WSIM Configurable Digital Signal Processor Simulator/Debugger

by

Wayland Ni

B.S., Computer Science and Engineering (2003)

Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 20, 2004

Author____
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by____
Charlie Sakamaki
VI-A Company Thesis Supervisor

Certified by____
Chris Terman
M.I.T. Thesis Supervisor

Accepted by____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

WSIM Configurable Digital Signal
Processor Simulator/Debugger
by
Wayland Ni

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 2004

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

This M.Eng. Thesis presents a design and implementation of a full-featured configurable Digital Signal Processor (DSP) simulator/debugger. The user will be able to set configurations in order to model a specific architecture design. The simulator will have a command interpreter to listen to and process commands given by the user. When supplied with an assembly program, the simulator will allow the user to step through the execution of the program cycle by cycle, as well as calculate statistics like instruction, resource, and cache profiling. Some of the main features of the simulator are a multiply-accumulate unit, memory with direct and indirect offset addressing, and loop instructions.

M.I.T. Thesis Supervisor: Chris Terman
Title: Senior Lecturer, MIT EECS

VI-A Company Thesis Supervisor: Charlie Sakamaki
Title: Senior Staff Engineer/Manager, QUALCOMM Inc.

Acknowledgements

First, I would like to thank my QUALCOMM supervisor, Charlie Sakamaki, for his expertise and direction in guiding me throughout the course of the project. I really enjoyed working with you and the rest of the team members and have learned a tremendous amount.

Next, I would like to thank my thesis advisor, Chris Terman, for his patience and support in putting up with my habit of procrastinating and waiting until the last minute. I really appreciate everything you've done in such a short time span.

I would also like to thank the MIT EECS VI-A program for making it possible for me to work on an industrial-based thesis. I especially would like to thank Prof. Markus Zahn, Kathleen Sullivan, and Lydia Wereminski for doing a great job with the program and coordinating all the details with the companies.

In addition, I would like to thank my academic advisor, Prof. Jeffrey Shapiro, and VI-A Faculty Advisor, Prof. Lizhong Zheng, for checking up on my progress and making sure my time at QUALCOMM was a success.

Finally, I would like to thank all of my family and friends, for encouraging me through this long and difficult process. Without all of you, this thesis would not have been possible. Thank you!

Table of Contents

Table of Figures

Chapter 1

Introduction

A Digital Signal Processor (DSP) is a specialized computer processor used to process audio, video, and other analog signals which have been converted to digital form. The main difference between a DSP and a general-purpose processor is that a DSP is usually dedicated for specific kinds of applications. A DSP has features designed to support high-performance, repetitive, numerically intensive tasks [1]. For example, in cellular phone chipset solutions, a DSP is used for computationally intensive applications such as voice encoding/decoding, MP3 music file playback, MIDI synthesis, and 2D/3D graphics functions [2]. The performance acceleration of DSP processors is achieved by features that include:

- Capability for single-cycle multiply-accumulate; some high-performance DSPs often have two multipliers that allow two multiply-accumulate operations on the same instruction cycle
- Complex addressing modes, for example, pre- and post-modification of address pointers, circular addressing, and bit-reversed addressing
- Specialized program flow control. DSP processors often provide a loop instruction that reduces the loop overhead by not spending any instruction cycles on updating and testing the loop counter or on jumping back to the top of the loop. Additionally, tight loops allow a single instruction to be repeated without any extra loop overhead
- Irregular instruction sets, so several operations can be encoded in a single instruction. Instead of restricting each instruction to a single operation as in general-purpose processors, DSPs may encode two additions, two multiplications, and several data moves into a single instruction [1].

The need for more specially tailored DSP processors has been brought about by the growth of computationally intensive applications, especially in mobile devices, which need to have low power consumption, but maintain high performance. As DSP architecture designs become more specific and more complex, the associated costs with fabricating new prototypes will start to mount. However, with a software-based simulator/debugger, architecture designers will be able to test out their designs and execute sample programs without spending the money to fabricate a new prototype.

This thesis presents the design and implementation of WSIM, a configurable text-based DSP simulator/debugger for the purposes of prototyping a DSP architecture. It allows the user to model a specific DSP architecture by easily configuring factors like instruction set, memory, and pipeline setup. After configuration, the simulator reads a DSP assembly program and produces a cycle-accurate simulation of the program's execution, while providing profiling information, including instruction execution counts, hardware resource usage counts, and cache performance.

The organization of this thesis is as follows. Chapter 2 describes some of the tools and technologies used in implementing WSIM. Chapter 3 discusses the interface to the simulator, in terms of the inputs and outputs. Chapter 4 outlines the overview of the system's design. Chapter 5 explains the user interface and serves as a user's guide. Chapter 6 talks about the major blocks of the system architecture. Chapter 7 explores the implementation details of the advanced features. Chapter 8 illustrates some test cases and examples used to examine the functionality of the simulator/debugger. Chapter 9 briefly summarizes related work in the field. Chapter 10 looks at possible future work to be done and concludes the thesis. The Appendix contains some sample source code for the simulator/debugger.

Chapter 2

Tools Background

SystemC, Tcl, and C++ are the main tools/languages used in the implementation of WSIM. This section briefly provides some background information on SystemC and Tcl.

SystemC

SystemC is an extension of the C++ programming language that enables modeling of hardware descriptions. It adds concepts to C++ such as concurrent process execution, timed events and data types. The class library is not a modification of C++, but a library of functions, data types and other language constructs that are legal C++ code [3]. Overall, SystemC really simplifies the process of modeling a DSP architecture.

Tcl

Tcl, or “tool command language,” is a simple scripting language for controlling and extending applications [4]. The major benefit of Tcl that we take advantage of is that it is embeddable. It has an interpreter that is a library of C procedures, so it can easily be incorporated into applications. We may easily add or remove commands as we please from the interpreter to suit our needs.

Chapter 3

System Interface

At the highest level, WSIM is a black box that takes as input a DSP assembly source file and configuration information about the target processor and produces profiling information to the user, as shown in Figure 1. The next few sections will describe each of these pieces.

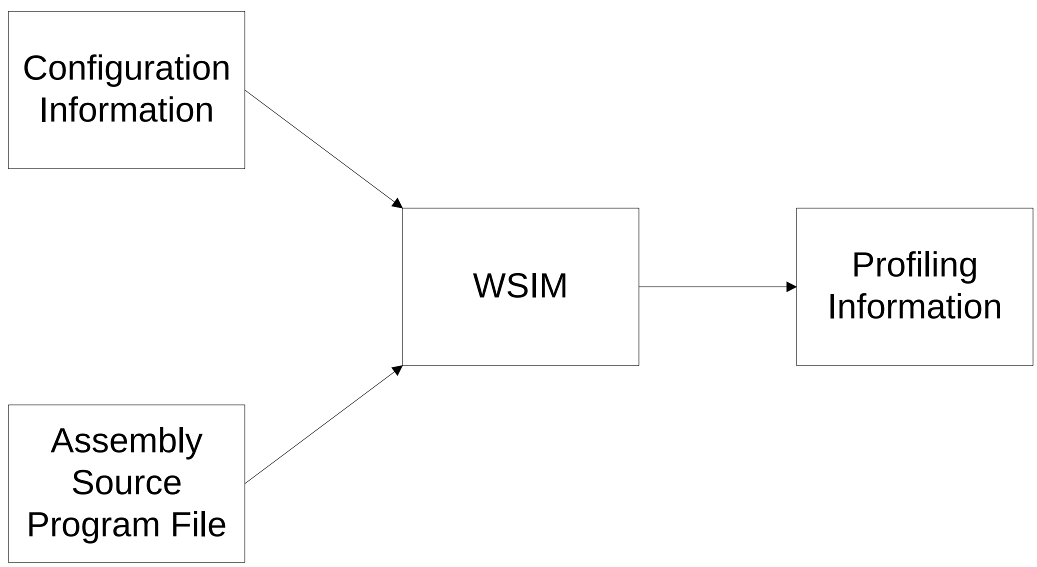

Figure 1: WSIM Interface Overview

Configuration Information

In order for WSIM to simulate the behavior and functionality of a particular DSP architecture design, we need to input the features and details of the design. These features include the register set, memory configuration, instruction set, and pipeline stages of the target processor. We will first introduce each of these features, and then show the details of the generic DSP we have chosen to model.

