Verilog Test Bench Design Code

Anand Rajan

Pennsylvania State University, University Park

Section 002R

CMPEN 331

Abstract

This final project serves as the culmination of all ideas and concepts presented throughout the class. In this project, the idea of the pipelining technique to construct a fast CPU is introduced, developed, and expanded to create a basic yet usable CPU. All five blocks that compose the CPU itself were constructed and perfected in previous lab sessions, and this lab serves as the presentation for it all.

The purpose of this project is, as mentioned, to construct a basic pipelined CPU. This warranted that each block should be able to handle its own instruction, as it would pass on that instruction to the next block at every clock cycle. This is done through the five blocks – Instruction Fetch, Decode, and Execute (IF, ID, EXE), Memory, and Write Back. Each block is composed of its own individual modules, which will be looked into in the next section. As a brief overview, the IF stage is charged with quite simply fetching the instruction. The PC provides a starting value and this stage is concerned with fetching the instruction at that PC value from the Instruction Memory module.

The Decode and Execute stage are charged with their self-explanatory tasks of decoding said instructing and proceeding to execute it. The instruction is decoded by splitting it up and processing each segment in different modules, most importantly the Control Unit. Depending on the values of the instruction, the output of this block will determine how the rest of the program proceeds. Execution of this instruction relies on the ALU module to perform the instruction specified by the opcode segment.

Finally, the memory block involves simply reading data memory at the provided index. The last write back stage assumes that the value obtained is now to be written into memory and performs just that.

This was the fundamental CPU built across the first 5 labs. As an honors options task as well as in the final project, forwarding of outputs in order to subvert data hazards was a requisite. While the initial structure of our CPU involved keeping the register file in the ID block, moving it to the WB block can optimize the circuit but at the same time, introduce data hazards. This is caused in the event that a value that needs to be written back is also being called upon by a previous instruction in the EXE or MEM stages.

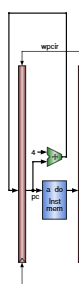
To prevent these hazards, forwarding was done. Rearranging the register file to the write back block, we must now introduce the addition of 2 new modules, namely 2 new forwarding multiplexers, to handle the data hazard. While register values are taken at each stage and sent back to the control unit, only the values of the RegWrite signal in each stage, as well as the source, target, and destination register values matter. At any given time, if a signal is to be written back, a hazard arises if the destination register of that write back is also the source or target register of an instruction in the EXE or MEM stages. The logic won’t be discussed thoroughly as the code implemented focuses on this logic. However, this logic was accounted for and the update of the control unit as well as the addition of the forwarding multiplexers complete this.

Having now provided a brief level overview of what will be done in this project, an introduction into the diagram and circuit itself will be provided.

Introduction

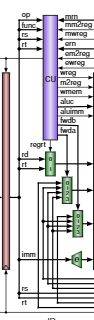
This section will serve to explain in detail each block and its purpose, as well as how it connects in an overarching manner to the rest of the circuit.

Starting with block 1, the information fetch stage, the only components involved are the IF register, the instruction memory, and a simple adder. The IF register contains the value of the PC (program counter) and this value is fed into both the adder as well as the instruction memory. The adder simply takes this input, adds 4 to it, and feeds it back into the IF register to update the PC for the next clock cycle. The instruction memory seeks the instruction found at the index given by PC and outputs it. This connects to the second block through the ID register.



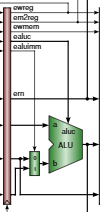
*Figure 1. Block 1 – Information Fetch Stage and circuitry*

The ID register only contains the value of the instruction. This block now acts on the next clock cycle as the IF block will now work on the next instruction. The instruction is broken down into several segments. The opcode and the function segments are fed into the control unit, as well as the source and target registers. The control unit uses this information to determine several things. Firstly, it outputs a high or low value for the output register (RegDst) using this information and feeds it into a multiplexer. This mux uses this as a selector bit to choose between the destination and the target register as the output register. Additionally, the control unit outputs several bits that will be of use to the ALU in the next block, such as the aluc, aluimm, and other information bits like wreg (RegWrite), m2reg (MemToReg), wmem (MemWrite), and the forwarding bits for the forwarding multiplexers. These bits are the selectors for the muxes of the same name that determines whether forwarding needs to be done or not. Lastly for this block, the immediate of the instruction is fed into a sign-extender (32 bit) and the source and target registers are also sent as inputs to the register file (found in block 5). This connects to the next block through the EXE register.

