Implementation of Alarm System on FPGA

Felsted, Samuel J.
felsteds@oregonstate.edu

Lamont, Sawyer J.
lamontsa@oregonstate.edu

Freund, Larsen
freundl@oregonstate.edu

December 2nd, 2024

1 Introduction

Spending time with loved ones over the holidays always brings up the question: "What are you learning in school?" This project aims to answer that question by demonstrating how a simple alarm clock can be created from the ground up using the principles of digital logic design. The logic was designed in Intel's Quartus development suite and tested in ModelSim. The final project was implemented on a DE10 FPGA and wired into a simple buzzer for a working prototpye (and something to show the grandparents!).

2 State Machine/Block Diagram

In our design, there are 2 fundamental states of the clock system and then various sub-states. The fundamental states are characterized by the passive state (displaying the time) and the active state (setting the alarm). The states are controlled by a single switch on device and thus the state can be represented by a 1 or a 0. The 7 segment displays on the FPGA display the current clock time of the current state.

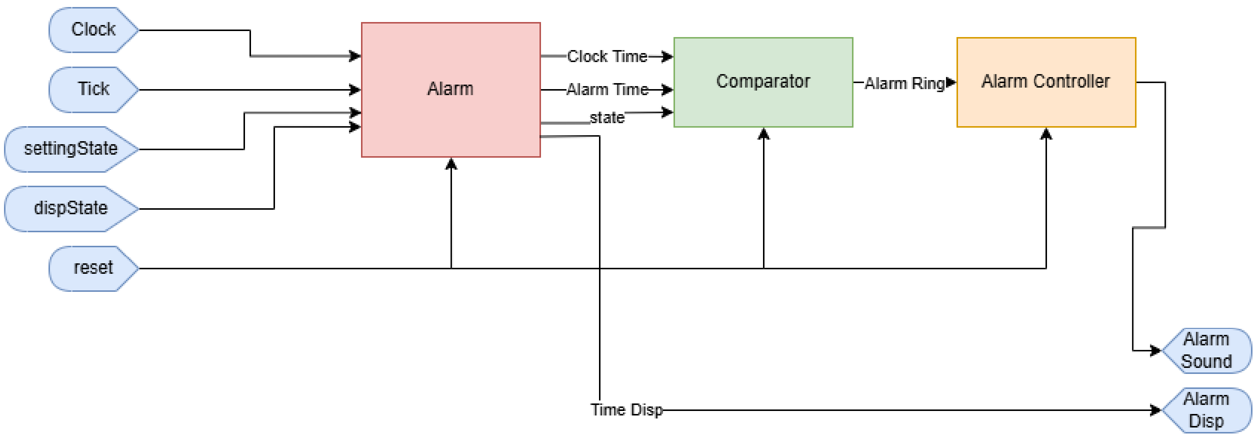


Figure 1: Block Diagram of Top Level

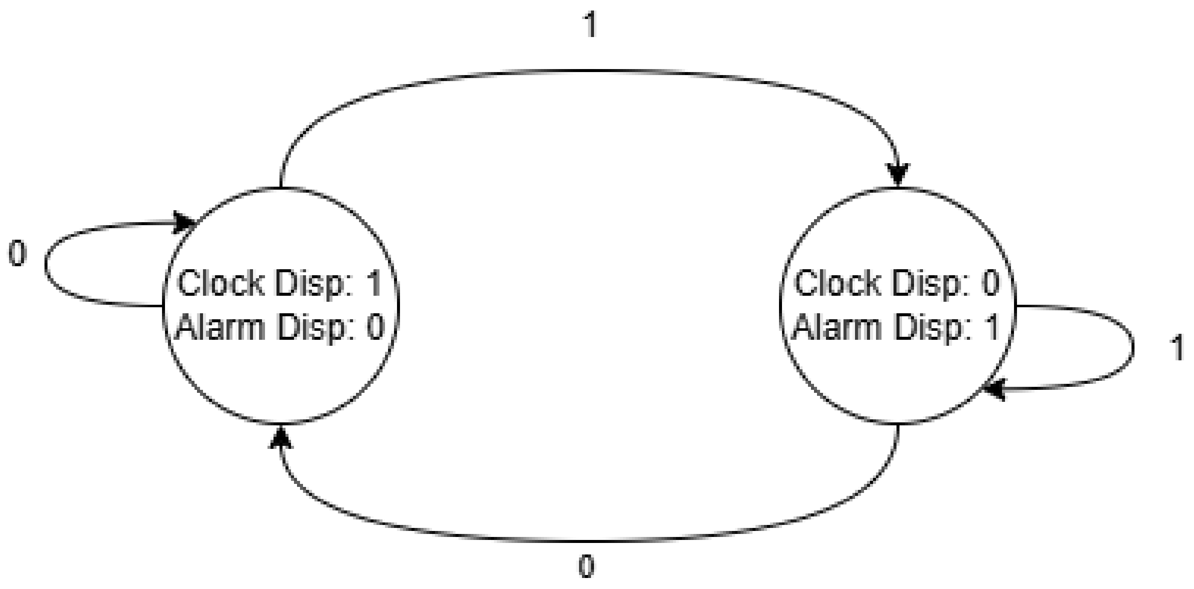


Figure 2: State Diagram of Time State

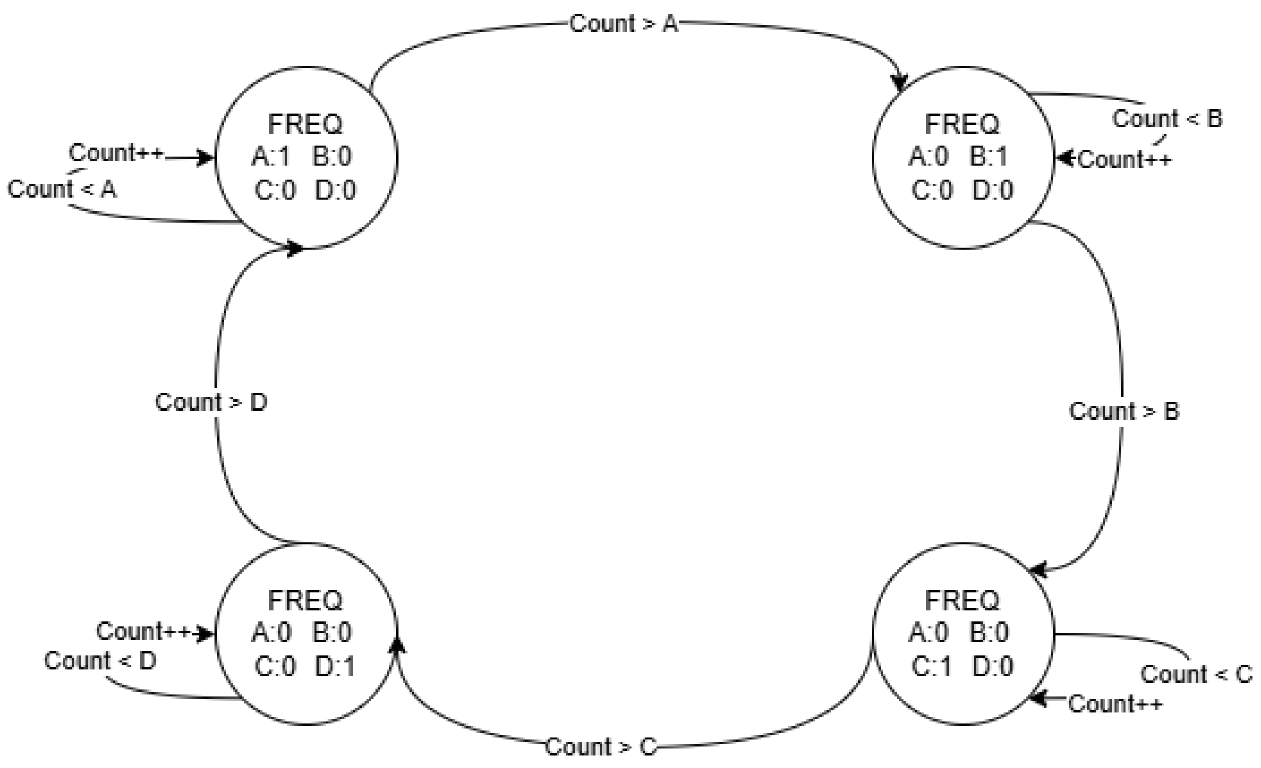


Figure 3: State Diagram of Frequency State

3 Modules

3.1 Clock

The clock module is designed to calculate both the alarm value and the clock value by converting a 50-MHz input clock signal into seconds using a **counter** and **comparator** module (Appendix B.1, B.2) [1]. This is achieved through a ripple-counter mechanism, where seconds are incremented until a value of 60 is reached, at which point the seconds counter resets, and a minute is added to the minute counter. A similar process governs the hour counter, which resets after 60 minutes. To ensure precision during these transitions, a synchronizer module is incorporated, allowing updates to the counters only during the clock's rising edge. This prevents any spurious or floating values during calculations. The module also features pause and set functionalities for enhanced control. The pause functionality, implemented using a multiplexer, disconnects the clock signal to temporarily halt time updates. The set functionality, handled by the **setTime**(Appendix B.3) module, allows users to adjust seconds, minutes, or hours individually. When a specific setting input is activated, user actions such as button presses increment the corresponding time value by one. For instance, enabling the seconds and hours settings simultaneously increments both counters with each user input. These integrated features provide a robust and user-configurable clock module with precise synchronization and flexibility. For a detailed schematic of the design, refer to Figure 10.

