CPE 166 Advance Logic Design

Lab Section 2

Lab 4

Shammah Thao

12/1/20

Table of Contents

Introduction ..............................................................................................................................3

Part 1: SRAM Design…………………....................................................................................3

Design Purpose .........................................................................................................................3

Verilog Data: .............................................................................................................................3

Verilog Code..............................................................................................................................4

Results Discussion .....................................................................................................................6

Part 2,3,4: Simplified Microprocessor Design…………...........................................................6

Design Purpose ..........................................................................................................................6

Verilog Data: ..............................................................................................................................8

Verilog Code..............................................................................................................................10

Results Discussion ....................................................................................................................17

Conclusion................................................................................................................................ 17

**Introduction:**

In this 4-part lab we were to design a SRAM design that reads and write data and a simplified Microprocessor. A SRAM is a type of random-access memory that uses latching circuitry to store each bit. It loses it data every time the power is lose making it faster than dram. A microprocessor, on the other hand, is a computer processor that is implemented on a single integrated circuit of MOSFET construction. The microprocessor is a multipurpose, clock-driven, register-based, digital integrated circuit that accepts binary data as input, processes it according to instructions stored in its memory, and provides results (also in binary form) as output. Microprocessors contain both combinational logic and sequential digital logic. Microprocessors operate on numbers and symbols represented in the binary number system.

**Part 1: SRAM**

Design Purpose:

The purpose of this part of the lab is to write in Verilog a sram that could read and write in 4 bits in a 32-address location. The coding would include 2 parts, the ram and the memory which was given to us and we would have to compete.

Verilog Design:

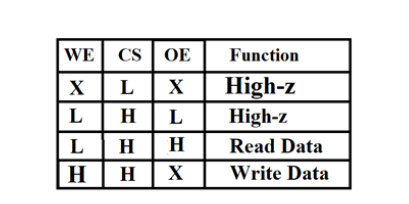


Figure 1: SRAM Function Table

Verilog Code:

Mem\_fsm

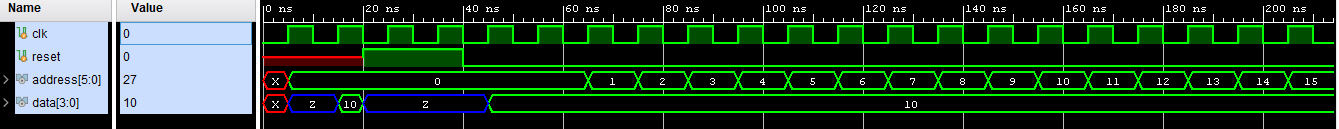
|  |  |
| --- | --- |
| Source Code | Testbench |
| `timescale 1ns / 1ps  module mem\_fsm(clk,reset,address,data,cs,we,oe  );input clk, reset;  output [3:0] address;  inout [3:0] data;  output cs, we, oe;  reg cs, we, oe;  reg [5:0] address;  reg [3:0] data\_reg;  reg [2:0] state;  parameter idle = 3’b000, s1= 3'b001, s2= 3'b010, s3=3'b011;  assign data = data\_reg;  always@(posedge clk or posedge reset)  begin  if (reset)  begin  state <= idle;  address <= 0;  end  else  case (state)  idle: begin  state <= s1;  address <= 0;  end  s1: begin  state <= s2;  address <= 0;  end  s2: begin  if(address == 32) begin  state <= s3;  end  else begin  address <= address + 1;  state <= s2;  end  end  s3: begin  if(address == 0) begin  state <= s1;  end  else begin  address <= address - 1;  state <= s3;  end  end  default: begin  state <= idle;  address <= 0;  end  endcase  end  always@(state)  begin  case (state)  idle: begin  cs = 0;  we = 0;  oe = 0;  data\_reg = 4'bZZZZ;  end  s1: begin  cs = 1;  we = 1;  oe = 0;  data\_reg = 4'b1010;  end  s2: begin  cs = 1;  we = 1;  oe = 0;  data\_reg = 4'b1010;  end  s3: begin  cs = 1;  we = 0;  oe = 1;  end  default: begin  cs = 0;  we = 0;  oe = 0;  data\_reg = 4'b1010;  end  endcase  end  endmodule |  |

