Final Project:

Logic Lab

Erich Wanzek

July 30, 2019

ABSTRACT:

In this lab I created a set of Verilog code modules to construct a 60 second timer. The project consisted of creating several clock frequency divider modules, decimal and heximal counter modules and a top level to perform all the desired functions of a simple 60 second timer. The outline for this report will appear as follows:

1.) **Project Analysis:**
- A.) Analysis of Part 1
- B.) Analysis of Part 2
- C.) Analysis of Part 3

2.) **Wave Diagram Pictures:**
- A.) Wave Diagram Pictures of Part 1
  1. Wave diagram for divide_by_2 module
  2. Wave diagram for divide_by_5 module
  3. Wave diagram for divide_by_10 module
- B.) Wave Diagram Pictures of Part 2
  1. Wave diagram for decimal_counter module

3.) **Verilog Code:**
- A.) Code from Part 1
  1. divide_by_5 module
  2. divide_by_2 module
  3. divide_by_10 module
- B.) Code from Part 2
  1. decimal_counter
  2. heximal_counter
- C.) Code from Part 3
  1. top_level module
  2. Hex_7seg_bitwise module

4.) **Test Case Video**

Video of the top level performing (NOT IN REPORT, INCLUDED IN THE REPORT ZIP FILE)

Project Analysis

Analysis Part 1:

In Part one of the final project I constructed three different verilog modules. These modules were the divide_by_5 module, the divide_by_2 module and the divide_by_10 module. These modules all divide an input clock signal by their stated division factor.

The first clock divider module I made was the divide_by_5 module. This module is a divide by 5 frequency counter with a 50% duty cycle. It was requested in the final project document to create the divider module by using flip flops however the LAB TA Yan Pang said it was acceptable to use a counting register variable to count clock pulses in a always @ loop.

To make the divide_by_5 clock frequency divider it is necessary to add two intermediate divider clock signals that both have half the period of a period of 5 for both the positive and negative edge of the incoming clock to be divided. This is because a divide by_5 clock is and odd factor divider so it is necessary to count both positive and negative clock edges to get the correct clock division for a 50% duty cycle. The divide_by_5 clock module uses two always @ loops that count the number of positive clock edges and number of negative clock edges of the incoming clock signal. The two always@ both have a counter register that counts from 0 to 4 the number of respective clock edges. The two always@loop output their counter register variable as another intermediate clock and the two intermediate clocks (one derived from the positive edge and one derived from the negative edge) are added together (logical OR) to combine the two clocks into the divide by 5 clock signal. A waveform simulation was performed to verify that the module worked correctly. The waveform simulation diagram can be seen in Wave Diagram section of this report.

The next module created was the divide_by_two module. This module divides a clock frequency by a factor of two with a 50% duty cycle. The module was constructed using one always @ loop triggered at the positive edge of a clock signal which contains a counting register that counts from 0 to 1 and repeats(effectively dividing by 2). A waveform simulation was run to verify that the module worked correctly. The waveform simulation diagram can be found in the Wave Diagram section of this report.

The final divider module created was the divide_by_10 module. This module divides a clock frequency by a factor of 10 with a 50% duty cycle. The module was constructed by instatning the divide_by_2 module and the divide_by_5 module wiring them in series with an intermediate clock wire to effectively multiply there two divisor factors together. A waveform simulation was performed to verify that the module worked correctly. The waveform simulation diagram can be seen in the Wave Diagram section of this report.

Analysis Part 2:

In this part of the final project, I created a decimal counter. This module counts positive edge clock pulses from 0 to 9 then it repeats. The counter operates similarly for the modules for dividing a clock signal, the circuit simply counts the number of clock pulses. In this case, each time the positive edge of the clock being inputted into the clock input is encountered, the circuit increments register that stores the current digit. When the counter hits the digit 9 it rolls back over to zero and outputs an Overflow bit value of 1 which is necessary to feed into the next counter for the next significant digit for the time circuit. A wave simulation was performed for the decimal counter, and upon analysis of the wave diagram it was determined that the decimal counter was functioning properly. The wave simulation diagram for the dcimal_counter module can be found in the wave diagram section of this report.

