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 encoder 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)`(note that the width of input and output bus is designated separately since Verilog, as an HDL, lacks exponential operation even in parameterization) 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.

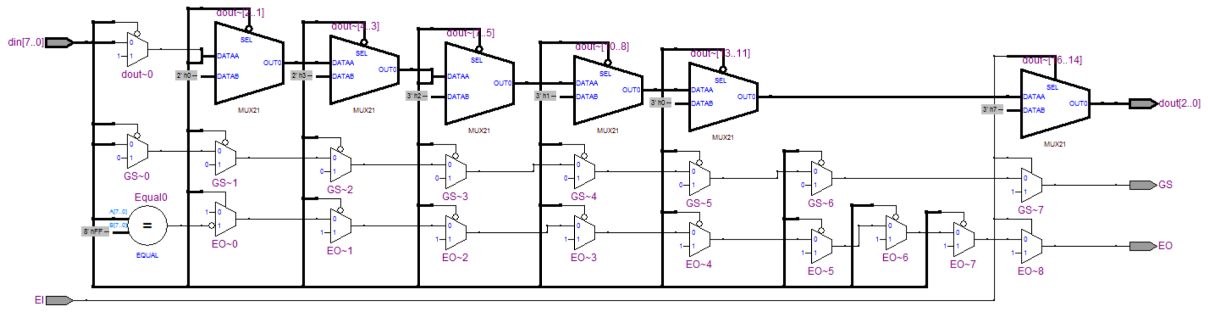


Fig. 1: Synthesis result of parameterized 8-3 priority encoder

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):

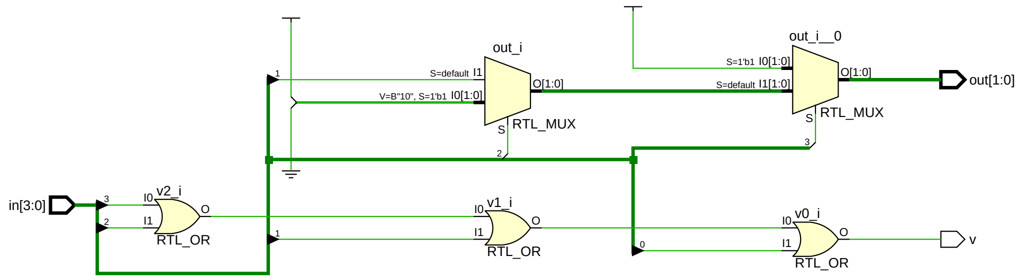
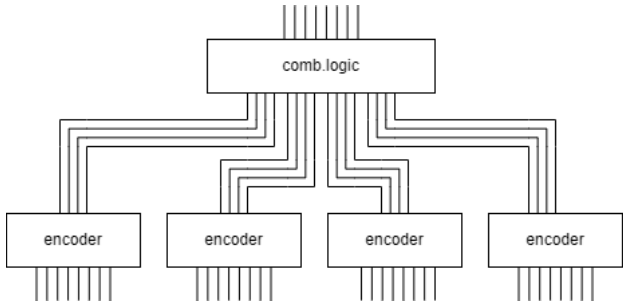


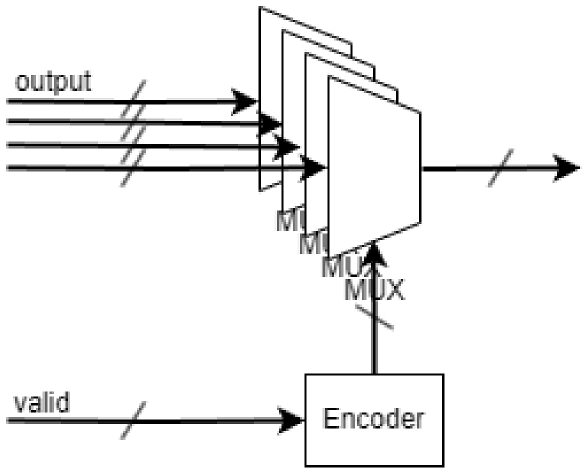
Fig. 2: Synthesis result of parameterized 4-2 priority encoder

B. *Tree Architecture*

The encoder organized in tree architecture is a reverse binary tree or quadruple tree, as depicted in left of Fig.(3). The leaf nodes at the bottom level are priority encoders with low input width, while the nodes at higher level are usually MUXs and gates that combines the output from bottom level leaf nodes with their priority.



