CMPE 125 Lab 7 Assignments

Dr. Donald Hung
Computer Engineering Department, San Jose State University

System-level Design (1)

Purpose:

1) To be familiar with system-level design methodologies, with an emphasis on the control portion
2) To be familiar with the procedure and tools for system-level development

Preparation:

Review lecture notes on finite state machine (FSM) and system-level design

Description:

The attached diagram (see Attachment 1) shows the control (FSM) and datapath (in dotted box) of a small calculator capable to perform ADD, SUBTRACT, AND, and XOR operations. The calculator will operate based on the system clock *CLK_db*. When idling, the calculator constantly watches for the *Go* signal and once the signal is detected, the calculator will do the following:

1) Load 3-bit input data (operands) *In1* and *In2* into the register file:

```
R1 ← In1;
R2 ← In2;
```

2) Executes the algorithm shown below, depending on the 2-bit operation control input *op*:

```
if (Op = 2'b11)            //Add
     R3 ← R1 + R2;
else if(Op = 2'b10)        //Subtract
     R3 ← R1 - R2;
else if(Op = 2'b01)        //And
     R3 ← R1 & R2;
else                       //Xor
     R3 ← R1 ^ R2;
```

3) Output the 3-bit result *Out* along with the flag signal *Done*, then return to the idle mode.

Tasks:

The lab is split into four tasks across two weeks. At the end of each week, have the instructor check your work. You will be assigned a grade for the task(s) that are due at that time.

Tasks for first week:

1. Functionally verify the datapath (DP).

   Verilog source code for the DP is provided (see Attachment 2). It connects all the building blocks in the DP (via module instantiation and port mapping). You need to write a Verilog testbench which generates the following stimuli:

   - Clock,
   - Control (s1, WA, WE, RAA, REA, RAB, REB, c, s2), and
   - Data (In1, In2),

   to be applied to the DP for its load and arithmetic/logic operations as specified on Page 1, and observe the DP's response (the signal Out). Keep in mind that each of these operations must be executed in exactly one clock cycle, therefore pay attention to the timing of the stimuli and response.

   Your testbench could be "eyeballing" with hard-coded timing for the stimuli, waveform viewer must be used in order to understand signal timing.

Tasks for second week:

2. Design and functionally verify the control unit (CU).

   You should use the systematic design methodology taught in class to design the CU (FSM), i.e., starting with constructing an ASM chart that describes the cycle-by-cycle operation of the system (the calculator in this case), then extract the *state transition diagram* and *output table* for describing (in HDL) the FSM's *next-state logic* and *output logic* respectively. The FSM must be tested thoroughly via functional verification.

   Use a **self-checking testbench** for functional verification of the control unit.

3. Integrate and functionally verify the entire system.

   You should write a top-level Verilog design code for the calculator by connecting the FSM with the DP, and write a Verilog testbench to verify the overall system's functional correctness. Hint: Remember to check the DONE flag to know when the state machine has finished running.

   Use a **self-checking testbench** for functional verification of the integrated system.

4. Validate the designed calculator using the Nexys4 DDR FPGA board (Attachment 3 contains Verilog source code of the utility blocks needed to set up the prototyping environment). Set the FPGA pin constraints based on the I/O arrangement specified below:
   - use an on-board push buttons to generate the start signal *Go*,
   - use another on-board push button for a manually generated system clock *CLK_db* (must be de-bounced, as shown in Attachment 1),
   - use six on-board DIP switches to enter the two 3-bit binary operands *In1* and *In2*,
   - use the two remaining on-board DIP switches to enter the 2-bit operation control input *op*,
   - use one of the on-board barcode LEDs to display the output signal *Done*,
   - use one on-board 7-segment LED to display the FSM's current state *CS* (this is for diagnosis purpose),

- use another on-board 7-segment LED to display the calculated result *Out*.

Use a debounced pushbutton for the clock so that you can step through states with each button press. Do not use the four second clock or 100 MHz system clock as the clock source.

