ELC 3338 Project Book

Keith Evan Schubert

March 14, 2016

Contents

Lab 1

Introduction

In the Labs for this course we will be building a 32-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 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, which is the size of data our computer will use (how many bits). We will be using 32-bits, but note that if we build things based around WORD, rather than the number 32, we can just change the value of WORD in the file and get a computer with a different size (say 64) 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 31 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 reset. Reset has higher priority, so if reset is asserted the register is cleared (Q

[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. :)

is set to zero), otherwise the value of D is stored it Q. That is it. A nice, simple module.

1.2 Testbench

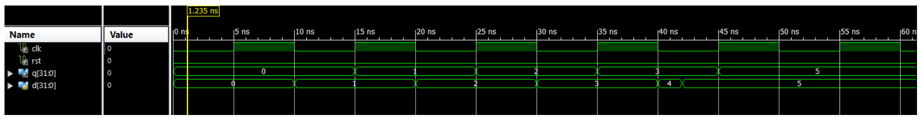
We now want to test this. To test it, we need to tell the simulator to build a copy (instantiate) the module, and then we will need to supply the inputs and look at the outputs. Consider the testbench in Listing 1.2.

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 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.

Figure 1.1: Timing diagram.



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

Alls 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 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.

[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.

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 (add one word offset) each cycle.
2. If the conditions of a conditional branch are met, the destination of the branch is the next instruction.
3. If an uncondtitional branch or jump occurs, the destination is the branch's destination.
4. If an interrupt or error occurs, then the next value of the program counter should be the appropriate handler.

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, then we just have to pass it a 1. Later in our computer we will need another adder, so this will let us re-use the code. We pass it a 1 because our memory will be word addressable. Word addressable means the smallest unit of memory that has its own address is a word. Most machines are byte addressable, because one ASCII character (a char in c/c++) is a byte. For a 32 bit word 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, see listing 2.1. We start by pulling in our definitions so we have our word size. We then declare our ports, which in this case I made all wires as I want to show you another way to define a unit, beside an always thread. 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. The actual definition is then one line. Assign means to take a command and make it permanently define the value of a wire. It can only be to a wire (not a reg) as it must continuously assign a value, and regs cannot be continuously driven. We then just give the simple code that the output is the sum of the two inputs, just like in C or an always thread. That is it.

Listing 2.1: Verilog code to make an adder.

```
`include "definitions.vh"

module adder(
    input [`WORD-1:0] Ain,
    input [`WORD-1:0] Bin,
    output [`WORD-1:0] add_out
    );
    assign add_out = Ain+Bin;
endmodule
```

In this lab you will make your own testbench. The easiest way to do this is to use ISE's built in testbench generator.

1. Add a new source. There are several ways to do this. You can right click on the FPGA in the source window (upper left window) and select add a new source, or go to the project menu and select add a new source.
2. When the dialog comes up, select Verilog testbench, give it a name and verify the path is where you want it.
3. In the next dialog window it will ask you which module to test, and select adder (assuming you have already included the adder, if not do so and repeat).
4. Finish.

It will make a test bench for you, create reg's for the inputs to the UUT and wires for the outputs. It also instantiates the module as UUT, and creates an initial thread to do the stimulus. You can now add whatever you feel is a good test for the adder. Do some easy to verify checks then some that will test carry and rolling over. What else do you think you should you check?

2.2 Input Selection via Mux

We will 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 (32 bits), but later we will also need to switch register addresses (5 bits). Rather than write two registers, we will make one and then use the parameter to change the size when they are declared.

Listing 2.2: 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 like you did for the adder. Note the parameter is not used, and it set the inputs and outputs to be the default of 8. We are going to change this to test it as a 5 bit mux. Find the line that starts `mux UUT(...` and change it to be `mux#(5) 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. Make sure you change the size of the regs and the wire that are sent to the UUT from 7:0 to 4:0. Now come up with good values to test your mux so you are confident it works. Ask yourself, what could go wrong, and how can I test it?

