ELC 3338 Project Book

Steve Potter

November 28, 2018

Contents

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.

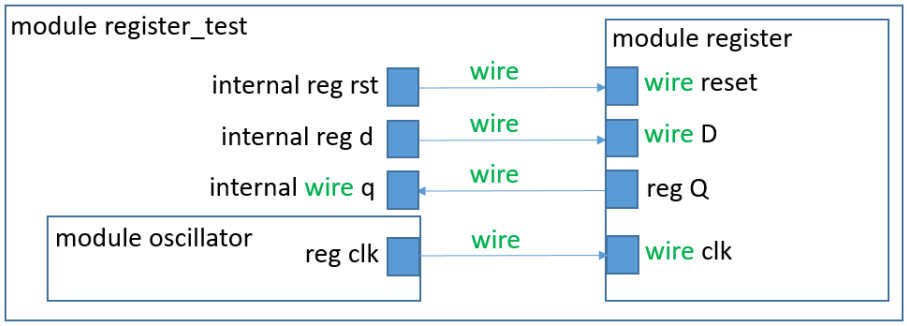
```
`include "definitions.vh"

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.

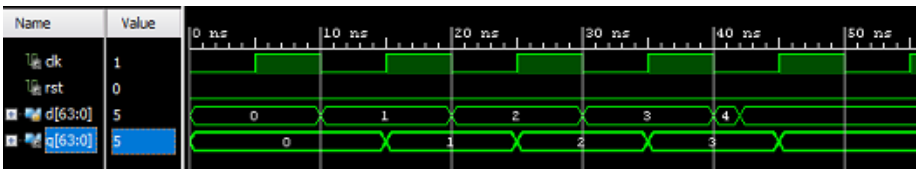
1.2 Testbench

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.

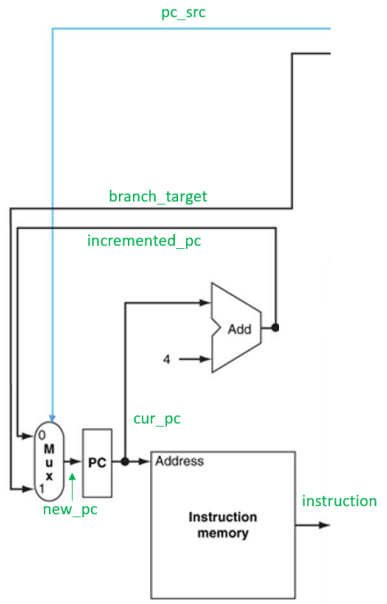
3.2 Fetch Stage

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'b11111000010011110000000101001001;
#(`CYCLE);

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

// STUR X9, [X10, #240]
instruction = `INSTR_LEN'b11111000000011110000000101001001;
```

```
#(`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

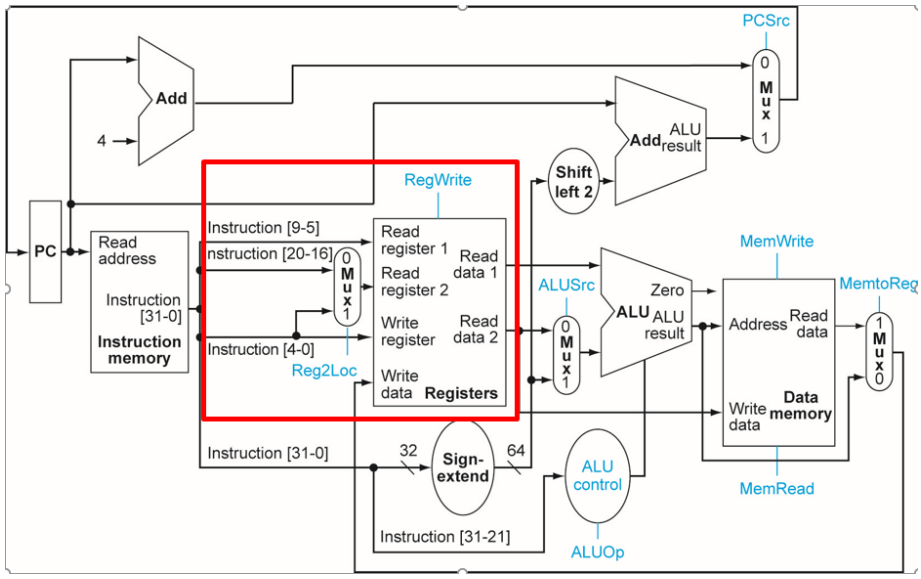
```
`include "definitions.vh"

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.

Lab 5

Control Unit and Sign Extender

5.1 Control Unit

Next, we will create the main control unit. You will create a new module called control (in control.v), 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. reg2_loc
2. uncondbranch
3. branch
4. mem_read
5. mem_to_reg
6. alu_op
7. mem_write
8. alu_src
9. reg_write

The supported instructions should include:

1. ADD
2. SUB
3. AND

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 4.4.6 in the textbook. Note that a value of X in a table entry indicates that it does not matter whether the value is 0 or 1. You do not want to use X in these cases, as this indicates undefined to Verilog. Instead, please use 0 in place of X.

5.2 Control Unit Test

The Control Unit is crucial to operation of your datapath, so it needs to be tested thoroughly. Every supported instruction (listed above) should be tested with the Control Unit Test. To facilitate testing, I have provided an Excel spreadsheet with 10 instructions listed across the top. I will refer to this table as the Expected Results Table. These are the instructions that you should use to test your Control Unit and your full Decode stage (once we finish it). The rows of the table are signals that will be inputs and outputs of the control module.

