# LABORATORY 9: ALARM CLOCK DESIGN

ELEC 3500

DUE 2019/12/07

AMIR KALMONI

100987101

JONATHAN PLANGGER

101070423

CARLETON UNIVERSITY

## TABLE OF CONTENTS

## 1. INTRODUCTION

This report was written as per the requirements of the ELEC 3500 course. The report covers the design of an alarm clock with the use of the software language Verilog and the Nexys 4 DDR FPGA. The sections of the design process for the clock presented in this report are the requirements for the design, the high-level design (block diagram) of the alarm clock, the low-level module design, the design of the innovations, and the conclusion of the results.

## 2. HIGH LEVEL DESIGN

The high-level design block diagram of the alarm clock is show below in Figure 2.0.

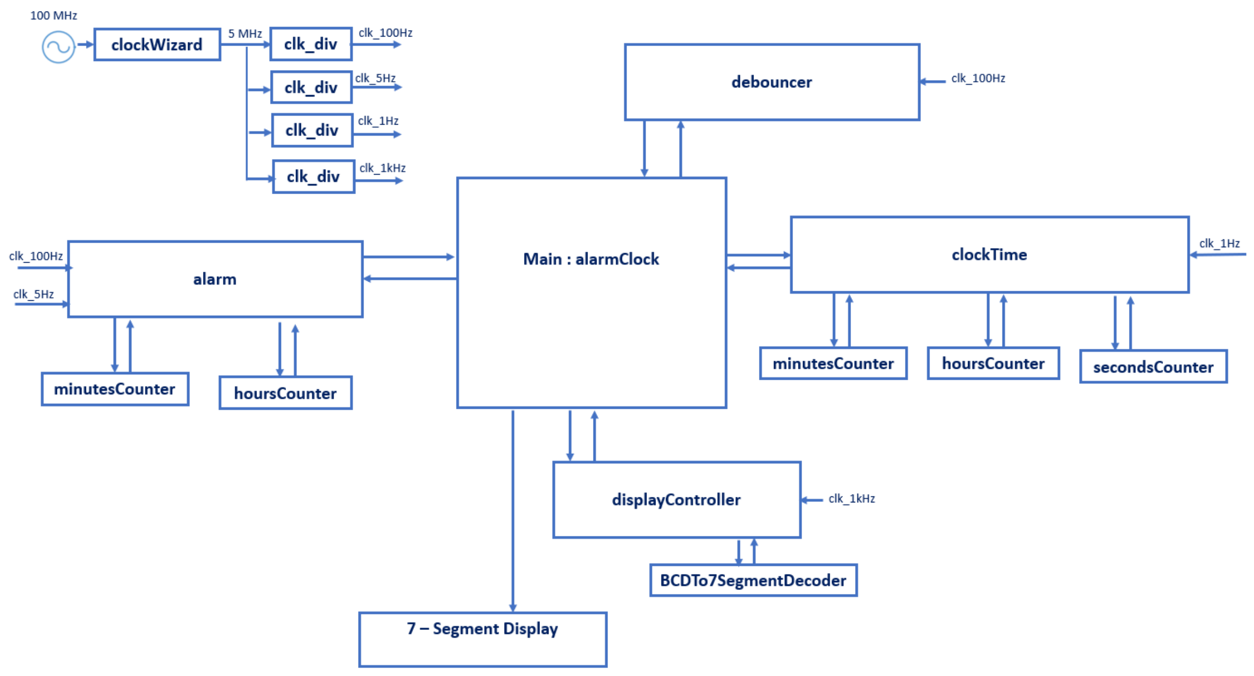


Figure 2.0: High Level Block Diagram

As can be seen in the figure above, the alarmClock module is the main module that takes in all inputs as well as sends out all the outputs of the alarm clock. The other sub modules will be described more thoroughly in section 3.0 but are in a general sense:

- clockTime: controls the internal clock of the alarm clock
- minutesCounter: counts the minutes
- hoursCounter: counts the hours
- secondsCounter: counts the seconds
- alarm: controls the value of the alarm and monitors if the clock time reaches the alarm time
- debouncer: debounces the button inputs to the system
- displayController: controls the output onto the 7-segment display
- BCDTo7SegmentDecoder: decodes the numerical values given to it by the displayController and returns display values to be used by the displayController
- clockWizard: IP that provides a 5MHz input clock

- clk_div: clock divider that reduces the frequency of the input clock by a parameterized amount

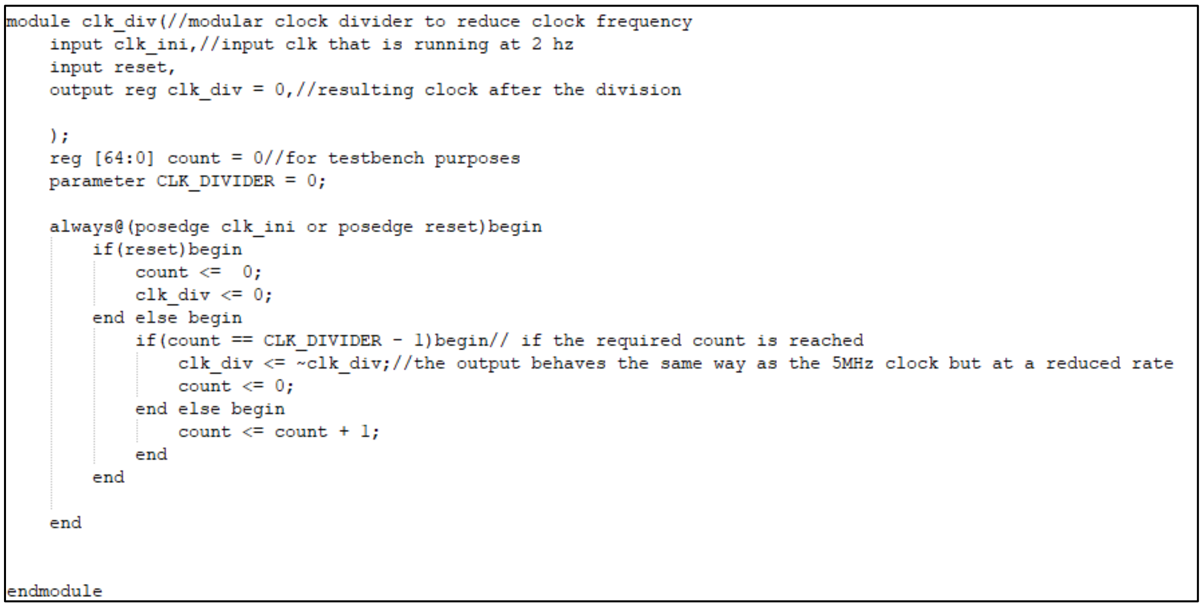
The alarm clock was therefore designed to have an internal clock that is internally monitored. When the alarm value is reached, the alarm notifies the master controller and the alarm triggers. With the display modules programmed, the main (alarmClock) module can change the output on the 7-segment display to display the internal values of the clock. With the general idea of the structure established, the details of the low-level modules can now be explained.

## 3. Low Level Module Design

This section was included in the report to detail the various modules alongside their respective codes and explain their function in the grand scheme of the design. The section will first cover the low-level modules used in the design and then progress to the higher level main file of alarmClock.

### 3.1 Clock Divider

This module was generated in order to reduce the initial value of the 5MHz clock into smaller such as clk_100Hz, clk_1Hz, clk_5Hz, clk_1kHz, to be used in other modules. The code used for the clock divider is shown in Figure 3.1 below.



```
module clk_div(//modular clock divider to reduce clock frequency
    input clk_ini,//input clk that is running at 2 hz
    input reset,
    output reg clk_div = 0,//resulting clock after the division

    );
    reg [64:0] count = 0//for testbench purposes
    parameter CLK_DIVIDER = 0;

    always@(posedge clk_ini or posedge reset)begin
        if(reset)begin
            count <=  0;
            clk_div <= 0;
        end else begin
            if(count == CLK_DIVIDER - 1)begin// if the required count is reached
                clk_div <= ~clk_div;//the output behaves the same way as the 5MHz clock but at a reduced rate
                count <= 0;
            end else begin
                count <= count + 1;
            end
        end

    end


endmodule
```

Figure 3.1: Clock Divider Code

The reduction of the inputted clock is done by first setting a counter value (count) and a value to be counted to (CLK_DIVIDER). Note that CLK_DIVIDER was initialized as a parameter of value of 0. This was done in order to allow the main file to overwrite the parameter value of this module by using the “defparam” statement therefore enabling the re-use of the exact same module for other desired output frequencies.

With the values set for the clock divider using defparam, the clock divider would then begin counting at each clock edge of the input clock (clk_ini) until it reached the desired value of CLK_DIVIDER. Once this is achieved, the output clock (clk_div) would then invert itself resulting in a time-varying signal similar to the input clock but with the desired frequency. This means that the module, as a whole, counts to the desired amount of Hz and, upon reaching it, outputs a clock signal which has the desired frequency.

