EECS 2070 02 Digital Design Labs 2020
Lab 6

學號：108062281　姓名：莊晴雯

1. 實作過程

In this lab6, we are required to implement a finite state machine by designing a monorail controller than can move either clockwise or counterclockwise and has three stops. The user can add people at each stops anytime which is represented by 12 LED lights (LED0-3: stop1, LED 4-7: stop2, LED 8-11: stop3) by using the right side numeric keypad on the keyboard (press 1 to increase passengers at stop1, press 2 to increase passengers at stop2, press 3 to increase passengers at stop3). Each LED lights that lights up indicates that there is someone waiting in queue to be picked up by the monorail. If there are no passengers queueing at all, then the monorail stops at the last stop.

The controller itself has three states:

1. STAY: the monorail stays at the same stop. If there are passengers waiting at the current stop that the monorail is in, the monorail will stay and pick up the passenger(s) one by one by frequency of clk/(2^26). Thus, the LED lights at the current stop will turn off one by one with frequency of clk/(2^26), indicating that the passengers were picked up by the monorail. The rightmost digit of the seven-segment display shows the symbol of either CLOCKWISE or COUNTERCLOCKWISE, depending on the monorail's direction.
2. CLOCKWISE: the monorail moves and switch stops in clockwise direction (1->2->3->1). It takes 4 clock cycle of clk/(2^26) to move from one stop to another. The passengers have to wait, in other words, were not picked up if the monorail is in this state. The second rightmost digit of the seven-segment display shows the animation of CLOCKWISE direction.
3. COUNTERCLOCKWISE: the monorail moves and switch stops in clockwise direction (3>2->1->3). It takes 4 clock cycle of clk/(2^26) to move from one stop to another. The passengers have to wait, in other words, were not picked up if the monorail is in this state. The second rightmost digit of the seven-segment display shows the animation of COUNTERCLOCKWISE direction.

The monorail has two modes:

1. Auto: The first mode that the monorail is in after being reset or first programmed. The monorail cannot change its mode to fixed before its first move to another stop. In this mode, the direction of the monorail depends on the nearest stop from current stop that has passengers queueing.
   Example: cur_stop: stop2; passengers queueing: stop1; next_stop: stop1, doesn't matter where the prev_stop is.
   Another example: cur_stop: stop2; prev_stop: stop1 (prev_state: CLOCKWISE); passengers queueing: stop1 and stop3; next_stop: stop3 (monorail keeps its trend (previous direction)).
2. Fixed: The monorail follows its current direction. LED[15] turns on.
   Example: cur_stop: stop1; direction: CLOCKWISE; next_stop; stop2, then stop3

To design this finite state machine, I first define stop1, stop2, and stop3 as 4'd1, 4'd2, and 4'd3 respectively, and the states STAY, CLOCKWISE, and COUNTERCLOCKWISE as 4'd4, 4'd5, and 4'd6 respectively

```
`define stop1 4'd1
`define stop2 4'd2
`define stop3 4'd3

`define STAY 4'd4
`define CLOCKWISE 4'd5
`define COUNTERCLOCKWISE 4'd6
```

Then for the wires and registers:

wire:

- from the inputs rst, clk, mode, PS2_DATA, PS2_CLK, LED.
- clk_div_16: clk frequency of clk/(2^16) for the buttons.
- clk_div_26: clk frequency of clk/(2^26) for the monorail controller.
- rst_debounced and rst_one_pulse: for the rst button.
- mode_debounced and mode_one_pulse: for the mode button.
- refresher: takes the value of the two most significant bits of the 20-bit refresh_counter
- key_down, last_change, been_ready, shift_down for the keyboard.

reg:

- for the outputs DISPLAY and DIGIT
- passenger_1, passenger_2, passenger_3: take 4 bits to represent the passengers waiting at each stops. Each of it will take the value of next_passenger_1, next_passenger_2, next_passenger_3 respectively. Initial value is 4'b0000 since there are no passengers waiting at the stops initially.
- next_passenger_1, next_passenger_2, next_passenger_3: take 4 bits to represent the passengers waiting at each stops, gets the value from user's

input from the keyboard by pressing either 1, 2, or 3 on the numeric keypad. Initial value is 4'b0000 since there are no passengers waiting at the stops initially.

- state: represents the state of the monorail controller, takes the value from next_state. Initial value is `STAY since the monorail is at `STAY state initially.
- next_state: the next upcoming state of the monorail which will later be assigned for reg state. Initial value is `STAY since the monorail is at `STAY state initially.
- dir: represents the direction of the monorail. I first set it as `CLOCKWISE.
- cur_pos: current position (current stop) of the monorail. Takes its value from next_pos. Initial value is `stop1.
- next_pos: next position (next stop) of the monorail. Initial value is `stop1.
- start: to let the mode button be available for pressing. In other words, the user can now press the mode button to switch between fixed and auto mode if start == 1'b1. Initial value is 1'b0 since the mode button is not available for pressing initially.
- auto: the monorail is in auto mode if auto == 1'b1, fixed mode if auto == 1'b0. Initial value is 1'b1 since the monorail is in auto mode initially.
- movement: used for monorail's clockwise animation to count until 4 clock cycles.
- movement_ctr: used for monorail's counterclockwise animation to count until 4 clock cycles.
- display_num: used for display on the seven-segment of the FPGA board.
- 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.
- flag_100 and flag_26: to get the timing when the passengers at each stop decreases by clk/(2^26) when the monorail is at that stop.
- flag_movement: to indicate that 4 clock cycle has passed, and that the new value of next_pos and next_state can be assigned to it. Otherwise, if flag_movement == 1'b0, which means that 4 clock cycle has not passed, then next_pos and next_state will keep its current values. This is used for the `CLOCKWISE state.
- flag_movement_ctr: to indicate that 4 clock cycle has passed, and that the new value of next_pos and next_state can be assigned to it. Otherwise, if flag_movement_ctr == 1'b0, which means that 4 clock cycle has not passed, then next_pos and next_state will keep its current values. This is used for the `COUNTERCLOCKWISE state.

- cnt and cnt3: count to prevent the monorail from skipping stops in the fixed mode.
- start_count, start_count2, start_count3: indicates that cnt and cnt3 can start counting.
- hold, hold2, hold3: holds the monorail from skipping stops.
- flag, flag_3, first, flag_first, in_3: all are flags to support above registers to avoid monorail from skipping stops.
- nums, key_num, last_key: used to support keyboard control.

