# Introduction

***Digital Design and Computer Architecture***

Modified by Kent Jones

# **HW5: MIPS Single-Cycle Processor**

In this lab you will expand the MIPS single-cycle processor using VHDL.

* You will load a test program and check that the instructions work in both the simulator and on a FPGA.
* Next, you will implement two new instructions, and write a new test program that confirms the new instructions work as well.
* By the end of this lab, you should understand the internal operation of the MIPS single-cycle processor.
* Please read and follow the instructions in this lab carefully. In the past, many students have lost points for silly errors like documenting the signals requested.

Before starting this lab, you should review the single-cycle implementation of the MIPS processor described in Section 7.3 of your text, *Digital Design and Computer Architecture*. The single-cycle processor schematic from the text is repeated in this document for your convenience. **This version of the MIPS single-cycle processor can execute the following instructions: add, sub, and, or, slt, lw, sw, beq, addi, and j.**

Our model of the single-cycle MIPS processor divides the machine into two major components: the **control** and the **datapath**. Each component is constructed from various functional blocks. For example, as shown in the figure on the last page of this lab, the datapath contains the 32-bit ALU that you designed in digital logic design, the register file, the sign extension logic, and five multiplexers to choose appropriate operands.

1. MIPS Single-Cycle Processor

The VHDL for the single-cycle MIPS entity is given in Section 7.6.1 of the text. Although the VHDL in the text is very close to being correct, there have been changes made to the code so that it will work for both synthesis and simulation. The most significant changes to the code are in the way the initial program loads into the FPGA’s internal RAM. The complete project currently resides at:  
  
**\\cs1\CS\_ClassData\401\_Computer\_Architecture\Session\_07\_MIPS\_Architecture\MIPS\_CPU\_FPGA\_AND\_SIM\_V4**

Copy this entire folder to your CS401 HW5 folder. Make sure that you map the network drive for CS\_Students! Browse to the mips subfolder and open **mips.xise** in the Xilinx ISE editor. Look at the **mips1** entity, which instantiates two major components, **controller** and **datapath**.

Now, take a look at the **controller** entity and its components. It contains two main components: **maindec** and **aludec**. The **maindec** entity produces all control signals except those for the ALU. The **aludec** entity produces the controls signals, **alucontrol[2:0]**, for the ALU. Make sure you thoroughly understand the controller entity. **Correlate signal names in the VHDL code with the wires on the schematic**.

After you thoroughly understand the controller entity, take a look at the datapath VHDL entity. The datapath has quite a few components. Make sure you understand the role of each component in the Datapath, and where it resides on the MIPS single-cycle processor hardware schematic.

The highest-level entity, **mips\_cpu**, includes both instruction and data memories as well as the processor. Each of these memories is a 64-word × 32-bit array. The instruction memory needs to contain some initial values that represent the program to execute. **NOTE: because this is designed to run on an FPGA, we have used a simple memory model where the first instruction starts at address 0x00000000. This differs from the actual MIPS architecture where instructions typically start at address 0x00400000**.

**2. A Test Program**

We will use the following simple program to test that basic instructions work:

# test1.asm

# 23 October 2005 David Harris [David\_Harris@hmc.edu](mailto:David_Harris@hmc.edu)

# modified by Kent Jones kjones@whitworth.edu

# Test MIPS instructions.

#Address Assembly Code # Machine Code

0x00000000 main: addi $2, $0, 5 # 20020005

0x00000004 addi $7, $0, 3 # 20070003

0x00000008 addi $3, $0, 0xc # 2003000c

0x00000010 or $4, $7, $2 # 00e22025

0x00000014 and $5, $3, $4 # 00642824

0x00000018 add $5, $5, $4 # 00a42820

0x0000001C beq $5, $7, end # 10a70008

0x00000020 slt $6, $3, $4 # 0064302a

0x00000024 beq $6, $0, around # 10c00001

0x00000028 addi $5, $0, 10 # 2005000a

0x0000002C around: slt $6, $7, $2 # 00e2302a

0x00000030 add $7, $6, $5 # 00c53820

0x00000034 sub $7, $7, $2 # 00e23822

0x00000038 j end # 0800000f

0x0000003C lw $7, 0($0) # 8c070000

0x00000040 end: sw $7, 71($2) # ac470047  
0x00000044 j main # 08000000

**Figure 1. MIPS assembly program: test1.asm / memfile.dat**

You can find this code supplementary lab material provided on the textbook website. The actual machine code version of this program is stored in the folder for this lab and is called **memfile.dat**. The machine code is already programmed into the memory of the current Xilinx project.

