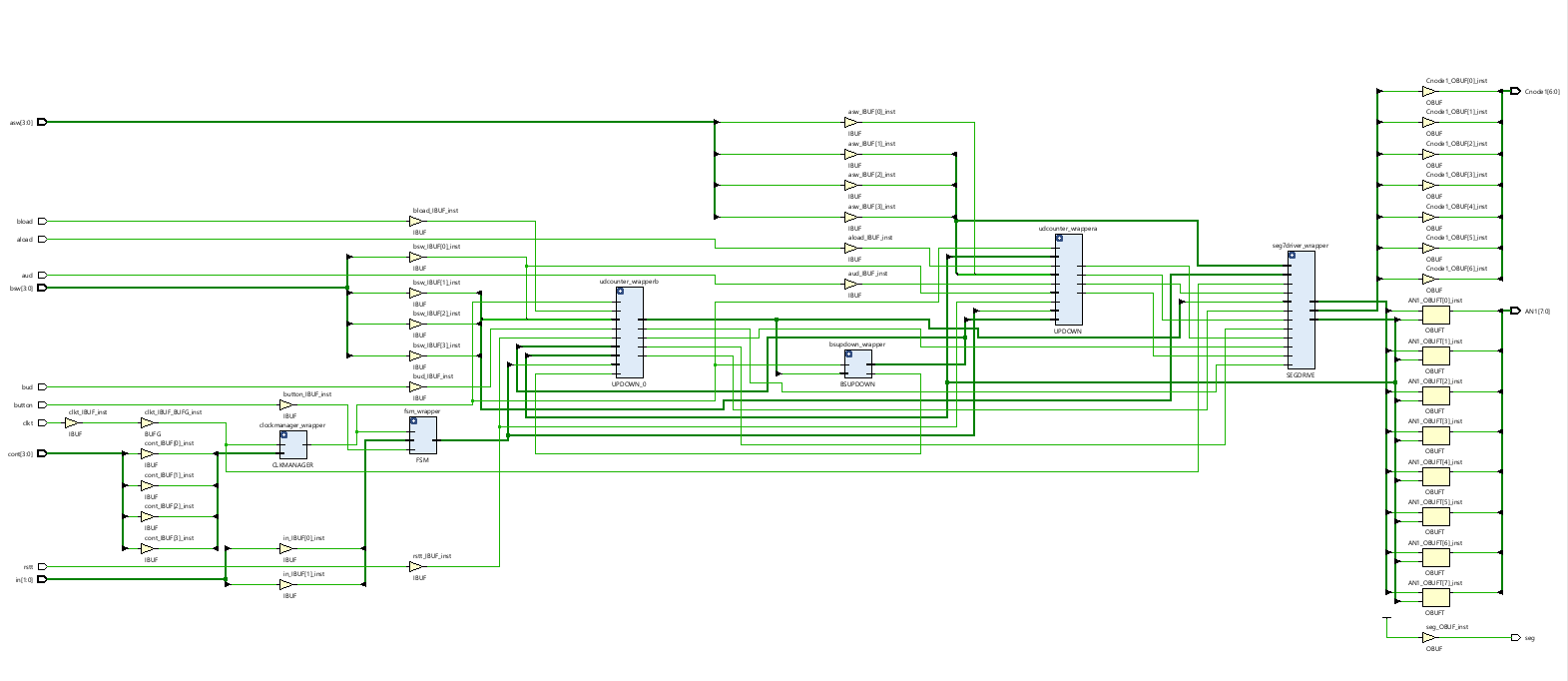
**ECE3300L Lab8 Group H Report (Sherwin Sathish & Mohamed Hamida)**

**SCHEMATIC:**

****

***Xdc for top.v:***

## This file is a general .xdc for the Nexys A7-100T

## To use it in a project:

## - uncomment the lines corresponding to used pins

## - rename the used ports (in each line, after get\_ports) according to the top level signal names in the project

## Clock signal

set\_property -dict { PACKAGE\_PIN E3 IOSTANDARD LVCMOS33 } [get\_ports { clkt}];# IO\_L12P\_T1\_MRCC\_35 Sch=clk100mhz

create\_clock -add -name sys\_clk\_pin -period 10.00 -waveform {0 5} [get\_ports {clkt}];

##Switches

set\_property -dict { PACKAGE\_PIN J15 IOSTANDARD LVCMOS33 } [get\_ports { asw[0] }]; #IO\_L24N\_T3\_RS0\_15 Sch=sw[0]

set\_property -dict { PACKAGE\_PIN L16 IOSTANDARD LVCMOS33 } [get\_ports { asw[1] }]; #IO\_L3N\_T0\_DQS\_EMCCLK\_14 Sch=sw[1]

set\_property -dict { PACKAGE\_PIN M13 IOSTANDARD LVCMOS33 } [get\_ports { asw[2] }]; #IO\_L6N\_T0\_D08\_VREF\_14 Sch=sw[2]

set\_property -dict { PACKAGE\_PIN R15 IOSTANDARD LVCMOS33 } [get\_ports { asw[3] }]; #IO\_L13N\_T2\_MRCC\_14 Sch=sw[3]

