Hardware Design

Lab 5 Report

Keyboard and Audio Modules

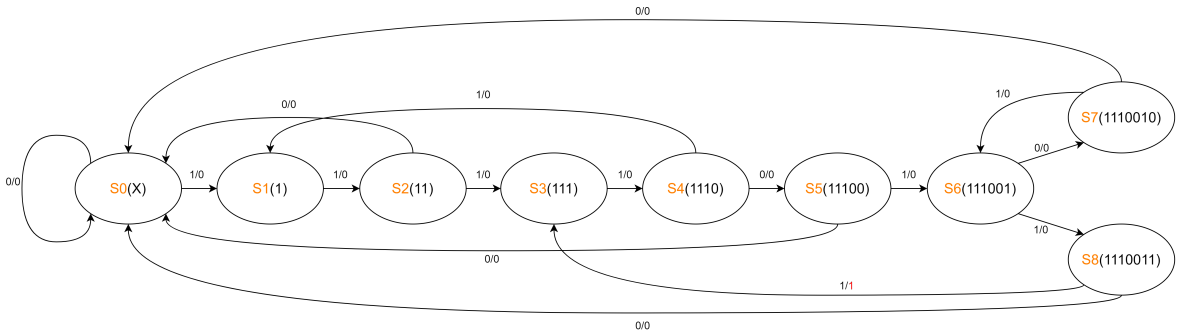
Team 01

112062122 謝佳晉　112062144 范升維

Table of Contents:

Advanced Q1. Sliding Window Sequence Detector

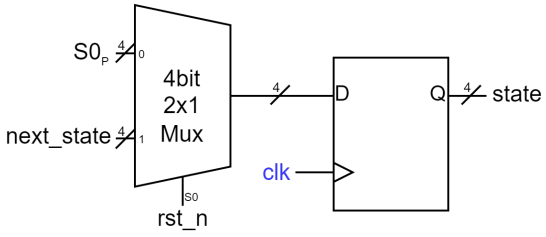
In advanced Q1, we were instructed to create a sliding window sequence detector. The detector takes in the input value and update the state and dec according to the value.



▲ Figure 1.1: state transition diagram

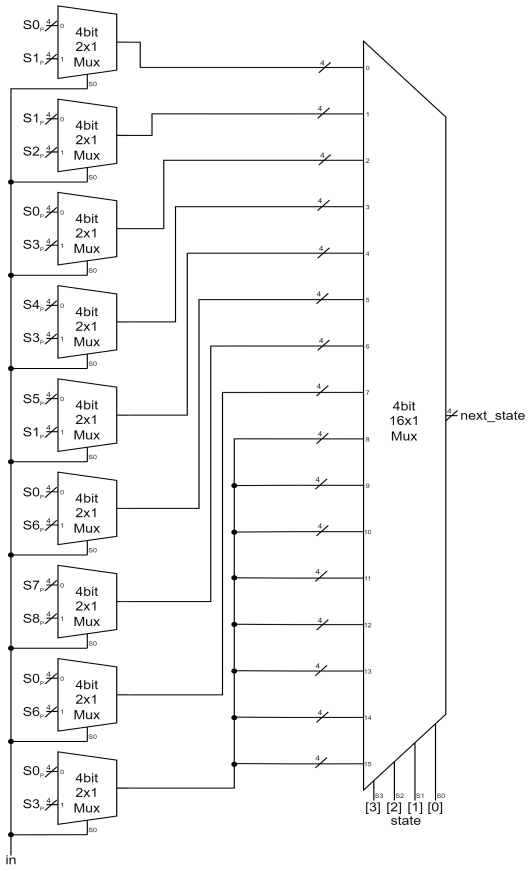
- 3 inputs: *clk, rst_n, in*
- 1 output: *dec*
- 2 internal signals: *state[3:0], next_state[3:0]*
- 9 states: *S0, S1, S2, S3, S4, S5, S6, S7*

First, we design the state transition mechanism as Figure 1.1. In *S6*, we divide the next states into *S7* and *S8* to deal with the repeating *2'b01* cases. If the state is *S7* and the input value is *1'b1*, state will go back to *S6*, detecting another *2'b01*. On the other hand, If the state is *S8* and the input value is *1'b1*, state will go back to *S3* and make *dec* output *1'b1*.

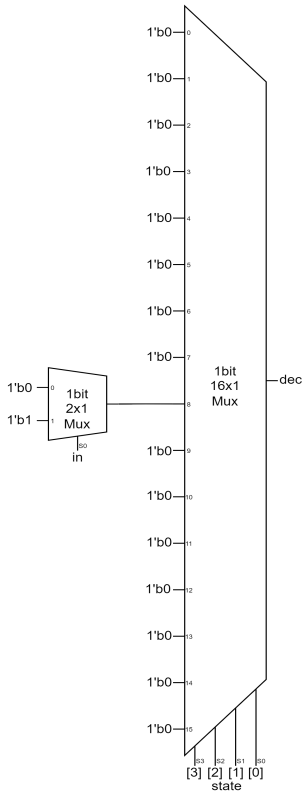


▲ Figure 1.2: *state* signal

In Figure 1.2, we show the sequential circuit design of *state[3:0]*. If *rst_n* is *1'b0*, state will go back to initial state *S0*. Otherwise, it will take in *next_state[3:0]*.



▲ Figure 1.3: *next_state* signal



▲ Figure 1.4: *dec* signal

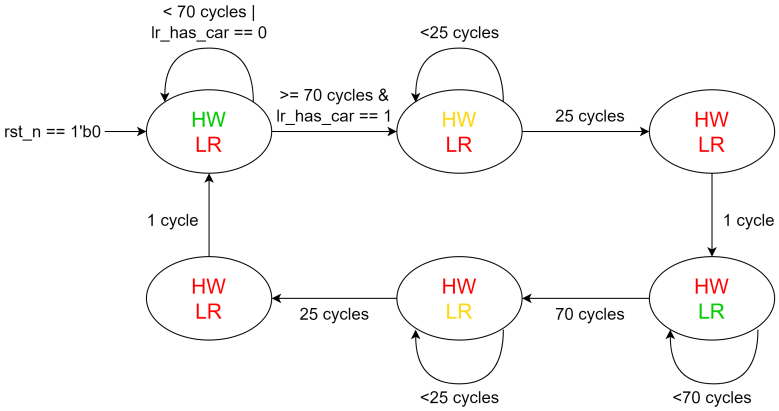
Figure 1.3 and Figure 1.4 shows our combinational circuit designs for *next_state[3:0]* and *dec*. We set *state[3:0]* as the rightmost MUXs' selection bits. We only have nine states, but the rightmost MUXs have 16 input ports, so default cases have to be considered. However, because states only update within the nine states, we can set the vacant input ports to whatever 4-bit values we want. For *next_state[3:0]*, we set them to the input value same as the case when state *S8*, while in *dec*, we set them to *1'b0*.

