CalPolyPomona
College of Engineering

Lab 3 Report

by

Jonathan Huynh #016137719

Adam Godfrey #015981472

Instructor: Dr. Mohamed Aly

Class: ECE 3300L .E02-OU - Verilog Design

July 2nd, 2025

Code with Explanation:

MUX2x1 Gate-Level Verilog

Code

```
module mux2x1_gatesMessy(
   input wire inputWireA,
   input wire inputWireB,
   input wire selectorSignal,
   output wire selectedOutput
);

   wire inverseOfSelector;
   wire wireAWhenSelectorLow;
   wire wireBWhenSelectorHigh;

   not(inverseOfSelector, selectorSignal);
   and(wireAWhenSelectorLow,
         inputWireA, inverseOfSelector);
   and(wireBWhenSelectorHigh,
         inputWireB, selectorSignal);
   or(selectedOutput,
         wireAWhenSelectorLow,
         wireBWhenSelectorHigh);
endmodule
```

Explanation:

This module **mux2x1_gatesMessy** implements a 2-to-1 multiplexer using basic logic gates: NOT, AND, and OR. It takes two data inputs (*inputWireA* and *inputWireB*), a control signal (selectorSignal), and produces one output (*selectedOutput*) based on the value of the selector. When selectorSignal is 0, the output is equal to inputWireA. When selectorSignal is 1, the output is equal to inputWireB.

Internally, the module first computes the inverse of the selector using a NOT gate. This inverted signal is then ANDed with *inputWireA*, enabling *inputWireA* to pass through only when the selector is low. Similarly, *inputWireB* is ANDed with the original selector signal, allowing it through only when the selector is high. The results of these two AND operations are then combined using an OR gate to produce the final output.

This code uses gate-level logic as per the instructions, however, this same functionality can be implemented more cleanly using a conditional operator or behavioral logic, making the code easier to read and maintain. This gate-level approach is useful for understanding how multiplexers work at the hardware level.

MUX16x1 Using Generate Loops

<u>Code</u>

```
module mux16x1_nested_generated(
   input wire [15:0] inputSwitchesAll,
   input wire [3:0] selectorBitsFromButtons,
   output wire finalMuxOutput
);

   wire [7:0] midStage1;
   wire [3:0] midStage2;
   wire [1:0] midStage3;

   genvar index;

   generate
      for (index = 0; index < 8; index = index + 1)
         mux2x1_gatesMessy m_stage1 (
         .inputWireA(inputSwitchesAll[2*index]),
         .inputWireB(inputSwitchesAll[2*index+1]),
         .selectorSignal(selectorBitsFromButtons[0]),
            .selectedOutput(midStage1[index])
         );

      for (index = 0; index < 4; index = index + 1)
                  mux2x1_gatesMessy m_stage2 (
         .inputWireA(midStage1[2*index]),
         .inputWireB(midStage1[2*index+1]),
         .selectorSignal(selectorBitsFromButtons[1]),
         .selectedOutput(midStage2[index])
         );

      for (index = 0; index < 2; index = index + 1)
         mux2x1_gatesMessy m_stage3 (
         .inputWireA(midStage2[2*index]),
         .inputWireB(midStage2[2*index+1]),
         .selectorSignal(selectorBitsFromButtons[2]),
         .selectedOutput(midStage3[index])
         );

      mux2x1_gatesMessy m_finalStage (
         .inputWireA(midStage3[0]),
         .inputWireB(midStage3[1]),
         .selectorSignal(selectorBitsFromButtons[3]),
         .selectedOutput(finalMuxOutput)
         );
   endgenerate
endmodule
```

<u>Explanation:</u>

The **mux16x1_nested_generated** module implements a 16-to-1 multiplexer using a multi-stage structure of smaller 2-to-1 multiplexers. It takes a 16-bit input vector (*inputSwitchesAll*), where each bit represents one of the 16 input values, and a 4-bit selector (*selectorBitsFromButtons*) that determines which of the 16 inputs is passed to the output (*finalMuxOutput*). The design uses the previously defined **mux2x1_gatesMessy** module and uses the generate block to instantiate the necessary logic in a structured, repetitive way.

