Lab 4'b0011

Deborah Hellen, Bernard Kahle, Ruby Spring

December 5, 2014

1 Introduction

The goal of this lab is to create a Central Processing Unit that allows us to run a simple program written in MIPS assembly. We use Verilog to write our CPU and its constituent components and Xilinx to synthesize it.

2 Assembly Program

We wrote a simple program in MIPS assembly that recursively calculates the Fibonacci number of 4 and of 10 and then adds them together and stores the result in $v0. Figure 1 below shows a simple diagram of the three addresses on the descending MIPS stack used in our program. The generic MIPS register names written in each address correspond to the order and content of the data pushed to the stack in the assembly code.

Figure 1: The order in which register data is pushed to and popped from the stack is indicated in the diagram above.

To execute this program, our CPU needs to be able to complete the following instructions:

1. add
2. add immediate
3. add immediate unsigned
4. set if less than immediate
5. store word
6. load word
7. jump and link
8. jump to return address
9. branch if equal
10. branch if not equal

3 CPU Design

Based on the relatively small number of commands that we need to execute to complete our assembly program, we decided to design a single-cycle CPU. We made this decision because we felt there would be little gain in the small latency reduction allowed by a mutli-cycle cpu running a single program containing only 10 instruction types. Fig. 2 below shows the layout of our single-cycle CPU design.

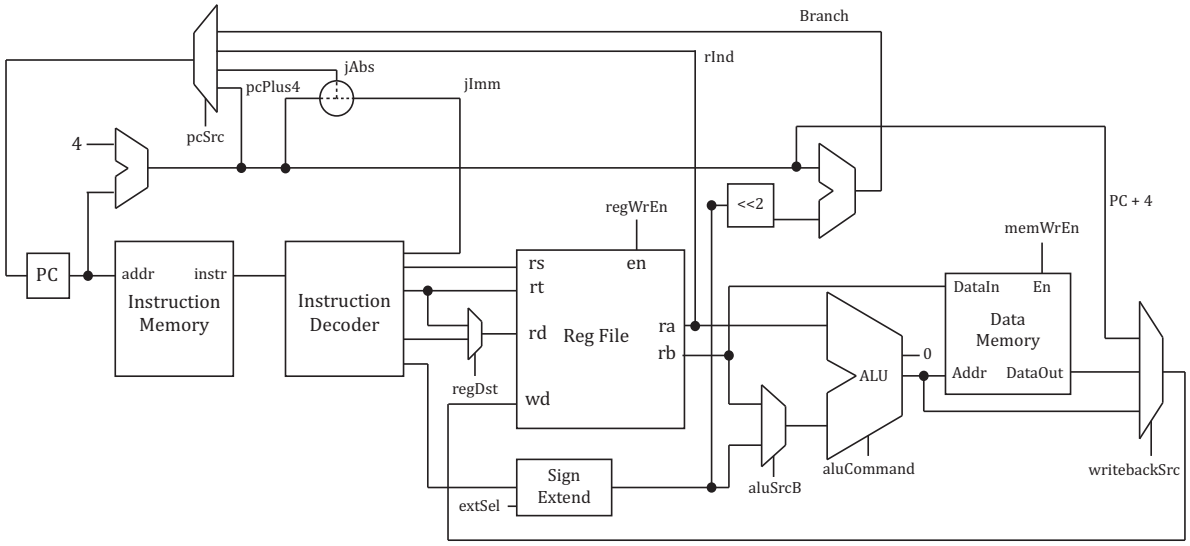

Figure 2: Shows the overall block diagram and data paths in our single cycle CPU. The design is as simple as possible; it's capabilities are limited to the instruction set required for running the Fibonacci assembly code.

Design Statistics from Xilinx

1. Resources Used in our Design: One 4000X32bit RAM, Two 32-bit adder, 32-bit adder/subtractor, 1024 flip-flops, two 32-bit latches, one 320bit comparator, one 32-bit 3 to 1 mux, two 32-bit 32 to 1 muxes, one 32bit 4 to 1 mux, and 1 logical left shifter
2. Maximum Clock Speed: not available
3. Total MIPS: not available

These design statistics are based on what we have as of 12/4/2014 at 10:40 PM. The CPU is not in its final working state and therefore these are not representative of a working Single-cycle CPU. Unfortunately, our Xilinx synthesis was unable to detect a clock in our design, so the maximum clock speed and the total MIPS were not available.

