

***PSIM: A PROCESSOR SIMULATOR FOR BASIC COMPUTER
ARCHITECTURE AND OPERATION EDUCATION***

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisor. This thesis does not include proprietary or classified information.

Michael Alexander Lusco

Certificate of Approval:

Dr. Charles E. Stroud
Professor, IEEE Fellow
Electrical and Computer Engineering

James R. Hansen
Director
University Honors College

***PSIM: A PROCESSOR SIMULATOR FOR BASIC COMPUTER ARCHITECTURE
AND OPERATION EDUCATION***

Michael Alexander Lusco

A Thesis

Submitted to the

Auburn University Honors College

In Partial Fulfillment of the

Requirements for

University Honors Scholar

Auburn, Alabama

April 24, 2010

*PSIM: A PROCESSOR SIMULATOR FOR BASIC COMPUTER ARCHITECTURE
AND OPERATION EDUCATION*

Michael Alexander Lusco

Permission is granted to Auburn University to make copies of this thesis at its discretion, upon the request of individuals or institutions and at their expense. The author reserves all publication rights.

Signature of Author

Date of Graduation

VITA

Michael Alexander Lusco, son of Michael Charles Lusco and Rhonda Joyce Lusco, was born June 4, 1988, in Birmingham, Alabama. He graduated high school from Oak Mountain High School in Birmingham, Alabama in 2006. He enrolled in Auburn University in the Fall of 2006 as a member of the University Honors Program. He graduated with a Bachelor of Engineering in Electrical and Computer Hardware degree and a minor in Business on May 14, 2010 and has been accepted into Graduate School at Auburn University starting the summer of 2010.

THESIS ABSTRACT

PSIM: A PROCESSOR SIMULATOR FOR BASIC COMPUTER ARCHITECTURE AND OPERATION EDUCATION

Michael A Lusco

Bachelor of Engineering, April 24, 2010

70 Typed Pages

Directed by Charles E. Stroud

As computer architectures have grown more complex, it has become increasingly more challenging for new students to understand the inner-workings necessary for a solid foundation in design and use of embedded processors. Without a basic understanding of the low-level interactions between components of a processor, it is often difficult for a student to fully utilize and understand a processor's architecture. This thesis discusses a method to aid students in learning the basics of computer architecture via a simulator program called the Processor SIMulator (PSIM).

The PSIM is a basic stored program computer architecture which graphically displays the underlying computer architecture while showing the detailed operation on a per clock cycle basis. Its instruction set consists of twenty-five instructions that can be combined to execute many of the more complex capabilities of more advanced architectures including conditional branches. In addition the PSIM includes an assembler for compiling assembly

language programs to PSIM machine code, the ability to display values in various formats including decimal and hexadecimal, and the ability to display and write to a file the contents of the program memory at any point in the simulation

ACKNOWLEDGMENTS

I would like to especially thank Dr. Stroud for his previous work and encouragement in helping me to complete my honors thesis. Without his constant encouragement and expertise, I would never have started the PSIM project which got me interested and ultimately laid the ground work for my entrance into Auburn's Graduate School.

TABLE OF CONTENTS

List of Tables	Error! Bookmark not defined.
List of Illustrations	x
Introduction.....	1
Background	3
Simulator Architecture.....	9
Graphical Interface.....	18
Simulator Code Structure.....	25
Summary and Conclusions	28
Appendix A: The Instruction Set	31
Appendix B: Example Programs	53
Appendix C: Controller Signals.....	56
Works Cited	59

LIST OF TABLES

Table 1 - DOS PSIM Instruction Set	4
Table 2 - Instruction Set of RISC-16 (5)	8
Table 3 - PSIM Instruction Set and Operand Count	14
Table 4 - PSIM Components.....	26

LIST OF ILLUSTRATIONS

Figure 1 - Original PSIM GUI.....	5
Figure 2 - The Simulator Architecture.....	10
Figure 3 - The Instruction Format.....	12
Figure 4 - Assembler Error List.....	15
Figure 5 - PSIM Home Screen.....	18
Figure 6 - From Left to Right: File Menu, Display Menu, Assembler Menu.....	19
Figure 7 - PSIM Instruction Information Window	21
Figure 8 - Bus Viewer.....	22
Figure 9 - Different Time Views	23
Figure 10 - Typical VHDL Process for a Counter	26
Figure 11- PSIM Process for a Counter	26
Figure 13 - PSIM Controller Block	56

This thesis follows the styling standards put forth in the *Guide to Preparation and Submission of Theses and Dissertations*, (2005) put forth by Auburn University.

This thesis was created using Microsoft Word 2007.

CHAPTER 1

Introduction

A thorough understanding of computer architecture is essential to the greater understanding of software and computer science. Often it is difficult to understand the inner workings of a computer when studying large-scale or complex architecture systems. The motivation behind the Processor SIMulator (PSIM) is to help bridge the learning gap by providing a small-scale, limited instruction set architecture whose internal operations and components can easily be understood by individuals with little or no previous exposure to the internal operation of a processor or computer architecture.

The PSIM display is a graphical environment where the internal components and their interconnections are displayed so that the user can easily understand the signal paths that data and control take during operation. The simulator consists of several basic components, including a simple arithmetic-logic unit (ALU), a number of user accessible and system registers, three multiplexors, an instruction read-only memory (ROM), and a data random access memory (RAM). In addition, the graphical user interface (GUI) provides additional options to the user; giving control over execution via control of the system clock as well as access to the processor's input register and instruction assembler. The initial idea of the simulator is based on a DOS-based simulator created by Dr. Stroud (1) which is described in more detail in Chapter 2. The primary goal of the current effort was to improve upon the original idea by providing a Windows based GUI with greater user interactivity. As the

project progressed additional instructions and architecture modifications were made to enhance the features and capabilities of the simulator in computer architecture and operation education.

This thesis provides a detailed background on processor simulators including the basis for this project developed by Dr. Stroud in Chapter 2 followed by a look at the architecture of the system and its instructions in Chapter 3. The user interface is detailed in Chapter 4 and the source code is discussed and analyzed in Chapter 5 before the paper is summarized and concluded in Chapter 6.

The PSIM project is currently maintained as an open source project located at <http://www.sf.net/projects/psimedu> and it can be downloaded and run on any Microsoft Windows computer with the Microsoft .NET Framework 3.5 install.

CHAPTER 2

Background

The main goal of PSIM is to be used as an education tool for individuals interested in embedded systems. When viewed macroscopically, embedded processor systems are often viewed as a black box; however, to really apply these systems it is important to understand the underlying architecture and to have a deeper understanding of how the core components interact. PSIM attempts to give the user a visual representation of a multi-cycle architecture so that a person interested in learning more about embedded processors can start with an easy to understand model and build up to more advanced architectures.

PSIM is not the first processor simulator which is focused education. In fact this PSIM is loosely based on the original PSIM by Charles Stroud (1). The original PSIM is a DOS simulator which simulates a simple 8-bit multi-cycle processor architecture which was originally based on the Simple Computer Architecture by Mano (2). This simple computer architecture consists of a total of four instructions including ADD, AND, JMP, INC.

The original PSIM expanded upon the basic architecture of the Simple Computer Architecture and added additional features for branching and an expanded instruction set (1). It has an 8-bit address bus allowing for up to 256 memory locations. It includes four 8-bit registers including an accumulator, data, input, output, and address register as well as a

1-bit carry register and 4-bit instruction register. The system has an ALU which can perform add, increment, complement, AND, NOR, and XOR functions. In addition the memory in the system is a shared program/data memory. The instruction set is summarized in Table 1 - DOS PSIM Instruction Set.

Instruction	Description	Instruction	Description
HLT	Halt's Processor	STA ADRS	Store AC in Mem
NOP	No Operation	STI ADRS	Store IN in Mem
INA	Increment AC	LDA ADRS	Load Mem into AC
CMA	Complement AC	LDO ADRS	Load Mem into OR
ISZ	Increment/Skip if zero	ADA	Add Mem to AC
LDI OPRD	Load OPRD in AC	AND ADRS	And Mem and AC
ADI	ADD OPRD in AC	NOR ADRS	NOR Mem and AC
BUN ADRS	Branch Unconditional	XOR ADRS	XOR Mem and AC

Table 1 - DOS PSIM Instruction Set

The key to the original PSIM was its ability to allow the user to single clock through the simulators execution allowing interested parties the ability to view the contents of the processors registers during execution of instructions. As seen in its GUI in Figure 1 - Original PSIM GUI, this allowed the program to be used to gain a deeper understand of how the processor fetched and executed different types of instructions. Unfortunately this simulator has become dated over time. Since the simulator program executable is 16-bit it is unable to be run in newer 64-bit versions of Microsoft Windows which only support 32-bit and 64-bit applications. In addition there are a lot of opportunities to expand the simulator with a full Windows GUI and a higher level of interactivity with the user.