Before writing your unit test, you should fill in every cell in this table with the expected results. This includes producing the machine code for each instruction. Then, use these machine code instructions as inputs to your Control Unit Test and verify the outputs. You should display the outputs on your simulation results in the same order that they are shown on the Expected Results Table. Then, you can go right down the table and easily verify that your results are correct.

Save the Expected Results Table on GitHub, because we will be adding to it in future labs. Note that for this lab, you do not need to have time values at the top of your expected results tables. The instructions give us sufficient granularity.

Note that, at this point, the values that are loaded in the registers do not matter, as we are just testing the Control Unit. Once we get to the full iDecode stage, we will need to address the register values.

The Verilog implementation between 'if/else' statements and 'case' statements differ. 'If/else' statements will create a series of nested muxes with two inputs each, whereas a case statement will produce one large mux with many inputs. To maximize speed, we should use case statements. But one of the challenges of this lab is dealing with opcodes of different sizes. Thankfully, Verilog has 'casex', which will only evaluate binary digits that are not labeled X. So for a CBZ instruction, you can fill in the last 3 digits with XXX and use 'casex'. To help you get started, I've included starter code for control.v in Listing 5.1.

The 'casex' syntax will also be very helpful in the Sign Extender. Also, please create macros in definitions.vh for the opcodes of each instruction and for the ALU Op values of each instruction type.

Listing 5.1: Control Module

```
`include "definitions.vh"

module control(
    input [10:0] opcode_bits,
    output reg reg2_loc,
    output reg uncondbranch,
    output reg branch,
    output reg mem_read,
    output reg mem_to_reg,
    output reg [1:0] alu_op,
    output reg mem_write,
    output reg alu_src,
    output reg reg_write
    );

    always @(*) begin
        casex (opcode_bits)
        // ADD, SUB, AND, ORR, LDUR, STUR, CBZ, and B should be
        // defined in definitions.vh.  CBZ and B instructions should
        // use X to fill in the extra bits in the 11 bit opcode
        // casex treats these bits as 'don't cares'
        `ADD: begin
                reg2_loc <=0;
                uncondbranch<=0;
                branch<=0;
                mem_read<=0;
                mem_to_reg<=0;
                // ALUOp_RTYPE, ALUOp_DTYPE, and ALUOp_BRANCH should
                // be defined in definitions.vh
                alu_op<=`ALUOp_RTYPE;
                mem_write<=0;
                alu_src<=0;
                reg_write<=1;
            end
        default: begin
                  reg2_loc<=0;
                  uncondbranch<=0;
                  branch<=0;
                  mem_read<=0;
                  mem_to_reg<=0;
                  alu_op<=`ALUOp_RTYPE;
```

```
                    mem_write<=0;
                    alu_src<=0;
                    reg_write<=0;
                end
        endcase
    end
endmodule
```

5.3 Sign Extender

The final major component of the Decode stage is the Sign Extender. The Sign Extender should use information in the instruction to create a 64-bit output value to use as an address or branch offset. The sign extender should support extending address values from the following instructions:

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

5.4 Sign Extender Test

The Sign Extender Test should utilize the instructions from the Control Unit test. Verify that your Sign Extended Output on the Simulation Results matches the value on your Expected Results table. Instructions that don't use a Sign Extended Output (like R-Type) should fall into the default case. Fill in the Expected Results Table with the default value. Note that for this lab, you do not need to have time values at the top of your expected results tables. The instructions give us sufficient granularity.

5.5 Your Assignment

You are to:

1. Fill in the entire Expected Results Table, starting with the version that I provide in the testfiles section in GitHub.
2. Create a Control Unit module.
3. Create a Control Unit Test and verify that the simulation results match your Expected Results Table.
4. Create a Sign Extender module.
5. Create a Sign Extender Test Module and verify that the simulation results match your Expected Results Table.
6. Write a lab report according to the LabN format.