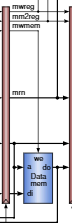


*Figure 2. Block 2 – Information Decode stage and circuitry*

Block 3, or the execute stage, is the calculation part of the CPU. It takes in the inputs from the EXE register and uses them in this block in the next clock cycle. The output of the second forwarding mux (QB) is fed into a mux in this stage, along with the output of the sign extender. The selector bit is ealuimm, and this mux’s output is fed into the ALU. The purpose of choosing with this mux is to determine whether the immediate is used in the calculation or not (in which case, the QB output of the register file is the default). The ALU is the center for arithmetic operations, and the inputs are A and B (corresponding to the output QA from the register file and output of the mux, either QB or extended immediate). Whether it be addition, subtraction, etc, the ALU will perform the chosen operation as given by aluc (determined by the control unit previously) and output it as ALUOUT. The result of this operation is fed into the next MEM register.

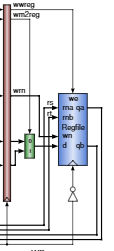
  
*Block 3. Execute stage and circuitry*

The memory block simply reads what is at this data memory. The data memory block uses the output from the ALU in the next clock cycle as an index and reads the memory it has stored at that index, should it be within range. The ALU output is still pipelined directly to the next register, but the output of the data memory is also checked and stored.



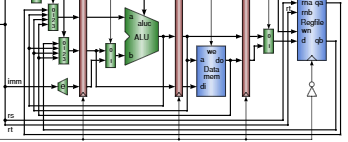
*Figure 4. Block 4 – Memory stage and circuitry*

Finally, the write back stage simply takes whatever values were written into the WB register (outputs of the MEM block) and in the next clock cycle, feeds the DM output into a multiplexer alongside the ALU output. Essentially, based on whether you want the memory to be sent from the register or not (MemToReg is high or low), the mux will choose between the newly calculated ALU output or the previously stored Data Memory value. This output is fed as the “d” input into the register file. The pipelined value of the output register from Block 2 is fed as the “wn” value into the RegFile. The primary inputs of ma and mb are given by the source and target registers, also sent directly from Block 2. The register file reads the contents of the file at indices ma and mb, and outputs the value stored there as qa and qb. This is changed based on whether the RegWrite signal is high (this is also fed into RegFile). If Write Back is enabled, the signals at wn and d would need to be written into memory.



*Figure 5. Block 5 – Write back stage and circuitry*

Otherwise, to keep in line with forwarding, the information in qa and qb would be sent back to Block 2 as inputs to the forwarding muxes. Other inputs include the ALU output (immediately after calculation), the ALU output after entering the next clock cycle, as well as the value stored in the Data Memory. Depending on whether a hazard arises with RS and RD, or RT and RD, or even whether it happens in the EXE or MEM block, this mux will output the forwarded value to ensure no hazards.



*Figure 6. Visualization and circuitry of the forwarding done, as well as rewiring from each block to the muxes. Control unit inputs are also taken.*

***As a preface, please note there were not enough I/O slots to display all signals in one go.***

Verilog Design Code

`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////

// Company: Pennsylvania State University, University Park

// Engineer: Anand Rajan

//

// Create Date: 04/25/2021 08:09:09 PM

// Design Name:

// Module Name: top

// Project Name: Final Lab

// Target Devices: XC7Z010-CLG400-1

// Tool Versions:

// Description: The project aims to develop a basic CPU through pipelining.

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

module top(input clk, output wwreg\_sig, output [4:0] rs\_sig, output [4:0] rt\_sig, output [4:0] wn\_sig, output [31:0] d\_sig);

// Block 1: IF stage

wire[7:0] adder\_to\_pc; // Output from adder

wire[7:0] pc\_out; // Output from PC register

wire[31:0] im\_to\_ifid; // Output from instruction memory

// Block 2: ID stage

wire[31:0] ifid\_out;

// Outputs from the Control Unit

wire wreg\_to\_idexe;

wire m2reg\_to\_idexe;

wire wmem\_to\_idexe;

wire[3:0] aluc\_to\_idexe;

wire aluimm\_to\_idexe;

wire RegDst;

// Output from Mux

wire [4:0]mux\_to\_idexe;

