# Name: Amon Ratna Sthapit

# Introduction

***Digital Design and Computer Architecture***

Modified by Kent Jones

# **MIPS 3: MIPS Single-Cycle Processor**

# 

|  |  |  |
| --- | --- | --- |
| **CATEGORY** | **POINTS** |  |
| **Exercise 1: Change MIPS Program (Individual)** |  | 20 |
| **Exercise 2: Compare Sim to Prediction (Individual)** |  | 20 |
| **Exercise 3: Running MIPS (Individual)** |  | 20 |
| **Group Project: MIPS Extension and Test (Group)** |  | 20 |
| **Code Review / Walkthrough / Presentation(Group)** |  | 20 |
| **TOTAL** |  | 100 |

# 

In this lab you will expand the MIPS single-cycle processor using VHDL. You will also turn in this document on blackboard with your answers pasted into the document. Each individual will turn in the document separately.

* 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 MIPS instructions, XORI and BNE. You will then write a new test program that confirms the new MIPS instructions work. In order to implement BNE you will have to make changes to the VHDL for the processor.**
* 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 failing to include screen snips of the simulation signal traces requested in the lab.

# MIPS Single-Cycle Processor

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* and covered in class. 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, addi, and, beq, j, lw, or, slt, sub, and sw**

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.

**Make a folder in your CS401 folder on CS1 called MIPS3. Copy the following folder to you new MIPS3 folder:**

[**\\cs1\CS\_ClassData\401\_Computer\_Architecture\2018\_spring\**mips\_fpga\_nexsys4\_ddr](file:///\\cs1\CS_ClassData\401_Computer_Architecture\2018_spring\mips_fpga_nexsys4_ddr)

Make sure that you have mapped the network drive to your folder on CS\_Students! Browse to the mips subfolder and open **mips.xpr** in the Xilinx Vivado editor. Look at the **top** entity in the **mips\_top.vhd** file.

The highest-level entity **for the mips processor itself** is called **mips\_top**, it contains the **instruction memories** and **data memories** and the **processor** itself. Each of the memories is a **64-word × 32-bit array**. The instruction memory needs to contain some initial values that represent the program to execute. Go ahead an open the elaborated design then double click on the **mips1** component. It should look like this at the top level. Notice that there is a data memory (**dmem1**), an instruction memory (**imem1**) and a mips processor (**mips1**). They are draw in reverse order by Vivado since we are pulling off the instruction read from imem1 and sending it out to the hex display (it also wraps back around into the instruction input of mips1). The data memory is then rendered on the left side of the image rather than the right…



Expand the mips1 component. You should see this:



Our model of the single-cycle MIPS processor divides the machine into two major components: the **control unit (cont)** and the **datapath unit (dp)**. 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.

# MIPS Control Unit

Now, take a look at the **controller** entity and its components.



The controller contains two main components:

* **Main decoder (md)** this entity produces all control signals **except those for the ALU**.
* **ALU decoder (ad)** This entity produces the controls signals, **alucontrol[2:0]**, for the ALU.

Make sure you thoroughly understand the controller entity. **Find and match signal names from the VHDL code with the wires shown on the mips schematic on the last page of this handout**.

Finally, take a look at the **datapath (dp)** VHDL entity. The datapath has quite a few components. You may find it easier look at the diagram in the book and the last page of this handout than at the elaborated design, however you should be able to located signals in both renderings. Make sure you understand the role of each component in the datapath unit, and where it resides on the MIPS single-cycle processor hardware schematic.

# Initial Test Program

The MIPS project comes with the instruction memory preloaded with a **memfile.dat that is contains the machine code for the following assembly language program called mipstest.asm**.

**NOTE: This is very important to understand. The test program stored in memfile.dat has ONLY 8 digits per line. Each line consists of a hex number that should only use digits and the lower case letters a,b,c,d,e,f. In the future, when writing your own programs, DO NOT put anything other than exactly 8 lower case hex digits on a line. If you don’t follow these rules, your program will not work and you may become confused as to why things don’t work.**

The current **simulation** **project** automatically **reads and loads the machine code from the memfile.dat** into the MIPS instruction memory during the **synthesis process**. This allows us to run small programs directly on the FPGA without having to implement a complex SDRAM interface. The current code is stored in an array of STD\_LOGIC\_VECTOR that forms the instruction memory for the machine.

