Hardware Design

Lab 6 Report

Peripheral Components: VGA, Mouse, and Dual FPGA

Team 01

112062122 謝佳晉　112062144 范升維

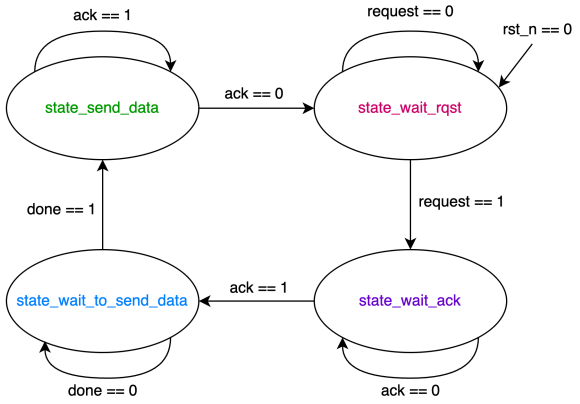
Table of Contents:

1. Dual FPGA Communication

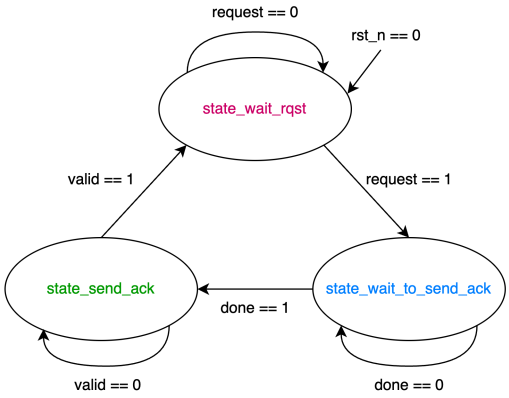
In this problem, we will build upon the template to complete the transmission protocol between two FPGAs. This includes state diagrams for both sides, transmission messages, and displaying the transmitted messages via seven-segment displays. Currently, the only incomplete portion in the template is the state diagram for the slave side.

1.1 State Diagram

The figure below are the state diagrams for both master and slave.