// Output from Register File

wire [5:0]qa\_to\_fmu;

wire [5:0]qb\_to\_fmu;

// Output from Sign Extender

wire[31:0] eimm\_to\_idexe;

// Block 3: EXE stage (next lab)

wire ewreg;

wire em2reg;

wire ewmem;

wire[3:0] ealuc;

wire ealuimm;

wire [4:0] emuxout;

wire [5:0] eqa;

wire [5:0] eqb;

wire [31:0] eeimm;

wire [31:0] mux\_to\_alu;

wire [31:0] aluout;

wire [31:0] fmu\_to\_idexe;

wire [31:0] fmu2\_to\_idexe;

// Block 4: MEM stage

wire mwreg;

wire mm2reg;

wire mwmem;

wire [4:0] mmuxout;

wire [31:0] maluout;

wire [5:0] mqb;

wire [31:0] dm\_to\_memwb;

// Block 5: WB stage

wire wwreg;

wire wm2reg;

wire [4:0] wmuxout;

wire [31:0] waluout;

wire [31:0] wdo;

wire [31:0] wbmuxout;

wire [1:0] forwardA;

wire [1:0] forwardB;

// Module Instantiations

PC pc(clk, adder\_to\_pc, pc\_out);

Adder adder(pc\_out, adder\_to\_pc);

IM im(pc\_out, im\_to\_ifid);

IFID ifid(clk, im\_to\_ifid, ifid\_out);

CU cu(ifid\_out[31:26], ifid\_out[5:0], ifid\_out[25:21], ifid\_out[20:16],

mmuxout, mm2reg, mwreg, emuxout, em2reg, ewreg,

wreg\_to\_idexe, m2reg\_to\_idexe, wmem\_to\_idexe, aluc\_to\_idexe, aluimm\_to\_idexe, RegDst, forwardA, forwardB);

Mux mux(ifid\_out[15:11], ifid\_out[20:16], RegDst, mux\_to\_idexe);

forwardingMux fmu(qa\_to\_fmu, aluout, maluout, dm\_to\_memwb, forwardA, fmu\_to\_idexe);

forwardingMux fmu2(qb\_to\_fmu, aluout, maluout, dm\_to\_memwb, forwardB, fmu2\_to\_idexe);

Signext ext(ifid\_out[15:0], eimm\_to\_idexe);

IDEXE idexe(clk, wreg\_to\_idexe, m2reg\_to\_idexe, wmem\_to\_idexe, aluc\_to\_idexe,

aluimm\_to\_idexe, mux\_to\_idexe, fmu\_to\_idexe, fmu2\_to\_idexe, eimm\_to\_idexe,

ewreg, em2reg, ewmem, ealuc, ealuimm, emuxout, eqa, eqb, eeimm);

Mux2 mux2(eqb, eeimm, ealuimm, mux\_to\_alu);

ALU alu(eqa, mux\_to\_alu, ealuc, aluout);

EXEMEM exemem(clk, ewreg, em2reg, ewmem, emuxout, aluout, eqb,

mwreg, mm2reg, mwmem, mmuxout, maluout, mqb);

DataMem dm(maluout, mqb, mwmem, dm\_to\_memwb);

MEMWB memwb(clk, mwreg, mm2reg, mmuxout, maluout, dm\_to\_memwb,

wwreg, wm2reg, wmuxout, waluout, wdo);

WBMux wbmux(waluout, wdo, wm2reg, wbmuxout);

Regfile regfile(!clk, ifid\_out[25:21], ifid\_out[20:16], wmuxout, wbmuxout, wwreg, qa\_to\_fmu, qb\_to\_fmu);

// assign mwreg\_sig = mwreg;

// assign mm2reg\_sig = mm2reg;

// assign mmuxout\_sig = mmuxout;

// assign maluout\_sig = maluout;

// assign do\_sig = dm\_to\_memwb;

// assign wwreg\_sig = wwreg;

// assign wm2reg\_sig = wm2reg;

// assign wmuxout\_sig = wmuxout;

// assign waluout\_sig = waluout;

// assign wdo\_sig = wdo;

assign wwreg\_sig = wwreg;

assign rs\_sig = ifid\_out[25:21];

assign rt\_sig = ifid\_out[20:16];

assign wn\_sig = wmuxout;

assign d\_sig = wbmuxout;

endmodule

module PC(input clk, input[7:0] a, output reg[7:0] q);

// Wire to set the value of PC to its next value; PC is a register effectively

always @(posedge clk)

begin

q <= a;

end

initial begin

q = 8'd100;

end

endmodule

module Adder(input [7:0]a, output reg[7:0] q);

// This module is not clock-dependent. It adds the input from PC register to the constant 4 and returns it.