NOTE: This is human readable assembly code and will NOT execute in our VHDL processor in this form! The actual file to run on our MIPS processor, memfile.dat, has only 8 lower case hex digits per line.

This is what a memfile.dat file should look like! It contains mips machine code! This code is stored in the instruction memory.

**memfile.dat**

20020005

2003000c

2067fff7

00e22025

00642824

00a42820

10a7000a

0064202a

10800001

20050000

00e2202a

00853820

00e23822

ac670044

8c020050

08000011

20020001

ac020054

08000000

# mipstest.asm

# David\_Harris@hmc.edu 9 November 2005

#

# Test the MIPS processor.

# add, sub, and, or, slt, addi, lw, sw, beq, j

# If successful, it should write the value 7 to address 84

# Assembly Description Address Machine

main: addi $2, $0, 5 # initialize $2 = 5 0 20020005

addi $3, $0, 12 # initialize $3 = 12 4 2003000c

addi $7, $3, -9 # initialize $7 = 3 8 2067fff7

or $4, $7, $2 # $4 <= 3 or 5 = 7 c 00e22025

and $5, $3, $4 # $5 <= 12 and 7 = 4 10 00642824

add $5, $5, $4 # $5 = 4 + 7 = 11 14 00a42820

beq $5, $7, end # shouldn't be taken 18 10a7000a

slt $4, $3, $4 # $4 = 12 < 7 = 0 1c 0064202a

beq $4, $0, around # should be taken 20 10800001

addi $5, $0, 0 # shouldn't happen 24 20050000

around: slt $4, $7, $2 # $4 = 3 < 5 = 1 28 00e2202a

add $7, $4, $5 # $7 = 1 + 11 = 12 2c 00853820

sub $7, $7, $2 # $7 = 12 - 5 = 7 30 00e23822

# Because there are 4 bytes per word, we only read from

# multiples of four (look in mips\_mem\_data.vhd).

# Thus, address 80 = 01010000 (ignore bottom two bits

# of the address and 010100 in binary is location 20.

# Look at the location 20 in the simulation to see 7

sw $7, 68($3) # Write 7 to loc 20 34 ac670044

lw $2, 80($0) # $2 = [80] = 7 38 8c020050

j end # should be taken 3c 08000011

addi $2, $0, 1 # shouldn't happen 40 20020001

end: sw $2, 84($0) # write adr 84 = 7 44 ac020054

j main # restart 48 08000000

**Figure 1. MIPS assembly program: mipstest.asm and the machine code file memfile.dat**

# Memory Model Used for Our VHDL MIPS Processor

**Because this simple mips processor was designed to run entirely on a single FPGA, we have used a simple memory model for both the instruction memory and the data memory. The first instruction memory location starts at address 0x00000000. The same is true for the data memory. This differs from the actual MIPS architecture where the first instructions typically start at address 0x00400000**. In the future, if you are able to integrate a memory controller into your processor you may choose to change this.

# How the HEX Code Machine Language Program is loaded at Synthesis Time

**To understand how loading hex code from memfile.dat into memory works**, you need to examine the code in the **mips\_mem\_instructions**.**vhd** file (double-click on the **imem** component in our VHDL project). The instruction memory consists of an array of 64 locations, each of which can store 32 bits. In this file we first define a hardware data type called ramtype:

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

We then use this data type to initialize the ram array object during synthesis as follows:

-- use the impure function to read RAM from a file and

-- store in the FPGA's ram memory

**signal** mem**:** ramtype **:=** InitRamFromFile**(**"memfile\_2.dat"**);**

The function, **InitRamFromFile("memfile.dat")** returns a RAM array to initialize the signal **mem**. It does this by reading the data file **memfile.dat**. The function creates 32 bit words in the mem array by reading one character at a time from the **memfile.dat** file. Each character represents a 4 bit hexadecimal value and after reading is converted to a four bit integer. Each 4 bit integer is shifted into its correct position and added to a temporary result. After constructing an entire temporary result (a 32 bit word) for each instruction, the code writes that instruction to the next instruction memory location in **RAM** which is returned to initialize **mem**. Here is the code to initialize the instruction memory”

