EECS 2070 02 Digital Design Labs 2020 Lab 5
學號：108062281　姓名：莊晴雯

1. 實作過程

In this lab5, we are required to design a Fancy Mask Vending Machine that has 5 states, namely INITIAL, DEPOSIT, AMOUNT, RELEASE, and CHANGE. The detailed spec for each state are as follows:

1) INITIAL state (clk/2^16)
   The first state of the vending machine that shows 0000 on the 7-segment display. The vending machine will be in this state when being programmed for the first time, after being reset, or after the all the changes were returned. The vending machine will directly go to the DEPOSIT state here.
2) DEPOSIT state (clk/2^16)
   In this state, customer can deposit money that is going to be used to buy masks. The money deposited is either per $5 or per $10, no other amount can be deposited other than these two. Customer can deposit money using money_5 (BTND) and money_10 (BTNU) button. The maximum balance that the customer can deposit is $50, even when the balance is $45, when customer presses money_10 button, the balance will still be $50. The price for each mask is $5 and in this state, the two rightmost 7-segment display will show the balance that the customer deposits and the two leftmost 7-segment digits will show the maximum amount of masks that the customer can buy, corresponding to the amount of money that the customer deposits. Note that the maximum amount of masks that can be bought are 9 pieces, even when the balance is $50.
   From this state, the customer can cancel the purchase by pressing button cancel (BTNC) and the machine will directly go to the CHANGE state to return all the deposited money. In addition, if the customer wishes to proceed, the customer can press check button to go to the AMOUNT state to adjust the number of masks that the customer wants to buy. Note that the customer can only operate those two buttons only if the balance is greater than 0. If balance is equal to 0, the machine will stay at the DEPOSIT state.

3) AMOUNT state (clk/2^16)

   In this state, the customer can adjust the number if masks that the customer wants to buy. The 7-segment display will initially show the maximum number of masks alongside with the amount of balance that the user inputs, corresponding to the previous DEPOSIT state. The two leftmost digits of the 7-segments display will then show the amount of masks while being adjusted by the customer. That is, if the customer presses count_down button (BTNL), the number of masks will decrease by one for each time the button was pressed. When, the number of mask reaches 1 and the customer keeps pressing count_down button, the number of mask will go back to the maximum amount of masks that the user can by corresponding to the deposited money. From this state, the customer can cancel the purchase by pressing button cancel (BTNC) and the machine will directly go to the CHANGE state to return all the deposited money. In addition, if the customer wishes to proceed, the customer can press check button to go to the RELEASE state.

4) RELEASE state (clk/2^27)

   In this state, it simulates the mask(s) that are dropping from the vending machine, indicating that the customer successfully bought the mask with the desired amount. This state will last for 5 seconds, and within this 5 seconds, the 16 LED lights will turn on and off repeatedly for 1 second each time, and the 7-segment display will show alphabetical message for the customer, in this case, I showed “TAKE”. After 5 seconds, the machine will automatically go to the CHANGE state to return the change for the customer, and all the LED lights will turn off at this time.

5) CHANGE state (clk/2^27)

   In this state, the changes after buying (or cancelling the purchase) will be returned to the customer by the calculation: (balance-(amount*5)). The sequence to return the change is to decrease by tens first if there are more than $10 to return, then the $5. The 7-segment display will display the amount of money decreasing which simulates dropping coins, with the frequency of clk/(2^27). After returning all the change, the machine will automatically go to the INITIAL state.

For this lab, I used several additional variables to support my design, namely:
Debounce and one pulse variables:

- money_5_debounced;
- money_10_debounced;
- cancel_debounced;
- check_debounced;
- count_down_debounced;
- money_5_one_pulse;
- money_10_one_pulse;
- cancel_one_pulse;
- check_one_pulse;
- count_down_one_pulse;

Clock with different frequencies:

- clk_div_16;
- clk_div_27;
- clk_div_29;

- ACT_DIGIT: takes the value of that will be used as the case for displaying the seven segment display.
- refresh_counter: uses 20 bits, increase by 20'd1 every time a positive edge of a clock occurs, and the two most significant bits will then be assigned to refresher.
- refresher: takes the value of the two most significant bits of the 20-bit refresh_counter.
- state: used to determine the current state of the machine and to determine what actions can be executed in the state.
- next_state: used to determine the next state of the current state if several conditions to change state is met.
- balance_digit1: stores the first digit of the amount of money deposited from the customer.
- balance_digit2: stores the first digit of the amount of money deposited from the customer.
- max: the maximum amount of masks that the customer can buy corresponding to the amount of money the customer deposits. The amount of max will decrease by one each time the customer presses the count_down button at the AMOUNT state.

- init_max: stores the initial value of max that will not change every time the customer presses the count_down button.
- money_digit1: first digit of change needed to be returned after subtracting the balance with the amount of mask(s) times 5.
- money_digit2: second digit of change needed to be returned after subtracting the balance with the amount of mask(s) times 5.
- change_digit1: takes the amount of money_digit1 to be used in clk_div_27.
- change_digit2: takes the amount of money_digit2 to be used in clk_div_27.
- return_digit1: takes the amount of balance_digit1 to be used in clk_div_27.
- return_digit2: takes the amount of balance_digit2 to be used in clk_div_27.
- digit1, digit2, digit3, digit4: each uses 4 bits, used to represent the digits on the seven segment display from leftmost side to rightmost side.
- sec: counter to count to 5 seconds in RELEASE state.
- change_state: flag that indicates the 5 second count in RELEASE state is over, and that the machine will change from RELEASE state to CHANGE state.
- multiply: used to get the price of the mask by multiplying the amount of mask to 5. Here, I add the amount of mask 5 times, which is why I need this variable.
- amt_1: stores the first digit of the price of the mask(s).
- amt_2: stores the second digit of the price of the mask(s).
- amount_digit1: stores the first final digit of the price of the mask(s).
- amount_digit2: stores the second final digit of the price of the mask(s).
- flag_check_dp: flag to indicate that the check button was pressed at DEPOSIT state (1'b1 if pressed, 1'b0 otherwise).
- flag_cancel_dp: flag to indicate that the cancel button was pressed at DEPOSIT state (1'b1 if pressed, 1'b0 otherwise).
- flag_check_amt: flag to indicate that the check button was pressed at AMOUNT state (1'b1 if pressed, 1'b0 otherwise).
- flag_cancel_amt: flag to indicate that the cancel button was pressed at AMOUNT state (1'b1 if pressed, 1'b0 otherwise).
- flag_buy: flag to indicate that the customer buys the mask(s) in the end before going to CHANGE state.
- flag_cancel: flag to indicate that the customer cancels the purchase of mask(s) in the end before going to CHANGE state.
- change_init: to indicate that all changes have been returned at the CHANGE state and that the machine can go back to the INITIAL state, also to reset everything back to the first place so that the machine can operate from the start.

