Contents

1 Introduction ........................................ 3
   1.1 Modulo 3 Incrementer .......................... 3
   1.2 Test Bench for Modulo 3 Incrementer .......... 5
   1.3 Your Assignment ............................... 7

2 Adder With Reservation Stations .................. 8
   2.1 A Simple ALU ................................. 8
   2.2 Test Bench for the Adder ..................... 13
   2.3 Your Assignment ............................... 18

3 Register File ....................................... 19
   3.1 Outline ...................................... 19
   3.2 Test Bench for the Register File .............. 20
   3.3 Your Assignment ............................... 25

4 Instruction Queue .................................... 26
   4.1 Outline ...................................... 26
   4.2 Test Bench for the Register File .............. 28
   4.3 Your Assignment ............................... 31

5 Tomasulo’s Method .................................. 33
   5.1 Your Assignment ............................... 33

6 Calculating the Pentagonal Numbers ............... 34
   6.1 Your Assignment ............................... 35
Lab 1

Introduction

In this lab manual we will build a simple Tomasulo machine, which will help you to understand HDL\(^1\) programming in Verilog, as well as how to exploit ILP\(^2\) using an out of order execution unit. I will assume you have seen some HDL programming before, and given the ease of picking up Verilog any HDL language should be enough. Verilog is one of the coolest languages I know. It is an object oriented, event driven, multi-threaded, intrinsically parallel language for simulation and synthesis of logic. There are lots of good language manuals out there so I will concentrate on an example.

1.1 Modulo 3 Incrementer

Listing 1.1: Verilog code of the modulo three incrementer.

```verilog
module mod3counter(
    input wire [1:0] num,
    output reg [1:0] mod3num
);

always@(*)
begin
    //modulo 3 counter

end
endmodule
```

Consider the code in Listing 1.1. This code declares a single module, which you can think of as a class. The module's name is mod3counter, which is

---

\(^1\)Hardware Description Language  
\(^2\)Instruction Level Parallelism
LAB 1. INTRODUCTION

descriptive, as we want to create a modulo 3 counter\(^3\). This is a fairly typical
digital design activity. A modulo 3 counter counts from 0 to 1 to 2 then back
to 0 again. Technically we are going to make a modulo 3 incremter, but
that is not that important here. Note that parameter passing requires explicit
identification of which way the data is to be passed (input, output). In the next
lab we will use a third option of bi-directional transport (inout) for a bus. We
also need to specify the type of data that is being passed (wire, reg). There are
other options but these are the most commonly used by far. If you don’t put
anything, Verilog will assume a wire.

Wires and regs are designed to model real elements, so for instance a wire
connects multiple devices, and allows any of them to assert the value being
passed to the others. Two different sources both applying different voltages will
cause a short down the line and you will get a value of ‘x’ meaning don’t know.
If all sources are isolated from the wire (it is disconnected somehow), then the
value isn’t zero (even in real life) rather we label it ‘z’ meaning high impedance
(isolated from a source, but it will adjust to any new source if it is applied. We
can even specify the degree of isolation, but not in this course. Thus a wire
needs to have something constantly driving it to have a legitimate value.

On the other hand, a register (reg for short) can connect to multiple outputs
and can have different sources, but it must be updated at distinct points not
constantly driven. For instance a register might update on rising edge of a clock
signal. Registers can hold their value, thus they can, and must be updated at
discrete periods of time. This makes a reg legitimate for a code block, but not
a wire. This is the most common mistake of novice (and some experienced)
Verilog programmers- driving registers or discretely assigning values to a wire.
Neither makes since (well I could drive a register, but it isn’t smart and Verilog
won’t let you). The underlying data structure to a wire is called a net, so you
will often see a message ‘not a net’ if you confuse a register and wire.

The final thing to notice is the \([1:0]\) part, which specifies the ‘wire’ is
actually a bundle of two wires. The most significant bit is on the right and is
numbered 1, while the least significant is on the left and is numbered 0. You
could have put the 1 and 0 in the opposite order, but that would open the
big-endian/little-endian debate. We will consistently keep the largest number
on the most significant bit, thus the least significant bit is at the lower address,
or little-endian.

Now notice that Listing 1.1 has block of code (see the begin and end) that
is begun by an always. Think of this as a thread which never terminates. The
at sign specifies the event to wake it up, and in this case it is a star, meaning
if any variable in the block changes. Should any variable input to the always
block change, the thread will be woken up and executed. The comments are
just like C++ and most programming syntax is like C.

