Lab 2: Program Counter

Madeline Stephens and Michael Hu

February 20, 2017

1 Introduction

Building upon the knowledge of registers established in Lab 1, Lab 2 addresses the program counter – a one word long register. This special register holds the address of the next instruction to be executed by the computer. While this is a seemingly simple function, the program counter must be able to work under a few different conditions. The program counter must be able to advance to the next instruction address with each clock cycle. However, this is complicated when unconditional branching, conditional branching, or interrupts are implemented. These commands require the program counter to move to an address that may not be the next in sequential order. To help handle these various situations, an incrementer and input selector are required. These components were fulfilled with a simple adder and multiplexer, respectively.

2 Interface

A simple adder was used to created an incrementor for the program counter. The adder consisted of two inputs and one output. The output of the adder, as could be expected, was the sum of the two inputs. It was required that the output of the incrementer be a wire, due to the fact that it must continuously assign a value. For a program counter to operate correctly, it must be able to decide between executing normal sequential stepping and branching. A basic multiplexer was used to fulfill this task. The mux used in this lab had two inputs, a selector, and an output. Based on whether the selector is set or not, the multiplexer connects one of the inputs to the output. This can be thought of like a railroad switch. Eventually, the multiplexer will be used to discern the correct value to pass into the program counter register. The program counter register will then pass its value to the adder (or incrementer) to get the address of the next instruction in memory. The adder and multiplexer were built using Verilog code that utilized the definintions file and were provided by Dr. Schubert.

3 Design

This lab focused on creating and testing the basic components needed to create a basic program counter. When combined with the program counter register, the simple adder and multiplexer will make up a program counter that can operate under various circumstances. Due to the multiplexer, the program counter will be able to function when faced with branching, looping, or any command that requires it to move to a non-sequential memory address.

4 Implementation

The incrementer was created by programming a simple adder using Verilog. This code can be found in Listing 1 on page 2. The adder must simply be passed a one, due to the fact that the memory of the computer being built will be word addressable. Additionally, all inputs and output of the adder are of size WORD due to this fact. Because the output of the adder must be continuously driven, it is important to note that is must be defined as a wire. In this case, this was done using a simple assign statement.

While slightly more involved, the multipexer code was still very simple to implement. The multiplexer code can be reference in Listing 2. Unlike the adder, the multiplexer had a size parameter. This allows the code to be reused, as only the parameter needs to be changed in order to change the size of the multiplexer. However, one must note that parameters can not be changed later with code. The multiplexer took in two inputs, a selector, and produced one output. The selector (referred to as "control" hereafter) is used to determine while input will be connected to the multiplexer output. The ternary operator in the output assign statement treats control as a Boolean statement. If control is HIGH, then Bin is connected to the output; if control is LOW, Ain is connected to the output. This function will allow the program counter to chose between simple incrementing or jumping to an address following a branch.

Listing 1: Verilog code for implementing an adder.

```
`include "definitions.vh"

module adder(
    input [`WORD-1:0] Ain,
    input [`WORD-1:0] Bin,
    output [`WORD-1:0] add_out
    );
    assign add_out = Ain+Bin;
endmodule
```

Listing 2: Verilog code for implementing a mux.

```
`include "definitions.vh"
```

```
module mux#(
    parameter SIZE=8)(
    input [SIZE-1:0] Ain,
    input [SIZE-1:0] Bin,
    input control,
    output [SIZE-1:0] mux_out
    );
    assign mux_out = control?Bin:Ain;
endmodule
```

5 Test Bench Design

To ensure the proper function of the aforementioned code, test benches were created and simulated. Various cases were used to test each module. These cases were chosen due to the fact that common errors could possibly occur under such conditions.

The code for the adder test bench can be found in Listing 3. The various test cases were chosen to test the basic function of the adder and to test its limitations. The first test case was simple addition check to ensure the adder could actually add small numbers with no errors. The next case was chosen because it was expected to cause a ripple-carry between bytes. The next two cases tested hexadecimal inputs, with the latter testing the largest value that could be properly stored in a register of size WORD. This number was found using the equation $2^n - 1$, with n being 32. The next case added intentionally caused overflow to test what the adder code would output. This was done by adding h'FFFFFFFF to itself. It was the adder was expected to output zero due to this error.