In addition to the original PSIM there is an educational simulator called the “Relatively Simple CPU” that is based on the “Very Simple CPU” architecture which are both by John D. C Carpinelli (3). The “Relatively Simple CPU” is a more advanced architecture than the “Very Simple CPU” as it supports up to 64K of memory with 16

instructions and has a java web interface which provides a look at the underlying architecture.

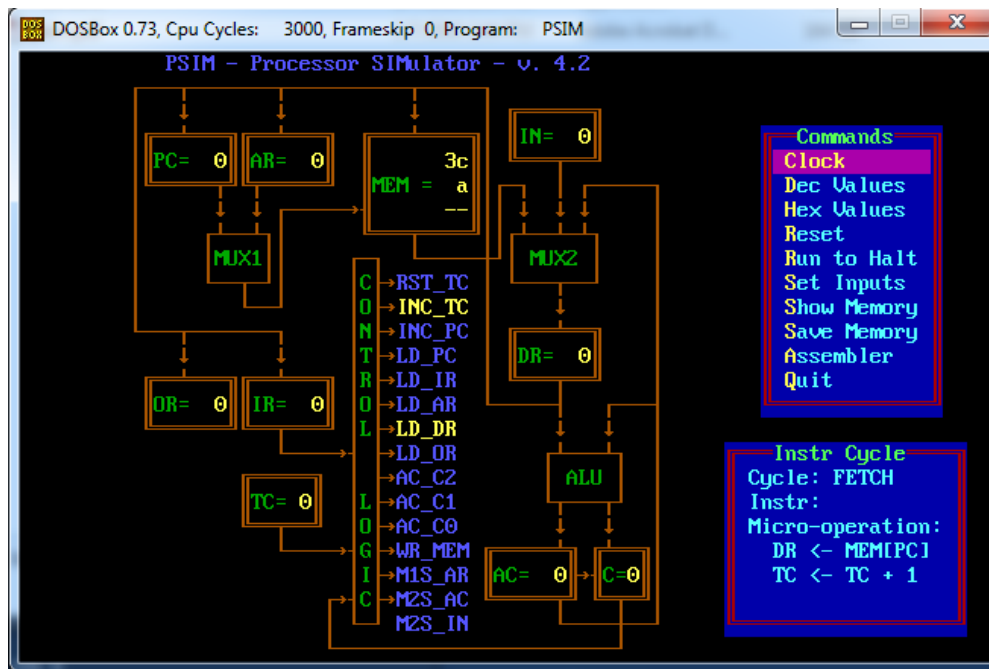


Figure 1 - Original PSIM GUI

The simulator is meant to be used as a companion for Mano’s book *Computer Systems Organization and Architecture*. As such it follows a similar paradigm to the PSIM by Stroud: it has an interface which can compile an assembly program where it can then be clocked and inspected as it runs. It also supports two different types of controllers including micro-coded and a traditional hard-wired control unit. The “Relatively Simple CPU” is accessible via the web via a java applet; while this does give it the advantage of system independence, its GUI is also more confusing than Stroud’s implementation as it requires multiple windows to view and run assembled programs. This can often be confusing as the system does not always function in the way one would expect. Once the simulator is running, it does provide a fair amount of feedback through visual animation and inspection of register values. As it stands the “Relatively Simple CPU” works well as a student companion

for the book; however, the tool does have a learning curve which impedes its ability to really give students a deeper understanding of the architecture. With improvements to the design and flow of the GUI, the “Relatively Simple CPU” would work better at improving a student’s understanding of a program’s execution at the architecture level.

Currently Stroud’s PSIM is in use as an educational aid for his VHDL class at Auburn University. It is used to introduce students to more advanced digital systems and introduce the concept of soft-core processor development for Field Programmable Gate Arrays (FPGA). According to Stroud, students seem to prefer the use of PSIM as it helps them to familiarize themselves with a simple architecture as well as gives them a platform to test simple programs (4). Compared to the “Relatively Simple Simulator” it has a simpler interface that presents the architecture and a few menu options. This helps prevent any confusion as well as help to lessen any learning curve associated with the software. It is this simple interface that the new PSIM attempts to emulate, while at the same time adding new instructions and features to the simulator.

The most similar education oriented simulator to the PSIM was the “RiSC-16” simulator (5). Like the PSIM, Jaumain sought to visually simulate an easy to understand architecture for students unfamiliar with the underlying details of computer architecture. The simulator that was developed simulates the “Ridiculously Simple Computer” or RiSC-16 developed by Professor Bruce Jacob at the University of Maryland (6). The simulator has a full GUI similar to the PSIM that allows different methods of clocking the processor (half-speed, single step, and full-speed) and visually shows the contents of the memory and registers as well as a block level diagram of the data path of the architecture. In addition it seeks to differentiate the different activities of the architecture components via colors and

Instruction	Assembly Format	Action
Add	add regA, regB, regC	Add contents of regB with regC, store result in regA.
Addi	addi regA, regB, Imm	Add contents of regB with Imm, store result in regA.
Nand	nand regA, regB, regC	Nand contents of regB with regC, store results in regA.
Lui	lui regA, Imm	Place the 10 ten bits of the 16-bit Imm into the 10 ten bits of regA, setting the bottom 6 bits of regA to zero.
Sw	sw regA, regB, Imm	Store value from regA into memory. Memory address is formed by adding Imm with contents of regB.
Lw	lw regA	Load value from memory into regA. Memory address is formed by adding Imm with contents of regB.
Beq	beq regA	If the contents of regA and regB are the same, branch to the address PC+1+Imm, where PC is the address of the beq instruction
Jalr	jalr regA	Branch to the address in regB. Store PC+1 into regA, where PC is the address of the jalr instruction.

easy to understand visual cues. The architecture itself is simple consisting of only eight instructions as shown in

Table 2 - Instruction Set of RISC-16. In addition its major logic components consist of eight registers, separated program and data memories (each 64K words), and an ALU with add, nand, and equality testing. Compared to the PSIM this architecture is slightly more limited (missing some of the branching and ALU capabilities discussed in Chapter 3); however, by using more colors and providing more visual feedback, the GUI does present some advantages over the PSIM. This simulator is in use by Jaumain as a two four hour lab courses for students interested in computer architectures. According to Jaumain students seem to prefer the simulators use in the classroom: “Compared to an ex-cathedra course and textbooks, they appreciate the ability to see events at their own rate and make their own experiments. (5)”

Clearly an educational processor simulator is a good addition for the classroom. It helps students by introducing them to the underlying components that make up a processor's architecture. In addition by giving students a solid foundation to build on and test with, the simulator aids them in understanding more complicated architectures such as multi-stage

Instruction	Assembly Format	Action
Add	add regA, regB, regC	Add contents of regB with regC, store result in regA.
Addi	addi regA, regB, Imm	Add contents of regB with Imm, store result in regA.
Nand	nand regA, regB, regC	Nand contents of regB with regC, store results in regA.
Lui	lui regA, Imm	Place the 10 ten bits of the 16-bit Imm into the 10 ten bits of regA, setting the bottom 6 bits of regA to zero.
Sw	sw regA, regB, Imm	Store value from regA into memory. Memory address is formed by adding Imm with contents of regB.
Lw	lw regA	Load value from memory into regA. Memory address is formed by adding Imm with contents of regB.
Beq	beq regA	If the contents of regA and regB are the same, branch to the address PC+1+Imm, where PC is the address of the beq instruction
Jalr	jalr regA	Branch to the address in regB. Store PC+1 into regA, where PC is the address of the jalr instruction.

pipelines. The new PSIM's goal is to fulfill that niche and to provide an easy to use platform for future Computer Engineers.

Table 2 - Instruction Set of RISC-16 (5)

CHAPTER 3

Simulator Architecture

PSIM Architecture

The PSIM simulates an 8-bit multi-cycle embedded processor meaning that every instruction takes multiple clock cycles to execute. It has several main components that make it a robust processor for simulation with many options. It includes the following major components:

- 256 x 16-bit Word Instruction ROM
- 256 x 8-bit Data RAM
- ALU with Zero and Carry flags provides the ability to perform basic arithmetic and logic operations
- Three user accessible 8-bit registers
- One 8-bit input register
- One 8-bit output register
- Support for JUMP and BRANCH instructions

A basic block diagram of the architecture and data path can be seen in Figure 2 - The Simulator Architecture. Each component is labeled (numbered) and a brief description is given related to each component's function in the architecture as follows:

1. PSIM Controller – Main PSIM state machine where each output controls a specific component, allowing it to perform operations.

