A FPGA Implementation of a MIPS RISC Processor for Computer Architecture Education

By:
Victor P. Rubio, B.S.
vrubio@gauss.nmsu.edu

Advisor:
Dr. Jeanine Cook
jecook@gauss.nmsu.edu

New Mexico State University

Las Cruces New Mexico

July 2004

ABSTRACT

A FPGA Implementation of a MIPS RISC Processor
for Computer Architecture Education

By

Victor P. Rubio, B.S.

Master of Science
New Mexico State University
Las Cruces, New Mexico

Chair of Committee:
Dr. Jeanine Cook
Electrical Engineering

Date: July 8, 2004
Room: T & B 104
Time: 1:30 PM

Computer organization and design is a common engineering course where students learn concepts of modern computer architecture. Students often learn computer design by implementing individual sections of a computer microprocessor using a simulation-only approach that limits a students experience to software design. This project targets the computer architecture courses and presents an FPGA (Field Programmable Gate Array) implementation of a MIPS (Microprocessor without Interlocked Pipeline Stages) RISC (Reduced Instruction Set Computer) Processor via VHDL (Very high speed integrated circuit Hardware Description Language) design. The goal of this project is to enhance the simulator based approach by integrating some hardware design to help the computer architecture students gain a hands-on experience in hardware-software integration and achieve a better understanding of both the MIPS single-cycle and pipelined processors as described in the widely used book, *Computer Organization and Design – The Hardware/Software Interface* by David A. Patterson and John L. Hennessy.

CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

INTRODUCTION

Computer organization and design is a common engineering course where students learn concepts of modern computer architecture. Students often learn computer design by implementing individual sections of a computer microprocessor using a simulation-only approach that limits a students experience to software design. As a result the students are not given the chance to implement and run their designs in real hardware, thus missing a good opportunity to gain a complete hands-on experience involves hardware-software integration.

This project targets the computer architecture courses and presents an FPGA (Field Programmable Gate Array) implementation design of a MIPS (Microprocessor without Interlocked Pipeline Stages) RISC (Reduced Instruction Set Computer) Processor using VHDL (Very high speed integrated circuit Hardware Description Language). Furthermore, the goal of this work is to enhance the simulator-based approach by integrating some hardware design to help the computer architecture students gain a better understanding of both the MIPS single-cycle and pipelined processor as described in the widely used book, *Computer Organization and Design – The Hardware/Software Interface* by David A. Patterson and John L. Hennessy [1].

FPGAS

An FPGA is a programmable logic device (PLD) that can be reprogrammed any number of times after it has been manufactured. Internally FPGAs contain gate arrays of pre-manufactured programmable logic elements called cells. A single cell can implement a network of several logic gates that are fed into flip-flops. These logic elements are internally arranged in a matrix configuration and are automatically connected to one another using a programmable interconnection network. The re-programmable nature of FPGAs makes them ideal for

educational purposes because it allows the students to attempt as many iteration as necessary to correct and optimized their processor design. The FPGAs are very desirable in the academic community because they can be recycled year after year and can be obtained at a relatively low cost.

Today there are many different manufacturers of FPGA devices including Actel, Altera, Atmel, Cypress, Lucent and Xilinx. However the biggest sponsors of the academic community are Altera and Xilinx, which boast "University Programs" offering discounts on software and hardware. By providing educational opportunities they promote education and research using these programmable logic device technologies. Both programs offer different development boards that come equipped with an FPGA device, oscillator, seven-segment LED display, a PS/2 keyboard/mouse port, a VGA video output display port, a micro-controller, prototyping connectors, voltage regulators, a parallel port and DC input jack. These development boards make an excellent resource for students to have available to them to learn digital logic design using industry-standard development tools and PLDs.

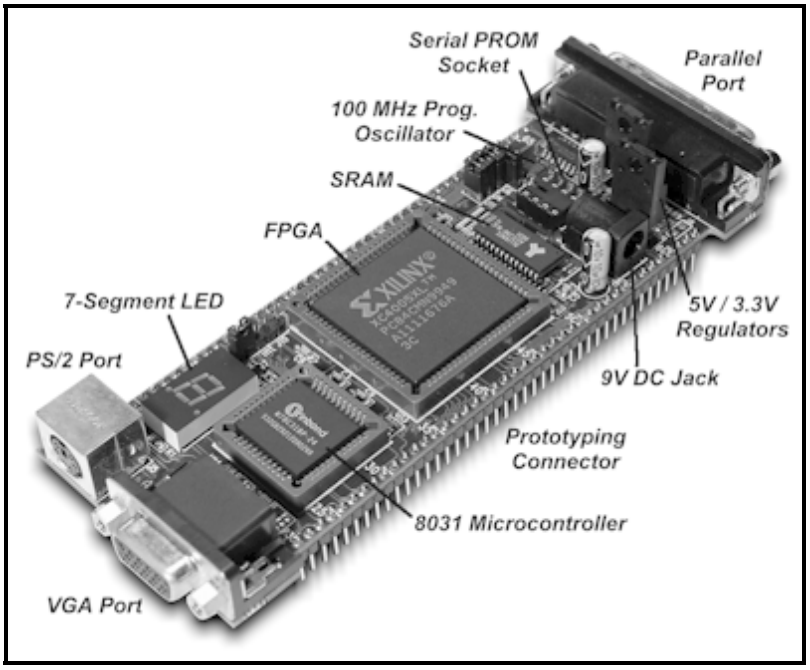

Figure 1.1 Xilinx XS-40005XL

The Xilinx University Program has a partnership with the Xess Corporation that offers the XS-40-0005XL Prototyping Board shown in Figure 1.1. The board comes equipped with an XC40005XL FPGA, 8031 microcontroller, and 32 KB of static random access memory (SRAM), which are used to configure the device. The XC40005XL FPGA is part of the Xilinx 4000 device family and has 9,000 usable gates. This device contains a more complex logic element called a configurable logic block (CLB). These CLBs contain three SRAM based lookup tables (LUT), which provide inputs into two flip flops and other CLBs as seen in Figure 1.2. The CLBs are arranged in a square matrix configuration so that they are interconnected via a programmable hierarchical interconnection network. The Xilinx 4000 device family contains between 100 to 3,136 CLBs.

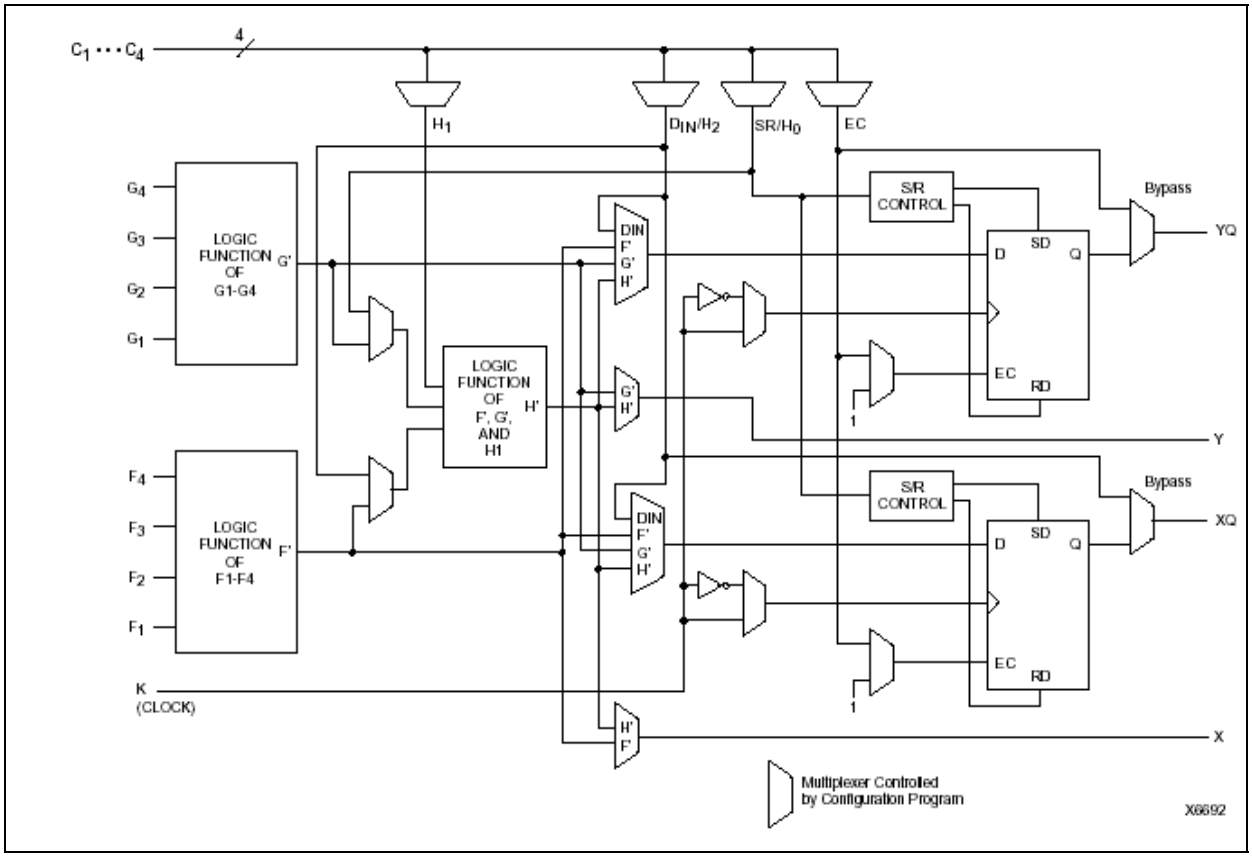

Figure 1.2 XC4000 Series CLB Block Diagram

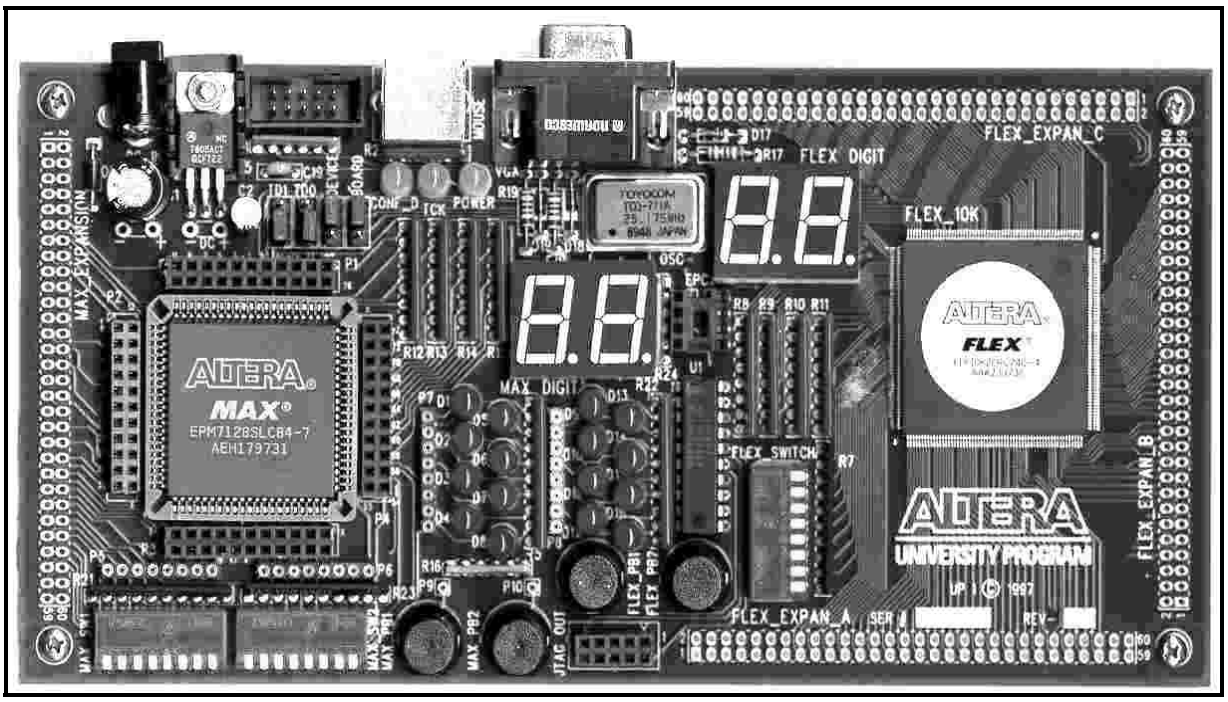

Figure 1.3 Altera UP2 Development Board

The Altera University Program offers the UP2 Development Board shown in Figure 1.3. The board comes equipped with both a EPF10K70 FPGA device based on SRAM technology and an EPM7128S PLD based on erasable programmable read-only memory (EEPROM) elements. The EPF10K70 is part of Alteras Flexible Logic Element matriX (FLEX) 10K family that comes with 10,000 to 250,00 gates. The EPF10K70 device equipped on the UP2 development board comes with 70,000 usable gates that use row and column programmable interconnections. This device contains nine embedded array blocks (EAB) that provide up to 2,048 bits of memory each and can be used to create random access memory (RAM), read only memory (ROM) or first-in first-out (FIFO) functions used to configure the device. The EPF10K70 has 3,744 logic elements (LE) each consisting of a four-input LUT, that are used to model any network of gates with four inputs and one output. The EPF10K70 device internally combines eight LEs to compose a logic array block (LAB). This device is equipped with 268 LABs. Figure 1.4 shows the block diagram for the FLEX10K70 device.

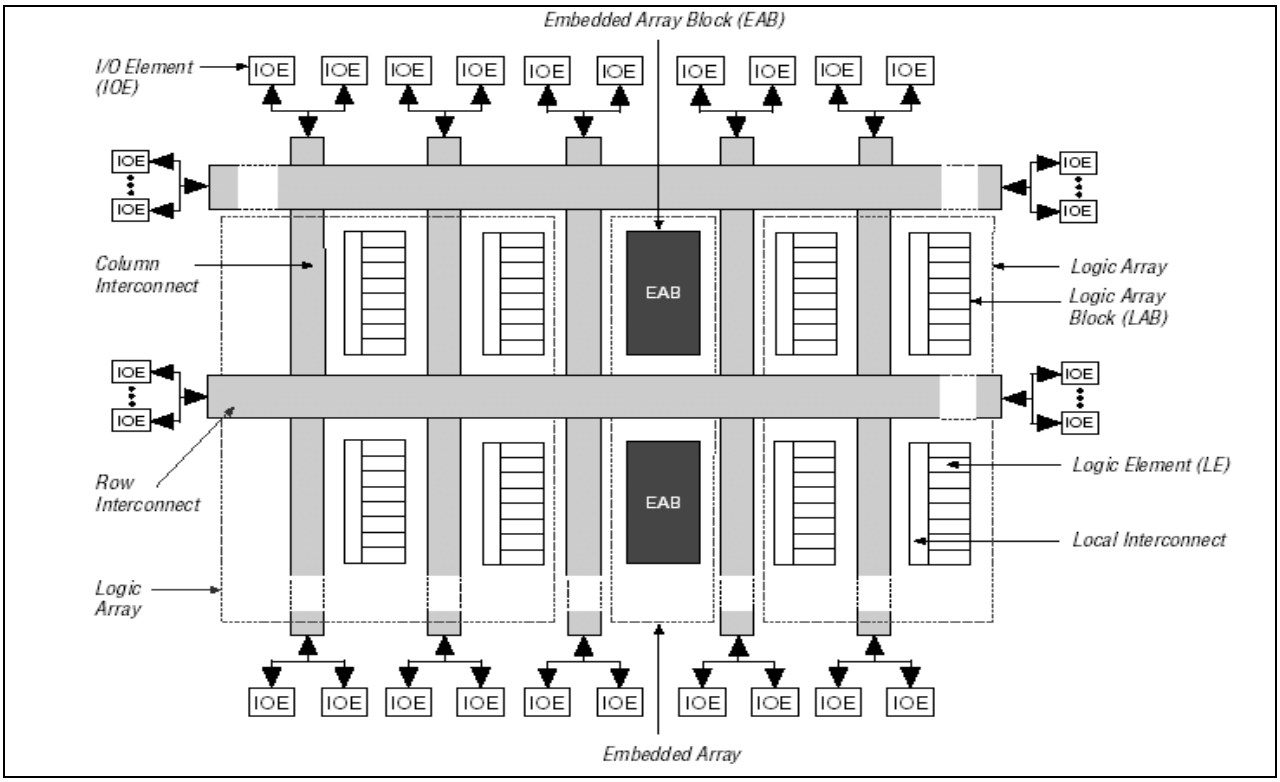

Figure 1.4 Altera FLEX10K70 Block Diagram

The Altera EPM7128S is a PLD device that belongs to the MAX 7000S family and comes with 2,500 usable gates. This device is configured using an erasable programmable read-only memory (EEPROM) element, whose configuration is retained even when the power is removed. The MAX7000S family uses logic elements called macrocells consisting of a programmable AND/OR network designed to implement Boolean equations as shown in Figure 1.5. The outputs of these networks are fed into programmable flip flops. Macrocells combined in groups of 16 create the MAX7000S device family LABs. The EPM7128S device contains a total of 160 macrocells and 8 LABs which are all interconnected via a programmable interconnect array (PIA). Figure 1.6 shows the Altera EPM7128S device block diagram.

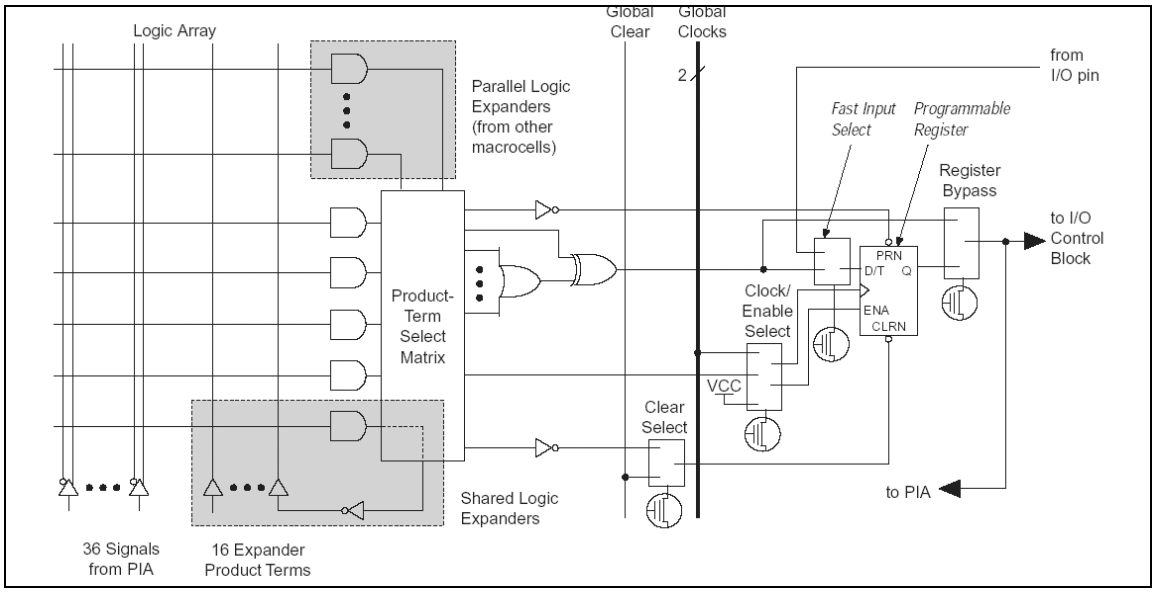

Figure 1.5 Altera EPM7128S Device Macrocell

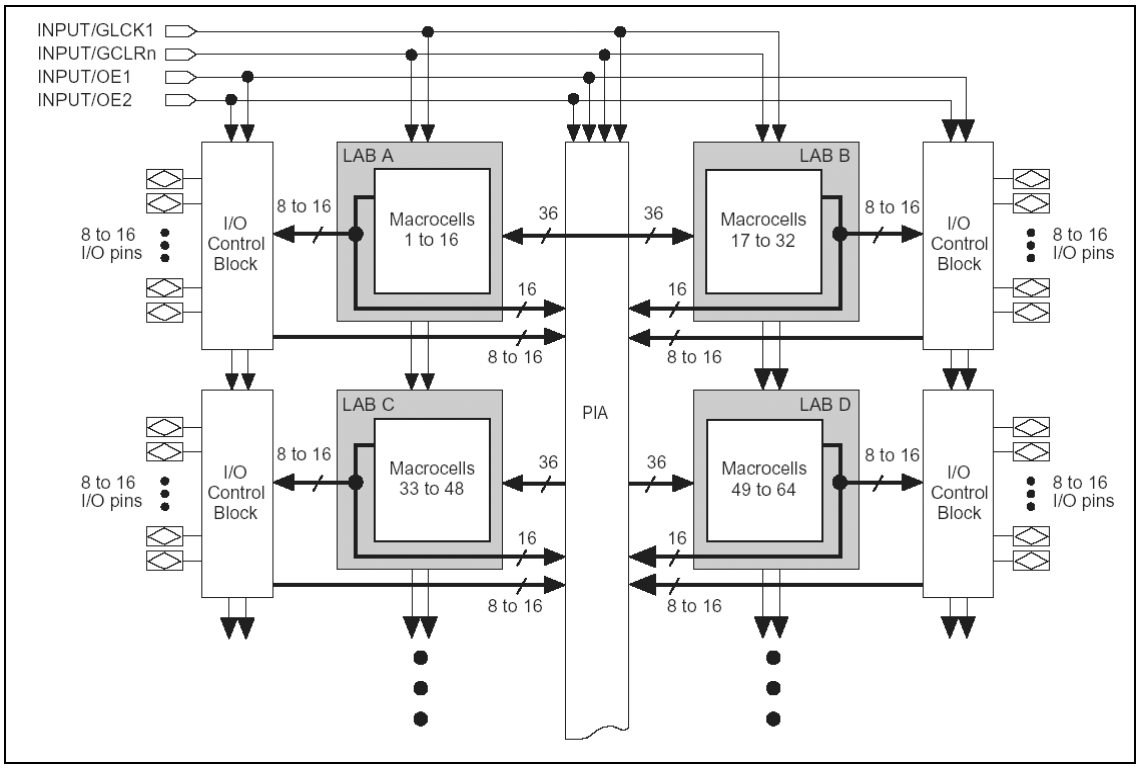

Figure 1.6 Altera EPM7128S Block Diagram

THE MIPS PROCESSOR

The MIPS instruction set architecture (ISA) is a RISC based microprocessor architecture that was developed by MIPS Computer Systems Inc. in the early 1980s. MIPS is now an

industry standard and the performance leader within the embedded industry. Their designs can be found in Canon digital cameras, Windows CE devices, Cisco Routers, Sony Play Station 2 game consoles, and many more products used in our everyday lives. By the late 1990s it was estimated that one in three of all RISC chips produced was a MIPS-based design [5]. MIPS RISC microprocessor architecture characteristics include: fix-length straightforward decoded instruction format, memory accesses limited to load and store instructions, hardwired control unit, a large general purpose register file, and all operations are done within the registers of the microprocessor. Due to these design characteristics, computer architecture courses in university and technical schools around the world often study the MIPS architecture. One of the most widely used tools that helps students understand MIPS is SPIM (MIPS spelled backwards) a software simulator that enables the user to read and write MIPS assembly language programs and execute them. SPIM is a great tool because it allows the user to execute programs one step or instruction at a time. This then allows the user to see exactly what is happening during their program execution. SPIM also provides a window displaying all general purpose registers which can also be used during the debug of a program. This simulator is another impressive tool that gives the computer architecture students an opportunity to visually observe how the MIPS processor works [6].

CAD SOFTWARE

Using a sophisticated computer aided design (CAD) software tool such as Altera's Multiple Array MatriX Programmable Logic User System (MAX+PLUS II), one can design complex logic circuits such as the MIPS Processor. MAX+PLUS II offers a graphical user interface including eleven fully integrated applications that offer: a variety of design entry

methods for hierarchical design, powerful logic synthesis, timing-driven compilation, partitioning, function and timing simulation, linked multi-device simulation, timing analysis, automatic error location, and device programming and verification. Figure 1.7 shows how the eleven integrated applications are grouped within MAX+PLUS II. The industry-standard design language VHDL can be used in the MAX+PLUS II text editor to specify logic circuits including the various components of the single-cycle and pipelined implementation of the MIPS processor. Once the VHDL code is complete, MAX+PLUS II will translate upon request, optimize, synthesize and save a text-based representation of a logic diagram. MAX+PLUS II can then fit the circuit design onto the device's logic elements and programmable interconnection network. This allows the designer to perform a simulation using actual logic gate and interconnect timing delays based on the assigned PLD. The final step is to download and configure the actual PLD with the program and perform a hardware verification of the design.

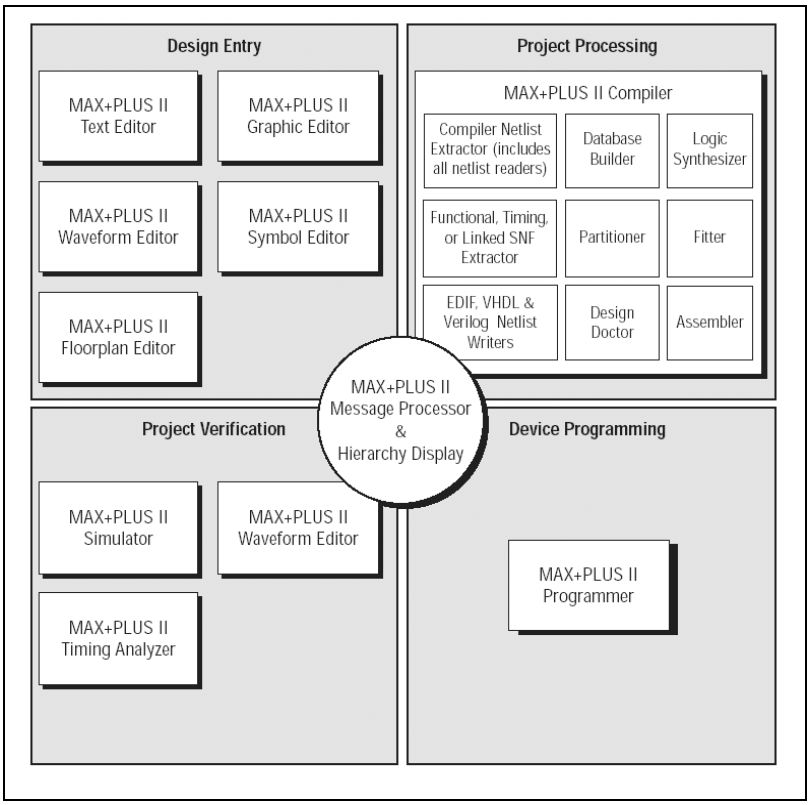

Figure 1.7 MAX+PLUS II Applications

2 RELATED WORK

In the early 1990s, Professors H. B. Diab and I. Demashkieh from the American University of Beirut in Beirut, Lebanon where the first to determine new methods of effective microprocessor architecture education. In their paper *A Re-configurable Microprocessor Teaching Tool* [8], they introduced an interactive, flexible, user-friendly software package to help describe how an 8-bit CPU functions internally as the master of a micro-computer system. The tool aids students by providing a graphical step by step animation of how the CPU works. This tool simulates the CPU control logic, internal registers, buses and memory contents at every clock edge. It can also simulate read and write cycle to/from memory and input/output devices. The software package also includes an assembler allowing students to choose either assembly code or machine code to run on the microprocessor, much like the newer SPIM simulator. Such a tool enables students to see first hand how the different parts of the microprocessor interact and how they combine into a working microprocessor, thus introducing one of the first simulator-based approaches to microprocessor instruction.

More recently in early 2000, Jan Gray, a software developer from Gray Research, initiated a strong campaign to use FPGA implementations in microprocessor instruction [9]. However, his work demonstrates optimizing CPU design to achieve a cost-effective integrated computer system in an FPGA.

One goal of this project is to use the ideas presented in [8] to create an effective method to teach microprocessor design by giving the computer architecture students a complete hands-on experience with hardware-software integration, a technique that several other universities have implemented to help teach computer design courses. One of the earliest examples of this arises

from Cornell University where in 1998 the EE 475 architecture class projects included a VHDL design and FGPA verification of a simple processor [14]. Most recently, in 2001 at Hiroshima City University located in Japan it is reported that within a 15 week time period, 7 out of 39 junior students succeeded in implementing hardware description language (HDL) descriptions of a superscalar RISC processor onto an FPGA. In 2002 in the same 15 week time period, 14 out of 47 junior students implemented varieties of superscalar CISC/RISC processor within an FPGA [15].

3 FPGA IMPLEMENTATION OF THE MIPS PROCESSOR

The main task of this project is to implement a single-cycle and pipelined representation of the MIPS processor onto an FPGA so that it models the processor presented in Chapters 5 and 6 of the book *Computer Organization and Design – The Hardware/Software Interface* by David A. Patterson and John L. Hennessy [1]. The Altera UP2 Development Board shown in Figure 3.1 was chosen to implement the VHDL design. The development board features one EPM7128S PLD and one FLEX10K70 FPGA. Each device has the following resources: a JTAG chain connection for the ByteBlaster II cable (used to program the device), two push-button switches, dual-digit seven-segment displays, and on-board oscillator (25.175 MHz). The EPM7128S device also has the following resources available to it: a socket-mounted 84-pin PLCC package, signal pins that are accessible via female headers, two octal dual inline package (DIP) switches, 16 LEDs, expansion port with 42 input/output pins and the dedicated global CLR, OE1, and OE2/GCLK2 pins. The FLEX10K70 device has the additional resources available to it: a socket for an EPC1 configuration device, one octal DIP switch, VGA video output display port, PS/2 mouse/keyboard port, three expansion ports each with 42 I/O pins and

seven global pins. Altera MAX+PLUS II 10.2 Baseline was the CAD design software chosen for the design platform.