2.3 Your Assignment

You are to:

1. Write a testbench for the adder in Listing 2.1.
2. Write a testbench for the mux in Listing 2.2.
3. Run a simulation and generate a timing diagram for each.
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 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. We will need an array of 32-bit numbers, which means the data type must be `reg[‘WORD-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 [`WORD - 1:0] instruction=`WORD'b0
    );

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

    always @(posedge(clk))
        instruction<= imem[pc];

    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 test fixture 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.

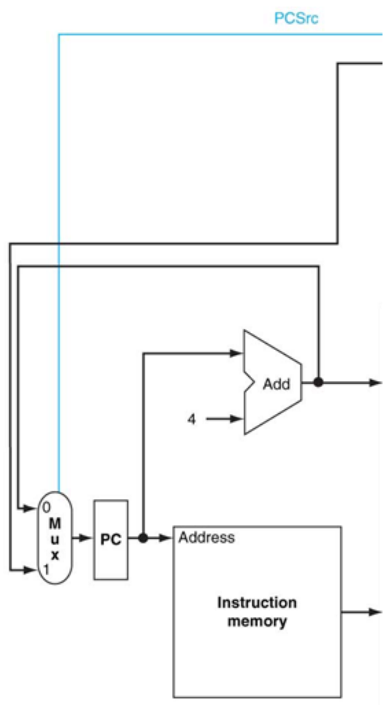
Any wire (or reg) in the figure that comes in or goes out are ports. In Figure 3.1, the blue wire is a control signal and comes ultimately from the control unit, which you will build in the next stage called decode. Wires (or regs) that are completely contained in the figure are local and are thus defined 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, and also, since I took this from the books images, the input to the adder is 4 not 1. We are staying with a word addressable machine, not a byte addressable machine, so stick with 1. 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 be `nPC_fetch-decode`. Note you get directionality also. Sometimes this helps, sometimes it is cumbersome, it mainly helps in debugging

[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.

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!

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.

Figure 3.1: Instruction Fetch Stage.



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=32'd1, SIZE=1024)(
    input clk,
    input reset,
    input PCSrc,
    input [`WORD-1:0] BrDest,
    output [`WORD-1:0] nPC,
    output [`WORD-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

Introduction

In our last lab, you built the first stage of our pipeline, Fetch. When you did you might have noticed that something odd happened. You might have noticed that the output of your instruction memory lagged by one cycle from the program counter. This is because they both are timed on the same edge of the clock. To handle this we we are going to need to put in a delay. We also want to buffer our outputs to ensure they stay constant through the entire cycle, which will allow us to calculate the new values, while allowing the next stage to operate off a finalized value. Consider the code in Listing 4.1. It illustrates the idea of a delay in updating.

Listing 4.1: Verilog code to build a buffer.

```
`include "definitions.vh"

module buffer_ifid#(parameter DELAY=0)(
    input clk,
    input reset,
    input [`WORD-1:0] nPC_if,
    input [`WORD-1:0] IR_if,
    output reg[`WORD-1:0] nPC_id=`ZERO,
    output reg[`WORD-1:0] IR_id=`ZERO
    );

    always @(negedge(clk), posedge(reset))
    begin
      #DELAY;
      if (reset)
      begin
        nPC_id<=`ZERO;
        IR_id<=`ZERO;
      end
      else
```

```
            begin
                nPC_id<=nPC_if;
                IR_id<=IR_if;
            end
        end
endmodule
```

4.1 Your Assignment

You are to:

1. Write a testbench for the buffer, run a simulation and generate a timing diagram.
2. Integrate the buffer into iFetch, add the delay to your instruction memory, and re-run your simulation to verify it works.
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 5

Starting to Decode

In our last lab, you finished the first stage of our pipeline, Fetch, by adding a buffer. Now we are going to move on to the first two components of the decode stage.

5.1 Sign Extender

Consider the code in Listing 5.1. It really has only one new line, which shows two uses of curly braces: concatenation and replication. Concatenation is done by `{a,b}`, which would concatenate the bits of a with the bits of b to form a longer sequence. Replication is done by `n{a}`, which copies a n times. If you look at our code it takes a number that is half a word long, and makes it a full word by copying the most significant bit. The replication is made generic by using the constant WORD, and calculating the bit locations. Unfortunately, the bit location (index) and number of times to copy it (n) have yet to be entered.