At this point you should notice that we have not given any real code to run.
That is now up to you. First you need to think what needs to be done. One

\(^3\)You might wonder why we want to do this, as our ultimate goal is to make a Tomasulo
machine. Well, this module is needed to walk through which reservation station is used next.
We will explain reservation stations in the next lab.
way to handle this is write a table of the inputs and outputs, which are quite short in this case. Put this in a comment just below the comment in the always block. Now you need to change the output conditioned on the input, so we will need a conditional statement. Try searching for both the if-else construct and the case statement. Which makes the most sense here? Code your choice in the always block. Notice we can change the output (mod3num) in a code block because it is a reg.

1.2 Test Bench for Modulo 3 Incrementer

Now that we have a module for a modulo 3 incrementer, we need to make an instance and simulate it for testing purposes. We could later use the same Verilog code to synthesize logic and program say an FPGA, but that is the subject of CSCI 311 and CSCI 521. We only need to simulate and test, so we need to write a test bench. The terminology goes back to the old days of hardwired testing (usually with wire wrap), where you would have a physical testing bench, which typically was set up to rapidly connect and test circuits.

In any case, our test bench will be another module, but this one will not have any inputs or outputs, and it will create an instance of our incrementer, see Listing 1.2. Note that there are no parenthesis since there are no parameters. This is a top level module. We will declare two local variables, a reg, to allow us to assign the input, and a wire to receive the output (driven by the reg in the modulo counter).

Listing 1.2: Verilog code of the modulo 3 counter test bench.

```verilog
module mod3counter_test;
reg [1:0] num_in;
wire [1:0] num_out;

mod3counter mod3inc(
 .num(num_in),
 .mod3num(num_out)
);

initial begin
  num_in <= 2;
  #10;
  num_in <= 1;
  #10;
  num_in <= 0;
  #10;
  num_in <= 5;
  #10;
end
```

4Field Programmable Gate Array
The instantiation involves declaring first the module name (your new type), followed by an instance name, and the specification of the parameters. Note that this is not a call to function, it is an instantiation, and the parameters are not values passed per-se, but rather hard connections between entities in the incrementer and those in the test bench. Don’t get this confused. You cannot instantiate a module in a thread, such as an always block, as code blocks manipulate values in hardware, they don’t make hardware. An instantiation makes hardware. I am declaring a new instance of a piece of hardware, just like a register or wire. Note also how I am passing the values. I am using Verilog-2001 conventions, which are vastly better the the Verilog-1995 conventions used in many books and online references. Verilog-1995 requires you to place the names of the external elements you are connecting between the parenthesis in the same order they are internally. This is basically what most normal programming languages do. You might wonder what is so bad about this, particularly given that the syntax I am using is much longer (around twice). Well, for small examples it is not bad, but for large ones it is easy to get things out of order, or pass the wrong size, which can cause buffer overflow errors (just like in regular programs). The Verilog-2001 syntax removes the need for the same order by creating the `.internal(external)` syntax. This allows us to explicitly connect the entities we want to. Purists might complain the code is not black box because you need to know names from inside the module being instantiated, but you need to know order and such with the other method so it cancels in my mind. The reality is you will get far fewer errors with the Verilog-2001 syntax, so I am requiring it. Note you can switch the order of the two ‘dot’ declarations without a problem as long as there is a comma between them.

Something new also appears in the main body, an initial block. This is another thread that starts at the beginning of the simulation (but after instantiations), and terminates when the last command is done. Inside the block, we have non-blocking assignments, `<=`. There are also blocking assignments like normal programming languages, `=`, but you shouldn’t use them unless you have a good reason. Blocking assignments do just what they say, they block the next statement from happening till they finish, but this doesn’t happen in real physical elements. One wire doesn’t pass its current, then the next one, it all happens at once. Multiple non-blocking statements happen in parallel. If we want to have them happen in order, with some time delay between them, then we should be precise in this and put in a time delay, by using the `#delay` syntax. This is just what I do. I set the value of the register then wait 10ns (the default simulation time is 1ns). The final delay is to give time before the simulation ends. This is only needed for some simulators, and though mine does not require it, you might have one that does so I include it for generality. It can’t hurt. The resulting timing diagram of the simulation is in Figure 1.1. Notice the values of the wire bundles are printed in the boxes and the timescale is read on the horizontal axis. Timing diagrams are immensely useful for debugging. I am
1.3. YOUR ASSIGNMENT

You are to:

1. Finish the modulo 3 incrementer in Listing 1.1.

2. Run a simulation and generate a timing diagram like I did (yours will look different because you are not using the same program, thus you can’t just use my graphic).

