LAB 5 REPORT

Sliding window sequence detector, Traffic light controller,

Greatest common divisor, Bonus: Booth multiplier,

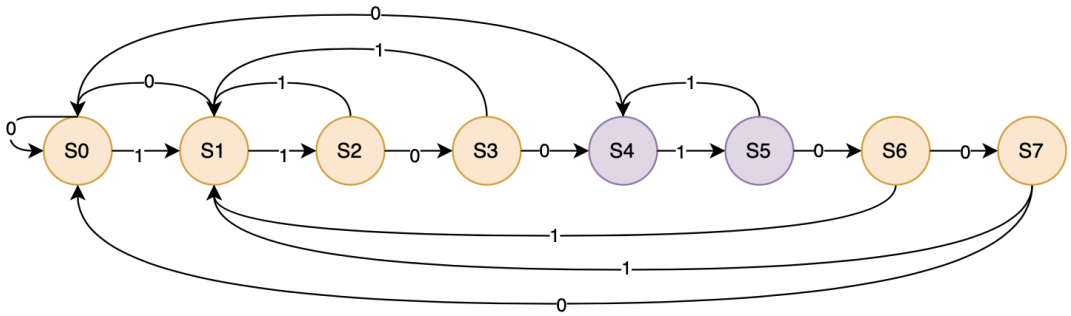
Mixed keyboard and audio module (FPGA),

Vending machine (FPGA)

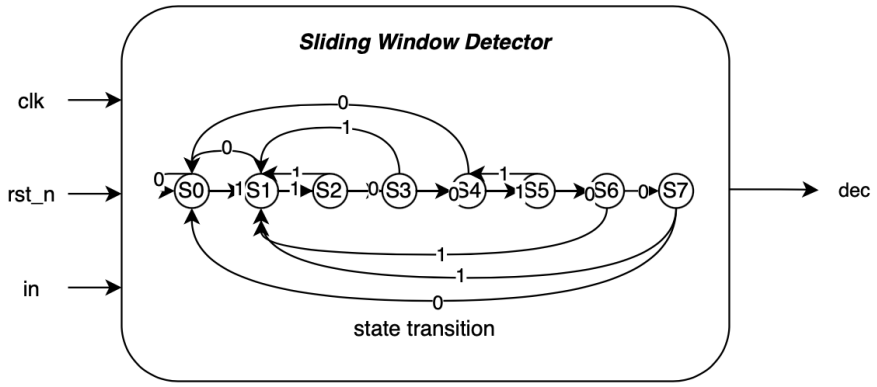
Team 37:

1. 徐美妮 Mary Madeline Nicole 109006205
2. 林之耀 Kevin Richardson Halim 109006277

I. Block diagrams and state transition diagrams

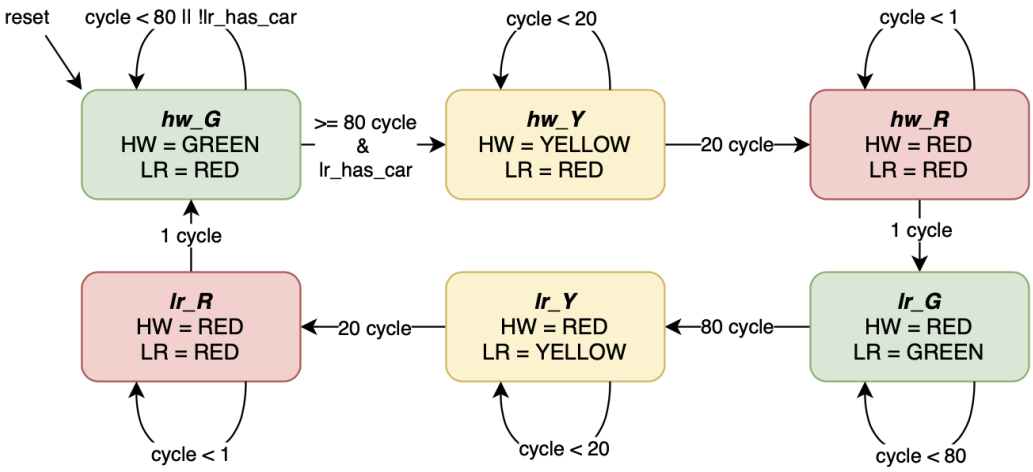


(a) State transition diagram

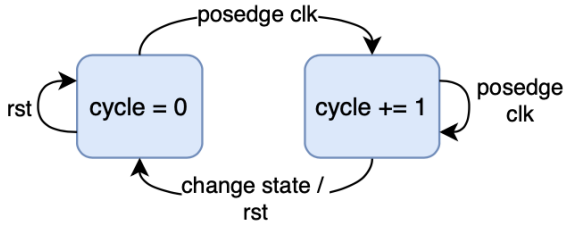


(b) Block diagram

1.1 Sliding Window Sequence Detector

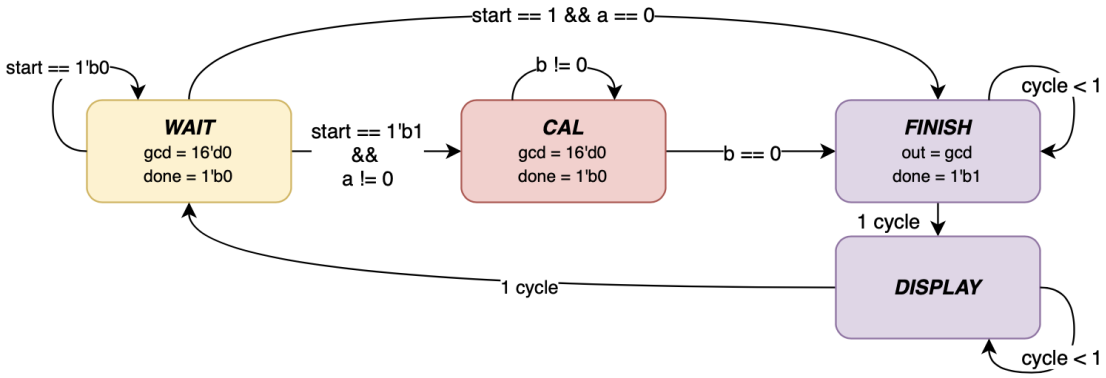


(a) State diagram for lights

