Digital Alarm

Ismaiel Sabet 900221277
Seif ElAnsari 900221511
Noor Emam 900222081

Table of Contents:

Abstract
As described in the project file, the goal of this project is to implement a Digital Alarm. The first step towards implementing this included reading up on the provided project description and understanding the basic functionality of the digital alarm. Following that, we implemented the Datapath (DP), Control Unit (CU), Algorithm State Machine (ASM), and finally the Logisim Evolution, which covered the second-minutes-hours counter and the seven segment display. Beyond that is implementation of the actual project in Verilog.

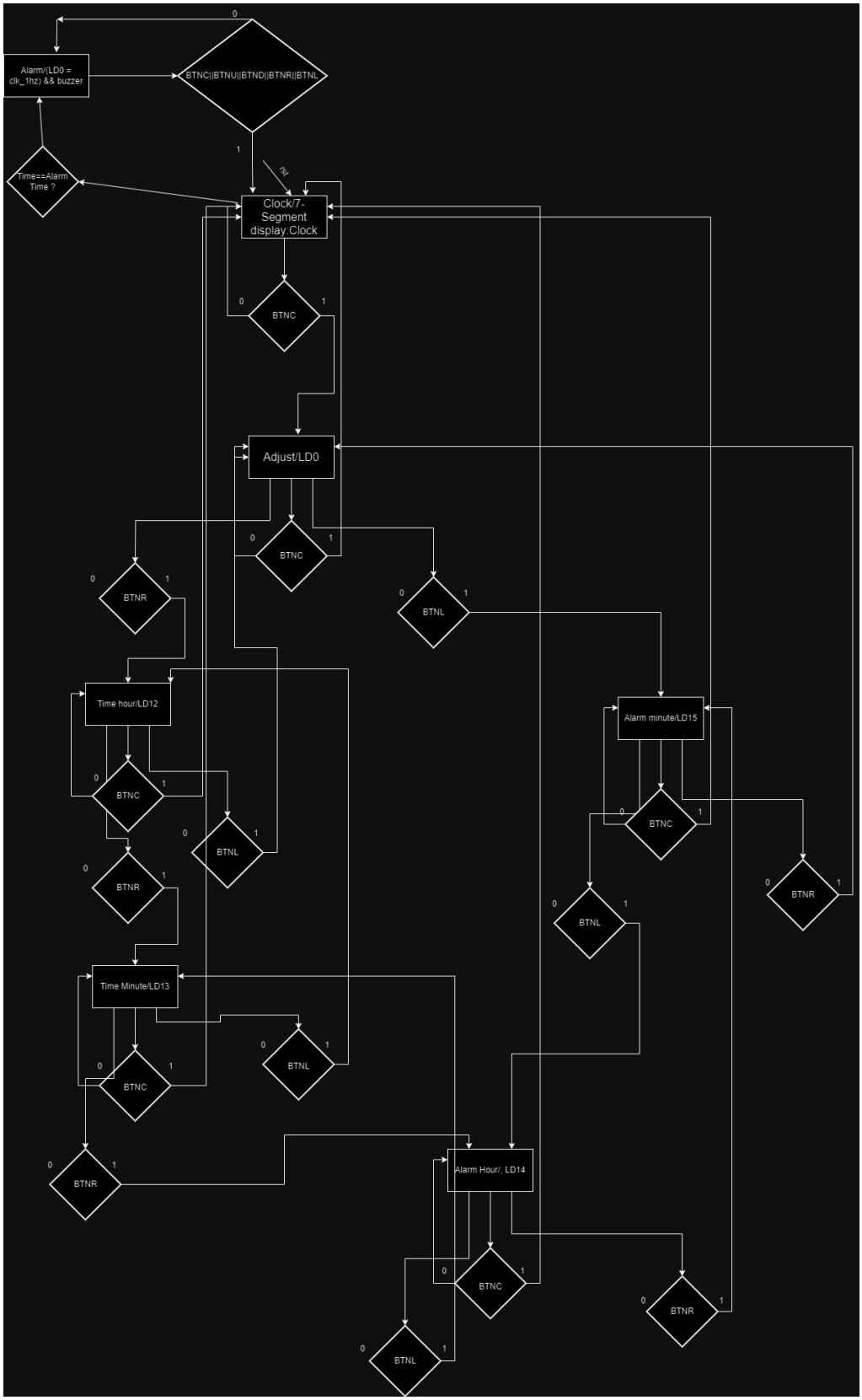
Design Process and Mindset
At the beginning of the project, we were all very excited to implement something that existed in the physical world. The first project was nice and creative, but it was more of a simulator. This one is a digital alarm, a device that each and every one of us uses everyday. So, there was definitely a desire to see how it works, and as such, an attempt to try and replicate such an important invention.

However and like on any project, our knowledge at the beginning was very limited and minute. We needed to research the logic behind building such a project, which was mostly localized in the DP, CU, and ASM charts. As such, each of us (the members of the group) took over exactly one of those 3 topics and researched them independently, in the hope that we could develop decent versions for the DP, CU, and ASM charts respectively. Further, we all shared our work together virtually and consistently reviewed everything together, as each of these 3 charts depend on one another.

When it came time to actually implement the project on Verilog, we opted for a completely different coding approach. In project 1, we did independent work and then shared our code together through Github. For this project, we decided to actually work in person, where 1 person would code and then we'd alternate through different tasks. This allowed all of us to be present while coding, but it also allowed each of us to be involved in every single step of the process of building this alarm. This approach does definitely have major cons, but its pros really outmatch the cons.