For the register set, the simulator needs to know the number of registers, the size of the registers, as well as whether the registers can be accessed partially. Partial access means that if only part of the register is needed, only part of the register is read from or written to. For example, Figure 2 shows a register file with four 32-bit registers R0-R4. However, if only the high 16 bits of a register are needed, we could use R0h, or if only the low 16 bits are needed, we could use R0l. In the design that we modeled, we decided to use eight 16-bit registers R0-R7, and four 32-bit long registers L0-L3, which can also be accessed partially with Lxh and Lxl. In addition, we have added four 16-bit address registers A0-A3 and four 16-bit address modifier registers AM0-AM3 to allow for advanced memory access methods which will be discussed later. To enable direct memory access (DMA), we also add four DMA pointers.

Register File

R0 (32)	R0h (16)	R0l (16)
R1 (32)	R1h (16)	R1l (16)
R2 (32)	R2h (16)	R2l (16)
R3 (32)	R3h (16)	R3l (16)

Figure 2: Sample register file with four 32-bit registers

In terms of memory configuration, we need to specify the number of memory segments used, the size and bit width of each segment, and whether each segment is random-access memory (RAM) or read-only memory (ROM). In our model, we just have one continuous segment of data memory. The segment is 64 KB of RAM with each address location storing 32 bits. In other processors, there may be up to three or more separate memory blocks, each with different parameters. Our instruction memory is not modeled like the data memory. Since no specific instruction encoding is used, we just have a simple array of `Instruction` data structures that store the assembly program.

The instruction set of a DSP can vary widely depending on the purpose of the specific DSP and the engineering tradeoffs in the design of the architecture. Instructions can usually be grouped into four broad types: computation, program flow, data move, and miscellaneous. Computation instructions include arithmetic logic unit (ALU) instructions such as add, subtract, and shift, multiply accumulate (MAC) instructions such as multiply and multiply-add/subtract combinations, as well as more specialized instructions like rounding, normalization, or filtering. Program flow instructions include jumps, branches, loops, function calls, interrupts, returns, and conditionals. Data move instructions involve loading from and storing to memory, and include register loads, immediate loads, direct loads/stores, and indirect loads/stores. Miscellaneous instructions can include null operation (NOP), stack instructions like pop or push, save and restore for context switches, and anything else the designer chooses. The instruction types we have chosen for our generic DSP are listed in Figure 3. Each instruction and its implementation will be explained in more detail later.

Instruction	Description
ADD R0 R1 R2	R0 = R1 + R2
ADDC R0 R1 immediate	R0 = R1 + immediate
SUB R0 R1 R2	R0 = R1 – R2
SUBC R0 R1 immediate	R0 = R1 - immediate
LDC16 R0 immediate	R0 = immediate (16 bits)
LDC32 L0 immediate	L0 = immediate (32 bits)
LSH R0 R1 R2	R0 = R1 << R2 (logical shift)
LSHC R0 R1 immediate	R0 = R1 << immediate (logical shift)
ASH R0 R1 R2	R0 = R1 << R2 (arithmetic shift)
ASHC R0 R1 immediate	R0 = R1 << immediate (arithmetic shift)
NOP	Null operation
MUL R0 R1 R2	R0 = R1 * R2
MULA R0 R1 R2 R3	R0 = R1 + R2 * R3
MULS R0 R1 R2 R3	R0 = R1 – R2 * R3
JMP label	Jump to label
MOV R0 R1	R0 = R1
BEQ label	Branch to label if ALU result == 0 (flags)
BNE label	Branch to label if ALU result != 0 (flags)
BLT label	Branch to label if ALU result < 0 (flags)
BLE label	Branch to label if ALU result <= 0 (flags)
BGT label	Branch to label if ALU result > 0 (flags)
BGE label	Branch to label if ALU result >= 0 (flags)

LDD R0 mem(address)	Load direct R0 = mem(address)
STD mem(address) R0	Store direct mem(address) = R0
LDI R0 A0 AM0	Load indirect and modify R0 = mem(A0); A0 = A0 + AM0
STI A0 AM0 R0	Store indirect and modify mem(A0) = R0; A0 = A0 + AM0
LDIO R0 A0 immediate	Load indirect with offset R0 = mem(A0 + immediate)
STIO A0 immediate R0	Store indirect with offset mem(0 + immediate) = R0
LDII R0 A0	Load indirect and increment R0 = mem(A0); A0 = A0 + 1
STII A0 R0	Store indirect and increment mem(A0) = R0; A0 = A0 + 1
LDID R0 A0	Load indirect and decrement R0 = mem(A0); A0 = A0 - 1
STID A0 R0	Store indirect and decrement mem(A0) = R0; A0 = A0 - 1
LDA DMA0 address	Load DMA address DMA0 = address
CALL function	Call function
RTF	Return from function
RTI	Return from interrupt
WAIT	Wait for one cycle
LOOPU label	Loop until label
LDLC R0	Load loop counter LC = R0
LDLCC immediate	Load loop counter LC = imm
TLOOP	Tight loop

Figure 3: Instruction set modeled in WSIM

Pipelining is an implementation technique that increases the instruction throughput of the processor. By dividing the pipeline into multiple stages, each stage can complete a part of a different instruction in parallel. Since multiple instructions are overlapped in execution, more instructions can exit the pipeline in the same amount of time. DSP pipelines can range anywhere from one stage to possibly seven or more stages. In our DSP model, we have chosen to work with a 3-stage pipeline. Figure 4 shows the three stages: fetch, decode, and execute. In the fetch stage, the processor computes the address of the next instruction and then proceeds to retrieve the next instruction from memory. In the decode stage, the processor figures out what the instruction does and what resources it will need. In the execute stage, the instruction is finally performed.

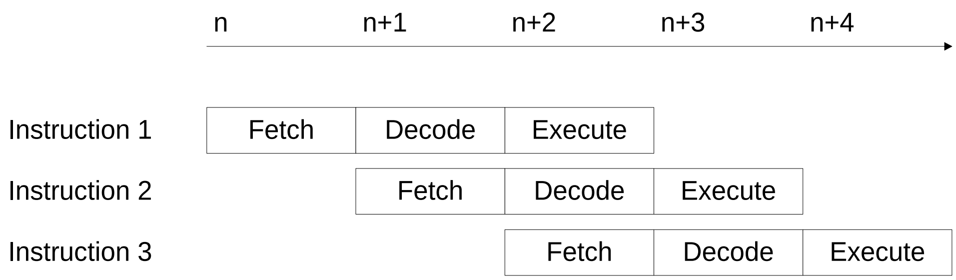

Figure 4: Three-stage pipeline

We have now been introduced to the four main parts of the configuration information needed to specify the target processor – register set, memory layout, instruction set, and pipeline stages – and have seen the details of the generic target processor we have modeled. Currently, these details have been hard-coded into the system, but are still relatively easy to modify. However, any changes to the processor design will require a recompilation of the code. One possible area of future work, which will be discussed in more detail later, is to allow a configuration file to specify the specifics of the processor at run-time. This work would involve designing a specification language, a parser for the configuration file, as well as a clean interface to the rest of the system.

DSP Assembly Source File

Another input to WSIM is a DSP assembly source file. At run-time, a source file may be loaded into the system for simulating and/or debugging. Source files consist of assembly instructions, program labels, user comments, and memory variables. The assembly instructions are chosen from the defined instruction set such as in Figure 3. Each instruction is listed on its own line and does not need any special characters to delimit it. Program labels are used to reference certain address locations the program can jump to. For instance, in order to call a function, the function label would need to precede the first instruction of the function. User comments are lines that begin with the string "##". Comments are used only for the programmer's benefit and are ignored by the simulator. Memory variables may be declared as single variables or arrays. An example of the declaration syntax is shown here:

```
#VAR aval=0x3535 22
#VAR bval[3]={0x01,0x02,0x03} 25
```

The first line sets the variable 'aval' to point to location 22 (or 0x16) in memory with initial value 0x3535. The second line sets the variable 'bval' to point to 3 consecutive locations starting with location 25 (or 0x19), with initial values of 0x01, 0x02, and 0x03. When declaring memory variables, the initial values are optional.

A couple examples of DSP test programs are included in Chapter 8. In future work, an improvement could be made in instruction formats. Instead of just using mnemonic instructions, a more complicated syntax may be developed. For example, assembly code resembling the C programming language can be much more readable to the user or programmer. As will be discussed later, this feature will require a more complex program parser.

Profiling Information

When testing a DSP architecture design, the engineer would like to know where the bottlenecks are and where the design could be made more efficient. The profiling information produced by WSIM could directly aid in this pursuit. The three classes of profiling we implement are instruction profiling, resource profiling, and cache profiling. In instruction profiling, the simulator simply keeps track of how many times each instruction is executed. Resource profiling remembers how many times each resource, which could be a register or memory location, is read from or written to. Resource profiling statistics are kept for the last cycle, a specified period, and for the entire execution of the program. The user may specify a period to start any time and may clear the statistics at any time. Based on these profiling results, the DSP designer can decide where to optimize the design. Chapter 9 also includes some example printouts of profiling statistics.