For the keyboard control, I referred to the sample keyboard control code provided:

```
parameter [8:0] LEFT_SHIFT_CODES  = 9'b0_0001_0010;
parameter [8:0] RIGHT_SHIFT_CODES = 9'b0_0101_1001;
parameter [8:0] KEY_CODES [0:19] = {
    9'b0_0100_0101, // 0 => 45
    9'b0_0001_0110, // 1 => 16
    9'b0_0001_1110, // 2 => 1E
    9'b0_0010_0110, // 3 => 26
    9'b0_0010_0101, // 4 => 25
    9'b0_0010_1110, // 5 => 2E
    9'b0_0011_0110, // 6 => 36
    9'b0_0011_1101, // 7 => 3D
    9'b0_0011_1110, // 8 => 3E
    9'b0_0100_0110, // 9 => 46

    9'b0_0111_0000, // right_0 => 70
    9'b0_0110_1001, // right_1 => 69
    9'b0_0111_0010, // right_2 => 72
    9'b0_0111_1010, // right_3 => 7A
    9'b0_0110_1011, // right_4 => 6B
    9'b0_0111_0011, // right_5 => 73
    9'b0_0111_0100, // right_6 => 74
    9'b0_0110_1100, // right_7 => 6C
    9'b0_0111_0101, // right_8 => 75
    9'b0_0111_1101  // right_9 => 7D
};

reg [15:0] nums;
reg [3:0] key_num;
reg [9:0] last_key;

wire shift_down;
wire [511:0] key_down;
wire [8:0] last_change;
wire been_ready;

assign shift_down = (key_down[LEFT_SHIFT_CODES] == 1'b1 || key_down[RIGHT_SHIFT_CODES] == 1'b1) ? 1'b1 : 1'b0;
```

```
always @ (*) begin
    case (last_change)
        KEY_CODES[00] : key_num = 4'b0000;
        KEY_CODES[01] : key_num = 4'b0001;
        KEY_CODES[02] : key_num = 4'b0010;
        KEY_CODES[03] : key_num = 4'b0011;
        KEY_CODES[04] : key_num = 4'b0100;
        KEY_CODES[05] : key_num = 4'b0101;
        KEY_CODES[06] : key_num = 4'b0110;
        KEY_CODES[07] : key_num = 4'b0111;
        KEY_CODES[08] : key_num = 4'b1000;
        KEY_CODES[09] : key_num = 4'b1001;
        KEY_CODES[10] : key_num = 4'b0000;
        KEY_CODES[11] : key_num = 4'b0001;
        KEY_CODES[12] : key_num = 4'b0010;
        KEY_CODES[13] : key_num = 4'b0011;
        KEY_CODES[14] : key_num = 4'b0100;
        KEY_CODES[15] : key_num = 4'b0101;
        KEY_CODES[16] : key_num = 4'b0110;
        KEY_CODES[17] : key_num = 4'b0111;
        KEY_CODES[18] : key_num = 4'b1000;
        KEY_CODES[19] : key_num = 4'b1001;
        default       : key_num = 4'b1111;
    endcase
end
```

Then I instantiated the two clock divider modules, along with debounce, one_pulse, and KeyboardDecoder:

```
clock_divider_26 clkdiv26(.clk(clk), .clk_div_26(clk_div_26));
clock_divider_16 clkdiv16(.clk(clk), .clk_div_16(clk_div_16));

debounce debounce_rst(.pb_debounced(rst_debounced), .pb(rst), .clk(clk_div_16));
one_pulse onepulse_rst(.pb_debounced(rst_debounced), .clk(clk_div_16), .pb_one_pulse(rst_one_pulse));
debounce debounce_mode(.pb_debounced(mode_debounced), .pb(mode), .clk(clk_div_16));
one_pulse onepulse_mode(.pb_debounced(mode_debounced), .clk(clk_div_16), .pb_one_pulse(mode_one_pulse));

KeyboardDecoder key_de (
    .key_down(key_down),
    .last_change(last_change),
    .key_valid(been_ready),
    .PS2_DATA(PS2_DATA),
    .PS2_CLK(PS2_CLK),
    .rst(rst),
    .clk(clk)
);
```

Then to get the exact timing for the passengers to decrease with frequency clk/(2^26), which I made in the always block of regular clk, I need to use flag for it to get the correct timing, which is why I have these two always blocks:

```
always@(posedge clk or posedge rst_one_pulse)
begin
   if(rst_one_pulse == 1'b1)
       flag_100 <= 1'b1;

   else
   begin
       if(flag_100 == flag_26)
           flag_100 <= ~flag_100;
   end
end

always@(posedge clk_div_26 or posedge rst_one_pulse)
begin
    if(rst_one_pulse == 1'b1)
        flag_26 = 1'b0;
    else
        flag_26 <= ~flag_26;
end
```

I then assign the values of passenger_1, passenger_2, and passenger_3 to the values of next_passenger_1, next_passenger_2, and next_passenger_3 respectively here:

```
always@(posedge clk or posedge rst_one_pulse)
begin
    if (rst_one_pulse == 1'b1)
    begin
        passenger_1 <= 4'd0;
        passenger_2 <= 4'd0;
        passenger_3 <= 4'd0;
    end

    else
    begin
        passenger_1 <= next_passenger_1;
        passenger_2 <= next_passenger_2;
        passenger_3 <= next_passenger_3;
    end
end
```

Then for the values of next_passenger_1, next_passenger_2, and next_passenger_3, which depends on the user's keyboard input, I designed it as follows in an always @(posedge clk or posedge rst_one_pulse):

As usual, if rst button happens to be pressed, then all values will be set back to 4'b000.

```
if (rst_one_pulse)
begin
    next_passenger_1 <= 4'd0;
    next_passenger_2 <= 4'd0;
    next_passenger_3 <= 4'd0;
end
```