For *next_state[3:0]*, we have to decide which state should the next state be based on *in* and the state at the moment, so we add 2x1 MUXs to 0^{th} to 8^{th} input ports.

In *dec*'s circuit, because we only have to take output value branch into consideration when state is *S8*, we add a 2x1 MUX to the 8^{th} input port, and set *1'b0* to other input ports.

Advanced Q2. Traffic Light Controller

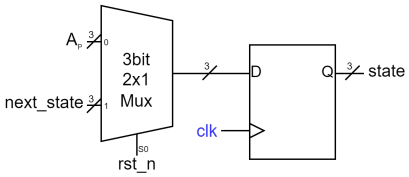
In advanced Q2, we were instructed to create a traffic light controller. The controller mechanism is taking in the input value and updating the state according to the value and clock cycles. We use a counter to keep track of the clock cycles.



▲ Figure 2.1: state transition diagram

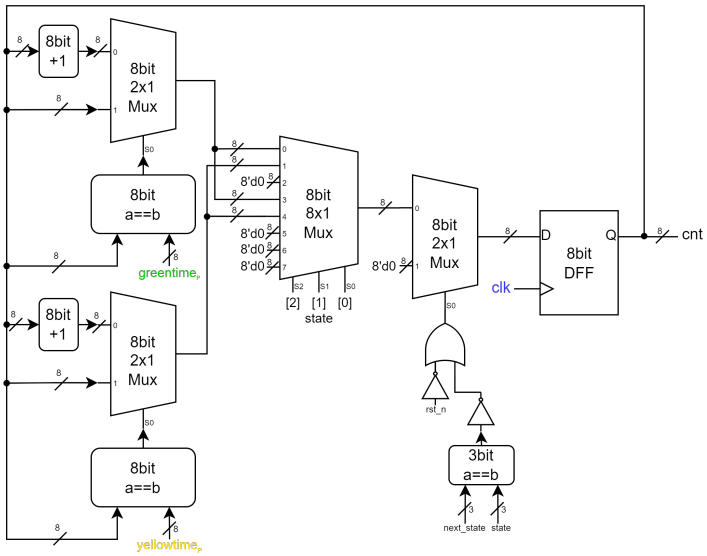
- 3 inputs: *clk, rst_n, lr_has_car*
- 2 outputs: *hw_light[2:0], lr_light[2:0]*
- 3 internal signals*: cnt[7:0], state[2:0], next_state[2:0]*
- 6 states: *A, B, C, D, E, F*
- other parameters: *greentime*(*8'd69*), *yellowtime*(*8'd24*)

Before designing our circuits for this problem, we complete the state transition diagram in the lab powerpoint, as shown in Figure 2.1. We add some go-back arrows and the *rst_n* case.



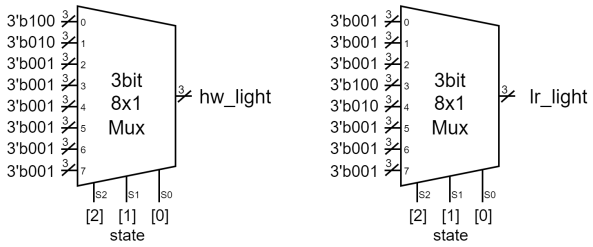
▲ Figure 2.2: *state* signal

In Figure 2.2, we show the sequential circuit design of *state[2:0]*. If *rst_n* is *1'b0*, state will go back to initial state *A*. Otherwise, it will take in *next_state[2:0]*.



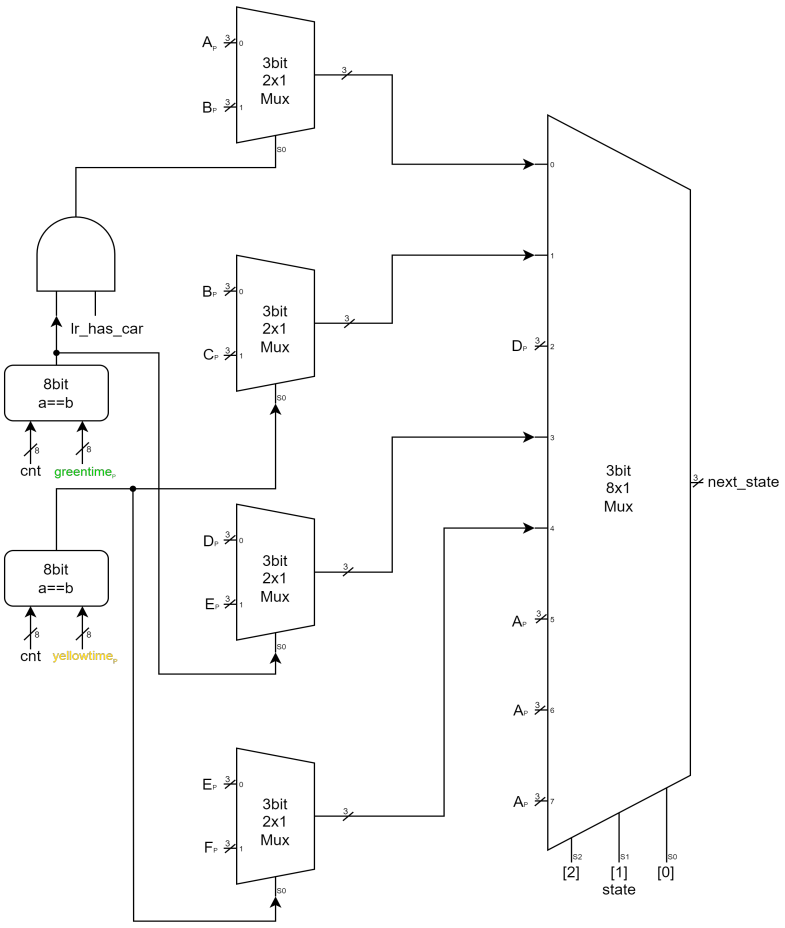
▲ Figure 2.3: *cnt* signal

Figure 2.3 shows our sequential circuit design for *cnt*. Because we have to set *cnt* to *8'd0* when *rst_n* == *1'b0* or state changes, we add a 2x1 MUX before the DFF and set the 1st input port to *8'd0*. For the 0th input port, it takes in the updated value of *cnt*. When state is *A* or *D*, *cnt* will increase if *cnt* < *greentime* and stop counting if *cnt* == *greentime* to make sure that state can change at exactly next cycle and in case of overflow. On the other hand, when state is *B* or *F*, *cnt* will increase if cnt < *yellowtime* and stop counting if *cnt* == *yellowtime* for the same reason.



▲ Figure 2.4: *hw_light* signal and *lr_light* signal

hw_light changes only in the front 3 states, and *lr_light* changes only in the back 3 states. Therefore, we use MUXs and plug in *state[3:0]* as selection.



