LAB 5

Keyboard and Audio Modules

Team 6

鄭聰明 **Jason Theodorus Pratama (109006236)**

鐵特德 **Tuguldur Tserenbaljir (109006271)**

Table of Contents

End of Table of Contents

I. Sliding Window Sequence Detector

a. Design

From the explanation of our professor, this is just like any other mealy machine, but this one is overlapping. So how to detect the overlap? In a nutshell, If you look at **Figure 1.1 S5's** state is currently on **"11100"**, if the input is 1 it goes to **S6** which is **"111001"** because if there are repeating **"01"**s on the input, the mealy machine will still continue on its sequence and will not break, to implement this, when the input on **S6** is **"0', it will go back to S5**, where this is as we so call

"**sliding window**", and will continue on a loop until on **S6** the input is 1, where it will continue its sequence to **S7** and so on.

While the rest of the circuit just follows the remaining sequence like in **Figure 1.1**

Here is the sequential circuit of the **Sliding Window Sequence Detector**

```
always @(posedge clk) begin
   if (rst_n == 1'b0) begin
      decc <= 1'b0;
      state <= S0;
   end
   else begin
      state <= next_state;
   end
end
```

And here is the Combinational Circuit of **Sliding Window Sequence Detector**

```
always @(*) begin
   case(state)
   S0: begin // IDLE
      if (in) begin
         next_state = S1;
         decc = 1'b0;
      end
      else begin
         next_state = S0;
         decc = 1'b0;
      end
   End
//// …. S1 → S7
    S8: begin
```

```
      if (in) begin
         next_state = S3;
         decc = 1'b0;
      end
      else begin
         next_state = S4;
         decc = 1'b0;
      end
   end
   endcase
end
```

b. Block Diagram & State Transition
   Where A/B
   A = in
   B = dec
   And Idle = Empty

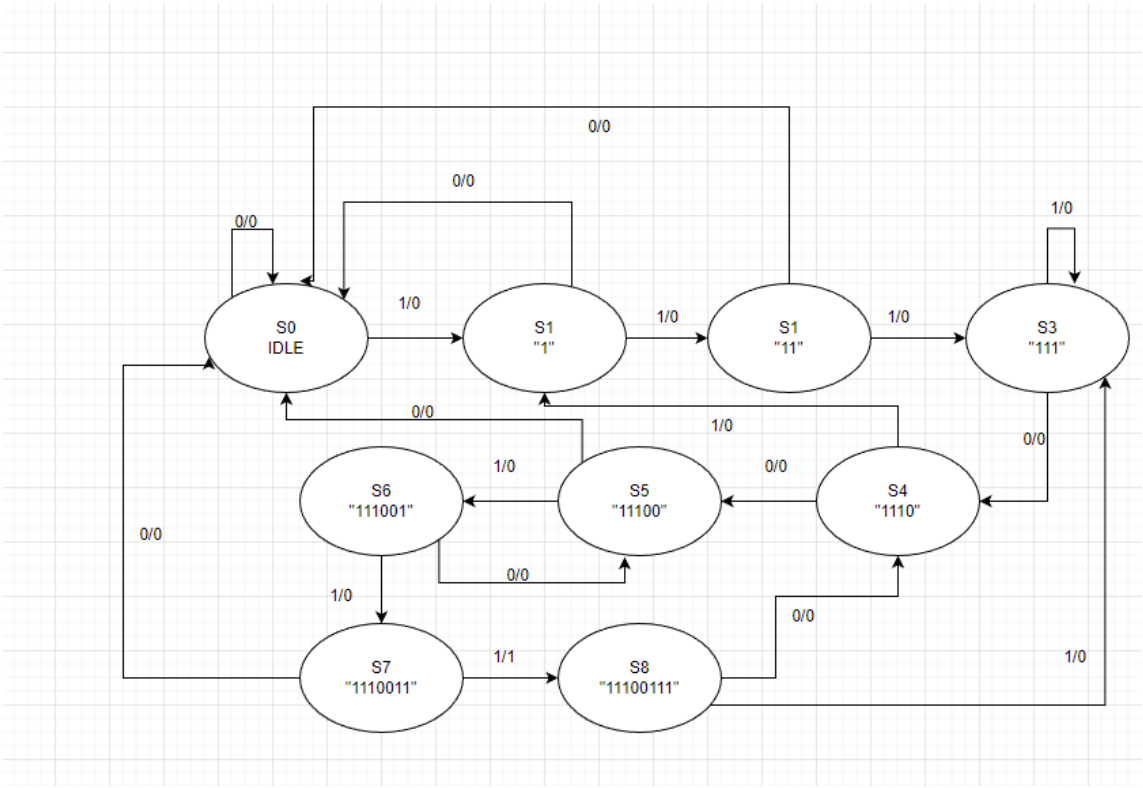


Figure 1.1 State Transition Diagram

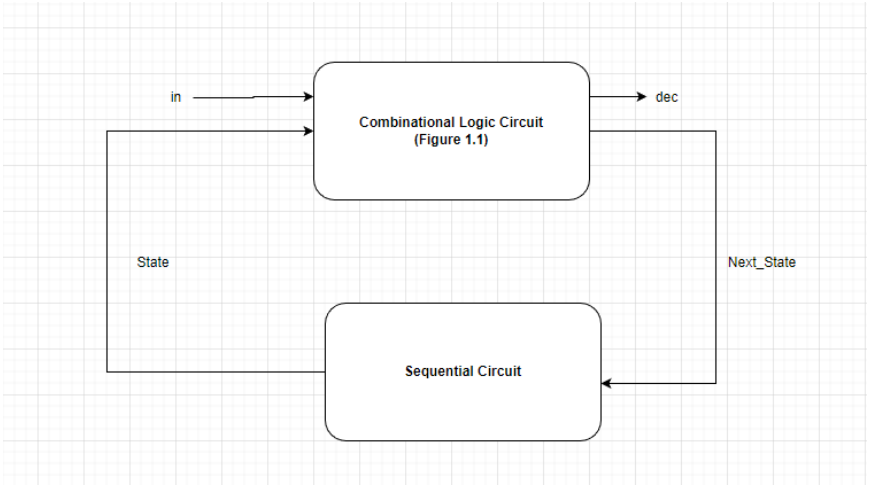


Figure 1.2 Block Diagram

c. Waveform & Testing

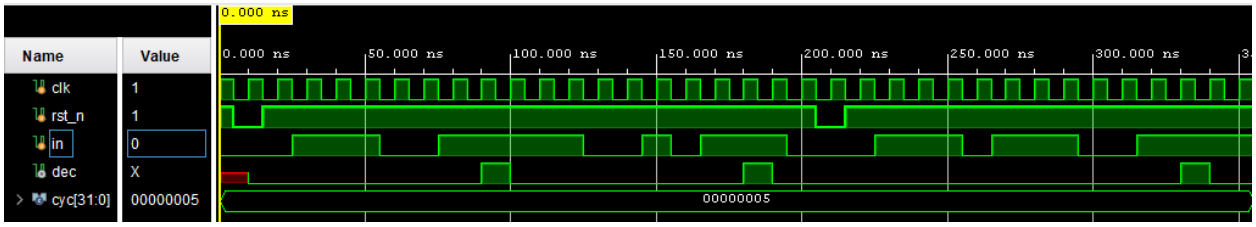


Figure 1.3 Waveform of Sliding Window Sequence Detector

For the testing of the inputs the input the sequence given from the slides which is

“1110011110010111”

And then after the 2nd rst_n the input is

“1110111001111”

Where after the 2nd rst_n is used to check overlap after a mismatch case, which in this case in **Figure 1.3** the dec is correct.

