Cole Schreiner
Due 11/10/2023
CSE100/L - F23 Logic Design Laboratory
Lab Section: 03
Professor Martine Schlag
University of California, Santa Cruz

Lab 4 Report - LED Match

Description

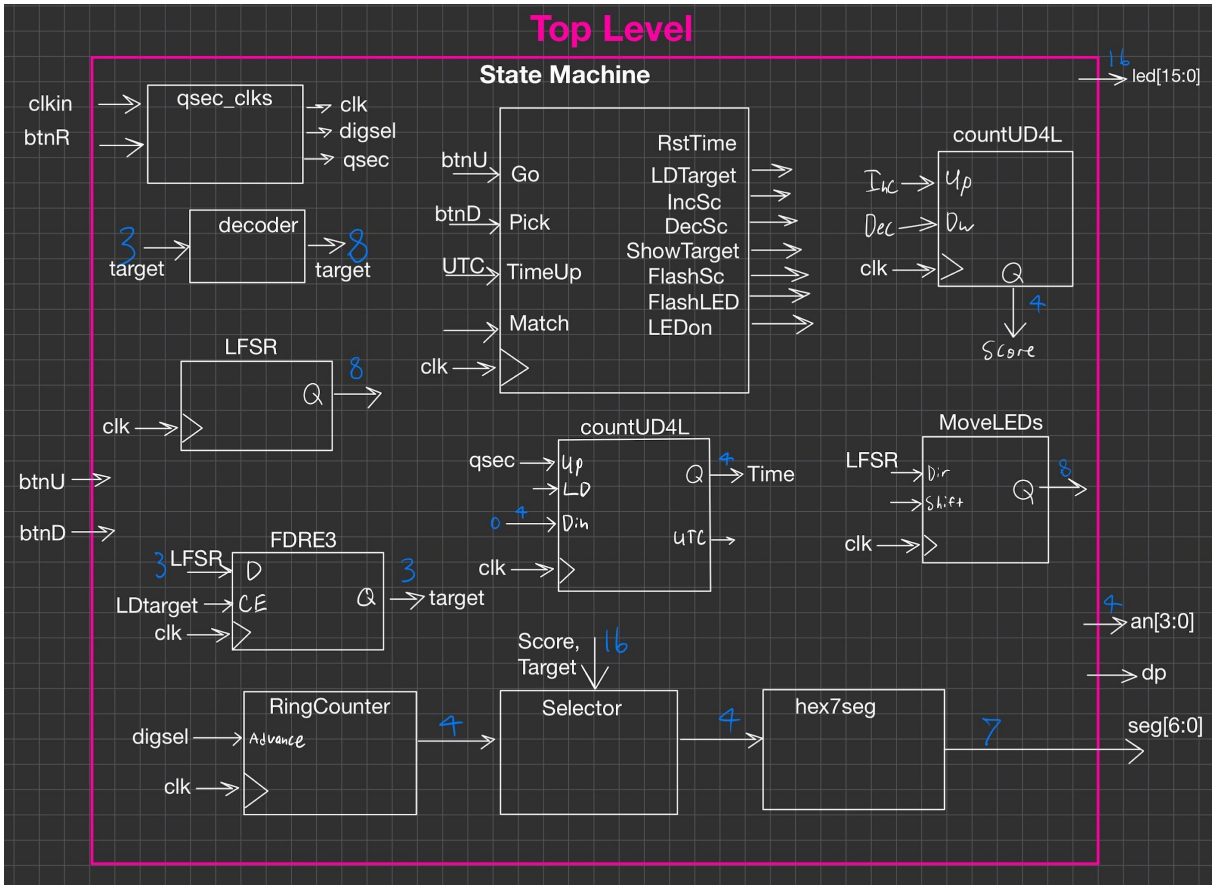
The objective of this lab was to engineer a state machine within a sequential circuit for the implementation of a game called “LED Match”. This was accomplished by implementing a series of circuits in Vivado through Verilog code. This involved creating a state machine that governs the sequential progression of the game, dictating state transitions and output logic, with the ultimate goal of implementing and testing the circuit on the BASYS3 board.

The game revolves around 3 buttons and 8 LEDs on the board. The game begins when the player presses the up button (btnU) to trigger the “Go” state of the state machine. From here, a random number is generated and displayed on a 7-segment display. Seconds later the LEDs on the board will light up 1 by 1, shifting left and right in a random fashion. The player is then tasked with trying to “Pick” the correct LED corresponding to the random number generated on the 7-segment display by pressing the down button (btnD). If the player is correct, they will see their score flash and increase by 1 on the leftmost 7-segment display and a new round will start by generating a new random number. If the player picks an LED at the wrong moment, the correct LED will flash, their score will decrease, and the next round begins. If the player becomes frustrated, they can press the button on the right (btnR) and reset the game entirely.

Design

Top_Level

The “Top_Level” module serves as the central hub for interconnecting all of the components for the LED Match game. It encompasses the integration of all modules: StateMachine, MoveLEDs, hex7seg, Selector, RingCounter, countUD4L, Decoder, LFSR, edgeDetector, and qsec_clks. As you can see from the diagram below, the entirety of the LED game is controlled by 4 input signals: the clock (clkin) and 3 buttons (btnR, btnU, btnD). The outputs of the Top_Level module are all indicators to the player about what is happening in the game. That is, 15 LEDs of which only the lower 8 are used (led [15:0]), the 4 anodes of the 7-segment displays (an[3:0]), the decimal point of the 7-segment displays which are unused (dp), and the 7 segments of the display (seg [6:0]).



The majority of the Top_Level module is wiring together all of the various components inside of it through the use of wires as well as plugging in each component’s inputs and outputs into each other. In the next section, we will see the

logic behind the State Machine so I will use that as an example for what the Top_Level does as it is a great way to express how I used wires and flip-flops in conjunction to call the instances of all modules.

```
//    State Machine
wire Go, Pick, TimeUp, Match;   // Inputs
wire RstTime, LDTarget, IncSc, DecSc, ShowTarget, FlashSc, FlashLED, LEDon;  // Outputs

wire [4:0] next, prev;
State_Machine THE_BRAIN (.clk(clk), .Go(Go), .Pick(Pick), .TimeUp(TimeUp), .Match(Match),
                .RstTime(RstTime), .LDTarget(LDTarget), .IncSc(Inc), .DecSc(Dec),
                .ShowTarget(ShowTarget), .FlashSc(FlashSc), .FlashLED(FlashLED), .LEDon(LEDon)
                ,. nextState(next), .prevState(prev)
                );

FDRE #(.INIT(1'b1)) FF_SM0 (.C(clk), .R(1'b0), .CE(1'b1), .D(next[0]), .Q(prev[0]));
FDRE #(.INIT(1'b0)) FF_SM1 (.C(clk), .R(1'b0), .CE(1'b1), .D(next[1]), .Q(prev[1]));
FDRE #(.INIT(1'b0)) FF_SM2 (.C(clk), .R(1'b0), .CE(1'b1), .D(next[2]), .Q(prev[2]));
FDRE #(.INIT(1'b0)) FF_SM3 (.C(clk), .R(1'b0), .CE(1'b1), .D(next[3]), .Q(prev[3]));
FDRE #(.INIT(1'b0)) FF_SM4 (.C(clk), .R(1'b0), .CE(1'b1), .D(next[4]), .Q(prev[4]));
```

In Top_Level, I created wires using the same names as all inputs and outputs of the state machine for better readability. When I call State_Machine, I am creating an instance of it which I called “THE BRAIN”. I then assign one of its inputs such as “.Go” to a Top_Level wire “(Go)” that Top_Level will use to feed other components into it and this is done for all components and each of their inputs/outputs. Below “THE_BRAIN” is a series of flip-flops used to logically facilitate the state switching of the state machine. I will dive further into the logic behind State_Machine below.