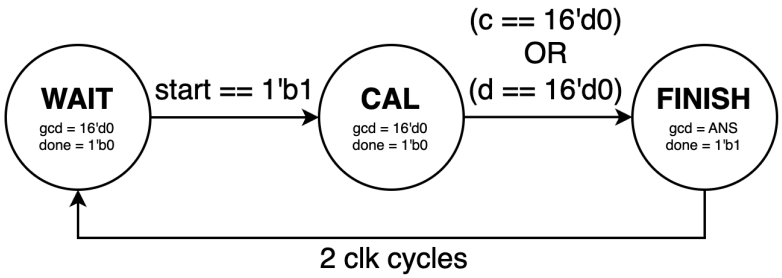
▲ Figure 2.5: *next_state* signal

Figure 2.5 shows our combinational circuit designs for *next_state[2:0]* (we set the initial value of *next_state[2:0]* to *A*). When state is *A*, *next_ state[2:0]* will update according to *cnt*, *lr_has_car*.

When state is *B*, *C*, *D*, *E*, *F*, *next_state[2:0]* will update only based on *cnt* values. However, when state is *C* or *F*, updating requires only one clock cycle. Therefore, we don't need to actually use the value of *cnt* but update *next_state[2:0]* to next value directly. For default cases, we simply plug in *A*.

Advanced Q3. Greatest Common Divider

In Q3, we are tasked with implementing the GCD (Greatest Common Divisor) that we are familiar with writing in C++. However, this time we cannot use division and modulo operators, so we need to implement it using repeated subtraction and conditional statements instead. First, let's look at the state diagram for this problem:

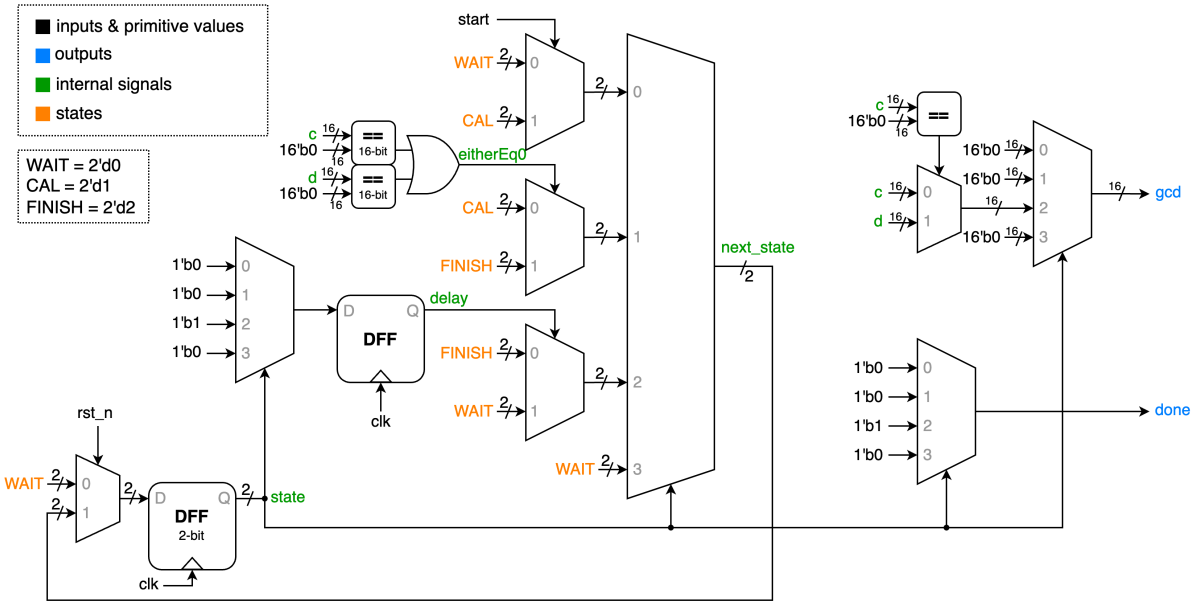


▲ Figure 3.1: State Diagram

As shown in the diagram, the transition to the next state only occurs when the conditions for moving to the next state are satisfied; otherwise, the current state is maintained.

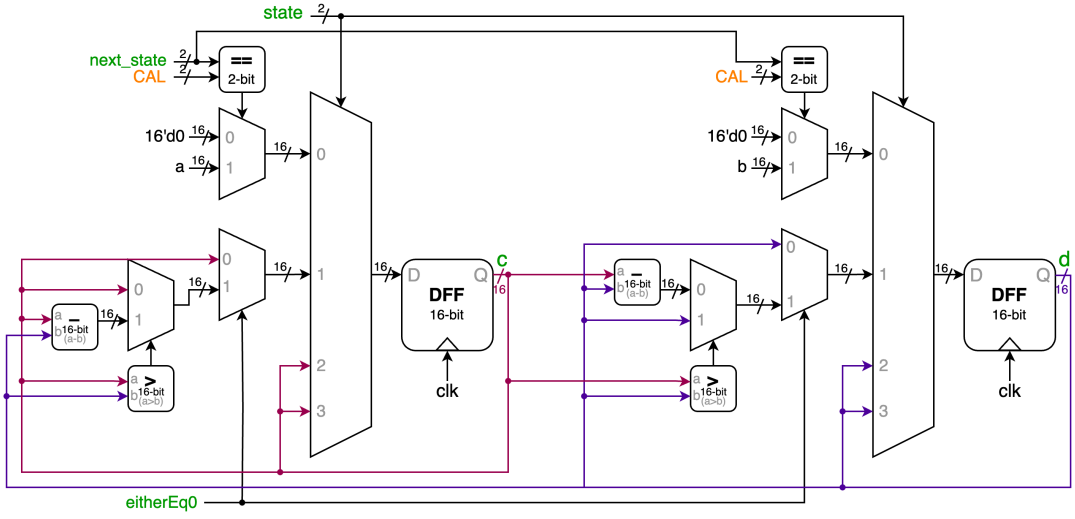
- 5 inputs: *clk, rst_n, start, a[15:0], b[15:0]*
- 2 outputs: *done, gcd[15:0]*
- 4 internal signals*: state[1:0], next_state[1:0], c[15:0], d[15:0], delay, eitherEq0,*
- 3 states: *WAIT(2'd0), CAL(2'd1), FINISH(2'd2)*

In addition to the specified I/O, we utilize two internal signals, c and d, to perform calculations during the *CAL* state. The *delay* signal is used to implement the two-clock-cycle waiting period at the end of the *Finish* state, while *eitherEq0* represents the condition where either c or d equals zero(*16'd0*), indicating whether the calculation in the *CAL* state has been completed.



▲ Figure 3.2: Top Module

In the Top Module shown above, it includes the signal generation for *state*, *next_state*, *delay*, *eitherEq0*, and the final outputs *gcd* and *done*. The circuit diagram it represents is quite intuitive and easy to understand. The *next_state* determines which value should be assigned to the state at the next clock edge, based on the current state and various conditions. The value of *gcd* is determined when the state is in *FINISH*: whichever of *c* or *d* equals *16'd0*, *gcd* will be assigned to the other one.