Lab 6

Finishing Decode

6.1 iDecode Module

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

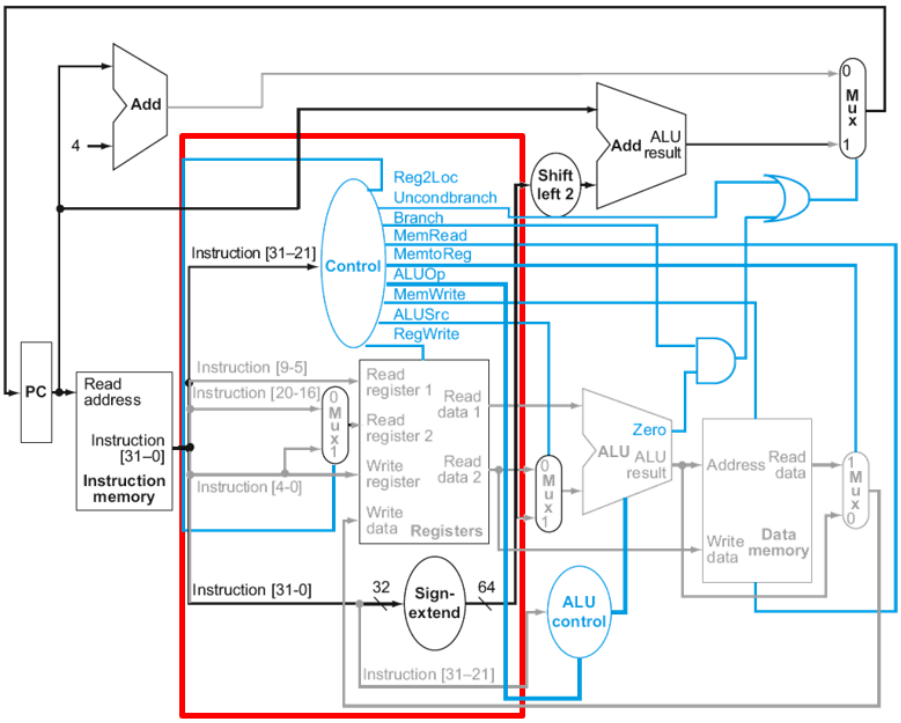
6.2 iDecode Unit Test

To verify that your iDecode module works correctly, you must first update your Expected Results Table. You should first add a row for every input and output of your iDecode module, then fill in all cells with expected values. If a particular data item is not relevant for a particular instruction (for instance, sign_extended_output on an R-type instruction), then just put an X in that cell. To ensure that we are testing each case, please update regData.data to reflect the following values:

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

Now create a unit test for iDecode by providing the inputs for iDecode and verifying the outputs of iDecode. For the instruction input, use the instructions from your Expected Results Table. For arithmetic instructions, use the test

Figure 6.1: Expected Results



bench to provide the correct value to the write_data input, since we do not yet have an ALU to do the calculations. Also, please use your test bench to provide a value of 20 to X9 in the first command (LDUR).

With your Expected Results Table in hand, you should be able to run the iDecode unit test and analyze your simulation outputs by going vertically down the simulation output, comparing the table to your simulation output. While certain values will be offset in time, a single instruction should fall within a single clock cycle.

6.3 Integrating iFetch and iDecode

Now that you have iDecode created and tested, you can test it with your iFetch module. To do this, you need to create a file called datapath.v. This will be your top-level file for your integrated datapath, analagous to the test bench files. In this file, you should have an instance of iFetch and iDecode. You should connect these two modules with wires by analyzing the full datapath diagram. Since we do not have a full datapath yet, we need to have some "test bench" aspects to datapath.v. Similarly to your iFetch test, you will want to create a reg for reset, pc_src, and branch_target. Since we do not have all aspects of branching implemented, please keep pc_src set to 0 for the duration of the test.

Please update your instrData.data file to only contain the 10 instructions in your Expected Results Table. The goal of datapath.v is to verify that the PC increments, each of these instructions is fetched at the appropriate time, and that the instruction executes properly. You can verify the execution by comparing your results with your expected results table.

6.4 Your Assignment

You are to:

1. Integrate all individual modules into the iDecode module.
2. Update your Expected Results Table to include all iDecode inputs and outputs.
3. Test the iDecode module with the instructions from the Expected Results Table.
4. Verify that your simulation results match your expected results.
5. Create datapath.v and integrate the iFetch and iDecode stages.
6. Verify that the results match the Expected Results Table.
7. Write a lab report according to the LabN format. The focus of the report is the iDecode module, including testing it with iDecode_test and integrating it with iFetch. It does not need to describe the details of the

submodules...you have already written a report on those. Please consider datapath.v to be a testbench when writing the report.

Lab 7

ALU and ALU Control

7.1 ALU

First we will build the ALU itself. The ALU has three inputs (two data inputs to act on, and a control input to determine the action perfomed) and two outputs (one data, and a logical flag). In the table shown in the lecture slides, you can see the meaning of the control bits used to determine what the ALU will calculate. You should use a case statement for the control bits to determine which ALU operation to perform. For each operation, you do not have to do anything fancy. You just need to use the math capability that verilog provides to make the calculation. You should give the the ALU control bits names in the definitions.vh file, and you should use these in your cases. Also don't forget to make a default case, which is needed to actually wire this up. Pick something fast for the default, thus usually a logic statement.

One last thing to note is the generation of the zero flag. There are several ways to handle this, but the easiest way to handle it is below:

1. In Verilog (like C), the statement $(y == 0)$ is an operation with a boolean output. You can thus say $x = (y == 0)$; to assign x to be the boolean value. The statement $x = (y == 0)$; is realizable as a digital comparator with y and 0 as inputs and x as the single bit output.

7.2 ALU Control

Now we need to build the controller to use the ALUOp field and the opcode field to generate the ALU control bits used above. Consider the table for ALU Op to ALU Control bits in the lecture slides. The ALU Control module is where you make this translation so that the ALU is told the proper operation to execute. Use case statements and include a default to handle undefined signals (use fast commands for undefined signals). Define the ALU operations in definitions.vh file to improve readability - you shouldn't need any numbers. This should be a

simple module with two inputs (ALUOp and function) and one output (control bits).

7.3 Your Assignment

You are to:

1. Create the ALU module.
2. Create an ALU testbench to test all relevant ALU operations, including testing the zero bit.
3. Create an ALU Control module.
4. Create an ALU Control testbench to test all of the opcode/ALUOp pairs that you will be using
5. You do not need to create Expected Results tables for these tests, as they are very short and simple.
6. Create a lab report in the LabN format.

Lab 8

Execute Stage

8.1 Execute

In the last lab, you created the ALU and ALU Control modules. Now we will finish the iExecute stage. The iExecute stage is represented by the red box in Figure 8.1. To finish the iExecute stage, you will need to add the following:

1. Mux to select the source of the second input into the ALU. You can reuse your mux that you created in the iFetch stage.
2. Shifter to left shift the sign extended branch address offset. You will need to create a new module for this.
3. Adder to add the branch address offset to the current PC. You can reuse your adder that you created in the iFetch stage.

