Hardware Design Lab 5 Report

Group Number: 30

Members: 112062130 侯佑勳
112062326 孔祥光

Contents:

Design & Testing

1. Sliding Window (Mealy Machine)
    - input: clk, rst_n, in
    - output: dec

Design

To detect the sequence “1110(01)+11”, we encode the states with 3-bit binaries like this: S0~S7 corresponds to 3’b000~3’b111, respectively. Below is the State Diagram.

These states represent what pattern has currently been detected:

S0: 0	S4:1110
S1: 1	S5: 11100
S2: 11	S6: 111001
S3: 111	S7: 1110011

Only when the state is **S7 and input is 1**, will the output be 1.

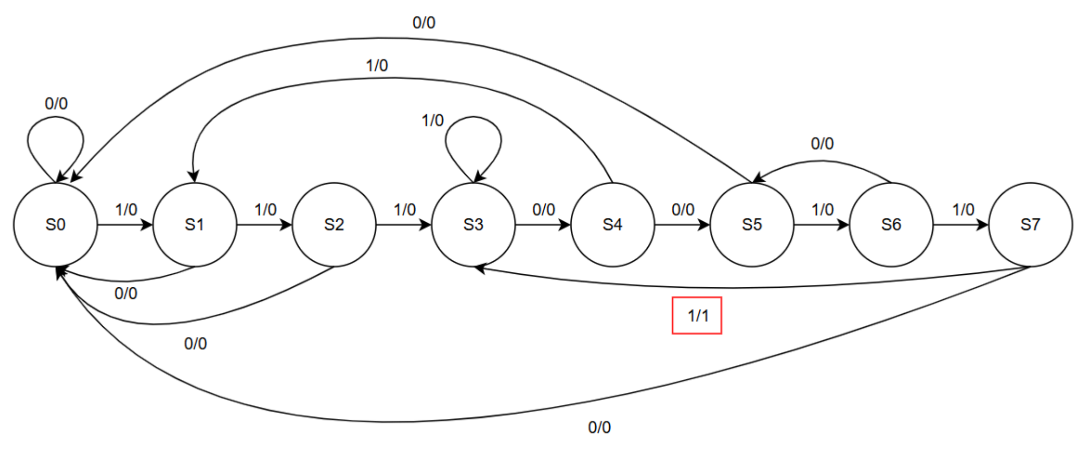


Figure 1-1 Sliding Window State Diagram (detect 1110(01)+11)

Since the next state is determined by the current state and input, we'll use a 16-1 MUX to generate the *next_state* signal, shown in Figure 1-2.

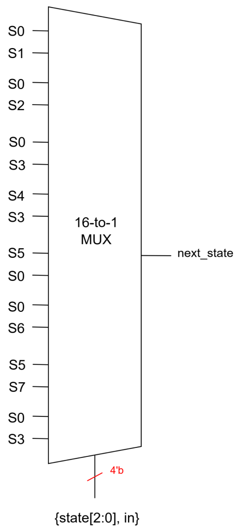


Figure 1-2 16-1 MUX for Determining the *next_state*

We use the circuit below to determine how to update the state, and only if the current state is S7 and input is 1, will dec be 1.

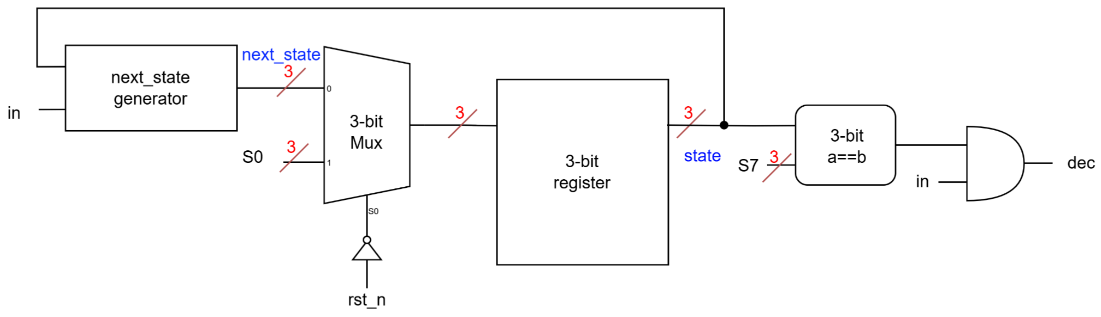


Figure 1-3 Mealy Machine

Testing

We use multiple waveforms to observe whether the dec is correct. For convenience, we use an array to restore the test_pattern and use a for loop to generate each digit in the test_pattern. The waveform below represents "test_pattern = 20'b1110_0101_0111_1001_0111" .

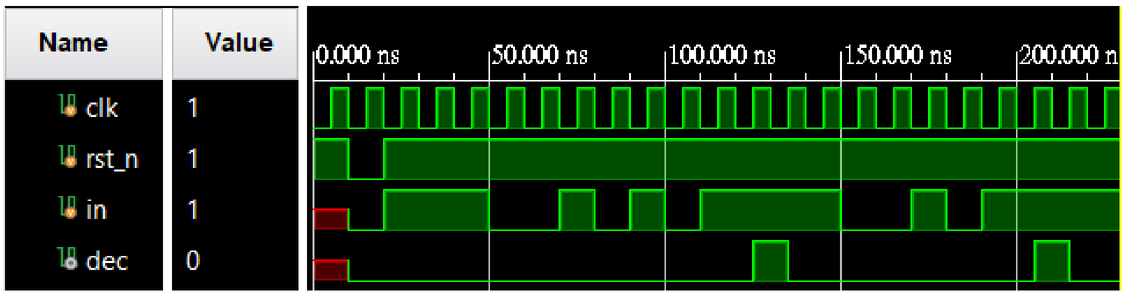


Figure 1-3 Mealy Machine

2. Traffic Light Controller
   - input: clk, rst_n, lr_has_car
   - output: [2:0] hw_light, [2:0] lr_light

Design

We design a 6 states moore machine to implement the traffic light controller. First we encode the states with 3-bit binaries like this: A~F corresponds to 3'b000~3'b101, respectively. The output(HW, LR) is only defined by the current state. Below is the State Diagram.