(a) Abstraction of Architecture



(b) Combinational Logic Selector

Fig. 3: Architecture Depiction

The internal implementation of the combinational logic block is shown in right of Fig.(3). an encoder layer and the adjacent upper combinational logic node, the combinational logic node consists of a set of multiplexers corresponding to the outputs of each bit from each encoder, as well as an encoder generating a line-select signal, with an input width equal to the number of encoders.

In the designs with building block encoder similiar to 74HC148, an encoder IC with *validity* output, the output is therefore composed by the output of the building blocks:

$$\begin{aligned} \mathbf{valid}[i] &= \text{OR}(\mathbf{input}[i]) && (1) \\ \mathbf{Q} &= i * W_{bb} + \mathbf{output}[\mathbf{valid}] && (2) \end{aligned}$$

where the bolded signal name represents bus, the block bracket represents indexing from bus, the variable i denotes the valid encoder (the location of the highest input bit), and the variable W_{bb} denotes the input width of the building block encoder.

For a priority encoder with input bitwidth W_{input}, output bitwidth W_{output}, it is obvious that the output bitwidth is the bottom 2 logarithm of the input bitwidth. To employ the tree architecture, the input is splitted the input bitwidth into $S = W_{input}/W_{bb}$ slices.

$$\begin{aligned} W_{output} &= \log_2 W_{input} && (3)\\ \Rightarrow W_{output} - \log_2 W_{bb} &= \log_2\left(\frac{W_{input}}{W_{bb}}\right) && (4)\\ &\Leftrightarrow \begin{cases}\log_2 W_{bb} = W_{output} - \log_2 S,\\ \log_2 S = W_{slice}\end{cases} && (5)\end{aligned}$$

Therefore, by converting the multiplication in Eq.(2) into bitwise operations, with respect to the power-2 integrity of S and W, the derivation is clear, that the lower W_{bb} bits of the output bit is the encoded output of each slice, and the higher W_{slice} bits is the serial number of the building block encoder with valid state.

For a 16-4 priority encoder in tree architecture, with the building block selected as 4-2 priority encoder with validity indication port, all the parameters can be evaluated from Eq.(5)

$$W_{output} = \log_2 16 = 4, S = 16/4 = 4, W_{slice} = 2, \log_2 W_{bb} = W_{output} - W_{slice} = 2 \quad (6)$$

Then the verilog modeling is performed intuitively:

```
`timescale 1us / 10ns
module encoder16x4 (
    input wire clk,
    input [15:0] in,
    output reg [3:0] out,
    output reg v
);
  wire [1:0] enc1, enc2, enc3, enc4;
  wire v1, v2, v3, v4;

  encoder4x2 e1 (
      .in (in[3:0]),
      .out(enc1),
      .v  (v1)
  );
  encoder4x2 e2 (
      .in (in[7:4]),
      .out(enc2),
      .v  (v2)
  );
  encoder4x2 e3 (
      .in (in[11:8]),
      .out(enc3),
      .v  (v3)
  );
  encoder4x2 e4 (
      .in (in[15:12]),
      .out(enc4),
      .v  (v4)
  );

  reg [3:0] result0, result1, v_sel;

  always @(posedge clk) begin
    v_sel   <= {v4, v3, v2, v1};
    result1 <= {enc1[1], enc2[1], enc3[1], enc4[1]};
    result0 <= {enc1[0], enc2[0], enc3[0], enc4[0]};
  end

  wire [1:0] sel_output;
  wire v_from_input;

  encoder4x2 e_select (
      .in (v_sel),
      .out(sel_output),
      .v  (v_from_input)
  );

  always @(posedge clk) begin
    out[3:2] <= sel_output;
    v <= v_from_input;
    case (sel_output)
      2'b11:   out[1:0] <= {result1[0], result0[0]};
      2'b10:   out[1:0] <= {result1[1], result0[1]};
      2'b01:   out[1:0] <= {result1[2], result0[2]};
      2'b00:   out[1:0] <= {result1[3], result0[3]};
      default: out[1:0] <= 2'b00;
    endcase
  end
endmodule
```

which results in the Fig.(4) after elaboration, and Fig.(5) after synthesis:

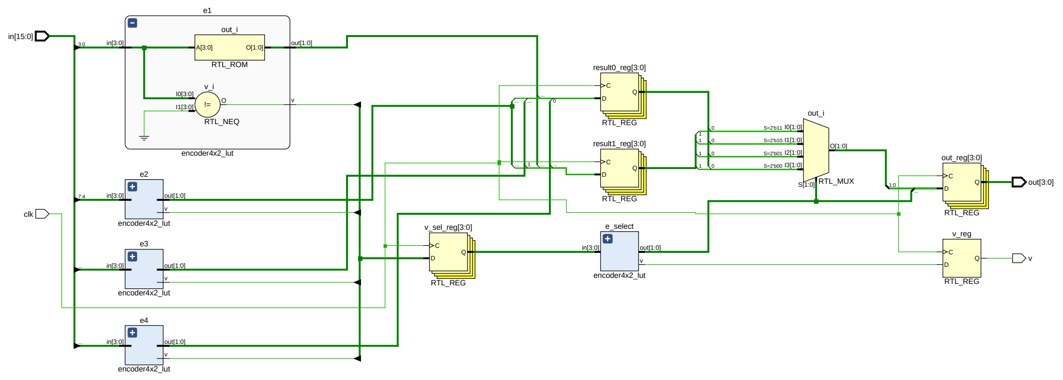


Fig. 4: Elaborated schematic of 16-4 encoder

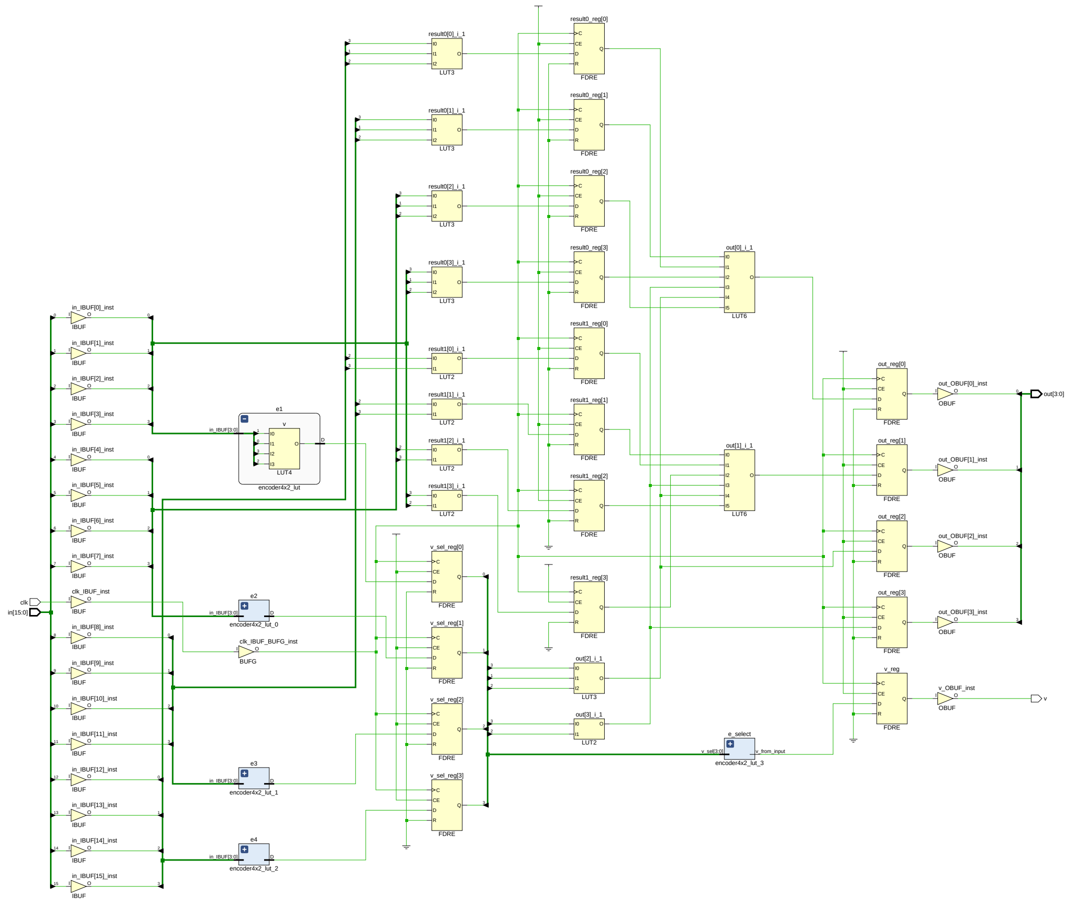


Fig. 5: Synthesized schematic of 16-4 encoder

II. Time Complexity Discussion and Optimization

The tree-structured prioritized encoder applies the idea of division and conquering method, which breaks down the big problem into small problems, and then combines the solutions of the small problems into the solution to the big problem. Each layer of the encoder works in parallel, effectively reducing the output delay.

A. Elaboration Stage Analysis

With the derivation in the section above, it is possible to portrait a model (6) of the internal data flow in the tree architecture, for the evaluation of the total time complexity (device gate delay):