3.2 Alarm

The alarm module is designed using two clock modules with distinct functionalities. The first module functions as a standard clock, taking a 50 MHz clock signal as input and incrementing time to provide a steady and reliable output. This module operates continuously to maintain an accurate time count. The second module serves as the alarm-setting interface. Unlike the first module, it does not increment with the passage of time; its primary purpose is to allow the user to view, set, and store the desired alarm time. Although it does not actively track time, it synchronizes with the clock module for consistent operation.

To switch between clock and alarm states, the **settingState** module (App. B.4) is employed. This module processes the system's state input and setting input, redirecting user commands to their respective destinations based on the current mode. The alarm module outputs three key pieces of information: the current time from the clock module, the set alarm time, and the system's state. For a detailed schematic of the design, refer to Figure 11.

3.3 Top Level

3.3.1 Display Driver

The Display Driver module uses one Alarm module and outputs the data needed to display it on a seven-segment display. To do this it will first decide whether to display the clock time or alarm time based on the state. This is done with a mux. The data outputted from the alarm has all the time data wrapped into one bus to minimize the amount of outputs needed. The data is split into seconds, minutes, and hours based on their bit locations. This data is then put into a **parser** module (App. B.5) to split the data into ones and tens. These are then each put into a **sevenSegDisplayDriver** module (App. B.6) [1] where the proper outputs are created to display the information. The DisplayDriver module also outputs the state, the clock data, and alarm data, allowing for comparison. For a detailed schematic of the design, refer to Figure 12.

3.3.2 Clock Comparator Module

The **clockStateComparator** module (App. B.7) is used to detect when the current time is equal to the time set on the alarm. These inputs are taken from the **Alarm** module (App. 11). When these states are equal, a register is set to HIGH, and the value is outputted to enable the alarm sound. Creating this in SystemVerilog is very straightforward. This will remain HIGH until the alarm is manually disabled or the RESET button is pushed.

