ELC 3338 Project Book

Steve Potter

August 26, 2020

Contents

Lab 1

Program Counter Register

During the course of this semester, we will build a 64-bit computer. To do this, we will make a synthesizable machine in Verilog, a common hardware description language (HDL).

A computer runs a program by executing individual instructions in sequential order. The instructions are stored in memory and are accessed by their memory address. During each clock cycle, an instruction is fetched from memory and executed on the processor. The memory address of the next instruction to fetch is stored in a register called the Program Counter (PC). During Lab 1, we will build and test the Program Counter register. In Lab 2, we build an incrementer (to count to the next instruction) and a mux (to select between the incremented count or a new starting value).

1.1 Program Counter Register

In order to make the Program Counter, we are going to make a Verilog module that explains how to build a register (a D flip-flop). Let me unpack the previous sentence:

1. Verilog is a Hardware Description Language (HDL).
2. We write Verilog code to tell Vivado how we want our register module to behave.
3. Vivado reads our Verilog code and synthesizes a realizable digital hardware design that meets the behavior that we specified. Thank you Vivado!
4. Vivado also simulates the behavior of the hardware, allowing you to test your design without building/programming hardware.

Consider the Verilog code in Listing 1.1. It is made up of three sections:

1. Header - this code includes a file name definitions.vh which contains information necessary for the program to run.

2. Port list (also known as interface) - specifies the signals coming into or going out of the module. In this case, there are three inputs and one output.

3. Body (also known as implementation) - describes the functionality of the module.

Listing 1.1: Verilog code to make a register.

```
`include "definitions.vh"

module register(
    input clk,
    input reset,
    input  [`WORD-1:0] D,
    output reg [`WORD-1:0] Q
    );
    always @(posedge(clk),posedge(reset)) begin
        if (reset==1'b1)
            Q<=`WORD'b0;
        else
            Q <= D;
    end
endmodule
```

The first part is the header. We will use this same header each time. It tells the Verilog compiler to get all the data from a file called definitions.vh. The extension vh is a Verilog header. We use this to specify common pieces of data we will use across our design, so that all the components we build will be consistent. By putting them in one file, we make it easier to maintain, and prevent mistakes that can happen easily by having multiple copies of these basic pieces of data. For our first lab, one item that we will be using from definitions.vh is WORD (set to 64), which is the size (in bits) of the memory addresses in our computer. Note that if we build things based around WORD, rather than the number 64, we can just change the value of WORD in the file and get a computer with a different size with a couple key strokes.

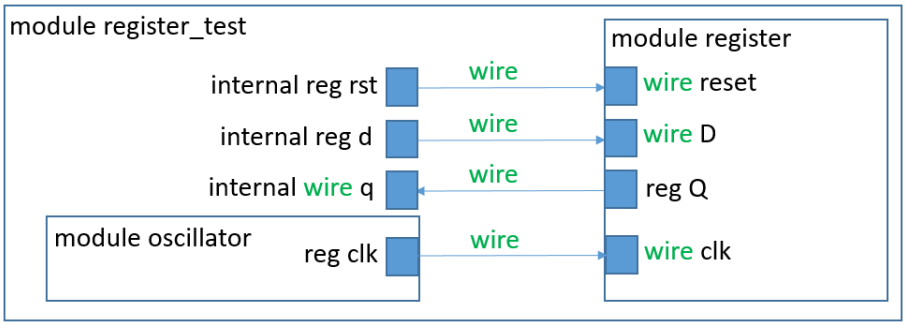
The second part is the port list or interface. In this area we specify what signals are coming in (input), going out (output), or could go either direction (inout). Ports can be defined as either "wire" or "reg". This can be confusing to some students. Think of it this way:

1. Wire

   (a) A wire is just a conductor that connects one component or module to another.

   (b) The value on a wire can only be changed by using combinational logic (as opposed to sequential logic).

Figure 1.1: Module Diagram.



- Internal regs are set in the 'initial' block or an 'always' block

(c) It has no memory, meaning that the value on the wire is driven by the results of combinational logic at that particular moment.

(d) Module inputs are always wires.

(e) Module outputs can be wires or regs.

2. Reg

(a) A reg more closely resembles a variable in software programming languages.

(b) A value of a reg can only be set by using sequential logic.

(c) A reg has memory, meaning that the value of the reg will remain the same until a sequential logic element updates it.