Listing 5.1: Verilog code to build a sign extender.

```
`include "definitions.vh"

module sign_extender(
    input [(`WORD/2)-1:0] in,
    output [`WORD-1:0] out
    );
    assign out = {{(n){in[(index)]}} , in };
endmodule
```

5.2 Control Unit

The control unit is the brains of our operation. It will have to take the first six bits of every instruction (the opcode), and set the control bits.

Figure 5.1: Control signal meanings and values.

Signal name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

	Execution/address calculation stage control lines				Memory access stage control lines			Write-back stage control lines	
Instruction	RegDst	ALUOp1	ALUOp0	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	1	1	0	0	0	0	0	1	0
`lw`	0	0	0	1	0	1	0	1	1
`sw`	X	0	0	1	0	0	1	0	X
`beq`	X	0	1	0	1	0	0	0	X

First lets discuss what signals we are setting. Almost all the boolean signals are described in Figure 5.1. The tricky one in the table is PCSrc, which can't be calculated yet, since we need to know whether the condition of the branch was met. All we know at the moment is that it is a branch, so that is the signal the control will send. Later after the condition has been checked in the execute stage, we can combine the branch signal with the result of the condition to generate PCSrc. ALUop is not described, though the value of both its bits is, since it is not a boolean signal. ALUop is a signal that tells the alu control unit, which we will build in the execute stage, what to do or to check the function field.

Consider the code in Listing 5.2. We introduce the switch statement, since it is the fastest way to check an integer indexed group of things, both in hardware and software. The other well known alternative is a group of nested if-then-else statements. In hardware[1] this makes a series of nested 2-input muxes, which means there will be log base 2 levels of muxes to go through. In complexity lingo, this is $O(\log(n))$ in time (how long it takes), and $O(1)$ in space (how big each mux is). A case statement is similar to a switch-case statement in C. In hardware[2] this makes a single large mux, which thus has only one level of evaluation. In complexity lingo this is $O(1)$ in time, and $O(n)$ in space. We care about time, so we will go with the case statement. Our case statement is

[1]This is similar for software, where if you code it well you can get log base 2 evaluations on average.

[2]In software, this makes a branch table which takes no conditions and only a single branch to resolve!

missing the signal values, which you can fill in from Figure 5.1. Note I have defined the ALUop values as constants to improve readability, so you should use them (see definitions.vh for the names).

Listing 5.2: Verilog code for the control unit.

```
`include "definitions.vh"

module control(
    input [5:0] Opcode,
    output reg  RegDst,
    output reg  Branch,
    output reg  MemRead,
    output reg  MemtoReg,
    output reg  [1:0] ALUOp,
    output reg  MemWrite,
    output reg  ALUSrc,
    output reg  RegWrite
    );
    always@(*) begin
        case (Opcode)
        `RTYPE: begin
                RegDst<=;
                Branch<=;
                MemRead<=;
                MemtoReg<=;
                ALUOp<=`ALUOp_R;
                MemWrite<=;
                ALUSrc<=;
                RegWrite<=;
            end
        `LW: begin
                RegDst<=;
                Branch<=;
                MemRead<=;
                MemtoReg<=;
                ALUOp<=;
                MemWrite<=;
                ALUSrc<=;
                RegWrite<=;
            end
        `SW: begin
                RegDst<=;
                Branch<=;
                MemRead<=;
                MemtoReg<=;
                ALUOp<=;
                MemWrite<=;
                ALUSrc<=;
                RegWrite<=;
            end
        `BEQ:  begin
                RegDst<=;
                Branch<=;
                MemRead<=;
                MemtoReg<=;
                ALUOp<=;
                MemWrite<=;
                ALUSrc<=;
                RegWrite<=;
            end
        default:  begin
                RegDst<=;
                Branch<=;
                MemRead<=;
                MemtoReg<=;
                ALUOp<=;
                MemWrite<=;
                ALUSrc<=;
                RegWrite<=;
            end
        endcase
    end
endmodule
```

5.3 Your Assignment

You are to:

1. Finish the sign extender and control unit.
2. Write a testbench for the sign extender and control unit, run them and generate the 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 the course LMS.