I first defined all 5 states at the very beginning as follows, so that I can use it by name to ease me while designing:

```
`define INITIAL 3'd0
`define DEPOSIT 3'd1
`define AMOUNT 3'd2
`define RELEASE 3'd3
`define CHANGE 3'd4
```

I then have the clock divider modules for frequency clk/(2^13), clk/(2^16), clk/(2^27), and clk/(2^29). I instantiated these modules on the main module like this:

```
clock_divider_13 clkdiv13(.clk(clk), .clk_div_13(clk_div_13));
clock_divider_16 clkdiv16(.clk(clk), .clk_div_16(clk_div_16));
clock_divider_27 clkdiv27(.clk(clk), .clk_div_27(clk_div_27));
clock_divider_29 clkdiv29(.clk(clk), .clk_div_29(clk_div_29));
```

As for the debounce and one pulse, I also instantiated them in the main module:

```
debounce debounce_money_5(.pb_debounced(money_5_debounced), .pb(money_5), .clk(clk_div_16));
debounce debounce_money_10(.pb_debounced(money_10_debounced), .pb(money_10), .clk(clk_div_16));
debounce debounce_cancel(.pb_debounced(cancel_debounced), .pb(cancel), .clk(clk_div_16));
debounce debounce_check(.pb_debounced(check_debounced), .pb(check), .clk(clk_div_16));
debounce debounce_count_down(.pb_debounced(count_down_debounced), .pb(count_down), .clk(clk_div_16));

one_pulse onepulse_money_5(.pb_debounced(money_5_debounced), .clk(clk_div_16), .pb_one_pulse(money_5_one_pulse));
one_pulse onepulse_money_10(.pb_debounced(money_10_debounced), .clk(clk_div_16), .pb_one_pulse(money_10_one_pulse));
one_pulse onepulse_cancel(.pb_debounced(cancel_debounced), .clk(clk_div_16), .pb_one_pulse(cancel_one_pulse));
one_pulse onepulse_check(.pb_debounced(check_debounced), .clk(clk_div_16), .pb_one_pulse(check_one_pulse));
one_pulse onepulse_count_down(.pb_debounced(count_down_debounced), .clk(clk_div_16), .pb_one_pulse(count_down_one_pulse));
```

To display all four digits simultaneously on the 7-segment display, I need to use refresher here. The refresher I implemented here takes the two most significant bits of the 20-bit refresh_counter, which will increase by 20'd1 every time a positive edge of a clock occurs.

```
always@(posedge clk)
begin
    refresh_counter <= refresh_counter + 20'd1;
end

assign refresher = refresh_counter[19:18];
```

Then in this always@(posedge clk or posedge rst or posedge change_init) block, I set the value of state to next_state, which holds the value of the corresponding state in this design depending on the actions.

However, if rst is equal to 1’b1 or if change_init is equal to 1’b1, the state will automatically become the INITIAL state. The always block for the transition of states looks like this:

```
always@(posedge clk or posedge rst or posedge change_init)
begin
    if (rst == 1'b1 || change_init == 1'b1)
    begin
        state <= `INITIAL;
    end

    else
    begin
        state <= next_state;
    end
end
```

I then have an always block that is specially used to control check and cancel buttons at the DEPOSIT state. The buttons will control the value of the flag variables: check_one_pulse for flag_check_dp, and cancel_one_pulse for flag_cancel_dp.

If flag_check_dp == 1’b1, then the machine will change its state from DEPOSIT to AMOUNT. If flag_cancel_dp == 1’b1, then the machine will change its state from DEPOSIT to CHANGE, just like if we press the check button the machine will change its state from DEPOSIT to AMOUNT, and if we press the cancel button the machine will change its state from DEPOSIT to CHANGE.

If rst or change_init is not equal to 1’b1, then flag_check_dp and flag_cancel_dp can change its values corresponding to the buttons. Here, in state DEPOSIT, if check_one_pulse == 1’b1 or cancel_one_pulse == 1’b1, that means the check (or cancel) button was pressed, then we will check the amount of balance digits first before proceeding, because if balance_digit1 and balance_digit2 are equal to 0, then the machine cannot change its state from DEPOSIT state, thus flag_check_dp and flag_cancel_dp will still be 1’b0 even though the buttons were pressed.