// PC register should take this value to set as new value.

wire[7:0] to\_add = 8'd4;

always @(\*)

begin

q <= (a + to\_add);

end

endmodule

module IM(input[7:0] addr, output reg [31:0] do);

reg [31:0] IM [0:511]; // Array of registers

integer a;

always @(\*)

begin

a = addr; // integer-cast

do = IM[a];

end

initial // Hardcoded based on the examples provided to be done

begin

IM[100] = 32'b10001100001000100000000000000000;

IM[104] = 32'b10001100001000110000000000000100;

IM[108] = 32'b10001100001001000000000000001000;

IM[112] = 32'b10001100001001010000000000001100;

IM[116] = 32'b00000000110000100101000000100000;

end

endmodule

module IFID(input clk, input [31:0]a, output reg[31:0] q);

// This is also clock-dependent. Outputs whatever it's inputted at pos clock edge.

always @(posedge clk)

begin

q <= a;

end

endmodule

module CU(input [5:0] op, input [5:0] func,

input [4:0] rs, input [4:0] rt, input [4:0] mmuxout, input mm2reg, input mwreg, input [4:0] emuxout, input em2reg, input ewreg,

output reg wreg, output reg m2reg, output reg wmem, output reg[3:0] aluc, output reg aluimm, output reg regrt,

output reg [1:0] fwda, output reg [1:0] fwdb);

always @(\*) begin

case(op) // Determination of these values depends on the opcode, and func for R-format

6'b000000: // R-format instruction

begin

wreg = 1'b1;

m2reg = 1'b0;

wmem = 1'b0;

regrt = 1'b1;

aluimm = 1'b0;

case(func) // Using truth table in Zybooks

6'b100000: // add

aluc = 4'b0010;

6'b100010: // subtract

aluc = 4'b0110;

6'b100100: // AND

aluc = 4'b0000;

6'b100101: // OR

aluc = 4'b0001;

6'b100110: // XOR

aluc = 4'b0010;

6'b000000: // shift left

aluc = 4'b0010;

6'b000010: // logical shift right

aluc = 4'b0110;

// Not in truth table

// 6'b000011: // arithmetic shift right

// aluc = 4'b0010;

// 6'b001000: // register jump

// aluc = 4'b0010;

endcase

end

// Commented cases remain to be done in second part of lab - only necessary cases done

// 6'b001000: // addi

// 6'b001100: // andi

// 6'b001101: // ori

// 6'b001110: // xori

6'b100011: // lw

begin

wreg = 1'b1;

m2reg = 1'b1;

wmem = 1'b0;

regrt = 1'b0;

aluimm = 1'b1;

aluc = 4'b0010;

end

6'b101011: // sw

begin

wreg = 1'b0;

m2reg = 1'bX;

wmem = 1'b1;

regrt = 1'bX;

aluimm = 1'b1;

aluc = 4'b0010;

end

// 6'b000100: // beq

// 6'b000101: // bne

// 6'b001111: // lui

// 6'b000010: // j

// 6'b000011: // jal

endcase

// Forwarding Unit

fwda <= 2'b00;

fwdb <= 2'b00;

if ((ewreg == 1) && (emuxout != 0) && (emuxout == rs)) begin // dest reg of exemem is same as source reg of prev instr

fwda <= 2'b01;

end else if ((ewreg == 1) && (emuxout != 0) && (emuxout == rt)) begin // dest reg of exemem is same as target reg of prev instr

fwdb <= 2'b01;

end else if ((mwreg == 1) && (mmuxout != 0) && (mmuxout == rs)) begin // same as above for memwb

if (emuxout != rs) begin

fwda <= 2'b10;

end else if (ewreg == 0) begin

fwda <= 2'b10;

end

end else if ((mwreg == 1) && (mmuxout != 0) && (mmuxout == rt)) begin // same as above for memwb

if (emuxout != rt) begin

fwdb <= 2'b10;

