Lab Report 2

Ethan Wyrick, Jack Landers
ELEN 122

```
module l2_SM(input clk,
                        input execute,
                        //input [3:0] input_data (part of instruction),
                        //input [1:0] regXaddr (part of instruction),
                        //input [1:0] regyaddr (part of instruction),
                        input [1:0] operation, // opcode (part of instruction)
                        output reg _Extern,
                        output reg Gout,
                        output reg Ain,
                        output reg Gin,
                        //output reg [1:0] AddrX,
                        //output reg [1:0] AddrY,
                        output reg RdX,
                        output reg RdY,
                        output reg WrX,
                        output reg add_sub,
                        output [3:0] cur_state);

    //defining all my states - 8 total
    parameter IDLE          = 4'b0000;
    parameter LOAD          = 4'b0001;
    parameter READ_Y    = 4'b0010;
    parameter READ_X    = 4'b0011;
    parameter ADD           = 4'b0100;
    parameter SUB           = 4'b0101;
    parameter MV            = 4'b0110;
    parameter WRITE_X   = 4'b0111;
    parameter DONE          = 4'b1000;

    reg [3:0] state = IDLE; // initial state being IDLE

    assign cur_state = state;

    initial begin //instead of reset
  state <= IDLE;
    end
```

```
/* output state logic
input: state
output: 8 control signals
TODO: you need to complete the output logic in the following always statement
*/
always@(*)
begin
  case(state)
    IDLE:
      begin
        _Extern = 1'b0;
        Gout = 1'b0;
        Ain = 1'b0;
        Gin = 1'b0;
        RdX = 1'b0;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
      end
    LOAD:
      begin
    _Extern = 1'b1;
          Gout = 1'b0;
          Ain = 1'b0;
          Gin = 1'b0;
          RdX = 1'b0;
          RdY = 1'b0;
          WrX = 1'b1;
          add_sub = 1'b0;
      end
    READ_Y:
      begin
                // fill in your code here
                    _Extern = 1'b0;
          Gout = 1'b0;
          Ain = 1'b1;
          Gin = 1'b0;
          RdX = 1'b0;
          RdY = 1'b1;
          WrX = 1'b0;
          add_sub = 1'b0;
      end
    READ_X:
      begin
```