If the balance is greater than 0, that is, at least one of the value between balance_digit1 or balance_digit2 is greater than 0, then flag_check_dp will be assigned to 1’b1 if check_one_pulse == 1’b1, and flag_cancel _dp will be assigned to 1’b1 if cancel_one_pulse == 1’b1.

```
if(check_one_pulse == 1'b1)
begin
    if(balance_digit1 > 0 || balance_digit2 > 0)
        flag_check_dp = 1'b1;
    else
        flag_check_dp = 1'b0;
end
```

```
if(cancel_one_pulse == 1'b1)
begin
    if(balance_digit1 > 0 || balance_digit2 > 0)
    begin
        flag_cancel_dp = 1'b1;
    end

    else
        flag_cancel_dp = 1'b0;
end
```

Else, if the check or cancel button was not pressed, then the value of flag_check_dp or flag_cancel_dp will still be 1'b0.

```
else
    flag_check_dp = 1'b0;
```

```
else
    flag_cancel_dp = 1'b0;
```

Hence, if rst == 1'b1 or change_init == 1'b1, then the value of flag_check_dp and flag_cancel_dp will both be set to 1'b0.

```
if(rst == 1'b1 || change_init == 1'b1)
begin
    flag_cancel_dp = 1'b0;
    flag_check_dp = 1'b0;
end
```

Full code:

```
always@(posedge clk_div_16 or posedge rst or posedge change_init)
begin
    if(rst == 1'b1 || change_init == 1'b1)
    begin
        flag_cancel_dp = 1'b0;
        flag_check_dp = 1'b0;
    end

    else
    begin
        if(state == `DEPOSIT)
        begin
            if(check_one_pulse == 1'b1)
            begin
                if(balance_digit1 > 0 || balance_digit2 > 0)
                    flag_check_dp = 1'b1;
                else
                    flag_check_dp = 1'b0;
            end

            else
                flag_check_dp = 1'b0;

            if(cancel_one_pulse == 1'b1)
            begin
                if(balance_digit1 > 0 || balance_digit2 > 0)
                begin
                    flag_cancel_dp = 1'b1;
                end

                else
                    flag_cancel_dp = 1'b0;
            end

            else
                flag_cancel_dp = 1'b0;
        end
    end
end
```

I did the same thing for the state AMOUNT, having an always block that is specially used to control check and cancel buttons at the AMOUNT state. The buttons will control the value of the flag variables: check_one_pulse for flag_check_amt, and cancel_one_pulse for flag_cancel_amt.

If flag_check_amt == 1'b1, then the machine will change its state from AMOUNT to RELEASE state. If flag_cancel_amt == 1'b1, then the machine will change its state from AMOUNT to CHANGE, just like if we press the check button the machine will change its state from AMOUNT to RELEASE, and if we press the cancel button the machine will change its state from AMOUNT to CHANGE.

If rst or change_init is not equal to 1'b1, then flag_check_amt and flag_cancel_amt can change its values corresponding to the buttons. Here, in state AMOUNT, if check_one_pulse == 1'b1 or cancel_one_pulse == 1'b1, that means the check (or cancel) button was pressed, then flag_check_amt will be assigned to 1'b1 if check_one_pulse == 1'b1, and flag_cancel_amt will be assigned to 1'b1 if cancel_one_pulse == 1'b1, without having to check the amount of balance like the previous DEPOSIT state.

Else, if the check or cancel button was not pressed, then the value of flag_check_amt or flag_cancel_amt will still be 1'b0.

```
always@(posedge clk_div_16 or posedge rst or posedge change_init)
begin
    if(rst == 1'b1 || change_init == 1'b1)
    begin
        flag_cancel_amt = 1'b0;
        flag_check_amt = 1'b0;
    end

    else
    begin
        if(state == `AMOUNT)
        begin
            if(check_one_pulse == 1'b1)
                flag_check_amt = 1'b1;
            else
                flag_check_amt = 1'b0;

            if(cancel_one_pulse == 1'b1)
            begin
                flag_cancel_amt = 1'b1;
            end

            else
                flag_cancel_amt = 1'b0;
        end
    end
end
```

Next, I have another always block that is used to control the change of one state to another state based on the actions taken on the buttons or by conditions. Here, we make use of flag_check_dp, flag_cancel_dp, flag_check_amt, and flag_cancel_amt we generated at the previous two always blocks.

As usual, if rst == 1'b1 or change_init == 1'b1, then the variables used in this always block will be reset to its initial value. Here we have flag_cancel which will be reset to 1'b0 and next_state which will be reset to `INITIAL.

```
if(rst == 1'b1 || change_init == 1'b1)
begin
    flag_cancel = 1'b0;
    next_state = `INITIAL;
end
```

Else, if both reset and change_init are not equal to 1'b1, then we can assign values to next_state. I divided the cases per state using if else statement.

1. First if statement is for state INITIAL. We won't need any buttons nor actions for it to change state because as described in the spec of lab5, when the machine is at INITIAL state, it will directly go to the DEPOSIT state. Hence, we can directly assign next_state at INITIAL state to DEPOSIT state.

```
if(state == `INITIAL)
begin
    next_state = `DEPOSIT;
end
```

2. At DEPOSIT state, we can change state depending on the buttons. If the check button is pressed, that is, flag_check_dp == 1'b1, then we assign next_state as `AMOUNT. Else, if cancel button was pressed (flag_cancel_dp == 1'b1), then we assign flag_cancel to 1'b1 to indicate that cancellation has been done, and assign next_state to `CHANGE. Else, if no buttons were pressed, then next_state will stay at DEPOSIT state.

```
else if(state == `DEPOSIT)
begin
    if(flag_check_dp == 1'b1)
        next_state = `AMOUNT;

    else if(flag_cancel_dp == 1'b1)
    begin
        flag_cancel = 1'b1;
        next_state = `CHANGE;
    end

    else
        next_state = `DEPOSIT;
end
```

3. At AMOUNT state, we can change state depending on the buttons. If the check button is pressed, that is, flag_check_amt == 1'b1, then we assign next_state as `RELEASE. Else, if cancel button was pressed (flag_cancel_amt == 1'b1), then we assign flag_cancel to 1'b1 to indicate that cancellation has been done, and assign next_state to `CHANGE. Else, if no buttons were pressed, then next_state will stay at AMOUNT state.

```
else if(state == `AMOUNT)
begin
    if(flag_check_amt == 1'b1)
    begin
        next_state = `RELEASE;
    end

    else if(flag_cancel_amt == 1'b1)
    begin
        flag_cancel = 1'b1;
        next_state = `CHANGE;
    end

    else
        next_state = `AMOUNT;
end
```

4. At RELEASE state, the state will change if the RELEASE state has been running for 5 seconds, and I have change_state variable to indicate that 5 seconds has passed. Therefore, if change_state == 1'b1, which means that 5 seconds has passed at RELEASE state, then the next_state will be assigned to `CHANGE. Else, next_state will stay at RELEASE.

```
else if(state == `RELEASE)
begin
    if(change_state == 1'b1)
        next_state = `CHANGE;

    else
        next_state = `RELEASE;
end
```

5. Lastly, at CHANGE state, it will only change the state only if the money has been returned until none is left. I used the variable change_init to indicate that all the money has been returned. If change_init == 1'b1, then the money has been returned, then next_state can be assigned back to `INITIAL. Else, then next_state will stay at `CHANGE.

```
else if(state == `CHANGE)
begin
    if(change_init == 1'b1)
        next_state = `INITIAL;

    else
        next_state = `CHANGE;
end
```

Next one is to implement what happens at DEPOSIT and AMOUNT which is at clk_div_16.

