Digital Logic Design – ECEN 3233 Spring 2009

Oklahoma State University

James E. Stine, Jr.

Project Part I: Building a PacMan Game

Introduction

In this first part of your project, you will build the MIPS single-cycle processor using Verilog that will be used to build one of the most popular video games in the market. It is important you try to complete this part to be able to move on to finish your project. More information will be given at a later time how to hand in the completed projected including all parts. Although this problem looks daunting it is not and this project will hopefully challenge you to see the true potential that digital systems can achieve along with a little hard work!

You will combine your ALU from Lab *5* with the code for the rest of the processor taken from the textbook. If you had problems with your code from Lab *5*, please see the TA who will help provide you with one to work with. Then you will load a test program and check that the instructions work. Next, you will implement two new instructions, 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, which is nothing more than a big complex digital system.

Before starting this lab, you should be familiar with 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 Figure *1* 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. The datapath is the major computation engine to your digital system, whereas, the control allows the datapath to work properly by controlling its function. The datapath is similar to your muscles and the control to your brain that controls the muscles to move. 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 *5*, 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 given in Section 7.6.1 of the text is the basic engine used to accomplish the task at hand. That is, it will be used to process each

command for the video game. In fact, the basic block in this lab is similar to current and past video gaming consoles. The project containing the module can also be found in the supplementary lab material available from the course website (inside the Lab 8 folder). Copy the folder to your own directory, so you can work on it.

Look at the `mips` module, which instantiates two sub-modules, `controller` and `datapath`. Then take a look at the controller module and its submodules. It contains two sub-modules: `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 Verilog code with the wires on the schematic.

After you have thoroughly understood the controller module, 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 has a question mark by it in the `Sources` window in Xilinx. You will need to add your Verilog module from lab 5. Do so by choosing `Project→Add Copy of Source` and add your `alu` module. 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. It is important to understand the instructions are nothing more than commands given to the digital logic to do some specific logic function. The commands or instructions are usually written within memory and called machine code. Each of the memories is a *64* word × *32*-bit array. The instruction memory needs to contain some initial values representing the program. The most convenient way to create a memory with initial values is with Xilinx's Core Generator.

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
#
# Test MIPS instructions.