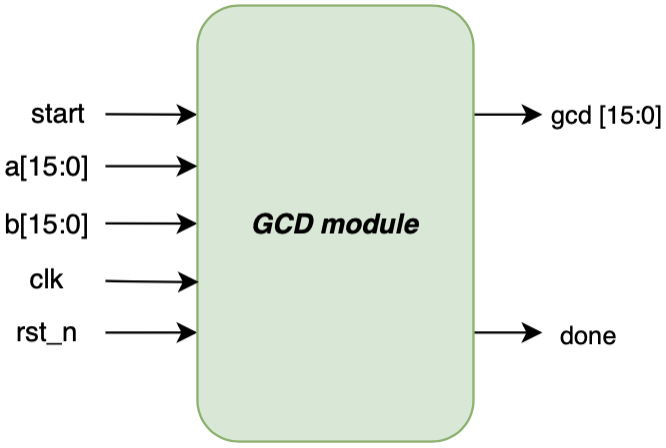


(b) State diagram for counting cycle

1.2 Traffic light controller for highway (HW) and local road (LR)

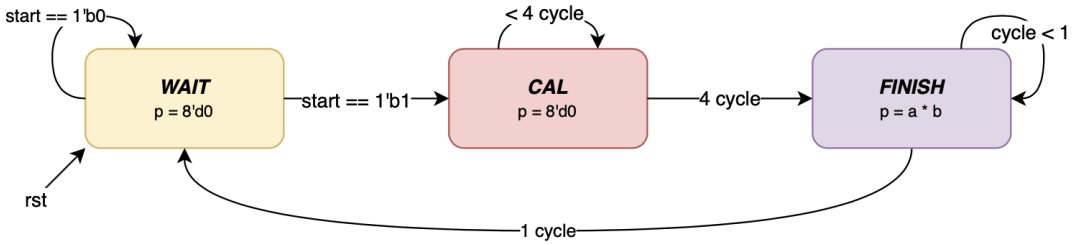


(a) State diagram

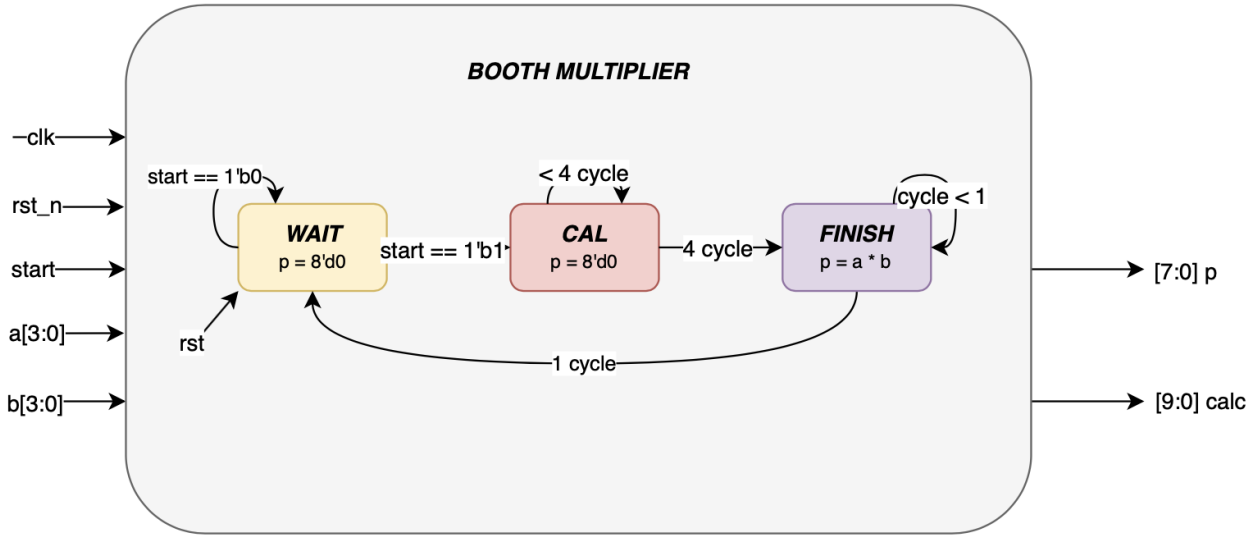


(b) Block diagram

1.3 Greatest common divisor

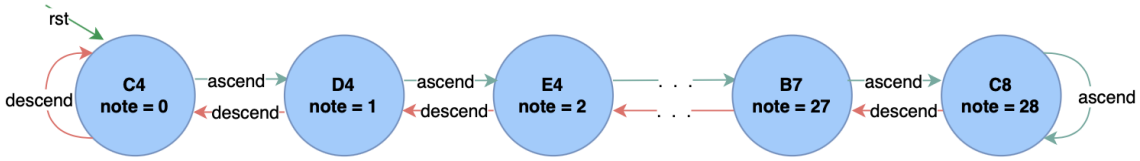


(a) State diagram

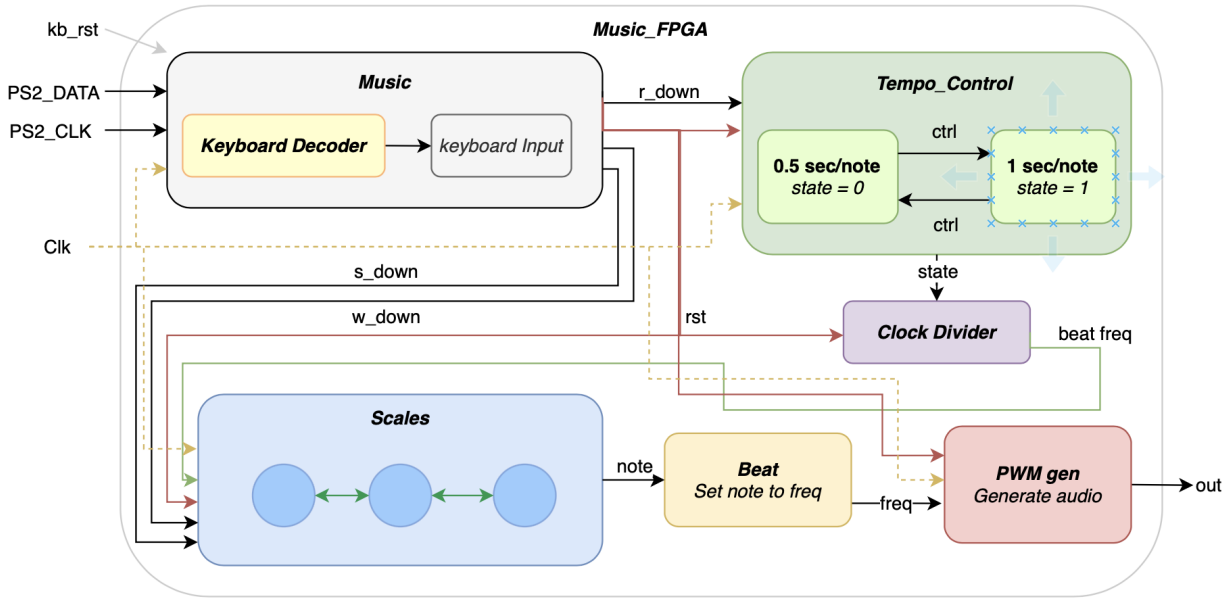


