Assignment - 01

EE1501: Digital Circuits Lab

S. Rohith Sai EE24BTECH11061

Contents

1 Introduction

This report presents the implementation and analysis of three digital circuits:

- A 4-to-2 priority encoder with valid output signal
- A 4-bit up counter with enable and asynchronous reset functionalities
- An 8-bit even parity generator

For each circuit, the design specifications, Verilog implementation, simulation results, and observations are detailed.

2 4-to-2 Priority Encoder

2.1 Specifications

- **Inputs**: `in[3:0]` - A 4-bit input signal.
- **Outputs**:
  - `out[1:0]` - A 2-bit binary output indicating the position of the highest-priority input that is high.
  - `valid` - A 1-bit output signal that is set to 1 if any input bit is high; otherwise, it is 0.
- **Priority Order**: `in[3] > in[2] > in[1] > in[0]`

2.2 Theory

A priority encoder is a combinational circuit that converts multiple input lines into a binary code based on priority levels. When multiple inputs are active simultaneously, the output corresponds to the highest-priority active input. The valid output indicates whether any input is active.

2.3 Truth Table

in[3]	in[2]	in[1]	in[0]	out[1:0]	valid
1	X	X	X	11	1
0	1	X	X	10	1
0	0	1	X	01	1
0	0	0	1	00	1
0	0	0	0	00	0

Table 1: Truth Table for 4-to-2 Priority Encoder (`X` represents don't care)

2.4 My Approach

To implement the 4-to-2 priority encoder, I used the `casez` statement within an `always` block. This allows the use of don't-care bits (represented by `z`) in the case items. The block checks for the highest priority active bit and assigns the corresponding output value. If none of the bits are high, the default case sets the output to `00` and the valid flag to `0`.

2.5 Verilog Code for the Priority Encoder

```
module A1_1 (
    input [3:0] in,
    output reg [1:0] out,
    output reg valid);

    // Using 'reg' for both 'out' and 'valid' because we are doing
       procedural assignment (always block)
    always @(*) begin
        valid = 1'b1;
        casez(in)
            4'b1zzz: out = 2'b11;
            4'b01zz: out = 2'b10;
            4'b001z: out = 2'b01;
            4'b0001: out = 2'b00;
            default: begin
                out = 2'b00;
                valid = 1'b0;
            end
        endcase
    end

endmodule
```

2.6 Simulation Waveform

Below is the waveform captured in GTKWave.

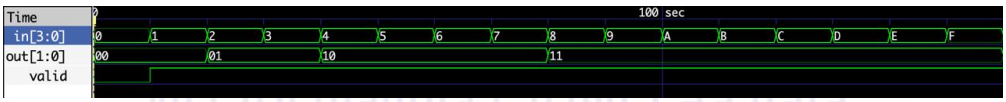

Figure 1: GTKWave waveform of the 4-to-2 priority encoder simulation.

2.7 Terminal Output

In addition to the waveform, the terminal prints the output for all input combinations during simulation. The captured screenshot of terminal output is shown below:

Time	in	out	valid
0	0000	00	0
10	0001	00	1
20	0010	01	1
30	0011	01	1
40	0100	10	1
50	0101	10	1
60	0110	10	1
70	0111	10	1
80	1000	11	1
90	1001	11	1
100	1010	11	1
110	1011	11	1
120	1100	11	1
130	1101	11	1
140	1110	11	1
150	1111	11	1

Figure 2: Terminal output showing `in`, `out`, and `valid` values over time.

2.8 Observations

- The waveform and terminal output show that for each input vector applied to `in`, the outputs `out` and `valid` behave correctly.
- When `in` is `0000`, the valid signal is low and `out` is `00`.
- For all other input combinations, `valid` is high and `out` represents the highest priority bit position.
- The `casez` statement effectively handles the priority logic, with `in[3]` having the highest priority and `in[0]` the lowest.
- The simulation confirms that the encoder works correctly for all possible input combinations.

3 4-bit Up Counter with Enable and Asynchronous Reset Functionalities

3.1 Specifications

- **Inputs**:
  - `clk` – Clock signal.
  - `reset` – Asynchronous reset signal (active high).
  - `enable` – Control signal to enable counting.
- **Outputs**:
  - `count[3:0]` – A 4-bit output representing the current count value.

3.2 Theory

A counter is a sequential circuit that goes through a predetermined sequence of states upon the application of clock pulses. The 4-bit up counter counts from 0 to 15 and then wraps around to 0. The asynchronous reset immediately sets the counter to 0 regardless of the clock state, while the enable signal controls whether the counter increments on the clock's rising edge.