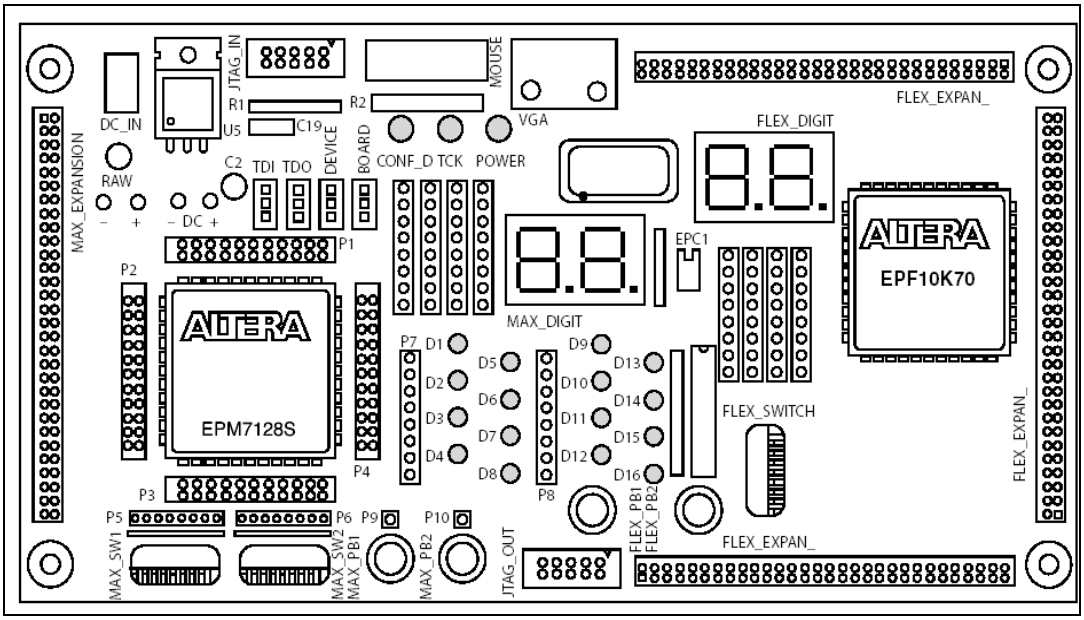

Figure 3.1 Altera UP2 Development Board [10]

3.1 THE MIPS INSTRUCTION SET ARCHITECTURE

As mentioned before MIPS is a RISC microprocessor architecture. The MIPS Architecture defines thirty-two, 32-bit general purpose registers (GPRs). Register $r0 is hard-wired and always contains the value zero. The CPU uses byte addressing for word accesses and must be aligned on a byte boundary divisible by four (0, 4, 8, …). MIPS only has three instruction types: I-type is used for the Load and Stores instructions, R-type is used for Arithmetic instructions, and J-type is used for the Jump instructions as shown in Figure 3.2. Table 3.1 provides a description of each of the fields used in the three different instruction types.

MIPS is a load/store architecture, meaning that all operations are performed on operands held in the processor registers and the main memory can only be accessed through the load and store instructions (e.g lw, sw). A load instruction loads a value from memory into a register. A store instruction stores a value from a register to memory. The load and store instructions use

the sum of the offset value in the address/immediate field and the base register in the $rs field to address the memory. Arithmetic instructions or R-type include: ALU Immediate (e.g. addi), three-operand (e.g. add, and, slt), and shift instructions (e.g. sll, srl). The J-type instructions are used for jump instructions (e.g. j). Branch instructions (e.g. beq, bne) are I-type instructions which use the addition of an offset value from the current address in the address/immediate field along with the program counter (PC) to compute the branch target address; this is considered PC-relative addressing. Table 3.2 shows a summary of the core MIPS instructions.

Field	Description
opcode	6-bit primary operation code
rd	5-bit specifier for the destination register
rs	5-bit specifier for the source register
rt	5-bit specifier for the target (source/destination) register or used to specify functions within the primary *opcode* REGIMM
address/immediate	16-bit signed *immediate* used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement
instr_index	26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address
sa	5-bit shift amount
function	6-bit function field used to specify functions within the primary *opcode* SPECIAL

Table 3.1 MIPS Instruction Fields [11]

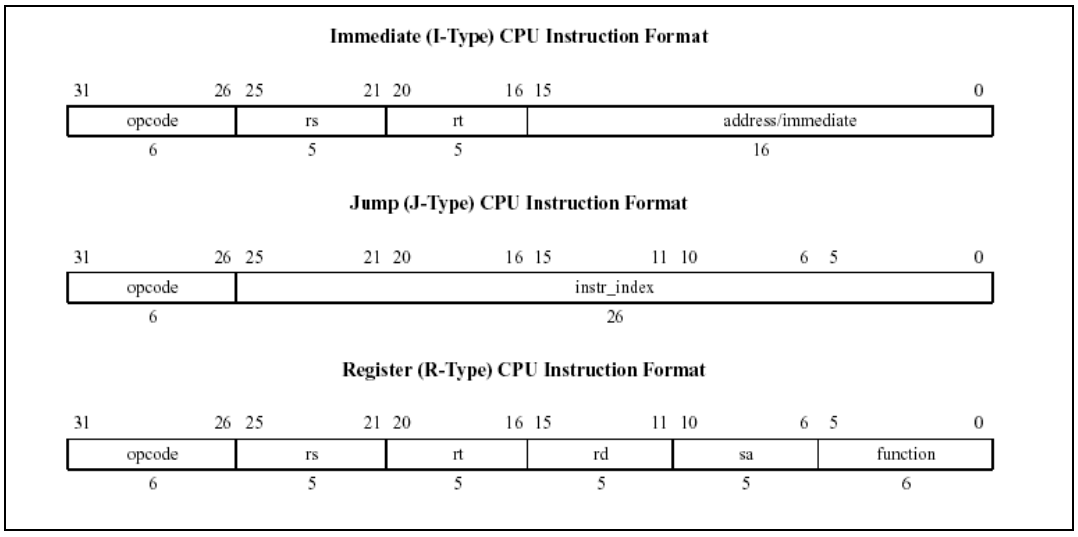

Figure 3.2 MIPS Instruction Types [11]

Instruction	Symbol	Format	Example	Meaning	Comments
Add	add	R	add $r1, $r2, $r3	$r1 = $r2 + $r3	overflow detected
Add Immediate	addi	I	addi $r1, $r2, 100	$r1 = $r2 + 100	plus constant
Add Unsigned	addu	R	addu $r1, $r2, $r3	$r1 = $r2 + $r3	overflow undetected
Subtract	sub	R	sub $r1, $r2, $r3	$r1 = $r2 - $r3	overflow detected
Subtract Unsigned	subu	R	subu $r1, $r2, $r3	$r1 = $r2 - $r3	overflow undetected
And	and	R	and $r1, $r2, $r3	$r1 = $r2 & $r3	bitwise logical and
Or	or	R	Or $r1, $r2, $r3	$r1 = $r2 | $r3	bitwise logical or
Shift Left Logical	sll	R	sll $r1, $r2, 10	$r1 = $r2 << 10	shift left by constant
Shift Right Logical	srl	R	srl $r1, $r2, 10	$r1 = $r2 >> 10	shift right by constant
Set Less Than	slt	R	slt $r1, $r2, $r3	if ($r2 < $r3) $r1 = 1 else 0	compare less than
Load Word	lw	I	lw $r1, 100($r2)	$r1 = mem($r2 + 100)	load word from mem to reg
Store Word	sw	I	sw $r1, 100($r2)	mem($r2 + 100) = $r1	store word from reg to mem
Branch on Equal	beq	I	beq $r1, $r2, 25	if ($r1 = $r2) goto PC + 4 + 100	equal test
Branch on Not Equal	bne	I	bne $r1, $r2, 25	if ($r1 != $r2) goto PC + 4 + 100	not equal test
Jump	j	J	j 100	goto 400	jump to target address

Table 3.2 MIPS Core Instructions

3.2 MIPS SINGLE-CYCLE PROCESSOR

The MIPS single-cycle processor performs the tasks of instruction fetch, instruction decode, execution, memory access and write-back all in one clock cycle. First the PC value is used as an address to index the instruction memory which supplies a 32-bit value of the next instruction to be executed. This instruction is then divided into the different fields shown in Table 3.1. The instructions opcode field bits [31-26] are sent to a control unit to determine the type of instruction to execute. The type of instruction then determines which control signals are

to be asserted and what function the ALU is to perform, thus decoding the instruction. The instruction register address fields $rs bits [25 - 21], $rt bits [20 - 16], and $rd bits[15-11] are used to address the register file. The register file supports two independent register reads and one register write in one clock cycle. The register file reads in the requested addresses and outputs the data values contained in these registers. These data values can then be operated on by the ALU whose operation is determined by the control unit to either compute a memory address (e.g. load or store), compute an arithmetic result (e.g. add, and or slt), or perform a compare (e.g. branch). If the instruction decoded is arithmetic, the ALU result must be written to a register. If the instruction decoded is a load or a store, the ALU result is then used to address the data memory. The final step writes the ALU result or memory value back to the register file.

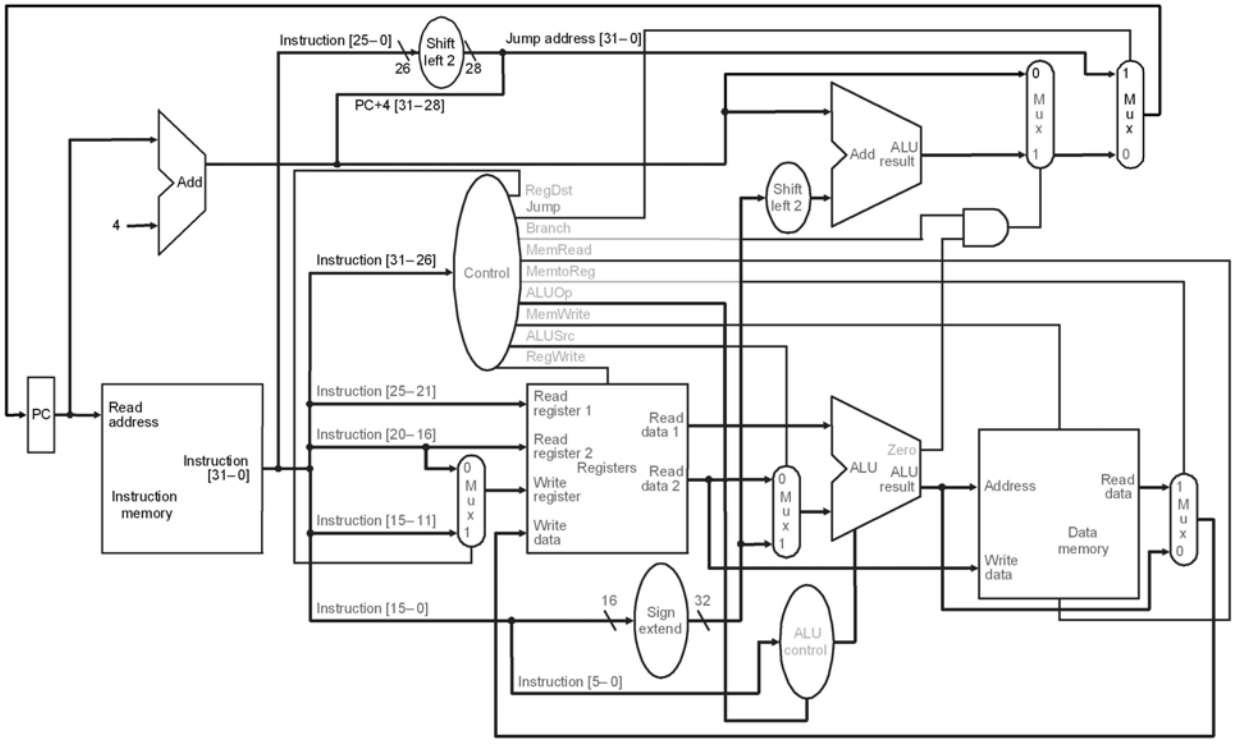

Figure 3.3 MIPS Single-cycle Processor

3.3 MIPS SINGLE-CYCLE PROCESSOR VHDL IMPLEMENTATION

The initial task of this project was to implement in VHDL the MIPS single-cycle processor using Altera MAX+PLUS II Text Editor to model the processor developed in [1]. A good VHDL reference and tutorial can be found in the appendices to the book *Fundamentals of Digital Logic with VHDL Design* by Stephen Brown and Zvonko Vranesic [12]. The *IEEE Standard VHDL Language Reference Manual* [13], also helped in the overall design of the VHDL implementation. The first part of the design was to analyze the single-cycle datapath and take note of the major function units and their respective connections.

The MIPS implementation as with all processors, consists of two main types of logic elements: combinational and sequential elements. Combinational elements are elements that operate on data values, meaning that their outputs depend on the current inputs. Such elements in the MIPS implementation include the arithmetic logic unit (ALU) and adder. Sequential elements are elements that contain and hold a state. Each state element has at least two inputs and one output. The two inputs are the data value to be written and a clock signal. The output signal provides the data values that were written in an earlier clock cycle. State elements in the MIPS implementation include the Register File, Instruction Memory, and Data Memory as seen in Figure 3.3. While many of logic units are straightforward to design and implement in VHDL, considerable effort was needed to implement the state elements.

It was determined that the full 32-bit version of the MIPS architecture would not fit onto the chosen FLEX10K70 FPGA. The FLEX10K70 device includes nine embedded array blocks (EABs) each providing only 2,048 bits of memory for a total of 2 KB memory space. The full 32-bit version of MIPS requires no less than twelve EABs to support the processor's register file, instruction memory, and data memory. In order for our design to model that in [1], the data

width was reduced to 8-bit while still maintaining a full 32-bit instruction. This new design allows us to implement all of the processor's state elements using six EABs, which can be handled by the FLEX10K70 FPGA device. Even though the data width was reduced, the design has minimal VHDL source modifications from the full 32-bit version, thus not impacting the instructional value of the MIPS VHDL model.

With our new design, the register file is implemented to hold thirty-two, 8-bit general purpose registers amounting to 32 bytes of memory space. This easily fits into one 256 x 8 EAB within the FPGA. The full 32-bit version of MIPS will require combining four 256 x 8 EABs to implement the register file. The register file has two read and one write input ports, meaning that during one clock cycle, the processor must be able to read two independent data values and write a separate value into the register file. Figure 3.4 shows the MIPS register file. The register file was implemented in VHDL by declaring it as a one-dimensional array of 32 elements or registers each 8-bits wide. (e.g. TYPE register_file IS ARRAY (0 TO 31) OF STD_LOGIC_VECTOR (7 DOWNTO 0)) By declaring the register file as a one-dimensional array, the requested register address would need to be converted into an integer to index the register file.(e.g. Read_Data_1 <= register_file (CONV_INTEGER (read_register_address1 (4 DOWNTO 0)))) Finally, to save from having to load each register with a value, the registers get initialized to their respective register number when the Reset signal is asserted. (e.g. $r1 = 1, $r2 = 2, etc.)

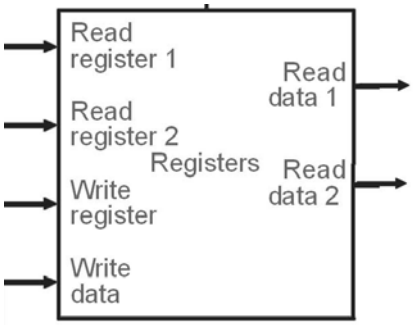

Figure 3.4 MIPS Register File

Altera MAX+PLUS II is packaged with a Library of Parameterized Modules (LPM) that allow one to implement RAM and ROM memory in Altera supported PLD devices. With our design this library was used to declare the instruction memory as a read only memory (ROM) and the data memory as a random access memory (RAM). Using the lpm_rom component from the LPM Library, the Instruction memory is declared as a ROM and the following parameters are set: the width of the output data port parameter lpm_width is set to 32-bits, the width of the address port parameter lpm_widthad is set to 8-bits, and the parameter lpm_file is used to declare a memory initialization file (.mif) that contains ROM initialization data. This allows us to set the indexed address data width to 8-bits, the instruction output to 32-bits wide, and enables us to initialize the ROM with the desired MIPS program to test the MIPS processor implementation. With these settings, four 256 x 8 EABs are required to implement the instruction memory. An example of the MIPS instruction memory can be seen in Figure 3.5 and the VHDL code implementation can be seen in Figure 3.6.

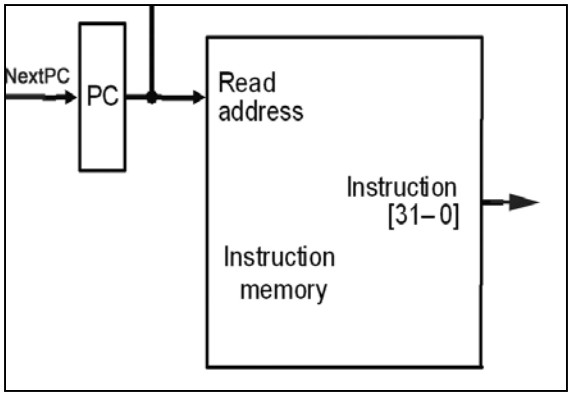

Figure 3.5 MIPS Instruction Memory

```
Instr Memory: LPM ROM

GENERIC MAP(
LPM_WIDTH              => 32,
LPM WIDTHAD            => 8,
LPM FILE               => "instruction memory.mif",
LPM OUTDATA            => "UNREGISTERED",
LPM ADDRESS CONTROL    => "UNREGISTERED")

PORT MAP (
        address        => PC,
        q              => Instruction );
```

Figure 3.6 VHDL – MIPS Instruction Memory

The data memory is declared using the lpm_ram_dq component of the LPM library. This component is chosen because it requires that the memory address to stabilize before allowing the write enable to be asserted high. The input Address width (lpm_widthad) and the Read Data

output width (lpm_width) are both declared as 8-bit wide, in lieu of our altered design. Using these settings allows us to use one 256 x 8 EAB instead of the 4 combined EABs required for the full 32-bit version of MIPS. An example of the MIPS data memory can be seen in Figure 3.7 and the VHDL code implementation can be seen in Figure 3.8.

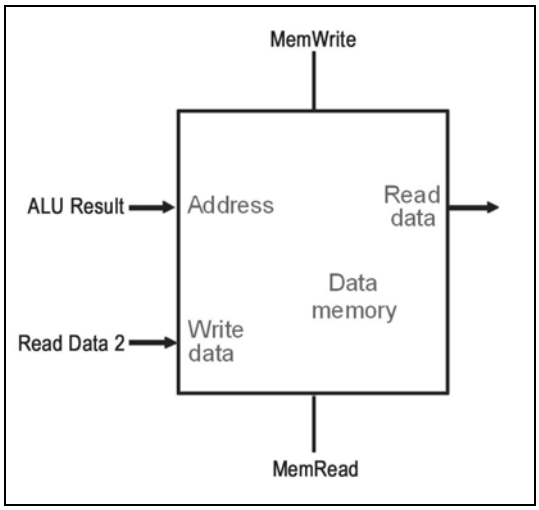

Figure 3.7 MIPS Data Memory

```
Data_Memory : LPM_RAM_DQ

       GENERIC MAP(
              LPM_WIDTH           => 8,
              LPM_WIDTHAD         => 8,
              LPM_FILE            => "data_memory.mif",
              LPM_INDATA          => "REGISTERED",
              LPM_ADDRESS_CONTROL => "UNREGISTERED",
              LPM_OUTDATA         => "UNREGISTERED")

       PORT MAP(
              inclock      => Clock,
              data         => Write_Data,
              address      => Address,
              we           => LPM_WRITE,
              q            => Read_Data);
```
Figure 3.8. VHDL – MIPS Data Memory

Once we determined how to declare the state elements of the MIPS processor it was time to implement the rest of the logic devices in VHDL. Because the final task is to pipeline the single-cycle implementation of the MIPS processor, we decided to modularize the single-cycle

implementation into the five different VHDL modules to be fully utilized later in the pipelined implementation of the MIPS processor. The five modules are: Instruction Fetch, Instruction Decode, Control Unit, Execution, and Data Memory as shown in Figure 3.9.

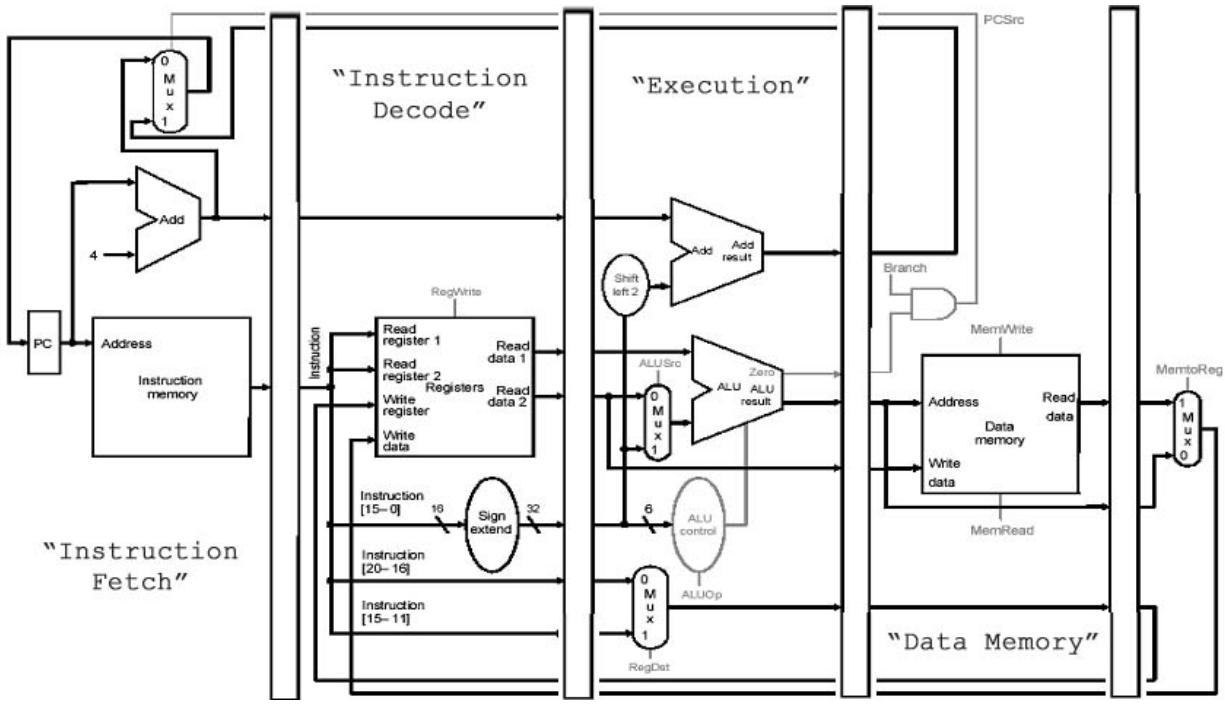

Figure 3.9 Modularized MIPS Single Cycle excluding Control Unit and signals.

With the decision to use five different modules to implement the single-cycle MIPS processor, the VHDL design becomes a two-level hierarchy. The top-level of the hierarchy is a structural VHDL file that connects the all five components of the single-cycle implementation, while the bottom-level contains the behavioral VHDL models of the five different components. Appendix C contains the top-level structural VHDL code for the MIPS single-cycle processor.

3.3.1 INSTRUCTION FETCH UNIT

The function of the instruction fetch unit is to obtain an instruction from the instruction memory using the current value of the PC and increment the PC value for the next instruction as

shown in Figure 3.10. Since this design uses an 8-bit data width we had to implement byte addressing to access the registers and word address to access the instruction memory. The instruction fetch component contains the following logic elements that are implemented in VHDL: 8-bit program counter (PC) register, an adder to increment the PC by four, the instruction memory, a multiplexor, and an AND gate used to select the value of the next PC. Appendix C contains the VHDL code used to create the instruction fetch unit of the MIPS single-cycle processor.

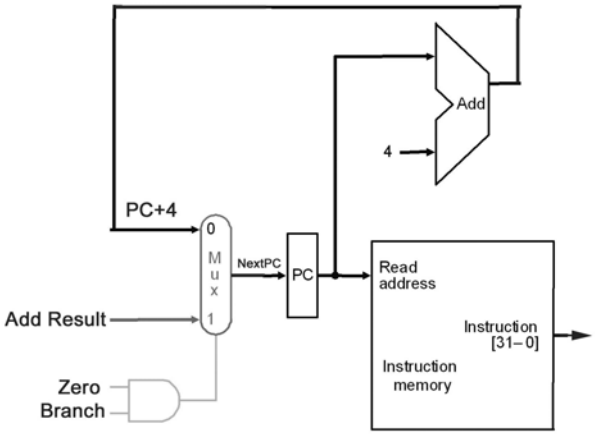

Figure 3.10 MIPS Instruction Fetch Unit

3.3.2 INSTRUCTION DECODE UNIT

The main function of the instruction decode unit is to use the 32-bit instruction provided from the previous instruction fetch unit to index the register file and obtain the register data values as seen in Figure 3.11. This unit also sign extends instruction bits [15 - 0] to 32-bit. However with our design of 8-bit data width, our implementation uses the instruction bits [7 – 0] bits instead of sign extending the value. The logic elements to be implemented in VHDL include several multiplexors and the register file, that was described earlier. Appendix C contains the VHDL code used to create the instruction decode unit of the MIPS single-cycle processor.

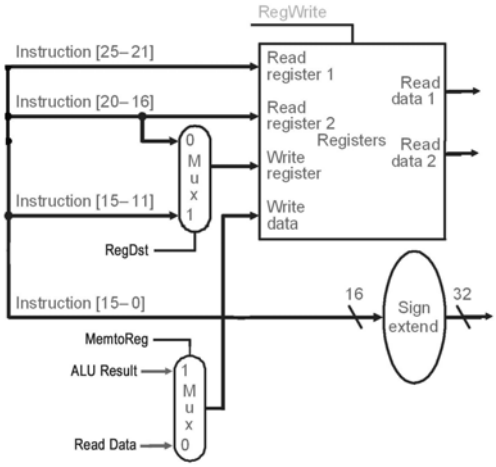

Figure 3.11 MIPS Instruction Decode Unit

3.3.3 THE CONTROL UNIT

The control unit of the MIPS single-cycle processor examines the instruction opcode bits [31 – 26] and decodes the instruction to generate nine control signals to be used in the additional modules as shown in Figure 3.12. The RegDst control signal determines which register is written to the register file. The Jump control signal selects the jump address to be sent to the PC. The Branch control signal is used to select the branch address to be sent to the PC. The MemRead control signal is asserted during a load instruction when the data memory is read to load a register with its memory contents. The MemtoReg control signal determines if the ALU result or the data memory output is written to the register file. The ALUOp control signals determine the function the ALU performs. (e.g. and, or, add, sbu, slt) The MemWrite control signal is asserted when during a store instruction when a registers value is stored in the data memory. The ALUSrc control signal determines if the ALU second operand comes from the register file or the sign extend. The RegWrite control signal is asserted when the register file

needs to be written. Table 3.3 shows the control signal values from the instruction decoded. Appendix C contains the VHDL code for the MIPS single-cycle control unit.

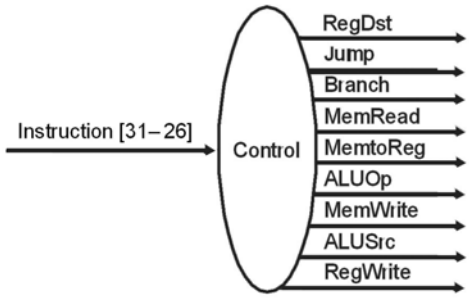

Figure 3.12 MIPS Control Unit

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Table 3.3 MIPS Control Signals

3.3.4 EXECUTION UNIT

The execution unit of the MIPS processor contains the arithmetic logic unit (ALU) which performs the operation determined by the ALUop signal. The branch address is calculated by adding the PC+4 to the sign extended immediate field shifted left 2 bits by a separate adder. The logic elements to be implemented in VHDL include a multiplexor, an adder, the ALU and the ALU control as shown in Figure 3.9 Appendix C contains the VHDL code used to create the execution unit of the MIPS single-cycle processor.

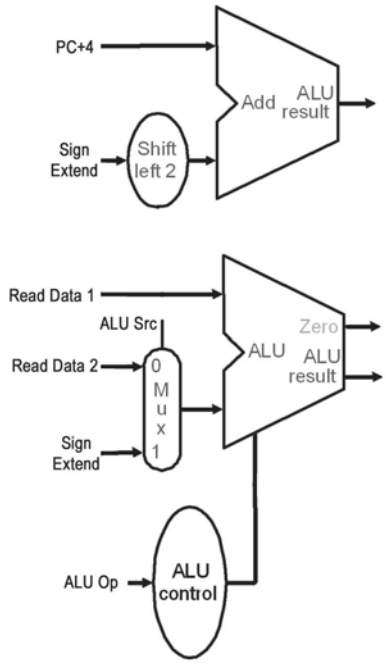

FIGURE 3.13 MIPS EXECUTION UNIT

3.3.5 DATA MEMORY UNIT

The data memory unit is only accessed by the load and store instructions. The load instruction asserts the MemRead signal and uses the ALU Result value as an address to index the data memory. The read output data is then subsequently written into the register file. A store instruction asserts the MemWrite signal and writes the data value previously read from a register into the computed memory address. The VHDL implementation of the data memory was described earlier. Figure 3.14 shows the signals used by the memory unit to access the data memory. Appendix C contains the complete VHDL code used to create the memory state of the MIPS single-cycle processor. Appendix D shows an example of MIPS single-cycle being simulated using Altera MAX+PLUS II waveform editor.