(b) Block diagram

1.4 Booth multiplier

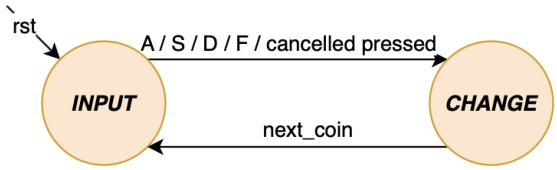


(a) State diagram of scales (C4 ← → C8)

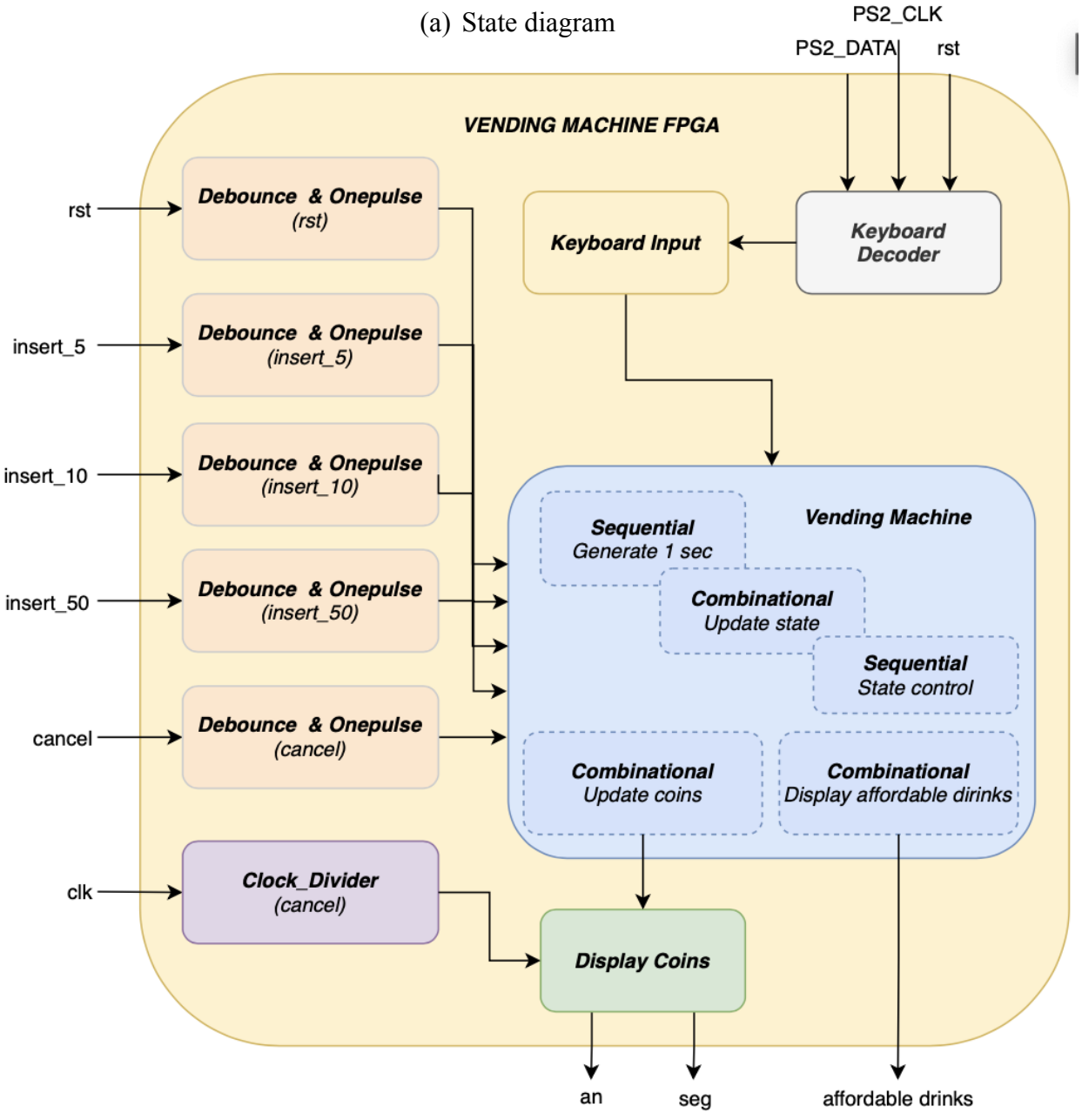


(b) Block diagram

1.5 Mixed Keyboard and Audio module



(a) State diagram



(b) Block Diagram

1.6 Vending machine

II. Design Process

First, start by making the block diagrams and state transition diagrams of the design.