Else, the values will be able to increase if been_ready && key_down[last_change] == 1'b1, and if key_num == 4'b0001, which indicates that 1 was pressed, then next_passenger_1 will increase its value from 4'b0000 to 4'b0001 to 4'b0011 to 4'b0111 to 4'b1111 every time it was pressed and when it reaches 4'b1111, it will stay the same even when the user presses the keyboard key. The same thing goes for next_passenger_2 and next_passenger_3 when 2 or 3 was pressed.

```
if(key_num == 4'b0001)
begin
    if(passenger_1 == 4'b0000)
        next_passenger_1 <= 4'b0001;
    else if(passenger_1 == 4'b0001)
        next_passenger_1 <= 4'b0011;
    else if(passenger_1 == 4'b0011)
        next_passenger_1 <= 4'b0111;
    else if(passenger_1 == 4'b0111)
        next_passenger_1 <= 4'b1111;
    else
        next_passenger_1 <= next_passenger_1;
end
```

```
else if(key_num == 4'b00010)
begin
    if(passenger_2 == 4'b0000)
        next_passenger_2 <= 4'b0001;
    else if(passenger_2 == 4'b0001)
        next_passenger_2 <= 4'b0011;
    else if(passenger_2 == 4'b0011)
        next_passenger_2 <= 4'b0111;
    else if(passenger_2 == 4'b0111)
        next_passenger_2 <= 4'b1111;
    else
        next_passenger_2 <= next_passenger_2;
end
```

```
else if(key_num == 4'b0011)
begin
    if(passenger_3 == 4'b0000)
        next_passenger_3 <= 4'b0001;
    else if(passenger_3 == 4'b0001)
        next_passenger_3 <= 4'b0011;
    else if(passenger_3 == 4'b0011)
        next_passenger_3 <= 4'b0111;
    else if(passenger_3 == 4'b0111)
        next_passenger_3 <= 4'b1111;
    else
        next_passenger_3 <= next_passenger_3;
end
```

To decrease the number of passengers one by one when the monorail is picking them up, it happens when flag_100 is equal to flag_26, to pick up the passengers one by one with frequency of clk/(2^26). The passengers can only be picked up when the monorail's state is `STAY. Therefore, inside the big if(flag_100 == flag_26) statement, there is another big if(state == `STAY) statement, then inside the if statement, there are conditions where the monorail is either at stop1, stop2, or stop3. When the monorail is at stop1, only the passengers at stop1 will decrease from 4'b1111 to 4'b0111 to 4'b0011 to 4'b0001 to 4'b0000 and when it reaches 4'b0000, it will stay the same. The same thing goes for stop2 and stop3. So the code will look like this:

```
if(flag_100 == flag_26)
```

```
if(state == `STAY)
```

```
if(cur_pos == `stop1)
begin
  if(passenger_1 == 4'b1111)
       next_passenger_1 <= 4'b0111;
   else if(passenger_1 == 4'b0111)
       next_passenger_1 <= 4'b0011;
   else if(passenger_1 == 4'b0011)
       next_passenger_1 <= 4'b0001;
   else if(passenger_1 == 4'b0001)
       next_passenger_1 <= 4'b0000;
    else
       next_passenger_1 <= next_passenger_1;
end

 else
   next_passenger_1 <= next_passenger_1;
```

```
if(cur_pos == `stop2)
begin
  if(passenger_2 == 4'b1111)
       next_passenger_2 <= 4'b0111;
   else if(passenger_2 == 4'b0111)
       next_passenger_2 <= 4'b0011;
   else if(passenger_2 == 4'b0011)
       next_passenger_2 <= 4'b0001;
   else if(passenger_2 == 4'b0001)
       next_passenger_2 <= 4'b0000;
    else
       next_passenger_2 <= next_passenger_2;
 end

else
    next_passenger_2 <= next_passenger_2;
```

```
 if(cur_pos == `stop3)
 begin
     if(passenger_3 == 4'b1111)
        next_passenger_3 <= 4'b0111;
    else if(passenger_3 == 4'b0111)
        next_passenger_3 <= 4'b0011;
    else if(passenger_3 == 4'b0011)
        next_passenger_3 <= 4'b0001;
    else if(passenger_3 == 4'b0001)
        next_passenger_3 <= 4'b0000;
     else
        next_passenger_3 <= next_passenger_3;
 end

 else
     next_passenger_3 <= next_passenger_3;
```

I then assign LED[3:0] as passenger_1, LED[7:4] as passenger_2, LED[11:8] as passenger_3, and LED[15] as 1'b0 if auto == 1'b1, and 1'b1 if auto == 1'b0.

```
assign LED[3:0] = passenger_1;
assign LED[7:4] = passenger_2;
assign LED[11:8] = passenger_3;
assign LED[15] = (auto == 1'b1) ? 1'b0 : 1'b1;
```

This part is for the auto and fixed mode part. If rst happens then auto will be set back to 1'b1. Otherwise, we then look at mode_one_pulse, if mode_one_pulse == 1'b1, which means that the mode button was pressed, then we need to chech if start is equal to 1'b1 or 1'b0. If start is equal to 1'b1, then the mode button is enabled, so auto's value can be changed, so I just negate it since it can only change from 1'b1 to 1'b0 or 1'b0 to 1'b1. Otherwise, if start == 1'b0, which means that the mode button was still not enabled, auto will remain as its own current value. Else, if the mode button was not pressed, auto will remain as its current value too.

```
always@(posedge clk or posedge rst_one_pulse or posedge mode_one_pulse)
begin
    if(rst_one_pulse == 1'b1)
    begin
        auto <= 1'b1;
    end

    else
    begin
        if(mode_one_pulse == 1'b1)
        begin
            if(start == 1'b1)
                auto <= ~auto;

            else
                auto <= auto;
        end

        else
            auto <= auto;
    end
end
```

Code for mode button, to control auto and fixed mode.

This is the part that handles the states and positions. state takes the value of next_state and cur_pos takes the value of next_pos in always@(posedge clk or posedge rst_one_pulse) block if rst_one pulse is not equal to 1'b1. If it is equal to 1'b1, then state will hold the value `STAY and cur_pos will hold the value `stop1 which is their initial values.

```
always@(posedge clk or posedge rst_one_pulse)
begin
    if(rst_one_pulse == 1'b1)
    begin
        state <= `STAY;
        cur_pos <= `stop1;
    end

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

In this always@(*) block, it will handle the change of states, in which values will be assigned to next_state. As per usual, if rst occurs then everything will be set back to its initial values, otherwise, everything works depending on their conditions. Inside the else block, there are three main conditions: if state == `STAY, if state == `CLOCKWISE, or if state == `COUNTERCLOCKWISE.

if(state == `STAY)

We first look if the current stop of the monorail has passengers queueing or not, and if no passengers are waiting at all stops, then the next_pos of the monorail will stay the same, which is why next_pos = cur_pos, and next_state will also be the same, which is `STAY, so next_state = `STAY.