State_Machine

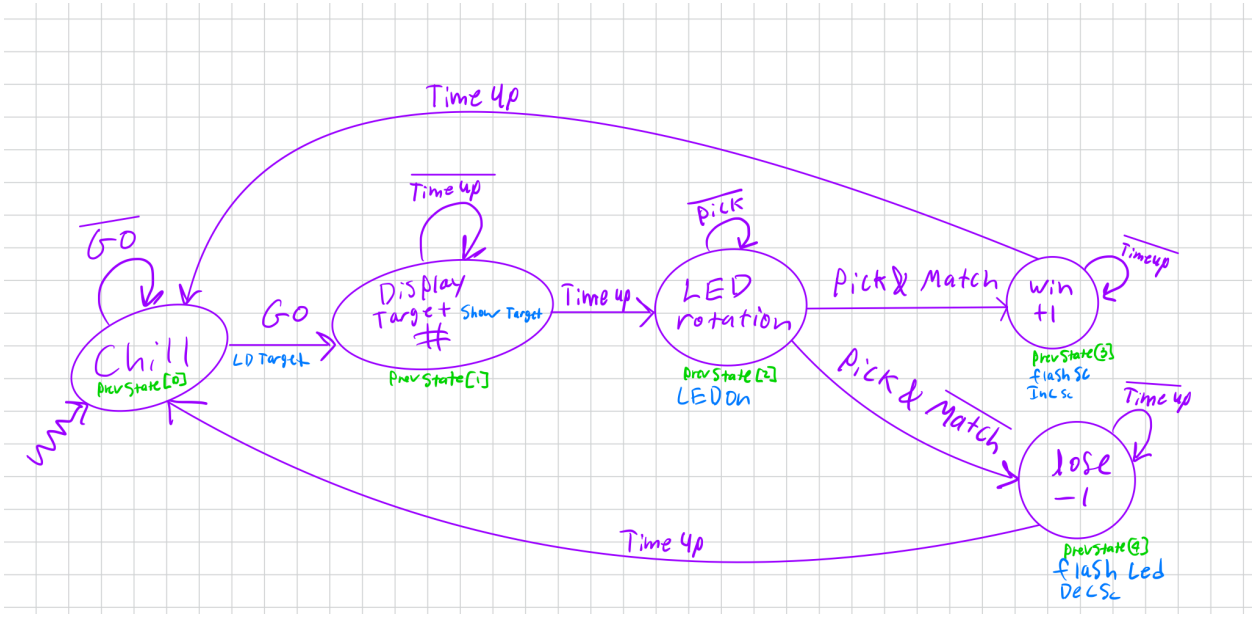
The State_Machine module is the foundation of this project. It is the core logic behind what has happened, what is happening, and what will be happening in the game. It orchestrates the transitions between the game's various states and functions. The system exists in a specific state at any given time, transitioning to other states in response to external inputs while generating outputs based on its current state. When crafting my state machine I thought it would be best to use “one-hot” since we recently learned about this method in class. One-hot essentially allows only 1 bit to be true or “hot” at a time, and this is exactly the logic required to run a game such as “LED Match”.

This module operates based on five key input signals: "Go", "Pick", "TimeUp", "Match", and "clk". "Go" serves as the initiator of the game, "Pick" captures the player"s choices, "TimeUp" denotes the end of specific timed events, "Match" confirms a correct match with the target, and "clk" ensures synchronized state transitions by functioning as

the clock signal. The inputs and outputs of State_Machine are as follows:

```
input Go, Pick, TimeUp, Match,
input clk,
input [4:0] prevState,
output [4:0] nextState,
output LDTarget, IncSc, DecSc, ShowTarget, FlashSc, FlashLED, LEDon, RstTime
```

The output signals govern various critical aspects of the game, including timer resets, loading the target number, score incrementation and decrementation, display control, and LED manipulation. Utilizing the one-hot state encoding approach, this module designates each state using a single bit in the state register. This ensures that only one bit is active at any given time, representing the current state of the system. A visual diagram of this can be seen below.



In my code, “nextState” is determined by the current state (aka prevState) and the input signals controlling the state machine such as the buttons and the clock. When the board is first turned on and loaded, it is in the state “Chill”, also called “nextState[0] in my code, and is awaiting an input signal. The input signal “Go” comes from btnU which transitions to the state “Display Target #” and begins generating a random number based on the current clk. This is nextState[1]. Then, the board displays that number on the rightmost 7-segment display and waits for another input signal. The next input signal it will receive is when “TimeUp” goes high. If “TimeUp” has not happened yet, it will remain in that state. When “TimeUp” does inevitably go high, it transitions to “nextState[2]” or the “LED rotation” state. From here, the game is running and is only waiting on input from the player. If the player doesn’t press btnD then the input signal “Pick” is never triggered and it will remain in that state. When the player does press btnU, they will either “Match” the number displayed, transitioning the state machine to

the “Win” state, or they will “Pick” and not “Match” which transitions to the “Lose” state. Both of these final states, nextState[3] and nextState[4], will restart the game. The logic lining up with the above diagram can be seen in my code below.

```
    // Assign states with one hot
//    Chill
    assign nextState[0] = (prevState[0] & ~Go) | (prevState[3] & TimeUp) | (prevState[4] & TimeUp);
//    Generate #
    assign nextState[1] = (prevState[0] & Go) | (prevState[1] & ~TimeUp);
//    LED rotation
    assign nextState[2] = (prevState[1] & TimeUp) | (prevState[2] & ~Pick);
//    Win
    assign nextState[3] = (prevState[2] & Pick & Match) | (prevState[3] & ~TimeUp);
//    Lose
    assign nextState[4] = (prevState[2] & Pick & ~Match) | (prevState[4] & ~TimeUp);

    // Assign outputs
    assign RstTime = (prevState[0]&Go) | (prevState[2]&Pick&Match) | (prevState[2]&Pick&~Match);
    assign LDTarget = prevState[0] & Go;
    assign IncSc = prevState[2]&Pick&Match;
    assign DecSc = prevState[2]&Pick&~Match;
    assign ShowTarget = (prevState[4]) | (prevState[3]) | (prevState[2]) | (prevState[1]);
    assign FlashSc = prevState[3];                  // Flash Score when you win
    assign FlashLED = prevState[4];         // When you pick
    assign LEDon = prevState[2];
```

Additionally, the “Assign outputs” section of that code is used for controlling the various reasons for all of these states. You need to “LDTarget” only after “prevState[0]”. We want to decrement the score “DecSc” if we are in prevState[2] and the player has picked “Pick” by pressing btnD, but not if they got the LED correct “~Match”. From this code we can see why I had to use the flip-flop logic in “Top_Level”. “prevState” was required as input to the State Machine in order to distinguish between past, present, and future states.