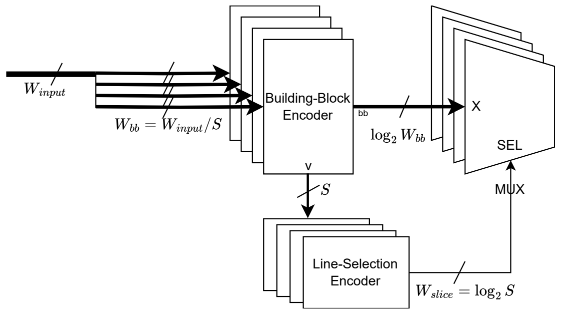
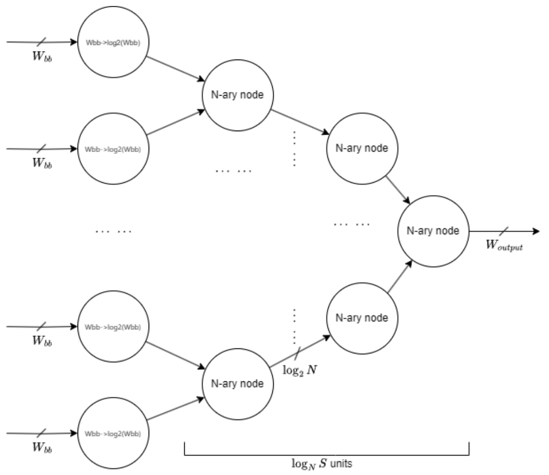


Fig. 6: Data flow and time complexity of the internal

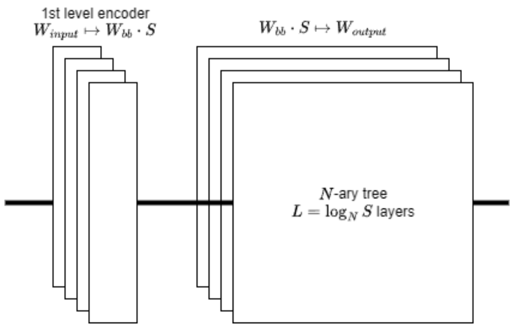
It's obvious that the time complexity of the cascaded architecture priority encoder is $\mathcal{O}(W_{in})$. If the division is only made once, which is equivalent to building block encoder and line selection generator are confined to cascaded architecture, the total delay is therefore obtained:

$$\begin{aligned} D &= \mathcal{O}(W_{bb}) + \mathcal{O}(S) = O\left(\frac{W_{input}}{S} + S\right) && (7) \\ \Rightarrow S &= \arg\min_{S}\left(\frac{W_{input}}{S} + S\right) && (8) \\ &= W_{input}^{\frac{1}{2}}, && (9) \\ D_{\min} &= 2S = 2W_{input}^{\frac{1}{2}} && (10) \end{aligned}$$