▲ Figure 1.1: master's state diagram



▲ Figure 1.2: slave's state diagram

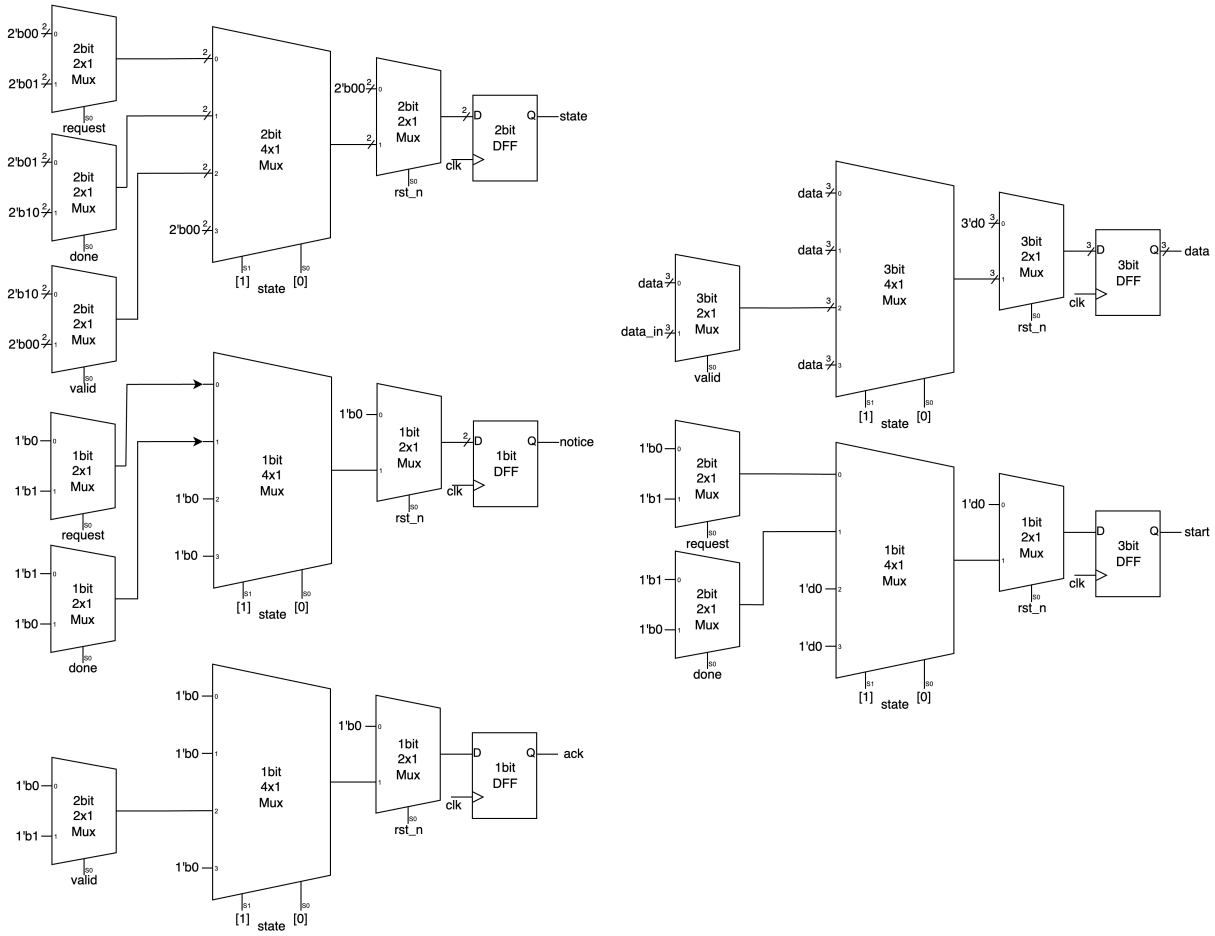
Initially, both master and slave units are in the wait_rqst state. When the master receives a request signal from the button input, it transmits a *request* signal to the slave. Upon receiving this *request*, the slave transitions to the wait_to_send_ack state and waits for one second until *done* is triggered. The slave then enters the send_ack state and continuously transmits an *ack* signal to the master, indicating its readiness to receive data.

The master, upon receiving the *ack*, transitions from wait_ack to wait_to_send_data state and waits for one second. After this delay, when the master's *done* signal is triggered, it enters the send_data state and begins transmitting data to the slave. Upon completing the data transmission, the master sends a *valid* signal to the slave. When the slave receives this *valid* signal, indicating the completion of master's transmission, it resets the *ack* signal to 0 to indicate the end of message reception and returns to the wait_rqst state.

The master, upon detecting the slave's *ack* signal reset to zero, confirms successful message reception and returns to the wait_rqst state, where it awaits the next *request* trigger.

1.2 Block Diagram

Below is the block diagram illustrating the slave state transitions. It is highly intuitive and straightforward, as it implements the communication protocol described in the previous section. With this implementation, message transmission between the two FPGA boards becomes operational.

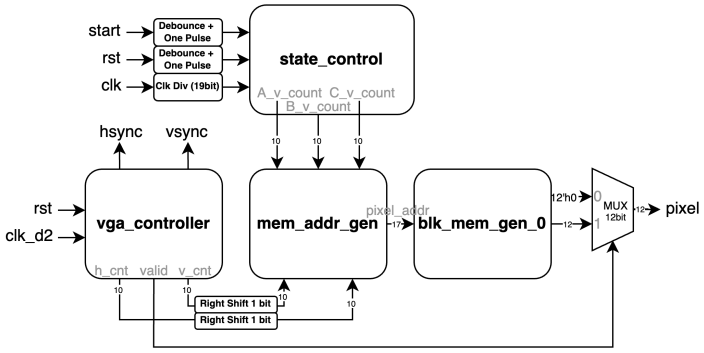


▲ Figure 1.3: slave_control

2. Slot Machine

2.1 Provided Programs and Required Tasks

In this project, the template has implemented the downward spinning functionality of the slot machine. Our tasks are to implement upward spinning and enable consecutive spins without requiring a reset. The provided template is shown in the figure below, which illustrates the key signals and relationships between circuit components:



▲ Figure 2.1: Abstract Block Diagram of Provided Template

- **state_control**: Outputs the frame indices for the slot machine's three numbers, providing mem_addr_gen with the necessary position data from the original image.
- **mem_addr_gen**: Generates image addresses using data from state_control and pixel rendering information from vga_controller.
- **blk_mem_gen_0**: A Vivado-configured module that retrieves stored pixel data based on provided addresses.
- **vga_controller**: Utilizes *clk_d2* (25MHz clock) to manage VGA interface and component communication signals.