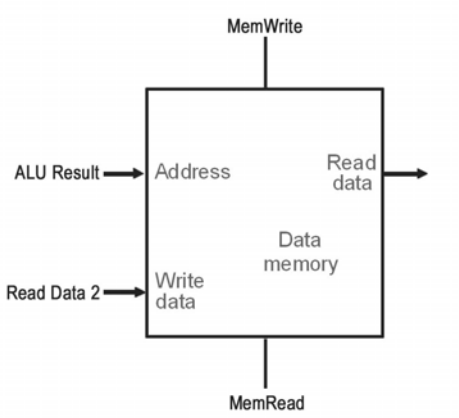

Figure 3.14 MIPS Data Memory Unit

3.4 MIPS PIPELINED PROCESSOR VHDL IMPLEMENTATION

Once the MIPS single-cycle VHDL implementation was completed, our next task was to pipeline the MIPS processor. Pipelining, a standard feature in RISC processors, is a technique used to improve both clock speed and overall performance. Pipelining allows a processor to work on different steps of the instruction at the same time, thus more instruction can be executed in a shorter period of time. For example in the VHDL MIPS single-cycle implementation above, the datapath is divided into different modules, where each module must wait for the previous one to finish before it can execute, thereby completing one instruction in one long clock cycle. When the MIPS processor is pipelined, during a single clock cycle each one of those modules or stages is in use at exactly the same time executing on different instructions in parallel. Figure 3.15 shows an example of a MIPS single-cycle non-pipelined (a.) versus a MIPS pipelined implementation (b.). The pipelined implementation executes faster, keep in mind that both implementations use the same hardware components.

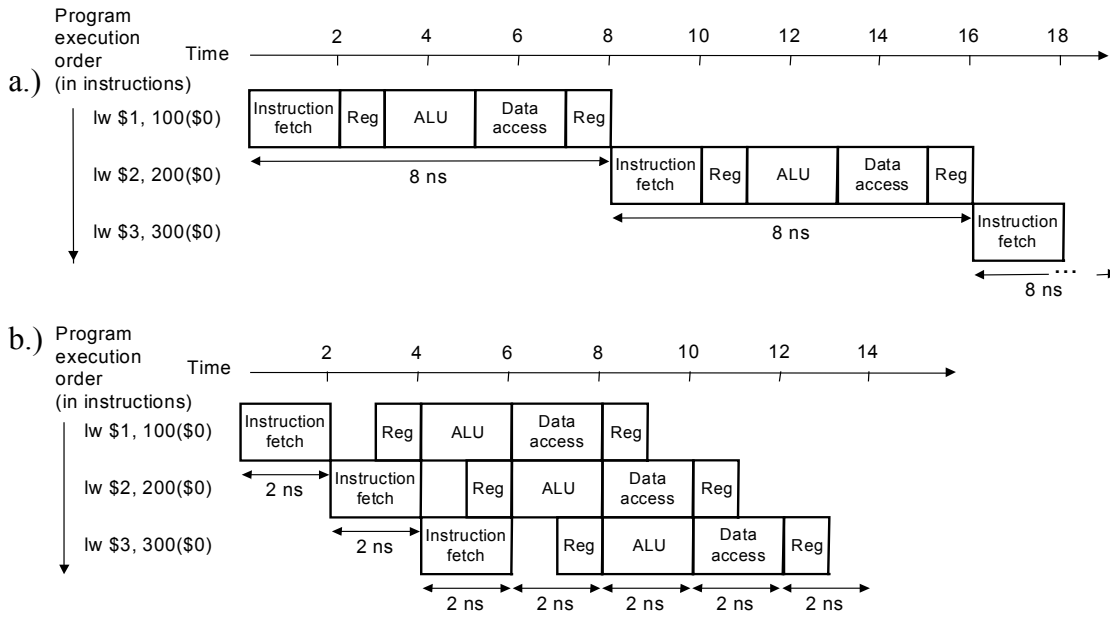

Figure 3.15. Single-cycle non-pipelined (a) vs. pipelined execution (b)

The MIPS pipelined processor involves five steps, the division of an instruction into five stages implies a five-stage pipeline:

1. Instruction Fetch (IF): fetching the instruction from the memory
2. Instruction Decode (ID): reading the registers and decoding the instruction
3. Execution (EX): executing an operation or calculating an address
4. Data Memory (MEM): accessing the data memory
5. Write Back (WB): writing the result into a register.

The key to pipelining the single-cycle implementation of the MIPS processor is the introduction of pipeline registers that are used to separate the datapath into the five sections IF, ID, EX, MEM and WB. Pipeline registers are used to store the values used by an instruction as it proceeds through the subsequent stages. The MIPS pipelined registers are labeled according to the stages they separate. (e.g. IF/ID, ID/EX, EX/MEM, MEM/WB) Figure 3.16 shows and example of a pipelined datapath excluding the control unit and control signal lines.

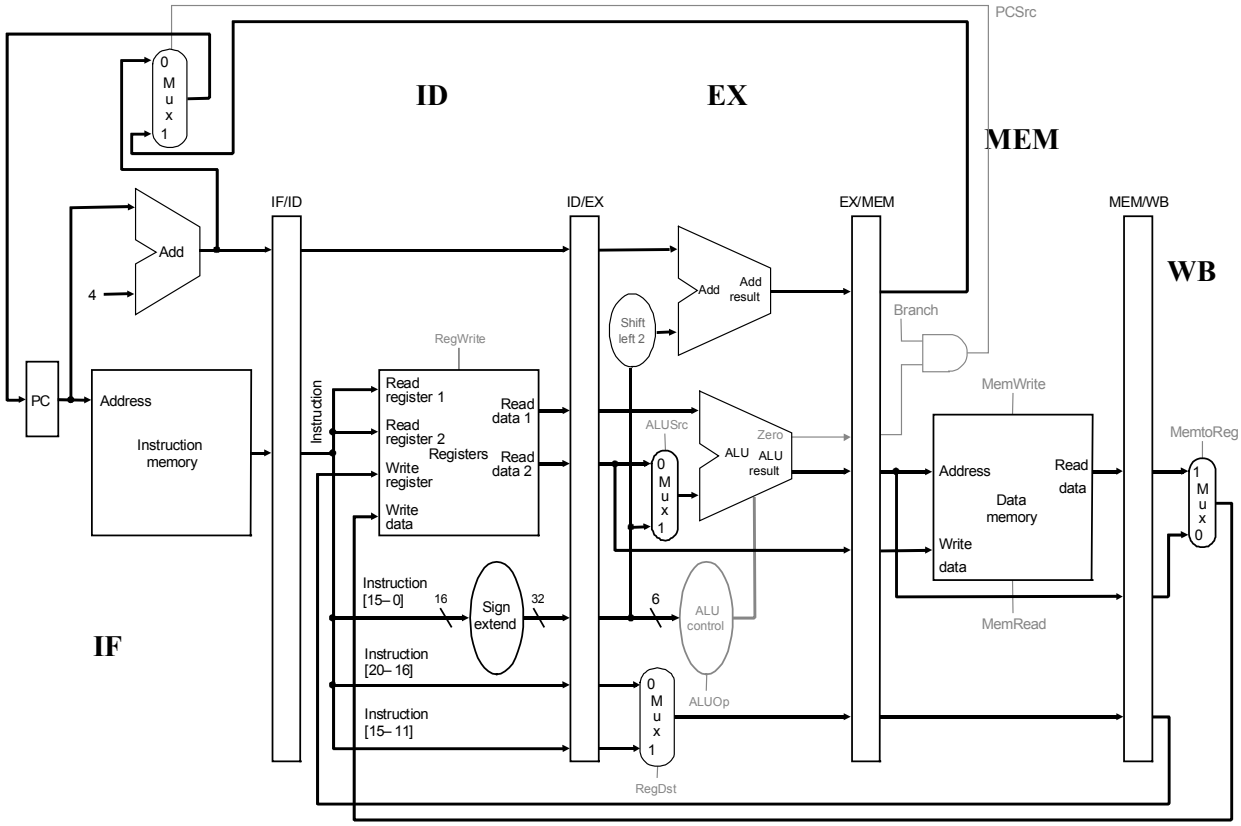

Figure 3.16 MIPS Pipelined Processor Datapath [1]

To implement the MIPS pipelined processor, pipeline registers are placed into the corresponding VHDL modules that generate the input to the particular pipeline register. For example, the Instruction Fetch component will generate the 32-bit instruction and the PC+4 value and store them into the IF/ID pipeline register. When that instruction moves to the Instruction Decode stages it extracts those saved values from the IF/ID pipeline register. Appendix F contains the complete VHDL code used to implement the MIPS pipelined processor data path. Appendices G shows an example of MIPS processor pipelined being simulated.

3.4.1 PIPELINE HAZARDS

Pipelining the MIPS processor introduces events called hazards, which prevent the next instruction in the instruction stream from being executing during its designated clock cycle. The

types of possible hazards include structural, data and control hazards. Structural hazards arise when the hardware cannot support the instructions that are ready to execute during the same clock cycle. Fortunately, MIPS is designed to be pipelined thus no structural hazards exist. However, if the MIPS processor had been designed with one memory to be shared between both instructions and data, then a structural hazard would occur.

Data hazards arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of the instructions in the pipeline, thus causing the pipeline to stall until the results are made available. One solution to this type of data hazard is called forwarding, which supplies the resulting operand to the dependant instruction as soon it has been computed. Figures 3.17 shows an example of pipelined data dependencies and Figure 3.18 shows how these dependencies are resolved using a forwarding unit. Appendix I MIPS Pipelined – Data Hazards and Forwarding Simulation, simulates this exact example.

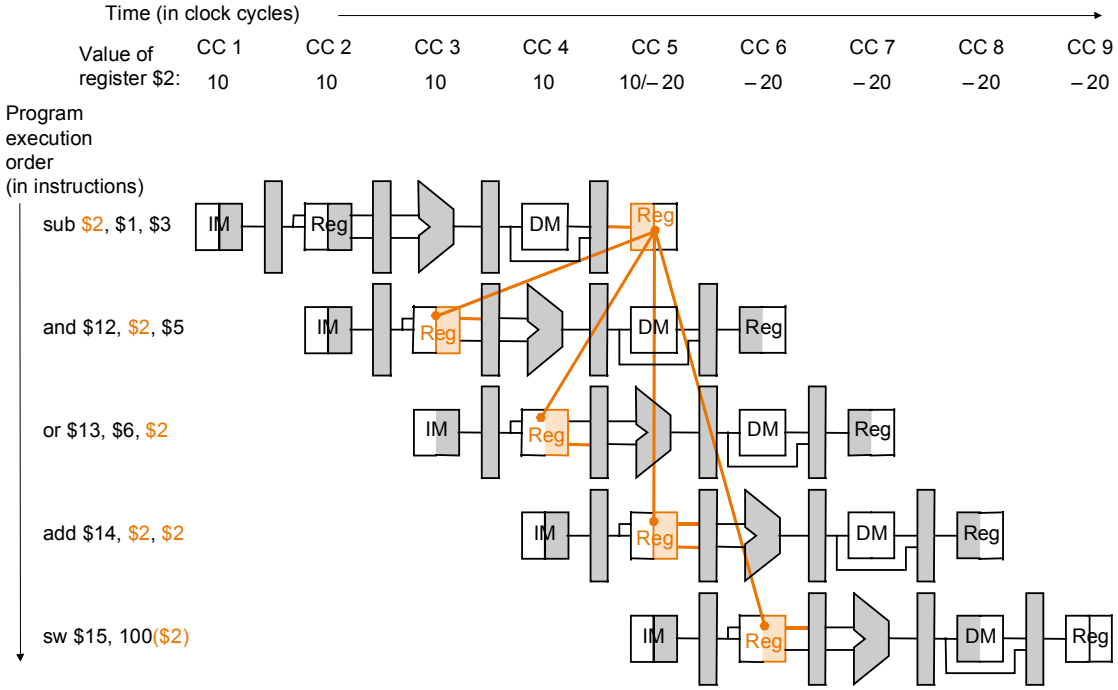

Figure 3.17 Pipelined Data Dependencies [1]

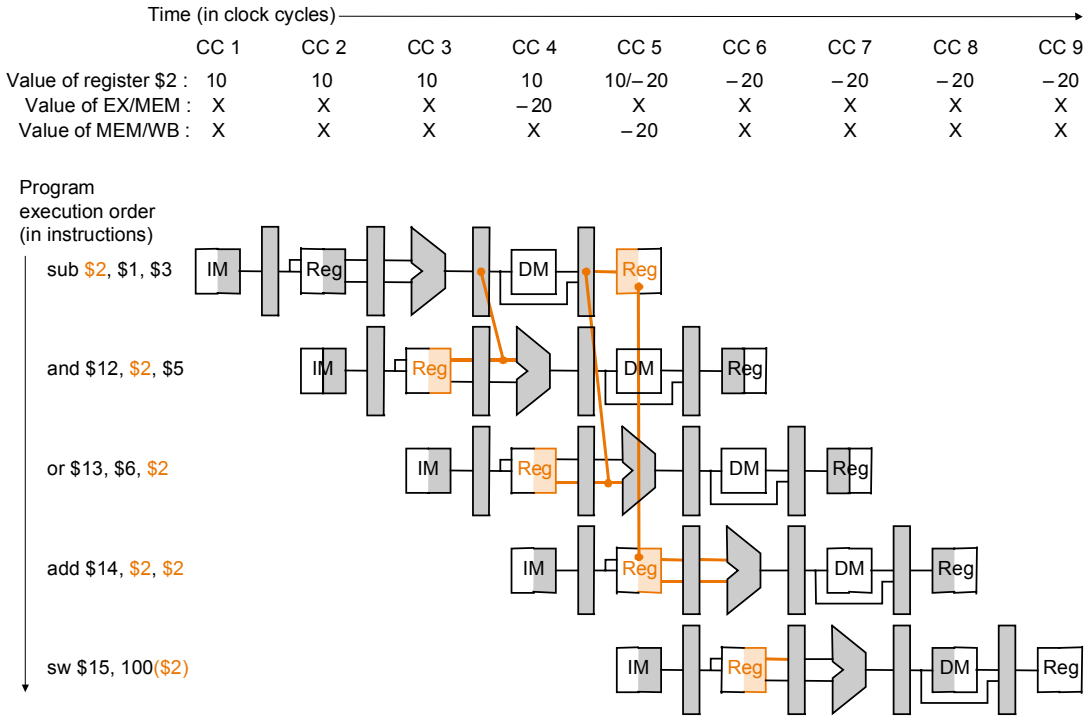

Figure 3.18 Pipelined Data Dependencies Resolved with Forwarding [1]

While forwarding is an exceptional solution to data hazards it does not resolve all of them. One instance is when an instruction attempts to read a register value that is going to be supplied by a previous load instruction that writes the same register, called a load-use hazard. At the same time the load instruction is reading data from memory, the subsequent instruction executing in the execution stage with the wrong data value. The only solution here is to stall the pipeline and wait for the correct data value being used as an operand. In order to detect such hazards, MIPS introduces a Hazard Detection Unit during the instruction decode stage so that it can stall the pipeline between a load instruction and the immediate instruction attempting to use the same register. Figure 3.19 show an example where the forwarding using can not resolve a data dependence, the solution is to this type of data hazard it to insert a stall shown in Figure 3.20 as a bubble. Appendix K MIPS Pipelined – Data Hazards and Stalls Simulation, shows this exact example.

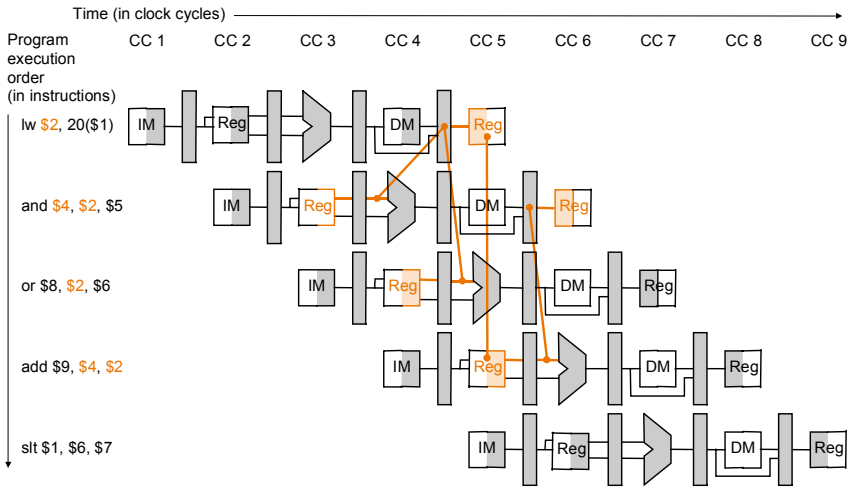

Figure 3.19 Pipelined Data Dependencies Requiring Stall [1]

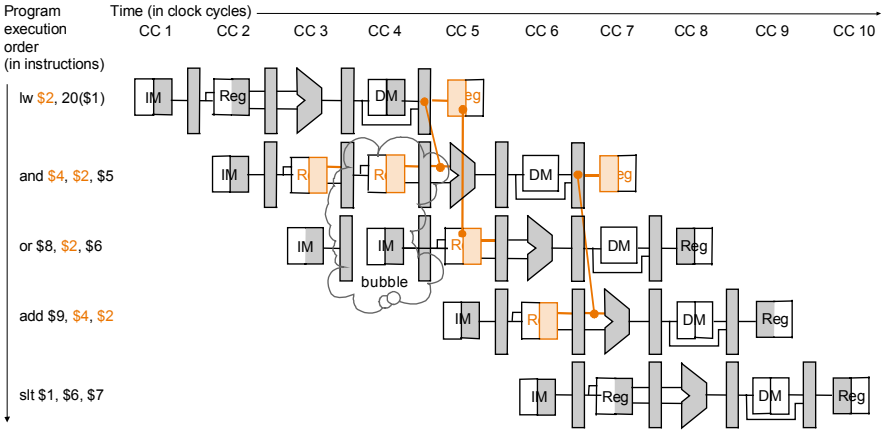

Figure 3.20 Pipelined Data Dependencies Resolved with Stall [1]

The last type of hazard is a control hazard also known as a branch hazard. These hazards occur when there is a need to make a decision based on the results of one instruction while other instructions continue executing. For example, a branch on equal instruction will branch to a non-sequential part of the instruction memory if the two register values compared are equal. While the register values are compared, other instructions continue to be fetched and decoded. If the branch is taken, the wrong instructions are fetched into the pipeline and must somehow be discarded. Figure 3.21 shows three instructions that need to be discarded after it is determined the branch instruction will be taken. A common solution to these hazards is to continue instruction execution as if the branch is not taken. If it is later determined that the branch is

taken, the instructions that were fetched and decoded must be discarded which can be achieved by flushing some of the pipeline registers. Flushing means that all values stored in the pipeline registers are discarded or reset. However in order to reduce the branch hazard to 1 clock cycle, the branch decision is moved from the memory pipeline stage to the instruction decode stage. By simply comparing the registers fetch it can be determined if a branch is to be taken or not. Appendix M MIPS Pipelined – Branch Hazard Simulation, shows an example of a branch instruction being taken, flushing the IF/ID pipeline register, and loading the new instruction determined from the branch address.

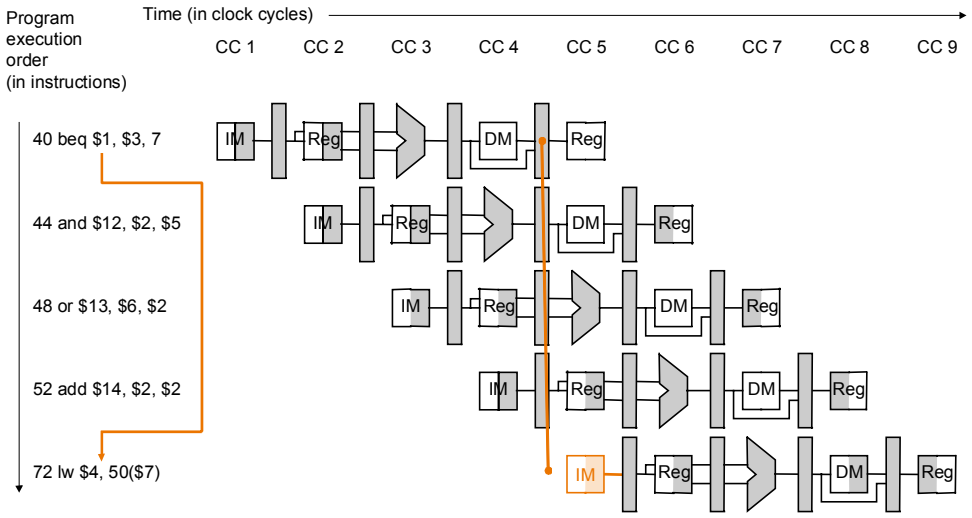

Figure 3.21 Pipelined Branch Instruction [1]

Appendix F MIPS Pipelined – VHDL Code, contains the complete VHDL code used to implement the MIPS pipelined processor including the solutions to resolve data and branch hazards. Appendices I, K, and M show example of MIPS pipelined being simulated. Appendix O MIPS Pipelined Final Dathpath and Control, shows the complete datapath and control that was implemented in VHDL code.

3.5 HARDWARE IMPLEMENTATION

Once the VHDL code was simulated and all operations were verified using Altera MAX+PLUS II Waveform Editor and Simulator, the design would then need to be prepared for the hardware implementation on to the Altera UP2 Development board. Preparing the VHDL design involves assigning VHDL code signals to device pins found on the UP2 board. For example the reset signal was assigned to an on-board push button (PB) switch allowing us to manually reset the processor. Table 3.3 shows the UP2 Board FLEX10K70 I/O pin assignments.

Pin Name	Pin	Pin Type	Function
Clock	91	Input	25.175 MHz System Clock
PB1	28	Input	Push-button 1
PB2	29	Input	Push-button 2
FLEX_switch_1	41	Input	FLEX DIP Switch 1
FLEX_switch_2	40	Input	FLEX DIP Switch 2
FLEX_switch_3	39	Input	FLEX DIP Switch 3
FLEX_switch_4	38	Input	FLEX DIP Switch 4
FLEX_switch_5	36	Input	FLEX DIP Switch 5
FLEX_switch_6	35	Input	FLEX DIP Switch 6
FLEX_switch_7	34	Input	FLEX DIP Switch 7
FLEX_switch_8	33	Input	FLEX DIP Switch 8
MSD_dp	14	Output	Most Significant Digit of Seven Segment Display Decimal Point Segment
MSD_a	6	Output	MSD Segment a
MSD_b	7	Output	MSD Segment b
MSD_c	8	Output	MSD Segment c
MSD_d	9	Output	MSD Segment d
MSD_e	11	Output	MSD Segment e
MSD_f	12	Output	MSD Segment f
MSD_g	13	Output	MSD Segment g
LSD_dp	25	Output	Least Significant Digit of Seven Segment Display Decimal Point Segment
LSD_a	17	Output	LSD Segment a
LSD_b	18	Output	LSD Segment b
LSD_c	19	Output	LSD Segment c
LSD_d	20	Output	LSD Segment d
LSD_e	21	Output	LSD Segment e
LSD_f	23	Output	LSD Segment f
LSD_g	24	Output	LSD Segment g
Blue	238	Output	VGA Video Signal – Blue Video Data
Green	237	Output	VGA Video Signal – Green Video Data
Red	236	Output	VGA Video Signal – Red Video Data
Horizontal_sync	240	Output	VGA Video Signal – Horizontal Synchronization
Vertical_sync	239	Output	VGA Video Signal – Vertical Synchronization

Table 3.4 FLEX10K70 Device I/O Pin Assignments

Programming or downloading the design to a UP2 board requires setting on-board jumpers that indicate which PLD device to program. The jumpers indicate if you want to program only the EMP7128S device, program only the FLEX10K70 device, program both devices or connect multiple UP2 boards together in a chain. The Altera *University Program UP2 Development Kit User Guide* [10], explains how to setup the jumpers to program the desired device. Once the jumpers are set the MAX+PLUS II software must be setup to configure the devices via a JTAG chain. The JTAG uses boundary-scan technology that allows one to perform downloading, debugging and diagnostics on a system through a small number of dedicated test pins. Once the software is properly setup, the design can then easily be downloaded using the ByteBlaster II download cable, a cable that provides a hardware interface to the UP2 via JTAG connector and a connection to the computer running MAX+PLUS II via a standard parallel port. For complete instructions on setting up the MAX+PLUS II software to program the devices via the ByteBlaster II download cable please see the *ByteBlaster MV Parallel Port Download Cable Data Sheet* by Altera [16]. By choosing to program and configure the FLEX10K70 FPGA device our design of the MIPS single-cycle processor easily fit onto this device. The VGA video output display on the UP2 board was used to display some hexadecimal output values of the major functional processor units and the two seven-segment display were configured to display the hexadecimal output values of the current program counter (PC). Finally one on-board push button switch was configured to used as the global reset signal and the other push button switch was configured to simulate a clock cycle when depressed. Both VHDL implementations of the MIPS single-cycle and pipelined processors were downloaded and used to configure the Altera FLEX10K70 device with plenty of room and resources to spare. Table 3.4 shows the number of resources utilized by each VHDL design.

Processor	Pins	Logic Cells	Memory Bits	EABs
Single-cycle	19/189	1238/3744	9472/18432	5/9
Pipelined	19/189	1423/3744	11240/18432	6/9

Table 3.5 FLEX10K70 Resource Utilization Table

4 RESULTS AND DISCUSSION

The work presented in this paper describes a functional FPGA implementation design of a MIPS single-cycle and pipelined processor designed using VHDL. The project was to model Chapters 5 and 6 from the widely used book *Computer Organization and Design – The Hardware/Software Interface* by David A. Patterson and John L. Hennessy, to help the computer architecture students gain a better understanding of the MIPS processor. The VHDL designs of the MIPS processor were all simulated to ensure that the processors were functional and operated just as described by Patterson and Hennessy. The Appendices D, G, I, K, and M presented in this paper show some specific examples presented by Patterson and Hennessy being simulated by the VHDL designs. The appendices first show the instruction memory initialization file, which is used to fill the instruction memory with the instructions to be executed as seen in Figure 4.1. The first column of numbers is the hexadecimal memory address of the instructions, which are indexed by the program counter (PC). The second column of characters is the actual 32-bit instruction represented using hexadecimal numbers. The third column of numbers is the PC value used to index the instruction memory to retrieve an instruction. The next four columns are the MIPS instruction's mnemonic description. Finally last columns are the pseudo instructions using the actual values used during the simulation.

```
--Initialized Instruction Memory
               --PC   Instruction
00: 8C2A0014;  --00:  LW  $10, 20 ($1)      $10 (0x0A) = MEM(0x01+0x14) = MEM(0x15) = 0x15
01: 00435822;  --04:  SUB $11, $2, $3       $11 (0x0B) = 0x02   - 0x03  = 0d-1 = 0XFF
02: 00856024;  --08:  AND $12, $4, $5       $12 (0x0C) = 0x04 AND 0x05  = 0d 4 = 0x04
03: 00C76825;  --0C:  OR  $13, $6, $7       $13 (0x0D) = 0x06  OR 0x07  = 0d 7 = 0x07
04: 01097020;  --10:  ADD $14, $8, $9       $14 (0x0E) = 0x08   + 0x09  = 0d17 = 0x11
```

Figure 4.1. Instruction Memory Initialization File

Figure 4.2 shows a screenshot of the Altera MAX+PLUS II Waveform Editor and Simulator results for the instructions shown in Figure 4.1. The first two rows depict the global clock and reset signals. The following rows are executed during the Instruction Fetch stage of the MIPS pipelined processor. The signals are the PC value, used to index the instruction memory and the 32-bit instruction that was index out of the instruction memory. Please note that these values correspond to those shown in Figure 4.1. The next group of signals are executed during the Instruction Decode stage of the pipelined processor. These values depict the register values indexed from the register file and various other signals used during hazard detection. The third group of signals are executed during the Execution stage of the pipelined processor. These signals show the two values fed into the ALU and the corresponding result. The following group of signals depict the memory stage of the pipelined processor. These signals are only used for the load and store instructions to access the data memory. In this example we can see that the memory address 0x15 was calculate during the execution unit is used to index or read the data memory and retrieve the value 0x15. In this specific case the data memory for address 0x15 was initialized with the same value 0x15. Finally the last group of signals are executed during the write back stage of the pipelined processor. The RegWrite_out signal tells us when the register file is going to be written. The WriteRegister__out signal is the actual register to be written and the RegWriteData_out is the value to be written.

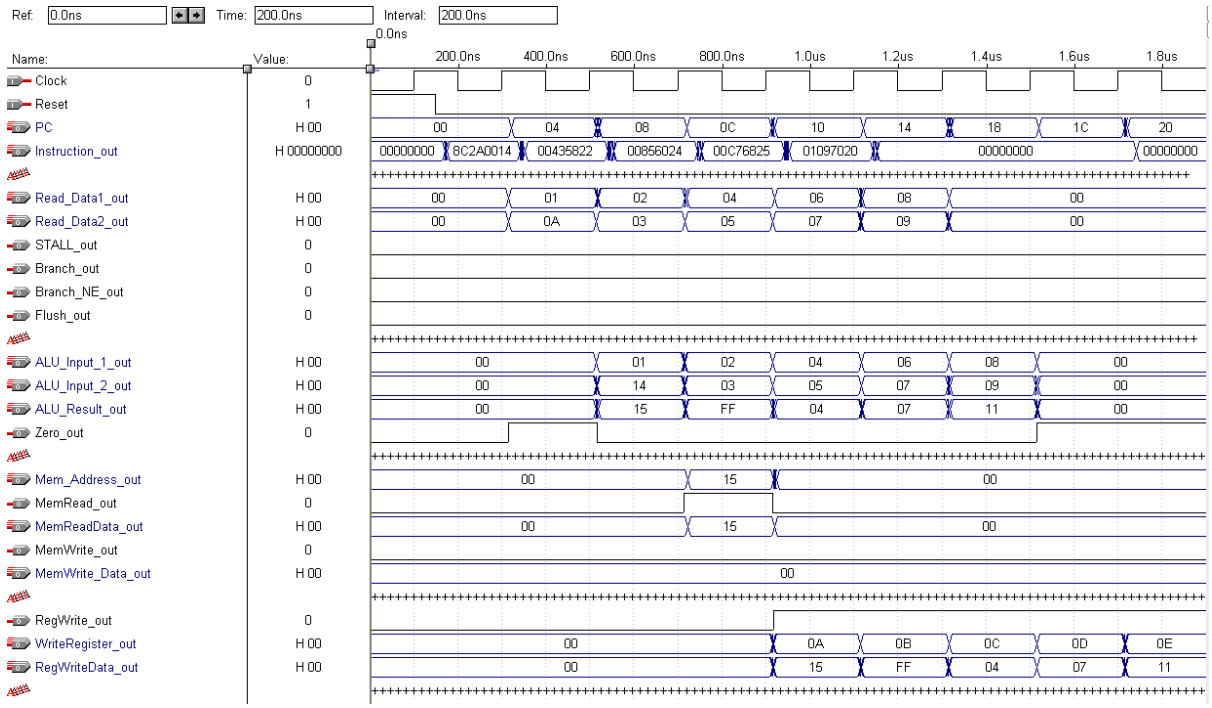

Figure 4.2 Altera MAX+PLUS II Waveform Editor and Simulator Screenshot

Every simulation shown in the appendices come accompanied by results obtained from the SPIM Simulator which was used to validate the results obtained by the Altera MAX+PLUS II Simulator. SPIM is a software simulator that runs programs written for the MIPS processor, it can read and write MIPS assembly language files to simulate programs. An example of the SPIM Simulator registers and console are shown in Figure 4.3. Part of SPIMs output is the value of the thirty two general purpose registers available in the MIPS instruction set architecture shown as R0 – R31. MIPS architecture developed a convention and suggested guidelines on how these register showed be used, those are the values seen in the parenthesis. For example registers $a0 - $a3 are used to pass arguments to routines, $t0 - $t9 are temporary registers and $s0 - $s7 hold long-lived values. The second part of the output shown is the Console. The console is SPIMs that displays all characters the program writes as output. In this example I've programmed the instructions shown in Figure 4.1 and outputted the SPIM results on the console.

