ELC 3338 Project Book

Steve Potter

September 20, 2017

Contents

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[1]. If you changed WORD to be 64, this would automatically become 63 down to 0, which would resize the input. Pretty cool[2]! 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

[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.

[2] I grew up in the 80's so I reserve the right to say cool, rad, tubular, or any other 80's-ism. :)

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

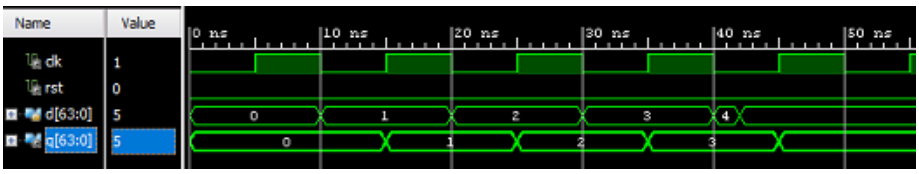
We now want to test this. To test it, we need to tell the simulator to build a copy (instantiate) the module, and then we will need to supply the inputs and look at the outputs. Consider the testbench in Listing 1.2.

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

Figure 1.1: Timing diagram.



top level simulation module - there are no ports. In the body (implementation) 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 become registers since you will want to specify a value and have it continue till 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 LaTeX for Your Write-up

All that is left is to write it up. I am going to have you use LaTeX to do your labs. 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. 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 LaTeX ? 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

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[3]. 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 TeX 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 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 I can describe in this brief intro, including commands to let you define new commands and environments. We

[3]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.

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.

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.

1.4 Your Assignment

You are to:

1. Finish the testbench in Listing 1.2.
2. Run a simulation and generate a timing diagram like I did.
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 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. A typical program counter has to deal with a variety of situations that could change the program counter.

1. The program counter should advance to the next address each cycle.

2. If the a branch is taken (from a conditional branch, unconditional branch, interrupt, or error), then the program counter should point to the branch's destination.

The most typical is the computer must fetch instructions in sequential order.

2.1 Incrementer

We will build an 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 cannot change later. 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] Ain,
    input [SIZE-1:0] Bin,
    input control,
    output [SIZE-1:0] mux_out
    );
    assign mux_out = control?Bin:Ain;
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 paramters 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 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 an integer, and gets stored in a room (memory location), which we can find by its room number (memory address). To get a piece of data, like an instruction, stored in memory we need to take its address, go to that location, and grab the value. A bunch of memory locations, accessed by an address is called an array. Arrays 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 4KB, which is impractical to setup using for loops and such, so we read it from memory. 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.

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. What needs to be tested? How will it be used? What can go wrong? Consider those questions and write a testbench and verify it's operation.

3.2 Fetch Stage

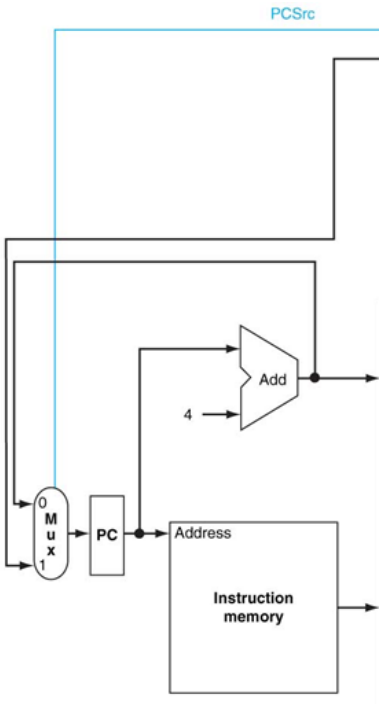
Now we need to connect it together. The components of our instruction fetch (sometimes called ifetch or just fetch) stage are shown in Figure 3.1.

Any wire (or reg) in the figure that comes in or goes out are ports. In Figure 3.1, the blue wire is a control signal and comes ultimately from the control unit, which you will build in the next stage called decode. Wires (or regs) that are completely contained in the figure are local and are thus defined in the module. This is important to notice when setting up your own modules, so try to figure out which each is before looking at the starter code. Equally important is determining the size of each signal (wire or reg), and following the convention (big or little endian). 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 get the names. In some cases the names don't matter, so a suitable name, that expresses the signal identity, location, and/or use is advisable. One nice convention[1] I have used at times would label the program counter signal in the fetch stage as `PC_fetch`. Similarly the next program counter signal passed from fetch to decode would

[1] Other conventions to consider are: CamelCase, underscore_separation. I honestly mix them when I am doing small projects or when I want to expose students to both, but in a formal design you should pick one and stick with it. Mistyping a variable is a frequent, and annoying, source of errors.

be `nPC_fetch-decode`. Note you get directionality also. Sometimes this helps, sometimes it is cumbersome, it mainly helps in debugging and modifying. Good names are slower now when it is easy (setup), and faster later when it is hard (debug), and is one of my favorite take-aways from the Agile design method called extreme programming. Most people are in a rush to get something done, so they make the design impossible - don't fall in this trap!