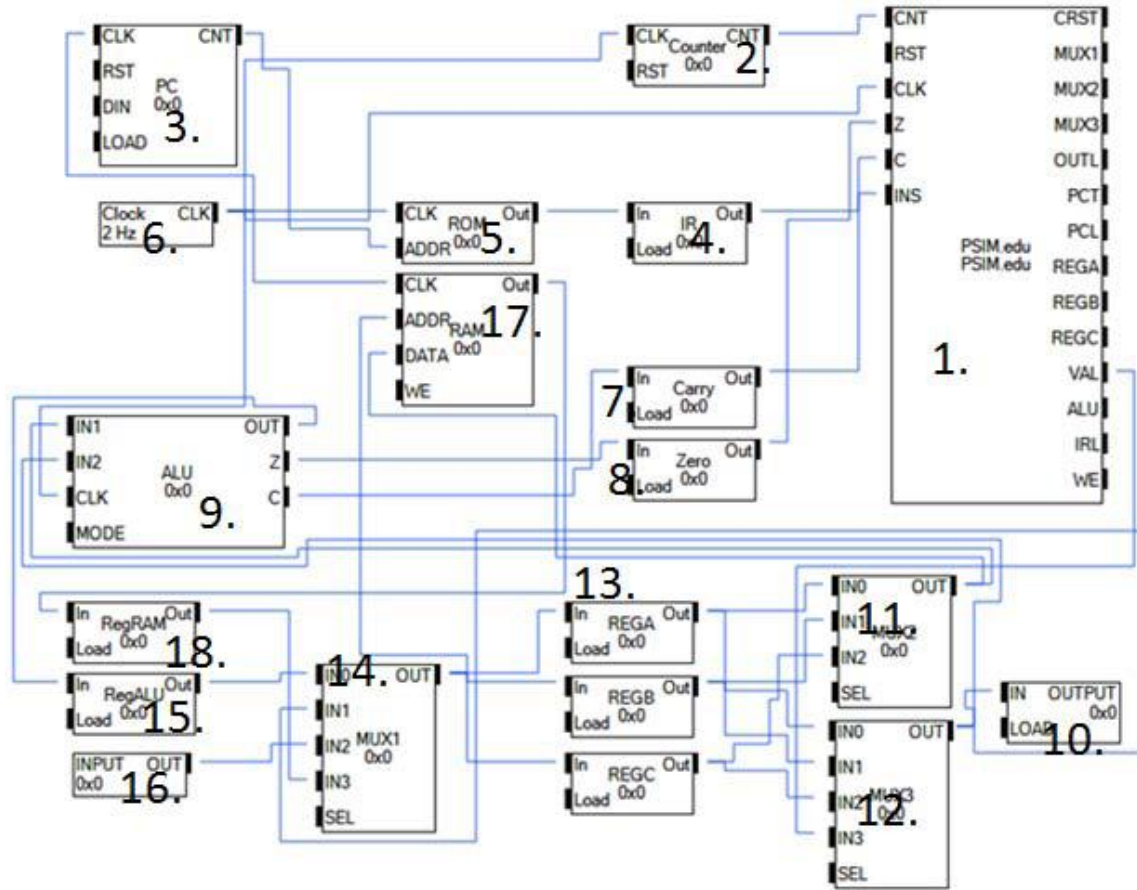


Figure 2 - The Simulator Architecture

2. Instruction Counter – This counter controls the instruction state and provides a way to track each setup of an instruction being processed.
3. Program Counter (PC) – The PC provides the current instruction ROM address to be retrieved. The address holds the instruction to be processed by the controller. The PC can be incremented or updated by the PSIM counter.
4. Instruction Register (IR) – The IR register holds the last instruction retrieved from the ROM
5. Instruction ROM – The Instruction ROM holds the compiled instructions to be executed.

6. Clock – A Simple clock that outputs a 50% duty cycle wave form at a user specified clock frequency.
7. Carry Register – The Carry Register holds the carry flag value from the last ALU operation.
8. Zero Register – The Zero Register holds the zero flag value from the last ALU operation.
9. ALU – The ALU handles all mathematic and logic operations.
10. Output Register – The Output Register holds the value on the output of the processor.
11. MUX 2 – Multiplexor (MUX) 2 selects a register output onto ALU input 1 or the data port of the Data RAM depending on the instruction.
12. MUX 3- MUX 3 selects the a register output or a user defined operand onto the ALU input 2, the output register, or back into MUX 1 to be reused for the address of the Data RAM or to be stored in another register.
13. Registers A, B, and C – These are user accessible registers that hold user defined values. They can be reset via the global reset of the system.
14. MUX1 – MUX1 selects an input value for Registers A, B and C and for the address port of the Data RAM. It selects between the output of the ALU, the INPUT register, the Data RAM, or from MUX 3.
15. ALU Register – The ALU Register holds the value from the last ALU operation.
16. Input Register – The Input Register holds any user specified value to be passed into the processor. It is set through the GUI and can be modified at any time.
17. Data RAM – This memory is a read and write memory that can be saved to and loaded from.

18. RAM Register – This register stores the Data RAM output value.

Instruction Architecture

All PSIM instructions consist of a single 16-bit word that holds all the contents of the instruction and its operands as illustrated in Figure 3 (taken from the Instruction Information Window of the GUI which is explained in more detail in Chapter 3).

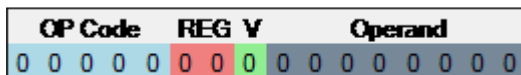


Figure 3 - The Instruction Format

The instruction is broken up into two main bytes. The first byte holds the operation code (OP Code) or the instruction code, the register to be operated on (REG) and a bit to denote if the value in the operand byte is a value (V) or an additional register address. All instructions fit this format (except for branch instructions which are a special case). This format does sacrifice some size since it requires two bytes per instruction and consequently a 16-bit instruction ROM; however, it also makes the instruction much easier to understand from a complexity stand-point as it is clearly separated into two distinct halves and is standardized among all instructions. This means it requires no special flags for instruction types making it easier for students to comprehend. The branch equal and branch not equal instructions are a unique case as they sacrifice a single bit of the operand section to allow a second register and address to be stored in the instruction. For this case the V bit and the most significant bit of the Operand section is used to store a second register for comparison with the register in REG and consequently uses the remaining 7-bits of the Operand section to store the absolute address to jump to if the branch is carried out. In practice there are four types of instruction formats:

- No operand instructions require only an op code to be a valid instruction code. The rest of the instruction should be zero. An example of this type of instruction is HALT.
- Single operand instructions require a valid 5-bit op code as well as either a register number (1-3 corresponding to register A-C) placed in the REG section or if only an immediate operand is required the V flag should be set to one and the operand value should be filled in with a valid 8-bit number. An example is JUMP.
- Two operands instructions require a valid 5-bit op code as well as a valid register (1-3 corresponding to register A-C) in the REG section. If the instruction requires two registers then the V bit should be zero and another register should be placed in the operand section of the instruction. If an immediate operand is required then the V bit should be a one and the 8-bit operand should be placed into the operand section of the instruction. An example of this type of instruction is ADD.
- Three operand instructions are a special case and require a slightly different instruction format since they require two registers and an operand value. The first register should be stored in the REG section of the instruction. The second register shares the V bit and the most significant bit of the operand section. Finally the remaining 7-bits of the operand section store up to a 7-bit operand. An example of this type of instruction is BEQ.

The Instruction Set

Though the maximum number of possible instructions is thirty-two (since there is a 5-bit OP Code), the simulator currently consists of twenty-five instructions ranging from basic

operations to arithmetic operations to jump and branch operations. The complete instruction set of the PSIM architecture and simulator is given in detail in Appendix A; however, for clarity the instructions are also summarized in Table 3.

Instruction	#	Instruction	#	Instruction	#	Instruction	#	Instruction	#
HALT	0	STM	2	NOT	1	LSR	2	BNE	3
LOAD	2	ADD	2	XOR	2	ROL	2	BRZ	1
LDCR	1	SUB	2	IN	1	ROR	2	BNZ	1
LDZR	1	AND	2	OUT	1	JUMP	1	BRC	1
LDM	2	OR	2	LSL	2	BEQ	3	BNC	1

Table 3 - PSIM Instruction Set and Operand Count

Since the PSIM is a multi-cycle architecture, each instruction takes a different number of clock cycles to execute. The maximum number of clock cycles for any instruction is five (BEQ and ALU operations) while the minimum is two. To facilitate this process a simple up-counter which increments every clock cycle is connected to the controller. The controller uses this counter value and the OP code to determine what values should be put on the outputs to the system components. Once the instruction is finished executing the counter is reset and a new instruction is fetched from the Instruction ROM.