-- function to initialize the instruction memory from a data file

**impure** **function** InitRamFromFile **(** RamFileName **:** **in** string **)** **return** RamType **is**

**variable** ch**:** character**;**

**variable** index **:** integer**;**

**variable** result**:** signed**((**width**-**1**)** **downto** 0**);**

**variable** tmpResult**:** signed**(**63 **downto** 0**);**

**file** mem\_file**:** TEXT **is** **in** RamFileName**;**

**variable** L**:** line**;**

**variable** RAM **:** ramtype**;**

**begin**

-- initialize memory from a file

**for** i **in** 0 **to** 63 **loop** -- set all contents low

RAM**(**i**)** **:=** std\_logic\_vector**(to\_unsigned(**0**,** width**));**

**end** **loop;**

index **:=** 0**;**

**while** not **endfile(**mem\_file**)** **loop**

-- read the next line from the file

**readline(**mem\_file**,** L**);**

result **:=** **to\_signed(**0**,**width**);**

**for** i **in** 1 **to** 8 **loop**

-- read character from the line just read

**read(**L**,** ch**);**

-- convert character to a binary value from a hex value

**if** '0' **<=** ch and ch **<=** '9' **then**

tmpResult **:=** result**\***16 **+** character'**pos(**ch**)** **-** character'**pos(**'0'**)** **;**

result **:=** tmpResult**(**31 **downto** 0**);**

**elsif** 'a' **<=** ch and ch **<=** 'f' **then**

tmpResult **:=** result**\***16 **+** character'**pos(**ch**)** **-** character'**pos(**'a'**)+**10 **;**

result **:=** tmpResult**(**31 **downto** 0**);**

**else** **report** "Format error on line " **&** integer'**image(**index**)**

**severity** error**;**

**end** **if;**

**end** **loop;**

-- set the width bit binary value in ram

RAM**(**index**)** **:=** std\_logic\_vector**(**result**);**

index **:=** index **+** 1**;**

**end** **loop;**

-- return the array of instructions loaded in RAM

**return** RAM**;**

**end** **function;**

## **Vivado Tips**: If You Have a “Missing File” in Vivado

Sometimes files may end up “missing” in Vivado and you need to remove them from your project. Right click and try to refresh the hierarchy.

## Important: Data Type Conversions in VHDL

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



# VHDL Test Bench Simulation

Now view the sources for Behavioral Simulation and look at the code for the test bench 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, **mips\_top**.

**Now run your simulation.** Find Simulation Sources, browses to **mips\_testbench**, select it, and then run the Simulation. If you zoom the output window as shown below you should be able to verify the hex codes of the instructions as they are read from memory and displayed on the output port (that would connect to a hex display on the NEXYS4 DDR board). Remember that the program has a jump to main at the end of the program and thus will continue to repeat.



# Use Vivado to Inspect the Instruction Memory

Let’s verify your that the hex code machine program was loaded into memory correctly! When you make your own microprocessor this will be the FIRST thing too look at… if your program does not load properly then there will be a mistake in your hex code file!

1. Set the run time to 10 ns.

1. Run for 10 ns.
2. Reset the simulation.



1. In the scope window drill down to imem1 and select it.
2. In the objects window expand mem, then scroll to the bottom of memory to see if the hex code instructions loaded from the data file correctly!

# Checking other Internal Signals during Simulation



1. Drill to the data path unit (dp) and click on **mainalu** to select it.
2. Click and drag the alucontrol, a[31:0], b[31:0], and result[31:0] signals to the simulation trace window.
3. Click and drag the alucontrol, a[31:0], b[31:0], and result[31:0] signals to the simulation trace window.
4. Now each time you click run for 10 ns, the simulation will display the instruction read from memory.

#### Changing the Machine Code Program & Predicting Control Signals

Now let’s add a different program to run on our hardware.