Using the MIPS register convention I have used the registers $s0 - $s7 to save all results. The results shown in the console can be used to validate the results obtained from the Altera MAX+PLUS II Simulator.

```
Registers
=========================
 PC     = 00000000     EPC    = 00000000     Cause  = 00000000  BadVAddr= 00000000
 Status = 00000000     HI     = 00000000     LO     = 00000000
                                 General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000015  R24 (t8) = 00000008
R1  (at) = 10010000  R9  (t1) = 00000000  R17 (s1) = ffffffff  R25 (t9) = 00000009
R2  (v0) = 0000000a  R10 (t2) = 00000002  R18 (s2) = 00000004  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000003  R19 (s3) = 00000007  R27 (k1) = 00000000
R4  (a0) = 10010093  R12 (t4) = 00000004  R20 (s4) = 00000011  R28 (gp) = 10008000
R5  (a1) = 00000000  R13 (t5) = 00000005  R21 (s5) = 00000000  R29 (sp) = 7fffe850
R6  (a2) = 7fffe858  R14 (t6) = 00000006  R22 (s6) = 00000000  R30 (s8) = 00000000
R7  (a3) = 00000000  R15 (t7) = 00000007  R23 (s7) = 00000000  R31 (ra) = 00000000


Console
=========================
00: $s0 =      0x15     = 0d21 = 0x00000015
01: $s1 = 0x02 SUB 0x03 = 0d-1 = 0xffffffff
02: $s2 = 0x04 AND 0x05 = 0d4  = 0x00000004
03: $s3 = 0x06 OR  0x07 = 0d7  = 0x00000007
04: $s4 = 0x08 ADD 0x09 = 0d17 = 0x00000011
```

Figure 4.3 SPIM Simulator Screenshot

The work presented illustrates the results achieved for the VHDL implementation of the MIPS single-cycle and pipelined processors. The hardest part to this design was learning to program with the hardware description language (HDL) VHDL. It took time getting used to its calling conventions and researching more efficient methods of implementing logic elements such as the instruction and data memory. Other problems I encountered involved getting the MIPS processor implemented on hardware. For example in order to display the current value of the program counter some design changes had to be made due to the on-board 25 MHz being to fast for this design project. My solution was to use an on-board pushbutton to emulate the clock single, thus every time the pushbutton was pressed one instruction was executed. Another solution was to include a frequency divider implemented in VHDL to reduce the clock speed down to 1 Hz, allowing us to see the PC values progress through the test program.

I remember the days when I was being taught about the MIPS single-cycle and pipelined processor and remembering how lost and confused I was with the material presented. The work presented in this project helped me gain a full and complete understanding of how the MIPS processor operates, wishing I had made available to me when I was being taught this material. This work helped me gain that complete hands-on experience of implementing a software design into actual hardware. No words can express the feelings and sense of pride felt when seeing my FPGA implementations execute through the instruction stream for the first time. A feeling not achievable through simulator based approaches or out of a textbook.

5 CLASSROOM INTEGRATION

A possible set of laboratory projects will now be suggested to enhance the hardware-software integration experience in the computer architecture courses. The first lab project I would suggest would be an: Introduction to SPIM, a MIPS Simulator. The purpose of this lab is to familiarize the students with SPIM, the MIPS instruction set, and microprocessor programming. This project would be advisable to break into 2/3 parts and/or assign homework assignments allowing the students to simulate MIPS assembly programs to get well acquainted with the calling conventions and debugging techniques to trace problems and locate errors within the programs. The paper *SPIM S20: A MIPS R2000 Simulator* [17] is a good reference manual to use for the SPIM simulator. An updated reprinted version of this article can also be found in Appendix A in the book *Computer Organization and Design – The Hardware/Software Interface* by David A. Patterson and John L. Hennessy [1]. The next proposed lab project would be an: Introduction to VHDL and MAX+PLUS II. The objective of this lab is to familiarize the students with the Altera MAX+PLUS II Text Editor, Waveform Editor and Simulator Tools.

The students are given the opportunity to design and implement various combinational logic circuits that can be loaded onto the prototyping board to test it. It is also advisable to assign homework assignment or introduce more labs on VHDL to student well familiarized with the software. An excellent series of VHDL and MAX+PLUS II tutorials can be found in the appendices to the book *Fundamentals of Digital Logic with VHDL Design* by Stephen Brown and Zvonko Vranesic [12]. The following proposed lab project would be more challenging and would be advisable to break the overall project into three or four different parts. Their goal is to implement a MIPS Single-cycle Processor in VHDL much like the project presented in this paper. The first part of the project would be to implement a behavioral VHDL model the control unit for a single-cycle datapath. The objective of this part is to familiarize the students with control unit which will used to control the subsequent parts. The next part of the project would be to implement a behavioral model the ALU and structural model of the register file, instruction and data memory. The following part of the project would include combining the units developed in the previous to parts to build a structural model of the MIPS single-cycle datapath. Other components such as multiplexors, adders, PC will need to be implemented to complete the datapath. Finally the students can implement their designs onto a development board and use the on-board resources such seven segment display to show the current program counter or the VGA video display output to display important signals. Extra credit can be offered to students who implement a deeper instruction set. The next project can feed off the previous one by implementing the MIPS Pipelined Processor from the previous single-cycle implementation. This project can also be divided into a number parts. The first being to implement the pipeline registers: IF/ID, ID/EX, EX/MEM, MEM/WB. Other parts would involve implementing the forwarding unit to take care of data hazards, a hazard control unit to take care of load-use

hazards and implementing branch flushing to handle branch hazards. Extra credit can be offered to students who can successfully implement MIPS processor overflow exception hardware. Additional advanced project can include: porting the design to a new FPGA device architecture, a VHDL implementation of the MIPS multi-cycle FSM controller, the implementation of a floating-point co-processor to the MIPS Processor, a VHDL synthesis model of another RISC processor's instruction set or implementing a superscalar processor. All of these suggested projects can be implemented in a computer architecture course to enhance the learning experience of the students.

6 FUTURE WORK

The main goal of this project was to present the positive effects an FPGA implementation could have on a students experience by integrating a hands-on approach to a simulation-only based class. This project was specifically targeted to the computer architecture courses that use the book *Computer Organization and Design – The Hardware/Software Interface*, to discuss the MIPS processor and instruction set. It would be a good idea to research how different courses like digital design, embedded systems, and digital signal processing could integrate using the FPGA devices in their courses. It would then be desirable to compromise a set of tutorials, reference manuals and laboratory projects to help the student grasp the important concepts of how these tools work. On a different note the VHDL implementations of the MIPS single-cycle and pipelined processors where download and configured onto the FLEX10K70 FPGA, however the processors remained in the prototyping stage. In the future it would be interesting to develop these devices for real world applications. Then one could test and see how the FPGA devices

operate in this type of environment. The classroom integration section presented earlier in this paper also offers different ideas for future course design projects.

7 CONCLUSION

In conclusion, the FPGA implementation of the MIPS processor and tools involved presented in this paper represent my goal of introducing FPGAs to help teach computer architecture courses by presenting the students with an enriching hands-on experience. While simulation is a good effective teaching tool, it can not model the excitement felt when ones own processor design boots up for the first time and operates in real hardware. The development board and tools introduced could easily be integrated into the computer architecture classes, where they could provide students with a enriching hands-on experience formerly unavailable to them. I am convinced that if professors where to integrate these tools into their classes the students would display a better understanding of the class lessons as well as an increased enthusiasm about the work being performed.

"I hear and forget. I see and I remember.
I do and I understand."
-- Chinese proverb

REFERENCES

[1] Patterson, D. A., Hennessy, J. L., *Computer Organization and Design: The Hardware/Software Interface*, 2nd edition, Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[2] Xilinx, *XC4000 Series Field Programmable Gate Arrays Product Specification*, ver. 1.6, 1999.

[3] Altera, *FLEX 10K Embedded Programmable Logic Device Family Data Sheet*, ver. 4.2., 2003.

[4] Altera, *MAX 7000 Programmable Logic Device Family Data Sheet*, ver. 6.02, 2002.

[5] MIPS Technologies, www.mips.com.

[6] SPIM, http://www.cs.wisc.edu/~larus/spim.html.

[7] Altera, *MAX+PLUS II Getting Started Manual*, ver. 6.0, 1995.

[8] Diab, H., Demashkieh, I., "A reconfigurable microprocessor teaching tool", *IEEE Proceedings A*, vol. 137, issue 5, September 1990.

[9] Gray, J., *Designing a Simple FPGA-Optimized RISC CPU and System-on-a-Chip*, 2000.

[10] Altera, *University Program UP2 Development Kit User Guide*, ver. 3.0, 2003.

[11] MIPS Technologies, *MIPS32™ Architecture For Programmers Volume I: Introduction to the MIPS32™ Architecture*, rev. 2.0, 2003.

[12] Brown, S., Vranesic, Z., *Fundamentals of Digital Logic with VHDL Design*, McGraw-Hill Publishers, 2002.

[13] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE, New York, NY, 2002. IEEE Standard 1076-2002.

[14] Land. B, *Electrical Engineering 475 Microprocessor Architectures*, http://instruct1.cit.cornell.edu/Courses/ee475/

[15] Takahashi, R., Ohiwa, H., "Situated Learning on FPGA for Superscalar Microprocessor Design Education", *IEEE Proceedings of the 16th Symposium on Integrated Circuits and System Design*, 2003.

[16] Altera, *ByteBlaster MV Parallel Port Download Cable*, ver. 3.3, 2002.

[17] Larus, J. R., "*SPIM S20: A MIPS R2000 Simulator*", 1993.

APPENDIX

APPENDIX A: RESOURCES

FPGA and PLD Vendors

[1] Altera – http://www.altera.com

[2] Xilinx – http://www.xilinx.com

[3] Actel – http://www.actel.com

[4] Atmel – http://www.atmel.com

FPGA Resources

[1] FPGA CPU News – http://www.fpgacpu.org

[2] FPGA and Programmable Logic Journal - http://www.fpgajournal.com/

[3] FPGA-FAQ - http://www.fpga-faq.com/

MIPS Resources

[1] MIPS Technologies – http://www.mips.com

[2] SPIM a MIPS Simulator – http://www.cs.wisc.edu/~larus/spim.html

VHDL Resources

[1] Hamburg VHDL archive - http://tech-www.informatik.uni-hamburg.de/vhdl/index.html

[2] VHDL Online - http://www.vhdl-online.de/~vhdl/

[3] VHDL Cookbook - http://techwww.informatik.unihamburg.de/vhdl/doc/cookbook/VHDL-Cookbook.pdf

APPENDIX B: ACRONYMS DEFINITIONS

1. FPGA - Field Programmable Gate Array
2. MIPS - Microprocessor without Interlocked Pipeline Stages
3. RISC - Reduced Instruction Set Computer
4. CPU - Central Processing Unit
5. VHDL - Very high speed integrated circuit Hardware Description Language
6. PLD - Programmable Logic Device
7. SRAM - Static Random Access Memory
8. CLB - Configurable Logic Block
9. LUT - LookUp Table
10. EEPROM - Erasable Programmable Read-Only Memory
11. FLEX - Flexible Logic Element matriX
12. EAB - Embedded Array Blocks
13. RAM - Random Access Memory
14. ROM - Read Only Memory
15. FIFO - First-In First-Out
16. LE - Logic Elements
17. LAB - Logic Array Block
18. PIA - Programmable Interconnect Array
19. ISA - Instruction Set Architecture
20. CAD - Computer Aided Design
21. HDL - Hardware Description Language
22. MAX+PLUS - Multiple Array matriX Programmable Logic User System
23. DIP - Dual Inline Package
24. GPR - General Purpose Register
25. PC - Program Counter
26. ALU - Arithmetic Logic Unit
27. LPM - Library of Parameterized Modules
28. IF - Instruction Fetch
29. ID - Instruction Decode
30. EX - Execute
31. MEM - MEMory
32. WB - Write Back
33. JTAG - Joint Test Action Group

APPENDIX C: MIPS SINGLE-CYCLE – VHDL CODE

```
-------------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
--
--
-- Filename: mips_sc.vhd
-- Description: VHDL code to implment the structual design of
-- the MIPS Single-cycle processor.  The top-level file connects
-- all the units together to complete functional processor.
-------------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY mips_sc IS

--Signals used for Simulation & Debug must be commented out if
--implementing the UP2 Hardware implementation
PORT(   PC                      : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
        ALU_Result_out          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
        Read_Data1_out          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
        Read_Data2_out          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
        Write_Data_out          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
        Write_Reg_out           : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
        Instruction_out         : OUT   STD_LOGIC_VECTOR (31 DOWNTO 0);
        Branch_out              : OUT   STD_LOGIC;
        Branch_NE_out           : OUT   STD_LOGIC;
        Zero_out                : OUT   STD_LOGIC;
        Jump_out                : OUT   STD_LOGIC;
        MemWrite_out            : OUT   STD_LOGIC;
        RegWrite_out            : OUT   STD_LOGIC;
        Clock, Reset            : IN    STD_LOGIC);

--Signals used for UP2 Board Implemntation
--All signals here must be assigned to a pin on
--the FLEX10K70 device.
--PORT( D1_a, D1_b, D1_c, D1_d,
--              D1_e, D1_f, D1_g,D1_pb        : OUT STD_LOGIC;
--              D2_a, D2_b, D2_c, D2_d,
--              D2_e, D2_f, D2_g,D2_pb        : OUT STD_LOGIC;
--              Clock, Reset, PB              : IN  STD_LOGIC);
END mips_sc;

ARCHITECTURE structure of mips_sc IS

--Declare all components/units/modules used that
--makeup the MIPS Single-cycle processor.
--COMPONENT debounce
--      PORT (  Clock   : IN STD_LOGIC;
 --             Pbutton : IN STD_LOGIC;
 --             Pulse   : OUT STD_LOGIC);
--END COMPONENT;

COMPONENT instrfetch
        PORT(   PC_Out          : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
                Instruction     : OUT   STD_LOGIC_VECTOR(31 DOWNTO 0);
                Add_Result      : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
                PC_plus_4_out   : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
                Branch          : IN    STD_LOGIC;
                Branch_NE       : IN    STD_LOGIC;
                Zero            : IN    STD_LOGIC;
                Jump            : IN    STD_LOGIC;
                Jump_Address    : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
```

```
                Clock, Reset    : IN    STD_LOGIC);
END COMPONENT;

COMPONENT operandfetch
        PORT (
                Read_Data_1     : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Read_Data_2     : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Write_Reg       : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
                RegWrite        : IN    STD_LOGIC;
                RegDst          : IN    STD_LOGIC;
                ALU_Result      : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                MemtoReg        : IN    STD_LOGIC;
                Read_data       : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Instruction     : IN    STD_LOGIC_VECTOR (31 DOWNTO 0);
                Sign_Extend     : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Jump_Instr      : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Clock, Reset    : IN    STD_LOGIC);
END COMPONENT;

COMPONENT controlunit IS
PORT(
                Opcode          : IN    STD_LOGIC_VECTOR (5 DOWNTO 0);
                RegDst          : OUT   STD_LOGIC;
                Branch          : OUT   STD_LOGIC;
                Branch_NE       : OUT   STD_LOGIC;
                MemRead         : OUT   STD_LOGIC;
                MemtoReg        : OUT   STD_LOGIC;
                ALU_Op          : OUT   STD_LOGIC_VECTOR (1 DOWNTO 0);
                MemWrite        : OUT   STD_LOGIC;
                ALUSrc          : OUT   STD_LOGIC;
                RegWrite        : OUT   STD_LOGIC;
                Jump            : OUT   STD_LOGIC;
                Clock, Reset    : IN    STD_LOGIC);
END COMPONENT;

COMPONENT execution IS
        PORT(
                Read_Data_1     : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Read_Data_2     : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Sign_Extend     : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Jump_Instr      : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Jump_Address    : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                ALUSrc          : IN    STD_LOGIC;
                Zero            : OUT   STD_LOGIC;
                ALU_Result      : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Funct_field     : IN    STD_LOGIC_VECTOR (5 DOWNTO 0);
                ALU_Op          : IN    STD_LOGIC_VECTOR (1 DOWNTO 0);
                Add_Result      : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                PC_plus_4       : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Clock, Reset    : IN    STD_LOGIC);
END COMPONENT;

COMPONENT datamemory IS
        PORT(
                Read_Data       : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Address         : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Write_Data      : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                MemRead         : IN    STD_LOGIC;
                MemWrite        : IN    STD_LOGIC;
                Clock, Reset    : IN    STD_LOGIC);
END COMPONENT;

--COMPONENT sevenseg_display IS
--      PORT(   Digit1          : IN    STD_LOGIC_VECTOR (3 DOWNTO 0);
--              Digit2          : IN    STD_LOGIC_VECTOR (3 DOWNTO 0);
--              D1seg_a         : OUT   STD_LOGIC;
--              D1seg_b         : OUT   STD_LOGIC;
--              D1seg_c         : OUT   STD_LOGIC;
--              D1seg_d         : OUT   STD_LOGIC;
--              D1seg_e         : OUT   STD_LOGIC;
--              D1seg_f         : OUT   STD_LOGIC;
```

```
--             D1seg_g        : OUT   STD_LOGIC;
--             D1pb           : OUT   STD_LOGIC;
--             D2seg_a        : OUT   STD_LOGIC;
--             D2seg_b        : OUT   STD_LOGIC;
--             D2seg_c        : OUT   STD_LOGIC;
--             D2seg_d        : OUT   STD_LOGIC;
--             D2seg_e        : OUT   STD_LOGIC;
--             D2seg_f        : OUT   STD_LOGIC;
--             D2seg_g        : OUT   STD_LOGIC;
--             D2pb           : OUT   STD_LOGIC;
--             Clock, Reset   : IN    STD_LOGIC);
--END COMPONENT;

--Signals used to connect VHDL Components
SIGNAL  Add_Result     : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  ALU_Result     : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  ALU_Op         : STD_LOGIC_VECTOR (1 DOWNTO 0);
SIGNAL  ALUSrc         : STD_LOGIC;
SIGNAL  Branch         : STD_LOGIC;
SIGNAL  Branch_NE      : STD_LOGIC;
SIGNAL  Instruction    : STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL  Jump           : STD_LOGIC;
SIGNAL   Jump_Address  : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL   Jump_Instr    : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  MemRead        : STD_LOGIC;
SIGNAL  MemtoReg       : STD_LOGIC;
SIGNAL  MemWrite       : STD_LOGIC;
SIGNAL  PC_Out         : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  PC_plus_4      : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  Read_data      : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  Read_Data_1    : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  Read_Data_2    : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  RegDst         : STD_LOGIC;
SIGNAL  RegWrite       : STD_LOGIC;
SIGNAL  Sign_Extend    : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  Write_Reg      : STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL  Zero           : STD_LOGIC;

--New Clock Signal used when PushButton is used
--to create a clock tick
SIGNAL   NClock        : STD_LOGIC;

BEGIN

        --Signals assigned to display output pins for simulator
        --These signals must be commented when implementing
        --the UP2 Development board.
        PC             <=     PC_Out;
        ALU_Result_out <=     ALU_Result;
        Read_Data1_out <=     Read_Data_1;
        Read_Data2_out <=     Read_Data_2;
        Write_Data_out <=     Read_data WHEN (MemtoReg = '1') ELSE ALU_Result;
        Write_Reg_out  <=     Write_Reg;
        Instruction_out<=     Instruction;
        Branch_out     <=     Branch;
        Branch_NE_out  <=     Branch_NE;
        Zero_out       <=     Zero;
        Jump_out       <=     Jump;
        MemWrite_out   <=     MemWrite;
        RegWrite_out   <=     '0' WHEN Write_Reg = "00000" ELSE RegWrite;

        --Connect each signal to respective line
        --NCLK: debounce PORT MAP(
        --             Clock          => Clock,
        --             PButton        => PB,
        --             Pulse          => NClock);

        IFE : instrfetch PORT MAP (
                       PC_Out         => PC_Out,
                       Instruction    => Instruction,
                       Add_Result     => Add_Result,
```

```
                PC_plus_4_out  => PC_plus_4,
                Branch         => Branch,
                Branch_NE      => Branch_NE,
                Zero           => Zero,
                Jump           => Jump,
                Jump_Address   => Jump_Address,
                Clock          => Clock,
                Reset          => Reset );

        ID : operandfetch PORT MAP (
                Read_Data_1    => Read_Data_1,
                Read_Data_2    => Read_Data_2,
                Write_Reg      => Write_Reg,
                RegWrite       => RegWrite,
                RegDst         => RegDst,
                ALU_Result     => ALU_Result,
                MemtoReg       => MemtoReg,
                Read_data      => Read_data,
                Instruction    => Instruction,
                Sign_Extend    => Sign_Extend,
                Jump_Instr     => Jump_Instr,
                Clock          => Clock,
                Reset          => Reset );

        CTRL : controlunit PORT MAP (
                Opcode         => Instruction (31 DOWNTO 26),
                RegDst         => RegDst,
                Jump           => Jump,
                Branch         => Branch,
                Branch_NE      => Branch_NE,
                MemRead        => MemRead,
                MemtoReg       => MemtoReg,
                ALU_Op         => ALU_Op,
                MemWrite       => MemWrite,
                ALUSrc         => ALUSrc,
                RegWrite       => RegWrite,
                Clock          => Clock,
                Reset          => Reset );

        EX : execution PORT MAP (
                Read_Data_1    => Read_Data_1,
                Read_Data_2    => Read_Data_2,
                Sign_Extend    => Sign_Extend,
                ALUSrc         => ALUSrc,
                Zero           => Zero,
                ALU_Result     => ALU_Result,
                Funct_field    => Instruction (5 DOWNTO 0),
                ALU_Op         => ALU_Op,
                Add_Result     => Add_Result,
                PC_plus_4      => PC_plus_4,
                Jump_Address   => Jump_Address,
                Jump_Instr     => Jump_Instr,
                Clock          => Clock,
                Reset          => Reset );

        MEM : datamemory PORT MAP (
                Read_Data      => Read_Data,
                Address        => ALU_Result,
                Write_Data     => Read_Data_2,
                MemRead        => MemRead,
                MemWrite       => MemWrite,
                Clock          => Clock,
                Reset          => Reset );

--      SSD: sevenseg_display PORT MAP(
--                      Digit1         => PC_Out (7 DOWNTO 4),
--                      Digit2         => PC_Out (3 DOWNTO 0),
--                      D1seg_a        => D1_a,
--                      D1seg_b        => D1_b,
--                      D1seg_c        => D1_c,
--                      D1seg_d        => D1_d,
```

```
--                    D1seg_e        => D1_e,
--                    D1seg_f        => D1_f,
--                    D1seg_g        => D1_g,
--                    D1pb           => D1_pb,
--                    D2seg_a        => D2_a,
--                    D2seg_b        => D2_b,
--                    D2seg_c        => D2_c,
--                    D2seg_d        => D2_d,
--                    D2seg_e        => D2_e,
--                    D2seg_f        => D2_f,
--                    D2seg_g        => D2_g,
--                    D2pb           => D2_pb,
--                    Clock          => NClock,
--                    Reset          => Reset);
END structure;


-----------------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
--
--
-- Filename: instrfetch.vhd
-- Description: VHDL code to implment the Instruction Fetch unit
-- of the MIPS single-cycle processor as seen in Chapter #5 of
-- Patterson and Hennessy book.  This file involves the use of the
-- LPM Components (LPM_ROM) to declare the Instruction Memory
-- as a read only memory (ROM).  See MAX+PLUS II Help on
-- "Implementing RAM & ROM (VHDL)" for details.
-----------------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_ARITH.ALL;
 USE IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY LPM;
 USE LPM.LPM_COMPONENTS.ALL;

ENTITY instrfetch IS
PORT(   SIGNAL PC_Out                 : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
               SIGNAL Instruction     : OUT   STD_LOGIC_VECTOR(31 DOWNTO 0);
               SIGNAL Add_Result      : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
               SIGNAL PC_plus_4_out   : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
               SIGNAL Branch          : IN    STD_LOGIC;
               SIGNAL Branch_NE       : IN    STD_LOGIC;
               SIGNAL Zero            : IN    STD_LOGIC;
               SIGNAL Jump            : IN    STD_LOGIC;
               SIGNAL Jump_Address    : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
               SIGNAL Clock, Reset    : IN    STD_LOGIC);
END instrfetch;

ARCHITECTURE behavior OF instrfetch IS
       SIGNAL PC              :       STD_LOGIC_VECTOR (9 DOWNTO 0);
       SIGNAL PC_plus_4       :       STD_LOGIC_VECTOR (9 DOWNTO 0);
       SIGNAL Next_PC         :       STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN

--Declare Instruction Memory as ROM
Instr_Memory: LPM_ROM

       GENERIC MAP(
              LPM_WIDTH                     => 32,
              LPM_WIDTHAD                   => 8,
              -- *.mif FILE used to initialize memory values
              LPM_FILE                      => "instruction_memory.mif",
              --LPM_FILE                    => "pipelining_example.mif",
              LPM_OUTDATA                   => "UNREGISTERED",
              LPM_ADDRESS_CONTROL           => "UNREGISTERED")
```

```
	PORT MAP (
		--Bits (9 DOWNTO 2) used to word-address instructions
		-- e.g. +1 not +4(byte-addressed)
		address		=> PC(9 DOWNTO 2),
		--Output of Instruction Memory is 32-bit instruction
		q		=> Instruction );

		-- Output Signals copied
		PC_Out		<= PC (7 DOWNTO 0);
		PC_plus_4_out	<= PC_plus_4 (7 DOWNTO 0);

		-- Increment PC by 4 by shifting bits
		PC_plus_4 (9 DOWNTO 2)	<= PC (9 DOWNTO 2) + 1;
		PC_plus_4 (1 DOWNTO 0)	<= "00";

		-- Mux to select NEXT_PC as Branch Address, PC + 4,  or Jump
		Next_PC  <= Add_result WHEN ((Branch = '1') AND
					(Zero = '1') AND (Branch_NE = '0') ) OR
					((Branch_NE = '1') AND (Zero = '0')) ELSE
				Jump_Address WHEN ( Jump = '1' ) ELSE
				PC_plus_4 (9 DOWNTO 2);

	PROCESS
		BEGIN
			WAIT UNTIL ( Clock'EVENT ) AND ( Clock = '1' );
			IF Reset = '1' THEN
				PC <="0000000000";
			ELSE
				PC (9 DOWNTO 2) <= Next_PC;
			END IF;
	END PROCESS;
END behavior;


-----------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
--
--
-- Filename: operandfetch.vhd
-- Description: VHDL code to implment the Operand Fetch unit
-- of the MIPS single-cycle processor as seen in Chapter #5 of
-- Patterson and Hennessy book.
--
-----------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_ARITH.ALL;
 USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY operandfetch IS
	PORT (  --Registers & MUX
		Read_Data_1	:	OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		Read_Data_2	:	OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		Write_Reg	:	OUT	STD_LOGIC_VECTOR (4 DOWNTO 0);
		RegWrite	:	IN	STD_LOGIC;
		RegDst		:	IN	STD_LOGIC;
		--Data Memory & MUX
		ALU_Result	:	IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		MemtoReg	:	IN	STD_LOGIC;
		Read_data	:	IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		--Misc
		Instruction	:	IN	STD_LOGIC_VECTOR (31 DOWNTO 0);
		Sign_Extend	:	OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		Jump_Instr	:	OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		Clock, Reset	:	IN	STD_LOGIC);
END operandfetch;
```

```
ARCHITECTURE behavior OF operandfetch IS
--Declare Register File as a one-dimensional array
--Thirty-two Registers each 8-bits wide
TYPE register_file IS ARRAY (0 TO 31) OF STD_LOGIC_VECTOR (7 DOWNTO 0);

        SIGNAL register_array         : register_file;
        SIGNAL read_register_address1 : STD_LOGIC_VECTOR (4 DOWNTO 0);
        SIGNAL read_register_address2 : STD_LOGIC_VECTOR (4 DOWNTO 0);
        SIGNAL write_register_address : STD_LOGIC_VECTOR (4 DOWNTO 0);
        SIGNAL write_register_address0: STD_LOGIC_VECTOR (4 DOWNTO 0);
        SIGNAL write_register_address1: STD_LOGIC_VECTOR (4 DOWNTO 0);
        SIGNAL write_data             : STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL instruction_15_0       : STD_LOGIC_VECTOR (15 DOWNTO 0);
        SIGNAL instruction_25_0       : STD_LOGIC_VECTOR (25 DOWNTO 0);

BEGIN

        --Copy Instruction bits to signals
        read_register_address1  <= Instruction (25 DOWNTO 21);
        read_register_address2  <= Instruction (20 DOWNTO 16);
        write_register_address0 <= Instruction (20 DOWNTO 16);
        write_register_address1 <= Instruction (15 DOWNTO 11);
        instruction_15_0        <= Instruction (15 DOWNTO 0);
        instruction_25_0        <= Instruction (25 DOWNTO 0);

        --Register File: Read_Data_1 Output
        Read_Data_1 <= register_array(CONV_INTEGER(read_register_address1 (4 DOWNTO 0)));

        --Register File: Read_Data_2 Output
        Read_Data_2 <= register_array(CONV_INTEGER(read_register_address2 (4 DOWNTO 0)));

        --Register File: MUX to select Write Register Address
        write_register_address <= write_register_address1 WHEN (RegDst = '1')
                                  ELSE write_register_address0;

        --Register File: MUX to select Write Register Data
        write_data <= ALU_result (7 DOWNTO 0) WHEN (MemtoReg = '0')
                      ELSE Read_data;

        --Sign Extend
        --NOTE: Due to 8-bit data width design
        --No sign extension is NEEDED
        Sign_Extend <= instruction_15_0 (7 DOWNTO 0);

        --Jump Instruction
        Jump_Instr <= instruction_25_0 (7 DOWNTO 0);

        --WB - Write Register
        Write_Reg  <= write_register_address;

        PROCESS
        BEGIN
                WAIT UNTIL ( Clock'EVENT ) AND ( Clock = '1' );
                IF (Reset = '1') THEN
                        --Reset Registers to own Register Number
                        --Used for testing ease.
                        FOR i IN 0 TO 31 LOOP
                                register_array(i) <= CONV_STD_LOGIC_VECTOR(i,8);
                        END LOOP;

                --Write Register File if RegWrite signal asserted
                ELSIF (RegWrite = '1') AND (write_register_address /= 0) THEN
                        register_array(CONV_INTEGER(write_register_address (4 DOWNTO 0)))
                        <= write_data;
                END IF;
        END PROCESS;
END behavior;


--------------------------------------------------------------------
```

```
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
--
--
-- Filename: control_unit.vhd
-- Description: VHDL code to implement the Control Unit
-- of the MIPS single-cycle processor as seen in Chapter #5 of
-- Patterson and Hennessy book.
----------------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_ARITH.ALL;
 USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY controlunit IS
PORT(   SIGNAL  Opcode          : IN  STD_LOGIC_VECTOR (5 DOWNTO 0);
        SIGNAL  RegDst          : OUT STD_LOGIC;
        SIGNAL  Branch          : OUT STD_LOGIC;
        SIGNAL  Branch_NE       : OUT STD_LOGIC;
        SIGNAL  MemRead         : OUT STD_LOGIC;
        SIGNAL  MemtoReg        : OUT STD_LOGIC;
        SIGNAL  ALU_Op          : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
        SIGNAL  MemWrite        : OUT STD_LOGIC;
        SIGNAL  ALUSrc          : OUT STD_LOGIC;
        SIGNAL  RegWrite        : OUT STD_LOGIC;
        SIGNAL   Jump           : OUT STD_LOGIC;
        SIGNAL  Clock, Reset    : IN  STD_LOGIC);
END controlunit;

ARCHITECTURE behavior OF controlunit IS

        SIGNAL R_format, LW, SW, BEQ, BNE, JMP, ADDI :       STD_LOGIC;
        SIGNAL Opcode_Out                            :       STD_LOGIC_VECTOR (5 DOWNTO 0);

BEGIN

        --Decode the Instruction OPCode to determine type
        --and set all corresponding control signals &
        --ALUOP function signals.
        R_format        <= '1' WHEN Opcode = "000000" ELSE '0';
        LW              <= '1' WHEN Opcode = "100011" ELSE '0';
        SW              <= '1' WHEN Opcode = "101011" ELSE '0';
        BEQ             <= '1' WHEN Opcode = "000100" ELSE '0';
        BNE             <= '1' WHEN Opcode = "000101" ELSE '0';
        JMP             <= '1' WHEN Opcode = "000010" ELSE '0';
        ADDI            <= '1' WHEN Opcode = "001000" ELSE '0';

        RegDst          <= R_format;
        Branch          <= BEQ;
        Branch_NE       <= BNE;
        Jump            <= JMP;
        MemRead         <= LW;
        MemtoReg        <= LW;
        ALU_Op(1)       <= R_format;
        ALU_Op(0)       <= BEQ OR BNE;
        MemWrite        <= SW;
        ALUSrc          <= LW OR SW OR ADDI;
        RegWrite        <= R_format OR LW OR ADDI;

END behavior;


----------------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
```

```
--
--
-- Filename: execution_unit.vhd
-- Description: VHDL code to implment the Execution Unit
-- of the MIPS single-cycle processor as seen in Chapter #5 of
-- Patterson and Hennessy book.
----------------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_ARITH.ALL;
 USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY execution IS
PORT(   --ALU Signals
        Read_Data_1     : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
        Read_Data_2     : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
        Sign_Extend     : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
        ALUSrc          : IN    STD_LOGIC;
        Zero            : OUT   STD_LOGIC;
        ALU_Result      : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
        --ALU Control
        Funct_field     : IN    STD_LOGIC_VECTOR (5 DOWNTO 0);
        ALU_Op          : IN    STD_LOGIC_VECTOR (1 DOWNTO 0);
        --Branch Adder
        Add_Result      : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
        PC_plus_4       : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
        --Jump Adress
        Jump_Instr      : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
        Jump_Address    : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
        --Misc
        Clock, Reset    : IN    STD_LOGIC);
END execution;

ARCHITECTURE behavior of execution IS

        SIGNAL A_input, B_input       :       STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL ALU_output             :       STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL Branch_Add             :       STD_LOGIC_VECTOR (8 DOWNTO 0);
        SIGNAL Jump_Add               :       STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL ALU_Control            :       STD_LOGIC_VECTOR (2 DOWNTO 0);

BEGIN

        --ALU Input
        A_input <= Read_Data_1;

        --MUX to select second ALU Input
        B_input <= Read_Data_2
                WHEN (ALUSrc = '0')
                ELSE (Sign_Extend (7 DOWNTO 0));

        --Set ALU Control Bits

        ALU_Control(2) <= ( Funct_field(1) AND ALU_Op(1) ) OR ALU_Op(0);
        ALU_Control(1) <= ( NOT Funct_field(2) ) OR ( NOT ALU_Op(1) );
        ALU_Control(0) <= ( Funct_field(1) AND Funct_field(3) AND ALU_Op(1) ) OR
                          ( Funct_field(0) AND Funct_field(2) AND ALU_Op(1) );

        --Set ALU_Zero
        Zero <= '1' WHEN ( ALU_output (7 DOWNTO 0) = "00000000") ELSE '0';

        --ALU Output:  Must check for SLT instruction and set correct ALU_output
        ALU_Result <= ("0000000" & ALU_output (7)) WHEN ALU_Control = "111"
                       ELSE ALU_output (7 DOWNTO 0);

        --Branch Adder
        Branch_Add <= PC_plus_4 (7 DOWNTO 2) + Sign_Extend (7 DOWNTO 0);
        Add_Result <= Branch_Add (7 DOWNTO 0);

        --Jump Address
        Jump_Add <= Jump_Instr (7 DOWNTO 0);
```

```
	Jump_Address <= Jump_Add;

	--Compute the ALU_output use the ALU_Control signals
	PROCESS (ALU_Control, A_input, B_input)
		BEGIN --ALU Operation
		CASE ALU_Control IS
			--Function: A_input AND B_input
			WHEN "000" => ALU_output <= A_input AND B_input;
			--Function: A_input OR B_input
			WHEN "001" => ALU_output <= A_input OR B_input;
			--Function: A_input ADD B_input
			WHEN "010" => ALU_output <= A_input + B_input;
			--Function: A_input ? B_input
			WHEN "011" => ALU_output <= "00000000";
			--Function: A_input ? B_input
			WHEN "100" => ALU_output <= "00000000";
			--Function: A_input ? B_input
			WHEN "101" => ALU_output <= "00000000";
			--Function: A_input SUB B_input
			WHEN "110" => ALU_output <= A_input - B_input;
			--Function: SLT (set less than)
			WHEN "111" => ALU_output <= A_input - B_input;
			WHEN OTHERS => ALU_output <= "00000000";
		END CASE;
	END PROCESS;
END behavior;


--------------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
--
--
-- Filename: data_memory.vhd
-- Description: VHDL code to implment the Instruction Fetch unit
-- of the MIPS single-cycle processor as seen in Chapter #5 of
-- Patterson and Hennessy book.  This file involves the use of the
-- LPM Components (LPM_RAM) to declare the Instruction Memory
-- as a random access memory (RAM).  See MAX+PLUS II Help on
-- "Implementing RAM & ROM (VHDL)" for details.
--------------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_ARITH.ALL;
 USE IEEE.STD_LOGIC_SIGNED.ALL;
LIBRARY LPM;
 USE LPM.LPM_COMPONENTS.ALL;

ENTITY datamemory IS
PORT(	Read_Data	: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
	Address		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
	Write_Data	: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
	MemRead		: IN	STD_LOGIC;
	MemWrite	: IN	STD_LOGIC;
	Clock, Reset	: IN	STD_LOGIC);
END datamemory;

ARCHITECTURE behavior OF datamemory IS
SIGNAL LPM_WRITE : STD_LOGIC;

BEGIN
	datamemory : LPM_RAM_DQ
	GENERIC MAP(
		LPM_WIDTH		=>	8,
		LPM_WIDTHAD		=>	8,
		LPM_FILE		=> "data_memory.mif",
		LPM_INDATA		=> "REGISTERED",
		LPM_ADDRESS_CONTROL	=> "UNREGISTERED",
```

```
                LPM_OUTDATA             => "UNREGISTERED")

        PORT MAP(
                inclock         => Clock,
                data            => Write_Data,
                address         => Address,
                we              => LPM_WRITE,
                q               => Read_Data);

        --Write Data Memory
        LPM_WRITE <= MemWrite AND (NOT Clock);

END behavior;


--------------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
--
--
-- Filename: debounce.vhd
-- Description: VHDL code to debounce the UP2 Board push buttons.
-- without this code pushing the button could actually result
-- in multiple contacts due to the spring found in the push button.
-- ONLY USED in HARDWARE IMPLEMENTATION
--------------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY debounce IS
PORT (  Clock           : IN STD_LOGIC;
        PButton         : IN STD_LOGIC;
        Pulse           : OUT STD_LOGIC);
END debounce;

ARCHITECTURE behavior OF debounce IS
        SIGNAL count    : STD_LOGIC_VECTOR (17 DOWNTO 0);

BEGIN
        PROCESS (Clock)
        BEGIN

        IF PButton = '1' THEN
                count <= "00000000000000000";

        ELSIF (Clock'EVENT AND Clock = '1') THEN
                IF (count /= "11111111111111111") THEN
                        count <= count + 1;
                END IF;
        END IF;

        IF (count = "11111111111111110") AND (PButton = '0') THEN
                Pulse <= '1';
        ELSE
                Pulse <= '0';
        END IF;

  END PROCESS;
END behavior;


--------------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
```

```
--
--
-- Filename: sevenseg_display.vhd
-- Description: VHDL code to implment the Most Significant Digit
-- (MSD) and Least Significant Digit (LSD) seven-segment displays
-- on the UP2 Development board.  A 4-bit hexidecimal number is
-- decoded and used to the respective MSD/LSD segments.
--
-- NOTE: seven segment display is assserted low
-- The pins are configured as follows:
--                 a
--               +---+
--             f | g | b
--               +---+
--             e |   | c
--               +---+ . (dp)
--                 d
----------------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_ARITH.ALL;
 USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY sevenseg_display IS
PORT(   Digit1, Digit2                        : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        D1seg_a,D1seg_b,D1seg_c,D1seg_d,
        D1seg_e,D1seg_f,D1seg_g,D1pb          : OUT STD_LOGIC;
        D2seg_a,D2seg_b,D2seg_c,D2seg_d,
        D2seg_e,D2seg_f,D2seg_g,D2pb          : OUT STD_LOGIC;
        Clock, Reset                          : IN  STD_LOGIC);
END sevenseg_display;

ARCHITECTURE behavior OF sevenseg_display IS
        SIGNAL data1, data2 : STD_LOGIC_VECTOR (6 DOWNTO 0);

BEGIN

        --Decode the first digit - MSD
        PROCESS (Digit1)
        BEGIN
          CASE Digit1 IS
                WHEN "0000" => data1 <= "1111110";
                WHEN "0001" => data1 <= "0110000";
                WHEN "0010" => data1 <= "1101101";
                WHEN "0011" => data1 <= "1111001";
                WHEN "0100" => data1 <= "0110011";
                WHEN "0101" => data1 <= "1011011";
                WHEN "0110" => data1 <= "1011111";
                WHEN "0111" => data1 <= "1110000";
                WHEN "1000" => data1 <= "1111111";
                WHEN "1001" => data1 <= "1111011";
                WHEN "1010" => data1 <= "1110111";
                WHEN "1011" => data1 <= "0011111";
                WHEN "1100" => data1 <= "1001110";
                WHEN "1101" => data1 <= "0111101";
                WHEN "1110" => data1 <= "1001111";
                WHEN "1111" => data1 <= "1000111";
                WHEN OTHERS => data1 <= "0000001";
          END CASE;
        END PROCESS;

        --Decode the second digit - LSD
        PROCESS (Digit2)
        BEGIN
          CASE Digit2 IS
                WHEN "0000" => data2 <= "1111110";
                WHEN "0001" => data2 <= "0110000";
                WHEN "0010" => data2 <= "1101101";
                WHEN "0011" => data2 <= "1111001";
                WHEN "0100" => data2 <= "0110011";
                WHEN "0101" => data2 <= "1011011";
```

```
            WHEN "0110" => data2 <= "1011111";
            WHEN "0111" => data2 <= "1110000";
            WHEN "1000" => data2 <= "1111111";
            WHEN "1001" => data2 <= "1111011";
            WHEN "1010" => data2 <= "1110111";
            WHEN "1011" => data2 <= "0011111";
            WHEN "1100" => data2 <= "1001110";
            WHEN "1101" => data2 <= "0111101";
            WHEN "1110" => data2 <= "1001111";
            WHEN "1111" => data2 <= "1000111";
            WHEN OTHERS => data2 <= "0000001";
      END CASE;
    END PROCESS;


    PROCESS
            BEGIN
                    WAIT UNTIL ( Clock'EVENT ) AND ( Clock = '1' );
                    IF (Reset = '1') THEN
                            --On Reset display two dases e.g. - -
                            D1seg_a <= '1';
                            D1seg_b <= '1';
                            D1seg_c <= '1';
                            D1seg_d <= '1';
                            D1seg_e <= '1';
                            D1seg_f <= '1';
                            D1seg_g <= '0';
                            D1pb    <= '1';
                            D2seg_a <= '1';
                            D2seg_b <= '1';
                            D2seg_c <= '1';
                            D2seg_d <= '1';
                            D2seg_e <= '1';
                            D2seg_f <= '1';
                            D2seg_g <= '0';
                            D2pb    <= '1';
                    ELSE
                            --Invert the data signal because
                            --seven segment display is assserted low
                            D1seg_a <= NOT data1(6);
                            D1seg_b <= NOT data1(5);
                            D1seg_c <= NOT data1(4);
                            D1seg_d <= NOT data1(3);
                            D1seg_e <= NOT data1(2);
                            D1seg_f <= NOT data1(1);
                            D1seg_g <= NOT data1(0);
                            D1pb    <= '1';
                            D2seg_a <= NOT data2(6);
                            D2seg_b <= NOT data2(5);
                            D2seg_c <= NOT data2(4);
                            D2seg_d <= NOT data2(3);
                            D2seg_e <= NOT data2(2);
                            D2seg_f <= NOT data2(1);
                            D2seg_g <= NOT data2(0);
                            D2pb    <= '1';
                    END IF;
    END PROCESS;
END behavior;
```