```
 if(((cur_pos == `stop1 && passenger_1 > 4'd0) || (cur_pos == `stop2 && passenger_2 > 4'd0) || (cur_pos == `stop3 && passenger_3 > 4'd0)) ||
    (passenger_1 == 4'd0 && passenger_2 == 4'd0 && passenger_3 == 4'd0))
begin
    next_pos = cur_pos;
    next_state = `STAY;
end
```

Else, we then have another two main conditions that we need to bring into consideration, which is when the monorail is in auto mode or fixed mode. If the monorail is in fixed mode (auto == 1'b0), at first I only used next_state = dir if(hold == 1'b0) else start_count = 1'b1 and next_state = `STAY to hold the position if there happens to be a condition where the monorail should go through a stop that has no passengers at all. Dir itself is the current direction that was 'saved', for example in the auto mode, my previous direction is clockwise, then dir will

become clockwise, but if I change to counterclockwise, then dir will become counterclockwise. I did this because previously my controller will skip the stop that has no passengers at all and go directly to the stop that has passengers, so I hold it for one clock cycle. However, I noticed that this cannot be implemented to all stops, so I made special conditions for certain stops that does not work using the method I have described above. The first one is for when ((dir == `CLOCKWISE && cur_pos == `stop2 && passenger_1 > 4'b0000 && passenger_3 == 4'b0000) || in_3 == 1'b1). If hold == 1’b0 or hold3 == 1’b0, which means that the monorails did not hold for a while to stop at the stop that has no passengers, then next_state will become dir. Else, if(((dir == `CLOCKWISE && cur_pos == `stop2 && passenger_1 > 4'b0000 && passenger_3 == 4'b0000) || (dir == `COUNTERCLOCKWISE && cur_pos == `stop3 && passenger_1 > 4'b0000 && passenger_2 == 4'b0000) || (dir == `COUNTERCLOCKWISE && cur_pos == `stop1 && passenger_2 > 4'b0000 && passenger_3 == 4'b0000)) || flag_3 == 1'b1), then start_count3 will become 1’b1 to start counting to hold the monorail at its position, and next_state will remain as `STAY. Else, start_count will become 1’b1 to start counting to hold the monorail at its position, and next_state will remain as `STAY. Here, start_counting3 and start_counting counts up to different values, which will be explained later below, so we need to reg variables.

```
if((dir == `CLOCKWISE && cur_pos == `stop2 && passenger_1 > 4'b0000 && passenger_3 == 4'b0000) || in_3 == 1'b1)
begin
    if(hold == 1'b0)
    begin
        next_state = dir;
    end

    else if(hold3 == 1'b0)
    begin
        next_state = dir;
    end

    else if(((dir == `CLOCKWISE && cur_pos == `stop2 && passenger_1 > 4'b0000 && passenger_3 == 4'b0000) ||
    (dir == `COUNTERCLOCKWISE && cur_pos == `stop3 && passenger_1 > 4'b0000 && passenger_2 == 4'b0000) ||
    (dir == `COUNTERCLOCKWISE && cur_pos == `stop1 && passenger_2 > 4'b0000 && passenger_3 == 4'b0000)) || flag_3 == 1'b1)
    begin
        start_count3 = 1'b1;
        next_state = `STAY;
    end

    else
    begin
        start_count = 1'b1;
        next_state = `STAY;
    end
end
```

The next else statement, which handles the other conditions, I put flag_first == 1’b0 because I had a tad bit of problem when displaying for the fisrt time when switching to fixed mode from auto, because the display will always show the current direction even when it was not supposed to. So I added flag_first for when it is the first time the monorail entered fixed mode from auto mode. Inside, we meet the condition if(((cur_pos == `stop1 && passenger_2 > 4'b0000) || (cur_pos == `stop2 && passenger_3 > 4'b0000) || (cur_pos == `stop3 && passenger_1 >

4'b0000)) || ((cur_pos == `stop1 && passenger_3 > 4'b0000) || (cur_pos == `stop2 && passenger_1 > 4'b0000) || (cur_pos == `stop3 && passenger_2 > 4'b0000))). That is, when other stops has passengers queueing, then first will be set to 1’b1, which will then be used to display the direction at the seven-segment display, and flag_first will be set to 1’b1. Else if this condition was not met, then first will stay as 1’b0 but flag_first will be 1’b1. Then, we use similar methods as how we handle the previous condition, this time using hold and hold2. When either one of then is equal to 1’b0, then next_state will become dir. Else, if we meet these conditions: (((dir == `CLOCKWISE && cur_pos == `stop2 && passenger_1 > 4'b0000 && passenger_3 == 4'b0000) || (dir == `COUNTERCLOCKWISE && cur_pos == `stop3 && passenger_1 > 4'b0000 && passenger_2 == 4'b0000) || (dir == `COUNTERCLOCKWISE && cur_pos == `stop1 && passenger_2 > 4'b0000 && passenger_3 == 4'b0000)) || flag == 1'b1), then start_count2 will become 1’b1 to start counting to hold the monorail at its position, and next_state will remain as `STAY. Else, start_count will become 1’b1 to start counting to hold the monorail at its position, and next_state will remain as `STAY. Here, start_counting2 and start_counting counts up to different values, which will be explained later below, so we need to reg variables. Else, if flag_first == 1’b1, then everything will stay the same as above except for the conditions to handle first and flag_first.

```
else
begin
    if(flag_first == 1'b0)
    begin
        if(((cur_pos == `stop1 && passenger_2 > 4'b0000) || (cur_pos == `stop2 && passenger_3 > 4'b0000) || (cur_pos == `stop3 && passenger_1 > 4'b0000)) ||
            ((cur_pos == `stop1 && passenger_3 > 4'b0000) || (cur_pos == `stop2 && passenger_1 > 4'b0000) || (cur_pos == `stop3 && passenger_2 > 4'b0000)))
            begin
                first = 1'b1;
                flag_first = 1'b1;
            end

        else
        begin
            first = 1'b0;
            flag_first = 1'b1;
        end

        if(hold == 1'b0)
        begin
            next_state = dir;
        end

        else if(hold2 == 1'b0)
        begin
            next_state = dir;
        end

        else if(((dir == `CLOCKWISE && cur_pos == `stop2 && passenger_1 > 4'b0000 && passenger_3 == 4'b0000) ||
        (dir == `COUNTERCLOCKWISE && cur_pos == `stop3 && passenger_1 > 4'b0000 && passenger_2 == 4'b0000) ||
        (dir == `COUNTERCLOCKWISE && cur_pos == `stop1 && passenger_2 > 4'b0000 && passenger_3 == 4'b0000)) || flag == 1'b1)
        begin
            start_count2 = 1'b1;
            next_state = `STAY;
        end

        else
        begin
            start_count = 1'b1;
            next_state = `STAY;
        end
    end
```

