Digital Design and Computer Architecture

Harris and Harris, © Elsevier, 2007

MIPS Single-Cycle Processor

(Truncated document from above source)

Introduction

You will combine your ALU with the code for the rest of the processor taken from the textbook. 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.

Please read and follow the instructions in this lab carefully. In the past, many students have lost points for silly errors like not printing all the signals requested.

Before starting this lab, you should be very 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. 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. The project containing the module can be found in the supplementary lab material available from the textbook website (or in the class directory). Copy the Lab08 folder to your own directory. Now open lab08_xx.ise from your local folder.

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. 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

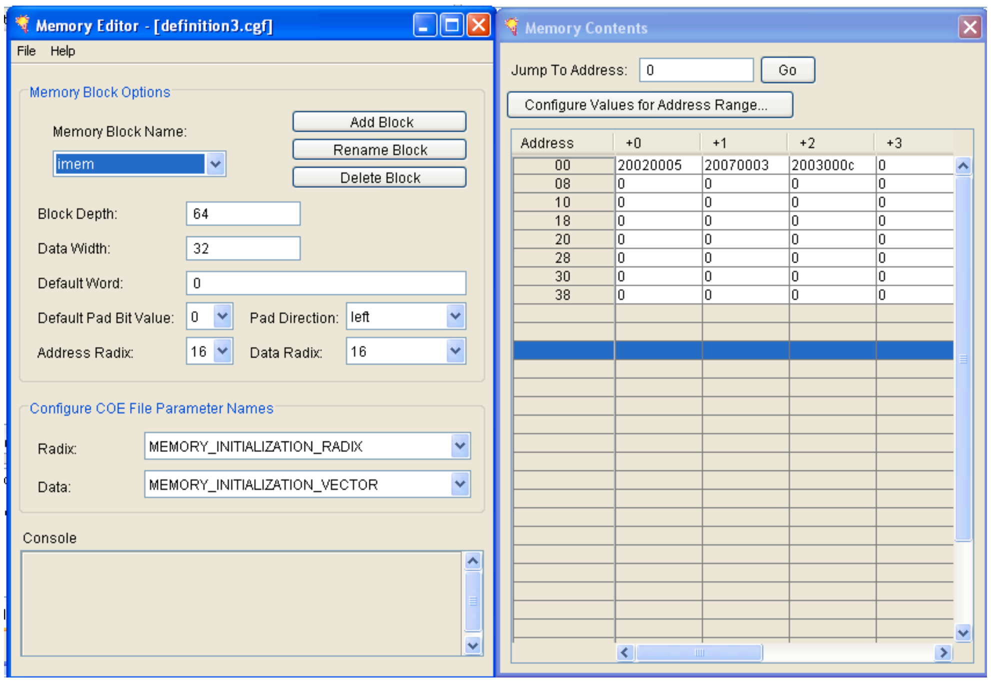
This code can be found in the supplementary lab material provided on the textbook website under the Lab08 directory.

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 Synplify Pro and Xilinx do 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`. 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.


Memory Editor - [definition3.cgf]
File
Help
Memory Block Options
Add Block
Rename Block
Delete Block
Configure COE File Parameter Names
Console
Memory Contents
Go
Configure Values for Address Range...

Memory Block Name: imem

Block Depth: 64

Data Width: 32

Default Word: 0

Default Pad Bit Value: 0   Pad Direction: left

Address Radix: 16   Data Radix: 16

Radix: MEMORY_INITIALIZATION_RADIX

Data: MEMORY_INITIALIZATION_VECTOR

Jump To Address: 0

Address	+0	+1	+2	+3
00	20020005	20070003	2003000c	0
08	0	0	0	0
10	0	0	0	0
18	0	0	0	0
20	0	0	0	0
28	0	0	0	0
30	0	0	0	0
38	0	0	0	0

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 Address 00, 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. 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 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 Next 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`. 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. 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` or click on the restart and run buttons).

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 `ori` and `bne` 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.

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 to help convert to machine code, but remember that PCSpim does not give the correct machine code for `beq` or `bne` and that the address for jumps is incorrect. Also, you'll need to modify the Verilog testbench to check for the correct "Simulation Succeeded" values at the end of the program. You might find it useful to create a table similar to that of Table 1.

```
# 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)
```

Figure 3. MIPS assembly program: test2.asm

test2.asm can be found in the supplementary lab material on the textbook website under the Lab08 directory. Again, for debugging, you might find it useful to make other signals from sub-modules visible in the ModelSim waveform. 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.