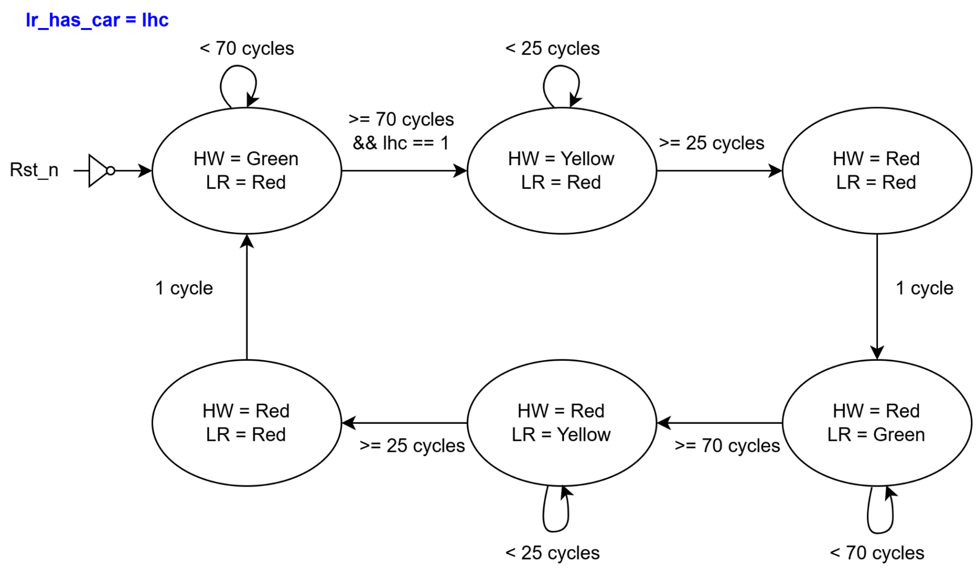


Figure 2-1 Traffic Light Controller State Diagram

We use the time we spent to judge whether we should go to the next state, so it's necessary to develop a counter to record time. Moreover, the counter needs to be reset after each state transmission. The reset value is 0, so the real counter condition should be deducted by 1

For instance:

- counter == 69 → Green changes to Yellow
- counter == 24 → Yellow Change to Red

To let the counter only exist in one always block, we design the count_reset signal to record whether the counter should be reset.

Figure 2-2 is how the counter works.
Figure 2-3 is how the count_reset is generated.

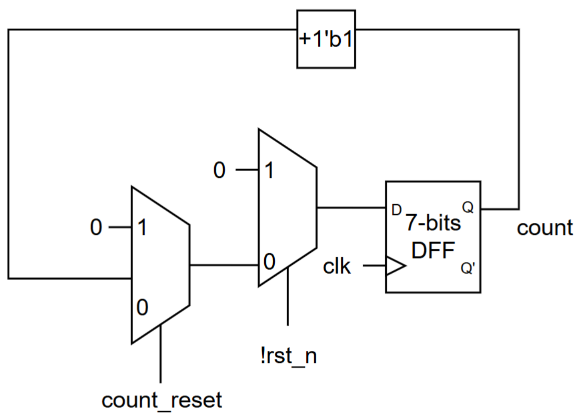


Figure 2-2 Counter

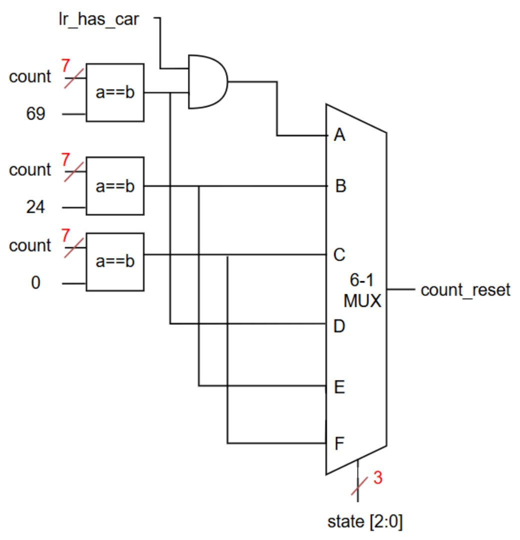


Figure 2-3 count_reset generator

The Figure 2-4 is the state changing circuit, the critical condition is also whether the time counts to a certain value.

