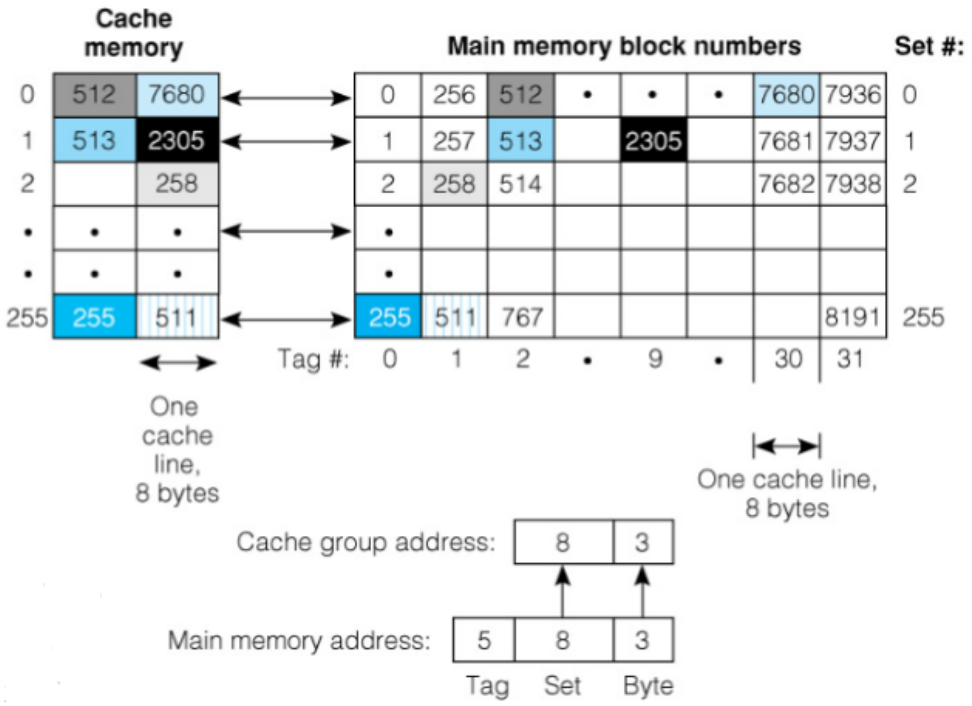

Figure 3.13: 2-way set associative cache organization[10]

Using an organization technique like direct-mapped or block-set associative makes it easier to decide which blocks to replace when the cache is full. Sometimes, however, using these methods makes no sense. For example, in the case of a program memory where the addresses will usually be called chronologically, it might make more sense to use fully associative rather than direct-mapped or block-set associative since the latter two will restrict how many relevant blocks of program memory will be able to be in cache at a time. The DMF RISC Processor uses 2-way set associative to map the main memories to the cache.

There are some general properties that can be used to decide on an effective replacement strategy for cache memory. These include temporal and spacial locality. Temporal locality is a property of most programs where if a specific address in the memory is used by the CPU, it will most likely be used again in the near future. Furthermore, spacial locality is a property that if an

address in memory is used by the CPU, it is likely that addresses nearby will also be used in the near future. This is especially true for the program memory, where generally, the addresses are needed in chronological order, with the exception of the case of a jump, call, or return.

Once it has been decided that a block in cache will be replaced, it is also important to keep track of if this block of memory needs to be written back to the main memory or not. If while in the cache, a block of memory is at all written to, a write-back of that block will need to be done to ensure that the main memory reflects the new state of that block of memory. It is important to note that some memory blocks that do not have a write enable, like ROM, will not require a write-back. This is true for program memory, in particular.

In the DMF RISC Processor, there are two cache memories, one for the data memory, and one for the program memory. Since the memory structure was Harvard and then DM and PM were separated, this was necessary. This can be seen in Figure 3.8.

3.2.3 Stalling

Stalling is required during any flow control instructions, such as jump, call, and return, as well as for load and store, which use a second instruction word as the address in data memory being accessed. The stalling is required for different reasons for each of these cases. For the flow control instructions, stalling is required because until the program counter is set to the new address, there is no way to get the next instructions to execute. For the load and store, stalling is implemented for only one clock cycle so that the instruction work with the memory address isn't implemented as an actual instruction. Additionally, because the data memory is being accessed, often a clock cycle is needed to get the information from the data memory.

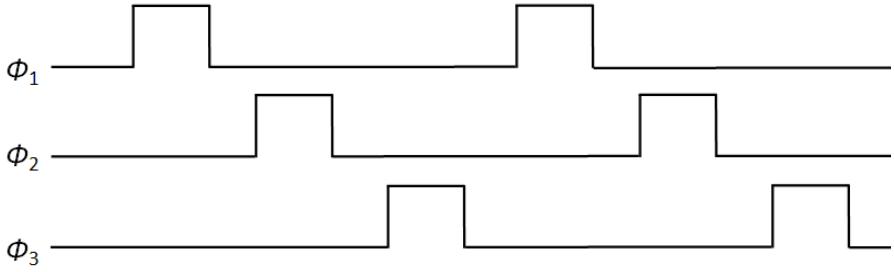

Figure 3.14: Clock Phases

3.2.4 Clock Phases

For the best efficiency, a PLL is used to generate three clocks with phases 30° phase shifts. The $\phi_1$clock was used as the CPU clock. The other two, $\phi_2$and $\phi_3$were used as the cache/main memory clocks. Which clock was used for which memory block depended on if the main memory was writing to cache or if cache was writing back to main memory. This clock layout made for the most efficient memory-CPU execution.

3.3 Instruction Details

This section details the implementation of the different instruction types.

3.3.1 Load and Store

Load and store instructions are used to transfer data in and out of memory to general purpose registers. The load/store operations require two instruction words. The first instruction word (IW0) contains the opcode and an R_d, R_i, and R_j field. The $R_i$field is used to determine the addressing mode of the instruction, the $R_d$field is used to tell the processor which general purpose register to load into, and the $R_j$field is used to tell the processor which general purpose register

to store from. The second instruction word (IW1) contains the address offset the CPU will use to load or store. The instruction word breakdowns can be found in Figure 3.15.

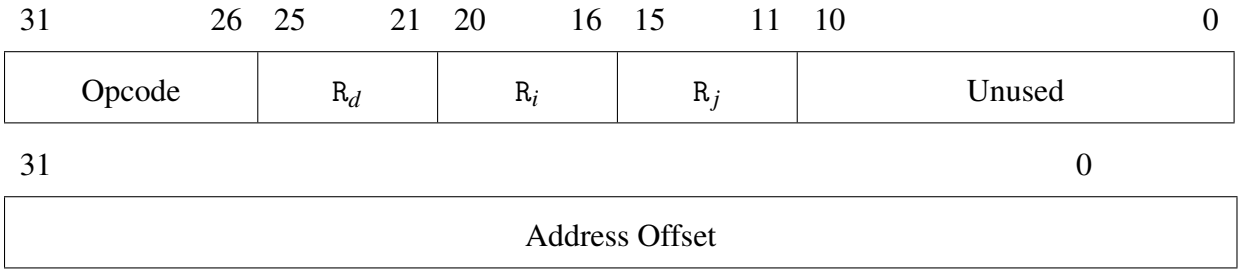

Figure 3.15: Load and Store IW0 (top) and IW1 (bottom)

The addressing mode used depends on the value in the $R_i$field. The possible addressing modes to be used are absolute, PC relative, SP, and register direct. A description of each of these and their corresponding $R_i$field value can be found in Table 3.1.

Table 3.1: Addressing Mode Descriptions

Mode	R_i	**Effective Address Value**
Absolute	0	The value in the address field
PC Relative	1	The value of the PC register + the value in the address field
SP	2	The value of the stack pointer (SP) register
Register Direct	3-31	The value of the R_i register operand

Table 3.2: Load and Store Instruction Details[4]

Instruction	Mnemonic	Opcode	Function
Load	LD	0x15	Load value from memory at the effective address or I/O peripheral into the R_d register
Store	ST	0x16	Store the value in the R_j register into memory at the effective address or I/O peripheral

3.3.2 Data Transfer

Data transfer instructions are the swap and copy instructions. These only require one instruction word since they do not have an address offset. The swap and copy instructions solely deal with register to register interaction. Swap swaps the values in two registers, while copy copies the value from one register into another. The instruction word layout and instruction descriptions can be seen in Figure 3.16 and Table 3.3, respectively.

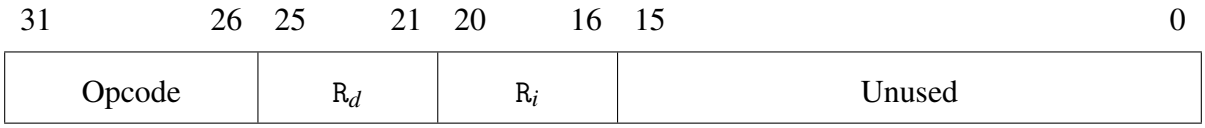

Figure 3.16: Data Transfer Instruction Word

Table 3.3: Data Transfer Instruction Details[4]

Instruction	Mnemonic	Opcode	Function
Copy	CPY	0x17	Copy the value from the R_i register into the R_d register
Swap	SWP	0x18	Swap the values in the R_d and R_i registers

3.3.3 Flow Control

Flow control instructions include all jumps, conditional and unconditional, as well as call and return. The distinct difference between a jump and a call is that a jump can not be returned from, whereas a call can. This is because during a call, the SR and PC are stored in stack so that upon return the values can be retrieved from the stack and stored back into their respective registers. For all of these, two instruction words are required, again an IW0 with the op-code and addressing mode (and for a jump, conditional information), and an IW1 with the address offset. The last 4 bits, C, N, V, Z, are all used to tell the condition of the jump. For a call and return, these field values are ignored or left as 0s, since only the opcode and IW1 are needed for the call and return instructions. The instruction word layouts can be found in Figure 3.17.

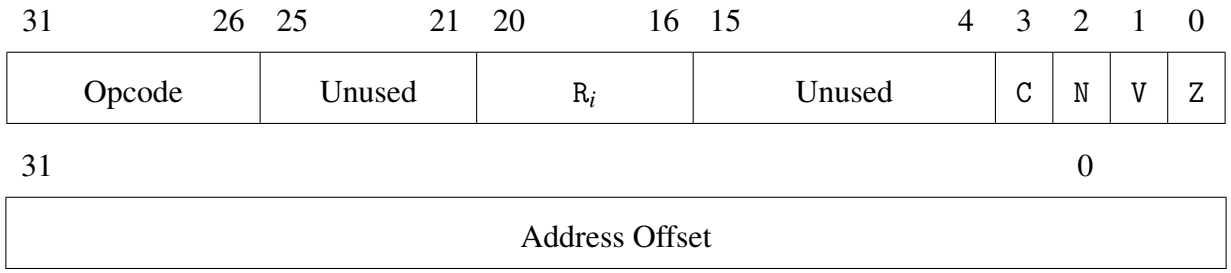

Figure 3.17: Flow Control IW0 (top) and IW1 (bottom)

For conditional jumps, the status register is checked to see if the corresponding bit meets the requirement for the jump before the jump is executed. This means sometimes there is a stall in the CPU and then no jump is made because the condition fails. There are methods that can be used to predict if the jump will fail or not, however these were not used in the DMF RISC Processor. The conditional and unconditional jump descriptions can be found in Table 3.4.

Table 3.4: Jump Condition Code Description[4]

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

Table 3.5: Flow Control Instruction Details[4]

Instruction	**Mnemonic**	**Opcode**	**Function**
Jump	JMP	0x19	Conditionally (or unconditionally) set the PC to the effective address
Call	CALL	0x1A	Write the PC followed by the SR onto the call stack, set the PC to the effective address
Return	RET	0x1B	Read the top of call stack into the SR, then read the next value into the PC

3.3.4 Manipulation Instructions

Manipulation instructions are some of the more basic instructions that manipulate the contents of registers. These include adding, subtracting, multiplying, and dividing, as well a logical manipulation, such as AND, OR, XOR, and NOT. These instructions all take R_i and $R_j$fields and use the contents in the corresponding registers to perform the manipulation. The R_d register is

the destination register because the result of the manipulation gets stored in this register. The R_i and R_j registers are referred to as the source registers. In some cases, like the multiply and divide instructions, the R_i register contents will also be replaced by results of the manipulation instruction.

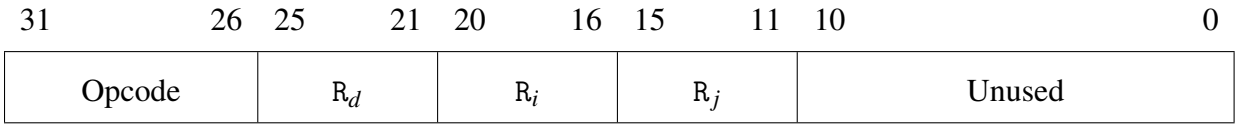

Figure 3.18: Manipulation Instruction Instruction Word

Table 3.6: Manipulation Instruction Details

Instruction	Mnemonic	Opcode	Function
Add	ADD	0x1	Store $R_i + R_j$ in R_d
Subtract	SUB	0x2	Store $R_i - R_j$ in R_d
Add constant	ADDC	0x3	Store$R_i + R_j$ in R_d, where R_j is a constant
Subtract constant	SUBC	0x4	Store $R_i - R_j$ in R_d, where R_j is a constant
Multiply	MUL	0x8	Store the most significant half of $R_i \times R_j$ in $R_i$and the least significant half R_d
Divide	DIV	0x9	Store the quotient of R_i/R_j in $R_d$and the remainder in R_i
Exclusive OR	XOR	0xA	Store $R_i \oplus R_j$ in R_d
Invert	NOT	0x11	Store $\sim R_i$ in R_d, ignores R_j field
AND	AND	0x13	Store $R_i \bullet R_j$ in R_d
OR	OR	0x14	Store R_i OR R_j in R_d

3.3.4.1 Shift and Rotate

Last, the shift and rotate instructions are used to shift and rotate values in registers a certain number of times. For these instructions, the value in the R_i register is the one shifted or rotated, and then stored in the R_d register

. The number of times the value is shifted or rotated is dependent on the value in the R_j field. In the DMF RISC Processor, the value will not shift more than 4 times to the right or left.

Table 3.7: Shift and Rotate Instruction Details

Instruction	**Mnemonic**	**Opcode**	**Function**
Shift right logical	SHRL	0xB	Shift R_i right logically by R_j bits and store in R_d
Shift right arithmetic	SHRA	0x12	Shift R_i right arithmetically by R_j bits and store in R_d
Rotate right	ROTR	0x5	Rotate R_i right by R_j bits and store in R_d
Rotate left	ROTL	0xC	Rotate R_i left by R_j bits and store in R_d
Rotate left through negative	RLN	0xD	Rotate R_i left through negative bit in SR by R_j bits and store in R_d
Rotate left through zero	RLZ	0xE	Rotate R_i left through zero bit in SR by R_j bits and store in R_d
Rotate right through negative	RRN	0xF	Rotate R_i right through negative bit in SR by R_j bits and store in R_d
Rotate right through zero	RRZ	0x10	Rotate R_i right through zero bit in SR by R_j bits and store in R_d

Chapter 4

GCC Back End Alterations

This chapter discusses what changes were made to the back end of GCC with respect to the OR1k files to match the opcodes of the DMF RISC processor.

4.1 OR1k Structure

The Open RISC 1000 Processor (OR1k) is supported by the GCC C compiler. This means there are a collection of files that can be used in the back end of GCC to port the intermediate language or RTL to be compatible binary machine code with the processor. The OR1k is the closest processor to the DMF RISC Processor, so it is the only one that could be used with some basic opcode manipulation.

4.1.1 General Structure

The OR1k is somewhat complicated in its instruction set as it has its instructions divided by addressing mode. Some instructions that have an immediate addressing mode (where a constant is used in the manipulation), have a standard 6-bit op code followed by a destination and source

register and then the 16-bit constant value. This, however, is not how all instructions work. Other instructions that use register-register for manipulation have a general 6-bit opcode in the beginning, the three registers (5-bits each, destination, A, and B), followed by some unused bits and then 4-5 bits of a specific opcode at the end of the instruction word. This differs from the DMF RISC processor majorly, and causes some misalignment when trying to match the OR1k instruction words to the DMF RISC processor's.

4.1.2 Alterations made to OR1K Back End

The alterations made to the OR1k filed in the back end of GCC were simply the changing of the opcode values.

4.2 GCC Back End

GCC's back end is used for porting the IR to the machine code compatible with a specific processor. By default, this processor is the machine the code is being compiled and run on, but other processor architectures can be chosen, like in this case, the OR1k processor.