Part two of the final project only called to develop a decimal counter that has a reset bit and only counts in the up direction. In Part 3 of the final project it is necessary to create a heximal counter which is very similar in operation and code structure to the decimal counter. Additionally it is called upon in part 3 of the project to add additional features and conditions for counting up and down and a start/stop hold switch to both the heximal and the decimal counter. All these additions to the heximal and decimal counter were implemented by adding more if else conditional statements to run the counter for the various conditions. THese extra features to the timers make it possible to make a fully functional 60 second counter in the top_level that can count up or down and start and stop at any time.

Analysis Part 3:

In part 3 of the final project, I constructed the top level for the 60 second timer circuit. The timer circuit itself physically consists of three hex displays and three switches on the FPGA board. The top level takes in inputs from switches 3 to 0 (SW[3:0]) and outputs to the HEX0 display, the HEX1 display, and the HEX2 display. The first switch is the start/stop hold switch which pauses the timer. The second switch is the direction switch which controls if the timer counts up or down. The third switch is the reset switch, which when placed to off, resets the timer to 59.9 seconds.

The first part of the verilog code for the top level consists of a cascade of divider modules to divide the onboard FPGA hardware clock signal of 50 MHz down to the desired 10 Hz signal to drive the first decimal counter for the 0.1 second place of the first hex display. The cascade of divider modules consists of one divide_by_5 module and six divide_by_10 modules cascaded in series.

The next part of the verilog code for the top level consists of two decimal counters and one heximal counter. The first decimal counter counts from 0 to 9 (or 9 to 0) for the 0.1 seconds place and outputs an overflow value into the clock input of the subsequent counter. The subsequent decimal counter counts from 0 to 9 (or 9 to 0) for the 1.0 seconds place of the timer and outputs an overflow value into the clock input of the subsequent heximal counter. The Heximal counter counts the 10.0s

of seconds place, its counts from 0 to 6 (or 6 to 0). The three counters combined together creates a 60 counter that counts from 00.0 seconds to 59.9 or vice versa.

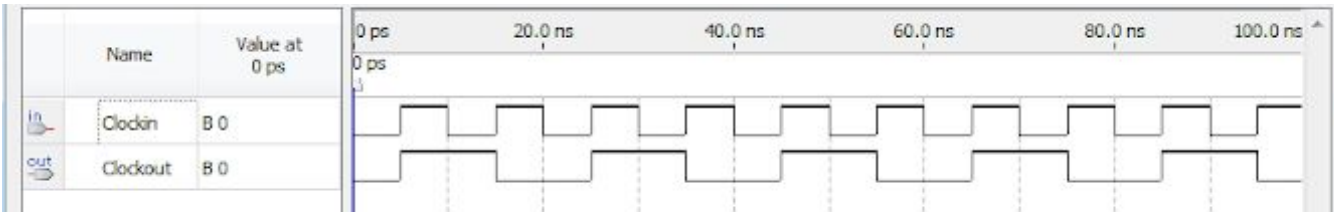
The final part of the verilog code for the top level consists of feeding in the 4bit wide output registers from the decimal and heximal counters into their corresponding instatiations of the hex_7seg_bitwise driver module which drives the corresponding HEX displays. The HEX0 display displays the output of the 1st decimal counter and represents 0.1 seconds. The HEX2 display displays the output of the 2nd decimal counter and represents 1.0 seconds. The HEX3 display displays the output of the heximal counter and represents 10.0 seconds to 60 seconds. The top level timer circuit was uploaded onto the Altera FPGA board and performed as designed. A video of the timer top level module in operation is included the Final Project ZIP file.