set\_property -dict { PACKAGE\_PIN R17 IOSTANDARD LVCMOS33 } [get\_ports { bsw[0] }]; #IO\_L12N\_T1\_MRCC\_14 Sch=sw[4]

set\_property -dict { PACKAGE\_PIN T18 IOSTANDARD LVCMOS33 } [get\_ports { bsw[1] }]; #IO\_L7N\_T1\_D10\_14 Sch=sw[5]

set\_property -dict { PACKAGE\_PIN U18 IOSTANDARD LVCMOS33 } [get\_ports { bsw[2] }]; #IO\_L17N\_T2\_A13\_D29\_14 Sch=sw[6]

set\_property -dict { PACKAGE\_PIN R13 IOSTANDARD LVCMOS33 } [get\_ports { bsw[3] }]; #IO\_L5N\_T0\_D07\_14 Sch=sw[7]

set\_property -dict { PACKAGE\_PIN T8 IOSTANDARD LVCMOS18 } [get\_ports { cont[0] }]; #IO\_L24N\_T3\_34 Sch=sw[8]

set\_property -dict { PACKAGE\_PIN U8 IOSTANDARD LVCMOS18 } [get\_ports { cont[1] }]; #IO\_25\_34 Sch=sw[9]

set\_property -dict { PACKAGE\_PIN R16 IOSTANDARD LVCMOS33 } [get\_ports { cont[2] }]; #IO\_L15P\_T2\_DQS\_RDWR\_B\_14 Sch=sw[10]

set\_property -dict { PACKAGE\_PIN T13 IOSTANDARD LVCMOS33 } [get\_ports { cont[3] }]; #IO\_L23P\_T3\_A03\_D19\_14 Sch=sw[11]

set\_property -dict { PACKAGE\_PIN H6 IOSTANDARD LVCMOS33 } [get\_ports { aud }]; #IO\_L24P\_T3\_35 Sch=sw[12]

set\_property -dict { PACKAGE\_PIN U12 IOSTANDARD LVCMOS33 } [get\_ports { bud }]; #IO\_L20P\_T3\_A08\_D24\_14 Sch=sw[13]

set\_property -dict { PACKAGE\_PIN U11 IOSTANDARD LVCMOS33 } [get\_ports { in[0] }]; #IO\_L19N\_T3\_A09\_D25\_VREF\_14 Sch=sw[14]

set\_property -dict { PACKAGE\_PIN V10 IOSTANDARD LVCMOS33 } [get\_ports { in[1] }]; #IO\_L21P\_T3\_DQS\_14 Sch=sw[15]

## LEDs

#set\_property -dict { PACKAGE\_PIN H17 IOSTANDARD LVCMOS33 } [get\_ports { sumbit[0] }]; #IO\_L18P\_T2\_A24\_15 Sch=led[0]

#set\_property -dict { PACKAGE\_PIN K15 IOSTANDARD LVCMOS33 } [get\_ports { sumbit[1] }]; #IO\_L24P\_T3\_RS1\_15 Sch=led[1]

#set\_property -dict { PACKAGE\_PIN J13 IOSTANDARD LVCMOS33 } [get\_ports { sumbit[2] }]; #IO\_L17N\_T2\_A25\_15 Sch=led[2]

#set\_property -dict { PACKAGE\_PIN N14 IOSTANDARD LVCMOS33 } [get\_ports { sumbit[3] }]; #IO\_L8P\_T1\_D11\_14 Sch=led[3]

#set\_property -dict { PACKAGE\_PIN R18 IOSTANDARD LVCMOS33 } [get\_ports { coutbit }]; #IO\_L7P\_T1\_D09\_14 Sch=led[4]

#set\_property -dict { PACKAGE\_PIN V17 IOSTANDARD LVCMOS33 } [get\_ports { LED[5] }]; #IO\_L18N\_T2\_A11\_D27\_14 Sch=led[5]

#set\_property -dict { PACKAGE\_PIN U17 IOSTANDARD LVCMOS33 } [get\_ports { LED[6] }]; #IO\_L17P\_T2\_A14\_D30\_14 Sch=led[6]

#set\_property -dict { PACKAGE\_PIN U16 IOSTANDARD LVCMOS33 } [get\_ports { LED[7] }]; #IO\_L18P\_T2\_A12\_D28\_14 Sch=led[7]

#set\_property -dict { PACKAGE\_PIN V16 IOSTANDARD LVCMOS33 } [get\_ports { LED[8] }]; #IO\_L16N\_T2\_A15\_D31\_14 Sch=led[8]

#set\_property -dict { PACKAGE\_PIN T15 IOSTANDARD LVCMOS33 } [get\_ports { LED[9] }]; #IO\_L14N\_T2\_SRCC\_14 Sch=led[9]

#set\_property -dict { PACKAGE\_PIN U14 IOSTANDARD LVCMOS33 } [get\_ports { LED[10] }]; #IO\_L22P\_T3\_A05\_D21\_14 Sch=led[10]

#set\_property -dict { PACKAGE\_PIN T16 IOSTANDARD LVCMOS33 } [get\_ports { LED[11] }]; #IO\_L15N\_T2\_DQS\_DOUT\_CSO\_B\_14 Sch=led[11]

#set\_property -dict { PACKAGE\_PIN V15 IOSTANDARD LVCMOS33 } [get\_ports { LED[12] }]; #IO\_L16P\_T2\_CSI\_B\_14 Sch=led[12]