For divide-conquer implementation with large width input bus, the cascaded architecture delay S itself is unsatisfying, suggesting that the N-branch tree architecture (7) should be applied.



(a) N-branch tree architecture illustration


1st level encoder
$W_{input} \mapsto W_{bb} \cdot S$
$W_{bb} \cdot S \mapsto W_{output}$
N-ary tree
$L = \log_N S$ layers

(b) N-branch tree computational block diagram

Fig. 7: N-branch tree architecture

With the illustration of Fig.(7 (b)) and respect to the power-2 integrity of W, S, N, take the notion at the output side $W = 2^w, S = 2^s, N = 2^n$ and then obtain the following time complexity objective:

$$\begin{aligned} TC(s,n) &= \mathcal{O}\left(\frac{W}{S} + L \cdot N\right) && (11)\\ &= \mathcal{O}\left(2^{w-s} + 2^{n}\frac{s}{n}\right) && (12)\end{aligned}$$

Since there is no implied constraints and both n and s are positive integers, the extremum can be directly derived by Hessian matrix:

$$\begin{aligned} \arg\min_{(s,n)} TC(s,n) &= \arg_{(s,n)}\left(\nabla\left(\frac{W}{S} + L \cdot N\right) = 0\right) && (13)\\ \Rightarrow (s,n) &= (w - \log_2 e, \log_2 e) && (14)\end{aligned}$$

and the extremum point satisfies the multiple variable minima condition

$$\det H|_{(s,n)} = \nabla(TC(i,j))\nabla^{T}(TC(i,j))|_{(s,n)} > 0 \tag{15}$$

Applying the integrity constraints to the solution in real domain $(s,n) = (w - \log_2 e, \log_2 e)$, further discussion reveals

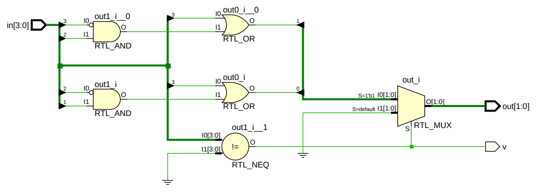
- For variable n, rounding to $n = 1$ or $n = 2$ results in the same value $2^n/n = 2$;
- For variable s, take $2^n/n = 2$ above results in $s = w - \lfloor \log_2(\log_2(e)) + 1 \rfloor = w - 1$

Therefore in conclusion,

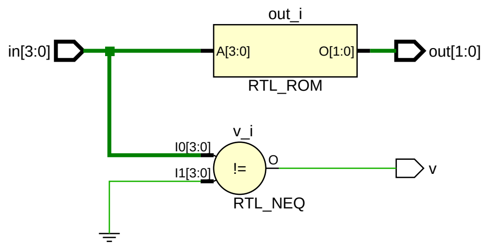
$$\arg\min_{(s,n)} TC(s,n), \{n,s,w\} \subseteq \mathbb{Z}^{+} \Rightarrow \begin{cases} n \in \{1,2\} \\ s = w - 1 \end{cases} \tag{16}$$

B. After Synthesis Stage Analysis

Vivado synthesizer optimizes the circuit into LUTs, analog switches and programmable ROMs after the synthesis. For low complexity modules like 4-2 priority encoder (8), either the cascaded implementation or the simplified combinational logic (using Carnot Diagram) are synthesized into a couple of LUTs with unit gate delay. For LUT implementation, the submodule is synthesized into an ROM, also with unit gate delay.