Wave Diagram Pictures

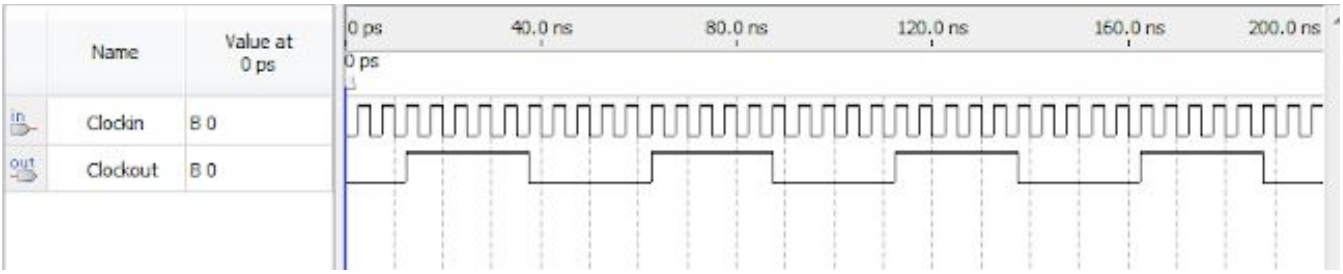
Wave Diagram Pictures from Part 1

Wave Simulation Diagram for divide_by_5 counter module

Wave Simulation Diagram for divide_by_2 counter module

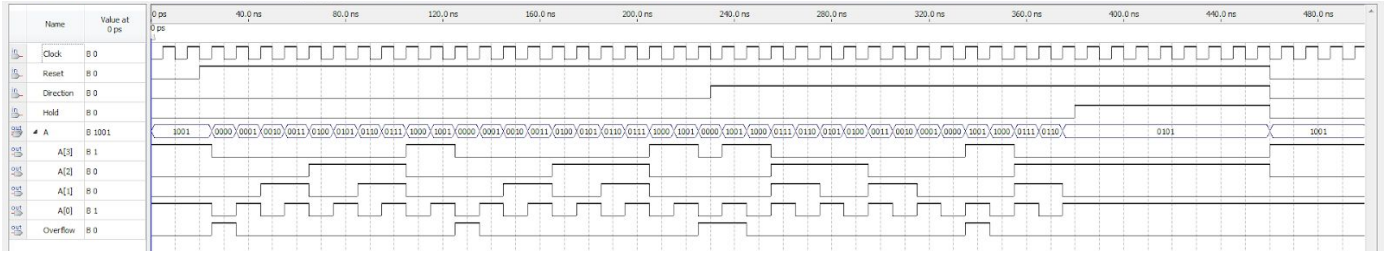


Wave Simulation Diagram for divide_by_10 counter module



Wave Diagram Pictures from Part 2

Wave Simulation Diagram for the decimal_coutner module



Verilog Codes

A.) Code from Part 1

1. divide_by_5 module

```
module divide_by_5(Clockin, Clockout); //Defines clock divide_by_5 module

        input Clockin; // Declares Clockin input
        output Clockout; //Declares Clockout output


        reg [2:0]counter1; //Declares 3 bit wide register counter1
        reg [2:0]counter2; //Declares 3 bit wide register counter2


        wire clkout1; //Declares wire for clkout1
        wire clkout2; //Declares wire for clkout2


        always @(posedge Clockin) begin //Initiates always @ loop with positive edge clock trigger
                if(counter1 == 4 ) //If counter1 reaches 4 reset counter to zero
                counter1 <= 0;                  //If counter1 reaches 4 reset counter to zero

                else
                counter1 <= counter1 + 1 ;  //else add value of 1 to counter1
        end


        always @(negedge Clockin) begin //Initiates always @ loop with negatve edge clock trigger
                if(counter2 == 4)       //If counter2 reaches 4 for reset counter to zero
                counter2 <= 0;          //If counter2 reaches 4 for reset counter to zero

                else
                counter2 <= counter2+ 1; //else add value of 1 to counter2
        end


        assign clkout1  = counter1[1]; //assings the value of clkout1 to the 2nd bit of the register counter1
        assign clkout2  = counter2[1]; //assings the value of clkout2 to the 2nd bit of the register counter2

        assign Clockout = clkout1 | clkout2; //Adds clkout1 to clkout2 by logical or


endmodule //ends module
```

2. divide_by_2 module

```
module divide_by_2(Clockin, Clockout); //Defines clock divide_by_2 module

        input Clockin;   //Declares Clockin input
        output Clockout; //Declares Clockout output

        reg [2:0]counter1; //Declares 3 bit wide register counter1

        wire clkout1; //Declares wire for clkout1

        always @(posedge Clockin) begin //Initiates always @ loop with positive edge clock trigger
                if(counter1 == 1 ) //If counter1 reaches 1, reset counter to zero
                counter1 <= 0;          //If counter1 reaches 1, reset counter to zero
                else
                counter1 <= counter1 + 1 ; //else add value of 1 to counter1
        end


        assign clkout1  = counter1[0]; //assigns the value of clkout1 to the 1st bit of the register counter1
        assign Clockout = clkout1;     //assigns Clockout to clkout1

endmodule //Ends module
```

