EECE 444

LAB 5: MIPS Single-Cycle Processor

Introduction

In this lab you will build a simplified MIPS single-cycle processor using Verilog. First you will design the single cycle control unit and combine your ALU from Lab 4 with the code for the rest of the processor from this lab. Then you will load a test program and confirm that the system works. Next, you will implement two new instructions, and then write a new test program that confirms the new instructions work as well. By the end of this lab, you should thoroughly understand the internal operation of the MIPS single-cycle processor.

Please read and follow the instructions in this lab carefully.

Before starting this lab, you should be very familiar with the single-cycle implementation of the MIPS processor. The single-cycle processor schematic is attached at the end of this lab assignment 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 units: the control and the datapath. Each unit 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 Lab 4, the register file, the sign extension logic, and five multiplexers to choose appropriate operands.

1. MIPS Single-Cycle Processor

The Verilog single-cycle MIPS module is uploaded on Blackboard. Copy them to your own lab5 folder.

Study the files until you are familiar with their contents. Look in mips.v at the `mips` module, which instantiates two sub-modules, `controller` and `datapath`. Then take a look at the datapath Verilog module. The datapath has quite a few submodules. Make sure you understand why each submodule is there and where each is located on the MIPS single-cycle processor schematic. You'll notice that the **`alu` module is not defined**. Copy your ALU from Lab 4 into your lab5 directory. Be sure the module name matches the instance module name (`alu`), and make sure the inputs and outputs are in the same order as they are expected in the `datapath` module.

After you are done studying the datapath, take a look at the controller module and its submodules. It contains two sub-modules: `maindec` and `aludec`. The `maindec` module will produce all control signals except those for the ALU. The `aludec` module will produces the control signal, `alucontrol[2:0]`, for the ALU**. The first part of this assignment** is to write these two Control submodules. When writing your code, correlate signal names in the Verilog code with the wires on the schematic. Start by filling out Table1 for the main decoder functionality and use Table 2 for the ALU decoder input `ALUOp`. Use X for any "don't care" signals. The internal structure of the control unit is shown in Figure 1.

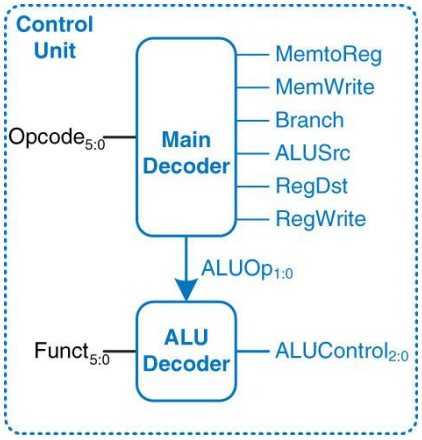


Figure 1: Control Unit internal structure

Table 1: Main Decoder Functionality

Instruction	$Op_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump			
R-type												
`lw`												
`sw`												
`beq`												
`addi`												
`j`												
`ori`												
`bne`												

Table 2: ALU Functionality

$ALUOp_{1:0}$	Meaning
00	Add
01	Subtract
10	Look at `funct` field
11	

The highest-level module, `top`, includes the instruction and data memories as well as the processors. Each of the memories is a 64-word × 32-bit array. The instruction memory needs to contain some initial values representing the program. The test program is given in Figure 2:

```
#          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
           sw   $7, 68($3)       # [80] = 7                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 mem[84] = 7       44         ac020054
```

Figure 2

Study the program until you understand what it does. The machine language code for the program is stored in memfile.dat (registers' names were emitted for simplicity). Examine the file carefully, you will be writing your own test file at the end of this lab.

2. Testing the single-cycle MIPS processor

In this section, you will test the processor with your controller and ALU.

In a complex system, if you don't know what to expect the answer should be, you are unlikely to get the right answer. Begin by predicting what should happen on each cycle when running the program. Complete the chart in Table 3 at the end of the lab with your predictions. What address will the final `sw` instruction write to and what value will it write?

Create a testbench "mipstest.v" and copy the code from the provided mipstest.v file into your file. Associate the testbench with the top Verilog file top.v.