Figure 3.1: Instruction Fetch Stage.



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). Again choose your names wisely. I have instantiated for you, but I don't have my connections done. Well I have one, and that is because it is a tricky one. An output in Verilog cannot be read inside the module that creates it. In our case nPC must be an output, but also needs to be read internally. I handle this by creating a local wire that can be read and assigning its value to the output. I thought this would be sneaky at the start so I wired it up. You must use the wiring diagram to hook the components together with the available wires.

The code, minus the connections is listed below in Listing 3.2. Once you have done your connections you should test how well the code works. You will only be able to see nPC and instruction coming out, so think how you can simulate the behavior of a computer, and what you should check. You know you checked your individual modules, but there could be errors, or unexpected behavior. Sometimes weird timings between modules causes signals to be missed and such. Think about what could happen and how you would have to recover (use of reset). Test sequential and branching. You can break your tests into several small tests too, which is often easier. Calling them something like "iFetch_test_sequential.v" to distinguish the contents is helpful later.

Listing 3.2: Starter code for the fetch stage.

```
`include "definitions.vh"

module iFetch#(parameter STEP='WORD'd4, SIZE=1024)(
    input clk,
```

```
    input reset,
    input PCSrc,
    input [`WORD-1:0] BrDest,
    output [`WORD-1:0] nPC,
    output [`INSTR_LEN-1:0] IR
    );
    wire [`WORD-1:0] PC;
    wire [`WORD-1:0] new_PC;
    wire [`WORD-1:0] nextPC;

    assign nPC=nextPC;

    mux#(`WORD) PCsel(
    .Ain(),
    .Bin(),
    .control(),
    .mux_out()
    );

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

    adder incrementer(
    .Ain(),
    .Bin(),
    .add_out()
    );

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

3.3 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 LaTeX following the lab format in `LabN.tex` and generate a pdf file.
5. Upload the pdf and all the Verilog files to the course LMS.

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 iDecode. This module will simply read inputs and assign appropriate output values. The outputs are listed for you in the provided starter code. 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.

Listing 4.1: Verilog starter code for the iDecode module.

```
`include "definitions.vh"
```

```
module iDecode(
    input clk,
    input reset,
    input [`WORD-1:0] nPC,
    input [`INSTR_LEN-1:0] Instruction,
    output [4:0]Rm_num,
    output [4:0]Rn_num,
    output [4:0]Rd_num,
    output [8:0]Address,
    output [10:0]Opcode
    );

