**California Polytechnic State University Pomona**

**DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING**

# Dgtl Circuit Dsgn Verilog Lab

**ECE 3300L Section E01**

**Report # 3**

**Prepared by**

**Arvin Ghaloosian (017153422)**
**Vittorio Huizar (016826784)**
Group H

**Presented to**

**Mohamed Aly (Mohamed El-Hadedy)**

**Due - July 2nd, 2025**

## DESIGN OVERVIEW:

For this lab, we designed and implemented a 16-to-1 multiplexer using both hardware description language (HDL) design principles and button-controlled selector logic. The output of the multiplexer (LED0) displays the value of one of the 16 switches (SW[15:0]), based on a 4-bit selector. The selector value is controlled entirely using the four push buttons (btnU, btnD, btnL, btnR) on the Nexys A7 board.

The design allows users to cycle through all 16 selector values by pressing button combinations, with the output updating dynamically.

## MODULE HIERARCHY:

- Top Module: top_mux_lab3
- Toggle Switch Debounce Module: toggle_switch
- Debouncer Module: debounce
- 16-to-1 Multiplexer: mux16x1
- 2-to-1 Multiplexer (used internally): mux2x1

## HOW THE DESIGN WORKS:

Inputs:
- clk (100 MHz system clock)
- rst (CPU reset button)
- SW[15:0] (switch inputs)
- btnU, btnD, btnL, btnR (push buttons for selector control)

Output:
- LED0 (Multiplexer output)

Button Functionality:
Each button toggles a single bit of the 4-bit selector sel[3:0].
- btnD → sel[0]
- btnR → sel[1]
- btnL → sel[2]
- btnU → sel[3]

The final selected switch (SW[sel]) drives LED0.

## Top Module:

```
module top_mux_lab3 (

  input clk,

  input rst,

  input [15:0] SW,

  input btnU, btnD, btnL, btnR,

  output LED0,

  output LED1,

  output LED2,

  output LED3,

  output LED4

);

  wire cleanD, cleanR, cleanL, cleanU;

  wire [3:0] sel;

  // Invert the active-low reset signal from the board to active-high for logic

  wire rst_active;

  assign rst_active = ~rst;

  // Debounce each button

  debounce dbD (.clk(clk), .btn_in(btnD), .btn_clean(cleanD));

  debounce dbR (.clk(clk), .btn_in(btnR), .btn_clean(cleanR));

  debounce dbL (.clk(clk), .btn_in(btnL), .btn_clean(cleanL));

  debounce dbU (.clk(clk), .btn_in(btnU), .btn_clean(cleanU));

  // Toggle each selector bit with debounced buttons

  toggle_switch t0 (.clk(clk), .rst(rst_active), .btn_raw(cleanD), .state(sel[0]));

  toggle_switch t1 (.clk(clk), .rst(rst_active), .btn_raw(cleanR), .state(sel[1]));

  toggle_switch t2 (.clk(clk), .rst(rst_active), .btn_raw(cleanL), .state(sel[2]));
  toggle_switch t3 (.clk(clk), .rst(rst_active), .btn_raw(cleanU), .state(sel[3]));

  // 16x1 MUX

  mux16x1 mux (.in(SW), .sel(sel), .out(LED0));

endmodule
```

## Explanation:

The top_mux_lab3.v module implements a 16-to-1 multiplexer controlled by pushbuttons on the Nexys A7 board. It takes the 16 slide switches (SW[15:0]) as data inputs and routes one selected switch to a single output LED (LED0). The 4-bit selector (sel[3:0]) that controls the MUX is toggled using the four pushbuttons (btnU, btnD, btnL, and btnR), where each button toggles a corresponding bit of the selector. To prevent input noise and bouncing issues from affecting the toggle logic, each button signal is first passed through a debounce module, generating clean button signals. Additionally, because the onboard reset button is active-low, the reset signal (rst) is inverted internally (rst_active = ~rst) so that the toggle switch modules receive an active-high reset input, ensuring proper reset behavior. When a user presses the buttons, the selector bits change state, choosing a different switch input, and the current value from the selected switch is shown on LED0. This design allows real-time manual control of the MUX select lines using button presses, demonstrating both digital selection logic and hardware input handling.