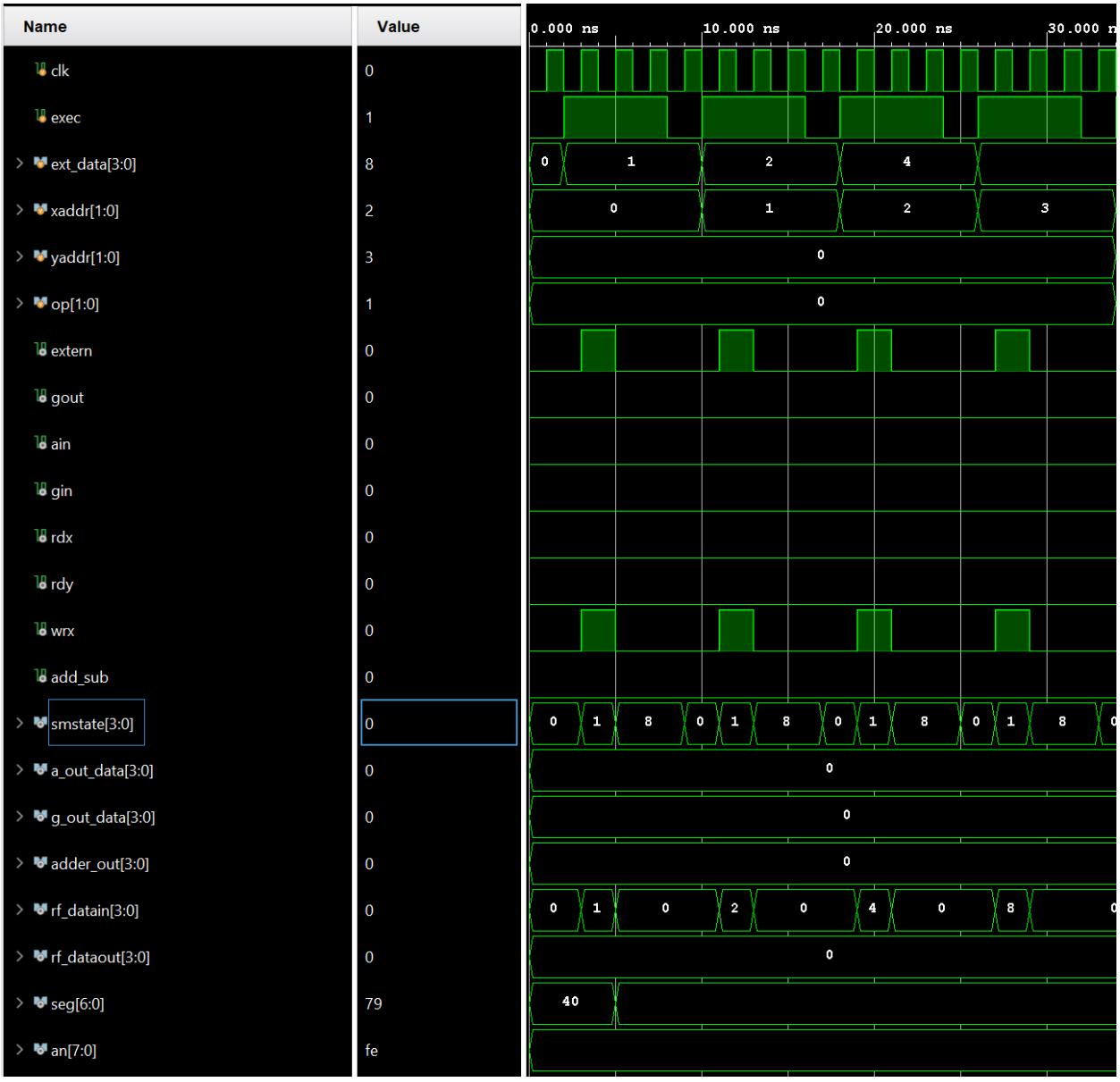
```
            // fill in your code here
                _Extern = 1'b0;
        Gout = 1'b0;
        Ain = 1'b1;
        Gin = 1'b0;
        RdX = 1'b1;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
    end
ADD:
    begin
            // fill in your code ADD
                _Extern = 1'b0;
        Gout = 1'b0;
        Ain = 1'b0;
        Gin = 1'b1;
        RdX = 1'b1;
        RdY = 1'b0;
        WrX = 1'b0;
        add_sub = 1'b0;
    end
SUB:
    begin
            // fill in your code here
                _Extern = 1'b0;
        Gout = 1'b0;
        Ain = 1'b0;
        Gin = 1'b1;
        RdX = 1'b0;
        RdY = 1'b1;
        WrX = 1'b0;
        add_sub = 1'b1;
    end
MV:
    begin
            // fill in your code here
        _Extern = 1'b0;
        Gout = 1'b0;
        Ain = 1'b0;
        Gin = 1'b1;
        RdX = 1'b0;
        RdY = 1'b0;
        WrX = 1'b0;
```

```
                add_sub = 1'b0;
            end
        WRITE_X:
            begin
                    // fill in your code here
                _Extern = 1'b0;
                Gout = 1'b1;
                Ain = 1'b0;
                Gin = 1'b0;
                RdX = 1'b0;
                RdY = 1'b0;
                WrX = 1'b1;
                add_sub = 1'b0;
            end
        DONE:
            begin
                    // fill in your code here
                        _Extern = 1'b0;
                Gout = 1'b0;
                Ain = 1'b0;
                Gin = 1'b0;
                RdX = 1'b0;
                RdY = 1'b0;
                WrX = 1'b0;
                add_sub = 1'b0;
            end
      endcase
    end

    /* Next state logic combined with flip-flops
    input: clk, execute (stop/go signal), operation (opcode)
    output: state
    TODO: you need to complete this Next state logic (combined with flip-flops) in the
following always statement
    */
    always@(posedge clk)
    begin

            case(state)
                    IDLE: begin
                                    if(execute == 1 && operation == 0) state <= LOAD;
                                    if(execute == 1 && operation == 1) state <= READ_Y;
                                    if(execute == 1 && operation == 3) state <= READ_Y;
                                    if(execute == 1 && operation == 2) state <= READ_X;
```

```
                end

                LOAD: begin
                        state <= DONE; // always go to the DONE state at the next
clock tick
                    end

                READ_Y: begin
                        // fill in your code here
            if(operation == 3)  state <= ADD;
            if(operation == 1)  state <= MV;
                    end

                READ_X: begin
                        // fill in your code here
                        state <= SUB;
                    end

                ADD: begin
                        // fill in your code here
            state <= WRITE_X;
                end

                SUB: begin
                        // fill in your code here
            state <= WRITE_X;
                    end

                MV: begin
                        // fill in your code here
            state <= WRITE_X;
                    end

                WRITE_X: begin
                        // fill in your code here
            state <= DONE;
                    end

                DONE: begin

                        //back to idle if execute back to low
                        if(execute == 0) state <= IDLE;

                    end
```

