EE332 Lab2: Divide-Conquer Implementation of 16-4 Priority Encoder

1st Qiu Kunyuan *EEE. Southern University of Science and Technology*
Shenzhen, PRC
11913019@mail.sustech.edu.cn

Abstract

This report focuses on the difference between cascaded structure and tree structure in the Verilog implementation of 16-4 priority encoder. The timing optimization of combinational logic circuits with a large number of inputs is carried out by the divide and conquer method, and the pipelining method is also employed. Therefore, the routing results are obtained with satisfactory delays and no race-hazard jitters.

Code for this experiment can be found at https://github.com/KagaJiankui/EE332-2024S/tree/master/lab3/lab3.srcs.

Index Terms

FPGA, programmable logic, priority coder, divide-conquer method, pipelining, -hazard condition, jitter, delay

I. Introduction

There are typically two main implementations of priority encoders in Verilog designs: cascaded architecture and tree architecture.

- **Cascaded architecture**: A cascaded architecture is a serial structure that groups input signals and encodes them within each group, and then these results are cascaded. Such a structure enjoys the simplicity of implementation, but suffers from the disadvantage that the delay increases with the increase in the number of input signals.
- **Tree architecture**: A tree architecture is a parallel structure that encodes all the input signals simultaneously. Benefit of this structure is its speed as all the operations are performed in parallel. However, the demerit of this structure is the requirement of more hardware resources.

A. Cascaded Architecture

The code of cascaded priority encoder is `if-else` statements organized along with the order of MSB to LSB. Thus, the cascaded priority encoderan can be easily modeled by the following Verilog code with parameter `digit`:

```verilog
module encoder
  #(
    parameter digit=8
    parameter digit_output=3
  )
  (
    input  [digit-1:0] x
    input  en,
    output reg [digit_output-1:0]y
  );
  integer i;
  always @(x or en) begin
    if (en) begin
      y = 0;
      for( i = 0; i <= digit - 1; i = i+1)
          if(x[i] == 1)  y = i[digit-1:0];
    end
    else  y = 0;
  end
endmodule
```

For 8-3 encoder, the module is instantialized with parameter `.digit(8), .digit_out(3)` that generates the following if-else statements,

```verilog
module encoder (
  input [7:0] x,
  input en,
  output reg [2:0] y,
  output reg v
);
  always @ (x or en)
    if (x>0 & en==1) begin
      if (x[7]) y = 3'b111;
      else if (x[6]) y = 3'b110;
      else if (x[5]) y = 3'b101;
      else if (x[4]) y = 3'b100;
      else if (x[3]) y = 3'b011;
      else if (x[2]) y = 3'b010;
      else if (x[1]) y = 3'b001;
      else if (x[0]) y = 3'b000;
      else y = 3'b000;
    end
    else begin
      y = 0;
      v = 0;
  end
endmodule
```

and the schematic (1) of the elaboration result is basically directly translated from the HDL description, which are cascaded MUXs.

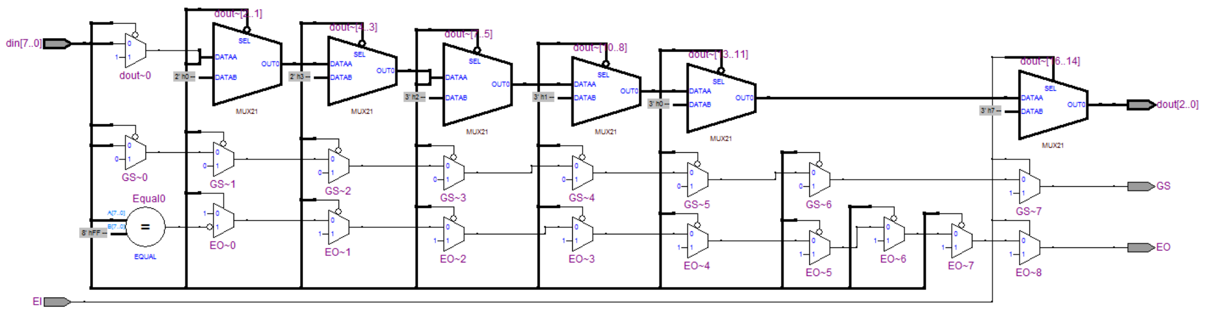

Fig. 1: Synthesis result of parameterized 8-3 priority encoder

For 4-2 encoder, the module is instantialized with parameter `.digit(4), .digit_out(2)`,

```verilog
`timescale 1us / 10ns
module encoder4x2_cas (
    input wire [3:0] in,
    output reg [1:0] out,
    output reg v
);

  always @(in) begin
    v = in[3] | in[2] | in[1] | in[0];
    if (in[3]) begin
      out = 2'b11;
    end else if (in[2]) begin
      out = 2'b10;
    end else if (in[1]) begin
      out = 2'b01;
    end else if (in[0]) begin
      out = 2'b00;
    end else begin
      out = 2'b00;
    end
  end

endmodule
```

which is elaborated into similiar schematic (2):

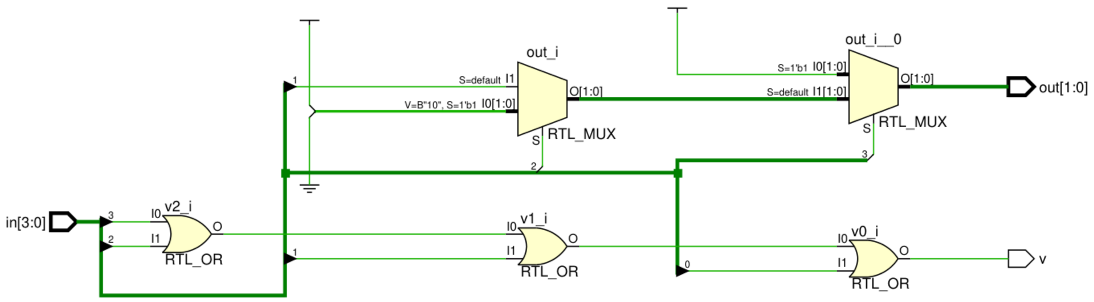

Fig. 2: Synthesis result of parameterized 4-2 priority encoder