

# ELC 3338 Project Book

Steve Potter

January 31, 2018

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Program Counter . . . . .	3
1.2	Testbench . . . . .	5
1.3	Using L <sup>A</sup> T <sub>E</sub> X for Your Write-up . . . . .	6
1.3.1	Background . . . . .	7
1.3.2	Compile Process . . . . .	8
1.4	Your Assignment . . . . .	8
<b>2</b>	<b>Program Counter</b>	<b>10</b>
2.1	Incrementer . . . . .	10
2.2	Input Selection via Mux . . . . .	11
2.3	Your Assignment . . . . .	12
<b>3</b>	<b>Fetch Stage</b>	<b>13</b>
3.1	Instruction Memory . . . . .	13
3.2	Fetch Stage . . . . .	14
3.3	Fetch Testbench . . . . .	16
3.4	Lab Writeup . . . . .	16
3.5	Your Assignment . . . . .	17
<b>4</b>	<b>Beginning to Decode</b>	<b>18</b>
4.1	Instruction Decode . . . . .	18
4.2	Register File . . . . .	19
4.3	Your Assignment . . . . .	21

# Lab 1

## Introduction

In the Labs for this course we will be building a 64-bit computer, so we can understand how it works and how we can make a synthesizable machine in a hardware description language (HDL) like Verilog. In this lab we will be building the counter that sequences all our computer's instructions.

A computer has to execute one instruction after another. We will be building a system to count sequentially from some starting number. Since this counter will be used to keep our program running in order, it is called the program counter. Our system needs to hold its value, count, and be able to change the starting value. We will break this into three components: a register (to hold the data), an incrementer (to count), and a mux (to select the count or a new starting value). For today we will just build the register, simulate it, and show how to write up the lab.

### 1.1 Program Counter

The register is called the program counter, since it holds the actual count. It is the heart of our system so it is where we will start. We are going to make a module that explains how to build a register (a D flip-flop) in Verilog. Consider the 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  clk ,
    input  reset ,
    input  ['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). For outputs only, we could also have the output driven by a register (reg) or not (wire). The other two types (input and inout) are only wires. If you don't specify anything for any of the port types, 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 wires. One is the clock, which specifies the timing, and the other is reset, which clears the contents (makes the 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<sup>1</sup>. 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.

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 in Q. That is it. A nice, simple

---

<sup>1</sup>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.

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 test_regs;

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

oscillator clk_gen(clk);

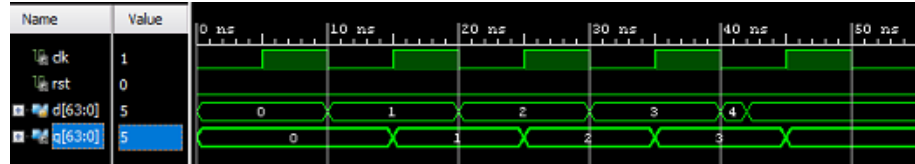
register UUT(
    .clk(clk),
    .reset(rst),
    .D(d),
    .Q(q)
);

initial
begin
    d<=WORD'd0; #CYCLE;
    d<=WORD'd1; #CYCLE;
    d<=WORD'd2; #CYCLE;
    d<=WORD'd3; #CYCLE;
    d<=WORD'd4; #('CYCLE/5);
    d<=WORD'd5; #('CYCLE*4/5);
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. In the body (implementation)

Figure 1.1: Timing diagram.



we have a bunch of things. First all the signals to our UUT must be declared. Outputs always must go to wires (the outputs are driving them, and only wires can be driven). Often inputs are registers since you will want to specify a value and have it continue until you give it a new value, though some can be wires if you had another unit that was supplying the values from its outputs. In our case, the clock signal will be driven by a module names 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 thread (it runs once at the start then never again) The initial thread sets the value of the input then waits a `CYCLE`. The last couple 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 L<sup>A</sup>T<sub>E</sub>X for Your Write-up

All that is left is to write it up. I am going to have you use L<sup>A</sup>T<sub>E</sub>X to do your labs. Note how I include files, programs, and images. It is worth noting that L<sup>A</sup>T<sub>E</sub>X will automatically make the table of contents and bibliography for you also. To top it off you only need a text editor as the files are just ASCII files. To generate the document you run the command `pdflatex` and pass it the main file, and it will generate a pdf.

Why use L<sup>A</sup>T<sub>E</sub>X ? There are lots, 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 (important given our math use).