Chapter 4

System Overview

Now that we have seen the inputs and outputs of WSIM as a high level, we will examine the framework of the system. WSIM is a text-based application that has a real-time command interpreter. The implementation uses SystemC for the architectural design, Tcl for the command interpreter, and C/C++ for the instruction parser and other functionality. We will next introduce the main blocks in our SystemC architecture and see how the command interpreter fits into the simulator.

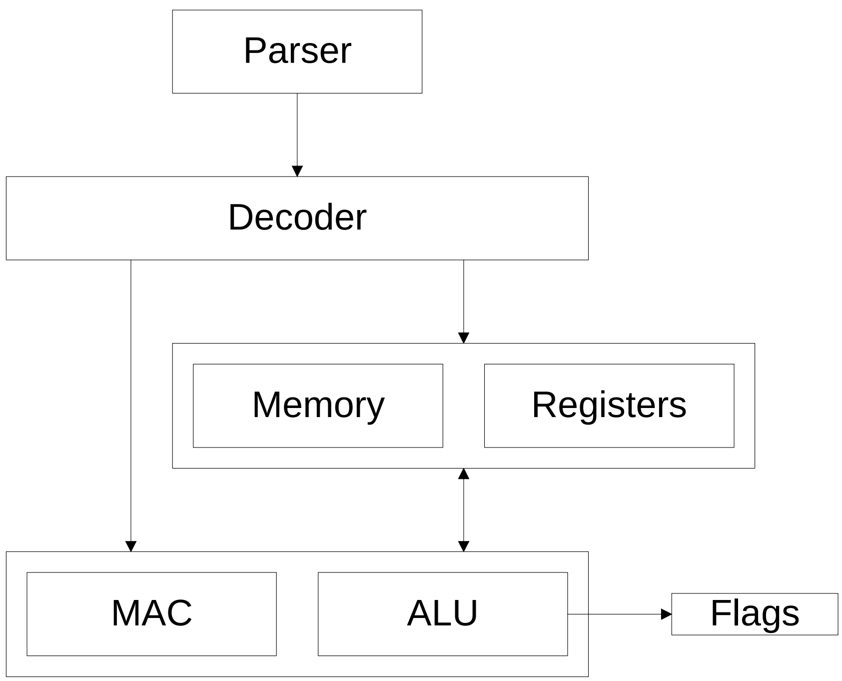

Figure 5: System overview

SystemC Processor Architecture

As shown in Figure 5, the main architectural blocks of the system are the parser, decoder, register file, memory, arithmetic logic unit (ALU), and multiply-accumulate unit (MAC). The flags module is a minor block that is just an addition to the ALU. The parser reads and parses an assembly program and stores the instructions in an instruction array. The decoder takes this instruction array and, upon command, executes each instruction one-by-one. For each instruction, the decoder must decide which registers or memory addresses, if any, must be read, as well as which functions, if any, the ALU or MAC must perform. The register file or memory then sends the corresponding data to the ALU or MAC, which in turn executes the specified operation and passes the result back to the register file or memory structure for write-back purposes. With the exception of the parser, each of these blocks is its own SystemC module. Since the parser is so tightly coupled with the decoder, we have just included the parser in the decode module.

Tcl Command Interpreter

Though the architecture of the processor is modeled with SystemC, the main driver of the simulator is the Tcl command interpreter. The user must issue commands to the interpreter, which will in turn run the parser, send instructions through the pipeline, or get profiling statistics. Upon startup, the system will begin execution in function sc_main. Function sc_main proceeds to instantiate each of the SystemC modules and connect them with signals. Then, the initialization sequence sets up all the hardware resources and internal data structures, as well as the Tcl interpreter. At this point, the system enters the command parser's infinite loop, which repeatedly issues a command prompt and processes commands. Pseudo code of this loop is shown below:

```
while(1) {
      Print command prompt
      Get command
      Evaluate command
}
```

We take advantage of the configurability of the Tcl interpreter by adding our own custom commands and getting rid of the commands we do not want. The next chapter, chapter 5, will describe each of the commands we implemented and serve as a user guide by showing how to use each of the commands.

Chapter 5

Tcl Interpreter Commands

In the last chapter, we explained that WSIM uses the Tcl command interpreter to drive the system. Figure 6 shows a table of all the commands we have implemented. This chapter will serve to explain each of these commands in more detail.

Command	Description
accounting clear	Clear used/assigned resources statistics
breakpoint	Set or delete a breakpoint
cache	Instruction or memory cache operation
continue	Run simulation until breakpoint
dma	Direct memory access
dump	Memory dump
getval	Get resource value
help	Print help
instructions	Print all instructions
interrupt	Set external interrupt
list	List program file
load	Load program file
lsr	List, Step, Register
profile	Instruction profile statistics
register	Print registers
reset	Reset simulator
run	Run until breakpoint
setval	Set resource value
stack	Print PC and loop stack data
statistics	Display resource usage statistics
step	Simulate one or more cycles

Figure 6: Table of interpreter commands

Clear Resource Statistics

Usage:

```
accounting clear
```

Resets the resource usage statistics for the period. It does not affect the cycle statistics or the total statistics. Cycle statistics only account for the last cycle executed. Total statistics are kept for the entire duration of the program. Period statistics can be reset by the user anytime, so the user may look at resource usage results from any point in the program to another.

Breakpoints

Usage:

b/breakpoint (<line_num>/label <label>)

b/breakpoint del/delete (<line_num>/label <label>/all)

b/breakpoint list

The `breakpoint` command is used to set, delete, or list breakpoints. The first version of the command sets a breakpoint at either a specific line number or at a given program label. The second version deletes the breakpoint at a given line number or program label, if one exists. The third version of the command just prints a list of all the existing breakpoints.

Caching

Usage:

[cache] is either ‘cache’ or ‘cachemem’

[cache] (on/off/lru/reset_stats/stats/tags)

[cache] delay <cycles>

[cache] log (stdout/<filename>)

[cache] size <total_cache_size> <block_size>

[cache] type (direct_mapped/fully_assoc/2_way/4_way)

[cache] algo (lru/random/lrr)

[cache] wbdelay <cycles>

The `cache` command and `cachemem` command are used to set up and simulate the instruction cache and data memory cache, respectively. The first version of the command allows the user to turn the cache on or off, show the statistics, reset the statistics, show the tags in the current cache, or show the least recently used array. The second version sets the cache controller delay on a cache miss. The third version allows the user to specify whether the cache notifications should be written to standard output or a file. The fourth version, which must be called before the cache is turned on, is used to set the total size of the cache and the size of each block. The fifth version lets the user specify the type of cache: direct_mapped (default), fully associative, 2-way set-associative, or 4-way set-associative. The sixth version of the command specifies which replacement strategy to use for the associative caches: least recently used, least recently replaced, or random. The final version of the command tells the simulator how long the write back delay should be.

Continue

Usage:

c/continue

Continues execution until a breakpoint is hit or the end of the program is reached.

Direct Memory Access

Usage:

dma <channel> (read/write) <filename> [<start> [<period>]]

Schedules a future DMA read or write request on one of four channels. The argument <start> specifies the cycle number the request should occur on. The argument <period> specifies that a DMA request should occur periodically every <period> cycles after <start>. If <start> is not given, the DMA request should occur on the ensuing cycle. If <period> is not given, the DMA request is a one-time event. A read request reads from the specified file and writes to the memory location pointed to by the specific channel's DMA pointer. A write request reads from that memory location and writes to the file.

Memory Dump

Usage:

dump [(<var>/mem) [<begin> [<end>]] [r <radix>]]

Dumps the contents of memory to the screen. If a variable name is given, the dump begins at the address of the variable. Otherwise, the dump starts at the address <begin> and goes until the address <end>. The default for <begin> is 0 and the default for <end> is <begin> + 128. The argument <radix> specifies whether the data should be displayed in decimal, hexadecimal (default), octal, or binary. If the command is called without any arguments, the next 128 values are displayed.

Get Resource Value

Usage:

getval (<resource>/cycle/FPC/DPC/EPC)

Returns the value of the specified resource or one of the special variables: cycle number, fetch program counter, decode program counter, or execute program counter.

Print Help

Usage:

h

help [<command>]

The command `h` displays a shortened version of help. The command `help` displays the detailed version of help. If <command> is specified, command-specific help is displayed.

Print Instructions

Usage:

instructions

Displays a list of every instruction in the instruction set.