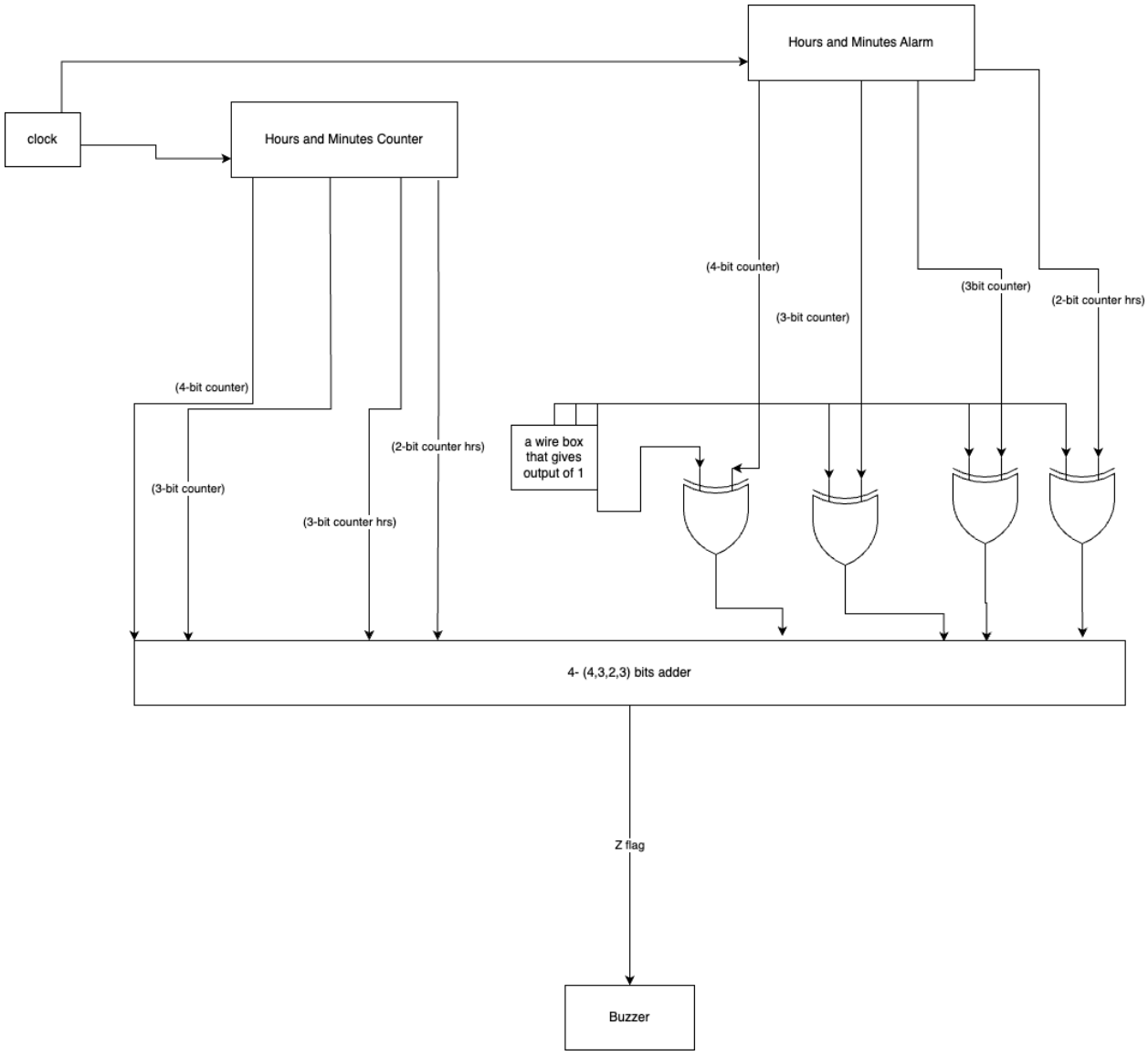
Design Files

ASM chart:


0
Alarm/(LD0 = clk_1hz) && buzzer
BTNC||BTNU||BTND||BTNR||BTNL
1
Time==Alarm Time ?
Clock/7-Segment display:Clock
BTNC
Adjust/LD0
BTNC
BTNR
BTNL
Time hour/LD12
Alarm minute/LD15
BTNC
BTNL
BTNR
BTNC
BTNL
BTNR
Time Minute/LD13
BTNL
BTNC
BTNR
Alarm Hour/, LD14
BTNC
BTNR
BTNL

The above figure details the ASM chart for our project. As one can tell, we implemented it using Moore rather than Mealy as we found it to be much simpler. There are 7 states: Clock, Adjust, Time hour, Time minutes, Alarm hour, Alarm minutes, and finally the actual Alarm, which is also a state in of itself. Our control signals are BTNC, BTNL, and BTNR, and through the decision blocks, we see the logical flow of the program from each state to the next. So for instance, clicking BTNC takes us from the default state i.e. Clock to the Adjust state. BTNR takes us from Time hours all the way to Alarm Minutes in that order, while BTNL does the reverse, and further, clicking BTNC at any time during any of these states always takes us back to the Clock.