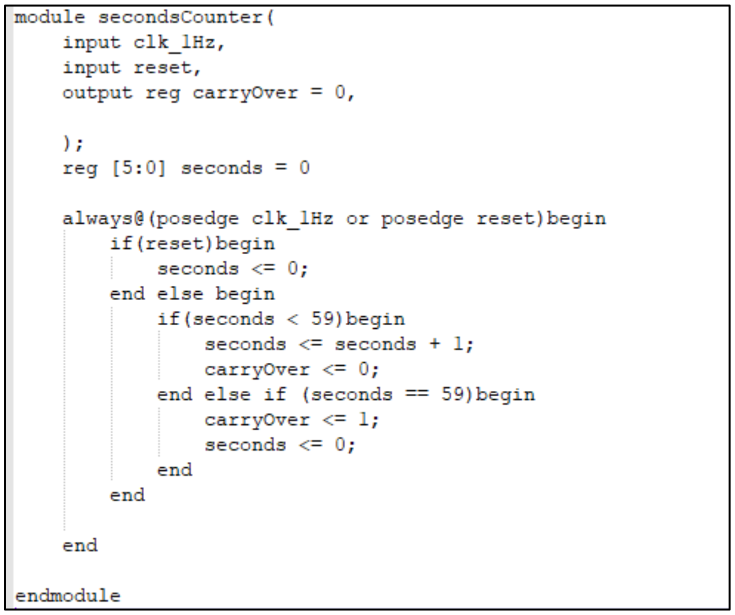
The CLK_DIVIDER value to be implemented in the main file using "defparam" is found using the following formula:

$$\text{CLK\_DIVIDER} = \frac{Input\ Clock\ Frequency}{Desired\ Clock\ Frequency}$$

Where, for the case of this design, the input clock frequency was always the one obtained by the clock wizard which was 5 MHz.

## 3.2 Seconds Counter

With the clock divider used to create a 1 Hz clock, the internal clock of the alarm itself could now be created. The internal clock was partitioned into 3 different modules, the first of which is the module that counts the seconds. The code for the seconds counter is shown in Figure 3.2 below.



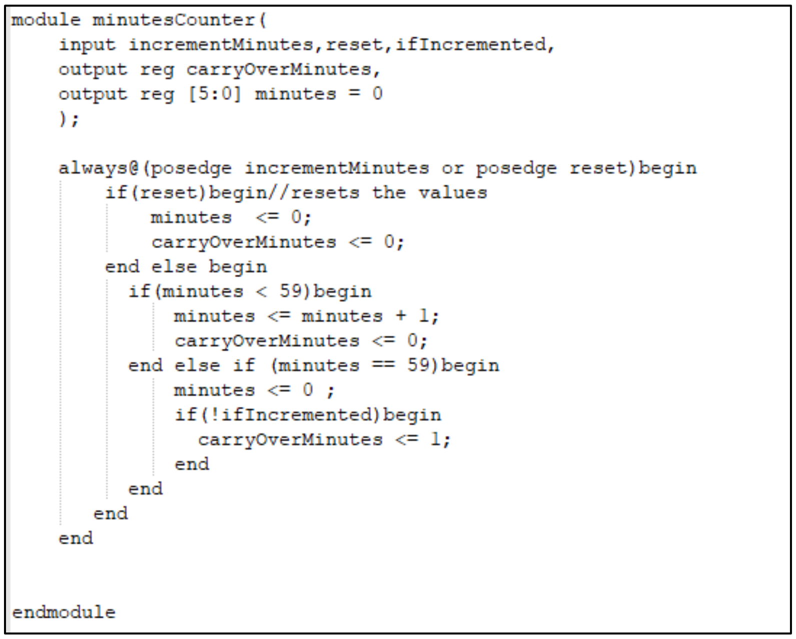
```
module secondsCounter(
    input clk_1Hz,
    input reset,
    output reg carryOver = 0,

    );
    reg [5:0] seconds = 0

    always@(posedge clk_1Hz or posedge reset)begin
        if(reset)begin
            seconds <= 0;
        end else begin
            if(seconds < 59)begin
                seconds <= seconds + 1;
                carryOver <= 0;
            end else if (seconds == 59)begin
                carryOver <= 1;
                seconds <= 0;
            end
        end

    end

endmodule
```

Figure 3.2: Seconds Counter

This module had the task of counting each second that occurred by incrementing the value of count each time the input clock became active high. Upon reaching the value of 59 seconds, count would be reset to 0 and the a carryOver signal would be output which would signal to the minutesCounter that the minutes should be incremented. The idea of this module is therefore to count every Hz, or rather, every second (due to $t = \frac{1}{f} = \frac{1}{1Hz} = 1s$) that occurs, and upon reaching the count of 60 (59 is due to starting the count at the value of 0), the seconds restart at 0 and the minutesCounter is alerted of this change. The connection between the secondsCounter and the minutesCounter is done in the higher module of clockTime.

## 3.3 Minutes Counter

The next step in creating an internal clock was in designing a module that would handle the minutes of the clock. This means that the module would have to be able to both take in the carryOver values of the secondsCounter as well as the button inputs used to change the time, the hours, and the minutes. The code used for this module is shown in Figure 3.3 below.



```
module minutesCounter(
    input incrementMinutes,reset,ifIncremented,
    output reg carryOverMinutes,
    output reg [5:0] minutes = 0
    );

    always@(posedge incrementMinutes or posedge reset)begin
        if(reset)begin//resets the values
            minutes  <= 0;
            carryOverMinutes <= 0;
        end else begin
          if(minutes < 59)begin
              minutes <= minutes + 1;
              carryOverMinutes <= 0;
          end else if (minutes == 59)begin
              minutes <= 0 ;
              if(!ifIncremented)begin
                carryOverMinutes <= 1;
              end
          end
       end
    end


endmodule
```

Figure 3.3: MinutesCounter Module Code

This module functioned in a semi-asynchronous manner where even though there was no clock as input, the module still worked off of the carryOver value obtained in the secondsCounter module which is a synchronous output. In addition, in order for the module to allow the incrementation of the values by buttons, the input incrementMinutes and ifIncremented were used for the circuit. The logic behind the incrementMinutes is not shown here but is rather shown in the higher level module of clockTime and is shown in the following relation:

incrementMinutes = (changeTime && changeMinutes) || carrySeconds

Where changeTime is button input to signals desire to change the value of the time, changeMinutes is the debounced (shown later) button input to change the minutes, and the carryMinutes is the output of the secondsCounter to signal a rollover in the seconds. This means that this module activates either when the required buttons are pressed or when the secondsCounter has counted 60 seconds. This way, the minutes can be incremented by both the buttons and the internal clock and no errors are created due to trying to drive reg values in 2 always blocks.

Once active, the module proceeds to continue to count the minutes values until it reaches 60 minutes and rolls resetting the minutes and setting out the carry signal (carryOverMinutes) to the hoursCounter module.

One important thing to note is the ifIncremented input being required in the design due to how, without its presence, the button inputs would cause both the minutes to roll over and the hours to increase by 1 which isn't good in a clock where the values can only be incremented upwards. If the button change in the minutes incremented the hours, the user may overshoot the desired hour and have to tap the changeHours button another 23 times to get it back to the desired hour. The logic behind the ifIncremented input is simply if both the changeTime and changeHours buttons are pressed at the same time. This is shown in the relation below:

ifIncremented = changeTime && changeHours

This input was therefore used to choose whether the minutesCounter should output a value of 1. The minutesCounter was programmed to not output a carry when the ifIncremented value was high.

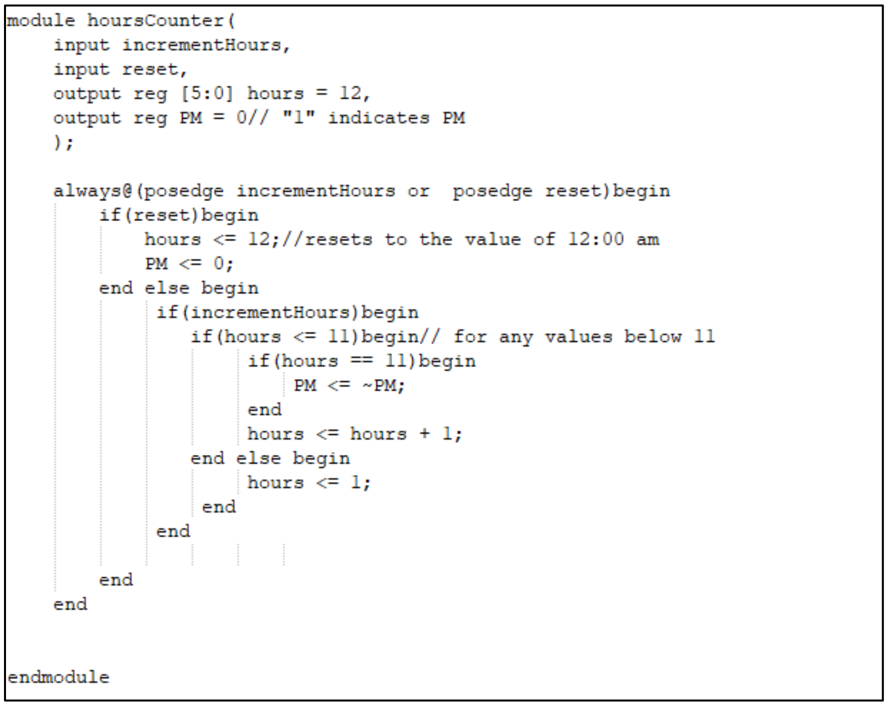
## 3.4 Hours Counter

The next module needed for the internal clock was the hoursCounter. This module has a similar structure as the minutesCounter where it receives the carry from the minutes as well as the button inputs to know when to increment the values of the hours. The logic for the incrementHours input is as following:

incrementHours = (changeTime && changeHours) || carryMinutes

Where the changeHours signal is the debounced signal of the hours button (hoursBTN) and the carryMinutes signal is the carry value from the previous minutesCounter module.

