ELC 3338 Project Book

Steve Potter

January 22, 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 wire clk,
    input wire reset,
    input  wire [`WORD-1:0] D,
    output reg [`WORD-1:0] Q=`WORD'b0
    );

    always @(posedge(clk),posedge(reset))begin
        if (reset==1'b1)
            Q<=`WORD'b0;
        else
            Q <= D;
    end

endmodule
```

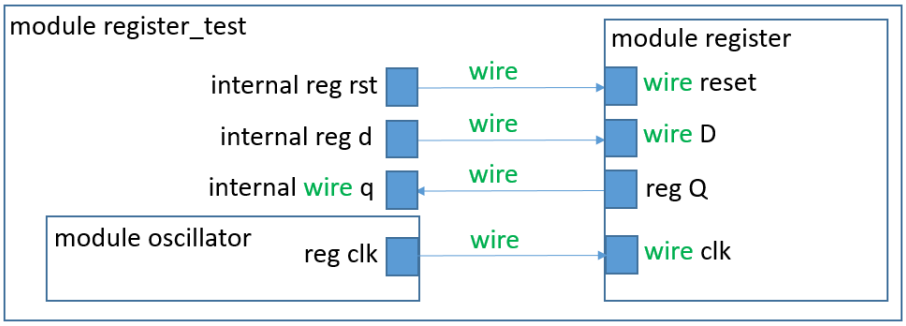
The first part is the header. We will use this same header each time. It tells the Verilog compiler to get all the data from a file called definitions.vh. The extension vh is a Verilog header. We use this to specify common pieces of data we will use across our design, so that all the components we build 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.

Figure 1.1: Module Diagram.



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

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

(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 (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.

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. Consider the testbench in Listing 1.2.

Listing 1.2: System Verilog code to test a register.

```
// include functions to verify functionality
`include "verification_functions.sv"

// create module register_test with no arguments
// no arguments implies that it is the top-level module
module register_test;

// import the verification package that was included above
import verification::*;

// create a 1-bit wire for use as a clock signal, oscillating between 0 and
wire clk;

// create a 1-bit reg to be used as the reset pin on the D Flip Flop
reg rst;

// create a 64 bit reg that will be used to set the D input of the Flip Flop
reg[`WORD - 1:0] d;
```

[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.

```
// create a 64 bit wire that will carry the Q output from the Flip Flop
wire [`WORD - 1:0] q;

// create a 64 bit reg that you will set to the correct result (cr)
reg [`WORD - 1:0] cr;

// create an instance of the oscillator module (provided) that will toggle the
// clock signal with a cycle time of 10ns
oscillator clk_gen(clk);

// create an instance of the register module called myreg
// use the clk, rst, in, and out signals as inputs/outputs
// to this instance of the register module
// note that the name that comes after the dot is the port name
// and the the name in parentheses is the signal name that connects
// to that port.  They can be the same but do not have to be the same
register myreg(
    .clk(clk),
    .reset(rst),
    .D(d),
    .Q(q)
    );

// the initial section is executed one time when the system starts up
// the initial section is a procedural block, meaning that regs must
// be used for signals that you will be manipulating in the initial section
initial
begin
    // call a verification function at the beginning to start the log
    begin_test();

    // set the reset pin to 0
    rst = 0;

    // set in to 0 (sets all 64 bits to a value of 0 in decimal)
    d=`WORD'd0;
    // wait for one cycle (10ns)
    #`CYCLE;

    // set the correct result (cr) to the value that you believe should
    // be produced on the Q output at this particular point in time
    cr=`WORD'd0;

    // call the verify function from the verification package
    // this function can be viewed in verification_functions.sv
```

```
    // the verify function compares the value of cr to the value
    // of ar (actual result, see function definition).  It also
    // compares the size of ar and cr
    verify(cr, $bits(cr), q, $bits(q));

    // repeat the previous steps using different input values and
    // different delays
    d='WORD'd1;
    #`CYCLE;
    cr='WORD'd9;
    verify(cr, $bits(cr), q, $bits(q));

    d='WORD'd2;
    #`CYCLE;
    cr='WORD'd8;
    verify(cr, $bits(cr), q, $bits(q));

    d='WORD'd3;
    #`CYCLE;
    cr='WORD'd7;
    verify(cr, $bits(cr), q, $bits(q));

    d='WORD'd4;
    #(`CYCLE/5);
    cr='WORD'd3;
    verify(cr, $bits(cr), q, $bits(q));

    d='WORD'd5;
    #(`CYCLE*4/5);
    cr='WORD'd5;
    verify(cr, $bits(cr), q, $bits(q));

    rst=1;
    #(`CYCLE/2);
    cr='WORD'd6;
    verify(cr, $bits(cr), q, $bits(q));