3.3.3 Wave Generator

The **waveGenerator** module (App. B.8) is used to generate waves of different frequency to create sound waves. Fundamentally, a wave is just a series of pulses on a set period. Using the relationship:

$$\text{Period} = \frac{1}{\text{Frequency}}$$

We can create waves of different frequencies to create various music notes as an output to the alarm. The module was created with the parameter **FREQ** (in Hertz) and the input clk to create a wave at a specified frequency. This is implemented using some math to convert the 50 MHz clock and the desired period (derived from the frequency) to a number of counts that the **Counter**(App. B.1) [1] module can count to. In order to create the desired wave, the output must be high for half of the period and low for the other half. This can be achieved by using a simple division operation. Bringing all of these principles together allows waves to be generated for variable frequencies.

3.3.4 Alarm Controller

The **alarmController** module (App. B.9) is used to generate a multi-note alarm. The first 4 parameters **(A, B, C, D)** represent the duration of each note in the alarm (in cycles). The current parameters are 100,000,000 for each note which should be 2 seconds per note (due to the FPGA internal clock being 50 MHz). Changing this will change the duration of each note. In addition, the next 3 parameters **(AFREQ, BFREQ, CFREQ)** correspond to the frequency of each note (in Hertz). The chosen numbers are approximately the frequency for C4 (262Hz), E4 (330Hz) and G4 (392Hz), which make up a standard C scale. To achieve the note duration, a counter is used for each note with a one-hot encoding state machine to control the enable of each wave generator used to create the notes. The module outputs a single output which represents the sound of the alarm. This is very customizable and could be used to play music (albeit at a very poor quality). For a detailed schematic of the entire top-level design design, refer to Figure 9.

4 Validation

To validate our results before implementing on the FPGA, we can simulate inputs in a control software. because of the modularity of our design, we can valid individual modules as pieces of a larger system to reduce complexity in our analysis.

4.1 Wave Generation

The simplest module to test is our **waveGenerator** module (App. B.8). All that is required is to drive the CLK and set a good testing parameter. Because the module is deisgned for a 50MHz input, we should set the frequency to be 5,000,000. This is an arbitrary number that will lead to a period of 10 seconds per cycle. In ModelSim, we have a period of 2 seconds per cycle. Running the do file for the simulation shows our expected period of 20 seconds. Both files are attached (Figure 4, App. C.1)

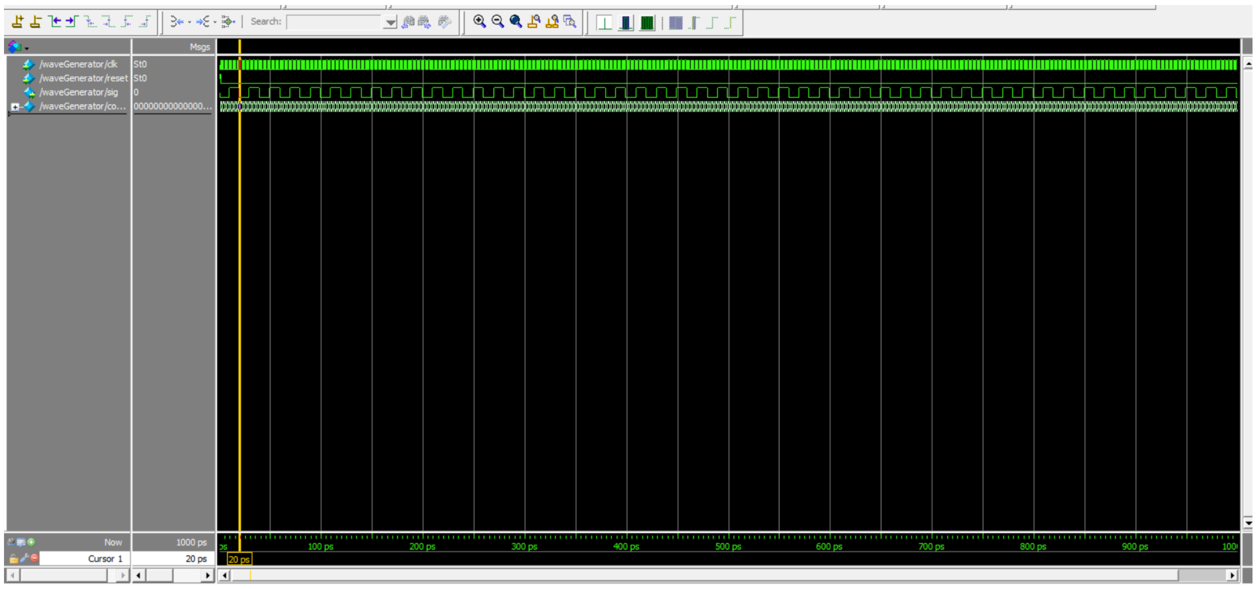


Figure 4: Simulated waveGenerator results

4.2 Alarm Controller

The next building block to test is the **alarmController** module (App. B.9). Simulating this has a near identical do file to the wave generator. However, different parameters need to be selected. The table below shows the full parameter list. These values were chosen to produce evenly timed waves that decrease in frequency until the 4th rest note. They take into account the internal mechanisms that account for the 50-MHz clock. The simulation and do file are of course attached (Figure 5, App. C.3). The results show exactly the expected results with the A section counting up to 100 and the output frequency being the greatest. Each subsequent section has its expected output and are all evenly timed.

Parameter	Value
A	100
B	100
C	100
D	100
AFREQ	5000000
BFREQ	4000000
CFREQ	3000000
N	32