These five modules should be included in a new module called iExecute. iExecute should consist of everything shown in the red box on Figure 8.1.

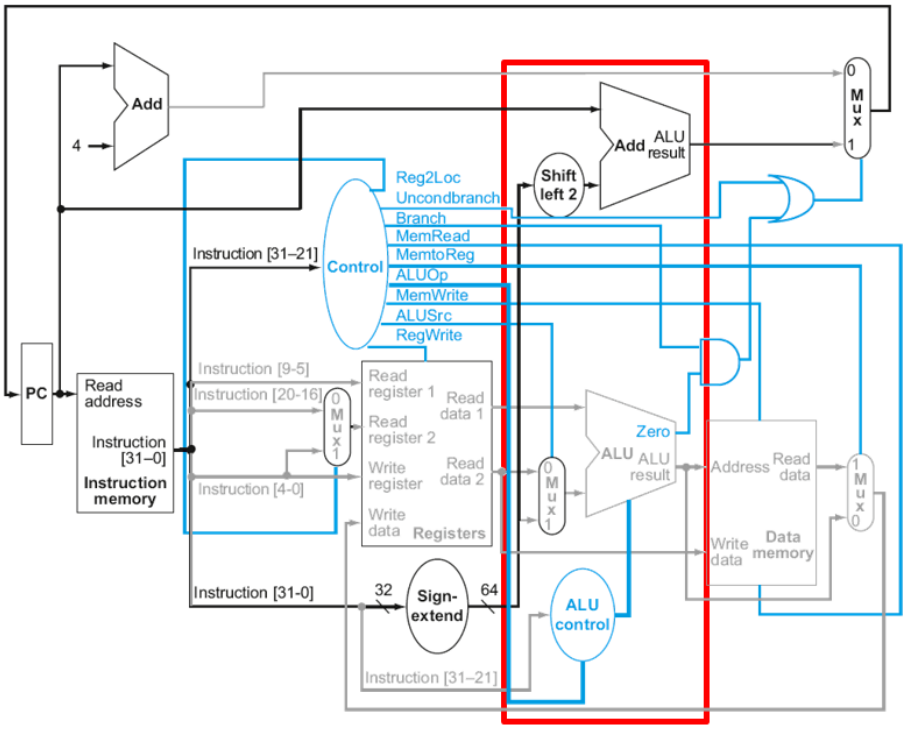
8.2 Your Assignment

I have provided a test bench, iExecute_test.v. Feel free to modify and improve it as you see fit.

Listing 8.1: iExecute Test Bench

```
`include "definitions.vh"

module iExecute_test;

reg [`WORD-1:0] pc_in;
reg [`WORD-1:0] read_data1;
reg [`WORD-1:0] read_data2;
```

Figure 8.1: Execute Stage



```
reg[`WORD-1:0] sign_extended_output;
wire[`WORD-1:0] branch_target;
wire [`WORD-1:0] alu_result;
wire zero;
reg [1:0] alu_op;
reg [10:0] opcode;
reg alu_src;

oscillator myOsc(clk);

iExecute myExecute(
    .pc_in(pc_in),
    .sign_extended_output_in(sign_extended_output),
    .alu_op_in(alu_op),
    .alu_src_in(alu_src),
    .opcode_in(opcode),
    .read_data1_in(read_data1),
    .read_data2_in(read_data2),
    .branch_target(branch_target),
    .alu_result(alu_result),
    .zero(zero)
    );

initial
begin
    pc_in<='WORD'd0;
    alu_src<=1'b0;
    sign_extended_output<='WORD'd520;
    read_data1<='WORD'd15;
    read_data2<='WORD'd10;
    opcode<=`ADD;
    alu_op<=2'b10;

    #`CYCLE;
    pc_in<=pc_in + 4;
    opcode<=`SUB;

    #`CYCLE;
    pc_in<=pc_in + 4;
    opcode<=`AND;

    #`CYCLE;
    pc_in<=pc_in + 4;
    opcode<=`ORR;

    #`CYCLE;
```

```
pc_in<=pc_in + 4;
alu_op<='ALUOp_DTYPE;
opcode<='LDUR;

#'CYCLE;
pc_in<=pc_in + 4;
alu_op<='ALUOp_DTYPE;
opcode<='STUR;

#'CYCLE;
pc_in<=pc_in + 4;
alu_op<='ALUOp_BRANCH;
opcode<='CBZ;

#'CYCLE;
pc_in<=pc_in + 4;
opcode<='B;

#'CYCLE;
pc_in<=pc_in + 4;
read_data1<='WORD'd15;
read_data2<='WORD'd15;
opcode<='ADD;
alu_op<='ALUOp_RTYPE;

#'CYCLE;
pc_in<=pc_in + 4;
opcode<='SUB;

#'CYCLE;
pc_in<=pc_in + 4;
opcode<='ADD;

#'CYCLE;
pc_in<=pc_in + 4;
read_data1<='WORD'd15;
read_data2<='WORD'd0;
opcode<='CBZ;
alu_op<='ALUOp_BRANCH;

#'CYCLE;
pc_in<=pc_in + 4;
read_data1<='WORD'd15;
read_data2<='WORD'd15;
opcode<='ADD;
alu_op<='ALUOp_RTYPE;
```

```
    #‘CYCLE;
    pc_in<=pc_in + 4;
    alu_src <=1’b1;
    alu_op<=‘ALUOp_DTYPE;
    opcode<=‘LDUR;

end
endmodule
```

You are to:

1. Complete the iExecute module
2. Use the iExecute_test to create an Expected Results Table. In the table, just have a column for each clock cycle, since each instruction is executed in one clock cycle.
3. Verify that your simulation results match your expected results.
4. Create a lab report in the LabN format.

