ELC 3338 Project Book

Steve Potter

April 23, 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
```

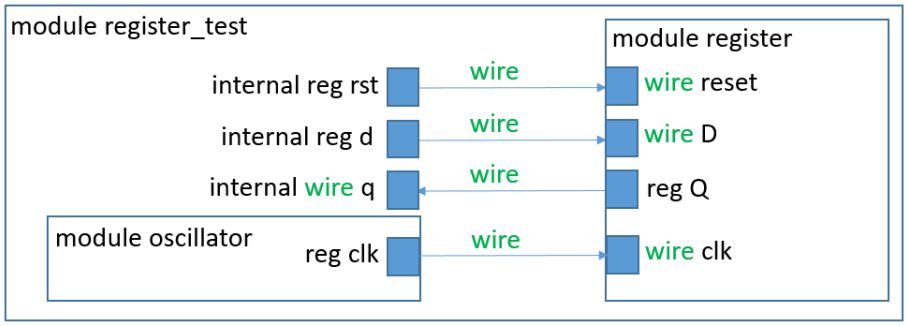
The first part is the header. We will use this same header each time. It tells the Verilog compiler to get all the data from a file called definitions.vh. The extension vh is a Verilog header. We use this to specify common pieces of data we will use across our design, so that all the components we build 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 1
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 t
// 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;
```

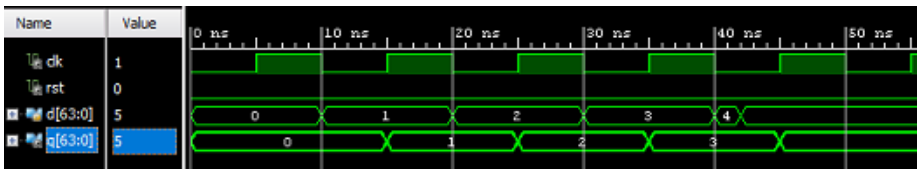
```
    verify(cr, $bits(cr), q, $bits(q));

    #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 simul
    #`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
    );

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 in-

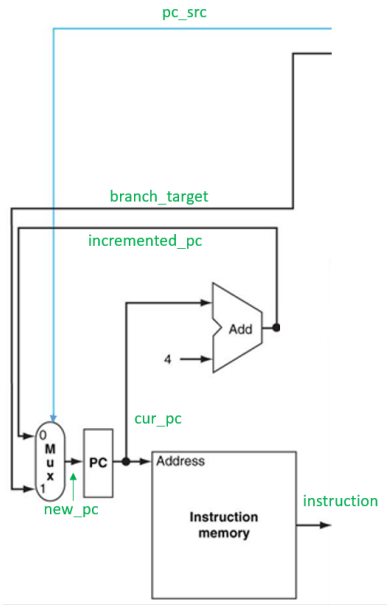
strData.data file, as this might change and it will be necessary for grading. Make sure to update the definition of IMEMFILE to point to your repository. The line of code to read the data in from the file is `$readmemh(‘IMEMFILE, imem);`

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

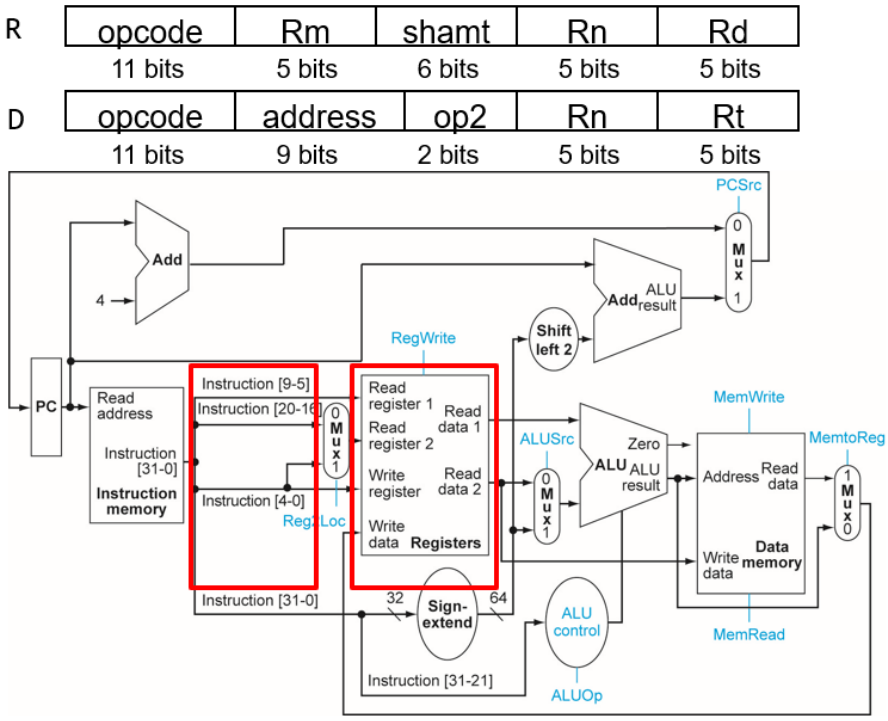
Next, we will create the register file. The register file is a piece of memory in the processor that holds the 32 register values that are used by most instructions

(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.