Second, the modules are written in verilog. We used combinational and sequential circuits to create our modules. Updates of the states are done sequentially, and the others are done combinatorially. Then we check all the spelling, syntax, and errors.

Third, the testbench is built. The testbench is very useful to check whether or not the module works as it should. The use of display and task when making the testbench makes it easier to spot mistakes at our module, by using the CAD server. We made the testbench according to the input changes of each module.

Finally, debugging and reiterating each line to fix some bugs that were faced. Such as wrong I/O, swapped variables, wrong naming, wrong behavior, typos, etc.

Modules:

1. Sliding Window Sequence Detector

This module detects the sequence 1100(10)+01. When the sequence reaches the end, it will output dec as 1. The pattern 10 in the bracket above can be repeated but must at least appear once. When the sequence mismatch, *dec* will be one.

This module has a total of 8 states. We reset and update the value of the states sequentially. The changes of the state are done combinatorially with case. Where each state will change their state to the next state according to the state diagram in part 1. The dec is assigned to 1 when the last state is state 7 and the input is 1.

2. Traffic Light controller

The Traffic light controller is designed for the cross section of a highway and a local road. As the highway has higher priority, it will stay green as long as possible. In this case, the Highway will turn red only if the local road's sensor

detects cars. The cycle of every greenlight is at least 80 cycles, 20 for yellow and 1 for red.

We created three parameters for the lights, which are Green, Red, and yellow. We also created 6 other parameters for the states of the lights. 3 states for the highway and the other 3 for the local road. We created a counter for the cycle, named cyc. We will increment the cycle for every clock cycle, and reset every next state. To mark the cycle for the green (80 cycles) and yellow (20 cycles), we used 2 wires named cyc80 and cyc20. Where we assign these two cycles to 1 if the cycle count is more than 80 cycles, and more than 20 cycles for the other one.

We update and reset the state and counter sequentially. Where the light output is updated combinatorially with case. To calculate the next state, we used another combinational block. If the state is Highway Green and cyc80 is 1, it will be updated depending on whether or not the local road has a car. If the local road has cars, the light will change to yellow and the counter will reset. However, if there are no cars on the local road, the highway will turn green as long as possible. For the yellow lights, if the counter reaches 20, it will go to the next state, else the counter keeps on incrementing until it reaches 20.

3. Greatest common divisor

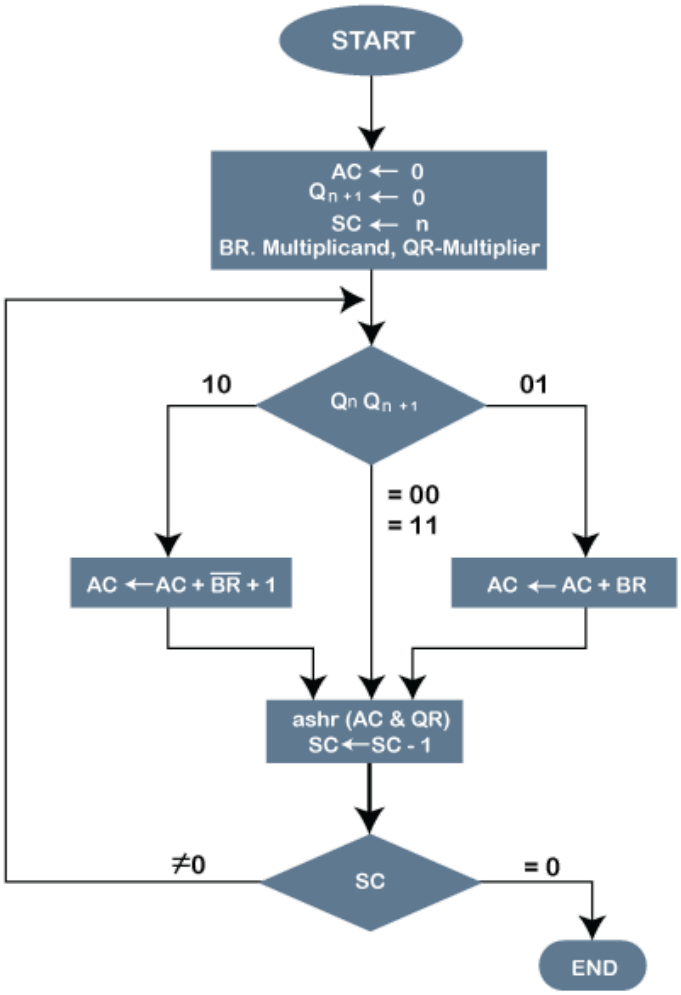
This module will calculate the greatest common divisor of two numbers a and b. Three states are used, WAIT, CAL, and FINISH. At the WAIT state, when start is 1, the operation will begin and inputs will be fetched. Else, it will stay in the WAIT state. It will also stay in WAIT if reset. After start, we then calculate the subtraction operation once per cycle at the CAL state. After calculating, the state will change to FINISH and stay there for 2 cycles with the output of the gcd and done = 1. Because the output will stay for 2 cycles after the gcd is counted, we created an extra state DISPLAY just to display the output for 1 cycle.