The current project automatically loads the machine code version of this program directly into the MIPS instruction memory at synthesis. This allows us to run small programs directly on the FPGA without having to implement the complex SDRAM interface. The current code implement an array of STD\_LOGIC\_VECTOR with the initial values read in from the **memfile.dat** file.

To understand how this works you will need to examine the code in the **mips\_mem**.**vhd** file. The instruction memory consists of an array of 64 locations, each of which can store 32 bits. We first define a hardware data type called ramtype:

**type** ramtype **is** **array** **(**63 **downto** 0**)** **of** STD\_LOGIC\_VECTOR**(**31 **downto** 0**);**

this data type is used to make the ram:

process **(** a **)** **is**

-- use an impure function to read RAM   
 -- from a file and store in the FPGA's ram memory

**variable** mem**:** ramtype **:=** InitRamFromFile**(**"memfile.dat"**);**

**begin**

rd **<=** mem**(** **to\_integer(**unsigned**(**a**))** **);**

**end process;**

The function, **InitRamFromFile("memfile.dat")** initializes the variable **mem**, by reading from the data file **memfile.dat** This function is defined inside the architecture of the imem object and works by reading each char from the file (which represents a 4 bit hexadecimal value) and converting each char to a four bit integer. These 4 bits are shifted into position and added to a temporary result. After constructing the entire 32 bit word for each instruction, the code writes the instruction to the next instruction memory location.

When you examine the code, you will notice the various data type conversions. The next figure demonstrates how to convert between various data types in VHDL.



**3. Synthesis**

Synthesize the highest level entity, **mips\_fpga\_interface**. Notice that the only necessary inputs to the highest level entity are **clk** and **rst\_n** (reset), and **led** (the leds are for verification).

View the synthesis report. There should be no errors. There will likely be some warnings. You should examine these.

Finally, view the RTL schematic.

**4. Testing the single-cycle MIPS processor**

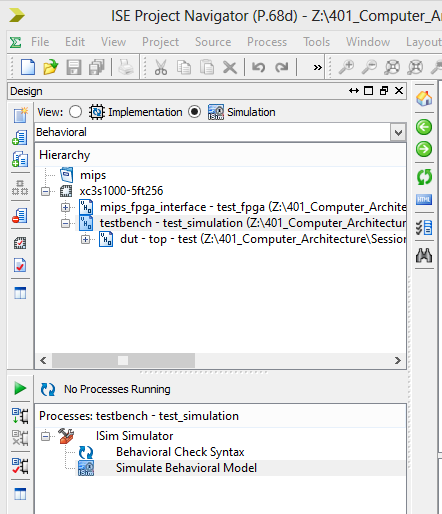
To test the processor, you will simulate running **test1.asm** on the VHDL code. The machine code for this program resides in **memfile.dat**. As described in section 2, this code loads into the instruction memory of the MIPS processor during synthesis.

In a complex system, if you don’t know what answer the system should produce, you won’t be able to debug the system. Begin by predicting what should happen on each cycle when running the program. To do this, complete the chart in Table 1 at the end of the lab. **What address will the final sw instruction (just before the jump) write to and what value will it write**? Address: $7 Value: 7

## VHDL TestBench

Now that you have predicted the output of the program, view the sources for Behavioral Simulation and look at the code for the testbench entity. The **mips\_testbench.vhd** code is for testing only and cannot be synthesized into hardware. It generates clock and reset inputs for the device under test, **top**. It also checks for a memory write and verifies the address and data being written. Do these match your expectations from your code analysis in Table 1?

Now you are ready to test your MIPS processor via the ISIM simulator. In the design view, select the testbench. Make sure to check **Simulation** in the view, and to select **Behavioral** directly underneath the view. Now, run the **Simulate Behavioral Model** process.



Now you can view your simulation output. You will see some warnings, these warnings occur because when the processor starts, some of its internal states are undefined. When the **alu** attempts to “add” undefined values, this causes a warning. Fortunately, despite the warnings, the machine should work correctly, and if you look carefully through the output you should see several **“simulation succeeded”** messages. This message occurs when the program runs and verifies the value loaded into memory. Remember that the program has a jump to main at the end of the program and thus will continue to repeat.