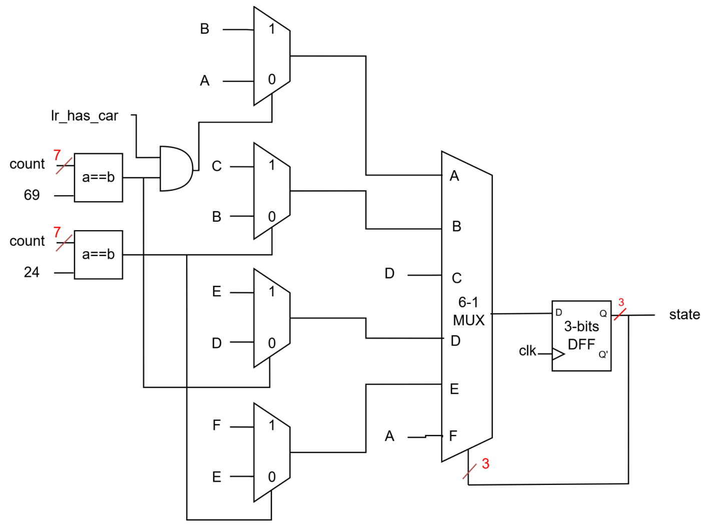


Figure 2-4 state changing circuit

Lastly, we use two MUX and the current state as a signal to determine the hw_light and lr_light.

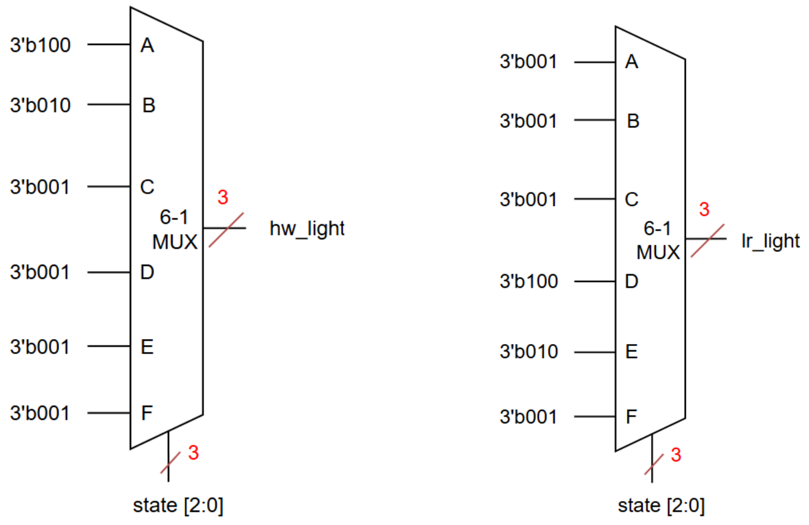


Figure 2-5 hw_light, lr_light 6-1 MUX

Testing

We use two cases to test whether the condition “lr_has_car” works well for the state transition from A to B. The difference between those two cases is when the lr_has_car signal becomes 1. It becomes 1 after 69 cycles since the rst_n is zero in the 1st case, and after 200 cycles in the 2nd case. We also observe the waveform to make sure whether each state transition is correct.

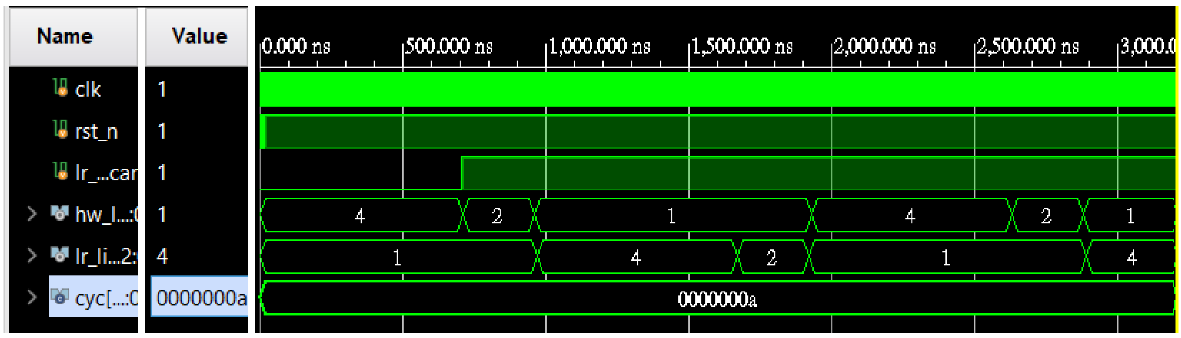


Figure 2-6 traffic light controller waveform situation 1st case

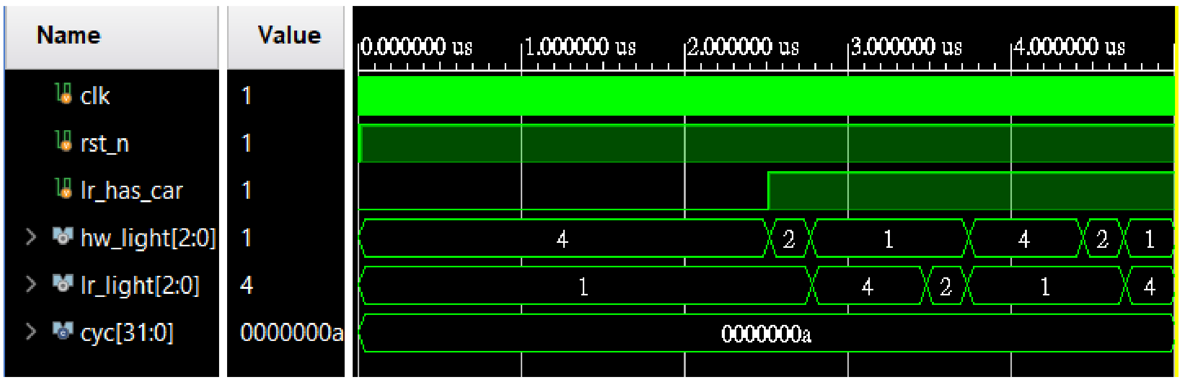


Figure 2-7 traffic light controller waveform situation 2nd case

3. Greatest Common Divisor
   - input: clk, rst_n, a[15:0], b[15:0], start
   - output: done, [15:0]gcd

Design

The state diagram in Figure 3-1 shows the intended behavior of the GCD module.