We reset and update the states sequentially with every positive edge clock. We calculate the gcd of the two numbers combinatorially if the current state is CAL. It will finish calculating if the value of b is zero. Not that the CAL will directly go to FINISH if the input of a is zero. For our module, when the state

reaches FINISH, it will go to another extra state after 1 cycle called DISPLAY and stay for 1 cycle. By this, the output will stay as their corresponding values for 2 cycles.

4. Booth Multiplier

The booth multiplier computes the product of two signed inputs. The booth multiplication mechanism is as follows:



Three states are used, WAIT, CAL, and FINISH. The state will stay at WAIT unless its reset or when start = 1. The value of a and b will be fetched when start = 1 and state will change into CAL. At the CAL state, the booth multiplication operation will be done per cycle and will finish after 4 cycles of calculations. The input for this module consists of *a* as the multiplicand, *b* as the

multiplier and *start* to trigger the module to start calculating the multiplied answer from *a* and *b*. The details on how the code worked will be explained below.

The variable inside this module is set to accept the -8 as a multiplicand, where the negation of -8 is -8 itself. Therefore, we used 5 bits to fetch the input where the most significant bit of it will copy the signed bit from the input *a* and named as *a_5bits*, and *neg_a_5bits* for the negative form with 2's complement method. Both of them will be stored in 10-bits register *a_pos* and *a_neg* with a form of {*a_5bits[4:0]*, 5'b00000} and {*a_5bits[4:0]*, 5'b00000}. The multiplier *b* will be stored on a pre-calculated answer on a 10-bit *calc* register with a form of {5'b00000, *b* , 1'b0}.

This algorithm takes 4 cycles, thus we used a register named *cyc* to count how many cycles of counting has been calculated. On each cycle we check two first least significant bits from the *calc* register. Whenever the two first least bits are the same (11 or 00) the *calc* won't do any addition, but when the two first least bits are 01, the *calc* will do an addition with *a_pos* and with *a_neg* when the two first least bits are 10. The product of this addition or not will be wired to *next_calc_unshifted*. The wire's name is unshifted since the final product on each cycle shall be arithmetically shifted to the right by 1 once and it will be stored to the *calc* with an intermediate wire named *next_calc_shifted*. The update happens each cycle for 4 cycles as mentioned before and the product after the fourth cycle will be registered to the output named *p* in the module and it only takes the 8-bit (between the most significant and the least significant bit).

The output will always be filled with an 8'd0 as a value whenever the process has not finished and the other registers will register the value to 10'd0 whenever they are not used.

5. (FPGA) Mixed Keyboard and Audio Module

This module will play the scales of piano starting from C4 to C8 and vice versa when it reaches the highest note. The scale can ascend or descend according to the key pressed *w* for ascend and *s* for descend. The key *r* is also used to control the speed of the scales.

We create a speed controller to control the tempo. We regulate several seconds per note, and this state is changed every time the *r* key is pressed. The Speed controller is a Moore machine where when the *r* is pressed, it switches between 0.5 sec/note or 1 sec/note. The clock divider in this module generates its output according to the output of the speed controller.

While the note_controller is used to regulate the current output sound frequency. It consists of two finite state machines. One is to control the current status whether the scales will ascend or descend. The other one is to read the above ascend or descend and switch between C4 and C8. The input will be *w* and *s* and the speed will be updated according to the speed given by the speed controller. The value in the speed controller is converted with ascend and descend. The note uses beat frequency as a signal to update with clock. The direction of the scales is determined according to the value of ascend.