The Assembler

The assembler is activated via the *Assemble Menu* by selecting *Load Program*. Two different file types can be loaded from the assembly menu: *psim* files which consist of one or more lines of assembly code to be assembled and loaded into the program ROM; *pbin* files are saved memory files and do not require assembly and instead are directly loaded into the Program ROM. When loading a file there is also an option to select from all files. In this case, if a file is selected that does not have either a *psim* or *pbin* file extension then by default it is treated as a *psim* file and an attempt will be made to assemble it.

While not 100% effective, the assembler does do some error checking for operand parameters and syntax errors. If the wrong type or an incorrect number of operands is used with an instruction the assembler will open an error dialog that will display any errors encountered during assembly and the line number that the error occurred. An example can be seen in Figure 4. The error checking for assembled programs is limited and focuses mostly around instances where any of the operands are unable to be determined or the incorrect type of operand is used. In addition there is error checking for labels and invalid instructions. There is no runtime checking or exception condition generated for infinite loops or other program logic errors (though the execution of the processor can be halted and reset by the user at any time if a logic error is noticed).

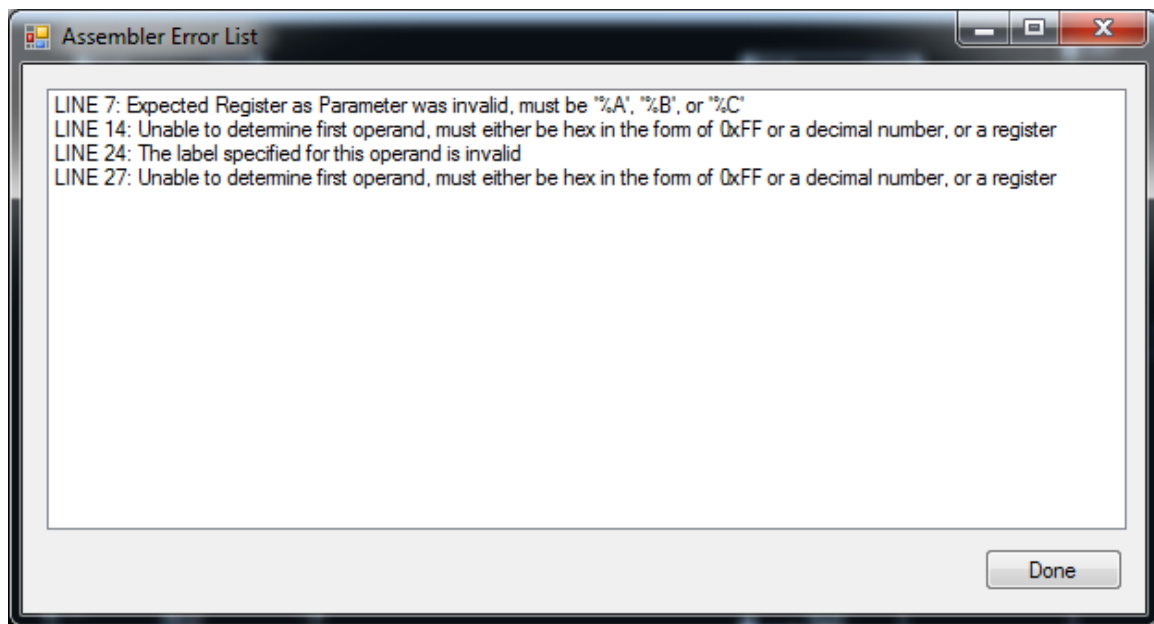


Figure 4 - Assembler Error List

PSIM Assembly Language

The PSIM language constructed to be a very straightforward assembly implementation. At a basic level only one instruction is allowed per a line with trailing whitespace ignored. The following list explains all of the special PSIM language constructs:

- Using a register:
 - To access the user registers you must use the following values: %a, %b, or %c.
 - An example would be the following: LOAD %a 5, which loads Register A with the decimal value 5.
- Using Hex Values:
 - The assembler will recognize both decimal and hexadecimal values. The default interpretation of a number is decimal; however, if the value is preceded by the '\$' symbol an attempt will be made by the assembler to interpret the value as hex.
 - An example would be Load %a, \$2A, which loads Register A with the hex value 0x2A.
- Using Comments:
 - Comments are denoted by a ';' character. If the first character in the line is a ';' then the remainder of the line will be ignored as a comment. There is no support for comments in line with instructions. All comments and all instructions must be a separate, single, line.
 - An example: ";" this is a comment about the program"
 - INVALID example: LOAD %a 5 ; this loads a with 5

- Using Labels

- Labels are used by the assembler to determine where to jump or branch to.

They appear by themselves on a line anywhere before or after the instruction to JUMP or branch to. They are denoted by a ‘:’ character.

- To use a label in an instruction as a destination for a branch or jump it must be prefixed with the ‘@’ symbol. This looks like the following and is only valid for JUMP and BRANCH instructions.

- :Reset
BEQ %a %b @End
JUMP @Reset
:End
HALT

For more information on the instruction set please view Appendix A or see example programs in Appendix B.

CHAPTER 4

Graphical Interface

The graphical interface (GUI) of the PSIM is Windows based and consists of a main home screen (Figure 5) and two support dialogs that provide more information about signals and running instructions.

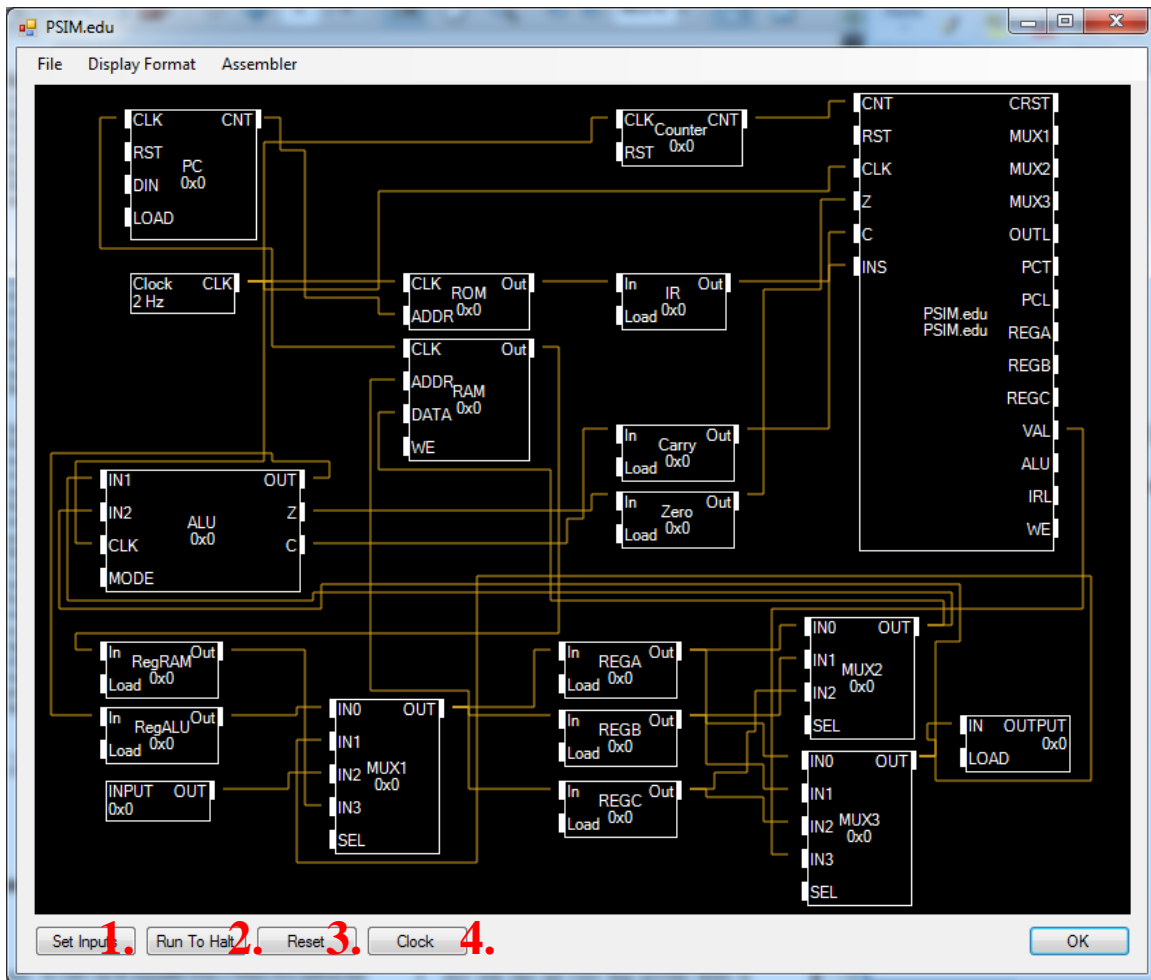


Figure 5 - PSIM Home Screen

The home screen consists of a display of the processor's current state. It includes menu options and buttons to begin working with the simulator. The buttons (in Figure 5, labeled 1-4, from left to right) have the following function:

1. Set Inputs – Sets the value to be placed on the input register.
2. Run To Halt – Starts the clock and allows the processor to run until it reaches a halt instruction¹.
3. Reset – Resets the processor and its registers to its known beginning state, it does not erase the ROM contents, but will reset the RAM.
4. Clock – Clocks the processor a single clock cycle. This allows the user to set through individual instructions as they execute.