## Debounce module:

```
module debounce (

  input clk,

  input btn_in,

  output reg btn_clean

);

  reg [2:0] shift_reg;

  always @(posedge clk) begin

    shift_reg <= {shift_reg[1:0], btn_in};

    if (shift_reg == 3'b111)

      btn_clean <= 1;

    else if (shift_reg == 3'b000)
    btn_clean <= 0;

  end

endmodule
```

## Explanation:

The debounce.v module is a simple hardware debouncer used to clean up noisy button inputs from mechanical pushbuttons on the FPGA board. When a button is pressed or released, mechanical contacts can bounce, causing multiple rapid transitions that would be falsely detected as multiple presses. This module samples the button input (btn_in) on every rising edge of the clock (clk) and stores the last three sampled values in a 3-bit shift register (shift_reg). The output (btn_clean) is set high only when all three sampled values are consistently high (3'b111), and it is set low when all three samples are low (3'b000). This ensures that only stable, clean transitions are passed through, preventing glitches and false triggering in the rest of the circuit that relies on button inputs.

## Toggle_switch module:

```
module toggle_switch (

  input clk,

  input rst,

  input btn_raw,

  output reg state

);

  wire btn_clean;

  reg btn_prev;

  debounce db (

    .clk(clk),

    .btn_in(btn_raw),

    .btn_clean(btn_clean)

  );

  always @(posedge clk) begin
```

```
    if (rst) begin
      state <= 0;
      btn_prev <= 0;
    end else begin
      if (btn_clean & ~btn_prev)
        state <= ~state;
      btn_prev <= btn_clean;
    end
  end
endmodule
```

## Explanation:

The toggle_switch.v module creates a toggleable state from a pushbutton input. It takes in a raw button signal (btn_raw), along with the clock (clk) and an active-high reset (rst). Inside the module, it first instantiates the debounce module to clean the incoming button signal, producing a stable btn_clean output. The toggle logic then uses edge detection by comparing the current and previous states of btn_clean. On each rising edge of the clock, if the button was just pressed (detected as a rising edge on the cleaned signal), the module toggles the state output from 0 to 1 or from 1 to 0. The reset signal (rst) can asynchronously clear the state back to 0. This allows each button press to flip the state, making it useful for incrementing selector bits in state-controlled designs like the 16-to-1 multiplexer in this lab.

## 16x1_mux module:

```
module mux16x1 (
  input [15:0] in,
  input [3:0] sel,
  output out
);
  wire [7:0] level1;
  wire [3:0] level2;
  wire [1:0] level3;
```

```
 genvar i;

 generate

  for (i = 0; i < 8; i = i + 1)

   mux2x1 m1 (.a(in[2*i]), .b(in[2*i+1]), .sel(sel[0]), .y(level1[i]));

  for (i = 0; i < 4; i = i + 1)

   mux2x1 m2 (.a(level1[2*i]), .b(level1[2*i+1]), .sel(sel[1]), .y(level2[i]));

  for (i = 0; i < 2; i = i + 1)

   mux2x1 m3 (.a(level2[2*i]), .b(level2[2*i+1]), .sel(sel[2]), .y(level3[i]));

  mux2x1 m4 (.a(level3[0]), .b(level3[1]), .sel(sel[3]), .y(out));

 endgenerate

endmodule
```

## Explanation:

The mux16x1.v module implements a 16-to-1 multiplexer by hierarchically connecting multiple 2-to-1 multiplexers. It takes a 16-bit input vector (in[15:0]), a 4-bit selector (sel[3:0]), and outputs a single bit (out). The selection process occurs in four levels. First, eight 2-to-1 muxes select between adjacent input pairs based on sel[0], generating an 8-bit intermediate signal (level1). Next, four 2-to-1 muxes use sel[1] to reduce level1 to four signals (level2). Then, two more 2-to-1 muxes use sel[2] to further reduce to two signals (level3). Finally, a single 2-to-1 mux at the top level selects the final output using sel[3]. This hierarchical structure efficiently implements the full 16-to-1 selection using basic 2-to-1 building blocks defined in the mux2x1 module.