The code used for this module is shown in Figure 3.4 below.



```
module hoursCounter(
    input incrementHours,
    input reset,
    output reg [5:0] hours = 12,
    output reg PM = 0// "1" indicates PM
    );

    always@(posedge incrementHours or  posedge reset)begin
        if(reset)begin
            hours <= 12;//resets to the value of 12:00 am
            PM <= 0;
        end else begin
            if(incrementHours)begin
               if(hours <= 11)begin// for any values below 11
                    if(hours == 11)begin
                        PM <= ~PM;
                    end
                    hours <= hours + 1;
               end else begin
                    hours <= 1;
                end
            end

        end
    end


endmodule
```

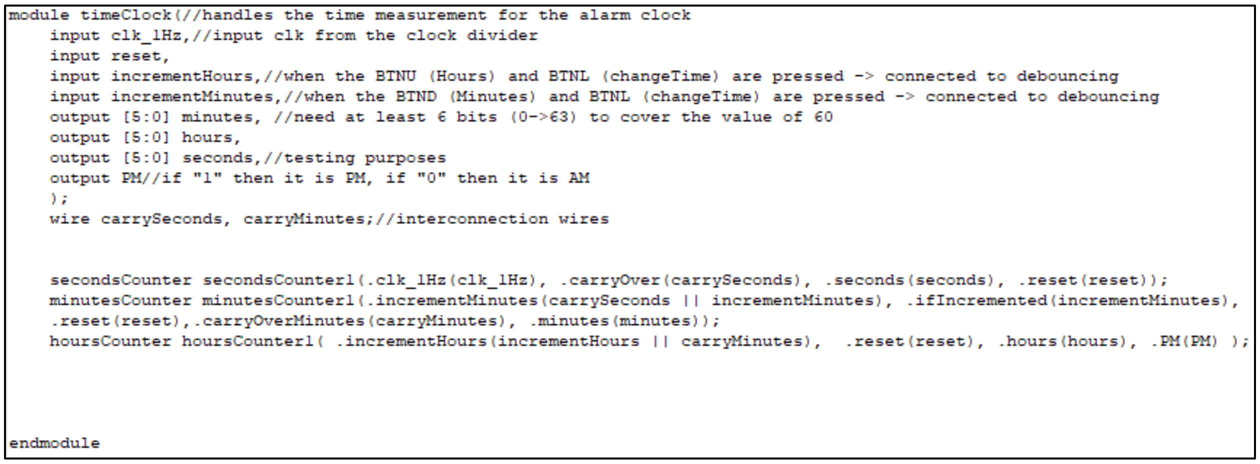
Figure 3.4: hoursCounter Module Code

Although the logic for the trigger conditions for this module is very similar to that of the minutesCounter, the modification of the hours was different due to the module needing to handle hours rather than minutes in addition to needing to handle whether the time was in PM or AM (due to the clock being a 12 hour clock). For this to be achieved, the hoursCounter was set up to count to the value of 12 where, upon reaching the value of 12, the value of the PM output bit is alternated (if it was 0 then it would become 1 and vice versa). This is seen in the if statement for the hours being equal to 11 where the PM is alternated at the same time as the hour being incremented to 12 which results in the hour turning to 12 at the same time as the PM value being changed due to non-blocking assignments.

The PM value, as the name suggests, indicates if the hours are in PM (if PM == 1) or in AM (if PM == 0).

## 3.5 TIME CLOCK

With the individual sub-modules described, the higher-level module of timeClock can now be fully described. The code used for this module is shown in Figure 3.5 below



```
module timeClock(//handles the time measurement for the alarm clock
    input clk_1Hz,//input clk from the clock divider
    input reset,
    input incrementHours,//when the BTNU (Hours) and BTNL (changeTime) are pressed -> connected to debouncing
    input incrementMinutes,//when the BTND (Minutes) and BTNL (changeTime) are pressed -> connected to debouncing
    output [5:0] minutes, //need at least 6 bits (0->63) to cover the value of 60
    output [5:0] hours,
    output [5:0] seconds,//testing purposes
    output PM//if "1" then it is PM, if "0" then it is AM
    );
    wire carrySeconds, carryMinutes;//interconnection wires


    secondsCounter secondsCounter1(.clk_1Hz(clk_1Hz), .carryOver(carrySeconds), .seconds(seconds), .reset(reset));
    minutesCounter minutesCounter1(.incrementMinutes(carrySeconds || incrementMinutes), .ifIncremented(incrementMinutes),
    .reset(reset),.carryOverMinutes(carryMinutes), .minutes(minutes));
    hoursCounter hoursCounter1( .incrementHours(incrementHours || carryMinutes),  .reset(reset), .hours(hours), .PM(PM) );



endmodule
```

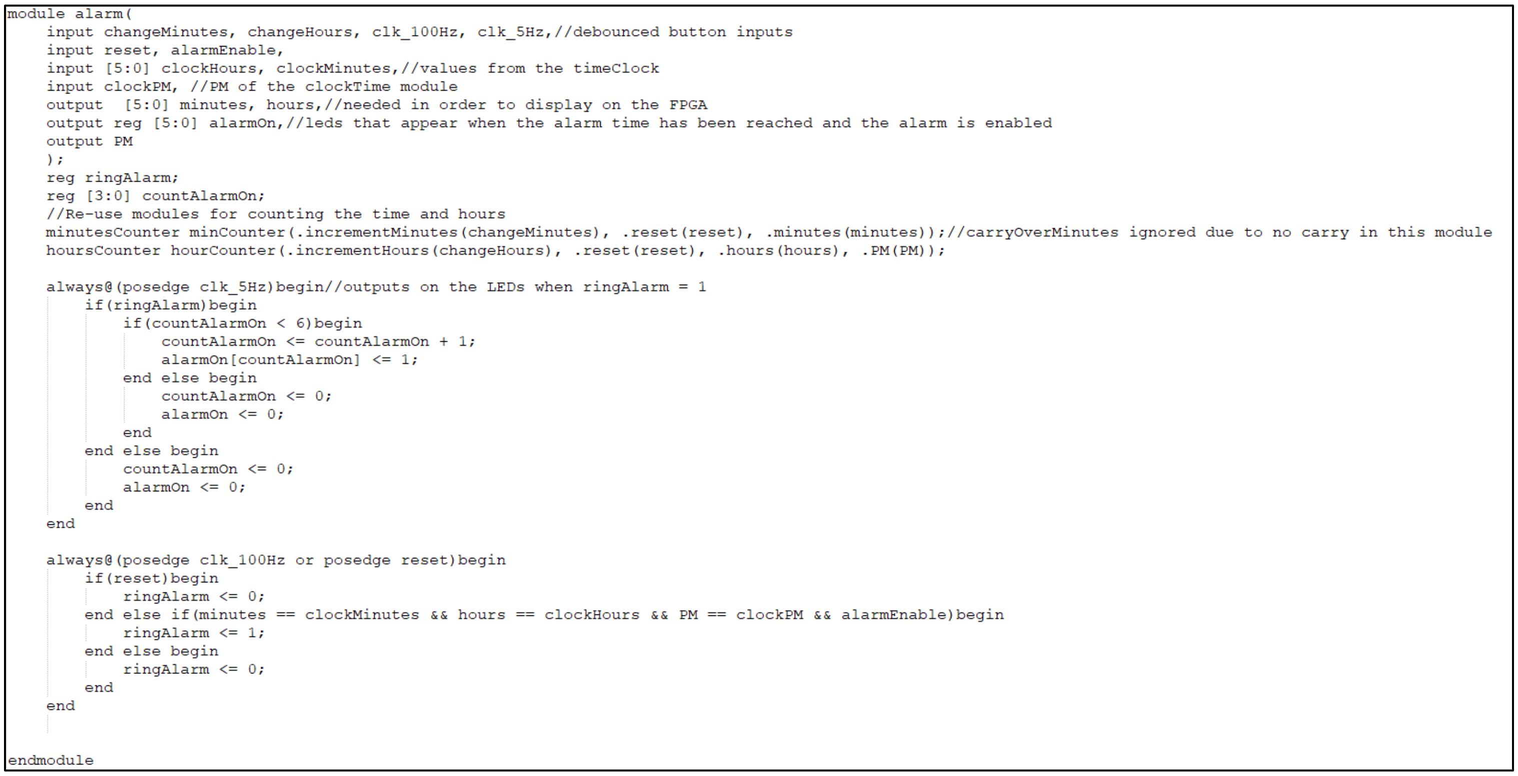
Figure 3.5: Time Clock Module Code

This module provided the necessary connections between the 3 counter modules as well as allowing for a centralized location from which the main module can access the values of the clock hours, minutes, and the condition of PM (if 0 or 1). Note that all of the inputs necessary for the function of the lower modules are present for this module.