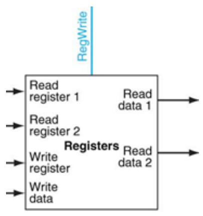
Lab 6

Register File

There are only two remaining parts of the decode stage: the register file (this week) and the buffer (next week).

Consider the interface of the register file, as depicted in Figure 6.1. Note that inputs enter on the left, controls on the top, and outputs exit on the right. We will be making a unit with 32 registers of size WORD. Also note that clock and reset are not included, but should be implicit, since we are building memory. We will need to read two registers at a time to do our arithmetic on two inputs (called A and B[1]) We will have one value to input, but only if write is asserted. Be careful of the timing - since the update is on the negative edge, just like the buffer which will drive it, a delay will speed things up.

Figure 6.1: Register file.



Notice that the input (writing to the registers) and the output (reading from the registers) will not be done at the same time due to the clock edge they happen on. This ensures that the data can update and be read in the same clock cycle. This will also require that the updating stage, called write back (WB or w/b for short), be a very small stage. Write back indeed is small, being composed of just one mux and the inputs to the registers.

One final note, it would be tempting to take the write signal for the register file directly from control, but don't! The write signal, the write address, and the write value must all arrive at the same time or you will update the wrong value or to the wrong place. We thus will pass all three signals along down the pipeline so the buffers at each stage will keep them coordinated. This is really important. The last buffer, mem-wb, will then be used to drive the update of the register file during the write back stage, when we know the values are good.

[1]These are the old names that was pretty universally used. I keep using them because it is a tip of the hat to where we came from.

6.1 Your Assignment

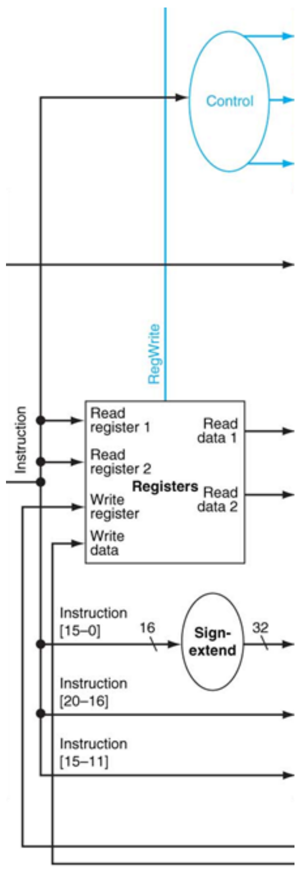
You are to:

1. Finish the register file.
2. Write a testbench for the register file, 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 the course LMS.

Lab 7

Completing Decode

Figure 7.1: iDecode



7.1 Buffer

Buffers are a pretty simple concept, so I won't spend a lot of time on it. I am giving you the entire buffer code from this stage (no delay included or needed). Your key part will be in testing and incorporating it.

7.2 Instruction Decode

Probably the biggest job will thus be creating your instances of each element and hooking them up. I have given you the basic code, but did not connect them. All the wires needed are already made, so it should be straightforward to hook them up. You will need to consult Figure 7.1, to do the hooking up. Note many signals come from the portions of the IR, based on the command type (R, I, J) and the corresponding bits of the field you want to access. In some cases (particularly the outputs), you will need to refer to the fields by their names (RT, RD). Don't forget that all outputs of the stage must be driven by the buffer, so they are timed correctly! The line coming across between the control and register file is nPC - it just passes through, and will be used in execute to calculate the branch target.

7.3 Your Assignment

You are to:

1. Write a buffer for the outputs of the decode stage.
2. Write a testbench for the buffer, run a simulation and generate a timing diagram.
3. Integrate the control, sign extender, register file, and buffer into iDecode.
4. Write a testbench for the iDecode, run a simulation and generate a timing diagram.
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 8

ALU and ALU Control

8.1 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 Figure 8.1 you can see the meaning of the control bits used to determine what the ALU will calculate. One way to build the ALU is to have it calculate each function and select the one you want. This is fast and simple but wasteful, thus you would need to determine if speed is a good enough reason to do this. In our case it is the approach we will take due to the simplicity. Generating all and selecting indicates a mux, and since there are many options it is a large mux, so a case statement is called for. The ALU control bits have been given names in the definitions.vh file, so be sure to use them as the individual 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, here are two (you can pick).