LFSR

The "LFSR" module is an 8-bit Linear Feedback Shift Register (L.F.S.R) which is a mechanism to generate pseudo-random sequences of bits. This design uses a shift register configuration with feedback, creating a sequence of bits that appears random.

The foundation of the LFSR is the XOR operation within the shift register, generating feedback by XORing specific bits within the register and circulating the result back as an input. Reading the LFSR at various points within its cycle can produce a pseudo-random 8-bit number.

MoveLEDs

The "MoveLEDs" module orchestrates the movement of an 8-bit LED sequence to the left or right based on the "Dir" and "Shift" input signals. This module operates as a circular shift register for the movement of the LED pattern. The "Dir" input signifies the direction of the shift, where a "0" indicates a leftward shift (aka “~Dir”) and a "1" denotes a rightward shift. This input works in conjunction with the "Shift" input, which serves as a trigger to initiate the shift operation. When "Shift" is "1", the LED sequence undergoes the shift operation, and when it"s "0", the LED states remain unchanged.

Decoder

The "Decoder" module functions as a 3-to-8 bit binary decoder, converting a 3-bit binary input denoted as "x" into an 8-bit one-hot-encoded output labeled as "y". This module uses combinational logic to generate specific output patterns based on the binary input. Each output bit represents a particular minterm of the input bits. Utilizing the provided truth table representation, each output bit ('y') is derived through a logical expression that defines the one-hot encoding.

countUD4L

The "countUD4L" module was implemented from our previous labs. It is a 4-bit up-down counter with load functionality. It operates in response to various inputs, including the system clock ('clk'), signals for increment ('btnU') and decrement ('btnD'), and a control for loading a 4-bit value ('LD'). This counter utilizes a 4-bit bus, 'Din,' to store the value that will be loaded when the 'LD' signal is asserted on the clock edge. The current value held by the counter is represented by the 4-bit bus 'Q'. The countUD4L module generates two output signals: 'UTC' (Up Terminal Count) and 'DTC' (Down Terminal Count). 'UTC' is active when the counter reaches its maximum value, specifically 15 in decimal ('4'b1111'). Conversely, 'DTC' is active when the counter reaches its minimum value of 0 ('4'b0000'). The logic of the countUD4L module is designed to handle various conditions involving the clock, increment, decrement, and load signals, allowing the counter to count up or down based on these inputs. The module operates using conditional logic that manages the state of the counter, either by incrementing, decrementing, or loading specific values as directed by the control inputs.

hex7seg

The "hex7seg" module takes a 4-bit binary input and outputs to a 7-segment display through combinational logic. This was acquired from previous labs by creating a truth table for the inputs of 0-6 and assigning logical statements for how to display those

numbers on the 7 display segments a-g. From here we created boolean expressions which we used to assign the segments in verilog.

Ring_Counter

The "RingCounter" module is a 4-bit ring counter, designed to cycle through a sequence of states triggered by a rising edge of the clock signal when the "Advance" input is active. For this lab, the ring counter's role was in managing the display multiplexing of the 7-segment display. The module incorporates a sequence of four flip-flops, each handling a specific bit in the 4-bit output "Ring_out." The connections between the flip-flops and the clock signal, as well as the "Advance" input, ensure that the counter progresses with each clock cycle and advances to the next state upon the "Advance" signal activation. The 4 flip-flops create a looping sequence that cycles through all possible states of the 4-bit output. This module allows anode activations on the 7-segment display, displaying vital information about the game to the player.

Selector

The "Selector" module extracts a group of 4 bits from a 16-bit input, determined by a 4-bit selector input. This module takes a 16-bit input and a 4-bit selector input to produce a 4-bit output. The purpose of the "Selector" module in the LED Match game is to facilitate the assignment of the leftmost and rightmost 7-segment displays for showing both the "Score" and the "Target." The "sel" input determines the specific bits from the 16-bit input "N" to be directed to the 4-bit output. Through a series of logical operations, the "Selector" module manipulates the bits of the 16-bit input based on the "sel" input.

Conclusion

This lab, like many of the past labs, was extremely difficult. Specifically, the logic behind the state machine had given me the most trouble. State machines were a fairly new concept for us going into this lab. Learning about this new concept in conjunction with applying it to our lab was an extremely difficult task. But by creating simulations and testing my code extensively, I was able to better understand the importance of creating a state machine. If I did this lab again, I would create simulation files and test the new modules before implementing them to my Top_Level module. This would've saved me a lot of time when debugging and trying to figure out where my logic was wrong.