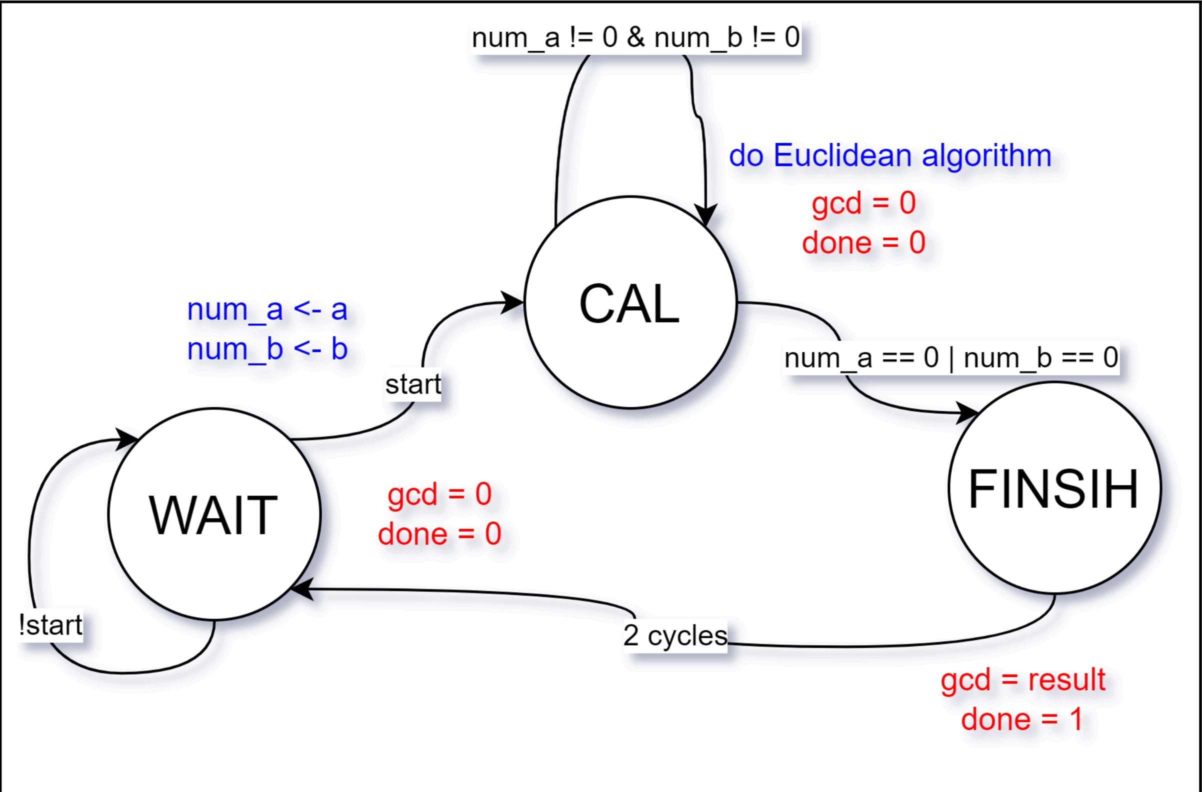


Figure 3-1 State Diagram of GCD module

And Figure 3-2 and 3-3 shows the simple FSM to achieve the above state transitions.