end else if (ewreg == 0) begin

fwdb <= 2'b10;

end

end

// the case of 11 would be in a load word dependency but this never arises in this case

end

endmodule

module Mux(input [4:0]rd, input [4:0]rt, input regrt, output reg [4:0]muxout);

always @(\*) begin

if (regrt) // select rd if RegDst is 1

muxout <= rd;

else // select rt if RegDst is 0

muxout <= rt;

end

endmodule

module Regfile(input clk, input [4:0]rs, input [4:0]rt, input [5:0] wn, input [31:0] d, input we, output reg [5:0]qa, output reg [5:0]qb);

reg [31:0] regfile [0:31];

integer a;

integer b;

integer c;

always @(posedge clk) begin

c = wn;

// if we is 1, write back is enabled

if (we)

regfile[c] <= d;

// qa outputs for rs and qb outputs for rt

a = rs; // this is to integer-cast the binary input

b = rt;

qa <= regfile[a];

qb <= regfile[b];

end

// asked to initialize all values in the regfile to 0

integer i;

initial begin

regfile[0] = 'h00000000;

regfile[1] = 'hA00000AA;

regfile[2] = 'h10000011;

regfile[3] = 'h20000022;

regfile[4] = 'h30000033;

regfile[5] = 'h40000044;

regfile[6] = 'h50000055;

regfile[7] = 'h60000066;

regfile[8] = 'h70000077;

regfile[9] = 'h80000088;

regfile[10] = 'h90000099;

for (i=11; i<32; i=i+1) begin

regfile[i] = 0;

end

end

endmodule

module Signext(input [15:0]imm, output reg [31:0]eimm);

// converts 16 bit imm to 32 bits

always @(\*) begin

eimm[31:0] <= { {16{imm[15]}}, imm[15:0] };

end

endmodule

module IDEXE(input clk, input wreg, input m2reg, input wmem, input [3:0]aluc, input aluimm, input [4:0]muxout, input [5:0]qa, input [5:0]qb, input [31:0]eimm,

output reg ewreg, output reg em2reg, output reg ewmem, output reg [3:0]ealuc, output reg ealuimm, output reg [4:0]emuxout,

output reg [5:0]eqa, output reg [5:0]eqb, output reg[31:0] eeimm);

// Nothing but a clock-dependent output-input setter

always @(posedge clk)

begin

ewreg <= wreg;

em2reg <= m2reg;

ewmem <= wmem;

ealuc <= aluc;

ealuimm <= aluimm;

emuxout <= muxout;

eqa <= qa;

eqb <= qb;

eeimm <= eimm;

end

endmodule

module Mux2(input [5:0] a, input [31:0] b, input sel, output reg [31:0] q);

// Not dependent on clock; chooses 0 or 1 based on ealuimm

always @(\*) begin

if (sel) // select eeimm if RegDst is 1

q <= b;

else // select eqb if RegDst is 0

q <= a;

end

endmodule

module ALU(input [5:0] a, input [31:0] b, input [3:0] aluc, output reg [31:0] r);

// Not dependent on clock; performs different ops based on aluc

always @(\*) begin

case(aluc) // Determination of these values depends on the aluc

4'b0000: // AND

begin

r <= a && b;

end

4'b0001: // OR

begin

r <= a || b;

end

4'b0010: // add

begin

r <= a + b;

end

4'b0010: // subtract

begin

r <= a - b;

end

4'b0010: // set on less than

begin

if (a < b) begin

r <= 32'b1;

end else begin

r <= 32'b0;

end

end

4'b0010: // NOR

begin

r <= ~(a || b);

end

endcase

end

endmodule

module EXEMEM(input clk, input ewreg, input em2reg, input ewmem, input [4:0]emuxout, input [31:0] aluout, input [5:0]eqb,

output reg mwreg, output reg mm2reg, output reg mwmem, output reg [4:0]mmuxout, output reg [31:0] maluout, output reg [5:0] mqb);

// Nothing but a clock-dependent output-input setter

always @(posedge clk)

begin

mwreg <= ewreg;

mm2reg <= em2reg;

mwmem <= ewmem;

mmuxout <= emuxout;

maluout <= aluout;

mqb <= eqb;

end

endmodule

module DataMem(input [31:0] a, input [5:0] di, input we, output reg [31:0] do);