The **Testbench** would be like the following :

```
@(negedge clk) rst_n = 1'b0;
@(negedge clk) rst_n = 1'b1;
//Given Waveform
@(negedge clk) in = 1'b1;
@(negedge clk) in = 1'b1;
@(negedge clk) in = 1'b1;
@(negedge clk) in = 1'b0;
@(negedge clk) in = 1'b0;
@(negedge clk) in = 1'b1;
@(negedge clk) in = 1'b1;
//Sequence dec == 1;
…….. And so on
```

II. Traffic Light Controller

a. Design

For this advanced question, we are to create a traffic controller for a highway and a local road. Right away the Finite State Diagram is given to us with 6 states. The states are green, yellow, and red lights all alternating with each other respective to their highway and local road whereas the lights are stored as **3'b001** (green), **3'b010**(yellow), and **3'b100**(red)

Firstly, we decided to finish the unfished Finite State Diagram before anything else. From the first and initial state of the highway being green and the local road red, it only switches to the next state when only the local road detects a car, and the clock cycles for at least 70 cycles. If not, it keeps cycling in the first state. With this logic excluding the fact of whether local roads of having a car or not, we implement the same logic in other states as well (**Figure 2.2**). We do not switch states when there is no car detected on the local road to have higher priority on the highway. Thus, once the full cycle completes and the state is back to the initial state **S0** then we check if there is a car on the local road.

```
S0:begin // hw green, lr red, atleast 70 clock
  if(count >= 69 && lr_has_car == 1'b1)begin
    // cycle until there is car in local road
    next_count = 0;
```

```
            next_state = S1;
            next_hw_light = 3'b010;
        end else begin
            next_count = count + 1;
            next_state = S0;
            next_hw_light = 3'b001;
        end
    end
```

After we finish our Finite State Diagram, we now got a good idea of how our programming should be structured. To start our program design, we set the values of the states **S0** to **S5 (3'b000** to **3'b101)**. In addition to higher priority of the highway being green, we set the initial state of the highway to green and local road to red. If the reset is 1'b0, our design will start at **S0** state. On the other hand, if the reset is 1'b1 we set the values of state, count, highway light, local road light to the next on the positive edge of the clock. During our thinking phase, we noticed that until one light turns green, the other light to be green will take atleast 3 states. For example, from state **S0** to **S2** (**Figure 2.2**).