Although not stated in the previous modules, the reset input is used to reset all of the values in the modules to the starting time of 12:00:00 AM (PM is 0).

## 3.6 The Alarm Module

The next logical step for the design was the alarm module which stored the value for the alarm on the alarm clock and checked if the clock value reaches the alarm value. The code used for this module is shown below in Figure 3.6.



```
module alarm(
    input changeMinutes, changeHours, clk_100Hz, clk_5Hz,//debounced button inputs
    input reset, alarmEnable,
    input [5:0] clockHours, clockMinutes,//values from the timeClock
    input clockPM, //PM of the clockTime module
    output  [5:0] minutes, hours,//needed in order to display on the FPGA
    output reg [5:0] alarmOn,//leds that appear when the alarm time has been reached and the alarm is enabled
    output PM
    );
    reg ringAlarm;
    reg [3:0] countAlarmOn;
    //Re-use modules for counting the time and hours
    minutesCounter minCounter(.incrementMinutes(changeMinutes), .reset(reset), .minutes(minutes));//carryOverMinutes ignored due to no carry in this module
    hoursCounter hourCounter(.incrementHours(changeHours), .reset(reset), .hours(hours), .PM(PM));

    always@(posedge clk_5Hz)begin//outputs on the LEDs when ringAlarm = 1
        if(ringAlarm)begin
            if(countAlarmOn < 6)begin
                countAlarmOn <= countAlarmOn + 1;
                alarmOn[countAlarmOn] <= 1;
            end else begin
                countAlarmOn <= 0;
                alarmOn <= 0;
            end
        end else begin
            countAlarmOn <= 0;
            alarmOn <= 0;
        end
    end

    always@(posedge clk_100Hz or posedge reset)begin
        if(reset)begin
            ringAlarm <= 0;
        end else if(minutes == clockMinutes && hours == clockHours && PM == clockPM && alarmEnable)begin
            ringAlarm <= 1;
        end else begin
            ringAlarm <= 0;
        end
    end


endmodule
```

Figure 3.6: Alarm Module Code

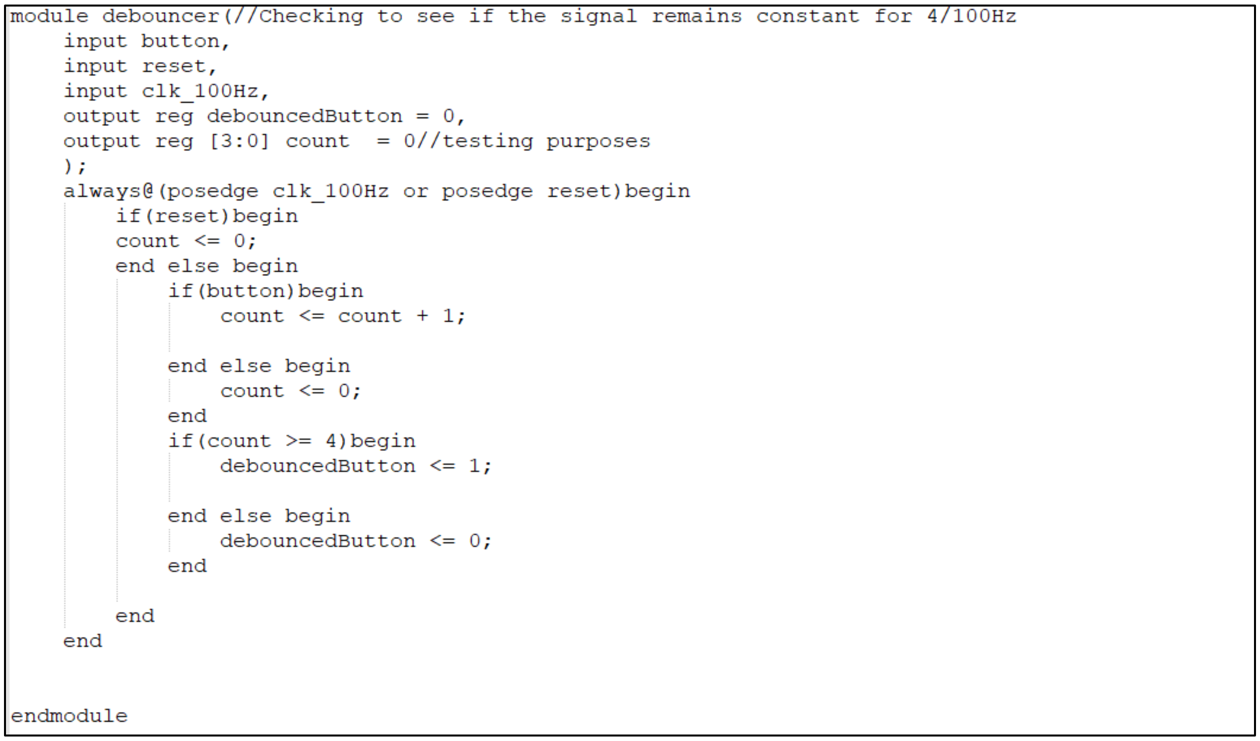
This module had a very similar function when compared to the clockTime module where the alarm module uses the same hours and minutes counters as clockTime. The differences lie in how there are no seconds counter as the alarm value must remain static and, in addition, the carryMinutes output is not used as the minutesCounter module should not change the hoursCounter module.

The detection of the matching of the clock time and the set alarm time is done by first using a 100 Hz clock (created by using the clock divider shown in section 4.1). The module proceeds to compare the values of time for the clock and the alarm and if they match the ringAlarm bit turns on notifying the module that the alarm time has been reached by the clock.

The first always block represents what happens when the alarm time is reached but, due to it being one of the innovations done for this project, it will not be covered in this section and will rather be covered in section 5.

## 3.7 The Button Debouncer

In every preceding module descriptions, the button inputs were referred to as changeHours and changeMinutes and not the actual inputs of hoursBTN and minutesBTN. This is due to those inputs needing to be debounced first before being used in the other modules due to how sensitive the FPGA buttons are to light touches. Buttons such as changeTime and changeAlarm do not need debouncing due to how they are on only when held but the hoursBTN and minutesBTN need to be able to provide inputs by single press and not count twice or more when being pressed due to uneven pressure. In order to ensure that the buttons were pressed for a sufficient amount of time before sending the signals to the other modules, the button inputs were debounced. The code used for this module is shown in Figure 3.7 below.



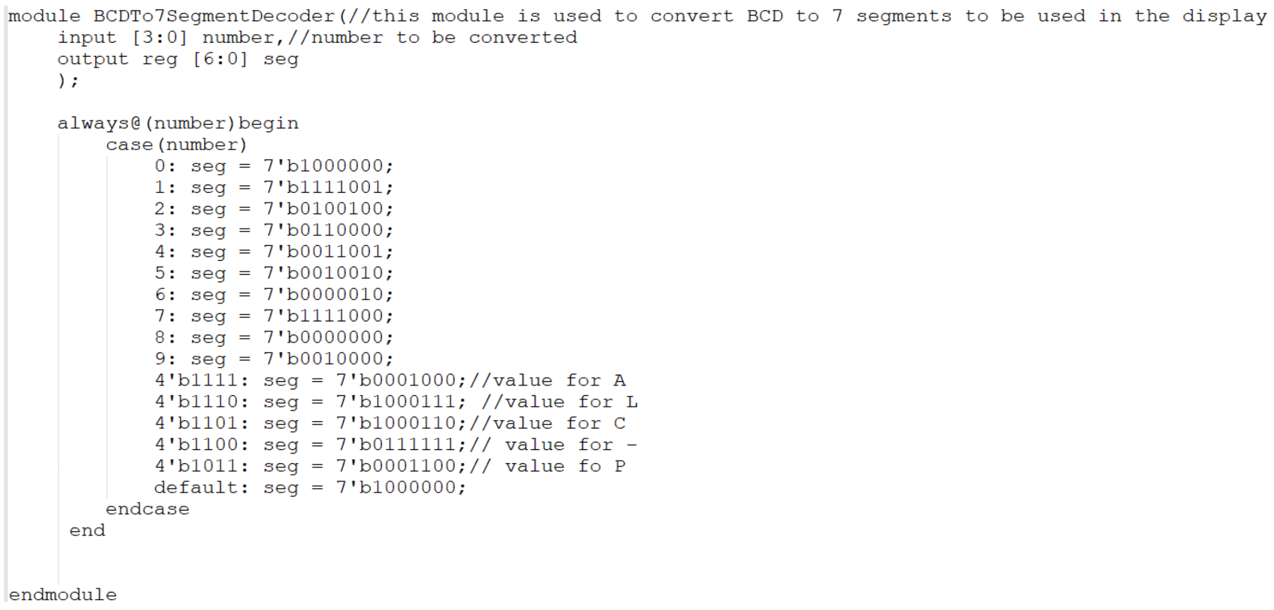
```
module debouncer(//Checking to see if the signal remains constant for 4/100Hz
    input button,
    input reset,
    input clk_100Hz,
    output reg debouncedButton = 0,
    output reg [3:0] count  = 0//testing purposes
    );
    always@(posedge clk_100Hz or posedge reset)begin
        if(reset)begin
        count <= 0;
        end else begin
            if(button)begin
                count <= count + 1;

            end else begin
                count <= 0;
            end
            if(count >= 4)begin
                debouncedButton <= 1;

            end else begin
                debouncedButton <= 0;
            end

        end
    end


endmodule
```

Figure 3.7: Debouncer Module Code

The general idea of the module is simple. The module samples an input from the button (button) every clock cycle of the 100 Hz clock and increments the value of count until it reaches the desired value of 4 samples and then outputs that the button was pressed (debouncedButton <= 1). If at any point during the sampling the button input stops being pressed, the count resets itself to 0. This allows the button input to become stable longer and for the user to not push the button twice while only trying to push it once.