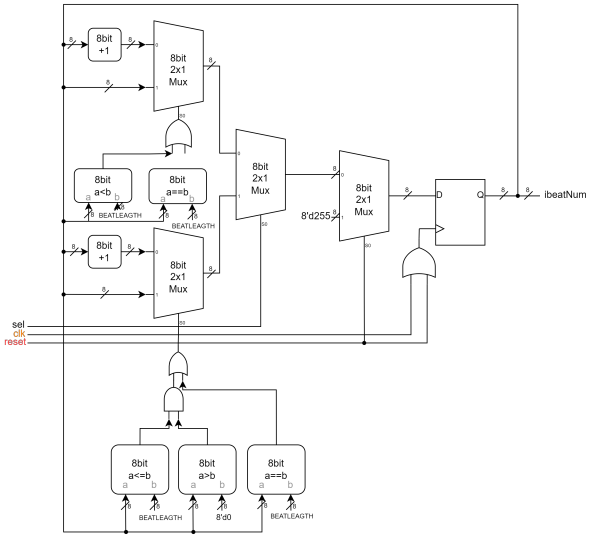
▲ Figure 3.3: *c* and *d* signal

Figure 3.3 shows the circuits for c and d, which share identical structures with only swapped signals. When in *WAIT* state and the $start==1'b1$, a and b are loaded into c and d respectively to prepare for calculation. During the calculation process, the larger number is subtracted from the smaller number until one of them becomes zero, at which point the *CAL* process ends. In the *FINISH* state, the values of c and d are maintained for output.

FPGA Demo 1: Music Box

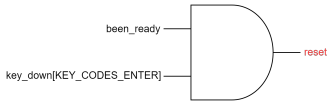
In this demonstration, we are required to make a ascending-pitch and descending-pitch music box. We modify “PlayerCtrl.v”, “Music.v”, and “Top.v” in Music_box_Sample_Code directory, and merge them with those files in the Music_box_Sample_Code directory originally and “KeyboardDecoder.v” in Keyboard_Sample_Code directory.

In “Music.v”, we modify the music from “Little Apple” to ascending pitches from C4 to C8 in order, which has 28 pitches in total.



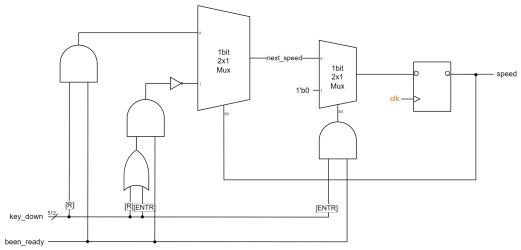
▲ Figure 4.1: modified PlayerCtrl

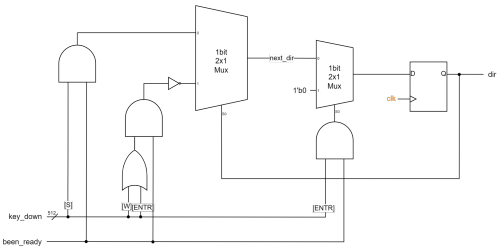
Figure 4.1 shows the modified version of PlayerCtrl. We add the mechanism of Reversing and Holding when reach the end of the direction. In order to change direction, we add an additional input port “sel” (linked with *speed*) to determine whether ascending or descending should be implemented at the moment. It can also help us to hold the note at the ends of ascending and descending without actually changing *ibeat*’s value. Note that we set the reset value of *ibeat* to *8’d255* in order to handle unwanted increment of *ibeat* because of collision of *reset* and *clk.*

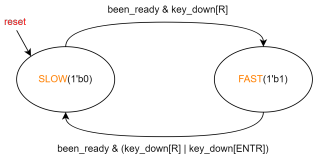


▲ Figure 4.2: *reset* signal

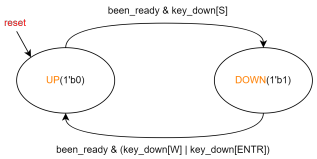
In Top module, we define *reset's* value to *been_ready && key_down[KEY_CODES_ENTER]*. As shown in Figure 4.2.





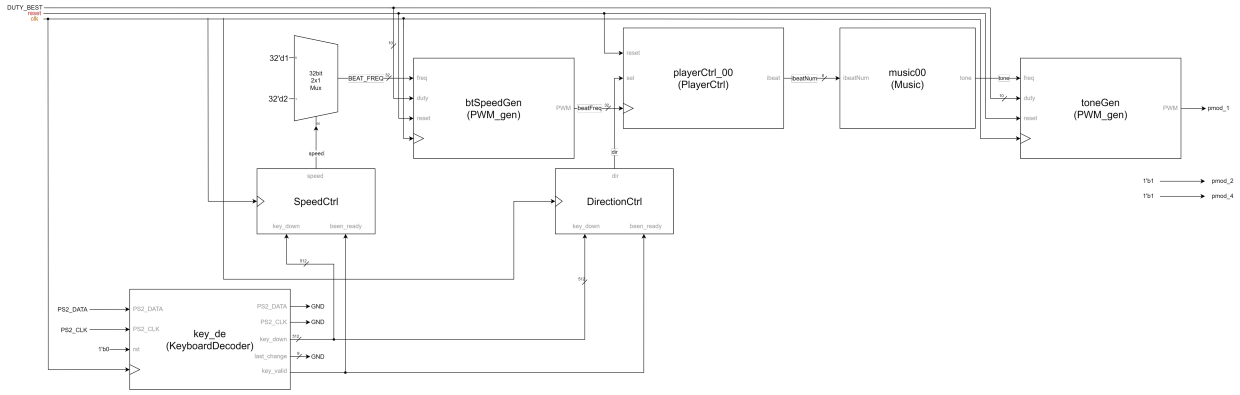


▲ Figure 4.3: SpeedCtrl and state transition diagram



▲ Figure 4.4: DirectionCtrl and state transition diagram

Figure 4.3 and 4.4 shows our design of SpeedCtrl and DirectionCtrl. They are similar to the state transition mechanism in the previous problems. We use *speed*/*dir* as state and use *next_speed*/*next_dir* as next_state.