Lab 5

Control Unit and Sign Extender

5.1 Control Unit

Next, we will create the main control unit. You will create a new module called control (in control.v) in 2_decode. The control module should use a portion of the instruction to determine the values of all control signals to be used in our processor. These signals include:

1. reg2_loc
2. uncondbranch
3. branch
4. mem_read
5. mem_to_reg
6. alu_op
7. mem_write
8. alu_src
9. reg_write

The supported instructions should include:

1. ADD
2. SUB
3. AND

Figure 5.1: Control Value Table

Instruction	Reg2Loc	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	0	0	0	1	0	0	0	1	0
LDUR	X	1	1	1	1	0	0	0	0
STUR	1	1	X	0	0	1	0	0	0
CBZ	1	0	X	0	0	0	1	0	1

4. ORR
5. LDUR
6. STUR
7. CBZ
8. B

You will need to evaluate the incoming opcode and set the value of the control lines according to Figure 5.1. Note that a value of X in a table entry indicates that it does not matter whether the value is 0 or 1. You do not want to use X in these cases, as this indicates 'undefined' to Verilog. Instead, please use 0 in place of X.

Note that you should not use a clk in this module.

5.2 Control Unit Test

The Control Unit is crucial to operation of your datapath, so it needs to be tested thoroughly. Every supported instruction (listed above) should be tested with the Control Unit Test. To facilitate testing, I have provided a testbench that includes all opcodes and an invalid opcode. You just need to fill in the 'cr' values.

The Verilog implementation between 'if/else' statements and 'case' statements differ. 'If/else' statements will create a series of nested muxes with two inputs each, whereas a case statement will produce one large mux with many inputs. To maximize speed, we should use case statements. Note that all Verilog case statements must have a default case. But one of the challenges of this lab is dealing with opcodes of different sizes. Thankfully, Verilog has 'casex', which will only evaluate binary digits that are not labeled X. So for a CBZ instruction, you can fill in the last 3 digits with XXX and use 'casex'. The 'casex' syntax will also be very helpful in the Sign Extender. Also, please create macros in definitions.vh for the opcodes of each instruction and for the ALU Op values of each instruction type. The values in the opcode macros can include X's when necessary so that they work appropriately with the casex. For opcode macros, use the instruction type in all caps. For instance, your macro

for a load instruction should be LDUR. For ALU Op macros, use the following macro names:

1. ALUOp_RTYPE
2. ALUOp_DTYPE
3. ALUOp_B
4. ALUOp_CBZ

The default case should set all control signals to 0 to "turn off" the datapath in case of an invalid opcode. The ALU Op should be R Type.

5.3 Sign Extender

The final major component of the Decode stage is the Sign Extender. The Sign Extender should use information in the instruction to create a 64-bit output value to use as an address or branch offset. You need to extract the constant value from the instruction (this will be different for each instruction type), place that extracted value in the least significant bits of the sign_extended_output, then sign extend that value. For positive values, the sign extender should fill extended bits with 0s. For negative values, the sign extender should fill the extended bits with 1s. The sign extender should support extending address values from the following instructions:

1. LDUR
2. STUR
3. CBZ
4. B

For R-Type instructions that do not have a value to sign extend, please use the default case and set the value to 0.

Note that you should not use a clk in this module.

5.4 Sign Extender Test

The Sign Extender Test is provided for you with all 'cr' values filled in. However, half of the instructions are missing. The comments indicate what the instructions should be, and you must fill in the missing instructions with the correct machine code.

5.5 Your Assignment

You are to:

1. Create a control module.
2. Update control_test.sv with the correct 'cr' values and verify that your control module works properly.
3. Create a Sign Extender module.
4. Update sign_extender_test.sv to fill in the missing instructions, then verify that your sign extender module works properly.
5. Rather than writing a lab report, please produce a landscape mode PDF file called Lab5_lastname.pdf that includes (in this order):
   (a) Your name and the lab number.
   (b) A snip of the Simulation Results for the control module. Please show opcodes in hex and everything else in binary.
   (c) A snip of the test results from the Tcl Console for the control 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 sign_extender 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 sign_extender module. This snip should show the entire log from BEGIN TEST RESULTS to END TEST RESULTS.
6. Upload Lab5_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 6

Finishing Decode

6.1 iDecode Module

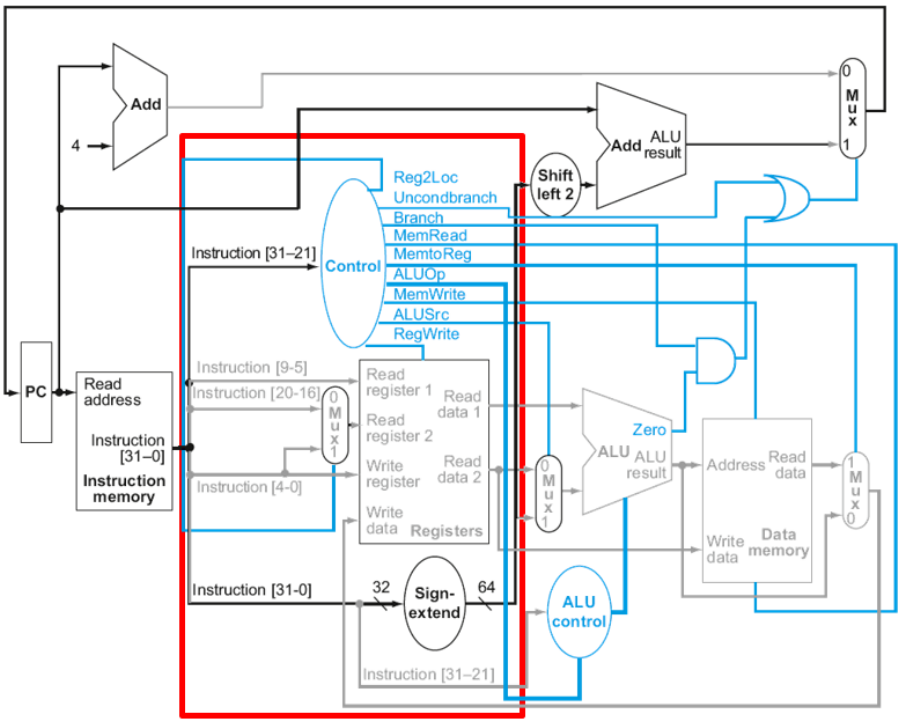
At this point, you have created all of the modules necessary to assemble the iDecode module. Now you need to create a new module called iDecode. The inputs and outputs can be determined by evaluating Figure 6.1. Any signal that crosses the boundaries of the red box is an input or output. Signals that do not cross the boundaries of the red box are signals that are internal to the iDecode module and should be declared internally in iDecode, with the exception of reg_write and reg2_loc. Please make sure to label signals consistently with lower case letters with words separated by underscores. For example, read_data1, write_data, alu_src.