(d) You can directly set a reg to a value using a procedural assignment.

(e) Regs can be used internally in a module (neither input nor output), or they can be used as module outputs. They cannot be used as module inputs.

If you don't specify anything for the port type, you will get a wire - it is the default. In our case we have four signals: three inputs (always wires), and one output that is a register. The first two inputs are single-bit wires. One is the clock, which specifies the timing, and the other is reset, which clears the contents of the register (makes them zero). The final input is the value we want to store in memory, and I have called it D, following the convention of digital logic. D has multiple bits that are numbered from WORD-1 down to 0. Thus the leftmost bit is 63 in this case, and the rightmost bit is 0[1]. The output Q

[1]If you want to be technical this is called little endian, since the little end (the least significant or unit bit) is going into the first memory location (bit 0). If you reversed the order by putting the 0 first and the WORD-1 last it would be big endian, since the big end (most significant bit) would go in the lowest addressed bit.

(also the digital logic conventional name) is a register (it will hold its value) and should also be of size WORD and follow the same bit order as the input D.

To help clarify this, please examine Figure 1.1, which shows the interconnection of the modules in this lab.

The final section is the body or implementation. It is composed of a single thread of code, that will keep running (hence always). It will run one time every time there is a positive edge (0 to 1 transition) for either the clock or reset. Reset has higher priority, so if reset is asserted the register is cleared (Q is set to zero), otherwise the value of D is stored it Q. That is it. A nice, simple module. Please note that the provided register module is fully operational. You do not need to modify it. This file is provided for you in ARM-Lab/code/1_fetch/register.v.

1.2 Testbench

We now want to test our register module using System Verilog. System Verilog is very similar to Verilog, but it adds the ability to verify that we get the results that we are expecting. To test the register module, we need to tell the simulator to build a copy (instantiate) of the register module, and then we will need to supply the inputs and evaluate the outputs to verify that the module works correctly. Evaluate the testbench, ARM-Lab/code/1_fetch/register_test.sv.

When evaluating this testbench module (register_test), notice that there are no ports. A testbench is providing all the signals to simulate the inputs to the unit under test (UUT) and thus does not need ports. This is how Verilog finds a top level simulation module - there are no ports. The clock signal will be driven by a module named oscillator, which will give us a square wave with period CYCLE, which is another constant defined in our definitions.vh file. The code thus makes an oscillator and a register, then runs the 'initial' section (it runs once at the start then never again).

This initial section of the testbench follows a relatively simple pattern:

1. The inputs to the register module are set to particular values.
2. The system delays for some amount of time. For instance, a one cycle delay is inserted with #CYCLE.
3. The value of er (expected result) is set to the output value that you expect to get from the register module.
4. The value and size of er are compared with the value and size of ar (actual result). The actual result is the output of your register module. I provide (in "verification_functions.sv", included at the top of the testbench) the verify function and a few other functions that allow us to easily verify the behavior of our system. Each time the 'verify' function is called, it reports and keeps track of whether the test passed (ar == er) or failed (ar != er).
5. At the end of the testbench, the final_result function is called to report the results of the test. This function will show the number of passing and failing test cases.

1.3 Your Assignment

You are to:

1. Evaluate the provided testbench. It is not an exhaustive testbench, but it tests a number of cases that commonly occur in our system. Note that you should not change the input values or timing of the testbench, nor should you add additional test cases to the testbench.

2. Create an Expected Results Table for your testbench. The idea behind the Expected Results Table is that you should identify how you think the system should operate. If you don't know how it should work, you will not know whether your simulation results are correct. The Expected Results Table should have a row for each signal in your simulation results (and the row order should match between your Expected Results Table and Simulation Results). The table should also have a column for each test case in the testbench. These test cases are the points in time that correspond to the 'verify' function calls in the testbench.

   I have provided an Expected Results Table for Lab 1 in ARM-Lab/tables. It includes everything you need, except that the row for er_q is not filled in. Please fill in this row with the expected value of q for each test case. Note that you don't need to show the clk signal in the Expected Results Table.

3. The provided testbench does not set er_q to the correct value, therefore causing your test to fail. Your job now is to take the values from your expected results table and enter these values in the testbench as er_q (expected result of q).

4. Run a behavioral simulation. Evaluate the timing diagram and verify that it matches the Expected Results Table. Also evaluate the printouts in the Tcl Console window in Vivado. These printouts will indicate the number of passes and fails that occurred in the test. If you chose the correct er values and all tests pass, then your module is verified to work properly.