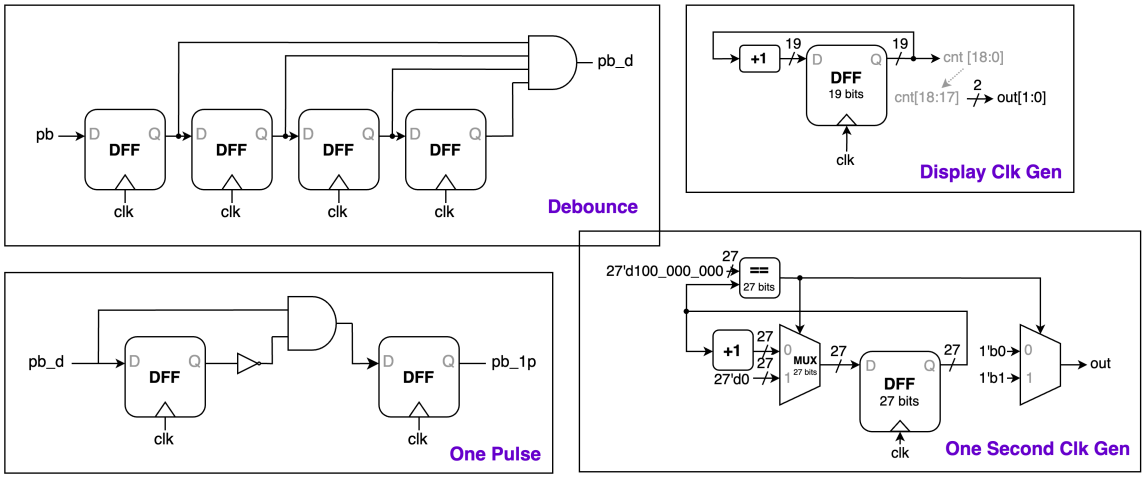
▲ Figure 4.4: Top module design

Figure 4.4 shows the overall design of our Top module.

We merge the KeyboardDecoder and the original Top module in music box's sample and modify PlayerCtrl to calculate *ibeatNum* according to the direction. Besides, we add additional signals, dir, speed, next_dir and next_speed, and update them according to keyboard states. In this way, we can handle the direction and speed changing much easier and clearer.

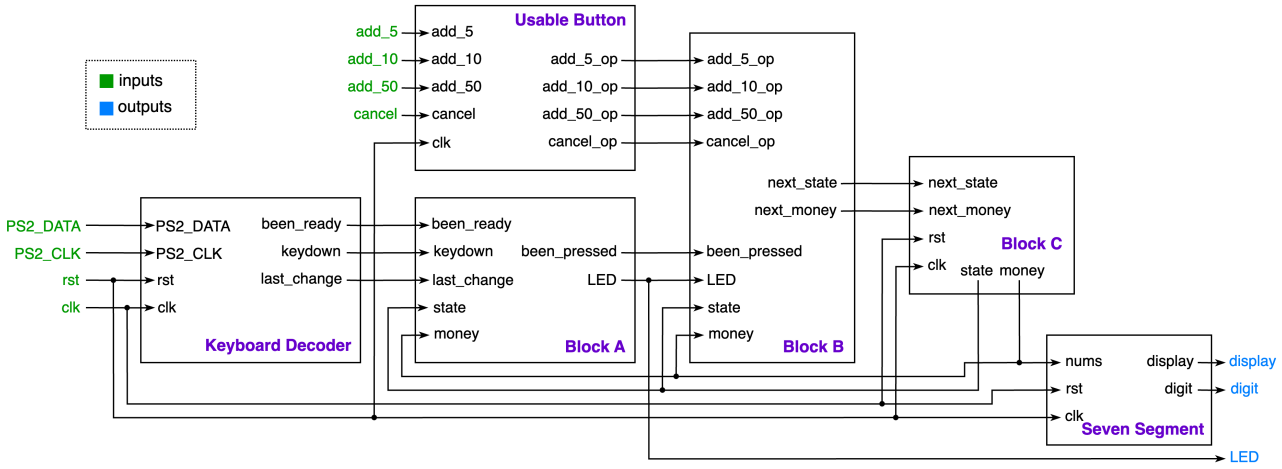
FPGA Demo 2: Vending Machine

First, let us introduce the basic components that we will use in this FPGA implementation problem. Since we will be using buttons and keyboards, we need to implement **Debounce** and **One Pulse** circuits. For Debounce, we can use a lower frequency clock, such as 100 to 500Hz, to generate an output of $1'b1$ only after receiving a prolonged stabilized signal. In this implementation, we use a simpler approach by increasing the number of DFFs in the Debounce circuit to 8 to achieve better debouncing effects. Additionally, we need a **One Second Clk Gen** because the problem requires deducting five dollars every second during the change-making simulation. Lastly, we need a **Display Clk Gen** for the Seven Segment display to properly alternate the display between the four digits of the seven-segment display at appropriate intervals.



▲ Figure 5.1: Basic Components

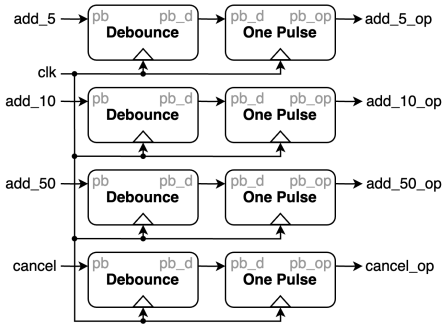
In this problem, we have only used two states, *WAIT* and *BACK*. *WAIT* represents the stage of accepting money, until a beverage is selected and the money is sufficient, at which point it will enter the *BACK* state. Additionally, when the cancel button is pressed, it will also enter the *BACK* state, initiating the change-giving stage. Once the change has been provided, it will reenter the *WAIT* state.



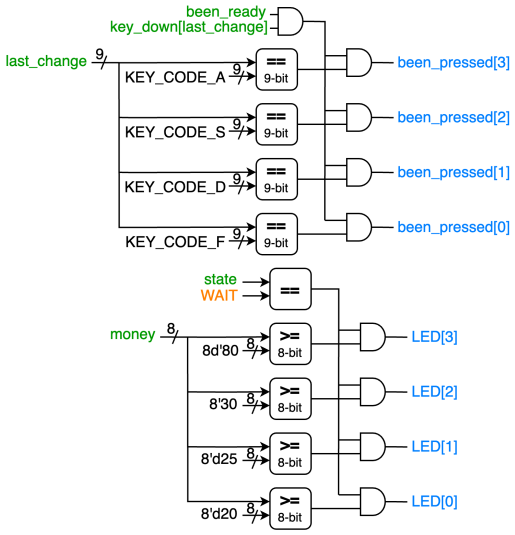
▲ Figure 5.2: Top Module

As shown in the image, this is the complete circuit diagram for this problem, which is divided into six blocks. The first block, **Keyboard Decoder**, uses the module provided by the Sample Code. Pressing and releasing a key will output a signal for one clock cycle, where *been_ready* indicates whether it is that particular clock cycle, and *keydown* is a 512-bit bus representing which keys are currently pressed, and *last_change* indicates which key was just triggered.

The next module, **Usable Button** (Figure 5.3), processes the button signals through debounce and one-pulse mechanisms. **Block A** (Figure 5.4) is the module that generates the *been_pressed* and *LED* internal signals. **Block B** (Figure 5.5) is the module that generates the *next_state* and *next_money*. **Block C** (Figure 5.6) is the module that updates the *state* and *money* on the positive clock edge. Finally, the **Seven Segment** (Figure 5.7) module displays the money on the seven-segment display according to the specified format, which we'll cover them in the following.