3. divide_by_10 module

```
module divide_by_10(Clockin, Clockout); //Defines clock divide_by_10 module

        input Clockin;   //Declares Clockin input
        output Clockout; //Declares Clockout output

        wire clock;  //Declares a clock wire

        divide_by_2(Clockin, clock);  //Instantiates the divide_by_2 module, feeds clock wire to next module
        divide_by_5(clock, Clockout); //Instantiates the divide_by_5 module, outputs clockout

endmodule //ends module
```

B.) Code from Part 2

1. decimal_coutner module

```
module decimal_counter(A, Overflow, Clock, Reset, Direction, Hold); //Declares the decimal_counter module

        input Clock, Reset, Direction, Hold; //Declares the Clock, Reset, Direction, Hold inputs
        output Overflow; //Declares Overflow output
        output [3:0]A;   //Declares 4 bit output A
        reg rOverflow;          //Declares regsiter rOverflow
        reg [3:0]rA;            //Declares 4 bit register rA

        always@(posedge Clock or negedge Reset) //Initiates always @ loop with positive edge clock trigger or negative edge
Rest trigger
                if (Reset ==1'd0) //if Reset is equal to decimal zero:
                        begin
                        rOverflow <= 1'b0; //Set rOverflow register to binary 0
                        rA <= 4'd9; //Set rA register to decimal 9 in binary
                        end

                //If register rA is less than decimal 9  and direction is set to binary zero and hold is set to binary zero:
                else if ((rA < 4'd9)&&(Direction == 1'b0)&&(Hold == 1'b0))
                        begin
                        rA <= rA + 1'b1;   //Add 1 binary bit to rA register
                        rOverflow <= 1'b0; //Set rOverflow register to binary 0
                        end

                //If register rA is equal to decimal 9 and direction is set to binary zero and hold is set to binary zero:
                else if ((rA == 4'd9)&&(Direction == 1'b0)&&(Hold == 1'b0))
                        begin
                        rA <=4'b0000; // Set rA register to binary 0000
                        rOverflow <= 1'b1; //Set rOverflow register to binary 1
                        end

                //If register rA is greater than decimal 0  and direction is set to binary 1 and hold is set to binary zero:
                else if ((rA > 4'd0)&&(Direction == 1'b1)&&(Hold == 1'b0))
                        begin
                        rA <= rA - 1'b1;   //Subtract 1 binary bit from rA register
                        rOverflow <= 1'b0; //Set rOverflow register to binary 0
                        end

                //If register rA is equal to decimal 9  and direction is set to binary one and hold is set to binary zero:
                else if ((rA == 4'd0)&&(Direction == 1'b1)&&(Hold == 1'b0))
                        begin
                        rA <=4'b1001; // Set rA register to binary 1001
                        rOverflow <= 1'b1; //Set rOverflow register to binary 1
                        end

        //else statement for all other scenarios
                else
                        begin
                        rA <= rA; //reset register rA to current rA value
                        rOverflow <= 1'b0; //Set rOverflow register to binary 0
                        end

        assign Overflow = rOverflow; //Assigns Overflow Output to register rOverflow
        assign A = rA; // Assigns Output A to register rA

endmodule //ends module
```