6.2 iDecode Testbench

To verify that your iDecode module works correctly, you must first fill in the Expected Results Table that has been provided in the testfiles directory. All rows and columns have been defined, but you must fill the expected values into each cell. Note that, at this point, you cannot fill in the rows below write_data. If a particular data item is not relevant for a particular instruction (for instance, sign_extended_output on an R-type instruction), then put N/A in that cell. It is important that we identify which signals are applicable and which signals are not applicable. Please do not use N/A for any control signals, as we always want to make sure these are set correctly.

To ensure that we are testing each case correctly and consistently, please update regData.data to reflect the following values and use these values when making filling in your Expected Results Table. Note that the instructions execute, so if X9 is updated by the first instruction, then you should use the updated value of X9 in the second instruction, etc. However, we will not be branching at this time, so the instructions will run in order.

Figure 6.1: Expected Results



1. X19 = 10
2. X20 = 30
3. X21 = 0
4. X22 = 16

Now update the provided testbench for iDecode to provide the inputs for iDecode and verifying the outputs of iDecode. For the instruction input, use the instructions from your Expected Results Table. For the outputs, use your Expected Results Table to fill in the 'cr' values for each instruction. For instructions that update the register file, use the test bench to provide the correct value to the write_data input, since we do not yet have an ALU to do the calculations. Also, please use your test bench to provide a value of 20 to X9 in the first command (LDUR).

In the testbench, when there is a value that is N/A in your Expected Results Table, please comment out the verify function call for that signal, since we do not have a real expected result to compare it to. Doing this, my testbench ends up having 119 test cases.

6.3 Your Assignment

You are to:

1. Integrate all individual modules into the iDecode module.
2. Update your Expected Results Table to include values for all iDecode inputs and outputs.
3. Update iDecode_test.sv with values from the Expected Results Table. Note that you do not need to add any test cases, change timing, etc.
4. Verify that your simulation results match your expected results.
5. Rather than writing a lab report, please produce a landscape mode PDF file called Lab6_lastname.pdf that includes (in this order):
   (a) Your name and the lab number.
   (b) A snip of your completed Expected Results Table.
   (c) A snip of the Simulation Results for the iDecode module. Please show instructions in hex, opcodes and control signals in binary and everything else in signed decimal.
   (d) A snip of the test results from the Tcl Console for the iDecode module. This snip should show the entire log from BEGIN TEST RESULTS to END TEST RESULTS.
6. Upload Lab6_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 7

Integrating Fetch and Decode

This lab will integrate the iFetch module with the iDecode module to make a single testbench that will verify that they function properly together. The result will be a system that automatically fetches instructions and then decodes the fetched instruction. To do this, you will update the provided fd_integration.s to include an instance of both the iFetch module and the iDecode module. The instruction output from the fetch module should be connected to the instruction input in the decode module.

You will also need to thoroughly test the outputs of each stage. Refer back to your iFetch_test.sv and iDecode_test.sv files to identify what you need to test. You will need to verify every item that you verified in each of those previous tests. We will be using the instructions and expected values from the Expected Results Table in the iDecode lab. Note that this time, rather than providing the instructions in the testbench, the iFetch module should provide the instructions. Make sure to also verify the cur_pc and instruction outputs of the iFetch module. Overall, the tests should have 139 steps. 119 come from the iDecode signals and 20 come from the iFetch signals.

7.1 Your Assignment

You are to:

1. Update the provided fd_integration.sv with values from the Expected Results Table and with expected outputs from iFetch.
2. Verify that your simulation results match your expected results.
3. Rather than writing a lab report, please produce a landscape mode PDF file called Lab7_lastname.pdf that includes (in this order):
   (a) Your name and the lab number.

   (b) A snip of your completed Expected Results Table.

   (c) A snip of the Simulation Results for the fd_integration test. Please show instructions in hex, opcodes and control signals in binary and everything else in signed decimal.

   (d) Copy and paste the entire log from BEGIN TEST RESULTS to END TEST RESULTS into your file. The results have gotten too long to use the snipping tool.

4. Upload Lab7_lastname.pdf file to Canvas.

5. 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 8

ALU and ALU Control

The goal of this lab is to build the ALU and ALU Control modules. These modules can be seen in the context of the datapath by viewing the datapath diagram from a previous lab, Figure 6.1. The ALU performs the arithmetic and logical operations, and the ALU Control module determines which operation should be performed by the ALU. During this lab, we will not put these modules together. Instead, they will be tested independently of each other.

