ELC 3338 Project Book

Steve Potter

February 10, 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
```

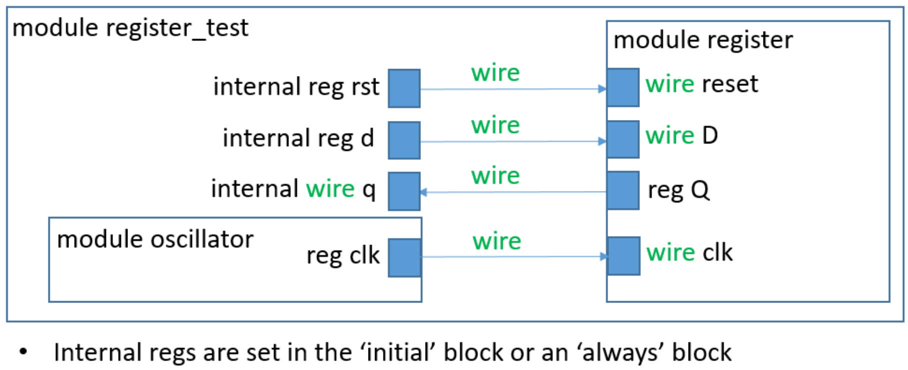
The first part is the header. We will use this same header each time. It tells the Verilog compiler to get all the data from a file called definitions.vh. The extension vh is a Verilog header. We use this to specify common pieces of data we will use across our design, so that all the components we build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



(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 Flo
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:

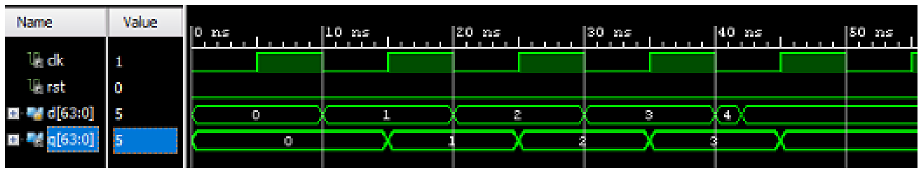
1. The inputs to the register module are set to particular values.

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)(
    input [SIZE-1:0] a_in,
    input [SIZE-1:0] b_in,
    input control,
    output [SIZE-1:0] mux_out
    );

    assign mux_out = control?b_in:a_in;

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.

Lab 3

Fetch Stage

We are ready to build our fetch stage. To do this, we will make one more module, our instruction memory. Then we will make a module to assemble all of our modules together into a working fetch stage.

3.1 Instruction Memory Module

Instructions are stored in memory and are accessed by using the address where they are stored. You can think of memory like a giant hotel for our data. Each piece of data is stored in a room (memory location), which we can find by its room number (memory address). To get a piece of data stored in memory (like an instruction) we need to take its address, go to that location, and grab the data. In Verilog, a bunch of memory locations that are accessed by an address is called an array. Arrays in Verilog are declared like they are in C; the data type is specified, then the name, then the array size.

To store the instructions, we will need an array of 32-bit numbers (definitions.vh defines INSTR_LEN as 32, please use this macro), which means the data type must be `reg['INSTR_LEN-1:0]`. After the name is specified (imem in this case), we are going to use a parameter called SIZE to specify how many elements the array has: `[SIZE-1:0]`. Therefore, your array will be defined as `reg['INSTR_LEN-1:0] imem [SIZE-1:0]`.