4.2.1 Altered Files

The files that needed to be altered to create machine code that is compatible with the DMF RISC Processor were limited to the OR1k files. In the file path ../toolchain/binutils/opcodes, there are a list of 8 files that coincide with creating the machine code for an OR1k processor, specifically. The files are:

- or1k-asm.c
- or1k-desc.c : tables of opcode output values

- or1k-desc.h : contains actual opcode values, as well as other opcode relevant enumerations (enums) used in or1k-desc.c
- or1k-dis.c
- or1k-ibld.c
- or1k-opc.c : contains some opcode structures, and also how the instructions will be printed in the disassembly file
- or1k-opc.h : contains some enums that are used in or1k-opc.c
- or1k-opinst.c

The main file that was edited was the or1k-opc.c since it contained the opcode values that needed to be altered. These specific opcodes were hard-coded into their shifted bit positions in an enum table, as can be seen in Figure 4.1. The last line of code in these blocks includes a hex number that is 8 digits long (or 32 bits, the length of one instruction word). The most significant six bits of these hex numbers are the opcode for the corresponding instruction. For example, the opcode of the immediate add instruction for the OR1k processor is 0x27, or decimal 39. Since this opcode needs to be in the most significant 6 bits, however, this is shifted up and the bits go from least significant spaces to most significant spaces, as can be seen in Figure 4.3. Since in the DMF RISC processor the opcode for an immediate add instruction is simply 0x03, this shifted to the left twice would be 0x0C, as seen in Figure 4.2.

```
/* l.addi $rD,$rA,$simm16 */
  {
    { 0, 0, 0, 0 },
    { { MNEM, ' ', OP (RD), ',', OP (RA), ',', OP (SIMM16), 0 } },
    & ifmt_l_lwz, { 0x9c000000 }
  },
```

Figure 4.1: or1k-opc.c before changes

```
/* l.addi $rD,$rA,$simm16 */
  {
    { 0, 0, 0, 0 },
    { { MNEM, ' ', OP (RD), ',', OP (RA), ',', OP (SIMM16), 0 } },
    & ifmt_l_lwz, { 0x0c000000 }
  },
```

Figure 4.2: or1k-opc.c after changes

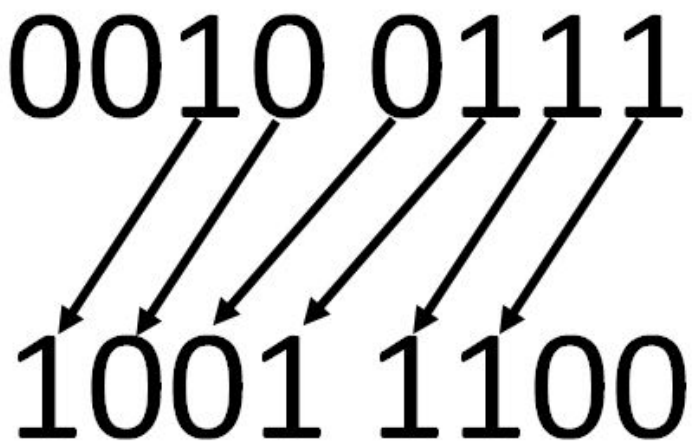

Figure 4.3: Bits (0x27) shifted to most significant 6 bits to make 0x9C

4.2.2 Resulting Output

The resulting output is a disassembly file with machine code that matches the instruction word format of the DMF RISC processor. This was obtained and can be seen the lines below in Figure 4.5 with addi instructions. When compared to Figure 4.4, its clear that the opcode for the addi instruction has been changed successfully. This was the only opcode change implemented, so this is the only opcode that will have a difference between Figure 4.4 and 4.5.

```
1
2   hello-world.elf:     file format elf32-or1k
3
4
5   Disassembly of section .text:
6
7   00000000 <main>:
8      0: 9c 21 ff f4   l.addi r1,r1,-12
9      4: d4 01 10 04   l.sw 4(r1),r2
10     8: 9c 41 00 0c   l.addi r2,r1,12
11     c: d4 01 48 08   l.sw 8(r1),r9
12    10: aa 20 00 05   l.ori r17,r0,0x5
13    14: d7 e2 8f f4   l.sw -12(r2),r17
14    18: 86 22 ff f4   l.lwz r17,-12(r2)
15    1c: 9e 31 00 01   l.addi r17,r17,1
16    20: d7 e2 8f f4   l.sw -12(r2),r17
17    24: 15 00 00 00   l.nop 0x0
18    28: e1 71 88 04   l.or r11,r17,r17
19    2c: 84 41 00 04   l.lwz r2,4(r1)
20    30: 85 21 00 08   l.lwz r9,8(r1)
21    34: 9c 21 00 0c   l.addi r1,r1,12
22    38: 44 00 48 00   l.jr r9
23    3c: 15 00 00 00   l.nop 0x0
```

Figure 4.4: Disassembly of hello-world.c for OR1k processor

```
1
2   hello-world.elf:     file format elf32-or1k
3
4
5   Disassembly of section .text:
6
7   00000000 <main>:
8      0: 0c 21 ff f4   l.addi r1, r1, -12
9      4: 00 01 10 04   l.jsw 4(r1), r2
10     8: 0c 41 00 0c   l.addi r2, r1, 12
11     c: 00 01 48 08   l.sw 8(r1), r9
12    10: 0e 22 00 05   l.addi r17, r2, 5
13    14: 03 e2 8f f4   l.sw -12(r2), r17
14    18: 02 22 ff f4   l.lwz r16, -12(r2)
15    1c: 0e 31 00 01   l.addi r17, r17, 1
16    20: 03 e2 8f f4   l.sw -12(r2), r17
17    24: 00 00 00 00   l.nop 0x0
18    28: 51 71 88 00   l.or r11,r17,r17
19    2c: 00 41 00 04   l.lwz r2, 4(r1)
20    30: 01 21 00 08   l.lwz r9, 8(r1)
21    34: 0c 21 00 0c   l.baddi r1, r1, 12
22    38: 00 00 48 00   l.jr r9
23    3c: 00 00 00 00   l.nop 0x0
```

Figure 4.5: Disassembly of hello-world.c for DMF RISC processor

4.2.3 Changes Not Implemented

A full scale port to the back end of GCC was not implemented for this project. A full scale port requires much more time and a deeper understanding of the GCC back end, the ABI of the processor/GCC, and much more. This, however, would be how the DMF RISC processor would be able to run any C code compiled to it, without having to fix the instruction word structure of the DMF RISC processor. Additionally, had the opcode porting with the OR1k processor worked, it would have only been with certain simple instructions, such as the manipulation and data transfer instructions. Most likely no instructions that accessed memory would have been functioning, as well as any flow control instructions, like jumps and calls. These would all require a bit more manipulation to the OR1k files and the instruction word formats to match them to the DMF RISC processor (or vice versa). At that point, doing a full port of the back end of GCC to the DMF

RISC processor would make for a more simple fully functioning processor to compiled C code.

Chapter 5

Tests and Results

This chapter discusses the results of the compiled C code run on the DMF RISC Processor.

5.1 Test of Basic Program

A program titled hello-world.c was used to test the process of compiling to the DMF RISC processor. Hello-world.c was a very short and simple program, and can be seen in Program Listing I.11. The program simply initializes an int variable (x), sets it equal to 5 and then increments it by 1. This will be a simple enough program to see some instructions be used, but not too complicated for GCC to generate assembly instructions not supported by the DMF RISC Processor.

5.1.1 Compiled Using GCC

The C code was compiled, and the new opcodes were able to be seen in the machine code that was run in the DMF processor. The machine code that was output from compiling hello-world.c can be seen in 4.5. This assembly code was inputted into the mif file that the DMF RISC Processor uses to initialize the program memory, and the program was tested. The output was successful,

as can be seen in 5.1.

5.1.2 Run in Processor

Register 17 is the one the OR1k back end of GCC decided to use for the variable declared in the hello-world.c program. It can be seen after the program counter (PC) switches to 0x8, the output of R17 becomes 5, which is the reflection of the declared integer x being initialized to 5. Then, a few instructions later, when the PC switches to 0xB, the value in register 17 switches to 6, a reflection of integer x being incremented by 1.

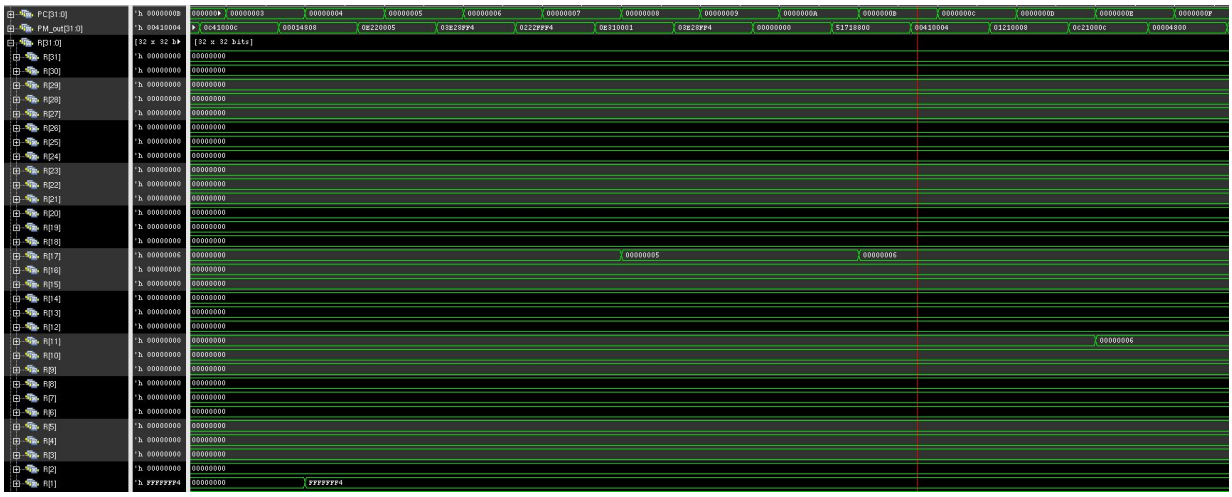

Figure 5.1: Output waveform for hello-world.c run on DMF RISC Processor

5.1.3 Challenges of using OR1k Back end

There were a few challenges to using the back end of a processor that was not designed exactly the same as the DMF RISC Processor. The issues are as follows:

1. The supported instructions are not the same. The OR1k has instructions that the DMF RISC Processor does not support, and vice versa. Therefore, when the GCC compiles the c code into assembly for the OR1k, all functionality can not necessarily be ported over

to the DMF RISC Processor. This is one reason why it may be more beneficial to do a complete back end port of GCC rather than simply an opcode port. The solution to this was instructions that were unsupported by the DMF RISC Processor were given opcodes of 0 (equivalent to a nop in the DMF RISC) so that it would not be mistaken for a different instruction. These instructions were unused or irrelevantly used in the assembly code generated for hello-world.c

2. Originally, the DMF RISC Processor was a 2-operand processor (destination = destination (operand) source) while the OR1k is a 3 operand (destination = source (operand) source). The DMF RISC Processor had to be manipulated a bit to make into a 3-operand processor to match the OR1k assembly code GCC outputted. It was too difficult to change the OR1k back end to support the 2-operand instructions.

3. It was very tedious and time-consuming to find the part of the back end where the opcodes needed to be changed. Eventually, they were found in or1k-opc.c, but the way the opcodes were hard coded in made them difficult to find. As talked about in section 4.2.1, the opcodes were pre-shifted to be in the most significant 6 bits of the instruction word. The opcodes would have been much easier to find if they were put into the code as is, and GCC shifted them into the most significant 6 bit positions. As much work as porting the entire back end of GCC to a new processor would have been, it might have taken a similar amount of time to figuring out the OR1k back end enough to alter the opcodes to match the DMF RISC Processor.

4. There is no control over how GCC converts the C code to assembly. There were so many extra unnecessary instructions generated by the GCC compilation process that might be necessary for the OR1k, but weren't for the much simpler DMF RISC. Luckily, these extra instructions were able to be ignored in the DMF RISC, but it takes up extra program

memory space to have unnecessary instructions, which could be eliminated by doing a full port to the back end of GCC.

Chapter 6

Conclusions

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

6.1 Future Work

In the future, the processor could be more fully integrated with the GCC compiler by improving two key things:

1. The instructions in the processor, and maybe the structure of their instruction words
2. The number of instructions/opcodes compatible with the GCC compiler

6.1.1 Processor Improvements

Since much of the time spent on this processor was to convert it from a 14-bit (original requirements from a class) to a 32-bit, extra critical instructions were not added to create a complete instruction set. Instructions such as a move instruction, as well as more data transfer instructions

that would add and remove items from stack (typically push and pop) were not implemented in this processor, but are used by the GNU Assembler.

Additionally, the processor could be converted to allow for three operands, a destination and two sources. Most processors operate like this nowadays, however, this processor was a very basic RISC design. It is much easier to port a processor to C, however, if it is capable of three operands.

Other components that would improve the processor functionality would be creating different features, as talked about in Section 2.1.2, such as out-of-order execution and branch predictions. Since this project was mostly to prove the availability of nearly any processor to be ported to GCC, these extra tasks were unnecessary, but could be implemented for future improvements.

6.1.2 More Compatible Instruction Words

The DMF RISC processor was not intended to be directly ported to GCC. Since instead the OR1k model was used and the simple binary of the opcodes were to be changed to be compatible with opcodes in the DMF RISC processor, the port was not done to its full potential. Using a tool called CGEN, new processors can be entirely ported to GCC for any C code to compile to cleanly. In the future, this would be another main goal, to create all the additional files needed (typically a list of 7/8 files) to completely port the back end of GCC to the DMF RISC processor. This would most likely be an easier task if the aforementioned changes to the processor were first made.

6.2 Project Conclusions

Unfortunately, due to the development environment, the port of the DMF RISC processor to the back end of GCC through the OR1k files was unsuccessful. That does not mean, however, that

with more work, this could not be done. Even still, processors and compilers were explored in Chapters 2 and 3, and a deeper understanding of their workings was obtained. Every project attempt will not be successful, and sometimes it is the failures we learn most from. In the end, a solid processor was designed and redesigned, and a GCC back end was altered to what should have made the machine code compatible. Much was learned from this experience, and that is the main goal of any academic project.

References

[1] Douglas W. Jones. The Ultimate RISC. *SIGARCH Comput. Archit. News*, 16(3):48, June 1988. doi:10.1145/48675.48683.

[2] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[3] Johan Bjareholt. RISC-V Compiler Performance:A Comparison between GCC and LLVM/clang. Master's thesis, Blekinge Institute of Technology, 2017.

[4] Connor Jan Goldberg. The Design of a Custom 32-bit RISC CPU and LLVM Compiler Backend. Master's thesis, Rochester Institute of Technology, August 2017.

[5] D. Bokan, M. Dukic, M. Popovic, and N. Cetic. Adjustment of GCC compiler frontend for embedded processors. In *Proc. 22nd Telecommunications Forum Telfor (TELFOR)*, pages 983–986, 2014.

[6] N. B. Jensen, P. Schleuniger, A. Hindborg, M. Walter, and S. Karlsson. Experiences with Compiler Support for Processors with Exposed Pipelines. In *Proc. IEEE Int. Parallel and Distributed Processing Symp. Workshop*, pages 137–143, 2015.

[7] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis,

Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.

[8] J. Kim, S. Lee, S. Moon, and S. Kim. Comparison of LLVM and GCC on the ARM Platform. In *Proc. 5th Int. Conf. Embedded and Multimedia Computing*, pages 1–6, 2010.

[9] H. El-Aawar. An application of complexity measures in addressing modes for CISC- and RISC-architectures. In *Proc. IEEE Int. Conf. Industrial Technology*, pages 1–7, 2008.

[10] Vincent P. Heuring and Harry F. Jordan. *Computer Systems Design and Architecture (2nd Edition)*. Prentice-Hall, Inc., USA, 2003.

[11] R. Stallman. The Free Software Movement and the GNU/Linux Operating System. In *Proc. 22nd IEEE Int. Conf. Software Maintenance*, page 426, 2006.

[12] E. Blem, J. Menon, and K. Sankaralingam. Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures. In *Proc. IEEE 19th Int. Symp. High Performance Computer Architecture (HPCA)*, pages 1–12, 2013.

[13] Mark Brehob, Travis E. Doom, Richard J. Enbody, William Howard Moore, Sherry Moore, Ron Sass, and Charles R. Severance. Beyond RISC-The Post-RISC Architecture Submitted to : IEEE Micro 3 / 96. In *IEEE Micro*, 2007.

[14] T. Jamil. RISC versus CISC. *IEEE Potentials*, 14(3):13–16, 1995.

[15] William H. Stallings. Reduced instruction set computer architecture. In *Proceedings of the IEEE*, volume 76, page 38, January 1988.

[16] William C. Hopkins. HLLDA Defies RISC: Thoughts on RISCs, CISCs, and HLLDAs. *SIGMICRO Newsl.*, 14(4):70, December 1983. URL: `https://doi-org.ezproxy.rit.edu/10.1145/1096419.1096430`, `doi:10.1145/1096419.1096430`.

[17] A. Akram and L. Sawalha. A Study of Performance and Power Consumption Differences Among Different ISAs. In *Proc. 22nd Euromicro Conf. Digital System Design (DSD)*, pages 628–632, 2019.