8.1 ALU

The ALU has three inputs:

1. a_in - the first input operand
2. b_in - the second input operand
3. alu_control - control signal used to tell the ALU what operation to perform

The ALU has two outputs:

1. alu_result - the result of the arithmetic/logic operation
2. zero - a flag indicating whether alu_result is zero

Figure 8.1 identifies the operation that corresponds to the alu_ctrl value. Note that we will not be implementing NOR, even though it is listed in the table. You should use a case statement to evaluate the alu_control bits to determine which ALU operation to perform. For each operation, you do not have to do anything fancy. You just need to use the math capability that verilog provides to make the calculation. To make the code readable, you must give the the ALU control bits names in the definitions.vh file, and you should use these in your cases. Please use the following names for your macros in defintions.vh:

1. ALU_AND

Figure 8.1: ALU Control Values

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	pass input b
1100	NOR

2. ALU_ORR
3. ALU_ADD
4. ALU_SUB
5. ALU_PASS

Also, don't forget to make a default case, which is needed to actually wire this up. Please use AND for the default case.

One last thing to note is the generation of the zero flag. The zero flag is determine by simply evaluating the alu_result. There are several ways to handle this, but this is the easiest way to handle it. In Verilog (like C), the statement $(y == 0)$ is an operation with a boolean output. You can thus say $x = (y == 0)$; to assign x to be the boolean value that $(y == 0)$ produced. The statement $x = (y == 0)$; is realizable as a digital comparator with y and 0 as inputs and x as the single bit output. Please note that you will not have signals called x and y. I just used these to explain the concept.

8.2 ALU Control

The ALU Control module has two inputs:

1. alu_op - 2 bit signal giving incomplete information about what ALU operation should be performed
2. opcode - the opcode of the instruction

The ALU Control module has one output:

1. alu_control - control signal used to tell the ALU what operation to perform

You have already defined the ALUOp macros in definitions.vh when you created the control module. You have also already defined the ALU Control macros in definitions.vh in the previous section. And you have macros defined for

Figure 8.2: ALU Op Table

ALUOp		Opcode field											Operation
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[24]	I[23]	I[22]	I[21]	
0	0	X	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	X	0111
1	X	1	0	0	0	1	0	1	1	0	0	0	0010
1	X	1	1	0	0	1	0	1	1	0	0	0	0110
1	X	1	0	0	0	1	0	1	0	0	0	0	0000
1	X	1	0	1	0	1	0	1	0	0	0	0	0001

instruction opcodes. Please use these macros when creating the ALU Control module.

Figure 8.1 identifies the operation that corresponds to the alu_ctrl value. This module can be implemented multiple ways, but we want to use the most efficient way possible. The most efficient method is to use a casex statement to evaluate the alu_op signal, utilizing the information in Figure 8.2. alu_op will provide all information necessary for D-Type and CB-Type instructions. For R-Type instructions, you should use the bits of the opcode to set the alu_ctrl value. When evaluating Figure 8.2 for R-Type instructions, certain bits of the instruction correspond to alu_ctrl bits. The magic decoder ring is listed below. Please note that the numbering system for the instruction bits includes the entire instruction, whereas you will just have the opcode to work with. Therefore, you will need to adjust the numbering to account for your 11-bit opcode.

1. ALU Ctrl [3] = 0
2. ALU Ctrl [2] = I[30]
3. ALU Ctrl [1] = I[24]
4. ALU Ctrl [0] = I[29]

Please use AND as the default case. For the B instruction, the ALU Op should be 00 from your control module. Since 00 is also the ALU Op for LDUR and STUR (which have ALU Control value of ADD), you can just allow the B instruction to also set ALU Control to ADD.

8.3 Your Assignment

You are to:

1. Create the ALU module.
2. Verify that your ALU module works correctly by running it against the provided test bench.
3. Create an ALU Control module.

4. Verify that your ALU module works correctly by running it against the provided test bench.
5. Rather than writing a lab report, please produce a landscape mode PDF file called Lab8_lastname.pdf that includes (in this order):
   (a) Your name and the lab number.
   (b) A snip of the Simulation Results for the each test bench. Please show instructions in hex, opcodes and control signals in binary and everything else in signed decimal.
   (c) For each test bench, copy and paste the entire log from BEGIN TEST RESULTS to END TEST RESULTS into your file.
6. Upload Lab8_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 9

Execute Stage

9.1 Execute

In the last lab, you created the ALU and ALU Control modules. Now we will finish the iExecute stage. The iExecute stage is represented by the red box in Figure 9.1. To finish the iExecute stage, you will need to add the following:

1. Mux to select the source of the second input into the ALU. You can reuse your mux that you created in the iFetch stage.
2. Shifter to left shift the sign extended branch address offset. You will need to create a new module for this.
3. Adder to add the branch address offset to the current PC. You can reuse your adder that you created in the iFetch stage. Please note the adder is near the top of the datapath diagram and it has an output named ALU result on the diagram. In fact, this is not a full ALU, but just and adder. And our output signal will be named branch_target, as this is what it really is.

