LABORATORY # 4

Sequential Logic Design

PART 1[1] (Finite State Machine) - Design of Flight Attendant Call System

PART 2 (Edge Detector) - Design of Rising Edge Detector

PART 3 (Time Multiplexing) - Design of Controller for Sequential Display of 4 Digits

[1] Design Example guided through by TA

Objectives

Lab 4 contains 3 parts: **Part 1** – implementation of a sequential circuit (state machine) with detailed guidance, **Part 2** – independent design of a rising edge detector state machine and **Part 3** – independent design of a controller for sequential display of four numbers on 7-segment displays. Lab4 focuses on using behavioral Verilog description to define state machines. Its purpose is to get familiar with:

i. Clock-synchronous state machine design, synthesis, and implementation
ii. Usage of Basys3 board's internal clocks
iii. Using clock divider circuit to reduce clock frequency
iv. Time-multiplexing of seven-segment displays

Equipment

- PC or compatible
- Digilent's Basys3 FPGA Evaluation Board

Software

Xilinx Vivado Design Software Suite

PART 1. Design of Flight Attendant Call System

In this FPGA application experiment, "*flight attendant call system*" will be implemented and tested. Because the design is a synchronous state machine, a clock source is needed to clock the registers. This lab will use the internal clock source of the Basys3 board which is a 100MHz oscillator connected to pin W5. For this part, the clock frequency is not important, and hence the 100MHz oscillator can directly be used as the clock source.

Specification

The Flight Attendant Call System functions according to the following rules -

Press CALL button: light turns on and *stays on* even after the button is released
Press CANCEL button: light turns off

Figure 1 shows the specification of the flight attendant call system.