Simulate your processor with Xilinx or ModelSim, whatever you prefer. Be sure to add all of the .v files, including the one containing your ALU. Add all of the signals from Table 3 to your waves window. (Not that many are not at the top level; you'll have to drill down into the appropriate part of the hierarchy to find them.)

Run the simulation. If all goes well, the testbench will print "Simulation succeeded." Look at the waveforms and check that they match your predictions in Table 3. If they don't, the problem is likely in your ALU, your controller or because you didn't properly add all of the files.

If you need to debug, you'll likely want to view more 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 values need to be output in hexadecimal and must be readable to get full credit.** After you have fixed any bugs, print out your final waveform.

3. Modifying the MIPS single-cycle processor

You now need to modify the MIPS single-cycle processor by adding the `ori` and `bne` instructions. First, modify the MIPS processor schematic at the end of this lab to show what changes are necessary. You can draw your changes directly onto the schematic. Then modify the main decoder and ALU decoder in Table 1 and Table 2, as required. Finally, modify the Verilog code as needed to include your modifications.

4. Testing your modified MIPS single-cycle processor

Finally, you'll need a test program to verify that your modified processor work. The program should check that your new instructions work properly and that the old ones didn't break. Use test2.asm shown in Figure 3.

```
# test2.asm
# 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)
```

Figure 3. MIPS assembly program: test2.asm

Convert the program to machine language and put it in a file named memfile2.dat. Modify imem to load this file. Modify the testbench to check for the appropriate address and data value indicating that the simulation succeeded. Run the program and check your results. Debug if necessary. When you are done, print out the waveforms as before and indicate the address and data value written by the `sw` instruction.

5. Synthesis

Modify the model by adding code to interface it to the input switches and LEDs on the Nexys4 board. Your interface must be able to halt operation of the MIPS processor and display the lower 16 bits of register $2 on 16 LEDs. Your interface must also divide the prototyping board's internal clock to provide the model with a slow clock.

1. You should add a new 'Halt' port to the MIPS top level. When Halt is high your processor should complete the current instruction and not proceed to the next instruction. When Halt goes back to zero you should keep executing the normal flow of instructions starting from the next instruction.
2. Map switches SW0 and SW1 to Reset and Halt, respectively.
3. Make necessary changes to your register file so that you can use register $2 as a top-level output and map it to LEDs [15:0].

4. The slow clock can be used to execute instructions in a manner that makes your output visible (when you implement this on the board). You may choose the frequency of the slow clock as you please. Using 100 MHz clock will make any program outputs on the LEDs blur.
5. You will not need to synthesize the testbench.
6. The instruction memory should be initialized with your instructions by using Verilog text-IO. Simply invoke the 'readmemh' function in an initial block (you should have done this already in the testing section).
7. Since the memory size if large and we will have a small number of instructions, you can use a for loop in the initial block to set the rest of the unused locations to zero.