[18] A. Raveendran, V. B. Patil, D. Selvakumar, and V. Desalphine. A RISC-V instruction set processor-micro-architecture design and analysis. In *Proc. Technology and Applications (VLSI-SATA) 2016 Int. Conf. VLSI Systems, Architectures*, pages 1–7, 2016.

[19] A. Kim, S. J. Hwang, and S. W. Kim. Effective Instruction Fetch Stage Design for 16-Bit Instruction Set Architecture. In *Proc. IEEE 8th Int. Conf. Computer and Information Technology Workshops*, pages 563–568, 2008.

[20] C. Park, M. Han, H. Lee, and S. W. Kim. Performance comparison of GCC and LLVM on the EISC processor. In *Proc. Information and Communications (ICEIC) 2014 Int. Conf. Electronics*, pages 1–2, 2014.

[21] L. Ghica and N. Tapus. Optimized retargetable compiler for embedded processors - GCC vs LLVM. In *Proc. IEEE Int. Conf. Intelligent Computer Communication and Processing (ICCP)*, pages 103–108, 2015.

[22] Bill Gatliff. Embedding with GNU: The GNU Compiler and Linker. *Embedded Systems Programming*, page 66, February 2000.

[23] William von Hagen. *The Definitive Guide to GCC, Second Edition (Definitive Guide)*. Apress, USA, 2006.

[24] J. Alves, M. Held, and M. Glesner. A code generator for an application specific pipelined processor. In *Proc. MELECON '94. Mediterranean Electrotechnical Conf*, pages 306–308 vol.1, 1994.

[25] John Wood and Harold C. Grossman. Interprocedural register allocation for risc machines. In *Proceedings of the 30th Annual Southeast Regional Conference*, ACM-SE 30, pages 188–195, New York, NY, USA, 1992. Association for Computing Machinery. URL: https://doi-org.ezproxy.rit.edu/10.1145/503720.503796, doi:10.1145/503720.503796.

[26] Seung Pyo Jung, Jingzhe Xu, Donghoon Lee, Ju Sung Park, Kang joo Kim, and Koon shik Cho. Design verification of 16 bit RISC processor. In *Proc. Int. SoC Design Conf*, volume 03, pages 13–14, 2008.

[27] K. Yi and Y. Ding. 32-bit RISC CPU Based on MIPS Instruction Fetch Module Design. In *2009 International Joint Conference on Artificial Intelligence*, page 754, 2009.

Appendix I

Source Code

I.1 DMF RISC Verilog Code Main File

```
1  module dmf_RISC621_cache_v (Resetn_pin, Clock_pin, SW_pin,
       Display_pin);
2  //
       ------------------------------------------------------------------------

3  // The code has been only tested for the instruction sequence
       currently
4  //    loaded in rom1.  Additional logic may be required for
       other instrcution
5  //    sequences. DxP October 2014.
6  //
       ------------------------------------------------------------------------
```

```
7    input Resetn_pin, Clock_pin;
8    input [4:0] SW_pin;      // Four switches and one push-button
9    output [7:0] Display_pin; // 8 LEDs
10 //
     ------------------------------------------------------------------

11 //-- Declare machine cycle and instruction cycle parameters
12 //
     ------------------------------------------------------------------

13   parameter [21:0] ADD_IC=22'b01, SUB_IC=22'b10, ADDC_IC=22'b11
       ,
14     SUBC_IC =22'b100, ROTR_IC=22'b101, MUL_IC=22'b1000, DIV_IC
         =22'b1001,
15     XOR_IC=22'b001010, SHRL_IC = 22'b001011, ROTL_IC=22'b001100
         , RLN_IC = 22'b001101,
16     RLZ_IC = 22'b001110, RRN_IC=22'b001111, RRZ_IC = 22'b010000
         , NOT_IC = 22'b010001,
17     SHRA_IC = 22'b010010, AND_IC = 22'b010011, OR_IC = 22'
         b010100, LD_IC = 22'b010101,
18     ST_IC = 22'b010110, CPY_IC = 22'b010111, SWAP_IC = 22'
         b011000, JMP_IC = 22'b011001,
19     CALL_IC = 22'b011010, RET_IC = 22'b011011;
20
```

```
21    parameter [3:0] JU = 4'b0000, JC1 = 4'b1000, JN1 = 4'b0100,
          JV1 = 4'b0010,
22        JZ1 = 4'b0001, JC0 = 4'b0111, JN0 = 4'b1011, JV0 = 4'b1101,
              JZ0 = 4'b1110;
23  //
      ------------------------------------------------------------------------

24  //-- Declare internal signals
25  //
      ------------------------------------------------------------------------

26    reg [31:0] R [31:0];
27    reg WR_DM, stall_mc0, stall_mc1, stall_mc2, stall_mc3;
28    reg [31:0] PC, IR3, IR2, IR1, MAB, MAX, MAeff, ADeff, SP,
          DM_in, IPDR;
29    reg [31:0] TA, TB, TALUH, TALUL, quotient, remainder, TBST;
30    reg [11:0] TSR, SR;
31    reg [7:0] Display_pin;
32    reg [32:0]  TALUout;
33    wire [31:0] PM_out, DM_out;
34    wire  Done_DM, Done_PM;
35    wire       C, Clock_not;
36    reg [63:0] product;
37    integer Ri1, Rj1, Ri2, Rj2, Ri3, Rj3, Rd1, Rd2, Rd3;
```

```
38 //
     ----------------------------------------------------------------------

39 // In this architecture we are using a combination of
     structural and
40 //  behavioral code.  Care has to be exercised because the
     values assigned
41 //    in the process are visible outside of it only during the
     next clock
42 //    cycle.  The CPU comprised of the DP and CU is modelled as
      a combination
43 //  of CASE and IF statements (behavioral).  The memories are
     called within
44 //  the structural part of the code.  We could model the
     memories as
45 //    arrays, but that would result in less than optimal memory
46 //    implementations.  Also, later on we will want to add an
     hierarchical
47 //    memory subsystem.
48 //
     ----------------------------------------------------------------------

49 // Structural section of the code.  The order of the
     assignments doesn't
50 //  matter.  Concurrency!
```

```
51 //
     ----------------------------------------------------------------------

52       assign   Clock_not = ~Clock_pin;
53
54       dmf_DM_system_v DM       (Resetn_pin, Clock_pin, WR_DM,
             MAeff, DM_in, DM_out, Done_DM);
55       dmf_PM_system_v   PM       (Resetn_pin, Clock_pin, PC,
             PM_out, Done_PM);
56
57 //
     ----------------------------------------------------------------------

58 //  Behavioral section of the code.  Assignments are evaluated
     in order, i.e.
59 //  sequentially. New assigned values are visible outside the
     always block
60 //  only after it is exit.  Last assigned value will be the
     exit value.
61 //
     ----------------------------------------------------------------------

62       always@(posedge Clock_not)
63 //
     ----------------------------------------------------------------------
```

```
64 // The reset is active low and clock synchronous.  For
       verification/simulation
65 //  purposes it is necessary in this case to initialize the
       value of some
66 //    registers.
67 //
       ------------------------------------------------------------------

68   if (Resetn_pin == 1)
69     begin
70       PC = 32'h00000000;
71       R[0] = 0; R[1] = 0; R[2] = 0; R[3] = 0; R[4] = 0;
72       R[5] = 0; R[6] = 0; R[7] = 0; R[8] = 0; R[9] = 0;
73       R[10] = 0; R[11] = 0; R[12] = 0; R[13] = 0; R[14] = 0;
74       R[15] = 0; R[16] = 0; R[17] = 0; R[18] = 0; R[19] = 0;
75       R[20] = 0; R[21] = 0; R[22] = 0; R[23] = 0; R[24] = 0;
76       R[25] = 0; R[26] = 0; R[27] = 0; R[28] = 0; R[29] = 0;
77       R[30] = 0; R[31] = 0;
78       // Necessary for sim
79 // The initialization of the stall_mc signals is necessary for
       the correct
80 //  startup of the pipeline.
81       stall_mc0 = 0; stall_mc1 = 1; stall_mc2 = 1; stall_mc3 =
           1;
```

```
82  // All IRs are initialized to the "don't care OpCode value 0
        xffff
83          IR1 = 32'h00000000; IR2 = 32'h00000000; IR3 = 32'
                h00000000;
84          TALUout = 0;
85          TALUH = 0;
86          TALUL = 0;
87          TSR = 0;
88          SR = 0;
89          product = 0;
90          TA = 0;
91          TB = 0;
92          Display_pin = 0;
93          Ri1 = 0; Ri2 = 0; Ri3 = 0;
94          Rj1 = 0; Rj2 = 0; Rj3 = 0;
95          Rd1 = 0; Rd2 = 0; Rd3 = 0;
96          SP = 32'h00003FF0; //may have to change
97          WR_DM = 1'b0;
98          //MAeff = 14'b0;
99          MAB = 32'b00000000;
100         MAX = 32'b00000000;
101       end
102     else if (Done_PM == 1'b0 || Done_DM == 1'b0) begin
103       SR = SR;
104       IR1 = IR1;
```

```
105         IR2 = IR2;
106         IR3 = IR3;
107         TA = TA;
108         TB = TB;
109         Ri1 = Ri1;
110         Rj1 = Rj1;
111         Ri2 = Ri2;
112         Rj2 = Rj2;
113         Ri3 = Ri3;
114         Rj3 = Rj3;
115         Rd1 = Rd1;
116         Rd2 = Rd2;
117         Rd3 = Rd3;
118       end
119       else  begin
120       WR_DM = 1'b0;
121   //
        ------------------------------------------------------------------------

122   // MC3 is executed first because its assignments might be
        needed by MC2 or MC1
123   //     to resolve data or control D/H.
124   //
        ------------------------------------------------------------------------
```

```
125   if (stall_mc3 == 0) begin case (IR3[31:26])
126 //
      ------------------------------------------------------------------------

127     LD_IC:  begin
128       if (MAeff[31:22] == 10'h3FF)
129         if (MAeff[21:0] == 22'hF) R[Rd3] = SP;
130         else R[Rd3] = {27'b000000000, SW_pin};

131       else R[Rd3] = DM_out; end
132     ST_IC:  begin
133       WR_DM = 1'b0;
134       if (MAeff[31:22] == 10'h3FF)
135         if (MAeff[21:0] == 22'hF) SP = TBST;
136         else Display_pin = TBST[7:0];
137       else ; end
138     CPY_IC: begin R[Rd3] = TALUL; end
139     SWAP_IC:  begin
140       R[Rd3] = TALUL;
141       R[Ri3] = TALUH; end
142     JMP_IC: begin case (IR3[3:0])
143       JU :  begin PC = ADeff; end
144       JC1:  begin
145         if (SR[11] == 1) PC = ADeff;
146         else PC = PC; end
```

```
147         JN1:   begin
148           if (SR[10] == 1) PC = ADeff;
149           else PC = PC; end
150         JV1:   begin
151           if (SR[9] == 1) PC = ADeff;
152           else PC = PC; end
153         JZ1:   begin
154           if (SR[8] == 1) PC = ADeff;
155           else PC = PC; end
156         JC0:   begin
157           if (SR[11] == 0) PC = ADeff;
158           else PC = PC; end
159         JN0:   begin
160           if (SR[10] == 0) PC = ADeff;
161           else PC = PC; end
162         JV0:   begin
163           if (SR[9] == 0) PC = ADeff;
164           else PC = PC; end
165         JZ0:   begin
166           if (SR[8] == 0) PC = ADeff;
167           else PC = PC; end
168         endcase
169         stall_mc0 = 1; end
170       CALL_IC: begin
171         MAeff = SP;
```

```
172         WR_DM = 1'b1;
173         DM_in = {{2{SR[11]}}, SR};
174         PC = MAB + MAX;
175         stall_mc0 = 1; end
176         //SP = SP - 1'b1; end
177         //stall_mc3 = 1; end
178       RET_IC: begin
179         PC = DM_out;
180         stall_mc0 = 1; end
181
182       MUL_IC, DIV_IC:  begin
183         R[Rd3] = TALUH;
184         R[Ri3] = TALUL;
185         SR = TSR;
186         Display_pin = R[IR3[3:0]][7:0]; end
187
188       ADD_IC, SUB_IC, ADDC_IC, SUBC_IC, NOT_IC, AND_IC, OR_IC,
              SHRA_IC,
189       ROTR_IC, ROTL_IC, XOR_IC, SHRL_IC, RRN_IC, RRZ_IC, RLN_IC,
              RLZ_IC:
190       begin
191         R[Rd3] = TALUH;
192         SR = TSR;
193         Display_pin = R[IR3[3:0]][7:0]; end
194
```

```
195       default:   ; endcase end
196   //
      ------------------------------------------------------------------------

197     if (stall_mc2 == 0) begin case (IR2[31:26])
198   //
      ------------------------------------------------------------------------

199       JMP_IC: begin
200         ADeff = MAB + MAX;
201         WR_DM = 1'b0;
202         stall_mc0 = 1; end
203       LD_IC: begin
204         MAeff = MAB + MAX;
205         WR_DM = 1'b0;
206         stall_mc0 = 0; end
207       ST_IC:   begin MAeff = MAB + MAX;
208         if (MAeff[31:26] != 10'h3FF)  begin
209           WR_DM = 1'b1;
210           DM_in = TB;
211           TBST = TB; end
212         else WR_DM = 1'b0;
213         stall_mc0 = 0; end
214       CPY_IC: begin TALUL = TB;
215         if(TALUL == 32'b0) TSR[8] = 1;
```

```
216         else TSR[8] = 0; end
217       SWAP_IC: begin TALUH = TA; TALUL = TB; end
218       CALL_IC: begin
219         WR_DM = 1'b1;
220         MAeff = SP;
221         DM_in = PC;
222         SP = SP - 1'b1;
223         stall_mc0 = 1; end
224         //stall_mc2 = 1; end
225       RET_IC: begin
226         WR_DM = 1'b0;
227         MAeff = SP;
228         //PC = DM_out;
229         SP = SP + 1'b1;
230         stall_mc0 = 1; end
231 //
      ------------------------------------------------------------------------

232 // For all assignments that target TALUH we use TALUout.  This
      is 15-bits wide
233 //  to account for the value of the carry when necessary.
234 //
      ------------------------------------------------------------------------

235       ADD_IC, ADDC_IC:  begin
```

```
236         TALUout = TA + TB;
237         TSR[11] = TALUout[14]; // Carry
238         TSR[10] = TALUout[13]; // Negative
239         TSR[9] = ((TA[13] ~^ TB[13]) & TA[13]) ^ (TALUout[13] & (
                TA[13] ~^ TB[13])); // V Overflow
240         if (TALUout[31:0] == 32'h0000) TSR[8] = 1;   // Zero
241         else TSR[8] = 0; TALUH = TALUout[31:0];   end
242       SUB_IC, SUBC_IC:   begin
243         TALUout = TA - TB;
244         TSR[11] = TALUout[14]; // Carry
245         TSR[10] = TALUout[13]; // Negative
246         TSR[9] = ((TA[13] ~^ TB[13]) & TA[13]) ^ (TALUout[13] & (
                TA[13] ~^ TB[13])); // V Overflow
247         if (TALUout[31:0] == 32'h0000) TSR[8] = 1;   // Zero
248         else TSR[8] = 0; TALUH = TALUout[31:0];   end
249       MUL_IC:   begin
250         product = TA * TB;
251         TALUH = product[63:32];
252         TALUL = product[31:0];
253         TALUout = {TALUH, TALUL[31]};
254         if (product == 0) TSR[8] = 1; //zero
255         else TSR[8] = 0;
256         TSR[10] = product[63]; end //negative
257       DIV_IC:   begin
258         quotient = TA / TB; remainder = TA % TB;
```

```
259         TALUH = quotient;
260         TALUL = remainder;
261         if (quotient == 0)  TSR[8] = 1; //zero
262         else TSR[8] = 0;
263         TSR[10] = quotient[31]; end //negative
264       NOT_IC: begin
265         TALUout = ~TA;
266         TALUH = TALUout[31:0];
267         if (TALUH[31:0] == 32'h00000000)  TSR[8] = 1; //zero
268         else TSR[8] = 0;
269         TSR[10] = TALUH[31]; end //negative
270       AND_IC: begin
271         TALUout = TA & TB;
272         TALUH = TALUout[31:0];
273         TSR[10] = TALUH[31]; // Negative
274         if (TALUH[31:0] == 32'h00000000) TSR[8] = 1;  // Zero
275         else TSR[8] = 0; end
276       OR_IC:  begin
277         TALUout = TA | TB;
278         TALUH = TALUout[31:0];
279         TSR[10] = TALUH[31]; // Negative
280         if (TALUH[31:0] == 32'h00000000) TSR[8] = 1;  // Zero
281         else TSR[8] = 0; end
282       XOR_IC: begin
283         TALUout = TA ^ TB;
```

```
284         TALUH = TALUout [31:0];
285         TSR[10] = TALUH[31]; // Negative
286         if (TALUH[31:0] == 32'h00000000) TSR[8] = 1;  // Zero
287         else TSR[8] = 0; end
288       SHRL_IC: begin //shift right logic
289         case (IR2[1:0])
290           2'b00: begin TALUH = TA; end
291           2'b01: begin TALUH[31]=1'b0; TALUH[30:0]=TA[31:1]; end
292           2'b10: begin TALUH[31:30]=2'b0; TALUH[29:0]=TA[31:2];
                end
293           2'b11: begin TALUH[31:29]=3'b0; TALUH[28:0]=TA[31:3];
                end endcase end
294       SHRA_IC:  begin
295         case (IR2[1:0])
296           2'b00:  begin TALUH = TA; end
297           2'b01:  begin TALUH[31]=TA[31]; TALUH[30:0]=TA[31:1];
                end
298           2'b10:  begin
299             TALUH[31]=TA[31]; TALUH[30]=TA[31]; TALUH[29:0]=TA
                  [31:2]; end
300           2'b11:  begin TALUH[31]=TA[31]; TALUH[30]=TA[31];
301             TALUH[29]=TA[31]; TALUH[28:0]=TA[31:3]; end endcase
                end
302       ROTL_IC: begin
303         case (IR2[1:0])
```

```
304           2'b00: begin TALUH = TA; end
305           2'b01: begin TALUH[0] = TA[31]; TALUH[31:1]=TA[30:0];
                 end
306           2'b10: begin TALUH[0] = TA[31]; TALUH[1] = TA[30];
                 TALUH[31:2] = TA[29:0]; end
307           2'b11: begin TALUH[0] = TA[31]; TALUH[1] = TA[30];
                 TALUH[2] = TA[29];
308                TALUH[31:3] = TA[28:0]; end endcase end
309       ROTR_IC: begin //rotate right
310         case (IR2[1:0])
311           2'b00: begin TALUH = TA; end
312           2'b01: begin TALUH[31] = TA[0]; TALUH[30:0]=TA[31:1];
                 end
313           2'b10: begin TALUH[31] = TA[0]; TALUH[30] = TA[1];
                 TALUH[29:0] = TA[31:2]; end
314           2'b11: begin TALUH[31] = TA[0]; TALUH[30] = TA[1];
                 TALUH[29] = TA[2];
315                TALUH[28:0] = TA[31:3]; end endcase end
316       RLN_IC: begin //roll left through N
317         case (IR2[1:0])
318           2'b00: begin TALUH = TA; end
319           2'b01: begin TALUH[0] = TSR[10]; TALUH[31:1] = TA
                 [30:0];
320                TSR[10] = TA[31]; end
```

```
321          2'b10: begin TALUH[0] = TA[31]; TALUH[1] = TSR[10];
                TALUH[31:2] = TA[29:0];
322                TSR[10] = TA[30]; end
323          2'b11: begin TALUH[0] = TA[30]; TALUH[1] = TA[31];
                TALUH[2] = TSR[10];
324                TALUH[31:3] = TA[28:0]; TSR[10] = TA[29]; end
                      endcase end
325       RLZ_IC: begin //roll left through Z
326         case (IR2[1:0])
327          2'b00: begin TALUH = TA; end
328          2'b01: begin TALUH[0] = TSR[8]; TALUH[31:1] = TA[30:0];
                 TSR[8] = TA[31]; end
329          2'b10: begin TALUH[0] = TA[31]; TALUH[1] = TSR[8];
                TALUH[31:2] = TA[29:0];
330                TSR[8] = TA[30]; end
331          2'b11: begin TALUH[0] = TA[30]; TALUH[1] = TA[31];
                TALUH[2] = TSR[8];
332                TALUH[31:3] = TA[28:0]; TSR[8] = TA[29]; end
                      endcase end
333       RRN_IC: begin
334         case (IR2[1:0])
335          2'b00: begin TALUH = TA; end
336          2'b01: begin TALUH[31]=TSR[10]; TALUH[30:0]=TA[31:1];
                TSR[10] = TA[0]; end
```

```
337           2'b10: begin TALUH[31]=TA[0]; TALUH[30]=TSR[10]; TALUH
                  [29:0]=TA[31:2];
338                 TSR[10] = TA[1]; end
339           2'b11: begin TALUH[31]=TA[1]; TALUH[30]=TA[0]; TALUH
                  [29]=TSR[10];
340                 TALUH[28:0]=TA[31:3]; TSR[10] = TA[2]; end endcase
                        end
341       RRZ_IC: begin//roll right through Z
342         case (IR2[1:0])
343           2'b00: begin TALUH = TA; end
344           2'b01: begin TALUH[31]=TSR[8]; TALUH[30:0]=TA[31:1];
                  TSR[8] = TA[0]; end
345           2'b10: begin TALUH[31]=TA[0]; TALUH[30]=TSR[8];
346                 TALUH[29:0]=TA[31:2]; TSR[8] = TA[1]; end
347           2'b11: begin TALUH[31]=TA[1]; TALUH[30]=TA[0]; TALUH
                  [29]=TSR[8];
348                 TALUH[28:0]=TA[31:3]; TSR[8] = TA[2]; end endcase
                     end
349
350       default: ;  endcase end
351   //
      ------------------------------------------------------------------------


352     if (stall_mc1 == 0) begin    case (IR1[31:26])
```

```
353  //
       ------------------------------------------------------------------------

354       LD_IC, JMP_IC:  begin //think this is done
355         MAB = PM_out;//unsure
356         if (Ri1 == 0) MAX = 0;
357         else if (Ri1 == 1) MAX = PC;
358         else if (Ri1 == 2) MAX = SP;
359         else begin
360           if (Rd2 == Ri1) MAX = TALUH;
361           else if ((IR2[31:27] == 5'b00100||IR2[31:26] == 6'
                b011000) && Ri2 == Ri1) MAX = TALUL;
362           else MAX = R[Ri1]; end
363         PC = PC + 1'b1;
364         stall_mc0 = 1; end //maybe
365       CALL_IC: begin
366         MAB = PM_out;//unsure
367         if (Ri1 == 0) MAX = 0;
368         else if (Ri1 == 1) MAX = PC;
369         else if (Ri1 == 2) MAX = SP;
370         else begin
371           if (Rd2 == Ri1) MAX = TALUH;
372           else if ((IR2[31:27] == 5'b00100||IR2[31:26] == 6'
                b011000) && Ri2 == Ri1) MAX = TALUL;
373           else MAX = R[Ri1]; end
```

```
374         PC = PC + 1'b1;
375         stall_mc0 = 1; //maybe
376         SP = SP - 1'b1; end
377       ST_IC: begin
378         MAB = PM_out;//unsure
379         if (Ri1 == 0) MAX = 0;
380         else if (Ri1 == 1) MAX = PC;
381         else if (Ri1 == 2) MAX = SP;
382         else begin
383           if (Rd2 == Ri1) MAX = TALUH;
384           else if ((IR2[31:27] == 5'b00100||IR2[31:26] == 6'
                b011000) && Ri2 == Ri1) MAX = TALUL;
385           else MAX = R[Ri1]; end
386         PC = PC+ 1'b1;
387         stall_mc0 = 1; //maybe
388         if (Rd2 == Rj1) TB = TALUH;
389         else if ((IR2[31:27] == 5'b00100||IR2[31:26] == 6'b011000
              ) && Ri2 == Rj1) TB = TALUL;
390         else
391         TB = R[Rj1];
392         TA = 4'b0; end
393       RET_IC: begin
394         MAeff = SP;
395         SR = DM_out[3:0];
396         SP = SP + 1'b1;
```

```
397         stall_mc0 = 1; end
398       CPY_IC: begin
399         if (Rd2 == Ri1) TB = TALUH;
400         else if ((IR2[31:27] == 5'b00100||IR2[31:26] == 6'b011000
              ) && Ri2 == Ri1) TB = TALUL;
401         else TB = R[Ri1];
402         TA = 4'b0; end
403       NOT_IC: begin
404         if (Rd2 == Ri1) TA = TALUH;
405         else if ((IR2[31:27] == 5'b00100||IR2[31:26] == 6'b011000
              ) && Ri2 == Ri1) TA = TALUL;
406         else TA = R[Ri1];
407         TB = 4'b0; end
408       SHRA_IC, ROTR_IC, RRN_IC, RRZ_IC, RLN_IC, RLZ_IC, SHRL_IC,
           ROTL_IC: begin
409         if (Rd2 == Ri1) begin TA = TALUH; TB = IR1[1:0]; end
410         else if ((IR2[31:27] == 5'b00100||IR2[31:26] == 6'b011000
              ) && Ri2 == Ri1) begin TA = TALUL; TB = IR1[1:0]; end
411         else begin TA = R[Ri1]; TB = IR1[1:0]; end end
412       ADDC_IC, SUBC_IC: begin
413         if (Rd2 == Ri1) TA = TALUH;
414         else if ((IR2[31:27] == 5'b00100||IR2[31:26] == 6'b011000
              ) && Ri2 == Ri1) TA = TALUL;
415         else TA = R[Ri1];
416         TB = {16'b0, IR1[15:0]}; end
```

```
417       default: begin
418       // ADD_IC, SUB_IC, AND_IC, OR_IC, MUL_IC, DIV_IC, XOR_IC,
             SWAP_IC:
419         if ((IR2[31:27] == 5'b00100||IR2[31:26] == 6'b011000) &&
               ((Rd2 == Ri1 && Rd2 == Rj1) || (Rd2 == Ri1 && Rd2 ==
               Rj1))) begin TA = TALUH; TB = TALUL; end
420         else if (Rd2 == Ri1 && Rd2 == Rj1) begin TA = TALUH; TB =
               TALUH;end
421         else if (Rd2 == Ri1) begin TA = TALUH; TB = R[Rj1]; end
422         else if (Rd2 == Rj1) begin TA = R[Ri1]; TB = TALUH; end
423         else if ((IR2[31:27] == 5'b00100||IR2[31:26] == 6'b011000
               ) && Ri2 == Ri1) begin TA = TALUL; TB = R[Rj1]; end
424         else if ((IR2[31:27] == 5'b00100||IR2[31:26] == 6'b011000
               ) && Rd2 == Rj1) begin TA = R[Ri1]; TB = TALUL; end
425         else begin TA = R[Ri1]; TB = R[Rj1]; end end
426       endcase
427     end
428   //
      ------------------------------------------------------------------

429   // The only data D/H that can occur are RAW.  These are
      automatically
430   //     resolved.  In the case of the JUMPS we stall until the
      adress of the
431   //     next instruction to be executed is known.
```

```
432 // The IR value 0xffff I call a "don't care" OpCode value.  It
    allows us to
433 //    control the refill of the pipe after the stalls of a jump
     emptied it.
434 //
    ------------------------------------------------------------------------


435     if (stall_mc2 == 0)
436       begin IR3 = IR2; Rd3 = Rd2; Ri3 = Ri2; Rj3 = Rj2;
             stall_mc3 = 0; end
437     else begin stall_mc3 =1; IR3 = 32'h00000000; Rd3 = 0; Ri3 =
          0; Rj3 = 0; end
438     if (stall_mc1 == 0) //&& IR2[13:8] != (JMP_IC || CALL_IC ||
          RET_IC)
439       begin IR2 = IR1; Rd2 = Rd1; Ri2 = Ri1; Rj2 = Rj1;
             stall_mc2 = 0; end
440     else begin stall_mc2 = 1; IR2 = 32'h00000000; Rd2 = 0; Ri2
        = 0; Rj2 = 0;  end
441     if (stall_mc0 == 0)
442       begin IR1 = PM_out; Rd1 = PM_out[25:21]; Ri1 = PM_out
             [20:16]; Rj1 = PM_out[15:11]; PC = PC + 1'b1; stall_mc1
              = 0; end
443     else begin stall_mc1 = 1; IR1 = 32'h00000000; Rd1 = 0; Ri1
        = 0; Rj1 = 0; end
444     if (IR3 == 32'h00000000)
```

```
445         begin stall_mc0 = 0; end
446 //
    ------------------------------------------------------------------------------


447 end
448 endmodule
```