External Interrupts

Usage:

interrupt (<addr>/<label>) [<start> [<period>]]

This command allows the user to simulate an external interrupt request. The interrupt handler can be specified either by its address or by its label. The <start> argument is the cycle number the interrupt is to occur on and <period>, if given, is the number of cycles until the interrupt request should occur again. If <period> is not given, the interrupt is a one-time event. If <start> is not given, the interrupt should occur on the following cycle.

List Program

Usage:

l/list

Prints the program source text from four lines before the earliest program counter to four lines after the latest program counter.

Load Program

Usage:

load <filename>

Loads a program file for simulation.

List, Step, Register

Usage:

lsr

This command is a combination of three other commands: list, step, and register. First, the program source file is displayed. Second, the system simulates one instruction cycle. Third, all of the registers are displayed.

Instruction Profiling

Usage:

p/profile [clear]

The `profile` command either prints out instruction profiling statistics or clears the statistics.

Print Registers

Usage:

r/register

Displays the values of all the registers in WSIM.

Reset Simulator

Usage:

reset

Resets all the hardware resources in WSIM.

Program Run

Usage:

run

Starts execution of the program until either a breakpoint is hit or the end of the program is reached.

Set Resource Value

Usage:

setval (<resource>/cycle) <val>

Set the value of <resource> or the cycle count to <val>.

Print Stack

Usage:

stack

Displays the program counter stack, loop counter stack, loop start stack, and loop end stack.

Resource Statistics

Usage:

stats/statistics [all/reg/mem/<resource...>]

Displays statistics of all resources, just the registers, just the memory, or just one specific resource.

Program Step

Usage:

s/step [<cycles>]

Simulates the execution of the program file for <cycles> number of cycles, or one cycle if <cycles> is not specified.

Chapter 6

SystemC Architecture Modules

From Figure 5 above, we have seen how the SystemC modules fit together to form the framework of the processor. In this chapter, we look at each of the modules separately and in more detail.

Instruction Parser

When the command interpreter receives a `load program` command, the instruction parser is called to read in the assembly program file. As mentioned before, everything in the program file should fall under one of four categories: instruction, label, comment, and memory variable. The parser reads in the file, one string at a time, delimited by white space, and decides which of the four categories that string falls under.

If the processed string is ‘#VAR’, it must be the beginning of a memory variable declaration. The parser then checks whether the variable is a single variable or an array, and whether an initial value is present. If an initial value is present, the value is written into the location in memory specified by the address. Then a new MemVar object is created and added to the global array of MemVar objects. Finally, the parser moves on to the first string of the next line.

If the first string of a line begins with the character sequence ‘##’, it must be a comment line. The parser then reads in the rest of the line and discards it, moving on to the first string of the next line.

If the first string of a line ends with the colon character ‘:’, that line is a program label. After checking for duplicate labels, a new Label object is created and added to the global array of labels. The parser then proceeds to the next line.

If a string does not fall under any of the first three categories, it must either be an instruction or an error by the assembly programmer. The string is checked against the list of instructions in the instruction set, and if a match is not found, an error statement is printed to the screen and the string is skipped. The parser would then go on to the next string, either on the same line, if it exists, or on the following line. If the first string does match with an instruction, a new Instruction object is created and added to the global array of instructions. The parser next scans the following strings to get the arguments of the instruction. Each Instruction object has a corresponding array for arguments. Since different instructions have different numbers of required arguments, we need to fill up the empty argument slots with the empty string "". In addition, when all the instructions have been read, we need to fill the empty instruction slots with "fake" instructions. These fake instructions and empty arguments just serve as placeholders and help avoid null pointer exceptions later in the pipeline.

Internal / External System Clock

There are actually two different clocks in WSIM: an external DSP instruction clock and an internal SystemC clock. The external instruction clock is the slower clock such that one instruction cycle is the same as NUM_CYCLES number of internal SystemC cycles, where NUM_CYCLES is defined in the header file *global.h*. When a `step` or `run` command is received by the interpreter, the interpreter calls the `clk_step` function in *main.cpp*. The `clk_step` function is basically a wrapper that simulates one external DSP instruction cycle by triggering the internal SystemC clock to run for NUM_CYCLES number of cycles. The SystemC clock is connected to each of the SystemC modules and triggers each of them to run for one cycle. Thus, for each processor instruction cycle we want to elapse, we need to simulate NUM_CYCLES number of SystemC cycles. This design was actually more complicated than it needed to be and is another candidate for possible future work. The simpler design would be to just have one SystemC cycle be equivalent to one processor instruction cycle. This idea will be discussed further in the chapter on future work.

Decoder

The decode module has a thread called `generate` that is sensitive to the SystemC clock signal. The `generate` thread is an infinite loop such that one processor instruction cycle, or NUM_CYCLES SystemC clock cycles, will result in a complete iteration of this loop. This behavior is accomplished by inserting NUM_CYCLES number of `wait` instructions in the loop. When triggered by the SystemC clock signal, the thread runs until it reaches a `wait` instruction and suspends. On the next clock cycle, the module resumes from right after the `wait` and keeps running until the next `wait` instruction. Thus, after NUM_CYCLES number of SystemC clock cycles, the point of execution will be at the same exact point in the loop. All of the SystemC modules are modeled in the same way.

At the simplest level, the execution loop in the `generate` thread carries out several tasks. First, it gets the program counter (PC) from the PC register. The PC is the address of the next instruction to be executed. Second, it looks up the instruction located at the address pointed to by the PC. Then, based on the specific instruction, the thread figures out what signals it needs to send to the MAC, ALU, register, and memory modules. Specifically, the ALU and MAC need to know what function to carry out and which inputs to accept, and which module to send the result to. The register and memory modules need to know which registers and memory locations to read from or write to. The final task the decode thread needs to complete is to increment the PC. These tasks comprise the most basic tasks necessary to simulate a simple processor. All the additional features we have implemented have added many more modifications to the decode module and will be described in detail in the chapter on features implementation.

To decide which signals to send to the other modules, we have implemented a long, straightforward `if-else if` structure with each specific instruction getting a block, much like a `switch-case` construct. Here is an example of the structure:

```
if (instruction == "ADD") {
...
} else if (instruction == "SUB") {
...
```

```
} else if (instruction == “MUL”) {
...
```

One more area of improvement in future work that will be discussed later is to get rid of this structure completely. If we want to be able to configure the instruction set of the target processor at runtime, we will not be able to use a structure like this. In the configuration file, we would have to convey which signals to send for each instruction type.

Register File

The register file has three read ports and one write port. The only job of this module is to output the contents of a register when needed and to write to registers when requested. The register file module has a thread called `update` that is sensitive to the SystemC clock signal. The `update` thread is an infinite loop like the `generate` method in the decode module. The loop just waits for the read signal, which specifies which register is to be read from, and the write enable signal, which specifies which register is to be written to. Additionally, there is a modification that will be described later to allow for read/write access to either the high bits or the low bits of a register.

Memory

The memory module is almost identical to the register file module, except that it only has one read port and one write port. It waits for signals from the decode module to decide which address location needs to be read from or written to, as well as whether the data comes from the ALU or MAC units.

Arithmetic Logic Unit

The ALU module waits for signals from the decode module and then determines which inputs to select, which function to perform, and where to send the output. A block diagram of the ALU is shown in Figure 7.

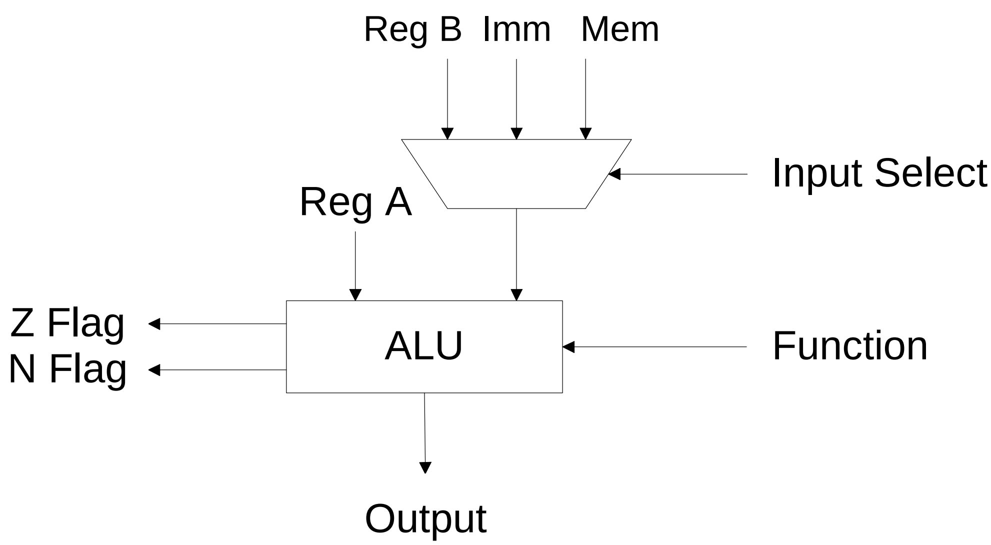

Figure 7: Block diagram of ALU

After the inputs are received, the module must sign extend the inputs because we are assuming that all data is signed. Since our inputs may be of varying bit lengths, such as 16, 32, or 64 bits, we just sign extend everything to a standard of 64 bits, and then perform all calculations as if all the inputs are 64 bits in length. After we perform the calculations, we mask the result to the correct length of the output, and send the result to the proper module. Depending on the result, we also send signals to the flag module to show if the result was zero or not, and if the result was negative or not. These flags are used in conditional branch instructions. The source code for the ALU SystemC module is included in the Appendix.

