Hardware Design and Lab: Lab5

111060013 EECS 26' 劉祐廷

Catalog

1. Advanced Question: Sliding Window Sequence Detector

A. Finite State Diagram

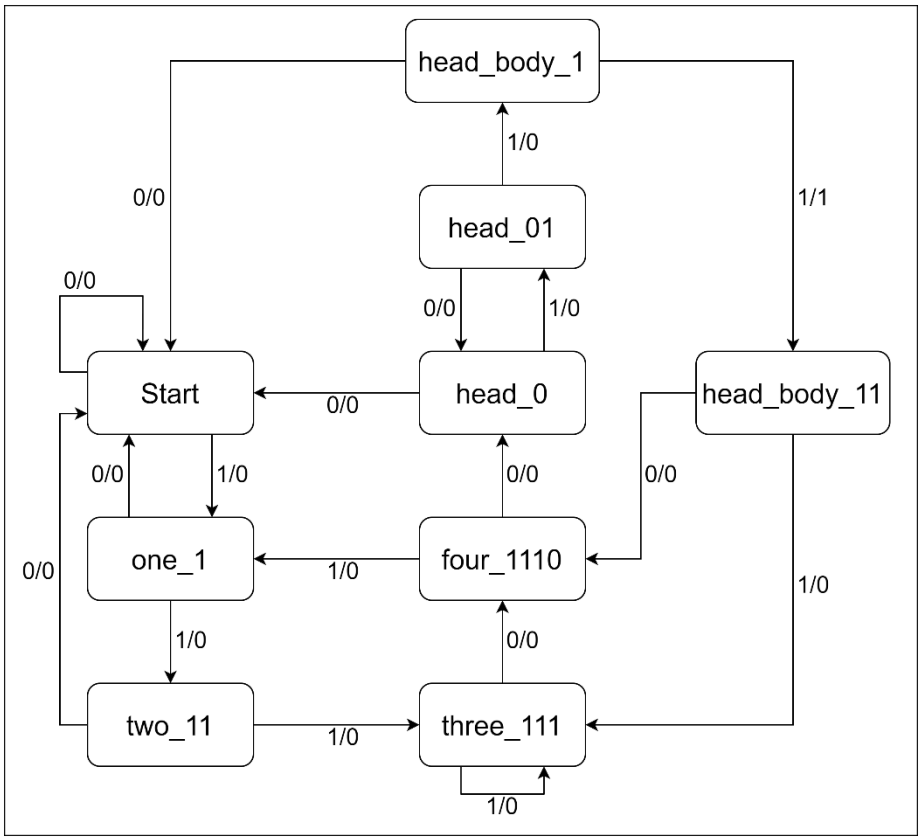


Figure 1.1

B. Explanation

Start: Detect 0 bit that match the pattern.
one_1: Detect the first bit **1** of the pattern.
two_11: Detect the second bit **1** of the pattern.
three_111: Detect the third bit **1** of the pattern.
four_1110: Detect the fourth bit **0** of the pattern.
head: It means 1110.
head_0: Detect **0** of the "several **01**".
head_01: Detect **1** of the "several **01**".
body: It means "several **01**".
head_body_1: Detect the second-to-last bit **1** of the pattern.
head_body_11: Detect the last bit **1** of the pattern

C. Testbench

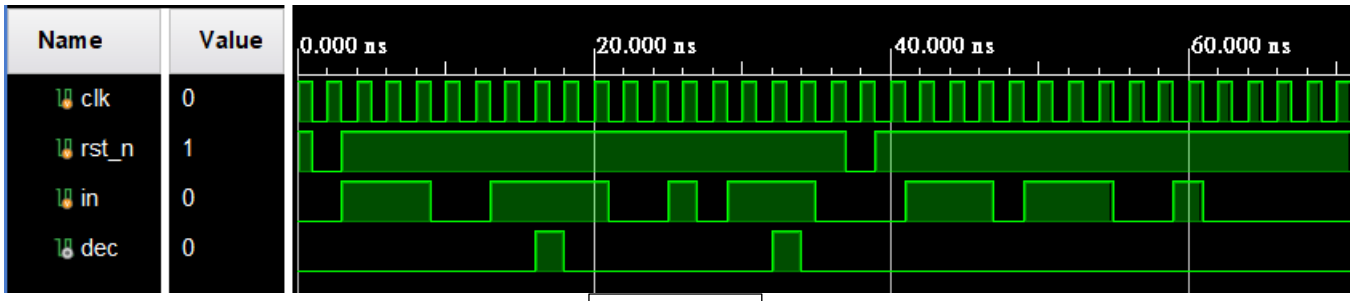


Figure 1.2

I use the testcase of the lab slide to check if there is something wrong and it seems that the design works properly.