Listing I.1: dmf$_R$*ISCProcessorMainCode*

I.2 DMF RISC Verilog Code Program Memory Cache Logic

```
1  module dmf_PM_system_v (Resetn, clock, Addr, out, Done);
2
3  input Resetn, clock;
4  input [31:0] Addr;
5  output reg Done;
6  output[31:0] out;
7
8  wire[25:0] TAG;
9  wire[1:0] group;
10 wire[3:0] word;
11 wire[25:0] dout_CAM_0, dout_CAM_1;
12 wire[25:0] din_CAM_0, din_CAM_1;
13 wire[3:0] mbits0, mbits1;
14 reg miss, cache_wren, wr_en_0, wr_en_1, rd_en_0, rd_en_1,
      process;
15 reg[6:0] cache_addr;
16 reg[31:0] PM_addr;
17 wire[31:0] cache_out, PM_out;
18 reg[3:0] replace;
19 reg[4:0] current_word;
20 wire c1, c2, c3;
21
22 assign TAG = Addr[31:6];
```

```
23 assign group = Addr[5:4];
24 assign word = Addr[3:0];
25
26 dmf_PLL_2        clock_pll_1   (clock, c1, c2, c3);
27
28 //assign PM_addr = writeback? PM_wr_addr : PM_rd_addr;
29 //assign PM_data = PM_wren? cache_out : 14'bz;
30
31 assign din_CAM_0 = wr_en_0 ? TAG : 26'bz;
32 assign din_CAM_1 = wr_en_1 ? TAG : 26'bz;
33
34
35 dmf_CAM_v        PM_CAM_0  (1'b1, wr_en_0, 1'b0, din_CAM_0, Addr
      [31:6], Addr[5:4], dout_CAM_0, mbits0);
36 dmf_CAM_v        PM_CAM_1  (1'b1, wr_en_1, 1'b0, din_CAM_1, Addr
      [31:6], Addr[5:4], dout_CAM_1, mbits1);
37 dmf_PM_v       PM_MM   (PM_addr, c2, PM_out);
38 dmf_PM_cache_v       PM_cache   (cache_addr, c3, PM_out,
      cache_wren, cache_out);
39 //dmf_2to4_decoder  DECODER   (group_in, match_out);
40
41 assign out = Done ? cache_out : 32'bz;
42
43 always@(posedge clock) begin
44   wr_en_0 = 1'b0;
```

```
45    wr_en_1 = 1'b0;
46    if (Resetn) begin
47      miss=1'b1; replace = 4'b0000;
48      Done = 1'b0; cache_wren = 0;
49       wr_en_0 = 1'b0; wr_en_1 = 1'b0;
50      current_word = 5'b00000; process = 0;
51    end
52    else begin
53 //     if we've been out of reset, need to check if its a miss
       or a hit
54      if (~process) begin
55        if(mbits0 == 4'b0000 && mbits1 == 4'b0000)
56          begin miss = 1'b1; Done = 1'b0; end
57
58        else begin miss = 1'b0; Done = 1'b0; end
59      end
60
61    //if it's a hit, we can just use the cache
62      if (miss == 1'b0) begin
63        process = 1;
64        cache_addr[5:4] = group; //cache address is 3 block bits
             and then 4 word bits
65
66        if (mbits0 != 4'b0000)begin //are we taking from Cache0?
67          cache_addr[6] = 1'b0;
```

```
68          replace[group] = 1'b0;
69          Done = 1;
70          process = 0;
71        end
72        else begin              //or Cache1?
73          cache_addr[6] = 1'b1;
74          replace[group] = 1'b1;
75          Done = 1;
76          process = 0;
77        end
78
79        cache_addr[3:0] = word;
80
81      end
82
83      //it's a miss, gotta replace
84      else begin
85        Done = 0;
86        process = 1;
87        cache_addr = {replace[group], group, current_word[3:0]};
88        PM_addr = {Addr[31:4], current_word[3:0]};
89        cache_wren = 1'b1;
90        current_word = current_word + 1'b1;
91
```

```
92          if (current_word == 5'b10001) begin //done with black
               transfer into cache
93            miss = 1'b0;
94            cache_wren = 1'b0;
95            current_word = 5'b00000;
96            if(replace[group]) begin //replacing things in CAM
97              wr_en_1 = 1'b1;
98            end
99            else begin
100             wr_en_0 = 1'b1;
101           end
102           //cache_addr={replace[group], group, word};
103         end
104       end
105     end
106 end
107 endmodule
```