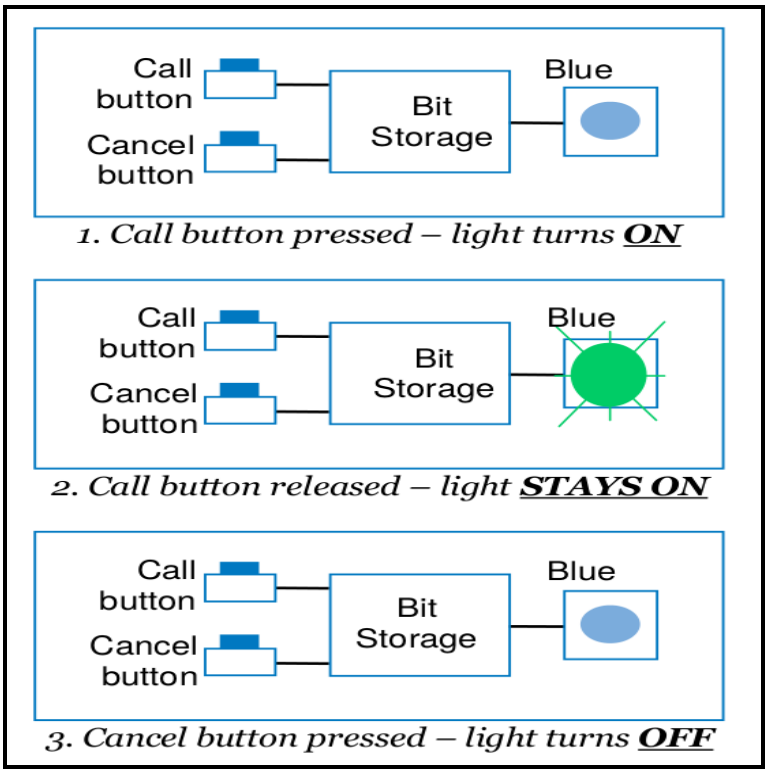
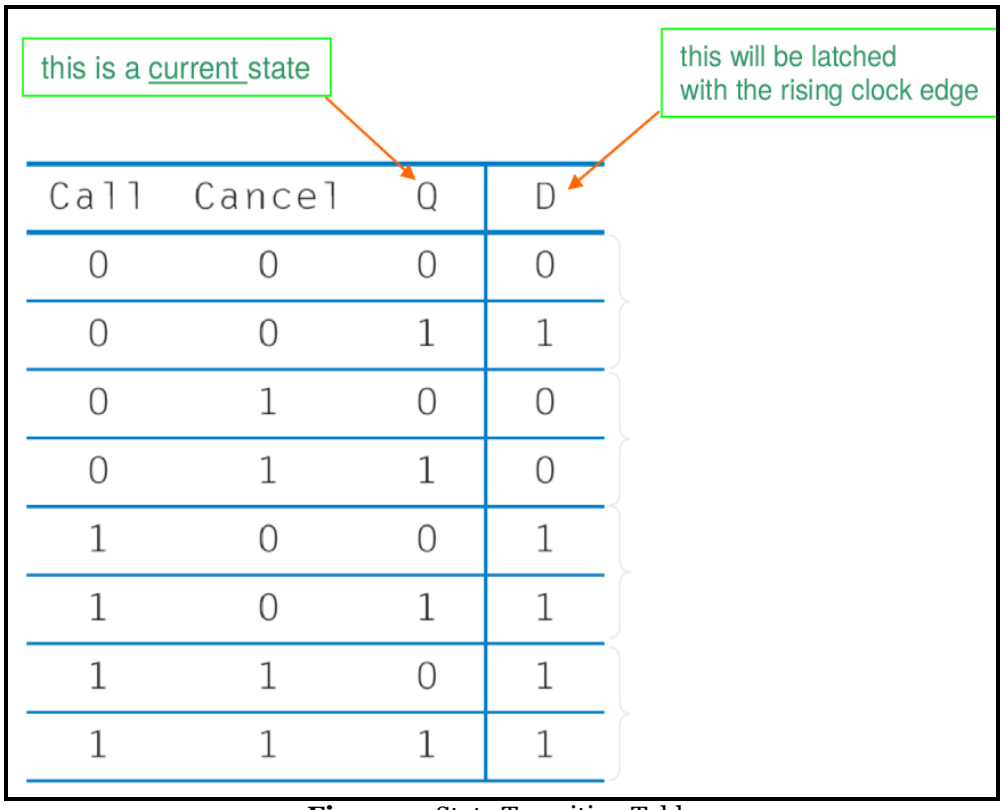


Figure 1. Flight Attendant Call System Specification

System Analysis and Implementation

Step 1 (Drawing the state table) – The first step in implementation is to come up with the state transition table as per the specification. It is easy to observe that there will be two possible states here – one when the light is on and the other when the light is off. Since 2 states require just one flip-flop for Finite State Machine (FSM) implementation, the state table can be drawn to express the next state as a function of the inputs (Call and Cancel) and the present state. It is highly recommended that you draw your own state tables for getting a better understanding of the system, and then compare it against the one provided in Figure 2.



Call	Cancel	Q	D
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Figure 2. State Transition Table

Note -> Although this system can be implemented using both structural and behavioral modeling, the aim is to practice behavioral modeling in this lab. Hence, all the parts will have to be implemented using behavioral modeling, *except for an additional exercise towards the end of this part*.

Step 2 (Writing the Verilog code) – The next step is write the Verilog code according to the state table obtained above. As done in the previous labs, a new

project needs to be created in Vivado with the right part number (*XC7A35TCPG236-1*). After creating the project, a new design source needs to be added to start writing the Verilog code, which is given below for this part. Note that the variable light_state corresponds to Q, and the variable next_state corresponds to D from the state table in Figure 2.

```
1  `timescale 1ns / 1ps
2
3  module flight_attendant_call_system(
4  input wire clk,
5  input wire call_button ,
6  input wire cancel_button ,
7  output reg light_state
8  );
9
10 reg next_state ;
11
12 // Combinational block
13 always @(*) begin
14     case ({call_button,cancel_button,light_state})
15     3'b000: next_state = 1'b0;
16     3'b001: next_state = 1'b1;
17     3'b010: next_state = 1'b0;
18     3'b011: next_state = 1'b0;
19     3'b100: next_state = 1'b1;
20     3'b101: next_state = 1'b1;
21     3'b110: next_state = 1'b1;
22     3'b111: next_state = 1'b1;
23     default : next_state = 1'b0 ;
24     endcase
25 end
26
27 // Sequential block
28 always @( posedge clk ) begin
29     light_state  <= next_state ;
30 end
31
32 endmodule
```

