ECE 469 Spring 2024 Lab 5: MIPS Single-Cycle CPU

Grading components:

- **(40%) Demo - to TA during week 9 and 10 Friday Lab hours**:
  - Week 9: Complete Table 1 for test_a, and test_b.
  - Week 10: Demonstrate Part A.

- **(60%) On gradescope Mon of spring break, the following items:**

(20pts) Part A
1. A completed version of Table 1.
2. An image of the simulation waveforms showing correct operation of the unmodified 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: `reset, pc, clk, 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 grader can read your bus values. Unreadable waveforms will receive no credit. Use several pages and multiple images as necessary.

(30pts) Part B + C:
1. List of the new instructions implemented, their encoding, marked up versions of the datapath schematic and decoder tables that include the new instructions.
2. Your SystemVerilog code for your modified MIPS with the changes highlighted and commented in the code.
3. The contents of your memfile_b.dat containing your test_b machine language code.
4. (Part C) The contents of your memfile_c.dat containing your test_c machine language code.
5. Simulation waveforms showing correct operation of your modified processor on the testbenches test_b, test_c.

(10pts) Overall:
1. RTL view from Quartus' compilation result of your MIPS processor: before and after the modification. Identify and highlight the changes of your modification parts.
2. Workload report: How many hours have you spent for this lab? Which activity takes the most significant amount of time? This will not affect your grade (unless omitted).

Part A Basic MIPS

Introduction

In this part of the project you will build a simplified MIPS single-cycle processor using SystemVerilog. You will combine your ALU from the previous project with the code for the rest of the processor taken from the textbook. 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, you should thoroughly understand the internal operation of the MIPS single-cycle processor.

Please read and follow the instructions. In the past, many students have lost points for silly errors like not printing all the signals requested, or not putting them in the order requested.

Before starting this project, you should be very familiar with the single-cycle implementation of the MIPS processor described in Ch 7.3 of your textbook. The single-cycle processor schematic from the text is repeated at the end of this 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, 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, the datapath contains the 32-bit ALU that you designed in Lab 3, the register file, the sign extension logic, and five multiplexers to choose appropriate operands.

1.   MIPS Single-Cycle Processor

The SystemVerilog single-cycle MIPS module is given in Ch 7.6 of the textbook.

Study the code until you are familiar with their contents. Look into the `mips` module, which instantiates two sub-modules, `controller` and `datapath`. Then take a look at the controller module and its two submodules: `maindec` and `aludec`. The `maindec` module produces all control signals except those for the ALU. The `aludec` module produces the control signal, `alucontrol[2:0]`, for the ALU. Make sure you thoroughly understand the controller module. Correlate signal names in the SystemVerilog code with the wires on the schematic.

After you thoroughly understand the controller module, take a look at the `datapath` 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. Use your ALU module from your previous project here. Be sure the module name matches the instance module name (`alu`), and make sure the inputs and outputs are in the same order as in they are expected in the `datapath` module.

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 7.60 of the textbook. Study the program until you understand what it does. The machine language code for the program should be stored in a file called memfile.dat.

2. Testing the single-cycle MIPS processor

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 1 at the end of the assignment with your predictions. What address will the final `sw` instruction write to and what value will it write? Below is Fig 7.60 on memfile.dat as your test bench (we'll call it test_a.asm).

```
#           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
```

Simulate your processor with ModelSim. Be sure to add all of the .sv files, including the one containing your ALU. Add all of the signals from Table 1 to your waves window. (Note 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 1. If they don't, the problem is likely in your ALU 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:

`reset`, `pc`, `clk`, `instr`, `aluout`, `writedata`, `memwrite`, and `readdata`.

All the values need to be output in hexadecimal, in the above order, and must be readable to get full credit.

Note: You can use the MIPS simulator "MARS" to help generating hex code for assembly code using its "memory dump" functionality. To use MARS to run the assembly code, remember to activate the Settings -> Memory Configuration -> Compact, Data at Address 0 option.

Part B: MIPS Plus

Implement the following 2 instructions:

{bne, slti}

```
bne has    op = 000101
slti has   op = 001010
```

1. Modifying the MIPS processor

You now need to modify the MIPS single-cycle processor by adding the `bne`, `slti` instructions. First, modify the MIPS processor schematic (at the last page) to show what changes are necessary. You can draw your changes directly onto the schematic. Then modify the main decoder and ALU decoder as required. Show your changes in the tables at the end too. Finally, modify the SystemVerilog code as needed to include your modifications.

2. Testing your modified MIPS single-cycle processor

Next, 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. The above provides an example `test_b.asm` for the two additional instruction `bne` and `slti`.