Listing I.2: Program Memory Cache Logic

I.3 DMF RISC Verilog Code Data Memory Cache Logic

```
1  module dmf_DM_system_v (Resetn, clock, wr_en, Addr, in, out,
       Done);
2
3  input Resetn, clock, wr_en;
4  input [31:0] in, Addr;
5  output reg Done;
6  output[31:0] out;
7
8  wire[25:0] TAG;
9  wire[1:0] group;
10 wire[3:0] word;
11 wire [25:0] dout_CAM_0, dout_CAM_1;
12 wire[25:0] din_CAM_0, din_CAM_1;//gonna have to rethink this
       whole ass mf thing
13 wire [3:0] mbits0, mbits1;
14 reg miss, cache_wren, DM_wren, wr_en_0, wr_en_1, rd_en_0,
       rd_en_1, process, writeback;
15 reg [6:0] cache_addr;
16 reg[31:0] DM_rd_addr, DM_wr_addr;
17 wire[31:0] cache_out, DM_out;
18 wire [31:0] DM_addr, DM_data, cache_data;
19 reg [3:0] cam0_dirtybit, cam1_dirtybit, replace;
20 reg [4:0] current_word;
```

```
21 wire c1, c2, c3, DM_clk, cache_clk;
22
23 assign TAG = Addr[31:6];
24 assign group = Addr[5:4];
25 assign word = Addr[3:0];
26
27
28 dmf_PLL_2        clock_pll_1    (clock, c1, c2, c3);
29
30 assign DM_addr = writeback? DM_wr_addr : DM_rd_addr;
31 //assign DM_data = DM_wren? cache_out : 14'bz;
32 assign cache_data = (~miss && wr_en) ? in : DM_out;
33 assign DM_clk = writeback ? c2 : c3;
34 assign cache_clk = writeback ? c3 : c2;
35
36 assign din_CAM_0 = wr_en_0 ? TAG : 26'bz;
37 assign din_CAM_1 = wr_en_1 ? TAG : 26'bz;
38
39 dmf_CAM_v        DM_CAM_1   (1'b0, wr_en_0, 1'b1, din_CAM_0, Addr
      [31:6], Addr[5:4], dout_CAM_0, mbits0);
40 dmf_CAM_v        DM_CAM_2   (1'b0, wr_en_1, 1'b1, din_CAM_1, Addr
      [31:6], Addr[5:4], dout_CAM_1, mbits1);
41 dmf_DM_cache_v       DM_cache   (cache_addr, DM_clk, cache_data,
      cache_wren, cache_out);
```

```
42 dmf_DM_v       DM_MM   (DM_addr, cache_clk, cache_out, DM_wren,
       DM_out);
43 //dmf_2to4_decoder  DECODER   (group_in, match_out);
44
45 assign out = Done ? cache_out : 32'bz;
46
47 always@(posedge clock) begin
48   wr_en_0 = 1'b0;
49   wr_en_1 = 1'b0;
50   cache_wren = 1'b0;
51   if (Resetn) begin
52     miss = 1'b1; replace = 4'b0000;
53     Done = 1'b0; cache_wren = 0; DM_wren = 0;
54     cam0_dirtybit = 4'b0000; cam1_dirtybit = 4'b0000;
55      wr_en_0 = 1'b0; wr_en_1 = 1'b0;
56     current_word = 5'b00000; process = 0; writeback = 0;
57 //      din_CAM_0 = 8'bz; din_CAM_1 = 8'bz;
58 //     TAG_0 = 8'b0; TAG_1 = 8'b0; group_0 = 2'b0; group_1 = 2'
       b0;
59   end
60   else begin
61     //Done = 0;
62 //     if we've been out of reset, need to check if its a miss
       or a hit     DONT NEED
63 //     if (Addr == 14'bzzzzzzzzzzzzzz)
```

```
64 //        Done = 1'b1;
65      if (~process) begin
66        if(mbits0 == 4'b0000 && mbits1 == 4'b0000)
67          begin miss = 1'b1; Done = 1'b0; end
68
69        else if (|(mbits0|| mbits1))
70          begin miss = 1'b0; Done = 1'b0; end
71        else Done = 1'b1;
72      end
73
74    //if it's a hit, we can just use the cache
75      if (Done == 1'b1)
76        Done = 1'b1;
77      else if (miss == 1'b0) begin
78        process = 1;
79
80        cache_addr[5:4] = group; //cache address is 3 block bits
             and then 4 word bits
81
82        if (mbits0 != 4'b0000)begin //are we taking from Cache0?
83          cache_addr[6] = 1'b0;
84          replace[group] = 1'b1;
85          Done = 1'b1;
86          process = 1'b0;
87        end
```

```
88          else begin              //or Cache1?
89            cache_addr[6] = 1'b1;
90            replace[group] = 1'b0;
91            Done = 1'b1;
92            process = 1'b0;
93          end
94
95          cache_addr[3:0] = word;
96
97          if(wr_en) begin         //if it's a write, need to set a
                few more things
98            cache_wren = 1'b1;
99            //cache_data = in;
100           cam0_dirtybit[group] = mbits0[group];
101           cam1_dirtybit[group] = mbits1[group];
102 //          if (cam0_dirtybit != 4'b0000) begin     DONT NEED
103 //            writeback0[wb_index0] = Addr;
104 //            wb_index0 = wb_index0 + 1'b1;
105 //          end
106 //          if (cam1_dirtybit != 4'b0000) begin
107 //            writeback1[wb_index1] = Addr;
108 //          end
109         end
110         else
111           cache_wren = 1'b0;
```

```
112
113         //just_reset = 1'b0;                    DONT NEED
114       end
115
116       //it's a miss, gotta replace
117       else begin
118         process = 1'b1;
119         if (replace[group] == 1'b0)
120           writeback = cam0_dirtybit[group];
121         else writeback = cam1_dirtybit[group]; //what's writeback
              ?
122         if (!writeback)begin                    //Nothing's been
              written to, can just replace the block
123           cache_addr = {replace[group], group, current_word
                [3:0]};
124           DM_rd_addr = {Addr[31:4], current_word[3:0]};
125           cache_wren = 1'b1;
126           DM_wren = 1'b0;
127           current_word = current_word + 1'b1;
128
129           if (current_word == 5'b10001) begin //done with black
                transfer into cache
130             miss = 1'b0;
131             cache_wren = 1'b0;
132             current_word = 5'b00000;
```

```
133             if(replace[group]) begin //replacing things in CAM
134               wr_en_1 = 1'b1;
135             end
136             else begin
137               wr_en_0 = 1'b1;
138             end
139             //cache_addr={replace[group], group, word};
140           end
141
142         end
143
144         if (writeback) begin          //shit's been written to,
              have to writeback and then replace
145           wr_en_0 = 1'b0; wr_en_1 = 1'b0; cache_wren = 1'b0;
               DM_wren = 1'b1;
146
147           if (replace[group]) begin
148             cache_addr = {1'b1, group, current_word[3:0]};
149             DM_wr_addr = {dout_CAM_1, group, current_word[3:0]};
                  end
150
151           else begin//if (replace[group])
152             cache_addr = {1'b0, group, current_word[3:0]};
153             DM_wr_addr = {dout_CAM_0, group, current_word[3:0]};
                  end
```

```
154
155             current_word = current_word + 1'b1;
156             if (current_word == 5'b10001) begin     //done with
                    writeback
157                 writeback = 1'b0;
158                 DM_wren = 1'b0;
159                 current_word = 5'b00000;
160                 if (replace[group])
161                     cam1_dirtybit[group] = 1'b0;
162                 else
163                     cam0_dirtybit[group] = 1'b0;
164             end
165
166
167         end
168
169
170     end
171
172
173   end
174 end
175 endmodule
```

Listing I.3: Data Memory Cache Logic

I.4 DMF RISC Verilog Code CAM Memory

```
1  module dmf_CAM_v (PM, we_n, rd_n, din, argin, addrs, dout,
       mbits);
2
3  //
       ------------------------------------------------------------------------
4  //-- Declare input and output port types
5  //
       ------------------------------------------------------------------------
6    input PM;
7    input we_n, rd_n;      //write and read enables
8    input [25:0]  din, argin; //data input and argument input
       busses
9    input [1:0] addrs;     //address input bus; points to 4
       locations
10   output  reg [25:0]  dout;
11   output  reg [3:0] mbits; //data output bus and mbits = match
       bits
12 //
       ------------------------------------------------------------------------
13 //-- Declare internal memory array
```

```
14 //
     ----------------------------------------------------------------------

15   reg [25:0] cam_mem [3:0]; //an array of 4x26 bit locations
16   integer i, int_addrs;
17 //
     ----------------------------------------------------------------------

18 //-- The WRITE procedural block.
19 //-- This enables a new tag value to be written at a specific
     location,
20 //--     using a WE, data input and address input busses as with
      any
21 //--     other memory.
22 //-- In the context of a cache, this happens when a new block
     is
23 //--     uploaded in the cache.
24 //
     ----------------------------------------------------------------------

25   always @ (we_n, din, addrs, argin, PM)
26     begin
27       if (PM)
28         begin
29           mbits = 4'b0000;
```

```
30          end
31
32        else
33          begin
34            if (argin === 26'bx)
35              mbits = 4'bx;
36            else
37              mbits = 4'b0000;
38          end
39
40        int_addrs = addrs;
41        dout = cam_mem[int_addrs];
42        if (we_n == 1)
43          begin
44            cam_mem[int_addrs] = din;
45          end
46
47        if(argin == cam_mem[int_addrs])
48          begin
49            mbits = 4'b0000;
50            mbits[int_addrs] = 1;
51          end
52      end
53  //
    ------------------------------------------------------------------
```

```
54 //-- The READ procedural block.
55 //-- This allows a value at a specific location to be read out,
56 //--     using a RD, data output and address input busses as
       with any
57 //--     other memory.
58 //-- In the context of a cache, this is not necessary. This
       functionality
59 //--     is provided here for reference and debugging purposes.
60 //
       ------------------------------------------------------------------

61 //  always @ (rd_n)
62 //    begin
63 //      int_addrs = addrs;
64 //      if (rd_n == 1)
65 //        begin
66 //          dout = cam_mem[int_addrs];
67 //        end
68 //      else
69 //        begin
70 //          dout = 8'b0;
71 //        end
72 //    end
```

```
73 //
     ----------------------------------------------------------------------

74 //-- The MATCH procedural block.
75 //-- This implements the actual CAM function.
76 //-- An mbit is 1 if the argument value is equal to the content
     of the
77 //--    memory location associated with it.
78 //
     ----------------------------------------------------------------------

79 //  always @ (argin)
80 //    begin
81 //      int_addrs = addrs;
82 //      mbits = 4'b0000;
83 //        if(argin == cam_mem[int_addrs])
84 //          begin
85 //            mbits[int_addrs] = 1;
86 //          end
87
88 //        for (i=0; i <= 3; i=i+1)
89 //          begin
90 //            if (argin == cam_mem[i])
91 //              begin
92 //                mbits[i] = 1;
```

```
93 //                  end
94 //              end
95 //      end
96 endmodule
```

Listing I.4: CAM Memory

I.5 DMF RISC Verilog Code Main PM

```
1  // megafunction wizard: %ROM: 1-PORT%
2  // GENERATION: STANDARD
3  // VERSION: WM1.0
4  // MODULE: altsyncram
5
6  // ============================================================
7  // File Name: dmf_PM_v.v
8  // Megafunction Name(s):
9  //       altsyncram
10 //
11 // Simulation Library Files(s):
12 //       altera_mf
13 // ============================================================
14 // ************************************************************
15 // THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
16 //
17 // 15.1.0 Build 185 10/21/2015 SJ Lite Edition
18 // ************************************************************
19
20
21 //Copyright (C) 1991-2015 Altera Corporation. All rights
      reserved.
```

```
22 //Your use of Altera Corporation's design tools, logic
      functions
23 //and other software and tools, and its AMPP partner logic
24 //functions, and any output files from any of the foregoing
25 //(including device programming or simulation files), and any
26 //associated documentation or information are expressly subject
27 //to the terms and conditions of the Altera Program License
28 //Subscription Agreement, the Altera Quartus Prime License
      Agreement,
29 //the Altera MegaCore Function License Agreement, or other
30 //applicable license agreement, including, without limitation,
31 //that your use is for the sole purpose of programming logic
32 //devices manufactured by Altera and sold by Altera or its
33 //authorized distributors.  Please refer to the applicable
34 //agreement for further details.
35
36
37 // synopsys translate_off
38 `timescale 1 ps / 1 ps
39 // synopsys translate_on
40 module dmf_PM_v (
41   address,
42   clock,
43   q);
44
```

```
45    input [13:0]  address;
46    input    clock;
47    output   [31:0]  q;
48  `ifndef ALTERA_RESERVED_QIS
49  // synopsys translate_off
50  `endif
51    tri1     clock;
52  `ifndef ALTERA_RESERVED_QIS
53  // synopsys translate_on
54  `endif
55
56    wire [31:0] sub_wire0;
57    wire [31:0] q = sub_wire0[31:0];
58
59    altsyncram  altsyncram_component (
60          .address_a (address),
61          .clock0 (clock),
62          .q_a (sub_wire0),
63          .aclr0 (1'b0),
64          .aclr1 (1'b0),
65          .address_b (1'b1),
66          .addressstall_a (1'b0),
67          .addressstall_b (1'b0),
68          .byteena_a (1'b1),
69          .byteena_b (1'b1),
```

```
70          .clock1 (1'b1),
71          .clocken0 (1'b1),
72          .clocken1 (1'b1),
73          .clocken2 (1'b1),
74          .clocken3 (1'b1),
75          .data_a ({32{1'b1}}),
76          .data_b (1'b1),
77          .eccstatus (),
78          .q_b (),
79          .rden_a (1'b1),
80          .rden_b (1'b1),
81          .wren_a (1'b0),
82          .wren_b (1'b0));
83    defparam
84      altsyncram_component.address_aclr_a = "NONE",
85      altsyncram_component.clock_enable_input_a = "BYPASS",
86      altsyncram_component.clock_enable_output_a = "BYPASS",
87      altsyncram_component.init_file = "dmfRISC621_rom1.mif",
88      altsyncram_component.intended_device_family = "Cyclone IV E
          ",
89      altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
90      altsyncram_component.lpm_type = "altsyncram",
91      altsyncram_component.numwords_a = 16384,
92      altsyncram_component.operation_mode = "ROM",
93      altsyncram_component.outdata_aclr_a = "NONE",
```

```
94      altsyncram_component.outdata_reg_a = "UNREGISTERED",
95      altsyncram_component.widthad_a = 14,
96      altsyncram_component.width_a = 32,
97      altsyncram_component.width_byteena_a = 1;
98
99
100 endmodule
101
102 // ============================================================
103 // CNX file retrieval info
104 // ============================================================
105 // Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
106 // Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
107 // Retrieval info: PRIVATE: AclrByte NUMERIC "0"
108 // Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
109 // Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
110 // Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
111 // Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
112 // Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
113 // Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
114 // Retrieval info: PRIVATE: Clken NUMERIC "0"
115 // Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
116 // Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
117 // Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
```

```
118 // Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "
    Cyclone IV E"
119 // Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
120 // Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
121 // Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
122 // Retrieval info: PRIVATE: MIFfilename STRING "dmfRISC621_rom1
    .mif"
123 // Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "16384"
124 // Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
125 // Retrieval info: PRIVATE: RegAddr NUMERIC "1"
126 // Retrieval info: PRIVATE: RegOutput NUMERIC "0"
127 // Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING
    "0"
128 // Retrieval info: PRIVATE: SingleClock NUMERIC "1"
129 // Retrieval info: PRIVATE: UseDQRAM NUMERIC "0"
130 // Retrieval info: PRIVATE: WidthAddr NUMERIC "14"
131 // Retrieval info: PRIVATE: WidthData NUMERIC "32"
132 // Retrieval info: PRIVATE: rden NUMERIC "0"
133 // Retrieval info: LIBRARY: altera_mf altera_mf.
    altera_mf_components.all
134 // Retrieval info: CONSTANT: ADDRESS_ACLR_A STRING "NONE"
135 // Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "
    BYPASS"
136 // Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "
    BYPASS"
```

```
137 // Retrieval info: CONSTANT: INIT_FILE STRING "dmfRISC621_rom1.
    mif"
138 // Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "
    Cyclone IV E"
139 // Retrieval info: CONSTANT: LPM_HINT STRING "
    ENABLE_RUNTIME_MOD=NO"
140 // Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
141 // Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "16384"
142 // Retrieval info: CONSTANT: OPERATION_MODE STRING "ROM"
143 // Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
144 // Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED
    "
145 // Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "14"
146 // Retrieval info: CONSTANT: WIDTH_A NUMERIC "32"
147 // Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
148 // Retrieval info: USED_PORT: address 0 0 14 0 INPUT NODEFVAL "
    address[13..0]"
149 // Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
150 // Retrieval info: USED_PORT: q 0 0 32 0 OUTPUT NODEFVAL "q
    [31..0]"
151 // Retrieval info: CONNECT: @address_a 0 0 14 0 address 0 0 14
    0
152 // Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
153 // Retrieval info: CONNECT: q 0 0 32 0 @q_a 0 0 32 0
154 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PM_v.v TRUE
```

```
155 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PM_v.inc FALSE
156 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PM_v.cmp FALSE
157 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PM_v.bsf TRUE
158 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PM_v_inst.v FALSE
159 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PM_v_bb.v TRUE
160 // Retrieval info: LIB_FILE: altera_mf
```