3.3 My Approach

To implement the 4-bit up counter, I used a positive-edge triggered `always` block with asynchronous reset. The reset condition has the highest priority and clears the counter to `0000`. If reset is inactive and enable is high, the counter increments by 1 on each clock cycle. Non-blocking assignments (`<=`) are used to ensure proper sequential logic behavior (Initially I forgot to use non-blocking assignments, which led to an error).

3.4 Verilog Code for the Counter

```
module A1_2 (
    input clk,
    input reset,
    input enable,
    output reg [3:0] count);

    always@(posedge clk or posedge reset) begin
        if (reset)
            count <= 4'b0000;   // We use '<=' in place of '=' for "
               non-blocking assignments" i.e., when we  use "always@
               (posedge clk)" block.
        else if (enable)
```

```
            count <= count + 1;
    end
endmodule
```

3.5 Simulation Waveform

Below is the waveform captured in GTKWave during simulation.

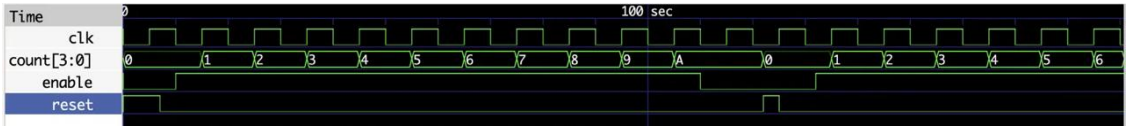

Figure 3: GTKWave waveform showing counter output with different enable and reset conditions.

3.6 Observations

- When `reset` is high, the output `count` is reset to `0000`, regardless of the clock or enable signals.
- When `enable` is low, the counter holds its value even though the clock continues to toggle.
- When `enable` is high and `reset` is low, the counter increments on each rising edge of the clock.
- Non-blocking assignments ensure that the count update occurs at the end of the clock cycle, maintaining correct sequential behavior.

4 Even parity generator for 8-bit input

4.1 Specifications

- **Inputs**:
  - `data[7:0]` – An 8-bit input vector representing the data for which parity is to be calculated.
- **Outputs**:
  - `parity` – A single-bit output representing the even parity bit.

4.2 Theory

A parity generator produces a parity bit based on the number of 1s in the input data. For even parity, the parity bit is set such that the total number of 1s (including the parity bit) is even. If the input contains an even number of 1s, the parity bit is 0; if the input contains an odd number of 1s, the parity bit is 1.

The truth table for even parity would be extensive for an 8-bit input (256 rows), but the general rule is:

- If number of 1s in data is even: parity = 0
- If number of 1s in data is odd: parity = 1

4.3 My Approach

The even parity is computed using a combinational logic block. Specifically, the Verilog reduction XOR operator ^ is used to calculate the parity of all 8 bits. The result of this operation yields the even parity as asked in the question.

4.4 Verilog Code for Even Parity Generator

```
module A1_3(
    input [7:0] data,
    output parity);

    assign parity = ^data; // XOR of all bits gives the even parity
       bit
endmodule
```

4.5 Simulation Waveform

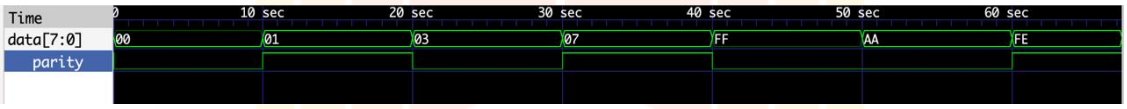

Figure 4: GTKWave output for the 8-bit even parity generator.

4.6 Terminal Output

In addition to the waveform, the terminal displays the parity output for each applied 8-bit input pattern during simulation. A screenshot of the terminal output is shown below:

Time	Data	Parity
0	00000000	0
10	00000001	1
20	00000011	0
30	00000111	1
40	11111111	0
50	10101010	0
60	11111110	1

Figure 5: Terminal output showing `data` and `parity` values at each simulation step.

4.7 Observations

- The output parity bit is `1` when the number of 1s in the input is odd, ensuring overall even parity.
- The output is `0` when the number of 1s is already even.
- The waveform confirms the correctness of the design for all tested inputs.
- The circuit is purely combinational (assign statement) and responds instantly to input changes.
- The XOR reduction operator (`^`) provides an elegant and concise implementation for parity calculation.