If rst == 1'b1 and change_init == 1'b1, then all variables used in this always block will be set back to its initial state, which are all 4'd0. Else, the vending machine will start working based on its states.

1. DEPOSIT

   In this state, if money_5 button was pressed, then the balance will increase by $5, and if money_10 was pressed, then the balance will increase by $10. We first take a look when money_5 button is pressed, which is inside the if(money_5_one_pulse == 1'b1) statement.

   We have several cases here, when balance is equal to $50, when balance is equal to $45 and the rest are just normal conditions.

   (1) When balance is equal to $50, that is, if(balance_digit1 == 4'd0 && balance_digit2 == 4'd5), then when money_5 is pressed, balance_digit1 and balance_digit2 will stop increasing since it has reached maximum and both max and init_max will be 4'd9, indicating the maximum number of masks the customer can buy.

   (2) When balance is equal to $45, that is, else if(balance_digit1 == 4'd5 && balance_digit2 == 4'd4), then when money_5 was pressed, balance_digit1 should be 4'd0 and balance_digit2 should be 4'd5, and both max and init_max will be 4'd9.

   On other conditions, since I implement the balance counter using two separate digits, then we have two cases here, when balance_digit1 is equal to 4'd0 and when balance_digit1 is equal to 4'd5. The way balance_digit2 counts depends on the value of balance_digit1 if money_5 was pressed.

   (3) If balance_digit1 is equal to 4'd0, then we can directly add 4'd5 to balance_digit1, and balance_digit2 will stay at the same value since we don't have carry here. Then both max and init_max will increase by 4'd1 since the price of each mask is $5.

   (4) If balance_digit1 is equal to 4'd5, then when being added by 4'd5, the value of balance_digit1 will be equal to 4'd0 (following the standard rule of addition) and the value of balance_digit2 will increase by 4'd1 since we now have carry from adding balance_digit1 to 4'd5. Then both max and init_max will increase by 4'd1 since the price of each mask is $5.

```
if(money_5_one_pulse == 1'b1)
begin
    if(balance_digit1 == 4'd0 && balance_digit2 == 4'd5)
    begin
        balance_digit1 <= balance_digit1;
        balance_digit2 <= balance_digit2;      (1)
        max <= 4'd9;
        init_max <= 4'd9;
    end

    else if(balance_digit1 == 4'd5 && balance_digit2 == 4'd4)
    begin
        balance_digit1 <= 4'd0;
        balance_digit2 <= balance_digit2 + 4'd1;      (2)
        max <= 4'd9;
        init_max <= 4'd9;
    end

    else
    begin
        if(balance_digit1 == 4'd0)      (3)
        begin
            balance_digit1 <= balance_digit1 + 4'd5;
            max <= max + 4'd1;
            init_max <= init_max + 4'd1;
        end

        if(balance_digit1 == 4'd5)      (4)
        begin
            balance_digit1 <= 4'd0;
            balance_digit2 <= balance_digit2 + 4'd1;
            max <= max + 4'd1;
            init_max <= init_max + 4'd1;
        end
    end
end
```

Now, when money_10 button was pressed, we also have several cases to deal with.

(1) When balance is equal to $45 or $50, which is if((balance_digit1 == 4'd5 && balance_digit2 == 4'd4) || (balance_digit1 == 4'd0 && balance_digit2 == 4'd5)), then when button money_10 was pressed, in both conditions, the amount of balance will increase to $50, which is 4'd0 for balance_digit1 and 4'd5 for balance_digit2. Then both max and init_max will be 4'd9.

(2) When balance is equal to $40, if(balance_digit1 == 4'd0 && balance_digit2 == 4'd4), then when button money_10 is pressed the balance will increase to $50, which can be implemented by balance_digit1 <= balance_digit1 and increasing the value of balance_digit1 by 4'd1. Then the max and init_max of the masks will be 4'd9 since it is the maximum amount of masks that can be bought.

(3) In other cases, we can directly increase the amount of balance_digit2 by 4'd1 and not worry about balance_digit1. Then, both max and init_max will increase by 2 since if customer deposits $10, it can be used to buy 2 pieces of mask since one piece of mask is $5.

```
else if(money_10_one_pulse == 1'b1)
begin
    if((balance_digit1 == 4'd5 && balance_digit2 == 4'd4) || (balance_digit1 == 4'd0 && balance_digit2 == 4'd5))
    begin
        balance_digit1 <= 4'd0;
        balance_digit2 <= 4'd5;
        max <= 4'd9;
        init_max <= 4'd9;
    end

    else if(balance_digit1 == 4'd0 && balance_digit2 == 4'd4)
    begin
        balance_digit1 <= balance_digit1;
        balance_digit2 <= balance_digit2 + 4'd1;
        max <= 4'd9;
        init_max <= 4'd9;
    end

    else
    begin
        balance_digit2 <= balance_digit2 + 4'd1;
        max <= max + 4'd2;
        init_max <= init_max + 4'd2;
    end
end
```

(1)

(2)

(3)

2. AMOUNT

In the AMOUNT state, customer can adjust the number of mask(s) to be bought by pressing the count_down button. The amount of mask will decrease by 1 each time the button was pressed, and will return back again to the initial maximum account if the customer keeps pressing count_down button when the amount of mask reaches 1.