Figure 8.1: ALU control bit meaning.

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set-on-less-than
1100	NOR

Figure 8.2: ALU control bit generation.

Instruction opcode	ALUOp	Instruction operation	Function code	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

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

2. Verilog gives you reducing logic by putting the gate before an array of bits, indicating all the bits are to be connected by that logic gate to produce a single output. This is a particularly elegant gate that is useful in many places including here. Think how you can determine something is zero by reducing with a gate, then possibly negating.

8.2 ALU Control

Now we need to build the controller to use the ALUOp field and the function field to generate the ALU control bits used above. Consider the table in Figure 8.2. The first two columns are the initial sorting on ALUOp. The second two columns explain the secondary sorting based on the function field used for R-Type commands. The final two columns explains what the output of the function should be. Selecting between many connected values indicates a mux and thus a case statement. A two layered sorting indicates a nested mux/nested case statement is needed. Be careful when you code as the nesting is only needed for the R-Type commands. In both the case statements include a default to handle undefined signals (use fast commands for undefined signals). Use the signal names defined in the 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).

8.3 Your Assignment

You are to:

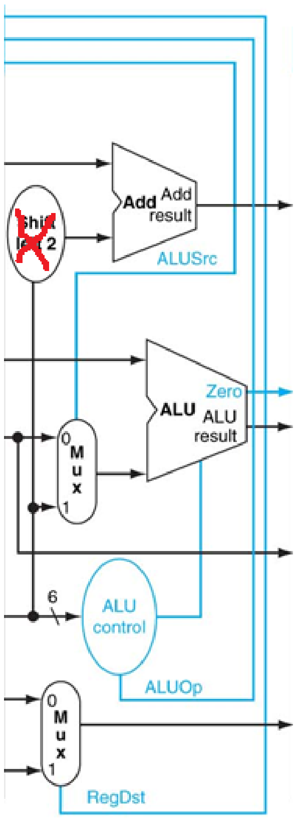
1. Finish the ALU and ALU control modules.

2. Test both modules with a testbench, 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 the course LMS.

Lab 9

Execute Stage

Figure 9.1: Execute



Note that I put a red 'x' over the left shift two unit. In our case it is not needed, as we are a word addressable machine, so the address is correct as is. The machine from the book is byte addressable, but word length for commands, thus the address must be multiplied by four (shifted left by two in binary) to handle this. Our simpler machine helps us here. Realize the left shift is easily handled by just appending two grounded wires on the right and dropping the two left most bits - a simple job for concatenation. Though simple it is still something and since unneeded, it is left off. The immediate value (from the sign extender) is passed directly to the adder.

9.1 Assemble

You have all the units you need to build the execute stage. Make sure you follow the wiring in Fig 9.1 carefully. Separate your signals into three categories: input, internal, output. In this stage all output comes from the buffer, so the execute stage outputs must come from the buffer outputs. I highlight the inputs, and wire them up first, marking each one complete with a red line when you do it. Once that is done go through the internal wires. There should be a wire for each already. When you use a wire mark it with a red line to keep track. You should finish quickly if you follow this methodology.

9.1.1 Buffer

The buffer is a simple design, just like the previous one and is provided to save time. Adapt your prior testbench to verify.

9.1.2 Mux and Adder

Your mux and adder have been tested and used already so they do not require further testing. Simply hook them up per Fig 9.1. Validation will be done by testing the entire unit.

9.1.3 ALU and Control

You built and tested these last time, so this time you only need to instantiate. Make sure you connect them as indicated in Fig 9.1.

9.2 Test

Now you need to test it. You should do a fake command of each type with data values that can test key functioning like overflow, etc. Look at the command list and I would put each command as a comment in your testbench, followed by the settings to test it. Do a couple data values for each, and it should be good. Make sure you have a reason for your choices.