Ram

|  |  |
| --- | --- |
| Source Code | Test bench |
| module SRAM (address, data, cs, we, oe);  input cs, we, oe;  input [5:0] address;  inout [7:0] data;  reg [7:0] data\_out;  reg [7:0] mem [0:1023];  assign data = (cs && oe && !we) ? data\_out : 8'bzzzzzzzz;  always @ (cs or we or data or address)  if (cs && we)  mem [address] = data;  always @ (cs or we or data or address or data)  begin  if (cs && !we && oe)  data\_out = mem[address];  end  endmodule |  |

Top

|  |  |
| --- | --- |
| Source Code | Testbench |
| module top( clk, reset, address, data, cs, we, oe );  input clk, reset, cs, we, oe;  output [7:0] address;  inout [3:0] data;  mem\_fsm g1(.clk(clk), .reset(reset), .address(address), .data(data), .cs(cs), .we(we), .oe(oe));  SRAM g2(.address(address), .data(data), .cs(cs), .we(we), .oe(oe));  endmodule | module top\_tb();  reg clk, reset;  wire [5:0] address;  wire [3:0] data;  top uut(.clk(clk),.reset(reset),.address(address),.data(data));  always begin  #5 clk = ~clk;  end  initial begin  clk = 0;  #20 reset = 1;  #20 reset = 0;  #1000 $stop;  end  endmodule |



Result Discussion:

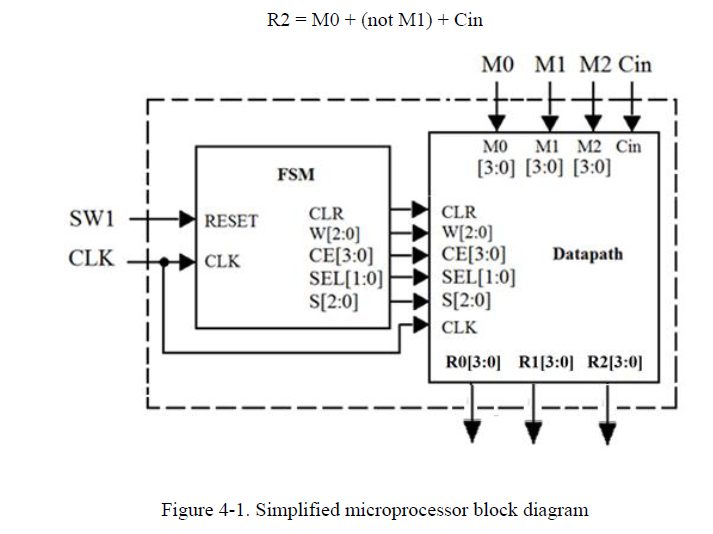
The result came out as intended. Getting the solution was easier than expected. I assumed we were supposed to add more states to the mem\_fsm which made it more complicated than it was. The shorter the code was the easier it was since it was just a simply fill in the blank. It was a good reminder of the Verilog coding along with fsm.