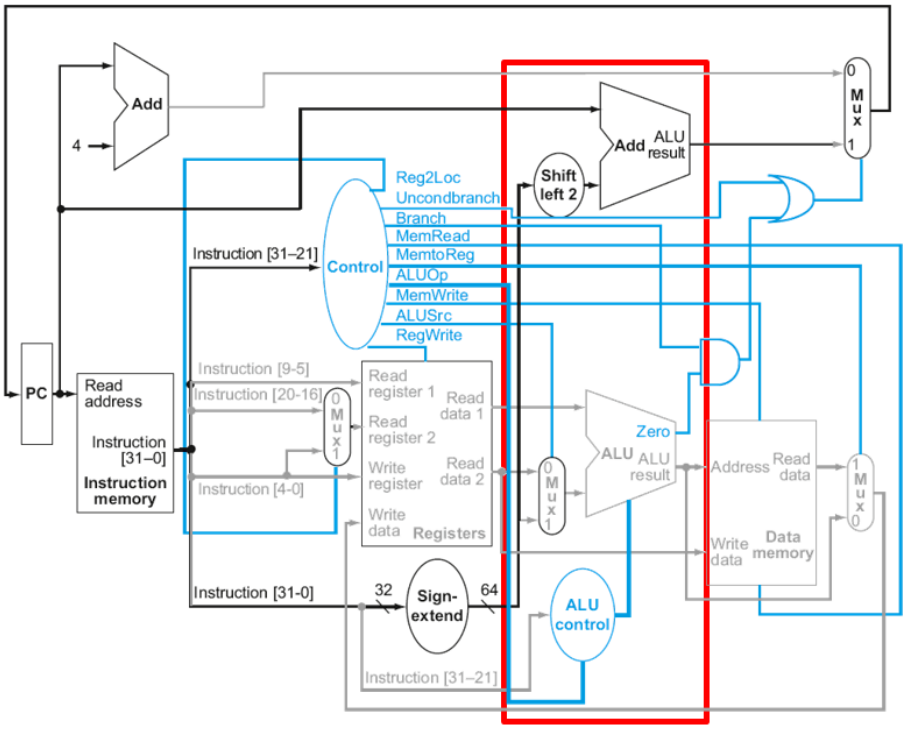
These five modules should be included in a new module called iExecute. iExecute should consist of everything shown in the red box on Figure 9.1.

The testbench will run the 10 instructions in your Expected Results Table. Therefore, your next step is to update your Expected Results Table by updating the 3 outputs of the iExecute stage. You also need to update 1 input to the table, PC. The PC value should start at 0 and increment by 4 for each instruction. Populate the iExecute output rows with expected values for each instruction. All of the inputs that you need to determine the expected result are in the table. Make sure to use N/A if the signal is not applicable for a given instruction.

Finally, you need to complete the testbench. I have provided the bulk of the testbench. The only updates that you need to make to the testbench are:

1. I have X for all cr values right now. Please update these to the correct values per your Expected Results Table.

Figure 9.1: Execute Stage



2. I have a verify function call for all 3 outputs for every instruction. If an output is N/A for a particular instruction, remove or comment out that verify. There should be a total of 20 test cases.

9.2 Your Assignment

You are to:

1. Complete the iExecute module
2. Update your Expected Results Table with the outputs from the iExecute stage. Also add a PC value.
3. Update iExecute_test.sv
4. Verify that your simulation results match your expected results.
5. Rather than writing a lab report, please produce a landscape mode PDF file called Lab9_lastname.pdf that includes (in this order):
   (a) Your name and the lab number.
   (b) A snip of your completed Expected Results Table.
   (c) A snip of the Simulation Results for the iExecute test. Please show instructions in hex, opcodes and control signals in binary and everything else in signed decimal.
   (d) Copy and paste the entire log from BEGIN TEST RESULTS to END TEST RESULTS into your file. The results have gotten too long to use the snipping tool.
6. Upload Lab9_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 10

Integrating Fetch and Decode

10.1 Integration

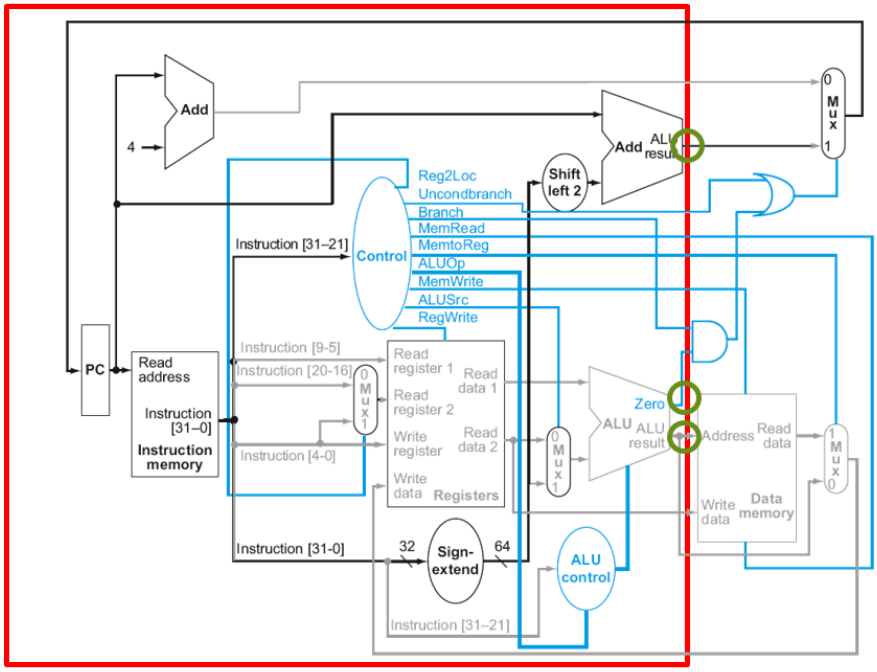
We now have working Fetch, Decode, and Execute modules. Now it is time to put them together to produce a system that can:

1. Update the program counter
2. Read the appropriate instruction from the instruction datafile
3. Read the correct registers
4. Update all control lines
5. Sign extend address data
6. Calculate Branch Target Addresses
7. Provide a zero bit for conditional branch instructions
8. Produce ALU results for R-Type and D-Type instructions

Once we can do all of this, we will be ready for the iMemory stage. We currently have the fetch and decode module integrated into fd_integration.sv. We also have a working execute module. Today we need to integrate the execute module with the fetch and decode stages. The new testbench should be called fde_integration.sv. Please reuse the instructions from your Expected Results Table that you used when you integrated fetch and decode. Once integrated, you should be able to produce a simulation that includes the outputs of fetch and decode as well as 3 new outputs from execute:

1. Branch Target

Figure 10.1: Execute Stage



2. ALU Result

3. Zero

These three new outputs are marked on Figure Figure 10.1. To verify these outputs, you should have already updated the Expected Results Table to include these three outputs. To test this integration, I recommend copying the contents of fd_integration.sv into fde_integration.sv and then modifying from there, adding your execute module, cr values, verify statements, etc that are necessary to verify the integration of the execute module. fd_integration had 139 test cases, so fde_integration should have 159 test cases.