The PWM gen converts the frequency to audio input and the Beat converts the tone to the correct frequency.

At our FPGA module, we process the keyboard input. The step to read keyboard input is very complicated. Among the keyboard keys, only one keyboard signal is allowed to be 1 at a time to avoid errors caused by pressing multiple keys at the same time.

6. (FPGA) Vending Machine

The vending machine provides four options: coffee, coke, oolong and water with prices 75, 50, 30 and 25 respectively. The user can input money with values 5, 10, and 50. With a maximum value of 100. They can cancel the money input and reset.

Only two states are needed, the input state and the change state. The insert state is where the user inputs money. And the change state represents the money minus item mode. It will decrease by 5 every 1 second until 0 dollars is transferred back to the insert state, and wait for the next input.

The clock divider produces the signal required when the seven segment display shows multiple digits. The keyboard input is processed at the FPGA main

module because it is quite complicated. Among all the keyboard keys, only 1 keyboard signal is allowed to be 1 at a time to avoid errors caused by pressing multiple keys at the same time. Debounce and onepulse is applied on five buttons. Because the original clk is used here, the time interval is very short. So the number is adjusted here.

The vending machine contains four parts, generally a signal that coins are enough to buy drinks, generate a one-second signal, update coins, and update state. The signal is for the coins to indicate if the coins are enough to buy the item are calculated based on whether the coins are greater than equal to the price of the drink. The use of the LED indicates and judges whether it is enough to buy the item at the vending machine to enter the change state.

When inserting the states, the coin and state are updated in real time according to the clock.

III. List of Contributions

徐美妮 Mary Madeline Nicole 109006205's contributions, such as :

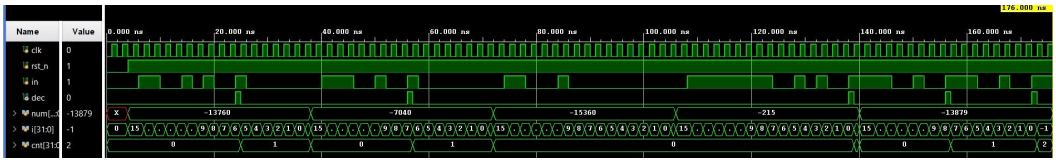
- Design and write module Sliding Window Sequence Detector
- Design and write module Traffic Light Controller
- Design and write module Greatest Common Divisor
- Design and write module Booth Multiplier
- Design and write module Mixed Keyboard and Audio
- Design and write module Vending Machine
- Design and write testbench Sliding Window Sequence Detector
- Design and write testbench Traffic Light Controller
- Design and write testbench Greatest Common Divisor
- Design and write testbench Booth Multiplier
- Design and write testbench Mixed Keyboard and Audio
- Design and write testbench Vending Machine
- Draw block diagrams and state transition diagrams,
- Simulate and synthesis modules in Vivado,
- Write the report.

林之耀 Kevin Richardson Halim 109006277's contributions, such as :

- Design and write module Sliding Window Sequence Detector
- Design and write module Traffic Light Controller
- Design and write module Greatest Common Divisor
- Design and write module Booth Multiplier
- Design and write module Mixed Keyboard and Audio
- Design and write module Vending Machine
- Design and write testbench Sliding Window Sequence Detector
- Design and write testbench Traffic Light Controller
- Design and write testbench Greatest Common Divisor
- Design and write testbench Booth Multiplier

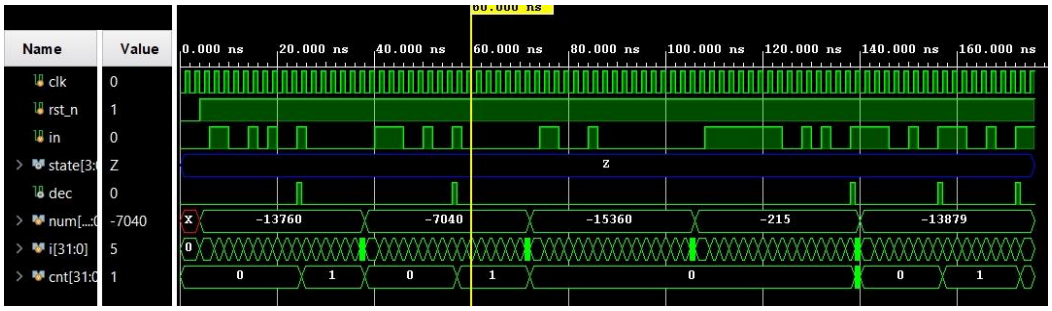
- Design and write testbench Mixed Keyboard and Audio
- Design and write testbench Vending Machine
- Design and write testbench Built in self test,
- Simulate and synthesis modules in Vivado,
- Write the report.