#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
```

Figure 1. MIPS assembly program: test1.asm

This code can be found in the supplementary lab material provided on the course website. The code is written in plain simple assembly code, which is nothing more than commands to your hardware. The commands or assembly instructions get translated or encoded to Machine Code, which tells the digital logic to do something specific. Although naïve designers initially thought that hardware should be more complex to do vast logic functions, it turns out that computers are more efficient when instructions or commands are simple.

To use this test code, it must be loaded into the MIPS instruction memory. It would be nice to be able to write Verilog code for a memory with some initial values (the test program), but Xilinx does not support this very well yet. Instead, we will use the Xilinx Core Generator feature to produce a memory with initial values.

The instruction memory, `imem`, will be constructed as a CoreGen ROM that will hold the program (the instructions) to execute. Create the ROM by choosing Project->New Source. Select the source type to be `IP(CoreGen and Architecture Wizard)` and name your memory "`imem`". You can use CoreGen to generate different kinds of modules. In this case, choose `Memories and Storage Elements→RAMs & ROMs→Distributed Memory v7.1`. Click `Next→Finish`. It will now ask you to enter the component name ("imem"), depth (64), width (32) etc. Choose ROM type memory and make sure none of the inputs or outputs are registered. On the left side, a diagram of the symbol will be shown. A is the 6-bit address and SPO is the 32-bit data output.

Notice that only a subset of the PC bits ($PC_{7:2}$) will be used to address the memory. Be sure you understand thoroughly why only these bits are used. When finished, you can click "Generate" on the lower left corner, and a memory component will be generated and listed in the Generated Modules in the CORE generator. At the same time, a symbol with the same name will be available for use in your Verilog files.

To load the ROM with instructions, you need to use Memory Editor in the CORE generator interface to input the contents of the ROM. To do so, highlight your CORE generated module (i.e. imem) in the `Sources in Project` window in your main Project Navigator window. Then, under the `Processes for Source` window, expand the Coregen option and double-click on `Manage Cores`.

Choose Tools→Memory Editor, and you will see a window like the one shown in Figure 2. In the case of instruction memory, you will define a *64* word (although you will only use *16* of the available words) × *32* bit memory.

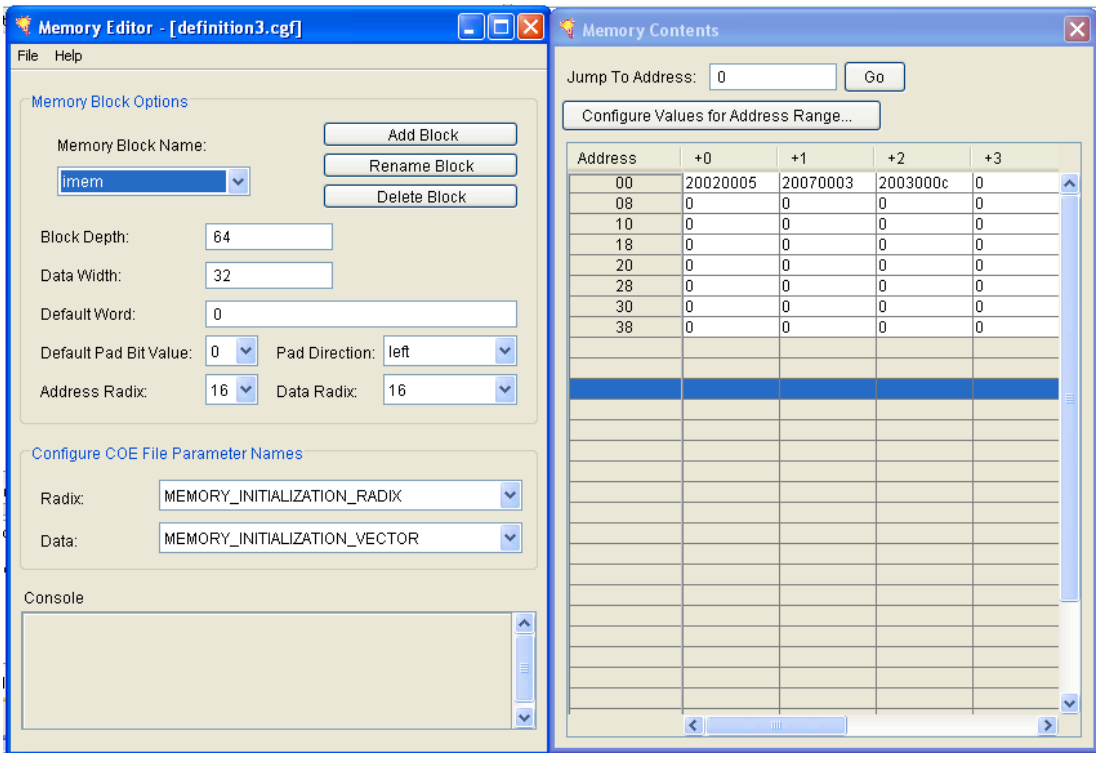


Figure 2. Memory Editor Interface

Click on "Add Block" to enter the Memory Block Name. Name it imem. Enter the Depth as *64* and the Data Width as *32*. Make sure the Radix for the Data is *16* (i.e. hexadecimal). Under Configure COE File Parameter Names, choose MEMORY_INITIALIZATION_RADIX for Radix and MEMORY_INITIALIZATION_VECTOR for Data. Now enter the instruction codes (given in Figure 2) into the contents of the ROM. For example instruction "20020005" is entered at Address 00 (Indicated as row 0, offset +0 in the window) in the ROM, instruction "20070003" is at Address 01, and so forth. The first three entries in the ROM are shown in Figure 4. The full set of entries is given in Figure 1 above. It is worth proofreading your entries because any typos will be very tedious to debug later.

When you finish entering all the machine instructions at the corresponding addresses, save your memory configuration by choosing `File→Save Memory Definition`. Now choose `File→Generate` to generate a ".coe" file that can be read by CoreGen. Click on the top option, COE File(s), and click OK. It should now inform you that defintion1_imem.coe has been generated. This file can be imported later to set up the complete 64 word × 32 bit instruction ROM. Now close the Memory Editor. You can now exit from the Memory Editor window by choosing `File→Exit`.

The main Core Generator window should still be open. Click on the "Generated IP" tab in the main window. Now right click on imem and select: `Recustomize→Under`

`Current Project Settings`. Click Next twice. In the last window, select `Load Coefficients`, and click on Load File. This will bring up a window that will allow you to browse for your ".coe" file that holds the definition of your ROM contents. Highlight the file and click Open. Then click Show Coefficients to spot-check that they match your expectations.

Now complete the setting of the ROM by clicking the "Generate" button. By doing so, the ROM "imem" is created with the stored instructions you entered and a template with the same name will also be ready to use in Verilog files.