At the top of Figure 5, the menu can be seen. There are three menus including File, Display Format, and Assembler. These menus can be seen in Figure 6.

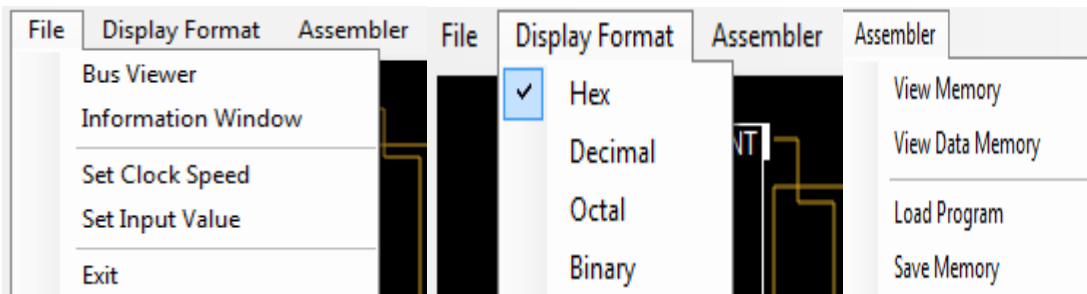


Figure 6 - From Left to Right: File Menu, Display Menu, Assembler Menu

The file menu has the following options:

- Bus Viewer – Launches a bus viewer window which defaults to viewing the clock.
This will be explained in more detail in the support dialog section.

¹ The Clock button will change to a pause button when the system is running, until it halts

- Information Window – Launches an information window which details each step the processor is executing while running. This will be explained in more detail in the support dialogs section.
- Set Clock Speed – Allows the user to set a new clock speed in hertz for the system to run at when it is running until halt.
- Set Input Value – Allows the user to set a new value to be placed into the input register of the system.

The display format menu adjusts the global display settings for numbers for the system. This affects all visual displays of numbers.

- Hex – All numbers displayed in the C hex format of i.e. 0xF.
- Decimal – All numbers are displayed in a normal decimal format i.e. 16.
- Octal – All numbers are displayed in base 8. A % designates octal i.e. %20.
- Binary – All numbers are displayed in binary. A b represents binary i.e. 10000b

The Assembler menu allows access for compiling new programs as well as viewing and saving the memory.

- View Memory – Brings up a box displaying the contents of the ROM in the simulator.
- View Data Memory – Brings up a box displaying the contents of the RAM in the simulator.
- Load Program – Attempts to assemble a new program into the memory. The assembler will attempt to assemble this program automatically. If there are any errors an error dialog will popup noting the suspected line numbers a general description of any errors.

- Save Memory – Saves the ROM contents as a *pbin* file for later use. This file can be re-loaded into the simulator at a later date without needed re-assembly.

Information Viewer

The information window in Figure 7 displays the contents of the instruction register as well as some information about the processors current instruction state. The instruction view at the top of the window displays the instruction register in its typical format so that flags, operand value, registers, and op codes can easily be understood. Below that the instruction information section displays some basic information about the currently executing instruction including what stage the processor is currently in. In addition it will display the current line number and instruction text that was assembled from the code file. In this above example the instruction was LDC 4, which is short hand for “LOAD %c 4”. It is easy to see that the processor decoded the instruction as “LOAD Register C with Value 4”.

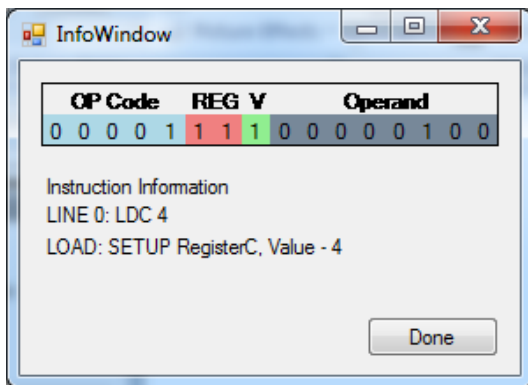


Figure 7 - PSIM Instruction Information Window

The information window is a support dialog; as such it can be closed at any time without affecting the execution of the processor. It can be opened at any time via the *File Menu* by selecting *Information Window* and is opened with the processor by default.

Bus Viewer

The bus viewer in Figure 8 displays a real time view of any pin of any component of the processor simulator. In Figure 8 the top bus (1) is the clock output and the bottom bus (2) is the mode select of the ALU. This dialog is a support dialog it can be closed or opened without affecting the execution of the processor. This dialog can be opened by going to the *File Menu* then selecting *Bus Viewer*. It will then be opened with the clock bus showing by default. Alternatively, any pin of a simulator component can be double clicked to open a bus viewer with that pin viewed. If the dialog is already opened, the new pin will be appended at the bottom of the viewer so that it can be compared to any other pins currently viewed. This dialog is not opened automatically with the processor.

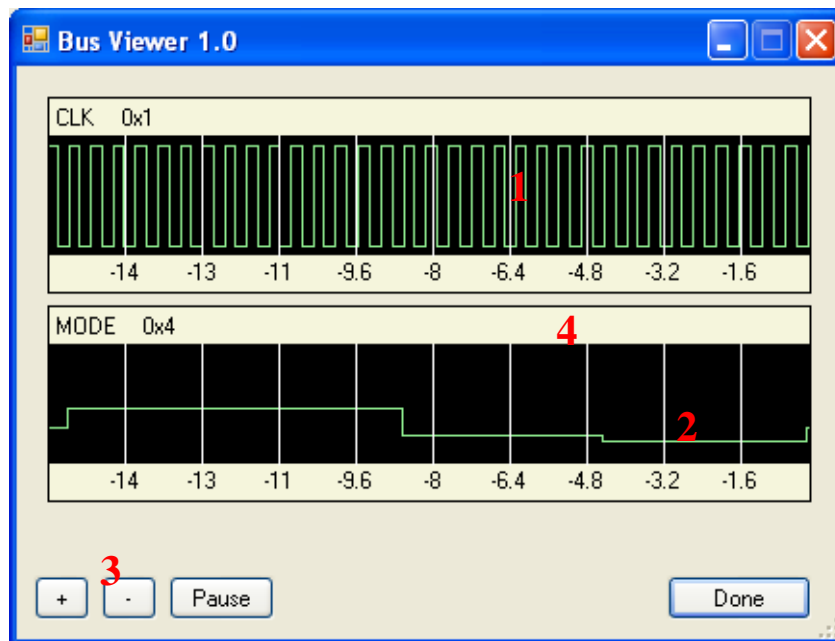


Figure 8 - Bus Viewer that is displaying the Clock output (1) and Register A's Output (2)

The output that is displayed is a type of histogram. Once opened the bus viewer begins to store up to 16 seconds of data about the signals history. The viewer samples data

from the signal every .004 seconds and can store a maximum of 1100 samples; however, this does not limit the viewer to only 4.4 seconds worth the data due to buffering.

The histogram time can be adjusted using the +/- buttons (Figure 8 #3). These buttons will adjust the time for all signals in the bus viewer. If only a single view is needed to be changed the +/- buttons on the individual signals can be used (Figure 8 #4). The maximum history window is 16 seconds, while the minimum is .01 second as shown in Figure 9. Instead of drawing the entire signal every time there is a change, the drawing code calculates the difference in the last time the graph was updated and the current time and draws only the portion of the signal that is new. This means that while only 4.4 seconds worth the actual history of a signal is stored, the graph can be display a history for as long as it has been drawn (it is arbitrary limited to 16 seconds in the viewer) since the drawing code just shifts the existing drawn image to the left when drawing the new part of the signal.