IV. Design Test



4.1 Sliding Window Sequence Detector

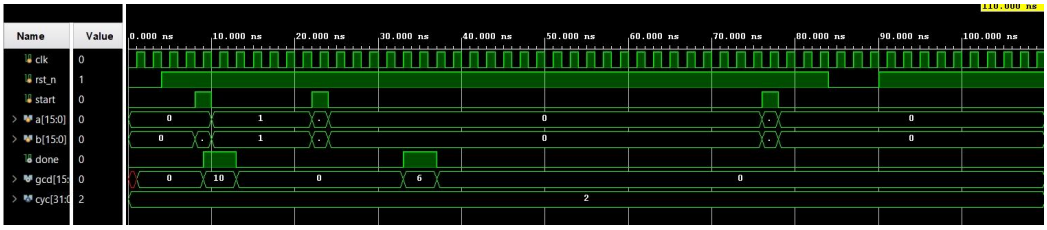
We test the Sliding Window Sequence Detector by inputting a predetermined output sequence. By choosing an input according to the detector expected output that matches or mismatch. This approach is to test whether our design is correct or if there are some problems. We use $display to print comments for every test case, also to mark if the result from our code is right or is an error. We test the output every negative edge, and generate the different sequences on the posedge. The sequences we generated are 16 bits. The sequence is tested by getting each bit one by one with a for loop at the initial block. We tested the code using waveform and ncverilog. Using some integer to count how many sequence appeared and compare it to the expected answer.



4.2 Traffic Light Controller

The traffic light controller was tested by first resetting the module. Next, we wait for more than 100 clock cycles to see whether the highway's light changes or not, and turn on the local road afterward. By mixing the input variable the wave of output will be seen and we could iterate the expected answer and the output given by the module. We used *$display* to mark whenever a car appeared on the local road. Then, we also mark *$display* when the car disappears from the road. To wait for the clock cycle, we used a for loop to count as much as the clock

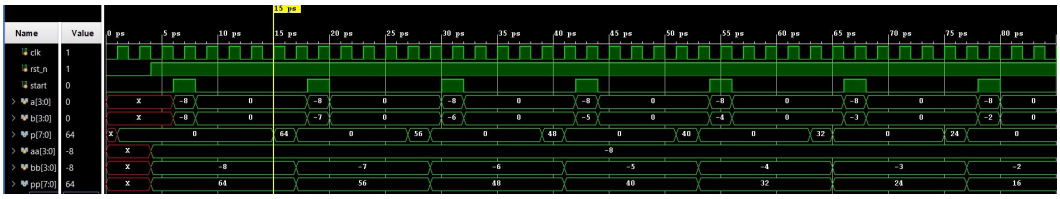
cycle we want to test. We did this so that we can test whether or not the clock cycles for each states are according to the criteria, and to make sure that the highway light will stay green as long as possible, and whenever the local road does not contain any vehicle, the green light will occur on the highway.



4.3 Greatest Common Divisor

The GCD module is tested by creating several types of test cases. First, we test when input *a* is 0. So the GCD should be output immediately. For the second test case, we input a and b with different numbers, and let the module calculate until b is 0. For the third one, we input random numbers and suddenly reset in the middle, to see if the module will do as the expected output or not.

These cases above cover the edge cases and conditions that might result in error. We simulate the module with vivado and ncverilog.



4.4 Booth Multiplier

The booth multiplier is tested by generating different values for a and b. We created a task test to check the correctness of the code, where it will print out ERROR when it outputs a different value to the desired output. For the booth multiplier, we also check when the signed bit is negative instead of positive. We also check the edge case for the most negative value of the a and b which is -8. We make sure that it generates the correct output even for the most negative bit. We check all the possible multiplication combinations of 2 signed 4-bits binary

represented, which is from -8 to 7 using for-loop. Inside the for-loop there are some procedure which is wait the module to calculate the answer on each 4 clock cycles with *task* name *Test* on each positive clock edge and change the input combination afterwards.

V. What we have learned from Lab 5

- ➔ Implement both keyboard and audio to the FPGA.
- ➔ Use Keyboard as user interface for FPGA
- ➔ Generate different frequencies for music notes.
- ➔ Implement Keyboard Decoder and Music