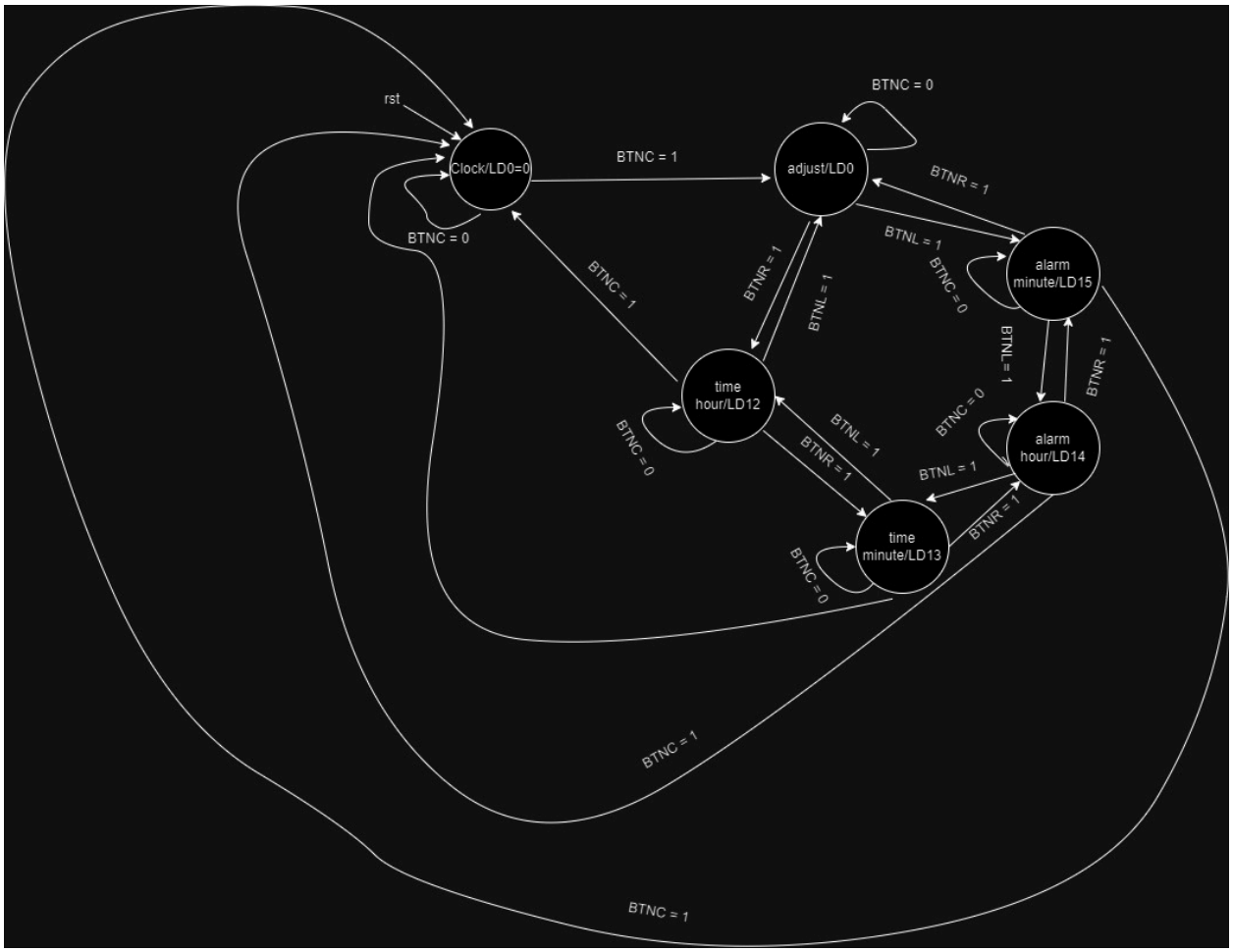
DP:



The above DP models the behavior of our project. We first feed a clock into the hours and minutes counter and the hours and minutes alarm. The hours and minutes

counter(through the use of a 2 bit counter (hours tens), a 3 bit counter (hours units), and a 4 bit counter and 3 bit counter) acts as our clock. The alarm also has the exact same counter signature, except that it's fed in an XOR gate with a 1. We know that the output of a 2 input XOR gate is 1 if and only if exactly one of the inputs is 1. We then add the output of both the hours and minute and the hours and minutes alarm and check for the z flag (if it's 1, then they're the same). We feed the z flag into the buzzer (it will emit sound if the z flag is 1 i.e. if the clock and the alarm are the same value).

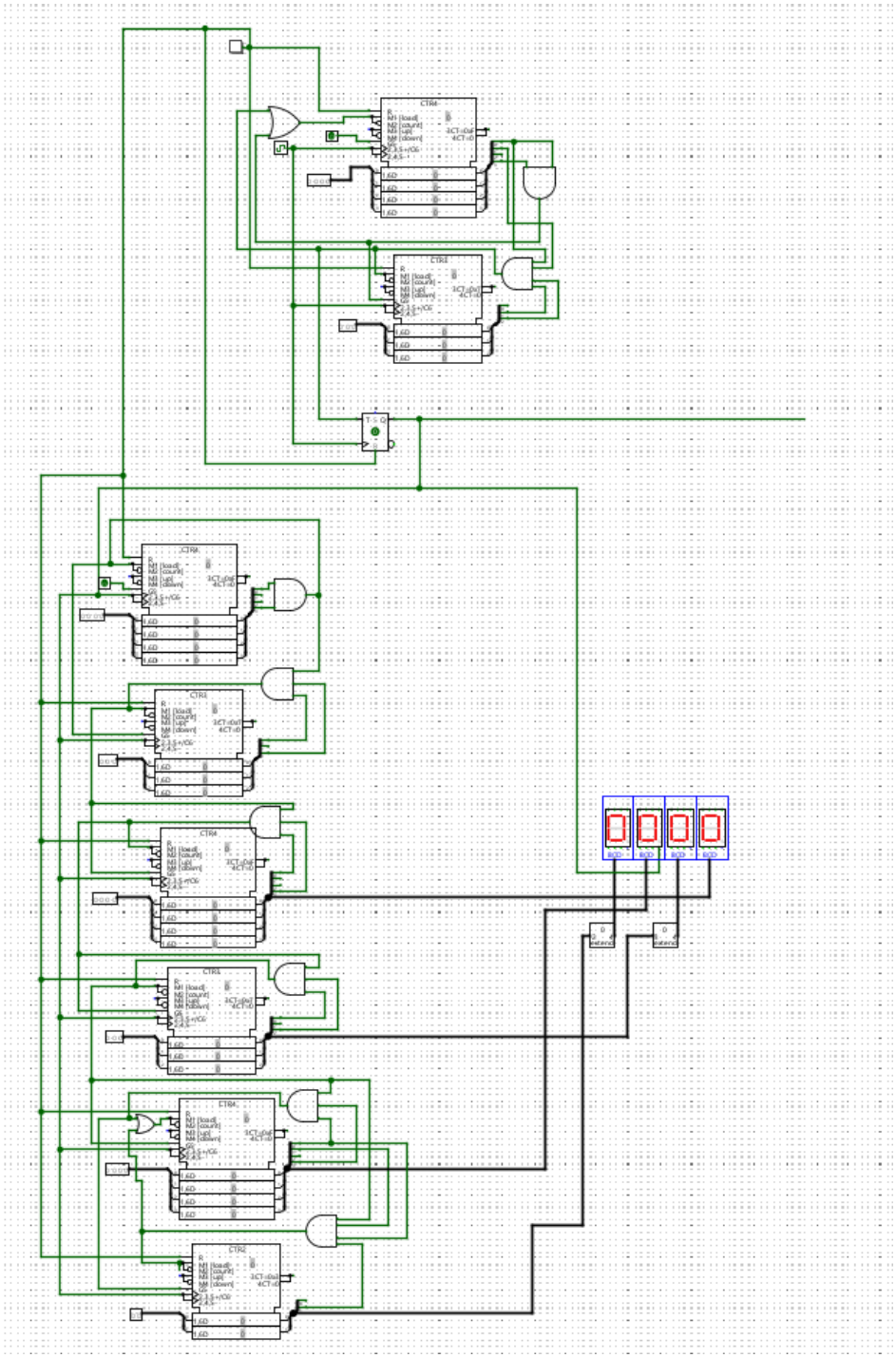
CU:



The above Control Unit (CU) describes the general states of the program and the control signals the cause any transition from one state to the next. We see that we always starts in Clock, and BTNC takes us from Clock to Adjust if it's 1. Following that step, any BTNC=1 always takes us back to Clock. BTNR goes through the sequence time-hour, time-minutes,alarm hour, alarm minutes (represented as the counterclockwise loop in the

figure), while BTNL goes in the reverse direction. Any click of BTNR or BTNL breaks the sequence and imposes the direction of the pressed button.

Logisim:



The above .circ file represents the basic functionality of the hours-minute-seconds counter, followed by the seven seg display. The actual counting mechanism (which is

very similar to the BCD counter taken in the course but expanded upon) was implemented in the course (as a bonus in the assignment), we just added the display. Note that this display is NOT multiplexed.

Verilog Files:

For the verilog files, we will begin by presenting the simpler modules before exploring the more complex module that is the Clock module, where all of the project's main functionality is rooted in,

Clock Divider:

```
module Clock_Divider #(parameter n = 50000000)(input clk, rst, output reg clk_out);

parameter WIDTH = $clog2(n);
reg [WIDTH-1:0] count;

always @ (posedge clk, posedge rst) begin
if(rst == 1'b1)
count  <= 0;
else if (count == n-1)
count <= 0;
else
count <= count +1;
end

always @ (posedge clk, posedge rst) begin

if(rst)
clk_out  <= 0;
else if (count == n-1)
clk_out <= ~clk_out;
end
endmodule
```

This function entails a simple clock divider and it's exactly similar to the one implemented in the lab.

Debouncer:

```
module Debouncer(input clk, rst, in, output out);

reg q1,q2,q3;
always@(posedge clk, posedge rst) begin
 if(rst == 1'b1) begin
 q1 <= 0;
 q2 <= 0;
 q3 <= 0;
 end
else begin
 q1 <= in;
 q2 <= q1;
 q3 <= q2;
 end
end
assign out = (rst) ? 0 : q1&q2&q3;
endmodule
```

This Debouncer module is the exact same as the one implemented in the lab.

Synchronizer:

```
module Synchronizer(input SIG, clk, output reg SIG1);
reg META;

always @ (posedge clk) begin
META <= SIG;
SIG1 <= META;
end
endmodule
```

The Synchronizer module is the exact same as the one implemented in the lab.

RisingEdge:

```
module RisingEdge (input clk, in, rst, output out);

reg [1:0] state;
reg [1:0] nextstate;
parameter [1:0] A=2'b00, B=2'b01, C=2'b10;

always @(posedge clk) begin
case(state)
A: if(in == 1) nextstate = B;
    else nextstate = A;
B:  if(in == 1) nextstate = C;
    else nextstate = A;
C:  if(in == 1) nextstate = C;
    else nextstate = A;
default: nextstate = A;
endcase
end

always @(posedge clk or posedge rst) begin
if(rst == 1)
state <= A;
else
state <= nextstate;
end

assign out = (state == B);

endmodule
```

The RisingEdge module is taken as is from the lab.

PushDownButton:

```
module PushDownButton(input button, clk, rst, output out);

wire med;
wire med2;
Debouncer Deb(.clk(clk),.rst(rst),.in(button),.out(med));
Synchronizer Sync(.SIG(med),.clk(clk),.SIG1(med2));
RisingEdge Edge(.clk(clk),.in(med2),.rst(rst),.out(out));
endmodule
```

PushDownButton is implemented in the project as in the lab as follows.

BCD7SEG:

```
module BCD7SEG# (parameter x =0)(input [3:0] number, clk,output reg [6:0] segments)

always @(posedge clk) begin
case(number)
0: segments = 7'b0000001;
1: segments = 7'b1001111;
2: segments = 7'b0010010;
3: segments = 7'b0000110;
4: segments = 7'b1001100;
5: segments = 7'b0100100;
6: segments = 7'b0100000;
7: segments = 7'b0001111;
8: segments = 7'b0000000;
9: segments = 7'b0000100;
default segments = 7'b1111111;
endcase
end
endmodule
```

This parametrized BCD7SEG module is the exact same as the one done in the lab.

Counter:

```
module Counter#(parameter x = 3, n = 5000000)(input clk, reset, enable, load, [x-1:0]data, output reg[x-1:0]count);

always @(posedge clk, posedge reset) begin
if(reset == 1) begin
count <= 0;
end
else if(load == 1) begin
count <= data;
end
else if(enable == 1) begin
if(count == n-1) begin
count <= 0;
end
else begin
count <= count +1;
end
end
end


endmodule
```

The idea of this parametrized Counter is similar to the Counter implemented in the lab. However, we added the load functionality and the counter functionality. As we can see,

when load is 1, we load in our data. We also added the “n-1” conditional resets the counter back to 0 (so for example a modulo 4 counter will load 0 when n-1 = 3). Else, we simply count.