Appendix

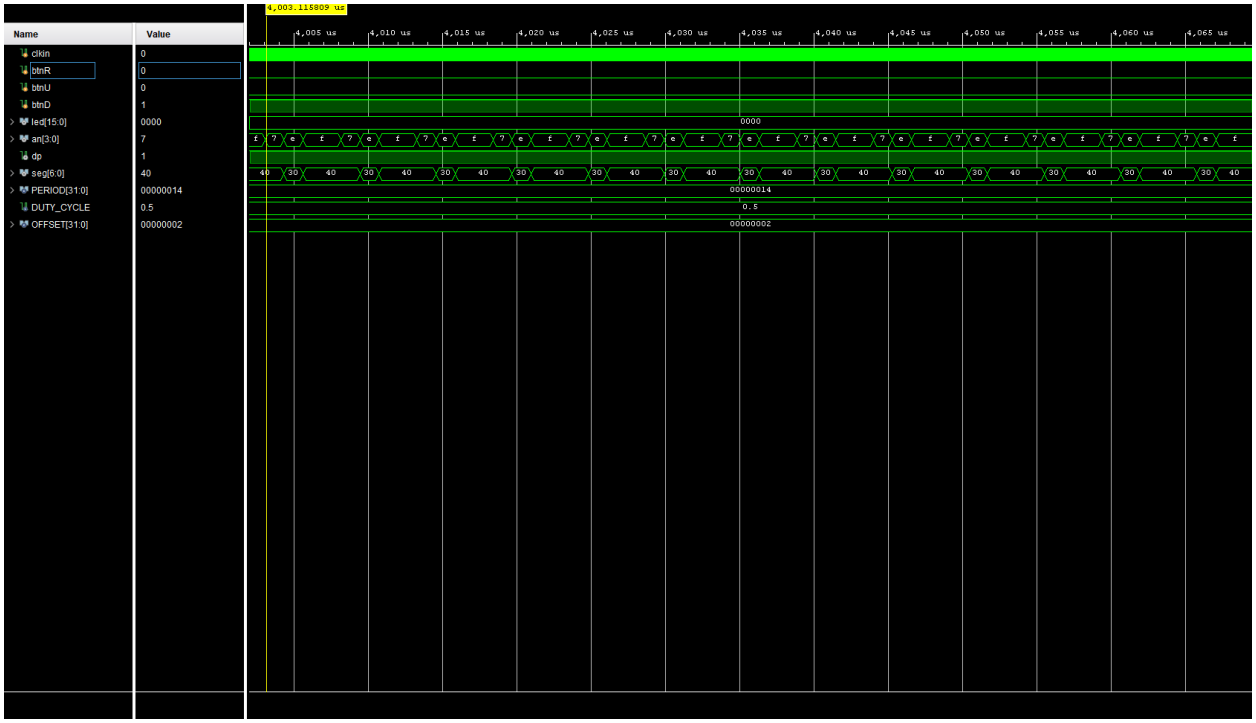


Fig. 1 Top_Level Simulation

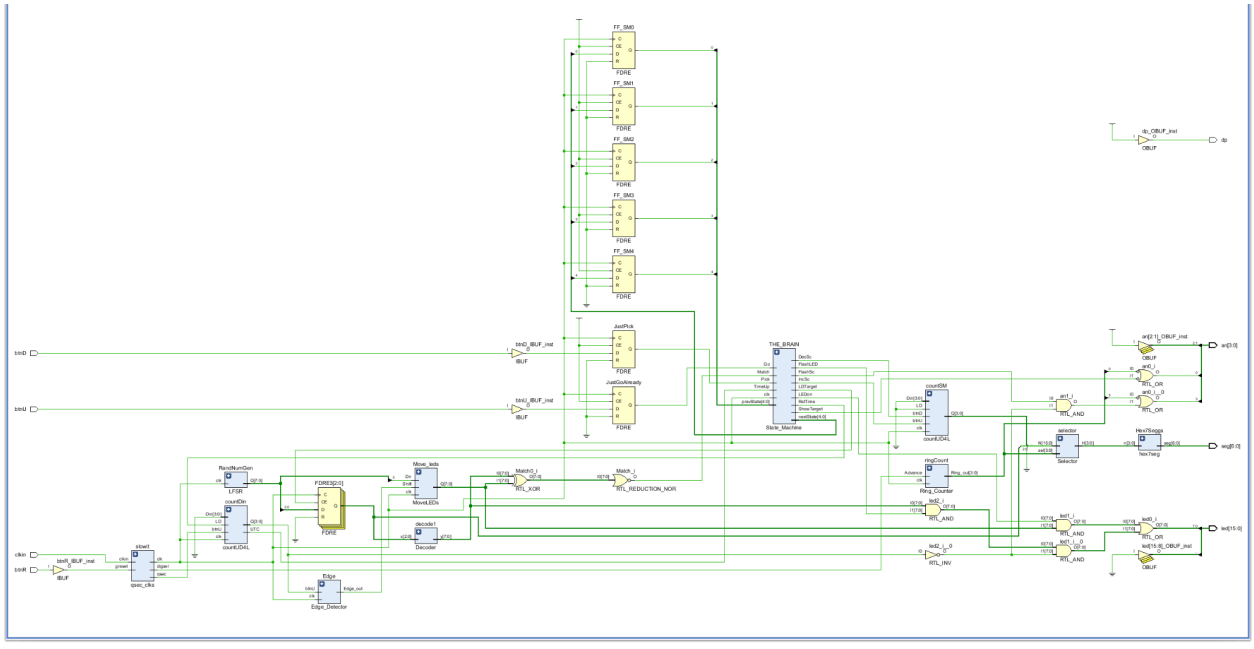


Fig. 2 Schematic

```
`timescale 1ns / 1ps

module Top_Level(
    input clkin, btnR, btnU, btnD,
    output [15:0] led,
    output [3:0] an,
    output dp,
    output [6:0] seg
    );

    assign led [15:8] = 1'b0;
    assign dp = 1'b1;
    assign an[1] = 1'b1;
    assign an[2] = 1'b1;
    // State Machine Stuff
    wire Go, Pick, TimeUp, Match;   // Inputs
    wire RstTime, LDTarget, IncSc, DecSc, ShowTarget, FlashSc, FlashLED, LEDon;  //
Outputs

//    qsec_clk
    wire clk, digsel, qsec;
    qsec_clks slowit (.clkin(clkin), .greset(btnR), .clk(clk), .digsel(digsel),
.qsec(qsec));

    //flipflop for btnU and Go
    FDRE #(.INIT(1'b0)) JustGoAlready (.C(clk), .R(1'b0), .CE(1'b1), .D(btnU), .Q(Go)

    //Pick
    FDRE #(.INIT(1'b0)) JustPick (.C(clk), .R(1'b0), .CE(1'b1), .D(btnD), .Q(Pick));

//    LFSR
    wire [7:0] LFSRout;
    LFSR RandNumGen (.clk(clk), .Q(LFSRout));

//    FDRE3
    wire [2:0] target;
    FDRE #(.INIT(1'b0)) FDRE3[2:0] (.C({3{clk}}), .R(1'b0), .CE(LDTarget),
.D(LFSRout[2:0]), .Q(target));


//    Decoder
    wire [7:0] decOut;
    Decoder decode1 (.x(target), .y(decOut));

//    countUD4L with Din
    wire [3:0] Time;
```

```
    wire UTC_out;
    countUD4L countDin (.clk(clk), .btnU(qsec), .LD(RstTime), .Din(4'b0), .Q(Time),
.UTC(UTC_out));
    assign TimeUp = UTC_out;

//    RingCounter
    wire [3:0] Ring_out;
    Ring_Counter ringCount(.Advance(digsel), .clk(clk), .Ring_out(Ring_out));

//    Selector
    wire [3:0] Sel_out;
    wire [3:0] Score;
    Selector selector (.N({Score, 8'h0,1'b0,target}), .sel(Ring_out), .H(Sel_out));

//    hex7seg
    hex7seg Hex7Seggs (.n(Sel_out), .seg(seg));

//    countUD4L for State Machine
    wire Inc, Dec;
    countUD4L countSM (.clk(clk), .LD(1'b0), .btnU(Inc), .btnD(Dec), .Din(4'b0),
.Q(Score));

    // Edge detect
    wire SlowTime;
   Edge_Detector Edge (.btnU(Time[0]), .clk(clk), .Edge_out(SlowTime));

//    MoveLEDs
    wire [7:0] Moveleds_out;
    MoveLEDs Move_leds (.Dir(LFSRout[5]), .Shift(SlowTime), .clk(clk),
.Q(Moveleds_out));

    // Match
    assign Match = ~|(decOut ^ Moveleds_out);


    //    State Machine
    wire [4:0] next, prev;
    State_Machine THE_BRAIN (.clk(clk), .Go(Go), .Pick(Pick), .TimeUp(TimeUp),
.Match(Match),
                             .RstTime(RstTime), .LDTarget(LDTarget), .IncSc(Inc),
.DecSc(Dec),
                             .ShowTarget(ShowTarget), .FlashSc(FlashSc),
.FlashLED(FlashLED), .LEDon(LEDon)
                             ,. nextState(next), .prevState(prev)
                             );
    FDRE #(.INIT(1'b1)) FF_SM0 (.C(clk), .R(1'b0), .CE(1'b1), .D(next[0]),
```

```
.Q(prev[0]));
    FDRE #(.INIT(1'b0)) FF_SM1 (.C(clk), .R(1'b0), .CE(1'b1), .D(next[1]),
.Q(prev[1]));
    FDRE #(.INIT(1'b0)) FF_SM2 (.C(clk), .R(1'b0), .CE(1'b1), .D(next[2]),
.Q(prev[2]));
    FDRE #(.INIT(1'b0)) FF_SM3 (.C(clk), .R(1'b0), .CE(1'b1), .D(next[3]),
.Q(prev[3]));
    FDRE #(.INIT(1'b0)) FF_SM4 (.C(clk), .R(1'b0), .CE(1'b1), .D(next[4]),
.Q(prev[4]));


//    Temp for testing
    assign an[3] = ~Ring_out[3] | (FlashSc & ~Time[0]);
    assign an[0] = ~Ring_out[0] | ~ShowTarget;
    assign led[7:0] = ({8{LEDon}} & Moveleds_out)|(decOut  & {8{FlashLED}} &
{8{~Time[0]}});

endmodule
```

```
module State_Machine(
    input Go, Pick, TimeUp, Match,
    input clk,
    input [4:0] prevState,
    output [4:0] nextState,
    output LDTarget, IncSc, DecSc, ShowTarget, FlashSc, FlashLED, LEDon, RstTime
    );

    // Assign states with one hot
//    Chill
    assign nextState[0] = (prevState[0] & ~Go) | (prevState[3] & TimeUp) |
(prevState[4] & TimeUp);
//    Generate #
    assign nextState[1] = (prevState[0] & Go) | (prevState[1] & ~TimeUp);
//    LED rotation
    assign nextState[2] = (prevState[1] & TimeUp) | (prevState[2] & ~Pick);
//    Win
    assign nextState[3] = (prevState[2] & Pick & Match) | (prevState[3] & ~TimeUp);
//    Lose
    assign nextState[4] = (prevState[2] & Pick & ~Match) | (prevState[4] & ~TimeUp);

    // Assign outputs
    assign RstTime = (prevState[0]&Go) | (prevState[2]&Pick&Match) |
(prevState[2]&Pick&~Match);
    assign LDTarget = prevState[0] & Go;
    assign IncSc = prevState[2]&Pick&Match;
    assign DecSc = prevState[2]&Pick&~Match;
    assign ShowTarget = (prevState[4]) | (prevState[3]) | (prevState[2]) |
(prevState[1]);
    assign FlashSc = prevState[3];                  // Flash Score when you win
    assign FlashLED = prevState[4];         // When you pick
    assign LEDon = prevState[2];


endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/30/2023 05:38:52 PM
// Design Name:
// Module Name: LFSR
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module LFSR(
    input clk,
    output [7:0] Q
    );
    wire [7:0] rnd;
    wire D_0;

    assign D_0 = rnd[7] ^ rnd[6] ^ rnd[5] ^ rnd[0];

    FDRE #(.INIT(1'b1)) RND0 (.C(clk), .R(1'b0), .CE(1'b1), .D(D_0), .Q(rnd[0]));
    FDRE #(.INIT(1'b0)) RND1 (.C(clk), .R(1'b0), .CE(1'b1), .D(rnd[0]), .Q(rnd[1]));
    FDRE #(.INIT(1'b0)) RND2 (.C(clk), .R(1'b0), .CE(1'b1), .D(rnd[1]), .Q(rnd[2]));
    FDRE #(.INIT(1'b0)) RND3 (.C(clk), .R(1'b0), .CE(1'b1), .D(rnd[2]), .Q(rnd[3]));
    FDRE #(.INIT(1'b0)) RND4 (.C(clk), .R(1'b0), .CE(1'b1), .D(rnd[3]), .Q(rnd[4]));