#set\_property -dict { PACKAGE\_PIN V14 IOSTANDARD LVCMOS33 } [get\_ports { LED[13] }]; #IO\_L22N\_T3\_A04\_D20\_14 Sch=led[13]

#set\_property -dict { PACKAGE\_PIN V12 IOSTANDARD LVCMOS33 } [get\_ports { LED[14] }]; #IO\_L20N\_T3\_A07\_D23\_14 Sch=led[14]

#set\_property -dict { PACKAGE\_PIN V11 IOSTANDARD LVCMOS33 } [get\_ports { LED[15] }]; #IO\_L21N\_T3\_DQS\_A06\_D22\_14 Sch=led[15]

## RGB LEDs

#set\_property -dict { PACKAGE\_PIN R12 IOSTANDARD LVCMOS33 } [get\_ports { LED16\_B }]; #IO\_L5P\_T0\_D06\_14 Sch=led16\_b

#set\_property -dict { PACKAGE\_PIN M16 IOSTANDARD LVCMOS33 } [get\_ports { LED16\_G }]; #IO\_L10P\_T1\_D14\_14 Sch=led16\_g

#set\_property -dict { PACKAGE\_PIN N15 IOSTANDARD LVCMOS33 } [get\_ports { LED16\_R }]; #IO\_L11P\_T1\_SRCC\_14 Sch=led16\_r

#set\_property -dict { PACKAGE\_PIN G14 IOSTANDARD LVCMOS33 } [get\_ports { LED17\_B }]; #IO\_L15N\_T2\_DQS\_ADV\_B\_15 Sch=led17\_b

#set\_property -dict { PACKAGE\_PIN R11 IOSTANDARD LVCMOS33 } [get\_ports { LED17\_G }]; #IO\_0\_14 Sch=led17\_g

#set\_property -dict { PACKAGE\_PIN N16 IOSTANDARD LVCMOS33 } [get\_ports { LED17\_R }]; #IO\_L11N\_T1\_SRCC\_14 Sch=led17\_r

##7 segment display

set\_property -dict { PACKAGE\_PIN T10 IOSTANDARD LVCMOS33 } [get\_ports { Cnode1[6] }]; #IO\_L24N\_T3\_A00\_D16\_14 Sch=ca

set\_property -dict { PACKAGE\_PIN R10 IOSTANDARD LVCMOS33 } [get\_ports { Cnode1[5] }]; #IO\_25\_14 Sch=cb

set\_property -dict { PACKAGE\_PIN K16 IOSTANDARD LVCMOS33 } [get\_ports { Cnode1[4] }]; #IO\_25\_15 Sch=cc

set\_property -dict { PACKAGE\_PIN K13 IOSTANDARD LVCMOS33 } [get\_ports { Cnode1[3] }]; #IO\_L17P\_T2\_A26\_15 Sch=cd

set\_property -dict { PACKAGE\_PIN P15 IOSTANDARD LVCMOS33 } [get\_ports { Cnode1[2] }]; #IO\_L13P\_T2\_MRCC\_14 Sch=ce

set\_property -dict { PACKAGE\_PIN T11 IOSTANDARD LVCMOS33 } [get\_ports { Cnode1[1] }]; #IO\_L19P\_T3\_A10\_D26\_14 Sch=cf

set\_property -dict { PACKAGE\_PIN L18 IOSTANDARD LVCMOS33 } [get\_ports { Cnode1[0] }]; #IO\_L4P\_T0\_D04\_14 Sch=cg

set\_property -dict { PACKAGE\_PIN H15 IOSTANDARD LVCMOS33 } [get\_ports { seg }]; #IO\_L19N\_T3\_A21\_VREF\_15 Sch=dp

set\_property -dict { PACKAGE\_PIN J17 IOSTANDARD LVCMOS33 } [get\_ports { AN1[0] }]; #IO\_L23P\_T3\_FOE\_B\_15 Sch=an[0]

set\_property -dict { PACKAGE\_PIN J18 IOSTANDARD LVCMOS33 } [get\_ports { AN1[1] }]; #IO\_L23N\_T3\_FWE\_B\_15 Sch=an[1]

set\_property -dict { PACKAGE\_PIN T9 IOSTANDARD LVCMOS33 } [get\_ports { AN1[2] }]; #IO\_L24P\_T3\_A01\_D17\_14 Sch=an[2]

set\_property -dict { PACKAGE\_PIN J14 IOSTANDARD LVCMOS33 } [get\_ports { AN1[3] }]; #IO\_L19P\_T3\_A22\_15 Sch=an[3]

set\_property -dict { PACKAGE\_PIN P14 IOSTANDARD LVCMOS33 } [get\_ports { AN1[4] }]; #IO\_L8N\_T1\_D12\_14 Sch=an[4]

set\_property -dict { PACKAGE\_PIN T14 IOSTANDARD LVCMOS33 } [get\_ports { AN1[5] }]; #IO\_L14P\_T2\_SRCC\_14 Sch=an[5]

set\_property -dict { PACKAGE\_PIN K2 IOSTANDARD LVCMOS33 } [get\_ports { AN1[6] }]; #IO\_L23P\_T3\_35 Sch=an[6]

set\_property -dict { PACKAGE\_PIN U13 IOSTANDARD LVCMOS33 } [get\_ports { AN1[7] }]; #IO\_L23N\_T3\_A02\_D18\_14 Sch=an[7]