Table 1: Parameters to wave generator

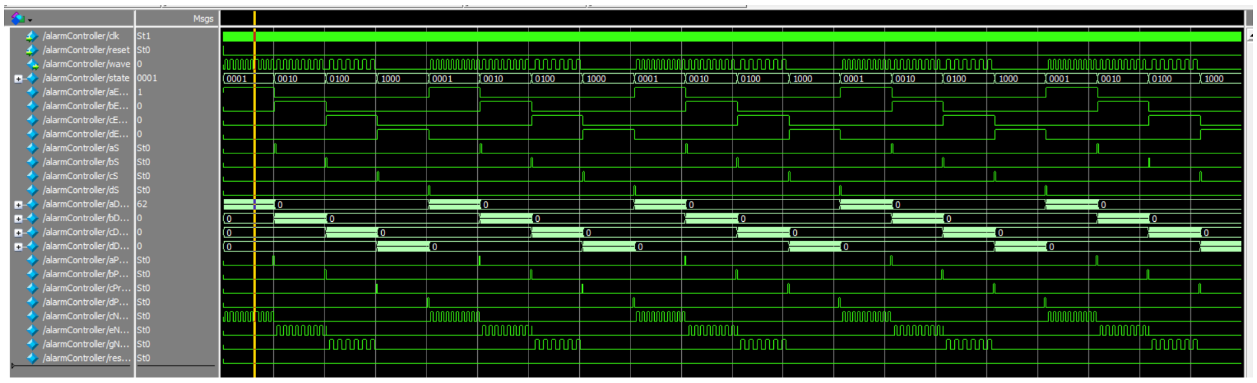


Figure 5: Simulated alarm controller results

4.3 Clock

Simulating the clock module is critical as both the alarm and the actual clock use it. Because of this, we only need to simulate one to prove the functionality of both. To test this module, we need to consider the fact it was designed for a 50-Mhz signal to pulse its internal clock. Thus for the simulation, we decided to use the manual control system to pulse the seconds so that we don't need to run the simulation for an extended amount of time and also we don't need to change the SystemVerilog code for the module. To test additional functionality, we decided to increment the mins at 50 cycles and the hours at 100 cycles. This can be seen in our simulation results which the first spike at setMin and the spike of setHours causing an increase in the /clock/min and /clock/hours.

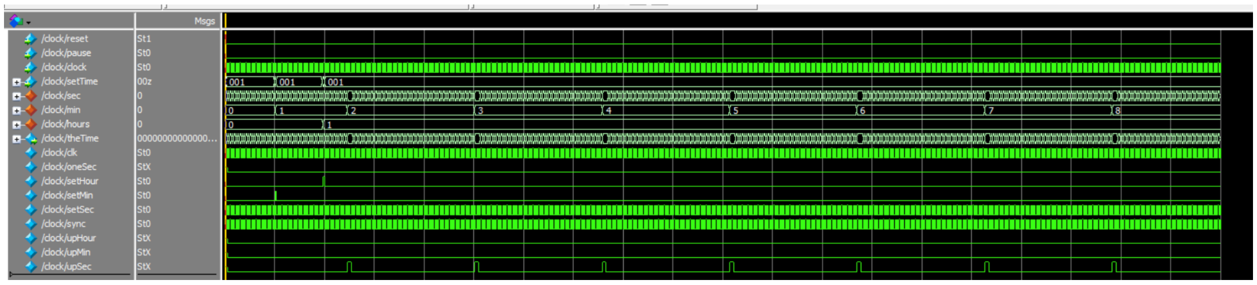


Figure 6: Simulated clock results

4.4 Final System

To simulate the final project, we need to show the ability to set a time and then see the alarm activate when the alarm time is equal to the clock time. To effectively simulate this, we chose the easy to display values from the previous simulations and used those methods to demonstrate the effectiveness of the design. We first enable the alarm as well as the reset in the clock state. Note how the /dp2/alarm is red as it has not been set yet. Next we switch into alarm mode with the state pin and use that to reset and then set the alarm to 2 using the manual set pin. Then we shift into clock mode and set the time to be 2. This enables the alarm which can be seen in the waveform in the /dp2/alarmPin simulation results in Figure 8. The code for this can be found in App. C.4.