APPENDIX D: MIPS SINGLE-CYCLE – SIMULATION

```
-- ============================================================================
-- VICTOR RUBIO
-- FILENAME: INSTRUCTION_MEMORY.MIF
-- DESCRIPTION: INSTRUCTION MEMORY INITIALIZATION FILE FOR MIPS
-- SINGLE-CYCLE PROCESSOR
-- ============================================================================
--256 X 32 ROM IMPLEMENTED USING
--FOUR EMBEDDED ARRAY BLOCKS ON FLEX10K70 DEVICE
DEPTH = 256;
WIDTH = 32;
--DISPLAY IN HEXIDECIMAL FORMAT
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT
BEGIN
  --INITIALIZED DATA MEMORY VALUES
  --00 : 55;
  --01 : AA;
  --02 : 11;
  --03 : 33;

  --DEFAULT INSTRUCTION MEMORY
  [00..FF] : 00000000;

                --PC : INSTRUCTION
  00: 8C410001;  --00 : LW  $1, 1(2)     --> $1 = MEM (0x02 + 0x01)
                 --                            = MEM (03)       = 0x33
  01: 00A41822;  --04 : SUB $3, $5, $4   --> $3 = 0x05 -   0x04 = 0x01
  02: 00E61024;  --08 : AND $2, $7, $6   --> $2 = 0x07 AND 0x06 = 0x06
  03: 00852025;  --0C : OR  $4, $4, $5   --> $4 = 0x04 OR  0x05 = 0x05
  04: 00C72820;  --10 : ADD $5, $6, $7   --> $5 = 0x06 +   0x07 = 0x0D
  05: 1421FFFA;  --14 : BNE $1, $1, -24  --> NOT TAKEN
  06: 1022FFFF;  --18 : BEQ $1, $2, -4   --> NOT TAKEN
  07: 0062302A;  --1C : SLT $6, $3, $2   --> $6 =  $3  <  $2 = 0x01
                 --                           = 0x01 < 0x06
  08: 10210002;  --20 : BEQ $1, $1, 2    --> TAKEN: 0x33 = 0x33
  09: 00000000;  --24 : NOP               --> NOP
  0A: 00000000;  --28 : NOP               --> NOP
  0B: AC010002;  --2C : SW  $1, 2         --> $1 = 0X33 = MEMORY(02)
  0C: 00232020;  --30 ; ADD $4, $1, $3   --> $4= 0x33 + 0x01 = 0x34
  0D: 08000000;  --34 : JUMP 0            --> JUMP TO PC = 00
END;
```

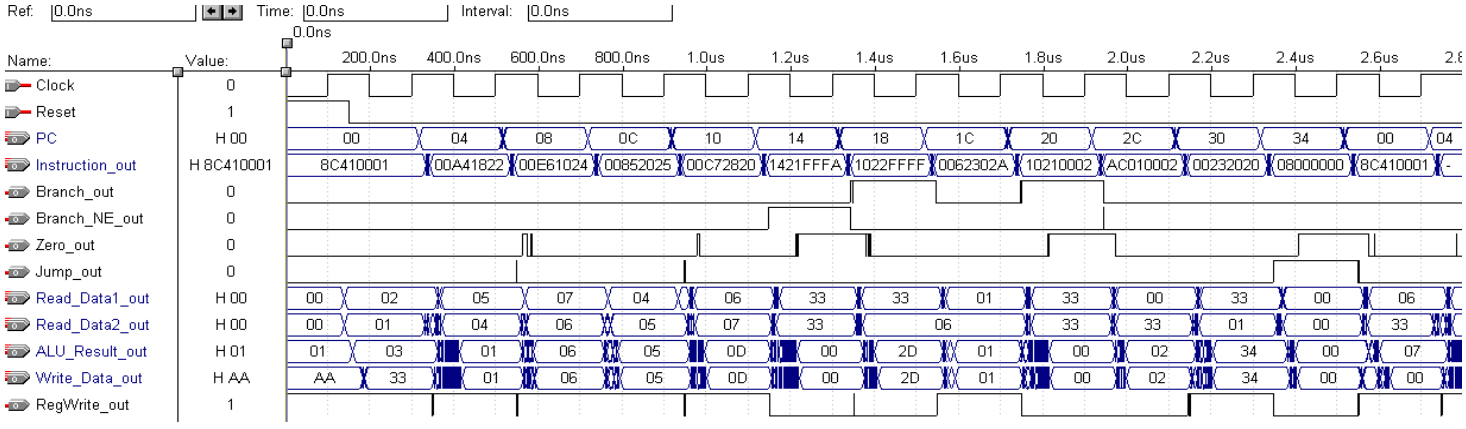

Figure D.1 MIPS Single-cycle Simulation Waveform

APPENDIX E: MIPS SINGLE-CYCLE – SIMULATION – SPIM VALIDATION

```
Registers
=========================
 PC     = 00000000     EPC    = 00000000     Cause  = 00000000     BadVAddr= 00000000
 Status = 00000000     HI     = 00000000     LO     = 00000000
                                 General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000033  R24 (t8) = 00000000
R1  (at) = 10010000  R9  (t1) = 00000005  R17 (s1) = 00000001  R25 (t9) = 00000000
R2  (v0) = 0000000a  R10 (t2) = 00000004  R18 (s2) = 00000006  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000007  R19 (s3) = 00000034  R27 (k1) = 00000000
R4  (a0) = 1001012f  R12 (t4) = 00000006  R20 (s4) = 0000000d  R28 (gp) = 10008000
R5  (a1) = 00000000  R13 (t5) = 00000003  R21 (s5) = 00000001  R29 (sp) = 7fffe838
R6  (a2) = 7fffe840  R14 (t6) = 00000000  R22 (s6) = 00000000  R30 (s8) = 00000000
R7  (a3) = 00000000  R15 (t7) = 00000000  R23 (s7) = 00000000  R31 (ra) = 00000000
                               Double Floating Point Registers
FP0 =00000000,00000000 FP8 =00000000,00000000 FP16=00000000,00000000 FP24=00000000,00000000
FP2 =00000000,00000000 FP10=00000000,00000000 FP18=00000000,00000000 FP26=00000000,00000000
FP4 =00000000,00000000 FP12=00000000,00000000 FP20=00000000,00000000 FP28=00000000,00000000
FP6 =00000000,00000000 FP14=00000000,00000000 FP22=00000000,00000000 FP30=00000000,00000000
                               Single Floating Point Registers
FP0 =00000000 FP8 =00000000 FP16=00000000 FP24=00000000
FP1 =00000000 FP9 =00000000 FP17=00000000 FP25=00000000
FP2 =00000000 FP10=00000000 FP18=00000000 FP26=00000000
FP3 =00000000 FP11=00000000 FP19=00000000 FP27=00000000
FP4 =00000000 FP12=00000000 FP20=00000000 FP28=00000000
FP5 =00000000 FP13=00000000 FP21=00000000 FP29=00000000
FP6 =00000000 FP14=00000000 FP22=00000000 FP30=00000000
FP7 =00000000 FP15=00000000 FP23=00000000 FP31=00000000


Console
=========================
00: $s0 =      0x33     = 0d51 = 0x00000033
01: $s1 = 0x05  - 0x04  = 0d1 = 0x00000001
02: $s2 = 0x07 AND 0x06 = 0d6 = 0x00000006
03: $s3 = 0x04  OR 0x05 = 0d5 = 0x00000005
04: $s4 = 0x06  +  0x07 = 0d13 = 0x0000000D
05: BNE Not Taken: 0x33 = 0x33
06: BEQ Not Taken: 0x33 != 0x06
07: $s5 = 0x01 SLT 0x06 = 0d1 = 0x00000001
08: BEQ Taken: 0x33 = 0x33
0C: $s3 = 0x33  +  0x01 = 0d52 = 0x00000034
```