The selection is done in four stages. In stage 1, eight 2-to-1 multiplexers select between pairs of inputs based on the least significant selector bit (*selectorBitsFromButtons[0]*), reducing the number of active signals from 16 to 8. Stage 2 further reduces the set from 8 to 4 using the next selector bit, and stage 3 narrows it down from 4 to 2. Finally, in stage 4, a single 2-to-1 multiplexer chooses between the remaining two signals based on the most significant selector bit, producing the final output.

Debounce Module

Code

```
module buttonDebouncer(
   input wire systemClock,
   input wire buttonInputRaw,
   output reg stableButtonOutput
);

   reg [2:0] debounceShiftRegister;

   always @(posedge systemClock) begin
         debounceShiftRegister <=
         {debounceShiftRegister[1:0],
         buttonInputRaw};

      if (debounceShiftRegister == 3'b111)
         stableButtonOutput <= 1;
      else if (debounceShiftRegister ==
         3'b000)
         stableButtonOutput <= 0;
   end
endmodule
```

Explanation:

The **buttonDebouncer** module is designed to eliminate noise and false triggers caused by mechanical bouncing in push-button signals. When a physical button is pressed or released, the electrical signal often fluctuates briefly before settling to a stable high or low state. This module uses a simple 3-sample shift register to ensure that only a consistently high or low signal over three clock cycles is considered a valid button press or release.

Internally, the module defines a 3-bit register called *debounceShiftRegister*. On every rising edge of the input *systemClock*, the newest value of *buttonInputRaw* is shifted into the least significant bit of the register, while older samples shift to the left. This keeps track of the last three samples of the button's state. If all three bits become 1 (i.e., the button has been consistently high for three cycles), the output *stableButtonOutput* is set to 1, indicating a stable press. If all three bits become 0, the output is cleared to 0, indicating a stable release.

Toggle Flip-Flop

Code

```
module toggleButtonSelector(
    input wire clockLineFromBoard,
    input wire resetWholeSystem,
    input wire bouncingButtonSignal,
    output reg toggledStateOut
);

    wire cleanButtonOutput;
    reg lastButtonMemory;

    buttonDebouncer debInst (
            .systemClock(clockLineFromBoard),
            .buttonInputRaw(bouncingButtonSignal),
            .stableButtonOutput(cleanButtonOutput)
    );

    always @(posedge clockLineFromBoard) begin
        if (resetWholeSystem) begin
            toggledStateOut <= 0;
            lastButtonMemory <= 0;
        end else begin
            if (cleanButtonOutput &&
            !lastButtonMemory)
                toggledStateOut <= ~toggledStateOut;
            lastButtonMemory <= cleanButtonOutput;
        end
    end
endmodule
```

Explanation:

The **toggleButtonSelector** module implements a toggle mechanism using a debounced button input. Its main function is to change (*toggle*) the state of the output (*toggledStateOut*) each time the button is pressed, but only once per press. This is commonly used in digital systems to switch modes or states using a single button.

To ensure reliable operation, the module first debounces the raw button input using the **buttonDebouncer** submodule. This removes glitches caused by mechanical bouncing, producing a clean and stable signal (*cleanButtonOutput*) that only changes when the button has actually been pressed or released. A register called *lastButtonMemory* stores the previous debounced button state, enabling edge detection.

On every rising edge of the clock (*clockLineFromBoard*), the module checks whether the system reset signal (*resetWholeSystem*) is active. If so, it resets both the toggle output and the last button state. If not, the module looks for a rising edge in the cleaned button signa. Specifically, it toggles the output only when the current button state is 1 and the last state was 0. This ensures that holding the button down does not repeatedly toggle the output, and only distinct presses trigger a change. The result is a clean and reliable toggle action on each button press.