// Not dependent on clock; array

reg [31:0] dmem [0:31];

integer x;

integer y;

always @(\*) begin

if (!we) // if we is 0, read

// do outputs value at address a

x = a; // this is to integer-cast the binary input

//b = rt;

do <= dmem[x];

// if we is 1, write, but this lab doesn't cover that

// input di seems relatively useless - function not specified in TB, lecture, or hints video

end

// asked to initialize all values in the dmem to those provided

integer i;

initial begin

for (i=0; i<32; i=i+1) begin

dmem[i] = 0;

end

dmem[0] = 'hA00000AA;

dmem[1] = 'h10000011;

dmem[2] = 'h20000022;

dmem[3] = 'h30000033;

dmem[4] = 'h40000044;

dmem[5] = 'h50000055;

dmem[6] = 'h60000066;

dmem[7] = 'h70000077;

dmem[8] = 'h80000088;

dmem[9] = 'h90000099;

end

endmodule

module MEMWB(input clk, input mwreg, input mm2reg, input [4:0] mmuxout, input [31:0] maluout, input [31:0] do,

output reg wwreg, output reg wm2reg, output reg [4:0] wmuxout, output reg [31:0] waluout, output reg [31:0] wdo);

// Nothing but a clock-dependent output-input setter

always @(posedge clk)

begin

wwreg <= mwreg;

wm2reg <= mm2reg;

wmuxout <= mmuxout;

waluout <= maluout;

wdo <= do;

end

endmodule

module WBMux(input [31:0] a, input [31:0] b, input sel, output reg [31:0] q);

// Not dependent on clock; chooses 0 or 1 based on selector

always @(\*) begin

if (sel) // select wdo if wm2reg is 1

q <= b;

else // select waluout if wm2reg is 0

q <= a;

end

endmodule

module forwardingMux(input [5:0] a, input [31:0] b, input [31:0] c, input [31:0] d, input [1:0] sel, output reg [31:0] q);

always @(\*) begin

if (sel == 2'b00)

q <= a;

else if (sel == 2'b01)

q <= b;

else if (sel == 2'b10)

q <= c;

else if (sel == 2'b11)

q <= d;

end

endmodule

Note that the device used, as listed in the prefacing comments, is XC7Z010-1CLG400-1, as prescribed by the laboratory instructions. The tradition of providing screenshots of the code for aesthetic and readability purposes will be omitted here due to the size of the top level code. The provision of the testbench will follow this section.

Verilog Test Bench Design Code

//////////////////////////////////////////////////////////////////////////////////

// Company: Pennsylvania State University, University Park

// Engineer: Anand Rajan

//

// Create Date: 04/25/2021 08:09:09 PM

// Design Name: Pipelining CPU

// Module Name: top

// Project Name: Lab 5

// Target Devices: XC7Z010-CLG400-1

// Tool Versions:

// Description: The project aims to develop a basic CPU through pipelining.

//

// Dependencies:

//

// Revision:

// Revision 0.01 - File Created

// Additional Comments:

//

//////////////////////////////////////////////////////////////////////////////////

`timescale 1ns/1ps

module testbench();

reg clk\_tb = 0;

wire [5:0] qa\_sig;

wire [5:0] qb\_sig;

wire mwreg\_sig;

wire mm2reg\_sig;

wire [4:0] mmuxout\_sig;

wire [31:0] maluout\_sig;

wire [31:0] do\_sig;

wire wwreg\_sig;

wire wm2reg\_sig;

wire [4:0] wmuxout\_sig;

wire [31:0] waluout\_sig;

wire [31:0] wdo\_sig;

top dut(clk\_tb, qa\_sig, qb\_sig, mwreg\_sig, mm2reg\_sig, mmuxout\_sig, maluout\_sig, do\_sig, wwreg\_sig, wm2reg\_sig, wmuxout\_sig, waluout\_sig, wdo\_sig); // Initializing an instance of top

always begin

#5;