Multiply Accumulate

The MAC module is very similar to the ALU module. It reads in three inputs, sign extends them to 64 bits, and performs an operation on them. The result is masked down to 32 bits and is sent to the register module and memory module. There are no flags associated with the MAC. A block diagram of the MAC module is shown below in Figure 8.

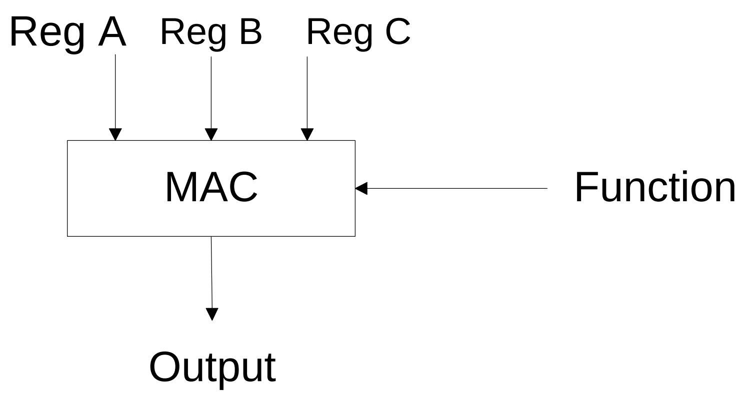

Figure 8: Block diagram of MAC

Since MAC instructions only accept inputs from registers, we do not need a multiplexer and an input select signal like in the ALU module.

Flags

The flags module is similar to the register module, but it only has to store two values, the zero flag and the negative flag. The zero flag being true indicates that the last result computed by the ALU was zero. The negative flag being true indicates that the last result computed by the ALU was negative. A combination of these two flags tells whether the last ALU result was positive, zero, or negative. These flags are used in conditional instructions such as BEQ, BLT, and BGE. BEQ, which stands for "branch equal," instructs the processor to jump if the last ALU result was equal to 0. BLT, which stands for “branch less than,” instructs the processor to jump if the last ALU result was less than zero. Finally, BGE, which stands for “branch greater than or equal to,” instructs the processor to branch if the last ALU result was greater than or equal to zero.

Chapter 7

Features Implementation

In this section, we introduce all of the advanced features of the simulator and describe how each is implemented.

Resource Value Representation and Resource Profiling

Hardware resources such as registers, memory, and ALU flags are represented by a `struct` data type called `RES_Value`. Each resource has a read value and a write value. This dual nature was designed to support parallel instructions, a popular feature of DSP architectures that may be added in the future. For example, consider the following parallel instruction:

```
R0 = R1, R2 = R0 + 1
```

If each resource only had one value instead of a read value and a write value, the result of the instructions would be different based on the order of execution of the two instructions. Since a parallel behavior is desired, the first instruction assigns R1 (read) to R0 (write) and the second instruction adds 1 to the value of R0 (read) before assigning the result to R2 (write). At the end of every cycle, the write value of each resource is copied to the read value.

The `RES_Value` data type also includes two separate pointers to two other `RES_Value` data types, named `AllResourcesNext` and `AllRegistersNext`. This way, one resource is linked to the next resource, in effect creating a linked list of all the resources in the system. This method makes it much easier to go through and perform an operation on all the resources. The head of the list is pointed to by the global variable `AllResourcesList`. The second list, pointed to by `AllRegistersList`, only includes the registers. Since every single memory location is represented as a resource and is included in the list `AllResourcesList`, the list `AllRegistersList` allows a more efficient search for a specific register because every single location of memory does not need to be traversed.

The remaining fields in the resource value data type are all used to support resource profiling. Two `bool` data types, `CycleUsedFlag` and `CycleAssignedFlag`, are used to indicate whether the resource was used or assigned to in the last instruction cycle. Four more `long` data types, `PeriodUsedCount`, `PeriodAssignedCount`, `TotalUsedCount`, and `TotalAssignedCount`, count the number of times the resource was used in the current period, assigned to in the current period, used in the entire program, and assigned to in the entire program, respectively. Two more pointers to `RES_Value` data types, `CycleAssignedNext` and `CycleUsedNext`, help connect a linked list of all the resources that were assigned to or used in the past cycle. Thus, the simulator only needs to traverse these reduced lists when updating the read and write fields of all the resources that were written to in each cycle.

Instruction Execution Profiling

Instruction profiling is implemented by adding a `count` variable in the `Instruction` class. Then, in the main loop of the decode module, we increment the variable for the instruction in the Execute stage of the pipeline.

Breakpoints

Breakpoints are used to help debug a program. The user may set a breakpoint on a specific instruction, either by giving the line number of the instruction or the name of the label directly preceding the instruction, if one exists. Like instruction profiling counts, breakpoint information is kept in the `Instruction` class. Each `Instruction` object has a flag that says whether a breakpoint exists at that instruction.

During each cycle of program execution, the simulator must check to see if a breakpoint has been reached. This check is performed in the main loop of the decode module. Before an instruction is simulated in the execution stage of the pipeline, the simulator makes sure there is no breakpoint set at that instruction. If a breakpoint does exist, the simulator ceases execution of the instruction and returns control to the command interpreter. The user must then command the simulator to start running again before that instruction gets executed.

Continuous Simulation

Usually, the user uses the `step` command or `lsr` command to step through one instruction at a time. If the user just wants to simulate the program indefinitely until a breakpoint is hit, the `run` or `continue` command will suffice. This behavior is implemented by using a global flag variable `SimContinuous` and letting the program keep running until the flag's value is set to false. The `while` loop is added in the `clk_step` function in *main.cpp*, around the `for` loop that triggers the SystemC clock. This portion of the code is shown here:

```
while(SimContinuous) {
        for (int i=0; i<NUM_CYCLES; i++) {
                clk.write(1);
                sc_cycle(10 NS);
                clk.write(0);
                sc_cycle(10 NS);
        }
}
```

The `SimContinuous` flag is set to false in the main loop in the decode module whenever a breakpoint is hit or when the final instruction has been executed. Otherwise, this loop will keep running.

Pipelining

As mentioned earlier, we have modeled a three-stage pipelined processor with a fetch stage, a decode stage, and an execute stage. To simulate this behavior, we first need three program counters, one for each stage: FPC, DPC, EPC. Since we also need to execute some tasks on the decode stage, we will need a separate `if-else if` structure to match instructions in the decode stage, very similar to the existing `if-else if` structure for instructions in the execute stage. This structure is added in the same place in *decode.cpp*.

In a pipelined architecture, certain instructions, such as jumps or branches, will require instructions already in the pipeline to be cancelled or flushed. An example is shown in Figure 9.

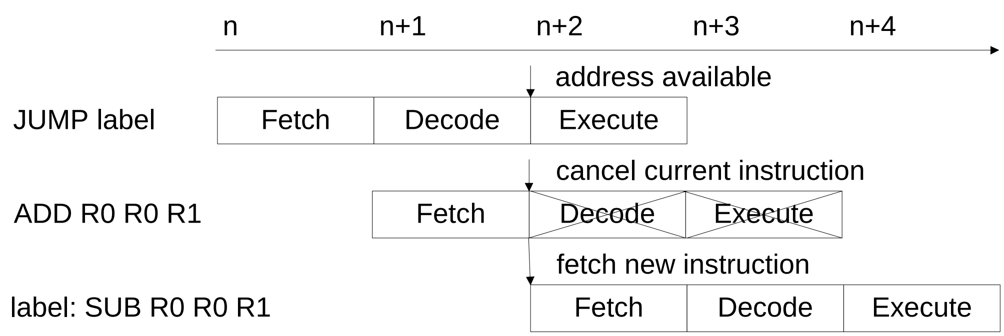

Figure 9: Pipeline Flow for Jump Instruction

In this example, the address of the location to jump to is not known until the beginning of cycle n+2. However, the instruction following the jump is already in the pipeline. Thus, we need a mechanism of flushing the instruction in the decode stage this cycle as well as the instruction in the execute stage in cycle n+3. Then, the processor is able to fetch the instruction from the new location.