Clock:

```
module Clock(input clk, reset, enable, Mode, Right, Left, Up, Down, output reg [6:0]segments, reg [3:0] anodes, reg dp, reg LD0, reg LD12, reg LD13, reg LD14, reg LD15, reg VDC);

wire clk_out;
wire clk_onehertz;
wire clk_segments;
wire clk_buttons;
wire [9:0] countmilliseconds;
wire [3:0] countseconds1;
wire [3:0] countseconds2;
wire [3:0] countminutes1;
wire [3:0] countminutes2;
wire [3:0] counthours1;
wire [3:0] counthours2;
wire [6:0] segments1; //minutes1 segment
wire [6:0] segments2; //minutes2 segment
wire [6:0] segments3; //hours1 segment
wire [6:0] segments4; //hours2 segment
wire [6:0] segments5; //hours1Adjust segment
wire [6:0] segments6; //hours2Adjust segment
wire [6:0] segments7; //minutes1Adjust segment
wire [6:0] segments8; //minutes2Adjust segment
wire [6:0] segments9; //hours1Alarm segment
wire [6:0] segments10; //hours2Alarm segment
wire [6:0] segments11; //minutes1Alarm segment
wire [6:0] segments12; //minutes2Alarm segment
wire ModeButton;
wire RightButton;
wire LeftButton;
wire UpButton;
wire DownButton;
reg [2:0] state;
reg [2:0] nextstate;
reg enableS1;
reg enableS2;
reg enableM1;
reg enableM2;
reg enableH1;
reg enableH2;
```

The above is the first picture of the Clock module. This is a large module, and it is there that most of the project’s functionality takes place. We have 17 parameters, ranging from inputs to outputs. We have the LD outputs and the buzzer output, but also the clock, reset, enable, mode, the 4 buttons, and the output segments (they will be used to to handle the seconds, minutes, and hours of both the time and the alarm). Most of the above code is simply just declarations and instantiations of vectors and variables that will be used in the upcoming screenshots.

```
reg anodeshifter;
reg [3:0] counthours1Adjust;
reg [3:0] counthours2Adjust;
reg [3:0] countminutes1Adjust;
reg [3:0] countminutes2Adjust;
reg [3:0] counthours1Alarm;
reg [3:0] counthours2Alarm;
reg [3:0] countminutes1Alarm;
reg [3:0] countminutes2Alarm;

parameter [2:0] Clock = 3'b000,  Adjust = 3'b001, time_hour = 3'b010, time_minute = 3'b011, alarm_hour = 3'b100, alarm_minute = 3'b101, VDC0 = 3'b110, VDC1 = 3'b111;

Clock_Divider#(50000000) One_Hertz(.clk(clk),.rst(reset),.clk_out(clk_onehertz)); //1Hz Clock
Clock_Divider#(50000) Frequency_Regulator(.clk(clk),.rst(reset),.clk_out(clk_out)); //1000Hz Clock
Clock_Divider#(150000) Segment_frequency(.clk(clk),.rst(reset),.clk_out(clk_segments)); //Segment Clock
Clock_Divider#(75000) Button_frequency(.clk(clk),.rst(reset),.clk_out(clk_buttons)); //Up and Down Clock
Counter#(10,1000) Milli(.clk(clk_out),.reset(reset),.count(countmilliseconds),.enable(enable),.load(1'b0),.data(10'b0000000000)); //Milliseconds Counter
Counter#(4,10) Sec1(.clk(clk_out),.reset(reset),.count(countseconds1),.enable(enableS1),.load(1'b0),.data(4'b0000)); //Seconds1 Counter
Counter#(4,6) Sec2(.clk(clk_out),.reset(reset),.count(countseconds2),.enable(enableS2),.load(1'b0),.data(4'b0000)); //Seconds2 Counter
Counter#(4,10) Min1(.clk(clk_out),.reset(reset),.count(countminutes1),.enable(enableM1),.load((state == time_minute)),.data((state== time_minute)? countminutes1Adjust : 4'b0000)); //Minutes1 Counter
Counter#(4,6) Min2(.clk(clk_out),.reset(reset),.count(countminutes2),.enable(enableM2),.load((state == time_minute)),.data((state== time_minute)? countminutes2Adjust : 4'b0000)); //Minutes2 Counter
Counter#(4,10) hr1(.clk(clk_out),.reset(reset),.count(counthours1),.enable(enableH1),.load((enableH1&counthours2[1]&counthours1[0]&counthours1[1]) ||(state== time_hour)),.data((state== time_hour)? counthours1Adjust : 4'b0000 )); //Hours1 Counter
Counter#(4,3) hr2(.clk(clk_out),.reset(reset),.count(counthours2),.enable(enableH2),.load((enableH1&counthours2[1]&counthours1[0]&counthours1[1]) ||(state == time_hour)),.data((state== time_hour)? counthours2Adjust : 4'b0000 )); //Hours2 Counter
BCD7SEG Seg1(.number(countminutes1),.segments(segments1),.clk(clk_segments)); //BCD7SEG for Minutes1
BCD7SEG Seg2(.number(countminutes2),.segments(segments2),.clk(clk_segments)); //BCD7SEG for Minutes2
BCD7SEG Seg3(.number(counthours1),.segments(segments3),.clk(clk_segments)); //BCD7SEG for Hours1
BCD7SEG Seg4(.number(counthours2),.segments(segments4),.clk(clk_segments)); //BCD7SEG for Hours2
BCD7SEG Seg5(.number(counthours1Adjust),.segments(segments5),.clk(clk_segments)); //BCD7SEG for Hours1Adjust
BCD7SEG Seg6(.number(counthours2Adjust),.segments(segments6),.clk(clk_segments)); //BCD7SEG for Hours2Adjust
BCD7SEG Seg7(.number(countminutes1Adjust),.segments(segments7),.clk(clk_segments)); //BCD7SEG for Minutes1Adjust
BCD7SEG Seg8(.number(countminutes2Adjust),.segments(segments8),.clk(clk_segments)); //BCD7SEG for Minutes2Adjust
BCD7SEG Seg9(.number(counthours1Alarm),.segments(segments9),.clk(clk_segments)); //BCD7SEG for Hours1Alarm
BCD7SEG Seg10(.number(counthours2Alarm),.segments(segments10),.clk(clk_segments)); //BCD7SEG for Hours2Alarm
BCD7SEG Seg11(.number(countminutes1Alarm),.segments(segments11),.clk(clk_segments)); //BCD7SEG for Minutes1Alarm
BCD7SEG Seg12(.number(countminutes2Alarm),.segments(segments12),.clk(clk_segments)); //BCD7SEG for Minutes2Alarm
PushDownButton ModeButton1(.button(Mode),.clk(clk_segments),.rst(reset),.out(ModeButton)); //PushDownButton for Mode
PushDownButton RightButton1(.button(Right),.clk(clk_segments),.rst(reset),.out(RightButton)); //PushDownButton for Right
PushDownButton LeftButton1(.button(Left),.clk(clk_segments),.rst(reset),.out(LeftButton)); //PushDownButton for Left
PushDownButton UpButton1(.button(Up),.clk(clk_buttons),.rst(reset),.out(UpButton)); //PushDownButton for Up
PushDownButton DownButton1(.button(Down),.clk(clk_buttons),.rst(reset),.out(DownButton)); //PushDownButton for Down

always@(posedge clk_out) begin
enableS1 = countmilliseconds[0] & countmilliseconds[1] & countmilliseconds[2] & countmilliseconds[5] & countmilliseconds[6] & countmilliseconds[7] & countmilliseconds[8] & countmilliseconds[9]; //Enable for Seconds1
enableS2 = enableS1 & countseconds1[0] & countseconds1[3]; //Enable for Seconds2
enableM1 = enableS2 & countseconds2[2] & countseconds2[0]; //Enable for Minutes1
enableM2 = enableM1&countminutes1[0] & countminutes1[3];  //Enable for Minutes2
enableH1 = enableM2& countminutes2[2]& countminutes2[0]; //Enable for Hours1
enableH2 = (enableH1 & counthours1[3] & counthours1[0]); //Enable for Hours2
end
```