    // Add code to populate outputs

endmodule
```

To test this, you can re-use your iFetch test. You will need to add an instance of the iDecode module to your iFetch test, connecting some of the outputs of the iFetch module to the inputs of the iDecode module.

Another aspect to the testing of this is that you must replace the instructions in fibI.data with real instructions that represent the C code:

A[12] = h + A[8] - A[4];

Assume the following:

1. X9 and X10 are temporary registers
2. h is already loaded in X21
3. The base address of A is in X22

You should end up with 5 instructions to represent this code. You should create a table that contains the contents of the instructions. It should be similar to the table in the Example problem on page 86 of the textbook (decimal format). You will then need to break the instructions down into their binary form and input the binary into fibI.data. Note that fibI.data should only be edited with a unicode text editor. You should replace the current fibI.data instructions with these instructions, deleting the old instructions.

4.2 Your Assignment

You are to:

1. Generate assembly code for the C statement above.
2. Generate a table of the instruction fields for each instruction (in decimal format).
3. Update your fibI.data file with these instructions
4. Populate the outputs of the iDecode module.
5. Add the iDecode module to the testbench, run a simulation and generate a timing diagram.
6. Verify that the outputs of your test match your table of instruction fields.
7. There will not be a submittal on Canvas today. However, please show me your results either today or at the beginning of lab time during the next class period. We will be adding more functionality next time.

Lab 5

Register File

5.1 Register File

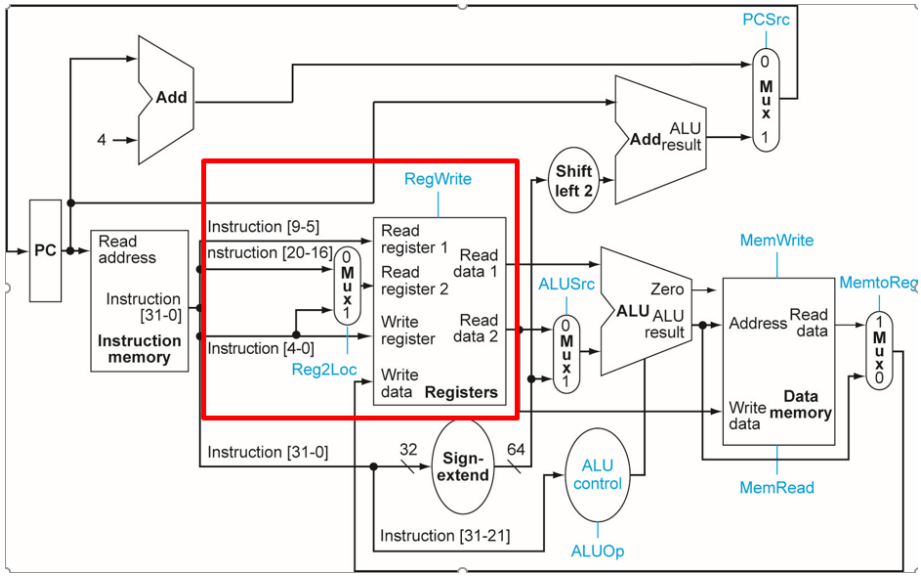
Next, we will create the register file. You will create a new module called regfile (in regfile.v), and this module will be part of your iDecode module. The regfile module should retrieve data from the registers as well as write to the registers when the regWrite flag is set and the clock edge rises. 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, fibR.data and read in during the initial block. In order to get results that match my provided results (and match the provided unit test), you should set X21=16, X9=33, X10=12 in fibR.data.

A unit test is provided, iDecode_test.v. It simply sends two commands (the add and subtract commands from last lab) and sets a few flags. Using that unit test, you should be able to produce the results shown in the lab manual. Note that in addition to the register file, you will also need to create a mux to determine the source of the second read register. Note that because we are sending the instructions from the test instead of from the iFetch routine, you will need to update your code from last lab to use procedural statements triggered on the clock edge. Otherwise, the instructions will not come in on the clock edge as expected. Timing is very important in this lab.

Listing 5.1: Verilog starter code for the iDecode module.

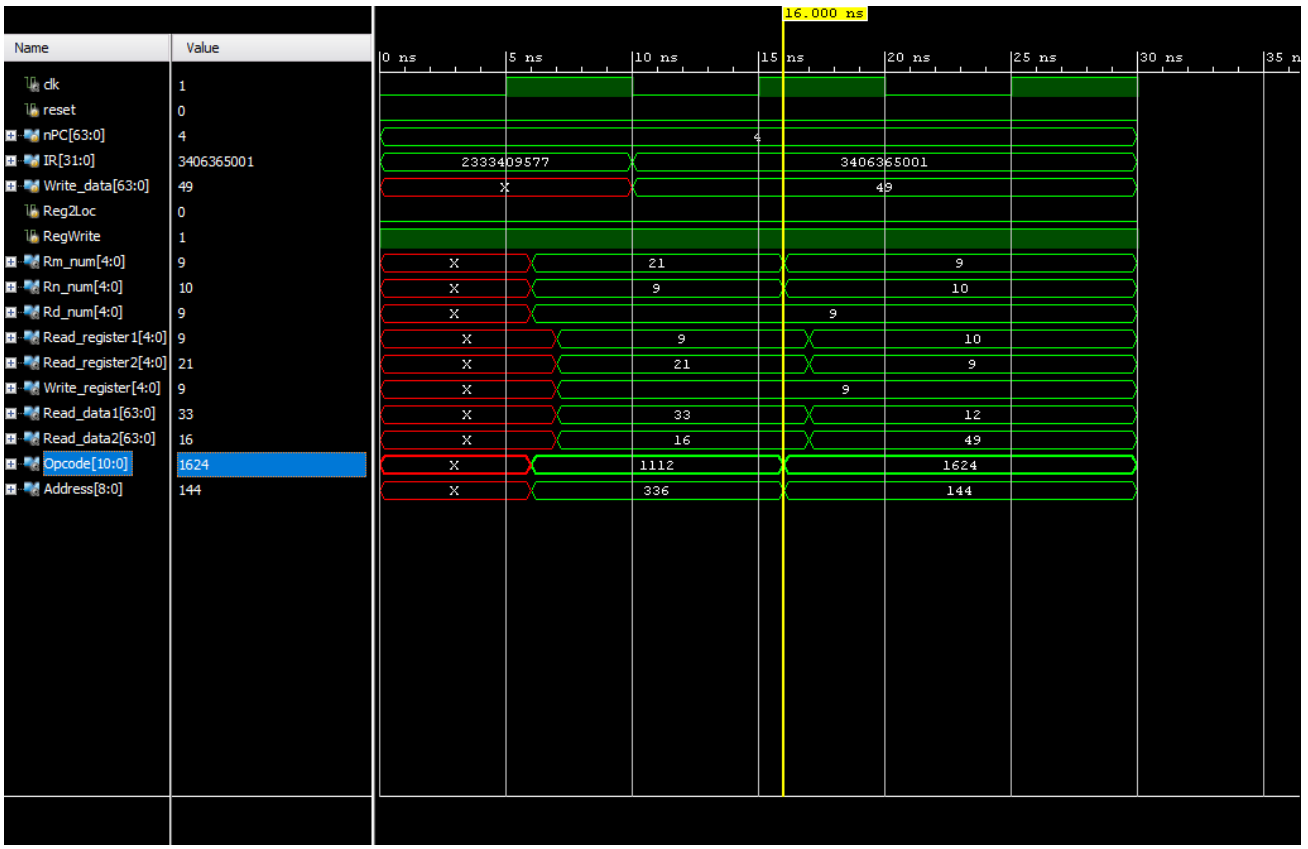
```
`include "definitions.vh"