## Step 1: Copy the hex code for memfile\_2.dat, given below in Figure 2, and store this hex code into a file called memfile\_2.dat Do NOT copy the assembly language code shown here for mipstest\_2.asm!

**Figure 2. MIPS assembly program: mipstest \_2.asm and memfile\_2.dat**

**mipstest\_2.asm**

**memfile\_2.dat**

20020005

20070003

2003000c

00e22025

00642824

00a42820

10a70008

0064302a

10c00001

2005000a

00e2302a

00c53820

00e23822

0800000f

8c070000

ac470047

#

# Test MIPS instructions mipstest\_2.asm

# Assembly Code # Machine Code

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

addi $7, $0, 3 # 20070003

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

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

and $5, $3, $4 # 00642824

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

beq $5, $7, end # 10a70008

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

beq $6, $0, around # 10c00001

addi $5, $0, 10 # 2005000a

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

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

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

j end # 0800000f

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

end: sw $7, 71($2) # ac470047

## Step 2: Now, save the new file memfile\_2.dat file into your project folder: **…\MIPS3\mips\_fpga\_nexsys4\_ddr\mips.srcs\sources\_1\imports\new**

## Step 3: Add this file to your Vivado project as follows:

* 1. In Vivado, select the option to add files to your project, ( **add file** )
  2. Click on the file type field in the dialog that appears and scroll through the file types in order to select **all files**,
  3. Browse to the folder where you just copied the file: **…\MIPS3\mips\_fpga\_nexsys4\_ddr\mips.srcs\sources\_1\imports\new**
  4. Select the **memfile\_2.dat** file.
  5. After you have added the new **memfile\_2.dat** file to your project (it should appear in the hierarchy under “data files”).
  6. Now, in the source file: **mips\_mem\_instructions.vhd** edit the which currently loads **memfile.dat**  and change the code so it loads **memfile\_2.dat** instead:

-- use the impure function to read RAM from a file and store

-- in the FPGA's ram memory

**signal** mem**:** ramtype **:=** InitRamFromFile**(**"memfile2.dat"**);**

Change this to memfile2.dat. If you get an error, you either put this in the wrong folder, did not add it to your project, or have the name incorrect…

## Step 4: Predict the Control Signals for the MIPS Program

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 will happen for each instruction** when running the program. This means that you should follow any branches or jumps in your prediction! **To do this complete Table 1 on the following page**. You will most likely need to add more rows to complete this table.

**Table 1. Prediction (by hand) of control signals for assembly program memfile\_2.dat**

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 drill down and look at these signals and others.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **TIME** | **RESET** | **PCSRC** | **PC** | **BRANCH** | **INSTR** | **SRCA** | **SRCB** | **ALUOUT** | **ZERO** | **WRITEDATA** | **MEMWRITE** | **READDATA** |
| **28 ns** | **0** | **0** | **00000000** | **0** | **20020005** | **00000000** | **00000005** | **00000005** | **0** | **00000005** | **0** | **UUUUUUUU** |
| **38 ns** | **0** | **0** | **00000004** | **0** | **20070003** | **00000000** | **00000003** | **00000003** | **0** | **UUUUUUUU** | **0** | **UUUUUUUU** |
| **48 ns** | **0** | **0** | **00000008** | **0** | **2003000c** | **00000000** | **0000000c** | **0000000c** | **0** | **UUUUUUUU** | **0** | **UUUUUUUU** |
| **58 ns** | **0** | **0** | **0000000c** | **0** | **00e22025** | **00000003** | **00000005** | **00000007** | **0** | **00000005** | **0** | **UUUUUUUU** |
| **68 ns** | **0** | **0** | **00000010** | **0** | **00642824** | **0000000c** | **00000007** | **00000004** | **0** | **00000007** | **0** | **UUUUUUUU** |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

#### Record Actual Signals and Compare to Prediction

## Now, you will run the program in simulation and record the actual control signal values. You will compare your predictions to what actually happens.

## Displaying Signals in Vivado’s ISIM Waveform Trace Window

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. Run the simulation for 10 – 50 ns. Make sure your signals are appearing correctly.