2. Advanced Question: Traffic Light Controller

A. Finite State Diagram

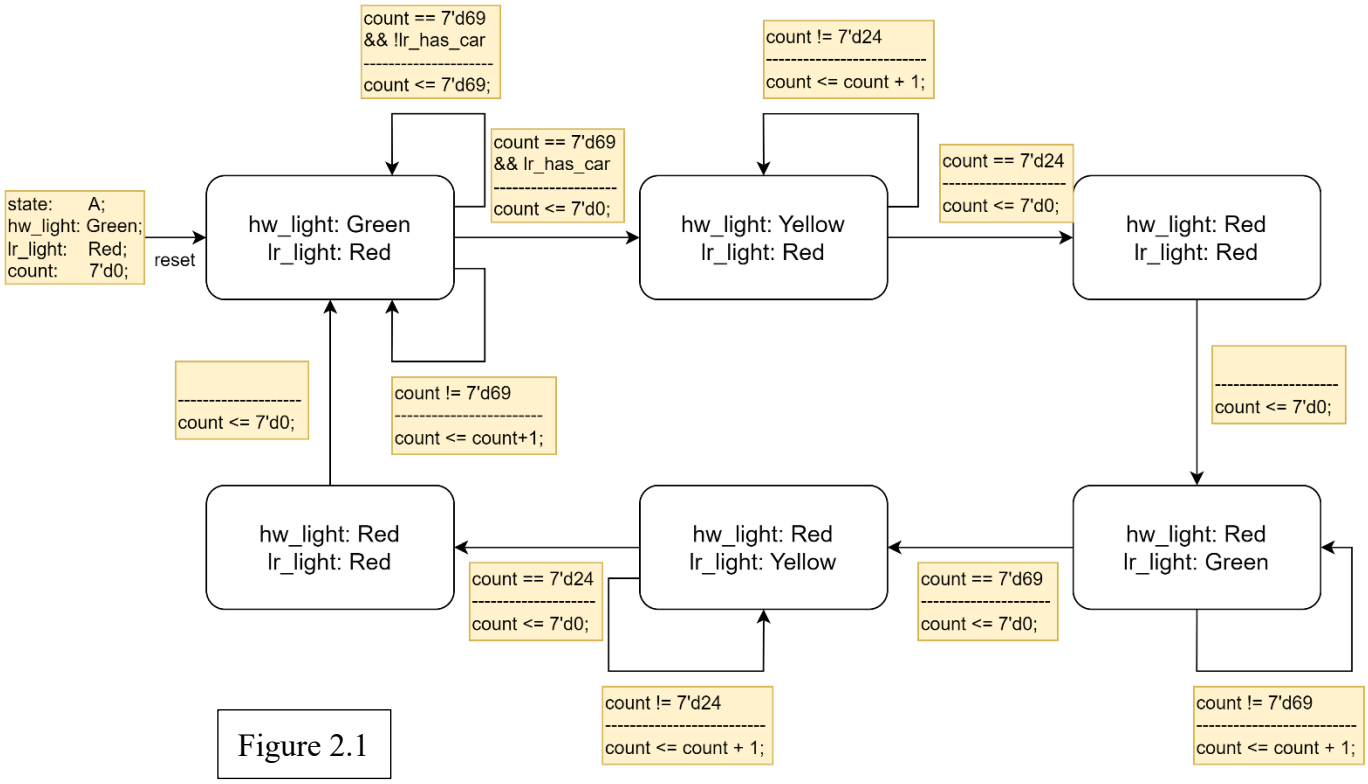


Figure 2.1

B. Explanation

I designed my circuit according to **Figure 2.1**. **count** remains **7'd69** is **lr_has_car == 1'b0** while the state is **hw_light: Green, lr_light: Red**, so that after counting 70 clock cycles, if **lr_has_car == 1'b1**, **hw_light** can change into yellow immediately.

C. Testbench

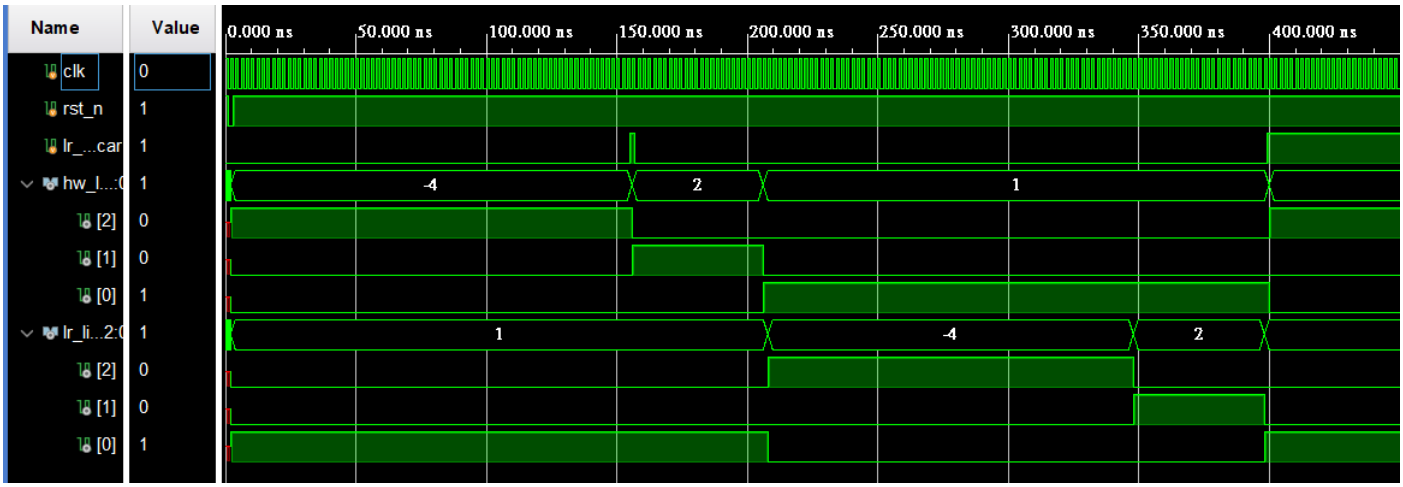


Figure 2.2

First, I test the case that **lr_has_car** is pulled up after 75 clock cycles, and it seems that nothing is wrong as **Figure 2.2** shows.

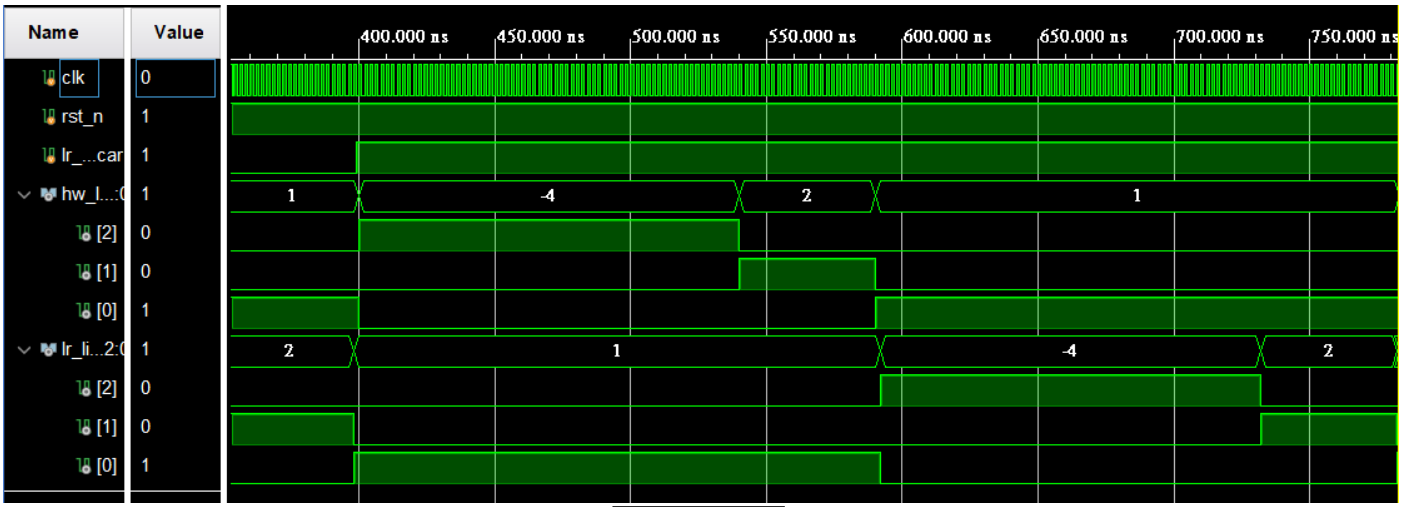


Figure 2.3

Second, I test the case that **lr_has_car** is pulled up from the beginning, and everything is correct as **Figure 2.3** shows.