```
    else
    begin
        if(hold == 1'b0)
        begin
            next_state = dir;
        end

        else if(hold2 == 1'b0)
        begin
            next_state = dir;
        end

        else if(((dir == `CLOCKWISE && cur_pos == `stop2 && passenger_1 > 4'b0000 && passenger_3 == 4'b0000) ||
        (dir == `COUNTERCLOCKWISE && cur_pos == `stop3 && passenger_1 > 4'b0000 && passenger_2 == 4'b0000) ||
        (dir == `COUNTERCLOCKWISE && cur_pos == `stop1 && passenger_2 > 4'b0000 && passenger_3 == 4'b0000)) || flag == 1'b1)
        begin
            start_count2 = 1'b1;
            next_state = `STAY;
        end

        else
        begin
            start_count = 1'b1;
            next_state = `STAY;
        end
    end
end
```

After having done dealing with the fixed mode, we can take a look at the auto mode. In the auto mode, start will become 1’b1 to enable the mode button, and then we have two conditions, for when dir == `CLOCKWISE or when dir == `COUNTERCLOCKWISE. If dir == `CLOCKWISE, we put the conditions that let next_state = `CLOCKWISE to be the priority, because there can be a case when two stops has passengers waiting at the same time. To prevent it to not following the trend, we need to set priorities. Therefore, if cur_stop is at stop1 and there are passengers queueing at stop2, or when cur_stop is at stop2 and there are passengers queueing at stop3, or when cur_stop is at stop3 and there are passengers queueing at stop1, then the next_state will be `CLOCKWISE. Else, if cur_stop is at stop1 and there are passengers queueing at stop3, or when cur_stop is at stop2 and there are passengers queueing at stop1, or when cur_stop is at stop3 and there are passengers queueing at stop2, then the next_state will be `COUNTERCLOCKWISE. Else, if the two conditions are not met, the monorail stays at `STAY state. Then if dir == `COUNTERCLOCKWISE, we switch the positions of the two conditions in the if(dir == `CLOCKWISE), so that it first look at , if cur_stop is at stop1 and there are passengers queueing at stop3, or when cur_stop is at stop2 and there are passengers queueing at stop1, or when cur_stop is at stop3 and there are passengers queueing at stop2, then the next_state will be `COUNTERCLOCKWISE, then proceed to if cur_stop is at stop1 and there are passengers queueing at stop2, or when cur_stop is at stop2 and there are passengers queueing at stop3, or when cur_stop is at stop3 and there are passengers queueing at stop1, then the next_state will be `CLOCKWISE, and lastly

if the two conditions are not met, the monorail stays at `STAY state.

```
else //auto == 1'b1
begin
    start = 1'b1;

    if(dir == `CLOCKWISE)
    begin
        if((cur_pos == `stop1 && passenger_2 > 4'b0000) || (cur_pos == `stop2 && passenger_3 > 4'b0000) || (cur_pos == `stop3 && passenger_1 > 4'b0000))
            next_state = `CLOCKWISE;

        else if((cur_pos == `stop1 && passenger_3 > 4'b0000) || (cur_pos == `stop2 && passenger_1 > 4'b0000) || (cur_pos == `stop3 && passenger_2 > 4'b0000))
            next_state = `COUNTERCLOCKWISE;

        else
            next_state = `STAY;
    end

    else if(dir == `COUNTERCLOCKWISE)
    begin
        if((cur_pos == `stop1 && passenger_3 > 4'b0000) || (cur_pos == `stop2 && passenger_1 > 4'b0000) || (cur_pos == `stop3 && passenger_2 > 4'b0000))
            next_state = `COUNTERCLOCKWISE;

        else if((cur_pos == `stop1 && passenger_2 > 4'b0000) || (cur_pos == `stop2 && passenger_3 > 4'b0000) || (cur_pos == `stop3 && passenger_1 > 4'b0000))
            next_state = `CLOCKWISE;

        else
            next_state = `STAY;
    end

    else
        next_state = `STAY;
end
```

else if(state == `CLOCKWISE)

start_count, start_ count 2, and start_ count 3 will all be set back to 1'b0, and for the next_pos, we see if flag_movement == 1'b1, which indicates that 4 clock cycles have been reached, then the monorail will switch position by adding 1 to cur_pos, but if cur_pos is stop3, then we need to set next_pos as stop1. Else, if 4 clock cycles have not been reached, or in other words, flag_movement == 1'b0, then next_pos will stay the same as cur_pos. As for the next_state, its is similar to how we handle next_pos, it is just next_state will either be `STAY if 4 clock cycle has been reaced, and `CLOCKWISE if 4 clock cycles have not been reached.

```
else if(state == `CLOCKWISE)
begin
    start_count = 1'b0;
    start_count2 = 1'b0;
    start_count3 = 1'b0;
    next_pos = (flag_movement == 1'b1) ? ((cur_pos == `stop3) ? `stop1 : (cur_pos + 4'd1)) : cur_pos;
    next_state = (flag_movement == 1'b1) ? `STAY : `CLOCKWISE;
end
```

else if(state == `COUNTERCLOCKWISE)

start_count, start_ count 2, and start_ count 3 will all be set back to 1'b0, and for the next_pos, we see if flag_movement_ctr == 1'b1, which indicates that 4 clock cycles have been reached, then the monorail will switch position by subtracting 1 to cur_pos, but if cur_pos is stop1, then we need to set next_pos as stop3. Else, if 4 clock cycles have not been reached, or in other words, flag_movement_ctr == 1'b0, then next_pos will stay the same as cur_pos. As for the next_state, its is similar to how we handle next_pos, it is just next_state will either be `STAY if 4 clock cycle has been reached, and `COUNTERCLOCKWISE if 4 clock cycles have not been reached.

```
else if(state == `COUNTERCLOCKWISE)
begin
    start_count = 1'b0;
    start_count2 = 1'b0;
    start_count3 = 1'b0;
    next_pos = (flag_movement_ctr == 1'b1) ? ((cur_pos == `stop1) ? `stop3 : (cur_pos - 4'd1)) : cur_pos;
    next_state = (flag_movement_ctr == 1'b1) ? `STAY : `COUNTERCLOCKWISE;
end
```

This marks the end of the always block that is dealing with next_state and next_pos. We can then look at the always@(posedge clk_div_26 or posedge rst_one_pulse).

If rst occurs, then everything in this always block will be set back to its initial value, otherwise, we have three conditions which are the states.