b. Block Diagram & State Transition

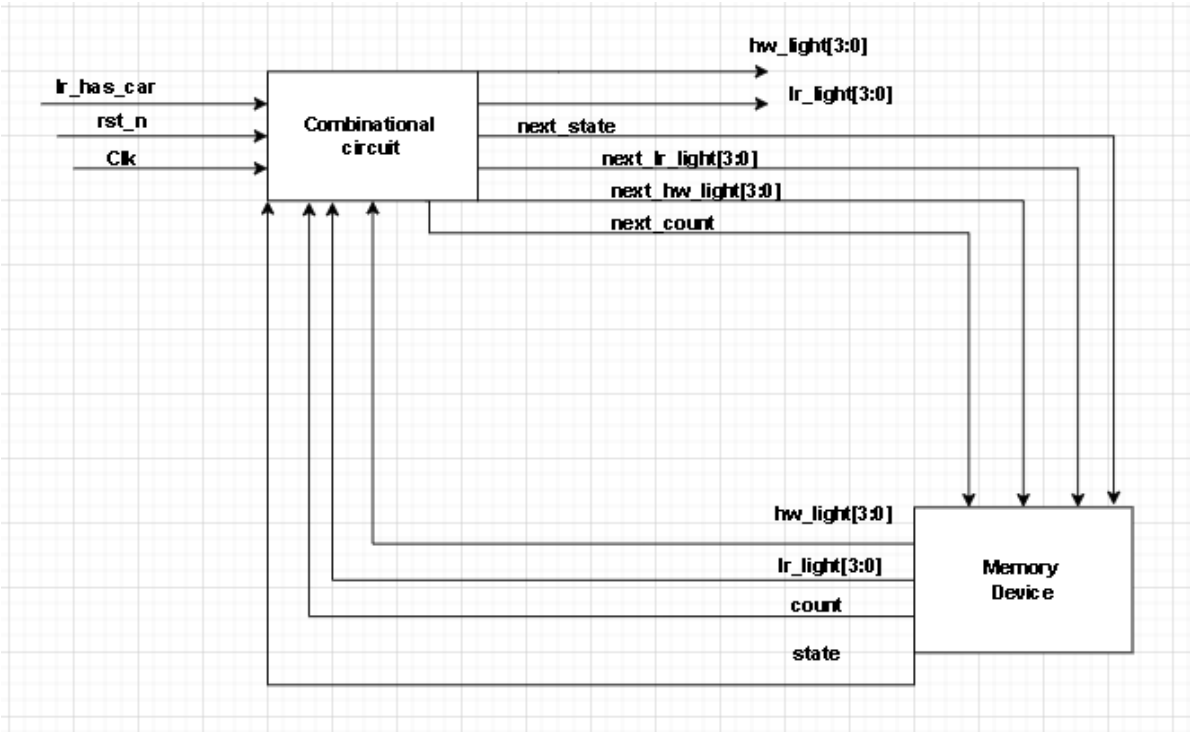


Figure 2.1 Block Diagram

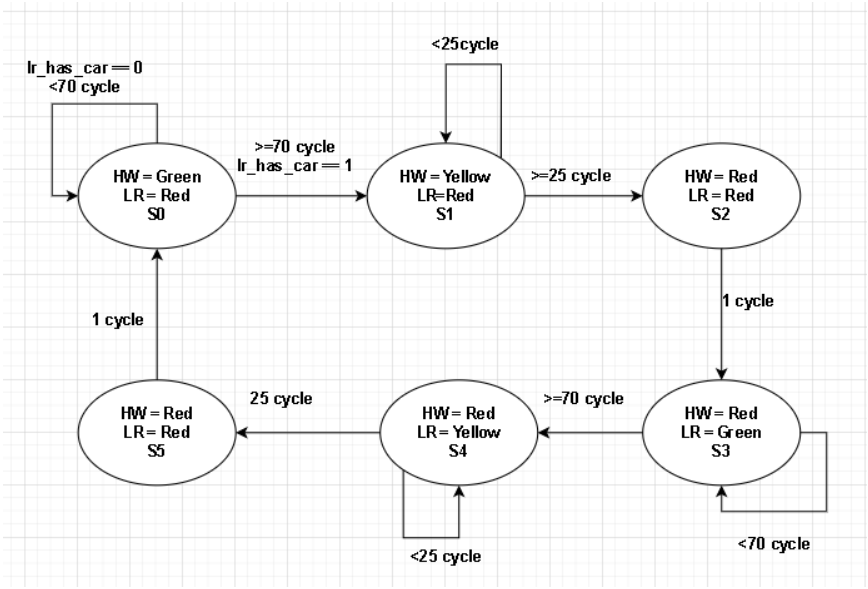
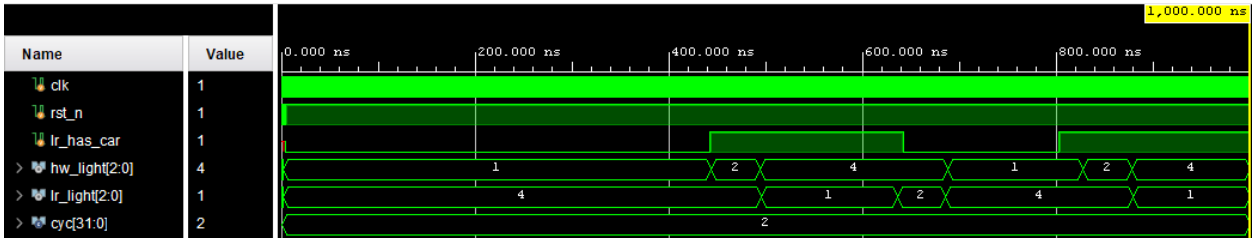


Figure 2.2 State Diagram

c. Waveform & Testing



The clock looks like a green line because we have many clock cycles for the lights to switch between cycles. We first start our test by setting **rst_n** to **1'b0** where it will stay in the initial state, and it was success. To continue our testing we set the **rst_n** to **1'b1** where the value of **lr_has_car** to **1'b0**, and we know that highway light should be green whereas the local road light should be red. To add, highway light stays green until there is a car detected on the local road. As we can see around the **400 ns** mark, we know it is going along our state diagram that we designed above. Then when highway light outputs **4** and local road outputs **1** we know that we can in the **S3** state where local road is green and highway is red (**1 == green, 2 == yellow, 4 ==red**).

After we see that one full cycle of highway light going green, yellow,red,red,red and then green in the initial state, we can say our implemented logic ran successfully. Thus, we test with another car in the local road right after highway light turns green. As we can see from the Waveform it works successfully.

Finally, we double check our wave to the Traffic light controller example timing diagram, given in the Lab5 slides.

III. Greatest Common Divisor

a. Design

Since we can not use loop modules and percentage division to calculate the Greatest Common Divisor, We researched from the internet and we learned that we should implement euclidean algorithm in this advanced question. To find the greatest common divisor, we get the difference positive difference of the smaller and larger number. The difference can not be negative since the inputs are not signed. So we should follow the logic shown below until one of them reaches zero.

```
if(reg_a != reg_b)begin
    if(reg_a > reg_b)begin
        reg_a <= reg_a - reg_b;
        reg_b <= reg_b;
    end else begin
        reg_b <= reg_b - reg_a;
        reg_a <= reg_a;
```