4 Test Strategy

Each major component of the CPU includes a test bench which we use to test the module in isolation. These tests are designed to go through a comprehensive sampling of all possible states and expected outputs of each component, and also check that it doesn't do anything extra. To test each component individually, run "do <component_name>.do"; for example, to run the test bench for the ALU, run "do alu.do".

We chose not to write separate test benches for each individual mux, the basic adders, the program counter, the left shift by two, or the jump concatenation bus because these were each simple modules with only a few lines of code that were much like or identical to ones that we have written and used successfully in previous labs.

4.1 Instruction Memory

To test the instruction memory module, we wrote a simple test bench that cycles through the instruction memory addresses and checks that each of the 43 commands in our assembly code is present at the expected address. If at any point in the cycle the memory at the address does not match the expected command, the test bench prints out an error detailing the expected command, the actual command, and the address at which the test failed.

Once we had addressed syntax errors, we did not find any errors in the instruction memory itself because we used a slightly modified version of the instruction memory provided and it was quite easy to implement.

4.2 Instruction Decoder

To run this test bench, use the command "do instructionDecoder.do" in our main CPU directory. If the test bench catches any errors, it will display an error message. If the test bench catches no errors, it will print out nothing.

Before writing a test bench in Verilog, we went through each possible case of the Instruction Decoder and wrote down what each control line or command should be set to for that case. This exercise allowed us to identify and notice some errors and forgotten control lines even before we ran a test bench or attempted to compile the module.

We then implemented a test bench in Verilog that cycles through each of our cases (each case is a MIPS op code) in our instruction decoder and checks that all of the appropriate control lines and flags are set. If the test bench finds an error, it prints out an error message to the command line detailing on which instruction the Instruction Decoder failed and gives the expected and actual output. An example error message is "beq (not equal) Failure: rs is 00011 when expected 00111." This allows us to see specifically which part of the instruction is not being read correctly. This proved useful for quick, easy debugging of this module.

Types of errors this can allow us to find:

1. This will allow us to determine if we have neglected to set or define all of our necessary control or command lines.
2. It will (and did) allow us to find incorrect commands.
3. This will also allow us to determine if any of our outputs of the instruction decoder are not the correct width.

4.3 Register File

To run the register file test bench, run "do registerFile.do" in the main CPU directory. If this test has run successfully, it will print out a "1" to the console.

We used Ruby's register file from the hw4 assignment in our CPU. To test this module, we used the test bench provided in the hw4 assignment with a couple of extensions. There are six tests in total. The first two test cases write data to a specific register in the Register File and then read back the data using the read ports to check that it has been correctly written. The third case tests the failure mode in which the write register is broken and is always written to. The fourth case tests the failure mode in which the decoder is broken and data is written to all of the registers, rather than only the specified one. The fifth case tests checks that Register Zero is always set to 0 and that we cannot write data to Register Zero. The sixth test case checks the failure mode that Port 2 always reads from Register 17 regardless of the specified register. If all test cases pass, dutPassed = 1. The waveform output of this test bench is shown in Figure 3. In this waveform, all of the tests have passed and therefore dutPassed is set to at the end of the test sequence.

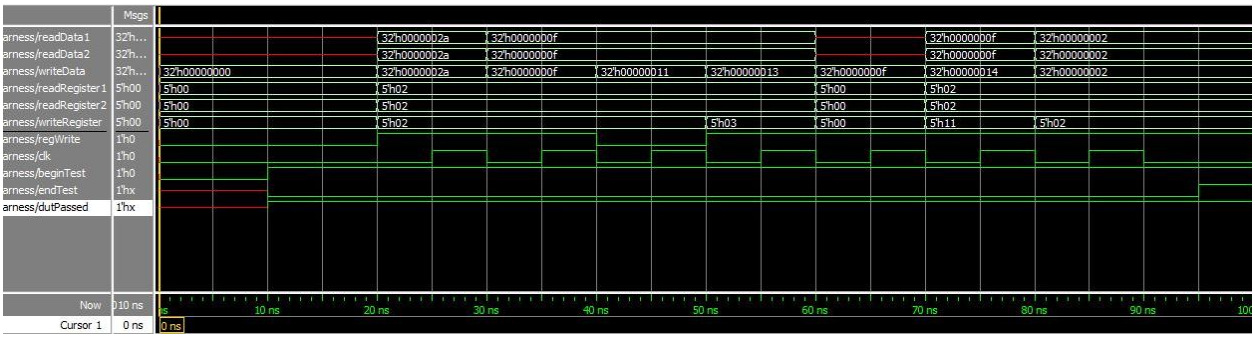

Figure 3: Shows the waveform of the test bench for the Register File. At the end of the series of six tests, the dutPassed register is set to 1, indicating that all of the tests passed.