Task 1: The system initially has a *start* signal as the trigger button for downward spinning. We need to add an *up* signal and rename *start* to *down*.

Task 2: Enable the slot machine to operate continuously with subsequent button presses without requiring a reset between spins.

2.2 Task 1: Enabling Upward Spinning

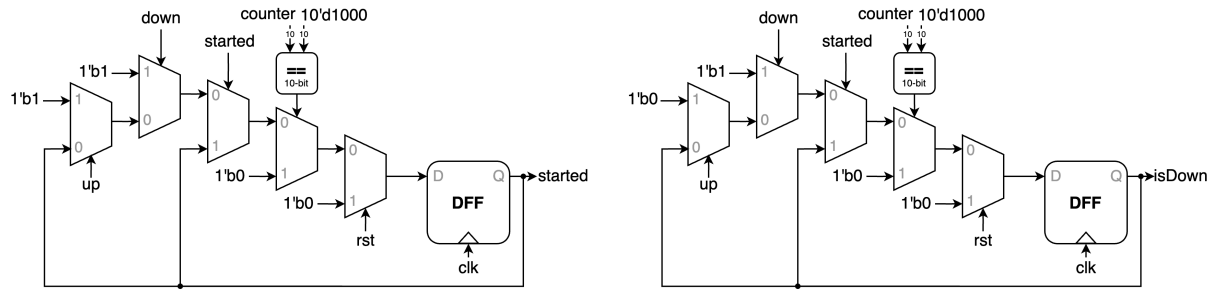
To do this, we expand the input by adding an *up* signal, which passes through Debounce and One Pulse before entering state_control. Our main modifications focus on its internal operations. While the three numbers' FSMs differ to control the sequence and speed, the timing relationships for upward and downward spinning remain unchanged. Therefore, we only need to modify the trigger conditions rather than the FSM itself.

Originally, a round begins from the STOP state when *counter==10'd0* and *down==1'b1*. Now, we expand this condition to start when *counter==10'd0* and *(down==1'b1 || up==1'b1)*, meaning either button press will initiate the spinning motion.

The state diagram operates by incrementing the counter from *10'd0* when the start button is pressed, continuing until it reaches *10'd1000*, which marks the completion of one round. The spinning speeds of the three numbers vary according to their respective state diagrams.

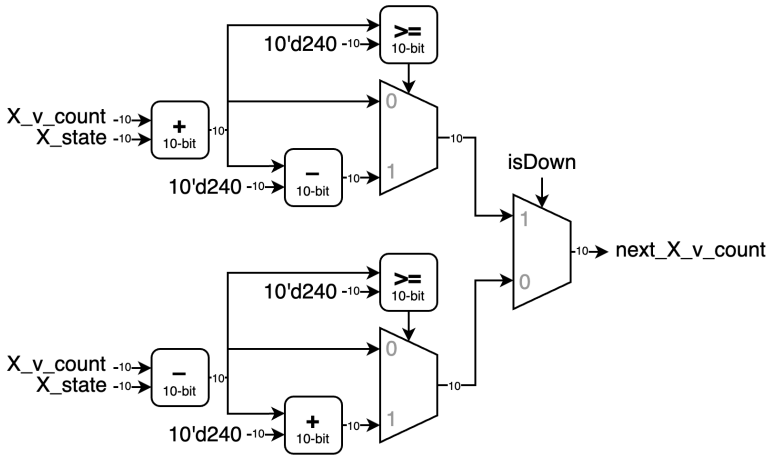
Our modification focuses on the *next_counter* conditions. We add the *up* signal alongside *down*. When both signals are non-zero and the counter equals zero, *next_counter* remains unchanged; otherwise, it increments by one. (This will require further adjustments in Task 2, as shown in the complete diagram in Figure 2.4)

To distinguish between upward and downward spinning directions, we introduced two additional signals: *isDown* and *started*. The *started* signal indicates whether the machine is in operation, while *isDown* determines the spinning direction (upward or downward). The circuit diagram is shown in Figure 2.2.