## 2x1_mux module:

```
module mux2x1 (

 input a, b,

 input sel,

 output y

);

 wire nsel, a1, b1;
```

```
  not (nsel, sel);

  and (a1, a, nsel);

  and (b1, b, sel);

  or  (y, a1, b1);

endmodule
```

## Explanation:

The mux2x1.v module implements a basic 2-to-1 multiplexer using simple logic gates. It takes two single-bit inputs (a and b), a single-bit selector signal (sel), and outputs one selected value (y). The logic works as follows: when sel is 0, the output y follows input a; when sel is 1, the output follows input b. This selection is done by inverting the selector (nsel), then ANDing it with a and ANDing sel with b, and finally ORing the two results to produce the correct output. This 2-to-1 mux is used as a fundamental building block within the larger 16-to-1 multiplexer (mux16x1.v).

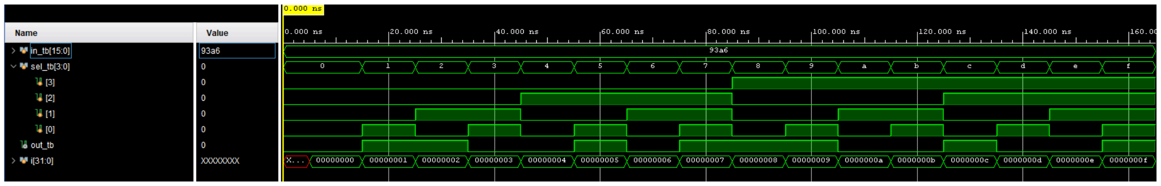
## XDC Snippet:

```
## Clock

set_property PACKAGE_PIN E3 [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]

## Reset

set_property PACKAGE_PIN C12 [get_ports rst]

set_property IOSTANDARD LVCMOS33 [get_ports rst]

## Switches (SW0 - SW15)

set_property PACKAGE_PIN J15 [get_ports {SW[0]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[0]}]

set_property PACKAGE_PIN L16 [get_ports {SW[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[1]}]

set_property PACKAGE_PIN M13 [get_ports {SW[2]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[2]}]
set_property PACKAGE_PIN R15 [get_ports {SW[3]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[3]}]

set_property PACKAGE_PIN R17 [get_ports {SW[4]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[4]}]

set_property PACKAGE_PIN T18 [get_ports {SW[5]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[5]}]

set_property PACKAGE_PIN U18 [get_ports {SW[6]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[6]}]

set_property PACKAGE_PIN R13 [get_ports {SW[7]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[7]}]

set_property PACKAGE_PIN T8 [get_ports {SW[8]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[8]}]

set_property PACKAGE_PIN U8 [get_ports {SW[9]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[9]}]

set_property PACKAGE_PIN R16 [get_ports {SW[10]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[10]}]

set_property PACKAGE_PIN T13 [get_ports {SW[11]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[11]}]

set_property PACKAGE_PIN H6 [get_ports {SW[12]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[12]}]

set_property PACKAGE_PIN U12 [get_ports {SW[13]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[13]}]

set_property PACKAGE_PIN U11 [get_ports {SW[14]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[14]}]

set_property PACKAGE_PIN V10 [get_ports {SW[15]}]

set_property IOSTANDARD LVCMOS33 [get_ports {SW[15]}]

## Buttons (btnU, btnD, btnL, btnR)

set_property PACKAGE_PIN M18 [get_ports {btnU}]
set_property IOSTANDARD LVCMOS33 [get_ports {btnU}]

set_property PACKAGE_PIN P18 [get_ports {btnD}]

set_property IOSTANDARD LVCMOS33 [get_ports {btnD}]

set_property PACKAGE_PIN P17 [get_ports {btnL}]

set_property IOSTANDARD LVCMOS33 [get_ports {btnL}]

set_property PACKAGE_PIN M17 [get_ports {btnR}]

set_property IOSTANDARD LVCMOS33 [get_ports {btnR}]

## LED0 (Mux Output)

set_property PACKAGE_PIN H17 [get_ports {LED0}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED0}]

## LED1-LED4 (sel[0]-sel[3] Debug)

set_property PACKAGE_PIN K15 [get_ports {LED1}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED1}]

set_property PACKAGE_PIN J13 [get_ports {LED2}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED2}]

set_property PACKAGE_PIN N14 [get_ports {LED3}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED3}]

set_property PACKAGE_PIN R18 [get_ports {LED4}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED4}]
```