Listing I.5: Main PM

I.6 dmf_RISC Verilog Code Main DM

```
1  // megafunction wizard: %RAM: 1-PORT%
2  // GENERATION: STANDARD
3  // VERSION: WM1.0
4  // MODULE: altsyncram
5
6  // ============================================================
7  // File Name: dmf_DM_v.v
8  // Megafunction Name(s):
9  //         altsyncram
10 //
11 // Simulation Library Files(s):
12 //         altera_mf
13 // ============================================================
14 // ************************************************************
15 // THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
16 //
17 // 15.1.0 Build 185 10/21/2015 SJ Lite Edition
18 // ************************************************************
19
20
21 //Copyright (C) 1991-2015 Altera Corporation. All rights
      reserved.
```

```
22 //Your use of Altera Corporation's design tools, logic
       functions
23 //and other software and tools, and its AMPP partner logic
24 //functions, and any output files from any of the foregoing
25 //(including device programming or simulation files), and any
26 //associated documentation or information are expressly subject
27 //to the terms and conditions of the Altera Program License
28 //Subscription Agreement, the Altera Quartus Prime License
       Agreement,
29 //the Altera MegaCore Function License Agreement, or other
30 //applicable license agreement, including, without limitation,
31 //that your use is for the sole purpose of programming logic
32 //devices manufactured by Altera and sold by Altera or its
33 //authorized distributors.  Please refer to the applicable
34 //agreement for further details.
35
36
37 // synopsys translate_off
38 `timescale 1 ps / 1 ps
39 // synopsys translate_on
40 module dmf_DM_v (
41   address,
42   clock,
43   data,
44   wren,
```

```
45    q);
46
47    input [13:0]  address;
48    input    clock;
49    input [31:0]  data;
50    input    wren;
51    output  [31:0]  q;
52  `ifndef ALTERA_RESERVED_QIS
53  // synopsys translate_off
54  `endif
55    tri1     clock;
56  `ifndef ALTERA_RESERVED_QIS
57  // synopsys translate_on
58  `endif
59
60    wire [31:0] sub_wire0;
61    wire [31:0] q = sub_wire0[31:0];
62
63    altsyncram  altsyncram_component (
64          .address_a (address),
65          .clock0 (clock),
66          .data_a (data),
67          .wren_a (wren),
68          .q_a (sub_wire0),
69          .aclr0 (1'b0),
```

```
70          .aclr1 (1'b0),
71          .address_b (1'b1),
72          .addressstall_a (1'b0),
73          .addressstall_b (1'b0),
74          .byteena_a (1'b1),
75          .byteena_b (1'b1),
76          .clock1 (1'b1),
77          .clocken0 (1'b1),
78          .clocken1 (1'b1),
79          .clocken2 (1'b1),
80          .clocken3 (1'b1),
81          .data_b (1'b1),
82          .eccstatus (),
83          .q_b (),
84          .rden_a (1'b1),
85          .rden_b (1'b1),
86          .wren_b (1'b0));
87    defparam
88      altsyncram_component.clock_enable_input_a = "BYPASS",
89      altsyncram_component.clock_enable_output_a = "BYPASS",
90      altsyncram_component.intended_device_family = "Cyclone IV E
          ",
91      altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
92      altsyncram_component.lpm_type = "altsyncram",
93      altsyncram_component.numwords_a = 16384,
```

```
94      altsyncram_component.operation_mode = "SINGLE_PORT",
95      altsyncram_component.outdata_aclr_a = "NONE",
96      altsyncram_component.outdata_reg_a = "UNREGISTERED",
97      altsyncram_component.power_up_uninitialized = "FALSE",
98      altsyncram_component.read_during_write_mode_port_a = "
          NEW_DATA_NO_NBE_READ",
99      altsyncram_component.widthad_a = 14,
100     altsyncram_component.width_a = 32,
101     altsyncram_component.width_byteena_a = 1;
102
103
104 endmodule
105
106 // ============================================================
107 // CNX file retrieval info
108 // ============================================================
109 // Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
110 // Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
111 // Retrieval info: PRIVATE: AclrByte NUMERIC "0"
112 // Retrieval info: PRIVATE: AclrData NUMERIC "0"
113 // Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
114 // Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
115 // Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
116 // Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
117 // Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
```

```
118 // Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
119 // Retrieval info: PRIVATE: Clken NUMERIC "0"
120 // Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
121 // Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
122 // Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
123 // Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
124 // Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "
      Cyclone IV E"
125 // Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
126 // Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
127 // Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
128 // Retrieval info: PRIVATE: MIFfilename STRING ""
129 // Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "16384"
130 // Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
131 // Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A
      NUMERIC "3"
132 // Retrieval info: PRIVATE: RegAddr NUMERIC "1"
133 // Retrieval info: PRIVATE: RegData NUMERIC "1"
134 // Retrieval info: PRIVATE: RegOutput NUMERIC "0"
135 // Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING
      "0"
136 // Retrieval info: PRIVATE: SingleClock NUMERIC "1"
137 // Retrieval info: PRIVATE: UseDQRAM NUMERIC "1"
138 // Retrieval info: PRIVATE: WRCONTROL_ACLR_A NUMERIC "0"
139 // Retrieval info: PRIVATE: WidthAddr NUMERIC "14"
```

```
140 // Retrieval info: PRIVATE: WidthData NUMERIC "32"
141 // Retrieval info: PRIVATE: rden NUMERIC "0"
142 // Retrieval info: LIBRARY: altera_mf altera_mf.
    altera_mf_components.all
143 // Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "
    BYPASS"
144 // Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "
    BYPASS"
145 // Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "
    Cyclone IV E"
146 // Retrieval info: CONSTANT: LPM_HINT STRING "
    ENABLE_RUNTIME_MOD=NO"
147 // Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
148 // Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "16384"
149 // Retrieval info: CONSTANT: OPERATION_MODE STRING "SINGLE_PORT
    "
150 // Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
151 // Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED
    "
152 // Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "
    FALSE"
153 // Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_PORT_A
    STRING "NEW_DATA_NO_NBE_READ"
154 // Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "14"
155 // Retrieval info: CONSTANT: WIDTH_A NUMERIC "32"
```

```
156 // Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
157 // Retrieval info: USED_PORT: address 0 0 14 0 INPUT NODEFVAL "
    address[13..0]"
158 // Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
159 // Retrieval info: USED_PORT: data 0 0 32 0 INPUT NODEFVAL "
    data[31..0]"
160 // Retrieval info: USED_PORT: q 0 0 32 0 OUTPUT NODEFVAL "q
    [31..0]"
161 // Retrieval info: USED_PORT: wren 0 0 0 0 INPUT NODEFVAL "wren
    "
162 // Retrieval info: CONNECT: @address_a 0 0 14 0 address 0 0 14
    0
163 // Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
164 // Retrieval info: CONNECT: @data_a 0 0 32 0 data 0 0 32 0
165 // Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
166 // Retrieval info: CONNECT: q 0 0 32 0 @q_a 0 0 32 0
167 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_DM_v.v TRUE
168 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_DM_v.inc FALSE
169 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_DM_v.cmp FALSE
170 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_DM_v.bsf TRUE
171 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_DM_v_inst.v FALSE
172 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_DM_v_bb.v TRUE
173 // Retrieval info: LIB_FILE: altera_mf
```

Listing I.6: Main DM

I.7 DMF RISC Verilog Code PM Cache

```
1  // megafunction wizard: %RAM: 1-PORT%
2  // GENERATION: STANDARD
3  // VERSION: WM1.0
4  // MODULE: altsyncram
5
6  // ============================================================
7  // File Name: dmf_PM_cache_v.v
8  // Megafunction Name(s):
9  //         altsyncram
10 //
11 // Simulation Library Files(s):
12 //         altera_mf
13 // ============================================================
14 // ************************************************************
15 // THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
16 //
17 // 15.1.0 Build 185 10/21/2015 SJ Lite Edition
18 // ************************************************************
19
20
21 //Copyright (C) 1991-2015 Altera Corporation. All rights
      reserved.
```

```
22 //Your use of Altera Corporation's design tools, logic
       functions
23 //and other software and tools, and its AMPP partner logic
24 //functions, and any output files from any of the foregoing
25 //(including device programming or simulation files), and any
26 //associated documentation or information are expressly subject
27 //to the terms and conditions of the Altera Program License
28 //Subscription Agreement, the Altera Quartus Prime License
       Agreement,
29 //the Altera MegaCore Function License Agreement, or other
30 //applicable license agreement, including, without limitation,
31 //that your use is for the sole purpose of programming logic
32 //devices manufactured by Altera and sold by Altera or its
33 //authorized distributors.  Please refer to the applicable
34 //agreement for further details.
35
36
37 // synopsys translate_off
38 `timescale 1 ps / 1 ps
39 // synopsys translate_on
40 module dmf_PM_cache_v (
41   address,
42   clock,
43   data,
44   wren,
```

```
45    q);
46
47    input [6:0]  address;
48    input    clock;
49    input [31:0]  data;
50    input    wren;
51    output  [31:0]  q;
52  `ifndef ALTERA_RESERVED_QIS
53  // synopsys translate_off
54  `endif
55    tri1     clock;
56  `ifndef ALTERA_RESERVED_QIS
57  // synopsys translate_on
58  `endif
59
60    wire [31:0] sub_wire0;
61    wire [31:0] q = sub_wire0[31:0];
62
63    altsyncram  altsyncram_component (
64          .address_a (address),
65          .clock0 (clock),
66          .data_a (data),
67          .wren_a (wren),
68          .q_a (sub_wire0),
69          .aclr0 (1'b0),
```

```
70          .aclr1 (1'b0),
71          .address_b (1'b1),
72          .addressstall_a (1'b0),
73          .addressstall_b (1'b0),
74          .byteena_a (1'b1),
75          .byteena_b (1'b1),
76          .clock1 (1'b1),
77          .clocken0 (1'b1),
78          .clocken1 (1'b1),
79          .clocken2 (1'b1),
80          .clocken3 (1'b1),
81          .data_b (1'b1),
82          .eccstatus (),
83          .q_b (),
84          .rden_a (1'b1),
85          .rden_b (1'b1),
86          .wren_b (1'b0));
87    defparam
88      altsyncram_component.clock_enable_input_a = "BYPASS",
89      altsyncram_component.clock_enable_output_a = "BYPASS",
90      altsyncram_component.intended_device_family = "Cyclone IV E
          ",
91      altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
92      altsyncram_component.lpm_type = "altsyncram",
93      altsyncram_component.numwords_a = 128,
```

```
94      altsyncram_component.operation_mode = "SINGLE_PORT",
95      altsyncram_component.outdata_aclr_a = "NONE",
96      altsyncram_component.outdata_reg_a = "UNREGISTERED",
97      altsyncram_component.power_up_uninitialized = "FALSE",
98      altsyncram_component.read_during_write_mode_port_a = "
          NEW_DATA_NO_NBE_READ",
99      altsyncram_component.widthad_a = 7,
100     altsyncram_component.width_a = 32,
101     altsyncram_component.width_byteena_a = 1;
102
103
104 endmodule
105
106 // ============================================================
107 // CNX file retrieval info
108 // ============================================================
109 // Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
110 // Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
111 // Retrieval info: PRIVATE: AclrByte NUMERIC "0"
112 // Retrieval info: PRIVATE: AclrData NUMERIC "0"
113 // Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
114 // Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
115 // Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
116 // Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
117 // Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
```

```
118 // Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
119 // Retrieval info: PRIVATE: Clken NUMERIC "0"
120 // Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
121 // Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
122 // Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
123 // Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
124 // Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "
      Cyclone IV E"
125 // Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
126 // Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
127 // Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
128 // Retrieval info: PRIVATE: MIFfilename STRING ""
129 // Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "128"
130 // Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
131 // Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A
      NUMERIC "3"
132 // Retrieval info: PRIVATE: RegAddr NUMERIC "1"
133 // Retrieval info: PRIVATE: RegData NUMERIC "1"
134 // Retrieval info: PRIVATE: RegOutput NUMERIC "0"
135 // Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING
      "0"
136 // Retrieval info: PRIVATE: SingleClock NUMERIC "1"
137 // Retrieval info: PRIVATE: UseDQRAM NUMERIC "1"
138 // Retrieval info: PRIVATE: WRCONTROL_ACLR_A NUMERIC "0"
139 // Retrieval info: PRIVATE: WidthAddr NUMERIC "7"
```

```
140 // Retrieval info: PRIVATE: WidthData NUMERIC "32"
141 // Retrieval info: PRIVATE: rden NUMERIC "0"
142 // Retrieval info: LIBRARY: altera_mf altera_mf.
    altera_mf_components.all
143 // Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "
    BYPASS"
144 // Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "
    BYPASS"
145 // Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "
    Cyclone IV E"
146 // Retrieval info: CONSTANT: LPM_HINT STRING "
    ENABLE_RUNTIME_MOD=NO"
147 // Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
148 // Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "128"
149 // Retrieval info: CONSTANT: OPERATION_MODE STRING "SINGLE_PORT
    "
150 // Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
151 // Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED
    "
152 // Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "
    FALSE"
153 // Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_PORT_A
    STRING "NEW_DATA_NO_NBE_READ"
154 // Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "10"
155 // Retrieval info: CONSTANT: WIDTH_A NUMERIC "32"
```

```
156 // Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
157 // Retrieval info: USED_PORT: address 0 0 10 0 INPUT NODEFVAL "
    address[9..0]"
158 // Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
159 // Retrieval info: USED_PORT: data 0 0 32 0 INPUT NODEFVAL "
    data[31..0]"
160 // Retrieval info: USED_PORT: q 0 0 32 0 OUTPUT NODEFVAL "q
    [31..0]"
161 // Retrieval info: USED_PORT: wren 0 0 0 0 INPUT NODEFVAL "wren
    "
162 // Retrieval info: CONNECT: @address_a 0 0 10 0 address 0 0 10
    0
163 // Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
164 // Retrieval info: CONNECT: @data_a 0 0 32 0 data 0 0 32 0
165 // Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
166 // Retrieval info: CONNECT: q 0 0 32 0 @q_a 0 0 32 0
167 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PM_cache_v.v TRUE
168 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PM_cache_v.inc
    FALSE
169 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PM_cache_v.cmp
    FALSE
170 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PM_cache_v.bsf
    TRUE
171 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PM_cache_v_inst.v
    FALSE
```

```
172 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PM_cache_v_bb.v
    TRUE
173 // Retrieval info: LIB_FILE: altera_mf
```