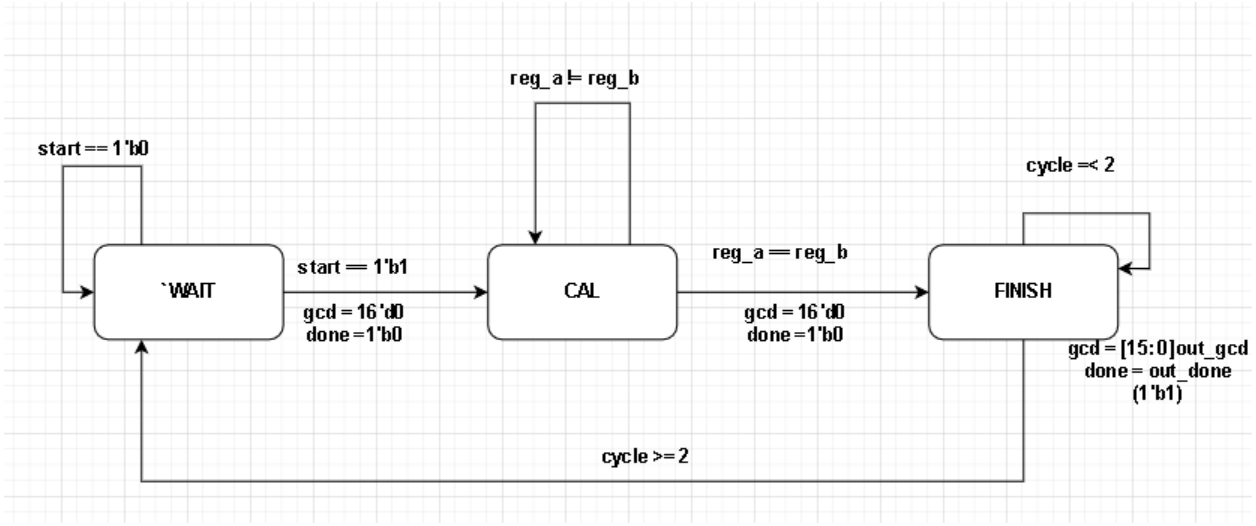
```
        end
    end
```

(Figure 3.1 euclidean algorithm implementation**)**

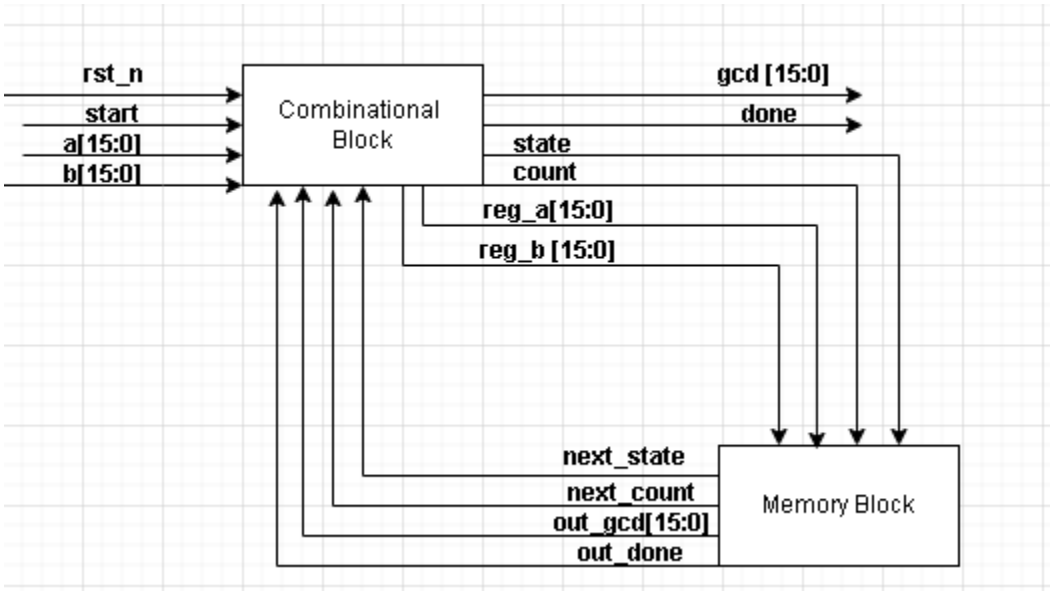
When one of then reaches zero, we switch the values and pass them as outputs. Thus, if they are equal to each other, we will just one of them. Before we started coding we complete our state transition diagram respect to the euclidean algorithm (**Figure 3.2**).

Given to us in the Lab5 slides, we will have 3 states which are **WAIT**, **CAL**, and **FINISH**. First, we iniliaze our states according to the State Transition Diagram. Next, we iniliaze our logic in the respective states. In the **WAIT** state, only when start has the value of **1'b1** we fetch and buffer our inputs and pass them into the next **CAL** state. The main logic of our **CAL** state will be the mplementation of euclidean algorithm (**Figure 3.1**) . Finally in our **FINISH** state, we count need to count the clock because in the slides the done should be outputed every 2 cycles and then back to the **WAIT** state.

b. Block Diagram & State Transition

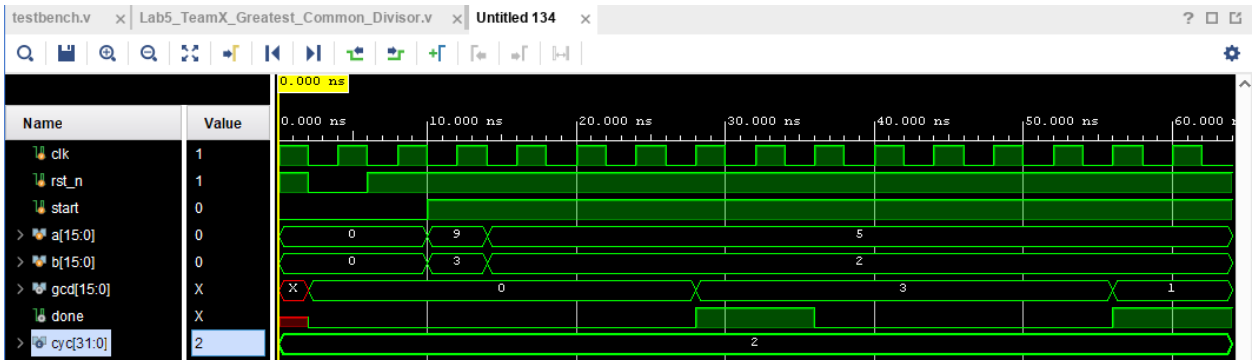


(Figure 3.2 State Transition)



(Figure 3.3 Block Diagram)

c. Waveform & Testing



At first the value of variables to fetch and buffer the inputs will be zero because the start value is **1'b0**. Thus, the current output will be **0**. Next, we test our logic by giving the pair of numbers (**9,3** ; **5,2**). The output of gcd will be **3,1** respectively and the done value should be high for **2** cycles as seen in the testbench waveform above. We used online gdc calculator to double check our answers from the waveform outputs.

IV. Booth Multiplier

a. Design.

Our idea for this advanced question was to reuse some of parts of advanced question of 3 (**GCD**) where we have 3 states (**WAIT,CAL,FINISH**). First, we complete our state transition diagram (**Figure 4.2**). Then, we read a lot

about the booth multiplier online. We found out that we should get the values of the inputs in binary where they can be negative or positive and that is why it is signed variables.

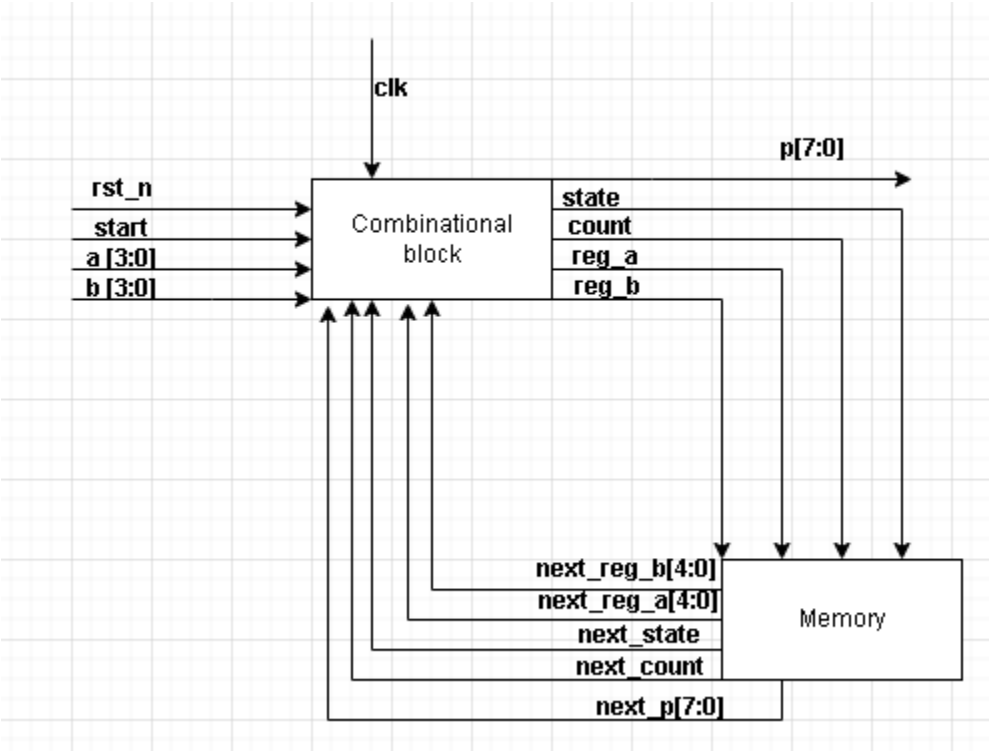
To start coding our Booth multiplier we first iniliaze the three states in the code. Our initial state will be **WAIT** where it will only transition into **CAL** if only **start** and **rst_n** variables are **1'b1**. If **rst_n** is **1'b0**, we set everything to 0 and wait in the **WAIT** state until we get the **1'b1**. Then we should get the values of the inputs in binary and buffer them, so that they get lost in state transition.