## 3.8 BCDTo7SegmentDecoder

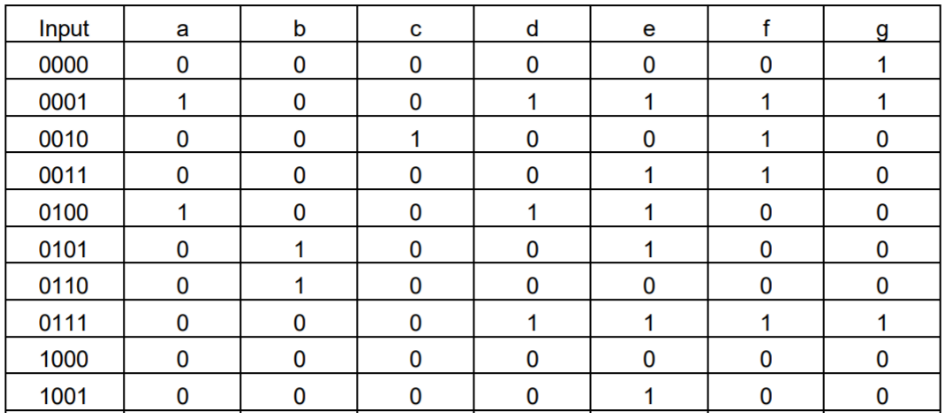
This module is a sub-module of the display controller (shown in the following section). The task of this module is to take in as input a number and decode it into the segment values used for the 7-segment display, The code used for this section is shown in Figure 3.8 below.



```
module BCDTo7SegmentDecoder(//this module is used to convert BCD to 7 segments to be used in the display
    input [3:0] number,//number to be converted
    output reg [6:0] seg
    );

    always@(number)begin
        case(number)
            0: seg = 7'b1000000;
            1: seg = 7'b1111001;
            2: seg = 7'b0100100;
            3: seg = 7'b0110000;
            4: seg = 7'b0011001;
            5: seg = 7'b0010010;
            6: seg = 7'b0000010;
            7: seg = 7'b1111000;
            8: seg = 7'b0000000;
            9: seg = 7'b0010000;
            4'b1111: seg = 7'b0001000;//value for A
            4'b1110: seg = 7'b1000111; //value for L
            4'b1101: seg = 7'b1000110;//value for C
            4'b1100: seg = 7'b0111111;// value for -
            4'b1011: seg = 7'b0001100;// value fo P
            default: seg = 7'b1000000;
        endcase
     end


endmodule
```

Figure 3.8: BCDTo7SegmentDisplay Code

The values that can be converted range from the integer values of 0 to 9 and 5 other values that are used to represent the letters A, L, C, and P which are used for the innovation. The seg values determined for the numbers were found using the truth table shown in Table 3.8 below.



| Input | a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|---|
| 0000 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0001 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0010 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0011 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0100 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0101 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0110 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0111 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1000 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1001 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

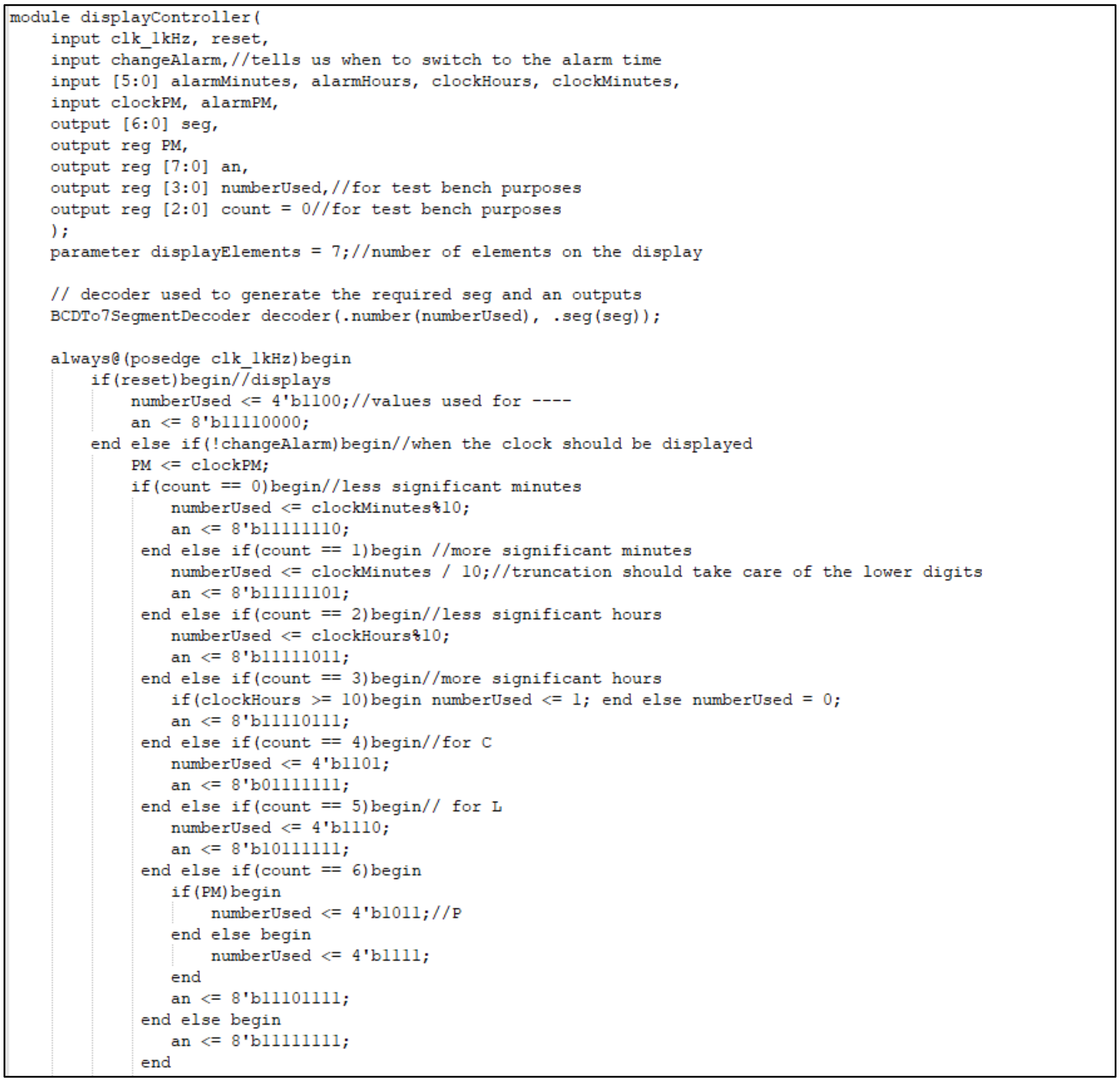
Table 3.8: Number Input Truth Table [1]

Note that the bits used for the seg values were actually in the reverse order as g is the most significant bit and a is the least significant bit. For example, on the table the seg values for an input of 0 is shown as 7'b0000001 but is flipped and put on the code as 7'b1000000.

The representation of the letters in seg values will not be shown in this section due to it being an innovation and will rather be shown in section 5.