Above is the second image of the Clock module. We have several declarations of regs at the beginning of the image (they will be used in upcoming always blocks) followed by the creation of our 8 states. The 8 states are the clock, adjust, time_hour, time_minute, alarm_hour, alarm_minute, and 2 states for the actual buzzer i.e. VDC0 and VDC1.

Our first clock divider counts at a frequency of 1 hz, so we got 50000000 by dividing f/2n = 1 and we solve for n for f = 100000000 (FPGA clock), which gives us n = 50000000.

The second clock divider because we want 1000 Hz to count the milliseconds (apply the same formula above).

The third clock divider handles the anode shifting, thus it's 150000.

The fourth clock divider handles the buttons, and as one can see, it's half the anode shifting divider.

We then have counters for the units and tens of the seconds, minutes,and hours, followed by creating a 7 seg display for every possible input vector we can have (so for example time minutes 1). After that, we make 4 buttons, one for each of the 4 directions.

```
always@(posedge clk_out) begin
enableS1 = countmilliseconds[0] & countmilliseconds[1] & countmilliseconds[2] & countmilliseconds[5] & countmilliseconds[6] & countmilliseconds[7] & countmilliseconds[8] & countmilliseconds[9]; //Enable for Seconds1
enableS2 = enableS1 & countseconds1[0] & countseconds1[3]; //Enable for Seconds2
enableM1 = enableS2 & countseconds2[2] & countseconds2[0]; //Enable for Minutes1
enableM2 = enableM1&countminutes1[0] & countminutes1[3];  //Enable for Minutes2
enableH1 = enableM2& countminutes2[2]& countminutes2[0]; //Enable for Hours1
enableH2 = (enableH1 & counthours1[3] & counthours1[0]); //Enable for Hours2
end

always@(posedge clk_segments) begin //Anode Shifter
if(anodes == 4'b0000) begin
anodes <= 4'b1111;
end
else if (anodes == 4'b1111) begin
anodes <= 4'b1110;
end
else if (anodes == 4'b1110) begin
anodes <= 4'b1101;
end
else if(anodes == 4'b1101) begin
anodes <= 4'b1011;
end
else if (anodes == 4'b1011) begin
anodes <= 4'b0111;
end
else if (anodes == 4'b0111) begin
anodes <= 4'b1110;
end
end
```

We have enables for each of the counters. As you can see they all depend on each other, so one will not count hours tens until we have both successfully counted the first and last bit of the hours units plus the minutes, which is in turn built on the seconds. This notion can be applied to all the other enable signals. After that, we initialize the anode based on what value they have.

```
always@(posedge clk_segments) begin //State Machine
    case(state)
    Clock: if(ModeButton==1) nextstate = Adjust;
            else if ((counthours1Alarm == counthours1) && (counthours2Alarm == counthours2) && (countminutes1Alarm == countminutes1) && (countminutes2Alarm == countminutes2)) nextstate = VDC0;
            else nextstate = Clock;
    Adjust: if(ModeButton==1) nextstate = Clock;
            else if(RightButton==1) nextstate = time_hour;
            else if(LeftButton==1) nextstate = alarm_minute;
            else nextstate = Adjust;
    time_hour: if(ModeButton==1) nextstate = Clock;
            else if(RightButton==1) nextstate = time_minute;
            else if(LeftButton==1) nextstate = Adjust;
            else nextstate = time_hour;
    time_minute: if(ModeButton==1) nextstate = Clock;
            else if(RightButton==1) nextstate = alarm_hour;
            else if(LeftButton==1) nextstate = time_hour;
            else nextstate = time_minute;
    alarm_hour: if(ModeButton==1) nextstate = Clock;
            else if(RightButton==1) nextstate = alarm_minute;
            else if(LeftButton==1) nextstate = time_minute;
            else nextstate = alarm_hour;
    alarm_minute: if(ModeButton==1) nextstate = Clock;
            else if(RightButton==1) nextstate = Adjust;
            else if(LeftButton==1) nextstate = alarm_hour;
            else nextstate = alarm_minute;
    VDC0: if((ModeButton==1)||(RightButton==1)||(LeftButton==1)||(UpButton==1)||(DownButton==1)) nextstate = VDC1;
            else nextstate = VDC0;
    VDC1: if(countminutes1Alarm != countminutes1) nextstate = Clock;
            else if(ModeButton==1) nextstate = Adjust;
            else nextstate = VDC1;
    default: nextstate = Clock ;
    endcase
end
```