Convert the program to machine language and put it in a file named `memfile_b.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.

```
# test_b.asm

L1:
addi $9, $0, 40
bne $9, $0, L2
addi $9, $9, 10
L2:
addi $8, $0,  -3
beq $9, $0, L2
slti $10, $8, -1
sw $10 12($9)
slti $10, $8, -5
loop:
addi $10, $10, 3
sw $10, -8($9)
addi $9, $9, -4
slti $11, $9, 30
bne $11, $0, out
j loop
out:
sw $9, 20($0)
```

Part C: MIPS Plus Plus

In addition to Part B, add the following instructions:

{sltu, ble}

```
sltu has op = 000000, func = 101011
```

sltu (set less than unsigned) should work like slt except assuming numbers to be
unsigned. For example:

```
if $8 = 0xFFFFFFFF, $9 = 0x00000000, then
slt $10, $8, $9 will result in $10 = 1 ($8 treated as negative number).
sltu $10, $8, $9 will result in $10 = 0 ($8 treated as a large positive
number).
```

ble (branch if less or equal) does not exist in standard MIPS instruction, thus you can
choose any unused op and format for its machine code. It should function as follows:

```
ble $8, $7, loop will go to loop when $8≤$7. Otherwise ($8>$7) go to the
next instruction.
```
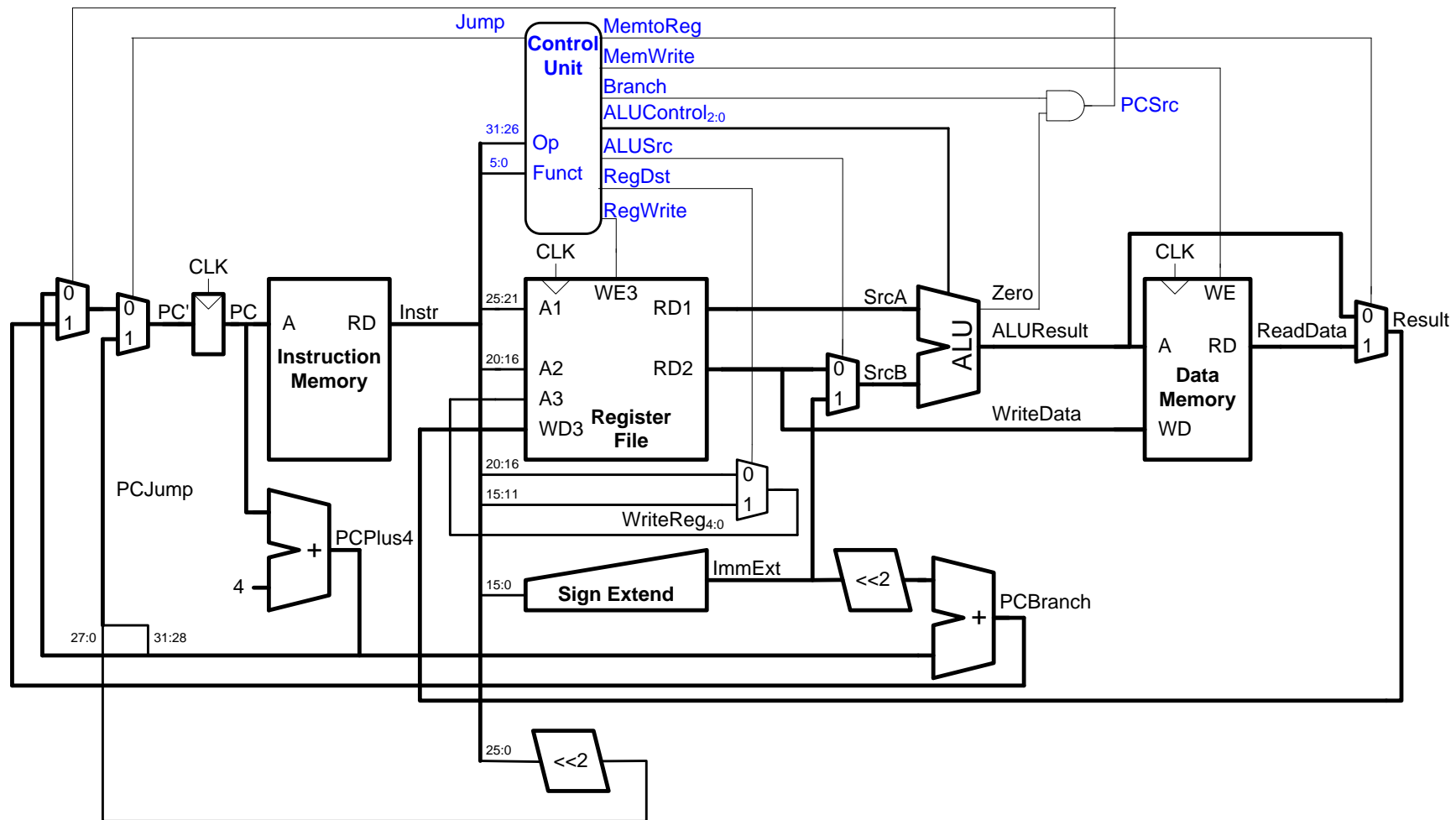
Use test_c.asm to verify your design.

```
# test_c.asm
addi  $8, $0,  3
addi $9, $0, 20
addi $7, $0, -4
loop:
sltu $10, $0, $8
slt $11, $0, $8
sub $10, $10, $7
sub $11, $11, $7
sw $10, 0($9)
sw $11, 4($9)
addi $8, $8, -3
addi $9, $9, 8
ble $7, $8, loop
sw $8, 16($0)
```

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										

Table 1. **First sixteen cycles of executing test_a.asm**

Single-cycle MIPS processor

Extended functionality. Main Decoder:

Instruction	Op$_{5:0}$	RegWrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUOp$_{1:0}$	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			
bne	000101											

Extended functionality. ALU Decoder:

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