EE312 Lab #2 Report

20180610 정의준

1. Introduction

In this lab, we have to design a vending machine. The machine gets coins and button inputs from the user. Then, the machine should behave like a typical vending machine. For example, when the coin is inserted, the machine should display all the products that the user can buy. There are other features that should be implemented too, such as returning the leftover coins in specific situations.

By working on this lab 2, we are able to learn how `always @(*)` statement and `always @(posedge clk)` statement are different, and how the blocking (=) and non-blocking (<=) operations should be used in the combinational and sequential logics. Also, we can learn how to make a simple finite state machine (FSM) with Verilog.

2. Design

The overall design of Lab 2 is very simple, because we only have to implement one module that does all the features of a vending machine. The input and output wires of the module is shown in the figure below.

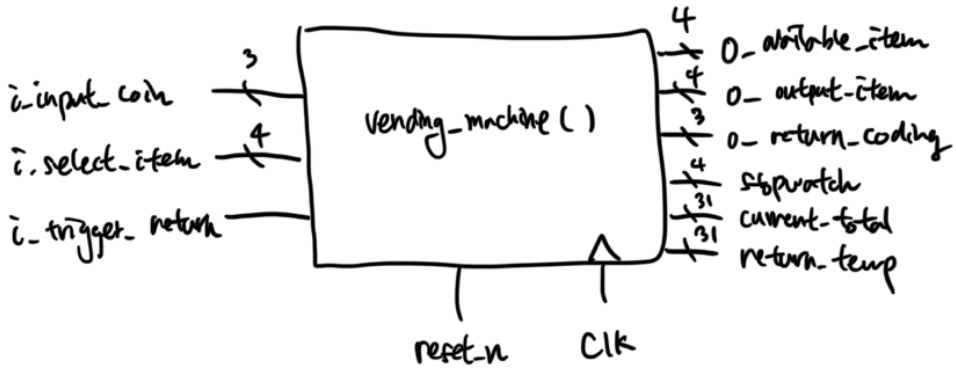

Figure 1 I/O ports of the vending machine module

For the inputs, two vectors (each for representing coin and button input) and one single-bit input is given. We don't have to care about the multiple inputs coming in at once, since the comment from the skeleton code says so.

reset_n is used as a power-on signal of the machine, and clk is used as a clock.

For the outputs, we have three vectors for displaying available items, output item, and the coin return. For the outputs (stopwatch, current_total, return_temp) which are not used in the testbench file, I just ignored them.

Because I wanted the machine to be FSM, I defined the state of the machine to be "the current total coin value." This was possible since we assumed that we will have an infinite number of coins and items. If it was not the case, I would have defined the state of the machine to be concatenation of the vectors that represent the total value and the total number of coins and items. The detailed description about how the sequential and logic gates are made will be specified in the implementation section.

3. Implementation

- Input part (Combinational Logic)

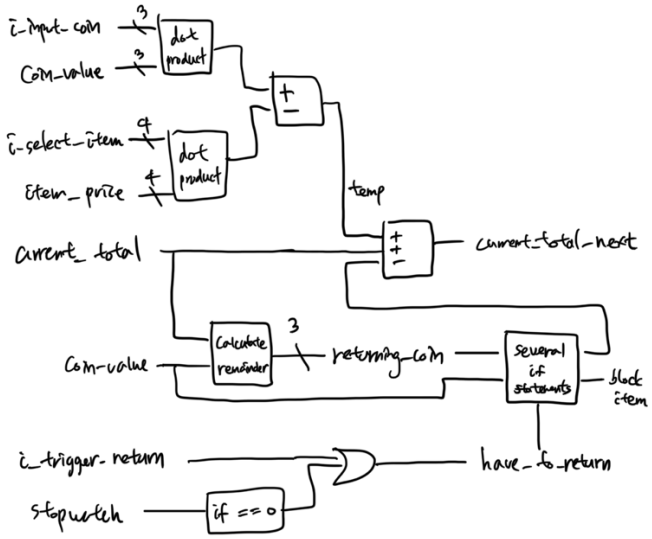

Figure 2 Overall scheme of the input part of the combinational logic circuit

First, I implemented the part where we calculate the next total value (current_total_next). Then, the current_total value will be updated to current_total_next on the rising edge by the sequential logic. To calculate current_total_next, we have to consider three situations: coin insert, item selection, and coin return.

The first "dot product box" calculates the value of the input coin and adds it to the current value. Obviously, in reality, the dot product box consists of number of products and sums.

The ‘box’ notation for the dot product is just for the simplicity. In reality, the box consists of several product and sum operators.

In the same way, we can calculate the total value of ordered items and subtract it from the current value.

‘have_to_return’ control signal activates when the stopwatch becomes zero or the return trigger is triggered.

The implementation for the coin return part is a bit tricky. First, the circuit is always converting the current value (which is int) into a bit vector ‘returning_coin’ that represents how many of each coin are needed to represent the current value. For instance, if current_total is 1800, returning_coin will be [3, 1, 1] since 3*100 + 1*500 + 1*1000 = 1800. So, if the ‘have_to_return’ control signal is enabled, we subtract the value of the coin that will soon be returned from the total.