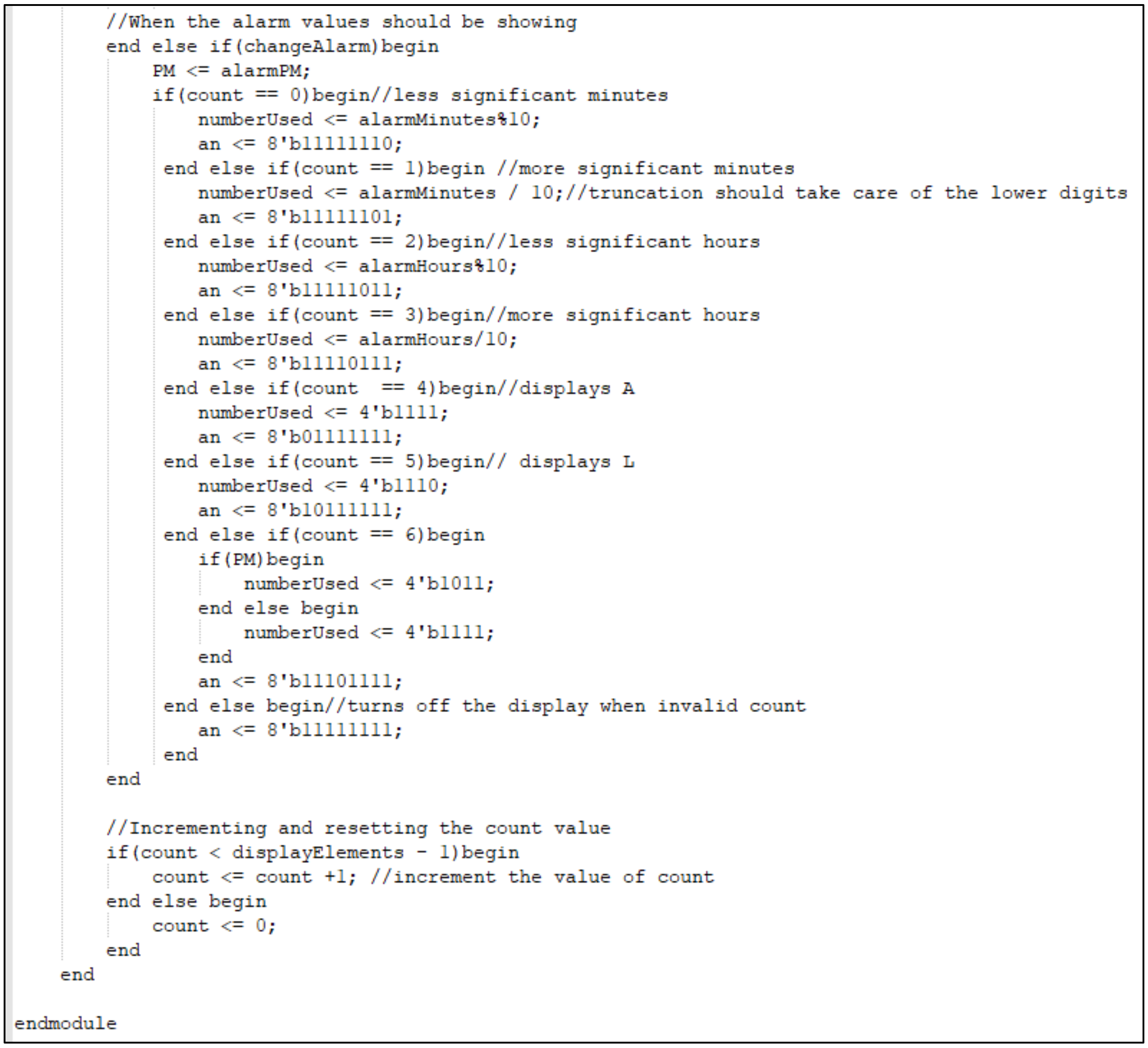
## 3.9 Display Controller

The final submodule to be used for the alarm clock design is the display controller which controls the 7-segment display. The controller is designed to take the time of both the clock and alarm and choose control which is being displayed by the use of the changeAlarm button input (if changeAlarm = 1 use the alarm time and vice versa). The code used for the display controller is shown below in Figures 3.9.1 and 3.9.2 below.



```
module displayController(
    input clk_1kHz, reset,
    input changeAlarm,//tells us when to switch to the alarm time
    input [5:0] alarmMinutes, alarmHours, clockHours, clockMinutes,
    input clockPM, alarmPM,
    output [6:0] seg,
    output reg PM,
    output reg [7:0] an,
    output reg [3:0] numberUsed,//for test bench purposes
    output reg [2:0] count = 0//for test bench purposes
    );
    parameter displayElements = 7;//number of elements on the display

    // decoder used to generate the required seg and an outputs
    BCDTo7SegmentDecoder decoder(.number(numberUsed), .seg(seg));

    always@(posedge clk_1kHz)begin
        if(reset)begin//displays
            numberUsed <= 4'b1100;//values used for ----
            an <= 8'b11110000;
        end else if(!changeAlarm)begin//when the clock should be displayed
            PM <= clockPM;
            if(count == 0)begin//less significant minutes
                numberUsed <= clockMinutes%10;
                an <= 8'b11111110;
             end else if(count == 1)begin //more significant minutes
                numberUsed <= clockMinutes / 10;//truncation should take care of the lower digits
                an <= 8'b11111101;
             end else if(count == 2)begin//less significant hours
                numberUsed <= clockHours%10;
                an <= 8'b11111011;
             end else if(count == 3)begin//more significant hours
                if(clockHours >= 10)begin numberUsed <= 1; end else numberUsed = 0;
                an <= 8'b11110111;
             end else if(count == 4)begin//for C
                numberUsed <= 4'b1101;
                an <= 8'b01111111;
             end else if(count == 5)begin// for L
                numberUsed <= 4'b1110;
                an <= 8'b10111111;
             end else if(count == 6)begin
                if(PM)begin
                    numberUsed <= 4'b1011;//P
                end else begin
                    numberUsed <= 4'b1111;
                end
                an <= 8'b11101111;
             end else begin
                an <= 8'b11111111;
             end
```

Figure 3.9.1: Display Controller Code (Part 1)



```
        //When the alarm values should be showing
        end else if(changeAlarm)begin
            PM <= alarmPM;
            if(count == 0)begin//less significant minutes
                numberUsed <= alarmMinutes%10;
                an <= 8'b11111110;
             end else if(count == 1)begin //more significant minutes
                numberUsed <= alarmMinutes / 10;//truncation should take care of the lower digits
                an <= 8'b11111101;
             end else if(count == 2)begin//less significant hours
                numberUsed <= alarmHours%10;
                an <= 8'b11111011;
             end else if(count == 3)begin//more significant hours
                numberUsed <= alarmHours/10;
                an <= 8'b11110111;
             end else if(count  == 4)begin//displays A
                numberUsed <= 4'b1111;
                an <= 8'b01111111;
             end else if(count == 5)begin// displays L
                numberUsed <= 4'b1110;
                an <= 8'b10111111;
             end else if(count == 6)begin
                if(PM)begin
                    numberUsed <= 4'b1011;
                end else begin
                    numberUsed <= 4'b1111;
                end
                an <= 8'b11101111;
             end else begin//turns off the display when invalid count
                an <= 8'b11111111;
             end
        end

        //Incrementing and resetting the count value
        if(count < displayElements - 1)begin
            count <= count +1; //increment the value of count
        end else begin
            count <= 0;
        end
    end

endmodule
```

Figure 3.9.2: Display Controller Code (Part 2)

Due to the way that the 7-segment display works and how only one value can be shown on one or more anodes (shown in Figure 3.9.3 below) at any given time. This meant that the segment values as well as which anode is being displayed at any given time had to be cycled in order to display all the different values of the clock or alarm time at the same "time".

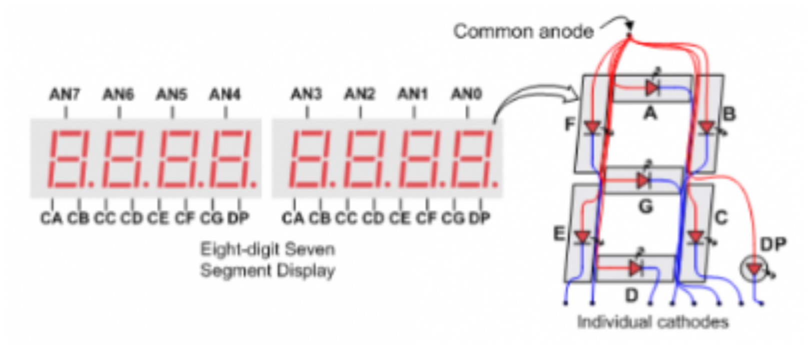


Figure 3.9.3: 7-Segment Display Configuration [2]

The display therefore was put on a 1 kHz clock and would show one of the values for 1/1000 seconds and continuously cycle through each anode displaying the desired values at the desired positions. This meant that a system had to be in place that could, each 1/1000 second, determine what value should be displayed, convert the display to segment values, and determine which anode it should be placed on. This is, in a nutshell, what the display controller does. It takes in the clock time and the alarm time and, with the changeAlarm button input, determines which time values should be displayed.

The next step for the controller is to determine what digit should be showing as numbers such as 45 have two digits, one digit being 4 and the other being 5. These two digits then have to be displayed on two different anodes which means two completely different outputs for the display controller. The way these input values are separated into 2 new values is through the use of integer truncation as well as modulus operations. For example, the first digit of the value 45 can be found by dividing the value by 10 and then, due to the number being an integer, the remaining value is the integer value of 4. This is shown below, where the highlighted value represent that the value discarded during the integer truncation.

$$45/10 = 4.5 = 4 \text{ (integer)}$$

A similar logic can be applied for finding the second digit where the remainder of the total value is the one that is being saved. This is shown below.