9.3 Your Assignment

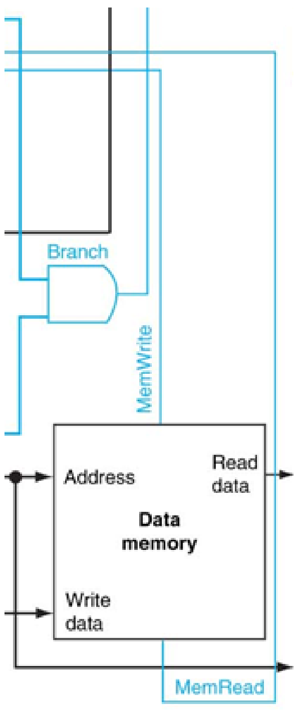
You are to:

1. Write a testbench for the buffer, run a simulation and generate a timing diagram.
2. Integrate the units into execute and run your simulation and generate a timing diagram to verify it works.
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 10

Memory

Figure 10.1: iDecode



10.1 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. And gates are a primitive in Verilog and are declared the same as a user defined module, i.e. by stating the type (and) followed by a name and then a parameter list in parenthesis. The output is first and there can be two or more inputs. Since it is a primitive you don't need to verify it and can just place it in the memory stage, making this a one line implementation.

10.2 Data Memory

This will be almost exactly like the instruction memory, with only two changes:

1. reading is now conditional on the 'read' control wire being high.
2. writing is now permissible if the 'write' control wire is high.

As such, take your instruction memory (it is the right size, you could also use your register memory, but that would require more modification) and add the two changes above then test.

10.3 Final Buffer

As before the buffer is simple but tedious so I am providing it. To save time, I am providing you a 'certified' buffer, so you don't need to test it.

10.4 Your Assignment

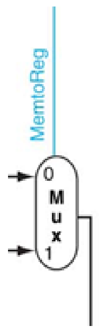
You are to:

1. Instantiate the AND gate directly in the memory stage.
2. Use instruction memory as a basis for data memory, and add the two new changes. Verify with a testbench and timing diagram.
3. Integrate data memory and the buffer into the memory stage. Verify it works with a testbench and timing diagram.
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 11

Write Back

Figure 11.1: 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 Pipeline

You are ready to assemble the pipeline. The basic pipeline file with all needed wires has been provided. You need to instantiate your modules and hook the wires up according to Fig 11.2. Here is my advice on how to do this simply and easily:

1. Open iFetch and copy the interface (from iFetch to the first ';') then paste it into pipeline after the comment stating 'Fetch'. This will become the instantiation.
2. Copy all the outputs in the interface again up to the space right after the comment stating 'wires from Fetch outputs'.
3. Change all the words outputs to wire in the section for wire instantiation. You are now ensured that the wires are the right size.
4. Since other modules could use the same names and this could cause conflicts, append `_IFID` after each wire name you just created to show it came from the IFID buffer. Make sure you change all the commas to semicolons and put a final semicolon on the last entry. Later you can use`_IDEX` for the

   IDEX buffer, etc. A few wires will come from a stage without buffering - such as PCSrc from the Mem stage, for which I suggest you use something like `_nbMEM`. The nb means no buffering and the MEM is for the stage. I like the stage time it is coming from to be the last thing because when I look in the timing diagram it reads nice to see the stage it comes from as the last (most noticeable) part of the signal name.

5. Back in the instantiation part (made in step 1), change everything before each port name to a '.' and put a pair of parenthesis after the port name.

6. Put `clk_a` and `reset` in to the clk and reset ports.

7. Put the appropriate wires you created in step 4 into the parenthesis of the outputs.

8. The inputs will come from later stages so you will place them when you create the appropriate stage.

9. repeat for the next stage till you finish Write Back.

This may seem tedious at first, but try it and you should find it goes fast and is fairly error-proof. I would split the stages between the partners in your group. Decode and Execute are the biggest. A roughly fair breakdown is one partner does before Fetch and Decode, the other does Execute, Memory, and Write Back.

Note the clock and reset are not shown. Make sure you use `clk_a` and not `clk`, the difference being `clk_a` can also start and stop (while preserving values) the pipeline. Be careful to pass back signals after the appropriate stage, some signals like exit after the ex-mem buffer.

Verify by running programs and testing the output.