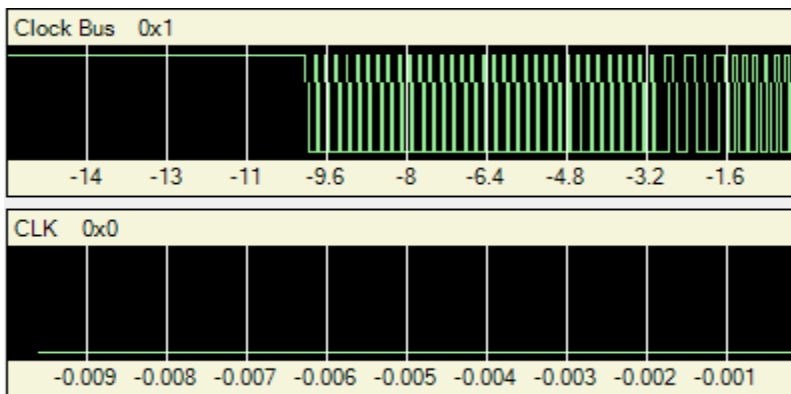


Figure 9 - Different Time Views

The viewer also allows the user to pause or capture a signal via the Pause button (Figure 8 #3) which stops pauses the capturing of all viewed signals or the 'C' button in (Figure 8 #4) which pauses only the effected signal. While a signal is paused no new information will be sampled by the signal viewer. A signal can still be zoomed in and out via the +/- buttons; however, only the existing history will be used. Once resumed, new data is

appended to the existing history, meaning any signal data occurring while the signal is paused will be lost.

All signals can be viewed in any numerical base supported by the simulator. By double-clicking in the blank margin of (Figure 8 #4), the numerical value will cycle through binary, octal, decimal, hex, and default (where default is whatever setting the simulator is set to display numbers in). To remove a signal from the bus viewer press the 'X' in (Figure 8 #4).

The goal of the PSIM GUI is to minimize any learning curve necessary to use the simulator while simultaneously providing the user with the maximum amount of detailed information possible. With the addition of the signal viewer and the information window, the PSIM provides a wealth of visual feedback to the user; however, by compartmentalizing these dialogs, the user can choose how much information they want to see about the architecture. These allow (7) a user to tailor the PSIM to his or her specific needs.

CHAPTER 5

Simulator Code Structure

The PSIM simulator is written in Microsoft .NET C# 3.0. C# is a managed language similar to Sun's Java. It has a syntax similar to that of C and C++ and provides a platform called the Common Language Runtime (CLR). The CLR is a platform of libraries similar to Java provided by Microsoft to perform certain functions such as graphical user interfaces or file input-output operations. Using these libraries simplifies the code development and in general helps simplify many other complex operations. More information on the C# language, its syntax or the CLR is available at (7).

Overall the PSIM is fairly complex from a code standpoint. To help simplify the development process the PSIM was built using four distinct code libraries described in **Error! Reference source not found..**

The PSIM project was designed so that components could be easily modeled and rapidly tested. To do this, the component code was modeled after the VHSIC Hardware Description Language (VHDL) process statement. In VHDL a process block is triggered when any of the inputs in its sensitivity list change. Often this is used for synchronous circuits to build state machines and edge-triggered components; however, if the sensitivity list includes every input it can be treated like a combinational circuit. Due to this flexibility the PSIM code was designed so that all pins could be attached to an event list in C# and get

File Name	Description
PSIMedu.exe	Main executable file which defines the main architecture to be simulated. It instantiates the components to be used as well as the interconnections between those components.
PSIMedu.Core.dll	Defines the instruction set and assembler, as well as all data types and IO such as the Bus, InputPin, and OutputPin class.
PSIMedu.Components.dll	Defines all of the components that can be used in an architecture (Table 3).
PSIMedu.Drawing.dll	Defines the drawing control that facilitates the display of the components used in an architecture. It also manages the display of bus data and information exchanged between components.

called when an input is changed. In Figure 10 an example VHDL process is given that

can work to create a synchronous counter. In comparison **Error!**

Reference source not found.

```

Process(CLK, RST) begin
  If (RST = '1') then
    value <= (others => '0');
  Elsif (CLK = '1' and CLK'event) then
    if (LOAD = '1') then
      value <= din;
    Else then
      value <= value + 1;
    End If;
  End If;
End Process;

```

Figure 10 - Typical VHDL Process for a Counter

```

if (_reset.GetValue()) {
  _value.Reset();
  _output.UpdateValue(_value.GetCopy());
}
else if (p == _clock && e.Value == true) {
  if (_load.GetValue()) {
    _value = _din.GetValue();
    _overflow.UpdateValue(false);
  }
  else {
    if (_value.Increase(1))
      _overflow.UpdateValue(true);
    else
      _overflow.UpdateValue(false);
  }
  _output.UpdateValue(_value.GetCopy());
}

```

Figure 11- PSIM Process for a Counter

shows an example of the same counter process in the PSIM processor to create a similar counter component. In the PSIM example a “sensitivity list” was created which included the clock and reset pin so that when the clock changes or a reset is triggered the function is automatically called to update any outputs as necessary. Overall the PSIM code was able to take advantage of C# events to accurately and easily emulate a VHDL process for component design; however, other ideas such as variable width buses proved to be much more difficult.

Data types proved to be one of the most difficult items to simulate. In C#, data types are of a fixed width usually in multiples of 8 bits: boolean (1-bit), byte (8-bit), short (16-bit), integer (32-bit), long (64-bit), etc. This limitation does not exist in hardware where signals are commonly just enough bits to accommodate the necessary data to be exchanged. To facilitate this difference, a new class had to be created for every data type that any component would use. These data classes are time consuming to implement due to their many supporting and repetitive methods. In fact, as implemented the bit data type (which represents a single 1 or 0) and the other implemented data types are around 1400 lines of code. In contrast the PSIM controller component which is the main processing unit for the architecture is only 800 lines of code.

In the future many improvements can be made to the underlying code of the PSIM especially considering the inefficiencies of the data type code. Future revisions should implement data types more closely resembling those of VHDL such as ‘bit’ and ‘bit_vector’ so that components can more easily be transcribed from VHDL for use in the PSIM.

CHAPTER 6

Summary and Conclusions

The PSIM simulator has evolved from its original inception (1), but there are a number of different improvements that can be made to both the simulator program and the underlying architecture that is simulated that can help improve the programs ability to function as an educational tool. Most importantly the simulator's architecture needs to be updated so that it can be accurately synthesized via a hardware description language such as the VHDL. To achieve this many of the components need to be rewritten such that they are triggered on the rising edge of the clock instead of being level sensitive. With a hardware implementation of the PSIM architecture, students could gain an even broader understanding of the architecture through first hand implementation use on an FPGA.

In addition to making adjustments to the current architecture, the simulator should be improved to allow the simulation of other architectures. The current architecture is a basic architecture that works well as an introduction to embedded processors; however, once a student has mastered this simple case there is no way to specify and simulate more complex architectures and components. This seriously impedes the ability of the PSIM to function as an educational aid for more advanced classes. To address this, a future version of PSIM should function more as a generic logic simulator. Ideally the PSIM should be able to accept any type of higher level logic component or architecture without having to be rebuilt or completely redesigned from the source code. With this enhancement the PSIM can be used

to simulate any architecture or component ranging from the more advanced pipeline architecture to a floating point co-processor. To facilitate this generalization of the PSIM, components will also need to be changed. With the current iteration of PSIM, any change to the architecture or any one of its digital components requires recompiling the PSIM from its source code. As a professor or demonstrator of PSIM, this is unacceptable as it requires a much deeper knowledge of the PSIM's source code than should be required. To remedy this problem, components should be moved out of the source code and into library files so that new logic components can be easily added and edited. Once external to the simulator, any user can design and simulate any architecture (provided that the component descriptive language is straightforward and easy to use).

The second version of PSIM is currently in development and seeks to address all of the above issues with the current PSIM. The PSIM has been modified to resemble a development environment, where architectures can be created via a drag and drop interface and new architectures can be quickly designed and simulated. This change improves the usability of the PSIM as an educational tool and also opens up the possibility for PSIM to be used as a research tool to understand and design new architectures. Though no development effort has been put forward to address research use, it is conceivable that more tools could be integrated into the PSIM to help researchers build new architectures such as clock cycle counters and efficiency analyzers.

Overall the PSIM project has started to achieve its original goal of providing a straightforward, easy to use, graphical tool that can be used to aid students in learning the architecture of embedded processor systems. With future improvements and development on

a new version the PSIM project can fully achieve its goals and even expand into new areas such as research and development.

Appendix A

The Instruction Set

Basic Instructions

Halt Instruction

The halt instruction ends the execution of the processor. This instruction is recommended to be placed at the end of ALL programs. Its OP Code is 0. It requires no parameters.

Format

- HALT

Aliases

This instruction has no aliases

More Information

The HALT instruction takes 2 clock cycles to execute, the first to FETCH, the second to decode and HALT.

Load Instruction

The load instruction takes two parameters the first must be a register, while the second can be either a register or a value. Its OP Code is 1.