3. Write up a lab report in \LaTeX{} following the lab format online and generate a pdf or ps file.

4. Make a zip or tar of all the Verilog and \LaTeX{} files, then upload to the course moodle.

using ModelSim XE from Mentor Graphics, but similar outputs are available from verilogo\text{r}, which is in the labs.

Figure 1.1: Modulo 3 timing diagram.
Lab 2

Adder With Reservation Stations

Now we get to jump in with both feet to the fun stuff. I suggest you read in advance 2.4-2.6 in the book on Tomasulo’s design for the IBM 360 floating point unit. You might also find chapter 28 in KOHW helpful.

2.1 A Simple ALU

We want to create a simple ALU, allowing addition, subtraction, and some logic operations. Typically such devices have no memory, so they just get two values and an operation passed in, and the result comes out. We want something a little more sophisticated. We want to be able to pass in two values and an operation, but we also want to be able to say if those values are valid values, or the address of something that will produce them, and thus wait for them to be produced and then calculate. We might have multiple operations before one finishes, so we want a queue of sorts (we will call these reservation stations) to hold the problems till they can be calculated. Further we might have multiple units returning results, so we need a bus for transmitting the answers. We will assume the values being calculated are 32 bit, 2’s complement values (specified by the signed keyword), and that there might be a total of up to $2^6$ reservation stations total for the computer (thus 6 bit addressing of the reservation stations). We will also assume the system commands are 6 bit, but that only the last 3 bits will specify the ALU operation (the first 3 bits specify the type of command, and thus the calculation unit). We will use 3 reservation stations for the ALU (in practice I would keep it to powers of 2 if possible, but I am doing this to force you to find out how to handle it). This results in a lot of inputs and outputs, see the template in Listing 2.1. Notice the inout declaration, which allows the bi-directional transmission needed for a bus. Some of the features will not be needed at the moment, they are included for later labs.
2.1. A SIMPLE ALU

Listing 2.1: Verilog code of the adder.

module adders(
    input wire clock, issue,
    input wire signed [31:0] A, B,
    input wire A_invalid, B_invalid,
    input wire [5:0] opcode,
    input wire CDB_xmit,
    inout wire signed [31:0] CDB_data,
    inout wire [5:0] CDB_source,
    inout wire CDB_write,
    output reg CDB_rts,
    output wire available,
    output wire [5:0] RS_available,
    output reg [5:0] issued,
    output wire [5:0] RS_executing,
    output reg error
);

    // constants
    // disconnected buses have high impedance
    parameter disconnected = 32’bz;
    // calculation delay
    parameter delay = 20;
    // updating delay
    parameter update_delay = 5;
    // no unit is calculating
    parameter none = 2’b00;
    // clear Qj/Qk, thus Vj/Vk valid
    parameter valid = 6’b000000;
    // error code
    parameter no_error = 1’b0;
    parameter all_rs_busy = 1’b1;
    // command not issued
    parameter not_issued = 6’b000000;
    // status of reservation station or alu
    parameter not_busy = 1’b0;
    parameter in_use = 1’b1;
    // is the value ready to send or not
    parameter ready = 1’b1;
    parameter not_ready = 1’b0;
    // used to clear a data value
    parameter clear = 0;
    // reservation station
    parameter no_rs = 6’b000000;
parameter adder_1 = 6'b000001;
parameter adder_2 = 6'b000010;
parameter adder_3 = 6'b000011;
// operation codes used by alu
parameter alu_add = 3'b000;
parameter alu_sub = 3'b001;
parameter alu_or = 3'b100;
parameter alu_and = 3'b101;
parameter alu_not = 3'b110;
parameter alu_xor = 3'b111;

// the result being written
reg signed [31:0] CDB_data_out;
// which rs is finished
reg [5:0] CDB_source_out;
// ready to write to CDB
reg CDB_write_out;
// operation to do
reg [5:0] operation [2:0];
// if values aren’t available here
// is where to get them
reg [5:0] Qj [2:0], Qk [2:0];
// values to calculate
reg signed [31:0] Vj [2:0], Vk [2:0];
// reservation station full indicators
reg Busy [2:0];
// alu calculating
reg Unit_Busy;
reg [1:0] adder_calculating;
// Reservation Station (RS) number of
// each station in this unit
reg [5:0] RS_num_of [2:0];
// we want to cycle the order the
// reservation stations are selected to
// go to execution to prevent starvation.
reg [1:0] Priority_Station;
wire [1:0] Second_Station;
wire [1:0] Last_Station;
// Intermediate wires in calculation of available RS
wire [5:0] RS_availability_of_Second_or_Last;
wire [5:0] RS_availability_of_Last;