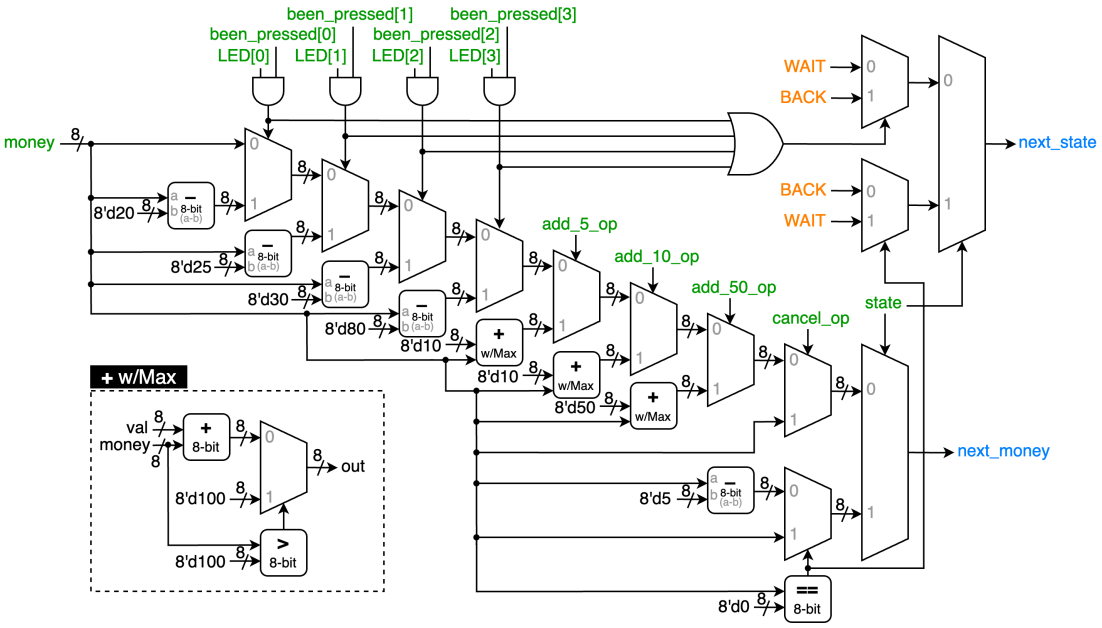


▲ Figure 5.3: Submodule – Usable Button



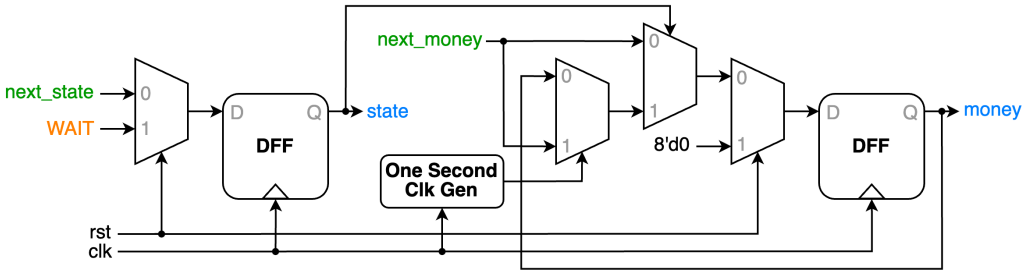
▲ Figure 5.4: Submodule – Block A

In Figure 5.4, the two signals we generate, *been_pressed[3:0]* and *LED[3:0]*, represent the following. *been_pressed[3:0]* indicates whether the purchase buttons for the 4 drinks, from most expensive to least expensive, have been pressed. *LED[3:0]* indicates which of the 4 drinks, from most expensive to least expensive, are currently within the user's budget and available for purchase.



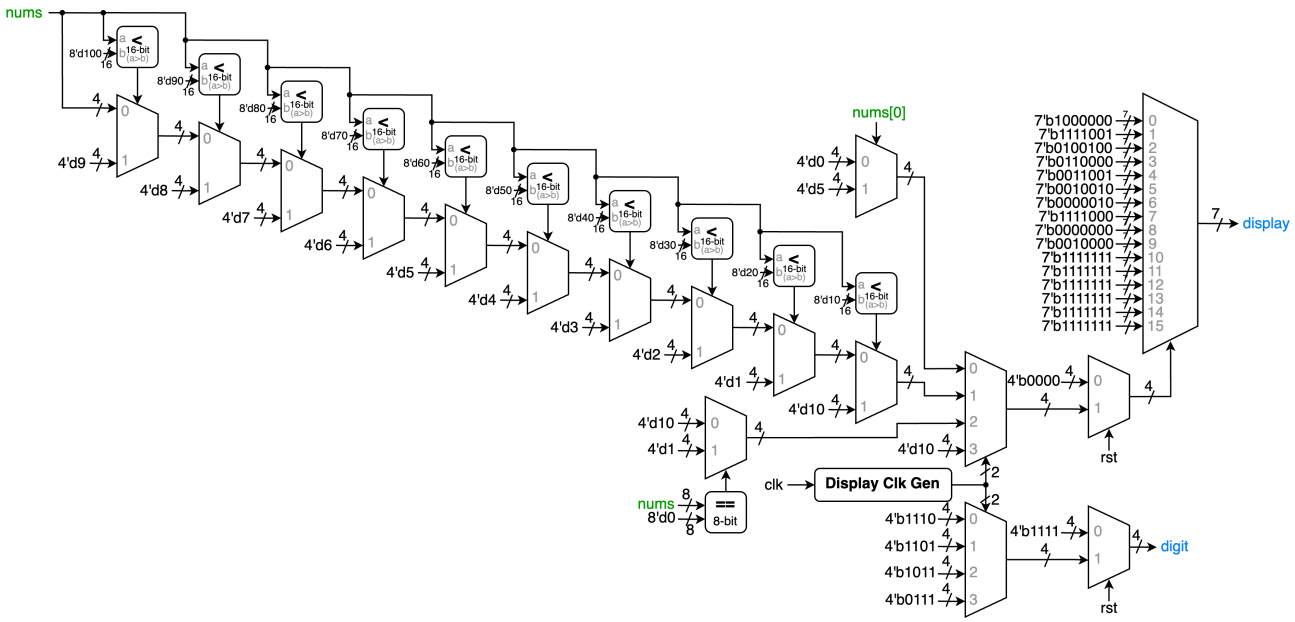
▲ Figure 5.5: Submodule – Block B

In Figure 5.5, **Block B** represents the main logic portion of the entire module. Firstly, the *next_state* section performs state transitions based on the previously explained reasons. As for *next_money*, it is dependent on the current state - in the *BACK* state, the value is decremented by *8'd5* until it reaches *8'd0*, while in the *WAIT* state, it is incremented or decremented accordingly based on whether a beverage is purchased or money is inserted. It is worth noting that an **Adder w/Max** is used here, as the maximum amount of money that can be inserted in this problem is 100 dollars, and any excess will be consumed. Therefore, the addition operation has an upper limit, while the subtraction does not, as it is included in the MUX's sel signal, which encompasses the check on whether the inserted money is sufficient to pay for the beverage. Consequently, if the money is insufficient, the subtraction operation will not be performed.



