ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

Thiết kế luận lý với HDL (TN) – CO1026

Báo cáo LAB 1

GVHD:Phạm Công Thái

Nhóm 7: (23207)
Lê Đình Anh Tài – 2312994
Phạm Công Võ – 2313946
Lê Thanh Phong – 2312618
Trần Minh Dương – 2310609
Bùi Tiến An - 2310002

BÀI THỰC HÀNH THIẾT KẾ LUẬN LÝ VỚI HDL

Nhóm 7: (23207)

Lê Đình Anh Tài – 2312994

Phạm Công Võ – 2313946

Lê Thanh Phong – 2312618

Trần Minh Dương – 2310609

Bùi Tiến An - 2310002

1.1. Bài 1

a. Design a 1-to-2 decoder using structure model as the following circuit:

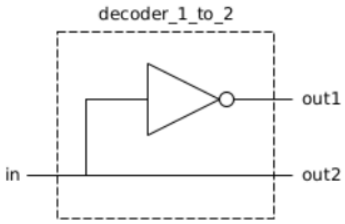


Figure 1: 1-to-2 decoder

b. Design a 2-to-1 multiplexer using structure model and hierarchical design (reuse the module decoder 1 to 2) as following circuit:

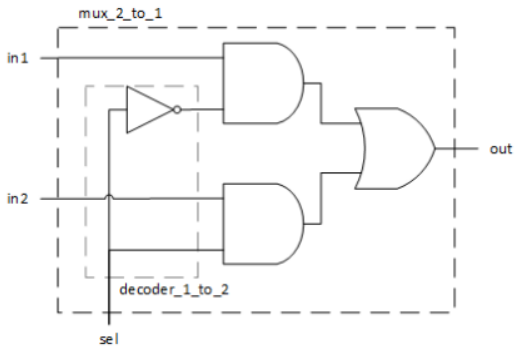
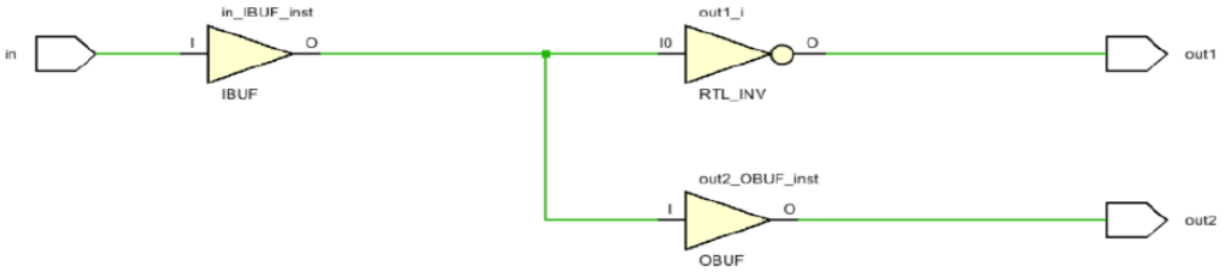