```
if(rst_one_pulse == 1'b1)
begin
    hold <= 1'b1;
    hold2 <= 1'b1;
    hold3 <= 1'b1;
    dir <= `CLOCKWISE;
    movement <= 4'd0;
    movement_ctr <= 4'd0;
    flag_movement <= 1'b0;
    flag_movement_ctr <= 1'b0;
    flag <= 1'b0;
    flag_3 <= 1'b0;
    cnt <= 4'd0;
    in_3 <= 1'b0;
end
```

If state == `CLOCKWISE, then hold, hold2, and hold3 will be set to 1'b1, and dir to `CLOCKWISE, flag and flag_3 will be set back to 1'b0, cnt and cnt_3 will be set back to 4'd0, then in_3 will be set back to 1'b0. For movement, if flag_movement is equal to 1'b0, then movement will keep increasing by 4'd1 in each clock cycle, else movement will hold its value. For flag_movement, when movement reached 4'd4, which means that 4 clock cycle has passed, flag_movement will become 1'b1, otherwise it will become 1'b0.

```
if(state == `CLOCKWISE)
begin
    hold <= 1'b1;
    hold2 <= 1'b1;
    hold3 <= 1'b1;
    flag <= 1'b0;
    flag_3 <= 1'b0;
    dir <= `CLOCKWISE;
    cnt <= 4'd0;
    cnt_3 <= 4'd0;
    in_3 <= 1'b0;
    movement <= (flag_movement == 1'b0) ? (movement + 4'd1) : movement;
    flag_movement <= (movement == 4'd4) ? 1'b1 : 1'b0;
end
```

If state == `COUNTERCLOCKWISE, then hold, hold2, and hold3 will be set to 1'b1, and dir to `COUNTERCLOCKWISE, flag and flag_3 will be set back to 1'b0, cnt and cnt_3 will be set back to 4'd0, then in_3 will be set back to 1'b0. For movement_ctr, if flag_movement_ctr is equal to 1'b0, then movement_ctr will keep increasing by 4'd1 in each clock cycle, else movement_ctr will hold its value. For flag_movement_ctr, when movement_ctr reached 4'd4, which means that 4 clock cycle has passed, flag_movement_ctr will become 1'b1, otherwise it will become 1'b0.

```
else if(state == `COUNTERCLOCKWISE)
begin
    hold <= 1'b1;
    hold2 <= 1'b1;
    hold3 <= 1'b1;
    flag <= 1'b0;
    flag_3 <= 1'b0;
    dir <= `COUNTERCLOCKWISE;
    cnt <= 4'd0;
    cnt_3 <= 4'd0;
    in_3 <= 1'b0;
    movement_ctr <= (flag_movement_ctr == 1'b0) ? (movement_ctr + 4'd1) : movement_ctr;
    flag_movement_ctr <= (movement_ctr == 4'd4) ? 1'b1 : 1'b0;
end
```

Else, if state == `STAY, we look at start_count, start_count2, and start_counr3 which I mention before to deal with the monorail skipping stops that has no passengers waiting while in fixed mode. If start_count == 1’b1, then cnt will increase by 4’d1 in one clock cycle of clk/(2^26) and by the time cnt reaches 4’d1, hold will become 1’b0 to stop the monorail from holding itself at the current stop that has no passengers. If start_count2 == 1’b1, then cnt will increase by 4’d1 in one clock cycle of clk/(2^26) and by the time cnt reaches 4’d2, hold2 will become 1’b0 to stop the monorail from holding itself at the current stop that has no passengers. Here, flag will become 1’b1 to let start_count2 to keep holding its value as 1’b1 and for the next_state to stay st `STAY. This is the perfect timing I got for certain conditions to hold the monorail, and the conditions are when (dir == `COUNTERCLOCKWISE && cur_pos == `stop3 && passenger_1 > 4'b0000 && passenger_2 == 4'b0000) || (dir == `COUNTERCLOCKWISE && cur_pos == `stop1 && passenger_2 > 4'b0000 && passenger_3 == 4'b0000). Next, if start_count3 == 1’b1, in_3 will become 1’b1 for it to keep entering the if((dir == `CLOCKWISE && cur_pos == `stop2 && passenger_1 > 4'b0000 && passenger_3 == 4'b0000) || in_3 == 1'b1) on the previous always block, cnt_3 will increase by 4’d1 in every clock cycle, and flag_3 will become 1’b1 to let start_count3 to keep holding its value as 1’b1 and for the next_state to stay st `STAY. By the time cnt_3 == 4’d6, hold3 will change its value to 1’b0 to let the monorail change its state. This is the perfect timing I got for the condition if(dir == `CLOCKWISE && cur_pos == `stop2 && passenger_1 > 4'b0000 && passenger_3 == 4'b0000). Then outside all if-else statements, we set flag_movement, flag_movement_ctr, movement, and movement_ctr back to its initial value to be used next.

```
else if(state == `STAY)
begin
    if(start_count == 1'b1)
    begin
        cnt <= cnt + 4'd1;

        if(cnt == 4'd1)
            hold <= 1'b0;
    end

    else if(start_count2 == 1'b1)
    begin
        cnt <= cnt + 4'd1;
        flag <= 1'b1;

        if(cnt == 4'd2)
            hold2 <= 1'b0;
    end

    else if(start_count3 == 1'b1)
    begin
        in_3 <= 1'b1;
        cnt_3 <= cnt_3 + 4'd1;
        flag_3 <= 1'b1;

        if(cnt_3 == 4'd6)
            hold3 <= 1'b0;
    end

    flag_movement <= 1'b0;
    flag_movement_ctr <= 1'b0;
    movement <= 4'd0;
    movement_ctr <= 4'd0;
end
```

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];
```

For the seven-segment display, I use refresher as the case to be able to display the digits simultaneously. Then for the rightmost digit, which displays the current position (current stop) of the monorail, I directly assign it as cur_pos since cur_pos holds the value 4'd1 for `stop1, 4'd2 for `stop2, and 4'd3 for `stop3.

```
2'b00:
begin
    DIGIT <= 4'b1110;
    display_num <= cur_pos;
end
```

For the second rightmost digit which displays the direction when the monorail is at state `STAY and animation when the monorail is at either state `CLOCKWISE or `COUNTERCLOCKWISE, I need to use if-else if-else statement to handle the conditions.

If state == `CLOCKWISE, I just need to add 4'd4, which shows the first animation of the clockwise direction, by movement, which will increase by 4'd1 in every clock cycle of clk/(2^26).

```
if(state == `CLOCKWISE)
    display_num <= 4'd4 + movement;
```

Else if state == `COUNTERCLOCKWISE, I just need to add 4'd9, which shows the first animation of the counterclockwise direction, by movement_ctr, which will increase by 4'd1 in every clock cycle of clk/(2^26).

```
else if(state == `COUNTERCLOCKWISE)
    display_num <= 4'd9 + movement_ctr;
```

Else, if state == `STAY, we have two conditions, whether the monorail is in auto mode or fixed mode. When the monorail is in fixed mode, under the condition if(dir == `CLOCKWISE), if((cur_pos == `stop1 && passenger_2 > 4'd0) || (cur_pos == `stop2 && passenger_3 > 4'd0) || (cur_pos == `stop3 && passenger_1 > 4'd0) || (cur_pos == `stop1 && passenger_3 > 4'd0) || (cur_pos == `stop2 && passenger_1 > 4'd0) || (cur_pos == `stop3 && passenger_2 > 4'd0)) display_num will hold the value of 4'd9 if first == 1'b1, that is, no display at all, and 4'd8 otherwise, which is the clockwise symbol. Else, if the above condition was not met, then the display will show nothing. That is, display_num holds the value 4'd9.