##Buttons

set\_property -dict { PACKAGE\_PIN C12 IOSTANDARD LVCMOS33 } [get\_ports { rstt }]; #IO\_L3P\_T0\_DQS\_AD1P\_15 Sch=cpu\_resetn

set\_property -dict { PACKAGE\_PIN N17 IOSTANDARD LVCMOS33 } [get\_ports { aload }]; #IO\_L9P\_T1\_DQS\_14 Sch=btnc

set\_property -dict { PACKAGE\_PIN M18 IOSTANDARD LVCMOS33 } [get\_ports { bload }]; #IO\_L4N\_T0\_D05\_14 Sch=btnu

set\_property -dict { PACKAGE\_PIN P17 IOSTANDARD LVCMOS33 } [get\_ports { button }]; #IO\_L12P\_T1\_MRCC\_14 Sch=btnl

#set\_property -dict { PACKAGE\_PIN M17 IOSTANDARD LVCMOS33 } [get\_ports { BTNR }]; #IO\_L10N\_T1\_D15\_14 Sch=btnr

#set\_property -dict { PACKAGE\_PIN P18 IOSTANDARD LVCMOS33 } [get\_ports { BTND }]; #IO\_L9N\_T1\_DQS\_D13\_14 Sch=btnd

**Code Detail:**

`timescale 1ns / 1ps

module CLKMANAGER

(

input clk,

input rst,

input [3:0] SW,

output reg clkout =0

);

reg [31:0] sel;

wire [4:0] temp = {1'b1,SW};

always@(posedge clk or posedge rst)

begin: DREG

if(rst)

sel<= 32'd0;

else

sel<=sel+1;

end

always@(posedge clk)

begin

case(SW)

5'd0: clkout <= sel[0];

5'd1: clkout <= sel[1];

5'd2: clkout <= sel[2];

5'd3: clkout <= sel[3];

5'd4: clkout <= sel[4];

5'd5: clkout <= sel[5];

5'd6: clkout <= sel[6];

5'd7: clkout <= sel[7];

5'd8: clkout <= sel[8];

5'd9: clkout <= sel[9];

5'd10:clkout <= sel[10];

5'd11:clkout <= sel[11];

5'd12:clkout <= sel[12];

5'd13:clkout <= sel[13];

5'd14:clkout <= sel[14];

5'd15:clkout <= sel[15];

5'd16:clkout <= sel[16];

5'd17:clkout <= sel[17];

5'd18:clkout <= sel[18];

5'd19:clkout <= sel[19];

5'd20:clkout <= sel[20];

5'd21:clkout <= sel[21];

5'd22:clkout <= sel[22];

5'd23:clkout <= sel[23];

5'd24:clkout <= sel[24];

5'd25:clkout <= sel[25];

5'd26:clkout <= sel[26];

5'd27:clkout <= sel[27];

5'd28:clkout <= sel[28];

5'd29:clkout <= sel[29];

5'd30:clkout <= sel[30];

5'd31:clkout <= sel[31];

endcase

end

endmodule

The purpose of the clock manager is to have a counter and a 4-bit mux to output a clk with various levels of frequency depending on the integer div in this case. First we set up the counter by using an always loop that accounts for the reset, and sets up the different frequencies for all 16 values. Then we created an always loop that takes effect when the switches are changed, and we used a case statement for all 16 values of the 4-bit switches. Depending on the clkout that is selected by the switches, the frequency could vary for each clkout. We used the non-blocking assignment <= so that the code doesn’t have to be done in order from top to bottom.

`timescale 1ns / 1ps

module UPDOWN(

input clk1,

input rst1,

input load,

input ud,

input [3:0] a,

output reg [3:0] tmp=0

);

always@(posedge clk1 or posedge rst1)

begin

if(load)

begin

tmp=a;

end

else

begin

if(rst1)

begin

if(ud)// 0 is up, 1 is down

tmp<= 4'd9;

else

tmp<= 4'd0;

end

else

begin

if(ud)

if(tmp==4'd0)

tmp <= 4'd9;

else

tmp <= tmp-1;

else

if(tmp==4'd9)

tmp <= 4'd0;

else

tmp <= tmp+1;

end

end

end

endmodule

`timescale 1ns / 1ps

module UPDOWN\_TB(

);

reg clk\_tb,rst\_tb;

reg l\_tb;

reg ud\_tb;

reg [3:0] atb;

wire [3:0] cnt\_tb;

UPDOWN COMP (

.clk1(clk\_tb),

.rst1(rst\_tb),

.load(l\_tb),

.ud(ud\_tb),

.a(atb),

.tmp(cnt\_tb)

);

initial

begin

clk\_tb = 0;

rst\_tb = 0;

ud\_tb = 0;

l\_tb= 1;

atb= 3;

end

always

begin

#5 clk\_tb = ~clk\_tb;

end

always

begin

#20 l\_tb = ~l\_tb;

end

endmodule

The purpose of this code is to count up or down in BCD. The always@(posedge or rst) was used because this is a sequential circuit , and the non-blocking assignment <= is used as well because this code runs in parallel. The if statement with en is used first because without the en the updown counter wouldn’t run. Next there is a if statement to see if the reset value is 1, if it is then we need to see whether the updown counter is 0 for up or 1 for down. This is because if we are counting from 9 to 0 then the value should reset to 9, whereas if we count from 0 to 9 it should reset to 0.