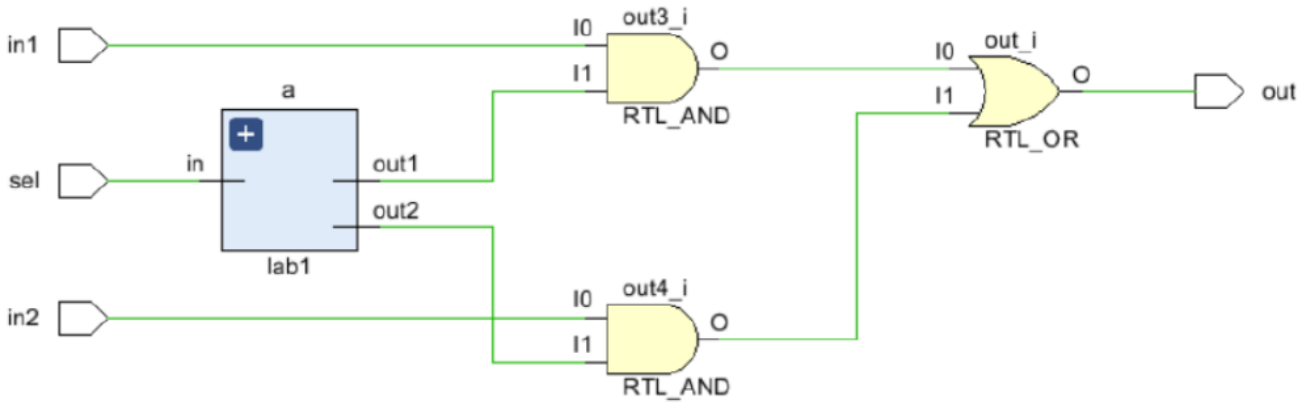
Figure 2: 2-to-1 multiplexer

Đây là một mô hình cấu trúc cho mạch 2-to-1 multiplexer:

```
module decoder_1_to_2(input in, output out1, output out2);
not not1(out1, in);
endmodule

module mux_2_to_1(input in1, input in2, input sel, output out);
    wire not_sel, and1_out, and2_out;
    decoder_1_to_2 dec(sel, not_sel);
    and and1(and1_out, in1, not_sel);
    and and2(and2_out, in2, sel);
    or(out, and1_out, and2_out);
endmodule
```

RTL Schematic





1.2 Bài 2

a. Write a test bench for the 2-to-1 Multiplexer in Exercise 1 then use Vivado Simulator to simulate the design, students can use the given example source code. Let's analyse the structure of a test bench then point out the differences between an RTL code and a test bench code.

Change the Radix, Format of signals and use zoom tool to evaluate the waveform.

Check the Tcl console window to see output of $monitor command.

b. Then, perform the Synthesis, compare the Synthesis's Schematic and the RTL Analysis's schematic.

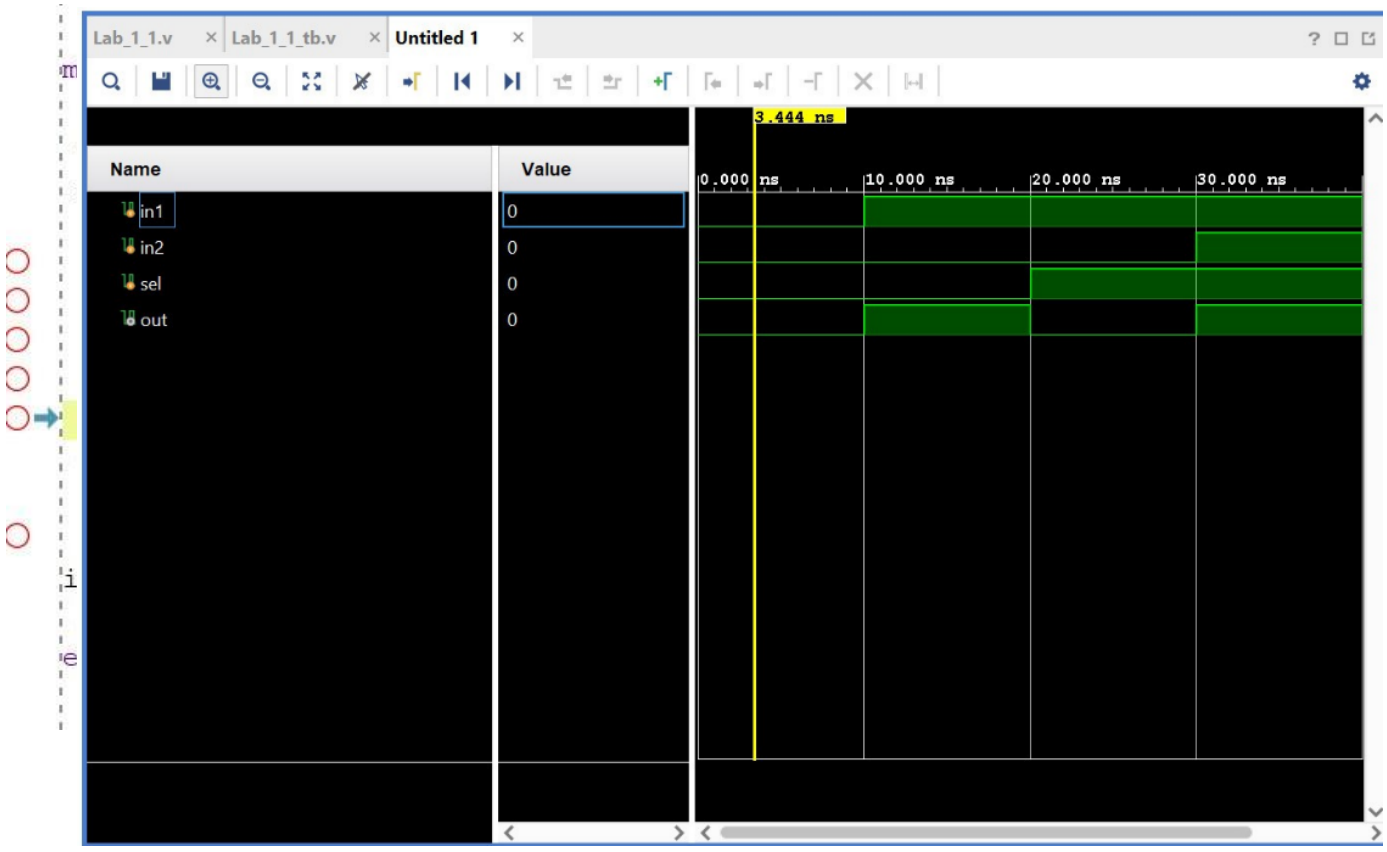
c. After that, run the Implementation, check the Utilization report in Project Summary for used resources.