In the **CAL** state, we get negative numbers and turn them into positive numbers to have a easier time calculating by using the 2's complement and adding **1'b1**. Then we check first two **1** and **0**'s of the binary number. If the binary number starts with **10,** we get the difference of two numbers and buffer them and pass them. Else if it starts with **01,** we would add them together and do the same process. Otherwise if it is **00** or **11**, we pass into the arithmetic shift. Then we will shift 4 times. Until we shift 4 times, we will follow the this process.

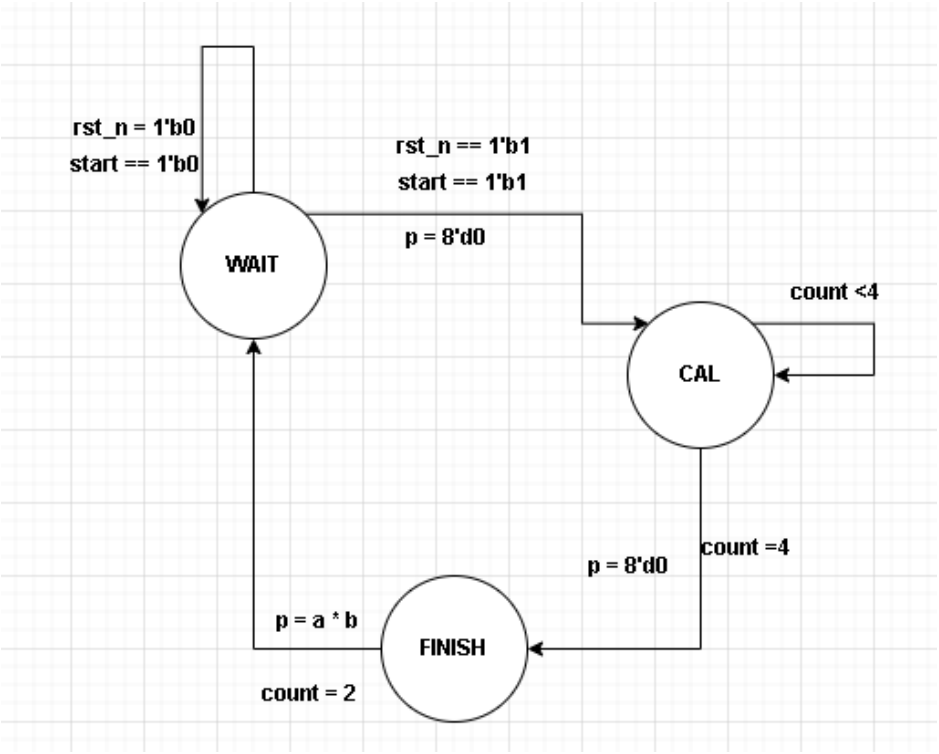
Finally, in the **FINISH** state, we go back to the initial **S0** state.

However, we were not able to debug our code in time to submit.

b. Block Diagram & State Transition



(Figure 4.1 Block Diagram)



(Figure 4.2 State Transition)

V. FPGA 1

For the 1st FPGA Question we have to implement the Audio Module, and Keyboard Module, which generate an **ascending** tone, and **descending** tone from **C4 to C8**

Most of the modules used are reused from the codes given from Keyboard Sample Code, And the Music Box.

Which the modules used are PlayerCtrl, Music, PWM_gen, KeyboardDecoder, and SevenSegment which i used to test the keyboard **inputs.**

At first, i tested my keyboard inputs which was outputting to the SevenSegment, and then i advanced to the Audio Module which i had some problems. Firstly i made my music module which states the C → A of Octave 4, which on the table would be like so

C4	NM1 = 32'd262
D4	NM2 = 32'd294
E4	NM3 = 32'd330

F4	NM4 = 32’d349
G4	NM5 = 32’d392
A4	NM6 = 32’d440
F4	NM7 = 32’d494

When these are already declared i made case statements like the Music.v in the basic lab which was Music Box like following

```
always @(*) begin
   case (ibeatNum)
      8'd0 : tone = `NM1; // Octave 4
                  //... NM7
      8'd7 : tone = `NM1 << 1; // Octave 5
                  //... NM7 << 1



      8'd14 : tone = `NM1 << 2; //Octave 6
                  //... NM7 << 2
      8'd21 : tone = `NM1 << 3; // Octave 7
                  //... NM7 << 3
      8'd28 : tone = `NM1 << 4; // Octave 8
                  //... NM7 << 4

      default : tone = `NM0;
   endcase
```

```
end
```

And for the **PWM_gen** i reused the Basic Lab code. At the Top my logic is there are 2 states of the machine which goes up and down and i made a counter for the beats so it will detect whether the musicbox has reached C8 / C4.

Which of the **pseudocode** like so

```
case(dir)
UP : begin
   if(beatlength != 8'd34) begin // Length of music is 8'd35
      next_ibeat = ibeat + 8'd1;
            Beatlength = beatlength + 1;
   end
   else begin
      next_ibeat = ibeat;
      if (been_ready && key_down[last_change] == 1'b1) begin
         if (last_change == KEY_CODE_S) begin // S
            next_dir = DOWN;
DOWN : begin
   if(beatlength != 8'd0) begin
      next_ibeat = ibeat - 8'd1;
            Beatlength = beatlength - 1;
   end
   else begin
      next_ibeat = ibeat;
      if (been_ready && key_down[last_change] == 1'b1) begin
         if (last_change == KEY_CODE_W) begin // W
            next_dir = UP;
         End
      Endcase
```

It is such a shame for us because our problem has been the ibeat wouldn't shift the music, we have been debugging this for 2 days , we changed multiple logics and rewriting the code, but nothing have worked. Although, we are sure the modules are correctly instantiated, we rechecked

the music box and Piano. I hope with this report Professor and the TAs would understand what we are trying to do. Sadly, because of the time limit we weren't able to work on the **R** button because the **W** and **S** didn't even work because of Logic Errors.

4

B. Problem

We had lots of problems in this FPGA Question, mostly about the Audio Modules, at first we thought that it was the speaker, so i directly connect the Music to play, and it was working, the keyboard was also working fine since we instantiated SevenSegment to output on the 7-Segment Display. When trying to implement the audio module it was stuck on C4 on start, after reset it was still also stuck on C4. i tried to remove all the logic and increase the ibeat no matter what conditions, and it worked. I guess it was a logic problem which needs more training and practicing. Funny thing is, this was similar to Ping Pong Counter Logic, we just realized it while writing the report. I guess we shouldve made a state transition first before writing the program.