10.2 Your Assignment

You are to:

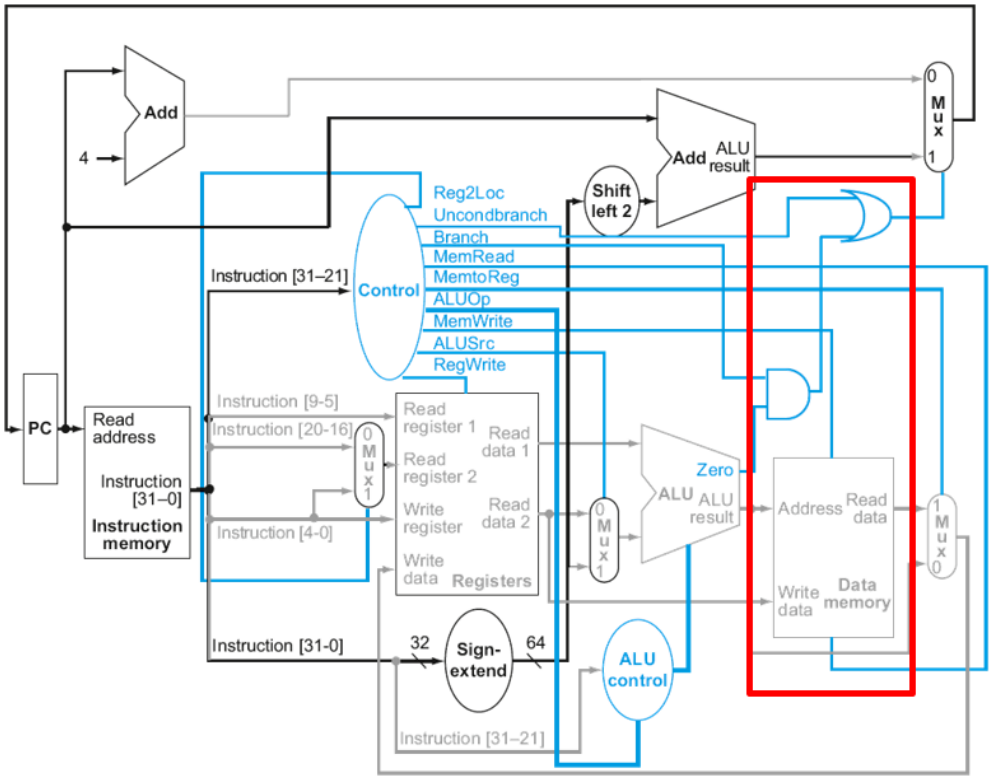
1. Create fde_integration.sv

2. Verify that your simulation results match your expected results.

3. Rather than writing a lab report, please produce a landscape mode PDF file called Lab10_lastname.pdf that includes (in this order):
   (a) Your name and the lab number.
   (b) A snip of your completed Expected Results Table.
   (c) A snip of the Simulation Results for the iExecute test. Please show instructions in hex, opcodes and control signals in binary and everything else in signed decimal.
   (d) Copy and paste the entire log from BEGIN TEST RESULTS to END TEST RESULTS into your file. The results have gotten too long to use the snipping tool.
4. Upload Lab10_lastname.pdf file to Canvas.
5. 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 11

Memory



11.1 Memory Stage

Today we will create the iMemory stage of our processor. This stage contains the data memory for the system that we use for load and store commands. It also contains the logic gates used to produce the pc_src signal that is used in the iFetch stage. Note that although the diagram shows the pc_src mux on the right side of the diagram, the mux is actually already implemented in the iFetch stage and belongs in the iFetch stage.

Because this module is so simple, we will put all of the code for the module in a new file called iMemory.v. This file will read and write data memory as well as determine the branch result.

11.2 Data Memory

The data memory will be similar to the register file memory, with two primary changes:

1. reading is now conditional on the MemRead control wire being high. If the MemRead flag is not high, then the read_data output should be set to Z (high impedance).

2. writing is now permissible if the MemWrite control wire is high.

In iMemory.v, copy the contents of register_file.v and modify it to meet the needs of the data memory module. I have provided a new data file, ramData.data that contains the initial values to be read into memory.

11.3 Branch Resolution

We now have all the information necessary to decide if the computer should branch or not. We have the signal 'branch' to tell us if it is a conditional branch command, and we have 'zero' to tell us if the condition was met. Both branch and zero must be true so we will combine them with an 'and' gate.

We also need an 'or' gate to 'or' together the output of the 'and' gate (above) and the uncondBranch control signal. These gates can be included directly in your iMemory module. While these are two separate gates on the Datapath Diagram, the code can be written as a single line of code (or you can write it as two lines, either way is fine).