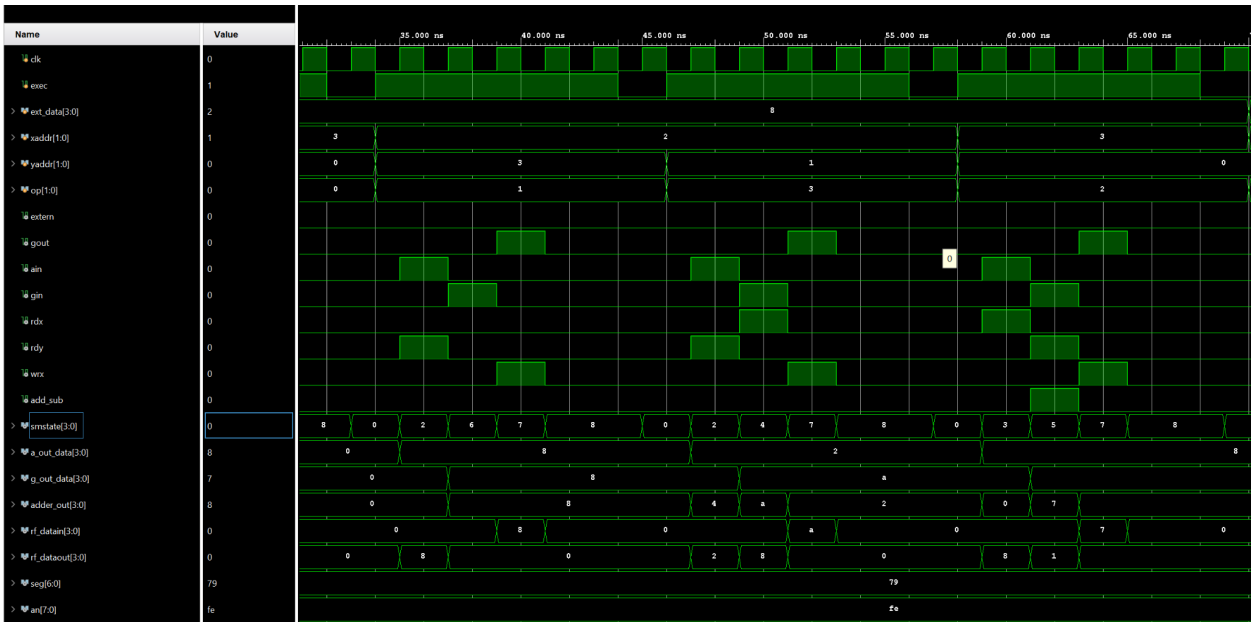
```
                        default: state <= IDLE;

                endcase

        end //end always


endmodule

/*

encodings
00 - load
01 - move
10 - subtract
11 - add

*/
```

Load Instructions:



In this section of the waveform, the state machine enters states 1(Load) and 8(Done), which load external data and write it to X and then signal that the operation is done

MV,ADD, and SUB:



In the first third of this waveform, the machine enters State 2(ReadY), which loads Y into latch A, and then state 6(Move), which passes it through the ALU, and finally state 7(WriteX), which deposits the value in X. In the second third, the machine Reads Y (2), and then enters state 4(Add), which loads X and adds it to Y within the ALU, and then writes this value to X(7). In the final state, we load X into latch A (State 3), and then load Y and subtract it from X (State 5), and then write to X (7).

In this lab, we were somewhat hindered by incorrect assertions of control signals from states, which we promptly fixed.Because of this, we were ready for our demonstration with the TA.

If this design were incapable of accessing 0 through deasserting both RdX and RdY, we could first load a number into Latch A, and then load it again in a SUB instruction, and then write this value to an address we don't need within X [(N-N) = 0 -> X]. Then, we could load the value we wish to move from Y into latch A, then add it to the 0 from X, and then write this value to X.