$$45\%10 = 4.\frac{5}{10} = 5 \text{ (integer)}$$

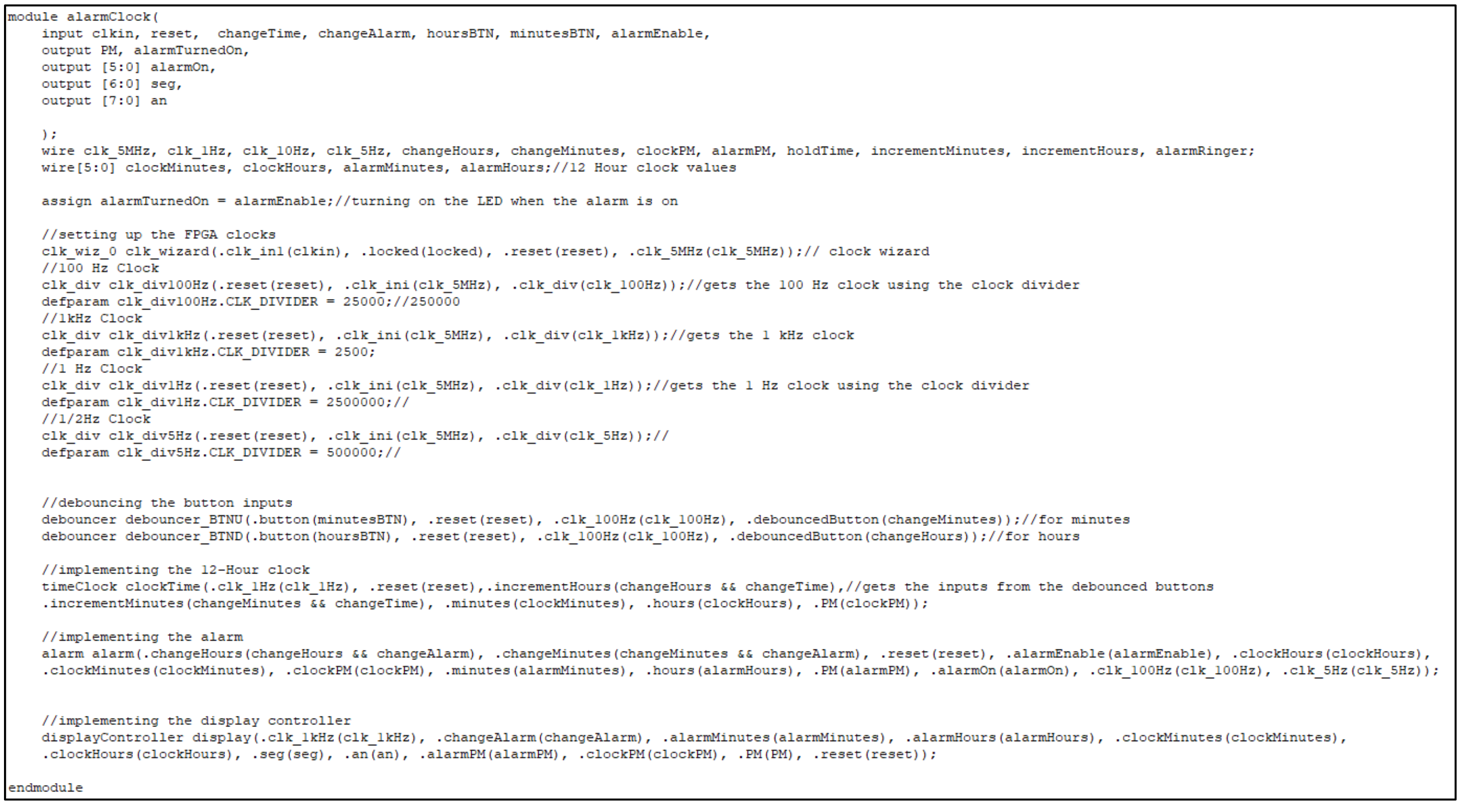
In this way, the values of the first and second digits can be obtained for the decoding into segment values. The decoding itself is done through the use of the BCDTo7SegmentDecoder previously shown in section 4.8.

With the segment values now found, the next task is in selecting which anode should be used. This is simply done with the use of an “an” output which tells the display which position they should be displayed on. Due to how they are anodes, the entered value of 0 indicates that that position should output and the entered value of 1 indicates that it shouldn’t output. For example, if you want to display on the leftmost anode you have to state that an = 8’b01111111.

With all of this done, the last part is to select when each number and position will be chosen. This is done simply through the use of a counter (count) which goes through the amount of display elements set by the displayElements parameter and ensures that each different output occurs during a full cycle. Due to the speed at which the module operates, there are no specific order that is required for the outputs as long as all of them are present.

## 3.10 Main Alarm Clock Module

With all of the submodules shown and described, the main module called alarmClock can now be approached. The code used for this module is shown in Figure 3.10 below.



```
module alarmClock(
    input clkin, reset,  changeTime, changeAlarm, hoursBTN, minutesBTN, alarmEnable,
    output PM, alarmTurnedOn,
    output [5:0] alarmOn,
    output [6:0] seg,
    output [7:0] an

    );
    wire clk_5MHz, clk_1Hz, clk_10Hz, clk_5Hz, changeHours, changeMinutes, clockPM, alarmPM, holdTime, incrementMinutes, incrementHours, alarmRinger;
    wire[5:0] clockMinutes, clockHours, alarmMinutes, alarmHours;//12 Hour clock values

    assign alarmTurnedOn = alarmEnable;//turning on the LED when the alarm is on

    //setting up the FPGA clocks
    clk_wiz_0 clk_wizard(.clk_in1(clkin), .locked(locked), .reset(reset), .clk_5MHz(clk_5MHz));// clock wizard
    //100 Hz Clock
    clk_div clk_div100Hz(.reset(reset), .clk_ini(clk_5MHz), .clk_div(clk_100Hz));//gets the 100 Hz clock using the clock divider
    defparam clk_div100Hz.CLK_DIVIDER = 25000;//250000
    //1kHz Clock
    clk_div clk_div1kHz(.reset(reset), .clk_ini(clk_5MHz), .clk_div(clk_1kHz));//gets the 1 kHz clock
    defparam clk_div1kHz.CLK_DIVIDER = 2500;
    //1 Hz Clock
    clk_div clk_div1Hz(.reset(reset), .clk_ini(clk_5MHz), .clk_div(clk_1Hz));//gets the 1 Hz clock using the clock divider
    defparam clk_div1Hz.CLK_DIVIDER = 2500000;//
    //1/2Hz Clock
    clk_div clk_div5Hz(.reset(reset), .clk_ini(clk_5MHz), .clk_div(clk_5Hz));//
    defparam clk_div5Hz.CLK_DIVIDER = 500000;//


    //debouncing the button inputs
    debouncer debouncer_BTNU(.button(minutesBTN), .reset(reset), .clk_100Hz(clk_100Hz), .debouncedButton(changeMinutes));//for minutes
    debouncer debouncer_BTND(.button(hoursBTN), .reset(reset), .clk_100Hz(clk_100Hz), .debouncedButton(changeHours));//for hours

    //implementing the 12-Hour clock
    timeClock clockTime(.clk_1Hz(clk_1Hz), .reset(reset),.incrementHours(changeHours && changeTime),//gets the inputs from the debounced buttons
    .incrementMinutes(changeMinutes && changeTime), .minutes(clockMinutes), .hours(clockHours), .PM(clockPM));

    //implementing the alarm
    alarm alarm(.changeHours(changeHours && changeAlarm), .changeMinutes(changeMinutes && changeAlarm), .reset(reset), .alarmEnable(alarmEnable), .clockHours(clockHours),
    .clockMinutes(clockMinutes), .clockPM(clockPM), .minutes(alarmMinutes), .hours(alarmHours), .PM(alarmPM), .alarmOn(alarmOn), .clk_100Hz(clk_100Hz), .clk_5Hz(clk_5Hz));


    //implementing the display controller
    displayController display(.clk_1kHz(clk_1kHz), .changeAlarm(changeAlarm), .alarmMinutes(alarmMinutes), .alarmHours(alarmHours), .clockMinutes(clockMinutes),
    .clockHours(clockHours), .seg(seg), .an(an), .alarmPM(alarmPM), .clockPM(clockPM), .PM(PM), .reset(reset));

endmodule
```

Figure 3.10: Alarm Clock Main Module

The main module is used to interconnect the outputs of all of the lower modules within the alarm clock as well as provide inputs and output into the system.

The only two significant things to note are the defparam statements that were used to overwrite the CLK_DIVIDER value in the clock dividers and how they were used to generate the required clocks for the submodules and the clock wizard. The clock wizard IP was used to manage the 100 MHz clock of the FPGA and divide it down into the 5MHz clock used for the clock dividers.

## 4. INNOVATIONS

This section covers the 3 different innovations that were implemented onto the alarm clock. These innovations were the displaying of "AL" and "CL" on the display when it showed the alarm time and clock time (respectively), the displaying of "A" and "P" on the rightmost anode of the first 4 anodes when the time displayed was in AM or PM (respectively), and the flashing of several LEDS in a cascading motion when the alarm time was reached.

### 4.1 AL AND CL DISPLAY FOR TIME

For this innovation, new seg values had to be developed and placed into the decoder. The determination of the values for the seg values was done using the diagram shown in Figure 4.1 below and putting a value of 0 for the segment that was desired to be on and a 1 for the segments that should be off.

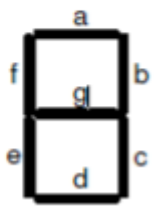


Figure 4.1: 7-Segment Breakdown [1]

Using the diagram above, Table 4.1 below could be generated for all of the letters used in the innovations.

| Letter | a | b | c | d | e | f | g |
|---|---|---|---|---|---|---|---|
| A | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| P | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| L | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| C | 0 | 1 | 1 | 0 | 0 | 0 | 1 |

Table 4.1: Letter to Segment Conversion

With the segment codes found, the new letter outputs could be added to the display decoder and display controller. This was done by placing two new entries for "AL" when changeAlarm was true and two new entries when for "CL" when changeAlarm was false. In addition, the value of displayElements was increased to accommodate the new elements into the display rotation. With this done, "AL" would be outputted onto the display screen when the changeAlarm button was pressed and "CL" would be present on the screen when changeAlarm was not pressed (when the screen was showing the clock time).

