Gabriel Emerson
ELEC 4200 – Lab 10
11/4/21

Lab 10: Sequential System Design Using ASM Charts

Goals:

The goal for this lab is to design and create a sequential system using ASM charts and methods. ASM is a new type of state machine that differs from the previous finite state machines that we have been using for the past few weeks. This is important because these machines are frequently used to make underlying systems in real world machines. This also helps the mindset of designing a machine based on a previously written chart or diagram.

Design Process:

Task 1 and 2 are the same program but doing two different things. The program is to design a 3 bit x 3 bit binary multiplier. This will use an accumulator, multiplier register, counter, and shifter. Task 3 will do a similar thing, but instead we will create a 4 bit x 4 bit multiplier using a ROM file. The first thing to do is create an ASM chart and then a block diagram. Then since we create a testbench for the first task, we can compare it to the simulation given in the lab writeup. An example chart and diagram are shown in Figure 1 and 2, then the given simulation is shown in Figure 3.

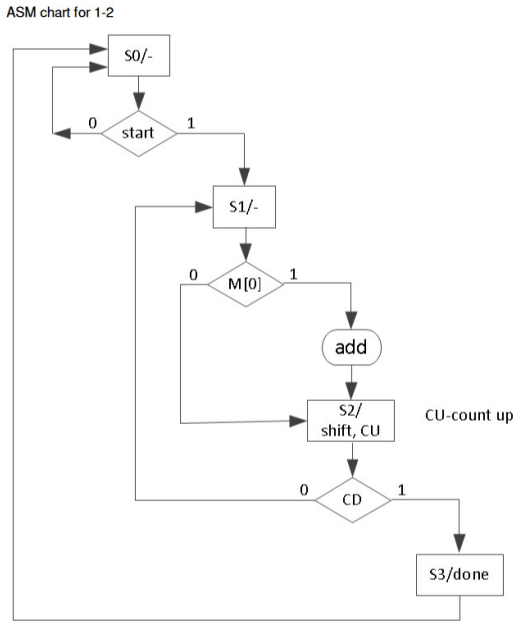

Figure 1

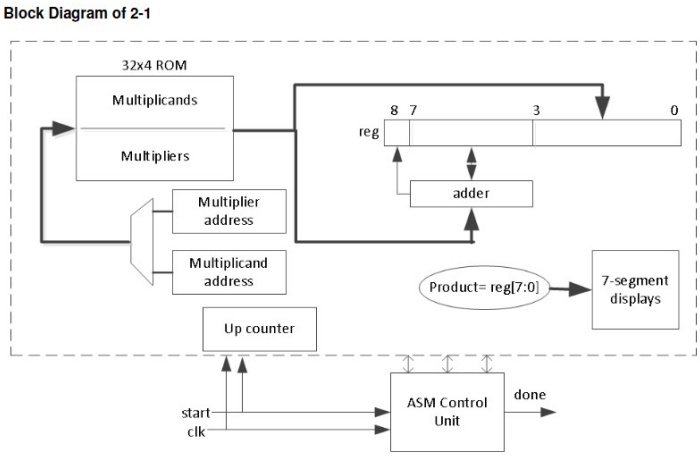

Figure 2

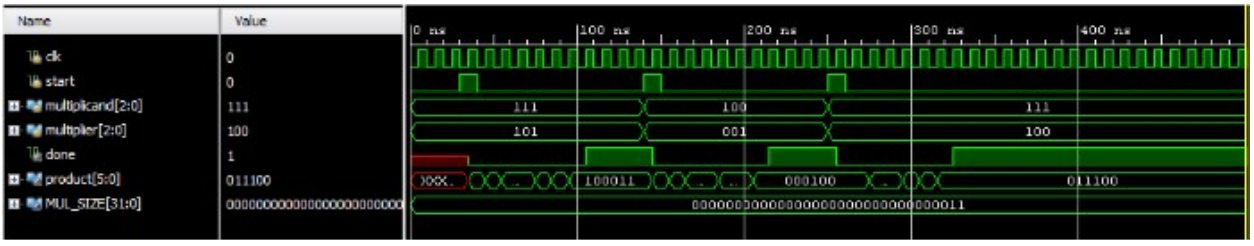

Figure 3

Detailed Design:

Since Task 1 and 2 are the same program, the design should be the same. However, when trying to design this myself, I could accomplish task 1, but not task 2. The code I designed in shown in Figure 4. However, since I could not get the second task to work, the code given that does work in shown in Figure 5. And since we simulated task 1, the testbench used for that simulation is shown in Figure 6.

```
module Lab10_1(multiplier, multiplicand, Clk, load, product, done, state);
    input [2:0] multiplier, multiplicand;
    input  Clk, load;
    output [5:0] product;
    output done;
    output [2:0] state;

    wire [5:0] sum; wire [ 4:0] post; wire carry; wire shift; wire add; wire count_done; wire lsb; wire count_up; wire bit_0; wire bit_1; wire bit_2;
    wire [2:0] plier;
    wire [2:0] plier_reg;
    reg [2:0] count;
    reg [6:0] accu;

    controller c1(.Clk(Clk), .load(load), .count_done(count_done), .lsb(lsb), .shift(shift), .add(add), .done(done), .bit_0(bit_0), .bit_1(bit_1), .bit_2(bit_2), .state(state));
    data_processor d1(.Clk(Clk), .load(load), .multiplicand(multiplicand),
     .multiplier(multiplier), .shift(shift), .add(add), .done(done), .lsb(lsb), .product(product),
      .count_done(count_done), .bit_0(bit_0), .bit_1(bit_1), .bit_2(bit_2));


endmodule


module controller(Clk, load, count_done, lsb, shift, add, done,bit_0,bit_1,bit_2,state);
    input lsb, load, count_done, Clk;
    output  shift, add, done, bit_0, bit_1, bit_2;
    reg done, add=0, shift=0, bit_0=0, bit_1=0, bit_2=0;
    //reg [2:0] state=3'b000, nextstate, oldstate=3'b000;
    output reg [2:0] state = 3'b000;
    reg [2:0] nextstate, oldstate=3'b000;
    parameter idle_state = 3'b000, load_state = 3'b001, bit_0_state = 3'b010, bit_1_state = 3'b011, bit_2_state = 3'b100, add_state = 3'b101, shift_state = 3'b110;
    parameter  finished_state = 3'b111;
    always @(posedge Clk)
    begin
    state <= nextstate;
    end
```

```
    always @(state or load or lsb)
    begin
        case(state)
            idle_state: begin
                            if (load==1)
                            begin
                                nextstate = bit_0_state;
                            end
                            else if(load==0)
                            begin
                                nextstate = idle_state;
                            end
                        end
            //load_state:  begin nextstate = bit_0_state; end
            bit_0_state: begin
                            if (lsb==1)
                            begin
                                nextstate = add_state;
                                oldstate = bit_0_state;
                            end
                            else if(lsb==0)
                            begin
                                nextstate = shift_state;
                                oldstate = bit_0_state;
                            end
                        end
            bit_1_state: begin
                            if (lsb==1)
                            begin
                                nextstate = add_state;
                                oldstate = bit_1_state;
                            end
                            else if(lsb==0)
                            begin
                                nextstate = shift_state;
                                oldstate = bit_1_state;
                            end
                        end
            bit_2_state: begin
                        //nextstate = add_state ;

                            if (lsb==1)
                            begin
                                nextstate = add_state;
                                oldstate = bit_2_state;
                            end
                            else if(lsb==0)
                            begin
                                nextstate = shift_state;
                                oldstate = bit_2_state;
                            end

                        end
            add_state: begin nextstate = shift_state; end
            shift_state: begin
                            if (oldstate == bit_0_state)
                            begin
                                nextstate = bit_1_state;
                            end
                            else if (oldstate == bit_1_state)
                            begin
                                nextstate = bit_2_state;
                            end
                            else if (oldstate == bit_2_state)
                            begin
                                nextstate = finished_state;
                            end
                        end
            finished_state: begin nextstate = idle_state; end
              default: nextstate = idle_state;
        endcase
    end

    always @(state)
    begin
        case(state)
            idle_state: begin
                            if(load==1)
                            begin
                                done=0; add=0; shift=0; bit_0=0; bit_1=0; bit_2=0;
                            end
                        end
            /* load_state: begin
                            done=0; add=0; shift=0; bit_0=0; bit_1=0; bit_2=0;
                        end
            */
            bit_0_state: begin
                            done=0; add=0; shift=0;  bit_0=1; bit_1=0; bit_2=0;
                        end
            bit_1_state: begin
                            done=0; add=0; shift=0;  bit_0=0; bit_1=1; bit_2=0;
                        end
            bit_2_state: begin
                            done=0; add=0; shift=0; bit_0=0; bit_1=0; bit_2=1;
                        end
            add_state:   begin
                            done=0; add=1; shift=0;
                        end
            shift_state: begin
                            begin done=0; add=0; shift=1;  end
                        end
            finished_state: begin
                                done =1; add=0; shift=0; bit_0=0; bit_1=0; bit_2=0;
                            end

            default: begin done=0; add=0; shift=0; bit_0=0; bit_1=0; bit_2=0; end
        endcase
    end


endmodule



module full_adder(input a, input b, input cin, output s, output cout);
    assign s = ( (a^b) ^ cin);
    assign cout = ( ((a^b) & cin) | (a&b) );
endmodule

module add_product(input [2:0] in1, input [2:0] in2, output [2:0]  out_put, output carr_out);
    wire outt1, outt2;
```

```
    full_adder sum1(.a(in1[0]), .b(in2[0]), .cin(1'b0), .s(out_put[0]), .cout(outt1) );
    full_adder sum2(.a(in1[1]), .b(in2[1]), .cin(outt1), .s(out_put[1]), .cout(outt2) );
    full_adder sum3(.a(in1[2]), .b(in2[2]), .cin(outt2), .s(out_put[2]), .cout(carr_out) );

endmodule

module data_processor(Clk,load, multiplicand, multiplier, shift, add, done, lsb, product, count_done, bit_0, bit_1,bit_2);
    input load, Clk, shift, add, done, bit_0,bit_1,bit_2;
    input [2:0] multiplicand;
    input [2:0] multiplier;
    output reg lsb;
    output reg count_done=0;
    output reg [5:0] product;

    reg [6:0] sum;
    reg [2:0] plier;
    reg [2:0] plier_reg;
    wire cout;
    wire [2:0] mul_reg;
    reg [2:0] count = 0;


    add_product add1(.in1(sum[5:3]) , .in2(multiplicand) , .out_put(mul_reg), .carr_out(cout));



    always @(posedge Clk)
    begin


        if(load) begin
            sum[6:0] = 7'b0000000;
            //count_done = 0;
            //count[2:0] <= 3'b000;
            sum[6:3] = 4'b0000;
            //sum[2:0] <= multiplier[2:0];
            //product[5:0] <= 6'b000000;
            //plier[2:0] <= 3'b000;
            plier_reg[2:0] = 4'b0000;
            //plier[2:0] <= multiplier[2:0];
            plier_reg[2:0] = multiplier[2:0];
            lsb = plier_reg[0];
            //lsb <= multiplier[0];

        end


        else if(bit_0==1)
        begin
            //lsb <= plier_reg[0];
            if(add)
            begin
                sum[6:3] = {cout, mul_reg};
            end

            if (shift)
            begin
                sum[6:0] = {1'b0, sum[6:1]};
                lsb = plier_reg[1];
            end
        end

        else if(bit_1==1)
        begin

            if(add)
            begin
                sum[6:3] = {cout, mul_reg};
            end

            if (shift)
            begin
                sum[6:0] = {1'b0, sum[6:1]};
                lsb = plier_reg[2];
            end
        end

        else if(bit_2==1)
        begin

            if(add)
            begin
                sum[6:3] = {cout, mul_reg};
                //product[5:0] <= sum[5:0];
            end

            if (shift)
            begin
                sum[6:0] = {1'b0, sum[6:1]};

                //product[5:0] = sum[5:0];
                //lsb <= plier_reg[1];
            end
        end

        if (done)
        begin
            product[5:0] <= sum[5:0];
        end
        product[5:0] = sum[5:0];
    end


endmodule
```