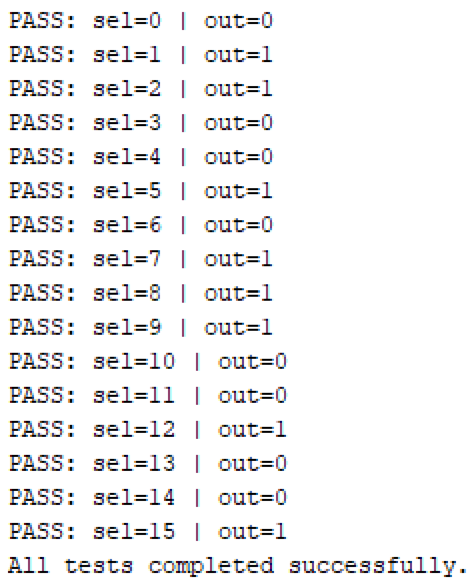
## Explanation:

This .xdc constraints file assigns the physical FPGA pins of the Nexys A7-100T board to the corresponding ports in the top_mux_lab3.v module. The 100 MHz system clock (clk) is mapped to pin E3, while the active-low CPU reset button (rst) is assigned to pin C12. All sixteen slide switches (SW[15:0]) are individually mapped to their appropriate I/O pins to serve as inputs for the 16-to-1 multiplexer. The four pushbuttons (btnU, btnD, btnL, and btnR), which control the selector bits for the multiplexer, are also mapped to their designated FPGA pins. The primary output (LED0), which displays the MUX output, is connected to pin H17. Additionally, four extra LEDs (LED1 through LED4) are mapped to indicate the current selector state (sel[3:0]) for easier debugging and visualization of button-controlled selector changes. This .xdc configuration ensures that all necessary signals from the Verilog design are correctly routed to the hardware inputs and outputs on the Nexys A7 board.

## TEST BENCH:



Name
Value
in_tb[15:0]
93a6
sel_tb[3:0]
0
[3]
0
[2]
0
[1]
0
[0]
0
out_tb
0
i[31:0]
XXXXXXXX
0.000 ns
20.000 ns
40.000 ns
60.000 ns
80.000 ns
100.000 ns
120.000 ns
140.000 ns
0 1 2 3 4 5 6 7 8 9 a b c d e f
00000000 00000001 00000002 00000003 00000004 00000005 00000006 00000007 00000008 00000009 0000000a 0000000b 0000000c 0000000d 0000000e 0000000f

