ELC 3338 Project Book

Steve Potter

September 12, 2018

Contents

1 Program Counter Register **3**
- 1.1 Program Counter Register . . . 3
- 1.2 Testbench . . . 6
- 1.3 Using LaTeX for Your Write-up . . . 7
  - 1.3.1 Background . . . 8
  - 1.3.2 Compile Process . . . 9
  - 1.3.3 Getting Started with LaTex . . . 9
- 1.4 Your Assignment . . . 10

2 Program Counter Incrementer and Mux **11**
- 2.1 Incrementer . . . 11
- 2.2 Input Selection via Mux . . . 12
- 2.3 Expected Results Table . . . 13
- 2.4 Your Assignment . . . 13

3 Fetch Stage **15**
- 3.1 Instruction Memory Module . . . 15
- 3.2 Fetch Stage . . . 16
- 3.3 Fetch Testbench . . . 17
- 3.4 Your Assignment . . . 18

4 Beginning to Decode **19**
- 4.1 Instruction Decode . . . 19
- 4.2 Register File . . . 21
- 4.3 Your Assignment . . . 24

Lab 1

Program Counter Register

During the course of this semester, we will build a 64-bit computer so that we can understand how it works. 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: a header (which has the include command), a port list or interface (which specifies the signals coming in or going out of our module), and a body or implementation (which describes how to build it).

Listing 1.1: Verilog code to make a register.

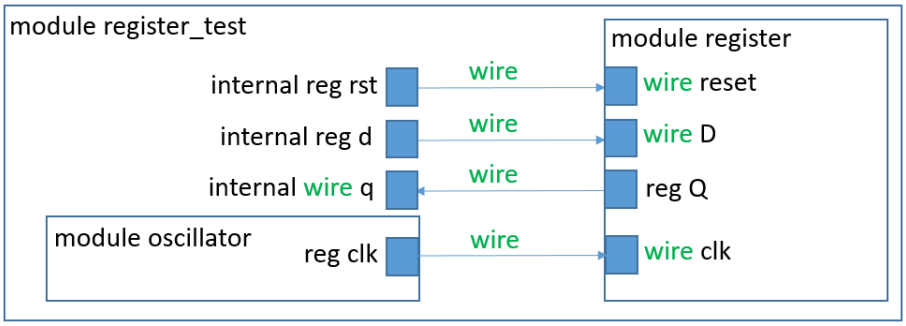
```
`include "definitions.vh"

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 component the piece of data we will be using is WORD (set to 64), which is the size of the memory addresses our computer will use (how many bits). Note that if we build things based around WORD, rather than the number 64, we can just change the value of WORD in the file and get a computer with a different size with a couple key strokes.

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

1. Wire
    (a) A wire is just a conductor that connects one component or module to another.
    (b) The value on a wire can only be changed by using combinational logic.
    (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.

Figure 1.1: Module Diagram.



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

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, 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 order as the input D.

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

[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.

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.

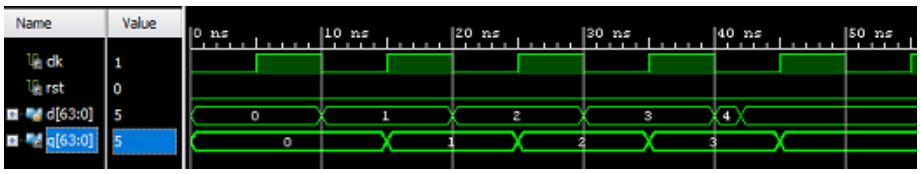
1.2 Testbench

We now want to test this. To test it, we need to tell the simulator to build a copy (instantiate) the module, and then we will need to supply the inputs and look at the outputs to verify that the module works correctly. Consider the testbench in Listing 1.2.

Listing 1.2: Verilog code to test a register.

```
`include "definitions.vh"


module register_test;

wire clk;
reg rst;
reg[`WORD - 1:0] d;
wire[`WORD - 1:0] q;

oscillator clk_gen(clk);

register UUT(
    .clk(clk),
    .reset(rst),
    .D(d),
    .Q(q)
    );

initial
begin
    rst = 0;
    d<=`WORD'd0; #`CYCLE;
    d<=`WORD'd1; #`CYCLE;
    rst = 1; #`CYCLE;
    rst = 0;
    d<=`WORD'd2; #`CYCLE;
    d<=`WORD'd3; #`CYCLE;
    d<=`WORD'd4; #(`CYCLE/5);
    d<=`WORD'd5; #(`CYCLE*4/5);
```

Figure 1.2: Timing diagram.