Step 3 (Creating the Verilog testbench) – After completing the design file, the next step is to write the testbench to verify the module. The testbench for this part is given below and can be used for simulation. The simulation waveform should be observed carefully to verify circuit behavior.

```
`timescale 1ns / 1ps

module tb_flight_attendant_call_system;

reg clk;
reg call_button;
reg cancel_button;

wire light_state;

flight_attendant_call_system u1 (
 .clk(clk),
 .call_button(call_button),
 .cancel_button(cancel_button),
 .light_state(light_state)
 );

initial
begin

clk = 0;
call_button = 0;
cancel_button = 0;

#10;

call_button = 1;
cancel_button = 0;

#10;

call_button = 0;
cancel_button = 1;

#10;

call_button = 1;
cancel_button = 1;

#10;

call_button = 0;
cancel_button = 0;

#10;

call_button = 1;
cancel_button = 0;

#10;

cancel_button = 1;

#20;

cancel_button = 0;

#20;
```

```
59
60    call_button = 0;
61    cancel_button = 1;
62
63    #20;
64
65    call_button = 0;
66    cancel_button = 0;
67
68    end
69
70    always
71    #5 clk = ~clk;
72
73    endmodule
```

Step 4 (Synthesis and Implementation) – If the behavioral simulation of the system is correct, the next step is to implement the system on the Basys3 board. The two buttons and an LED on the Basys3 board can be used to emulate the call button, the cancel button, and the light respectively, as shown in Figure 3.

