Lab 10: Sequential System Design Using ASM Charts

Ethan Cook, ELEC 4200 - section 001

April 9th, 2024

1 Introduction

The purpose of this lab is for students to design sequential systems using Algorithmic State Machines (ASM) charts. ASM charts are used to create complex control units. After completing this lab, students should have an in-depth experience creating complex control units. All code in this report is written with Verilog and implemented with Vivado 2018.1 and a Nexys A7 board.

2 Task 1

The goal for the first task is to design a 3-bit by 3-bit binary multiplier. The multiplier will be made up of three parts: the data processor, the control unit, and the overlying module. The data processor contains a 3-bit accumulator, a 3-bit multiplier register, a 3-bit adder, a counter, and a 3-bit shifter. The control unit contains a least significant bit (LSB), a start signal, a cnt_done signal, and a clk as input. It generates start, shift, add, and done flags. Before developing the model, students should create an ASM chart for the control unit. The chart is found in Figure 1. The code and simulation results, after running the code through a test bench, is shown in Figures 2 and 3, respectively.

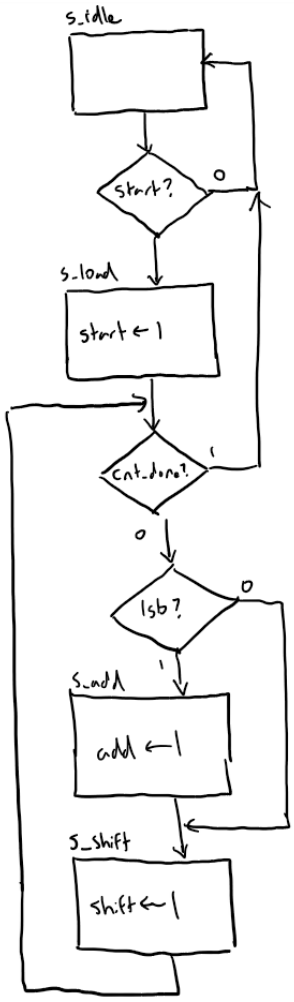

Figure 1: 3-Bit by 3-Bit Binary Multiplier Control Unit ASM Chart

```
module multiplier_3x3(
    input clk, input start, input [2:0] multiplicand, input [2:0] multiplier,
    output done, output [5:0] product);
wire start_f, add_f, shift_f, lsb, cnt_done;
dp_multiplier_3x3 dp (clk, start_f, add_f, shift_f, multiplicand, multiplier, lsb, cnt_done, product);
cu_multiplier_3x3 cu (clk, start, lsb, cnt_done, start_f, add_f, shift_f, done);
endmodule

module dp_multiplier_3x3(
    input clk, input start_f, input add_f, input shift_f, input [2:0] multiplicand, input [2:0] multiplier,
    output lsb, output cnt_done, output [5:0] P);

reg [2:0] acc;
reg [2:0] mult_reg;
reg [1:0] counter;
reg carry;

assign lsb = mult_reg[0];
assign cnt_done = (counter == 2'b11);
assign P = {acc, mult_reg};

always @ (posedge clk) begin
    if (start_f) begin mult_reg <= multiplier; acc <= 0; counter <= 0; end
    if (add_f) begin {carry, acc} <= acc + multiplicand; end
    if (shift_f) begin counter <= counter + 1; mult_reg <= {acc[0], mult_reg[2:1]}; acc <= {carry, acc[2:1]}; end
end

endmodule

module cu_multiplier_3x3(
    input clk, input start, input lsb, input cnt_done,
    output reg start_f, output reg add_f, output reg shift_f, output reg done_f);
reg [1:0] state, next_state;
parameter s_idle = 2'b00, s_load = 2'b01, s_add = 2'b10, s_shift = 2'b11;

always @ (posedge clk)
    state <= next_state;

always @ (state or start or lsb or cnt_done) begin
    next_state = s_idle;
    start_f = 0; shift_f = 0; add_f = 0;
    case (state)
        s_idle: if (start) begin next_state = s_load; start_f = 1; done_f = 0; end
                else next_state = s_idle;
        s_load: if (cnt_done) begin next_state = s_idle; done_f = 1; end
                else if (lsb) begin next_state = s_add; add_f = 1; end
                else begin next_state = s_shift; shift_f = 1; end
        s_add:  begin next_state = s_shift; shift_f = 1; end
        s_shift:    if (cnt_done) begin next_state = s_idle; done_f = 1; end
                    else if (lsb) begin next_state = s_add; add_f = 1; end
                    else begin next_state = s_shift; shift_f = 1; end
        default: next_state = s_idle;
    endcase
end
```