APPENDIX F: MIPS PIPELINED – VHDL CODE

```
--------------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
--
--
-- Filename: mips_pipe.vhd
-- Description: VHDL code to implment the structual design of
-- the MIPS pipelined processor.  The top-level file connects
-- all the behavioral units together to complete functional
-- processor.
--
-- Signals with a _p, _pp, or _ppp sufix designate the number
-- of pipeline registers that signals runs through.
-- e.g. _p = 1 pipeline register, _pp = 2 pipeline registers, etc.
--------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;

ENTITY mips_pipe IS

        --THESE SIGNAL ARE USED ONLY FOR THE SIMULATOR
        PORT(   --IF
                PC                      : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                --PC_Plus_4            : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                --Next_PC               : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Instruction_out         : OUT   STD_LOGIC_VECTOR (31 DOWNTO 0);
                --ID
                Read_Data1_out          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Read_Data2_out          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                --EX
                ALU_Input_1_out         : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                ALU_Input_2_out         : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                ALU_Result_out          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                --Add_Result_out        : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
                Branch_out              : OUT   STD_LOGIC;
                Branch_NE_out           : OUT   STD_LOGIC;
                Zero_out                : OUT   STD_LOGIC;
                --MEM
                MemRead_out             : OUT   STD_LOGIC;
                MemReadData_out         : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                MemWrite_out            : OUT   STD_LOGIC;
                Mem_Address_out         : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                MemWrite_Data_out       : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                --WB
                RegWrite_out            : OUT   STD_LOGIC;
                WriteRegister_out       : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
                RegWriteData_out        : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);

                --FORWARDING UNIT LINES
                --EXMEM_RegWrite_out    : OUT   STD_LOGIC;
                --EXMEM_ALU_Result_out  : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                --EXMEM_Register_Rd_out: OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
                --MEMWB_RegWrite_out    : OUT   STD_LOGIC;
                --MEMWB_Register_Rd_out: OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
                --MEMWB_Read_Data_out   : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                --IDEX_Register_Rs_out  : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
                --IDEX_Register_Rt_out  : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
                ForwardA_out            : OUT   STD_LOGIC_VECTOR (1 DOWNTO 0);
                ForwardB_out            : OUT   STD_LOGIC_VECTOR (1 DOWNTO 0);

                --HAZARD DETECTION LINES
                --IDEX_MemRead_out      : OUT   STD_LOGIC;
```

```
                --IDEX_Register_Rt_out : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
                --IFID_Register_Rs_out : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
                --IFID_Register_Rt_out : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
                STALL_out             : OUT   STD_LOGIC;
                --HDU_RegWrite_out     : OUT   STD_LOGIC;
                --HDU_MemWrite_out     : OUT   STD_LOGIC;

                --BRACH HAZARD
                IF_Flush_out          : OUT   STD_LOGIC;
                IF_ReadData1_out      : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                IF_ReadData2_out      : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                IF_SignExtend_out     : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                IF_Branch_out         : OUT   STD_LOGIC;
                IF_BranchNE_out       : OUT   STD_LOGIC;
                IF_PCPlus4_out        : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                IF_AddResult_out      : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                IF_Zero_out           : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Clock, Reset          : IN    STD_LOGIC);

--Signals used for UP2 Board Implemntation
--All signals here must be assigned to a pin on
--the FLEX10K70 device.
--PORT( D1_a, D1_b, D1_c, D1_d,
--      D1_e, D1_f, D1_g,D1_pb        : OUT   STD_LOGIC;
--      D2_a, D2_b, D2_c, D2_d,
--      D2_e, D2_f, D2_g,D2_pb        : OUT   STD_LOGIC;
--      Clock, Reset, PB              : IN    STD_LOGIC);

END mips_pipe;

ARCHITECTURE structure of mips_pipe IS

--Declare all components/units/modules used that
--makeup the MIPS Pipelined processor.
COMPONENT instrfetch
        PORT(   PC_Out                : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
                Instruction_p         : OUT   STD_LOGIC_VECTOR(31 DOWNTO 0);
                PC_plus_4_p           : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
                NXT_PC                : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
                Stall                 : IN    STD_LOGIC;
                --BRANCH HAZARD
                Read_Data_1           : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Read_Data_2           : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Sign_Extend           : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Branch                : IN    STD_LOGIC;
                Branch_NE             : IN    STD_LOGIC;
                PC_plus_4             : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
                IFFlush               : IN    STD_LOGIC;
                IFFlush_p             : OUT   STD_LOGIC;
                IFFlush_pp            : IN    STD_LOGIC;
                IFFlush_ppp           : OUT   STD_LOGIC;
                --OUTPUTS FOR branch hazard DEBUG--
                IF_ReadData1          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                IF_ReadData2          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                IF_SignExtend         : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                IF_Branch             : OUT   STD_LOGIC;
                IF_BranchNE           : OUT   STD_LOGIC;
                IF_PCPlus4            : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                IF_AddResult          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                IF_Zero               : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Clock, Reset          : IN    STD_LOGIC);
END COMPONENT;

COMPONENT operandfetch
        PORT (  Read_Data_1_p         : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Read_Data_2_p         : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Write_Data            : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                RegWrite_ppp          : IN    STD_LOGIC;
                RegWriteOut           : OUT   STD_LOGIC;
                Write_Address_pp      : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
                Write_Address_ppp     : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
```

```
                Read_Data_p           : IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
                MemtoReg_ppp          : IN   STD_LOGIC;
                ALU_Result_pp         : IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Instruction_p         : IN   STD_LOGIC_VECTOR (31 DOWNTO 0);
                Sign_Extend_p         : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
                Write_Address_0_p     : OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
                Write_Address_1_p     : OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
                Write_Address_2_p     : OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
                RegWriteData          : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
                Instruction_pp        : OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
                PC_plus_4_p           : IN   STD_LOGIC_VECTOR(7 DOWNTO 0);
                PC_plus_4_pp          : OUT  STD_LOGIC_VECTOR(7 DOWNTO 0);
                --HAZARD DETECTION UNIT
                IDEX_MemRead          : IN   STD_LOGIC;
                IDEX_Register_Rt      : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
                IFID_Register_Rs      : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
                IFID_Register_Rt      : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
                -------HDU Output lines------
                IDEXMemRead_out       : OUT  STD_LOGIC;
                IDEXRegister_Rt_out   : OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
                IFIDRegister_Rs_out   : OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
                IFIDRegister_Rt_out   : OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
                --BRANCH HAZARDS (CONTROL HAZARDS)
                Branch_p              : IN   STD_LOGIC;
                Branch_NE_p           : IN   STD_LOGIC;
                Add_Result_p          : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
                --IF_Flush            : OUT  STD_LOGIC;
                Branch_pp             : OUT  STD_LOGIC;
                Branch_NE_pp          : OUT  STD_LOGIC;
                Clock, Reset          : IN   STD_LOGIC);
END COMPONENT;

COMPONENT controlunit IS
PORT(           Opcode                : IN   STD_LOGIC_VECTOR (5 DOWNTO 0);
                RegDst_p              : OUT  STD_LOGIC;
                ALU_Op_p              : OUT  STD_LOGIC_VECTOR (1 DOWNTO 0);
                ALUSrc_p              : OUT  STD_LOGIC;
                MemWrite_p            : OUT  STD_LOGIC;
                Branch_p              : OUT  STD_LOGIC;
                Branch_NE_p           : OUT  STD_LOGIC;
                MemRead_p             : OUT  STD_LOGIC;
                MemtoReg_p            : OUT  STD_LOGIC;
                RegWrite_p            : OUT  STD_LOGIC;
                IF_Flush              : OUT  STD_LOGIC;
                --HAZARD DETECTION UNIT
                IDEX_MemRead          : IN   STD_LOGIC;
                IDEX_Register_Rt      : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
                IFID_Register_Rs      : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
                IFID_Register_Rt      : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
                Stall_out             : OUT  STD_LOGIC;
                Clock, Reset          : IN   STD_LOGIC);
END COMPONENT;

COMPONENT execution IS
        PORT(   Read_Data_1           : IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Read_Data_2           : IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Sign_Extend_p         : IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
                ALUSrc_p              : IN   STD_LOGIC;
                Zero_p                : OUT  STD_LOGIC;
                ALU_Result_p          : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
                Funct_field           : IN   STD_LOGIC_VECTOR (5 DOWNTO 0);
                ALU_Op_p              : IN   STD_LOGIC_VECTOR (1 DOWNTO 0);
                PC_plus_4_pp          : IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
                RegDst_p              : IN   STD_LOGIC;
                Write_Address_0_p     : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
                Write_Address_1_p     : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
                Write_Address_p       : OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
                MemtoReg_p            : IN   STD_LOGIC;
                MemtoReg_pp           : OUT  STD_LOGIC;
                Read_Data_2_pp        : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
                MemRead_p             : IN   STD_LOGIC;
```

```
               RegWrite_p            : IN   STD_LOGIC;
               MemRead_pp            : OUT  STD_LOGIC;
               RegWrite_pp           : OUT  STD_LOGIC;
               MemWrite_p            : IN   STD_LOGIC;
               MemWrite_pp           : OUT  STD_LOGIC;
               --FORWARDING UNIT SIGNALS
               EXMEM_RegWrite        : IN   STD_LOGIC;
               EXMEM_ALU_Result      : IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
               EXMEM_Register_Rd     : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
               MEMWB_RegWrite        : IN   STD_LOGIC;
               MEMWB_Register_Rd     : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
               MEMWB_Read_Data       : IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
               IDEX_Register_Rs      : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
               IDEX_Register_Rt      : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
               ALU1                  : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
               ALU2                  : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
               forwardA              : OUT  STD_LOGIC_VECTOR (1 DOWNTO 0);
               forwardB              : OUT  STD_LOGIC_VECTOR (1 DOWNTO 0);
               EXMEMRegWrite         : OUT  STD_LOGIC;
               EXMEMALU_Result       : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
               EXMEMRegister_Rd      : OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
               MEMWBRegWrite         : OUT  STD_LOGIC;
               MEMWBRegister_Rd      : OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
               MEMWBRead_Data        : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
               IDEXRegister_Rs       : OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
               IDEXRegister_Rt       : OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
               Clock, Reset          : IN   STD_LOGIC);
END COMPONENT;

COMPONENT datamemory IS
       PORT(   Read_Data_p           : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
               Address               : IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
               Write_Data            : IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
               Read_Data_2_ppp       : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
               MemRead_pp            : IN   STD_LOGIC;
               MemWrite_pp           : IN   STD_LOGIC;
               MemRead_ppp           : OUT  STD_LOGIC;
               MemWrite_ppp          : OUT  STD_LOGIC;
               MemtoReg_pp           : IN   STD_LOGIC;
               RegWrite_pp           : IN   STD_LOGIC;
               MemtoReg_ppp          : OUT  STD_LOGIC;
               RegWrite_ppp          : OUT  STD_LOGIC;
               ALU_Result_p          : IN   STD_LOGIC_VECTOR (7 DOWNTO 0);
               ALU_Result_pp         : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
               Write_Address_p       : IN   STD_LOGIC_VECTOR (4 DOWNTO 0);
               Write_Address_pp      : OUT  STD_LOGIC_VECTOR (4 DOWNTO 0);
               Reg_WriteData         : OUT  STD_LOGIC_VECTOR (7 DOWNTO 0);
               Clock, Reset          : IN   STD_LOGIC);
END COMPONENT;

COMPONENT debounce
       PORT (  Clock                 : IN   STD_LOGIC;
               PButton               : IN   STD_LOGIC;
               Pulse                 : OUT  STD_LOGIC);
END COMPONENT;

COMPONENT sevenseg_display IS
       PORT(   Digit1                : IN   STD_LOGIC_VECTOR (3 DOWNTO 0);
               Digit2                : IN   STD_LOGIC_VECTOR (3 DOWNTO 0);
               D1seg_a               : OUT  STD_LOGIC;
               D1seg_b               : OUT  STD_LOGIC;
               D1seg_c               : OUT  STD_LOGIC;
               D1seg_d               : OUT  STD_LOGIC;
               D1seg_e               : OUT  STD_LOGIC;
               D1seg_f               : OUT  STD_LOGIC;
               D1seg_g               : OUT  STD_LOGIC;
               D1pb                  : OUT  STD_LOGIC;
               D2seg_a               : OUT  STD_LOGIC;
               D2seg_b               : OUT  STD_LOGIC;
               D2seg_c               : OUT  STD_LOGIC;
               D2seg_d               : OUT  STD_LOGIC;
```

```
               D2seg_e               : OUT   STD_LOGIC;
               D2seg_f               : OUT   STD_LOGIC;
               D2seg_g               : OUT   STD_LOGIC;
               D2pb                  : OUT   STD_LOGIC;
               Clock, Reset          : IN    STD_LOGIC);
END COMPONENT;

--Signals used to connect VHDL Components
SIGNAL  ALU_Op_p               : STD_LOGIC_VECTOR (1 DOWNTO 0);
SIGNAL  Add_Result_p           : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  Add_Result_pp          : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  ALU_Result_p           : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  ALU_Result_pp          : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  ALU1                   : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL   ALU2                  : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  ALUSrc_p               : STD_LOGIC;
SIGNAL  Branch_p               : STD_LOGIC;
SIGNAL  Branch_pp              : STD_LOGIC;
SIGNAL  Branch_NE_p            : STD_LOGIC;
SIGNAL  Branch_NE_pp           : STD_LOGIC;
SIGNAL  forwardA               : STD_LOGIC_VECTOR (1 DOWNTO 0);
SIGNAL  forwardB               : STD_LOGIC_VECTOR (1 DOWNTO 0);
SIGNAL  Instruction            : STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL  Instruction_p          : STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL  Instruction_pp         : STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL  MemRead_p              : STD_LOGIC;
SIGNAL  MemRead_pp             : STD_LOGIC;
SIGNAL   MemRead_ppp           : STD_LOGIC;
SIGNAL  MemtoReg_p             : STD_LOGIC;
SIGNAL  MemtoReg_pp            : STD_LOGIC;
SIGNAL  MemtoReg_ppp           : STD_LOGIC;
SIGNAL  MemWrite_p             : STD_LOGIC;
SIGNAL  MemWrite_pp            : STD_LOGIC;
SIGNAL   MemWrite_ppp          : STD_LOGIC;
SIGNAL  PC_Out                 : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  PC_plus_4_p            : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  PC_plus_4_pp           : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  NXT_PC                 : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL  Read_Data_1_p          : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  Read_Data_2_p          : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  Read_Data_p            : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  Read_Data_2_pp         : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  Read_Data_2_ppp        : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  RegDst_p               : STD_LOGIC;
SIGNAL  RegWrite_p             : STD_LOGIC;
SIGNAL  RegWrite_pp            : STD_LOGIC;
SIGNAL  RegWrite_ppp           : STD_LOGIC;
SIGNAL   RegWriteOut           : STD_LOGIC;
SIGNAL   RegWriteData          : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL   Reg_WriteData         : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  Sign_Extend_p          : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  Write_Address_0_p      : STD_LOGIC_VECTOR (4 DOWNTO 0); --Rd
SIGNAL  Write_Address_1_p      : STD_LOGIC_VECTOR (4 DOWNTO 0); --Rt
SIGNAL  Write_Address_2_p      : STD_LOGIC_VECTOR (4 DOWNTO 0); --Rs
SIGNAL   Write_Address_p       : STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL  Write_Address_pp       : STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL  Write_Address_ppp      : STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL  Zero_p                 : STD_LOGIC;
---FORWARDING UNIT SIGNALS
SIGNAL  EXMEMRegWrite          : STD_LOGIC;
SIGNAL  EXMEMALU_Result        : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  EXMEMRegister_Rd       : STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL  MEMWBRegWrite          : STD_LOGIC;
SIGNAL  MEMWBRegister_Rd       : STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL  MEMWBRead_Data         : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  IDEXRegister_Rs        : STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL  IDEXRegister_Rt        : STD_LOGIC_VECTOR (4 DOWNTO 0);
--HAZARD DETECTION UNIT SIGNALS
SIGNAL  STALLout               : STD_LOGIC;
SIGNAL  IDEXMemRead_out        : STD_LOGIC;
SIGNAL  IDEXRegister_Rt_out    : STD_LOGIC_VECTOR (4 DOWNTO 0);
```

```
SIGNAL  IFIDRegister_Rs_out    : STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL  IFIDRegister_Rt_out    : STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL  StallInstruction       : STD_LOGIC_VECTOR (31 DOWNTO 0);
--BRANCH HAZARD
SIGNAL  IF_Flush               : STD_LOGIC;
SIGNAL  IF_Flush_p             : STD_LOGIC;
SIGNAL  IF_Flush_pp            : STD_LOGIC;
SIGNAL  IF_Flush_ppp           : STD_LOGIC;
SIGNAL  IF_ReadData1           : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  IF_ReadData2           : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  IF_SignExtend          : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  IF_Branch              : STD_LOGIC;
SIGNAL  IF_BranchNE            : STD_LOGIC;
SIGNAL  IF_PCPlus4             : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  IF_AddResult           : STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL  IF_Zero                : STD_LOGIC_VECTOR (7 DOWNTO 0);
--New Clock Signal used when PushButton is used
--to create a clock tick
SIGNAL  NClock                 : STD_LOGIC;

BEGIN

        --Signals to assign to output pins for SIMULATOR
        Instruction_out        <=      Instruction;
        PC                     <=      PC_Out;
        --PC_Plus_4            <=      PC_plus_4_p;
        --Next_PC              <=      NXT_PC;
        Read_Data1_out         <=      Read_Data_1_p;
        Read_Data2_out         <=      Read_Data_2_p;
        ALU_Input_1_out        <=      ALU1;
        ALU_Input_2_out        <=      ALU2;
        ALU_Result_out         <=      ALU_Result_p;
        --Add_Result_out       <=      Add_Result_p;
        Zero_out               <=      Zero_p;
        MemRead_out            <=      MemRead_ppp;
        MemReadData_out        <=      Read_Data_p WHEN MemRead_ppp = '1' ELSE "00000000";
        MemWrite_out           <=      MemWrite_ppp;
        Mem_Address_out        <=      ALU_Result_pp WHEN MemRead_ppp = '1'
                                         OR MemWrite_ppp = '1' ELSE "00000000";
        MemWrite_Data_out      <=      Read_Data_2_ppp WHEN MemWrite_ppp = '1' ELSE "00000000";
        RegWrite_out           <=      '0' WHEN Write_Address_ppp = "00000" ELSE RegWriteOut;
        WriteRegister_out      <=      Write_Address_ppp;
        RegWriteData_out       <=      RegWriteData;
        --FORWARDING UNIT LINES
        --EXMEM_RegWrite_out   <=      EXMEMRegWrite;
        --EXMEM_ALU_Result_out <=      EXMEMALU_Result;
        --EXMEM_Register_Rd_out<=      EXMEMRegister_Rd;
        --MEMWB_RegWrite_out   <=      MEMWBRegWrite;
        --MEMWB_Register_Rd_out<=      MEMWBRegister_Rd;
        --MEMWB_Read_Data_out  <=      MEMWBRead_Data;
        --IDEX_Register_Rs_out <=      IDEXRegister_Rs;
        --IDEX_Register_Rt_out <=      IDEXRegister_Rt;
        ForwardA_out           <=      forwardA;
        ForwardB_out           <=      forwardB;
        --HAZARD DETECTION LINES
        --IDEX_MemRead_out     <=      IDEXMemRead_out;
        --IDEX_Register_Rt_out <=      IDEXRegister_Rt_out;
        --IFID_Register_Rs_out <=      IFIDRegister_Rs_out;
        --IFID_Register_Rt_out <=      IFIDRegister_Rt_out;
        STALL_out              <=      STALLout;
        StallInstruction       <=      Instruction WHEN STALLout = '0' ELSE Instruction_pp;
        --HDU_RegWrite_out     <=      RegWrite_p;
        --HDU_MemWrite_out     <=      MemWrite_p;
        --BRANCH HAZARD
        IF_Flush_out           <=      IF_Flush_p;
        Branch_out             <=      Branch_p;
        Branch_NE_out          <=      Branch_NE_p;
        IF_ReadData1_out       <=      IF_ReadData1;
        IF_ReadData2_out       <=      IF_ReadData2;
        IF_SignExtend_out      <=      IF_SignExtend;
        IF_Branch_out          <=      IF_Branch;
```

```
IF_BranchNE_out        <=     IF_BranchNE;
IF_PCPlus4_out         <=     IF_PCPlus4;
IF_AddResult_out       <=     IF_AddResult;
IF_Zero_out            <=     IF_Zero;

--Connect the MIPS Components
IFE : instrfetch PORT MAP (
                       PC_Out                => PC_Out,
                       Instruction_p         => Instruction,
                       PC_plus_4_p           => PC_plus_4_p,
                       NXT_PC                => NXT_PC,
                       Stall                 => STALLout,
                       --BRANCH HAZARD
                       Read_Data_1           => Read_Data_1_p,
                       Read_Data_2           => Read_Data_2_p,
                       Sign_Extend           => Sign_Extend_p,
                       Branch                => Branch_p,
                       Branch_NE             => Branch_NE_p,
                       PC_plus_4             => PC_plus_4_p,
                       IFFlush               => IF_Flush,
                       IFFlush_p             => IF_Flush_p,
                       IFFlush_pp            => IF_Flush_p,
                       IFFlush_ppp           => IF_Flush_ppp,
                       IF_ReadData1          => IF_ReadData1,
                       IF_ReadData2          => IF_ReadData2,
                       IF_SignExtend         => IF_SignExtend,
                       IF_Branch             => IF_Branch,
                       IF_BranchNE           => IF_BranchNE,
                       IF_PCPlus4            => IF_PCPlus4,
                       IF_AddResult          => IF_AddResult,
                       IF_Zero               => IF_Zero,
                       Clock                 => Clock,
                       Reset                 => Reset );

ID : operandfetch PORT MAP (
                       Read_Data_1_p         => Read_Data_1_p,
                       Read_Data_2_p         => Read_Data_2_p,
                       Write_Data            => Reg_WriteData,
                       RegWrite_ppp          => RegWrite_ppp,
                       RegWriteOut           => RegWriteOut,
                       Write_Address_pp      => Write_Address_pp,
                       Write_Address_ppp     => Write_Address_ppp,
                       Read_Data_p           => Read_Data_p,
                       MemtoReg_ppp          => MemtoReg_ppp,
                       ALU_Result_pp         => ALU_Result_pp,
                       Instruction_p         => StallInstruction,
                       Sign_Extend_p         => Sign_Extend_p,
                       Write_Address_0_p     => Write_Address_0_p,
                       Write_Address_1_p     => Write_Address_1_p,
                       Write_Address_2_p     => Write_Address_2_p,
                       RegWriteData          => RegWriteData,
                       Instruction_pp        => Instruction_pp,
                       PC_plus_4_p           => PC_plus_4_p,
                       PC_plus_4_pp          => PC_plus_4_pp,
                       --HAZARD DETECTION UNIT
                       IDEX_MemRead          => MemRead_p,
                       IDEX_Register_Rt      => Write_Address_0_p,
                       IFID_Register_Rs      => Instruction (25 DOWNTO 21),  --Rs
                       IFID_Register_Rt      => Instruction (20 DOWNTO 16),  --Rt
                       -------HDU Output lines------
                       IDEXMemRead_out       => IDEXMemRead_out,
                       IDEXRegister_Rt_out   => IDEXRegister_Rt_out,
                       IFIDRegister_Rs_out   => IFIDRegister_Rs_out,
                       IFIDRegister_Rt_out   => IFIDRegister_Rt_out,
                       --BRANCH HAZARDS
                       Branch_p              => Branch_p,
                       Branch_NE_p           => Branch_NE_p,
                       Branch_pp             => Branch_pp,
                       Branch_NE_pp          => Branch_NE_pp,
                       Clock                 => Clock,
                       Reset                 => Reset );
```

```
CTRL : controlunit PORT MAP (
                      Opcode                => Instruction (31 DOWNTO 26),
                      RegDst_p              => RegDst_p,
                      ALU_Op_p              => ALU_Op_p,
                      ALUSrc_p              => ALUSrc_p,
                      MemWrite_p            => MemWrite_p,
                      Branch_p              => Branch_p,
                      Branch_NE_p           => Branch_NE_p,
                      MemRead_p             => MemRead_p,
                      MemtoReg_p            => MemtoReg_p,
                      RegWrite_p            => RegWrite_p,
                      IF_Flush              => IF_Flush,

                      --HAZARD DETECTION UNIT
                      IDEX_MemRead          => MemRead_p,
                      IDEX_Register_Rt      => Write_Address_0_p,
                      IFID_Register_Rs      => Instruction (25 DOWNTO 21),  --Rs
                      IFID_Register_Rt      => Instruction (20 DOWNTO 16),  --Rt
                      Stall_out             => STALLout,

                      Clock                 => Clock,
                      Reset                 => Reset);

EX : execution PORT MAP (
                Read_Data_1                 => Read_Data_1_p,
                Read_Data_2                 => Read_Data_2_p,
                Sign_Extend_p               => Sign_Extend_p,
                ALUSrc_p                    => ALUSrc_p,
                Zero_p                      => Zero_p,
                ALU_Result_p                => ALU_Result_p,
                Funct_field                 => Instruction_pp (5 DOWNTO 0),
                ALU_Op_p                    => ALU_Op_p,
                PC_plus_4_pp                => PC_plus_4_pp,
                RegDst_p                    => RegDst_p,
                Write_Address_0_p           => Write_Address_0_p,
                Write_Address_1_p           => Write_Address_1_p,
                Write_Address_p             => Write_Address_p,
                MemtoReg_p                  => MemtoReg_p,
                MemtoReg_pp                 => MemtoReg_pp,
                Read_Data_2_pp              => Read_Data_2_pp,
                MemRead_p                   => MemRead_p,
                RegWrite_p                  => RegWrite_p,
                MemRead_pp                  => MemRead_pp,
                RegWrite_pp                 => RegWrite_pp,
                MemWrite_p                  => MemWrite_p,
                MemWrite_pp                 => MemWrite_pp,
                EXMEM_RegWrite              => RegWrite_pp,
                EXMEM_ALU_Result            => ALU_Result_p,
                EXMEM_Register_Rd           => Write_Address_p,
                MEMWB_RegWrite              => RegWrite_ppp,
                MEMWB_Register_Rd           => Write_Address_pp,
                MEMWB_Read_Data             => Reg_WriteData,
                IDEX_Register_Rs            => Write_Address_2_p,
                IDEX_Register_Rt            => Write_Address_0_p,
                ALU1                        => ALU1,
                ALU2                        => ALU2,
                forwardA                    => forwardA,
                forwardB                    => forwardB,
                EXMEMRegWrite               => EXMEMRegWrite,
                EXMEMALU_Result             => EXMEMALU_Result,
                EXMEMRegister_Rd            => EXMEMRegister_Rd,
                MEMWBRegWrite               => MEMWBRegWrite,
                MEMWBRegister_Rd            => MEMWBRegister_Rd,
                MEMWBRead_Data              => MEMWBRead_Data,
                IDEXRegister_Rs             => IDEXRegister_Rs,
                IDEXRegister_Rt             => IDEXRegister_Rt,
                Clock                       => Clock,
                Reset                       => Reset );

MEM : datamemory PORT MAP (
```

```
                        Read_Data_p                 => Read_Data_p,
                        Address                     => ALU_Result_p,
                        Write_Data                  => Read_Data_2_pp,
                        Read_Data_2_ppp             => Read_Data_2_ppp,
                        MemRead_pp                  => MemRead_pp,
                        MemWrite_pp                 => MemWrite_pp,
                        MemRead_ppp                 => MemRead_ppp,
                        MemWrite_ppp                => MemWrite_ppp,
                        MemtoReg_pp                 => MemtoReg_pp,
                        RegWrite_pp                 => RegWrite_pp,
                        MemtoReg_ppp                => MemtoReg_ppp,
                        RegWrite_ppp                => RegWrite_ppp,
                        ALU_Result_p                => ALU_Result_p,
                        ALU_Result_pp               => ALU_Result_pp,
                        Write_Address_p             => Write_Address_p,
                        Write_Address_pp            => Write_Address_pp,
                        Reg_WriteData               => Reg_WriteData,
                        Clock                       => Clock,
                        Reset                       => Reset);

--      NCLK: Debounce PORT MAP(
--                      Clock                       => Clock,
--                      PButton                     => PB,
--                      Pulse                       => NClock);

--      SSD: sevenseg_display PORT MAP(
--                      Digit1                      => PC_Out (7 DOWNTO 4),
--                      Digit2                      => PC_Out (3 DOWNTO 0),
--                      D1seg_a                     => D1_a,
--                      D1seg_b                     => D1_b,
--                      D1seg_c                     => D1_c,
--                      D1seg_d                     => D1_d,
--                      D1seg_e                     => D1_e,
--                      D1seg_f                     => D1_f,
--                      D1seg_g                     => D1_g,
--                      D1pb                        => D1_pb,
--                      D2seg_a                     => D2_a,
--                      D2seg_b                     => D2_b,
--                      D2seg_c                     => D2_c,
--                      D2seg_d                     => D2_d,
--                      D2seg_e                     => D2_e,
--                      D2seg_f                     => D2_f,
--                      D2seg_g                     => D2_g,
--                      D2pb                        => D2_pb,
--                      Clock                       => NClock,
--                      Reset                       => Reset);

END structure;


-- ---------------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
--
-- Filename: instr_fetch.vhd
-- Description: VHDL code to implment the Instruction Fetch unit
-- of the MIPS Pipelined processor as seen in Chapter #6 of
-- Patterson and Hennessy book.  This file involves the use of the
-- LPM Components (LPM_ROM) to declare the Instruction Memory
-- as a read only memory (ROM).  See MAX+PLUS II Help on
-- "Implementing RAM & ROM (VHDL)" for details.
--
-- Signals with a _p, _pp, or _ppp sufix designate the number
-- of pipeline registers that signals runs through.
-- e.g. _p = 1 pipeline register, _pp = 2 pipeline registers, etc.
-- ---------------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
```

```
 USE IEEE.STD_LOGIC_ARITH.ALL;
 USE IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY LPM;
 USE LPM.LPM_COMPONENTS.ALL;

ENTITY instrfetch IS
	PORT(	--Program Counter
		PC_Out			: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		--Instruction Memory
		Instruction_p		: OUT	STD_LOGIC_VECTOR (31 DOWNTO 0);
		--PC + 4
		PC_plus_4_p		: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		NXT_PC			: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		--Hazard Detection Unit
		Stall			: IN	STD_LOGIC;
		--BRANCH HAZARD
		Read_Data_1		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		Read_Data_2		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		Sign_Extend		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		Branch			: IN	STD_LOGIC;
		Branch_NE		: IN	STD_LOGIC;
		PC_plus_4		: IN	STD_LOGIC_VECTOR (7 DOWNTO 0);
		IFFlush			: IN	STD_LOGIC;
		IFFlush_p		: OUT	STD_LOGIC;
		IFFlush_pp		: IN	STD_LOGIC;
		IFFlush_ppp		: OUT	STD_LOGIC; --Assert if Flush used!
		--OUTPUTS FOR branch hazard DEBUG--
		IF_ReadData1		: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		IF_ReadData2		: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		IF_SignExtend		: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		IF_Branch		: OUT	STD_LOGIC;
		IF_BranchNE		: OUT	STD_LOGIC;
		IF_PCPlus4		: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		IF_AddResult		: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		IF_Zero			: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		Clock, Reset		: IN	STD_LOGIC);
END instrfetch;

ARCHITECTURE behavior OF instrfetch IS
	SIGNAL PC			: STD_LOGIC_VECTOR (9 DOWNTO 0);
	SIGNAL PCplus4			: STD_LOGIC_VECTOR (9 DOWNTO 0);
	SIGNAL Next_PC			: STD_LOGIC_VECTOR (7 DOWNTO 0);
	SIGNAL Instruction		: STD_LOGIC_VECTOR (31 DOWNTO 0);
	SIGNAL zero			: STD_LOGIC_VECTOR (7 DOWNTO 0);
	SIGNAL add_result		: STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN

--Instruction Memory as ROM
Instr_Memory: LPM_ROM

	GENERIC MAP(
		LPM_WIDTH		=> 32,
		LPM_WIDTHAD		=> 8,
		-- *.mif FILE used to initialize memory values
		--LPM_FILE		=> "instruction_memory.mif",
		--LPM_FILE		=> "pipelining_example.mif",
		LPM_FILE		=> "dependencies_example.mif",
		--LPM_FILE		=> "forwarding_example.mif",
		--LPM_FILE		=> "hazard_detection_example.mif",
		--LPM_FILE		=> "branch_hazard_detection_example.mif",
		LPM_OUTDATA		=> "UNREGISTERED",
		LPM_ADDRESS_CONTROL	=> "UNREGISTERED")

	PORT MAP (
		--Bits (9 DOWNTO 2) used to word-address instructions
		-- e.g. +1 not +4(byte-addressed)
		address		=> PC (9 DOWNTO 2),
		--Output of Instruction Memory is 32-bit instruction
		q		=> Instruction );
```

```
		-- Copy Output Signals
		PC_Out		<= PC (7 DOWNTO 0);
		NXT_PC		<= Next_PC;

		-- Adder to increment PC by 4 by shifting bits
		PCplus4 (9 DOWNTO 2)  <= PC (9 DOWNTO 2) + 1;
		PCplus4 (1 DOWNTO 0)  <= "00";

		--New Branch Compare - compare Read_Data_1 XOR Read_Data_2:
		zero <= Read_Data_1 (7 DOWNTO 0) XOR Read_Data_2 (7 DOWNTO 0);
		add_result <= PC_plus_4 (7 DOWNTO 2) + Sign_Extend (7 DOWNTO 0);

		-- Mux to select Branch Address or PC + 4 or Jump  (PCSrc Mux)
		Next_PC <= add_result WHEN ( IFFlush_pp = '1' AND ( zero = "00000000" )
				AND ( Branch = '1') ) OR ( IFFlush_pp = '1'
				AND ( zero /= "00000000" ) AND ( Branch_NE = '1') )
				ELSE  PCplus4 (9 DOWNTO 2);

		IF_ReadData1   <= Read_Data_1;
		IF_ReadData2   <= Read_Data_2;
		IF_SignExtend  <= Sign_Extend;
		IF_Branch      <= Branch;
		IF_BranchNE    <= Branch_NE;
		IF_PCPlus4     <= PC_plus_4;
		IF_AddResult   <= add_result;
		IF_Zero        <= zero;
		IFFlush_p      <= IFFlush;

		--Branch Flushing Unit to flush IF/ID Pipeline Register
		Instruction_p <= "00000000000000000000000000000000"
				WHEN (Reset = '1')
				OR ( IFFlush_pp = '1' AND ( zero = "00000000" )
				AND ( Branch = '1') ) OR ( IFFlush_pp = '1'
				AND ( zero /= "00000000" ) AND ( Branch_NE = '1') )
				ELSE Instruction;
	PROCESS
	BEGIN
		WAIT UNTIL ( Clock'EVENT ) AND ( Clock = '1' );
		IF Reset = '1' THEN
			PC		<= "0000000000";
			PC_plus_4_p	<= "00000000";
			IFFlush_ppp	<= '0';
		ELSE
			IF (Stall = '1')  THEN
				--AVOID Writing any signals for Load-Use Data Hazard
			ELSE
				PC (9 DOWNTO 2)		<= Next_PC (7 DOWNTO 0);
				PC_plus_4_p		<= PCplus4 (7 DOWNTO 0);

				--Assert if Flush Signal Used!!!
				IF ( IFFlush_pp = '1' AND ( zero = "00000000" )
					AND ( Branch = '1') ) OR
					( IFFlush_pp = '1' AND ( zero /= "00000000" )
					AND ( Branch_NE = '1') ) THEN
						IFFlush_ppp <= '1';
					ELSE
						IFFlush_ppp <= '0';
					END IF;
				END IF;
			END IF;
	END PROCESS;
END behavior;


-----------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
--
```

```
--
-- Filename: operand_fetch.vhd
-- Description: VHDL code to implment the Operand Fetch unit
-- of the MIPS pipelined processor as seen in Chapter #6 of
-- Patterson and Hennessy book.
--
-- Signals with a _p, _pp, or _ppp sufix designate the number
-- of pipeline registers that signals runs through.
-- e.g. _p = 1 pipeline register, _pp = 2 pipeline registers, etc.
----------------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_ARITH.ALL;
 USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY operandfetch IS
       PORT (
              --Registers / MUX
              Read_Data_1_p          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
              Read_Data_2_p          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
              --Data Memory / MUX  --> WRITEBACK (LW/R-format)
              RegWrite_ppp           : IN    STD_LOGIC;
              RegWriteOut            : OUT   STD_LOGIC;
              Write_Address_pp       : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
              Write_Address_ppp      : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
              Read_Data_p            : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
              MemtoReg_ppp           : IN    STD_LOGIC;
              ALU_Result_pp          : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
              Write_Data             : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
              --Misc
              Instruction_p          : IN    STD_LOGIC_VECTOR (31 DOWNTO 0);
              Sign_Extend_p          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
              Write_Address_0_p      : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
              Write_Address_1_p      : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
              Write_Address_2_p      : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
              RegWriteData           : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
              Instruction_pp         : OUT   STD_LOGIC_VECTOR(31 DOWNTO 0);
              PC_plus_4_p            : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
              PC_plus_4_pp           : OUT   STD_LOGIC_VECTOR(7 DOWNTO 0);
              -------HAZARD DETECTION UNIT-------
              IDEX_MemRead           : IN    STD_LOGIC;
              IDEX_Register_Rt       : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
              IFID_Register_Rs       : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
              IFID_Register_Rt       : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
              -------HDU Output lines-----
              IDEXMemRead_out        : OUT   STD_LOGIC;
              IDEXRegister_Rt_out    : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
              IFIDRegister_Rs_out    : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
              IFIDRegister_Rt_out    : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
              ------BRANCH HAZARDS (CONTROL HAZARDS)
              Branch_p               : IN    STD_LOGIC;
              Branch_NE_p            : IN    STD_LOGIC;
              --Add_Result_p         : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
              Branch_pp              : OUT   STD_LOGIC;
              Branch_NE_pp           : OUT   STD_LOGIC;
              --Misc.
              Clock, Reset           : IN    STD_LOGIC);
END operandfetch;

ARCHITECTURE behavior OF operandfetch IS
--Declare Register File as a one-dimensional array
--Thirty-two Registers each 8-bits wide
TYPE register_file IS ARRAY (0 TO 31) OF STD_LOGIC_VECTOR (7 DOWNTO 0);

       SIGNAL register_array         : register_file;
       SIGNAL read_register_address1 : STD_LOGIC_VECTOR (4 DOWNTO 0);
       SIGNAL read_register_address2 : STD_LOGIC_VECTOR (4 DOWNTO 0);
       SIGNAL instruction_15_0       : STD_LOGIC_VECTOR (15 DOWNTO 0);
       --PIPELINED SIGNAL
       SIGNAL write_register_address0: STD_LOGIC_VECTOR (4 DOWNTO 0);
       SIGNAL write_register_address1: STD_LOGIC_VECTOR (4 DOWNTO 0);
```

```
        SIGNAL write_register_address2: STD_LOGIC_VECTOR (4 DOWNTO 0);
        SIGNAL sign_extend            : STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL read_data_1            : STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL read_data_2            : STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL writedata              : STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL instruction            : STD_LOGIC_VECTOR (31 DOWNTO 0);
        SIGNAL write_address          : STD_LOGIC_VECTOR (4 DOWNTO 0);
        SIGNAL stall                  : STD_LOGIC;
        SIGNAL Branch_Add, ADD_Result : STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL ifflush                : STD_LOGIC;
        SIGNAL bne                    : STD_LOGIC;
        SIGNAL result                 : STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN

        --Copy Instruction bits to signals
        read_register_address1  <= Instruction_p (25 DOWNTO 21); --Rs
        read_register_address2  <= Instruction_p (20 DOWNTO 16); --Rt
        write_register_address0 <= Instruction_p (20 DOWNTO 16); --Rt
        write_register_address1 <= Instruction_p (15 DOWNTO 11); --Rd
        write_register_address2 <= Instruction_p (25 DOWNTO 21); --Rs
        instruction_15_0       <= Instruction_p (15 DOWNTO 0);
        bne                     <= Branch_NE_p;

        --Register File: Read_Data_1 Output
        read_data_1 <= register_array(CONV_INTEGER(read_register_address1 (4 DOWNTO 0)));

        --Register File: Read_Data_2 Output
        read_data_2 <= register_array(CONV_INTEGER(read_register_address2 (4 DOWNTO 0)));

        --Sign Extend
        --NOTE: Due to 8-bit data width design
        --No sign extension is NEEDED
        sign_extend <= instruction_15_0 (7 DOWNTO 0);

        --Register File: MUX to select Write Register Data
        writedata <= Read_Data_p WHEN MemtoReg_ppp = '1' ELSE ALU_Result_pp;

        --Copy Instruction
        instruction <= Instruction_p;

        --Process to ensure writes happen on 1st half of clock cycle
        PROCESS (Clock, Reset)
        BEGIN
                IF (Reset = '1') THEN
                        --Reset Registers own Register Number
                        FOR i IN 0 TO 31 LOOP
                                register_array(i) <= CONV_STD_LOGIC_VECTOR(i,8);
                        END LOOP;
                ELSIF (Clock'EVENT AND Clock='0') THEN
                        --Write Register File if RegWrite signal asserted
                        IF ((RegWrite_ppp = '1') AND (Write_Address_pp /= "00000")) THEN
                                register_array(CONV_INTEGER(Write_Address_pp (4 DOWNTO 0)))
                                <= writedata;
                        END IF;
                END IF;
        END PROCESS;

        --Process to ensure read happen on 2nd half of clock cycle
        PROCESS
        BEGIN
                WAIT UNTIL ( Clock'EVENT AND Clock = '1');
                        IF Reset = '1' THEN
                                Read_Data_1_p         <= "00000000";
                                Read_Data_2_p         <= "00000000";
                                Sign_Extend_p         <= "00000000";
                                Write_Address_0_p     <= "00000";
                                Write_Address_1_p     <= "00000";
                                Write_Address_2_p     <= "00000";
                                RegWriteData          <= "00000000";
                                Instruction_pp        <= "00000000000000000000000000000000";
```

```
                        Write_Address_ppp     <= "00000";
                        RegWriteOut           <= '0';
                        PC_plus_4_pp          <= "00000000";
                ELSE
                        Read_Data_1_p         <= read_data_1;
                        Read_Data_2_p         <= read_data_2;
                        Sign_Extend_p         <= sign_extend;
                        Write_Address_0_p     <= write_register_address0;
                        Write_Address_1_p     <= write_register_address1;
                        Write_Address_2_p     <= write_register_address2;
                        RegWriteData          <= writedata;
                        Instruction_pp        <= Instruction_p;
                        Write_Address_ppp     <= Write_Address_pp;
                        RegWriteOut           <= RegWrite_ppp;
                        PC_plus_4_pp          <= PC_plus_4_p;
                        ----HAZARD DETECTION UNIT OUTPUT LINES----
                        IDEXMemRead_out       <= IDEX_MemRead;
                        IDEXRegister_Rt_out   <= IDEX_Register_Rt;
                        IFIDRegister_Rs_out   <= IFID_Register_Rs;
                        IFIDRegister_Rt_out   <= IFID_Register_Rt;
                        ---BRANCH HAZARD
                        Branch_pp             <= Branch_p;
                        Branch_NE_pp          <= Branch_NE_p;
                END IF;
        END PROCESS;
END behavior;


--------------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
--
-- Filename: control_unit.vhd
-- Description: VHDL code to implement the Control Unit
-- of the MIPS pipelined processor as seen in Chapter #6 of
-- Patterson and Hennessy book.
--
-- Signals with a _p, _pp, or _ppp sufix designate the number
-- of pipeline registers that signals runs through.
-- e.g. _p = 1 pipeline register, _pp = 2 pipeline registers, etc.
--------------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_ARITH.ALL;
 USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY controlunit IS
PORT(
        SIGNAL  Opcode          : IN    STD_LOGIC_VECTOR (5 DOWNTO 0);
        SIGNAL  RegDst_p        : OUT   STD_LOGIC;
        SIGNAL  ALU_Op_p        : OUT   STD_LOGIC_VECTOR (1 DOWNTO 0);
        SIGNAL  ALUSrc_p        : OUT   STD_LOGIC;
        SIGNAL  MemWrite_p      : OUT   STD_LOGIC;
        SIGNAL  Branch_p        : OUT   STD_LOGIC;
        SIGNAL  Branch_NE_p     : OUT   STD_LOGIC;
        SIGNAL  MemRead_p       : OUT   STD_LOGIC;
        SIGNAL  MemtoReg_p      : OUT   STD_LOGIC;
        SIGNAL  RegWrite_p      : OUT   STD_LOGIC;
        SIGNAL  IF_Flush        : OUT   STD_LOGIC;

        --HAZARD DETECTION UNIT
        IDEX_MemRead            : IN    STD_LOGIC;
        IDEX_Register_Rt        : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
        IFID_Register_Rs        : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
        IFID_Register_Rt        : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
        SIGNAL  Stall_out       : OUT   STD_LOGIC;
        SIGNAL  Clock, Reset    : IN    STD_LOGIC);
END controlunit;
```

```
ARCHITECTURE behavior OF controlunit IS
	SIGNAL R_format, LW, SW, BEQ, BNE, ADDI	: STD_LOGIC;
	SIGNAL Opcode_Out				: STD_LOGIC_VECTOR (5 DOWNTO 0);
	SIGNAL ifflush					: STD_LOGIC;
	--PIPELINED SIGNAL
	--EX
	SIGNAL RegDst, ALU_Op0, ALU_Op1, ALUSrc	: STD_LOGIC;
	--MEM
	SIGNAL Branch, Branch_NE, MemWrite, MemRead	: STD_LOGIC;
	--WB
	SIGNAL MemtoReg, RegWrite			: STD_LOGIC;
	--HDU
	SIGNAL stall					: STD_LOGIC;

BEGIN

	--Decode the Instruction OPCode to determine type
	--and set all corresponding control signals &
	--ALUOP function signals.
	R_format	<=	'1' WHEN Opcode = "000000" ELSE '0';
	LW		<=	'1' WHEN Opcode = "100011" ELSE '0';
	SW		<=	'1' WHEN Opcode = "101011" ELSE '0';
	BEQ		<=	'1' WHEN Opcode = "000100" ELSE '0';
	BNE		<=	'1' WHEN Opcode = "000101" ELSE '0';
	ADDI		<=	'1' WHEN Opcode = "001000" ELSE '0';
	--EX
	RegDst		<=	R_format;
	ALU_Op1		<=	R_format;
	ALU_Op0		<=	BEQ OR BNE;
	ALUSrc		<=	LW OR SW OR ADDI;
	--MEM
	Branch		<=	BEQ;
	Branch_NE	<=	BNE;
	ifflush		<=	BEQ OR BNE;
	MemRead		<=	LW;
	MemWrite	<=	SW;
	--WB
	MemtoReg	<=	LW;
	RegWrite	<=	R_format OR LW OR ADDI;

	--HAZARD DETECTION UNIT
	PROCESS
	BEGIN
		--DETECT A LOAD/USE HAZARD  e.g. LW $2, 20($1) followed by ADD $4, $2, $1
		IF ( (IDEX_MemRead = '1') AND
			((IDEX_Register_Rt = IFID_Register_Rs) OR
			 (IDEX_Register_Rt = IFID_Register_Rt))) THEN
			stall <= '1';  --LOAD/USE HAZARD
		ELSE
			stall <= '0';  --NO HAZARD
		END IF;
	END PROCESS;


	PROCESS
		BEGIN
			WAIT UNTIL ( Clock'EVENT ) AND ( Clock = '1' );
			IF (Reset = '1' OR stall = '1') THEN
				--WB
				MemtoReg_p	<= '0';
				RegWrite_p	<= '0';
				--Mem
				MemWrite_p	<= '0';
				MemRead_p	<= '0';
				Branch_p	<= '0';
				Branch_NE_p	<= '0';
				--EX
				RegDst_p	<= '0';
				ALU_Op_p(1)	<= '0';
				ALU_Op_p(0)	<= '0';
```

```
                        ALUSrc_p        <= '0';
                        IF_Flush        <= '0';
                        --HDU
                        Stall_out       <= stall;
                ELSE
                        --WB
                        MemtoReg_p      <= MemtoReg;
                        RegWrite_p      <= RegWrite;
                        --Mem
                        MemWrite_p      <= MemWrite;
                        MemRead_p       <= MemRead;
                        Branch_p        <= Branch;
                        Branch_NE_p     <= Branch_NE;
                        --EX
                        RegDst_p        <= RegDst;
                        ALU_Op_p(1)     <= ALU_Op1;
                        ALU_Op_p(0)     <= ALU_Op0;
                        ALUSrc_p        <= ALUSrc;
                        IF_Flush        <= ifflush;
                        --HDU
                        Stall_out       <= stall;
                END IF;
        END PROCESS;
END behavior;

--------------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
--
-- Filename: execution_unit.vhd
-- Description: VHDL code to implment the Execution Unit
-- of the MIPS pipelined processor as seen in Chapter #6 of
-- Patterson and Hennessy book.
--
-- Signals with a _p, _pp, or _ppp sufix designate the number
-- of pipeline registers that signals runs through.
-- e.g. _p = 1 pipeline register, _pp = 2 pipeline registers, etc.
--------------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_ARITH.ALL;
 USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY execution IS
        PORT(
                --ALU
                Read_Data_1             : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Read_Data_2             : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Sign_Extend_p           : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                ALUSrc_p                : IN    STD_LOGIC;
                Zero_p                  : OUT   STD_LOGIC;
                ALU_Result_p            : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                --ALU Control
                Funct_field             : IN    STD_LOGIC_VECTOR (5 DOWNTO 0);
                ALU_Op_p                : IN    STD_LOGIC_VECTOR (1 DOWNTO 0);
                PC_plus_4_pp            : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                --Register File - Write Address
                RegDst_p                : IN    STD_LOGIC;
                Write_Address_0_p       : IN    STD_LOGIC_VECTOR (4 DOWNTO 0); --Rt
                Write_Address_1_p       : IN    STD_LOGIC_VECTOR (4 DOWNTO 0); --Rd
                Write_Address_p         : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
                --MISC/PIPELINED SIGNALS
                MemtoReg_p              : IN    STD_LOGIC;
                MemtoReg_pp             : OUT   STD_LOGIC;
                Read_Data_2_pp          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                MemRead_p               : IN    STD_LOGIC;
                RegWrite_p              : IN    STD_LOGIC;
                MemRead_pp              : OUT   STD_LOGIC;
```

```
                RegWrite_pp             : OUT   STD_LOGIC;
                MemWrite_p              : IN    STD_LOGIC;
                MemWrite_pp             : OUT   STD_LOGIC;
                --FORWARDING UNIT SIGNALS
                EXMEM_RegWrite          : IN    STD_LOGIC;
                EXMEM_ALU_Result        : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                EXMEM_Register_Rd       : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
                MEMWB_RegWrite          : IN    STD_LOGIC;
                MEMWB_Register_Rd       : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
                MEMWB_Read_Data         : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                IDEX_Register_Rs        : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
                IDEX_Register_Rt        : IN    STD_LOGIC_VECTOR (4 DOWNTO 0);
                ALU1                    : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                ALU2                    : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                forwardA                : OUT   STD_LOGIC_VECTOR (1 DOWNTO 0);
                forwardB                : OUT   STD_LOGIC_VECTOR (1 DOWNTO 0);
                EXMEMRegWrite           : OUT   STD_LOGIC;
                EXMEMALU_Result         : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                EXMEMRegister_Rd        : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
                MEMWBRegWrite           : OUT   STD_LOGIC;
                MEMWBRegister_Rd        : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
                MEMWBRead_Data          : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                IDEXRegister_Rs         : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
                IDEXRegister_Rt         : OUT   STD_LOGIC_VECTOR (4 DOWNTO 0);
                ----Misc
                Clock, Reset            : IN    STD_LOGIC);
END execution;

ARCHITECTURE behavior of execution IS

        SIGNAL A_input, B_input, Binput       : STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL ALU_output                     : STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL ALU_Control                    : STD_LOGIC_VECTOR (2 DOWNTO 0);
        --PIPELINED SIGNALS
        SIGNAL ALU_Result                     : STD_LOGIC_VECTOR (7 DOWNTO 0);
        SIGNAL Zero                           : STD_LOGIC;
        SIGNAL Write_Address                  : STD_LOGIC_VECTOR (4 DOWNTO 0);
        --FORWARDING SIGNALS
        SIGNAL Forward_A, Forward_B           : STD_LOGIC_VECTOR (1 DOWNTO 0);

BEGIN

        --FORWARDING UNIT PART II
        PROCESS --(EXMEM_RegWrite, EXMEM_ALU_Result, EXMEM_Register_Rd, MEMWB_RegWrite,
                -- MEMWB_Register_Rd, MEMWB_Read_Data, IDEX_Register_Rs, IDEX_Register_Rt)
        BEGIN
                -- Forward A
                IF ( (MEMWB_RegWrite = '1') AND
                        (MEMWB_Register_Rd /= "00000") AND
                        (EXMEM_Register_Rd /= IDEX_Register_Rs) AND
                        (MEMWB_Register_Rd = IDEX_Register_Rs) ) THEN
                        Forward_A <= "01";     --MEM HAZARD
                ELSIF ( (EXMEM_RegWrite = '1') AND
                        (EXMEM_Register_Rd /= "00000") AND
                        (EXMEM_Register_Rd = IDEX_Register_Rs) ) THEN
                        Forward_A <= "10";     --EX HAZARD
                ELSE
                        Forward_A <= "00";     --NO HAZARD
                END IF;

                -- Forward B
                IF ( (MEMWB_RegWrite = '1') AND
                         (MEMWB_Register_Rd /= "00000") AND
                         (EXMEM_Register_Rd /= IDEX_Register_Rt) AND
                         (MEMWB_Register_Rd = IDEX_Register_Rt) ) THEN
                         Forward_B <= "01"; --MEM HAZARD
                ELSIF ( (EXMEM_RegWrite = '1') AND
                        (EXMEM_Register_Rd /= "00000") AND
                        (EXMEM_Register_Rd = IDEX_Register_Rt) ) THEN
                        Forward_B <= "10"; --EX HAZARD
                ELSE
```

```
               Forward_B <= "00"; --NO HAZARD
        END IF;
END PROCESS;

-- Mux for A Input
WITH Forward_A SELECT
   A_input <= Read_Data_1 WHEN "00",
                MEMWB_Read_Data WHEN "01",
                EXMEM_ALU_Result WHEN "10",
                "11111111" WHEN others;

-- Mux for B Input
WITH Forward_B SELECT
   Binput <= Read_Data_2 WHEN "00",
               MEMWB_Read_Data WHEN "01",
               EXMEM_ALU_Result WHEN "10",
               "11111111" WHEN others;

--ALU Input MUX  -- Need to Allow ALUSrc to Select Sign_Extend or from ForwardB Mux
B_input <= (Sign_Extend_p (7 DOWNTO 0)) WHEN (ALUSrc_p = '1') ELSE Binput;

--ALU Control Bits
ALU_Control(2) <= ( Funct_field(1) AND ALU_Op_p(1) ) OR ALU_Op_p(0);
ALU_Control(1) <= ( NOT Funct_field(2) ) OR ( NOT ALU_Op_p(1) );
ALU_Control(0) <= ( Funct_field(1) AND Funct_field(3) AND ALU_Op_p(1) ) OR
                  ( Funct_field(0) AND Funct_field(2) AND ALU_Op_p(1) );

--Set ALU_Zero
Zero <= '1' WHEN ( ALU_output (7 DOWNTO 0) = "00000000") ELSE '0';

--Register File : Write Address
Write_Address <= Write_Address_0_p WHEN RegDst_p = '0' ELSE Write_Address_1_p;

--ALU Output:  Must check for SLT instruction and set correct ALU_output
ALU_Result <= "0000000" & ALU_output (7) WHEN ALU_Control = "111"
                ELSE ALU_output (7 DOWNTO 0);

--Compute the ALU_output use the ALU_Control signals
PROCESS (ALU_Control, A_input, B_input)
        BEGIN --ALU Operation
        CASE ALU_Control IS
               --Function: A_input AND B_input
               WHEN "000" => ALU_output <= A_input AND B_input;
               --Function: A_input OR B_input
               WHEN "001" => ALU_output <= A_input OR B_input;
               --Function: A_input ADD B_input
               WHEN "010" => ALU_output <= A_input + B_input;
               --Function: A_input ? B_input
               WHEN "011" => ALU_output <= "00000000";
               --Function: A_input ? B_input
               WHEN "100" => ALU_output <= "00000000";
               --Function: A_input ? B_input
               WHEN "101" => ALU_output <= "00000000";
               --Function: A_input SUB B_input
               WHEN "110" => ALU_output <= A_input - B_input;
               --Function: SLT (set less than)
               WHEN "111" => ALU_output <= A_input - B_input;
               WHEN OTHERS => ALU_output <= "00000000";
        END CASE;
END PROCESS;

PROCESS
BEGIN
        WAIT UNTIL(Clock'EVENT) AND (Clock = '1');
        IF Reset = '1' THEN
               Zero_p                <= '0';
               ALU_Result_p          <= "00000000";
               Write_Address_p       <= "00000";
               MemtoReg_pp           <= '0';
               RegWrite_pp           <= '0';
               Read_Data_2_pp        <= "00000000";
```

```
                        MemRead_pp              <= '0';
                        MemWrite_pp             <= '0';
                ELSE
                        Zero_p                  <= Zero;
                        ALU_Result_p            <= ALU_Result;
                        Write_Address_p         <= Write_Address;
                        MemtoReg_pp             <= MemtoReg_p;
                        Read_Data_2_pp          <= Read_Data_2;
                        MemRead_pp              <= MemRead_p;
                        RegWrite_pp             <= RegWrite_p;
                        MemWrite_pp             <= MemWrite_p;
                        ALU1                    <= A_input;
                        ALU2                    <= B_input;
                        forwardA                <= Forward_A;
                        forwardB                <= Forward_B;
                        EXMEMRegWrite           <= EXMEM_RegWrite;
                        EXMEMALU_Result         <= EXMEM_ALU_Result;
                        EXMEMRegister_Rd        <= EXMEM_Register_Rd;
                        MEMWBRegWrite           <= MEMWB_RegWrite;
                        MEMWBRegister_Rd        <= MEMWB_Register_Rd;
                        MEMWBRead_Data          <= MEMWB_Read_Data;
                        IDEXRegister_Rs         <= IDEX_Register_Rs;
                        IDEXRegister_Rt         <= IDEX_Register_Rt;
                END IF;
        END PROCESS;
END behavior;


--------------------------------------------------------------------
-- Victor Rubio
-- Graduate Student
-- Klipsch School of Electrical and Computer Engineering
-- New Mexico State University
-- Las Cruces, NM
--
-- Filename: data_memory.vhd
-- Description: VHDL code to implment the Instruction Fetch unit
-- of the MIPS pipelined processor as seen in Chapter #6 of
-- Patterson and Hennessy book.  This file involves the use of the
-- LPM Components (LPM_RAM) to declare the Instruction Memory
-- as a random access memory (RAM).  See MAX+PLUS II Help on
-- "Implementing RAM & ROM (VHDL)" for details.
--
-- Signals with a _p, _pp, or _ppp sufix designate the number
-- of pipeline registers that signals runs through.
-- e.g. _p = 1 pipeline register, _pp = 2 pipeline registers, etc.
--------------------------------------------------------------------
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_ARITH.ALL;
 USE IEEE.STD_LOGIC_SIGNED.ALL;
LIBRARY LPM;
 USE LPM.LPM_COMPONENTS.ALL;

ENTITY datamemory IS
        PORT(
                Read_Data_p             : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Address                 : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Write_Data              : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                Read_Data_2_ppp         : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                ---Pipelined Signals
                MemRead_pp              : IN    STD_LOGIC;
                MemRead_ppp             : OUT   STD_LOGIC;
                MemWrite_pp             : IN    STD_LOGIC;
                MemWrite_ppp            : OUT   STD_LOGIC;
                MemtoReg_pp             : IN    STD_LOGIC;
                MemtoReg_ppp            : OUT   STD_LOGIC;
                RegWrite_pp             : IN    STD_LOGIC;
                RegWrite_ppp            : OUT   STD_LOGIC;
                ALU_Result_p            : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                ALU_Result_pp           : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
```

```
		Write_Address_p		: IN	STD_LOGIC_VECTOR (4 DOWNTO 0);
		Write_Address_pp	: OUT	STD_LOGIC_VECTOR (4 DOWNTO 0);
		Reg_WriteData		: OUT	STD_LOGIC_VECTOR (7 DOWNTO 0);
		Clock, Reset		: IN	STD_LOGIC);
END datamemory;

ARCHITECTURE behavior OF datamemory IS

	--Internal Signals
	SIGNAL LPM_WRITE		: STD_LOGIC;
	SIGNAL ReadData			: STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN
	datamemory : LPM_RAM_DQ
	GENERIC MAP(
		LPM_WIDTH		=> 8,
		LPM_WIDTHAD		=> 8,
		LPM_FILE		=> "data_memory.mif",
		LPM_INDATA		=> "REGISTERED",
		LPM_ADDRESS_CONTROL	=> "UNREGISTERED",
		LPM_OUTDATA		=> "UNREGISTERED")

	--READ DATA MEMORY
	PORT MAP(
		inclock			=> Clock,
		data			=> Write_Data,
		address			=> Address,
		we			=> LPM_WRITE,
		q			=> ReadData);

	--WRITE DATA MEMORY (SW)
	LPM_WRITE <= MemWrite_pp AND (NOT Clock);

	PROCESS
	BEGIN
		WAIT UNTIL Clock'EVENT AND Clock = '1';
			IF Reset = '1' THEN
				Read_Data_p	<= "00000000";
				MemtoReg_ppp	<= '0';
				RegWrite_ppp	<= '0';
				ALU_Result_pp	<= "00000000";
				Write_Address_pp <= "00000";
				MemRead_ppp	<= '0';
				MemWrite_ppp	<= '0';
				Read_Data_2_ppp <= "00000000";
			ELSE
				Read_Data_p	<= readdata;
				MemtoReg_ppp	<= MemtoReg_pp;
				RegWrite_ppp	<= RegWrite_pp;
				ALU_Result_pp	<= ALU_Result_p;
				Write_Address_pp <= Write_Address_p;
				MemRead_ppp	<= MemRead_pp;
				MemWrite_ppp	<= MemWrite_pp;
				Read_Data_2_ppp <= Write_Data;

				IF (MemtoReg_pp = '1') THEN
					Reg_WriteData <= ReadData;
				ELSE
					Reg_WriteData <= ALU_Result_p;
				END IF;

			END IF;
	END PROCESS;
END behavior;
```