    #`CYCLE;
    rst=0;
    #(`CYCLE/2);
    cr='WORD'd5;
    verify(cr, $bits(cr), q, $bits(q));

    rst=1;
    #(`CYCLE/2);
    cr='WORD'd4;
```

```
    verify(cr, $bits(cr), q, $bits(q));

    #2;
    d='WORD'd345;
    cr='WORD'd3;
    verify(cr, $bits(cr), q, $bits(q));

    #3;
    rst=0;
    cr='WORD'd2;
    verify(cr, $bits(cr), q, $bits(q));

    #'CYCLE;
    cr='WORD'd1;
    verify(cr, $bits(cr), q, $bits(q));

    // Add an extra cycle delay so that we can see the results on the simulation
    #'CYCLE;

    // call the final_result function to tally the results of the test
    // this function is part of the verification package
    final_result();

    $finish;
end
endmodule
```

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:

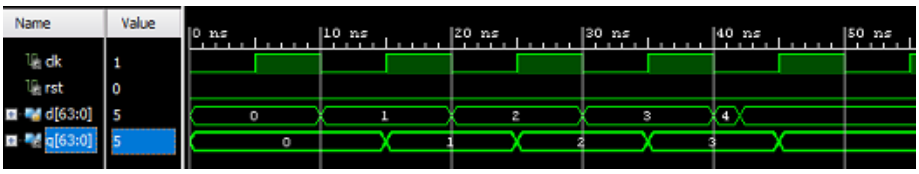
1. The inputs to the register module are set to particular values.

2. The system delays for some amount of time. For instance, a one cycle delay is inserted with #CYCLE.

3. The value of cr (correct result) is set to the output value that you expect to get from the register module.

4. The value and size of cr are compared with the value and size of ar (actual result). The actual result is the output of your register module. I provide

Figure 1.2: Timing diagram.



(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 verify is called, it keeps track of whether the test passed (ar == cr) or failed (ar != cr).

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 testbench in Listing 1.2. 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. An example Expected Results Table is at ARM-Lab/testfiles/Lab1_Register_ExpectedResultsTable.xlsx. The idea behind the Expected Results Table is that you 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 point in the testbench. These test points are the points in time that correspond to the 'verify' function calls in the testbench. To complete the table, fill in each cell with the expected value. Note that you don't need to show the clk signal in the Expected Results Table. See the Lab1 Expected Results Table Excel file in the testfiles section of my git repository.

3. The provided testbench does not set the cr 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 the cr (correct result).

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 cr values and all tests pass, then your module is verified to work properly.

5. Rather than writing a lab report, please produce a landscape mode single page 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. Make sure to show all values in decimal form and don't cut off the signal names on the left.

   (d) A snip of the test results from the Tcl Console. This snip should show the entire log from BEGIN TEST RESULTS to END TEST RESULTS.

   (e) I have included a sample in the testfiles directory of my git repository.

6. Upload Lab1_lastname.pdf file to Canvas.

7. Zip up your ARM-Lab directory and submit it on Canvas as well. I will 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 Incrementer

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. 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, 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 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 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. One interesting addition in this block of code is the addition of a size parameter. Parameters are passed before the normal ports and are used to configure the code to meet a requirement at the time of construction. Note parameters are constants and cannot be changed later in the module. The = 8 defines the default value if nothing is specified. In this case 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), 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. Please look at the starter code in this lab document for direction on how to add a parameter to the module.

Listing 2.1: Verilog code to make a mux.

```
`include "definitions.vh"

module mux#(
    parameter SIZE=8)(
    // add ports here
    );

    // Add module code here

endmodule
```

Look at the testbench provided for the mux. Note that if the parameter is not set by the testbench, the mux module will set the inputs and outputs to be the default of 8. We are going to change this to test it as a 64 bit mux and a

5-bit mux. Notice how the size of the mux is set, since you will need to do this in future labs.

2.3 Your Assignment

You are to:

1. Create an adder module.
2. Use the provided adder_test.sv to verify that the adder works properly. Note that you cannot/should not make any changes to the test bench. The correct results are already in the test bench.
3. Create a mux module.
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. Rather than writing a lab report, please produce a landscape mode single page 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. Make sure to show all values in decimal form and don't cut off the signal names on the left.
   (c) A snip of the test results from the Tcl Console for the adder. This snip should show the entire log from BEGIN TEST RESULTS to END TEST RESULTS.
   (d) A snip of the Simulation Results for the mux. Make sure to show all values in decimal form and don't cut off the signal names on the left.
   (e) A snip of the test results from the Tcl Console for the mux. This snip 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. I will run your code against my correct testbench to verify that your code and testbench work correctly.