// use tri state gates to connect the CDB
// output registers to the CDB, with
// CDB_xmit as control, thus handling writes
assign CDB_data =
    CDB_xmit ? CDB_data_out : disconnected;
assign CDB_source =
    CDB_xmit ? CDB_source_out : disconnected;
assign CDB_write =
    CDB_xmit ? CDB_write_out : disconnected;

// available indicates that at least one RS is not busy
assign available = ~(Busy[0] & Busy[1] & Busy[2]);
// assign RS_availability_of_Last =
// finish this line
// assign RS_availability_of_Second_or_Last =
// finish this line
assign RS_available =
    ~Busy[Priority_Station] ?
    RS_num_of[Priority_Station] :
    RS_availability_of_Second_or_Last;

// RS executing, so issue unit can track
// assign RS_executing =
// finish this line

// cycle the order of the stations
mod3counter mod3count1(
   .num(Priority_Station),
   .mod3num(Second_Station)
);
// calculate the last station

initial
begin
    CDB_rts<=not_ready;
    CDB_data_out<=clear;
    CDB_source_out<=no_rs;
    CDB_write_out<=not_ready;
    Priority_Station<=no_rs;
    Unit_Busy<=not_busy;
    Busy[0]<=not_busy;
    Busy[1]<=not_busy;
    Busy[2]<=not_busy;
    issued<=not_issued;
    error<=no_error;
    RS_num_of[0]<=adder_1;
    RS_num_of[1]<=adder_2;
    adder_calculating<=none;
end
always@(posedge clock)
begin
    // clean messages out each cycle
    issued<=not_issued;
    error<=no_error;
end

always@(negedge CDB_xmit)
begin
    // clean up write when finished
    CDB_rts<=not_ready;
    // CDB_write_out // finish this line
    // Unit_Busy // finish this line
    // Busy[adder_calculating] // finish this line
    // adder_calculating // finish this line
end

always@(posedge clock)
begin
    // each cycle handle execution
end

always@(negedge clock)
begin
    // handle updates
end

always@(negedge clock)
begin
    // handle issue
end
endmodule

Let me take a moment to discuss programming style. Note that I am putting each type of declaration on a separate line. This is done for readability. Note that I also try to use names that not only tell you what they are, but that in the case of boolean values (such as A_invalid) they tell you how to interpret the true false values. You should do this, it makes your code vastly more readable without introducing ugly comments that only serve to explain your poor naming conventions. In some cases I have used names I don’t like, for instance A or Qj,
2.2. **Test Bench for the Adder**

When we have an adder that works, or even before if we wanted to test parts of it, we need to write a test bench to see if it works. Since I have already
explained the basic idea and flow of them in the last lab, I will just present a test bench that you can modify and tune, in Listing 2.2. I am manually faking a clock by flipping the clock bit ever clock period. I also just set the equations I want to calculate. As a point of interest, I deliberately send more values than the reservation stations can hold to test the error flagging facility. You should also see that when I set a value as invalid, the system interprets the number as the address of the reservation station that will produce it. Other than that it is pretty straight forward in a long sort of way.

Listing 2.2: Verilog code of the adder test bench.

```verilog
module adders_test;

    parameter clock_period = 10;

    // operation codes used by alu
    parameter alu_add = 3'b000;
    parameter alu_sub = 3'b001;
    parameter alu_or = 3'b100;
    parameter alu_and = 3'b101;
    parameter alu_not = 3'b110;
    parameter alu_xor = 3'b111;

    // inputs
    reg clk;
    reg issue_command;
    reg [5:0] command;
    reg signed [31:0] A,B;
    reg A_invalid, B_invalid;
    reg CDB_xmit;

    // i/o used just as output
    wire signed [31:0] data_out;
    wire [5:0] data_from_rs_num;
    wire data_valid;

    // outputs
    wire CDB_rts;
    wire available;
    wire [5:0] issued_to_rs_num, RS_available;
    wire [5:0] RS_executing;
    wire error;

initial begin
    clk <= 0;
    issue_command <= 0;
    command <= 0;
```
2.2. TEST BENCH FOR THE ADDER

```vhdl
A <= 0;
B <= 0;
A_invalid <= 0;
B_invalid <= 0;
CDB_xmit <= 0;

#clock_period clk=˜clk;
#clock_period clk=˜clk;

issue_command <= 1;
command <= alu_add;
A <= 5;
B <= 5;
A_invalid <= 0;
B_invalid <= 0;
CDB_xmit <= 0;

#clock_period clk=˜clk;
#clock_period clk=˜clk;

issue_command <= 1;
command <= alu_sub;
A <= 3;
B <= 13;
A_invalid <= 1;
B_invalid <= 0;
CDB_xmit <= 0;

#clock_period clk=˜clk;
#clock_period clk=˜clk;

issue_command <= 1;
command <= alu_and;
A <= 15;
B <= 60;
A_invalid <= 0;
B_invalid <= 0;
CDB_xmit <= 0;

#clock_period clk=˜clk;
#clock_period clk=˜clk;

issue_command <= 1;
command <= alu_add;
A <= 25;
B <= 35;
```
A\_invalid <= 0;
B\_invalid <= 0;
CDB\_xmit <= 0;