3. Advanced Question: Greatest Common Divisor

A. Block Diagram

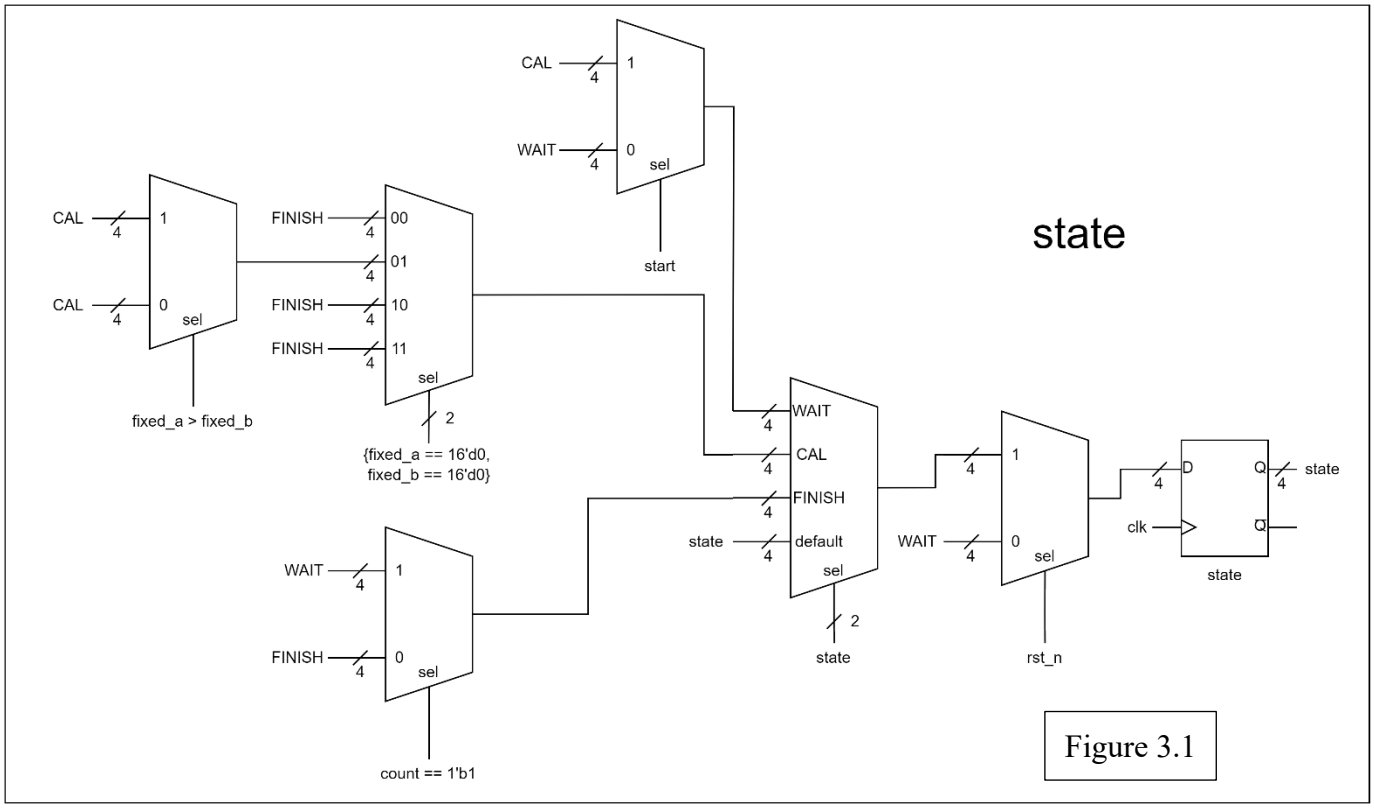


Figure 3.1

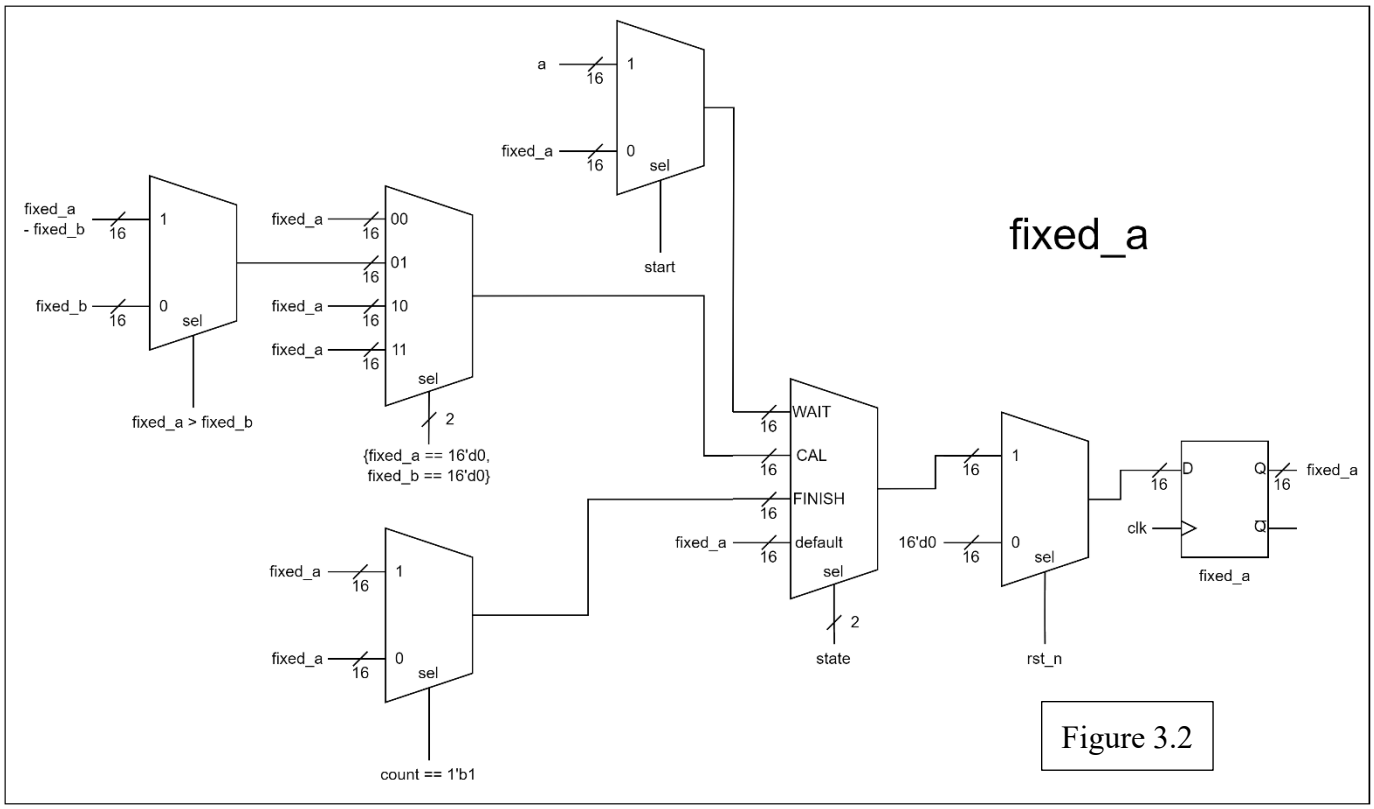


Figure 3.2

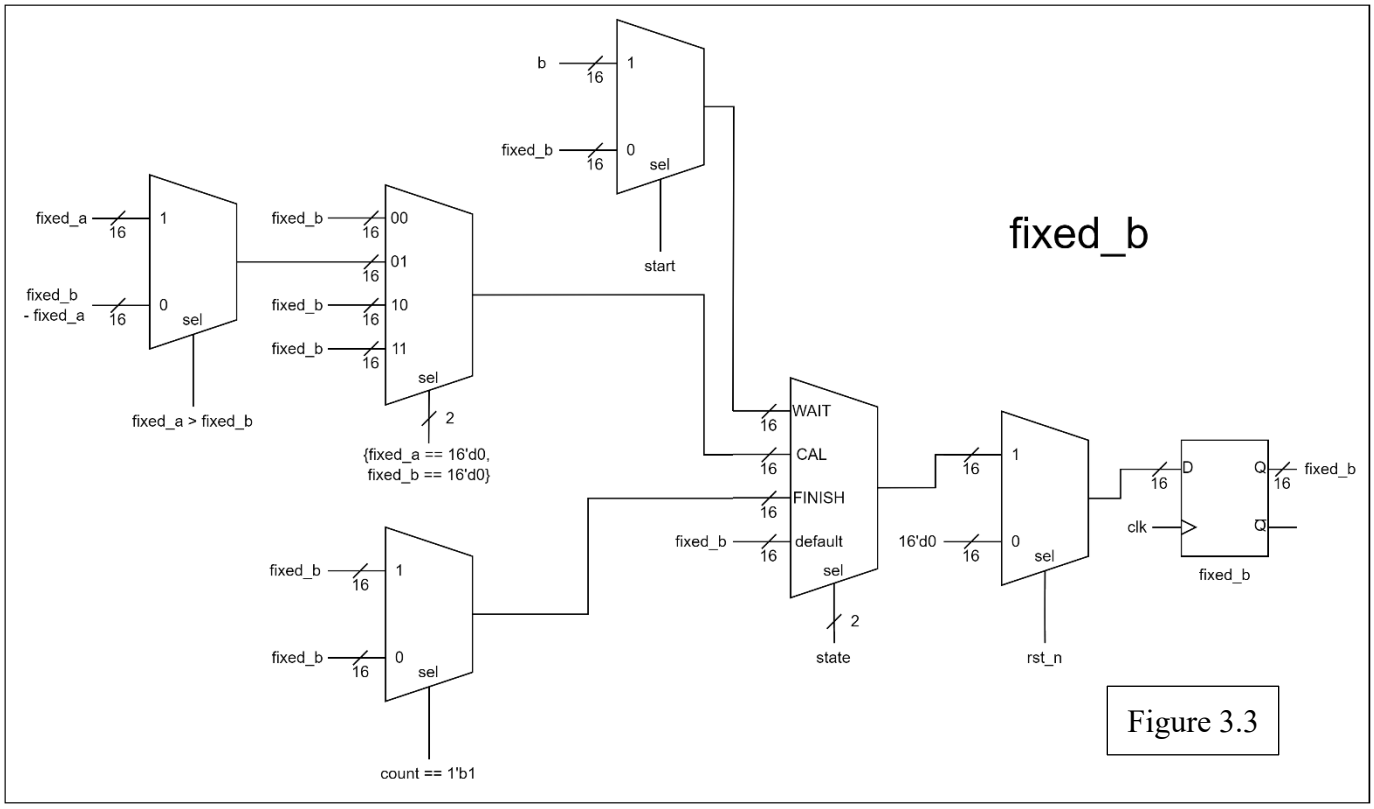