## Debugging your MIPS Processor

For debugging, you will likely need to make other signals from sub-components visible in the trace window of the ISIM simulator. To do this, expand the **testbench** entity in the **Instance and Process Name** window by clicking on the triangle beside it. Now, expand the **dut** (device under test) entity, followed by the **mips1** entity, followed by the **cont** (controller) entity. Within the **cont** entity you will see the **ad** (alu decoder) entity and the **md** (main decoder) entity. Click and drag the **md** (main decoder) entity to the waveform window. Also click and drag the **ad** (alu decoder) entity to the waveform window as well.

Now that you have the controller entity signals on the waveform window, you will need to restart the simulation to update the waveform. let it run for a bit, and then pause the simulation by clicking on the pause button on the tool bar.

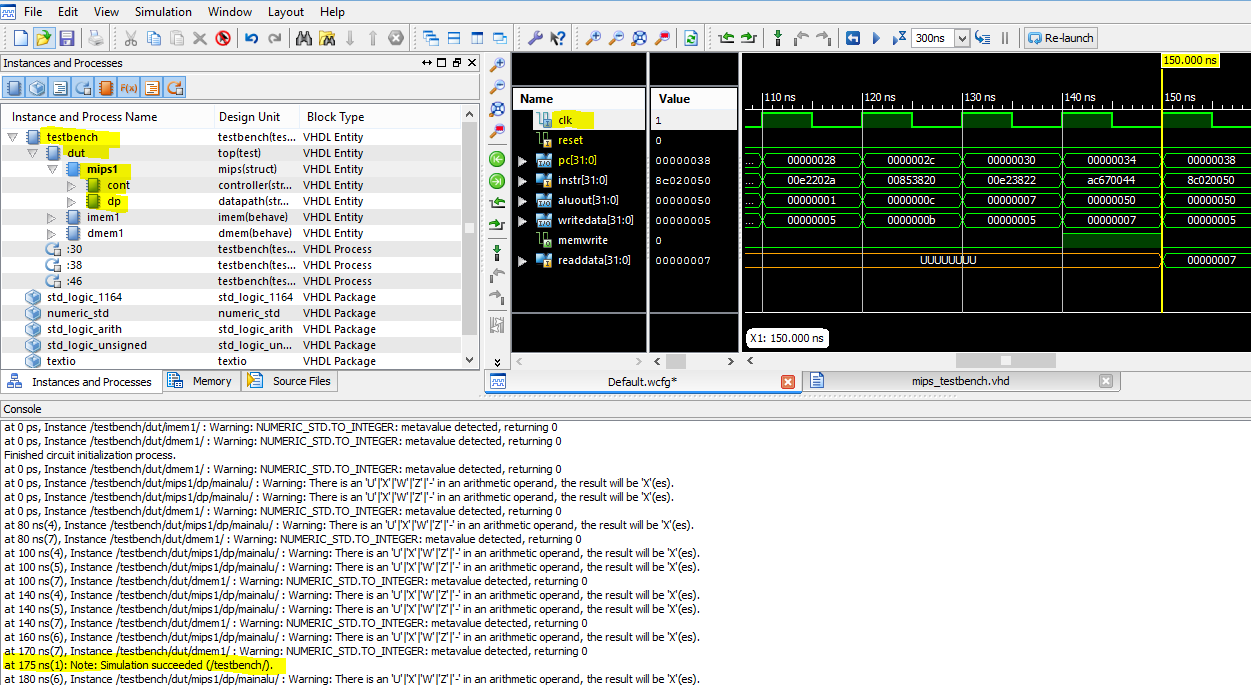
You can remove signals from the waveform window by right-clicking on them. You can also change how signals are displayed on the waveform window by right-clicking and selecting a **hexadecimal radix**.

You can change the order of the signals on the waveform by dragging them to a new position. For this assignment you will be required to have some specific signals in a specific order.

During debug, you’ll likely want to view several internal signals. **However, on the final waveform that you turn in, show ONLY the following signals in this order: clk, reset, pc, instr, aluout, writedata, memwrite, and readdata. All the multi-bit signal values must be output in hexadecimal and must be readable to get full credit.**

Before you print out your final waveform, ctrl-scroll in the waveform window to set the zoom so that the hex values are clearly visible. **Print out the waveform from 30 ns to 330 ns. with the specified signals in the specified order.** You will have to figure out how to print a waveform. You can do this through print-preview setup, or, you can take screen shots and combine the images in Gimp or some other image editing program.