    FDRE #(.INIT(1'b0)) RND5 (.C(clk), .R(1'b0), .CE(1'b1), .D(rnd[4]), .Q(rnd[5]));
    FDRE #(.INIT(1'b0)) RND6 (.C(clk), .R(1'b0), .CE(1'b1), .D(rnd[5]), .Q(rnd[6]));
    FDRE #(.INIT(1'b0)) RND7 (.C(clk), .R(1'b0), .CE(1'b1), .D(rnd[6]), .Q(rnd[7]));


    assign Q = rnd;

endmodule
```

```
module MoveLEDs(
    input Dir, Shift, clk,
    output [7:0] Q
    );
    wire [7:0] a, b;
    //Dir = Shift left
    // ~Dir = shift right
    assign a[0] = (b[7] & Dir) | (b[1] & ~Dir);
    assign a[1] = (b[0] & Dir) | (b[2] & ~Dir);
    assign a[2] = (b[1] & Dir) | (b[3] & ~Dir);
    assign a[3] = (b[2] & Dir) | (b[4] & ~Dir);
    assign a[4] = (b[3] & Dir) | (b[5] & ~Dir);
    assign a[5] = (b[4] & Dir) | (b[6] & ~Dir);
    assign a[6] = (b[5] & Dir) | (b[7] & ~Dir);
    assign a[7] = (b[6] & Dir) | (b[0] & ~Dir);

    FDRE #(.INIT(1'b1)) FF_0 (.C(clk), .CE(Shift), .D(a[0]), .Q(b[0]));

    FDRE #(.INIT(1'b0)) FF_7_0 [7:1] (
    .C({7{clk}}),
    .CE({7{Shift}}),
    .D({a[7:1]}),
    .Q({b[7:1]})
    );
    assign Q = b;
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/30/2023 05:15:40 PM
// Design Name:
// Module Name: Decoder
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Decoder(
    input [2:0] x,
    output [7:0] y
    );

    assign y[0] = ~x[2] & ~x[1] & ~x[0];
    assign y[1] = ~x[2] & ~x[1] & x[0];
    assign y[2] = ~x[2] & x[1] & ~x[0];
    assign y[3] = ~x[2] & x[1] & x[0];
    assign y[4] = x[2] & ~x[1] & ~x[0];
    assign y[5] = x[2] & ~x[1] & x[0];
    assign y[6] = x[2] & x[1] & ~x[0];
    assign y[7] = x[2] & x[1] & x[0];

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/17/2023 02:42:13 PM
// Design Name:
// Module Name: countUD4L
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module countUD4L(

    input clk,         //  the system clock,
    input btnU,            //  (increment)
    input btnD,            //  (decrement)
    input LD,           //  (load control)
    input [3:0] Din,          //  the 4-bit bus Din that will be loaded on the clock
edge if LD is high.

    output [3:0] Q,           //  a 4-bit bus Q which is the current value held by
the counter,
    output UTC,          //  the signal UTC (btnU Terminal Count) which is 1 only
when the counter is at 4'b1111 (15 in decimal), and
    output DTC           //  the signal DTC (Down Terminal Count) which is 1 only
when the counter is at 4'b0000.
    );
    wire Dwire [3:0];
    wire Inc;
    assign Inc = (btnU ^ btnD | LD);

    assign Dwire[0] = ((Q[0]^Inc) & btnU & ~btnD) | (((Q[0]^Inc) & ~btnU & btnD) &
~LD) | (LD & Din[0]);
    assign Dwire[1] = ((Q[1]^ (Inc & Q[0])) & btnU & ~btnD) | (((Q[1]^ (Inc &
~Q[0])) & ~btnU & btnD) & ~LD) | (LD & Din[1]);
```

```
    assign Dwire[2] = ((Q[2]^ (Inc & Q[0] & Q[1])) & btnU & ~btnD) | (((Q[2]^ (Inc &
~Q[0] & ~Q[1])) & ~btnU & btnD) & ~LD) | (LD & Din[2]);
    assign Dwire[3] = ((Q[3]^ (Inc & Q[0] & Q[1] & Q[2])) & btnU & ~btnD) | (((Q[3]^
(Inc & ~Q[0] & ~Q[1] & ~Q[2])) & ~btnU & btnD) & ~LD) | (LD & Din[3]);

    assign UTC = &(Q[3:0]);
    assign DTC = &(~Q[3:0]);

    FDRE #(.INIT(1'b0)) b0_ff (.C(clk), .R(1'b0), .CE(Inc), .D(Dwire[0]), .Q(Q[0]));
    //  From "Using Flip-Flops in Vivado"
https://classes.soe.ucsc.edu/cse100/Fall23/lab/FDRE/FDRE.html
    FDRE #(.INIT(1'b0)) b1_ff (.C(clk), .R(1'b0), .CE(Inc), .D(Dwire[1]), .Q(Q[1]));
    FDRE #(.INIT(1'b0)) b2_ff (.C(clk), .R(1'b0), .CE(Inc), .D(Dwire[2]), .Q(Q[2]));
    FDRE #(.INIT(1'b0)) b3_ff (.C(clk), .R(1'b0), .CE(Inc), .D(Dwire[3]), .Q(Q[3]));

endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/10/2023 01:16:42 PM
// Design Name:
// Module Name: hex7seg
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module hex7seg(
    input [3:0] n,
    output [6:0] seg
    );
//    a
    assign seg[0] = (~n[3] & ~n[2] & ~n[1] & n[0]) | (~n[3] & n[2] & ~n[1] & ~n[0])
| (n[3] & ~n[2] & n[1] & n[0]) | (n[3] & n[2] & ~n[1] & n[0]);
//    b
    assign seg[1] = (~n[3] & n[2] & ~n[1] & n[0]) | (~n[3] & n[2] & n[1] & ~n[0]) |
(n[3] & ~n[2] & n[1] & n[0]) | (n[3] & n[2] & ~n[1] & ~n[0]) | (n[3] & n[2] & n[1] &
~n[0]) | (n[3] & n[2] & n[1] & n[0]);
//    c
    assign seg[2] = (~n[3] & ~n[2] & n[1] & ~n[0]) | (n[3] & n[2] & ~n[1] & ~n[0]) |
(n[3] & n[2] & n[1] & ~n[0]) | (n[3] & n[2] & n[1] & n[0]);
//    d
    assign seg[3] = (~n[3] & ~n[2] & ~n[1] & n[0]) | (~n[3] & n[2] & ~n[1] & ~n[0])
| (~n[3] & n[2] & n[1] & n[0]) | (n[3] & ~n[2] & ~n[1] & n[0]) | (n[3] & ~n[2] &
n[1] & ~n[0]) | (n[3] & n[2] & n[1] & n[0]);
//    e
    assign seg[4] = (~n[3] & ~n[2] & ~n[1] & n[0]) | (~n[3] & ~n[2] & n[1] & n[0]) |
(~n[3] & n[2] & ~n[1] & ~n[0]) | (~n[3] & n[2] & ~n[1] & n[0]) | (~n[3] & n[2] &
n[1] & n[0]) | (n[3] & ~n[2] & ~n[1] & n[0]);
//    f
    assign seg[5] = (~n[3] & ~n[2] & ~n[1] & n[0]) | (~n[3] & ~n[2] & n[1] & ~n[0])
```

```
| (~n[3] & ~n[2] & n[1] & n[0]) | (~n[3] & n[2] & n[1] & n[0]) | (n[3] & n[2] &
~n[1] & n[0]);
//    g
    assign seg[6] = (~n[3] & ~n[2] & ~n[1] & ~n[0]) | (~n[3] & ~n[2] & ~n[1] & n[0])
| (~n[3] & n[2] & n[1] & n[0]) | (n[3] & n[2] & ~n[1] & ~n[0]);
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/17/2023 01:58:07 PM
// Design Name:
// Module Name: Ring_Counter
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Ring_Counter(
    input Advance,
    input clk,

    output [3:0] Ring_out
    );

    FDRE #(.INIT(1'b1)) b0_ff (.C(clk), .R(1'b0), .CE(Advance), .D(Ring_out[3]),