d. Add the Arty-Z7 constraint file to the project, assign pin for the design as follow:

- **in1: btn[0], in2: btn[1]**
- **sel: sw[0]**
- **out: led[0]**

then, generate bitstream file and program the FPGA to test the implemented circuit on board.

Đây là test bench cho mạch 2-to-1 Multiplexer ở bài 1:



4.3. Bài 3

a. Design a half-adder circuit using structural model.

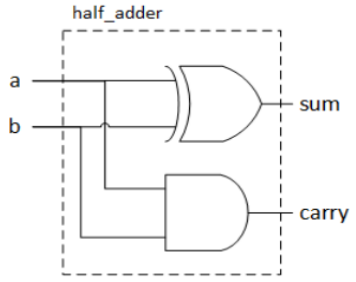


Figure 3: Half adder

Trong mạch này, cổng XOR được sử dụng để tính sum và carry đầu ra. Các bit đầu vào a và b được kết nối với đầu vào của cổng XOR. Đầu ra của cổng XOR đầu tiên (XOR b) tạo ra tổng đầu ra của bộ half-adder. Đầu ra của cổng XOR thứ hai ((XOR b) XOR (a VÀ b)) tạo ra đầu ra carry của bộ half-adder.

Mạch cấu trúc cho bộ half-adder:

```
module Add_half(c_out, sum, a, b);
output sum, c_out;
input a, b;
xor sum_bit(sum, a, b);
and carry_bit(c_out, a, b);
endmodule
```

b. Design a full-adder circuit using structural model. Reuse the half-adder module

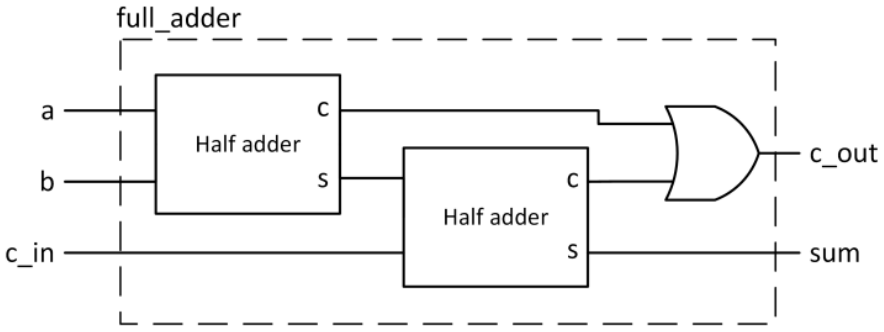


Figure 4: Full adder

Write a test bench to evaluate operations of the implemented full-adder circuit. Test the implemented circuit on Arty-Z7 board using switches/buttons and LEDs.

Bonus: show the operands and sum on 7-seg LEDs.

Để thiết kế một mạch full-adder bằng cách sử dụng mô hình cấu trúc, chúng ta sẽ cần kết hợp hai half-adder và một cổng OR. Chúng ta có thể sử dụng lại mô-đun half-adder cho mục đích này.