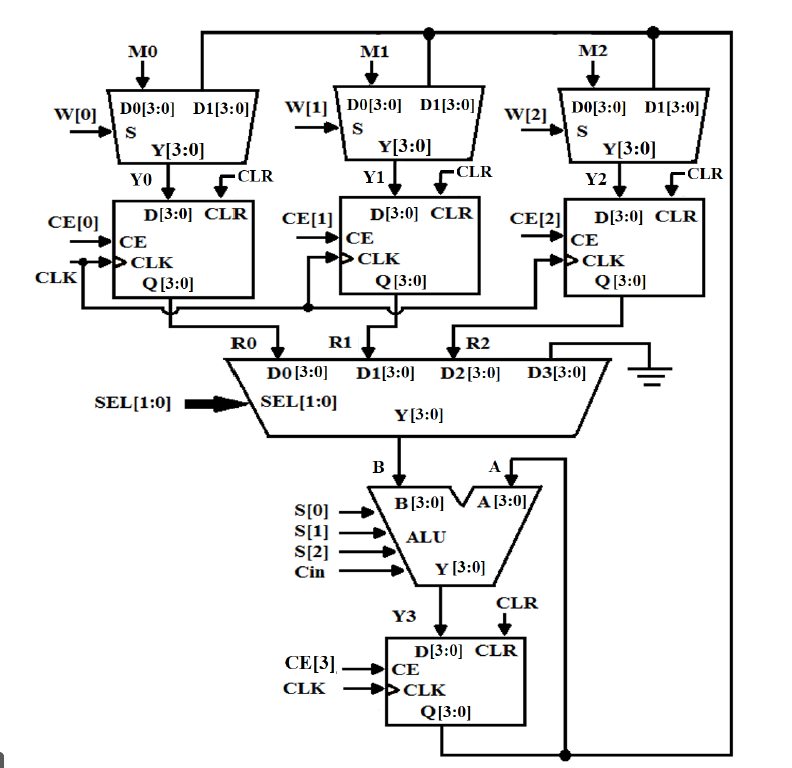
**Part 2,3,4: Simplified Microprocessor Design**

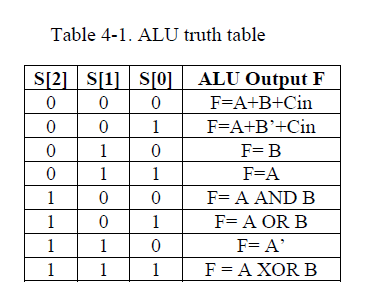
Design Purpose:

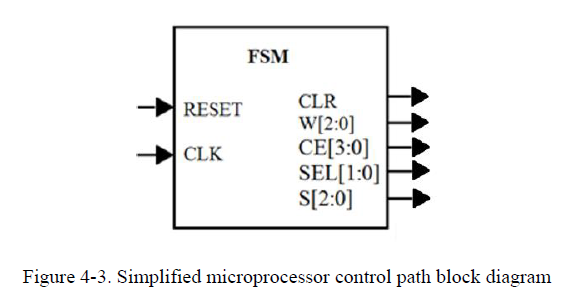
The purpose of this was to make a simplified microprocessor diagram with two input into a fsm which leads to a datapath that accepts an additional 4 inputs, giving us 3 outputs in arrays. There was about 3 part to this lab which are required to be join together in a hierarchy format.

Verilog Design





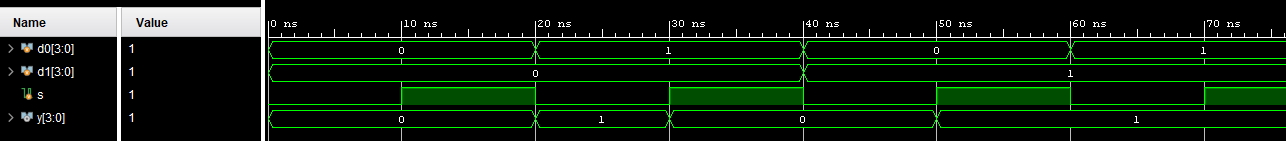




Verilog Code:

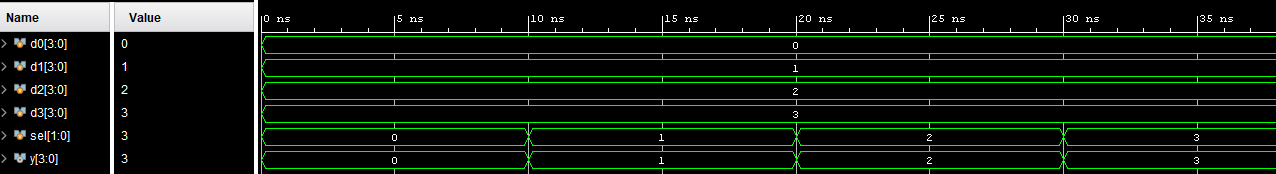
Mux 2 to 1

|  |  |
| --- | --- |
| Source Code | Testbench |
| module mux2to1(d0, d1, s, y);  input[3:0] d0, d1;  input s;  output reg[3:0] y;  always@(d0 or d1 or s) begin  if(s) y = d1;  else y = d0;  end  endmodule | module mux2to1\_tb;  reg[3:0] d0, d1;  reg s;  wire[3:0] y;  mux2to1 u1(d0, d1, s, y);  initial begin  d1 = 0; d0 = 0; s = 0;  #10 d1 = 0; d0 = 0; s = 1;  #10 d1 = 0; d0 = 1; s = 0;  #10 d1 = 0; d0 = 1; s = 1;  #10 d1 = 1; d0 = 0; s = 0;  #10 d1 = 1; d0 = 0; s = 1;  #10 d1 = 1; d0 = 1; s = 0;  #10 d1 = 1; d0 = 1; s = 1;  #10 $stop;  end  endmodule |



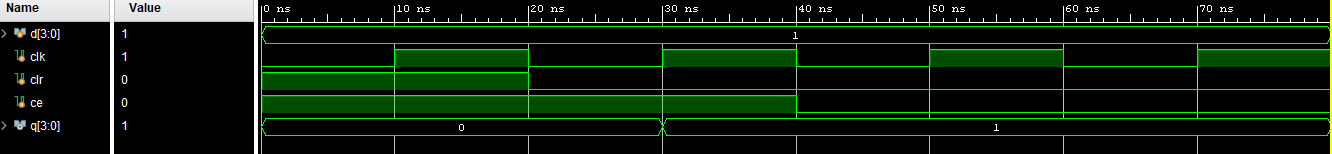
Mux4to1

|  |  |
| --- | --- |
| Source Code | Testbench |
| module mux4to1(d0, d1, d2, d3, sel, y);  input[3:0] d0,d1,d2,d3;  input[1:0] sel;  output reg[3:0] y;  always @(d0 or d1 or d2 or d3 or sel) begin  case (sel)  2'b00: y = d0;  2'b01: y = d1;  2'b10: y = d2;  2'b11: y = d3;  endcase  end  endmodule | module mux4to1\_tb;  reg[3:0] d0, d1, d2, d3;  reg[1:0] sel;  wire[3:0] y;  mux4to1 u1(d0, d1, d2, d3, sel, y);  initial begin  d0 = 0; d1 = 1; d2 = 2; d3 = 3; sel = 2'b00;  #10 d0 = 0; d1 = 1; d2 = 2; d3 = 3; sel = 2'b01;  #10 d0 = 0; d1 = 1; d2 = 2; d3 = 3; sel = 2'b10;  #10 d0 = 0; d1 = 1; d2 = 2; d3 = 3; sel = 2'b11;  #10 $stop;  end  endmodule |



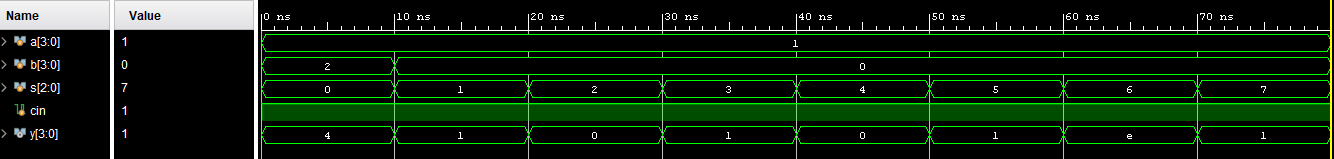
Diff

|  |  |
| --- | --- |
| Source Code | Testbench |
| module diff (clk, clr, d, q, ce);  input[3:0] d;  input clk, ce, clr;  output reg[3:0] q;  always@(posedge clr or posedge clk)  begin  if(clr) q <= 0;  else if(ce) q <= d;  end  endmodule | module diff\_tb;  reg[3:0] d;  reg clk, clr, ce;  wire[3:0] q;  diff u1(clk, clr, d, q, ce);  initial clk = 0;  always #10 clk = ~clk;  initial begin  clr = 1; d = 1; ce = 1;  #20 clr = 0;  #20 ce = 0;  #40 $stop;  end  endmodule |