When you view the imem module in your Processes in Source window, there should now no longer be a question mark next to it. If there is still a question mark, you need fix your errors. You can edit your core generated module by double-clicking on it. You can also look at the equivalent Verilog code by choosing `CoreGen`→`View Verilog Functional Model`.

3. Synthesis

Synthesize the highest level module, `top` although this may take a little bit of time. Notice that the only necessary inputs to the highest level module are clk and reset. The other signals are there for verification purposes only.

View the synthesis report. There should be no errors or warnings (you can confirm this by looking at the errors tab at the bottom of the screen).

View the RTL schematic. If it does not look as you expected, fix the errors and resynthesize.

4. Testing the single-cycle MIPS processor

To test the processor, you will simulate running test1.asm on the Verilog code. 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 lab with your predictions. What address will the final `sw` instruction write to and what value will it write?

View the sources for Behavioral Simulation and look at the testbench module. This Verilog code is for testing your module only and is not synthesizeable. 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. Make sure the testbench file is highlighted in the Sources window and click on ModelSim Simulator -> Simulate Behavioral Model. If the simulation fails being unable to find the imem, go to the Sources for Behavioral Simulation panel, remove imem.xco, and add imem.v (which was generated by CoreGen). Then close ModelSim and try simulating the testbench again. If you should need to synthesize later, replace imem.v with imem.xco again.

Now, you can view your simulation. It should print "simulation succeeded" and display the correct values of the major internal busses. If it does not perform as you expected, check your Table 1 entries and see if you made a mistake. You can also check your coefficients file that you entered into the instruction ROM (imem). You can do so by opening the file in the memory editor. (To do so, highlight the file in the Sources window and choose Manage Cores in the Processes window. Now choose Tools -> Memory Editor. Now choose File -> Open Memory Definition. Edit the definition if needed.)

If there are no errors in either your table or ROM, modify your Verilog code and fix any bugs. Good engineers know that designs never work the first time, so please try to allocate time for debugging. For debugging, you will likely need to make other signals from sub-modules visible in the higher-level module. View the "Workspace" window in the main ModelSim window. Expand the hierarchy of the modules until you find the module that has the internal signal you would like to view. Double-click on the module and the corresponding Verilog file will open in the far-right window and an objects window listing all the signals will open. You will likely need to expand the objects window to be able to read the signal names. Click on the signal you would like to view in the waveform window and drag it to the waves window. After you have dragged the signals you would like to view to the waves window, you will need to resimulate. You can refer to previous lab handouts if you've forgotten how to do this. (Hint: `restart -f`, then `run 1000`).

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 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.

5. Modifying the MIPS single-cycle processor

You now need to modify the MIPS single-cycle processor by adding the `lbl` and `sll` instructions. First, modify the MIPS processor schematic (on 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 of the lab. Finally, modify the Verilog code as needed to include your modifications. Please refer to sections 6.2, 6.3, and 6.4 of your textbook for detailed explanation of these instructions.

6. Testing your modified MIPS single-cycle processor

Enter the program, test2.asm, given in Figure 5 into your instruction memory (imem). You need to figure out what the machine code is for the instructions. Also comment each line of code of test2.asm. You can use PCSpim, a freely available tool on the Internet, to help convert to machine code, but remember that PCSpim does not give the correct machine code for `beq` and that the address for jumps is incorrect. Or, you can examine Appendix B in your textbook to find the corresponding opcode and RTL documentation. Also, you'll need to modify the Verilog testbench to check for the correct "Simulation Succeeded" values at the end of the program. A testbench and DO file have already been included for you on the ecen3233 website that might be helpful. You might also find it useful to create a table similar to that of Table 1.