11.3 Your Assignment

You are to:

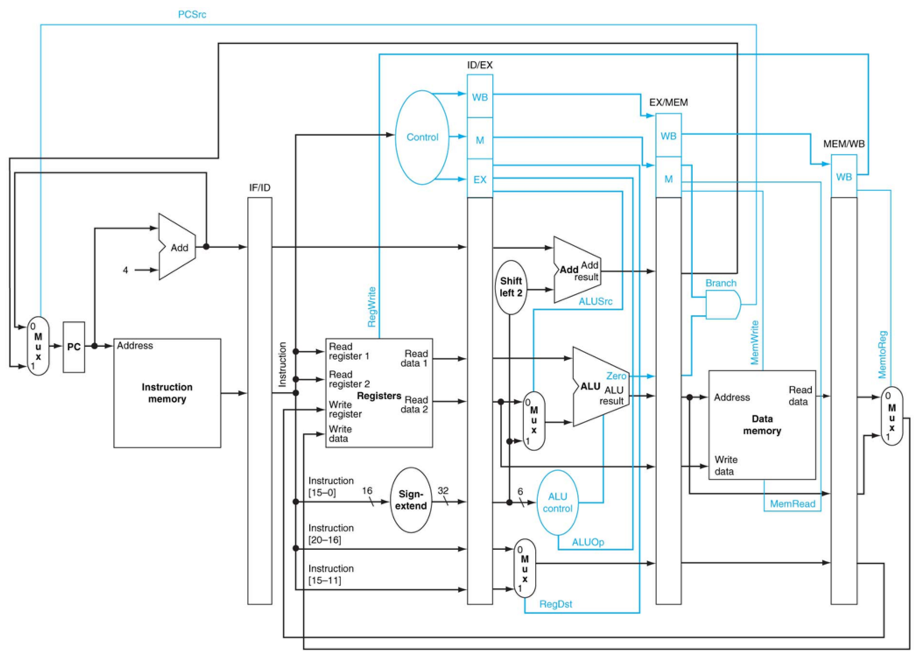
1. Create the Writeback stage consisting of one Mux.

2. Integrate all five stages into the file pipeline.v. Verify with test programs.

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.

Figure 11.2: Full Pipeline.



Lab 12

Coding

This is an test lab to help you debug and verify your pipeline. At the moment you don't have a hazard detection circuit to do a stall/slip or a forwarding unit to pass values to where they are needed so you will need to put four no-ops between each real instruction to ensure everything finishes. If you have time you can implement forwarding at the end of the lab and earn extra credit!

12.1 Implementing Code

Pick a snipet of MIPS code that you know how it works and what it will produce from the text or one of my sample tests. Copy the three sum testfiles and rename them to put your machine code in. Assemble each line of your chosen snipet into MIPS machine code and replace the instructions (not the no-ops) in the files. Specify the starting data values in the registers and data memory by changing the contents of your newly renamed R and D files. Change the file names pointed to in the 'definitions.vh' file to the new ones. Run your simulation and verify the functioning of the code! I suggest each partner pick one snipet to do, so you will get two verifications.

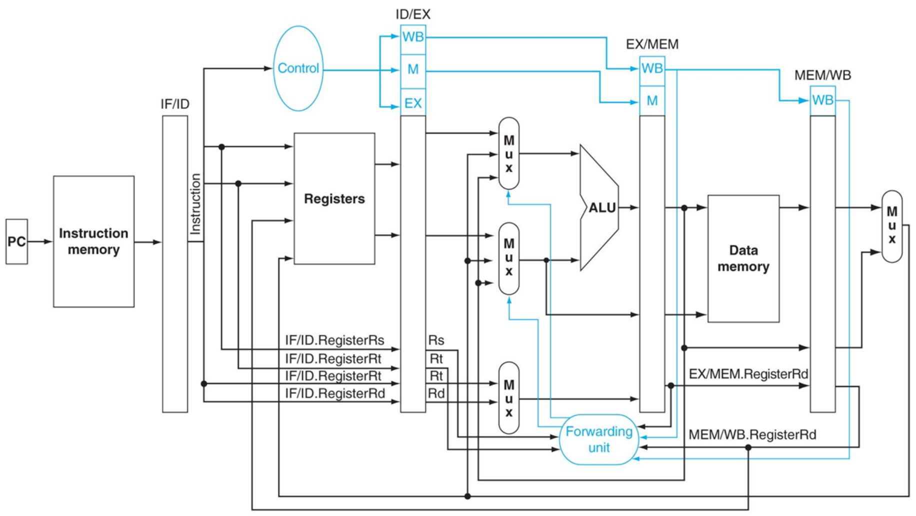
12.2 Forwarding

If you finished the pipeline and verified it successfully then congratulations you hit a great milestone! Now see if you can modify the pipeline you built to include the forwarding unit in Fig 12.1. This will allow R-Type commands to go back to back, i.e. no four command delay between them.