Contents of Report:

1) Cover page.
2) A list of successfully accomplished tasks.
3) For Tasks 1 – 3, you should include the following:
   a) A test plan for functional verification
   b) Commented Verilog source code
   c) Captured verification results (waveforms/simulation log files)

   In addition, for Task 2 (the control unit), you should also include the following:
   d) The ASM chart describing the system's cycle-by-cyle operation
   e) The *state transition diagram* and the *output table* extracted from the ASM chart
4) For Task 4 (validation), you should describe the hardware prototyping setup and your test plan for hardware validation
5) Descriptions and analysis on your observations (for all tasks)
6) Detailed descriptions of the task(s) that you cannot, or did not accomplish, including reason(s) and/or your analysis.

Report checklist

Follow the “CMPE 125 Lab Report Guidelines” document on Canvas.

Your report should include these sections:

- Cover page *(with name, SID, and the Lab Record filled out with your name, TA’s name, and lab demo score)*

- Introduction *(1 paragraph)*

- Design methodology *(1 paragraph for each module: CU, DP, CU+DP)*
    - Block diagrams:
        - Control unit (CU)
        - Data path (DP)
        - Combined top-level calculator (CU+DP)
    - Other diagrams required:
        - ASM chart
        - State transition diagram (“bubble” diagram)
        - Output table
        - FSM Structural Diagram

    - Note: the block diagrams must adhere closely to the lab report guidelines.

- Test plan for simulation *(1 paragraph for each module: CU, DP, CU+DP)*
    - TCL console output is required for each module.
    - Console output from simulator (as a picture), one for each module.

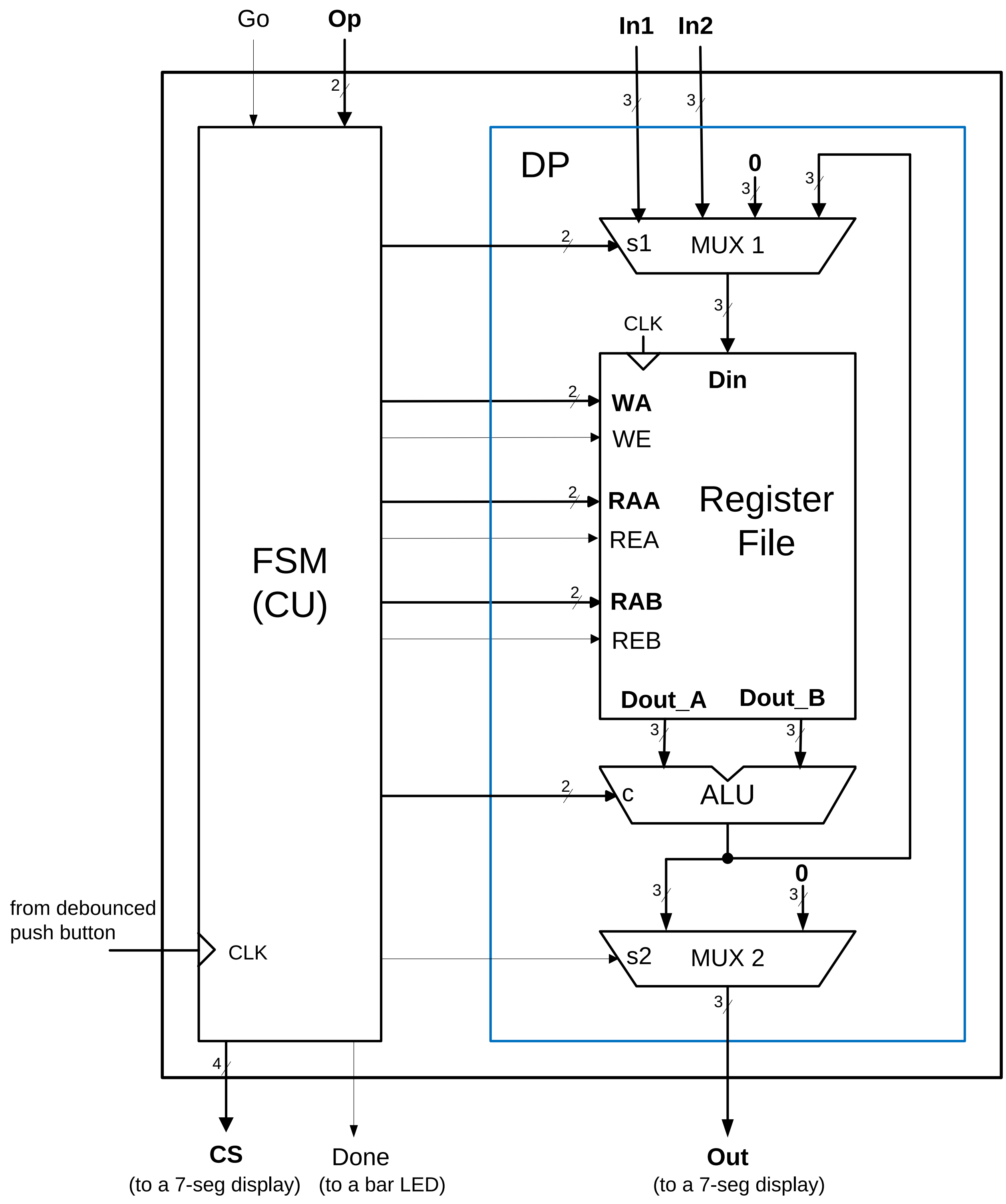
- Test plan for FPGA board *(1 paragraph)*
    - Include photograph of FPGA board showing results with a caption that clearly labels the state of the switches and LEDs.