Figure 2: 3-Bit by 3-Bit Binary Multiplier Code

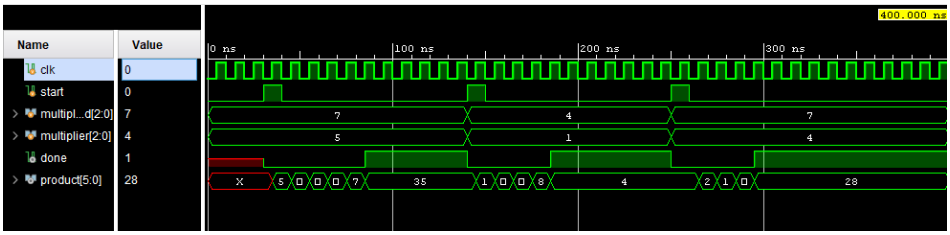

Figure 3: 3-Bit by 3-Bit Binary Multiplier Simulation Result

3 Task 2

The goal for task 2 is to implement the design in the first task onto the FPGA board. After assigning the inputs to switches and a button and assigning the outputs to LED's, the implementation should be verified. No figures are needed for this task.

4 Task 3

The goal for task 3 is to modify the design in task 2 to perform 4-bit by 4-bit binary multiplication. The 4-bit multiplicand and multipliers must be stored in 32x4 ROM. The input to the design will

be multiplicand and multiplier addresses using switches. The design should take in a 5 MHz clock. The right most three 7-segment displays should show the decimal result. The design should then be implemented and verified. The code for task 3 is shown in Figures 4, 5, and 6.

```
module multiplier_4x4_adv(
    input clk_100MHz, input start, input [3:0] multiplicand_address, input [3:0] multiplier_address,
    output done, output [6:0] seg, output [7:0] AN);

wire start_f, add_f, shift_f, lsb, cnt_done;
wire [7:0] product;
reg [3:0] INPUT [31:0];
wire [3:0] multiplicand, multiplier;
wire [11:0] product_bcd;

assign multiplicand = INPUT[{1'b0, multiplicand_address}];
assign multiplier = INPUT[{1'b1, multiplier_address}];

wire clk_5MHz, locked;

clk_wiz_0 instance_name (.clk_5MHz(clk_5MHz), .locked(locked), .clk_100MHz(clk_100MHz));

dp_multiplier_4x4 dp (clk_5MHz, start_f, add_f, shift_f, multiplicand, multiplier, lsb, cnt_done, product);
cu_multiplier_4x4 cu (clk_5MHz, start, lsb, cnt_done, start_f, add_f, shift_f, done);

bcd_converter con (product, product_bcd);

wire [6:0] seg0, seg1, seg2;

bcd_to_seg(product_bcd[3:0], 1'b0, seg0);
bcd_to_seg(product_bcd[7:4], ~|product_bcd[11:4], seg1);
bcd_to_seg(product_bcd[11:8],~|product_bcd[11:8], seg2);

wire [1:0] dig_sel;
reg [13:0] refresh_count;

always @ (posedge clk_5MHz)
    refresh_count <= refresh_count + 1;

assign dig_sel = refresh_count[13:12];

seg_display(seg0, seg1, seg2, dig_sel, AN, seg);


initial $readmemb ("ROM_data.mem", INPUT, 0, 31);

endmodule

module seg_display(input [6:0] seg0, input [6:0] seg1, input [6:0] seg2, input [1:0] seg_sel, output reg [7:0] AN, output reg [6:0] seg);

always @ (*) begin
    seg = seg0;
    AN = 8'b11111111;
    if (seg_sel == 2'b00) begin
        seg = seg0;
        AN = 8'b11111110;
    end else if (seg_sel == 2'b01) begin
        seg = seg1;
```