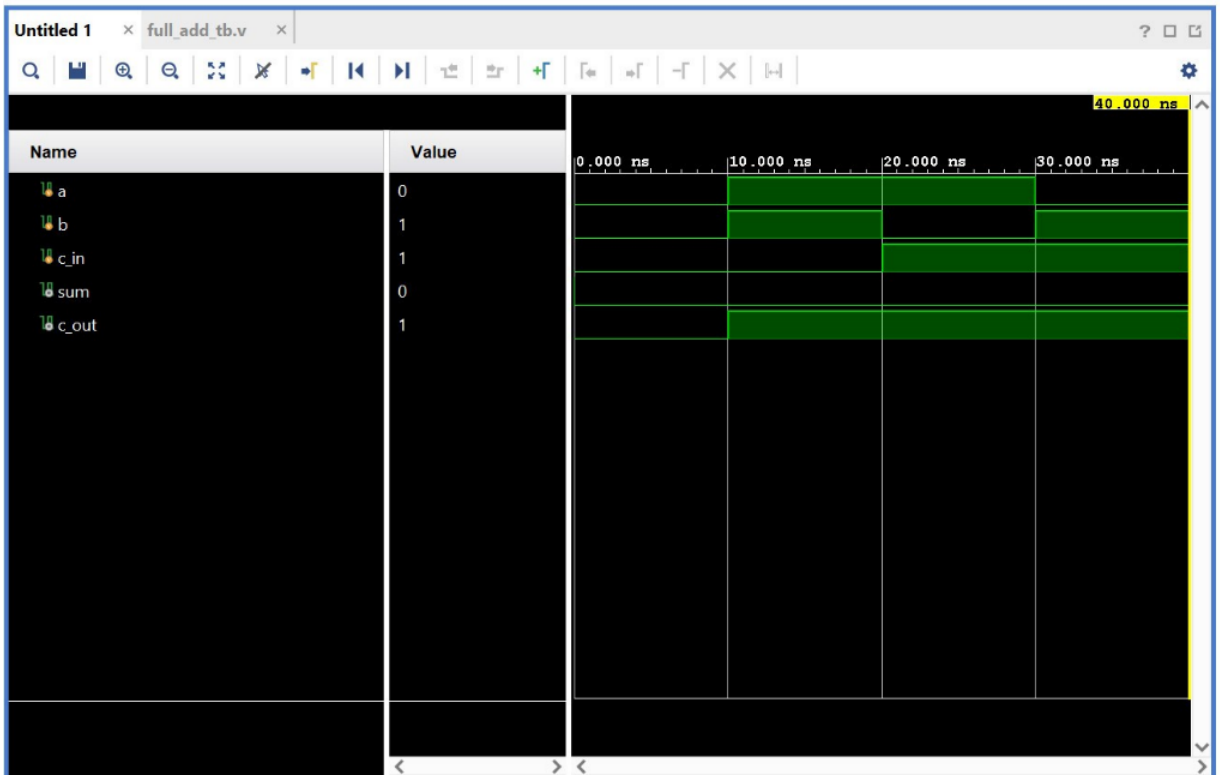
Đây là mô hình cấu trúc cho mạch full-adder:

```
module Add_half(c_out, sum, a, b);
output sum, c_out;
input a, b;
xor sum_bit(sum, a, b);
and carry_bit(c_out, a, b);
endmodule

module Add_full(c_out, sum, a, b, c_in);
output sum, c_out;
input a, b, c_in;
wire w1, w2, w3;
Add_half AH1(.sum(w1), .c_out(w2), .a(a), .b(b));
Add_half AH2(.sum(sum), .c_out(w3), .a(c_in), .b(w1));
or carry_bit(c_out, w2, w3);
endmodule
```

Trong mô hình này, full-adder nhận ba đầu vào: A, B và Cin và tạo ra hai đầu ra: Sum và Cout. Bộ half-adder đầu tiên lấy A và B, và tạo ra một tổng tạm thời và đầu ra mang theo. Half-adder thứ 2 lấy tổng tạm thời và Cin, và tạo ra tổng cuối cùng và một sản lượng mang tạm thời khác. Cuối cùng, một cổng OR kết hợp hai đầu ra mang tạm thời để tạo ra đầu ra mang cuối cùng Cout.