2. heximal_counter module

```
module hex_counter(A, Overflow, Clock, Reset, Direction, Hold); //Declares the hex_counter module

        input Clock, Reset, Direction, Hold; //Declares the Clock, Reset, Direction, Hold inputs
        output Overflow;  //Declares Overflow output
        output [3:0]A;                  //Declares 4 bit output A

        reg rOverflow;                  //Declares regsiter rOverflow
        reg [3:0]rA;                    //Declares 4 bit register rA

        always@(posedge Clock or negedge Reset) //Initiates always @ loop with positive edge clock trigger or negative edge
Rest trigger

                //if Reset is equal to decimal zero:
                if (Reset ==1'd0)
                        begin
                        rOverflow <= 1'b0;  //Set rOverflow register to binary 0
                        rA <= 4'd5;         //Set rA register to decimal 5 in binary
                        end

                //If register rA is less than decimal 5  and direction is set to binary zero and hold is set to binary zero:
                else if ((rA < 4'd5)&&(Direction == 1'b0)&&(Hold == 1'b0))
                        begin
                        rA <= rA + 1'b1;     //Add 1 binary bit to rA register
                        rOverflow <= 1'b0;  //Set rOverflow register to binary 0
                        end

                //If register rA is greater than decimal 0  and direction is set to binary 1 and hold is set to binary zero:
                else if ((rA > 4'd0)&&(Direction == 1'b1)&&(Hold == 1'b0))
                        begin
                        rA <= rA - 1'b1;    //Subtract 1 binary bit from rA register
                        rOverflow <= 1'b0;  //Set rOverflow register to binary 0
                        end

                //If register rA is equal to decimal 5 and direction is set to binary zero and hold is set to binary zero:
                else if ((rA == 4'd5)&&(Direction == 1'b0)&&(Hold == 1'b0))
                        begin
                        rA <=4'b0000;       //Set rA register to binary 0000
                        rOverflow <= 1'b1;  //Set rOverflow register to binary 1
                        end

                //If register rA is equal to decimal 5  and direction is set to binary one and hold is set to binary zero:
                else if ((rA == 4'd0)&&(Direction == 1'b1)&&(Hold == 1'b0))
                        begin
                        rA <=4'b0101;       //Set rA register to binary 0101
                        rOverflow <= 1'b1;  //Set rOverflow register to binary 1
                        end

                //else statement for all other scenarios:
                else
                        begin
                        rA <= rA;                       //reset register rA to current rA value
                        rOverflow <= 1'b0; //Set rOverflow register to binary 0
                        end

        assign Overflow = rOverflow; //Assigns Overflow Output to register rOverflow
        assign A = rA; // Assigns Output A to register rA

endmodule //ends module
```

B.) Code from Part 3

1. top_level module

```
module top_level(CLOCK_50,SW, HEX0, HEX1, HEX2, LEDR);// Defines the top_level for the timer circuit
        input wire CLOCK_50; // Declares input hardware 50MHz Clock
        input wire [2:0]SW; //Declares 3 input switches
        output [2:0]LEDR; //Declares 3 output LEDR wires

        assign LEDR[2:0] = SW[2:0]; // Assigns the 3 LEDRS to the 3 switches

        wire Reset; // Declares the Reset wire
        wire Direction; // Declares the Direction wire
        wire Hold; // Declares the Hold wire

        assign Reset = SW[2];       //Assign Reset wire to switch 2
        assign Direction = SW[1];         //Assign Direction wire switch 1
        assign Hold = SW[0];        //Assign Hold wire to switch 0

        output wire [6:0]HEX0; //defines output wires for HEX0 display.
        output wire [6:0]HEX1; //defines output wires for HEX1 display.
        output wire [6:0]HEX2; //defines output wires for HEX2 display.

        wire c1, c2, c3, c4, c5, c6, c7; //Declare intermediate clock wires

        wire [3:0]A0; //Declare 3 bit Data A0 wire
        wire [3:0]A1; //Declare 3 bit Data A1 wire
        wire [3:0]A2; //Declare 3 bit Data A2 wire


        //Instantiate the divider modules and cascade outputs to get correct clock frequency

        divide_by_5 inst0(CLOCK_50, c1); // Instantiate the divide_by_5 module, output 10MHz
        divide_by_10 inst1 (c1, c2); //Instantiate the divide_by_10 module #1 output 1MHz
        divide_by_10 inst2 (c2, c3); //Instantiate the divide_by_10 module #2 output 0.1MHZ
        divide_by_10 inst3 (c3, c4); //Instantiate the divide_by_10 module #3 output 0.01MHz
        divide_by_10 inst4 (c4, c5); //Instantiate the divide_by_10 module #4 output 1KHz
        divide_by_10 inst5 (c5, c6); //Instantiate the divide_by_10 module #5 output 0.1KHz
        divide_by_10 inst6 (c6, c7); //Instantiate the divide_by_10 module #6 output 0.1HZ


        wire Overflow0, Overflow1, Overflow2; //Declare the Overflow wires to feed into successive counter modules


        decimal_counter inst9(A0, Overflow0, c7, Reset, Direction, Hold); //Instantiate the decimal_counter module
        decimal_counter inst10(A1, Overflow1, Overflow0, Reset, Direction, Hold); //Instantiate the decimal_counter module
        hex_counter inst11(A2, Overflow2, Overflow1, Reset, Direction, Hold); //Instantiate the hex_counter module


        hex_7seg_bitwise inst12(A0, HEX0); //Instantiate the hex_7seg_bitwise for HEX display 0
        hex_7seg_bitwise inst13(A1, HEX1); //Instantiate the hex_7seg_bitwise for HEX display 1
        hex_7seg_bitwise inst14(A2, HEX2); //Instantiate the hex_7seg_bitwise for HEX display 2

endmodule //ends module
```