Figure 4: 4-Bit by 4-Bit Binary Multiplier Code, Decimal Output Part 1

```
        AN = 8'b11111101;
    end else if (seg_sel == 2'b10) begin
        seg = seg2;
        AN = 8'b11111011;
    end else begin
        seg = 7'b1111111;
        AN = 8'b11110111;
    end
end

endmodule

module bcd_to_seg(input [3:0] bcd, input hide, output reg [6:0] seg);

always @ (bcd) begin
    seg = 7'b1111111;
    if (hide) seg = 7'b1111111;
    else begin
        case (bcd)
            0: seg = 7'b0000001;
            1: seg = 7'b1001111;
            2: seg = 7'b0010010;
            3: seg = 7'b0000110;
            4: seg = 7'b1001100;
            5: seg = 7'b0100100;
            6: seg = 7'b0100000;
            7: seg = 7'b0001111;
            8: seg = 7'b0000000;
            9: seg = 7'b0000100;
            default: seg=7'b1111111;
        endcase
    end
end

endmodule

module bcd_converter (input [7:0] num, output [11:0] bcd_out);

assign bcd_out[3:0] = num % 10;
assign bcd_out[7:4] = num/10 % 10;
assign bcd_out[11:8] = num/100;

endmodule

module dp_multiplier_4x4(
    input clk, input start_f, input add_f, input shift_f, input [3:0] multiplicand, input [3:0] multiplier,
    output lsb, output cnt_done, output [7:0] P);

reg [3:0] acc;
reg [3:0] mult_reg;
reg [2:0] counter;
```

Figure 5: 4-Bit by 4-Bit Binary Multiplier Code, Decimal Output Part 2

```
reg carry;

assign lsb = mult_reg[0];
assign cnt_done = (counter == 3'b100);
assign P = {acc, mult_reg};

always @ (posedge clk) begin
    if (start_f) begin mult_reg <= multiplier; acc <= 0; counter <= 0; carry <= 0; end
    if (add_f) begin {carry, acc} <= acc + multiplicand; end
    if (shift_f) begin counter <= counter + 1; mult_reg <= {acc[0], mult_reg[3:1]}; acc <= {carry, acc[3:1]}; carry <= 0; end
end

endmodule

module cu_multiplier_4x4(
    input clk, input start, input lsb, input cnt_done,
    output reg start_f, output reg add_f, output reg shift_f, output reg done_f);

reg [1:0] state, next_state;
parameter s_idle = 2'b00, s_load = 2'b01, s_add = 2'b10, s_shift = 2'b11;

always @ (posedge clk)
    state <= next_state;

always @ (state or start or lsb or cnt_done) begin
    next_state = s_idle;
    start_f = 0; shift_f = 0; add_f = 0;
    case (state)
        s_idle: if (start) begin next_state = s_load; start_f = 1; done_f = 0; end
                else next_state = s_idle;
        s_load: if (cnt_done) begin next_state = s_idle; done_f = 1; end
                else if (lsb) begin next_state = s_add; add_f = 1; end
                else begin next_state = s_shift; shift_f = 1; end
        s_add:  begin next_state = s_shift; shift_f = 1; end
        s_shift:    if (cnt_done) begin next_state = s_idle; done_f = 1; end
                    else if (lsb) begin next_state = s_add; add_f = 1; end
                    else begin next_state = s_shift; shift_f = 1; end
        default: next_state = s_idle;
    endcase
end

endmodule
```

Figure 6: 4-Bit by 4-Bit Binary Multiplier Code, Decimal Output Part 3

5 Conclusion

Overall, this was an exceptional lab to master designing control units with ASM charts. I ran into no issue. I thoroughly enjoyed this lab and do not have any suggestions on how to improve it.