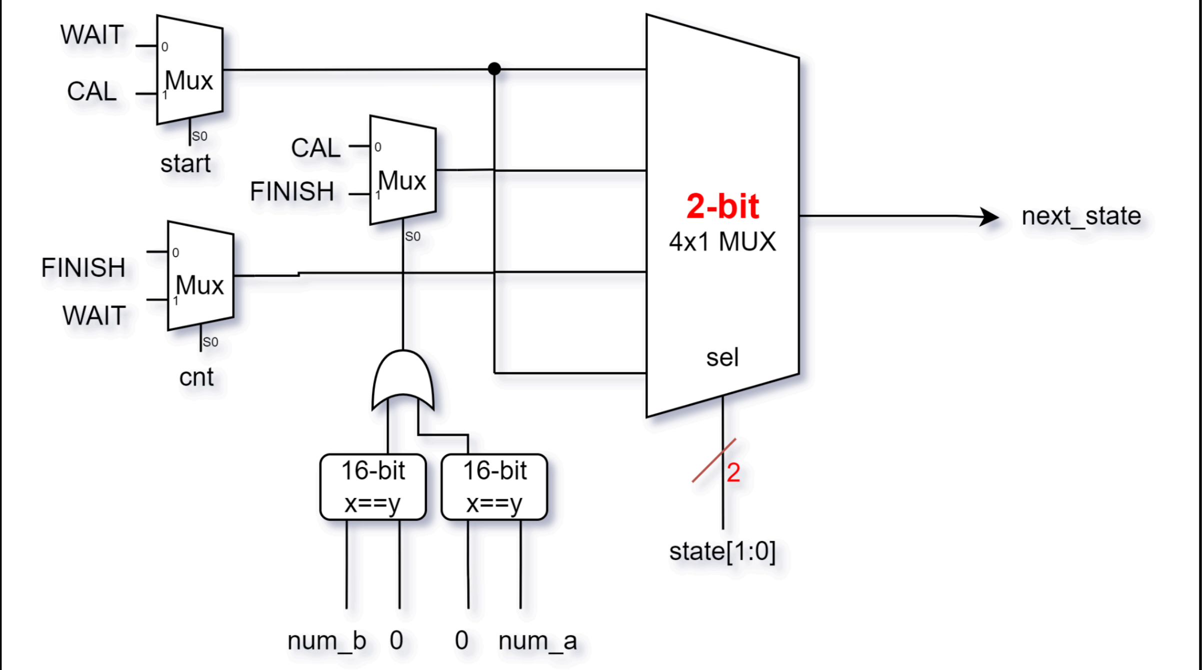


Figure 3-2 next_state generator for FSM

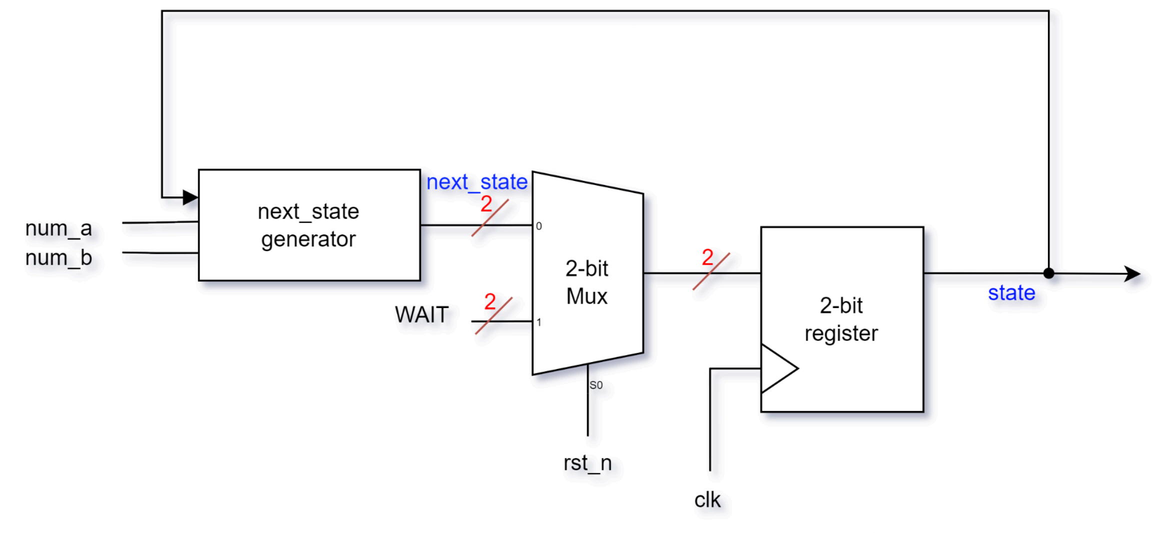


Figure 3-3 FSM

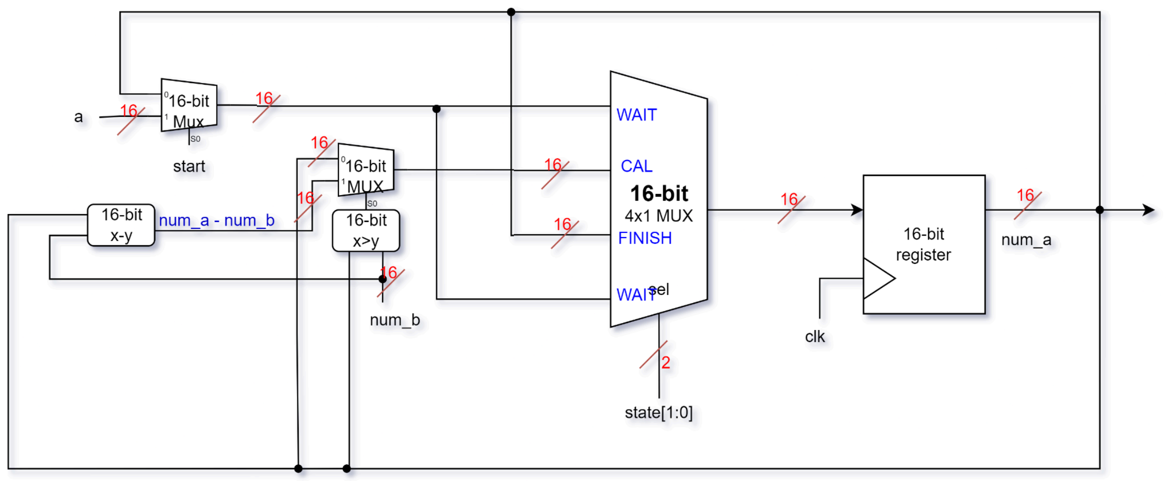


Figure 3-4 A buffer to store the operands
(this example shows register for `num_a`, that for `num_b` is similar)

Since the input could change during calculation, we'll buffer the input before we start the operations, storing them in two 16-bit registers, shown in Figure 3-4.
The circuit may seem complex at first, but the underlying concept is rather simple. We update the content of the buffer registers `num_a`, `num_b` according to the current state and inputs:

- WAIT
  - start: fetch the input `a, b` to update the current value
  - !start: keep the current value
- CAL: update the value of the buffers using the Euclidean algorithm
  - num_a > num_b:
    (`num_a`, `num_b`) ← (`num_a - num_b`, `num_b`)
  - otherwise:
    (`num_a`, `num_b`) ← (`num_a`, `num_b - num_a`)
- FINISH: keep the current value

Finally put everything together and obtain the whole module.
(it's rather messy since everything is a feedback of each other.)