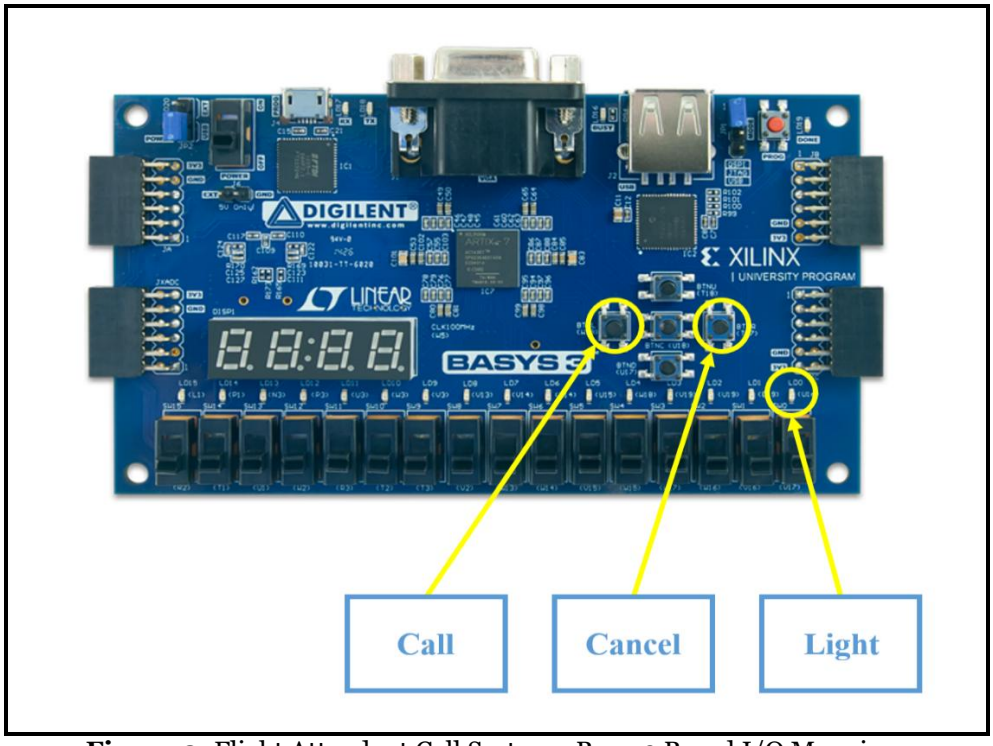


Figure 3. Flight Attendant Call System - Basys3 Board I/O Mappings

To map the ports of the module to the I/Os of the Basys board, a constraints file needs to be created, as done in Lab 3. The uncommented lines of the constraints file are given below.

```
// Clock signal
set_property PACKAGE_PIN W5 [get_ports {clk}]
      set_property IOSTANDARD LVCMOS33 [get_ports {clk}]
      create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}]

//Buttons
set_property PACKAGE_PIN W19 [get_ports {call_button}]
      set_property IOSTANDARD LVCMOS33 [get_ports {call_button}]
set_property PACKAGE_PIN T17 [get_ports {cancel_button}]
      set_property IOSTANDARD LVCMOS33 [get_ports {cancel_button}]

//LED
set_property PACKAGE_PIN U16 [get_ports {light_state}]
      set_property IOSTANDARD LVCMOS33 [get_ports {light_state}]
```

After adding the constraints file, synthesis and implementation needs to be run, as done in lab 3.

Step 5 (Generate Bitstream and Program Device) – The final step to implement the system on the board is to generate the bitstream file after the implementation step and program the Basys3 board with the generated bitstream file. Now, your design can be verified on the board.

Additional Exercise – Dataflow Modeling

The aim of this exercise is to appreciate the fact that the next state can always be expressed *in terms of the current state and the inputs using a Boolean expression*. The code snippet for the dataflow modeling is given below. Only the assign statement needs to be completed using the minimized expression of next_state to be derived from the state table in Figure 2. As discussed before, the variable light_state corresponds to Q, and the variable next_state corresponds to D in the state table. Use a K-map to obtain the minimal expression for next_state.

```
1  `timescale 1ns / 1ps
2
3  module flight_attendant_call_system_dataflow(
4  input wire clk,
5  input wire call_button ,
6  input wire cancel_button ,
7  output reg light_state
8  );
9
10 wire next_state ;
11
12 // Combinational block
13 assign next_state = /*Fill the code with the minimized expression obtained from the state table */;
14
15 // Sequential block
16 always @( posedge clk ) begin
17     light_state  <= next_state ;
18 end
19
20 endmodule
```

The same testbench as provided above can be used to verify the behavior of the dataflow-modeled system, just by changing the name of the instantiated module. Post simulation, the same constraints file can be used to implement the design on the Basys3 board.

PART 2. Rising-Edge Detector

Specification

The rising edge detector is a circuit that generates a short, one-clock-cycle pulse (called a tick) when the input signal changes from '0' to '1'. A switch can be used to emulate an input signal. The clock should be adjusted to a sufficiently low frequency (~1Hz) so that the LED one-clock-cycle flash can be seen clearly.

Design and Implementation

The edge detector module can be implemented using two blocks – a combinational block which computes the FSM's next state and a sequential block which updates and stores the state on the clock edge. The module definition is given below but needs to be completed as guided by the commented lines.