We implement the flushing mechanism by using a global variable `Flushed` that keeps track of which stages need to be flushed. The first bit in `Flushed` represents the execute stage, while the second bit represents the Decode stage. The fetch stage will not ever need to be flushed. If we want to flush the instruction in the decode stage, we OR the variable with the number 2: Flushed = Flushed | 2. If we want to flush the instruction in the execute stage, we OR the variable with the number 1: Flushed = Flushed | 1. At the end of every cycle, the bits of the variable are shifted one bit to the right: Flushed = Flushed >> 1. Now we just have to examine the variable before we execute the decode and execute stages in the decode module. Here is the pseudo code that makes it work:

```
if (Flushed & 0x2)
     replace decode instruction with NOP
if (Flushed & 0x1)
     replace execute instruction with NOP
...
Flushed = Flushed >> 1
```

With a pipelined processor, there must also be some restrictions on the order of instructions. For example, there must be a NOP between an ALU instruction and a conditional branch instruction or a call instruction. Figure 10 shows what would happen if there was no NOP between the ALU and a BEQ instruction.

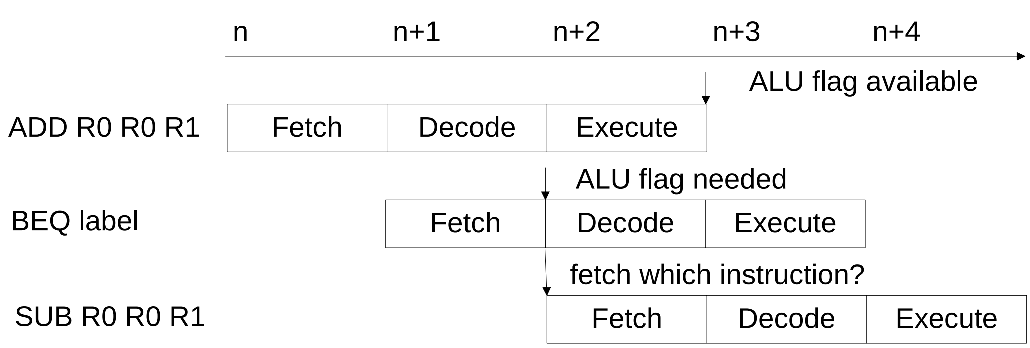

Figure 10: No NOP between ADD and BEQ

The ALU flags are not set until after the execute stage of the ADD instruction. However, the BEQ instruction needs the flags before the decode stage so the processor knows which instruction to fetch next. There will be a conflict if the NOP is not inserted. Figure 11 shows what would happen with the NOP.

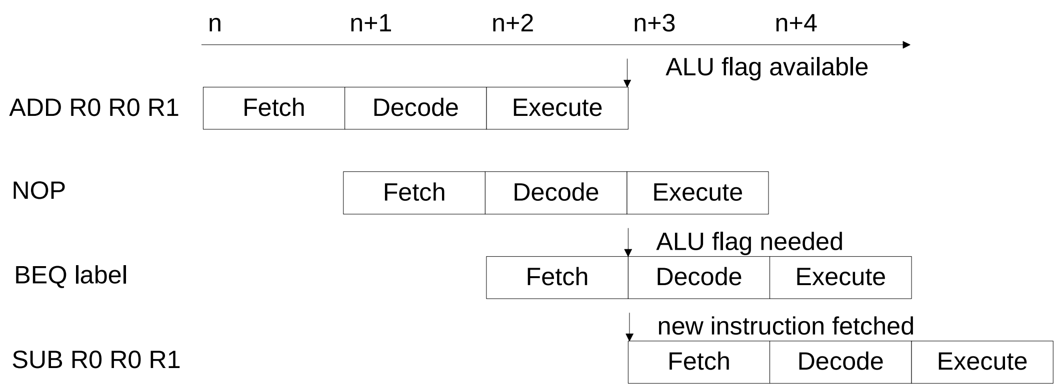

Figure 11: NOP inserted between ADD and BEQ

Since the NOP is added, the BEQ instruction does not need the value of the ALU flags until one cycle later, exactly when the flags will be available. Restrictions like these must be checked for in the program file parser or else unpredictable program behavior may occur. With the addition of parallel instructions and longer pipelines, the number of conflicting instructions will keep rising.

Function Calls

Like JUMP and BRANCH instructions, a CALL instruction just tells the processor to start executing at another location, specified by a label. However, unlike JUMP and BRANCH instructions, a function call saves the address of the instruction immediately following the call and is able to return to that instruction when the function concludes. The RTF instruction, or return from function, is used to end the function and return.

If the program has a recursive function or a series of nested function calls, the processor would need to save multiple addresses at once, and also remember the order of the function calls. To apply this functionality, we chose to implement a program counter (PC) stack. Whenever a function call occurs, the address of the return instruction is pushed on to the stack. When a function returns, the address is popped off the stack. The depth of the stack is defined by a global variable called `PC_STACK_SIZE` in the file *global.h* and is currently set to 16.

We also need to save the state of the ALU flags on calls and restore the flags on returns. Since we only use 16-bit addresses and can store 32 bits at each location in the stack, we can let the 17^{th} and 18^{th} bits represent the Z flag and the N flag respectively.

Loop Instructions

Loop instructions are a way to support zero-overhead program loops. After the loop counter LC is loaded, the loop until instruction (LOOPU) specifies a label indicating the last instruction in the loop. For example, consider this two-instruction loop:

A: LDLCC 10
B: LOOPU done
C: ADDC R0 R0 1
D: done: ADDC R1 R1 1

The order of instructions executed would be A, B, C, D, C, D, C … Each additional iteration requires two instruction cycles. Now let us look at the same two-instruction loop without using the loop instructions:

A: start: ADDC R0 R0 1
B: ADDC R1 R1 1
C: JUMP start
D: NOP

The order of execution would be A, B, C, D, A, B, C, D … Thus, every iteration requires four instruction cycles. The NOP serves as a placeholder because whatever instruction that follows the JUMP instruction will be cancelled.

To implement the loop instructions, the processor records and saves the loop start and loop end address when the LOOPU instruction is called. During each instruction cycle, the decode module must check to see if the loop end address is the same as the fetch program counter address. If the addresses are the same, and the loop counter is greater than one, the next instruction in the fetch stage becomes the one at the loop start address and the loop counter is decremented.

Since nested loops are possible, we need to use stacks like with the program counter. We implement three stacks: the loop counter stack, the loop start stack, and the loop end stack. When the LOOPU is executed, the loop counter, loop start, and loop end values are all pushed on to their respective stacks. When we exit a loop, or when the loop end is reached and the loop counter is not greater than one, we pop the values off all three stacks. The depth of the stacks is defined by a global variable called `LOOP_STACK_SIZE` in the file *global.h* and is currently set to 16.

An additional instruction, TLOOP, allows for one-instruction loops with zero-overhead. The only difference is that the loop start address and loop end address are identical.

External Interrupts

As mentioned earlier, the user may schedule either one-time or periodic external interrupts to occur in the future. We represent the interrupt requests with a linked list of `Interrupt` objects called `AllScheduledInterrupts`. Each `Interrupt` object stores the cycle number of its next scheduled request, the periodicity, the address of the interrupt handler, and a pointer to the next `Interrupt` object in the linked list. A scheduling algorithm is used to sort the linked list in order of time until the next interrupt. When the user schedules the interrupt, the `Interrupt` object is inserted into the correct spot in the list. When an interrupt occurs, the object is taken out from the front of the list. Then, if the interrupt is periodic, the next request time is calculated and the object is inserted back into the list at the proper location. If the interrupt is one-time, the object is discarded.

At the beginning of each instruction cycle, the decode module checks the first element of the interrupt linked list to see if an interrupt is happening that cycle. If so, the PC and ALU flags are saved on the PC stack, much like a function call. The simulator then jumps to the interrupt handler and must cancel and flush the proper instructions. For example, if the instruction in the execute stage is a program flow instruction such as

JUMP or BRANCH, all three stages are flushed. Otherwise, the instruction in the execute stage is allowed to execute and the instructions in the fetch and decode stage are flushed. When the interrupt handler completes, it calls the RTI function, which pops the PC stack and returns to the correct instruction.