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

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

T<sub>E</sub>X 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, T<sub>E</sub>X is 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<sup>2</sup>. 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. You will thus interact with several different programs, an ASCII text editor (to write the files), a T<sub>E</sub>X application to compile them, a pdf or dvi viewer to look at the output, and potential helper apps like dvi2ps, dvi2pdf, and their viewers. Since L<sup>A</sup>T<sub>E</sub>X 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. L<sup>A</sup>T<sub>E</sub>X 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 I can 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

---

<sup>2</sup>For instance, note the chapter, section, and subsection commands in the tex files. L<sup>A</sup>T<sub>E</sub>X assigns a number, records it, the title, and page so it can automatically put it in the table of contents for you.

to include figures, code, and equations. If you want to learn more, then I have links to free manuals online at [r2labs.org](http://r2labs.org).

### 1.3.2 Compile Process

One thing that will help you a lot in working with  $\text{\LaTeX}$  is how the compile process works.  $\text{\TeX}$  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.  $\text{\TeX}$  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  $\text{\TeX}$  (we didn't even have  $\text{\LaTeX}$  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  $\text{\TeX}$  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  $\text{\LaTeX}$  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.  $\text{\LaTeX}$  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  $\text{\LaTeX}$  reads all this in and places both the bibliography and the cross references. This means that to get a bibliography in you must run  $\text{\LaTeX}$  BibTeX,  $\text{\LaTeX}$  then  $\text{\LaTeX}$  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.

## 1.4 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 like I did.
3. Write up a lab report in L<sup>A</sup>T<sub>E</sub>X following the lab format in `LabN.tex` and generate a pdf file.
4. Upload the pdf and all the Verilog files to the course LMS.

## Lab 2

# Program Counter

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 way 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  $32/8 = 4$  bytes to a word or each instruction would be 4 addresses later. The book follows this convention so it will have 4 when it increments its 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 **Ain** and **Bin**. The output should be **add\_out**.

## 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 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 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
    );
    assign mux_out = control?b_in:a_in;
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 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 parameters you don't need to. Now come up with good values to test your mux so you are confident it works.

## 2.3 Your Assignment

You are to:

1. Write an incrementer by creating an adder.
2. Write a testbench for the adder.
3. Write a testbench for the mux in Listing.
4. Run a simulation and generate a timing diagram for each.
5. Write up a lab report in  $\text{\LaTeX}$  following the lab format in `LabN.tex` and generate a pdf file.
6. Upload the pdf and all the Verilog files to the course LMS.

## Lab 3

# Fetch Stage

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

### 3.1 Instruction Memory

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. A bunch of memory locations, 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 (mem in this case), we are going to use a parameter called SIZE to specify how big the array is: `[SIZE-1:0]`.

The other interesting thing about this code is how to initialize the memory. The default size of the memory is 1024 bytes, 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 14 instructions. However, you will need to update definitions.vh to point to your group's testfiles section rather than mine, or else it will 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];
    integer i;

    //handle output
    always @(posedge clk)
        instruction <= imem[pc/4];

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

endmodule
```

The code is given in Listing 3.1. How will it be used? What needs to be tested? Consider those questions and write a testbench and verify its operation.

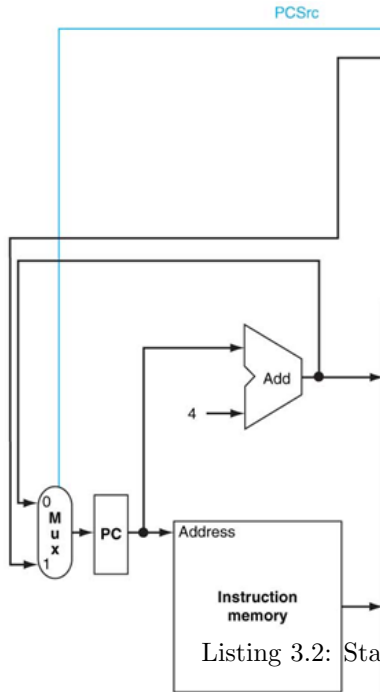
## 3.2 Fetch Stage

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) in the figure that comes in or goes out are input or output ports in 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 that it must be output from the iFetch module, I 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.

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 not every wire has a name. If a wire is unlabeled, it is worth looking at other figures (like your text in chapter 4) to figure out what the signal is.

Figure 3.1: Instruction Fetch Stage.



**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 (build one) in the iFetch module. Again choose your names wisely. I have instantiated them for you in Listing 3.2, but I haven't made the connections yet. You must use knowledge gained from the wiring diagram to hook the components together with the available wires. You should not need to create any more wires and registers. It is just a matter of connecting the components together correctly.

The code, minus the connections, is listed below in Listing 3.2.