State Diagram and Verilog Code - Start by drawing a state transition diagram consisting of 3 states that will be required to represent this system. It is not necessary to draw the state table as the Verilog implementation can be written directly from the state diagram using the case and if-else statements. Also, note that the input called "*signal*" is the one on which the rising edge needs to be detected and "*outedge*" is the output on which the status of the rising edge needs to be shown.

```
1  `timescale 1ns / 1ps
2
3  module rising_edge_detector(
4      input clk,
5      input signal,
6      input reset,
7      output reg outedge
8  );
9      wire slow_clk ;
10
11     reg [1:0] state ;
12     reg [1:0] next_state ;
13
14     clkdiv c1(clk, reset, slow_clk );
15
```

```
16      // Combinational logic.
17
18      always @(*) begin
19
20          case (state)
21
22              2'b00   : begin
23                  outedge = 1'b0;
24                  if (~signal)
25                      next_state =       ,
26                  else
27                      next_state =
28                  end
29
30              2'b01   : /*Insert your code here */;
31
32              2'b10   : /*Insert your code here */;
33
34              default : begin
35                  next_state = 2'b00 ;
36                  outedge = 1'b0;
37                  end
38
39        endcase
40
41      end
42
43      // Sequential logic
44
49  always@(posedge slow_clk or posedge reset)begin
50      if(reset)
51          state <= 2'b00;
52      else
53          state <= next_state;
54  end
```

If the board's internal clock is directly used, the flashing of the LED cannot be seen because the frequency, 100MHz, is too high for the human eye to detect a one-clock flash of the LED. This problem can be solved by using a much slower clock (~1Hz). Hence, the slower clock needs to be derived from the faster clock using a method known as clock division. The module definition of the clock divider circuit is given below.

```
1  `timescale 1ns / 1ps
2
3  module clkdiv(
4      input clk,
5      input reset,
6      output clk_out
7  );
8
9      reg [15:0] COUNT;
10
11       assign clk_out=COUNT[15];
12
13       always @(posedge clk)
14       begin
15       if (reset)
16         COUNT = 0;
17       else
18         COUNT = COUNT + 1;
19       end
20
21     endmodule
```

In the above module, the signal "clk" is coming from the board, while the signal "clk_out" is the one that drives the implemented FSM.

Important Note –> In the code above, the counter is specified to have the width of 16 bits. This is a somewhat arbitrary number. The actual width which is required for observing the flashing of LED is likely to be even larger. However, if such a counter is directly simulated, the simulator will need to go through 2^16 cycles before setting the counter, which will result in a very long simulation time. To allow the simulation to conclude within reasonable time, the width of the counter should be set to 2 bits for behavioral simulation. However, ***when the design is to be implemented on the board, the counter width needs to be adjusted such that the resulting clock frequency meets the problem specifications***, i.e., the flashing of the LED can be clearly observed with naked eye (the width may not be 16 for that).

Behavioral Simulation - Please create a test bench to conduct the behavioral simulation of the FSM. Pay attention to the above note while doing the behavioral simulation of the module.

Implementation - After behavioral simulation, the constraints file needs to be created according to the following mapping specifications –

- "*clk*" input port to clock pin W5
- "*signal*" input port to switch V17
- "*reset*" input port to button U18

"*outedge*" output port to LED U16

Using the constraints file, the design needs to be implemented to generate the bitstream file. The Basys3 board needs to be programmed with the generated bitstream file for the verification of the design on the board.

PART 3. Controller for Sequential Display of 4 Digits

Objective

The Digilent Basys3 Board contains four seven-segment LED displays with decimal points. The goal of this experiment is to sequentially (one-at-a-time) display four different numbers on the four LED displays. The design challenge comes from the fact that while all four 4-bit numbers are available at the same time, only one number should be shown (on one of four 7-segment displays) at any given time. This problem will be solved by designing a controller (FSM) circuit to control the display of the four displays sequentially using a time-multiplexing scheme, in which the four displays share eight common signals to light the segments but need to be carefully controlled via their enable signals to allow only one display to be on at any given time.

Design and Implementation

Block Diagram and Verilog Code - From lab 3, recall that the signals controlling the 7-segment display are active-low (i.e., enabled when a signal is '0'). The example of displaying '3' on the right-most LED is shown in Figure 4. Note that the enable signal (i.e., an) is '1110' for this case.