Top-Level Module

Code

```
module top_module_mux_lab3(
    input wire clkFromBoard,
    input wire resetButtonActiveHigh,
    input wire [15:0] physicalSwitchInputs,
    input wire upBtn, downBtn, leftBtn, rightBtn,
    output wire singleLEDOutput
    );

    wire [3:0] selectorLineForMUX;

    toggleButtonSelector selBit0 (
            .clockLineFromBoard(clkFromBoard),

            .resetWholeSystem(resetButtonActiveHigh),
            .bouncingButtonSignal(downBtn),
            .toggledStateOut(selectorLineForMUX[0])
            );

            toggleButtonSelector selBit1 (
            .clockLineFromBoard(clkFromBoard),
            .resetWholeSystem(resetButtonActiveHigh),
            .bouncingButtonSignal(rightBtn),
            .toggledStateOut(selectorLineForMUX[1])
            );

            toggleButtonSelector selBit2 (
            .clockLineFromBoard(clkFromBoard),
            .resetWholeSystem(resetButtonActiveHigh),
            .bouncingButtonSignal(leftBtn),
            .toggledStateOut(selectorLineForMUX[2])
            );

            toggleButtonSelector selBit3
            (.clockLineFromBoard(clkFromBoard),
            .resetWholeSystem(resetButtonActiveHigh),
            .bouncingButtonSignal(upBtn),
            .toggledStateOut(selectorLineForMUX[3])
            );

            mux16x1_nested_generated dataMUX (
            .inputSwitchesAll(physicalSwitchInputs),
            .selectorBitsFromButtons(selectorLineForM
            UX),
            .finalMuxOutput(singleLEDOutput)
            );
endmodule
```

Explanation:

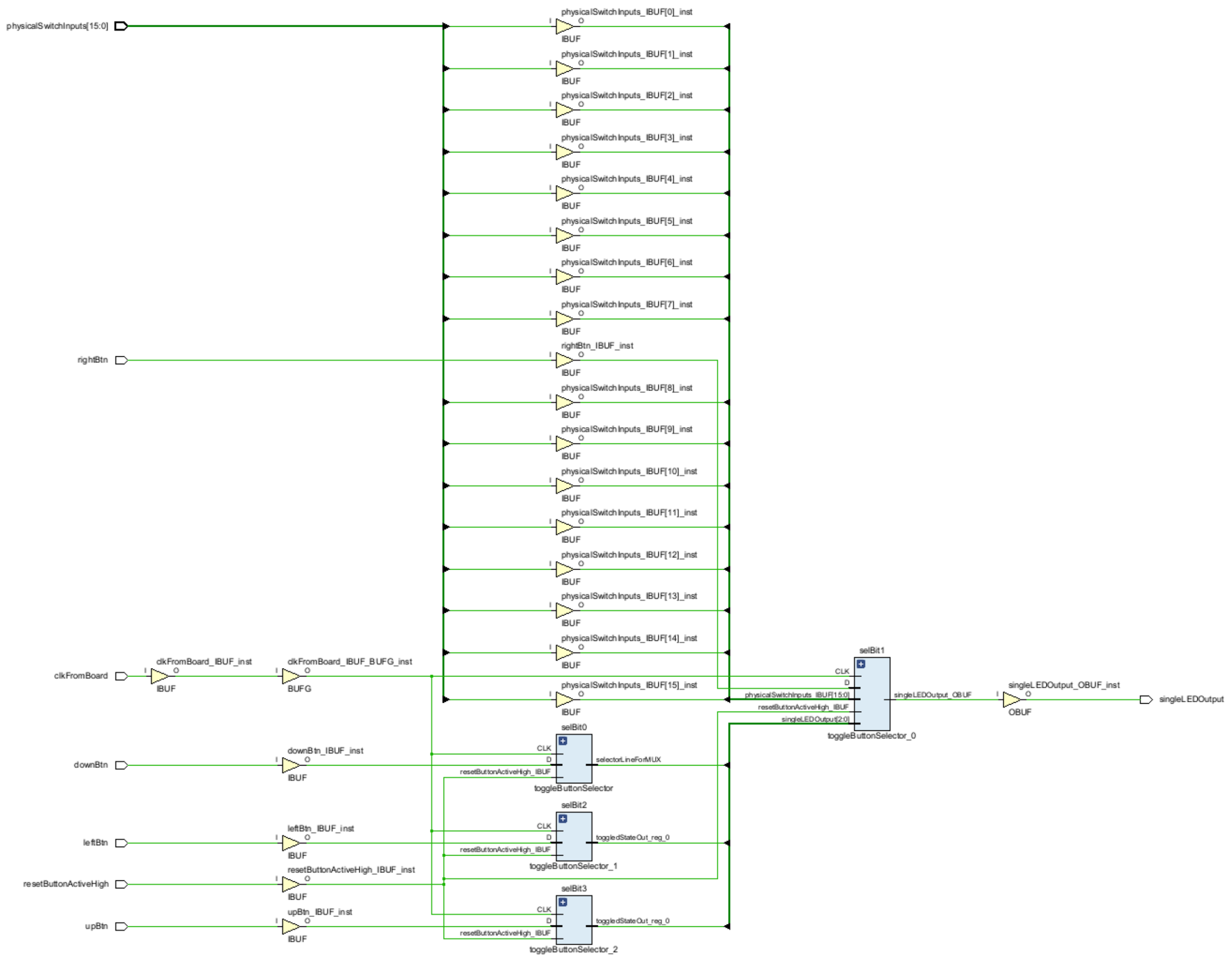
The **top_module_mux_lab3** is the top module of the verilog code that integrates multiple modules to create an interactive 16-to-1 multiplexer system controlled by physical buttons and switches. It allows a user to select one of 16 switch inputs using four directional buttons (*upBtn, downBtn, leftBtn, rightBtn*) and displays the selected value on a single output LED (*singleLEDOutput*). The system is synchronized using a board-provided clock (*clkFromBoard*) and can be reset with an active-high reset signal (*resetButtonActiveHigh*).