Figure 3.3

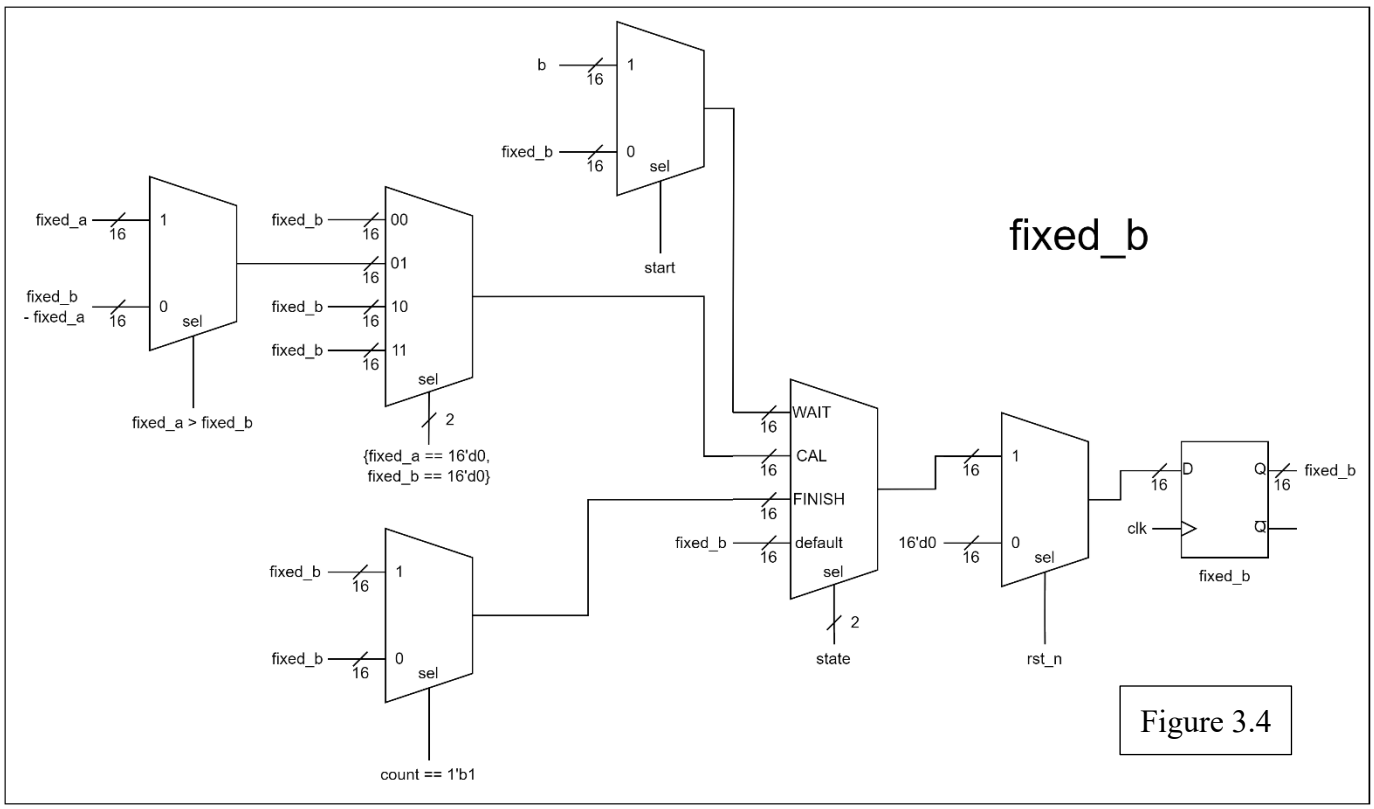


Figure 3.4

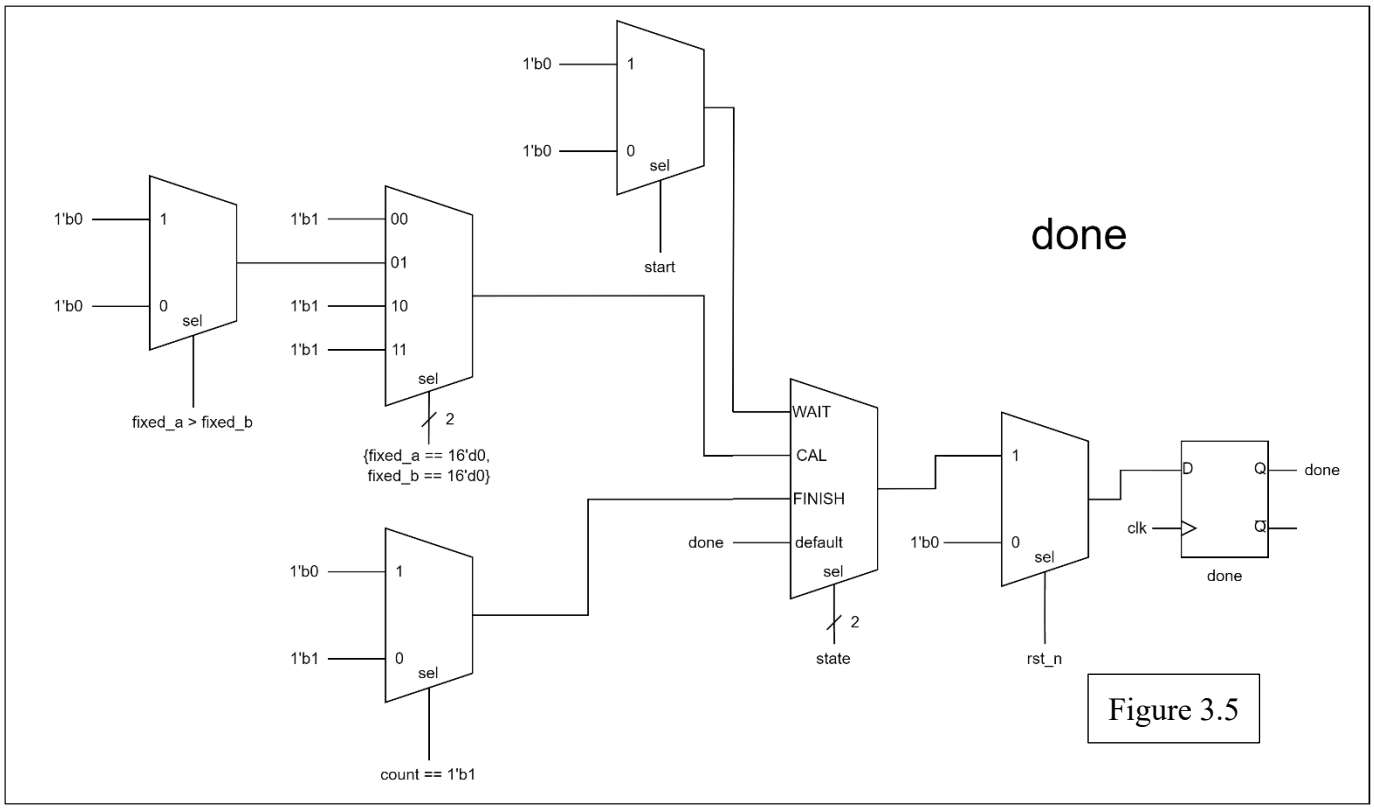


Figure 3.5

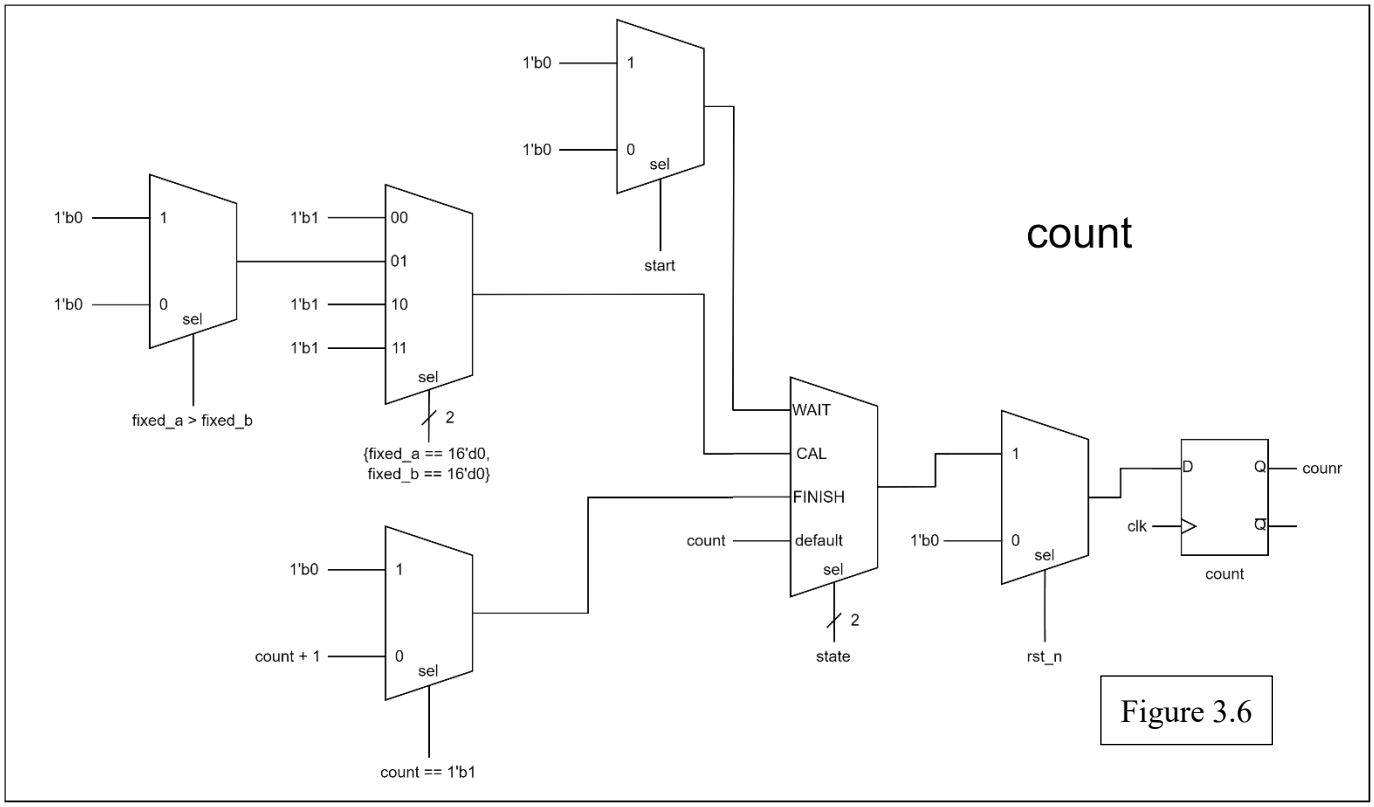


Figure 3.6

B. Finite State Diagram