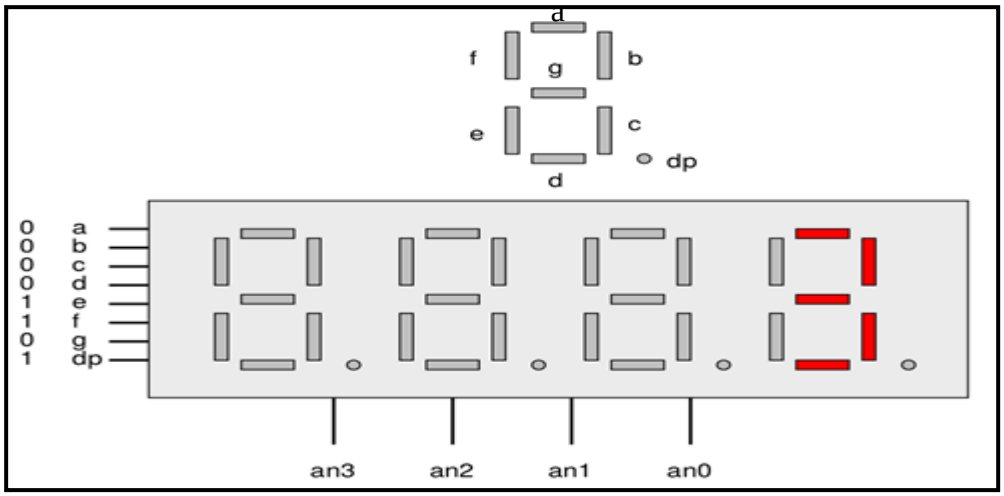


Figure 4. '3' displayed on one seven-segment display

Time-multiplexing the four seven segment displays means that the four displays are enabled in turn and not all at a time, as shown in the simplified timing

diagram in Figure 5. The frequency of the enable signal needs to be sufficiently low so that the human eye can distinguish the on and off intervals of the 7-segment displays and would perceive that all four displays are lit one after the other. Therefore, at the end of the design process, it is required that we adjust the frequency of the enable signal to allow easy visual observation.

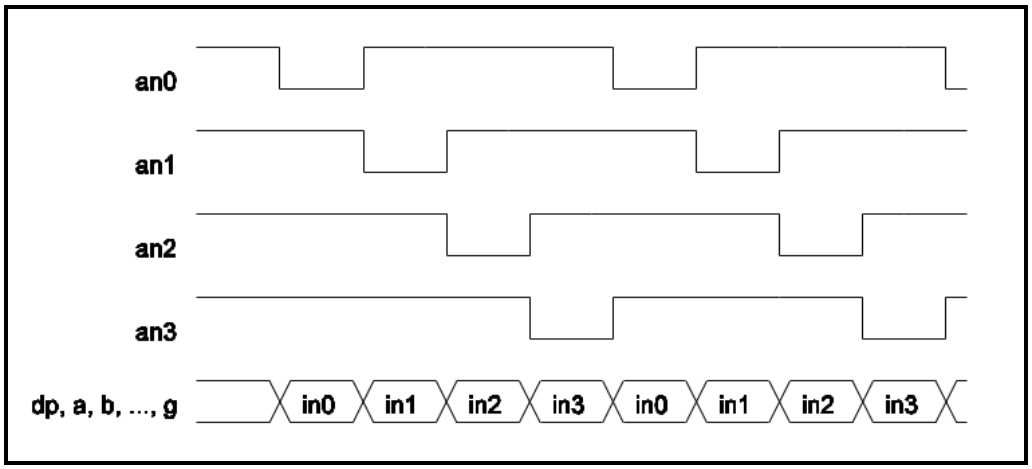


Figure 5. Time-multiplexing: the 4 enable signals (an0-an3) made active (low) sequentially

The block diagram in Figure 6 below has two objectives: (a) places the desired system in the context of a general controller architecture that involves a state register (D Flip Flops) and the combinational circuit for outputs and next state; and (b) suggests one possible realization of the required functionality. This figure should be used as a guide to implement the system.