▲ Figure 2.2: *started* and *isDown* signals

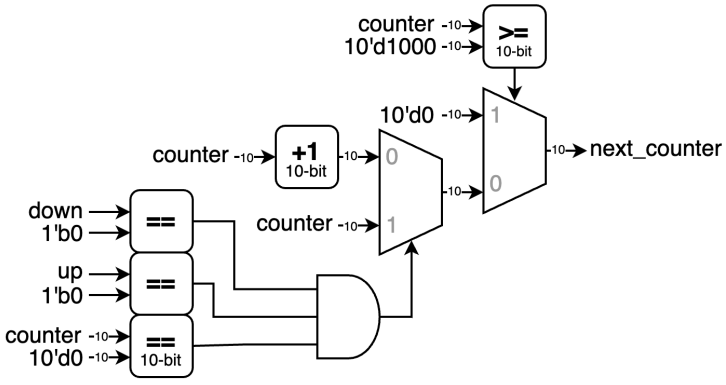
With these two signals, we only need to modify *next_X_v_count*(where X represents the respective numbers), which controls the position data sent to mem_addr_gen. Using a MUX to split the original downward spinning, the position control changes from addition to subtraction, thus completing the implementation, as shown in Figure 2.3.



▲ Figure 2.3: *next_X_v_count* signal

2.3 Task 2: Continuous Usage Without Reset Required

Following Task 1's handling of the *next_counter* signal, continuous operation can be achieved by simply adding a condition at the beginning to reset the counter to *10'd0* when counter equals *10'd1000*. The complete circuit diagram for *next_counter* is shown in Figure 2.4 below.



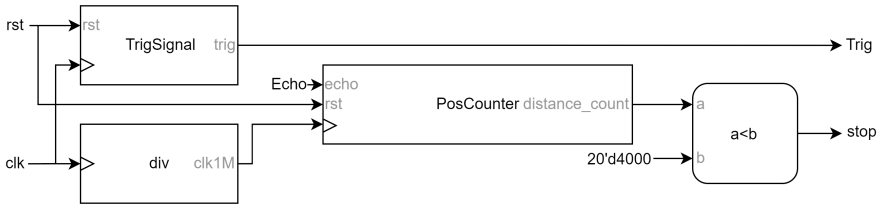
▲ Figure 2.4: *next_counter* signal

3. The Car

We are asked to modify the sample code to make the car complete the tracks. The sample code are divided into four parts, which are sonic, motor, tracker sensor, and top.

3.1 Sonic Part

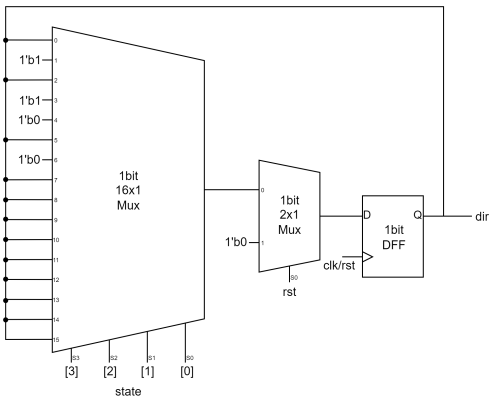
In the sonic part, we modified the sonic_top module. In order to let the car stop when the distance between it and obstacle in front of it is less than 40cm, we set *stop* to (dis < 20'd4000). The overall design of sonic part is shown in Figure 3.1.