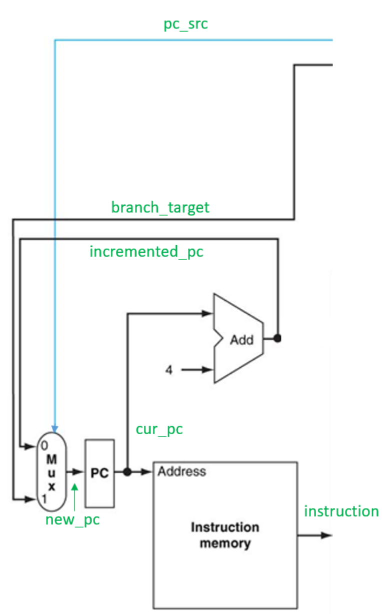
Now we need to populate this array with instructions. Rather than populating the array element by element, we will read the instruction values in from a file called instrData.data that I have provided in the testfiles area of the ARM-Lab repository. Note that we are just initializing the array with the values from the file. The values should only be read from the file once at the beginning of the simulation. Then we will access the imem array for an instruction value. To read the file and put the contents into our imem array, we will use $readmemh to read in hexadecimal values from instrData.data. $readmemb could be used if we chose to format our data file in binary rather than hexadecimal. It is very important that we use a macro in definitions.vh for the name and path of the instrData.data file, as this might change and it will be necessary for grading. Make sure to update the definition of IMEMFILE to point to your repository. The line of code to read the data in from the file is `$readmemh('IMEMFILE, imem);`

Once the array is populated, we can access imem and provide the instruction that corresponds to the requested address. Instructions should only be updated on the positive edge of the clock. I have provided a testbench for this module. Create a file called instr_mem.v in code/1_fetch directory and write your module code here. Test it against instr_mem_test.sv and verify that the module works as expected. The test will compare your instruction output to the instruction values that I provided in instrData.data.

3.2 Fetch Stage

Now we need to connect our modules together to make a fetch stage. The components of our instruction fetch (sometimes called ifetch or just fetch) stage are shown in Figure 3.1.

Figure 3.1: Instruction Fetch Stage.



Any wire (or reg) that comes into or goes out of the figure are input or output ports of the iFetch module. In Figure 3.1, the blue wire is a control signal and comes ultimately from the control unit, which you will build in the decode stage. Wires (or regs) that are completely contained in the figure are local to the iFetch module and are thus defined internally in the module (not an input or output). The one exception to this is the current program counter (cur_pc). While there is no reason (at this point) that it must be output from the iFetch module, you must still make it an output so that it shows up on your simulation results, helping you to keep track of the program counter for the instruction that is currently executing. Also, it will be required when we start pipelining our datapath in Lab 12. And it is required to verify functionality with the testbench.

While the input and output signals are easily identified by the diagram, you must also determine the size of each signal and whether it is a wire or reg. When you look at the figure I cut from a figure in the book, note that I labeled every wire on the diagram in green. For the sake of consistency and debugging, it is required that you use these names.

IMPORTANT NOTE: Throughout your entire project, your signal names should follow the convention of the Freescale Semiconductor Verilog guide, which states that signal names should be all lower case, with words separated by an underscore.

Once you have figured out all your connecting signals (wires and regs), you should identify the components you are going to use. We have already created the modules, so now we just need to tell Verilog to instantiate them in the iFetch module and connect them together. Create the iFetch module in a new file called iFetch.v in the code/1_fetch directory.

I have provided a testbench called iFetch_test.sv. Use this testbench to determine if your fetch stage is working properly. As we progress through this lab project, you will learn how critical timing is. Please look at the cur_pc value and the instruction value and verify that the instruction that was fetched is the correct instruction, according to instrData.data and the current program counter. Note that no instruction should be fetched in the first 5ns, as this is a half clock cycle and does not have a rising edge.

I have included a file called delay.v in code/0_common. It includes a module that inputs a clock signal and outputs a clock signal that is delayed by some number of ns. This will be useful when resolving timing issues. Throughout this entire lab project, the delay module should only be instantiated in the top-level module (testbench). If you need the output of the delay function in a lower-level module, add a port to the lower-level module and pass it through that port.

3.3 Your Assignment

You are to:

1. Create the instruction memory module.
2. Verify the module by running against the provided testbench.
3. Create the fetch stage.
4. Verify the module by running against the provided testbench.
5. Rather than writing a lab report, please produce a landscape mode PDF file called Lab3_lastname.pdf that includes (in this order):
   (a) Your name and the lab number.
   (b) A snip of the Simulation Results for the instr_mem module. Please show instructions in hex and everything else in signed decimal.
   (c) A snip of the test results from the Tcl Console for the instr_mem module. This snip should show the entire log from BEGIN TEST RESULTS to END TEST RESULTS.
   (d) A snip of the Simulation Results for the fetch module. Please show instructions in hex and everything else in signed decimal.