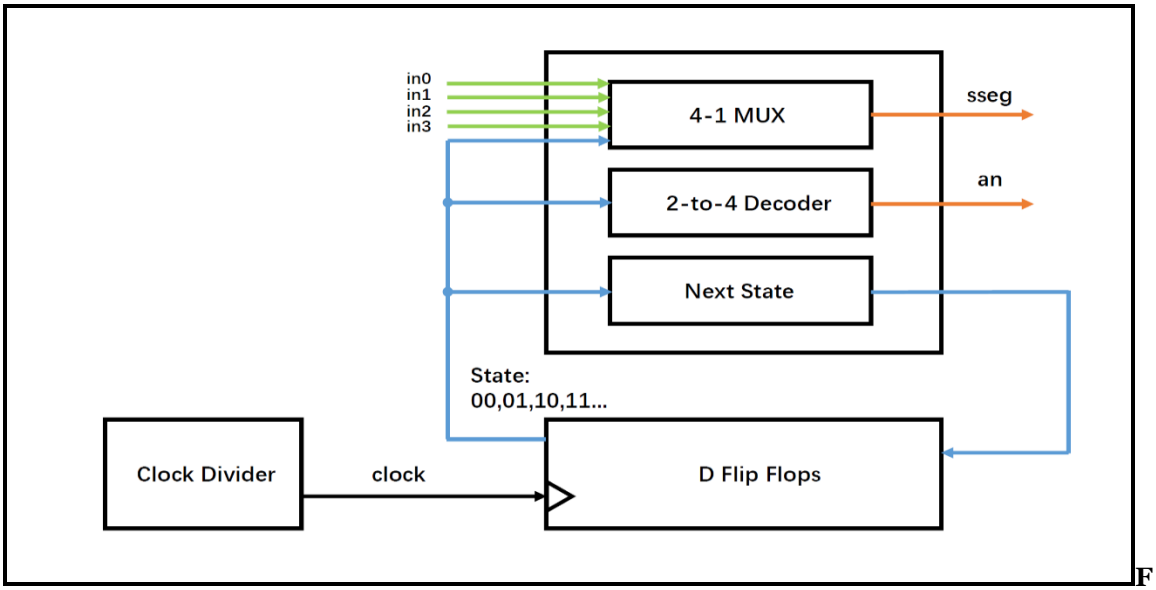


Figure 6. The display controller architecture and suggested implementation

To convert a binary code into a 7-segment hex code, the following Verilog code serves as a look-up table. The code below must be used as a template and should be completed at the commented places. (Note that the seven-segment display can take values A, b, C, d, E, F for inputs greater than 9 respectively, as this is the hex code and not BCD. Hence, it should not be turned off for inputs greater than 9 as was done in lab 3 and the segments corresponding to the letters should be on.)

```
1  `timescale 1ns / 1ps
2
3  module hexto7segment(
4      input [3:0] x,
5      output reg [6:0] r
6      );
7      always @(*)
8          case (x)
9              4'b0000 :  r = 7'b0000001;
10             4'b0001 :  /*Your code here*/;
11             4'b0010 :  /*Your code here*/;
12             4'b0011 :  /*Your code here*/;
13             4'b0100 :  /*Your code here*/;
14             4'b0101 :  /*Your code here*/;
15             4'b0110 :  /*Your code here*/;
16             4'b0111 :  /*Your code here*/;
17             4'b1000 :  /*Your code here*/;
18             4'b1001 :  /*Your code here*/;
19             4'b1010 :  /*Your code here*/;
20             4'b1011 :  /*Your code here*/;
21             4'b1100 :  /*Your code here*/;
22             4'b1101 :  /*Your code here*/;
23             4'b1110 :  /*Your code here*/;
24             4'b1111 :  /*Your code here*/;
25         endcase
26 endmodule
```

