Lab 2: Design Practice for Amy Crypto Machine

You don't need to submit this design practice.

Objective

1. Getting familiar with sequential Verilog behavior modeling.
2. Practicing finite-state machines (FSMs) in Verilog.
3. Practicing the memory data structure in Verilog
4. Practicing the for-loop in Verilog

Design Description

This practice includes several Verilog techniques that will be used in Lab 2.

1. Storing multiple input data to the storage/memory. In this practice, we use the for-loop to initialize the memory data structure and manipulate the content.
2. Dealing with multiple test patterns. In the test stimulus, the next input data may come within a few cycles after finishing previous test pattern.
3. Using Verilog concatenation operators.

We provide a sample design, lab2_practice, for your reference. Fig. 1 is its block diagram. The test stimulus will generate a random number of the **in_data** ranging from 10 to 63 with the **in_valid** being high.

Figure 1: lab2_practice block diagram

In addition, we also provide a sample finite state machine (FSM) in Fig. 2 to process the input data.

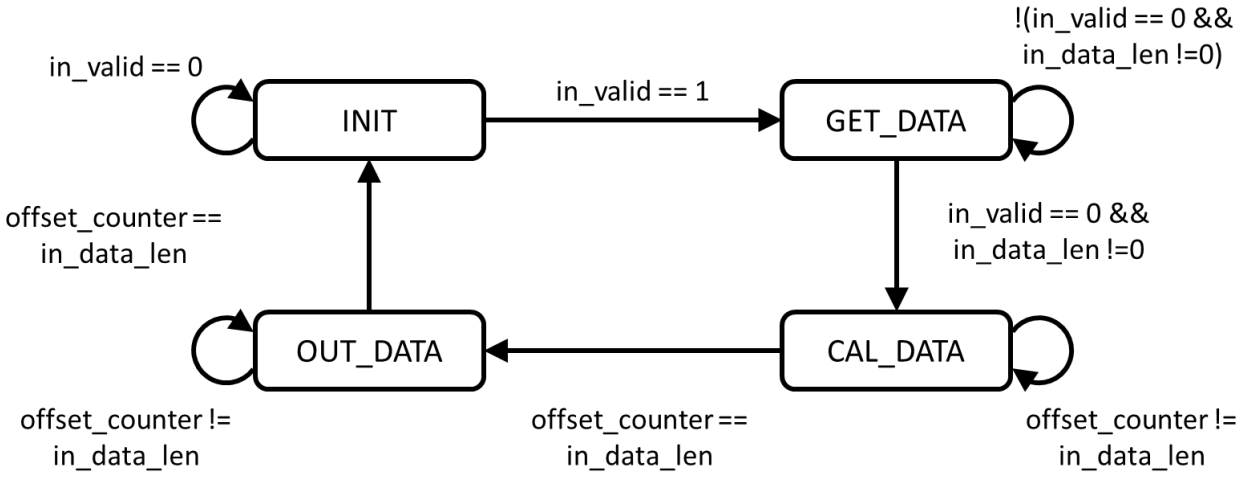

Figure 2: finite state machine of lab2_practice.

- **INIT**:
  After reset, the state will be the **INIT** state waiting for the **in_valid** to be high. Whenever the **in_valid** becomes high, the state machine will enter the **GET_DATA** state to process the input data.

- **GET_DATA:**
  In the **GET_DATA** state, if the **in_valid** is pulled low, we get all input data. The **in_data_len** signal is to ensure that we have at least one data.

- **CAL_DATA:**
  After the input data are all received, the next state is **CAL_DATA**. First, the input data is increased by one. Second, if the first bit and last bit are **2'b00**, the output data is **8'd0**; if the first bit and last bit are **2'b11**, the output data is **8'b11111111**. Here, we use (**offset_counter == in_data_len)** to indicate that all input data are calculated. The **offset_counter** is a counter which will increase every cycle.

- **OUT_DATA:**
  Finally, the state will transit to the **OUT_DATA** state. In this state, we output the data to the test stimulus (testbench) for verification. For detailed implementation, you can check out **lab2_practice.v**

Action Items

a. Calculating data
You should calculate the input data based on the following equation.

$$c_i = p_i + 1,$$

where the p is the in_data; the c is the calculated data, and the i is the index of current data.

Example:

Assume that we get the following input data {1, 2, 3, 4, 5} as the index of {1, 2, 3, 4, 5}, which is the same as input sequence. First, we increase the input data as follows.

Index	in_data	Increased Data
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6

Second, the increased data will be decoded based on its first bit and last bit.

First Bit and Last Bit	Output Data
2'b00	8'b0
2'b11	8'b11111111
Others	No Change

Examples:

- If the increased data is 5 (i.e., $\{00000101\}_2$), the final output data will be $\{00000101\}_2$.
- If the increased data is 133 (i.e., $\{10000101\}_2$), the final output data will be $\{11111111\}_2$.

b. IO list:

Input Signals	Bit Width	Definition
clk	1	Clock signal
rst	1	Reset signal
in_data	8	Input data
in_valid	1	Input data valid: 1 (high) for valid

Output Signals	Bit Width	Definition
out_data	8	Output data
out_valid	1	Output data valid: 1 (high) for valid

c. Specification

1、Reset
- The reset signal (rst) is the active-high synchronous reset.
- The design will be reset only at the beginning of the simulation. All output signals should be reset to 0 after the reset!