Above is the third picture of the Clock module. This part of the state is our FSM, implemented as a Moore machine. As described above, we have 8 states and we just describe the state transition in the above part of the code. The default state is always the clock.

```
always@(posedge clk_segments) begin //State Machine Reset
    if(reset == 1) state <= Clock;
    else state <= nextstate;
end

always@(posedge clk_segments) begin //Heirarchy of States
if(state == Clock) begin //Clock State
    LD0 <=0;
    LD12 <=0;
    LD13 <=0;
    LD14 <=0;
    LD15 <=0;
    if(anodes == 14) begin
    segments <= segments1;
    dp <= 1;
    end
    else if (anodes ==13) begin
    segments <= segments2;
    dp <=1;
    end
    else if (anodes == 11) begin
    segments <= segments3;
    dp <= clk_onehertz;
    end
    else if(anodes == 7) begin
    segments <= segments4;
    dp <= 1;
    end
end
else if(state == Adjust) begin //Adjust State
    LD0 <= 1;
    LD12 <= 0;
    LD13 <= 0;
    LD14 <= 0;
    LD15 <= 0;
    if(anodes == 14)
    segments <= 7'b0001000;
    else if (anodes ==13)
    segments <= 7'b0001000;
    else if (anodes == 11)
    segments <= 7'b0001000;
    else if(anodes == 7)
    segments <= 7'b0001000;
end
```

The first part of the above code describes the reset functionality of the code. We also detail the states in much more in depth when it comes to the actual outputs.

```
else if(state == time_hour) begin //Time Hour State
    LD0 <= 1;
    LD12 <= 1;
    LD13 <= 0;
    LD14 <= 0;
    LD15 <= 0;
    if(anodes == 14)
    segments <= 7'b1111110;
    else if (anodes ==13)
    segments <= 7'b1111110;
    else if (anodes == 11)
    segments <= segments5;
    else if(anodes == 7)
    segments <= segments6;

    //Adjusting Hours and all edge cases

    if(UpButton == 1) begin
        if(counthours1 == 9 || (counthours1 == 3 && counthours2 == 2)) begin
            counthours1Adjust <= 0;
            if(counthours2 == 2) begin
                counthours2Adjust <= 0;
            end
            else begin
                counthours2Adjust <= counthours2 + 1;
            end
        end
        else begin
            counthours1Adjust <= counthours1 + 1;
            counthours2Adjust <= counthours2;
        end
    end
    else if(DownButton == 1) begin
        if(counthours1 == 0 && counthours2 ==0) begin
            counthours1Adjust <= 3;
            counthours2Adjust <= 2;
        end
       else if(counthours1 == 0) begin
            counthours1Adjust <= 9;
            counthours2Adjust <= counthours2 - 1;
        end
        else begin
            counthours1Adjust <= counthours1 - 1;
            counthours2Adjust <= counthours2;
        end
    end

end
```

The above code continues the trend carried on by the previous image. We see the actual counting done in the above picture, whether up or down (refer to the last line).

```
else if(state == time_minute) begin //Time Minute State
    LD0 <= 1;
    LD12 <= 0;
    LD13 <= 1;
    LD14 <= 0;
    LD15 <= 0;
    if(anodes == 14)
    segments <= segments7;
    else if (anodes ==13)
    segments <= segments8;
    else if (anodes == 11)
    segments <= 7'b1111110;
    else if(anodes == 7)
    segments <= 7'b1111110;

    //Adjusting Minutes and all edge cases

    if(UpButton == 1) begin
        if(countminutes1 == 9) begin
            countminutes1Adjust <= 0;
            if(countminutes2 == 5) begin
                countminutes2Adjust <= 0;
            end
            else begin
                countminutes2Adjust <= countminutes2 + 1;
            end
        end
        else begin
            countminutes1Adjust <= countminutes1 + 1;
            countminutes2Adjust <= countminutes2;
        end
    end
    else if(DownButton == 1) begin
        if(countminutes1 == 0 && countminutes2 ==0) begin
            countminutes1Adjust <= 9;
            countminutes2Adjust <= 5;
        end
       else if(countminutes1 == 0) begin
            countminutes1Adjust <= 9;
            countminutes2Adjust <= countminutes2 - 1;
        end
        else begin
            countminutes1Adjust <= countminutes1 - 1;
            countminutes2Adjust <= countminutes2;
        end
    end
end
```

Similar to the above picture (just detailing the other states).

```
else if(state == alarm_hour) begin //Alarm Hour State
    LD0 <= 1;
    LD12 <= 0;
    LD13 <= 0;
    LD14 <= 1;
    LD15 <= 0;
    if(anodes == 14)
    segments <= 7'b1111110;
    else if (anodes ==13)
    segments <= 7'b1111110;
    else if (anodes == 11)
    segments <= segments9;
    else if(anodes == 7)
    segments <= segments10;

    //Adjusting Alarm Hours and all edge cases

    if(UpButton == 1) begin
        if(counthours1Alarm == 9 || (counthours1Alarm == 3 && counthours2Alarm == 2)) begin
            counthours1Alarm <= 0;
            if(counthours2Alarm == 2) begin
                counthours2Alarm <= 0;
            end
            else begin
                counthours2Alarm <= counthours2Alarm + 1;
            end
        end
        else begin
            counthours1Alarm <= counthours1Alarm + 1;
            counthours2Alarm <= counthours2Alarm;
        end
    end
    else if(DownButton == 1) begin
        if(counthours1Alarm == 0 && counthours2Alarm ==0) begin
            counthours1Alarm <= 3;
            counthours2Alarm <= 2;
        end
       else if(counthours1Alarm == 0) begin
            counthours1Alarm <= 9;
            counthours2Alarm <= counthours2Alarm - 1;
        end
        else begin
            counthours1Alarm <= counthours1Alarm - 1;
            counthours2Alarm <= counthours2Alarm;
        end
    end
end
```