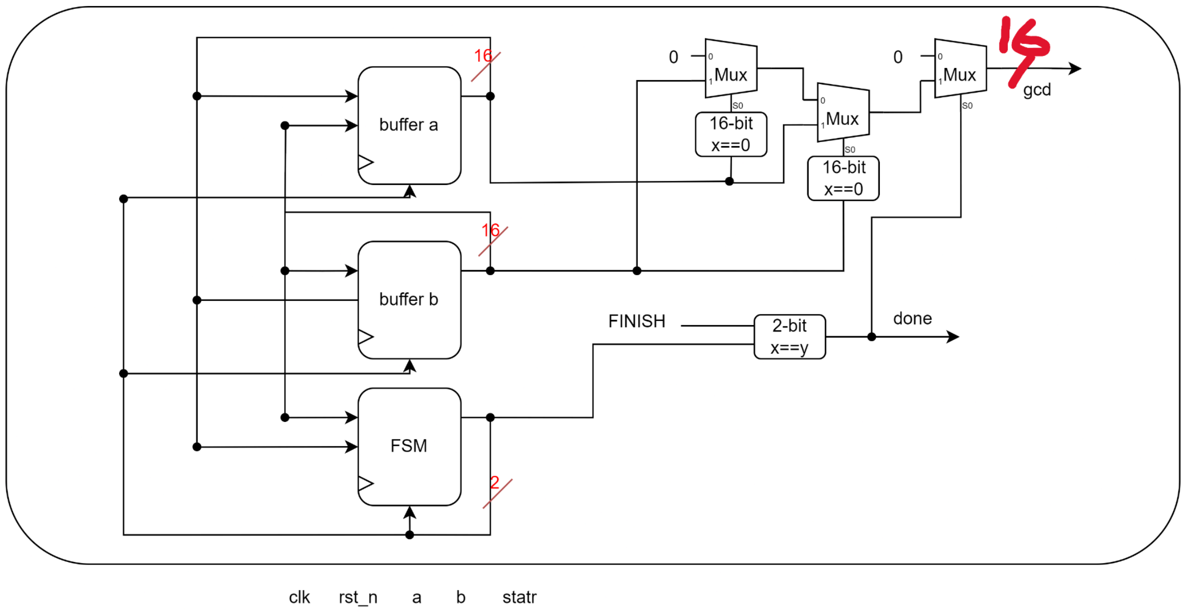


Figure 3-5 GCD module

Testing

Figure 3-6 Simulation Waveform for GCD module

FPGA

1. Music Step Player
   - inout: PS2_DATA, PS2_CLK
   - input: clk
   - output: pmod_1, pmod_2, pmod_4
   - some important signal:

     direction: determine to ascend or descend
     fast: determine 1sec/note or 0.5sec/note
     [4:0] ibeatNum: determine which note to display
     [31:0] freq: the display note's frequency
     beatFreq: determine the playing tempo
     rst: reset or not

Design
In this FPGA question, we use some submodules from Music demo code (including PWM_gen, PlayerCtrl, Music) and the KeyboardDecoder, Oneplus module from Keyboard demo code.

First, we add a fast signal to determine the ascending rate. If the fast signal is 1, the count_max will be divided by 2 again, which makes the cycle shorter. Figure 4-1 is the PWM_gen module.

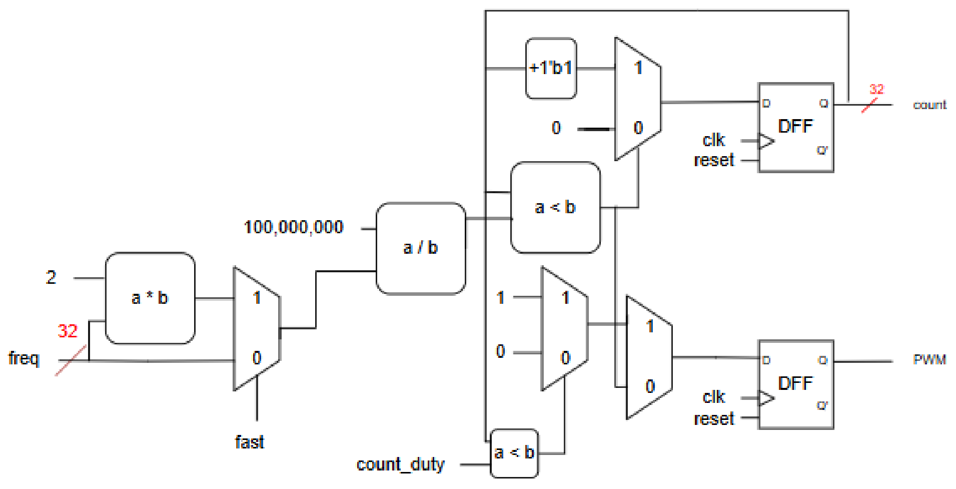


Figure 4-1 PWM_gen module

Second, we add a direction signal to the PlayCtrl module. Moreover, we add a reset_done signal to hold a C4(ibeatNum = 0) for one beatFreq cycle because of the following situation:

We reset the PlayerCtrl module, and the `ibeat` should change to zero(the pitch is C4). However, because the `clk` of the PlayCtrl is the beatFreq from the PWM_gen, the beatFreq will change to 1 after 1 system clock in the PWM_gen module. If the `beatFreq`(the clock in PlayCtrl) becomes 1, the pitch will start to ascend and we won't hear the pitch C4.
Figure 4-2 is this module.