- Conclusion *(1 paragraph)*
    - In addition if you could not complete any part of the lab explain clearly what went wrong and how you could have remedied the problem.

- Appendix
    - Any tables or other diagrams go here that don’t fit in the other sections.

- Source code
    - Use a fixed-width font such as `Courier New` for the source code.
    - Clearly label the file name of each source file you include.
    - Include **all** source code you used in the project, including constraints.

Attachment 1: System Schematics



Microarchitecture of the Small Calculator

Attachment 2: Verilog Source Code for the Datapath (DP)

```
module DP(in1, in2, s1, clk, wa, we, raa, rea, rab, reb, c, s2, out);

input [2:0] in1, in2;
input [1:0] s1, wa, raa, rab, c;
input we, rea, reb, s2, clk;
output [2:0] out;
wire [2:0] mux1out;
wire [2:0] douta;
wire [2:0] doutb;
wire [2:0] aluout;
// instantiate the building blocks
MUX1 U0(.in1(in1), .in2(in2), .in3(3'b000), .in4(aluout), .s1(s1),
.m1out(mux1out));
RF U1(.clk(clk), .rea(rea), .reb(reb), .raa(raa), .rab(rab), .we(we), .wa(wa),
.din(mux1out), .douta(douta), .doutb(doutb));
ALU U2(.in1(douta), .in2(doutb), .c(c), .aluout(aluout));
MUX2 U3(.in1(aluout), .in2(3'b000), .s2(s2), .m2out(out));

endmodule //DP


/* Source Code for the Building Blocks Used */

module MUX1(in1, in2, in3, in4, s1, m1out);

input [2:0] in1, in2, in3, in4;
input [1:0] s1;
output reg [2:0] m1out;

always @ (in1, in2, in3, in4, s1)
begin
      case (s1)
            2'b11:            m1out = in1;
            2'b10:            m1out = in2;
            2'b01:            m1out = in3;
            default:          m1out = in4; // 2'b00
      endcase
end

endmodule //MUX1


module RF(clk, rea, reb, raa, rab, we, wa, din, douta, doutb);

input clk, rea, reb, we;
input [1:0] raa, rab, wa;
input [2:0] din;
output reg [2:0] douta, doutb;
reg [2:0] RegFile [3:0];

always @(rea, reb, raa, rab)
begin
      if (rea)
            douta = RegFile[raa];
      else douta = 3'b000;
      if (reb)
            doutb = RegFile[rab];
      else doutb = 3'b000;
end
```

```
always@(posedge clk)
begin
      if (we)
            RegFile[wa] = din;
      else
            RegFile[wa] = RegFile[wa];
end

endmodule //RF


module ALU (in1, in2, c, aluout);

input [2:0] in1, in2;
input [1:0] c;
output reg [2:0] aluout;

always @ (in1, in2, c)
begin
      case (c)
            2'b00:            aluout = in1 + in2;
            2'b01:            aluout = in1 - in2;
            2'b10:            aluout = in1 & in2;
            default:          aluout = in1 ^ in2; // 2'b11;
      endcase
end

endmodule //ALU


module MUX2 (in1, in2, s2, m2out);

input [2:0] in1, in2;
input s2;
output reg [2:0] m2out;

always @ (in1, in2, s2)
begin
if (s2)
      m2out = in1;
else
      m2out = in2;
end

endmodule //MUX2
```