## 4.2 A and P Output for AM and PM

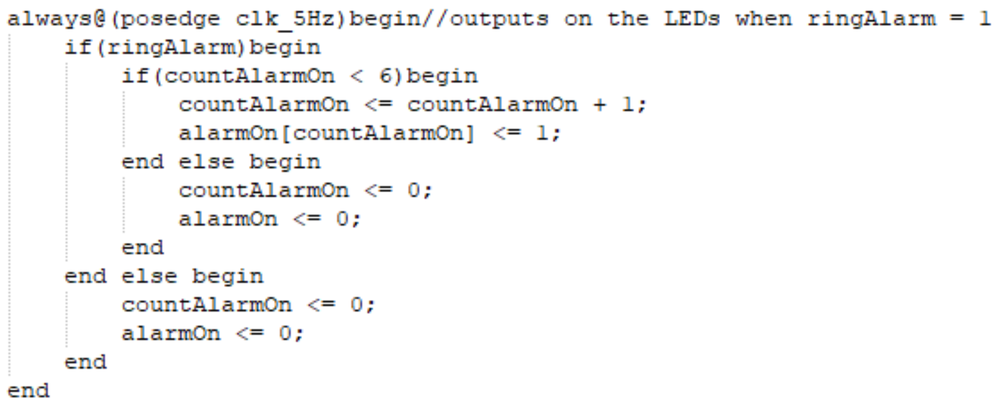
This innovation follows suit with the previous one where A would be outputted onto the fourth anode (when counting from left to right) when the time was in AM and P when the time is in PM. This change was done to clearly tell the user that the time that was reached was in PM/AM due to how the standard tell LED15 was often overlooked and not obvious.

The addition was designed in the similar manner where the segment codes were found and added to the display decoder and a new reference value was implemented so that the display controller could access the letter.

The last part of this innovation design is the determination of when to output "A" and when to output "P". This was done using an alarmPM and clockPM as inputs to the display controller and enabling the appropriate one with the use of the if statements.

## 4.3 Cascading LEDs for the Alarm

The last innovation was the addition of the cascading of the LEDs when the alarm was reached. This innovation was added in order to make the reaching of the clock time more obvious due to how the original ½ second interval LED1 was not apparent enough. The innovation was done by modifying the alarm module and adding a 5Hz clock. With the new clock implemented, the output alarmOn would be increased in size to contain 6 LEDs. The LEDs would then be turned on starting from right to left until the max count of LEDs was reached where they would all be turned off and cycled through the same thing again. This would occur as long as the clock time matched the alarm time. The code used for this section is a sub-section of the alarm module (see section 4.6) and is shown in Figure 4.3 below.



```
always@(posedge clk_5Hz)begin//outputs on the LEDs when ringAlarm = 1
    if(ringAlarm)begin
        if(countAlarmOn < 6)begin
            countAlarmOn <= countAlarmOn + 1;
            alarmOn[countAlarmOn] <= 1;
        end else begin
            countAlarmOn <= 0;
            alarmOn <= 0;
        end
    end else begin
        countAlarmOn <= 0;
        alarmOn <= 0;
    end
end
```

Figure 4.3: Cascading LED Code

## 5. Design Verification

Some of the modules outlined in section 3 were subjected to simulations in order to verify the validity of the design prior to implementing them. Due to the input clock being 100 MHz for the main alarmClock module, a simulation could not be done for this module. The following section will show the simulation results to the test benches conducted on the modules.

### 5.1 Clock Time Simulation

The first simulation that was done was in verifying the function of the clockTime module. The most significant portion of the output wave received during the simulation is shown below in Figure 5.1.

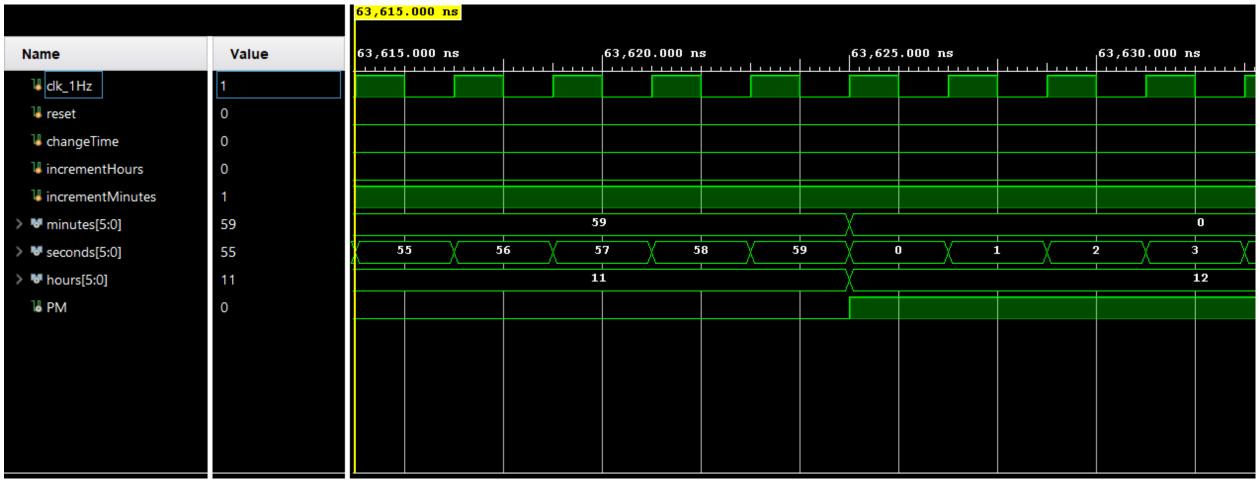


Figure 5.1: Clock Time Output Waveform

The design can be seen as being valid from the above figure due to how the minutes, seconds, and hours all increment the way they should (11h59min59secs -> 12h00min00sec). In addition, the PM value can be seen to change upon the change from 11 AM to 12 PM.

### 5.2 The Button Debouncer Simulation

The debouncer testbench output is shown in Figure 5.2 below.

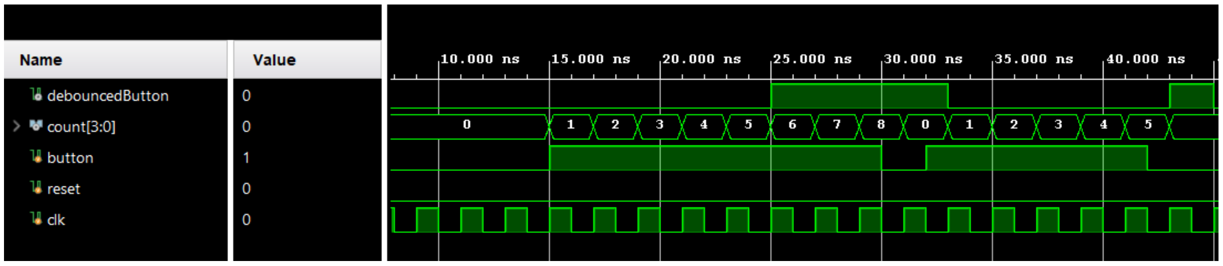


Figure 5.2: Button Debouncer Test Bench

From the above waveform, the debouncer can be seen as working due to how upon reaching the desired amount of inputs, it begins to output the debouncedButton value.

## 5.3 The Clock Divider Simulation

Figure 5.3 below show the output waveform for the testbench of the clock divider.

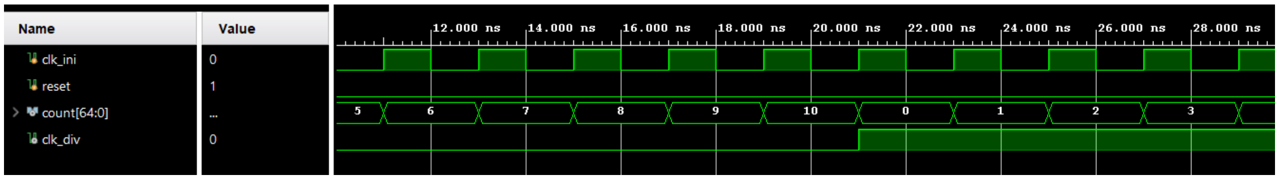


Figure 5.3: Clock Divider Output Waveform

This testbench was constructed with the CLK_DIVIDER value of 10 which means that the output clock should be 1/10 of the input clock. As can be seen in the figure, the clock divider works as intended and divides the input clock by the desired amounts.

## 6. Conclusion

In conclusion, the design requirements set by the lab manual was reached where the clock designed measured time and rang when the set alarm time was reached. The innovations added onto the design were the modification of the displayController and decoder to allow for new letters to be outputted onto the display when either the clock or alarm time were showing (“CL” for clock and “AL” for alarm) and when the time changed from AM to PM with the output of “A” and “P”. In addition, the alarm was made to output a cascade of LEDs turning on when the set alarm time was reached.

## 7. REFERENCES

[1] – “Alarm Clock Lab”, ELEC 3500, Date Accessed: 2019-12-07

[2] – Nexys 4 DDR Reference Manual,
https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual,
Date Accessed: 2019-12-07