# Assignment - 01


EE1501: Digital Circuits Lab

**S. Rohith Sai EE24BTECH11061**

# Contents

# 1 4-to-2 Priority Encoder

## 1.1 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]`

## 1.2 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`.

## 1.3 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
```

## 1.4 Simulation Waveform

Below is the waveform captured in GTKWave.


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

## 1.5 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.

### 1.6 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.

## 2 4-bit Up Counter with Enable and Asynchronous Reset Functionalities

### 2.1 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.

### 2.2 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).

### 2.3 Verilog Code for the Counter

```
module A1_2 (
    input clk, // Clock signal
    input reset, // Asynchronous reset signal (active high {1 is
       high})
    input enable, // Control signal to enable counting
    output reg [3:0] count);

    always@(posedge clk) begin
        if (reset)              // Counter is cleared when reset is
           high
```

```
            count <= 4'b0000;   // We use '<=' in place of '=' for "
               non-blocking assignments" i.e., when we  use "always@
               (posedge clk)" block.
        else if (enable)        // Counting takes place only when
           enable is high
            count <= count + 1;
    end
endmodule
```

### 2.4 Simulation Waveform

Below is the waveform captured in GTKWave during simulation.

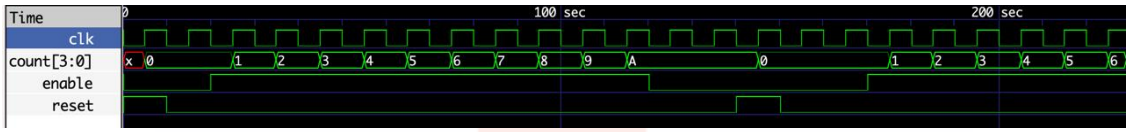


Figure 3: GTKWave waveform showing counter output with different enable and reset conditions.

### 2.5 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.
- The waveform and terminal output match the expected behavior of a 4-bit up counter with asynchronous reset and enable.

## 3 Even parity generator for 8-bit input

### 3.1 Specifications

- **Inputs**:
  - `data[7:0]` – A 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.

## 3.2 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 odd parity. To obtain the even parity, the result is complemented using the bitwise NOT operator `~`.

## 3.3 Verilog Code for Even Parity Generator

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

    assign parity = ~^data; // XOR of all bits give the odd parity
       bit, hence NOT of the output gives the even parity bit.
endmodule
```

## 3.4 Simulation Waveform


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

## 3.5 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 | 1 |
| 10 | 00000001 | 0 |
| 20 | 00000011 | 1 |
| 30 | 00000111 | 0 |
| 40 | 11111111 | 1 |
| 50 | 10101010 | 1 |
| 60 | 11111110 | 0 |

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

## 3.6 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.