Figure 4

```
module Multiplier_3_3(Product, done, Multiplicand, Multiplier, Start, clock, reset_b);// Default configuration: five-bit datapath
    //parameter dp_width = 3; // Set to width of datapath
    output [5:0] Product;
    //output reg Ready;
    output reg done;
    input [2: 0] Multiplicand, Multiplier;
    input Start, clock, reset_b;

    //parameter BC_size= 3; // Size of bit counter
    parameter S_idle= 3'b001, // one-hot code
              S_add= 3'b010,
              S_shift= 3'b100;
    reg [2:0] state, next_state;
    reg [2:0] A, B, Q; // Sized for datapath
    reg C;
    reg [1:0] P;
    reg Load_regs, Decr_P, Add_regs, Shift_regs;
    // Miscellaneous combinational logic
    assign Product = {A, Q};
    wire Zero = (P == 0); // counter is zero // Zero = ~|P; // alternative
    //wire Ready = (state == S_idle); // controller status

    // control unit
    always @ (posedge clock, negedge reset_b) begin//next state assignment
        if (~reset_b) state <= S_idle;
        else state <= next_state;
    end

    always @(state, Start, Q[0], Zero) begin//next state logic
        Decr_P = 0;  //acc[3] will be carry bit
        Load_regs = 0;
        Add_regs=0;
        Shift_regs=0;
        next_state = S_idle; //default case

        case(state)
            S_idle : begin if(Start) begin next_state = S_add;
                                           Load_regs = 1; end end
            S_add : begin next_state = S_shift; Decr_P=1;
                          if (Q[0]) Add_regs=1; end
            S_shift : begin Shift_regs=1;
                            if(Zero) next_state = S_idle;
                            else next_state = S_add; end
        endcase
    end

    //Datapath Unit
    always @(posedge clock) begin //outputs
         if (Load_regs) begin
            P <= 3;
            A <= 0; //accumulator
            C <= 0;
            B <= Multiplicand;
            Q <= Multiplier;
         end
         if (P == 0) done <= 1;
         if (P > 0) done <= 0;
         if (Add_regs) {C,A} <= A+B;
         if (Shift_regs) {C,A,Q} <= {C,A,Q} >> 1;
         if (Decr_P) P <= P-1;
    end
endmodule
```