```
end

endmodule
```

Like our register it starts with our standard header, but this time 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 them. 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 nice 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). The initial section sets the value of the inputs then waits a CYCLE. The last couple of delays are not full cycles. I did this for two reasons:

1. To show you how to make Verilog do calculations for you.
2. To remind you that the input won't necessarily be nice and perfectly timed to your register. Unsynchronized signals happen, and is a frequent cause of problems, hence the need to test.

This is by no means an exhaustive testbench, but run it and look at the output. Does it do what you expect? What else might you want to test? Add this to your testbench and run it again to see if the register works.

1.3 Using LaTeX for Your Write-up

This section was originally written by Dr. Schubert, so any first person references are from Dr. Schubert.

All that is left is to write it up. I am going to have you use LaTeX to do your lab reports. Note how I include files, programs, and images. It is worth noting that LaTeX will automatically make the table of contents and bibliography for you also.

Why use LaTeX ? There are lots of reasons, but here are a few that matter in this course:

1. It typesets programs from the actual source, no need to copy the program and have spell checkers and grammar editors mess things up.

2. It quickly and correctly handles equations.
3. It automatically handles table of contents and bibliographies.
4. It is free, and generates high quality documents (book quality) - it is open source since before open source.
5. It is used in publication of research documents.
6. It is the only large program believed to be error free in its source code, and have no missing features (development is complete!)

1.3.1 Background

TEX refers to both a language for typesetting and the program (compiler actually) that does the typesetting. LATEX is a macro package which sits on top of TEX and provides additional functionality, and has become synonymous with the language variant (dialect) of TEX which it created. Since LATEX is hugely popular and really useful, TEX and LATEX have become synonymous to most people, and I will treat it so from now on. A note on pronunciation: TEX is in Greek letters - tau epsilon chi and hence is pronounced 'tek' not tex (similar for LATEX which is pronounced 'lay-tek' not latex).

TEX is not a WYSIWYG (what you see is what you get) typesetting program like many editors you are familiar with, as it was designed to be a tagged language like the more recent html (yes, TEXis older). The idea is not to spend time thinking about how it should look, but rather to classify what it is and let the automated standards set the text by what the text is[2]. To provide flexibility and extension (and it was designed by one of the greatest computer scientists, Donald Knuth) it was set up as a programming language with a compiler. Since LATEX is a programming language, we have a comment character % that I had to escape by putting a \before it to make it print. Whitespace past the first space (word separation) is ignored, except for a blank line, which means start a new paragraph. More than one blank line is ignored. To get more space, you issue a command, such as `\vspace{.25in}`, which puts a quarter inch of vertical space. LATEX also knows pt (points), px (pixels), pc (pica), mm (millimeters), cm (centimeters), em (width of an 'm'), and many more. By default the space is not placed if it does not separate some object (i.e. at the top of a page), but you can force it by using `\vspace*{.25in}`. Starred commands are just versions of the main command.

There are many more commands than we could describe in this brief intro, including commands to let you define new commands and environments. We will not need too many fancy commands, we only need to describe the commands to include figures, code, and equations. If you want to learn more, then I have links to free manuals online at r2labs.org.

[2]For instance, note the chapter, section, and subsection commands in the tex files. LATEX assigns a number, records it, the title, and page so it can automatically put it in the table of contents for you.

1.3.2 Compile Process

One thing that will help you a lot in working with LaTeX is how the compile process works. TeX is a two pass compiler, but it does only one pass each time it runs. Allow me a brief introduction to compilers, which is a great course if you can take it.

When you are compiling a file you have control statements (branches, loops, conditional execution statements like if or switch/case) that require you to know how many program lines ahead or behind something is in the assembled code, which you will not know at the start. While you are often just putting in a flag or label to be handled by the assembler later, you in truth don't even know if they actually put the destination of the transfer of control, and thus have an error. One easy way of handling this is to run through the process twice, collecting labels and such the first time and then doing the compile the second time through, which is what a two pass compiler does. TeX collects all the labels, notes all the chapter, section, and other structures, identifies all the bibliography references, and so on and puts them in a special auxiliary file for the next pass. It will also create a DVI file, which has most things right, but will lack table of contents, references, bibliography, and such. The second time through it already has the information before the file runs so it reads that first and uses it to create a fully correct output.

A logical question at this point is why not just have it run twice on its own? Well, in the 1980's computers were small and slow, so each run of TeX(we didn't even have LaTeX at first) took an appreciable amount of time. If you know the compile process, there are times you only have to run things once, like small spelling changes not in a title, chapter, etc. Allowing people to do only one pass at a time was a big advantage (some TeX compiles I had to do could take 10 minutes even in the 1990's). Bibliographies are handled by an external program called BibTeX, which reads the .aux file to find the references (thus you need to run LaTeX first), then pulls the data from the .bib files you specify in the calling command in your .tex file and creates a .bbl file. The .bbl file contains all the info formatted how the bibliography should look. LaTeX reads this in the first pass and copies it over to the .aux file and resolves the links to the text references. The next run of LaTeXreads all this in and places both the bibliography and the cross references. This means that to get a bibliography in you must run LaTeX BibTeX, LaTeX then LaTeX once more. You only need to do this if you add new reference, which in the labs will be once, provided you don't delete those intermediary files.

1.3.3 Getting Started with LaTex

Now that you have some background knowledge, we need to learn how to build a document on your computer. There are many ways to do this, including text editors and command line tools. I prefer using a more user-friendly editing and buidling environment. While there are numerous options available, I choose to use TexStudio. It is installed on all ECS computers, and it is available for

free download at home. I would recommend opening LabN.tex and building it before making any changes. Then make some changes, rebuild, and view those changes in the PDF that is generated. Steps to build a document in TexStudio are:

1. Open TexStudio on your lab computer
2. Use the menus open LabN.tex
3. Click on the double green arrow icon near the top. If you hover over it, it says "Build & View".
4. This should produce a PDF document on the right side of the TexStudio window

This document should build properly as long as you don't modify it. Once you start editing, it is possible that you will get compile errors. These errors are listed in the bottom pane of TexStudio. Like many compilers, they are sometimes cryptic and don't lead you directly to the problem. The most common problem (by far) is using an underscore with using the escape character (backslash) first. For example, look at the fetch1.tex file to see how I made this example_of_how_to_use_underscores.

Note that all code and image references are relative to where the .tex document is located in the file system. It is important that you maintain the same file structure that I gave you so that these references are simple and consistent.

1.4 Your Assignment

You are to:

1. Finish the testbench in Listing 1.2 by testing several additional cases. For instance, what happens when D is set at different points during the clock cycle, or if D is set for longer than a single clock cycle. Also, reset is not currently being tested. Does reset work properly? Does the register work properly after reset has been cleared?
2. Run a simulation and generate a timing diagram.
3. Write up a lab report in LaTeX following the lab format in `LabN.tex` and generate a pdf file.
4. Upload the pdf and all the Verilog files to Canvas.

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 (add 4 bytes) 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 and a testbench for the adder. 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 outputs based on how the selector bit is set. If the selector is 0 then input 0 is connected to the output, and if the selector is 1 then input 1 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 problem 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 starter code is located in ARM-Lab/code/0_common/mux.v.

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
    );

    // add mux code here

endmodule
```