C. Block Diagram

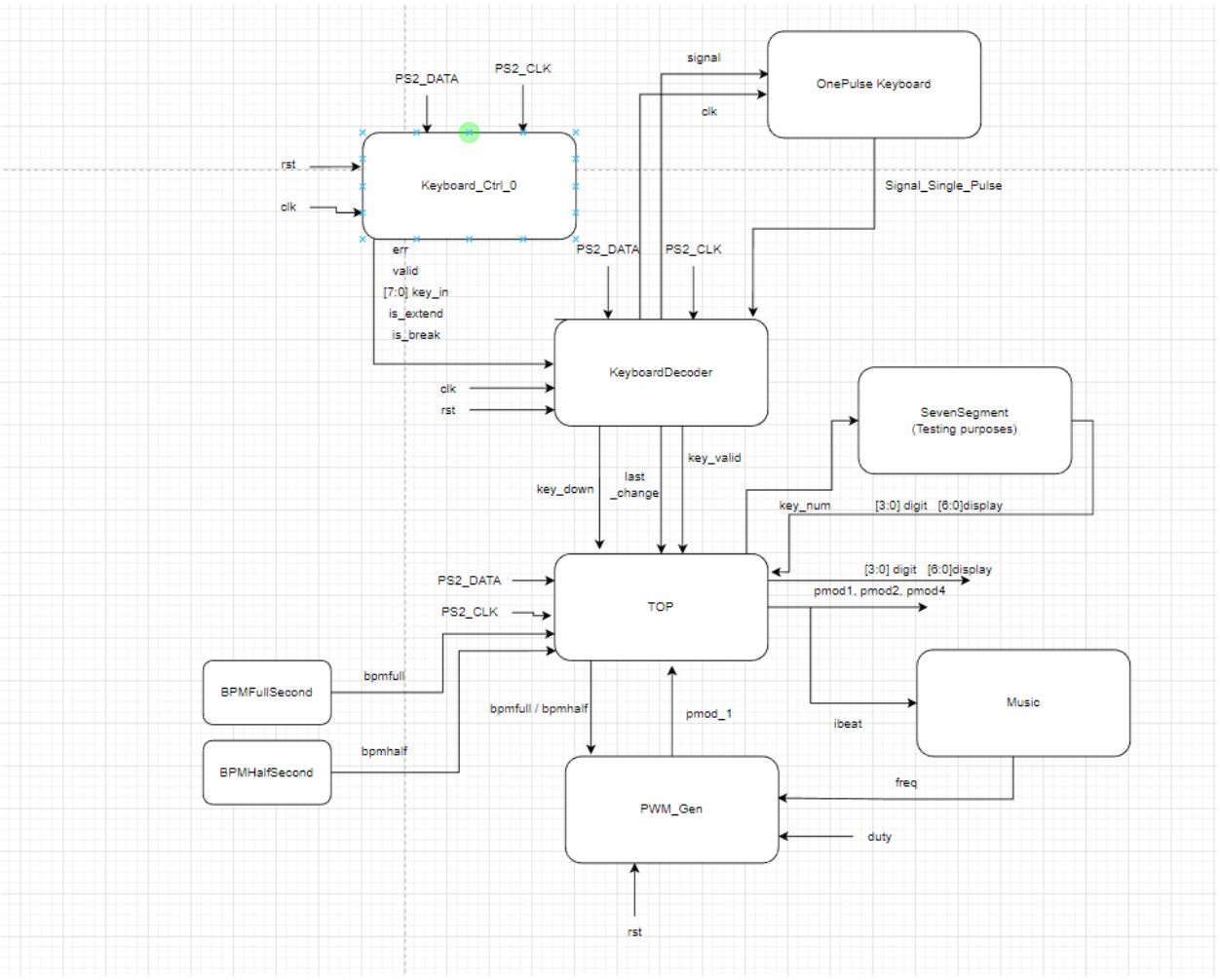


Figure 5.1 Block Diagram of FPGA 1

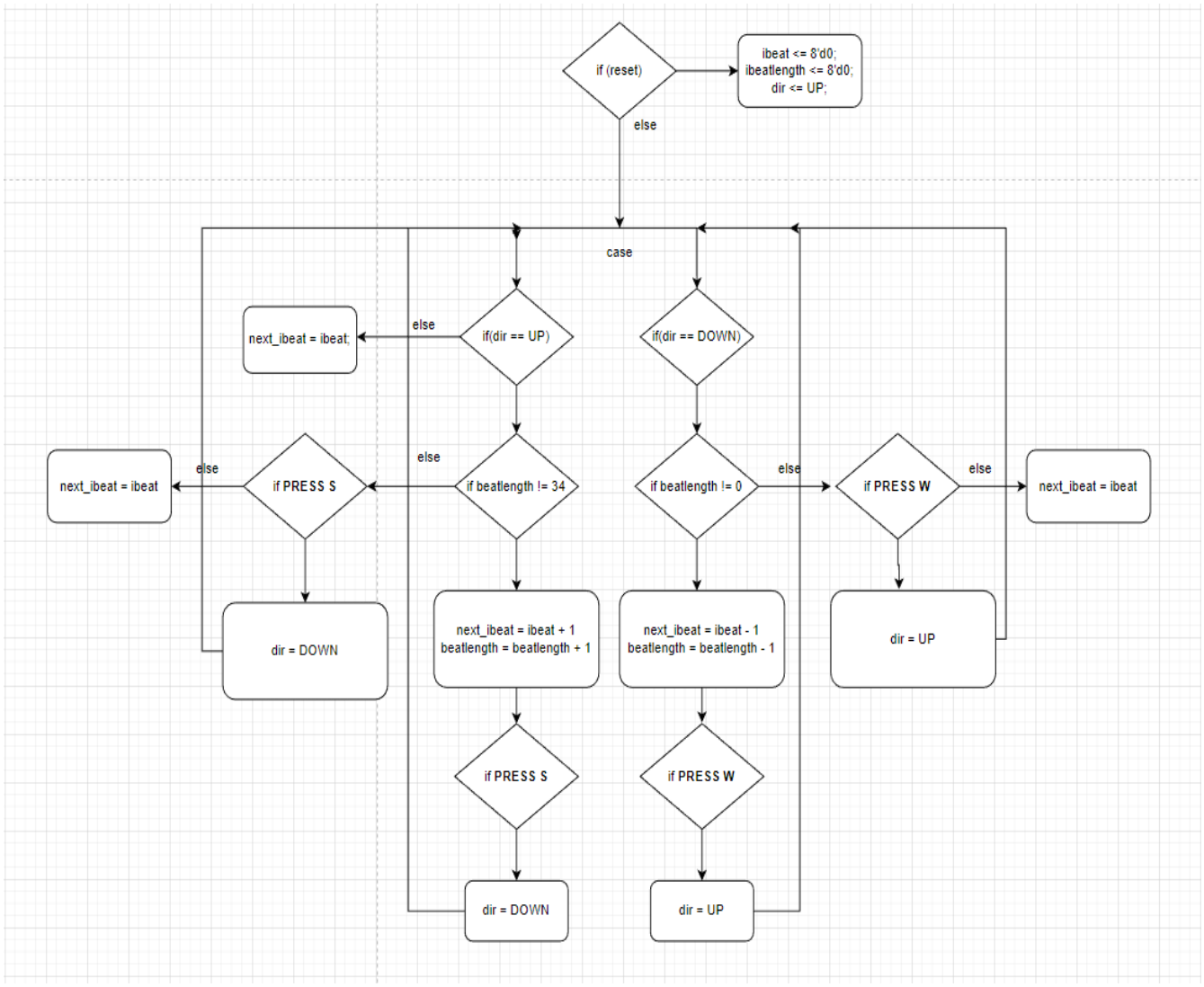


Figure 5.2 State Transition of FPGA 1

VI. FPGA 2

For FPGA2, we tried to speedrun it but sadly we failed, we were too frustrated on the FPGA1, sadly we didn't put in enough time for FPGA2. On the brightside, we have written pseudocodes before starting the FPGA Questions itself, we figure that the Vending Machine is easier, and we weren't quite sure why we didn't allocate the time to the **Vending Machine** instead of the **FPGA 1**

For the vending machine we also implemented a similar logic, where there are 2 states which is the **BUY** State and **CHANGE** state, where during the **BUY State** you can insert the coins, and

buy the things from the vending machine while on the **CHANGE** state it returns the change of $5 every second, to simulate the coins being returned.

Here is the **pseudocode** of the logic

```
always @(*) begin
   case(state)
      BUY : begin
         if (balance > 100) begin
            next_balance = 8'd100;
         else if (balance  < 101) begin
            if (buttonCancel) begin
               next_balance = balance;
               next_state = CHANGE; //return change
            else begin
               if (buttonLeft) begin
                  next_balance = balance + 8'd5; //insert 5
               if (buttonRight) begin
                  next_balance = balance + 8'd50;
               if (buttonCenter) begin
                  next_balance = balance + 8'd10;
         else begin
            if (been_ready && key_down[last_change] == 1'b1) begin
               if (last_change == KEY_CODE_A) begin // coffee
                  if(balance >= 8'd80) begin
                     next_balance = balance - COFFEE;
                  else begin
                     next_balance = balance - COFFEE;
                     next_state = BUY;
               else if (last_change == KEY_CODE_S) begin
                  if(balance >= 8'd30) begin
                     next_balance = balance - COKE;

                  end
                  else begin