`timescale 1ns / 1ps

module FSM(

input [1:0] in,

input but,

input clk,

output [1:0] outasm,

output outsra,

output outsrb

);

reg state;

reg [1:0] next\_state;

reg [1:0] next\_state2;

always@(posedge clk)

if (but)

state = ~state;

else

state = state;

always@(state)

case (state)

1'b0://asm

begin

next\_state = in;

end

1'b1://rot or shift for bar shift

begin

next\_state2[0] = in[0];

next\_state2[1] = in[1];

end

endcase

assign outasm = next\_state;

assign outsra = next\_state2[0];

assign outsrb = next\_state2[1];

Endmodule

`timescale 1ns / 1ps

module FSM(

input [1:0] in,

input but,

//input clk,

output [1:0] outasm,

output outsra,

output outsrb

);

reg state;

reg [1:0] next\_state;

reg [1:0] next\_state2;

/\* always@(posedge clk)

begin

if (but)

state = ~state;

else

state = state;

end\*/

always@(but)

begin

case (but)

1'b0://asm

begin

next\_state = in;

end

1'b1://rot or shift for bar shift

begin

next\_state2[0] = in[0];

next\_state2[1] = in[1];

end

endcase

end

assign outasm = next\_state;

assign outsra = next\_state2[0];

assign outsrb = next\_state2[1];

endmodule

The fsm in this circuit passes either the add, subtract, multiply switches or the rotate and shift option for the barrel shifter will occur. The code is created so that when a button is pressed either one of these options will come to pass. This is to account for not enough I/O ports being available.

`timescale 1ns / 1ps

module BSUPDOWN(

input clk1,

output [1:0] s0,

output [1:0] s1

);

reg [3:0] tmp;

always@(posedge clk1)

begin

tmp <= tmp+1;

end

assign s0[0] = tmp[0];

assign s0[1] = tmp[2];

assign s1[0] = tmp[1];

assign s1[1] = tmp[3];

endmodule

The barrel shifter updown counter continuously counts up for a 4-bit number tmp and the digits are separated to equal different parts of s0 and s1 in the actual barrel shifter code to ensure randomization. The first and third digit of tmp go into the first barrel shifter as s0 and s1, the second and last digit go to the other barrel shifter for the same purpose.

`timescale 1ns / 1ps

module BARSHI(

input [3:0] B,

input [1:0] SW,

input sel,

output reg [3:0] P

);

reg [3:0] sr;

always@(\*)

begin

if(sel)

begin

case(SW) // rotate

2'b00:

begin

sr[3]=B[3];

sr[2]=B[2];

sr[1]=B[1];

sr[0]=B[0];

end

2'b01:

begin

sr[3]=B[0];

sr[2]=B[3];

sr[1]=B[2];

sr[0]=B[1];

end

2'b10:

begin

sr[3]=B[1];

sr[2]=B[0];

sr[1]=B[3];

sr[0]=B[2];

end

2'b11:

begin

sr[3]=B[2];

sr[2]=B[1];

sr[1]=B[0];

sr[0]=B[3];

end

endcase

end

else

begin//shift

case(SW)

2'b00:

begin

sr[3]=B[3];

sr[2]=B[2];

sr[1]=B[1];

sr[0]=B[0];

end

2'b01:

begin

sr[3]=0;

sr[2]=B[3];

sr[1]=B[2];

sr[0]=B[1];

end

2'b10:

begin

sr[3]=0;

sr[2]=0;

sr[1]=B[3];

sr[0]=B[2];

end

2'b11:

begin

sr[3]=0;

sr[2]=0;

sr[1]=0;

sr[0]=B[3];

end

endcase

end

P = sr;

end

endmodule

`timescale 1ns / 1ps

module BARSHI\_TB(

);

reg [3:0] Btb;

reg [1:0] SWtb;

reg seltb;

wire [3:0] Ptb;

BARSHI COMP(

.B(Btb),

.SW(SWtb),

.sel(seltb),

.P(Ptb)

);

initial

begin

Btb = 7;

SWtb = 2;

seltb = 1;

end

endmodule

For the barrel shifting code I created two separate cases to account for the option of shifting or rotating and the output is based on a two level 4 mux rows truth table. The 0 case represents rotation, while the 1 case represents shifting. The option to select is coming from the finite state machine in this case.