Create a testbench 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

Figure 2.1: Expected Results Table

	0-10	10-20	20-30	30-35	35-55	55-65
a	0	55	55	1	1	1
b	5	5	59000	59000	24	8
add_out	5	60	59055	59001	25	9

8. We are going to change this to test it as a 64 bit mux. In your testbench, instead of creating your mux module using `mux UUT(...`, define the parameter as 64 by changing it to be `mux#(64) UUT(...` You can also do the dot notation as was done for the ports, but there are usually so few paramters you don't need to. Now come up with good values to test your mux so you are confident it works.

2.3 Expected Results Table

In order to verify that our modules work properly, we will create an Expected Results Table and compare our expected results with our simulation results. The Expected Results Table is not only critical for your own verification of your module, but it is also something that I will use heavily in grading the lab reports. In your lab reports, I need to be able to easily compare your expected results with your actual results. Of course, I will examine your test bench code as well so that I can check that your expected results are correct as well. The Expected Results Table should be done in Excel. It should have simulation time values across the top and signal names along the left-hand side. The order of the signal names in your table must match the order of the signal names in your simulation results. Then you should fill in each block of the table with the expected value for that particular signal at that particular time. Please use decimal numbers for all values in the expected results and the simulation results. See Figure 2.1 for an example Expected Results Table.

2.4 Your Assignment

You are to:

1. Write an adder.
2. Write a testbench for the adder.
3. Create an Expected Results Table for the adder.
4. Update the mux starter code to operate as a mux.
5. Write a testbench for the mux.
6. Create an Expected Results Table for the mux.

7. Run a simulation and generate a timing diagram for each testbench.
8. Compare your Simulation Results with your Expected Results Table and resolve discrepancies.
9. Write up a lab report in LaTeX following the lab format in `LabN.tex` and generate a pdf file.
10. Upload the pdf and all the Verilog files to Canvas.

Lab 3

Fetch Stage

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

3.1 Instruction Memory Module

The 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), 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]`.