clk\_tb = ~clk\_tb; // Clock Rule

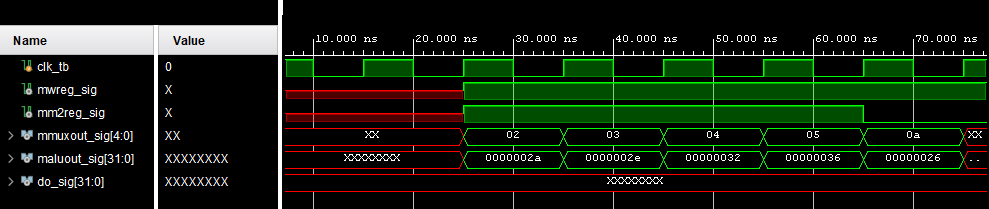
end

endmodule

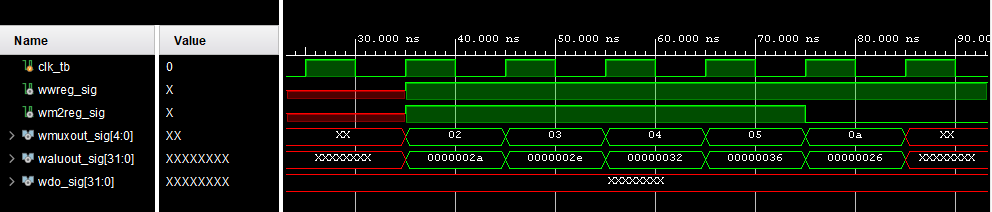
Note that this function really only requires the controlling of the clock through the testbench. These outputs are displayed as outputs for simplicity but the $display keyword could also have been used in place of this.

The corresponding waveforms for the simulation of this design and testbench design code is also provided below the code screenshot. It should be stressed that the signals become X once more after 75/85 ns due to the lack of instructions past the PC’s of 100, 104, 108, 112, 116. Since most of these blocks perform calculations or selections, there is no selection to be made if there is no instruction derived from the IM. As such, they are X’s. The signals that come as outputs from the Control Unit extend forever PURELY BECAUSE OF THE CASE STATEMENT. No default statement was introduced since no such case was presented to account for this, and as such, with the lack of instructions, the control unit continues to output the same values as before infinitely. **THIS IS NOT A BUG. THIS CAN BE CHANGED WITH MORE INSTRUCTIONS OR SPECIFICATION OF A DEFAULT CASE.** All values for each signal are found on the signal and not on the left-hand sidebar. Refer to the signals themselves (those without values are logic signals).

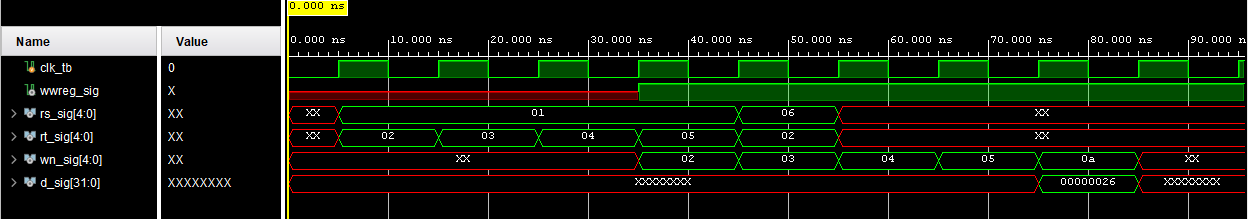
Furthermore, in providing these outputs, it will be seen that the Data Memory’s output is X, and the reason for this is because the input index given by the ALU output falls outside the range of permissible indices for DM values (that is, the first index provided is 42, and the maximum DM index is 31). **THIS IS ALSO NOT A BUG.** If the DM were expanded (though no explicit amount was specified for this), and more values were filled in it, it would be populated, and thus produce an output. However, since it does not in fact ever access any index (since all inputs are higher than 31), this is not clearly seen.

  
*Image 7. Screenshot of Final Waveform (inputs to MEMWB)*

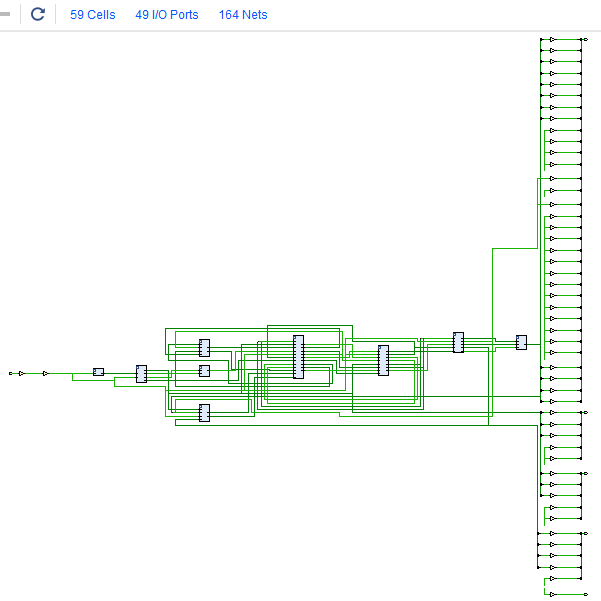
*(values of DM explained as above in the red text)*