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.

## Step 1: Setup the Waveform Trace with the Required Signals for Exercise 2

**For the final waveform that include in this document, show ONLY the following signals in this order: clk, reset, pcsrc, pc, branch, instr, srca, srcb, aluout, zero, writedata, memwrite, readdata**

## Step 2: Screen Snip the Waveform Trace Displays:

**For multi-bit signal values, make sure the signal trace shows these as hexadecimal values. They must also be readable to be awarded full credit!**

1. **Screen snip and insert a waveform trace from 0 ns to 60 ns here:**
2. **Screen snip and insert a waveform trace from 60 ns to 130 ns here:**
3. **Screen snip and insert a waveform trace from 130 ns to 170 ns here:**

## Step 3: Create Table 2 from the screen snip traces: Highlight any rows that are different from the predictions made previously.

Now given the waveform traces, fill out table 2 using the information from these traces. You may copy the data that is correct from table 1 given above. Highlight any rows that differed from your prediction given in table 1.

**Table 2. Actual signal values the assembly program memfile\_2.dat**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **TIME** | **RESET** | **PCSRC** | **PC** | **BRANCH** | **INSTR** | **SRCA** | **SRCB** | **ALUOUT** | **ZERO** | **WRITEDATA** | **MEMWRITE** | **READDATA** |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

#### Run MIPS single-cycle on NEXYS4 and Double Clock Speed

This folder has a MIPS processor wired to the FPGA 7-segment display. The 7-segment display shows the hex code of the machine code program that’s currently running.

* This has a process that generates a very slow clock (around 1 or 2 secs)
* Every clock cycle the processor outputs the instruction HEX code to the LEDs.
* **Switch 0** is the reset signal. If you switch it on, the program counter will reset to zero and the first instruction code will be displayed. Switch it off and the program will continue running.
* It also outputs selected bits to the single LEDs ( these can be changed and are helpful for debugging).

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… ☺

## STEP 1: Go ahead and synthesize the processor, and load the MIPS bit file into the fpga. Run the program and watch the LED’s flash.

Record the HEX codes here:

Do the instruction HEX codes match those in the simulation?

## STEP 2: Increase the clock speed to about half a second.

Identify the clock divider VHDL code that generates the clock signal for the mips processor. Double this speed from its current value. Synthesize and verify that the program runs twice as fast.

***Paste the code you modified to double the clock speed here:***

#### Exercise 4: Extending MIPS: Adding the xori and bne instructions to the MIPS single-cycle processor and Test Program Design

As a group you must work on this part of the project together. Make sure that each of you in your group have roughly equal time at the computer while the other person looks for errors, discusses the design process etc.

As a group refresh your memory of this paper: <https://collaboration.csc.ncsu.edu/laurie/Papers/Kindergarten.PDF>

* Share everything – work together at the same terminal and not separately.
* Play fair – don’t do all the work by yourself, OR hog the computer when typing.
* Don’t hit you partner – make sure to share tasks so one partner is not tempted to browse the web, text, etc. Give you partner the benefit of the doubt but also encourage them to contribute.
* Put things back where you found them – put away negative thoughts and don’t dwell on them. Focus on the positives.
* Clean up your own mess – pair programming will help you catch mistakes early and often.
* Don’t take things too seriously – have fun and be proactive.

Record the times spent working together here:

|  |
| --- |
| Meeting time 1: |
| Meeting time 2: |
| Meeting time 3: |

|  |
| --- |
| Meeting time 4: |
| Meeting time 5: |
| Meeting time 6: |

## Step 1: Directly modify the MIPS processor schematic (given here) to show what changes are necessary to the hardware to add XORI and BNE. Draw your changes directly onto this schematic.