\#clock\_period clk = \sim clk;
\#clock\_period clk = \sim clk;

issue\_command <= 0;
command <= 0;
A <= 0;
B <= 0;
A\_invalid <= 0;
B\_invalid <= 0;
CDB\_xmit <= 0;
\#clock\_period clk = \sim clk;

\#clock\_period clk = \sim clk;
\#clock\_period clk = \sim clk;

\#clock\_period clk = \sim clk;
\#clock\_period clk = \sim clk;

\#clock\_period clk = \sim clk;
\#clock\_period clk = \sim clk;

\#clock\_period clk = \sim clk;
\#clock\_period clk = \sim clk;

CDB\_xmit <= 1;
\#clock\_period clk = \sim clk;
\#clock\_period clk = \sim clk;

CDB\_xmit <= 0;
\#clock\_period clk = \sim clk;
\#clock\_period clk = \sim clk;

\#clock\_period clk = \sim clk;
\#clock\_period clk = \sim clk;

\#clock\_period clk = \sim clk;
\#clock\_period clk = \sim clk;

CDB\_xmit <= 1;
\#clock\_period clk = \sim clk;
\#clock\_period clk = \sim clk;
CDB\_xmit<=0;

#clock\_period clk=\~clk;
#clock\_period clk=\~clk;

#clock\_period clk=\~clk;
#clock\_period clk=\~clk;

#clock\_period clk=\~clk;
#clock\_period clk=\~clk;

CDB\_xmit<=1;

#clock\_period clk=\~clk;
#clock\_period clk=\~clk;

CDB\_xmit<=0;

#clock\_period clk=\~clk;
#clock\_period clk=\~clk;

end

adders my\_adders(
.clock(clk),
.issue(issue\_command),
.A(A),
.B(B),
.A\_invalid(A\_invalid),
.B\_invalid(B\_invalid),
.opcode(command),
.CDB\_xmit(CDB\_xmit),
.CDB\_data(data\_out),
.CDB\_source(data\_from\_rs\_num),
.CDB\_write(data\_valid),
.CDB\_rts(CDB\_rts),
available(available),
.RS\_available(RS\_available),
.issued(issued\_to\_rs\_num),
.RS\_executing(RS\_executing),
.error(error)
);
endmodule

When I run this on my adder and modulo 3 incremeniter, I get the timing diagram in Figure 2.1. Again, yours will look similar, hopefully. The answers
Figure 2.1: Adder timing diagram.

are output only when the CDB_xmit is set to true, i.e. when transmit (xmit) permission is given for the bus. Note the adder holds the values till then.

2.3 Your Assignment

You are to:

1. Finish the adder in Listing 2.1.

2. Run a simulation and generate a timing diagram like I did (yours will look different because you are not using the same program, thus you can’t just use my graphic).

3. Write up a lab report in \LaTeX following the lab format online and generate a pdf or ps file.

4. Make a zip or tar of all the Verilog and \LaTeX files, then upload to the course moodle.
Lab 3

Register File

3.1 Outline

As last time, I am including an outline of the code module, with strategic parts left out, see Listing 3.1. This code is actually easier than last lab, but I don’t plan to give as much help on it. Just remember how we have done our designs in the past couple labs. Come up with a description of an algorithm and put that in the comments, then figure out how to code that. This module is done much easier if you use a for loop (just like C syntax). You will need a loop variable, and Verilog provides an integer data type to handle this.