APPENDIX G: MIPS PIPELINED – PIPELINING SIMULATION

```
-----------------------------------------------------------------
-- Victor Rubio
-- Filename: pipelining_example.mif
-- Description: Instruction Memory Initialization file for MIPS
-- Pipelined Processor.
-- Example obtained from P&H pg. #471
-----------------------------------------------------------------

--256 x 32 ROM implemented using
--four Embedded Array Blocks on Flex10K70 device
DEPTH = 256;
WIDTH = 32;

--Display in Hexidecimal Format
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT
BEGIN

  --Initialized Instruction Memory
                --PC   Instruction
  00: 8C2A0014;  --00:  LW  $10, 20 ($1)      $10 (0x0A) = MEM(0x01+0x14) = MEM(0x15)  = 0x15
  01: 00435822;  --04:  SUB $11, $2, $3       $11 (0x0B) = 0x02   - 0x03  = 0d-1 = 0XFF
  02: 00856024;  --08:  AND $12, $4, $5       $12 (0x0C) = 0x04 AND 0x05  = 0d 4 = 0x04
  03: 00C76825;  --0C:  OR  $13, $6, $7       $13 (0x0D) = 0x06  OR 0x07  = 0d 7 = 0x07
  04: 01097020;  --10:  ADD $14, $8, $9       $14 (0x0E) = 0x08   + 0x09  = 0d17 = 0x11

  --Initialized  Data Memory Values
  --00: 55;
  --01: AA;
  --02: 11;
  --03: 33;
  --15: 15;

END;
```