Format

- LOAD %a %b
- LOAD %a 5

Aliases

- LDA – Loads a value into a
- LDB – Loads a value into b
- LDC – Loads a value into c
- Example: LDA 5 or LDB %a

Additional Information

The LOAD instruction takes 3 clock cycles to complete: one to fetch, the second for decode and setup, the third to latch the values and begin the next cycle.

LDCR Instruction

The LDCR instruction loads the carry flag into a user register. This can be used to perform addition with carry or other more complex operations. It takes a single parameter that is a register. Its OP Code is 16.

Format

- LDCR %a

Aliases

This instruction has no aliases

Additional Information

The LDCR instruction takes 3 clock cycles to complete: one to fetch, the second to decode and setup, the third to latch the values and begin the next cycle.

LDZR Instruction

The LDZR instruction loads the zero flag into a user register. It takes a single parameter that is a register. Its OP Code is 17.

Format

- LDZR %a

Aliases

This instruction has no aliases

Additional Information

The LDZR instruction takes 3 clock cycles to complete: one to fetch, the second to decode and setup, the third to latch the values and begin the next cycle.

LDM Instruction

The LDM instruction loads a value at a memory address into a register. It takes a register value and either a register holding the memory address or an operand with the memory address. Its OP Code is 23.

Format

- LDM %a 5
- LDM %a %b

Aliases

- LDMA – Loads register A with the memory value at an address
- LDMB – Loads register B with the memory value at an address
- LDMC – Loads register C with the memory value at an address
- LDMA 5 – Loads register a with the memory value at address 5

Additional Information

The LDM instruction takes 4 clock cycles to complete: one to fetch, the second to decode and setup, the third to latch the value from the RAM, and the fourth to latch the value into a register

STM Instruction

The STM instruction stores a registers value into a memory location. It takes a register as its source and a register that holds its destination address. Unfortunately due to architecture limitations, this instruction cannot take an operand for its memory address. Its OP Code is 24.

Format

- STM %a %b

Aliases

- STMA – Loads register A with the memory value at an address
- STMB – Loads register B with the memory value at an address
- STMC – Loads register C with the memory value at an address
- STMA %b – Loads register a with the memory value at the address in register b

Additional Information

The STM instruction takes 3 clock cycles to complete: one to fetch, the second to decode and setup, the third to save the value into the RAM at the correct address.

Arithmetic Instructions

Add Instruction

The add instruction takes two operands. The first operand must be a register while the second can either be a register or a value. It adds two numbers together and stores them in the register provided in operand one. Its OP Code is 2.

Format

- ADD %a %b
- ADD %c \$A3

Aliases

- ADDA – Adds a value or register to a
- ADDB – Adds a value to register to b
- ADDC – Adds a value to register to c
- Example: ADDA %b

Addition Information

The Add instruction takes 4 clock cycles to complete: one to fetch, the second to setup the ALU and its inputs, the third to push the values back out to the register, the fourth to latch the values and move to the next cycle.

Sub Instruction

The sub instruction takes two operands. The first operand must be a register while the second can either be a register or a value. It subtracts two numbers together and stores them in the register provided in operand one. Its OP Code is 3.

Format

- SUB %a %b
- SUB %c \$A3

Aliases

- SUBA – Subtracts a value or register to A
- SUBB – Subtracts a value to register to B
- SUBC – Subtracts a value to register to C
- Example: SUBA 4

Addition Information

The Sub instruction takes 4 clock cycles to complete: one to fetch, the second to setup the ALU and its inputs, the third to push the values back out to the register, the fourth to latch the values and move to the next cycle.

Logical Instructions

AND Instruction

The AND instruction takes two operands. The first operand must be a register while the second can either be a register or a value. It will take the two operands and “and” them and store the result in the register provided in operand one. Its OP Code is 4.

Format

- AND %a %b
- AND %a \$F

Aliases

- ANDA – ANDs a value or register with A
- ANDB – ANDs a value or register with B
- ANDC – ANDs a value or register with C
- Example: ANDA 8

Additional Information

The AND instruction takes 4 clock cycles to complete: one to fetch, the second to setup the ALU and its inputs, the third to push the values back out to the register, the fourth to latch the values and move to the next cycle.

OR Instruction

The OR instruction takes two operands. The first operand must be a register while the second can either be a register or a value. It will take the two operands and “or” them and store the result in the register provided in operand one. Its OP Code is 5.

Format

- OR %a %b
- OR %a \$f

Aliases

- ORA – ORs a value or register with A
- ORB – ORs a value or register with B
- ORC – ORs a value or register with C
- Example: ORA %b

Additional Information

The OR Instruction takes 4 clock cycles to complete: one to fetch, the second to setup the ALU and its inputs, the third to push the values back out to the register, the fourth to latch the values and move to the next cycle.

NOT Instruction

The NOT instruction takes one operand. The operand must be a register. It will take the complement of the register and put the new value back into the register. Its OP Code is 6.

Format

- NOT %a

Aliases

- NOTA – Complements register A, takes no parameters
- NOTB – Complements register B, takes no parameters
- NOTC – Complements register C, takes no parameters
- Example: NOTA

Additional Information

The NOT Instruction takes 4 clock cycles to complete: one to fetch, the second to setup the ALU and its inputs, the third to push the values back out to the register, the fourth to latch the values and move to the next cycle

XOR Instruction

The XOR instruction takes two operands. The first operand must be a register while the second operand may be either a register or value. This will take the two operands XOR them together and store the result into the register provided in operand one. Its OP Code is 7.

Format

- XOR %a %c
- XOR %b 5

Aliases

- XORA – XORs register A with a value or register
- XORB – XORs register B with a value or register
- XORC – XORs register C with a value or register
- Example: XORA \$37

Additional Information

The XOR instruction takes 4 clock cycles to complete: one to fetch, the second to setup the ALU and its inputs, the third to push the values back out to the register, the fourth to latch the values and move to the next cycle.

IO Instructions

IN Instruction

The IN instruction takes a single operand. The first operand must be a register to place the input registers value into. Its OP code is 8.

Format

- IN %a

Aliases

- INA – Loads the input into register A
- INB – Loads the input into register B
- INC – Loads the input into register C
- Example: INA

Additional Information

The IN instruction takes 3 clock cycles to complete: one to fetch, the second to setup the destination register, the third to latch the input into the register and move to the next cycle.

OUT Instruction

The OUT instruction takes a single operand. The first operand must be a register whose output will be placed into the output register. Its OP code is 9.

Format

- OUT %a

Aliases

- OUTA – Loads register A's value into the Output register
- OUTB – Loads register B's value into the Output register
- OUTC – Loads register C's value into the Output register
- Example: OUTA

Additional Information

The OUT instruction takes 3 clock cycles to complete: one to fetch, the second to setup the multiplexor and output register, the third to latch the data.

Shift Instructions

LSL Instruction

The LSL instruction takes two operands. The first is a register to shift and the second operand is either a register or a value giving the number of positions to shift. The instruction will shift whatever value given to the left as many times as needed then place the value back into the register. Its OP code is 10.

Format

- LSL %a 2
- LSL %b %a

Aliases

- LSLA – Shifts register A's value
- LSLB – Shifts register B's value
- LSLC – Shifts register C's value
- Example: LSLA 2

Additional Information

The LSL instruction takes 4 clock cycles to complete: one to fetch, the second to setup the ALU inputs and perform the operation, the third to setup the register to save the value into, and the fourth to latch the data and start the next cycle.

LSR Instruction

The LSR instruction takes two operands. The first is a register to shift and the second operand is either a register or a value giving the number of positions to shift. The instruction will shift whatever value given to the right as many times as needed then place the value back into the register. Its OP code is 11.

Format

- LSR %a 2
- LSR %b %a

Aliases

- LSRA – Shifts register A's value
- LSRB – Shifts register B's value
- LSRC – Shifts register C's value
- Example: LSRA 2

Additional Information

The LSR instruction takes 4 clock cycles to complete: one to fetch, the second to setup the ALU inputs and perform the operation, the third to setup the register to save the value into, and the fourth to latch the data and start the next cycle.

ROL Instruction

The ROL instruction takes two operands. The first is a register to shift and the second operand is either a register or a value giving the number of positions to rotate. The instruction will rotate the value left taking its most significant bit and making it the LSB. The remaining bits will all shift left. Its OP code is 12.

Format

- ROL %a 2
- ROL %b %a

Aliases

- ROLA – Rotates register A’s value
- ROLB – Rotates register B’s value
- ROLC – Rotates register C’s value
- Example: ROLA 2

Additional Information

The ROL instruction takes 4 clock cycles to complete: one to fetch, the second to setup the ALU inputs and perform the operation, the third to setup the register to save the value into, and the fourth to latch the data and start the next cycle.