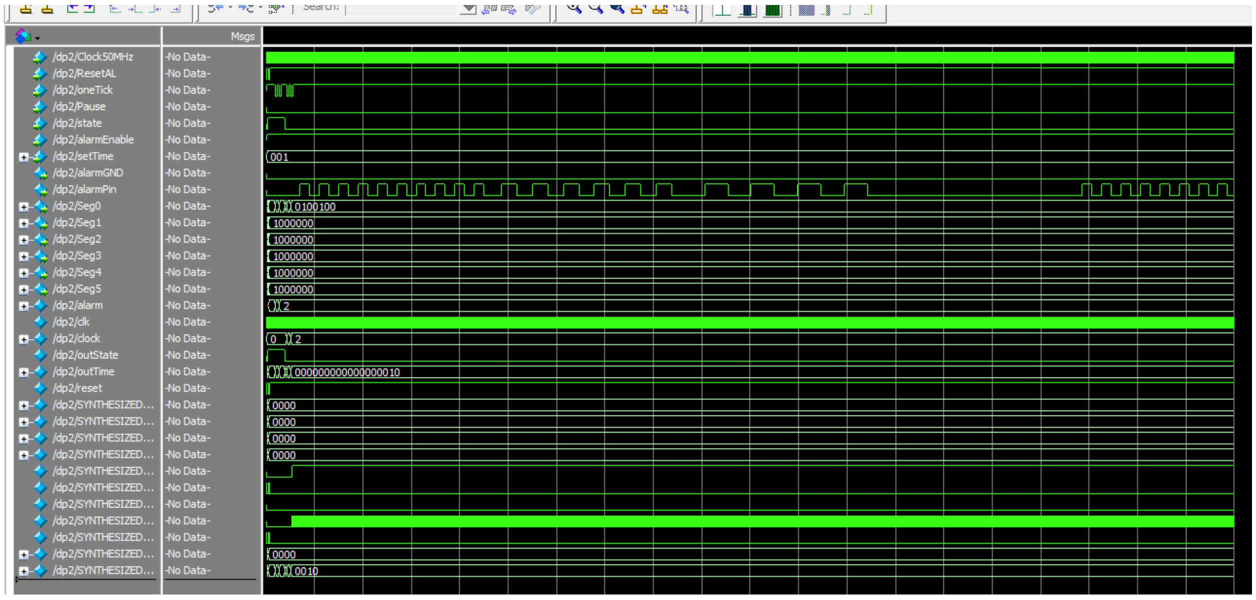


Figure 7: Simulated large scale results

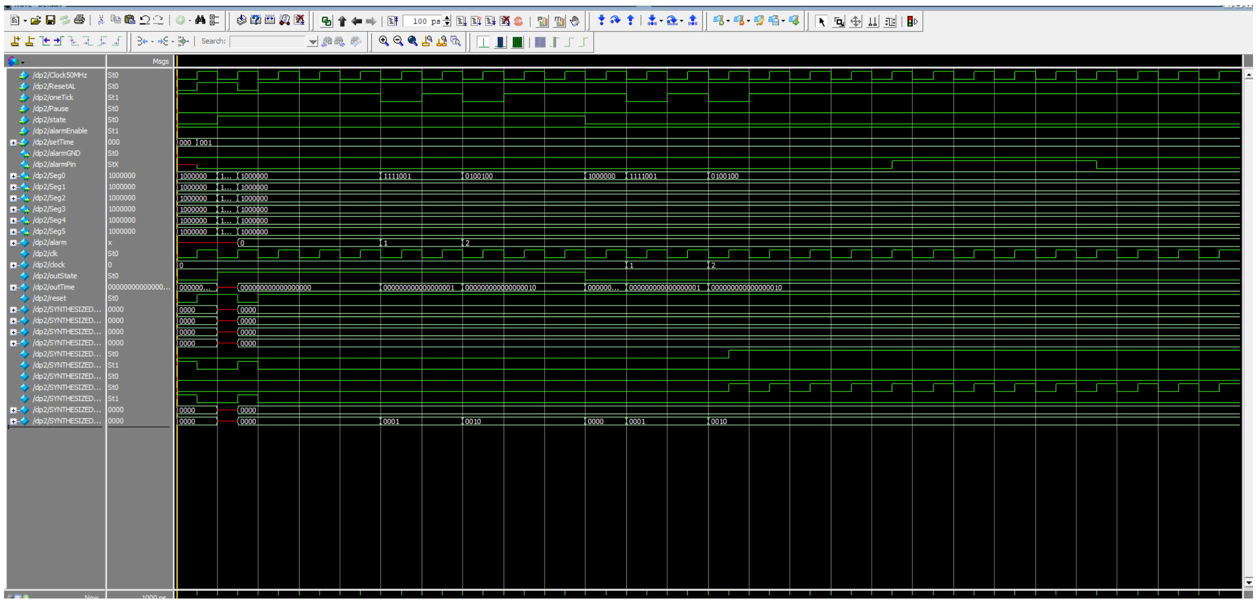


Figure 8: first 100 seconds results

5 Implementation and Video

The DE10 FPGA that was provided in ECE 272 was used to implement the final project. A breadboard was used to wire the output of the buzzer wave to an actual piezoelectric buzzer. The video can be found here: https://media.oregonstate.edu/media/t/1_2pao9m75

6 Reflection

Due to the many different moving parts of this project, it was easily one of the most challenging assignments of this course. Despite this, we were very successful in designing and executing our project.

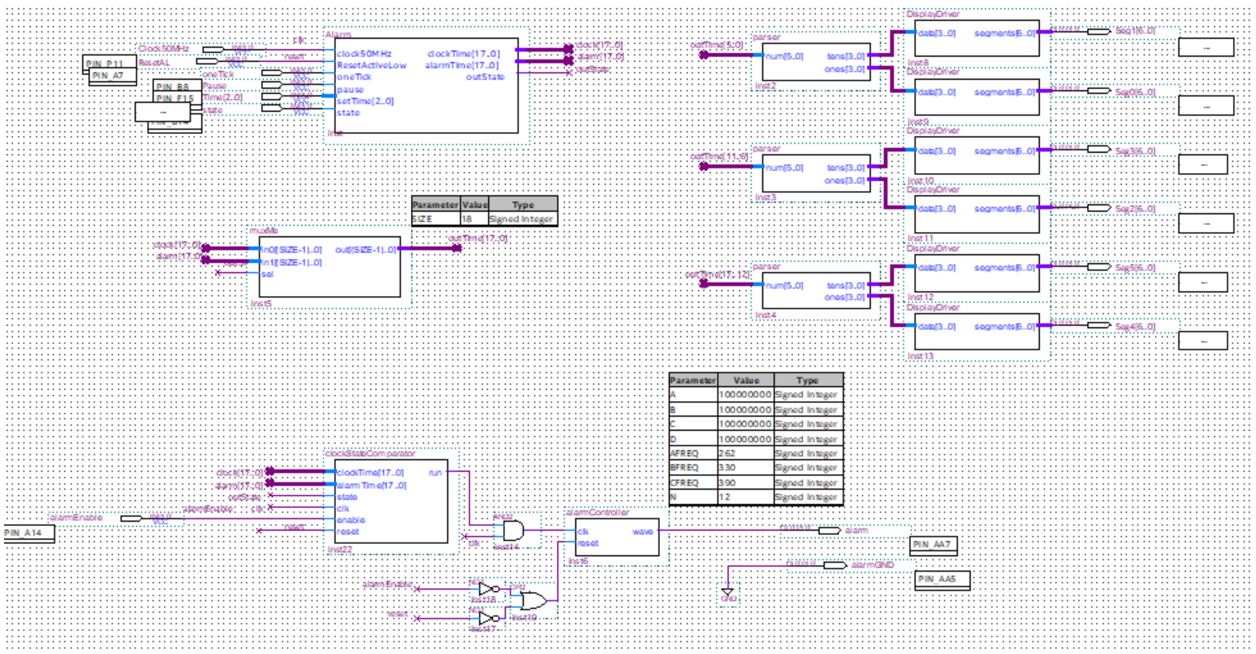
We used the clock from ECE 272 Lab 5 as a starting point for implementing this project. This gave us a starting point in addition to the counter and sync given from the textbook [1]. We then scaled the project to include a state machine for setting the alarm in addition to the clock time. Afterward, we completed the alarm functionality and allowed the state machine to control the display of the system as well. Finally, we developed the buzzer features and then implemented them on an FPGA. Our group had access to a breadboard and buzzer as well leading to the buzzer functionality on the device being fully operational.