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

6. Upload Lab3_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.

Lab 4

Beginning to Decode

4.1 Instruction Decode

The next stage in the datapath is the iDecode stage. The iDecode stage evaluates the binary instructions (an output of the iFetch stage) and determines what needs to be done. There are many aspects to the iDecode stage, and some get fairly complex. But today we will begin the process of decoding an instruction by decomposing the instructions into the key parts of R-Type and D-Type instructions:

1. opcode
2. address (used only in D-Type instructions)
3. rm_num (used only in R-Type instructions)
4. rn_num
5. rd_num (though the book uses Rt for D-type instructions, we will use Rd for the last operand of D-type instructions)

To do this, you will create a new module called instruction_parse. This module will simply read an input and assign appropriate output values. These outputs should be assigned using continuous assignments. The input is a 32-bit instruction. Outputs are listed for you above. Although R-type and D-type instructions have different operands, you can treat them the same for now. For instance, you can still assign an Address field on an R-type instruction, and you can still assign an Rm field on a D-type instruction. When we create the Control Module in a future lab, the control signals will drive what fields of the instruction are used and what fields are ignored. Notice how, because of the commonality of instruction format, Opcode, Rn, and Rd are all universal across these instruction types. Please remember to use the style specified in the previous lab, where all items are lower case with underscores separating them. For instance, for Rd, you should use the signal name rd_num. Appending num

Figure 4.1: Instruction Parse Test Output

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

LDUR X9, [X22, #240]
+++ Step 1: Pass: |rn_num| time = 2 ns | cr = 22 | ar = 22 | cr_bits = 5 | ar_bits = 5 +++
+++ Step 2: Pass: |rd_num| time = 2 ns | cr = 9 | ar = 9 | cr_bits = 5 | ar_bits = 5 +++
+++ Step 3: Pass: |address| time = 2 ns | cr = 240 | ar = 240 | cr_bits = 9 | ar_bits = 9 +++
+++ Step 4: Pass: |opcode| time = 2 ns | cr = 7c2 | ar = 7c2 | cr_bits = 11 | ar_bits = 11 +++

ADD X10, X21, X9
+++ Step 5: Pass: |rm_num| time = 12 ns | cr = 9 | ar = 9 | cr_bits = 5 | ar_bits = 5 +++
+++ Step 6: Pass: |rn_num| time = 12 ns | cr = 21 | ar = 21 | cr_bits = 5 | ar_bits = 5 +++
+++ Step 7: Pass: |rd_num| time = 12 ns | cr = 10 | ar = 10 | cr_bits = 5 | ar_bits = 5 +++
+++ Step 8: Pass: |opcode| time = 12 ns | cr = 458 | ar = 458 | cr_bits = 11 | ar_bits = 11 +++

STUR X10, [X23, #64]
+++ Step 9: Pass: |rn_num| time = 22 ns | cr = 23 | ar = 23 | cr_bits = 5 | ar_bits = 5 +++
+++ Step 10: Pass: |rd_num| time = 22 ns | cr = 10 | ar = 10 | cr_bits = 5 | ar_bits = 5 +++
+++ Step 11: Pass: |address| time = 22 ns | cr = 64 | ar = 64 | cr_bits = 9 | ar_bits = 9 +++
+++ Step 12: Pass: |opcode| time = 22 ns | cr = 7c0 | ar = 7c0 | cr_bits = 11 | ar_bits = 11 +++

Pass Count = 12
Fail Count = 0

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

on the end of the name indicates that this is the register number, not the value from the register.

To test this module, you will need to finish instr_parse_test.sv. I have provided some starter code for the testbench as well as detailed comments that describe what you need to do. The testbench will feed the module with instructions. The instructions are specified in the testbench and are very similar (yet slightly different) to the instructions that were encoded in the lecture on Machine Code. You will need to update the testbench by:

1. Creating signals to be used in the testbench
2. Setting the instruction to the correct value per the instruction listed in the comments
3. Add code to verify that each output of your instruction_parse module is correct

The final output of your testbench should match the output shown in Figure 4.1.