(a) Simplified combinational logic implementation



(b) LUT implementation

Fig. 8: Alternative implementation of 4-2 encoder submodule

Thus in the time complexity object Eq.(11), the unit submodule delay suggests that

$$\frac{W}{S} = 1, N = 1 \Rightarrow TC(s,n) = \mathcal{O}(L) = \mathcal{O}(\frac{s}{n}) \tag{17}$$

To minify the possibility of outbounded length in critical paths produced by synthesizer, the input widths W_{bb} are selected to 4, and the number of layers is $L = 1$. Therefore, parameters selection (6) is still appropriate for the 16-4 priority encoder design.

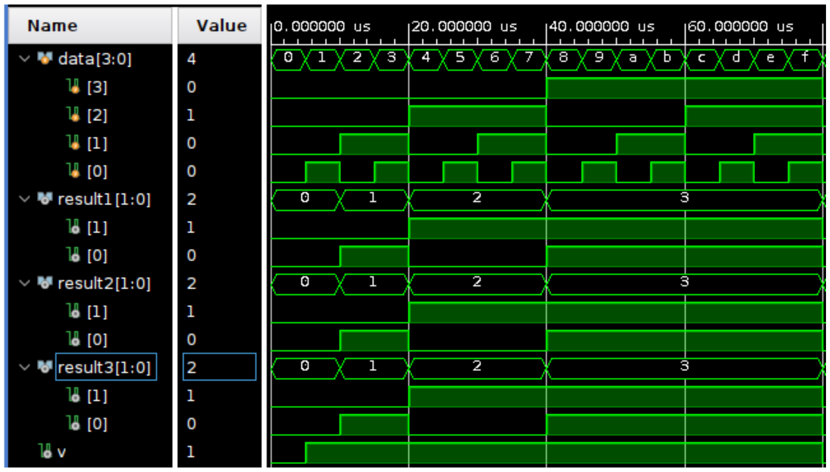


Fig. 9: Timing simulation after synthesis of 4-2 priority encoder

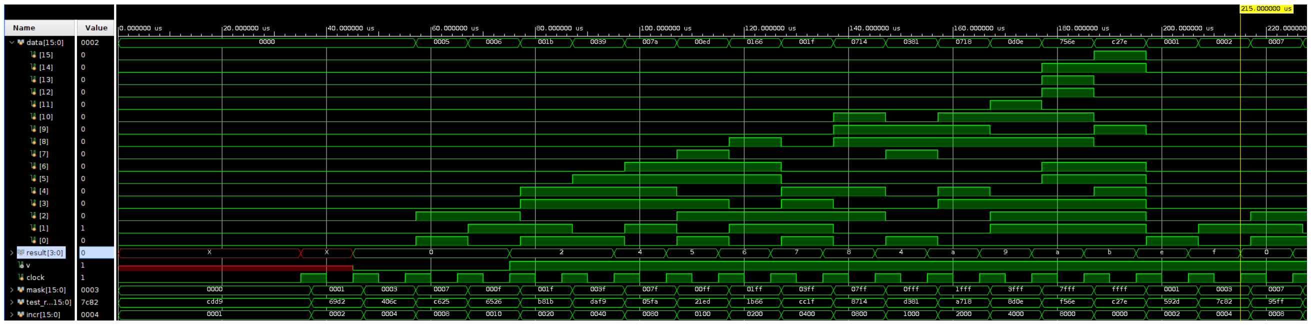


Fig. 10: Timing simulation after synthesis of 16-4 priority encoder, using combinational logic submodule

III. Simulation Results

With the elaboration and discussion above, the modeling and simulation are pretty straightforward. Figures below shows the simulation result of 4-2 encoder submodule and the final 16-4 encoder submodule:

In Fig.(9), signal `result1` is the output of cascaded architecture submodule, `result2` is the output of LUT submodule, and signal `result3` is the output of combinational logic (Carnot simplification on design time) submodule.

In Fig.(10), the consistency of the 16-4 priority encoder (with 2 layer pipelining) is verified. Note the time required for the initialization of registers at the beginning.