5. Produce a landscape mode PDF called Lab1_lastname.pdf that includes (in this order):

   (a) Your name and the lab number.

   (b) A snip (using the Snipping Tool) of your Expected Results Table.

   (c) A snip of the Simulation Results (the black window that shows each signal). Make sure to show all values in decimal form and don't cut off the signal names on the left.

   (d) The test results copied and pasted from the Tcl Console. The results should show the entire log from BEGIN TEST RESULTS to END TEST RESULTS.

   (e) I have included a sample, ARM-Lab/reports/Lab1_Potter.pdf to give you an idea of what I'm looking for.

6. Upload Lab1_lastname.pdf file to Canvas.

7. Zip up your ARM-Lab directory and submit it on Canvas as well. Please make sure that you give me the ARM-Lab directory rather than the ARM-Project directory. I do not want the project files in the ARM-Project directory. Before you submit your zip file, make sure that it contains directories like code, manual, tables, etc. I will extract your zip file and run your code against my correct testbench to verify that your code and testbench work correctly. Since I give you working register.v code in this lab, this is pretty easy. In future labs, you must create your own module code.

Lab 2

Program Counter Incrementer and Mux

As mentioned in the last lab, the program counter is a register that is one word in length. It holds the address in memory of the next instruction to be fetched and executed. There are several ways that the program counter is updated:

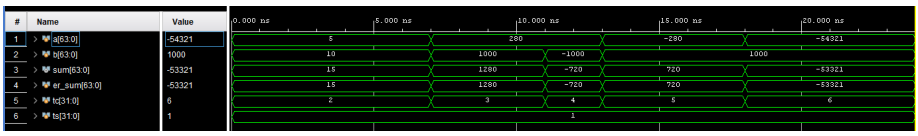
1. If the program does not branch (via an if statement, while loop, etc), then the program counter should advance to the next address (by adding 4 to the current PC) each clock cycle.
2. If the conditions of a conditional branch are met, then the program counter should be updated with the branch destination address.
3. If an unconditional branch or jump occurs, then the program counter should be updated with the branch destination address.
4. If an interrupt or error occurs, then the program counter should be updated with the interrupt or error handler address.

The instructions will be fetched in sequential order the majority of the time.

2.1 Adder

We will build a program counter incrementer by making a simple adder. Later in our computer we will need another adder, so we will re-use this code. Therefore, the adder code should be stored in ARM-Lab/code/0_common because it will be used in multiple stages. When used as the program counter, we will pass it a 4 because each instruction is 32-bits long (even though it is a 64-bit computer) and we want to increment to the next instruction in memory. Most machines are byte addressable, because one ASCII character (a char in c/c++) is a byte. For a machine with 32-bit instructions like we are using, that would mean that each instruction would be 4 bytes later in memory ($32/8 = 4$ bytes). Therefore,

Figure 2.1: Adder Test Simulation Results



we will be adding 4 to the program counter each time we want to increment the program counter.

An adder is very simple in Verilog. There are two inputs (the two numbers to be added) and one output (the result). All the ports are size 'WORD because they hold 64-bit integers.

In this lab you will make your own adder module. Your adder module should be called 'adder' and should have inputs of `a_in` and `b_in`. The output should be `add_out`. HINT: this should be very easy. Verilog is a Hardware Description Language, so use Verilog to describe what you want to do. Don't make it complicated. The adder code should be stored in ARM-Lab/code/0_common/adder.v. You will need to create this file.

2.2 Adder Test Bench

I have provided an incomplete test bench for the adder. While I put most of the infrastructure in place, you will need to fill in the following details. I labeled these spots with comments that start with TODO so that you can see where you need to do your work.

1. Create an instance of the adder module

2. Fill in the details of each test case so that your simulation results and test log match the results shown in Figure 2.1 and Figure 2.2. Pay careful attention to detail to make sure your results are identical to these figures. This includes making sure that the timing matches.

2.3 Input Selection via Mux

We will also need to be able to choose between normal advancing (sequential stepping) and branching (loops, if statements, etc.). We will use a multiplexor (mux) to do this. A mux is a simple device that connects one of the inputs to the output based on how the control bit is set. If the control bit is 0 then input a is connected to the output, and if the selector is 1 then input b is connected to the output. The heading and port list is provided for you in Listing 2.1. You just need to fill in the body to make it operate like a mux.