4.2 Register File

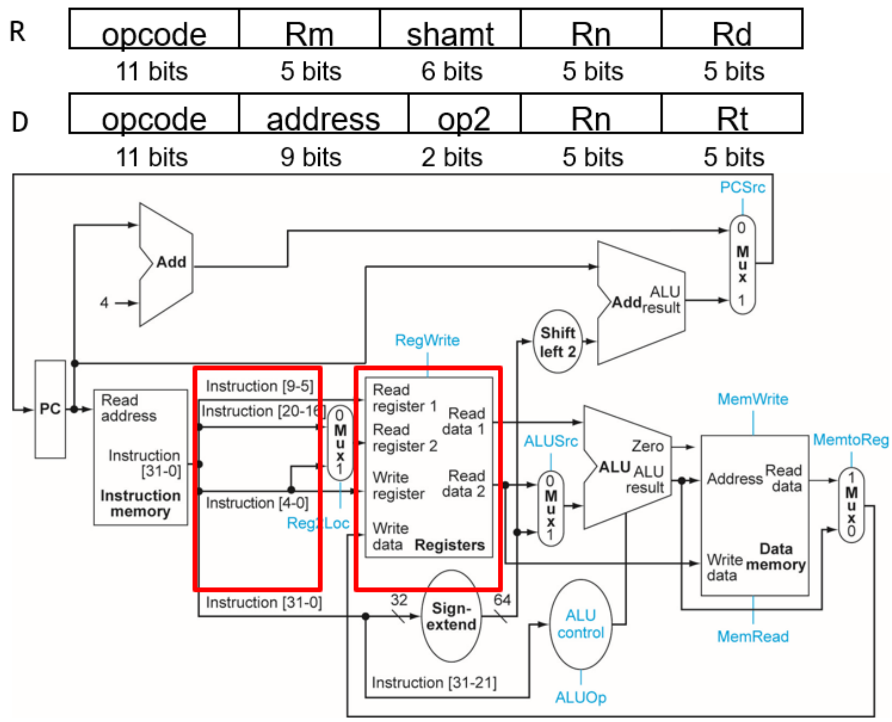
Next, we will create the register file. The register file is a piece of memory in the processor that holds the 32 register values that are used by most instructions

(X0-X31). You will create a new module called regfile (in regfile.v). The regfile module should retrieve data from the registers on the rising edge of read_clk as well as write to the registers on the rising edge of write_clk when the regWrite flag is set. Two different clocks are used here because the regfile will be read at a different time than it is written to. The regfile should use a verilog reg array, similar to the array used in instruction memory. Since we don't currently have the ability to do loads and stores (since we don't have data memory yet), the values for the registers should be stored in a datafile, regData.data and copied into the array during the initial block, just like we did with the instr_mem.v file. regData.data is provided for you. The regfile module will have a lot of similarities to the instr_mem module, so I recommend reusing concepts and code from the instr_mem module.

Inputs to the module should include a signal called read_clk and a signal called write_clk as well as all inputs shown on the Register file in Figure 4.2. Don't forget reg_write. This is a control signal that determines whether data should be written to the register. Some instruction write to registers, others do not. The outputs should be the outputs of the Register file in Figure 4.2. Use names such as read_register1, read_data2, etc.

I have provided the majority of the testbench for this module, regfile_test.sv. It provide input values and verifies that the outputs match expected behavior. Notice how the testbench utilizes the delay module to create different clocks for read_clk and write_clk. The testbench is designed to test a variety of scenarios and to verify the timing aspects of this module. Your only job on the testbench is to fill in the correct result (cr) values in the testbench. They are currently populated with X.

Figure 4.2: Instruction Parse and Regfile Diagrams



4.3 Your Assignment

You are to:

1. Create an instruction_parse module as described above.

2. Update instr_parse_test and verify the functionality of the instruction_parse module. For this testbench, please use hex for instructions and opcodes and unsigned decimal for all other signals.

3. Create a regfile module.

4. Update the regfile_test module as described above and verify the functionality of the regfile module. For this testbench, please use signed decimal for all signals.

5. Rather than writing a lab report, please produce a landscape mode PDF file called Lab4_lastname.pdf that includes (in this order):

   (a) Your name and the lab number.

   (b) A snip of the Simulation Results for the instruction_parse module. Please show instructions and opcodes in hex and everything else in unsigned decimal.

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

   (d) A snip of the Simulation Results for the regfile module. Please show everything else in signed decimal.

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

6. Upload Lab4_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.