The same thing goes for the condition dir == `COUNTERCLOCKWISE, it is just that we only need to replace the value of 4'd8 to 4'd13, and it became like if((cur_pos == `stop1 && passenger_2 > 4'd0) || (cur_pos == `stop2 && passenger_3 > 4'd0) || (cur_pos == `stop3 && passenger_1 > 4'd0) || (cur_pos == `stop1 && passenger_3 > 4'd0) || (cur_pos == `stop2 && passenger_1 > 4'd0) || (cur_pos == `stop3 && passenger_2 > 4'd0)) display_num will hold the value of 4'd13, which is the counterclockwise symbol, and eliminate the condition for first since everything works perfectly fine here.

```
if(auto == 1'b0)
begin
   if(dir == `CLOCKWISE)
    begin
        if((cur_pos == `stop1 && passenger_2 > 4'd0) || (cur_pos == `stop2 && passenger_3 > 4'd0) || (cur_pos == `stop3 && passenger_1 > 4'd0))
            display_num <= (first == 1'b1) ? 4'd9 : 4'd8;

        else if((cur_pos == `stop1 && passenger_3 > 4'd0) || (cur_pos == `stop2 && passenger_1 > 4'd0) || (cur_pos == `stop3 && passenger_2 > 4'd0))
            display_num <= (first == 1'b1) ? 4'd9 : 4'd8;

        else
           display_num <= 4'd9;
    end

    else if(dir == `COUNTERCLOCKWISE)
    begin
       if((cur_pos == `stop1 && passenger_2 > 4'd0) || (cur_pos == `stop2 && passenger_3 > 4'd0) || (cur_pos == `stop3 && passenger_1 > 4'd0))
            display_num <= 4'd13;

        else if((cur_pos == `stop1 && passenger_3 > 4'd0) || (cur_pos == `stop2 && passenger_1 > 4'd0) || (cur_pos == `stop3 && passenger_2 > 4'd0))
            display_num <= 4'd13;

        else
           display_num <= 4'd9;
    end
end
```

Else, if the monorail is in auto mode, we need to have priority for different values of dir. If dir == `CLOCKWISE, we need to prioritize the display for clockwise symbol. The condition for the clockwise symbol to occur was when ((cur_pos == `stop1 && passenger_2 > 4'd0) || (cur_pos == `stop2 && passenger_3 > 4'd0) || (cur_pos == `stop3 && passenger_1 > 4'd0)), and for the counterclockwise symbol it is ((cur_pos == `stop1 && passenger_3 > 4'd0) || (cur_pos == `stop2 && passenger_1 > 4'd0) || (cur_pos == `stop3 && passenger_2 > 4'd0)). Apart from these two conditions, there would not be any display. For when dir == `COUNTERCLOCKWISE, we just need to switch the position of the if-else if conditions inside the if(dir == `CLOCKWISE) block.

```
else if(auto == 1'b1)
begin
    if(dir == `CLOCKWISE)
    begin
       if((cur_pos == `stop1 && passenger_2 > 4'd0) || (cur_pos == `stop2 && passenger_3 > 4'd0) || (cur_pos == `stop3 && passenger_1 > 4'd0))
            display_num <= 4'd8;

        else if((cur_pos == `stop1 && passenger_3 > 4'd0) || (cur_pos == `stop2 && passenger_1 > 4'd0) || (cur_pos == `stop3 && passenger_2 > 4'd0))
            display_num <= 4'd13;

        else
            display_num <= 4'd9;
    end

    else if(dir == `COUNTERCLOCKWISE)
    begin
        if((cur_pos == `stop1 && passenger_3 > 4'd0) || (cur_pos == `stop2 && passenger_1 > 4'd0) || (cur_pos == `stop3 && passenger_2 > 4'd0))
            display_num <= 4'd13;

        else if((cur_pos == `stop1 && passenger_2 > 4'd0) || (cur_pos == `stop2 && passenger_3 > 4'd0) || (cur_pos == `stop3 && passenger_1 > 4'd0))
            display_num <= 4'd8;

        else
            display_num <= 4'd9;
    end
end
```

For the other digits on the seven-segment display, we don't need to display anything so we just need to set the value of display_num to 4'd9, the same as default case.

(4'd14 is the same as 4'd9, both shows nothing).

```
2'b10:
begin
    DIGIT <= 4'b1011;
    display_num <= 4'd9;
end

2'b11:
begin
    DIGIT <= 4'b0111;
    display_num <= 4'd9;
end

default:
begin
    DIGIT <= 4'b1111;
    display_num <= 4'd14;
end
```