`timescale 1ns / 1ps

module ALU(

input [3:0] opa,

input [3:0] opb,

input [1:0] asm,

output [7:0] opc

);

reg [7:0] ropc=0;

always@(\*)

begin

case(asm)

2'b00:

begin

ropc <= 0;

end

2'b01://ADD

begin

ropc <=opa+opb;

end

2'b10://SUBTRACT

begin

ropc= opa-opb;

end

2'b11: //MULTIPICATION

begin

ropc = opa \* opb;

end

endcase

end

assign opc = ropc;

endmodule

`timescale 1ns / 1ps

module alu\_tb();

reg [3:0] opatb;

reg [3:0] opbtb;

reg [1:0] asmtb;

wire [7:0] opctb;

ALU alu\_tb(

.opa(opatb),

.opb(opbtb),

.asm(asmtb),

//output to 7seg

.opc(opctb)

);

initial

begin: TEST

asmtb = 3;

opatb = 4;

opbtb = 2;

#100

asmtb = 2;

opatb = 4;

opbtb = 4;

#100

asmtb = 1;

opatb = 9;

opbtb = 1;

#100

asmtb = 3;

opatb = 1;

opbtb = 9;

#1000

$finish;

end

endmodule

The alu code takes 2 single digit numbers and either adds, subtracts, or multiplies them to yield a 2 digit answer. This is a much simplified version because signs weren’t necessary here, but there are 4 cases which depended on the asm chooser from the finite state machine. In addition to the add, subtract, and multiply options, there is also a null option if 00 is chosen from the two switches that control the operations.

`timescale 1ns / 1ps

module segdisplaydriver(

input [31:0] inDigit,

output reg [6:0] Cnode,

output dp,

output reg [7:0] AN,

input nexysCLK, // 100MHz

output reg divided\_clk = 0 // 10kHz => 10ms period, 0.5ms ON, 0.5ms OFF

);

reg [3:0] singledigit = 0;

reg [3:0] refreshcounter = 0;

// Calculate division value = 100MHz / (2 \* desired frequency) - 1 => 10kHz => 4999

localparam div\_value = 25000;

integer counter\_value = 0;

always @(posedge nexysCLK)

begin

if (counter\_value == div\_value) // For every (div\_value) clock cycles, reset counter back to 0

counter\_value <= 0; // Use <= for parallel & same time, = for sequential - one after the other

else

counter\_value <= counter\_value + 1;

end

// divide clock

always @(posedge nexysCLK)

begin

if (counter\_value == div\_value)

divided\_clk <= ~divided\_clk; // Flip signal

else

divided\_clk <= divided\_clk; // Keep signal the same

end

/\*CLOCK DIVIDER CODE\*/

always @(posedge divided\_clk)

begin

refreshcounter <= refreshcounter + 1;

end

always @(refreshcounter)

begin

case(refreshcounter)

4'b0000: singledigit = inDigit[3:0]; // digit 1 value (right digit)

4'b0001: singledigit = inDigit[7:4]; // digit 2 value

4'b0010: singledigit = inDigit[11:8]; // digit 3 value

4'b0011: singledigit = inDigit[15:12]; // digit 4 value

4'b0101: singledigit = inDigit[19:16]; // digit 5 value missing 4’b0010

4'b0110: singledigit = inDigit[23:20]; // digit 6 value

4'b0111: singledigit = inDigit[27:24]; // digit 7 value

4'b1000: singledigit = inDigit[31:28]; // digit 8 value (left digit)

endcase

end

always @(refreshcounter)

begin

case(refreshcounter)

4'b0000: AN = 8'b11111110; // digit 1 ON (right digit)

4'b0001: AN = 8'b11111101; // digit 2 ON

4'b0010: AN = 8'b11111011; // digit 3 ON

4'b0011: AN = 8'b11110111; // digit 4 ON

4'b0100: AN = 8'b11101111; // digit 5 ON

4'b0101: AN = 8'b11011111; // digit 6 ON

4'b0110: AN = 8'b10111111; // digit 7 ON

4'b0111: AN = 8'b01111111; // digit 8 ON (left digit)

default: AN = 8'bZZZZZZZZ;

endcase

end

assign dp = 1'b1;

always@(singledigit)

begin

case (singledigit)

4'd0: Cnode<= 7'b0000001; //0

4'd1: Cnode<= 7'b1001111; //1

4'd2: Cnode<= 7'b0010010; //2

4'd3: Cnode<= 7'b0000110; //3

4'd4: Cnode<= 7'b1001100; //4

4'd5: Cnode<= 7'b0100100; //5

4'd6: Cnode<= 7'b0100000; //6

4'd7: Cnode<= 7'b0001111; //7

4'd8: Cnode<= 7'b0000000; //8

4'd9: Cnode<= 7'b0000100; //9

4'd10:Cnode<= 7'b0001000; //A

4'd11:Cnode<= 7'b1100000; //B

4'd12:Cnode<= 7'b0110001; //C

4'd13:Cnode<= 7'b1000010; //D

4'd14:Cnode<= 7'b0110000; //E

4'd15:Cnode<= 7'b0111000; //F

default: Cnode = 7'b0000000; //DEFAULT CASE EVERYTHING ON

endcase

end

endmodule

`timescale 1ns / 1ps

module SEGDRIVE\_TB(

);

reg [31:0] tmp\_SW\_tb;

wire [6:0] Cnode\_tb;

wire dp\_tb;

wire [7:0] AN\_tb;

SEGDRIVE COMP (

.tmp\_SW(tmp\_SW\_tb),

.Cnode(Cnode\_tb),

.dp(dp\_tb),

.AN(AN\_tb)

);

initial

begin

//dp\_tb = 1;

tmp\_SW\_tb=0;

#10

tmp\_SW\_tb=4;

#1000

$finish;

end

endmodule

This code takes an input of 32 switches and depending on when the switches change value, one of the many possible outputs will be selected to display on the 7 segment display. AN and dp are assigned in the beginning to make sure it is only one digit without the decimal place being shown. The different cases range from 0 to 9, and the 32-bit switch value selects the specific digit that needs to be modified. In this particular code 0 represents on, while 1 represents off.