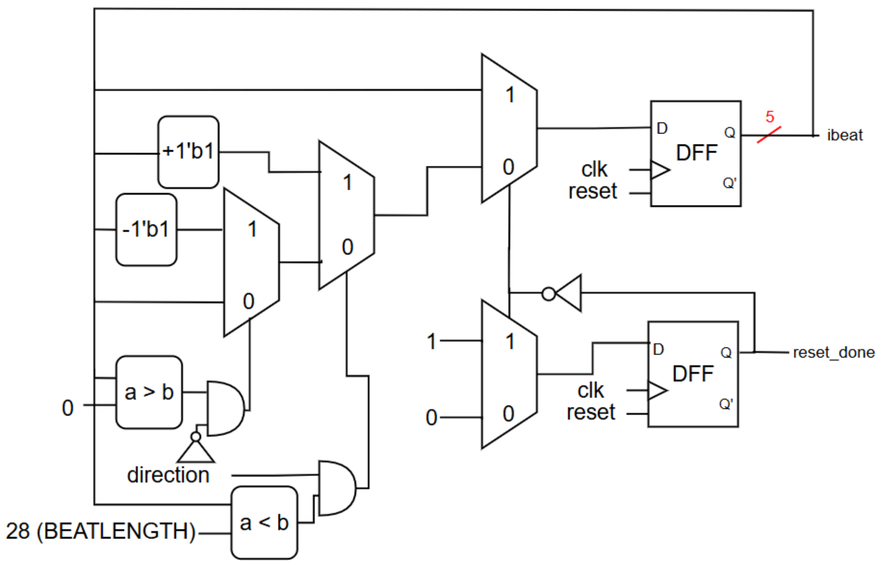


Figure 4-2 PlayCtrl module

Third, we add 29 pitches (C4~C8) to the Music module, and use ibeatNum to choose which pitch should be displayed.

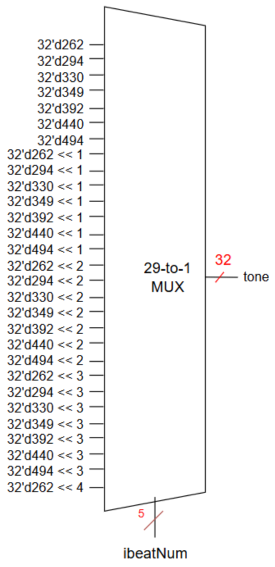


Figure 4-3 Music module

Below is the whole circuit.
We detect the key w, s, r in a sequential always block to change the direction or rate respectively. For the reset signal, we directly assign whether the enter_code is pressed to it. We also add onepulse function to the reset signal.

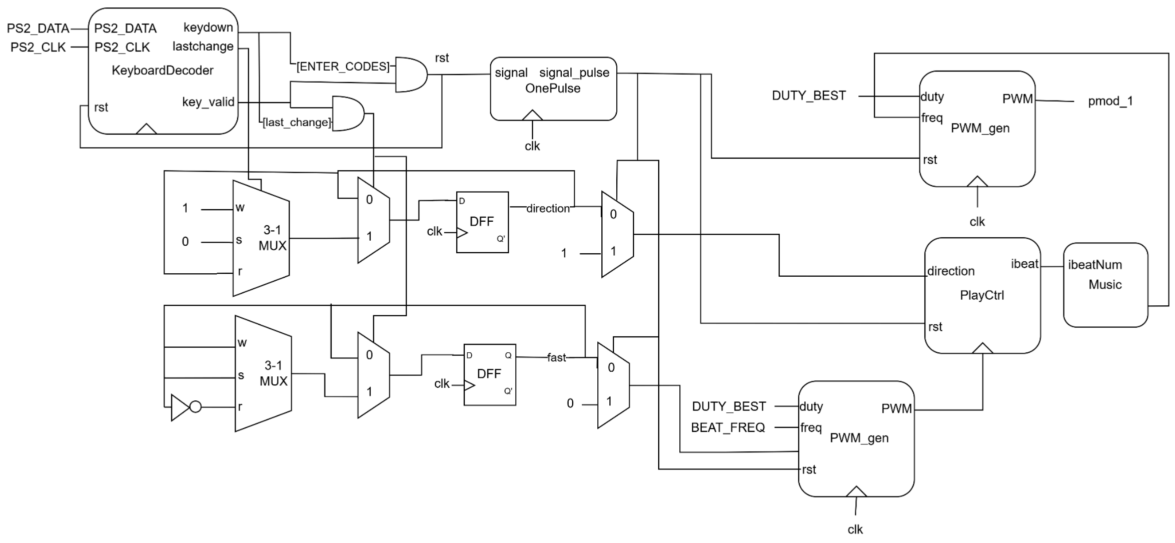
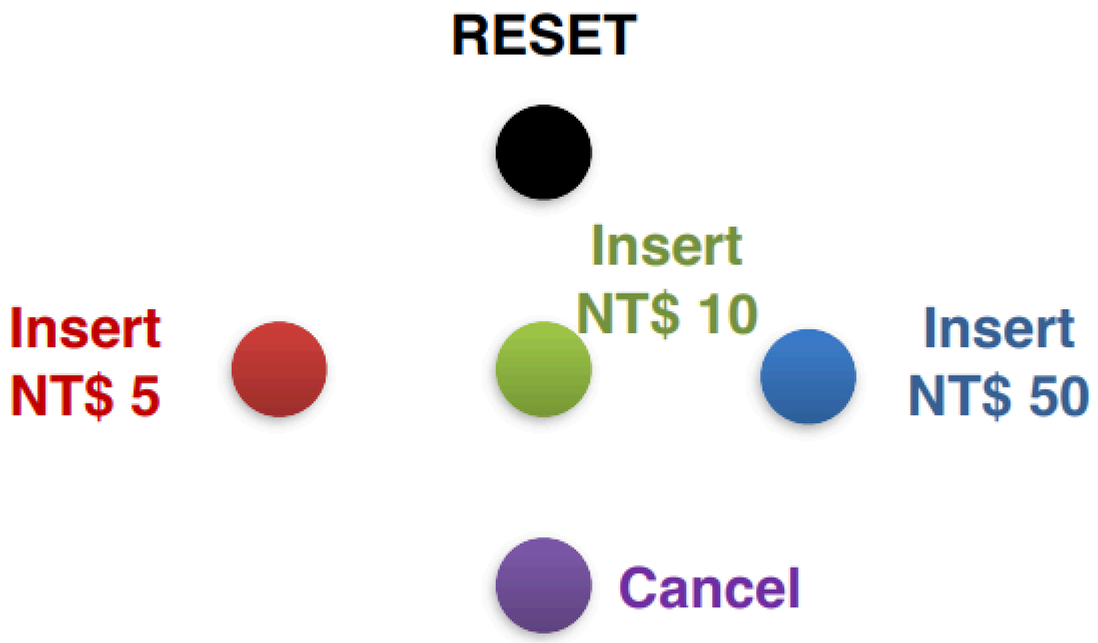