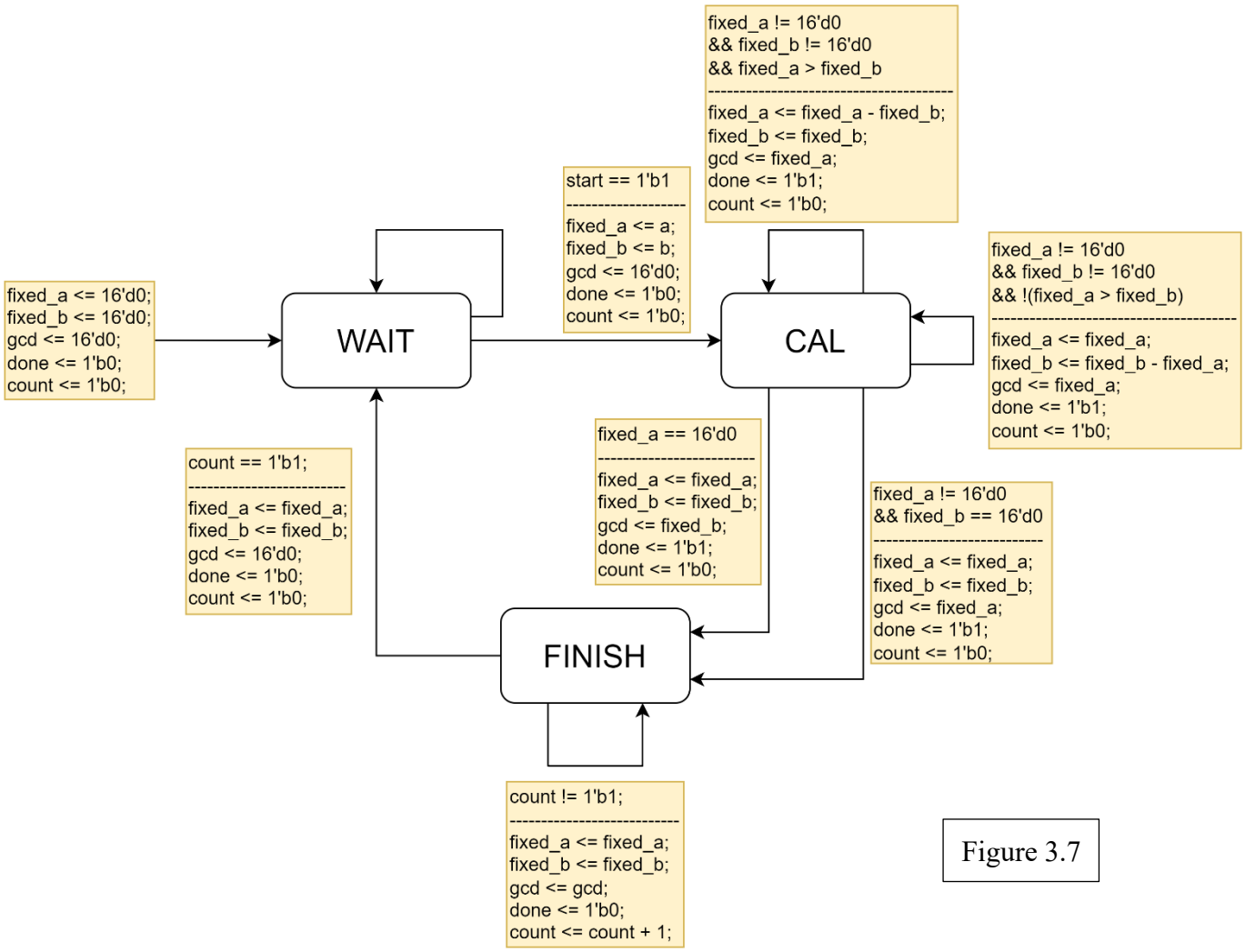


Figure 3.7

C. Explanation

In this problem, most of rules are provided in the lab slide. Therefore, I just follow the slide and use moore machine to realize the circuits. **Figure 3.1** to **Figure 3.6** show that the circuits I designed. Because **a, b** can change at any time, I use **fixed_a** and **fixed_b** to store the value while **state** change from **WAIT** to **CAL**.