Direct Memory Access

DMA requests are very similar to interrupt requests. The user specifies when the request should occur, whether the request is one-time or periodic, whether the request is a read or write, and the channel number to use. Like interrupt requests, all of this information is stored in `DMA` objects, which are linked together in a list in order of time until execution. The same scheduling algorithm is also used. As with interrupts, DMA requests are checked at the beginning of each instruction cycle. However, unlike interrupts, when a DMA request occurs, the pipeline performs a vertical stall, instead of flushing or canceling instructions. A vertical stall means that all instructions stay in the same pipeline cycle and are suspended until the DMA is complete. An example timing diagram is shown in Figure 12.

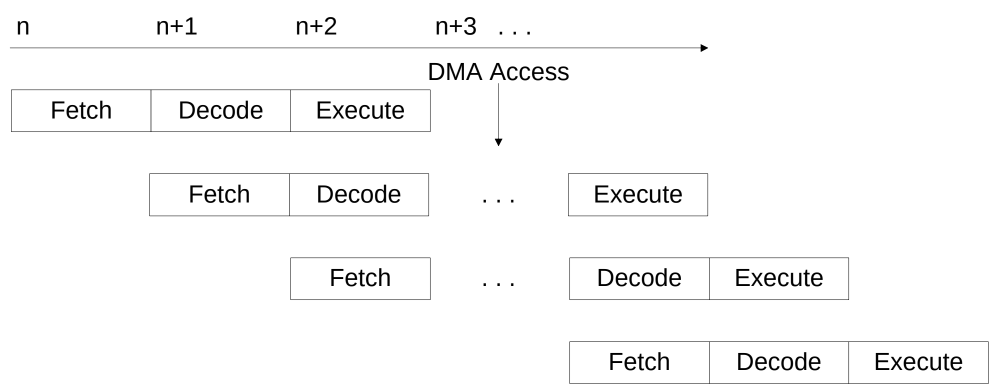

Figure 12: Timing diagram of DMA access

During the DMA access, data is either read from a file and written to memory, or read from memory and written to the file. Depending on the specified channel (0-3), the respective DMA pointer (DMA0-DMA3) is used to point to the read/write location in memory.

Cache Simulation

The user may simulate instruction or memory cache by specifying several parameters: total cache size, block size, type of cache, replacement strategy, and length of write back delay. When the cache is turned on, the decode module must call the function `SimICache` at the beginning of every instruction cycle. This function checks for a cache hit or miss, and updates the statistics accordingly. The straightforward representation of the cache just uses several multi-dimensional arrays to keep track of tags, least recently used queues, and least recently replaced queues. The results of the cache simulation have no effect on the rest of the simulator.

Chapter 8

Testing

The simulator was tested with several DSP assembly programs. Most of the tests were very basic and just tested the functionality of specific instructions. However, a couple of the tests were a little more interesting. The list of tests is shown in the following table.

Test code name	Purpose
ash	Tests ASH instruction
ashc	Tests ASHC instruction
beq	Tests BEQ instruction
bge	Tests BGE instruction
bgt	Tests BGT instruction
ble	Tests BLE instruction
blt	Tests BLT instruction
bne	Tests BNE instruction
ldst	Tests memory variables and all memory access methods
lsh	Tests LSH instruction
lshc	Tests LSHC instruction
mul	Tests MUL instruction
mula	Tests MULA instruction
mulb	Tests MULB instruction
prime	Writes increasing prime numbers to memory
sum100	Sums integers from 1 to 100

Figure 13: Table of test programs

Here is the code for the *sum100* test:

```
LDC16  R0 0
LDC16  R1 0

Jump:
ADDC R0 R0 1
ADD  R1 R0 R1
SUBC R2 R0 100
NOP
BNE  Jump

LDC16  R3 0xABCD
LDC16  R4 0xABCD
LDC16  R5 0xABCD
LDC16  R6 0xABCD
LDC16  R7 0xABCD
```

This program just demonstrates the functionality of some simple ALU instructions and a BNE instruction. Each time through the loop, R0 is incremented by one, and added to the running sum in R1. The SUBC instruction helps to check for when R0 reaches 100.

The *prime* program, which is somewhat more interesting, is shown here:

```
##R3 is the value to be tested
LDC16 R3 1
##A0 is the address register
LDC16 A0 0

##Manually store the number 2
LDC16 R6 2
STII A0 R6

start:
ADDC R3 R3 2
##R4 cycles through all odd numbers to find a factor
LDC16 R4 1

inside:
ADDC R4 R4 2
SUB R5 R3 R4
NOP
BEQ write

MOV R0 R3
MOV R1 R4
CALL mod
SUBC R2 R2 0
```

```
NOP
BEQ start
NOP
JMP inside

write:
STII A0 R3
JMP start

mod:
##Returns R0 (mod R1) in R2
SUB R0 R0 R1
NOP
BGE mod
ADD R2 R0 R1
RTF
```

This program writes increasing prime numbers to memory. It demonstrates the use of function calls, jumps, branches, and memory accesses. R3, which holds the value being prime-tested in each iteration, is incremented by 2 every time a prime is found or disproved. The loop beginning at the label `inside` checks whether each odd number from 3 up to the number tested to see whether it is a factor. The function `mod` is called to return the value R0 (mod R1) in the register R2. If the result is 0, then R1 is a factor of R0. This program does not have a stopping point, so it will continue to find prime numbers until the user chooses to stop the simulation. Below is a memory dump after 10,000 instruction cycles have been executed:

```
MEM(0x0000):      2      3      5      7     11     13     17     19
MEM(0x0008):     23     29     31     37     41     43     47     53
MEM(0x0010):     59     61     67   ????   ????   ????   ????   ????
MEM(0x0018):   ????   ???? ?????   ????   ????   ????   ????   ????
```

A program listing and display of registers is shown below, also after 10,000 cycles:

```
File listing:
            16: NOP
            17: JMP   inside
        write:
            18: STII  A0 R3
            19: JMP   start
        mod:
```

```
E===>        20: SUB   R0 R0 R1
 D-->        21: NOP
  F->        22: BGE   mod
             23: ADD   R2 R0 R1
             24: RTF
             25: BAD
             26: BAD

WSIM Core Registers:
R0     + 0x0010 +
R1     + 0x000b +
R2     + 0x0008 +
R3     + 0x0047 +
R4     + 0x000b +
R5     + 0x003c
R6     + 0x0002 +
R7       0x????
L0       0x????????
L1       0x????????
L2       0x????????
L3       0x????????
A0     + 0x0013 +
A1       0x????
A2       0x????
A3       0x????
AM0      0x????
AM1      0x????
AM2      0x????
AM3      0x????
CYCLE  * 0x2710 *
DMA0     0x????
DMA1     0x????
DMA2     0x????
DMA3     0x????
Z           + 0 *
N           + 0 *
```

The "*" and "+" characters to the right of the register values indicate that the register was read from in the last cycle or in the current period, respectively. The "*" and "+" characters to the left of the values indicate that the register was written to in the last cycle or in the current period, respectively.

Here is a print out of the instruction execution profiling at the same point in the program:

```
Address   Inst Count   Instruction
-------   ----------   -----------
 00000             3   LDC16    R3     1
 00001             1   LDC16    A0     0
```

```
00002          1  LDC16   R6     2
00003          1  STII    A0     R6
00004         35  ADDC    R3     R3     2
00005         35  LDC16   R4     1
00006        301  ADDC    R4     R4     2
00007        301  SUB     R5     R3     R4
00008        301  NOP
00009        301  BEQ     write
0000a        301  MOV     R0     R3
0000b        283  MOV     R1     R4
0000c        283  CALL    mod
0000d        565  SUBC    R2     R2     0
0000e        282  NOP
0000f        282  BEQ     start
00010        282  NOP
00011        266  JMP     inside
00012        284  STII    A0     R3
00013         18  JMP     start
00014       1341  SUB     R0     R0     R1
00015       1323  NOP
00016       1323  BGE     mod
00017       1323  ADD     R2     R0     R1
00018        282  RTF
00019        282  BAD
```

Here is a printout of the resource profiling statistics:

Resource	Assigned	Used	Total Assigned	Total Used
R0	1606	1605	1606	1605
R1	283	1605	283	1605
R2	564	282	564	282
R3	36	637	36	637
R4	336	885	336	885
R5	301	0	301	0
R6	1	1	1	1
Z	3693	10000	3693	10000
N	3693	10000	3693	10000
A0	20	19	20	19
FPC	10000	10000	10000	10000
DPC	10000	10000	10000	10000
EPC	10000	10000	10000	10000
CYCLE	10000	10000	10000	10000
PC0	283	564	283	564
PC_STACK_PTR	565	847	565	847
LCPTR	0	10000	0	10000
LAPTR	0	10000	0	10000
MEM(0)	1	0	1	0
MEM(1)	1	0	1	0
MEM(2)	1	0	1	0
MEM(3)	1	0	1	0

MEM(4)	1	0	1	0
MEM(5)	1	0	1	0
MEM(6)	1	0	1	0
MEM(7)	1	0	1	0
MEM(8)	1	0	1	0
MEM(9)	1	0	1	0
MEM(10)	1	0	1	0
MEM(11)	1	0	1	0
MEM(12)	1	0	1	0
MEM(13)	1	0	1	0
MEM(14)	1	0	1	0
MEM(15)	1	0	1	0
MEM(16)	1	0	1	0
MEM(17)	1	0	1	0
MEM(18)	1	0	1	0

Chapter 9

Related Work

MIT's course called Computation Structures provides two simulators as teaching tools, JSim and BSim. JSim is a digital circuit construction and analysis tool. It allows the user to build from transistors and gates up to a full RISC processor. It has a simple editor but does not have the functionality to parse an assembly program and execute it. It also does not provide the advanced functionality needed to simulate a DSP processor.

BSim is more similar to this project. It is a simulator for the Beta processor, which is the RISC processor studied in the course. It shows the execution state of the processor when running supplied assembly code. Both of these applications are written in Java, whereas this project is to be implemented in C/C++. The other major difference is that BSim is geared towards a RISC processor, not a DSP processor. Therefore it does not support some advanced capabilities like MACs, loops, and complex addressing techniques. However, both of these tools provide good models of simulation tools.

Chapter 10

Future Work

Although WSIM is capable of simulating a simple DSP architecture, many improvements can be made. First, instead of configuring the processor or hardware description in the source code and having to recompile, a configuration file model can be created. This work involves designing a unique format and language for the file itself, writing a parser for the configuration file, and interfacing the results of the parser to the rest of the simulator.

Instead of only accepting mnemonic assembly source code, a more complicated parser can be written to accept more user-friendly code. The code could look more like a higher level-language like C or Java. For example, instead of the instruction ADDC R0 R1 R2, we could just write R0 = R1 + R2. Many DSP processors already accept this type of source code, so this idea would expand the scope to more users.

Many processors today allow for parallel instructions. Supporting this feature would also allow WSIM to model a wider scope of existing DSP architectures. Although the implementation of parallel instructions should not be too difficult, there are many sticky points because certain instructions have restrictions as to which instructions they may be in parallel with. This problem is caused by a bottleneck of limited hardware resources.

As mentioned earlier, several more ideas for future work include synchronizing the DSP instruction cycle clock and the internal SystemC clock, automating the signals generated by instructions in the decode module, and adding some more advanced processor features like external device ports, a set of kernel registers, direct memory exchange (DME), and more memory modules.

References

[1] Comp.dsp Usenet news group. http://www.bdti.com/faq/3.htm.

[2] QUALCOMM Incorporated Press Release for MSM6250. 11/12/02. http://www.qualcomm.com/press/pr/releases2002/press1115.html.

[3] J. Bhasker, '*A SystemC Primer*,' Star Galaxy Publishing, 2002.

[4] John K. Ousterhout, '*Tcl and the Tk Toolkit*,' Addison-Wesley, 1994.

[5] Using the GNU Compiler Collection. http://www.delorie.com/gnu/docs/gcc/gcc_toc.html.

[5] *esim*: A Structural Design Language for Computer Architecture Education. http://www.cse.ucsc.edu/~elm/Software/Esim/index.html.

[6] M. Mernik, M. Lenic., E. Avdicausevic, V. Zumer, "Compiler/Interpreter Generator System LISA," *Proceedings of the 33rd Hawaii International Conference on System Sciences*, Sept. 2000.

[7] MIT Course Website: Computation Structures, http://6004.lcs.mit.edu/, 2003.

Appendix

Sample Source Code

```
//alu.h
#ifndef ALU_H
#define ALU_H

struct alu : sc_module {
    sc_in<SC_LongLong> in1;     //input 1 - reg1
    sc_in<SC_LongLong> in2a;    //input 2a - reg2
    sc_in<SC_LongLong> in2b;    //input 2b - imm
    sc_in<SC_LongLong> in2c;    //input 2c - mem
    sc_in<long> in2_sel;        //0 for 2a,1 for 2b,2 for 2c
    sc_in<long> in1width;       //input 1 width
    sc_in<long> in2awidth;      //input 2a width
    sc_in<long> in2bwidth;      //input 2b width
    sc_in<long> in2cwidth;      //input 2c width
    sc_in<AluOp> fn;            //alu function
    sc_out<SC_LongLong> out;    //output
    sc_out<long> reg_outwidth;  //reg output width
    sc_out<long> mem_outwidth;  //mem output width
    sc_in<RegMemWE> regs_mem_we;//write to reg or mem
    sc_out<bool> z_flag;        //zero
    sc_out<bool> n_flag;        //negative
    sc_in<bool>    clk;         //clock

    void exec();                //method implementing
                                //functionality

    //Constructor
    SC_CTOR( alu ) {
        SC_THREAD( exec );   //Declare exec as SC_THREAD and
                             //dont_initialize();
        sensitive_pos << clk;  //make it sensitive to
                               //positive clock edge
    }

};

#endif
```

```
//alu.cpp
#include "math.h"
#include "systemc.h"
#include "types.h"
#include "alu.h"

//Definition of exec method
void alu::exec()
{
  signed long long a, b;           // Inputs
  long a_width, b_width, r_width;  // Input widths
  static signed long long result;  // ALU result output
  unsigned long long atmp;         // Unsigned version of a
  long sel;                        // Input 2 select
                                   // 0(reg), 1(imm), 2(mem)
  AluOp op;                        // Operation to execute
  SC_LongLong temp;     // Signal to send long long variable

  while(1)
    {
      // Manually synchronize with rest of system
      wait();
      wait();

      sel = in2_sel.read(); // Input2 select from decode
      op = fn.read();       // Operation (from decode)
      a = in1.read().Num;   // Input1 (from reg)
      a_width = in1width.read(); // Input1 width

      // Signal from decode that tells to write back to regs
      // or mem
      // Get the width. If not regs or mem, assume 64 bits
      if(regs_mem_we.read() == REGALUWE)
        {
          r_width = reg_outwidth.read();
        }
      else if(regs_mem_we.read() == MEMALUWE)
        {
          r_width = mem_outwidth.read();
        }
      else
        {
          r_width = 64;
        }

      // Read input2 (2=mem, 1=imm, 0=reg) and get width
      if(sel == 2)
        {
          b = in2c.read().Num;
          b_width = in2cwidth.read();
        }
      else if(sel == 1)
        {
          b = in2b.read().Num;
```

```
      b_width = in2bwidth.read();
    }
  else
    {
      b = in2a.read().Num;
      b_width = in2awidth.read();
    }

  // Sign extend a and b if negative
  if(a_width != 64 && a >> (a_width - 1))
    {
      a = a | ((long long)-1 << a_width);
    }

  if(b_width != 64 && b >> (b_width - 1))
    {
      b = b | ((long long)-1 << b_width);
    }

  // Perform operation
  switch(op)
    {
    case ADD:        //add
      result = a+b;
      break;
    case SUB:        //sub
      result = a-b;
      break;
    case A:          //a
      result = a;
      break;
    case B:          //b
      result = b;
      break;
    case ASH:        //arithmetic shift
      if(b >= 0)
        {
          result = a<<b;
        }
      else
        {
          result = a>>-b;
        }
      break;
    case LSH:        //logical shift
      // For logical shift, make A unsigned, mask the
      // digits past a_width, and do shift
      atmp = (unsigned) a;
      atmp=atmp&((unsigned long long)pow(2,a_width)-1);
      if(b >= 0)
        result = atmp<<b;
      else
        result = atmp>>-b;
      break;
```

```
        default:
          break;
        }

      // Mask result to correct width
      result=result&((unsigned long long)pow(2, r_width)-1);

      // Send result to regs and mem
      temp.Num = result;
      out.write(temp);

      // Write flags
      if(result == 0)
        z_flag.write(1);
      else
        z_flag.write(0);

      // Negative flag
      if(result & (0x1 << (r_width-1)))
        n_flag.write(1);
      else
        n_flag.write(0);

      // Sync
      wait();
      wait();
      wait();
    }
} // end of exec method
```