**Figure 3: Single-cycle MIPS processor**

## Step 2: Modify the tables below for the main decoder and the ALU decoder in order to account for XORI and BNE. You may or may not wish to add more control signals, so there are extra columns added just in case you need them. You are not required to use them.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Op5:0** |  | **RegWrite** | **RegDst** | **AluSrc** | **Branch** | **MemWrite** | **MemtoReg** | **ALUOp1:0** | **Jump** |  | **Funct** |
| 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** | **1** | **1** | **0** | **0** | **0** | **10** | **0** |  | **100101** |
| bne | 000101 |  | **0** | **X** | **0** |  | **0** | **X** |  | **0** |  |  |
| xori | 001110 |  |  |  | **1** |  |  |  | **10** | **0** |  | **??????** |

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

## Step 3: Finally, modify the VHDL code as needed to allow the processor to work correctly. **Make sure a copy of your modified processor is on CS1 in your MIPS3 folder.** Add any code that you modified to this document directly here in step 3, with comments as to what you modified and why:

## Step 4: Group Project: Test your modified MIPS single-cycle processor

For this part of the project as a group you will create your own test program for the modified MIPS processor. You may if you wish add more instructions, but, you are not required to at this time. You may also if you wish modify the I/O for the processor, but, you are not required to at this time.

1. **Your group will need to generate your own original mips test program called mipstest\_3.asm Place the assembly code for your program here:**
2. **Generate the machine code for your original mipstest\_3.asm. Create a hex file called memfile\_3.dat that will contain the hexcode for mipstest\_3.asm described above. You will need to add this file to the Vivado project like you did earlier and further, you will need to modify the VHDL impure function code to load the file memfile\_3.dat. Include the hexcode directly here:**

**NOTE**: You may if you wish use PCSpim to generate part of the machine code for **memfile\_3.dat** from the file **test\_3.asm**. **However**, **remember that PCSpim will NOT give you the correct machine code for any of the jump or branching statements in your program! The addresses for these jumps and branches will be incorrect. Our machine code starts the text segment at address 0x00000000 instead of 0x00400000 also our memory starts at address 0x00000000 in data memory rather than 0x10000000.** You will need to figure out how to generate the correct the machine code instructions so that the code works on our mips machine!

1. **Generate a waveform that shows your test program is working correctly on the MIPS simulation. You must use the following signals (you may add more at the end of this list if you wish, but these must be in this order):** **clk, reset, pcsrc, pc, branch, instr, srca, srcb, aluout, zero, writedata, memwrite, readdata**i n that order. **Make sure all your waveforms are readable and show values in hexadecimal.   
     
   Take enough screen shots of the simulation trace (as you did before) and include them in this document directly here to demonstrate that your code is working. Make sure the waveforms are readable:**

## Step 5: Testing your modified MIPS program on the FPGA

Make sure your program can run on the FPGA. Remember the clock on the FPGA is slowed down in order to verify that the program works correctly. You may increase the speed of the clock on the FPGA, but, then you will need some hardware modifications to prove your program is working…

What to Turn In

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

1. This entire document with your entries filled out for **Exercises 1 to 3 (must be done individually)**.
2. **The group project. You must work together for the group project and record your time working together and the answers to the questions. Do NOT split the work, work separately, and then combine answers.**
3. **This time each individual must submit this document on blackboard directly individually.**
4. A complete **MIPS3** folder with the mips project code **vhdl** and **dat** files for the exercises AND for the group project in each of your individual MIPS3 folders folder on CS1.
5. DON’T FORGET: Follow instructions exactly for the simulation waveforms and tables. These are complex projects to implement and to grade. Because of the high standards we have for our CS students, we want you to do this correctly. For the waveforms, you will need to take multiple screen shots of the simulation trace and include them in this document as required.

#### Exercise 5: Code Walkthrough MIPS 3 Presentations

On the due date for this design, groups of two will take 5 minutes (max) and present their test algorithm design to the class. You will be graded on whether you present the following items. **Do not use more than 4 or 5 slides to summarize your algorithm.**

Minute 1: What hardware changes did you make to MIPS to accommodate XORI and BNE?   
 Show a diagram of the changes.

Minute 2: A neat assembly language listing of your MIPS test program. You must include comments.

Minute 3: A listing of the VHDL that was changed.

Minutes 4 - 5: A discussion of the process and result. A demo of your program running on the FPGA.

* What went well? What not so well?
* Did you get both simulation and FPGA working?
* Did you do anything above and beyond the requirements (not required)?