What to Turn In

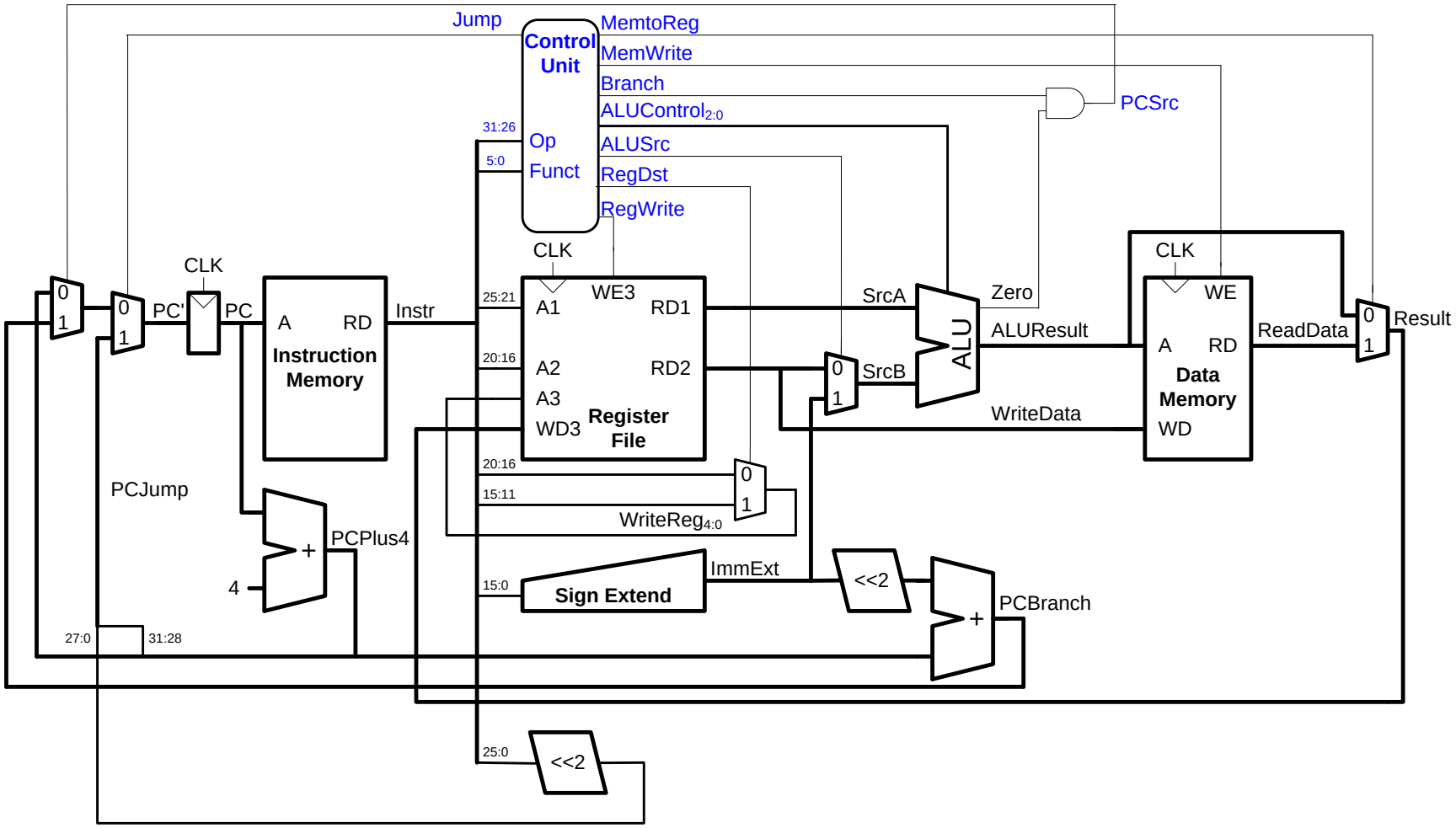
Please turn in each of the following items, clearly labeled and in the following order:

1. **Please indicate how many hours you spent on this lab.** This will not affect your grade (unless omitted), but will be helpful for calibrating the workload for next future labs.
2. A completed version of Table 3.
3. An image of the simulation waveforms showing correct operation of the processor. Does it write the correct value to address 84?

   The simulation waveforms should give the signal values in hexadecimal format and should be in the following order: `clk`, `reset`, `pc`, `instr`, `aluout`, `writedata`, `memwrite`, and `readdata`. Do not display any other signals in the waveform. Check that the waveforms are zoomed out enough that the TA can read your bus values. Unreadable waveforms will receive no credit. Use multiple images as necessary.
4. Marked up versions of the datapath schematic and decoder tables that adds the `ori` and `bne` instructions.
5. Your Verilog code for your modified MIPS computer (including `ori` and `bne` functionality) with the changes highlighted and commented in the code.
6. The contents of your memfile2.dat containing your test2 machine language code.
7. An image of the simulation waveforms showing correct operation of your modified processor on the new program. What address and data value are written by the `sw` instruction?

Table 3: First sixteen cycles of executing mipstest.asm

Cycle	reset	pc	instr	branch	srca	srcb	aluout	zero	pcsrc	writedata	memwrite	read data
1	1	00	`addi $2,$0,5` `20020005`	0	0	5	5	0	0	0	0	x
2	0	04	`addi $3,$0,12` `2003000c`	0	0	c	c	0	0	0	0	x
3	0	08	`addi $7,$3,-9` `20067fff7`	0	c	-9	3	0	0	0	0	x
4	0	0C										



Single-cycle MIPS processor