Listing 3.1: Verilog code of the register file.

```verilog
module Registers(
    input wire clock, issue,
    input wire [4:0] A_address, B_address, dest,
    input wire [31:0] In_data,
    input wire [5:0] In_source, RS_calculating_value,
    input wire write,
    output reg [31:0] A_out, B_out,
    output reg A_invalid, B_invalid
);

parameter not_redirected=0;
parameter true=1'b1;
parameter false=1'b0;
parameter num_of_regs=32;

reg [31:0] reg_file [31:0];
reg [5:0] redirection [31:0];
integer i;

initial
    begin
```
LAB 3. REGISTER FILE

for (i=0; i<num_of_regs; i=i+1)
begin
    reg_file[i]=i;
    redirection[i]=not_redirected;
end
end

always@ (negedge clock)
begin
    // handle issues
end

always@ (posedge clock)
begin
    // handle updates
end
endmodule

3.2 Test Bench for the Register File

By now you are hopefully comfortable with the concept of designing a test bench. There is nothing surprising in this one at all, see Listing 3.2. The timing diagram I get is also straightforward, see Figure 3.1.

Listing 3.2: Verilog code of the register file test bench.

module Registers_test;

    parameter clock_period = 10;

    // inputs
    reg clk;
    reg issue_command;
    reg [4:0] A_address, B_address, dest;
    reg signed [31:0] CDB_data;
    reg [5:0] CDB_source, RS_producing_result;
    reg CDB_write;

    wire signed [31:0] A,B;
    wire A_invalid, B_invalid;

    initial
    begin

3.2. TEST BENCH FOR THE REGISTER FILE

```vhdl
clk <= 0;

// reading variables
issue_command <= 0;
A_address <=0;
B_address <=0;
dest <=0;
RS-producing_result <=0;

// update variables
CDB_data<=0;
CDB_source<=0;
CDB_write<=0;

#clock_period clk=˜clk;
#clock_period clk=˜clk;

// reading variables
issue_command <= 1;
A_address <=1;
B_address <=2;
dest <=3;
RS-producing_result <=1;

// update variables
CDB_data<=0;
CDB_source<=0;
CDB_write<=0;

#clock_period clk=˜clk;
#clock_period clk=˜clk;

// reading variables
issue_command <= 0;
A_address <=0;
B_address <=0;
dest <=0;
RS-producing_result <=0;

// update variables
CDB_data<=0;
CDB_source<=0;
```
CDB_write <= 0;

#clock_period clk=˜clk;
#clock_period clk=˜clk;

// reading variables
issue_command <= 1;
A_address <= 3;
B_address <= 7;
dest <= 7;
RS-producing_result <= 2;

// update variables
CDB_data <= 0;
CDB_source <= 0;
CDB_write <= 0;

#clock_period clk=˜clk;
#clock_period clk=˜clk;

// reading variables
issue_command <= 0;
A_address <= 0;
B_address <= 0;
dest <= 0;
RS-producing_result <= 0;

// update variables
CDB_data <= 25;
CDB_source <= 1;
CDB_write <= 1;

#clock_period clk=˜clk;
#clock_period clk=˜clk;

// reading variables
issue_command <= 0;
A_address <= 0;
B_address <= 0;
3.2. TEST BENCH FOR THE REGISTER FILE

dest <= 0;
RS-producing_result <= 0;

// update variables
CDB_data <= 0;
CDB_source <= 0;
CDB_write <= 0;

#clock_period clk = ~clk;
#clock_period clk = ~clk;

// reading variables
issue_command <= 1;
A_address <= 12;
B_address <= 13;
dest <= 15;
RS-producing_result <= 3;

// update variables
CDB_data <= 21;
CDB_source <= 2;
CDB_write <= 1;

#clock_period clk = ~clk;
#clock_period clk = ~clk;

// reading variables
issue_command <= 0;
A_address <= 0;
B_address <= 0;
dest <= 0;
RS-producing_result <= 0;

// update variables
CDB_data <= 0;
CDB_source <= 0;
CDB_write <= 0;
 Registers myregs(  .clock(clk),  .issue(issue_command),  .A_address(A_address),  .B_address(B_address),  .dest(dest),  .In_data(CDB_data),  .In_source(CDB_source),  .RS_calculating_value(RS-producing_result),  .write(CDB_write),  .A_out(A),  .B_out(B),  .A_invalid(A_invalid),  .B_invalid(B_invalid) );

endmodule
3.3 Your Assignment

You are to:

1. Finish the register file in Listing 3.1. Note: you will need to add a few ports to handle errors at issue time, and should expand the use of the parameters to remove “magic numbers.”

2. Run a simulation and generate a timing diagram like I did (yours will look different because you are not using the same program, thus you can’t just use my graphic).