Nothing went critically wrong during the development of our project and our team was very effective. Discord proved to be a valuable resource, as the instant messaging and file-sharing it provided allowed us to work at all times. We also became very avid Latex enjoyers as it became very convenient for scientific writing.

References

[1] David Money Harris, Sarah L. Harris (2013) *Digital Design and Computer Architecture*, Morgan Kaufmann international

A Schematics



Parameter	Value	Type
SIZE	18	Signed Integer

Parameter	Value	Type
A	100000000	Signed Integer
B	100000000	Signed Integer
C	100000000	Signed Integer
D	100000000	Signed Integer
AFREQ	262	Signed Integer
BFREQ	330	Signed Integer
CFREQ	390	Signed Integer
N	12	Signed Integer

Figure 9: Quartas Schematic of Highest Level

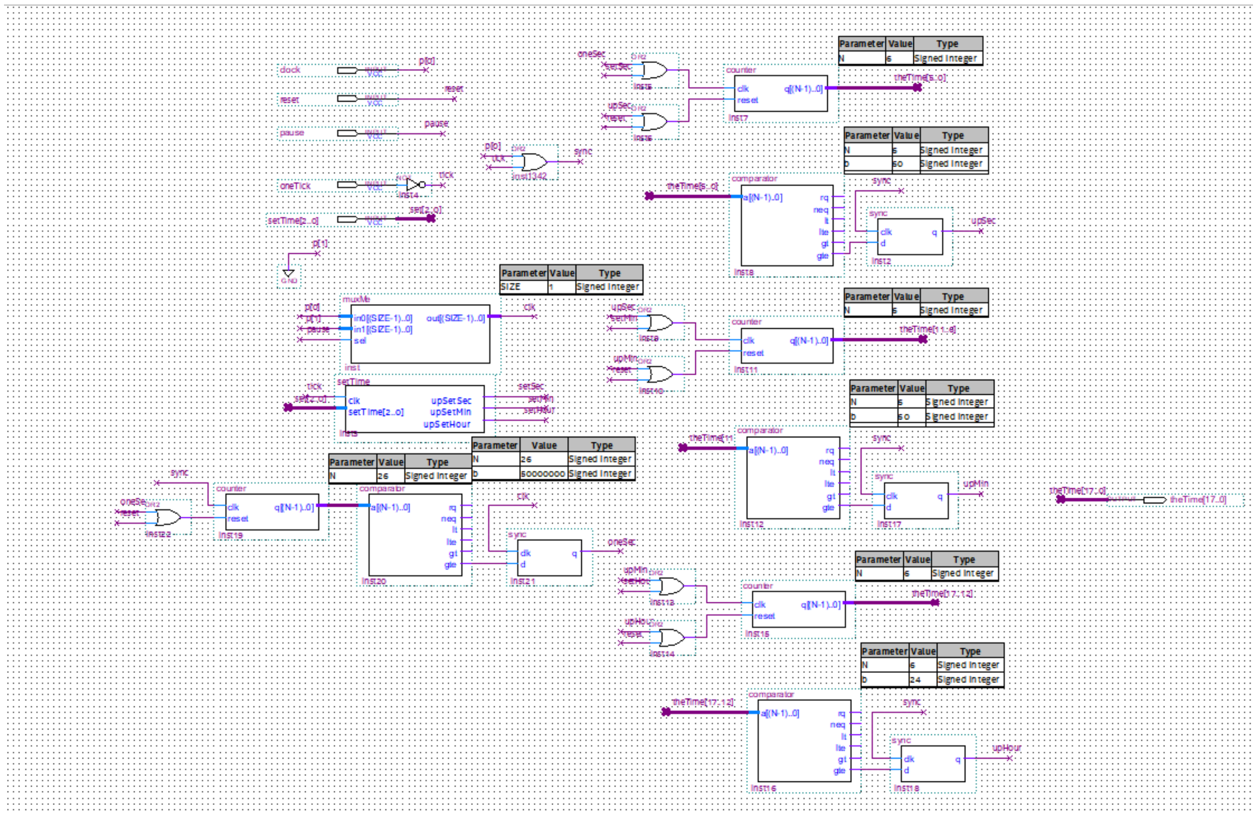


Figure 10: Quartas Schematic of Clock

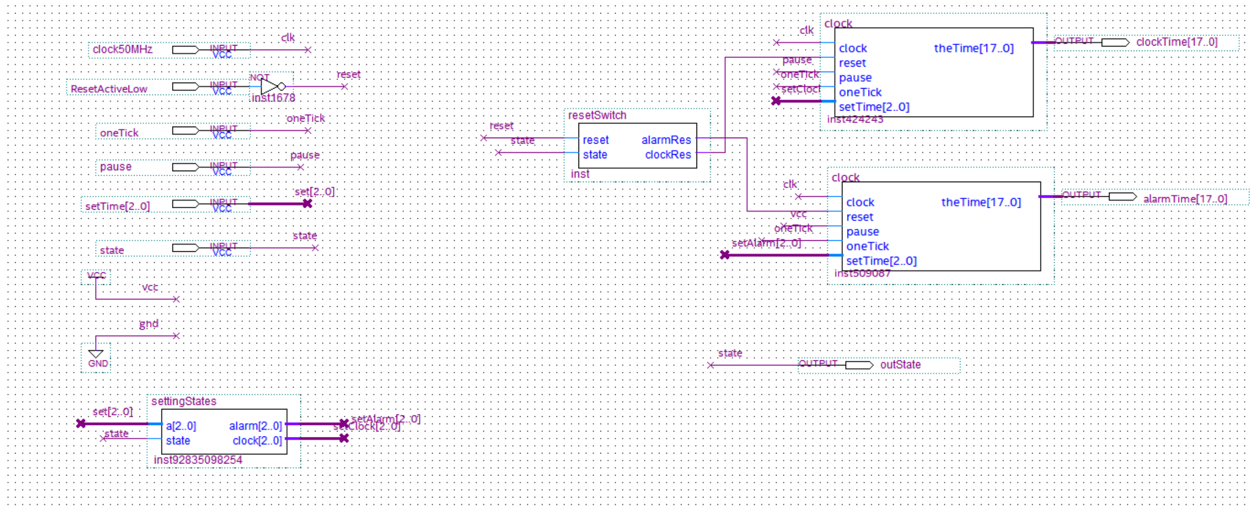


Figure 11: Quartas Schematic of Alarm

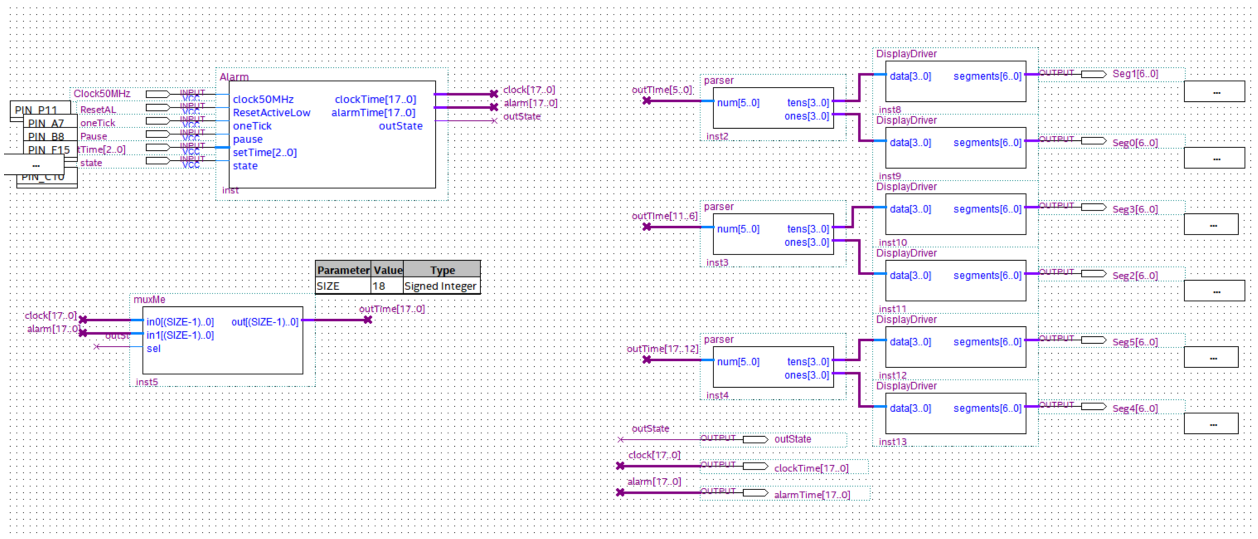


Figure 12: Quartas Schematic of Display Driver

B System Verilog Code

B.1 Counter

```
module counter # (parameter N= 6)
 (input logic clk,
  input logic reset,
  output logic [N-1:0] q);

 always_ff @(posedge clk, posedge reset)
  if(reset) q <= 0;
  else   q <= q + 1;
endmodule
```

B.2 Comparator

```
module comparator #(parameter N = 6, parameter b = 60)
    (input logic [N-1:0] a,
     output logic rq, neq, lt, lte, gt, gte);

     assign eq = (a == b);
     assign neq = (a != b);
     assign lt = (a < b);
     assign lte = (a <= b);
     assign gt = (a > b);
     assign gte = (a >= b);
endmodule
```

B.3 setTime

```
module setTime(
    clk, setTime, upSetSec, upSetMin, upSetHour
    );

    input wire clk;
    input wire [2:0] setTime;
    output wire upSetSec;
    output wire upSetMin;
    output wire upSetHour;

    wire [2:0] set;