Lab 9

Integrating Fetch and Decode

9.1 Integration

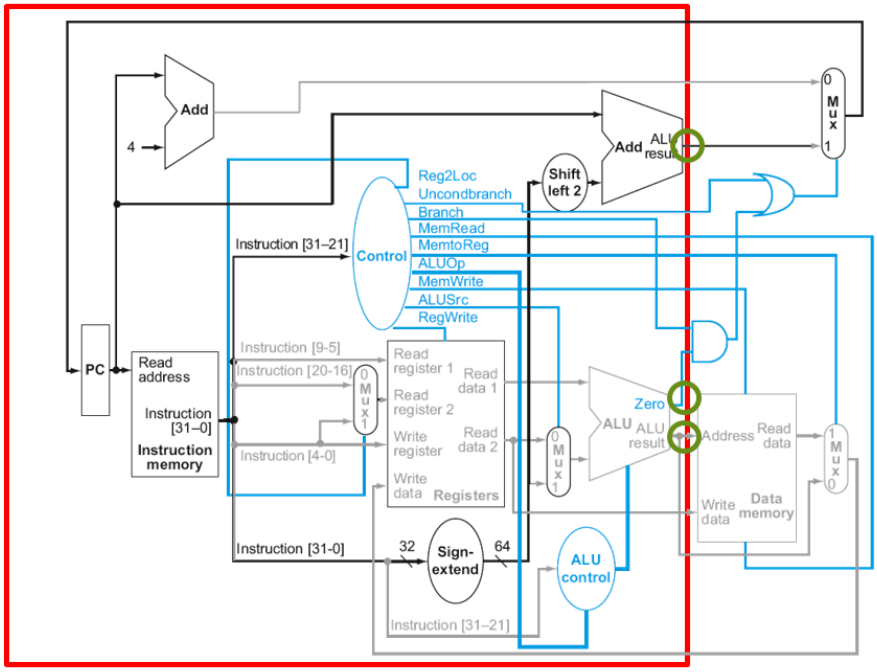
We now have working Fetch, Decode, and Execute modules. Now it is time to put them together to produce a system that can:

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

Once we can do all of this, we will be ready for the iMemory stage. We currently have the fetch and decode module integrated into datapath.v. We also have a working execute module. Today we need to integrate the Execute module into datapath.v. Please reuse the instructions from your Expected Results Table that you used when you integrated fetch and decode. Please make sure that datapath.v is the "top module" in your project. Once integrated, you should be able to produce a simulation that includes 3 new outputs:

1. Branch Target
2. ALU Result

Figure 9.1: Execute Stage



3. Zero

These three new outputs are marked on Figure Figure 9.1. To verify these outputs, you should update the Expected Results Table to include these three outputs. Then you should verify this table against your simulation results.

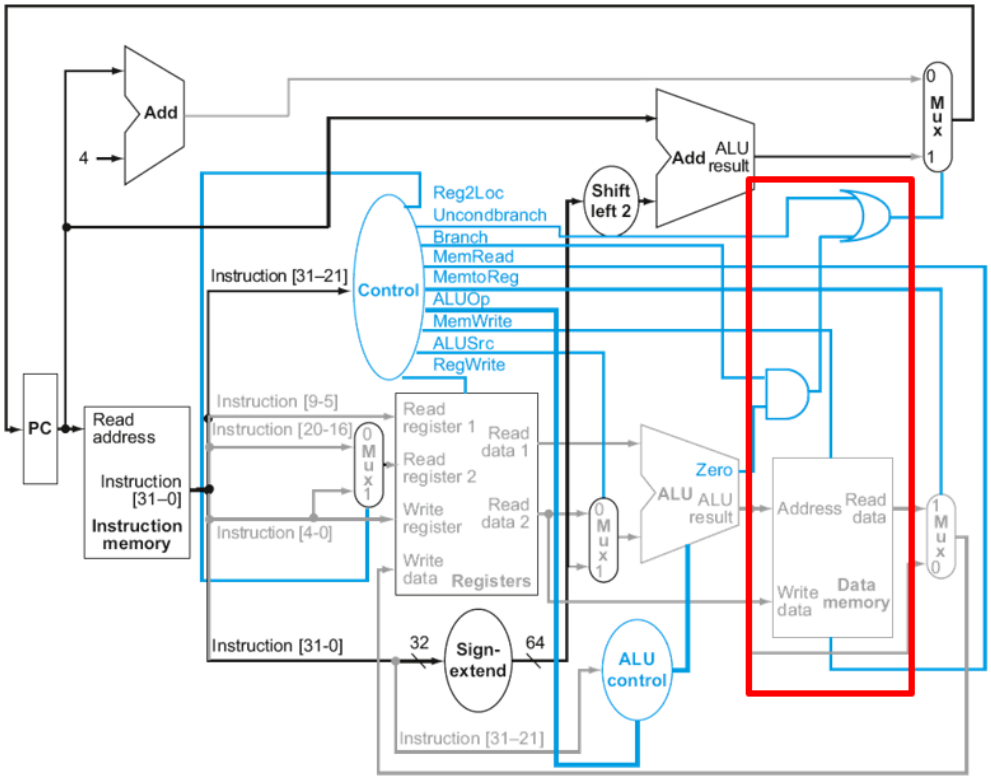
9.2 Your Assignment

You are to:

1. Integrate your iExecute module into your datapath.v that includes iFetch and iExecute.
2. Update your Expected Results Table to include the new outputs.
3. Verify that your simulation results match your expected results.
4. Create a lab report in the LabN format that focuses on the integration of the iExecute stage. The only inputs that you need to list are the inputs that you set in the initial section of datapath.v. The only outputs are the signals that are leaving the red box in Figure 9.1.