`timescale 1ns / 1ps

module top(

//INPUTS FROM FPGA

input clkt,

input rstt,

input [3:0] asw,

input [3:0] bsw,

input [3:0] cont,

input aud,

input aload,

input bud,

input bload,

input [1:0] in,

input button,

//OUTPUT TO 7SEG DISPLAY

// output [7:0] opc,

//output signc1,

//7SEG DISPLAY INPUTS

output [6:0] Cnode1,

output [7:0] AN1,

output seg

);

wire [31:0] seg7Digit;

//Send input changes to 7seg display

assign seg7Digit[3:0] = asw;

assign seg7Digit[7:4] = bsw;

// assign seg7Digit[11:8] = tmp\_cnta;

//assign seg7Digit[19:16] = tmp\_cntb;

wire [7:0] opc;

wire slowclk\_out;

CLKMANAGER clockmanager\_wrapper(

.clk(clkt),

.SW(cont),

.clkout(slowclk\_out)

);

//wire [3:0] tmp\_cnta;

UPDOWN udcounter\_wrappera(

.rst1(rstt),

.load(aload),

.clk1(slowclk\_out),

.tmp(seg7Digit[11:8]),

.a(asw),

.ud(aud)

);

//wire [3:0] tmp\_cntb;

UPDOWN udcounter\_wrapperb(

.rst1(rstt),

.load(bload),

.clk1(slowclk\_out),

.tmp(seg7Digit[19:16]),

.a(bsw),

.ud(bud)

);

wire [1:0] sel1;

wire [1:0] sel2;

BSUPDOWN bsupdown\_wrapper(

.clk1(slowclk\_out),

.s0(sel1),

.s1(sel2)

);

wire [1:0] amscon;

FSM fsm\_wrapper(

.in(in),

.but(button),

.clk(slowclk\_out),

.outasm(amscon),

.outsra(sra),

.outsrb(srb)

);

wire sra;

wire srb;

BARSHI barshi\_wrappera(

.B(seg7Digit[11:8]),

.SW(sel1),

.sel(sra),

.P(seg7Digit[15:12])

);

BARSHI barshi\_wrapperb(

.B(seg7Digit[19:16]),

.SW(sel2),

.sel(srb),

.P(seg7Digit[23:20])

);

ALU alu\_wrapper(

.opa(seg7Digit[15:12]),

.opb(seg7Digit[23:20]),

.asm(asmcon),

.opc(seg7Digit[31:24])

);

SEGDRIVE seg7driver\_wrapper(

.nexysCLK(clkt),

.inDigit(seg7Digit),

.Cnode(Cnode1),

.dp(seg),

.AN(AN1)

);

endmodule

`timescale 1ns / 1ps

module top\_tb();

reg [3:0] opatb;

reg [3:0] opbtb;

reg audtb;

reg budtb;

reg altb;

reg bltb;

reg [1:0] asmtb;

reg but1;

reg clk\_tb;

wire [6:0] Cnode\_tb;

wire dp\_tb;

wire [7:0] AN\_tb;

top top\_t(

.clkt(clk\_tb),

.asw(opatb),

.bsw(opbtb),

.aud(audtb),

.bud(budtb),

.aload(altb),

.bload(bltb),

.in(asmtb),

.button(but1),

//output to 7seg

.Cnode1(Cnode\_tb),

.seg(dp\_tb),

.AN1(AN\_tb)

);

initial

begin: TEST

asmtb = 2;

opatb = 4;

opbtb = 2;

audtb = 1;

budtb = 1;

but1 = 1;

clk\_tb=0;

#100

asmtb = 1;

opatb = 3;

opbtb = 6;

audtb = 0;

budtb = 1;

but1 = 0;

altb=1;

bltb=0;

#100

asmtb = 2;

opatb = 5;

opbtb = 2;

audtb = 1;

budtb = 0;

but1 = 1;

#100

asmtb = 3;

opatb = 8;

opbtb = 9;

audtb = 1;

budtb = 1;

but1 = 0;

#1000

$finish;

end

always

begin

#5 clk\_tb = ~clk\_tb;

end

endmodule

The top file basically instantiates all the previous files and establishes connections to make the circuit display the desired results. So initially the output of the clock manager goes to the two updown counters that were instantiated, the finite state machine that was created, and the upcounter that connects to the s0 and s1 of the two barrel shifters that were created. This is to ensure a synchronous clock for the entire circuit. The output of the updowns connect to the barrel shifters as well as the 7-seg display, and the output of the barrel shifters connect to the two input digits of the alu as well as the 7-seg display. To take care of this a 32-bit wire is created and specific indexes can be addressed like seg7digit[23:20]. This allows one output to go two places through wiring. From the barrel shifter updown counter, the first and third digit of tmp go into the first barrel shifter as s0 and s1, the second and last digit go to the other barrel shifter for the same purpose. The fsm in this circuit passes either the add, subtract, multiply switches or the rotate and shift option for the barrel shifter will occur. The seg7Digit wire establishes most of the connections described, while also displaying the interconnections on the 7-seg display.