Figure 5

```
module tb_3_3();
    reg clk, start, reset_b;
    reg [2:0] mcand;
    reg [2:0] mplier;
    wire done;
    wire [5:0] product;


    Multiplier_3_3 DUT(.done(done), .clock(clk), .Product(product), .Multiplicand(mcand),
                       .Multiplier(mplier), .Start(start), .reset_b(reset_b));

    initial begin
        clk=0; start=0; mcand=3'b111; mplier = 3'b101;
        #30 start = 1;
        #10 start= 0;
        #100 start=1; mcand = 3'b100; mplier = 3'b001;
        #10 start=0;
        #100 start=1; mcand = 3'b111; mplier = 3'b100;
        #10 start=0;
        end
    always #5 clk=~clk;
endmodule
```

Figure 6

Task 3 was different by adding in a bit to make it a 4x4 bit multiplier, however, it also changed how the multiplier worked. Instead of using multiple lower level systems to add/shift bits into the correct place, task 3 uses a ROM file and the clock wizard to get the correct product. The code is shown below in Figure 7.

```
module multiplier_4x4(Ready, Mcand, Mplier, Start, clock, segments, AN, reset_b);
    parameter dp_width = 4; // Set to width of datapath
    wire [7:0] Product;
    output reg Ready; //also done
    output [6:0] segments;
    output [7:0] AN;
    wire DCM_lock;
    reg [3:0] ones, tens, hundreds;
    input [3:0] Mcand, Mplier;
    input Start, clock, reset_b;

    //parameter BC_size= 3; // Size of bit counter
    parameter S_idle= 3'b001, // one-hot code
              S_add= 3'b010,
              S_shift= 3'b100;
    reg [2:0] state, next_state;
    reg [3:0] A, B, Q; // Sized for datapath
    reg C;
    reg [1:0] P;
    reg Load_regs, Decr_P, Add_regs, Shift_regs;

    // Miscellaneous combinational logic
    assign Product = {A, Q};
    wire Zero = (P == 0); // counter is zero // Zero = ~|P; // alternative
    //wire Ready = (state == S_idle); // controller status

    integer counter = 0;

//    reg [3:0] ROM [31:0];
//    wire [3:0] Multiplicand, Multiplier;

//    assign Multiplicand = MY_ROM[Mcand];
//    assign Multiplier = MY_ROM[Mplier+16];
//    initial $readmemb ("mcandsandmpliers.mem", MY_ROM, 0, 31);

    wire [3:0] multiplicand;
    wire [3:0] multiplier;
    reg [3:0] ROM [31:0];
    assign multiplicand = ROM[Mcand];
    assign multiplier = ROM[Mplier + 16];
    initial $readmemb ("memory.mem", ROM, 0, 31);

    // control unit
    always @ (posedge clock, negedge reset_b) begin//next state assignment
        if (~reset_b) state <= S_idle;
        else state <= next_state;
    end

    always @(state, Start, Q[0], Zero) begin//next state logic
        Decr_P = 0;  //acc[3] will be carry bit
        Load_regs = 0;
        Add_regs=0;
        Shift_regs=0;
        next_state = S_idle; //default case

        case(state)
            S_idle : begin if(Start) begin next_state = S_add;
                                    Load_regs = 1; end end
            S_add : begin next_state = S_shift; Decr_P=1;
                    if (Q[0]) Add_regs=1; end
            S_shift : begin Shift_regs=1;
                            if(Zero) next_state = S_idle;
                            else next_state = S_add; end
            default : next_state = S_idle;
        endcase
    end

    //Datapath Unit
    always @(posedge clock) begin //outputs
         if (Load_regs) begin
            P <= 4; //P=dp_width;
            A <= 0;        //accumulator
            C <= 0;
            B <= multiplicand;
            Q <= multiplier;
         end
         if (P == 0) Ready <= 1;
         if (P > 0) Ready <= 0;
         if (Add_regs) {C,A} <= A+B;
         if (Shift_regs) {C,A,Q} <= {C,A,Q} >> 1;
         if (Decr_P) P <= P-1;
    end

    //convert product bits to bcd
    //reg [3:0] ones;
    //reg [3:0] tens;
    //reg [3:0] hundreds;
    always @ (Product)
    begin
        ones = Product % 10;
        tens = (Product / 10) % 10;
        hundreds = Product / 100;
    end


    //7 segment display
    wire clk_src_5MHz;
    wire clk_src_500Hz;
    wire lock_signal;
    reg pulse_at_500Hz;

    //Error with both clk_src_5Mhz and 500Hz
    clk_wiz_0 clk_5MHz(.clk_in1(clock), .clk_out1(clk_src_5MHz), .locked(DCM_lock));

    clock_divider divider(.clock_in(clk_src_5MHz), .clock_out(clk_src_500Hz));

    initial begin
        pulse_at_500Hz = 0;
    end

    always @(posedge clk_src_500Hz)
        begin
            pulse_at_500Hz = ~pulse_at_500Hz;
        end

        //Display on 7 segment display
    bcd_to_7_seg seg1(.clk_in(clk_src_500Hz), .ones(ones), .segments(segments), .AN(AN), .tens(tens), .hundreds(hundreds));

//    bcd_converter_7_seg seg7(.tens(tens), .ones(ones), .hundreds(hundreds), .v(Product), .segments(segments), .AN(AN),
//            .reset(reset_b), .clk_in(clock), .DCM_lock(DCM_lock));

endmodule
```