11.4 Test Bench

For this lab, we will just have a single test bench that will test the entire iMemory stage. The provided test bench is complete except that the cr values are not filled in. Please fill in these values with the correct results that you expect to get.

11.5 Your Assignment

You are to:

1. Create a new module called iMemory as described above.
2. Update the testbench cr values to verify that the iMemory stage works properly.
3. Rather than writing a lab report, please produce a landscape mode PDF file called Lab11_lastname.pdf that includes (in this order):
   (a) Your name and the lab number.
   (b) A snip of the Simulation Results for the iMemory test. Please show instructions in hex, opcodes and control signals in binary and everything else in signed decimal.
   (c) Copy and paste the entire log from BEGIN TEST RESULTS to END TEST RESULTS into your file. The results have gotten too long to use the snipping tool.
4. Upload Lab11_lastname.pdf file to Canvas.
5. 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 12

Write Back

12.1 Mux

This stage consists on only one item, a mux to select between the output of memory and the output of the ALU. The control is the MemtoReg control line, see Fig 12.1. Since the mux has already been tested it does not need a testbench. The stage thus has only 3 inputs (2 data and 1 control) and one output, the result.

12.2 Datapath

You are ready to assemble the full non-pipelined datapath shown in Fig 12.2. To do this, you will need to combine all 5 stages into datapath.sv. datapath.sv is the top-level file, so it is the testbench. Verify operation of your datapath by running your set of instructions in instrData.data and testing the output. These instructions are the 10 instructions from the Expected Results Table.

Now that we have a writeback stage, we do not need to set write_data in the initial section of datapath.sv. Rather, you should connect write_data from the WriteBack stage to the Decode stage. Because we now have a memory stage, we should no longer need to set pc_src in the inital section of datapath.sv. However, our test instructions are not meant to run like a program and would yield strange results. So for right now, we want to keep pc_src hard-coded to 0 in datapath.sv. To do this, we will use a new reg called pc_src_tmp and set it to 0. Then we will use pc_src_tmp as the input to the iFetch module. pc_src will be a wire that is an output of the iMemory module. In part 2 of this lab, we will get rid of pc_src_tmp and connect pc_src from iMemory to iFetch.

I have provided a mostly complete datapath.sv. The initial section is complete, including the cr values and all verifies. Your job is to add your modules at the top and to define the clocks for those modules. I have already added iFetch because I wanted to try to make it clear what we are doing with pc_src_tmp.

Figure 12.1: Write Back

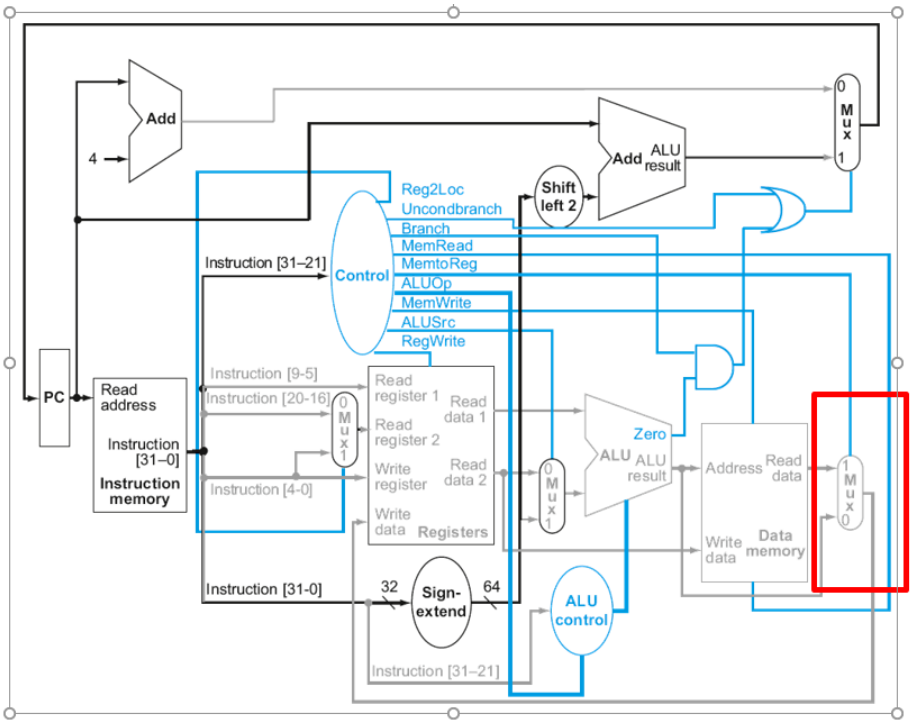
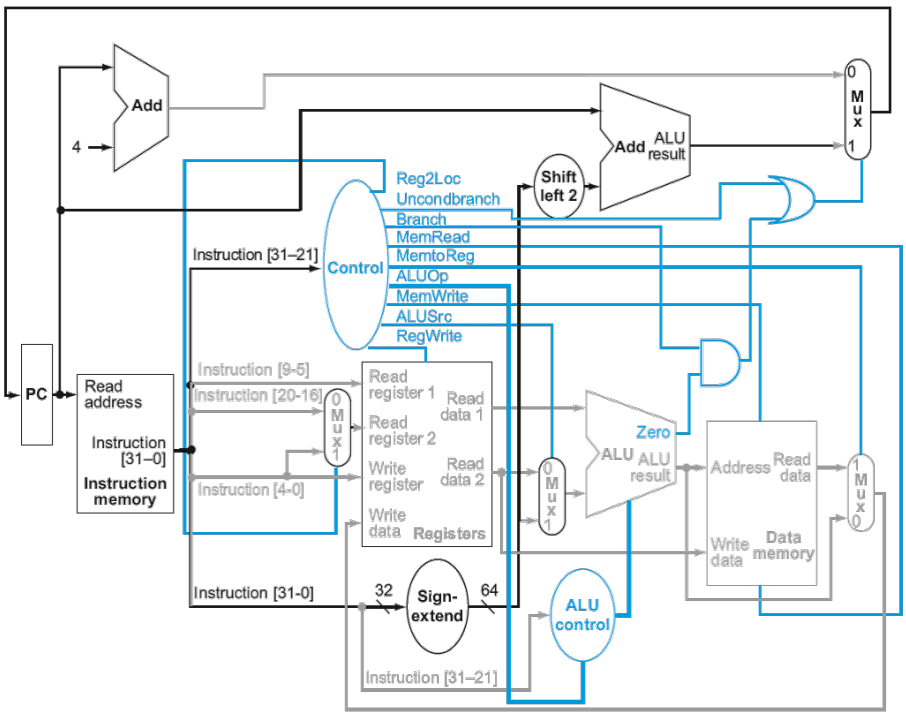


Figure 12.2: Full Non-Pipelined Datapath



Once you add your modules and clocks, you should be able to run datapath.sv and get 175 test results (hopefully all passing!).