D. Testbench

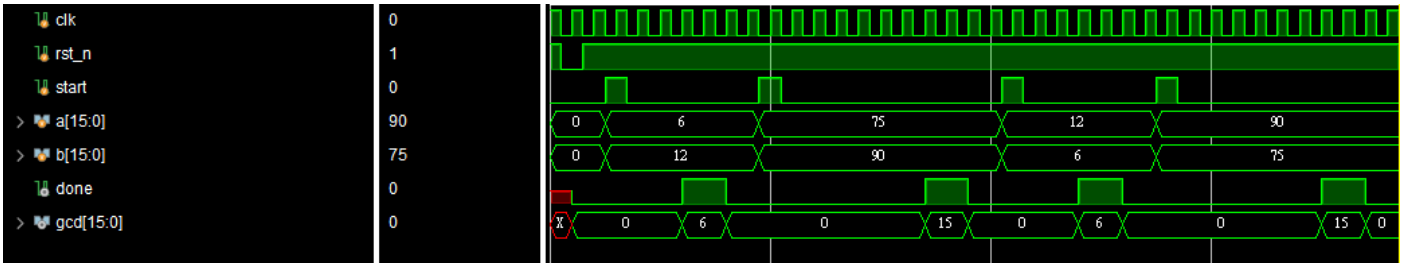


Figure 3.8

To test the design, I set **a** and **b** to different numbers and then change them and calculate them again. In this way, I can test if the Euclidean algorithm works properly. It seems that everything is right as **Figure 3.8** shows.

4. Advanced Question: Booth Multiplier

A. Finite State Diagram

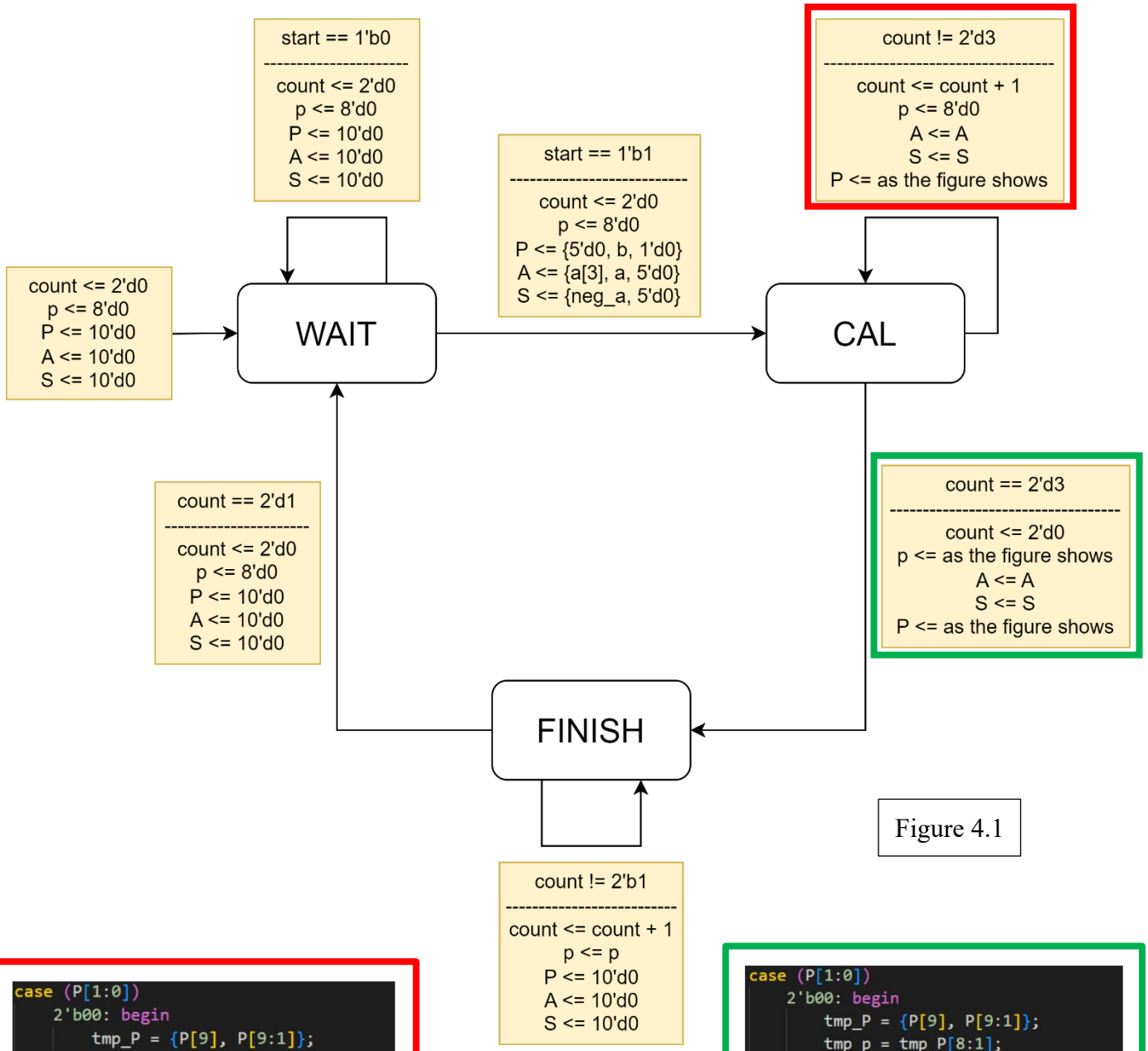


Figure 4.1

```
case (P[1:0])
    2'b00: begin
        tmp_P = {P[9], P[9:1]};
    end
    2'b01: begin
        tmp_P = {sumPA[9], sumPA[9:1]};
    end
    2'b10: begin
        tmp_P = {sumPS[9], sumPS[9:1]};
    end
    2'b11: begin
        tmp_P = {P[9], P[9:1]};
    end
    default: begin
        tmp_P = P;
    end
endcase
```

```
case (P[1:0])
    2'b00: begin
        tmp_P = {P[9], P[9:1]};
        tmp_p = tmp_P[8:1];
    end
    2'b01: begin
        tmp_P = {sumPA[9], sumPA[9:1]};
        tmp_p = tmp_P[8:1];
    end
    2'b10: begin
        tmp_P = {sumPS[9], sumPS[9:1]};
        tmp_p = tmp_P[8:1];
    end
    2'b11: begin
        tmp_P = {P[9], P[9:1]};
        tmp_p = tmp_P[8:1];
    end
    default: begin
        tmp_P = P;
        tmp_p = p;
    end
endcase
```

B. Explanation

In this problem, I designed a Moore machine according to **Figure 4.1**. Because Booth method needs the complement of the multiplicand, I use 5 bits for multiplicand to prevent from the error cause by overflow while it is equal to -8.