Test bench cho mạch full-adder:

```
22
23  module Add_full_tb;
24    reg a, b, c_in;
25    wire sum, c_out;
26
27    Add_full uut(
28      .sum(sum),
29      .c_out(c_out),
30      .a(a),
31      .b(b),
32      .c_in(c_in)
33    );
34
35    initial begin
36      $display("a b c_in | sum c_out");
37      $monitor("%b %b %b   | %b %b", a, b, c_in, sum, c_out);
38      a = 0;b = 0;c_in = 0;
39      #10; a = 1;b = 1;c_in = 0;
40      #10; a = 1;b = 0;c_in = 1;
41      #10; a = 0;b = 1;c_in = 1;
42      #10;
43      $finish;
44    end
45  endmodule
```



Bảng test bench này đặt các giá trị của A, B và Cin cho tất cả các kết hợp có thể có và mô phỏng mạch full-adder. Câu lệnh $monitor hiển thị các giá trị của A, B, Cin, Sum và Cout cho mỗi tổ hợp đầu vào.

c. Design a 4-bit ripple carry adder using structural model. Reuse the implemented full-adder. Write a test bench to simulate the implemented circuit.

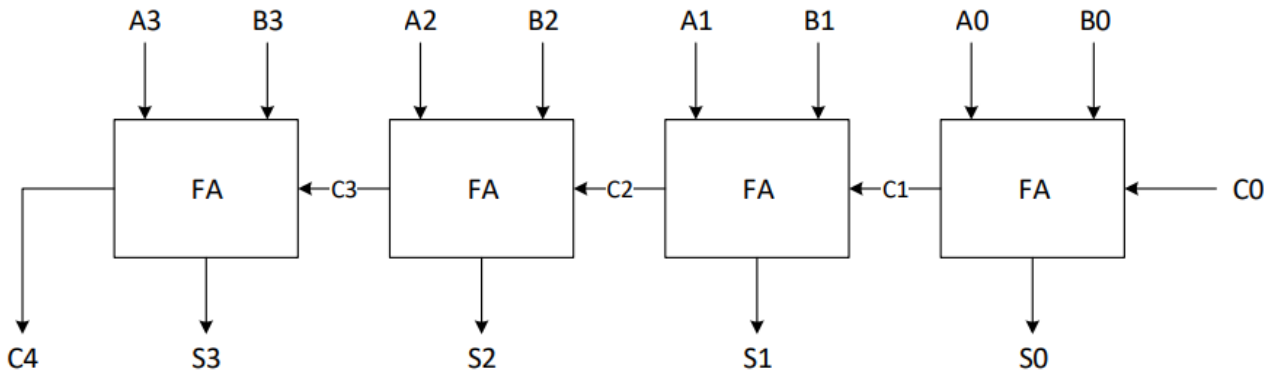


Figure 5: 4-bit ripple carry adder

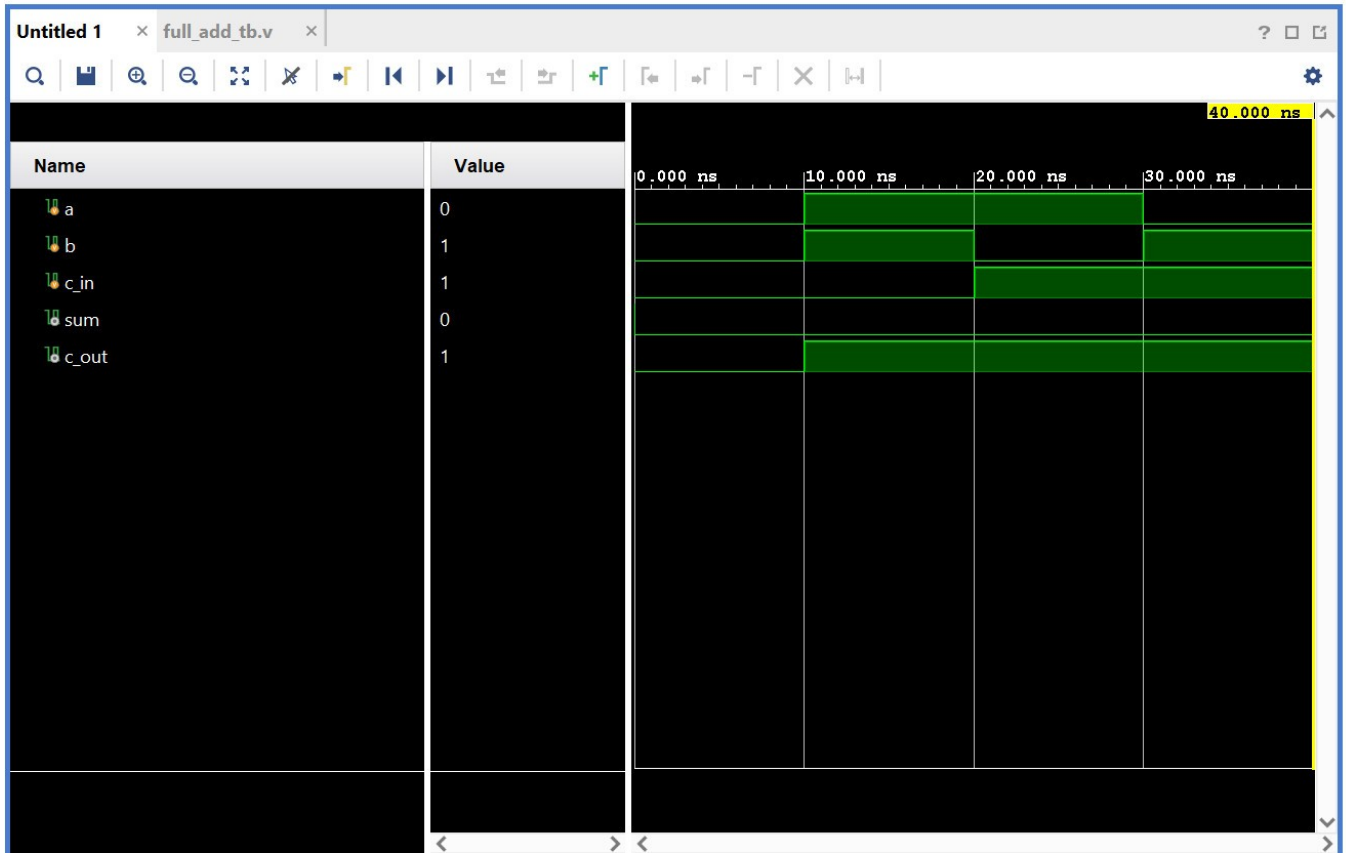
Đây là một mô hình cấu trúc cho bộ cộng 4-bit ripple carry bằng cách sử dụng mô-đun full-adder:

```
module ripple_carry_adder(input [3:0] A, input [3:0] B, output reg [3:0] sum,
output reg carry);