The other interesting thing about this code is how to initialize the memory. The default size of the memory is 1024 elements, so we do not want to initialize this memory element by element in the code. Fortunately Verilog gives you two functions to do this automatically: $readmemb and $readmemh. The last letter specifies the base (binary or hexadecimal) of the data in the file. White space separates fields, but the underscore character is ignored and thus can be used to make the values in a number more readable. The readmemb function will be used to read the file ‘IMEMFILE and store the bits in the imem array. This is done one time on initialization. Then, you can access that data in imem at any time after that.

‘IMEMFILE is defined in definitions.vh, and I provide this file, which contains instructions. However, you will need to update definitions.vh to point to

your testfiles section rather than mine, or else it will look for mine and not find the file.

Listing 3.1: Instruction Memory

```
`include "definitions.vh"

module instr_mem#(
    parameter SIZE=1024)(
    input clk,
    input [`WORD - 1:0] pc,
    output reg [`INSTR_LEN - 1:0] instruction
    );

    reg[`INSTR_LEN - 1:0] imem [SIZE-1:0];

    //handle output
    always @(posedge(clk))
        // add one line of code here

    //initialize memory from file
    initial
        $readmemb(`IMEMFILE, imem);

endmodule
```

Starter code for the instruction memory module is given in Listing 3.1. There is only one line of code missing from this code. Fill in the line of code inside the always block to complete the module. Write a testbench and expected results table. Verify that it operates correctly. In order to confidently verify correct operation, you are required to create an Expected Results Table for your testbench and compare your simulation results with it. Please show the instruction values in hexadecimal in the Expected Results Table and the Simulation Results.

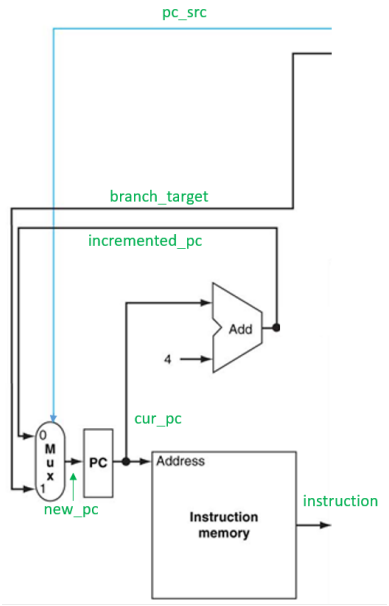
3.2 Fetch Stage

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

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. 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, I strongly recommend making it an output so that it shows up on your simulation results,

helping you to keep track of which instruction is currently executing. Also, it will be required when we start pipelining our datapath in Lab 12.

Figure 3.1: Instruction Fetch Stage.



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.

3.3 Fetch Testbench

Once you have made your connections, you should test the operation of the iFetch module by creating iFetch_test.v. You know you checked your individual modules, but there could be errors, or unexpected behavior when you put them together. Sometimes weird timings between modules causes signals to be missed and such.

Your testbench should create an instance of the iFetch module, set the inputs into the iFetch module, and verify that the correct outputs are produced. You should test both sequential operation (PC incrementing by 4) and branching. When you test branching, keep in mind that my provided instruction file (instrData.data) only contains 14 instructions, so don't branch beyond the end of the file.

As we progress through this lab, 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 delay by some number of ns. This will be useful when resolving timing issues. Make sure to switch back and forth between sequential and branching to make sure that this works properly.

In order to confidently verify correct operation, you are required to create an Expected Results Table for your testbench and compare your simulation results with it.

3.4 Your Assignment

You are to:

1. Finish the instruction memory module.
2. Write a testbench for the instruction memory module.
3. Create an Expected Results Table for this testbench.
4. Verify the results of this testbench by comparing the Expected Results Table with the Simulation Results.
5. Finish the fetch stage.
6. Write a testbench for the fetch stage.
7. Create an Expected Results Table for this testbench.
8. Verify the results of this testbench by comparing the Expected Results Table with the Simulation Results.
9. Write up a lab report in LaTeX following the lab format in `LabN.tex` and generate a pdf file.
10. Upload the pdf and all the Verilog files to Canvas.

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 instr_parse. This module will simply read inputs 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 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 create an instr_parse_test.v that will feed the module with a clock signal and instructions. Since we are not integrating with our fetch module yet, your test bench should manually set the instruction values. I am providing the testbench, shown in Listing 4.1.. For instruction inputs, it uses three of the four instructions that we recently decomposed in the lecture on machine code. I modified the ADD instruction slightly to use X10 as the destination register. I do not include the ADDI instruction because we will not be implementing immediate instructions in lab. Please make an Expected Results Table and use it to verify that your instructions are being parsed correctly.