2、Input data
- The in_data will be valid only when the in_valid is high.

3、Output data
- The out_valid should be high when the out_data is ready.

4、Simulation time
- Please follow Lab 1 tutorial to change the runtime to 100,000 ns.

5、Simulation
- Please refer to Appendix A for the simulation guide.

e. PATTERN testing guide

1. Reset
   - The PATTERN module will check whether your output signals are set to 0 after reset.
2. Input data
   - Each test case will have a random number of the in_data ranging from 10~63.
   - After finishing one test case, the next in_data will come with a 2~10 cycle delay.
3. Output data
   - The PATTERN module will check if you have the same number of the out_data as the in_data you got.
   - The PATTERN module will check the correctness of the out_data.
   - The out_valid and in_valid cannot be high at the same time.

Design Explanation

a. Store multiple input data to memory.

To store multiple data into memory, we first declare a memory data structure as Fig. 3.

```
// output_data_save
reg [7:0] output_data_save[63:0], next_output_data_save[63:0];
```

Figure 3: Memory declaration.

Since there are 64 8-bit data, it is better to use the for-loop to initialize all of them (refer to Fig. 4). Note that the for-loop statement for combinational circuits in Verilog represents an unfolded parallel hardware. The loop boundary (e.g., MAX_INPUT_MAX in Fig. 4) must be a constant. So, all **output_data_save** will be initialized **at the same time**. However, the for-loop in C/C++ behaves sequentially. That is, the for-loop will initialize the **output_data_save[0]** first. Then, it checks if **i** is exceed the boundary. If not, it goes to the next **i,** and so on**.**

```
always @(posedge clk ) begin
    if (rst) begin
        for (i=0; i<=MAX_INPUT_LEN; i=i+1) begin
            output_data_save[i] <= 0;
        end
    end
    else begin
        for (i=0; i<=MAX_INPUT_LEN; i=i+1) begin
            output_data_save[i] <= next_output_data_save[i];
        end
    end
end
```

Figure 4: Memory initialization using for-loop.

After the initialization, we can start storing data to the memory as in Fig. 5. Firstly, we assign the **next_output_data_save** the same data as **out_data_save**, which put in a default value to prevent from synthesizing latches. Since the **in_data** is valid only when **in_valid** is high, we use an if statement to check the current value of **in_valid**, utilize the **offset_counter** to identify the index of the incoming data. After we get all data, we can move on to the next state.

```
GET_DATA : begin
    for (i=0; i<=MAX_INPUT_LEN; i=i+1) begin
        next_output_data_save[i] = output_data_save[i];
    end
    if (in_valid) begin
        next_output_data_save[offset_counter] = in_data;
    end
end
```

Figure 4: Memory writing using for-loop.

b. Dealing with multiple test patterns. In the test stimulus, the next input data may come within a few cycles after finishing a test pattern.

After finishing one test pattern, the next pattern may come within a few cycles. We should initialize the state machine after one test pattern is finished. Fig. 5 shows the sample code segment. After the **OUTPUT_DATA** state, the next state will be **INIT**, and we should initialize all variables. The state can transmit to the **GET_DATA** state only when the **in_valid** is high (i.e., the next input data comes).

```
always @(*) begin
    case(state)
        INIT : begin
            if (in_valid) begin
                next_state = GET_DATA;
            end
            else begin
                next_state = INIT;
            end
        end
        GET_DATA : begin
            if (!in_valid && in_data_len != 0) begin
                next_state = CALCULATE_DATA;
            end
            else begin
                next_state = GET_DATA;
            end
        end
        CALCULATE_DATA : begin
            if (offset_counter == in_data_len) begin
                next_state = OUTPUT_DATA;
            end
            else begin
                next_state = CALCULATE_DATA;
            end
        end
        OUTPUT_DATA : begin
            if (offset_counter == in_data_len) begin
                next_state = INIT;
            end
            else begin
                next_state = OUTPUT_DATA;
            end
        end
        default : begin
            next_state = INIT;
        end
    endcase
end
```

Figure 5: State machine example.

c. Using the for-loop to manipulate the memory data structure.

We showed how to initialize the memory using the for-loop. In Fig. 6, we demonstrate how to manipulate the memory data structure. First, we assign the **next_output_data_save** the same data as **out_data_save**, and then do the calculation.

```
CALCULATE_DATA : begin
    for (i=0; i<=MAX_INPUT_LEN; i=i+1) begin
        next_output_data_save[i] = output_data_save[i];
    end

    if (offset_counter != in_data_len) begin

        // + 1
        next_output_data_save[offset_counter] = output_data_save[offset_counter] + 1;

        // select
        case({next_output_data_save[offset_counter][7], next_output_data_save[offset_counter][0]})
            2'b00 : begin
                next_output_data_save[offset_counter] = 8'd0;
            end
            2'b11 : begin
                next_output_data_save[offset_counter] = 8'b11111111;
            end
            default : begin
                next_output_data_save[offset_counter] = next_output_data_save[offset_counter];
            end
        endcase

    end

end
```

Figure 6: calculation example.

d. Verilog concatenation operators.

In Verilog, we can use **{}** to concatenate two or more vectors as one. As shown in Fig. 6, we use **{next_output_data_save[offset_counter][7], next_output_data_save[offset_counter][0]}** to concatenate the first bit and last bit of **next_output_data_save** and use it for the case statement to decode the data.