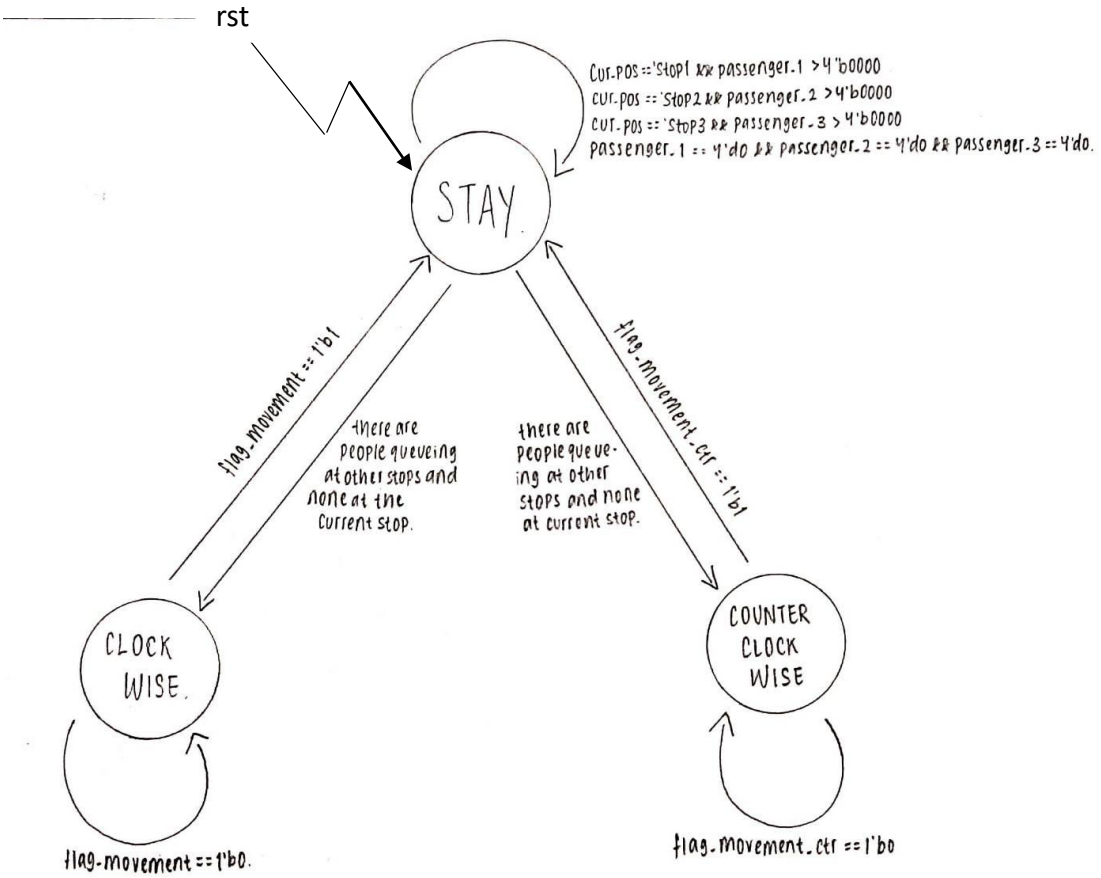
```
always @ (*) begin
    case (display_num)
        4'd0 : DISPLAY = 7'b1000000;    //0000
        4'd1 : DISPLAY = 7'b1111001;    //0001
        4'd2 : DISPLAY = 7'b0100100;    //0010
        4'd3 : DISPLAY = 7'b0110000;    //0011
        4'd4 : DISPLAY = 7'b1111111;
        4'd5 : DISPLAY = 7'b1101111;    //1st clockwise
        4'd6 : DISPLAY = 7'b1001111;
        4'd7 : DISPLAY = 7'b1001110;
        4'd8 : DISPLAY = 7'b1001100;
        4'd9 : DISPLAY = 7'b1111111;
        4'd10 : DISPLAY = 7'b1111011;   //1st counterclockwise
        4'd11 : DISPLAY = 7'b1111001;
        4'd12 : DISPLAY = 7'b1111000;
        4'd13 : DISPLAY = 7'b1011000;
        4'd14 : DISPLAY = 7'b1111111;
        default : DISPLAY = 7'b1111111;
    endcase
end
```

For the numbers to appear correctly on the digits of the seven-segment display, the binary digit 0 is used activate a single segment instead of 1, so I designed the case which will take display_num to determine which number to display on the digit, as shown below. The case is put inside an always@(*) block because this part will be used every time it is called.

The last part were the clock divider modules for frequency of clk/(2^16) and clk/(2^26).

```
module clock_divider_16(clk, clk_div_16);
    parameter n = 16;
    input clk;
    output clk_div_16;

    reg[15:0] num = 16'd0;
    wire [15:0] next_num;

    always @(posedge clk)
    begin
        num = next_num;
    end

    assign next_num = num+1;
    assign clk_div_16 = num[n-1];
endmodule
```

```
module clock_divider_26(clk, clk_div_26);
    parameter n = 26;
    input clk;
    output clk_div_26;

    reg[25:0] num = 26'd0;
    wire [25:0] next_num;

    always @(posedge clk)
    begin
        num = next_num;
    end

    assign next_num = num+1;
    assign clk_div_26 = num[n-1];
endmodule
```



As we can see in the state diagram, when the controller is being reset, the controller goes back to STAY state. The controller will stays at STAY state if there are passengers queueing at its current stop (to pick up passengers), in which the conditions to fulfill are when ((cur_pos == `stop1 && passenger_1 > 4'd0) || (cur_pos == `stop2 && passenger_2 > 4'd0) || (cur_pos == `stop3 && passenger_3 > 4'd0)), and when no passengers are queueing at every stops, or when being written as a code: (passenger_1 == 4'd0 && passenger_2 == 4'd0 && passenger_3 == 4'd0). The controller will go from STAY state to either CLOCKWISE when flag_movement == 1’b1, or COUNTERCLOCKWISE when flag_movement_ctr == 1’b1. flag_movement and flag_movement_ctr are flags to indicate that 4 clock cycle has passed and the monorail can change state to STAY, as mentioned in the spec. Otherwise, when flag_movement or flag_movement_ctr are still equal to 1’b0, the monorail will stay at CLOCKWISE or COUNTERCLOCKWISE state while displaying the animation of the monorail. As for the monorail to change its state from either CLOCKWISE or COUNTERCLOCKWISE to STAY, the conditions to fulfill are when there are people queueing at other stops and none at the monorail’s current stop.

In fixed mode, the monorail can either switch from STAY to CLOCKWISE and vice versa, or STAY to COUNTERCLOCKWISE and vice versa. When the controller is in auto mode, the monorail can go to CLOCKWISE or COUNTERCLOCKWISE, after going through STAY, because there is no way that the monorail can go to CLOCKWISE state directly after COUNTERCLOCKWISE, and vice versa.

2. 學到的東西與遇到的困難

I struggled a lot with timings in this lab. I used the special condition to handle the errors occurred when my monorail jumps from one stop to another when there are no people waiting at the stop that the monorail was supposed to stop before continuing to another. There was no time left for me to debug from beginning until end since I found this mistake the day before demo day, which was the reason behind the usage of special condition. I also used the trial-and-error method to get the perfect timing for the monorail to stop at the empty stop for one clock cycle. Which is why I had different values of cnt and cnt_3 to change the value of hold, hold2, and hold3, and all the additional flags I used was to support this kind of implementation. I also used to have so many variables in this lab that in the end, I come to realize that I don't need to use that much variables to support my design so I eliminated quite a few variables that results in a complicated coding style that makes it hard to debug. In addition, I still find my code quite messy and I realize that I need to learn more and practice more to sharpen my coding style skills and make my code neat.

3. 想對老師或助教說的話

I apologize for several unused registers that I declared such as test and test2 which I previously used for debugging, also for some commented codes or parts since I had no time left to clean up my code before submitting it.