module iDecode(
    input clk,
    input reset,
    input [`WORD-1:0] nPC,
    input [`INSTR_LEN-1:0] Instruction,
    input [`WORD-1:0] Write_data,
```

Figure 5.1: Expected Results




```
    input Reg2Loc,
    input RegWrite,
    output reg [4:0]Rm_num,
    output reg [4:0]Rn_num,
    output reg [4:0]Rd_num,
    output reg [4:0]Read_register1,
    output reg [4:0]Read_register2,
    output reg [4:0]Write_register,
    output ['WORD-1:0]Read_data1,
    output ['WORD-1:0]Read_data2,
    output reg [8:0]Address,
    output reg [10:0]Opcode
    );

endmodule
```

Figure 5.2: Expected Results



5.2 Your Assignment

You are to:

1. Create a regfile module and add it to the iDecode module.
2. Re-use your mux from previous labs and add it to the iDecode module.
3. Import the provided iDecode_test.v file.
4. Verify that your results match the expected results.
5. There will not be a submittal on Canvas today. However, please show me your results either today or at the beginning of lab time during the next class period. We will be adding more functionality next time.

Lab 6

Control Unit and Sign Extender

6.1 Control Unit

Next, we will create the main control unit. You will create a new module called control (in control.v), and this module will be part of your iDecode module. 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. Reg2Loc
2. Uncondbranch
3. Branch
4. MemRead
5. MemtoReg
6. ALUOp
7. MemWrite
8. ALUSrc
9. RegWrite

The supported instructions should include:

1. ADD
2. SUB
3. ADD

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

You will need to evaluate the incoming instruction and determine what value to set for each control line. Inputs and outputs should match Figure 4.23 in the textbook.

6.2 Sign Extender

The final 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 offset. The sign extender should support extending address values from the following instructions:

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

6.3 iDecode Test

I've made additional updates to the iDecode test, including remove Reg2Loc and RegWrite, since these will now be produced by the control unit. The unit test now has instructions LDUR, LDUR, ADD, SUB, STUR, CBZ. When you complete this lab, you should be able to get correct control unit outputs for these instructions as well as produce correct sign extended address values. This is not an exhaustive test, and you can modify it as you feel necessary to improve testing.

Listing 6.1: Updated Verilog starter code for the iDecode_test module.

```
`include "definitions.vh"

module iDecode_test;

wire clk;
reg reset;
reg [`WORD-1:0] nPC;
reg [`INSTR_LEN-1:0] IR;
```

```
reg [`WORD-1:0] Write_data;
wire [4:0] Rm_num;
wire [4:0] Rn_num;
wire [4:0] Rd_num;
wire [4:0] Read_register1;
wire [4:0] Read_register2;
wire [4:0] Write_register;
wire [`WORD-1:0] Read_data1;
wire [`WORD-1:0] Read_data2;
wire [10:0] Opcode;
wire [8:0] Address;
wire [`WORD-1:0] SignExtendedOutput;
wire [`WORD-1:0] nPCout;
wire Reg2Loc;
wire Branch;
wire MemRead;
wire MemtoReg;
wire [1:0] ALUOp;
wire MemWrite;
wire ALUSrc;
wire RegWrite;