.Q(Ring_out[0]));
    FDRE #(.INIT(1'b0)) b1_ff (.C(clk), .R(1'b0), .CE(Advance), .D(Ring_out[0]),
.Q(Ring_out[1]));
    FDRE #(.INIT(1'b0)) b2_ff (.C(clk), .R(1'b0), .CE(Advance), .D(Ring_out[1]),
.Q(Ring_out[2]));
    FDRE #(.INIT(1'b0)) b3_ff (.C(clk), .R(1'b0), .CE(Advance), .D(Ring_out[2]),
.Q(Ring_out[3]));
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/17/2023 02:04:49 PM
// Design Name:
// Module Name: Selector
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Selector(
    input [15:0] N,
    input [3:0] sel,

    output [3:0] H
    );
    assign H[3] = (sel[3]&N[15]) | (sel[2]&N[11]) | (sel[1]&N[7]) | (sel[0]&N[3]);
    assign H[2] = (sel[3]&N[14]) | (sel[2]&N[10]) | (sel[1]&N[6]) | (sel[0]&N[2]);
    assign H[1] = (sel[3]&N[13]) | (sel[2]&N[9]) | (sel[1]&N[5]) | (sel[0]&N[1]);
    assign H[0] = (sel[3]&N[12]) | (sel[2]&N[8]) | (sel[1]&N[4]) | (sel[0]&N[0]);

endmodule
```

```
`timescale 1ns / 1ps

module TopLevelSim();
    reg clkin, btnR, btnU, btnD;
    wire [15:0] led;
    wire [3:0] an;
    wire dp;
    wire [6:0] seg;

    Top_Level UUT(.clkin(clkin), .btnR(btnR), .btnU(btnU), .btnD(btnD),
                  .led(led), .an(an), .dp(dp), .seg(seg)
    );

    parameter PERIOD = 20;
    parameter real DUTY_CYCLE = 0.5;
    parameter OFFSET = 2;



    initial    // Clock process for clkin
    begin
        #OFFSET
          clkin = 1'b1;
        forever
        begin
            #(PERIOD-(PERIOD*DUTY_CYCLE)) clkin = ~clkin;
        end
    end

    initial
    begin


//        clkin, btnR, btnU, btnD

        // set all values low
        //assign clkin = 1'b0;
        assign btnR = 1'b0;
        assign btnU = 1'b0;
        assign btnD = 1'b0;

        //START SIMULATION TESTING HERE
        //Display target#
        #4000
        assign btnR = 1'b1;
```

```
        #100
        assign btnR = 1'b0;
        assign btnU = 1'b1;

        #100
        assign btnR = 1'b0;
        #100
        assign btnU = 1'b0;

        #100
        assign btnD = 1'b0;

        #100
        assign btnD = 1'b1;

end
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/17/2023 02:05:39 PM
// Design Name:
// Module Name: Edge_Detector
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module Edge_Detector(
    input btnU,
    input clk,

    output Edge_out
    );
    wire x;
    assign Edge_out = btnU & ~x;
    FDRE #(.INIT(1'b0)) Edge_D1 (.C(clk), .R(1'b0), .CE(1'b1), .D(btnU), .Q(x));
endmodule
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 10/27/2022 11:19:41 AM
// Design Name:
// Module Name: lab5_clks
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision: 10/27/22
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////



module qsec_clks(
    input clkin,
    input greset,  //btnR
    output clk,
    output digsel,
    output qsec//,
    //output fastclk
);

       wire clk_int;
       //assign fastclk = clk_int;
       clk_wiz_0 my_clk_inst (.clk_out1(clk_int), .reset(greset), .locked(),
.clk_in1(clkin));
       clkcntrl4 slowclk (.clk_int(clk_int), .seldig(digsel), .clk_out(clk),
.qsec(qsec));

       STARTUPE2 #(.PROG_USR("FALSE"),//Activate program event security feature.
Requires encrypted bitstreams.
                   .SIM_CCLK_FREQ(0.0)   // Set the Configuration Clock
Frequency(ns) for simulation.
                                      )
       STARTUPE2_inst (.CFGCLK(dummy),  // 1-bit output: Configuration main clock
output
```

```
                   .CFGMCLK(), // 1-bit output: Configuration internal oscillator
clock output
                   .EOS(),     // 1-bit output: Active high output signal indicating
the End Of Startup.
                   .PREQ(),// 1-bit output: PROGRAM request to fabric output
                   .CLK(),  // 1-bit input: User start-up clock input
                   .GSR(greset),  // 1-bit input: Global Set/Reset input (GSR cannot
be used for the port name)
                   .GTS(),  // 1-bit input: Global 3-state input (GTS cannot be used
for the port name)
                   .KEYCLEARB(), // 1-bit input: Clear AES Decrypter Key input from
Battery-Backed RAM (BBRAM)
                   .PACK(), // 1-bit input: PROGRAM acknowledge input
                   .USRCCLKO(), // 1-bit input: User CCLK input
                   .USRCCLKTS(), // 1-bit input: User CCLK 3-state enable input
                   .USRDONEO(), // 1-bit input: User DONE pin output control
                   .USRDONETS() // 1-bit input: User DONE 3-state enable output
                 );  // End of STARTUPE2_inst instantiation

endmodule

module clk_wiz_0
 (// Clock in ports
  // Clock out ports
  output        clk_out1,
  // Status and control signals
  input         reset,
  output        locked,
  input         clk_in1
 );
  // Input buffering
  //------------------------------------
wire clk_in1_clk_wiz_0;
wire clk_in2_clk_wiz_0;
  IBUF clkin1_ibufg
   (.O (clk_in1_clk_wiz_0),
    .I (clk_in1));



  // Clocking PRIMITIVE
  //------------------------------------

  // Instantiation of the MMCM PRIMITIVE
  //    * Unused inputs are tied off
  //    * Unused outputs are labeled unused
```

```
wire        clk_out1_clk_wiz_0;   //this is the output before BUFG
wire        clk_out2_clk_wiz_0;
wire        clk_out3_clk_wiz_0;
wire        clk_out4_clk_wiz_0;
wire        clk_out5_clk_wiz_0;
wire        clk_out6_clk_wiz_0;