Therefore, when count_down_one_pulse == 1'b1, which is when the count_down button was pressed, and if max is greater than 4'd1, then the value of max will decrease by 4'd1. Else, if max is already equal to 4'd1 and the count_down button was pressed, then max will be assigned to init_max, the initial amount of maximum number of masks that can be bought. Else, if the button was not pressed, then max will stay at its same value.

At the same time, I am also calculating the total price of the mask(s). I had several cases here, when max <= 4'd7, and when 4'd7 < max <= 4'd9. I separated these because using my method of counting, I will need larger than 4 bits to store the value of the price, but at that time it is too late for me to change everything so I added these cases.

```
if(max <= 4'd7)
begin
    if(multiply != 4'd5)
    begin
        if(amt_1 >= 4'd10)
        begin
            amt_1 <= amt_1 + max - 4'd10;
            amt_2 <= amt_2 + 4'd1;
            multiply <= multiply + 4'd1;
        end

        else
        begin
            amt_1 <= max + amt_1;
            amt_2 <= amt_2 + 4'd0;
            multiply <= multiply + 4'd1;
        end
    end

    else
    begin
        if(amt_1 >= 4'd10)
        begin
            amount_digit1 <= amt_1 - 4'd10;
            amount_digit2 <= amt_2 + 4'd1;
            amt_1 <= 4'd0;
            amt_2 <= 4'd0;
            multiply <= 4'd0;
        end

        else
        begin
            amount_digit1 <= amt_1;
            amount_digit2 <= amt_2;
            amt_1 <= 4'd0;
            amt_2 <= 4'd0;
            multiply <= 4'd0;
        end
    end
end
```

If max <= 4'd7, I count the price by adding max to itself for 5 times, which is why I used the register multiply that will count for 5 times, and max will stop calculating when multiply reaches 4'd5. In the if(multiply != 4'd5), we can have two cases, one is when amt_1 >= 4'd10 and other one is when amt_1 < 4'd10. The first time max goes in here will be in the latter case, when amt_1 < 4'd10, because the initial amount of amt_1 is 4'd0. When amt_1 < 4'd10, I assign the value of amt_1 to max + amt_1, and we don't need to worry about amt_2 since there are no carries, then multiply will increase by one.

Other case is when amt_1 >= 4'd10. Here, after adding amt_1 to max, we also need to subtract it by 4'd10 and add amt_2 to 4'd1, because we are using two digits to count. So this illustrates moving the tens from amt_1 to amt_2. Same thing, multiply will also increase by 4'd1 here.

We repeat the same method until multiply reached 4'd5 and then we check if amt_1 itself is greater or equal to 4'd10 or not. If yes, then we subtract amt_1 with 4'd10 and add amt_2 with 4'd1, and assign both of them to amount_digit1 and amount_digit2 respectively. Then we also need to set amt_1, amt_2, and multiply back to their initial values, which is 4'd0.

If amt_1 is less than 4'd10, then we can directly assign amount_digit1 and amount_digit2 to the value of amt_1 and amt_2, and set amt_1, amt_2, and multiply back to their initial values, which is 4'd0.

When 4’d7 < max <= 4’d9, it can happen when max == 4’d8 or max == 4’d9. If max == 4’d8, then amount_digit1 will immediately be set to 4’d0 and amount_digit2 will be set to 4’d4, which is the price of 8 masks, $40. Then if max is equal to 4’d9, then amount_digit1 will immediately be set to 4’d5 and amount_digit2 will be set to 4’d4, which is the price of 9 masks, $45. Not to forget we also need to set amt_1, and amt_2 back to their initial values, which is 4’d0.

```
else if(4'd7 < max <= 4'd9)
begin
    if(max == 4'd8)
    begin
        amount_digit1 <= 4'd0;
        amount_digit2 <= 4'd4;
        amt_1 <= 4'd0;
        amt_2 <= 4'd0;
    end

    else if(max == 4'd9)
    begin
        amount_digit1 <= 4'd5;
        amount_digit2 <= 4'd4;
        amt_1 <= 4'd0;
        amt_2 <= 4'd0;
    end
end
```

After getting the price of the masks, we can now subtract the balance with the price of the mask to get the amount of change to be given back to the customer. I used another register here, money_digit1 and money_digit2 to get the amount of change to be returned back to the customer. There might be two conditions in this case:

a. When balance_digit1 == 4’d0 and amount_digit1 == 4’d5
   if(amount_digit1 == 4'd5 && balance_digit1 == 4'd0)
   This case can occur in conditions like 30-15, and we need to have borrow. So I directly assign money_digit1 to 4’d5 and money_digit2 to (balance_digit2 – 4’d1 – amount_digit2). Using this formula, we can get the correct amount of change to be returned to the customer.
b. The second case is the usual case for subtraction where we don’t need to have borrow from the second digit. Therefore, we can directly subtract balance_digit1 with amount_digit1, balance_digit2 with amount_digit2, and assign their values to money_digit1 and money_digit2 respectively.

```
if(amount_digit1 == 4'd5 && balance_digit1 == 4'd0)
begin
    money_digit1 <= 4'd5;
    money_digit2 <= balance_digit2 - 4'd1 - amount_digit2;
end

else
begin
    money_digit1 <= balance_digit1 - amount_digit1;
    money_digit2 <= balance_digit2 - amount_digit2;
end
```