3. Write up a lab report in \LaTeX{} following the lab format online and generate a pdf or ps file.

4. Make a zip or tar of all the Verilog and \LaTeX{} files, then upload to the course moodle.
Lab 4

Instruction Queue

4.1 Outline

By now, I hope you are comfortable with the idea of developing the algorithms for a piece of hardware. The instruction queue needs to handle fetching instructions from memory, issuing instructions when a reservation station is available, tracking the status of an instruction, and removing the instruction from the queue when it is finished. Memory can be modeled as an array of 32 bit values for simplicity. I suggest you have a program counter that walks through the memory each cycle and fetches to the next available spot in the queue if there is one. My sample interface is in Listing 4.1.

Listing 4.1: Verilog code of the instruction queue.

```verilog
module instruction_queue(
    input wire clock,
    input wire issue_error,
    input wire adder_available,
    input wire[5:0] adder_RS_available,
    input wire[5:0] RS_issued,
    input wire[5:0] RS_executing_adder,
    input wire adder_rts,
    input wire[5:0] RS_finished,
    output reg[5:0] operation,
    output reg[2:0] execution_unit,
    output reg[4:0] Dest_address,
    output reg[4:0] A_address,
    output reg[4:0] B_address,
    output reg issue
);

parameter TRUE =1'b1;
parameter FALSE =1'b0;
```
parameter LAST_INSTRUCTION_ELEMENT = 1023;
parameter INSTRUCTION_ADDRESS_BITS = 10;
parameter LAST_QUEUE_ELEMENT = 7;
parameter QUEUE_ADDRESS_BITS = 3;
parameter CLEAR = 0;
parameter OPCODE_HIGH = 31;
parameter OPCODE_LOW = 26;
parameter EXEC_UNIT_HIGH = 31;
parameter EXEC_UNIT_LOW = 29;
parameter SOURCE1_HIGH = 25;
parameter SOURCE1_LOW = 21;
parameter SOURCE2_HIGH = 20;
parameter SOURCE2_LOW = 16;
parameter DESTINATION_HIGH = 15;
parameter DESTINATION_LOW = 11;
parameter BUSY_MASK = 4'b0001;
parameter ISSUE_MASK = 4'b0010;
parameter EXECUTE_MASK = 4'b0100;
parameter WRITE_BACK_MASK = 4'b1000;
parameter ADDER = 3'b000;
parameter alu_add = 3'b000;
parameter alu_sub = 3'b001;
parameter alu_or = 3'b100;
parameter alu_and = 3'b101;
parameter alu_not = 3'b110;
parameter alu_xor = 3'b111;

integer i, j;

reg issued_this_clock = 0;
reg [31:0] Instruction_Memory [LAST_INSTRUCTION_ELEMENT:0];
reg [INSTRUCTION_ADDRESS_BITS:0] PC = 0;
reg next_queue_location;
reg [31:0] Instruction [LAST_QUEUE_ELEMENT:0];
reg [5:0] RS_Holding [LAST_QUEUE_ELEMENT:0];
reg [3:0] Status [LAST_QUEUE_ELEMENT:0];
reg [QUEUE_ADDRESS_BITS-1:0] Queue_End = CLEAR;

initial
begin
    issue <= FALSE;
    for (i = 0; i <= LAST_QUEUE_ELEMENT; i = i + 1)
        begin
            Instruction[i] <= CLEAR;
            RS_Holding[i] <= CLEAR;
Status[i] <= CLEAR;
end
for (i=0; i<=32; i=i+1)
begin
    Instruction_Memory[i] <= {adder, alu_add, 5'b00001, 5'b00011, 5'b00111, 11'b0};
end
endmodule

4.2 Test Bench for the Register File

By now you are hopefully comfortable with the concept of designing a test bench. There is nothing surprising in this one either, see Listing 4.2. The timing diagram I get is also straightforward, see Figure 4.1.

Listing 4.2: Verilog code of the instruction queue test bench.

module Instruction_queue_test;

    reg clk=0;
    reg error=0;
    reg available=0;
    reg [5:0] RS_available=0;
    reg [5:0] RS_issue;
    reg [5:0] RS_exec=0;
    reg adder_rts=0;
    reg [5:0] RS_xmit=0;

    wire [5:0] opcode;
    wire [2:0] exec_unit;
    wire [4:0] A_address;
    wire [4:0] B_address;
    wire [4:0] Dest_address;
    wire issue;

    parameter clk_period=10;

initial
begin
    #clk_period clk=˜clk;
    RS_issue<=RS_available;
    #clk_period clk=˜clk;

    error <=0;
    available <=1;
    RS_available <=1;
    RS_issue <=0;
    RS_exec <=0;
    adder_rts <=0;
    RS_xmit <=0;