 wire [3:0] carry_in;
 assign carry_in[0] = 1'b0;
 assign carry_in[1] = carry;
 assign carry_in[2] = carry_in[1];
 assign carry_in[3] = carry_in[2];

full_adder fa0(.a(A[0]), .b(B[0]), .c(carry_in[0]), .sum(sum[0]),
.carry(carry_in[1]));
 full_adder fa1(.a(A[1]), .b(B[1]), .c(carry_in[1]), .sum(sum[1]),
.carry(carry_in[2]));
 full_adder fa2(.a(A[2]), .b(B[2]), .c(carry_in[2]), .sum(sum[2]),
.carry(carry_in[3]));
 full_adder fa3(.a(A[3]), .b(B[3]), .c(carry_in[3]), .sum(sum[3]),
.carry(carry));
endmodule
```

Trong thiết kế này, trước tiên chúng tôi khai báo một **carry_in** dây được sử dụng để nhân giống mang từ bit này sang bit khác. Chúng tôi đặt bit đầu tiên của **carry_in** thành 0 và các bit còn lại được đặt thành đầu ra mang của bit trước đó. Sau đó, chúng tôi khởi tạo bốn phiên bản của mô-đun **full_adder**, kết nối các đầu vào **A** và **B** với đầu vào **a** và **b** của bộ cộng đầy đủ, tương ứng. Bit **carry_in** tương ứng được kết nối với đầu vào **c** của mỗi bộ cộng đầy đủ. **Sum đầu** ra của mỗi bộ cộng đầy đủ được kết nối với bit tương ứng của **tổng** đầu ra của mô-đun cộng mang gợn sóng. Cuối cùng, đầu ra **carry** của bộ **full_adder** cuối cùng được kết nối với đầu ra **carry** của mô-đun ripple carry adder.