In Fig.(11), the consistency of the 16-4 priority encoder (with 2 layer pipelining) is verified, the identical critical path between the combinational logic design and the ROM LUT design after synthesis results in identical waveform. Note the time required for the initialization of registers at the beginning.

IV. Appendix. Code

A. Testbench for 16-4 priority encoder

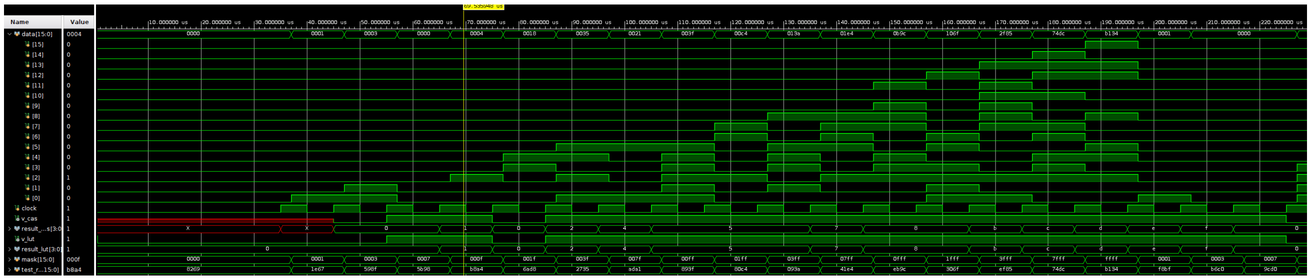


Fig. 11: Timing simulation after synthesis of 16-4 priority encoder, using LUT submodule

```verilog
`timescale 1us / 10ns
/*
 Company:
 Engineer:

 Create Date: 03/19/2024 04:49:04 PM
 Design Name:
 Module Name: tb_encoder_16to4
 Project Name:
 Target Devices:
 Tool Versions:
 Description:

 Dependencies:

 Revision:
 Revision 0.01 - File Created
 Additional Comments:
*/



module tb_encoder_16x4_lut;
  reg [15:0] data;
  reg [15:0] mask;
  reg [15:0] test_rand;
  reg [15:0] incr;
  wire [3:0] result_cas, result_lut;
  wire v_cas, v_lut;
  reg clock;
  parameter reg [32:0] powerupDelay = 30;

  encoder16x4_cas encoder (
      .clk(clock),
      .in (data),
      .out(result_cas),
      .v  (v_cas)
  );

  encoder16x4_lut encoder_lut (
      .clk(clock),
      .in (data),
      .out(result_lut),
      .v  (v_lut)
  );

  initial begin
    clock = 1'b0;
    data = 16'h0000;
    incr = 16'h0001;
    mask = 16'h0000;
    test_rand = {$urandom} & 16'hFFFF;
    #powerupDelay;  // register initialization on powerUp
    forever begin
      #5;
      clock = ~clock;
    end
  end

  always @(posedge clock) begin
    #2;
    if (mask == 16'hffff) begin
      incr = 16'h0001;
      mask = 16'h0000;
    end
    incr = incr << 1;
    test_rand = {$urandom} & 16'hFFFF;
    mask = incr - 16'h0001;
    data = test_rand & mask;
  end

endmodule
```

B. Testbench for 4-2 priority encoder

```
`timescale 1us / 10ns


/* verilator lint_off UNOPTFLAT */

module tb_encoder_4to2;
reg [3:0] data;
wire [1:0] result1;
wire [1:0] result2;
wire [1:0] result3;
wire v;

encoder4x2_cas enc_cas(
  .in(data),
  .out(result1),
  .v(v)
);

encoder4x2_lut enc_lut(
  .in(data),
  .out(result2),
  .v(v)
);

encoder4x2 enc_tree(
  .in(data),
  .out(result3),
  .v(v)
);

initial begin
  data=4'b0000;
end

always begin
  #5;
  if (data>15) begin
    data=0;
  end
  data=data+1;
end

endmodule
```