Similar to the above picture. As we can see, Count hours alarm is always loaded with 23 at the beginning.

```
else if(state == alarm_minute) begin //Alarm Minute State
    LD0 <= 1;
    LD12 <= 0;
    LD13 <= 0;
    LD14 <= 0;
    LD15 <= 1;
    if(anodes == 14)
    segments <= segments11;
    else if (anodes ==13)
    segments <= segments12;
    else if (anodes == 11)
    segments <= 7'b1111110;
    else if(anodes == 7)
    segments <= 7'b1111110;

    //Adjusting Alarm Minutes and all edge cases

    if(UpButton == 1) begin
        if(countminutes1Alarm == 9) begin
            countminutes1Alarm <= 0;
            if(countminutes2Alarm == 5) begin
                countminutes2Alarm <= 0;
            end
            else begin
                countminutes2Alarm <= countminutes2Alarm + 1;
            end
        end
        else begin
            countminutes1Alarm <= countminutes1Alarm + 1;
            countminutes2Alarm <= countminutes2Alarm;
        end
    end
    else if(DownButton == 1) begin
        if(countminutes1Alarm == 0 && countminutes2Alarm ==0) begin
            countminutes1Alarm <= 9;
            countminutes2Alarm <= 5;
        end
       else if(countminutes1Alarm == 0) begin
            countminutes1Alarm <= 9;
            countminutes2Alarm <= countminutes2Alarm - 1;
        end
        else begin
            countminutes1Alarm <= countminutes1Alarm - 1;
            countminutes2Alarm <= countminutes2Alarm;
        end
    end
end
```

Similar to the above picture. We also have load 59 for the minutes of the alarm at the beginning as shown above.

```
else if(state == VDC0) begin //Alarm State
    LD0 <= clk_out;
    LD12 <= 0;
    LD13 <= 0;
    LD14 <= 0;
    LD15 <= 0;
    if(anodes == 14)
    segments <= segments1;
    else if (anodes ==13)
    segments <= segments2;
    else if (anodes == 11)
    segments <= segments3;
    else if(anodes == 7)
    segments <= segments4;
    VDC <= clk_onehertz;

end
else if(state == VDC1) begin //Pseudo Clock State for Alarm
    LD0 <= 0;
    LD12 <= 0;
    LD13 <= 0;
    LD14 <= 0;
    LD15 <= 0;
    if(anodes == 14)
    segments <= segments1;
    else if (anodes ==13)
    segments <= segments2;
    else if (anodes == 11)
    segments <= segments3;
    else if(anodes == 7)
    segments <= segments4;
    VDC <= 0;
end
end

endmodule
```

Similar to the above picture. As this is the final picture in the module, we are just implementing the buzzer state in the above picture.

Constraint:

```
##7 Segment Display

set_property -dict { PACKAGE_PIN W7   IOSTANDARD LVCMOS33 } [get_ports {segments[6]}]
set_property -dict { PACKAGE_PIN W6   IOSTANDARD LVCMOS33 } [get_ports {segments[5]}]
set_property -dict { PACKAGE_PIN U8   IOSTANDARD LVCMOS33 } [get_ports {segments[4]}]
set_property -dict { PACKAGE_PIN V8   IOSTANDARD LVCMOS33 } [get_ports {segments[3]}]
set_property -dict { PACKAGE_PIN U5   IOSTANDARD LVCMOS33 } [get_ports {segments[2]}]
set_property -dict { PACKAGE_PIN V5   IOSTANDARD LVCMOS33 } [get_ports {segments[1]}]
set_property -dict { PACKAGE_PIN U7   IOSTANDARD LVCMOS33 } [get_ports {segments[0]}]
set_property -dict { PACKAGE_PIN V7   IOSTANDARD LVCMOS33 } [get_ports dp]

##Clock

set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports clk]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]

##Switches

set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports {reset}]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports {enable}]

##Anodes

set_property -dict { PACKAGE_PIN U2   IOSTANDARD LVCMOS33 } [get_ports {anodes[1]}]
set_property -dict { PACKAGE_PIN U4   IOSTANDARD LVCMOS33 } [get_ports {anodes[2]}]
set_property -dict { PACKAGE_PIN V4   IOSTANDARD LVCMOS33 } [get_ports {anodes[3]}]
set_property -dict { PACKAGE_PIN W4   IOSTANDARD LVCMOS33 } [get_ports {anodes[0]}]

##Buttons

set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports Mode]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports Up]
set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports Left]
set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports Right]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports Down]

## LEDs

set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {LD0}]
set_property -dict { PACKAGE_PIN P3    IOSTANDARD LVCMOS33 } [get_ports {LD12}]
set_property -dict { PACKAGE_PIN N3    IOSTANDARD LVCMOS33 } [get_ports {LD13}]
set_property -dict { PACKAGE_PIN P1    IOSTANDARD LVCMOS33 } [get_ports {LD14}]
set_property -dict { PACKAGE_PIN L1    IOSTANDARD LVCMOS33 } [get_ports {LD15}]

##Pmod Header JB
set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVCMOS33 } [get_ports {VDC}];#Sch name = JB2
```

The picture above details the constraint file of the project.

Individual Contribution:

- Ismaiel Sabet
    - Designed the ASM chart
    - Assisted in the CU
    - Wrote the report
    - Developed the clock divider
    - Developed the clock and counter modules
- Seif ElAnsari
    - Designed the CU
    - Assisted in the ASM chart
    - Made the Logisim simulation
    - Developed the PushButtonDetector
    - Developed the clock and counter modules
    - Developed the buzzer sound feature
- Noor Emam
    - Designed the DP
    - Developed the clock and counter
    - Developed the synchronizer
    - Developed the rising edge detector
    - Developed the buzzer sound feature

Conclusion:

To wrap this up, we successfully implemented a Digital Alarm for this project. We went through the different stages of the project development cycle, and as such, built each successive attempt at work on the previous stage.

However, the project was not free of problems. We struggled with the DP and the logisim at the beginning, and throughout the actual implementation of the verilog code, we suffered many problems including but not limited to a very fast clock, a very slow clock, lack of transition between states, problem with data input, and problems with the buzzer.

But nonetheless, it was a fresh and exciting experience, one that was stimulating and insightful!

References:

Dr Shalan's slides