4.4 Sign Extender

To run the test cases for this module, run the command "do signExtend.do" in our CPU directory. If the test has run successfully, the test bench shown in Figure 4.

We use the sign extender to extend the immediate from a 16-bit integer to a 32-bit integer. To test the sign extender, we three cases of negative sign extension and one case of positive sign extension for both signed and unsigned extension. We also check that the 'extSel' flag is set appropriately; it should be high for all of our unsigned extensions and low for signed extensions. We need both positive and negative extension in our CPU because the 16-bit immediate can be positive or negative. We also need unsigned extension because we use the add immediate unsigned command in our Fibonacci program.

Fig. 4 shows the output of our test bench for the Sign Extender. The test bench shows that the sign extender works appropriately for these eight examples, which represent the relevant cases that the module must handle.

```
# Test Cases:
# Type       |    16-bit Input    |      32-bit Output                 |  extSel
# Signed     |  1111111111111000  |  11111111111111111111111111111000  |  0
#
# Signed     |  1111111111100010  |  11111111111111111111111111100010  |  0
#
# Signed     |  0000000000000111  |  00000000000000000000000000000111  |  0
#
# Signed     |  1000000000000000  |  11111111111111111000000000000000  |  0
#
# Unsigned   |  1000000000000000  |  00000000000000001000000000000000  |  1
#
# Unsigned   |  1111111111100010  |  00000000000000001111111111100010  |  1
#
# Unsigned   |  0000000000000111  |  00000000000000000000000000000111  |  1
#
# Unsigned   |  1000000000000000  |  00000000000000001000000000000000  |  1
# 0 ns
# 1680 ns
```

Figure 4: Shows the output of our test bench for the sign extender.

Types of design errors this can help us detect:

1. This test bench will also help us ensure that we always have a 32-bit immediate which will then go into the aluSrcB mux.
2. This will (and did!) allow us to notice that we were setting "extSel" when we were not supposed to.

4.5 ALU

To run the test bench for the ALU, run the command "do alu.do" in the main CPU directory. If teh test has run correctly, the test bench should appear exactly as shown below in Figure 5.

We chose to re-write an ALU in behavioral verilog rather than using one of the structural ALUs that we created in mp1. We chose to do this so that we would have simpler, more readable code to debug. This also allowed us to only include the operations in the ALU that we need for our specific instruction set. These operations are Add, Subtract, and Set if Less Than. We wrote a series of test cases that demonstrates appropriate functionality for each of these three important operations.

For the Add operation we test one case that should have carryout and no overflow, one case that should have carryout and overflow, one case that should have no carryout and no overflow, and one case that should have overflow and no carryout. We also test a fifth case in which we expect to have both overflow and carryout and also for the zero flag to be set.

For the Subtract operation we test one case in which we subtract two negative numbers and one case in which we subtract two positive numbers. We chose not to explicitly check the carryout and overflow cases for the Subtract operation because carryout and overflow are handled using the same code that handles carryout and overflow for Addition, which has already been tested.