See the following figure to see what your output should look like:



1. Running the MIPS single-cycle processor on the FPGA

After finishing your simulation, you will run the MIPS processor on the FPGA. To do this, exit the simulator, then, switch the design view back to **Implementation** from **Simulation**. Select the top level vhdl file, **mips\_fpga\_interface.vhd.** Don’t forget to set the floats in the properties for **Generate Programming File**.

Before you actually generate the bit file, you should examine the top level code. This has a process that generates a slow clock (1 second) and also outputs the values of the program counter to the leds 5 downto 0. It also outputs the memwrite signal to led(6).

As noted earlier, the actual program has a jump which will jump back to the starting address (address 0 of program memory) in order to run the program over and over again… ☺

Go ahead and synthesize the processor, and load the MIPS bit file into the fpga. Run the program and watch the LED’s flash. Does the memwrite signal turn on appropriately?

Yes. The memwrite signal turns on appropriately.

1. Adding the ori and bne instructions to the MIPS single-cycle processor

Will you need to modify the MIPS single-cycle processor hardware in order to add the **ori** and **bne** instructions? If so, you should directly modify the MIPS processor schematic (on the last page) to show what changes are necessary. **If you do need to make hardware changes you can draw your changes directly onto this schematic.**

To finish adding the instructions ori and bne:

1. **Modify the VHDL for the main decoder and ALU decoder as required.**
2. **Show your changes in the tables at the end of the lab.**
3. **Finally, modify the VHDL code as needed to include your modifications.**
4. Testing your modified MIPS single-cycle processor
5. **You need to determine the machine code for instructions given in test2.asm.** **Also comment each line of code of test2.asm**. You can use PCSpim to help convert to machine code, **and store the machine code in a file called memfile2.dat**. **Remember that PCSpim will not give you the correct machine code for j, beq, bne because the address for the jumps will be incorrect – this is because our machine starts the text segment at address 0x00000000.**
6. You will need to modify the VHDL code to load memfile2.dat instead of memfile.dat.
7. Also, you’ll need to modify the VHDL testbench to **check for the correct “Simulation Succeeded” values at the end of the program**. Look at the testbench that we provided you to figure out how to test for the correct value. You might also find it useful to create a table similar to that of **Table 1**.
8. Generate a final waveform to turn in and only include the following signals in this order: **clk, reset, pc, instr, aluout, writedata, memwrite, and readdata,** in that order. **Make sure all your waveforms are readable and show values in hexadecimal.**

# test2.asm

# 23 March 2006 S. Harris sharris@hmc.edu

#

# Test MIPS instructions.

#Assembly Code

main: ori $t0, $0, 0x8000

addi $t1, $0, -32768

ori $t2, $t0, 0x8001

beq $t0, $t1, there

slt $t3, $t1, $t0

bne $t3, $0, here

j there

here: sub $t2, $t2, $t0

ori $t0, $t0, 0xFF

there: add $t3, $t3, $t2

sub $t0, $t2, $t0

sw $t0, 82($t3)  
 j main

**Figure 3. MIPS assembly program: test2.asm**

**8. Testing your modified MIPS program on the FPGA**

Run your modified program on the FPGA. Does the FPGA perform as expected?

# 9. What to Turn In

Please turn in each of the following items, clearly labeled and in the following order:

1. The complete HW5 folder with the modified vhdl files, memfile.dat and memfile2.dat underneath your CS401 folder on CS\_Students. Make sure the vhdl files are separated from the Xilinx ISE generated files for clarity.
2. **Please indicate how many hours you spent on this lab.** This will not affect your grade, but will be helpful for calibrating the workload for next semester’s labs.
3. A completed version of Table 1.
4. Marked up versions of the datapath schematic and decoder tables that adds the ori and bne instructions.
5. A printout of your VHDL code for your modified MIPS computer (including ori and bne functionality) with the changes highlighted:
6. Simulation waveforms of:

* mips\_testbench.v for test1.asm
* mips\_testbench.v for test2.asm (with your modified single-cycle MIPS processor)

The simulation waveforms should give the signal values in hexadecimal format and **show ONLY the following signals in this order: clk, reset, pc, instr, aluout, writedata, memwrite, and readdata. All the values need to be output in hexadecimal (except memwrite) and must be readable to get full credit.**