                     next_balance = balance - COKE;
```

```
                next_state = BUY;
            else if (last_change == KEY_CODE_D) begin
                if (balance >= 8'd25) begin
                    next_balance = balance - OOLONG;
                else begin
                    next_balance = balance - OOLONG;
                    next_state = BUY;
            else if (last_change == KEY_CODE_F) begin
                if (balance >= 8'd20) begin
                    next_balance = balance - WATER;
                else begin
                    next_balance = balance - WATER;
                    next_state = BUY;

            else begin
                next_balance = balance;
                next_state = BUY;
    CHANGE : begin
        while (balance != 8'd0)
            next_balance = balance - 8'd5;
            if (balance == 8'd0)
                next_state = BUY;
endcase
```

Where if the balance is above 100 it will set the balance back to 100, because the max amount of the balance is 100, then if the balance is <= 100, the following table is set on what to do
Which the prices of the items are already declared on parameter
Where COFFEE is declared to 80, COKE to 30, OOLONG to 25 and WATER to 20

cancel	Change state to **CHANGE**
buttonLeft	Next_balance = balance + 8’d5
buttonRight	Next_balance = balance + 8’d50

buttonCenter	Next_balance = balance + 8'd10
rst_n	Balance <= 8'd0 State <= BUY
Keyboard A	Balance - COFFEE
Keyboard S	Balance - COKE
Keyboard D	Balance - OOLONG
Keyboard F	Balance - Water

Then the balance will go through seven segment display to be outputted
Then for the FPGA Buttons we ran it first through Debounce and Onepulse in order to get the signal for the buttons. And not forgetting the keyboard decoder for the A,S,D,F

B. Problems

As said from previously, it was kind of time problem for us, because we were too focused on the Audio and Keyboard Module. Before starting the FPGAs we have written pseudocodes to implement it further, so we decided to do FPGA 1 first, the FPGA 2 from the code we wrote worked until the initial balance display, but we didn't put in enough time to debug it.