Lab 10

Memory



10.1 Memory Stage

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

10.2 Branch Resolution

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

We also need an 'or' gate to 'or' together the output of the branch 'and' gate (above) and the uncondBranch control signal. These gates can be included in your iMemory module as one line commands. They do not need to be explicity tested, as they will be thoroughly tested when we integrate the system.

10.3 Data Memory

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

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

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

Create a data_mem.v file, copy the contents of your register file memory and modify it to meet the needs of the data memory module.

10.4 Your Assignment

You are to:

1. Create a new module called iMemory.

2. Instantiate the AND and OR gates directly in the iMemory stage.

3. Create a new module called data_mem to handle data memory access. Also create or modify a ramData.data file that contains initial values to be read into memory.

4. Integrate data_mem into the iMemory stage.

5. Write a testbench to verify that the iMemory stage works properly. Use a combination of ramData.data and testbench to test various scenarios. Make sure to use STUR instructions to set values in memory locations, then read those values back using the LDUR function.

6. Create a lab report in the LabN format.

Lab 11

Write Back

11.1 Mux

This stage consists on only one item, a mux to select between the output of memory and the output of the ALU. The control is the MemtoReg control line, see Fig 11.1. Since the mux has already been tested it does not need a testbench. The stage thus has only 3 inputs (2 data and 1 control) and one output, the result.

11.2 Datapath

You are ready to assemble the full non-pipelined datapath shown in Fig 11.3. To do this, you will need to combine all 5 stages into datapath.v. Stages include:

1. iFetch
2. iDecode
3. iExecute
4. iMemory
5. iWriteBack

Verify by running your set of instructions in instrData.data and testing the output. Pay particular attention to make sure that the Rd register is updated appropriately by R-Type and D-Type instructions. You should no longer set write_data in datapath.v. Rather, you should connect write_data from the WriteBack stage to the Decode stage. For right now, keep pc_src hard-coded to 0 in datapath.v. Even though we could connect it now, our test instructions are not meant to run like a program and would yield strange results. Each instruction should execute as expected according to your Expected Results Table.

Figure 11.1: Write Back

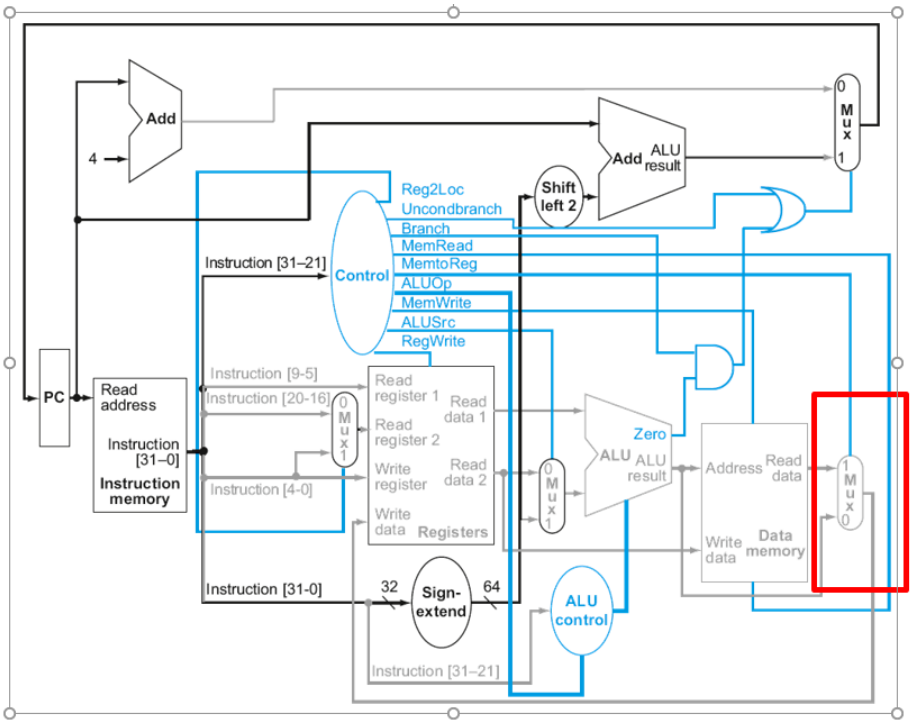
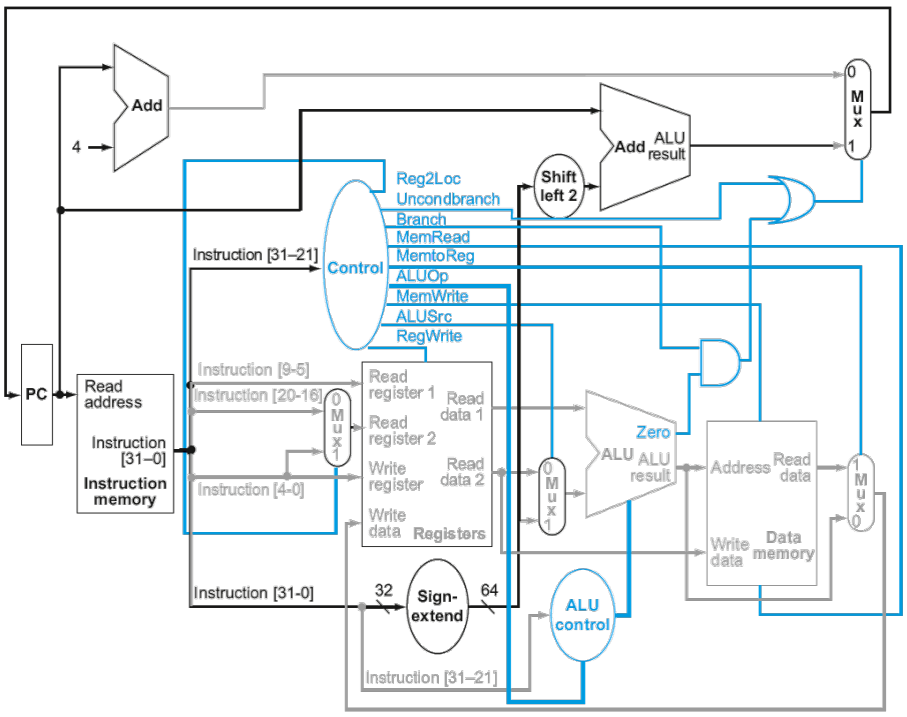