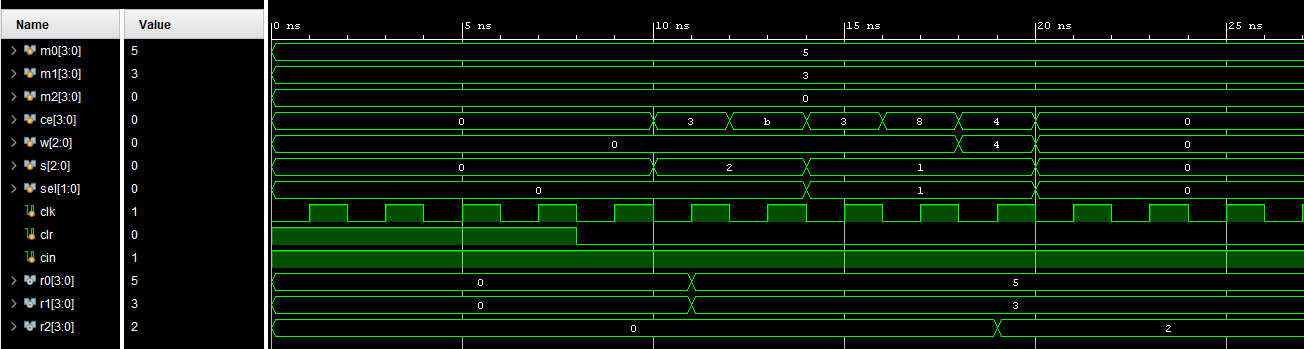
ALU

|  |  |
| --- | --- |
| Source Code | Testbench |
| module alu(a, b, s, cin, y);  input[3:0] a, b;  input[2:0] s;  input cin;  output reg[3:0] y;  always@(a or b or s or cin or y) begin  case(s)  3'b000: y = a + b + cin;  3'b001: y = a + ~b + cin;  3'b010: y = b;  3'b011: y = a;  3'b100: y = a & b;  3'b101: y = a | b;  3'b110: y = ~a;  3'b111: y = a ^ b;  endcase  end  endmodule | module alu\_tb;  reg[3:0] a, b;  reg[2:0] s;  reg cin;  wire[3:0] y;  alu u1(a, b, s, cin, y);  initial begin  s = 0; a = 1; b = 2; cin = 1;  #10 s = 1; a = 1; b = 0; cin = 1;  #10 s = 2; a = 1; b = 0; cin = 1;  #10 s = 3; a = 1; b = 0; cin = 1;  #10 s = 4; a = 1; b = 0; cin = 1;  #10 s = 5; a = 1; b = 0; cin = 1;  #10 s = 6; a = 1; b = 0; cin = 1;  #10 s = 7; a = 1; b = 0; cin = 1;  #10 $stop;  end  endmodule |