▲ Figure 3.1: sonic part

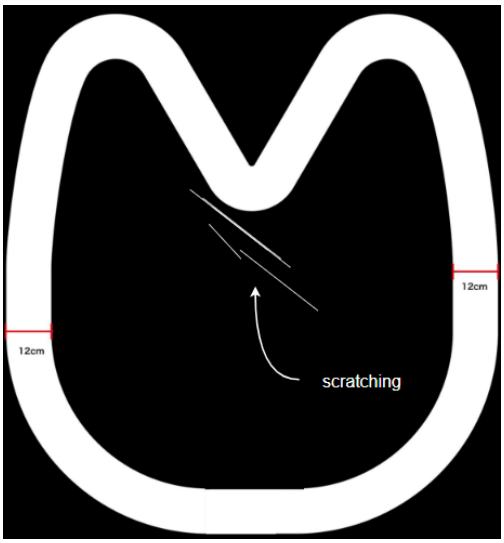
3.2 Tracker Sensor Part

In the tracker sensor part, we modified the tracker_sensor module. We design our *state* to be 4bit to include in the *STAY_L* and *STAY_R* states, which we use when the sensors are all out of track. In addition to *state*, we also add signal *dir* to record which direction it should turn before the car goes out of track. It can helps us decide on whether *STAY_L* and *STAY_R* the car should switch to.



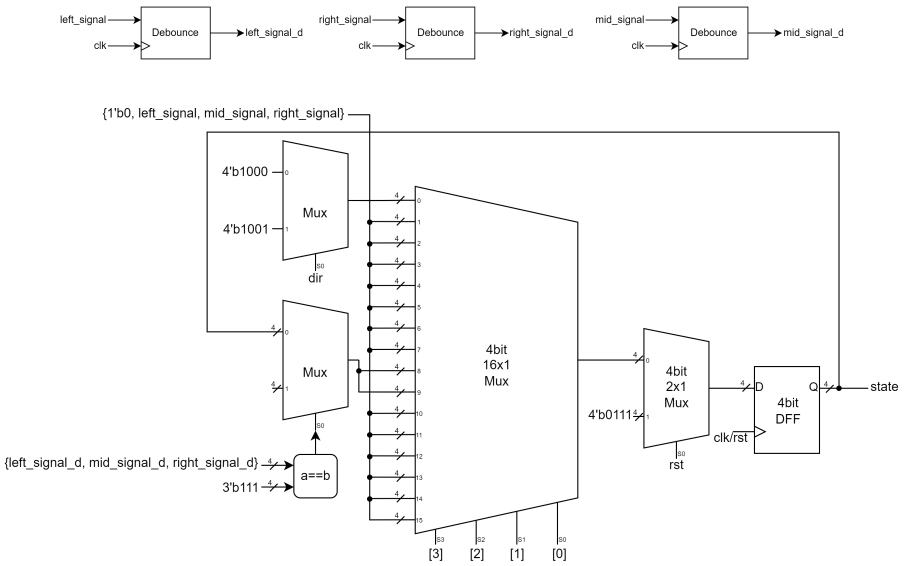
▲ Figure 3.2: diagram of *dir*

At first, our state transition is simply based on sensors' output signals. However, we discover that the car will randomly turn its direction before going back to track from the black area. It's because of the scratching on black areas.



▲ Figure 3.3: scratching the black area

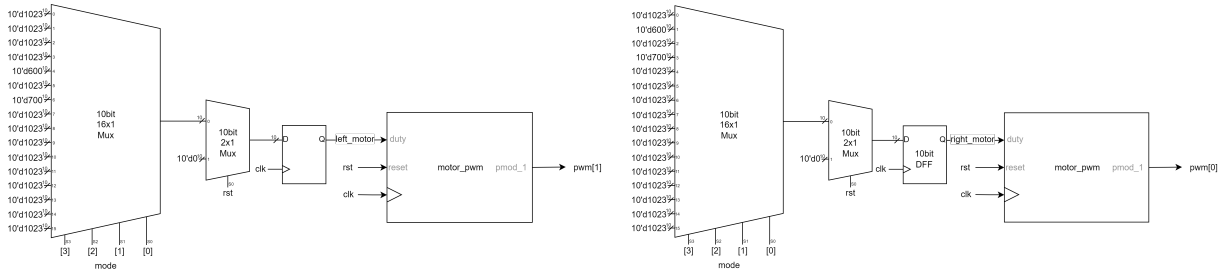
When the sensors passes the scratching, they will output *1'b1*. The not-wanted outputs are like glitches. Therefore, after a long term of testing, we add debounce on sensors' outputs and let the car keep its state on *STAY_L* and *STAY_R* until the sensors are all on track.