3. RELEASE

This time we use clk/(2^27), and I assign the values of money_digit1 and money_digit2 to change_digit1 and change_digit2 respectively, which will be used for the CHANGE state. At this state, the 16 LED lights will turn on and off repeatedly so I negate the LED, and the LED will be negated every clk/(2^27) which is 1 second, so it satisfies the condition described in the spec. Also, this state will last for 5 seconds so I have the counter to 5, which is sec, and it will increase by 4'd1 each time clk/(2^27) occurs, which is 1 second, and by the time sec reaches 5, we know that 5 seconds has passed. Therefore, when sec reaches 4'd5, we are ready to change the state from RELEASE to CHANGE, so I assign flag_buy to 1'b1, which indicates that the customer has successfully bought the mask(s), and change_state to 1'b1 which is used as a flag to change the state. Then if rst == 1'b1 or change_init == 1'b1, then everything here will be set back to its initial value.

```
if(state == `RELEASE)
begin
    change_digit1 <= money_digit1;
    change_digit2 <= money_digit2;

    LED <= ~LED;
    sec <= sec + 4'd1;

    if(sec == 4'd5)
    begin
        flag_buy <= 1'b1;
        change_state <= 1'b1;
    end
end
```

4. CHANGE

We also use clk/(2^27) here, and we have two possibilities in this CHANGE state, that is, if the customer cancels the purchase or the customer successfully done the purchase. If the customer has successfully done the purchase, it indicates that flag_cancel == 1'b0, so we can take a look inside the if(flag_cancel == 1'b0) statement.

Here, I used change_digit1 and change_digit2 which is the amount of change after buying the mask(s). In this state, the amount of change will decrease sequentially from tens to five, so we may have a few conditions to satisfy.

A) If both change_digit1 and change_digit2 are equal to 4'd0, that is, all the change are returned, then change_digit1 and change_digit2 will stay at 4'd0, and change_init will be assigned to 1'b1 indicating that the machine can go back to INITIAL state.

B) Else, if there are still changes to be returned, then in order to return it sequentially as described in the spec, we have another conditions.

a) if(change_digit1 == 4'd5 && change_digit2 == 4'd0)
   Since only the first digit has amount and it is 4'd5, then we can directly subtract change_digit1 with 4'd5 and leave change_digit2 as it is since it is already 4'd0 to begin with.
b) if(change_digit2 > 4'd0 && change_digit1 == 4'd5)
   When the second digit has amount, we subtract the amount of second digit by 4'd1 first until it reaches 4'd0, then we can proceed to subtract change_digit1 by 5'd5.
c) if(change_digit2 > 4'd0 && change_digit1 == 4'd0)
   If only the second digit has amount and the first digit is 4'd0, then we can directly subtract the second digit with 4'd1 until it reaches 4'd0 and leave change_digit1 as it is since it was already 4'd0 to begin with.

```
if(flag_cancel == 1'b0)
begin
    if(change_digit1 == 4'd0 && change_digit2 == 4'd0)
    begin
        change_digit1 <= 4'd0;
        change_digit2 <= 4'd0;
        change_init <= 1'b1;
    end                                          (A)

    else                                         (B)
    begin
        if(change_digit1 == 4'd5 && change_digit2 == 4'd0)
        begin
            change_digit1 <= change_digit1 - 4'd5;
        end                                              (a)

        else if(change_digit2 > 4'd0 && change_digit1 == 4'd5)
        begin
            if(change_digit2 == 4'd0)
            begin                                        (b)
                change_digit2 <= 4'd0;
                change_digit1 <= change_digit1 - 4'd5;
            end

            else
                change_digit2 <= change_digit2 - 4'd1;
        end

        else if(change_digit2 > 4'd0 && change_digit1 == 4'd0)
        begin
            change_digit2 <= change_digit2 - 4'd1;
        end                                              (c)
    end
end
```

After being done with the first condition, we can take a look at the second condition, which is when flag_cancel == 1'b1, that is, the purchase was cancelled.

I used the registers return_digit1 and return_digit2 which was assigned with the value of balance_digit1 and balance_digit2 respectively. Then, how we decrease the value of return_digit1 and return_digit2 is the same as how we decrease the value of change_digit1 and change_digit2 in the previous condition:

A) If both return_digit1 and return_digit2 are equal to 4'd0, that is, all the money are returned, then return_digit1 and return_digit2 will stay at 4'd0, and change_init will be assigned to 1'b1 indicating that the machine can go back to INITIAL state.

B) Else, if there are still money to be returned, then in order to return it sequentially as described in the spec, we have another conditions.

   a) if(return_digit1 == 4'd5 && return_digit2 == 4'd0)
      Since only the first digit has amount and it is 4'd5, then we can directly subtract return_digit1 with 4'd5 and leave return_digit2 as it is since it is already 4'd0 to begin with.

   b) if(return_digit2 > 4'd0 && return_digit1 == 4'd5)
      When the second digit has amount, we subtract the amount of second digit by 4'd1 first until it reaches 4'd0, then we can proceed to subtract return_digit1 by 5'd5.

   c) if(return_digit2 > 4'd0 && return_digit1 == 4'd0)
      If only the second digit has amount and the first digit is 4'd0, then we can directly subtract the second digit with 4'd1 until it reaches 4'd0 and leave return_digit1 as it is since it was already 4'd0 to begin with.

```
else
begin
    if(return_digit1 == 4'd0 && return_digit2 == 4'd0)
    begin
        return_digit1 <= 4'd0;
        return_digit2 <= 4'd0;
        change_init <= 1'b1;
    end

    else
    begin
        if(return_digit1 == 4'd5 && return_digit2 == 4'd0)
        begin
            return_digit1 <= return_digit1 - 4'd5;
            return_digit2 <= 4'd0;
        end

        else if(return_digit2 > 4'd0 && return_digit1 == 4'd5)
        begin
            if(return_digit2 == 4'd0)
            begin
                return_digit2 <= 4'd0;
                return_digit1 <= return_digit1 - 4'd5;
            end

            else
                return_digit2 <= return_digit2 - 4'd1;
        end

        else if(return_digit2 > 4'd0 && return_digit1 == 4'd0)
        begin
            return_digit2 <= return_digit2 - 4'd1;
        end
    end
end
```

(A) (B) (a) (b) (c)

Also, if rst occurs or if change_init == 1'b1, then everything will be set back to its initial value.

```
if(rst == 1'b1 || change_init == 1'b1)
begin
    LED <= 16'd0;
    sec <= 4'd0;
    flag_buy <= 1'b0;
    change_state <= 1'b0;
    change_digit1 <= 4'd0;
    change_digit2 <= 4'd0;
    change_init <= 1'b0;
    return_digit1 <= 4'd0;
    return_digit2 <= 4'd0;
    flag_transition = 1'b0;
end
```

Lastly, all that is left to be explained is regarding the 7-segment display. At always@(posedge clk_div_13), I have the cases for all states because each state used different registers.