Figure 11.2: Full Non-Pipelined Datapath



Datapath.v should be your top-level Verilog file (no additional test drivers are necessary).

11.3 Your Assignment Part 1

You are to:

1. Create the Writeback stage consisting of one Mux.
2. Integrate all five stages into the file datapath.v.
3. Update your Expected Results Table to include the iWriteBack stage.
4. Run simulations to verify that your results match your Expected Results table.
5. Do not write up a lab report yet. There will be one final test to add before we submit it.

11.4 Division

To further verify your datapath operation, you should create a new set of datafiles to implement the division code shown below. You should first write assembly code, then translate it into binary. One restriction is that the only non-zero value in your regData.data should be X22, which can be used as the base address for the array A. Otherwise, all other data must be loaded from memory via the ramData.data and LDUR commands. Please pay attention to the comments in division.c.

Listing 11.1: C code for doing simple division.

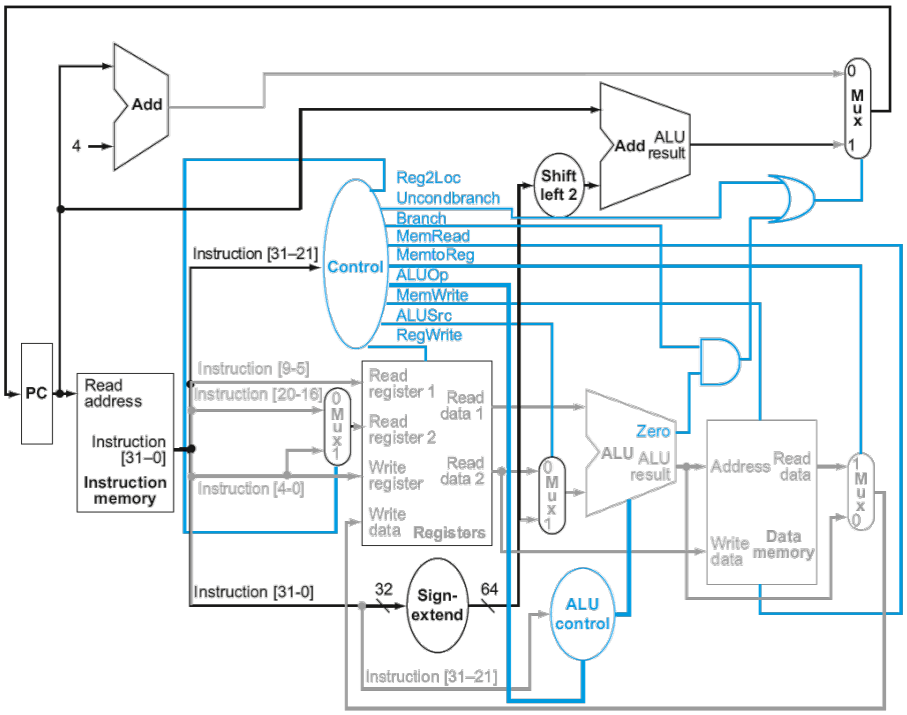
```
// Rules:
// The register file must be set to all 0s except that you
// can store the base address of A in X22.  Base address of A
// should be non-zero.  Everything else must be loaded from memory.
// This is a simple division problem where having a remainder will
// break it, so choose an x that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


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

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

A[2] = z;
```

11.5 Your Assignment Part 2

You are to:

1. Implement the assembly and binary code for the Division C Code shown above.
2. Verify that the divison works correctly.
3. Write up a lab report according to the LabN format.

Figure 11.3: Full Non-Pipelined Datapath



Lab 12

Pipeline Fetch and Decode

12.1 Pipeline Fetch and Decode

Now that we have a working single-cycle datapath, we are going to break it apart and start pipelining it. Make sure to keep a copy of your working single cycle datapath for reference. To begin the pipelining process, we will first pipeline the iFetch and iDecode stages.

The advantage of pipelining includes the ability to reduce the clock cycle time by doing only one stage at a time. We have been using a 10ns clock cycle. Now that we are pipelining, let's start conservative and go down to a 4ns clock cycle. Please keep it at 4ns for now to maintain consistency across the class.