Each of the four buttons is connected to a **toggleButtonSelector** module, which debounces the button signal and turns it into a toggle bit. This toggle bit is stored in one bit of the 4-bit *selectorLineForMUX* signal. Specifically, pressing *downBtn* toggles selector bit 0, *rightBtn* toggles bit 1, *leftBtn* toggles bit 2, and *upBtn* toggles bit 3. These four bits together form a binary number that selects which of the 16 input switches (physicalSwitchInputs) is routed to the output through the **mux16x1_nested_generated** module.

The **mux16x1_nested_generated** is built from smaller 2-to-1 multiplexers that uses the 4-bit selector to choose one of the 16 switch inputs. The chosen input's value is then output on *singleLEDOutput*, effectively letting the user navigate through the switch inputs using button presses.

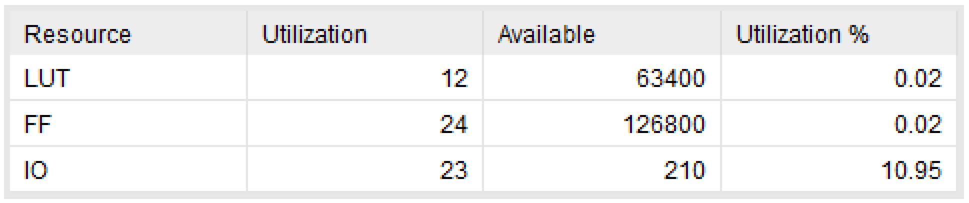
Screenshot Proofs:

Schematic:

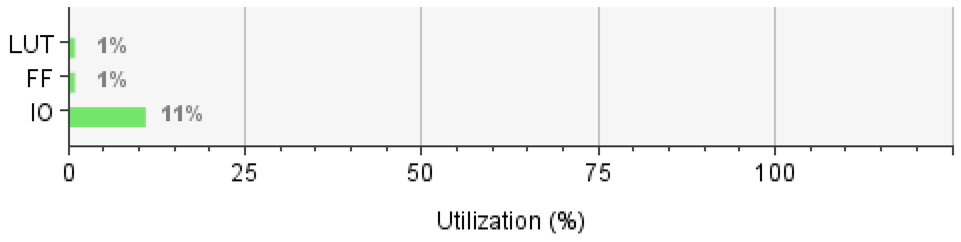


physicalSwitchInputs[15:0]
physicalSwitchInputs_IBUF[0]_inst
physicalSwitchInputs_IBUF[1]_inst
physicalSwitchInputs_IBUF[2]_inst
physicalSwitchInputs_IBUF[3]_inst
physicalSwitchInputs_IBUF[4]_inst
physicalSwitchInputs_IBUF[5]_inst
physicalSwitchInputs_IBUF[6]_inst
physicalSwitchInputs_IBUF[7]_inst
rightBtn
rightBtn_IBUF_inst
physicalSwitchInputs_IBUF[8]_inst
physicalSwitchInputs_IBUF[9]_inst
physicalSwitchInputs_IBUF[10]_inst
physicalSwitchInputs_IBUF[11]_inst
physicalSwitchInputs_IBUF[12]_inst
physicalSwitchInputs_IBUF[13]_inst
physicalSwitchInputs_IBUF[14]_inst
physicalSwitchInputs_IBUF[15]_inst
IBUF
clkFromBoard
clkFromBoard_IBUF_inst
clkFromBoard_IBUF_BUFG_inst
BUFG
selBit1
toggleButtonSelector_0
singleLEDOutput_OBUF
singleLEDOutput_OBUF_inst
OBUF
singleLEDOutput
downBtn
downBtn_IBUF_inst
selBit0
selectorLineForMUX
toggleButtonSelector
leftBtn
leftBtn_IBUF_inst
selBit2
toggledStateOut_reg_0
toggleButtonSelector_1
resetButtonActiveHigh
resetButtonActiveHigh_IBUF_inst
resetButtonActiveHigh_IBUF
upBtn
upBtn_IBUF_inst
selBit3
toggleButtonSelector_2

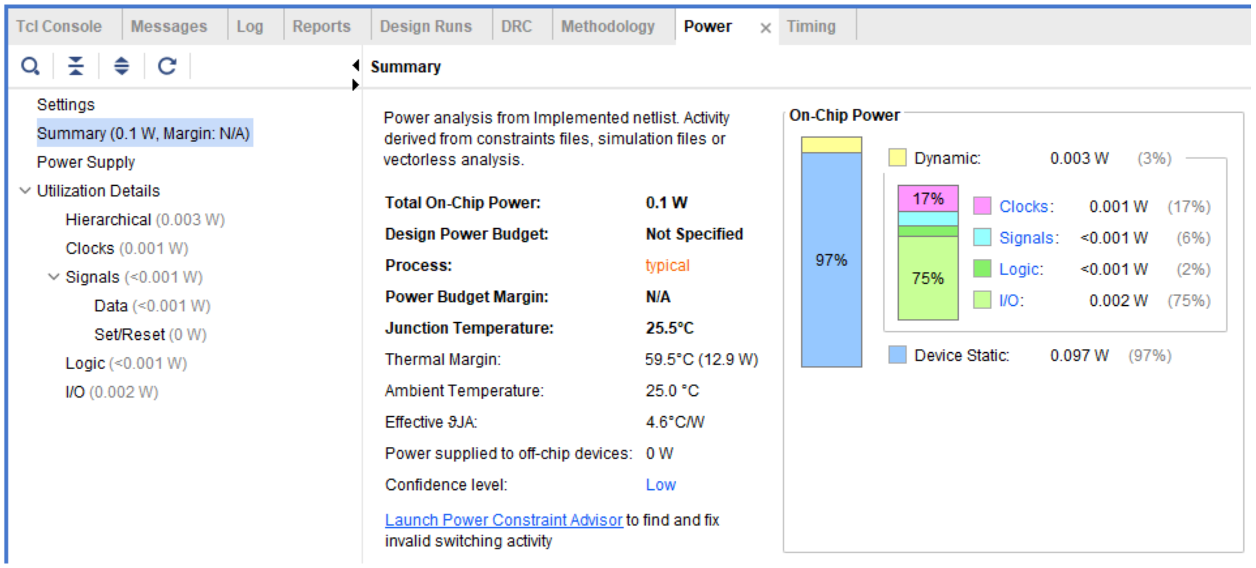
Resource Utilization Table:



Resource	Utilization	Available	Utilization %
LUT	12	63400	0.02
FF	24	126800	0.02
IO	23	210	10.95



Power Utilization:



Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

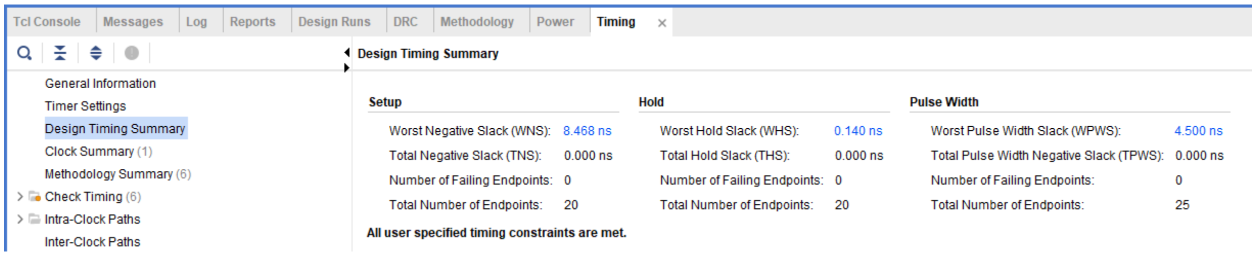
Total On-Chip Power:	**0.1 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.5°C**
Thermal Margin:	59.5°C (12.9 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	4.6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	0.003 W	(3%)
Clocks:	0.001 W	(17%)
Signals:	<0.001 W	(6%)
Logic:	<0.001 W	(2%)
I/O:	0.002 W	(75%)
Device Static:	0.097 W	(97%)

Timing Summary:

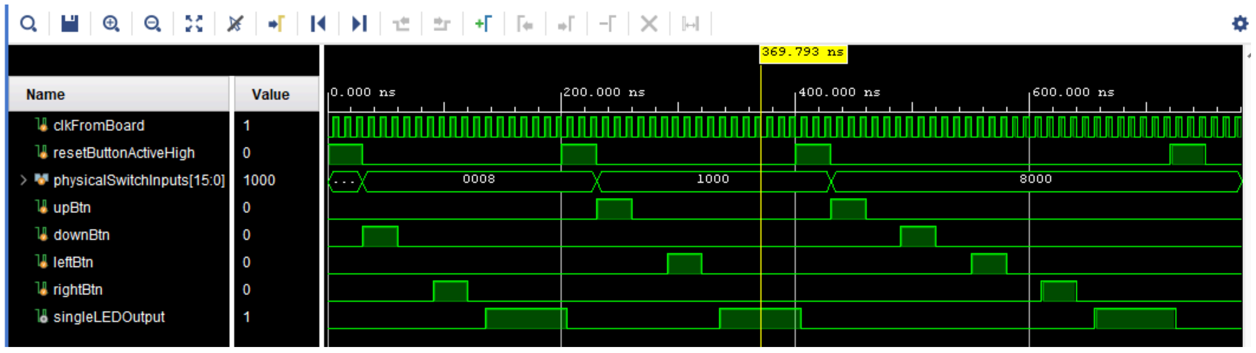


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	8.468 ns	Worst Hold Slack (WHS):	0.140 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	20	Total Number of Endpoints:	20	Total Number of Endpoints:	25

All user specified timing constraints are met.

Simulation Waveform:



Name	Value
clkFromBoard	1
resetButtonActiveHigh	0
physicalSwitchInputs[15:0]	1000
upBtn	0
downBtn	0
leftBtn	0
rightBtn	0
singleLEDOutput	1

Partner Contributions:

Team Member	**Contribution**	**% Effort**
Jonathan Huynh	Implementation, Debugging, Demo, Written Report	50%
Adam Godfrey	Verilog Code, Test Bench	50%