Figure 7

Verification:

Task 1 firstly is tested by using a testbench to simulate the results, then both of these tasks are verified by programming them to the board and visually verifying the results. When verifying, we simply load the multiplicand and multiplier in the inputs and watch to see what the output is. In the case of task 1 and 2, we can also be sure to check if the done signal has gone high when the output number is correct. The simulation results for task 1 is shown below in Figure 8.

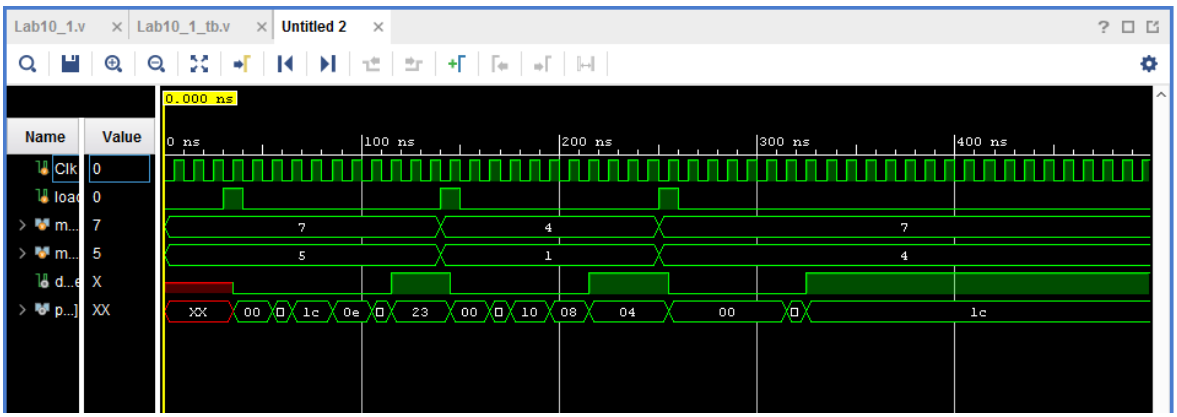

Figure 8

Conclusion:

What I learned in this lab is how ASM charts can be used to design complex control units. I also designed digital system to perform binary multiplication using the ASM chart technique to develop the control unit which interfaced to the datapath processing unit. I also learned more about testbenches, the clock_wizard, and ROM files. This lab did not go very well in comparison to the few weeks before but since this is the last lab, after learning this last bit about ASM's, we do not need these tasks to build off another.