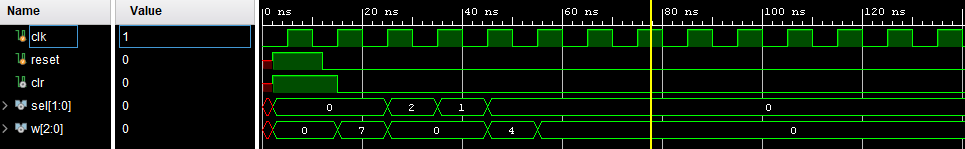
Datapath

|  |  |
| --- | --- |
| Source Code | Testbench |
| module dp(m0, m1, m2, cin, clr, w, ce, sel, s, clk, r0, r1, r2);  input clk, clr, cin;  input[3:0] m0,m1,m2,ce;  input[2:0] w, s;  input[1:0] sel;  output[3:0] r0, r1, r2;  wire [3:0] y0, y1, y2, y3, a, b;  mux2to1 g1(.d0(m0),.d1(a), .s(w[0]), .y(y0));  mux2to1 g2(.d0(m1),.d1(a), .s(w[1]), .y(y1));  mux2to1 g3(.d0(m2),.d1(a), .s(w[2]), .y(y2));  diff d1(.clk(clk), .clr(clr), .ce(ce[0]),.d(y0),.q(r0));  diff d2(.clk(clk), .clr(clr), .ce(ce[1]),.d(y1),.q(r1));  diff d3(.clk(clk), .clr(clr), .ce(ce[2]),.d(y2),.q(r2));  diff d4(.clk(clk), .clr(clr), .ce(ce[3]),.d(y3),.q(a));  mux4to1 g4(.d0(r0),.d1(r1),.d2(r2),.d3(4'b0000),.sel(sel),.y(b));  alu a1(.a(a),.b(b),.s(s),.cin(cin),.y(y3));  endmodule | module dp\_tb;  reg[3:0] m0, m1, m2, ce;  reg[2:0] w, s;  reg[1:0] sel;  reg clk, clr, cin;  wire [3:0] r0, r1, r2;  dp uut(m0, m1, m2, cin, clr, w, ce, sel, s, clk, r0, r1, r2);  initial begin  clr = 1'b1; w = 3'b000; ce = 4'b0000; sel = 2'b00; s = 3'b000; clk = 1'b0; m2 = 4'b0000;  m0 = 4'b0101; m1 = 4'b0011;  cin = 1;  end  always #1 clk = ~clk;  initial begin  #8; clr = 1'b0;  #2; ce = 4'b0011; sel = 2'b00; s = 3'b010;  #2; ce = 4'b1011;  #2; ce = 4'b0011; sel = 2'b01; s = 3'b001;  #2; ce = 4'b1000;  #2; ce = 4'b0100; w = 3'b100;  #2; w = 3'b000; ce = 4'b0000; sel = 2'b00; s = 3'b000;  #8; $stop;  end  endmodule |