wire        clk_out7_clk_wiz_0;

wire [15:0] do_unused;
wire        drdy_unused;
wire        psdone_unused;
wire        locked_int;
wire        clkfbout_clk_wiz_0;
wire        clkfbout_buf_clk_wiz_0;
wire        clkfboutb_unused;
wire clkout0b_unused;
wire clkout1_unused;
wire clkout1b_unused;
wire clkout2_unused;
wire clkout2b_unused;
wire clkout3_unused;
wire clkout3b_unused;
wire clkout4_unused;
wire        clkout5_unused;
wire        clkout6_unused;
wire        clkfbstopped_unused;
wire        clkinstopped_unused;
wire        reset_high;

MMCME2_ADV
#(.BANDWIDTH            ("OPTIMIZED"),
  .CLKOUT4_CASCADE      ("FALSE"),
  .COMPENSATION         ("ZHOLD"),
  .STARTUP_WAIT         ("FALSE"),
  .DIVCLK_DIVIDE        (1),
  .CLKFBOUT_MULT_F      (9.125),
  .CLKFBOUT_PHASE       (0.000),
  .CLKFBOUT_USE_FINE_PS ("FALSE"),
  .CLKOUT0_DIVIDE_F     (36.500),
  .CLKOUT0_PHASE        (0.000),
  .CLKOUT0_DUTY_CYCLE   (0.500),
  .CLKOUT0_USE_FINE_PS  ("FALSE"),
  .CLKIN1_PERIOD        (10.0))
mmcm_adv_inst
  // Output clocks
 (
```

```
    .CLKFBOUT            (clkfbout_clk_wiz_0),
    .CLKFBOUTB           (clkfboutb_unused),
    .CLKOUT0             (clk_out1_clk_wiz_0),  // this is the output clock before BU
    .CLKOUT0B            (clkout0b_unused),
    .CLKOUT1             (clkout1_unused),
    .CLKOUT1B            (clkout1b_unused),
    .CLKOUT2             (clkout2_unused),
    .CLKOUT2B            (clkout2b_unused),
    .CLKOUT3             (clkout3_unused),
    .CLKOUT3B            (clkout3b_unused),
    .CLKOUT4             (clkout4_unused),
    .CLKOUT5             (clkout5_unused),
    .CLKOUT6             (clkout6_unused),
     // Input clock control
    .CLKFBIN             (clkfbout_buf_clk_wiz_0),
    .CLKIN1              (clk_in1_clk_wiz_0),
    .CLKIN2              (1'b0),
     // Tied to always select the primary input clock
    .CLKINSEL            (1'b1),
    // Ports for dynamic reconfiguration
    .DADDR               (7'h0),
    .DCLK                (1'b0),
    .DEN                 (1'b0),
    .DI                  (16'h0),
    .DO                  (do_unused),
    .DRDY                (drdy_unused),
    .DWE                 (1'b0),
    // Ports for dynamic phase shift
    .PSCLK               (1'b0),
    .PSEN                (1'b0),
    .PSINCDEC            (1'b0),
    .PSDONE              (psdone_unused),
    // Other control and status signals
    .LOCKED              (locked_int),
    .CLKINSTOPPED        (clkinstopped_unused),
    .CLKFBSTOPPED        (clkfbstopped_unused),
    .PWRDWN              (1'b0),
    .RST                 (reset_high));
  assign reset_high = reset;

  assign locked = locked_int;

// Clock Monitor clock assigning
//--------------------------------------
// Output buffering
//-----------------------------------
```

```
  BUFG clkf_buf
   (.O (clkfbout_buf_clk_wiz_0),
    .I (clkfbout_clk_wiz_0));

  BUFG clkout1_buf
   (.O   (clk_out1),
    .I   (clk_out1_clk_wiz_0));
endmodule


// clk_int is clkin divided by 2
module clkcntrl4(
     input clk_int,
     output seldig,
     output clk_out,
     output qsec);

   wire XLXN_70;
   wire XLXN_72;
   wire XLXN_75;
   wire XLXN_77;
   wire XLXN_78;
   wire XLXN_79;
   wire qsec3, qsec2, qsec0;


   count4_MSCLK_clkcntrl4  XLXI_37 (.C(clk_int),
                                    .CE(1'b1),
                                    .R(1'b0),
                                    .CEO(XLXN_72),
                                    .Q0(),
                                    .Q1(),
                                    .Q2(),
                                    .Q3(),
                                    .TC());

   count4_MSCLK_clkcntrl4  XLXI_38 (.C(clk_int),
                                    .CE(XLXN_72),
                                    .R(1'b0),
                                    .CEO(XLXN_70),
                                    .Q0(),
                                    .Q1(),
                                    .Q2(),
                                    .Q3(),
                                    .TC());
```

```
//   (* HU_SET = "XLXI_39_75" *)
   count4_MSCLK_clkcntrl4  XLXI_39 (.C(clk_int),
                                    .CE(XLXN_70),
                                    .R(1'b0),
                                    .CEO(),
                                   .Q0(),
                                    .Q1(),
                                    .Q2(),
                                    .Q3(XLXN_77),
                                    .TC());
   //Second counter starts here
//(* HU_SET = "XLXI_40_76" *)
 count4_MSCLK_clkcntrl4  XLXI_40 (.C(clk_out),    // clk_int in simulation,
                                    .CE(1'b1),
                                    .R(1'b0),
                                    .CEO(),
                                    .Q0(),
                                    .Q1(),
                                    .Q2(),
                                    .Q3(),
                                    .TC(XLXN_75));    // digsel either

//Third counter starts here for real qsec

   count4_MSCLK_clkcntrl4  XLXI_49 (.C(clk_out),    // clk_int in simulation,
                                    .CE(1'b1),
                                    .R(qsec3),
                                    .CEO(XLXN_78),
                                    .Q0(),
                                    .Q1(),