A) INITIAL state

In this state, I displayed balance_digit1 and balance_digit2 for digit1 and digit2 of the 7-segment display which were initially zero, and max for digit3, and 4'd0 for digit4 since it always has the value 0.

B) DEPOSIT state

Everything was the same as INITIAL state, except for the fact that balance_digit1 and balance_digit2 can increase its value and not remain as zero.

C) AMOUNT state

Everything was the same as DEPOSIT state, just that both balance_digit1 and balance_digit2 hold their values meanwhile max can change its value depending on the count_down button.

D) RELEASE state

Digit1, digit2, digit3, and digit4 displays the word "TAKE", so I directly assign digit1 as 4'd13, digit2 ad 4'd12, digit3 ad 4'd11, and digit4 as 4'd10.

E) CHANGE state

The display for digit1 and digit2 depends on flag_cancel. If flag_cancel == 1'b0, that means the purchase was cancelled, then digit1 and digit2 will hold the value of change_digit1 and change_digit2. Else if the purchase was not cancelled (flag_cancel == 1'b1), then digit1 and digit2 will hold the value of return_digit1 and return_digit2. Then both digit3 and digit4 will hold the value of 4'd0.

F) Default

The default case is to set all digits to hold the value of 4'd0.

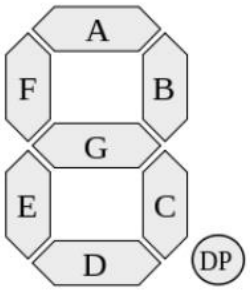
```
always@(posedge clk_div_13)
begin
    if(state == `INITIAL)
    begin
        digit1 <= balance_digit1;
        digit2 <= balance_digit2;
        digit3 <= max;
        digit4 <= 4'd0;
    end

    else if(state == `DEPOSIT)
    begin
        digit1 <= balance_digit1;
        digit2 <= balance_digit2;
        digit3 <= max;
        digit4 <= 4'd0;
    end

    else if(state == `AMOUNT)
    begin
        digit1 <= balance_digit1;
        digit2 <= balance_digit2;
        digit3 <= max;
        digit4 <= 4'd0;
    end

    else if(state == `RELEASE)
    begin
        digit1 <= 4'd13;
        digit2 <= 4'd12;
        digit3 <= 4'd11;
        digit4 <= 4'd10;
    end

    else if(state == `CHANGE)
    begin
        digit1 <= (flag_cancel == 1'b0) ? change_digit1 : return_digit1;
        digit2 <= (flag_cancel == 1'b0) ? change_digit2 : return_digit2;
        digit3 <= 4'd0;
        digit4 <= 4'd0;
    end

    else
    begin
        digit1 <= 4'd0;
        digit2 <= 4'd0;
        digit3 <= 4'd0;
        digit4 <= 4'd0;
    end

end
```

```
always@(*)
begin
    case(ACT_DIGIT) //digit
        4'd0: DISPLAY <= 7'b0000001;
        4'd1: DISPLAY <= 7'b1001111;
        4'd2: DISPLAY <= 7'b0010010;
        4'd3: DISPLAY <= 7'b0000110;
        4'd4: DISPLAY <= 7'b1001100;
        4'd5: DISPLAY <= 7'b0100100;
        4'd6: DISPLAY <= 7'b0100000;
        4'd7: DISPLAY <= 7'b0001111;
        4'd8: DISPLAY <= 7'b0000000;
        4'd9: DISPLAY <= 7'b0000100;
        4'd10: DISPLAY <= 7'b1110000; //T
        4'd11: DISPLAY <= 7'b0000010; //A
        4'd12: DISPLAY <= 7'b0101000; //K
        4'd13: DISPLAY <= 7'b0110000; //E
        default: DISPLAY <= 7'b1111111;
    endcase
end
```

As usual, for the numbers to appear correctly on the digits of the seven segment display, because this time to the binary digit 0 is used activate a single segment instead of 1, so I designed the case which will take ACT_DIGIT to determine which number to display on the digit, as shown below, and modified a bit of the constrains on this part for it to match with my main source code. The case is put inside an always@(*) block because this part will be used every time it is called.

Constraints for display, as referred from the lecture PPT:

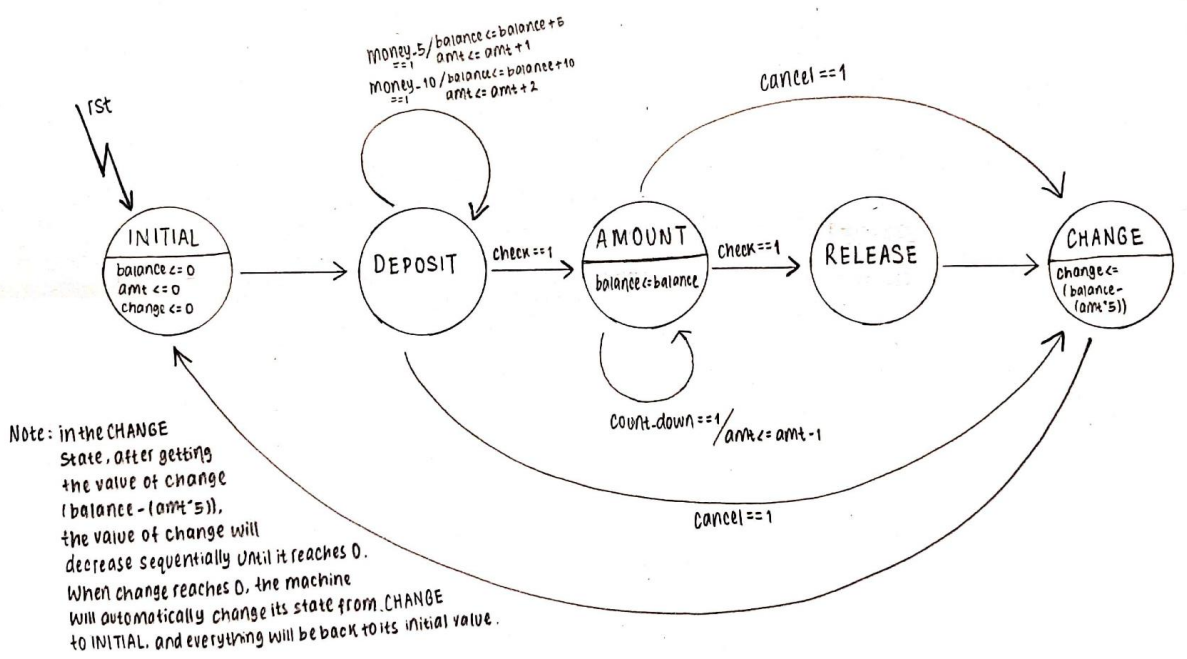
```
# 7 segment display
 set_property PACKAGE_PIN W7 [get_ports {DISPLAY[6]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[6]}]
 set_property PACKAGE_PIN W6 [get_ports {DISPLAY[5]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[5]}]
 set_property PACKAGE_PIN U8 [get_ports {DISPLAY[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[4]}]
 set_property PACKAGE_PIN V8 [get_ports {DISPLAY[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[3]}]
 set_property PACKAGE_PIN U5 [get_ports {DISPLAY[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[2]}]
 set_property PACKAGE_PIN V5 [get_ports {DISPLAY[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[1]}]
 set_property PACKAGE_PIN U7 [get_ports {DISPLAY[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {DISPLAY[0]}]
```



The ACT_DIGIT used as the case for the seven segment display is obtained from this part of code, which takes digit1 to be displayed on the rightmost digit (DIGIT <= 4'b1110), digit2 to be displayed on the 2nd rightmost digit (DIGIT <= 4'b1101), digit3 to be displayed on the 2nd leftmost digit (DIGIT <= 4'b1011) and digit4 to be displayed on the leftmost digit (DIGIT <= 4'b0111).

```
always@(posedge clk_div_13)
begin
    case(refresher)
        2'b00:
        begin
            DIGIT <= 4'b1110;
            ACT_DIGIT <= digit1;
        end

        2'b01:
        begin
            DIGIT <= 4'b1101;
            ACT_DIGIT <= digit2;
        end

        2'b10:
        begin
            DIGIT <= 4'b1011;
            ACT_DIGIT <= digit3;
        end

        2'b11:
        begin
            DIGIT <= 4'b0111;
            ACT_DIGIT <= digit4;
        end

        default:
        begin
            DIGIT <= 4'b1110;
            ACT_DIGIT <= 4'd0;
        end
    endcase
end
```

I used refresher as the case which I have designed and was explained before, to display all the digits simultaneously. Then, as matched to how the digits were activated, I assigned ACT_DIGIT with the correct variables, which will be used as the case to activate the seven segment display correctly. The case is put inside an always@(posedge clk_div_13) block as described in the spec. For the default I assign it to 4'd0.



From the corresponding state diagram, the first state when we start the machine is the INITIAL state. In the INITIAL state, the machine will directly go to the deposit state so that the customer can put money inside the machine, by pressing money_5 or money_10 button. When the customer presses money_5, balance will be added by 5 and amt of mask will be added by 1. When customer presses money_10, balance will increase by 10 and the amt of mask will increase by 2, since the price of each mask is $5. When pressing money_5 or money_10, the machine will still stay at DEPOSIT state, so the arrow still points to the DEPOSIT state. When the machine is in DEPOSIT state, customer can press either cancel or check button to cancel the purchase and directly go to the CHANGE state, or continue with the purchase and go to the AMOUNT state to adjust the number of masks. This action can only be done if balance is greater than 0. In AMOUNT state, the customer can adjust the number of masks to be bought by pressing count_down button, and the number of mask(s) will decrease by 1 every time the button was pressed, but when the amount of mask is already 1 and the customer still presses the count_down button, the amount of mask will go back to the initial maximum amount of masks that can be bought by the customer corresponding to the balance deposited to the machine. When pressing count_down button, the machine will not change its state and will stay at AMOUNT state. Then in the AMOUNT state, the customer can cancel the purchase by pressing the cancel button and the machine will directly go to the CHANGE state, or continue with the purchase and the machine will go to RELEASE state. Then in the RELEASE state, the machine will be in this state for 5

seconds, and then automatically move to CHANGE state to return all the changes to the customer. In the RELEASE state, all LED lights will light up and will turn off for 1 second in the 5 seconds period, and the 7-segment display will show “TAKE”. After 5 seconds, the machine will automatically go to CHANGE state to return all the changes to the customer sequentially. At first in the CHANGE state, the 7-segment display will first show the amount of change to be returned using the formula change = (balance-(amt*5)). Then the amount of change will decrease sequentially until it reaches 0 and when it reaches 0, the machine will go back to INITIAL state and everything will be set back to its initial values. In any state, if the customer resets the machine, everything will be reset back to its initial state and values.

2. 學到的東西與遇到的困難

My variables used to be messed up and all jumbled so I decided to separate each registers for each states and maybe for some conditions also. This way, I can easily detect where the error might be when it happened. I once also messed up the buttons which is used for state transition and it did not work as how I wanted it to be which is why I decided to use flag for some buttons in each state.

3. 想對老師或助教說的話

Everything was great overall. I am looking forward to learn more about good coding style and learning more from common mistakes so that I could make a simpler and neater design in the upcoming labs instead of jumbling up all the variables and mess everything up, resulting in a hard to debug code.