**NOTE: Be sure the waveforms match your expectations. Check that the waveforms are zoomed out enough that the instructor can read your bus values. Unreadable waveforms will receive no credit. Use several pages as necessary.**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Cycle | reset | pc | instr | branch | srca | srcb | aluresult | zero | pcsrc | writedata | memwrite | read data |
| 1 | 1 | 00 | addi $2,$0,5  20020005 | 0 | 0 | 5 | 5 | 0 | 0 | x | 0 | 0 |
| 2 | 0 | 04 | addi $7,$0,3  20070003 | 0 | 0 | 3 | 3 | 0 | 0 | x | 0 | 0 |
| 3 | 0 | 08 | addi $3,$0,0xc  2003000c | 0 | 0 | 12 | 12 | 0 | 0 | x | 0 | 0 |
| 4 | 0 | 0C | or $4, $7, $2  00e22025 | 0 | 3 | 5 | 7 | 0 | 0 | 5 | 0 | 0 |
| 5 |  | 10 | and $5, $3, $4  00642824 | 0 | 12 | 7 | 4 | 0 | 0 | 7 | 0 | 0 |
| 6 |  | 14 | add $5, $5, $4  00a42820 | 0 | 4 | 7 | 11 | 0 | 0 | 7 | 0 | 0 |
| 7 |  | 18 | beq $5, $7, end  10a70008 | 1 | 11 | 3 | 8 | 1 | 0 | 3 | 0 | 0 |
| 8 |  | 1C | slt $6, $3, $4  0064302a | 0 | 12 | 7 | 0 | 1 | 0 | 7 | 0 | 0 |
| 9 |  | 20 | beq $6, $0, around  10c00001 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 10 |  | 28 | slt $6, $7, $2  00e2302a | 0 | 3 | 5 | 1 | 0 | 0 | 5 | 0 | 0 |
| 11 |  | 2C | add $7, $6, $5  00c53820 | 0 | 1 | 11 | 12 | 0 | 0 | 11 | 0 | 0 |
| 12 |  | 30 | sub $7, $7, $2  00e23822 | 0 | 12 | 5 | 7 | 1 | 0 | 5 | 0 | 0 |
| 13 |  | 34 | j end  0800000f | X | 0 | 15 | 0 | 0 | X | 0 | 0 | 0 |
| 14 |  | 3C | sw $7, 71($2)  ac470047 | 0 | 5 | 71 | 76 | 1 | 0 | 7 | 1 | 0 |

**Table 1. First fourteen cycles of executing assembly program test1.asm**

Remember, *branch* is asserted (1) when the instruction is a branch (beq) instruction. *aluout* is the output of the ALU at each cycle. *zero* is high (1) only if *aluout* is 0. *pcsrc*, a signal in the datapath, is low (0) when *nextpc* should be pc+4. *pcsrc* is high (1) when the *nextpc* should be the branch target address (*pcbranch*). You will notice that all of these signals are not available from the top-level entity (mips). For debugging, you might want to look at these signals and others.



26

Op(0)

Figure 2

op(0)

branch

**Figure 1: Single-cycle MIPS processor**

PcSrc

Figure 2

zero

# Extended functionality. Main Decoder:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Op5:0** | **RegWrite** | **RegDst** | **AluSrc1:0** | **Branch** | **MemWrite** | **MemtoReg** | **ALUOp1:0** | **Jump** |  |  |  |
| R-type | 000000 | 1 | 1 | 0 | 0 | 0 | 0 | 10 | 0 |  |  |  |
| lw | 100011 | 1 | 0 | 1 | 0 | 0 | 1 | 00 | 0 |  |  |  |
| sw | 101011 | 0 | X | 1 | 0 | 1 | X | 00 | 0 |  |  |  |
| beq | 000100 | 0 | X | 0 | 1 | 0 | X | 01 | 0 |  |  |  |
| addi | 001000 | 1 | 0 | 1 | 0 | 0 | 0 | 00 | 0 |  |  |  |
| j | 000010 | 0 | X | X | X | 0 | X | XX | 1 |  |  |  |
| ori | 001101 | 1 | 0 | 1 | 0 | 0 | 0 | 10 | 0 |  |  |  |
| bne | 000101 | 0 | X | 0 | 1 | 0 | X | 01 | 0 |  |  |  |

**Extended functionality. ALU Decoder:**

|  |  |
| --- | --- |
| **ALUOp1:0** | **Meaning** |
| 00 | Add |
| 01 | Subtract |
| 10 | Look at funct field |
| 11 |  |