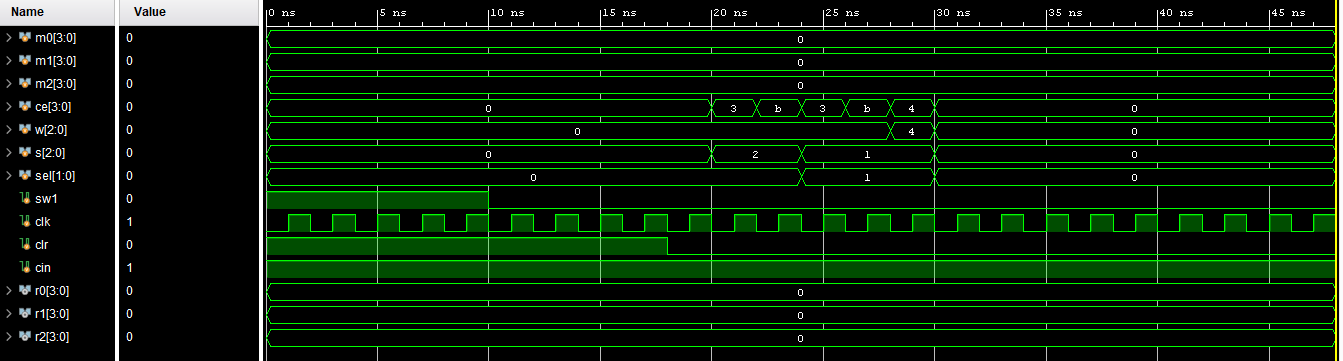
FSM

|  |  |
| --- | --- |
| Source Code | Testbench |
| module fsm(clk, reset, clr, w, ce, sel, s);  input clk, reset;  output reg clr;  output reg [2:0] w, s;  output reg [1:0] sel;  output reg [3:0] ce;  reg [2:0] cs, ns;  parameter s0=0, s1=1, s2=2, s3=3, s4=4, s5=5;  always@(posedge clk or posedge reset) begin  if( reset ) begin  cs <= s0;  end else begin  cs <= ns;  end  end  always@(cs) begin  case(cs)  s0 : begin  clr = 1'b1;  w = 3'b000;  s = 3'b000;  sel = 2'b00;  ce = 4'b0000;  ns <= s1;  end  s1 : begin  clr = 1'b0;  w = 3'b101;  s = 3'b000;  sel = 2'b00;  ce = 4'b0111;  ns <= s2;  end  s2 : begin  clr = 1'b0;  w = 3'b100;  s = 3'b010;  sel = 2'b11;  ce = 4'b1000;  ns <= s3;  end  s3 : begin  clr = 1'b0;  w = 3'b100;  s = 3'b001;  sel = 2'b10;  ce = 4'b1000;  ns <= s4;  end  s4 : begin  clr = 1'b0;  w = 3'b000;  s = 3'b000;  sel = 2'b00;  ce = 4'b0100;  ns <= s5;  end  s5 : begin  clr = 1'b0;  w = 3'b000;  s = 3'b000;  sel = 2'b00;  ce = 4'b0000;  ns <= s5;  end  default : ns = s0;  endcase  end  endmodule | module fsm\_tb;  reg clk, reset;  wire clr;  wire [1:0] sel;  wire [2:0] w, s;  wire [3:0] ce;  fsm uut(clk, reset, clr, ce, w, s, sel);  always  begin  #5 clk = ~clk;  end  initial  begin  clk = 0;  #2 reset = 1;  #10 reset = 0;  #150  $stop;  end  endmodule |



Top

|  |  |
| --- | --- |
| Source Code | Testbench |
| module top(sw1, clk, m0, m1, m2, cin, r0, r1, r2);  input clk, sw1, cin;  input [3:0] m0, m1, m2;  output[3:0] r0,r1,r2;  wire clr;  wire [1:0] sel;  wire [2:0] w,s;  wire [3:0] ce, r0,r1,r2;  fsm g1(.reset(sw1), .clk(clk), .clr(clr), .w(w), .ce(ce), .sel(sel), .s(s));  dp g2(.m0(m0), .m1(m1), .m2(m2), .cin(cin), .clr(clr), .w(w), .ce(ce), .sel(sel), .s(s),.clk(clk), .r0(r0), .r1(r1), .r2(r2));  endmodule | module top\_tb;  reg[3:0] m0,m1,m2,ce;  reg[2:0] w,s;  reg[1:0] sel;  reg sw1, clk, clr, cin;  wire [3:0] r0,r1,r2;  top utt(sw1, clk, m0, m1, m2, cin, r0, r1, r2);  initial  begin  sw1= 1;  clr =1'b1; w= 3'b000; ce= 4'b0000; sel=2'b00; s = 3'b000; clk = 1'b0; m2 = 4'b0000;  m0=4'b0000; m1= 4'b0000;  cin = 1;  end  always #1 clk =~clk;  initial begin  #10; sw1 = 0;  #8; clr = 1'b0;  #2; ce = 4'b0011; sel = 2'b00; s= 3'b010;  #2; ce = 4'b1011;  #2; ce = 4'b0011; sel = 2'b01; s= 3'b001;  #2; ce = 4'b1011;  #2; ce = 4'b0100; w= 3'b100;  #2; w= 3'b000; ce = 4'b0000; sel =2'b00; s = 3'b00;  #10;  #8; $stop;  end  endmodule |



Result Discussion:

The result came out as it should. There was difficulty combining all the coding together since it has been a while since working with Verilog but looking back at old project does help a lot. The only issue I had was logic issue and sometime there might be a mistake in a random while that makes it so the whole file does not work. The testbench at the end also confused me a bit since I overthink to the point where I was confused on what we are testing, but at the end the result came out just fine.

**Conclusion:**

This part of the lab does sum everything that we learn up till now into one project. This coding requires me to think about the very beginning of the course which was a great thing since I could relearn what was already taught. Each part target specific thing such a fsm, state machine coding, and hierarchy design. Although it was quite difficult to jump back into coding that we have done in a while. I am more familiar with verilog coding compared to HDML coding so I would prefer that over the other. Overall, this lab was helpful in being a reminder of how to code hierarchy along with fsm and state machine.