Make a copy of your project and call the new one forwarding or something. Many people have ruined their project and not had an easy way to restart, please be careful.

Go one step at a time and implement everything in the picture. The components are simple, so I don't think you should have a problem. You will need to make new wires and break old connections. You will also have to add ports

Figure 12.1: Pipeline with forwarding.



to your existing modules. I suggest picking a direction - inside out or outside it - meaning start from inside the lowest level module in a stage and work out till the stage is done, or work from the interface of the stage to the lowest level module (outside in). Either is fine, one will fit your style better.

If you finish it, write it up and name it 'LabForwarding' so I can find it easily. You have now hit the ultra elite group of about the top 10% of all computer organization students I have taught. Great job!

12.3 Your Assignment

You are to:

1. Finish your pipeline if not done.
2. Each partner should convert a snipet of code, run it, and verify the solution.
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.

For extra credit you can:

1. Add forwarding, then run a simulation and generate a timing diagram to verify.

2. Write up a lab report in LaTeX following the lab format in `LabN.tex` and generate a pdf file as ‘LabForwarding.pdf’.
3. Upload the pdf and a zip of all the Verilog files for forwarding (to keep the separate) to the course LMS.

Lab 13

Wrapping up

As before, many will need time to debug. Make sure you methodically try to break down errors. If you are having a problem be sure to have me or a lab assistant help you with the code. Don't keep quite, a different set of eyes is invaluable to solving a problem.

13.1 Performance of Forwarding

If you finished both the pipeline and forwarding then you can do one more addition for extra credit, and squeeze out every last bit of digital fun before the semester tragically must end. You must do forwarding first, so if you just finished the pipeline in this last lab then please go to the previous one and do forwarding if you want extra credit. Forwarding is needed to test the performance. On the non-pipelined system each command must finish before the next can start. On the pipelined system this is not so. You are going to assess performance by calculating the speedup of your system with forwarding.

1. Run a snipet of code that has a loop on your original pipeline with all the no-ops intact. Verify the solution. Take the time that the final answer is calculated and call it T_{old}.

2. Run the same snipet of code on your forwarding pipeline with the no-ops between RType commands removed. Verify the solution. Take the time that the final answer is calculated and call it T_{new}.

3. Calculate the speedup by $S = \frac{T_{old}}{T_{new}}$.

4. Repeat this for the looping the following number of times: 1, 2, 4, 8, 16, 32, 64. Plot the speedup versus number of times you looped. Explain what you see.

If you finish, you have entered the top 1% of all time! Superb job!

13.2 Your Assignment

If you haven't already, you are to:

1. Finish your pipeline if not done.
2. Each partner should convert a snipet of code, run it, and verify the solution.
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 in the last labs space.

For extra credit you can:

1. Add forwarding, then run a simulation and generate a timing diagram to verify.
2. Write up a lab report in LaTeX following the lab format in `LabN.tex` and generate a pdf file as 'LabForwarding.pdf'.
3. Upload the pdf and a zip of all the Verilog files for forwarding (to keep the separate) to the course LMS.

For more extra credit you can:

1. Do a perfomance tests on your pipeline with and without forwarding as described above.
2. Write up a lab report in LaTeX following the lab format in `LabN.tex` and generate a pdf file as 'LabPerformance.pdf'.
3. Upload the pdf and the test files you used to the course LMS. I should have your verilog from earlier. If you had to change anything make a zip of the entire pipeline so I don't have to piecemeal it.

If you are done and don't want to do more, then don't upload anything so I know there is nothing to grade. The last lab is extra credit.

Lab 14

Bye!

God Bless!

and

Soli Deo Gloria!