Listing 3: Verilog code for testing an adder.

```
`timescale 1ns / 1ps

`include "definitions.vh"

module adder_test;

        // Inputs
        reg [`WORD-1:0] Ain;
        reg [`WORD-1:0] Bin;

        // Outputs
        wire [`WORD-1:0] add_out;

        // Instantiate the Unit Under Test (UUT)
        adder uut (
```

```
        .Ain(Ain),
        .Bin(Bin),
        .add_out(add_out)
    );

    initial begin
        // Initialize Inputs
        Ain = 0;
        Bin = 0;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here

        Ain = 1;
        Bin = 1;
        //Expected result: d'2

        #100;

        Ain = 4;
        Bin = 8;
        //Expected result: d'12
        //will produce ripple carry

        #100;

        Ain = 32'hFF00;
        Bin = 32'h00FF;
        //Expected result: h'0000 FFFF

        #100;


    Ain = 32'hFFFF0000;
    Bin = 32'h0000FFFF;
    //Expected result: h'FFFFFFFF
    // largest unsigned value

    #100;

    Ain = 32'hFFFFFFFF;
    Bin = 32'd1;
    // Expected Result:0
    //(Will cause overflow)
```

```
                #100;

                Ain = 32'd1;
                Bin = 32'd0;
                //Expected Result: 1
                //(Testing adding zero)


        end

endmodule
```

Due to the parameter used in the multiplexer code, definitions were added to multiplexer test bench to boost re-usability. Two different sizes of the multiplexer were tested – 5-bit and 16-bit. Test cases were constructed so that various number combinations were used and control was varied. Initially, the 5-bit multiplexer was tested. Referring to Listing 4, MUXSIZE was set to 5 and SIZE was set to 16 (i.e.- remained unchanged). This was an oversight which led to an expected error. In the second text case, the expected output was 56. However, because the multiplexer was only five bits, only the last five bits of 56 (b'111000) were used, causing the actual output value to be 24 (b'11000). Additionally, due to the fact that Vivado did not know what to do with them, the remaining eleven bits were filled with Z. The last test ran on the mux was with the parameter defined as 16 (that is, with MUXSIZE and SIZE set to 16). This test produced expected results.

Listing 4: Verilog code for testing a mux.

```
`timescale 1ns / 1ps
`define MUXSIZE 16
`define SIZE 5

module mux_test;

        reg [15:0] Ain;
        reg [15:0] Bin;
        reg control;
        wire [15:0] mux_out;

        mux#(`SIZE) uut (
                .Ain(Ain),
                .Bin(Bin),
                .control(control),
                .mux_out(mux_out)
        );
```

```
        initial begin
                Ain <= 0;
                Bin <= 0;
                control <= 0;
        #10;

        Ain <= 'SIZE'd10;
        Bin <= 'SIZE'd56;
        control <= 'SIZE'd0;
        //Expected: 10

        #10;
        control <= 1; //Expected 56
        #10;

         Ain <= 'SIZE'd3;
         Bin <= 'SIZE'd2;
         control <= 'SIZE'd1;
         //Expected: 2

         #10;


         Ain <= 'SIZE'd10;
         Bin <= 'SIZE'd0;
         control <= 'SIZE'd0;
                //Expected: 10

        end

endmodule
```

6 Simulation

The timing diagram for the adder test can be found in Figure 1 on page **??**. The adder test bench simulation provided exactly the expected results.

The simulation waveforms for the intial 5-bit multiplexer test can be found in Figure 2. Note the blue waveforms and the fact that they are marked with Z, not a distinct value or what was expected. However, it can be seen that the last five bits of the output match what was expected.

Lastly, the timing diagram for the 16-bit multiplexer test can be found in Figure 3. The output of the simulation was as expected, and no other errors were found.

Figure 1: Timing diagram for the adder test.

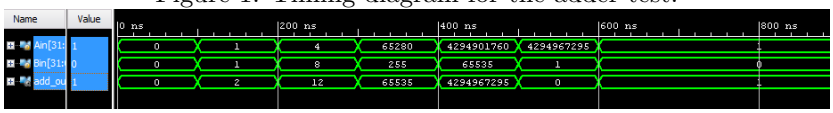
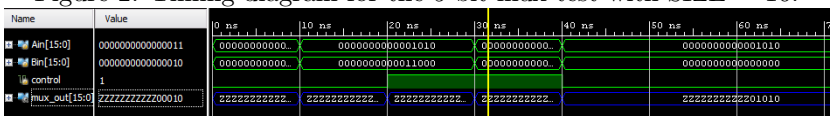


Figure 2: Timing diagram for the 5-bit mux test with SIZE = 16.



7 Conclusions

In order to create a properly functioning program counter, an incrementer and input selector are required. To ensure this, a simple adder and multiplexer were built and tested. While a few errors were encountered, these components functioned as expected. The simulation waveforms created from the various tests provide insight into how these components function and how important details like size definitions and parameters can be. While the adder performed exactly as expected, the multiplexer was found to be a bit more complicated. For a more thorough investigation, it may behoove the user to test the multiplexer with additional size parameters.

Despite the few errors encountered, the components operated correctly. Thus, by implementing the adder and multiplexer created in Lab 2, a fully functioning program counter could be constructed.

Figure 3: Timing diagram for the 16-bit mux test.