Figure 2.2: Adder Test Log

```
******* BEGIN TEST RESULTS *******

Test Case 1
Inputs: a = 5 | b = 10
+++ Step 1: Pass: |sum| time = 7 ns | er = 15 | ar = 15 | er_bits = 64 | ar_bits = 64 +++

Test Case 2
Inputs: a = 280 | b = 1000
+++ Step 1: Pass: |sum| time = 11 ns | er = 1280 | ar = 1280 | er_bits = 64 | ar_bits = 64 +++

Test Case 3
Inputs: a = 280 | b = -1000
+++ Step 1: Pass: |sum| time = 13 ns | er = -720 | ar = -720 | er_bits = 64 | ar_bits = 64 +++

Test Case 4
Inputs: a = -280 | b = 1000
+++ Step 1: Pass: |sum| time = 18 ns | er = 720 | ar = 720 | er_bits = 64 | ar_bits = 64 +++

Test Case 5
Inputs: a = -54321 | b = 1000
+++ Step 1: Pass: |sum| time = 24 ns | er = -53321 | ar = -53321 | er_bits = 64 | ar_bits = 64 +++

Pass Count = 5
Fail Count = 0

******* END TEST RESULTS *******
```

One interesting addition in this block of code is the addition of a size parameter. Parameters allow you to pass a value into a module, making the module more flexible and reusable. In the mux code, the parameter is specificed by #(parameter SIZE=8). Parameters are constants, so they cannot be explicitly changed within the module. Rather, they are specified when an instance of this module is created. In the provided mux_test.sv, the mux module is instantiated by: mux #(‘WORD) UUT_64. Note that ‘WORD is a macro (from definitions.vh) that defines another name for the value 64. So the test bench passes in a parameter value of 64, and the mux module uses 64 anywhere the term SIZE is used. If no parameter is specified when the instance is created, then the parameter will be assigned the default value. $SIZE = 8$ defines the default value for the SIZE parameter in this module.

Note parameters are constants and cannot be changed later in the module. In this lab, we are using parameters to set the number of wires that compose the inputs and output. In our lab project, we will need some muxes to switch entire words (64 bits) like this one, but later we will also need to switch register addresses (5 bits). Rather than write two muxes, we will make one and then use the parameter to change the size when they are declared. The mux code should be stored in ARM-Lab/code/0_common/mux.v.

The provided testbench instantiates both a 64 bit mux and a 5-bit mux so that two different mux sizes can be tested, ensuring that the mux works now and will work when we use it later in the semester.

Listing 2.1: Verilog code to make a mux.

```
`include "definitions.vh"

module mux#(
    parameter SIZE=8)(
    input [SIZE-1:0] a_in,
    input [SIZE-1:0] b_in,
    input control,
    output [SIZE-1:0] mux_out
    );

  // TODO: Add body of mux here

endmodule
```

2.4 Your Assignment

You are to:

1. Create an adder module.
2. Update adder_test.sv as described above.
3. Fill in the body of the mux module in mux.v.
4. Use the provided mux_test.sv to verify that the mux works properly. Note that you cannot/should not make any changes to the test bench. The correct results are already in the test bench.
5. Produce a landscape mode PDF called Lab2_lastname.pdf that includes (in this order):
   (a) Your name and the lab number.
   (b) A snip of the Simulation Results for the adder_test. Make sure to show all values in decimal form and don't cut off the signal names on the left.
   (c) The adder_test results copied and pasted from the Tcl Console. The results should show the entire log from BEGIN TEST RESULTS to END TEST RESULTS.
   (d) A snip of the Simulation Results for the mux_test. Make sure to show all values in decimal form and don't cut off the signal names on the left.
   (e) The mux_test results copied and pasted from the Tcl Console. The results should show the entire log from BEGIN TEST RESULTS to END TEST RESULTS.

6. Upload Lab2_lastname.pdf file to Canvas.

7. Zip up your ARM-Lab directory and submit it on Canvas as well. Please make sure that you give me the ARM-Lab directory rather than the ARM-Project directory. I do not want the project files in the ARM-Project directory. Before you submit your zip file, make sure that it contains directories like code, manual, tables, etc. I will extract your zip file and run your code against my correct testbench to verify that your code and testbench work correctly.