2. Hex_7seg_bitwise module

```
module hex_7seg_bitwise (X, segment); //module name 7 segment display, part 1e

        input wire [3:0]X; //defines a wire that's 4 bits wide
        output wire [6:0]segment; //defines an output wire that is 7 bits wide

        // Base for copying and pasting to make typing segments easier: A[3]&B[2]&C[1]&D[0] |

        // S0 = Sum(1,4,11,13)
        // Segment[0] = ~A[3]&~B[2]&~C[1]&D[0] | ~A[3]&B[2]&~C[1]&~D[0] | A[3]&B[2]&~C[1]&D[0] | A[3]&~B[2]&C[1]&D[0];
        assign segment[0] = ~X[3]&~X[2]&~X[1]&X[0] | ~X[3]&X[2]&~X[1]&~X[0] | X[3]&X[2]&~X[1]&X[0] | X[3]&~X[2]&X[1]&X[0];


        // S1 = Sum(5,6,11,12,14,15)
        // Segment[1] = A[3]&B[2]&~D[0] | ~A[3]&B[2]&~C[1]&D[0] | B[2]&C[1]&~D[0] | A[3]&C[1]&D[0];
        assign segment[1] = X[3]&X[2]&~X[0] | ~X[3]&X[2]&~X[1]&X[0] | X[2]&X[1]&~X[0] | X[3]&X[1]&X[0];


        // S2 = Sum(2,12,14,15)
        // Segment[2] = ~A[3]&~B[2]&C[1]&~D[0] | A[3]&B[2]&~D[0] | A[3]&B[2]&C[1];
        assign segment[2] = ~X[3]&~X[2]&X[1]&~X[0] | X[3]&X[2]&~X[0] | X[3]&X[2]&X[1];


        // S3 = Sum(1,4,7,9,10,15)
        // Segment[3] = ~A[3]&B[2]&~C[1]&~D[0] | ~B[2]&~C[1]&D[0] | B[2]&C[1]&D[0] | A[3]&~B[2]&C[1]&~D[0]
        assign segment[3] = ~X[3]&X[2]&~X[1]&~X[0] | ~X[2]&~X[1]&X[0] | X[2]&X[1]&X[0] | X[3]&~X[2]&X[1]&~X[0];


        // S4 = Sum(1,3,4,5,7,9)
        // Segment[4] = ~A[3]&B[2]&~C[1] | ~A[3]&D[0] | ~B[2]&~C[1]&D[0]
        assign segment[4] = ~X[3]&X[2]&~X[1] | ~X[3]&X[0] | ~X[2]&~X[1]&X[0];


        // S5 = Sum(1,2,3,7,13)
        // Segment[5] = ~A[3]&~B[2]&D[0] | ~A[3]&C[1]&D[0] | ~A[3]&~B[2]&C[1] | A[3]&B[2]&~C[1]&D[0]
        assign segment[5] = ~X[3]&~X[2]&X[0] | ~X[3]&X[1]&X[0] | ~X[3]&~X[2]&X[1] | X[3]&X[2]&~X[1]&X[0];


        // S6 = Sum(0,1,7,12)
        // Segment[6] = ~A[3]&~B[2]&~C[1] |  ~A[3]&B[2]&C[1]&D[0] |  A[3]&B[2]&~C[1]&~D[0]
        assign segment[6] = ~X[3]&~X[2]&~X[1] |  ~X[3]&X[2]&X[1]&X[0] |  X[3]&X[2]&~X[1]&~X[0];

endmodule //ends module
```