Figure 4-4 Whole Circuit

2. Vending Machine
    - inout: PS2_DATA, PS2_CLK (keyboard I/O)
    - input: clk, btns



    - output: digit[3:0], display[6:0], LED[3:0]
    - some important signals: cash, drink_avail[3:0], divided_clocks

Utility & I/O Modules
We define some divided clocks within the submodules where the duty cycles are different from the system clock, such as a 100Hz clock for debounce, a 1Hz clock for returning change, and a 1000Hz clock for time-multiplexing the 7-seg display.

Also, since we're using buttons, the input from them must be debounced and onepulsed before use.

A **Keyboard Decoder** is used to handle the I/O from the keyboard. We subscribe to the changes in key A, S, D, F, as they are used for the selection of drinks.

Finally, the output end is we have a **SevenSegmentDisplay** module to handle concurrent display of 3 digits, plus decoding 4-bit numbers to 7-segment display positions.

Submodules

The only submodule is the **Vendor FSM**, which emulates the behavior of the vending machine. There are two states, IDLE and CHANGE. When IDLE, the machine waits for coin insertion and drink selection. When CHANGE, the machine returns 5$ change each second.

Top Module

Connecting the processed input (reset, clear, insert_five, insert_ten, insert_fifty) to the Vendor FSM, and that to the output processor, is our final module. See Figure 5-1.

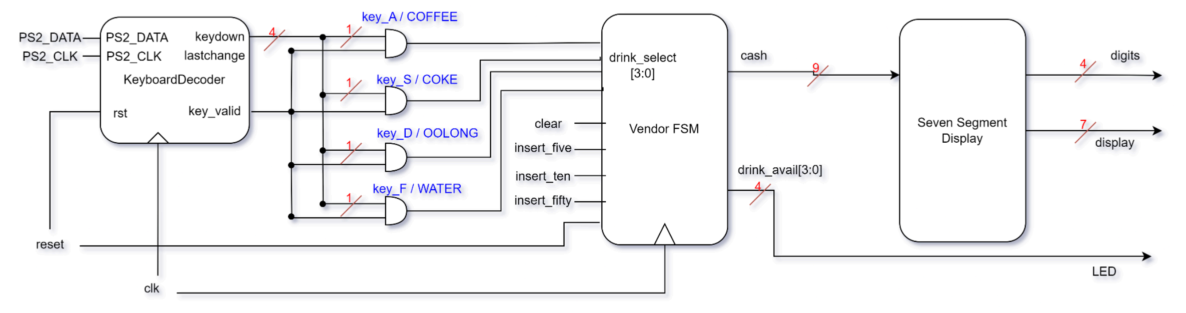


Figure 5-1 Vending Machine FPGA

What Have We Learned?

112062130 侯佑勳
I have learned how keyboards and audio modules work in this lab. For the keyboard, the most difficult parts are how to detect when the key is pressed and released. I used an assignment to implement this function. For the audio part, it’s essential to understand how every submodule works, in that way, we can implement new features more conveniently.

112062326 孔祥光
In this lab I’ve familiarized myself with basic keyboard & audio I/O of FPGA. Other than that, I’ve also learned more about the best practices for designing FSM and modules with complex behaviors.

Contribution

112062130 侯佑勳
- Sliding Window Mealy Machine (&tb)
- Traffic Light FSM (&tb)
- Booth Multiplier
- Music Step Player FPGA

112062326 孔祥光
- GCD (&tb)
- Vending Machine FPGA
- Booth Multiplier tb