```
'include "definitions.vh"

module iFetch#(parameter STEP='WORD'd4, SIZE=1024)(
    input  clk ,
    input  reset ,
    input  pc_src ,
    input  ['WORD-1:0] branch_target ,
    output ['WORD-1:0] incremented_pc ,
    output ['INSTR_LEN-1:0] instruction ,
    output ['WORD-1:0] cur_pc
);
wire  ['WORD-1:0] new_pc;

mux#('WORD) pc_mux(
    .a_in() ,
    .b_in() ,
    .control() ,
    .mux_out()
);

register pc_register(
```

```

        .clk(),
        .reset(),
        .D(),
        .Q()
    );

    adder_incrementer(
        .a_in(),
        .b_in(),
        .add_out()
    );

    instr_mem#(SIZE) instr_mem(
        .clk(),
        .pc(),
        .instruction()
    );
endmodule

```

### 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 set the inputs into the iFetch stage 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. Make sure to switch back and forth between sequential and branching to make sure that this works properly,

### 3.4 Lab Writeup

For the Lab 3 report, only describe iFetch.v, iFetch\_test.v, and instr\_mem\_test.v. You have already described the other files, except instr\_mem.v, which I provided. iFetch.v is all about the module connections and how the modules interact with each other. Focus the Interface, Design, and Implementation parts of your lab



report on that aspect. Then describe `iFetch_test.v`, and `instr_mem_test.v` in the Test Bench Design section, and show the results for both.

## 3.5 Your Assignment

You are to:

1. Write a testbench for the memory in Listing 3.1.
2. Finish the fetch stage and write a testbench to verify it.
3. Run the simulations and generate a timing diagrams.
4. Write up a lab report in  $\text{\LaTeX}$  following the lab format in `LabN.tex` and generate a pdf file.
5. 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 that 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 (used only in R-Type instructions)
4. Rn
5. Rd (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 on the clock edge with non-blocking procedural assignments. While this might not seem important now, it will become important later. The inputs should be a `clk` signal and 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. In future labs, we will begin treating the instructions differently and ignore the unnecessary fields. 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, you will need to create an `instr_parse_test.v` that will feed the module with a clock signal and instructions. Then you can verify that your instructions are being parsed correctly.

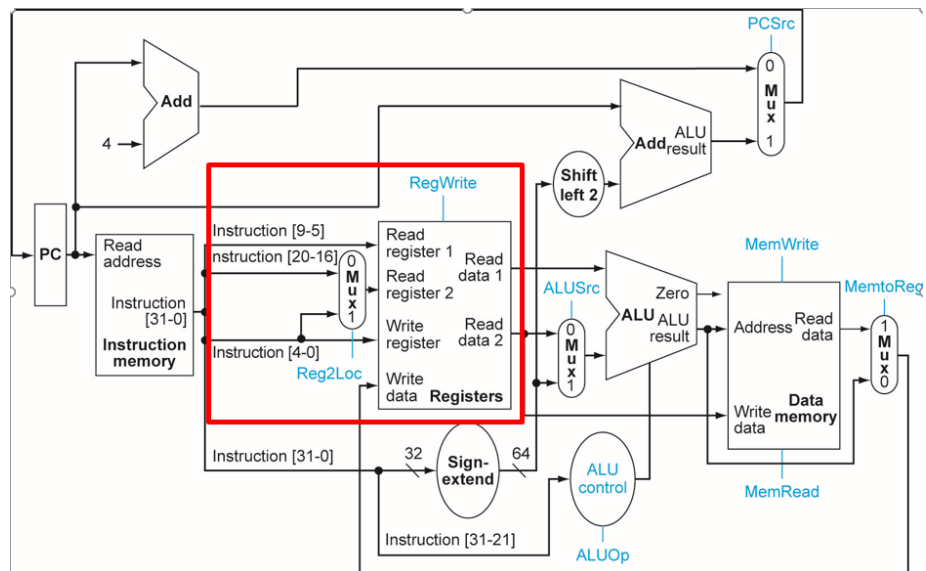
## 4.2 Register File

Next, we will create the register file. 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. The `regfile` should use a verilog `reg` array. You do not need to 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 read in during the initial block, just like we did with the `instr_mem.v` file.

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. 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.

Figure 4.1: Expected Results



## 4.3 Your Assignment

You are to:

1. Create an `instr_parse` module as described above.
2. Create an `instr_parse` test module and verify that the instruction is being parsed properly.
3. Create a `regfile` module as described above.
4. Create a `regfile_test` module and verify that the values are being stored and retrieved from the `regfile` properly.