C. Block Diagram and State Transition

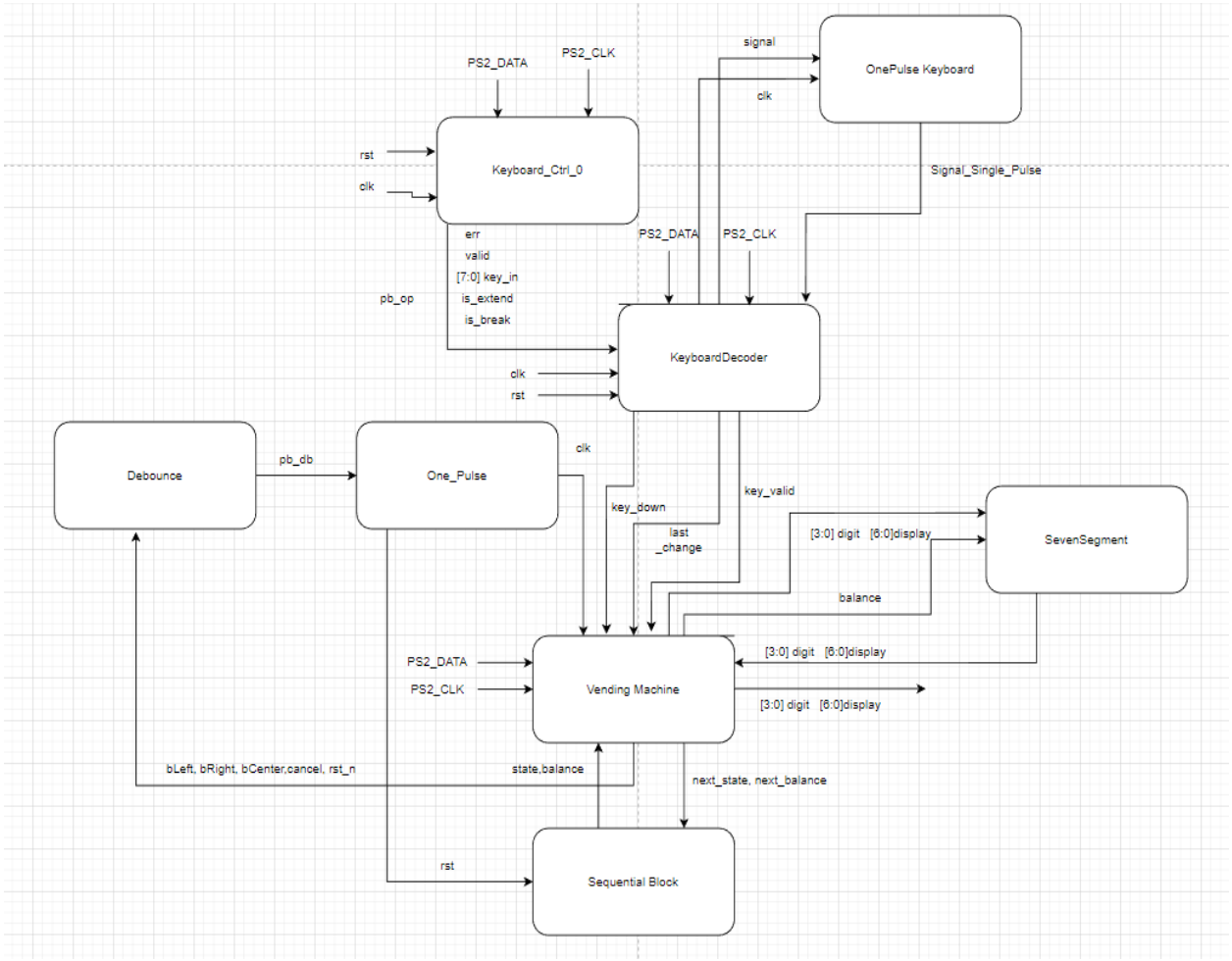


Figure 6.1 Block Diagram of Vending Machine

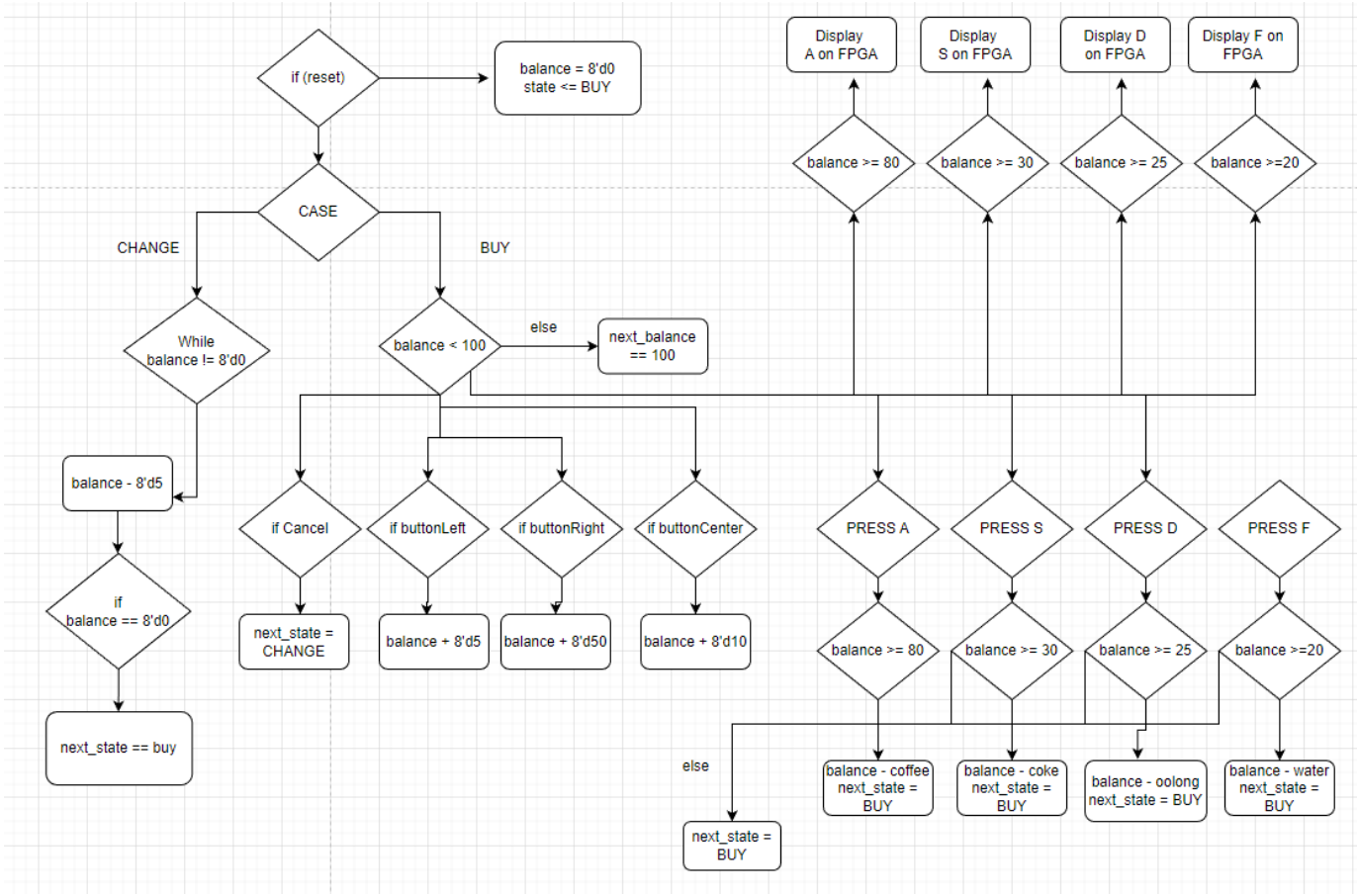


Figure 6.2 State Transition of Vending Machine

VII. Problems we faced

We had a discussion about the FPGA Questions were the ones that needs more time doing, We concluded that it was about work management, if we had put in more time to the FPGA Implemention we were sure that it would work.

There were also external problems where Jason had his Data Structure midterm on the middle of the Lab week, so he weren't able to allocate more of his time to do FPGA Problems.

VIII. What we learned from this lab

We mostly learned about Finite State Machines, and implementing them on Overlapping conditions, Traffic Light, and GCD Calculator. We learned more about signed and unsigned inputs. Thus, we learned to code more comparatively in verilog and debug more

efficiently. Learn to use keyboard , speaker, and audio modules for FPGA.

IX. Contributions

Jason was in charge of question 1, FGPA 1 and FPGA 2. While Tuguldur was in charge of question 2, 3, and 4. Jason helped Tudo to write testbench for Question 2, and debug codes for question 2 and 3. Tuguldur helped to connect the speaker to the FPGA Board for FPGA 1.