```
# Adapted from 'test2.asm', originally written by David M. Harris.
# Test MIPS instructions for the Extended Verilog code.

#Assembly Code
main:       addi $t0,  $0, 0x8000
            addi $t1,  $0, -32768
            or   $t2, $t0, $t1
            beq  $t0, $t1, there
            slt  $t3, $t1, $t0
            beq  $t3,  $0, here
            j    there
here:       sub  $t2, $t2, $t0
            addi $t3, $t2, 7
there:      sw   $t2, 40($0)
            lbu  $t2, 42($0)
            sll  $t4, $t2, 8
            sw   $t4, 44($0)
```

Figure 3. MIPS assembly program: test2.asm

Again, for debugging, you might find it useful to make other signals from sub-modules visible in the higher-level module. The DO file labels each signal for its respective

module (i.e. datapath vs. control) and it may be prudent to inspect how this is done inside the DO file. However, in the final waveform that you turn in, 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.

What you should get working

Please try complete the following tasks by April 4, 2008 or as soon as you can. The next section of the project should be posted by next week allowing anyone to get a jump on the next section. The tasks below will help you understand the ideas presented in this project part I.

1. A completed version of Table 1.
2. Marked up versions of the datapath schematic and decoder tables that adds the `lbu` and `sll` instructions.
3. A printout of your Verilog code for your modified MIPS computer (including `lbu` and `sll` functionality) with the changes highlighted:
4. Simulation waveforms of:
   - top.v for test1.asm
   - top.v for test2.asm (with your modified single-cycle MIPS processor)

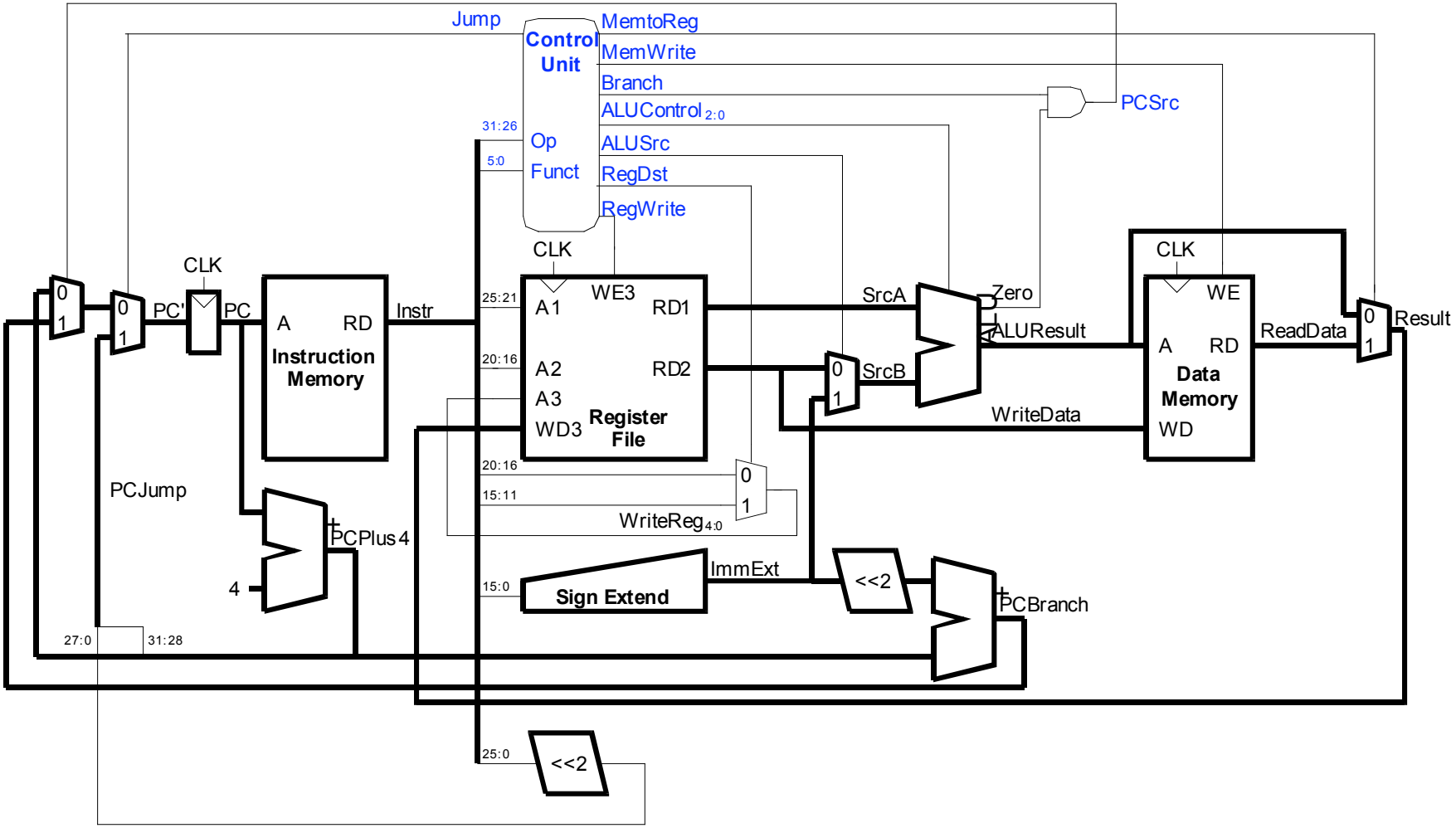
   The simulation waveforms should give the signal values in hexadecimal format and should be in the following order: `clk`, `reset`, `pc`, `instr`, `aluout`, `writedata`, `memwrite`, and `readdata`. Do not display any other signals in the waveform.

   Be sure the waveforms match your expectations. Check that the waveforms are zoomed out enough that you can read your bus values.

Cycle	reset	Pc	Instr	branch	srca	srcb	Aluresult	zero	pcsrc	write data	Mem write	read data
1	1	00	`addi $2,$0,5` `20020005`	0	0	5	5	0	0	0	0	0
2	0	04	`addi $7,$0,3` `20070003`	0	0	3	3	0	0			0
3	0	08	`addi $3,$0,0xc` `2003000c`	0	0	12	12	0	0			
4	0	0C										
5												
6												
7												
8												
9												
10												
11												
12												
13												
14												

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 module (mips). For debugging, you might want to be able to look at these signals and others.



Single-cycle MIPS processor

Extended functionality. Main Decoder:

Instruction	$Op_{5:0}$	RegWrite	RegDst	$AluSrc_{1:0}$	Branch	MemWrite	MemtoReg	$ALUOp_{1:0}$	Jump			
R-type	000000	1	1	00	0	0	0	10	0			
`lw`	100011	1	0	01	0	0	1	00	0			
`sw`	101011	0	X	01	0	1	X	00	0			
`beq`	000100	0	X	00	1	0	X	01	0			
`addi`	001000	1	0	01	0	0	0	00	0			
`j`	000010	0	X	XX	X	0	X	XX	1			
`lbu`	100100											
`sll`	000000 (R-type)											

Extended functionality. ALU Decoder:

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