This will affect your timing and will require you to update the clocks in your datapath. The basic clk for each stage should be clk (no delays). You will still need delayed clocks for items that are delayed within a stage such as read_clk and write_clk. The first goal for your pipeline is to get a series of instructions to execute in pipeline form, where a new instruction is fetched while the previous instruction is being decoded. You should verify this by examining the Read_Register values, Read_Data values, control signal values, etc. To test this, use your set of instructions that corresponds to your Expected Results table. For right now, just focus on the Fetch and Decode stages.

12.2 Pipeline Buffering and Forwarding

The next step in developing your pipeline is to add the stage buffers that we have seen in the textbook. The idea here is that every input to a stage must be buffered so that you do not operate on values that are currently being changed. The code for this is not complex, but the planning and thought process can be complex because some signals need to be passed through to future stages. Remember, your signals should only go from one stage to the next stage. For example, you should not have signals that go from Decode to Memory. The signal must go from Decode to Execute, then Execute to Memory. If you skip

a stage, the data will be incorrect due to timing issues. To aid your efforts, I require that you complete the PipelineAnalysisTemplate.xlsx. This allows you to figure out what signals need to be passed between stages and develops a naming convention that will help you when you start implementing.

In the spreadsheet, you should start with my template, which has many (not all) of the signals that are in your datapath. Look at your current connections in datapath.v. Wherever you see an output from a module, put that in the spreadsheet as a source. Wherever you see an input to a module, put that in the spreadsheet as a destination. In many cases, you will see that there is space in the spreadsheet between the source and destination. This means that the signal must be buffered and forwarded from one stage to the next. Fill in the space between the source and destination with signals to be forwarded. Please use the naming convention that I show on the spreadsheet....this might be the most crucial part...even if you don't understand why, just trust me on this one. The signal names should match your current signal names, but they should have a suffix added to them that indicates which stage outputted the data. For instance, when mem_to_reg was buffered in iExecute and forwarded to iMemory, the signal should be name mem_to_reg_ie because iExecute outputs the data.

12.3 Your Assignment

You are to:

1. Save a copy of your non-pipelined datapath.v.
2. Comment out Execute, Memory, and WriteBack stages from datapath.v.
3. Pipeline iFetch and iDecode stages as described above.
4. Test with your set of instructions from your Expected Results table.
5. Verify that stages are processing the correct instructions at the correct times.
6. Complete PipelineAnalysisTemplate.xlsx
7. Create a lab report with LabN format.

Lab 13

Pipelining without Branching or Forwarding

13.1 Overview

Now that we have pipelined the fetch and decode stages, we can add the register buffers between each stage and get a simple pipeline working. This pipeline will not include any data forwarding or branch prediction. We will handle data hazards by using assembly code with appropriately placed nop commands. For now, we will avoid control hazards by keeping pc_src set to 0, which will keep the system from branching (even when you run a branch instruction).

Only after you have completed the spreadsheet, then you can move on to writing Verilog code. You will want to update datapath.v to use the names from the spreadsheet. This will include adding a lot of new signals. Then you will want to update your modules to account for these new signals. Use your spreadsheet to make the following updates:

1. Add input ports to stage modules as needed. This will be particularly relevant for signals that are passed through. You should not use the _id, _ie, etc naming convention within the module. You should either use no suffix or the suffix _in or _out.

2. Update all modules to buffer all inputs into the module. This will keep inputs from changing while they are being used. On the positive edge of the clock, you should use procedural assignments to copy all inputs into another register (reg). This register can either be an output reg (when appropriate) or a local reg (if the signal does not need to be output). Then do all of the module's processing on the buffered regs. Try to change as few names as possible within the modules. Use names that keep you from having to update your signal names to the lower level modules. In modules that do not currently have a clock, you will need to add the clock.

3. Use the information in the spreadsheet to add output ports to stage modules as needed.

Again, try to change as few names as possible within the modules. You want to use the stage-specific names (_ie, etc) in datapath.v rather than in the modules, to the extent possible.

Use the instructions from your Expected Results Table to test your pipeline. Insert NOP instructions where appropriate in your instrData.data. To insert a NOP instruction, add a 32-bit line with all zeros.

13.2 Your Assignment

You are to:

1. Implement the spreadsheet in your datapath.v
2. Update your modules to buffer the appropriate values into registers
3. Use your expected results table instructions to test the pipeline and correct any issues
4. Once your pipeline is working, go back and try to reduce the cycle time as much as possible (without having to re-architect your solution).
5. Submit a lab report using the LabN format.

Lab 14

Pipelining with Branching

14.1 Overview

Once your pipeline is working with the simple set of commands that I provided, it is time to add branching.

We will put the following restrictions on our efforts to branch:

1. Use Branch Not Taken method of prediction
2. You should move the branch decision hardware to the Decode stage as mentioned in the lecture.
3. When a branch is taken, you must zero the control lines and set PCWrite and IF/IDWrite to 0. You will need to add PCWrite and IF/IDWrite. Branch hazards must be detected by a new module, the Hazard Detection Unit.
4. Instructions used for this should be the division problem instructions. You need to insert nops (all zeros) in the instruction file where necessary. The only reason to add nops in the instruction file is a data hazard

14.2 Your Assignment

You are to:

1. Implement Branch Not Taken Prediction
2. Create an instruction file as described above, based on the division problem instruction file
3. Update your modules to detect and respond to branch hazards
4. Use the instruction to test the pipeline and correct any issues

5. Submit picture(s) of your simulation results. Also submit a zip file of your repository.