Also, the ‘block_item’ vector is a temporary register that controls the order of the coin returns. In my code, the coin with a larger value is returned first. So, if returning_coin is [3, 1, 1], the machine will return the coin in the following order in each rising edge: 1000, 500, 100, 100, 100. I will put the code for the ‘several if statements’ part in the below, but I think it is inconsequential (which means not important).

```
if (returning_coin_2 > 0) begin
    block_item_1 = 1;
    block_item_0 = 1;
    if (have_to_return)
        current_total_nxt = current_total_nxt - kkCoinValue[2];
end
else begin
    block_item_1 = 0;
    if (returning_coin_1 > 0) begin
        block_item_0 = 1;
        if (have_to_return)
            current_total_nxt = current_total_nxt - kkCoinValue[1];
    end
    else begin
        if (returning_coin_0 > 0) begin
            block_item_0 = 0;
            if (have_to_return)
                current_total_nxt = current_total_nxt - kkCoinValue[0];
        end
        else begin
            block_item_0 = 0;
            block_item_1 = 1;
        end
    end
end
```

Figure 3 Code for returning the coin in order

- Output part (Combinational Logic)

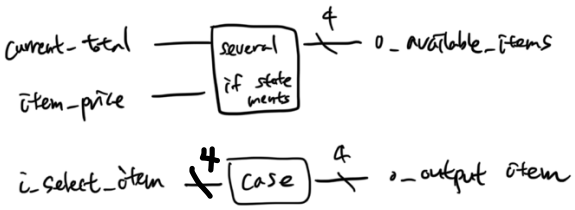

Figure 4 Scheme for output circuit

Luckily, the output part is much simpler. The first box illustrates where the machine compares the item prices and the current total value and displays the items that the user can buy. The second box is for selecting item based on the pressed button.

- Sequential Logic

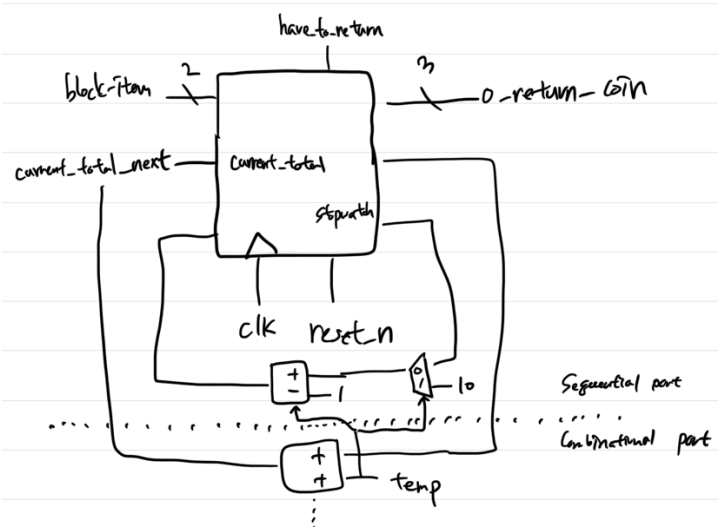

Figure 5 Sequential logic circuit

Finally, the sequential part, where the circuit is activated on a rising edge of the clock. First, if reset_n is not activated, the machine resets current_total, stopwatch, and have_to_return flag.

If reset_n is activated, the state is updated by the code `current_total <= current_total_nxt;`. Then, it decreases stopwatch based on the current total value and the temp value. The idea is that temp value becomes non-zero if the user puts in the coin, or buys an item. If temp is not one, the stopwatch is reset to 10. The stopwatch should not be working when the current value is zero.

Then, the coin return part. If the have_to_return flag is active, the machine makes an element of o_return_coin into 1 based on the value of 'block_item'. I arbitrarily made four cases, where

the coin returns 1000 coin, 500 coin, 100 coin, and no coin. The code for this case statement is attached below. You can see that the four cases of 'block_item' match with **Fig 3**.

```
if (have_to_return) begin
    case ({block_item_0, block_item_1})
        {1'b1, 1'b1}:
            o_return_coin <= 3'b100;
        {1'b1, 1'b0}:
            o_return_coin <= 3'b010;
        {1'b0, 1'b0}:
            o_return_coin <= 3'b001;
        default:
            o_return_coin <= 3'b000;
    endcase
```

Figure 6 code for returning coins

4. Evaluation

My vending machine only changes states by the code `current_total <= current_total_nxt;`, which lets the structure of the code simple. Also, I tried not to define new redundant registers that can harm the simplicity and the intuitiveness of the code.

I tried to use minimum number of conditional statements in the `always @(posedge clk)` part as possible. This is the reason why there are so many if statements in the combinational logic part. This way of designing helped me to prevent the machine making unintended hazards.

My implementation has passed 10/10 tests of the vending_machine_TB.v.

5. Discussion

At the first time when I saw the skeleton code, I had a hard time understanding the scheme and the role of individual registers because I was not used to the Verilog language. Also, even after I understood the code, I had a hard time about deciding how I should each state of this FSM. Luckily, because we didn't have to care about the amount of item and coin, I simply set the state equal to the value of total money, and finished the implementation.

Also, at the first time, I put too many conditional statements and temporary registers in the rising-edge part. So, the code did not work as I expected. This was because I didn't understand the difference between the difference between Verilog and the other programming languages. But, after I understood the concept that the Verilog is "designing a virtual logic circuit," I was

able to simplify the rising-edge part and make suitable logic gates and control signals to implement the vending machine.

6. Conclusion

In conclusion, I was able to finish this lab after I understood the nature of the Verilog language and designing a virtual logic circuit. Specifically, I was able to understand and know the differences between 'sequential' and 'combinational' circuit. I saw that two circuits should be carefully designed to make no hazards. Drawing circuits on a paper before coding helped me to efficiently design the circuit.

Also, I am happy that I was able to implement this lab with a vending machine that only changes states when the code `current_total <= current_total_nxt;` is run. I wonder whether there are other ways we can implement this machine with much less registers and conditional statements.