After the design of the hex to 7-segment decoder above, the next step is to design the top module for this system, as shown in Figure 6. The structure of the code is given below and it needs to be completed at the commented places. Note that there are 4 inputs for each 7-segment display and hence, the bus width of sw input is 16. Four hex to 7-segment decoders will be designed to be instantiated inside the top module to decode each set of 4 input bits. Further, note that there is also a clk_div_disp module instantiated, which has the same functionality as the clkdiv module designed for the second part but may have different width requirement.

```
`timescale 1ns / 1ps

module time_multiplexing_main(
    input clk,
    input reset,
    input [15:0] sw,
    output [3:0] an,
    output [6:0] sseg);

    wire [6:0]  in0, in1, in2, in3;
    wire slow_clk;

    // Module instantiation  of hexto7segment decoder
    hexto7segment c1 (.x(sw[3:0]), .r(in0));
    hexto7segment c2 (.x(sw[7:4]), .r(in1));
    hexto7segment c3 (.x(sw[11:8]), .r(in2));
    hexto7segment c4 (.x(sw[15:12]), .r(in3));

    // Module instantiation of the clock divider
    clk_div_disp c5 (.clk(clk), .reset(reset), .slow_clk(slow_clk));

    // Module instantiation of the multiplexer
    time_mux_state_machine c6(
      .clk (slow_clk),
      .reset (reset),
      .in0 (in0),
      .in1 (in1),
      .in2 (in2),
      .in3 (in3),
      .an (an),
      .sseg (sseg));
endmodule
```

The next step is to implement the controller circuit with the top module called time_mux_state_machine. The structure of the Verilog code of the module is given below and needs to be completed at the commented places.

```
`timescale 1ns / 1ps

module time_mux_state_machine(
    input clk,
    input reset,
    input [6:0] in0,
    input [6:0] in1,
    input [6:0] in2,
    input [6:0] in3,
    output reg [3:0] an,
    output reg [6:0] sseg
    );

    reg [1:0] state ;
    reg [1:0] next_state;

```

```
17      always @ (*) begin
18          case(state)           // State transition
19              2'b00: next_state = 2'b01;
20              2'b01: /*Your code here*/;
21              2'b10: /*Your code here*/;
22              2'b11: /*Your code here*/;
23          endcase
24       end
25
26      always @(*) begin
27          case (state)          // Multiplexer
28              2'b00 : sseg = in0;
29              2'b01 : /*Your code here*/;
30              2'b10 : /*Your code here*/;
31              2'b11 : /*Your code here*/;
32          endcase
33
34          case (state)          // Decoder
35              2'b00 : an = 4'b1110;
36              2'b01 : /*Your code here*/;
37              2'b10 : /*Your code here*/;
38              2'b11 : /*Your code here*/;
39          endcase
40      end
41
42      always @(posedge clk or posedge reset) begin
43          if(reset)
44              state <= 2'b00;
45          else
46              state <= next_state;
47      end
48  endmodule
```

Important Note -> It is required that the four 7-segment displays are ON and show different numbers one after the other and not simultaneously, i.e., only one of the displays should be on at any given time. Therefore, the internal clock of the board should be divided using the clock divider circuit and then used, like in part 2. It is required that you come up with the appropriate counter width based on your experiments.

Behavioral Simulation - Please create a test bench to conduct the behavioral simulation of the FSM. Please note that the counter width should be made small for simulation purposes to avoid high simulation time, as discussed in part 2.

Implementation - After behavioral simulation, the constraints file needs to be created according to the following mapping specifications –

- "*clk*" input port to internal clock W5
- "*reset*" input port to button U18
- "*sw[0],..,sw[15]*" input ports to switches V17, V16, W16, W17, W15, V15, W14, W13, V2, T3, T2, R3, W2, U1, T1, R2 respectively
- "*an[0], an[1], an[2], an[3]*" output ports to U2, U4, V4, W4 respectively in 7-segment display

- "*sseg[6],..,sseg[0]*" to W7, W6, U8, V8, U5, V5, U7 respectively in 7-segment display

Using the constraints file, the design needs to be implemented to generate the bitstream file. The Basys3 board needs to be programmed with the generated bitstream file for the verification of the design on the board.