▲ Figure 5.6: Submodule – Block C

In the diagram shown in Figure 5.6, this represents **Block C**. At the clock edge, the "next" signals are read in, and some additional conditions are applied. For instance, if the current state is *BACK*, the money value is only updated from *next_money* once per second.

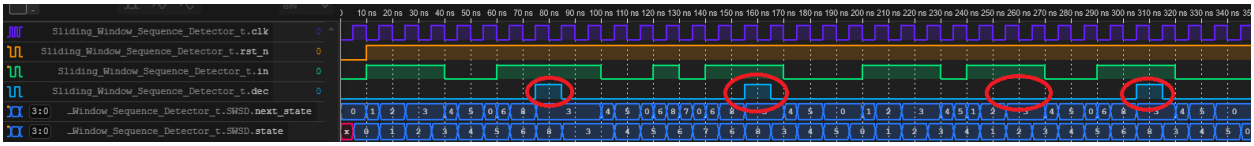


▲ Figure 5.7: Submodule – Seven Segment

Finally, Figure 5.7 depicts the **Seven Segment** display. The component in the top right is the **Number Decoder**, which can convert the *num[3:0]* input into the corresponding 7-bit signal for the seven-segment display. Preceding this is a four-to-one MUX, which allows the seven-segment display to cyclically display the four digits, ensuring that all four digits can be shown. However, for this specific problem, leading zeros cannot be displayed, so the fourth digit is permanently turned off (using *4'd10*, though *4'd10* to *4'd15* would all work with the implemented Number Decoder). The third digit will only be illuminated when the money value is *8'd100*, and will remain off in all other cases. For the first and second digits, while they should initially appear quite similar, since the ones digit can only be 0 or 5 for this problem, we can simply check for 0 or 5 and handle them individually, with only the second digit needing a more comprehensive check. The implementation of the cycling display is handled by the digit component in the bottom right. Finally, the MUX with *rst* as the sel signal before the display and digit components allows the seven-segment display to be turned off when the reset button is pressed, giving a clearer indication of the reset action.

Testbenches

A. Sliding Window Sequence Detector



In this testbench, we combine the waveform on the spec and special cases, listed below:

(circle from left to right: *C1*, *C2*, *C3*, *C4*)

Case *C1*. sequence with exactly one (01) in the middle, match

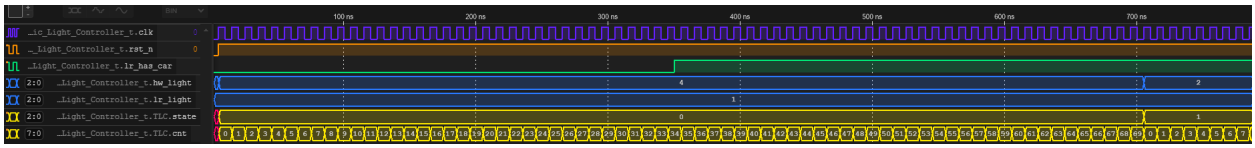
Case *C2*. sequence with repeated (01) in the middle, match

Case *C3*. sequence without (01) in the middle, mismatch

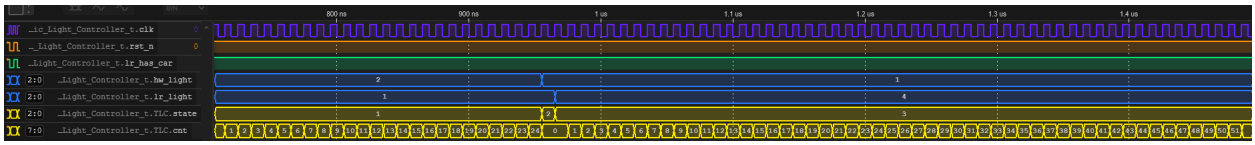
Case *C4*. just show “detecting whenever the sequence occur, setting dec to 1’b1”

B. Traffic Light Controller

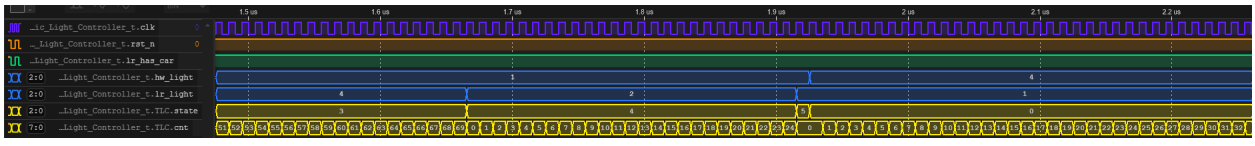
In question B, we wrote testbench by referring to the waveform in ppt and add some special cases.



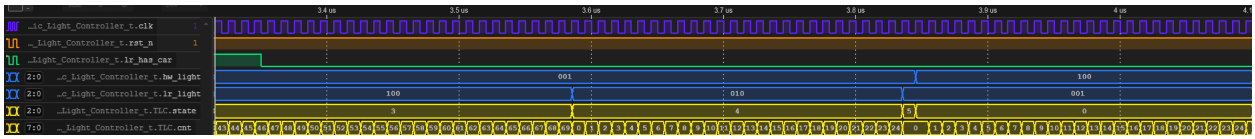
▲Set lr_has_car to 1’b1 when state is A



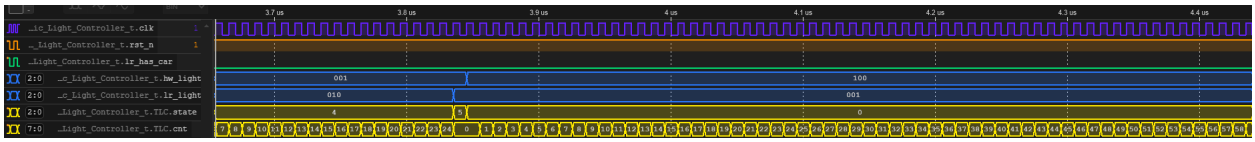
▲When reach 70 cycles, because lr_has_car is 1’b1, state changes to B.