 *Image 8. Screenshot of Final Waveform (outputs of MEMWB)*

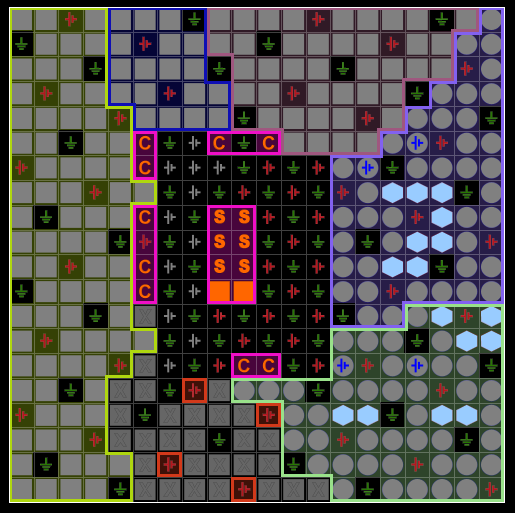
*(values of DMout explained as above in the red text)*



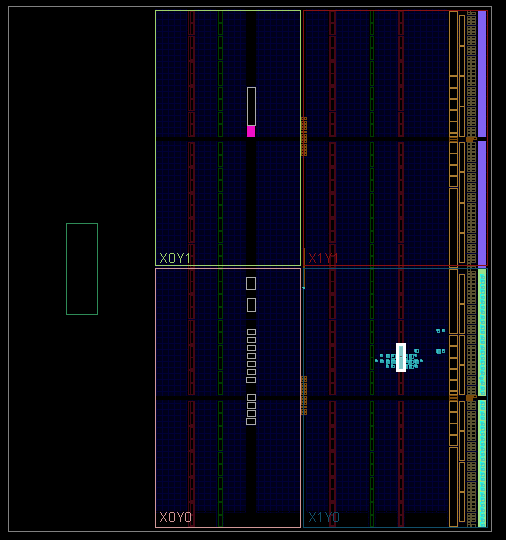
*Image 9. Screenshot of Final Waveform (inputs of RegFile)*



*Image 10. Design Schematics from the Synthesis of the Project (****SYNTHESIZED DESIGN –*** *xc7z010clg400-1)*



*Image 11. Snapshot of the I/O Planning*



*Image 12. Snapshot of the Floor Planning*

Images 10 through 12 provide the design schematic of the synthesized design and snapshots of the I/O and Floor planning of the same. It was difficult to discern which was which due to GUI errors, so the same choices of images were taken as the first lab. Note that the schematic provided could not be zoomed or resized in any way to make it clearer or larger. The information of the schematic is provided just above the design, in the same bar where the number of cells and such are mentioned. Specifics can be conducted within the project itself. The vertical layout was indeterminable though, and as such, cannot be zoomed in on without cutting a large portion out (or without having to post 20 or so snapshots).

With this, the requirements specified by the lab’s instructions are concluded. To recap, the specified device was used in this lab, which was done in Vivado. The Verilog design code as well as test bench code were both provided (albeit not with the 2x line spacing condition in both cases, but with 1x in the larger one in order to fit the code into as compact a space as possible – which still ended up being large). The code accounted for all blocks as well as implementation of forwarding of the basic CPU requested. Snapshots of the same were also provided in order to supplement readability and aesthetics such as indentations and comments. The waveform resulting from the verification of my design using the simulation software were also provided, with all the specified signals shown. The waveforms cover a timing period from 0 till 85 ns, since this is all the time necessary for the instructions to have come to pass. Additionally, the design was synthesized and the design schematics of this as well as the I/O and Floor planning designs were provided in snapshots (images 10 through 12). The cover page requirements were met to the best of my ability, and this project will be uploaded as a Word file (with no use of LaTex).