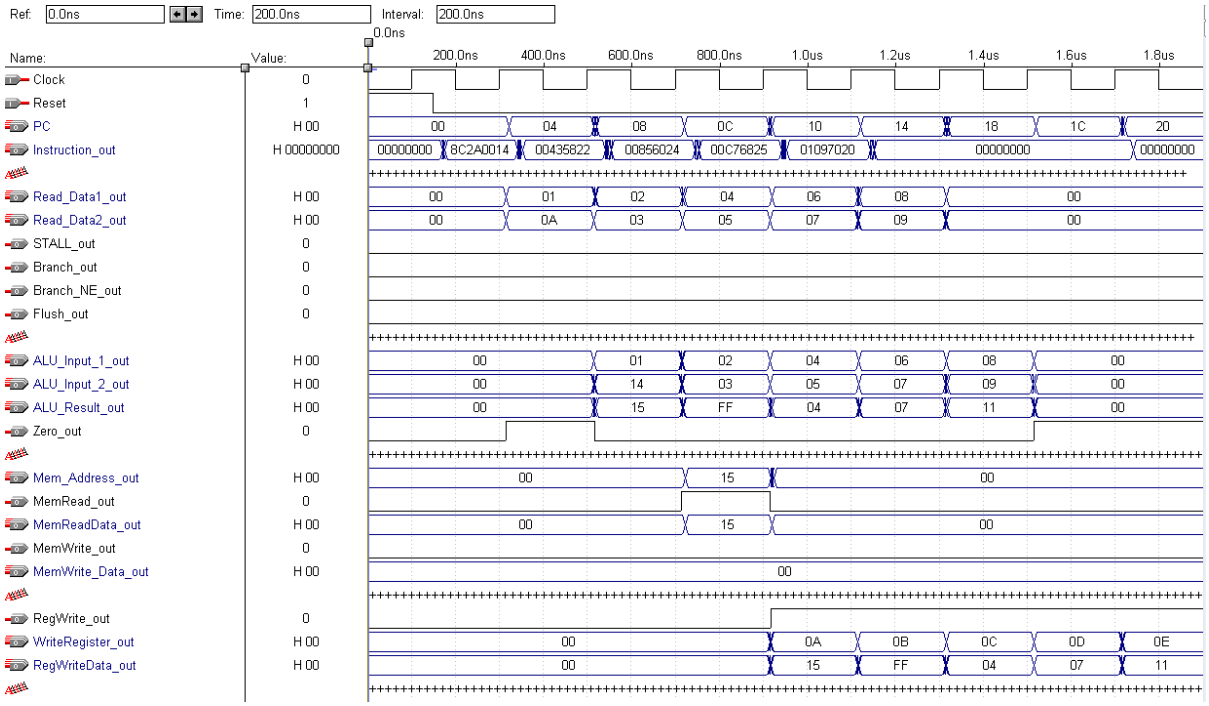

Figure G.1 MIPS Pipelined Simulation Waveform

APPENDIX H: MIPS PIPELINED – PIPELINING SIMULATION – SPIM VALIDATION

```
Registers
=========================
 PC     = 00000000     EPC    = 00000000     Cause  = 00000000     BadVAddr= 00000000
 Status = 00000000     HI     = 00000000     LO     = 00000000
                                  General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000015  R24 (t8) = 00000008
R1  (at) = 10010000  R9  (t1) = 00000000  R17 (s1) = ffffffff  R25 (t9) = 00000009
R2  (v0) = 0000000a  R10 (t2) = 00000002  R18 (s2) = 00000004  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000003  R19 (s3) = 00000007  R27 (k1) = 00000000
R4  (a0) = 10010093  R12 (t4) = 00000004  R20 (s4) = 00000011  R28 (gp) = 10008000
R5  (a1) = 00000000  R13 (t5) = 00000005  R21 (s5) = 00000000  R29 (sp) = 7fffe850
R6  (a2) = 7fffe858  R14 (t6) = 00000006  R22 (s6) = 00000000  R30 (s8) = 00000000
R7  (a3) = 00000000  R15 (t7) = 00000007  R23 (s7) = 00000000  R31 (ra) = 00000000
                                Double Floating Point Registers
FP0 =00000000,00000000 FP8 =00000000,00000000 FP16=00000000,00000000 FP24=00000000,00000000
FP2 =00000000,00000000 FP10=00000000,00000000 FP18=00000000,00000000 FP26=00000000,00000000
FP4 =00000000,00000000 FP12=00000000,00000000 FP20=00000000,00000000 FP28=00000000,00000000
FP6 =00000000,00000000 FP14=00000000,00000000 FP22=00000000,00000000 FP30=00000000,00000000
                                Single Floating Point Registers
FP0 =00000000 FP8 =00000000 FP16=00000000 FP24=00000000
FP1 =00000000 FP9 =00000000 FP17=00000000 FP25=00000000
FP2 =00000000 FP10=00000000 FP18=00000000 FP26=00000000
FP3 =00000000 FP11=00000000 FP19=00000000 FP27=00000000
FP4 =00000000 FP12=00000000 FP20=00000000 FP28=00000000
FP5 =00000000 FP13=00000000 FP21=00000000 FP29=00000000
FP6 =00000000 FP14=00000000 FP22=00000000 FP30=00000000
FP7 =00000000 FP15=00000000 FP23=00000000 FP31=00000000


Console
=========================
00: $s0 =      0x15     = 0d21 = 0x00000015
01: $s1 = 0x02 SUB 0x03 = 0d-1 = 0xffffffff
02: $s2 = 0x04 AND 0x05 = 0d4 = 0x00000004
03: $s3 = 0x06 OR  0x07 = 0d7 = 0x00000007
04: $s4 = 0x08 ADD 0x09 = 0d17 = 0x00000011
```

APPENDIX I: MIPS PIPELINED – DATA HAZARDS AND FORWARDING SIMULATION

```
-----------------------------------------------------------------
-- Victor Rubio
-- Filename: dependencies_example.mif
-- Description: Instruction Memory Initialization file for MIPS
-- Pipelined Processor.
-- Example obtained from P&H pg. #477-481
-----------------------------------------------------------------

--256 x 32 ROM implemented using
--four Embedded Array Blocks on Flex10K70 device
DEPTH = 256;
WIDTH = 32;

--Display in Hexidecimal Format
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT
BEGIN

  --Default Instruction Memory Vales
  [00..FF] : 00000000;

  --Initialized Instruction Memory
                --PC   Instruction
  00: 00231022; --00:  SUB $2,  $1, $3   --> $2  = 0x01   - 0x03 = 0d-2 = 0xFE
  01: 00456024; --04:  AND $12, $2, $5   --> $12 = 0xFE and 0x05 = 0x04
  02: 00C26825; --08:  OR  $13, $6, $2   --> $13 = 0x06  or 0xFE = 0xFE
  03: 00427020; --0C:  ADD $14, $2, $2   --> $14 = 0xFE   + 0xFE = 0x1FC
  04: AC4F000A; --10:  SW  $15, 100 ($2) --> mem(100 + $2) = $15 = mem(0x64 + 0xFE)
                --                                               = mem(0x162) = 0x0F
END;
```

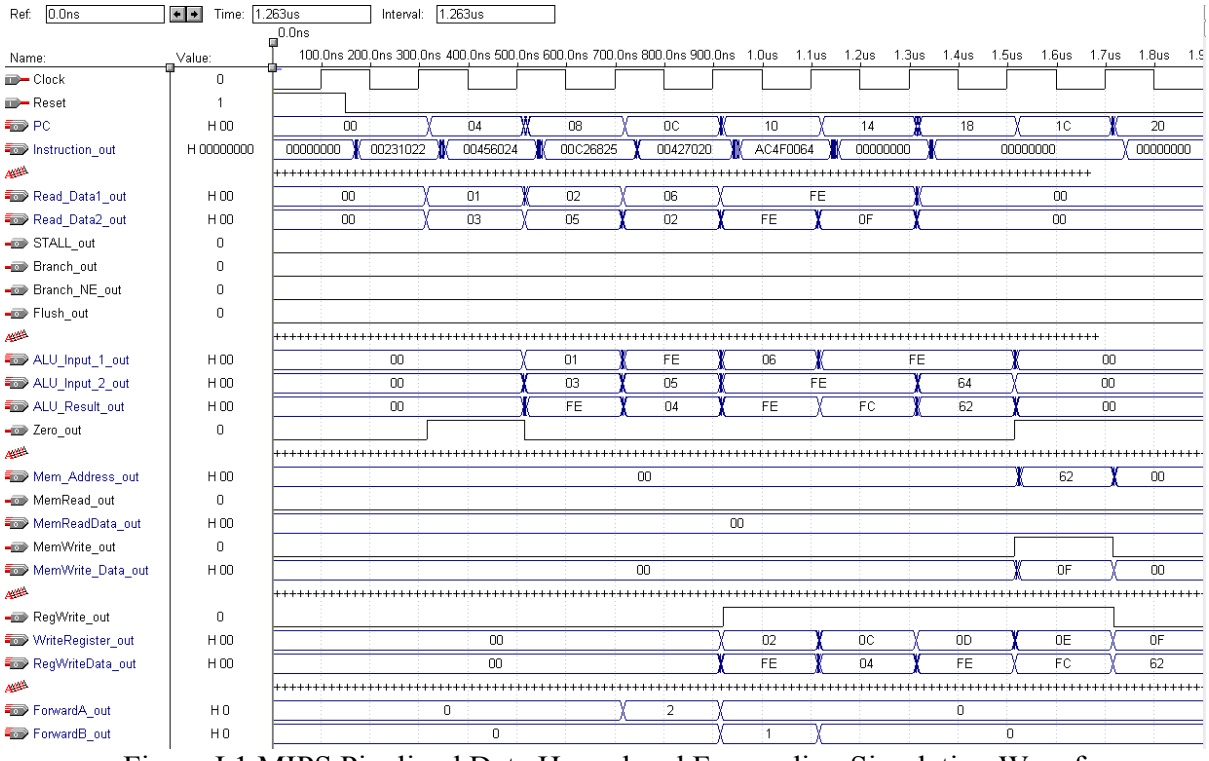

Figure I.1 MIPS Pipelined Data Hazard and Forwarding Simulation Waveform

APPENDIX J: MIPS PIPELINED – DATA HAZARDS AND FORWARDING SIMULATION – SPIM VALIDATION

```
Registers
=========================
 PC     = 00000000     EPC    = 00000000     Cause  = 00000000     BadVAddr= 00000000
 Status = 00000000     HI     = 00000000     LO     = 00000000
                                  General Registers
R0  (r0) = 00000000  R8  (t0) = 00000001  R16 (s0) = fffffffe    R24 (t8) = 00000000
R1  (at) = 10010000  R9  (t1) = 00000003  R17 (s1) = 00000004  R25 (t9) = 00000000
R2  (v0) = 0000000a   R10 (t2) = 00000005  R18 (s2) = fffffffe    R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 000001fc   R27 (k1) = 00000000
R4  (a0) = 10010076  R12 (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000
R5  (a1) = 00000000  R13 (t5) = 00000006  R21 (s5) = 00000000  R29 (sp) = 7fffe850
R6  (a2) = 7fffe858   R14 (t6) = 000000fe   R22 (s6) = 00000000  R30 (s8) = 00000000
R7  (a3) = 00000000  R15 (t7) = 00000000  R23 (s7) = 00000000  R31 (ra) = 00000000
                               Double Floating Point Registers
FP0 =00000000,00000000 FP8 =00000000,00000000 FP16=00000000,00000000 FP24=00000000,00000000
FP2 =00000000,00000000 FP10=00000000,00000000 FP18=00000000,00000000 FP26=00000000,00000000
FP4 =00000000,00000000 FP12=00000000,00000000 FP20=00000000,00000000 FP28=00000000,00000000
FP6 =00000000,00000000 FP14=00000000,00000000 FP22=00000000,00000000 FP30=00000000,00000000
                               Single Floating Point Registers
FP0 =00000000 FP8 =00000000 FP16=00000000 FP24=00000000
FP1 =00000000 FP9 =00000000 FP17=00000000 FP25=00000000
FP2 =00000000 FP10=00000000 FP18=00000000 FP26=00000000
FP3 =00000000 FP11=00000000 FP19=00000000 FP27=00000000
FP4 =00000000 FP12=00000000 FP20=00000000 FP28=00000000
FP5 =00000000 FP13=00000000 FP21=00000000 FP29=00000000
FP6 =00000000 FP14=00000000 FP22=00000000 FP30=00000000
FP7 =00000000 FP15=00000000 FP23=00000000 FP31=00000000


Console
=========================
00: $s0 = 0x01 SUB 0x03 = 0d-2 = 0xfffffffe
01: $s1 = 0xFE AND 0x05 = 0d4 = 0x00000004
02: $s2 = 0x06 OR  0xFE = 0d-2 = 0x fffffffe
03: $s3 = 0xFE ADD 0xFE = 0d508 = 0x000001fc
```

APPENDIX K: MIPS PIPELINED – DATA HAZARDS AND STALLS SIMULATION

```
-----------------------------------------------------------------
-- Victor Rubio
-- Filename: hazard_stall.mif
-- Description: Instruction Memory Initialization file for MIPS
-- Pipelined Processor.
-- Example obtained from P&H pg. #489 - 496
-----------------------------------------------------------------

--256 x 32 ROM implemented using
--four Embedded Array Blocks on Flex10K70 device
DEPTH = 256;
WIDTH = 32;
--Display in Hexidecimal Format
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT
BEGIN
  --Initialized Data Memory Values
  --00 : 55;
  --01 : AA;
  --02 : 11;
  --03 : 33;
  --Default Instruction Memory Vales
  [00..FF] : 00000000;
                --PC:  Instruction      Description:
00: 8C220014; --00:  LW  $2, 20($1)   $2 = mem(20 + $1)  = mem(0x14 + 0x01) = 0x15
01: 00452024; --04:  AND $4, $2, $5   $4 = 0x15 AND 0x05 = 0x05
02: 00464025; --08:  OR  $8, $2, $6   $8 = 0x15  OR 0x06 = 0x17
03: 00824820; --0C:  ADD $9, $4, $2   $9 = 0x05  +  0x15 = 0x1A
04: 00C7082A; --10:  SLT $1, $6, $7   $1 = 0x06 SLT 0x07 = 0x01

END;
```

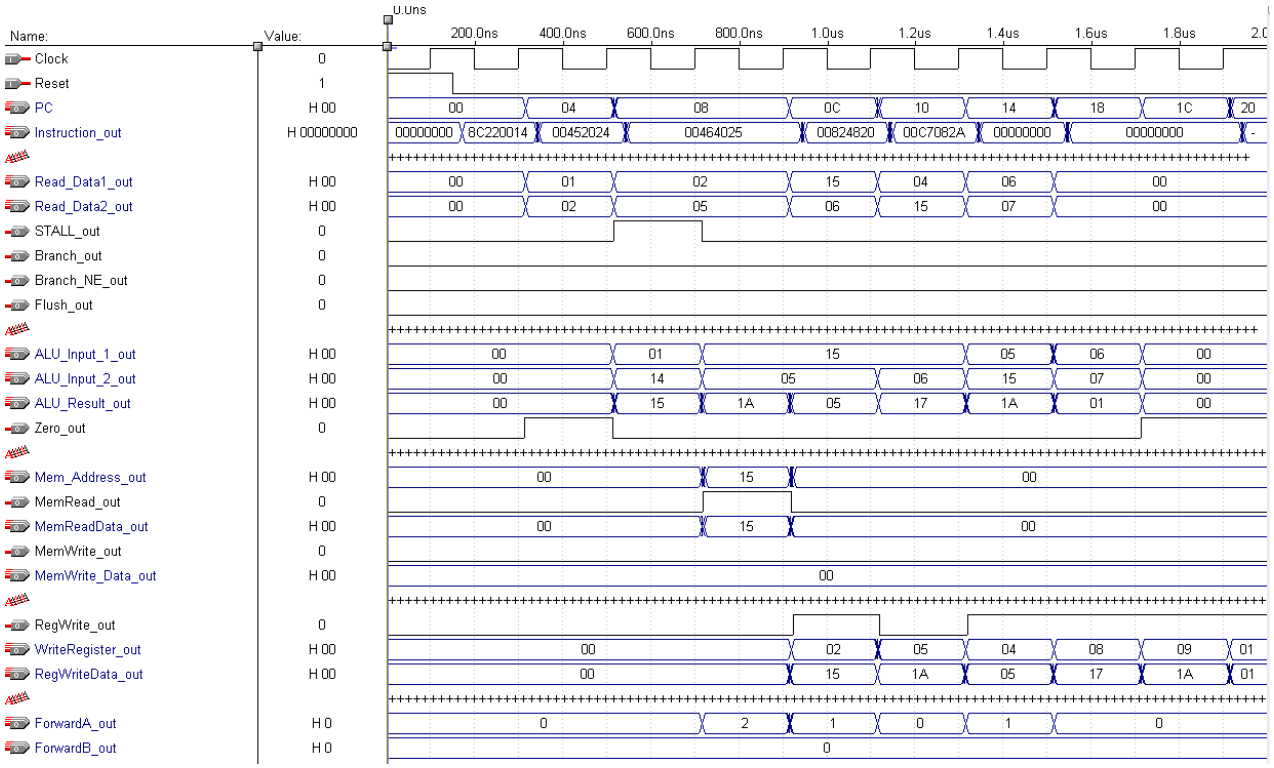

Figure K.1 MIPS Pipelined Data Hazard and Stall Simulation Waveform

APPENDIX L: MIPS PIPELINED – DATA HAZARDS AND STALLS SIMULATION – SPIM VALIDATION

```
Registers
=========================
 PC     = 00000000     EPC    = 00000000     Cause  = 00000000     BadVAddr= 00000000
 Status = 00000000     HI     = 00000000     LO     = 00000000
                                 General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000015  R24 (t8) = 00000006
R1  (at) = 10010000  R9  (t1) = 00000000  R17 (s1) = 00000005  R25 (t9) = 00000007
R2  (v0) = 0000000a  R10 (t2) = 00000000  R18 (s2) = 00000017  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000005  R19 (s3) = 0000001a  R27 (k1) = 00000000
R4  (a0) = 10010098  R12 (t4) = 00000000  R20 (s4) = 00000001  R28 (gp) = 10008000
R5  (a1) = 00000000  R13 (t5) = 00000006  R21 (s5) = 00000000  R29 (sp) = 7fffe850
R6  (a2) = 7fffe858  R14 (t6) = 00000000  R22 (s6) = 00000000  R30 (s8) = 00000000
R7  (a3) = 00000000  R15 (t7) = 00000000  R23 (s7) = 00000000  R31 (ra) = 00000000
                               Double Floating Point Registers
FP0 =00000000,00000000 FP8 =00000000,00000000 FP16=00000000,00000000 FP24=00000000,00000000
FP2 =00000000,00000000 FP10=00000000,00000000 FP18=00000000,00000000 FP26=00000000,00000000
FP4 =00000000,00000000 FP12=00000000,00000000 FP20=00000000,00000000 FP28=00000000,00000000
FP6 =00000000,00000000 FP14=00000000,00000000 FP22=00000000,00000000 FP30=00000000,00000000
                               Single Floating Point Registers
FP0 =00000000 FP8 =00000000 FP16=00000000 FP24=00000000
FP1 =00000000 FP9 =00000000 FP17=00000000 FP25=00000000
FP2 =00000000 FP10=00000000 FP18=00000000 FP26=00000000
FP3 =00000000 FP11=00000000 FP19=00000000 FP27=00000000
FP4 =00000000 FP12=00000000 FP20=00000000 FP28=00000000
FP5 =00000000 FP13=00000000 FP21=00000000 FP29=00000000
FP6 =00000000 FP14=00000000 FP22=00000000 FP30=00000000
FP7 =00000000 FP15=00000000 FP23=00000000 FP31=00000000


Console
=========================
00: $s0 =      0x15     = 0d21 = 0x00000015
01: $s1 = 0x15 AND 0x05 = 0d5 = 0x00000005
02: $s2 = 0x15  OR 0x06 = 0d23 = 0x00000017
03: $s3 = 0x05   + 0x15 = 0d26 = 0x0000001a
04: $s4 = 1 = if 0x06 < 0x07 = 0d1 = 0x00000001
```

APPENDIX M: MIPS PIPELINED – BRANCH HAZARD SIMULATION

```
-----------------------------------------------------------------
-- Victor Rubio
-- Filename: branch_hazard.mif
-- Description: Instruction Memory Initialization file for MIPS
-- Pipelined Processor.
-- Example obtained from P&H pg. #496-500
-----------------------------------------------------------------

--256 x 32 ROM implemented using
--four Embedded Array Blocks on Flex10K70 device
DEPTH = 256;
WIDTH = 32;
--Display in Hexidecimal Format
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT
BEGIN
  --Default Instruction Memory Vales
  [00..FF] : 00000000;
  --Initialized Instruction Memory

                --PC   Instruction
00: 00885022;   --00:  SUB $10, $4, $8 --> $10 (A) = $4 - $8 = 0xFC
01: 10630007;   --04:  BEQ $3,  $3, 7  --> if $3 = $3, 03 + 03 + 07 = 09
02: 00456024;   --08:  AND $12, $2, $5 --> $12 (C)= 2 & 5 = 0
03: 00466825;   --0C:  OR  $13, $2, $6 --> $13 (D)= 2 OR 6 = 0x06
04: 00827020;   --10:  ADD $14, $4, $2 --> $14 (E)= 4 + 2 = 0x06
05: 00C7782A;   --14:  SLT $15, $6, $7 --> if 6 < 7 ... $15 (F)= 1
06: 00000020;   --18:  nop
07: 00000020;   --1C:  nop
08: 00000020;   --20:  nop
09: 00452020;   --24:  ADD $4, $2, $5  --> $4 = 0x02  +  0x05 = 0x07
```

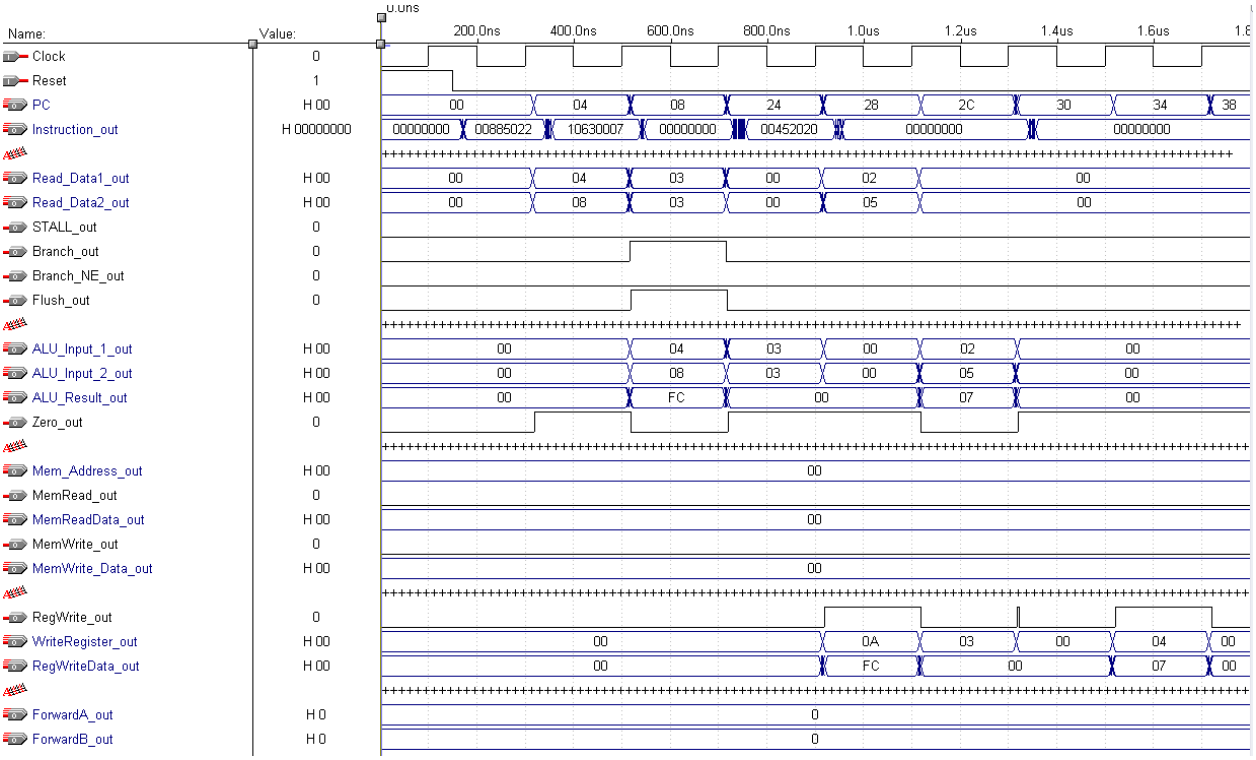

Figure M.1 MIPS Pipelined Branch Hazard Simulation Waveform

APPENDIX N: MIPS PIPELINED – BRANCH HAZARD SIMULATION – SPIM VALIDATION

```
Registers
=========================
 PC     = 00000000     EPC    = 00000000     Cause  = 00000000     BadVAddr= 00000000
 Status = 00000000     HI     = 00000000     LO     = 00000000
                            General Registers
R0  (r0) = 00000000  R8  (t0) = 00000004  R16 (s0) = fffffffc  R24 (t8) = 00000000
R1  (at) = 10010000  R9  (t1) = 00000008  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 0000000a  R10 (t2) = 00000003  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000003  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 100100f0  R12 (t4) = 00000002  R20 (s4) = 00000000  R28 (gp) = 10008000
R5  (a1) = 00000000  R13 (t5) = 00000005  R21 (s5) = 00000000  R29 (sp) = 7fffe850
R6  (a2) = 7fffe858  R14 (t6) = 00000000  R22 (s6) = 00000007  R30 (s8) = 00000000
R7  (a3) = 00000000  R15 (t7) = 00000000  R23 (s7) = 00000000  R31 (ra) = 00000000
                            Double Floating Point Registers
FP0 =00000000,00000000 FP8 =00000000,00000000 FP16=00000000,00000000 FP24=00000000,00000000
FP2 =00000000,00000000 FP10=00000000,00000000 FP18=00000000,00000000 FP26=00000000,00000000
FP4 =00000000,00000000 FP12=00000000,00000000 FP20=00000000,00000000 FP28=00000000,00000000
FP6 =00000000,00000000 FP14=00000000,00000000 FP22=00000000,00000000 FP30=00000000,00000000
                            Single Floating Point Registers
FP0 =00000000 FP8 =00000000 FP16=00000000 FP24=00000000
FP1 =00000000 FP9 =00000000 FP17=00000000 FP25=00000000
FP2 =00000000 FP10=00000000 FP18=00000000 FP26=00000000
FP3 =00000000 FP11=00000000 FP19=00000000 FP27=00000000
FP4 =00000000 FP12=00000000 FP20=00000000 FP28=00000000
FP5 =00000000 FP13=00000000 FP21=00000000 FP29=00000000
FP6 =00000000 FP14=00000000 FP22=00000000 FP30=00000000
FP7 =00000000 FP15=00000000 FP23=00000000 FP31=00000000


Console
=========================
00: $s0 = 0x04  SUB 0x08 = 0d-4 = 0xfffffffc
01: beq $3, $3 ---> TAKEN
06: $s6 = 0x02  ADD 0x05 = 0d7 = 00000007
```

APPENDIX O: MIPS PIPELINED FINAL DATHPATH AND CONTROL

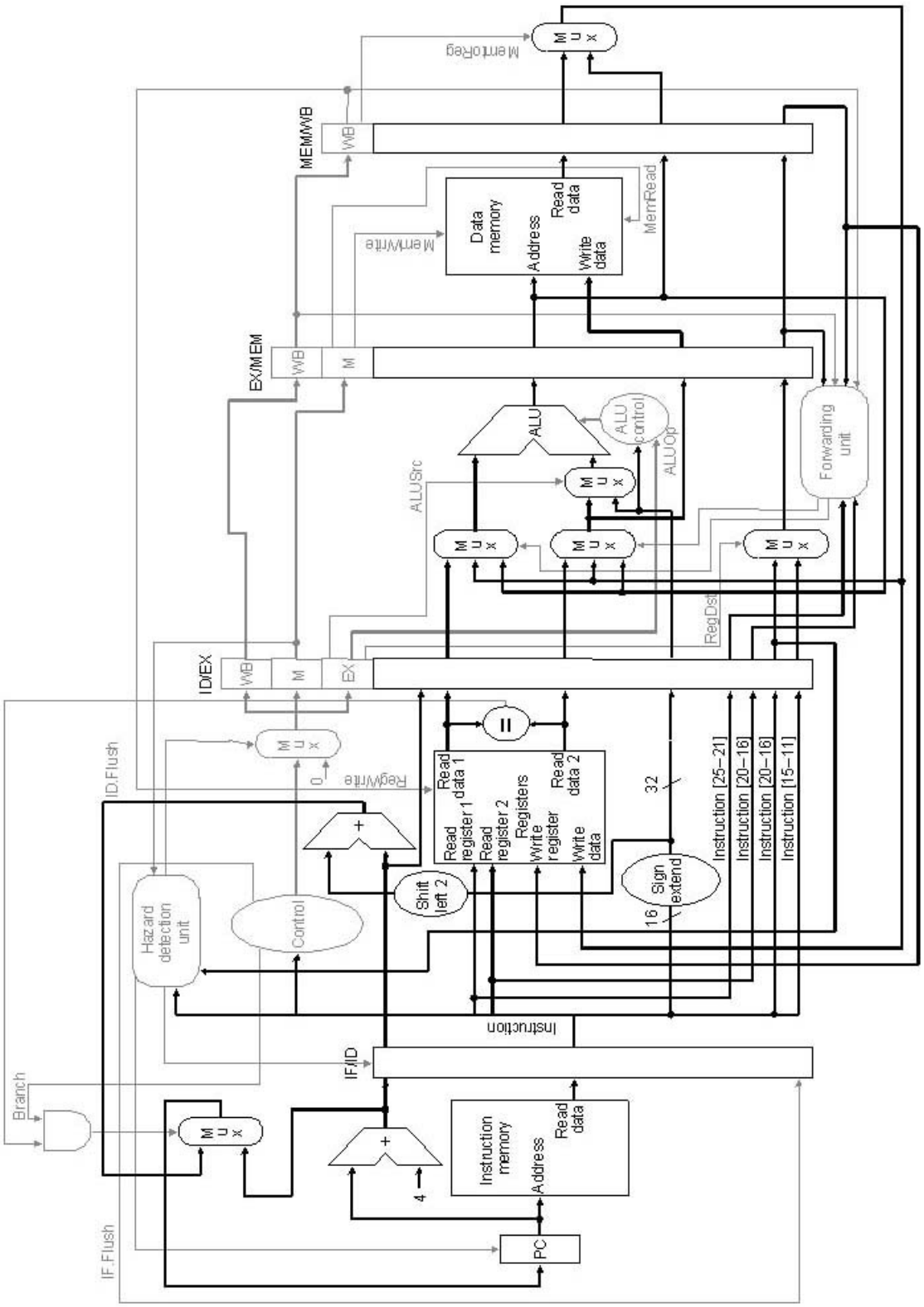

Figure O.1 MIPS Pipelined Final Dathpath and Control