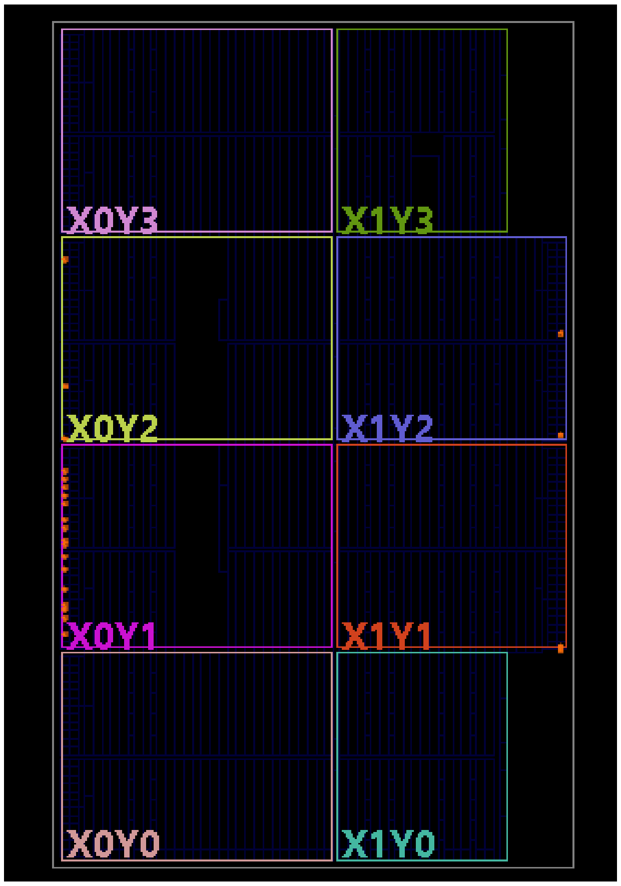


```
PASS: sel=0 | out=0
PASS: sel=1 | out=1
PASS: sel=2 | out=1
PASS: sel=3 | out=0
PASS: sel=4 | out=0
PASS: sel=5 | out=1
PASS: sel=6 | out=0
PASS: sel=7 | out=1
PASS: sel=8 | out=1
PASS: sel=9 | out=1
PASS: sel=10 | out=0
PASS: sel=11 | out=0
PASS: sel=12 | out=1
PASS: sel=13 | out=0
PASS: sel=14 | out=0
PASS: sel=15 | out=1
All tests completed successfully.
```

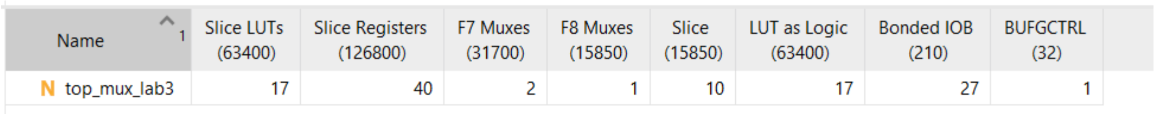
We developed a self-checking testbench that simulates all 16 selector values. It does this by simulating button presses in sequence, which mimic actual hardware button toggling. At each selector state, the corresponding switch is set high, and the testbench checks if LED0 matches the expected output.

This helped catch timing issues and debounce errors.

## SYNTHESIS SCHEMATIC:



## RESOURCE UTILIZATION:



| Name | Slice LUTs (63400) | Slice Registers (126800) | F7 Muxes (31700) | F8 Muxes (15850) | Slice (15850) | LUT as Logic (63400) | Bonded IOB (210) | BUFGCTRL (32) |
|---|---|---|---|---|---|---|---|---|
| N top_mux_lab3 | 17 | 40 | 2 | 1 | 10 | 17 | 27 | 1 |

## CONTRIBUTIONS:

Arvin Ghaloosian (50%)
- Wrote top-level module and mux design
- Conducted hardware testing on Nexys A7 board
- Verified timing and output in hardware
- Recorded video demo and added comments

Vittorio Huizar (50%)
- Implemented toggle switch and debounce modules
- Handled Vivado project setup and pin constraints
- Created and debugged the self-checking testbench
- Documented simulation and synthesis results

## REFLECTION:

This lab helped reinforce the importance of debouncing mechanical inputs, and how small glitches can cause major functional errors in digital designs. The toggle logic taught us how button states can be captured and stored, effectively simulating a simple state machine for selection.

We also gained more experience with hierarchical module design and multi-level mux implementations, building from 2x1 muxes up to a 16x1 tree structure.

Creating the testbench was more difficult than previous labs since we had to simulate sequential button presses while monitoring output over time.

## Link to youtube video demo:

https://youtu.be/rtr-PkvdIVU