ROR Instruction

The ROR instruction takes two operands. The first is a register to shift and the second operand is either a register or a value giving the number of positions to rotate. The instruction will rotate the value stored in the register to the right, taking its LSB and making it the MSB. It will shift the other bits to the right. Its OP code is 13.

Format

- ROR %a 2
- ROR %b %a

Aliases

- RORA – Rotates register A's value
- RORB – Rotates register B's value
- RORC – Rotates register C's value
- Example: RORA 2

Additional Information

The ROR instruction takes 4 clock cycles to complete: one to fetch, the second to setup the ALU inputs and perform the operation, the third to setup the register to save the value into, and the fourth to latch the data and start the next cycle.

Branch and Jump Instructions

JUMP Instruction

The JUMP instruction takes a single operand. This operand must be a label in the format of @label. The label must correspond to a label previously declared in the assembly file. The assembler will translate this to the appropriate value. Its OP code is 14.

Format

- JUMP @label

Aliases

This instruction has no aliases.

Additional Information

The Jump instruction takes 2 clock cycles to complete. One to fetch the instruction, and the second to change the program counter's value.

BEQ Instruction

The BEQ instruction takes three operands. The first and second operand must be registers

whose equality should be checked. The third operand must be a label in the form of @label.

At assembly the assembler will change this to the appropriate value. If the registers are equal

then the branch will be taken to the label provided. Its OP code is 15.

Format

- BEQ %a, %b, @label

Aliases

This instruction has no aliases.

Additional Information

The branch when equal instruction takes 3 clock cycles to complete. One to fetch the

instruction, the second to setup the equality, the third to branch if equal.

BNE Instruction

The BNE instruction takes three operands. The first and second operand must be registers whose equality should be checked. The third operand must be a label in the form of @label. At assembly the assembler will change this to the appropriate value. If the registers are not equal then the branch will be taken to the label provided. Its OP code is 18.

Format

- BNE %a, %b, @label

Aliases

This instruction has no aliases.

Additional Information

The branch when equal instruction takes 3 clock cycles to complete. One to fetch the instruction, the second to setup the equality, the third to branch if not equal.

BRZ Instruction

The BRZ instruction takes a single operand that must be a label in the form of @label. At assembly the assembler will change this to the appropriate value. If the zero register is a one then the branch will be taken to the label provided. Its OP code is 19.

Format

- BRZ @label

Aliases

This instruction has no aliases.

Additional Information

The branch when zero instruction takes 2 clock cycles to complete. The first cycle fetches the instruction, the second checks the zero flag and branches if one.

BNZ Instruction

The BNZ instruction takes a single operand that must be a label in the form of @label. At assembly the assembler will change this to the appropriate value. If the zero register is a zero then the branch will be taken to the label provided. Its OP code is 20.

Format

- BNZ @label

Aliases

This instruction has no aliases.

Additional Information

The branch when not zero instruction takes 2 clock cycles to complete. The first cycle fetches the instruction, the second checks the zero flag and branches if zero.

BRC Instruction

The BRC instruction takes a single operand that must be a label in the form of @label. At assembly the assembler will change this to the appropriate value. If the carry register is a one then the branch will be taken to the label provided. Its OP code is 21.

Format

- BRC @label

Aliases

This instruction has no aliases.

Additional Information

The branch when carry instruction takes 2 clock cycles to complete. The first cycle fetches the instruction, the second checks the carry flag and branches if one.

BNC Instruction

The BNC instruction takes a single operand that must be a label in the form of @label. At assembly the assembler will change this to the appropriate value. If the carry register is a zero then the branch will be taken to the label provided. Its OP code is 22.

Format

- BNC @label

Aliases

This instruction has no aliases.

Additional Information

The branch when carry instruction takes 2 clock cycles to complete. The first cycle fetches the instruction, the second checks the carry flag and branches if zero.

Appendix B
Example Programs

Simple Multiplier

```
; Simple Multiplier  
; Multiplies an input value by 5  
INA  
LDB 5  
LDC 0  
  
; Multiplication Loop  
:Mult  
ADDC %a  
SUBB 1  
BNZ @Mult  
  
; Output the value in Register C  
OUTC  
HALT
```

Overflow Detector

```
; Overflow Detector
; Increases the number $FA until it overflows then halts
LDA $FA
LDC 1

:Increase A by one, if there is not a carry repeat
ADDA 1
BNC @Increase

HALT
```

Byte Maker

```
; Combine two four bit words to make a byte
; Load $03 into A and $0E into B
LDA $03
LDB $0E

; Left Shift B by 4 and or it with A
LSLB 4
ORA %b

; Outputs $E3
OUTA
HALT
```

Memory Store and Load

```
; STORE and LOAD from Memory
```

```
LDA 1
```

```
LDB 5
```

```
STMA %b
```

```
; Stores 5 into memory address 3
```

```
LDA 5
```

```
LDB 3
```

```
STMA %b
```

```
; Stores 2 into memory address 9
```

```
LDA 2
```

```
LDB 9
```

```
STMA %b
```

```
; Load mem[3] into A
```

```
LDMA 3
```

```
; Load mem[%a] into C
```

```
LDMC %a
```

```
HALT
```

Appendix C

Controller Signals

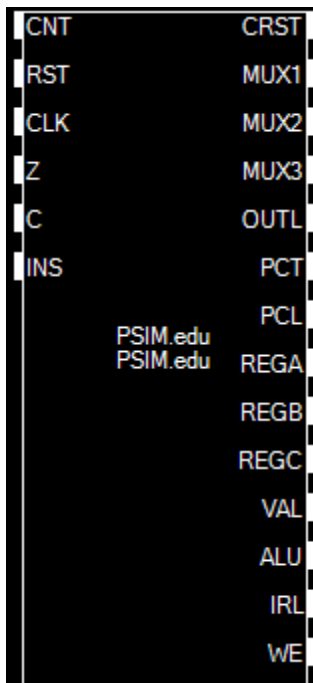


Figure 12 - PSIM Controller Block

The PSIM controller block has many different input and output ports which control different functionality of the different logic blocks.

This section will explain each input and outputs signal in more detail and give a better understanding of how the state machine operates.

Inputs

The PSIM controller has 6 input ports:

- CNT – This is the current counter value provided by the PSIM counter. This helps in state management and allows each instruction to take a variable amount of clock cycles
- RST – Active high reset pin that resets the state machine
- CLK – The clock signal

- Z – The ALU zero flag
- C – the ALU carry flag
- INS – The value in the instruction register

Outputs

The PSIM controller has 13 drawn output ports plus an additional port that is not drawn:

- CRST – Resets the PSIM counter, this is done after every instruction is created
- MUX1 – This is the MUX1 select pin which selects a value into the registers
- MUX2 – This is the MUX2 select pin which selects a value to output from the registers
- MUX3 – This is the MUX3 select pin which selects a value to output from the registers or a instruction value
- OUTL – This is the load signal for the output register, raising this causes the output register to load any value on its input
- PCT – Program Counter Trigger port, raising this increments the program counter
- PCL – Program Counter Load port, raising this causes the PC to load whatever value is on its DIN port
- REGA, REGB, REGC – Each of these ports corresponds to the load pin on the appropriate user register, raising this will cause whatever value is on its input to be loaded
- VAL – If a value is associated with an instruction, this will be where it is outputted. This pin is attached to the 4th input of MUX3 so it can be used in operations
- ALU – This is where the current ALU output mode select is given to the ALU. For example the ALU mode of 0 corresponds to addition

- IRL – Instruction Register Load, when raised the instruction register will load the contents of the ROM's currently selected address
- RALU – A pin not shown for drawing purposes, this pin causes the register after the ALU to load the value of an ALU operation so it can be latched and used later
- WE – The write enable signal for the RAM
- RLD – The RAM Register Load Signal

Works Cited

1. *PSIM Processor SIMulator (version 4.2)*. **Stroud, Charles**. July 23, 2003. available at www.eng.auburn.edu/~strouce/ausim.html.
2. **Mano, Morris**. *Computer Engineering Hardware Design*. s.l. : Prentice Hall, 1988. pp. 280-292.
3. **Carpinelli, John D**. Companion Website for Computer Systems Organization and Architecture. *Pearson Education*. [Online] www.awl.com/carpinelli.
4. **Stroud, Charles**. *Personal Communication*. October 2009.
5. *Educational Simulation of the RiSC Processor*. **Jaumain, Marc**. 2007, International Conference for Engineering Education, pp. 436-441.
6. *The RiSC-16 Instruction-Set Architecture*. **Jacob, Bruce**. Fall 2000, ENEE 446: Digital Computer Design. Available: <http://www.ece.umd.edu/~blj/RiSC/RiSC-isa.pdf>.
7. **Microsoft**. The C# Language. *Visual C# Developer Center*. [Online] <http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx>.