Listing I.7: PM Cache

I.8 dmf_RISC Verilog Code DM Cache

```
1  // megafunction wizard: %RAM: 1-PORT%
2  // GENERATION: STANDARD
3  // VERSION: WM1.0
4  // MODULE: altsyncram
5
6  // ============================================================
7  // File Name: dmf_DM_cache_v.v
8  // Megafunction Name(s):
9  //       altsyncram
10 //
11 // Simulation Library Files(s):
12 //       altera_mf
13 // ============================================================
14 // ************************************************************
15 // THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
16 //
17 // 15.1.0 Build 185 10/21/2015 SJ Lite Edition
18 // ************************************************************
19
20
21 //Copyright (C) 1991-2015 Altera Corporation. All rights
      reserved.
```

```
22 //Your use of Altera Corporation's design tools, logic
       functions
23 //and other software and tools, and its AMPP partner logic
24 //functions, and any output files from any of the foregoing
25 //(including device programming or simulation files), and any
26 //associated documentation or information are expressly subject
27 //to the terms and conditions of the Altera Program License
28 //Subscription Agreement, the Altera Quartus Prime License
       Agreement,
29 //the Altera MegaCore Function License Agreement, or other
30 //applicable license agreement, including, without limitation,
31 //that your use is for the sole purpose of programming logic
32 //devices manufactured by Altera and sold by Altera or its
33 //authorized distributors.  Please refer to the applicable
34 //agreement for further details.
35
36
37 // synopsys translate_off
38 `timescale 1 ps / 1 ps
39 // synopsys translate_on
40 module dmf_DM_cache_v (
41   address,
42   clock,
43   data,
44   wren,
```

```
45    q);
46
47    input [6:0]   address;
48    input    clock;
49    input [31:0]   data;
50    input    wren;
51    output  [31:0]  q;
52  `ifndef ALTERA_RESERVED_QIS
53  // synopsys translate_off
54  `endif
55    tri1     clock;
56  `ifndef ALTERA_RESERVED_QIS
57  // synopsys translate_on
58  `endif
59
60    wire [31:0] sub_wire0;
61    wire [31:0] q = sub_wire0[31:0];
62
63    altsyncram  altsyncram_component (
64          .address_a (address),
65          .clock0 (clock),
66          .data_a (data),
67          .wren_a (wren),
68          .q_a (sub_wire0),
69          .aclr0 (1'b0),
```

```
70          .aclr1 (1'b0),
71          .address_b (1'b1),
72          .addressstall_a (1'b0),
73          .addressstall_b (1'b0),
74          .byteena_a (1'b1),
75          .byteena_b (1'b1),
76          .clock1 (1'b1),
77          .clocken0 (1'b1),
78          .clocken1 (1'b1),
79          .clocken2 (1'b1),
80          .clocken3 (1'b1),
81          .data_b (1'b1),
82          .eccstatus (),
83          .q_b (),
84          .rden_a (1'b1),
85          .rden_b (1'b1),
86          .wren_b (1'b0));
87    defparam
88      altsyncram_component.clock_enable_input_a = "BYPASS",
89      altsyncram_component.clock_enable_output_a = "BYPASS",
90      altsyncram_component.intended_device_family = "Cyclone IV E
            ",
91      altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
92      altsyncram_component.lpm_type = "altsyncram",
93      altsyncram_component.numwords_a = 128,
```

```
94      altsyncram_component.operation_mode = "SINGLE_PORT",
95      altsyncram_component.outdata_aclr_a = "NONE",
96      altsyncram_component.outdata_reg_a = "UNREGISTERED",
97      altsyncram_component.power_up_uninitialized = "FALSE",
98      altsyncram_component.read_during_write_mode_port_a = "
          NEW_DATA_NO_NBE_READ",
99      altsyncram_component.widthad_a = 7,
100     altsyncram_component.width_a = 32,
101     altsyncram_component.width_byteena_a = 1;
102
103
104 endmodule
105
106 // ============================================================
107 // CNX file retrieval info
108 // ============================================================
109 // Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
110 // Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
111 // Retrieval info: PRIVATE: AclrByte NUMERIC "0"
112 // Retrieval info: PRIVATE: AclrData NUMERIC "0"
113 // Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
114 // Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
115 // Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
116 // Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
117 // Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
```

```
118 // Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
119 // Retrieval info: PRIVATE: Clken NUMERIC "0"
120 // Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
121 // Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
122 // Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
123 // Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
124 // Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "
        Cyclone IV E"
125 // Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
126 // Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
127 // Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
128 // Retrieval info: PRIVATE: MIFfilename STRING ""
129 // Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "128"
130 // Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
131 // Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A
        NUMERIC "3"
132 // Retrieval info: PRIVATE: RegAddr NUMERIC "1"
133 // Retrieval info: PRIVATE: RegData NUMERIC "1"
134 // Retrieval info: PRIVATE: RegOutput NUMERIC "0"
135 // Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING
        "0"
136 // Retrieval info: PRIVATE: SingleClock NUMERIC "1"
137 // Retrieval info: PRIVATE: UseDQRAM NUMERIC "1"
138 // Retrieval info: PRIVATE: WRCONTROL_ACLR_A NUMERIC "0"
139 // Retrieval info: PRIVATE: WidthAddr NUMERIC "7"
```

```
140 // Retrieval info: PRIVATE: WidthData NUMERIC "32"
141 // Retrieval info: PRIVATE: rden NUMERIC "0"
142 // Retrieval info: LIBRARY: altera_mf altera_mf.
    altera_mf_components.all
143 // Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "
    BYPASS"
144 // Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "
    BYPASS"
145 // Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "
    Cyclone IV E"
146 // Retrieval info: CONSTANT: LPM_HINT STRING "
    ENABLE_RUNTIME_MOD=NO"
147 // Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
148 // Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "128"
149 // Retrieval info: CONSTANT: OPERATION_MODE STRING "SINGLE_PORT
    "
150 // Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
151 // Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED
    "
152 // Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "
    FALSE"
153 // Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_PORT_A
    STRING "NEW_DATA_NO_NBE_READ"
154 // Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "10"
155 // Retrieval info: CONSTANT: WIDTH_A NUMERIC "32"
```

```
156 // Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
157 // Retrieval info: USED_PORT: address 0 0 10 0 INPUT NODEFVAL "
    address[9..0]"
158 // Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
159 // Retrieval info: USED_PORT: data 0 0 32 0 INPUT NODEFVAL "
    data[31..0]"
160 // Retrieval info: USED_PORT: q 0 0 32 0 OUTPUT NODEFVAL "q
    [31..0]"
161 // Retrieval info: USED_PORT: wren 0 0 0 0 INPUT NODEFVAL "wren
    "
162 // Retrieval info: CONNECT: @address_a 0 0 10 0 address 0 0 10
    0
163 // Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
164 // Retrieval info: CONNECT: @data_a 0 0 32 0 data 0 0 32 0
165 // Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
166 // Retrieval info: CONNECT: q 0 0 32 0 @q_a 0 0 32 0
167 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_DM_cache_v.v TRUE
168 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_DM_cache_v.inc
    FALSE
169 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_DM_cache_v.cmp
    FALSE
170 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_DM_cache_v.bsf
    TRUE
171 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_DM_cache_v_inst.v
    FALSE
```

```
172 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_DM_cache_v_bb.v
    TRUE
173 // Retrieval info: LIB_FILE: altera_mf
```