**Corner Cases/Error :**

***One corner case can be A-D can be displayed, to cope this either the default case in the case statements should be 9, or the user simply just shouldn’t input those values.***

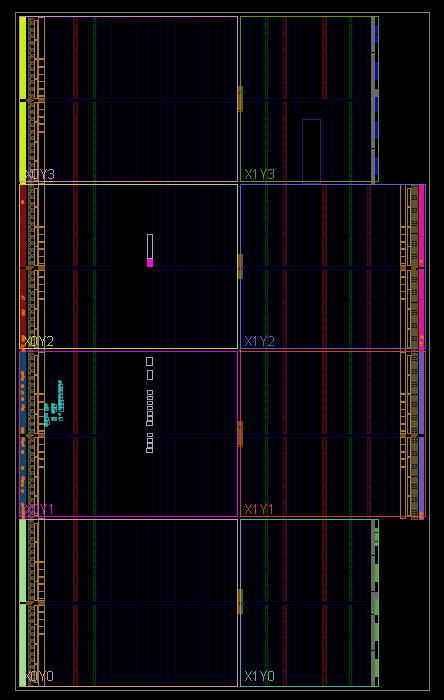
***Another corner case can be the CLKMANAGER Testbench. At first the testbench wasn’t working because of :***

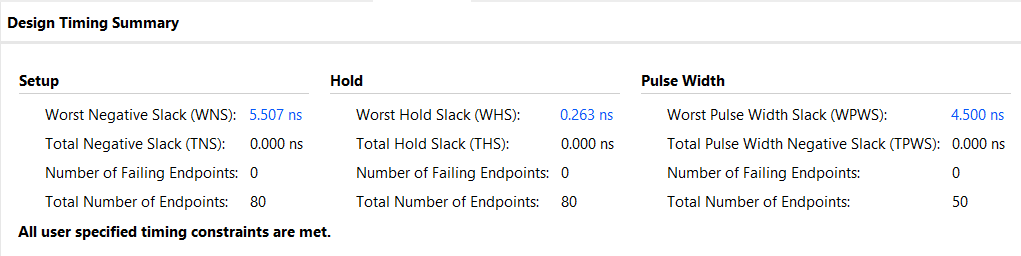
always@(SW)

begin

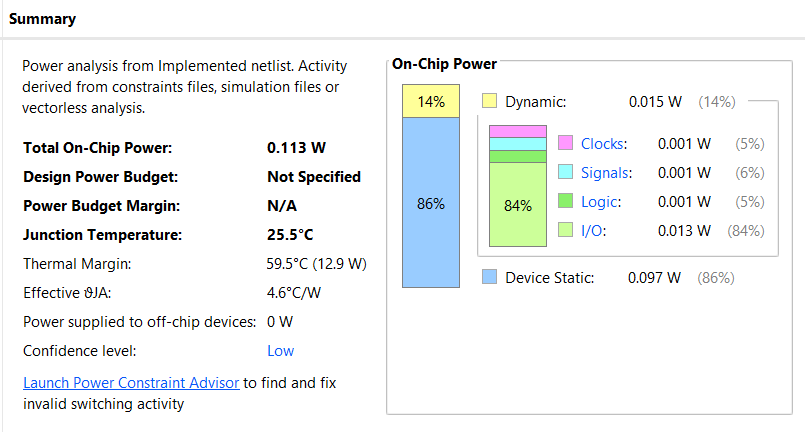
***The problem is that the loop can only run if the SW changes, so we had to change SW to posedge clk or posedge rst, so that the code could effectively work in the testbench.***

**IMPLEMENTED DESIGN/ TIMING SUMMARY:**

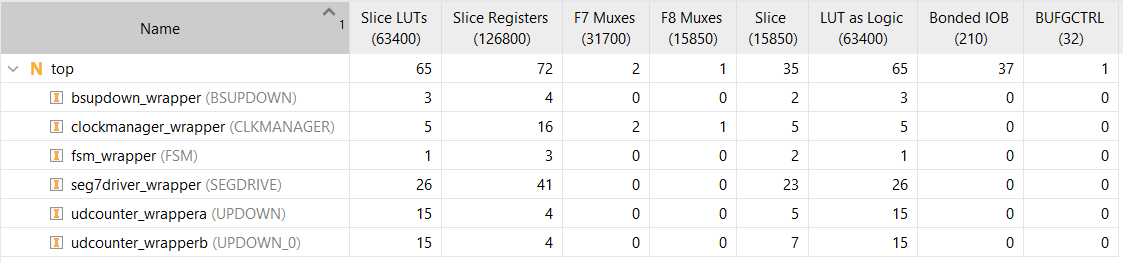
****

****

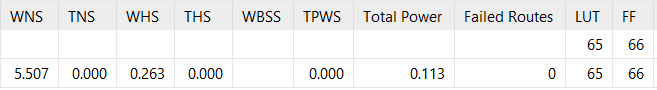
**POWER SUMMARY:**

****

**UTILIZATION:**

****

**RESOURCE USAGE:**

****