    assign upSetSec = clk & set[0];
    assign upSetMin = clk & set[1];
    assign upSetHour = clk & set[2];
    assign set = setTime;

endmodule
```

B.4 settingState

```
module settingStates (
    input logic [2:0] a,      // Determine sec, min, hour setting
    input logic state,        // state signal
    output logic [2:0] alarm, // letting alarm get set
    output logic [2:0] clock  // letting clock
);
    always_comb begin
        // Default assignments to avoid unintended latches
        alarm = 3'b000;
        clock = 3'b000;

        if (state) begin
            alarm = a;
        end else begin
            clock = a;
        end
    end
endmodule
```

B.5 Parser

```
module parser (
    input logic [5:0] num,
    output logic [3:0] tens,
    output logic [3:0] ones
);

    assign tens = (num / 10) % 10;
    assign ones = num % 10;

endmodule
```

B.6 SevenSegDisplayDriver

```
module DisplayDriver(input logic [3:0] data,
      output logic [6:0] segments);
 always_comb
  case(data)
  0: segments=7'b100_0000;
  1: segments=7'b111_1001;
  2: segments=7'b010_0100;
  3: segments=7'b011_0000;
  4: segments=7'b001_1001;
  5: segments=7'b001_0010;
  6: segments=7'b000_0010;
  7: segments=7'b111_1000;
  8: segments=7'b000_0000;
  9: segments=7'b001_1000;
  default: segments=7'b111_1111;
 endcase
endmodule
```

B.7 clockStateComparator

```
module clockStateComparator(
  input logic clockTime[17:0],
  input logic alarmTime[17:0],
  input logic state,
  input logic clk,
  input logic enable,
  input logic reset,
  output logic run
);

  reg atTime;

  always_ff @(posedge clk  or negedge reset) begin
    if (~reset)
    atTime <= 0;
        else if (~enable) begin
            atTime <= 0;
        end else if  (clockTime == alarmTime && ~state) begin
            atTime <= 1;
        end
    end


 assign run = atTime & enable;


endmodule
```