Listing I.8: DM Cache

I.9 DMF RISC Verilog Code PLL

```
1  // megafunction wizard: %ALTPLL%
2  // GENERATION: STANDARD
3  // VERSION: WM1.0
4  // MODULE: altpll
5
6  // ============================================================
7  // File Name: dmf_PLL_2.v
8  // Megafunction Name(s):
9  //       altpll
10 //
11 // Simulation Library Files(s):
12 //       altera_mf
13 // ============================================================
14 // ************************************************************
15 // THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
16 //
17 // 18.0.0 Build 614 04/24/2018 SJ Lite Edition
18 // ************************************************************
19
20
21 //Copyright (C) 2018  Intel Corporation. All rights reserved.
22 //Your use of Intel Corporation's design tools, logic functions
23 //and other software and tools, and its AMPP partner logic
```

```
24 //functions, and any output files from any of the foregoing
25 //(including device programming or simulation files), and any
26 //associated documentation or information are expressly subject
27 //to the terms and conditions of the Intel Program License
28 //Subscription Agreement, the Intel Quartus Prime License
      Agreement,
29 //the Intel FPGA IP License Agreement, or other applicable
      license
30 //agreement, including, without limitation, that your use is
      for
31 //the sole purpose of programming logic devices manufactured by
32 //Intel and sold by Intel or its authorized distributors.
      Please
33 //refer to the applicable agreement for further details.
34
35
36 // synopsys translate_off
37 `timescale 1 ps / 1 ps
38 // synopsys translate_on
39 module dmf_PLL_2 (
40   inclk0,
41   c0,
42   c1,
43   c2);
44
```

```
45    input    inclk0;
46    output     c0;
47    output     c1;
48    output     c2;
49
50    wire [0:0] sub_wire2 = 1'h0;
51    wire [4:0] sub_wire3;
52    wire  sub_wire0 = inclk0;
53    wire [1:0] sub_wire1 = {sub_wire2, sub_wire0};
54    wire [2:2] sub_wire6 = sub_wire3[2:2];
55    wire [1:1] sub_wire5 = sub_wire3[1:1];
56    wire [0:0] sub_wire4 = sub_wire3[0:0];
57    wire  c0 = sub_wire4;
58    wire  c1 = sub_wire5;
59    wire  c2 = sub_wire6;
60
61    altpll  altpll_component (
62          .inclk (sub_wire1),
63          .clk (sub_wire3),
64          .activeclock (),
65          .areset (1'b0),
66          .clkbad (),
67          .clkena ({6{1'b1}}),
68          .clkloss (),
69          .clkswitch (1'b0),
```

```
70          .configupdate (1'b0),
71          .enable0 (),
72          .enable1 (),
73          .extclk (),
74          .extclkena ({4{1'b1}}),
75          .fbin (1'b1),
76          .fbmimicbidir (),
77          .fbout (),
78          .fref (),
79          .icdrclk (),
80          .locked (),
81          .pfdena (1'b1),
82          .phasecounterselect ({4{1'b1}}),
83          .phasedone (),
84          .phasestep (1'b1),
85          .phaseupdown (1'b1),
86          .pllena (1'b1),
87          .scanaclr (1'b0),
88          .scanclk (1'b0),
89          .scanclkena (1'b1),
90          .scandata (1'b0),
91          .scandataout (),
92          .scandone (),
93          .scanread (1'b0),
94          .scanwrite (1'b0),
```

```
95          .sclkout0 (),
96          .sclkout1 (),
97          .vcooverrange (),
98          .vcounderrange ());
99      defparam
100       altpll_component.bandwidth_type = "AUTO",
101       altpll_component.clk0_divide_by = 1,
102       altpll_component.clk0_duty_cycle = 50,
103       altpll_component.clk0_multiply_by = 1,
104       altpll_component.clk0_phase_shift = "0",
105       altpll_component.clk1_divide_by = 1,
106       altpll_component.clk1_duty_cycle = 50,
107       altpll_component.clk1_multiply_by = 1,
108       altpll_component.clk1_phase_shift = "1667",
109       altpll_component.clk2_divide_by = 1,
110       altpll_component.clk2_duty_cycle = 50,
111       altpll_component.clk2_multiply_by = 1,
112       altpll_component.clk2_phase_shift = "3333",
113       altpll_component.compensate_clock = "CLK0",
114       altpll_component.inclk0_input_frequency = 20000,
115       altpll_component.intended_device_family = "Cyclone IV E",
116       altpll_component.lpm_hint = "CBX_MODULE_PREFIX=dmf_PLL_2",
117       altpll_component.lpm_type = "altpll",
118       altpll_component.operation_mode = "NORMAL",
119       altpll_component.pll_type = "AUTO",
```

```
120     altpll_component.port_activeclock = "PORT_UNUSED",
121     altpll_component.port_areset = "PORT_UNUSED",
122     altpll_component.port_clkbad0 = "PORT_UNUSED",
123     altpll_component.port_clkbad1 = "PORT_UNUSED",
124     altpll_component.port_clkloss = "PORT_UNUSED",
125     altpll_component.port_clkswitch = "PORT_UNUSED",
126     altpll_component.port_configupdate = "PORT_UNUSED",
127     altpll_component.port_fbin = "PORT_UNUSED",
128     altpll_component.port_inclk0 = "PORT_USED",
129     altpll_component.port_inclk1 = "PORT_UNUSED",
130     altpll_component.port_locked = "PORT_UNUSED",
131     altpll_component.port_pfdena = "PORT_UNUSED",
132     altpll_component.port_phasecounterselect = "PORT_UNUSED",
133     altpll_component.port_phasedone = "PORT_UNUSED",
134     altpll_component.port_phasestep = "PORT_UNUSED",
135     altpll_component.port_phaseupdown = "PORT_UNUSED",
136     altpll_component.port_pllena = "PORT_UNUSED",
137     altpll_component.port_scanaclr = "PORT_UNUSED",
138     altpll_component.port_scanclk = "PORT_UNUSED",
139     altpll_component.port_scanclkena = "PORT_UNUSED",
140     altpll_component.port_scandata = "PORT_UNUSED",
141     altpll_component.port_scandataout = "PORT_UNUSED",
142     altpll_component.port_scandone = "PORT_UNUSED",
143     altpll_component.port_scanread = "PORT_UNUSED",
144     altpll_component.port_scanwrite = "PORT_UNUSED",
```

```
145       altpll_component.port_clk0 = "PORT_USED",
146       altpll_component.port_clk1 = "PORT_USED",
147       altpll_component.port_clk2 = "PORT_USED",
148       altpll_component.port_clk3 = "PORT_UNUSED",
149       altpll_component.port_clk4 = "PORT_UNUSED",
150       altpll_component.port_clk5 = "PORT_UNUSED",
151       altpll_component.port_clkena0 = "PORT_UNUSED",
152       altpll_component.port_clkena1 = "PORT_UNUSED",
153       altpll_component.port_clkena2 = "PORT_UNUSED",
154       altpll_component.port_clkena3 = "PORT_UNUSED",
155       altpll_component.port_clkena4 = "PORT_UNUSED",
156       altpll_component.port_clkena5 = "PORT_UNUSED",
157       altpll_component.port_extclk0 = "PORT_UNUSED",
158       altpll_component.port_extclk1 = "PORT_UNUSED",
159       altpll_component.port_extclk2 = "PORT_UNUSED",
160       altpll_component.port_extclk3 = "PORT_UNUSED",
161       altpll_component.width_clock = 5;
162
163
164 endmodule
165
166 // ============================================================
167 // CNX file retrieval info
168 // ============================================================
169 // Retrieval info: PRIVATE: ACTIVECLK_CHECK STRING "0"
```

```
170 // Retrieval info: PRIVATE: BANDWIDTH STRING "1.000"
171 // Retrieval info: PRIVATE: BANDWIDTH_FEATURE_ENABLED STRING
        "1"
172 // Retrieval info: PRIVATE: BANDWIDTH_FREQ_UNIT STRING "MHz"
173 // Retrieval info: PRIVATE: BANDWIDTH_PRESET STRING "Low"
174 // Retrieval info: PRIVATE: BANDWIDTH_USE_AUTO STRING "1"
175 // Retrieval info: PRIVATE: BANDWIDTH_USE_PRESET STRING "0"
176 // Retrieval info: PRIVATE: CLKBAD_SWITCHOVER_CHECK STRING "0"
177 // Retrieval info: PRIVATE: CLKLOSS_CHECK STRING "0"
178 // Retrieval info: PRIVATE: CLKSWITCH_CHECK STRING "0"
179 // Retrieval info: PRIVATE: CNX_NO_COMPENSATE_RADIO STRING "0"
180 // Retrieval info: PRIVATE: CREATE_CLKBAD_CHECK STRING "0"
181 // Retrieval info: PRIVATE: CREATE_INCLK1_CHECK STRING "0"
182 // Retrieval info: PRIVATE: CUR_DEDICATED_CLK STRING "c0"
183 // Retrieval info: PRIVATE: CUR_FBIN_CLK STRING "c0"
184 // Retrieval info: PRIVATE: DEVICE_SPEED_GRADE STRING "Any"
185 // Retrieval info: PRIVATE: DIV_FACTOR0 NUMERIC "1"
186 // Retrieval info: PRIVATE: DIV_FACTOR1 NUMERIC "1"
187 // Retrieval info: PRIVATE: DIV_FACTOR2 NUMERIC "1"
188 // Retrieval info: PRIVATE: DUTY_CYCLE0 STRING "50.00000000"
189 // Retrieval info: PRIVATE: DUTY_CYCLE1 STRING "50.00000000"
190 // Retrieval info: PRIVATE: DUTY_CYCLE2 STRING "50.00000000"
191 // Retrieval info: PRIVATE: EFF_OUTPUT_FREQ_VALUE0 STRING
        "50.000000"
```

```
192 // Retrieval info: PRIVATE: EFF_OUTPUT_FREQ_VALUE1 STRING
    "50.000000"
193 // Retrieval info: PRIVATE: EFF_OUTPUT_FREQ_VALUE2 STRING
    "50.000000"
194 // Retrieval info: PRIVATE: EXPLICIT_SWITCHOVER_COUNTER STRING
    "0"
195 // Retrieval info: PRIVATE: EXT_FEEDBACK_RADIO STRING "0"
196 // Retrieval info: PRIVATE: GLOCKED_COUNTER_EDIT_CHANGED STRING
     "1"
197 // Retrieval info: PRIVATE: GLOCKED_FEATURE_ENABLED STRING "0"
198 // Retrieval info: PRIVATE: GLOCKED_MODE_CHECK STRING "0"
199 // Retrieval info: PRIVATE: GLOCK_COUNTER_EDIT NUMERIC
    "1048575"
200 // Retrieval info: PRIVATE: HAS_MANUAL_SWITCHOVER STRING "1"
201 // Retrieval info: PRIVATE: INCLK0_FREQ_EDIT STRING "50.000"
202 // Retrieval info: PRIVATE: INCLK0_FREQ_UNIT_COMBO STRING "MHz"
203 // Retrieval info: PRIVATE: INCLK1_FREQ_EDIT STRING "100.000"
204 // Retrieval info: PRIVATE: INCLK1_FREQ_EDIT_CHANGED STRING "1"
205 // Retrieval info: PRIVATE: INCLK1_FREQ_UNIT_CHANGED STRING "1"
206 // Retrieval info: PRIVATE: INCLK1_FREQ_UNIT_COMBO STRING "MHz"
207 // Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "
    Cyclone IV E"
208 // Retrieval info: PRIVATE: INT_FEEDBACK__MODE_RADIO STRING "1"
209 // Retrieval info: PRIVATE: LOCKED_OUTPUT_CHECK STRING "0"
210 // Retrieval info: PRIVATE: LONG_SCAN_RADIO STRING "1"
```

```
211 // Retrieval info: PRIVATE: LVDS_MODE_DATA_RATE STRING "Not
    Available"
212 // Retrieval info: PRIVATE: LVDS_MODE_DATA_RATE_DIRTY NUMERIC
    "0"
213 // Retrieval info: PRIVATE: LVDS_PHASE_SHIFT_UNIT0 STRING "deg"
214 // Retrieval info: PRIVATE: LVDS_PHASE_SHIFT_UNIT1 STRING "deg"
215 // Retrieval info: PRIVATE: LVDS_PHASE_SHIFT_UNIT2 STRING "deg"
216 // Retrieval info: PRIVATE: MIG_DEVICE_SPEED_GRADE STRING "Any"
217 // Retrieval info: PRIVATE: MIRROR_CLK0 STRING "0"
218 // Retrieval info: PRIVATE: MIRROR_CLK1 STRING "0"
219 // Retrieval info: PRIVATE: MIRROR_CLK2 STRING "0"
220 // Retrieval info: PRIVATE: MULT_FACTOR0 NUMERIC "1"
221 // Retrieval info: PRIVATE: MULT_FACTOR1 NUMERIC "1"
222 // Retrieval info: PRIVATE: MULT_FACTOR2 NUMERIC "1"
223 // Retrieval info: PRIVATE: NORMAL_MODE_RADIO STRING "1"
224 // Retrieval info: PRIVATE: OUTPUT_FREQ0 STRING "100.00000000"
225 // Retrieval info: PRIVATE: OUTPUT_FREQ1 STRING "100.00000000"
226 // Retrieval info: PRIVATE: OUTPUT_FREQ2 STRING "100.00000000"
227 // Retrieval info: PRIVATE: OUTPUT_FREQ_MODE0 STRING "0"
228 // Retrieval info: PRIVATE: OUTPUT_FREQ_MODE1 STRING "0"
229 // Retrieval info: PRIVATE: OUTPUT_FREQ_MODE2 STRING "0"
230 // Retrieval info: PRIVATE: OUTPUT_FREQ_UNIT0 STRING "MHz"
231 // Retrieval info: PRIVATE: OUTPUT_FREQ_UNIT1 STRING "MHz"
232 // Retrieval info: PRIVATE: OUTPUT_FREQ_UNIT2 STRING "MHz"
```

```
233 // Retrieval info: PRIVATE: PHASE_RECONFIG_FEATURE_ENABLED
    STRING "1"
234 // Retrieval info: PRIVATE: PHASE_RECONFIG_INPUTS_CHECK STRING
    "0"
235 // Retrieval info: PRIVATE: PHASE_SHIFT0 STRING "0.00000000"
236 // Retrieval info: PRIVATE: PHASE_SHIFT1 STRING "30.00000000"
237 // Retrieval info: PRIVATE: PHASE_SHIFT2 STRING "60.00000000"
238 // Retrieval info: PRIVATE: PHASE_SHIFT_STEP_ENABLED_CHECK
    STRING "0"
239 // Retrieval info: PRIVATE: PHASE_SHIFT_UNIT0 STRING "deg"
240 // Retrieval info: PRIVATE: PHASE_SHIFT_UNIT1 STRING "deg"
241 // Retrieval info: PRIVATE: PHASE_SHIFT_UNIT2 STRING "deg"
242 // Retrieval info: PRIVATE: PLL_ADVANCED_PARAM_CHECK STRING "0"
243 // Retrieval info: PRIVATE: PLL_ARESET_CHECK STRING "0"
244 // Retrieval info: PRIVATE: PLL_AUTOPLL_CHECK NUMERIC "1"
245 // Retrieval info: PRIVATE: PLL_ENHPLL_CHECK NUMERIC "0"
246 // Retrieval info: PRIVATE: PLL_FASTPLL_CHECK NUMERIC "0"
247 // Retrieval info: PRIVATE: PLL_FBMIMIC_CHECK STRING "0"
248 // Retrieval info: PRIVATE: PLL_LVDS_PLL_CHECK NUMERIC "0"
249 // Retrieval info: PRIVATE: PLL_PFDENA_CHECK STRING "0"
250 // Retrieval info: PRIVATE: PLL_TARGET_HARCOPY_CHECK NUMERIC
    "0"
251 // Retrieval info: PRIVATE: PRIMARY_CLK_COMBO STRING "inclk0"
252 // Retrieval info: PRIVATE: RECONFIG_FILE STRING "dmf_PLL_2.mif
    "
```

```
253 // Retrieval info: PRIVATE: SACN_INPUTS_CHECK STRING "0"
254 // Retrieval info: PRIVATE: SCAN_FEATURE_ENABLED STRING "1"
255 // Retrieval info: PRIVATE: SELF_RESET_LOCK_LOSS STRING "0"
256 // Retrieval info: PRIVATE: SHORT_SCAN_RADIO STRING "0"
257 // Retrieval info: PRIVATE: SPREAD_FEATURE_ENABLED STRING "0"
258 // Retrieval info: PRIVATE: SPREAD_FREQ STRING "50.000"
259 // Retrieval info: PRIVATE: SPREAD_FREQ_UNIT STRING "KHz"
260 // Retrieval info: PRIVATE: SPREAD_PERCENT STRING "0.500"
261 // Retrieval info: PRIVATE: SPREAD_USE STRING "0"
262 // Retrieval info: PRIVATE: SRC_SYNCH_COMP_RADIO STRING "0"
263 // Retrieval info: PRIVATE: STICKY_CLK0 STRING "1"
264 // Retrieval info: PRIVATE: STICKY_CLK1 STRING "1"
265 // Retrieval info: PRIVATE: STICKY_CLK2 STRING "1"
266 // Retrieval info: PRIVATE: SWITCHOVER_COUNT_EDIT NUMERIC "1"
267 // Retrieval info: PRIVATE: SWITCHOVER_FEATURE_ENABLED STRING
      "1"
268 // Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING
      "0"
269 // Retrieval info: PRIVATE: USE_CLK0 STRING "1"
270 // Retrieval info: PRIVATE: USE_CLK1 STRING "1"
271 // Retrieval info: PRIVATE: USE_CLK2 STRING "1"
272 // Retrieval info: PRIVATE: USE_CLKENA0 STRING "0"
273 // Retrieval info: PRIVATE: USE_CLKENA1 STRING "0"
274 // Retrieval info: PRIVATE: USE_CLKENA2 STRING "0"
275 // Retrieval info: PRIVATE: USE_MIL_SPEED_GRADE NUMERIC "0"
```

```
276 // Retrieval info: PRIVATE: ZERO_DELAY_RADIO STRING "0"
277 // Retrieval info: LIBRARY: altera_mf altera_mf.
    altera_mf_components.all
278 // Retrieval info: CONSTANT: BANDWIDTH_TYPE STRING "AUTO"
279 // Retrieval info: CONSTANT: CLK0_DIVIDE_BY NUMERIC "1"
280 // Retrieval info: CONSTANT: CLK0_DUTY_CYCLE NUMERIC "50"
281 // Retrieval info: CONSTANT: CLK0_MULTIPLY_BY NUMERIC "1"
282 // Retrieval info: CONSTANT: CLK0_PHASE_SHIFT STRING "0"
283 // Retrieval info: CONSTANT: CLK1_DIVIDE_BY NUMERIC "1"
284 // Retrieval info: CONSTANT: CLK1_DUTY_CYCLE NUMERIC "50"
285 // Retrieval info: CONSTANT: CLK1_MULTIPLY_BY NUMERIC "1"
286 // Retrieval info: CONSTANT: CLK1_PHASE_SHIFT STRING "1667"
287 // Retrieval info: CONSTANT: CLK2_DIVIDE_BY NUMERIC "1"
288 // Retrieval info: CONSTANT: CLK2_DUTY_CYCLE NUMERIC "50"
289 // Retrieval info: CONSTANT: CLK2_MULTIPLY_BY NUMERIC "1"
290 // Retrieval info: CONSTANT: CLK2_PHASE_SHIFT STRING "3333"
291 // Retrieval info: CONSTANT: COMPENSATE_CLOCK STRING "CLK0"
292 // Retrieval info: CONSTANT: INCLK0_INPUT_FREQUENCY NUMERIC
    "20000"
293 // Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "
    Cyclone IV E"
294 // Retrieval info: CONSTANT: LPM_TYPE STRING "altpll"
295 // Retrieval info: CONSTANT: OPERATION_MODE STRING "NORMAL"
296 // Retrieval info: CONSTANT: PLL_TYPE STRING "AUTO"
```

```
297 // Retrieval info: CONSTANT: PORT_ACTIVECLOCK STRING "
    PORT_UNUSED"
298 // Retrieval info: CONSTANT: PORT_ARESET STRING "PORT_UNUSED"
299 // Retrieval info: CONSTANT: PORT_CLKBAD0 STRING "PORT_UNUSED"
300 // Retrieval info: CONSTANT: PORT_CLKBAD1 STRING "PORT_UNUSED"
301 // Retrieval info: CONSTANT: PORT_CLKLOSS STRING "PORT_UNUSED"
302 // Retrieval info: CONSTANT: PORT_CLKSWITCH STRING "PORT_UNUSED
    "
303 // Retrieval info: CONSTANT: PORT_CONFIGUPDATE STRING "
    PORT_UNUSED"
304 // Retrieval info: CONSTANT: PORT_FBIN STRING "PORT_UNUSED"
305 // Retrieval info: CONSTANT: PORT_INCLK0 STRING "PORT_USED"
306 // Retrieval info: CONSTANT: PORT_INCLK1 STRING "PORT_UNUSED"
307 // Retrieval info: CONSTANT: PORT_LOCKED STRING "PORT_UNUSED"
308 // Retrieval info: CONSTANT: PORT_PFDENA STRING "PORT_UNUSED"
309 // Retrieval info: CONSTANT: PORT_PHASECOUNTERSELECT STRING "
    PORT_UNUSED"
310 // Retrieval info: CONSTANT: PORT_PHASEDONE STRING "PORT_UNUSED
    "
311 // Retrieval info: CONSTANT: PORT_PHASESTEP STRING "PORT_UNUSED
    "
312 // Retrieval info: CONSTANT: PORT_PHASEUPDOWN STRING "
    PORT_UNUSED"
313 // Retrieval info: CONSTANT: PORT_PLLENA STRING "PORT_UNUSED"
314 // Retrieval info: CONSTANT: PORT_SCANACLR STRING "PORT_UNUSED"
```

```
315 // Retrieval info: CONSTANT: PORT_SCANCLK STRING "PORT_UNUSED"
316 // Retrieval info: CONSTANT: PORT_SCANCLKENA STRING "
    PORT_UNUSED"
317 // Retrieval info: CONSTANT: PORT_SCANDATA STRING "PORT_UNUSED"
318 // Retrieval info: CONSTANT: PORT_SCANDATAOUT STRING "
    PORT_UNUSED"
319 // Retrieval info: CONSTANT: PORT_SCANDONE STRING "PORT_UNUSED"
320 // Retrieval info: CONSTANT: PORT_SCANREAD STRING "PORT_UNUSED"
321 // Retrieval info: CONSTANT: PORT_SCANWRITE STRING "PORT_UNUSED
    "
322 // Retrieval info: CONSTANT: PORT_clk0 STRING "PORT_USED"
323 // Retrieval info: CONSTANT: PORT_clk1 STRING "PORT_USED"
324 // Retrieval info: CONSTANT: PORT_clk2 STRING "PORT_USED"
325 // Retrieval info: CONSTANT: PORT_clk3 STRING "PORT_UNUSED"
326 // Retrieval info: CONSTANT: PORT_clk4 STRING "PORT_UNUSED"
327 // Retrieval info: CONSTANT: PORT_clk5 STRING "PORT_UNUSED"
328 // Retrieval info: CONSTANT: PORT_clkena0 STRING "PORT_UNUSED"
329 // Retrieval info: CONSTANT: PORT_clkena1 STRING "PORT_UNUSED"
330 // Retrieval info: CONSTANT: PORT_clkena2 STRING "PORT_UNUSED"
331 // Retrieval info: CONSTANT: PORT_clkena3 STRING "PORT_UNUSED"
332 // Retrieval info: CONSTANT: PORT_clkena4 STRING "PORT_UNUSED"
333 // Retrieval info: CONSTANT: PORT_clkena5 STRING "PORT_UNUSED"
334 // Retrieval info: CONSTANT: PORT_extclk0 STRING "PORT_UNUSED"
335 // Retrieval info: CONSTANT: PORT_extclk1 STRING "PORT_UNUSED"
336 // Retrieval info: CONSTANT: PORT_extclk2 STRING "PORT_UNUSED"
```

```
337 // Retrieval info: CONSTANT: PORT_extclk3 STRING "PORT_UNUSED"
338 // Retrieval info: CONSTANT: WIDTH_CLOCK NUMERIC "5"
339 // Retrieval info: USED_PORT: @clk 0 0 5 0 OUTPUT_CLK_EXT VCC "
    @clk[4..0]"
340 // Retrieval info: USED_PORT: c0 0 0 0 0 OUTPUT_CLK_EXT VCC "c0
    "
341 // Retrieval info: USED_PORT: c1 0 0 0 0 OUTPUT_CLK_EXT VCC "c1
    "
342 // Retrieval info: USED_PORT: c2 0 0 0 0 OUTPUT_CLK_EXT VCC "c2
    "
343 // Retrieval info: USED_PORT: inclk0 0 0 0 0 INPUT_CLK_EXT GND
    "inclk0"
344 // Retrieval info: CONNECT: @inclk 0 0 1 1 GND 0 0 0 0
345 // Retrieval info: CONNECT: @inclk 0 0 1 0 inclk0 0 0 0 0
346 // Retrieval info: CONNECT: c0 0 0 0 0 @clk 0 0 1 0
347 // Retrieval info: CONNECT: c1 0 0 0 0 @clk 0 0 1 1
348 // Retrieval info: CONNECT: c2 0 0 0 0 @clk 0 0 1 2
349 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PLL_2.v TRUE
350 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PLL_2.ppf TRUE
351 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PLL_2.inc FALSE
352 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PLL_2.cmp FALSE
353 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PLL_2.bsf FALSE
354 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PLL_2_inst.v FALSE
355 // Retrieval info: GEN_FILE: TYPE_NORMAL dmf_PLL_2_bb.v FALSE
356 // Retrieval info: LIB_FILE: altera_mf
```

```
357 // Retrieval info: CBX_MODULE_PREFIX: ON
```

Listing I.9: PLL

I.10 DMF RISC Verilog Code Testbench

```
1  `timescale 1 ps/1 ps
2
3  module test;
4    reg Resetn_tb, Clock_tb;
5    reg [4:0] SW_in_tb;
6    wire [7:0] Display_out_tb;
7    integer i;
8
9  dmf_RISC621_cache_v top (Resetn_tb, Clock_tb, SW_in_tb,
       Display_out_tb);
10
11 initial begin
12 $sdf_annotate("sdf/dmf_RISC621_cache_v_tsmc18_scan.sdf", test.
       top);
13 //
       ------------------------------------------------------------------

14 //  Resetn_tb, Clock_tb, SW_in_tb, Display_out_tb
15 //
       ------------------------------------------------------------------

16 //-- Test Vector 1 (40ns): Reset
```

```
17 //
     ----------------------------------------------------------------------

18 for (i=0; i<5; i=i+1)
19   apply_test_vector(1, 0, 5'b00000);
20 //
     ----------------------------------------------------------------------

21 //-- All other test vectors
22 //
     ----------------------------------------------------------------------

23   for (i=0; i<800; i=i+1)
24     apply_test_vector(0, 0, 5'b00000);
25 end
26
27 task apply_test_vector;
28   input Resetn_int, Clock_int;
29   input [4:0] SW_in_int;
30
31   begin
32     Resetn_tb = Resetn_int; Clock_tb = Clock_int;
33     SW_in_tb = SW_in_int;
34     #20000;
35     Clock_tb = 1;
```

```
36        #20000;
37     end
38  endtask
39  endmodule
```

Listing I.10: testbench

I.11 Hello-World C Test Code

```
1  /*
       ----------------------------------------------------------------
2  //
       //
3  //   hello-world.c                                                 //
4  //                                                                 //
       //
5  //   This file is part of the Amber project                        //
6  //   http://www.opencores.org/project,amber                        //
7  //                                                                 //
       //
8  //   Description                                                   //
9  //   Simple stand-alone example application.                       //
```

```
10 //
   //
11 //  Author(s):                                                  //
12 //      - Conor Santifort, csantifort.amber@gmail.com           //
13 //
   //
14 ////////////////////////////////////////////////////////////////////////
15 //
   //
16 // Copyright (C) 2010 Authors and OPENCORES.ORG                 //
17 //
   //
18 // This source file may be used and distributed without         //
19 // restriction provided that this copyright statement is not    //
   //
```

```
20 // removed from the file and that any derivative work contains
       //
21 // the original copyright notice and the associated disclaimer.
       //
22 //
     //
23 // This source file is free software; you can redistribute it
       //
24 // and/or modify it under the terms of the GNU Lesser General
       //
25 // Public License as published by the Free Software Foundation;
       //
26 // either version 2.1 of the License, or (at your option) any
       //
27 // later version.
                                                          //
28 //
     //
29 // This source is distributed in the hope that it will be
           //
30 // useful, but WITHOUT ANY WARRANTY; without even the implied
       //
```

```
31 // warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
         //
32 // PURPOSE.  See the GNU Lesser General Public License for more
      //
33 // details.
                                                                  //
34 //


    //
35 // You should have received a copy of the GNU Lesser General
       //
36 // Public License along with this source; if not, download it
      //
37 // from http://www.opencores.org/lgpl.shtml
                           //
38 //


    //
39 ----------------------------------------------------------------
    */
40
41 /* Note that the stdio.h referred to here is the one in
42    mini-libc. This applications compiles in mini-libc
43    so it can run stand-alone.
44 */
```

```
45 #include "stdio.h"
46
47 main ()
48 {
49     //printf ("Hello, World!\n");
50     /* Flush out UART FIFO */
51     //printf ("                ");
52     //_testpass();
53     //
54   int x;
55
56   x = 5 ;
57   x = x + 1;
58 }
```

Listing I.11: Hello-World C Test Code