For "Set if Less Than" we expect the least significant bit to be set to 1 if A is less than B and all other bits of the output to be 0. In our test bench, we test a case where A is positive and B is negative, a case where A is positive and B is positive (but A is greater than B), and a case where A is negative and B is positive. For the first two cases, we expect the least significant bit to be 0 and for the third case we expect the least significant bit to be 1. Our expectations match the results shown in the figure.

```
# Command |                A                                 B                 | Cout               result               | Zero  OFL  | What are we testing?
#   000   | 11101010111011000010000100011011  11110011101101101110100000100010  |  0   11011110100110011000100100111101  |   0    0   | Addition
#
#   000   | 00010000000000000000000000000000  11110000000000000000000000000000  |  0   00000000000000000000000000000000  |   1    0   | Addition
#
#   000   | 00000000000000000000000000000010  00000000000000000000000000000010  |  0   00000000000000000000000000000100  |   0    0   | Addition
#
#   000   | 10000000000000000000000000000000  10000000000000000000000000000000  |  0   00000000000000000000000000000000  |   1    0   | Addition
#
#   000   | 01010000000000000000000000000000  01010000000000000000000000000000  |  0   10100000000000000000000000000000  |   0    0   | Addition
#
#
#   001   | 01101101001111000011101001011010  01111000110000100010101001100001  |  0   11110100011110100000111111111001  |   0    0   | Subtract
#
#   001   | 11011110011111000011110100100100  11101110110101010001010000100  |  0   11110001000100011011001010100000  |   0    0   | Subtract
#
#
#   011   | 00010011111011100001110000000111  11110100010101010110010100100011  |  0   00000000000000000000000000000001  |   0    0   | SLT
#
#   011   | 00000000000000000000000000000011  00000000000000000000000000000010  |  0   00000000000000000000000000000000  |   1    0   | SLT
#
```

Figure 5: Shows the output of our test bench for the ALU.

4.6 Data Memory

We tested our data memory by running through three test scenarios. First, we ensure that there is no data written when 'regWrEn'is set to 0 by checking that dataOut is not equal to dataIn. Second, we check that if regWrEn is set to 1, dataIn is equal to dataOut, which indicates to us that data has been successfully written. Third, it checks that data is not present in an empty address by checking that dataOut of the known address is equal to 0. If any of these test cases fail, the relevant error details are printed to the console output.

4.7 CPU

The test bench for the entire CPU can be run with "do DoFibonacci.do". This test bench relies heavily on generating and displaying waveforms to ensure that elements of the CPU are being accessed at the

correct time. Specifically, we walk through each clock cycle of the waveform and compare each signal to the values that Mars reports should be being written, read, or computed for each associated instruction. Using the waveforms allowed us to catch timing issues, where values were changed in clock cycles later than intended. Additionally, by displaying most of the current values within the CPU we were able to notice small errors like an incorrect ALU command. The do file will display these waveforms when run in ModelSim.

Some of the various types of design errors that our test bench collection can catch:

1. Well-written memory tests help us catch data leaks.
2. Displaying waveforms for individual components and the overall CPU allows us to catch timing inconsistencies and errors.

5 Debugging

First, we ensured that each of our individual components worked in isolation by writing and running robust test benches that allowed us to quickly debug and check outputs on the console and in waveforms. After we ensured that each module worked in isolation, we wired together the entire system according to our diagram in Figure 2.

Initially, we had a number of syntax errors in our main CPU due to naming inconsistencies across many different files. We fixed this by converting all of our files and module names to camel case and adhering strictly to this naming scheme.

After dealing with the syntax errors that prevented compilation, we debugged the CPU by generating and checking waveforms. By looking at waveforms, we noticed that the program counter was only being updated every other clock cycle.

During the debugging process we realized that we had not carefully considered which elements needed to be clocked and which did not when creating our overall system diagram. Initially, we had only the program counter clocked, but this produced a number of timing problems. Deciding which elements to clock and which ones to leave un-clocked was one of the major challenges in our debugging process.

6 Work Plan Analysis

Attached in our supplementary folder is our final work plan update. For the most part, we stuck to our work plan very well. Until we had to stitch the entire design together, we were well on schedule and relatively close to our expected deadlines. Unfortunately, after stitching together the whole CPU, our debugging process took far longer than we expected. After we had tested each unit in isolation, we spent approximately 3 hours on 12/3/14 attempting to debug the CPU and then another 10 on 12/4/14. Unfortunately this did not result in a perfectly working CPU. Our actual total time was over twice our expected total time.

7 Conclusions and Future Work

Our CPU is close to functional, but currently it gets stuck in an infinite loop and never accesses the $stop command.

The majority of our debugging time in this lab was spent addressing timing issues. This highlighted for us the extreme challenge that timing presents, even in a single-cycle CPU design.

We believe that the issue our CPU has is one that Eric briefly mentioned in class on 12/4/2014. We appear to be "silently" generating some registers which cause some data writes to be delayed by one clock cycle. However, we cannot find an elegant solution to this issue without causing our Verilog code to run in

an infinite loop.

To extend this lab, we could create a Multi-Cycle CPU. This would require dealing with hazards that we did not directly deal with in this lab such as Exec being used by Store and R-type at the same time.