B.8 waveGenerator

```
module waveGenerator#(
    parameter FREQ = 8
)
(
    input logic clk,
    input logic reset,
    output logic sig
);

    localparam integer HALF_PERIOD = 50_000_000 / (2 * FREQ);

    logic [31:0] counter;

    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            counter <= 0;
            sig <= 0;
        end else if (counter >= HALF_PERIOD - 1) begin
            counter <= 0;
            sig <= ~sig;
        end else begin
            counter <= counter + 1;
        end
    end

endmodule
```

B.9 alarmController

```
module alarmController #(
    parameter int A = 100000000,
    parameter int B = 100000000,
    parameter int C = 100000000,
    parameter int D = 100000000,
  parameter int AFREQ = 262,
  parameter int BFREQ = 330,
  parameter int CFREQ = 390,
    parameter int N = 12
) (
    input logic clk,
    input logic reset,
    output logic wave
);

    logic [3:0] state;

    logic aEnable, bEnable, cEnable, dEnable;

    assign aEnable = (state == 4'b0001);
    assign bEnable = (state == 4'b0010);
    assign cEnable = (state == 4'b0100);
    assign dEnable = (state == 4'b1000);

    always_ff @(posedge clk or posedge reset) begin
        if (reset)
            state <= 4'b0001;
        else begin
            case (state)
                4'b0001: if (aS) state <= 4'b0010;
                4'b0010: if (bS) state <= 4'b0100;
                4'b0100: if (cS) state <= 4'b1000;
                4'b1000: if (dS) state <= 4'b0001;
                default: state <= 4'b0001;
            endcase
        end
    end

    wire [N-1:0] aData, bData, cData, dData;
    wire aPresync, bPresync, cPresync, dPresync;
    wire aS, bS, cS, dS;

    counter #(.N(N)) aCounter (
        .clk(clk & aEnable),
        .reset(reset | ~aEnable),
        .q(aData)
    );
```

```
comparator #(.b(A - 1), .N(N)) aComparator (
    .a(aData),
    .gte(aPresync)
);
sync aSync (
    .clk(clk),
    .d(aPresync),
    .q(aS)
);

counter #(.N(N)) bCounter (
    .clk(clk & bEnable),
    .reset(reset | ~bEnable),
    .q(bData)
);
comparator #(.b(B - 1), .N(N)) bComparator (
    .a(bData),
    .gte(bPresync)
);
sync bSync (
    .clk(clk),
    .d(bPresync),
    .q(bS)
);

counter #(.N(N)) cCounter (
    .clk(clk & cEnable),
    .reset(reset | ~cEnable),
    .q(cData)
);
comparator #(.b(C - 1), .N(N)) cComparator (
    .a(cData),
    .gte(cPresync)
);
sync cSync (
    .clk(clk),
    .d(cPresync),
    .q(cS)
);

counter #(.N(N)) dCounter (
    .clk(clk & dEnable),
    .reset(reset | ~dEnable),
    .q(dData)
);
comparator #(.b(D - 1), .N(N)) dComparator (
    .a(dData),
    .gte(dPresync)
);
sync dSync (
    .clk(clk),
    .d(dPresync),
    .q(dS)
);
```

```
    wire cNote, eNote, gNote, resetNote;

    waveGenerator #(.FREQ(AFREQ)) aNoteWave (
        .clk(clk & aEnable),
        .reset(~aEnable | reset),
        .sig(cNote)
    );

    waveGenerator #(.FREQ(BFREQ)) bNoteWave (
        .clk(clk & bEnable),
        .reset(~bEnable | reset),
        .sig(eNote)
    );

    waveGenerator #(.FREQ(CFREQ)) cNoteWave (
        .clk(clk & cEnable),
        .reset(~cEnable | reset),
        .sig(gNote)
    );

    waveGenerator #(.FREQ(1)) resetWave (
        .clk(clk),
        .reset(~reset),
        .sig(resetNote)
    );


    assign wave = reset ? resetNote :
                  (aEnable ? cNote :
                  (bEnable ? eNote :
                  (cEnable ? gNote : 1'b0)));

endmodule
```

C Do Files

C.1 waveDO

```
vsim -gui work.waveGenerator
add wave *

force reset 1 @ 0
force reset 0 @ 1

force clk 0 @ 0, 1 @ 1 -r 2

run 1000
```

C.2 acDO

```
vsim -gui work.alarmController
add wave *

force reset 1 @ 0
force reset 0 @ 1

force clk 0 @ 0, 1 @ 1 -r 2


run 10000
```

C.3 clockDO

```
vsim -gui work.clock
add wave *

force reset 1 @ 0
force reset 0 @ 1

force clock 0 @ 0, 1 @ 1 -r 2
force tick 0 @ 0, 1 @ 1 -r 2
force pause 0 @ 0

force setTime[0] 1 @ 1
force setTime[1] 0 @ 0


force setTime[1] 0 @ 0

force setTime[1] 1 @ 50
force setTime[1] 0 @ 52

force setTime[2] 0 @ 0

force setTime[2] 1 @ 98
force setTime[2] 0 @ 100

run 1000
```

C.4 dp2DO

```
vsim -gui work.dp2
add wave *


force ResetAL 0 @ 0
force ResetAL 1 @ 1

force Clock50MHz 0 @ 0, 1 @ 1 -r 2
force oneTick 0 @ 0
force Pause 0 @ 0

force state 0 @ 0
force alarmEnable 1 @ 0

force oneTick 1 @ 0

force setTime[0] 0 @ 0
force setTime[0] 1 @ 1

force setTime[1] 0 @ 0
force setTime[2] 0 @ 0


# Set Alarm
force state 1 @ 2

force ResetAL 0 @ 3
force ResetAL 1 @ 4

force oneTick 0 @ 10
force oneTick 1 @ 12

force oneTick 0 @ 14
force oneTick 1 @ 16

force state 0 @ 20

force oneTick 0 @ 22
force oneTick 1 @ 24

force oneTick 0 @ 26
force oneTick 1 @ 28



run 1000
```