12.3 Your Assignment Part 1

You are to:

1. Update your Expected Results Table to include the iWriteBack stage.
2. Create the iWriteback stage consisting of one Mux.
3. Integrate all five stages into the file datapath.sv.
4. Run simulations to verify that your results match your Expected Results table.
5. There will be one final test to add before we submit it. However, please save copies of the following:

(a) A snip of the Simulation Results for datapath.sv. Please show instructions in hex, opcodes and control signals in binary and everything else in signed decimal.

(b) Copy and paste the entire log from BEGIN TEST RESULTS to END TEST RESULTS into your file. The results have gotten too long to use the snipping tool.

12.4 Division

To fully verify your LEGv8 processor, we are going to write a program that can divide two numbers, run that program and verify that we get the correct result. The program that we will run is shown in the file division.c, shown below. At the top of that file, I've listed some rules about the contents of the regData file and the ramData file. In previous semesters, I have made students translate this C code into assembly, then translate that assembly into machine code, put that machine code in instrData, update regData, and update ramData. This semester, to lighten your load, I am providing 3 new test files:

1. instrData_division.data
2. regData_division.data
3. ramData_division.data

While I am providing these files, please open the up and analyze the contents. To utilize these files, you need to copy them into your testfiles directory and update definitions.vh to point to these files. instrData_division contains the machine code for the C code listed below. The assembly translation is also shown below. regData_division sets all register to 0 except for A22, which it sets to 24. This value is the base address of the array A. This is the only non-zero value allowed in regData. ramData_division has all values set to 0 except for the values of A[0], A[1], A[2], and A[3]. I set the dividend to 56 and the divisor to 8. The quotient is initially set to 0. A[3] is set to 1. This is necessary because we do not have immediate instructions in our processor.

To further simplify the lab, I have provided a test bench (division.sv) that is complete except for the modules and clocks. Please make sure that for this part, you connect pc_src from iMemory to iFetch. These instructions are meant to run as a program and correct branch decisions are vital to its operation. The test bench verification is very simple. It has one test case, which verifies the value that is being stored by the final instruction. Given that I have set it up to do 56/8, the result should be 7. Rather than verifying hundreds of steps along the way (which we've already done), we will very simply check to see if the division works. Feel free to modify ramData_division.data to use a different dividend and divisor...but make sure that they are divisible. 57/8 (or anything like that) will not work properly, as it will never break out of the loop.

Listing 12.1: C code for doing simple division.

```
// Rules:
// The register file must be set to all 0s except that you
// can store the base address of A in X22.  Base address of A
// should be non-zero.  Everything else must be loaded from memory.
// This is a simple division problem where having a remainder will
// break it, so choose an x that is divisible by y.


int x = A[0];  // dividend
int y = A[1];  // divisor
int z = A[2];    // quotient, should initially be set to 0
int one = A[3];

while(x != 0)
{
  x = x - y;
  z = z + one;
}

A[2] = z;
```

Listing 12.2: Assembly code for doing simple division.

```
LDUR X9, [X22, #0]
LDUR X10, [X22, #8]
LDUR X11, [X22, #16]
LDUR X11, [X22, #24]
CBZ X9, 4
SUB X9, X9, X10
ADD X11, X11, X12
B -3
STUR X11, [X22, #16]
```

12.5 Your Assignment Part 2

You are to:

1. Add your modules and clocks to division.sv.

2. Run the simulation, analyze the results and verify that the divison works correctly.

3. Rather than writing a lab report, please produce a landscape mode PDF file called Lab12_lastname.pdf that includes (in this order):

   (a) Your name and the lab number.

   (b) A snip of the Simulation Results for both the datapath.sv test and division.sv. Please show instructions in hex, opcodes and control signals in binary and everything else in signed decimal.

   (c) Copy and paste the entire log for both datapath.sv and division.sv from BEGIN TEST RESULTS to END TEST RESULTS into your file. The results have gotten too long to use the snipping tool.

4. Upload Lab12_lastname.pdf file to Canvas.

5. 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.

CONGRATULATIONS! YOU'VE JUST BUILT A SIMPLE ARMv8 PROCESSOR!!!!