C. Testbench

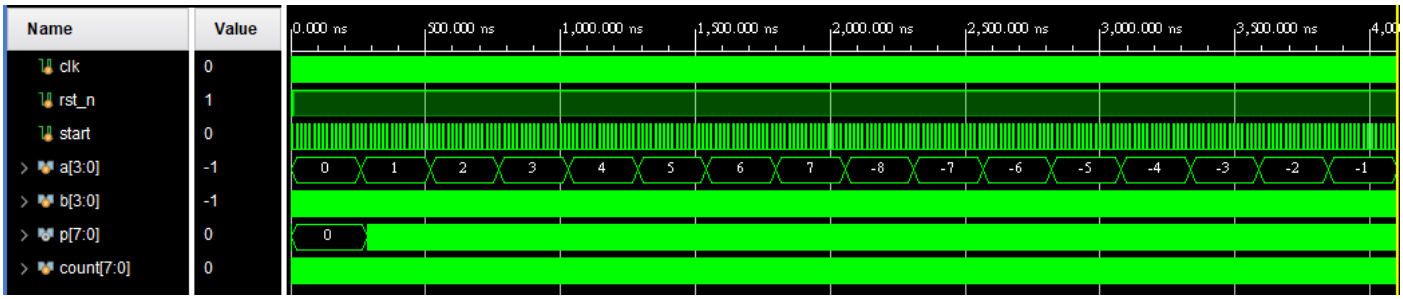


Figure 4.2

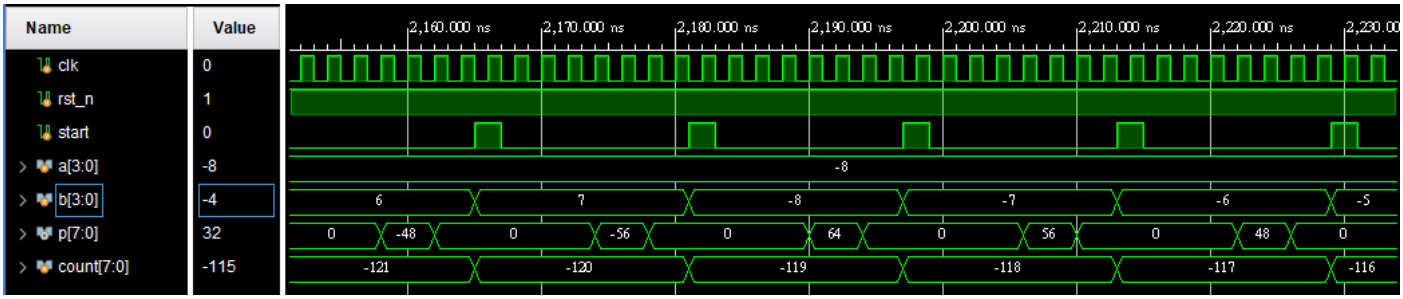
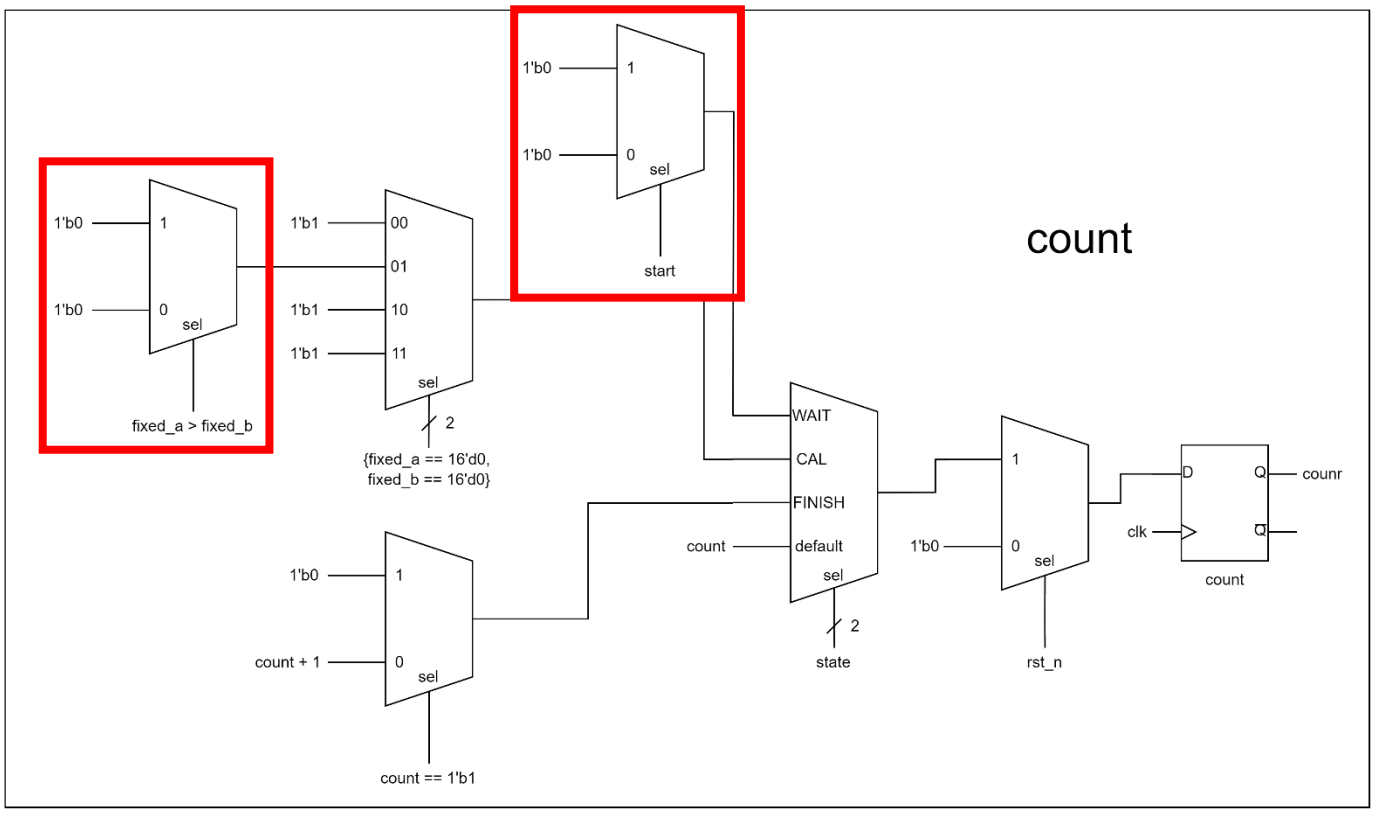


Figure 4.3

As **Figure 4.3** shows, I test every possible case to check if my design is correct or not. And for the overflow problem mentioned in **Explanation** above is prevented. We can see that while -8 is the multiplicand, the results are correct.

5. What I Have Learned

In this lab I've found that after I realized the circuits according to the finite state diagram, there may be some redundant multiplexers in my circuits, such as whatever the selection signal is 1 or 0, the output is the same. After drawing out the block diagram, I can clearly see that which mux is redundant and can be deleted. However, close to the end of the semester, there are a lot of projects that I need to do, which steals my time away. Therefore I didn't have time to simplify my code to reduce redundant mux.



For example, in **Figure 3.6**, these two mux can be replaced by a wire.