▲ Figure 3.4: sensor signals debounce and *state* design

3.3 Motor Part

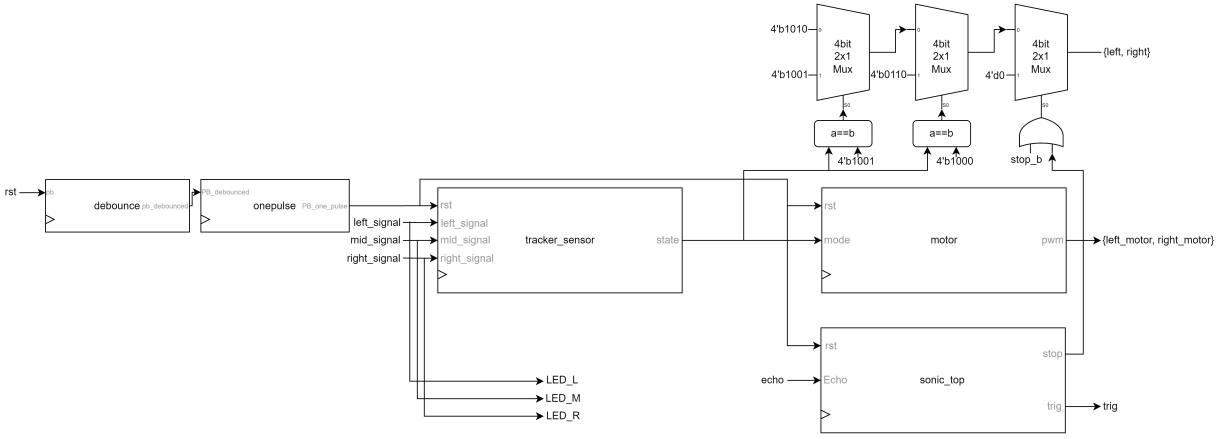
In the motor part, we modified the motor module. We design different pwm combinations according to *mode*(state). For states of one sensor out, we set one of the wheel's pwm to *10'd1023*(100%) and the other wheel's to *10'd700*(68.4%). For states of two sensors out, we set one of the wheel's pwm to *10'd1023*(100%) and the other wheel's to *10'd600*(58.7%). For states of all sensors out, we set both of the wheels' pwms to *10'd1023*(100%) and invert one of the wheel's direction. In this way, the car can turn back to track faster.



▲ Figure 3.5: motor_part

3.4 Top Part

In the top part, we combine the above three parts, add some signals(*LED_L*, *LED_M*, *LED_R*, *stop_b*) for debugging, and change left motor's and right motor's direction according to the states and stop signals. When *stop* and *stop_b* are positive, {*left*, *right*} is changed to *4'd0*. When the car is in state *STAY_L* and *STAY_R*, {*left*, *right*} is changed to *4'b0110* and *4'b1001*. Under other conditions, {*left*, *right*} is set to *4'b1010*. The overall design is shown in Figure 3.6.



▲ Figure 3.6: top part

What have we learned from Lab 6?

Throughout the semester, we underwent six intensive laboratory sessions that progressively built our understanding. We began with gate-level construction of basic modules, advanced to Data Flow Modeling and Behavioral Modeling in the middle stages, and learned about sequential circuits. Eventually, we tackled complex interfaces including keyboards, audio systems, VGA, mouse inputs, dual FPGA transmission, and even successfully developed a track-following vehicle. This journey demonstrated the remarkable flexibility and versatility of FPGA technology.

However, the final stages presented unprecedented challenges, particularly in hardware connectivity. The dual FPGA setup and the vehicle project, in particular, required extensive wiring configurations. Issues with the .xdc file configuration or incorrect pin assignments led to significant problems. We initially spent considerable time debugging the Verilog code, only to eventually discover that the issues stemmed from wiring connections.

This experience emphasized a crucial lesson: it is essential to thoroughly identify and verify all required inputs and outputs before beginning implementation. Only when this foundational work is properly completed can the program logic be correctly demonstrated. This understanding highlighted the critical relationship between hardware configuration and software functionality.

Contributions

	Dual FPGA	Slot Machine	The Car	What we learned
wrote Verilog modules				
drew diagrams				
wrote reports				

Both
謝佳晉
范升維