Listing 4.1: Instruction Parse Testbench

```
`include "definitions.vh"

module instr_parse_test;

reg [`INSTR_LEN-1:0] instruction;
wire [4:0]rm_num;
wire [4:0]rn_num;
wire [4:0]rd_num;
wire [8:0]address;
wire [10:0]opcode;

instruction_parse parser(
.instruction(instruction),
.rm_num(rm_num),
.rn_num(rn_num),
.rd_num(rd_num),
.address(address),
.opcode(opcode)
    );

initial
begin
// LDUR X9, [X10, #240]
instruction = `INSTR_LEN'b11111000010011110000001010010 01;
#(`CYCLE);

// ADD X10, X21, X9
instruction = `INSTR_LEN'b10001011000010010000001010101010;
#(`CYCLE);

// STUR X9, [X10, #240]
instruction = `INSTR_LEN'b11111000000011110000001010010 01;
```

```
#(`CYCLE);

$finish;
end
endmodule
```

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. You should not use the register module that you used for your program counter. 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. Starter code is provided and shown in Listing 4.2.

Listing 4.2: Regfile Starter Code

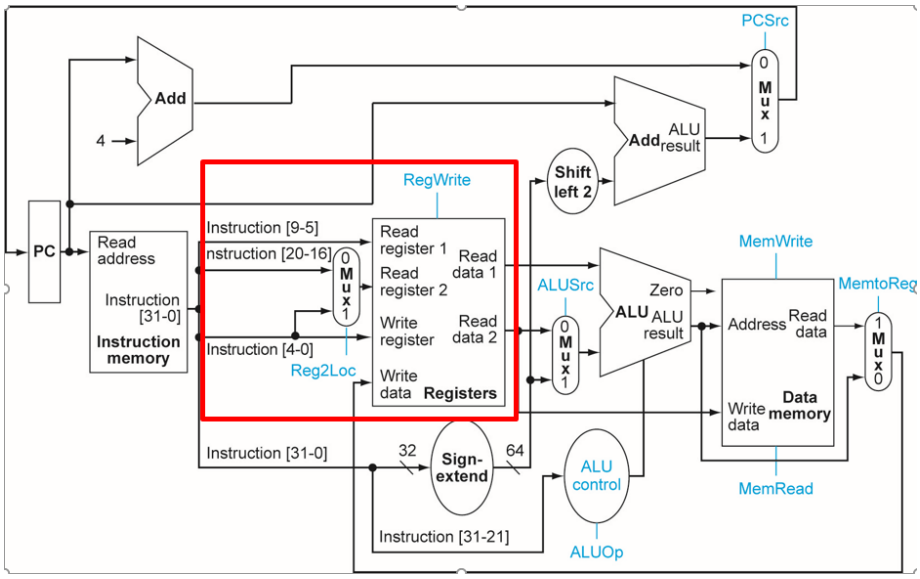
```
`include "definitions.vh"

module regfile#(
    parameter SIZE=32)(
    input read_clk,
    input write_clk,
    input reg_write,
    input [4:0] read_register1,
    input [4:0] read_register2,
    input [4:0] write_register,
    input [`WORD-1:0] write_data,
    output reg [`WORD-1:0] read_data1,
    output reg [`WORD-1:0] read_data2
    );
    reg[`WORD-1:0] rf [SIZE-1:0];

    // handle input

    always @(posedge(write_clk))begin
        // insert code to write to rf when appropriate
    end
```

```
    //handle output
    always @(posedge(read_clk)) begin
        // insert code to read from rf when appropriate
    end

    //initialize memory from file
    initial
        $readmemb('RMEMFILE, rf);

endmodule
```

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 4.1. 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 4.1. Use names such as read_register1, read_data2, etc.

You will need to write a testbench, regfile_test.v, for this module as well. It should provide values for each input and verify that the outputs match expected behavior. You should use the delay module in your testbench to create different clocks for read_clk and write_clk. Please make sure to write to the register file in your test bench, then read from that register to verify that your register writes worked properly. Also make sure to test the reg_write flag to make sure that it operates as expected.

Figure 4.1: Expected Results



4.3 Your Assignment

You are to:

1. Create an instr_parse module as described above.
2. Create and Expected Results Table for instr_parse_test.
3. Use the instr_parse test module and verify that the instruction is being parsed properly by comparing to the Expected Results Table.
4. Finish the regfile module.
5. Create a regfile_test module.
6. Create and Expected Results Table for instr_parse_test.
7. Verify that the values are being stored and retrieved from the regfile properly by comparing the results with the Expected Results Table.
8. Write a lab report according to the LabN format.