                                    .Q2(),
                                    .Q3(),
                                    .TC());

   count4_MSCLK_clkcntrl4  XLXI_45 (.C(clk_out),
                                    .CE(XLXN_78),
                                    .R(qsec3),
                                    .CEO(XLXN_79),
                                    .Q0(),
                                    .Q1(),
                                    .Q2(),
                                    .Q3(),
                                    .TC());

   count4_MSCLK_clkcntrl4  XLXI_44 (.C(clk_out),
                                    .CE(XLXN_79),
```

```
                                    .R(qsec3),
                                    .CEO(),
                                    .Q0(),
                                    .Q1(qsec2),
                                    .Q2(qsec0),
                                    .Q3(),
                                    .TC());

   AND2  I_12222 (.I0(qsec0),
                  .I1(qsec2),
                  //.I2(XLXN_79),
                  //.I3(),
                  .O(qsec3));


`ifdef XILINX_SIMULATOR
   BUF  XLXI_336 (.I(XLXN_75),.O(seldig));
   BUF  XLXI_398 (.I(XLXN_75),.O(qsec));
   BUFG  XLXI_399 (.I(clk_int),.O(clk_out));
`else
   BUF  XLXI_336 (.I(XLXN_75),.O(seldig));
   BUF  XLXI_398 (.I(qsec3),.O(qsec));
   BUFG  XLXI_401 (.I(XLXN_77),.O(clk_out));
`endif

endmodule


//module FTCE_MXILINX_clkcntrl4(C,
//                              CE,
//                              CLR,
//                              T,
//                              Q);

//   parameter INIT = 1'b0;

//    input C;
//    input CE;
//    input CLR;
//    input T;
//   output Q;

//   wire TQ;
//   wire Q_DUMMY;

//   assign Q = Q_DUMMY;
```

```
//   XOR2  I_36_32 (.I0(T),
//                  .I1(Q_DUMMY),
//                  .O(TQ));
//   ///(* RLOC = "X0Y0" *)
//   FDCE  I_36_35 (.C(C),
//                  .CE(CE),
//                  .CLR(CLR),
//                  .D(TQ),
//                  .Q(Q_DUMMY));
//endmodule
`timescale 1ns / 1ps

module count4_MSCLK_clkcntrl4(C,
                CE,
               R,
               CEO,
               Q0,
               Q1,
               Q2,
               Q3,
               TC
    );
    input C;
    input CE;
    input R;
   output CEO;
   output Q0;
   output Q1;
   output Q2;
   output Q3;
   output TC;

    wire [3:0] D;

    assign D[0] = (~Q0);
    assign D[1] = (Q1^Q0);
    assign D[2] = (Q2^(Q1&Q0));
    assign D[3] = (Q3^(Q2&Q1&Q0));
    assign TC = Q3&Q2&Q1&Q0;
    assign CEO = TC&CE;

    FDRE #( .INIT(1'b0) ) MSCLK_CNT4ff_0 (.C(C), .R(R), .CE(CE), .D(D[0]), .Q(Q0));
    FDRE #( .INIT(1'b0) ) MSCLK_CNT4ff_1 (.C(C), .R(R), .CE(CE), .D(D[1]), .Q(Q1));
    FDRE #( .INIT(1'b0) ) MSCLK_CNT4ff_2 (.C(C), .R(R), .CE(CE), .D(D[2]), .Q(Q2));
    FDRE #( .INIT(1'b0) ) MSCLK_CNT4ff_3 (.C(C), .R(R), .CE(CE), .D(D[3]), .Q(Q3));
```

```
endmodule

//module CB4CE_MXILINX_clkcntrl4(C,
//                               CE,
//                               CLR,
//                               CEO,
//                               Q0,
//                               Q1,
//                               Q2,
//                               Q3,
//                               TC);

//    input C;
//    input CE;
//    input CLR;
//   output CEO;
//   output Q0;
//   output Q1;
//   output Q2;
//   output Q3;
//   output TC;

//   wire T2;
//   wire T3;
//   wire XLXN_1;
//   wire Q0_DUMMY;
//   wire Q1_DUMMY;
//   wire Q2_DUMMY;
//   wire Q3_DUMMY;
//   wire TC_DUMMY;

//   assign Q0 = Q0_DUMMY;
//   assign Q1 = Q1_DUMMY;
//   assign Q2 = Q2_DUMMY;
//   assign Q3 = Q3_DUMMY;
//   assign TC = TC_DUMMY;
//   (* HU_SET = "I_Q0_69" *)
//   FTCE_MXILINX_clkcntrl4 #( .INIT(1'b0) ) I_Q0 (.C(C),
//                                .CE(CE),
//                                .CLR(CLR),
//                                .T(XLXN_1),
//                                .Q(Q0_DUMMY));
//   (* HU_SET = "I_Q1_70" *)
//   FTCE_MXILINX_clkcntrl4 #( .INIT(1'b0) ) I_Q1 (.C(C),
//                                .CE(CE),
//                                .CLR(CLR),
```

```
//                                .T(Q0_DUMMY),
//                                .Q(Q1_DUMMY));
//   (* HU_SET = "I_Q2_71" *)
//   FTCE_MXILINX_clkcntrl4 #( .INIT(1'b0) ) I_Q2 (.C(C),
//                                .CE(CE),
//                                .CLR(CLR),
//                                .T(T2),
//                                .Q(Q2_DUMMY));
//   (* HU_SET = "I_Q3_72" *)
//   FTCE_MXILINX_clkcntrl4 #( .INIT(1'b0) ) I_Q3 (.C(C),
//                                .CE(CE),
//                                .CLR(CLR),
//                                .T(T3),
//                                .Q(Q3_DUMMY));
//   AND4  I_36_31 (.I0(Q3_DUMMY),
//                  .I1(Q2_DUMMY),
//                  .I2(Q1_DUMMY),
//                  .I3(Q0_DUMMY),
//                  .O(TC_DUMMY));
//   AND3  I_36_32 (.I0(Q2_DUMMY),
//                  .I1(Q1_DUMMY),
//                  .I2(Q0_DUMMY),
//                  .O(T3));
//   AND2  I_36_33 (.I0(Q1_DUMMY),
//                  .I1(Q0_DUMMY),
//                  .O(T2));
//   VCC  I_36_58 (.P(XLXN_1));
//   AND2  I_36_67 (.I0(CE),
//                  .I1(TC_DUMMY),
//                  .O(CEO));
//endmodule
```