iDecode Decoder(
        .clk(clk),
        .reset(reset),
        .nPC(nPC),
        .Instruction(IR),
        .Write_data(Write_data),
        .nPCout(nPCout),
        .SignExtendedOutput(SignExtendedOutput),
        .Reg2Loc(Reg2Loc),
        .Branch(Branch),
        .MemRead(MemRead),
        .MemtoReg(MemtoReg),
        .ALUOp(ALUOp),
        .MemWrite(MemWrite),
        .ALUSrc(ALUSrc),
        .RegWrite(RegWrite),
        .Rm_num(Rm_num),
        .Rn_num(Rn_num),
        .Rd_num(Rd_num),
        .Read_register1(Read_register1),
        .Read_register2(Read_register2),
        .Write_register(Write_register),
        .Read_data1(Read_data1),
```

```
        .Read_data2(Read_data2),
        .Address(Address),
        .Opcode(Opcode)
        );

oscillator myOsc(clk);

initial
begin
reset=0;
nPC=4;
#(`CYCLE/2);
// X21 should be set to 16 in fibR.data
// X9 should be set to 33 in fibR.data
// X10 should be set to 12 in fibR.data
IR = `INSTR_LEN'b11111000010001000000001011001001;
nPC=nPC+4;
Write_data=33;
#(`CYCLE);

IR = `INSTR_LEN'b11111000010000100000001011001010;
nPC=nPC+4;
Write_data=12;
#(`CYCLE);

// Add X21 + X9 and put in X9
IR = `INSTR_LEN'b10001011000101010000000100101001;
nPC=nPC+4;
#(`CYCLE);

Write_data=49;
nPC=nPC+4;
// Subtract X9-X10 and put in X9
IR = `INSTR_LEN'b11001011000010010000000101001001;
#(`CYCLE);

IR = `INSTR_LEN'b11111000000001100000001011001001;
nPC=nPC+4;
#(`CYCLE);

IR = `INSTR_LEN'b10110100111111100000001011001001;
nPC=nPC+4;
#(`CYCLE);

end
endmodule
```

6.4 Your Assignment

You are to:

1. Create a control module and add it to the iDecode module.
2. Create a sign extender module and add it to the iDecode module.
3. Import the updated iDecode_test.v file and update it as you see fit.
4. Verify that your results match what you expect.
5. There will not be a submittal on Canvas today. However, please show me your results either today or at the beginning of lab time during the next class period. Next time, we will make sure that your iDecode stage is working properly, integrate it with the iFetch stage, verify that the first two stages work properly, then finally write up a lab report.

Lab 7

Finishing Decode

7.1 Integration and Verification

At this point, you have created all of the modules necessary for the decode module. Now you need to put them all together and verify that they work properly. You should be able to support all of the instructions listed below, and you should include at least one of each of these instructions in your unit test. When adding instructions to your unit test, consider testing items like negative numbers in branch addresses and any other relevant untested cases. Also, for R-type instructions in your unit test, please provide the numerical result via the Write_data line since we do not have an ALU to perform calculations.

The supported instruction set includes:

1. ADD
2. SUB
3. AND
4. ORR
5. LDUR
6. STUR
7. CBZ
8. B

Each instruction should complete during a single clock cycle and should produce expected results. To verify this, you need to create an expected results table (probably in Excel) that includes a column for every instruction and a row for every data item shown on your simulation output. You should order the rows identically to the order of your simulation outputs for easy verification. Then fill in the rows for each instruction with your expected result. Please use

decimal numbers unless binary seems more appropriate. If a particular data item is not relevant for a particular instruction (for instance, address on an R-type instruction), then just put an X in that cell.

Once your expected results table is complete, you should be able to analyze your simulation output and go vertically down the simulation output and compare it to your expected results table, and it should match. While certain values will be offset in time, a single instruction should fall within a single clock cycle.

7.2 Your Assignment

You are to:

1. Integrate all individual modules into the iDecode module.
2. Test the iDecode module with a variety of instructions, including at least one of each instruction listed above.
3. Create your expected results table for each instruction in your test.
4. Verify that your simulation results match your expected results.
5. Write a lab report for the entire iDecode stage. This should include information about each module within iDecode, the iDecode module itself, and the iDecode_test. The goal is to concisely but completely describe the iDecode module. The report should follow the LabN.tex format and should include the following additional items:
   (a) Simulation Result Images
   (b) Expected Results Table
   (c) All code used in the iDecode module, including the iDecode_test
   (d) The elaborated schematic that Vivado produces. Please make sure to expand the iDecode module so that we can see the contents of it.