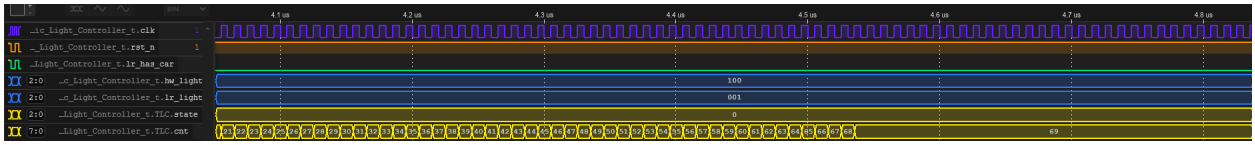
▲The next five states are just cycle-base.



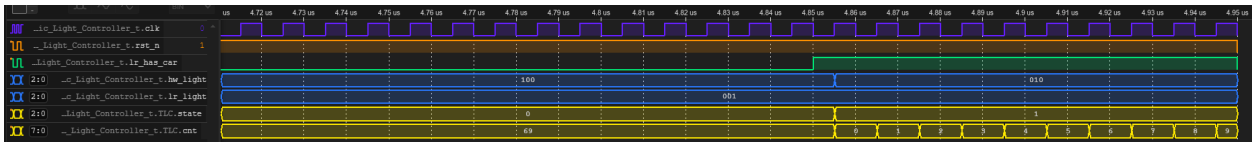
▲Change lr_has_car to 1’b0, preparing for next case.



▲lr_has_car keep begin 1’b0. When <70 cycles, cnt calculate normally.

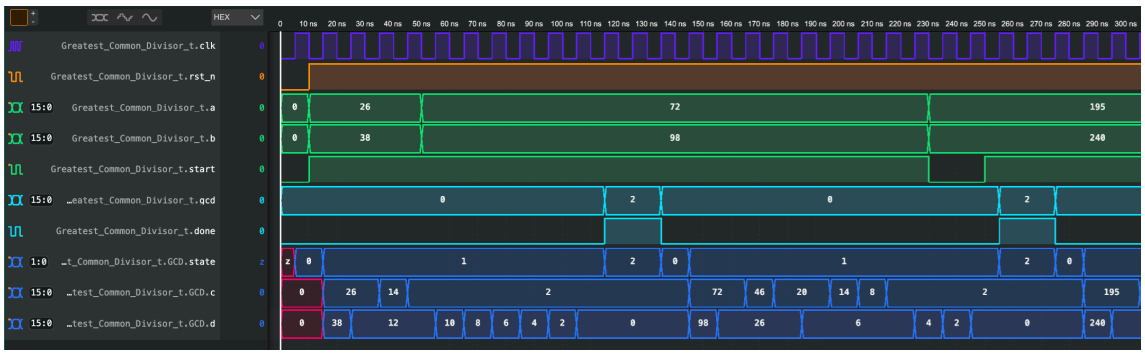


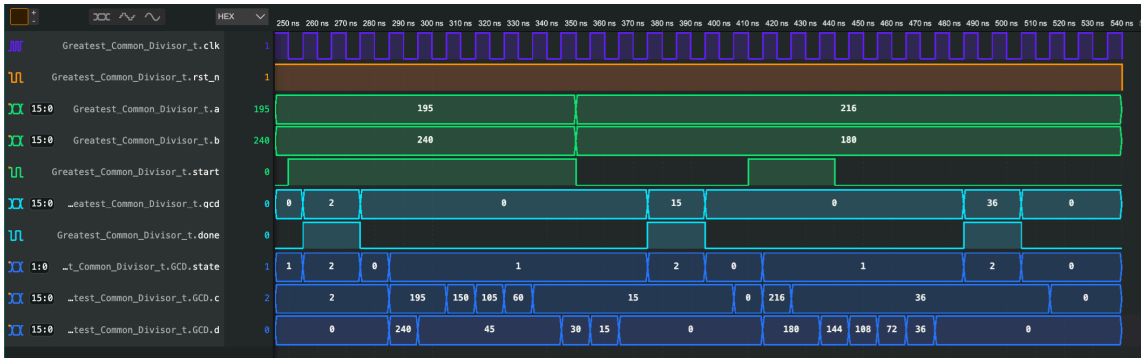
▲When >=70 cycles, cnt keep its value.



▲Because >=70 cycles, state changes upon lr_has_car changes to 1’b1.

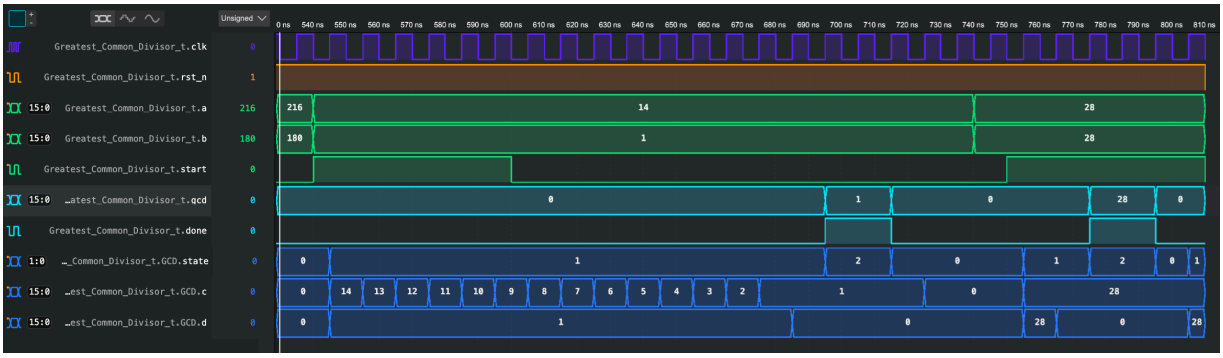
C. Greatest Common Divider





Since the problem statement did not provide any example waveforms, I tested a few sets of input values: (26, 38)->2, (72, 98)->2, (195, 240)->15, (216, 180)->36. From the waveforms, we can observe the calculation process of the internal signals c and d, as well as the changes in the *state*, which are shown in the dark blue traces. Additionally, the done output is correctly asserted for two clock cycles.

On top of those, we can see when the current state is *CAL*, the alteration of *start* or input numbers a and b will not affect the calculation process.



Finally, we test two edge case. One is that one of the number is 1, we can see the result in the waveform is correct. The other is having the same number as two inputs, the result is also correct.

What have we learned from Lab 5?

From Lab1 to Lab4, we have completed many interesting small modules, but in this Lab5, the addition of a keyboard and a speaker has expanded the controllable space. Additionally, through this Lab5, we have gained a more thorough understanding of FSM usage. Furthermore, in the two demos, we have utilized techniques learned in the past, such as the seven-segment display, Debounce, and One Pulsed, among others.

Contributions

謝佳晉：

wrote Verilog modules: Q1, Q2, Q3, Q4, D1
drew diagram: Q1, Q2, D1
wrote report: Q1, Q2, D1, tb

范升維：

wrote Verilog modules: Q1, Q2, Q3, D2
drew diagram: Q3, D2
wrote report: Q3, D2, tb, what we learned
organized whole report