    #clk_period clk=˜clk;
    RS_issue<=RS_available;
    #clk_period clk=˜clk;

    error <=0;
    available <=1;
    RS_available <=2;
    RS_issue <=0;
    RS_exec <=0;
    adder_rts <=0;
    RS_xmit <=0;

    #clk_period clk=˜clk;
    RS_issue<=RS_available;
    #clk_period clk=˜clk;

    error <=0;
    available <=1;
    RS_available <=3;
    RS_issue <=0;
    RS_exec <=0;
    adder_rts <=0;
    RS_xmit <=0;

    #clk_period clk=˜clk;
    RS_issue<=RS_available;
    #clk_period clk=˜clk;

    error <=0;
    available <=0;
RS_available <=0;
RS_issue <=0;
RS_exec<=3;
adder_rts <=0;
RS_xmit <=0;

#clk_period clk=˜clk;
RS_issue<=RS_available;
#clk_period clk=˜clk;

error <=0;
available <=0;
RS_available <=0;
RS_issue <=0;
RS_exec<=0;
adder_rts <=1;
RS_xmit <=0;

#clk_period clk=˜clk;
RS_issue<=RS_available;
#clk_period clk=˜clk;

error <=0;
available <=0;
RS_available <=0;
RS_issue <=0;
RS_exec<=0;
adder_rts <=0;
RS_xmit <=3;

#clk_period clk=˜clk;
RS_issue<=RS_available;
#clk_period clk=˜clk;

error <=0;
available <=1;
RS_available <=3;
RS_issue <=0;
RS_exec<=2;
adder_rts <=0;
RS_xmit <=0;
4.3 Your Assignment

You are to:

1. Finish the instruction queue in Listing 4.1.

2. Run a simulation and generate a timing diagram like I did (yours will look different because you are not using the same program, thus you can't just use my graphic).
3. Write up a lab report in \LaTeX{} following the lab format online and generate a pdf or ps file.

4. Make a zip or tar of all the Verilog and \LaTeX{} files, then upload to the course moodle.
Lab 5

Tomasulo’s Method

Now it is up to you to combine the previous modules in a main module to make a functioning Tomasulo’s machine.

5.1 Your Assignment

You are to:

1. Write your own module to interface the previously developed ones.

2. Run a simulation and generate a timing diagram. This time there is no testbench, you should only have to set up instruction memory.

3. Write up a lab report in \LaTeX following the lab format online and generate a pdf or ps file.

4. Make a zip or tar of all the Verilog and \LaTeX files, then upload to the course moodle.
Lab 6

Calculating the Pentagonal Numbers

The pentagonal numbers are a mathematical sequence that is an example of figurative numbers. Figurative numbers are the number of dots on an appropriately nested sequence of polygons. In the case of the pentagonal numbers, the polygons are pentagons that share a vertex.

The first several pentagonal numbers are thus \( \{1, 5, 12, 22, 35, 51, 70, 92, \ldots \} \).

The challenge is then how do we calculate any pentagonal number using only addition? Sounds daunting, but this is where we stand on the shoulders of giants. Isaac Newton came up with a method of differences, which is easily understood from calculus. Assume we have a sequence of numbers, which is described by a polynomial. If we kept taking derivatives you would eventually end up with a zero as the answer, which means the previous derivative was a constant, say \( k_0 \). That means the derivative before it is just a sum of the constants \( k_0 \), with some initial value, say \( k_1 \). We can then use those numbers to add to calculate the next level up, and so on, till we eventually reach the actual polynomial. Interestingly this only involves addition. This cool property was noted by Charles Babage, who used it as the basis of the difference engine, one
6.1. YOUR ASSIGNMENT

of the early predecessors of the computer.

<table>
<thead>
<tr>
<th>Pentagonal Numbers</th>
<th>1</th>
<th>5</th>
<th>12</th>
<th>22</th>
<th>35</th>
<th>51</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt; Difference</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt; Difference</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.1 Your Assignment

You are to:

1. Modify instruction memory and the initial conditions of the register file to calculate the first 6 pentagonal numbers.

2. Run a simulation and generate a timing diagram.

3. Write up a lab report in L<sup>\text{\LaTeX}</sup> following the lab format online and generate a pdf or ps file.

4. Make a zip or tar of all the Verilog and L<sup>\text{\LaTeX}</sup> files, then upload to the course moodle.