Attachment 3: Verilog Source Code for Utility Blocks Needed by FPGA Prototyping

```
   module clk_gen(clk100MHz, rst, clk_4sec, clk_5KHz);
   input clk100MHz, rst;
   output clk_4sec, clk_5KHz;
   reg clk_4sec, clk_5KHz;
   integer count, count1;
   always@(posedge clk100MHz)
   begin
     if(rst)
     begin
```

```
        count = 0;
        count1 = 0;
        clk_4sec = 0;
        clk_5KHz  =0;
   end
   else
   begin
        if(count == 200000000)
        begin
             clk_4sec = ~clk_4sec;
             count = 0;
        end
        if(count1 == 10000)
        begin
             clk_5KHz = ~clk_5KHz;
             count1 = 0;
        end
        count = count + 1;
        count1 = count1 + 1;
   end
end
endmodule // end clk_gen

module bcd_to_7seg(BCD, s0, s1, s2, s3, s4, s5, s6);
output s0, s1, s2, s3, s4, s5, s6;
input [3:0] BCD;
reg s0, s1, s2, s3, s4, s5, s6;
always @ (BCD)
begin // BCD to 7-segment decoding
case (BCD) // s0 – s6 are active low
 4'b0000: begin s0=0; s1=0; s2=0; s3=1; s4=0; s5=0; s6=0; end
 4'b0001: begin s0=1; s1=0; s2=1; s3=1; s4=0; s5=1; s6=1; end
 4'b0010: begin s0=0; s1=1; s2=0; s3=0; s4=0; s5=1; s6=0; end
 4'b0011: begin s0=0; s1=0; s2=1; s3=0; s4=0; s5=1; s6=0; end
 4'b0100: begin s0=1; s1=0; s2=1; s3=0; s4=0; s5=0; s6=1; end
 4'b0101: begin s0=0; s1=0; s2=1; s3=0; s4=1; s5=0; s6=0; end
 4'b0110: begin s0=0; s1=0; s2=0; s3=0; s4=1; s5=0; s6=0; end
 4'b0111: begin s0=1; s1=0; s2=1; s3=1; s4=0; s5=1; s6=0; end
 4'b1000: begin s0=0; s1=0; s2=0; s3=0; s4=0; s5=0; s6=0; end
 4'b1001: begin s0=0; s1=0; s2=1; s3=0; s4=0; s5=0; s6=0; end
 default: begin s0=1; s1=1; s2=1; s3=1; s4=1; s5=1; s6=1; end
endcase
end
endmodule  // end bcd_to_7seg

module led_mux (clk, rst, LED0, LED1, LED2, LED3, LEDOUT, LEDSEL);
input clk, rst;
input [7:0] LED0, LED1, LED2, LED3;
output[3:0] LEDSEL;
output[7:0] LEDOUT;
reg [3:0] LEDSEL;
reg [7:0] LEDOUT;
```

```
   reg [1:0] index;
   always @(posedge clk)
   begin
      if(rst)
            index = 0;
      else
            index = index + 2’d1;
   end
   always @(index,LED0,LED1,LED2,LED3)
   begin
      case(index)
      0:    begin
            LEDSEL = 4'b1110;
            LEDOUT = LED0;
            end
      1:    begin
            LEDSEL = 4'b1101;
            LEDOUT = LED1;
            end
      2:    begin
            LEDSEL = 4'b1011;
            LEDOUT = LED2;
            end
      3:    begin
            LEDSEL = 4'b0111;
            LEDOUT = LED3;
            end
      default: begin
                        LEDSEL = 0; LEDOUT = 0;
                 end
      endcase
   end
endmodule  // end led_mux

module button_debouncer #(parameter depth = 16) (
    input wire clk,                 /* 5 KHz clock */
    input wire button,              /* Input button from constraints */
    output reg debounced_button);

    localparam history_max = (2**depth)-1;

    /* History of sampled input button */
    reg [depth-1:0] history;

    always @ (posedge clk)
    begin
        /* Move history back one sample and insert new sample */
        history <= { button, history[depth-1:1] };

        /* Assert debounced button if it has been in a consistent state
throughout history */
        debounced_button <= (history == history_max) ? 1'b1 : 1'b0;
    end
endmodule
```