Dưới đây là test bench cho bộ cộng mang gợn sóng 4 bit:

```
module testbench();
reg [3:0] A, B;
wire [3:0] sum;
wire carry;
ripple_carry_adder rca(.A(A), .B(B), .sum(sum), .carry(carry));
initial begin
$dumpfile("test.vcd");
$dumpvars(0, testbench);
A = 4'b0000;
B = 4'b0000;
#10 $display("A = %b, B = %b, sum = %b, carry = %b", A, B, sum, carry);
A = 4'b0001;
B = 4'b0001;
#10 $display("A = %b, B = %b, sum = %b,
```



Mô hình cấu trúc đã cho thực hiện một bộ cộng mang gợn 4 bit bằng cách sử dụng mô-đun full adder được triển khai trước đó.

Trong bộ cộng mang gợn sóng, mỗi bit của toán hạng **A** và **B** được thêm vào bằng cách sử dụng một bộ full adder. Đầu ra carry của mỗi bộ full adder được kết nối với đầu vào carry của bộ full adder tiếp theo, lan truyền carry từ bit này sang bit tiếp theo. Cuối cùng, đầu ra carry của bộ full adder cuối cùng là đầu ra carry tổng thể của bộ cộng 4 bit. **Đầu** ra Sum của mỗi bộ full adder được kết nối với bit tương ứng của **Sum** đầu ra của bộ cộng 4 bit.

Mô-đun full adder lấy ba đầu vào, **a**, **b** và **c**, lần lượt là hai toán hạng và đầu vào carry. Mô-đun xuất ra Sum và Carry cho bit đó của bộ cộng. Tổng được tính bằng XOR của ba đầu vào, trong khi carry được tính bằng cổng AND và OR.

Test bench khởi tạo toán hạng **A** và **B** thành 0 và sau đó thành 1. Bàn kiểm tra sau đó hiển thị đầu vào và đầu ra của bộ cộng 4 bit sau mỗi lần lặp. Các tác vụ hệ thống **$dumpfile** và **$dumpvars** được sử dụng để tạo tệp VCD để phân tích dạng sóng.

4.3 Bài 3

Give the 2-bit comparator circuit as Figure 6 with $A = \{A1, A0\}$ and $B = \{B1, B0\}$ are 2 2-bit input numbers, A_gt_B is active if $A > B$, A_lt_B is active if $A < B$ and A_eq_B is active if $A = B$.

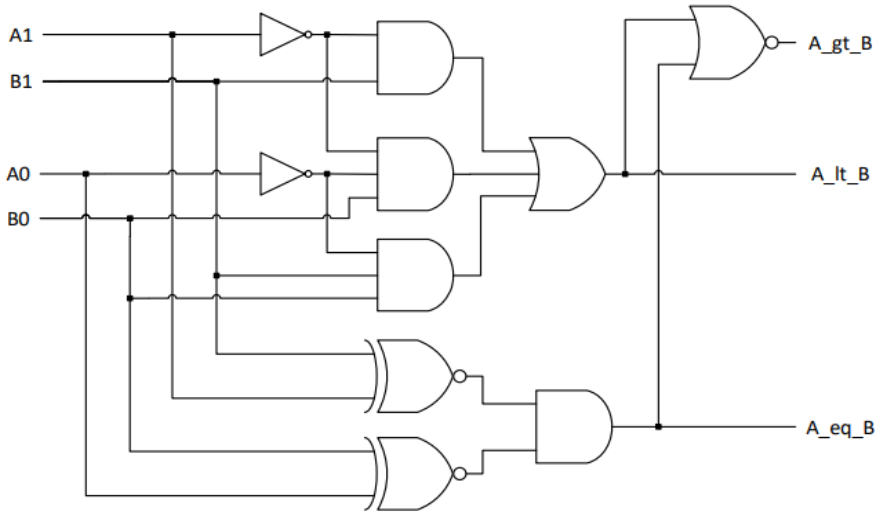


Figure 6: 2-bit comparator

Let's design a 4-bit comparator following below steps:

• Analyse the functions of each output of the 2-bit comparator, then determine the functions of 4-bit comparator outputs.

• Conceptualize the design of 4-bit comparator from 2-bit comparators (the 2-bit comparator can be partitioned into smaller blocks). Draw a block diagram that describes the idea.

• Draw a diagram to shows the designed circuit hierarchy.

• Implement the designed circuit using Verilog HDL structural model

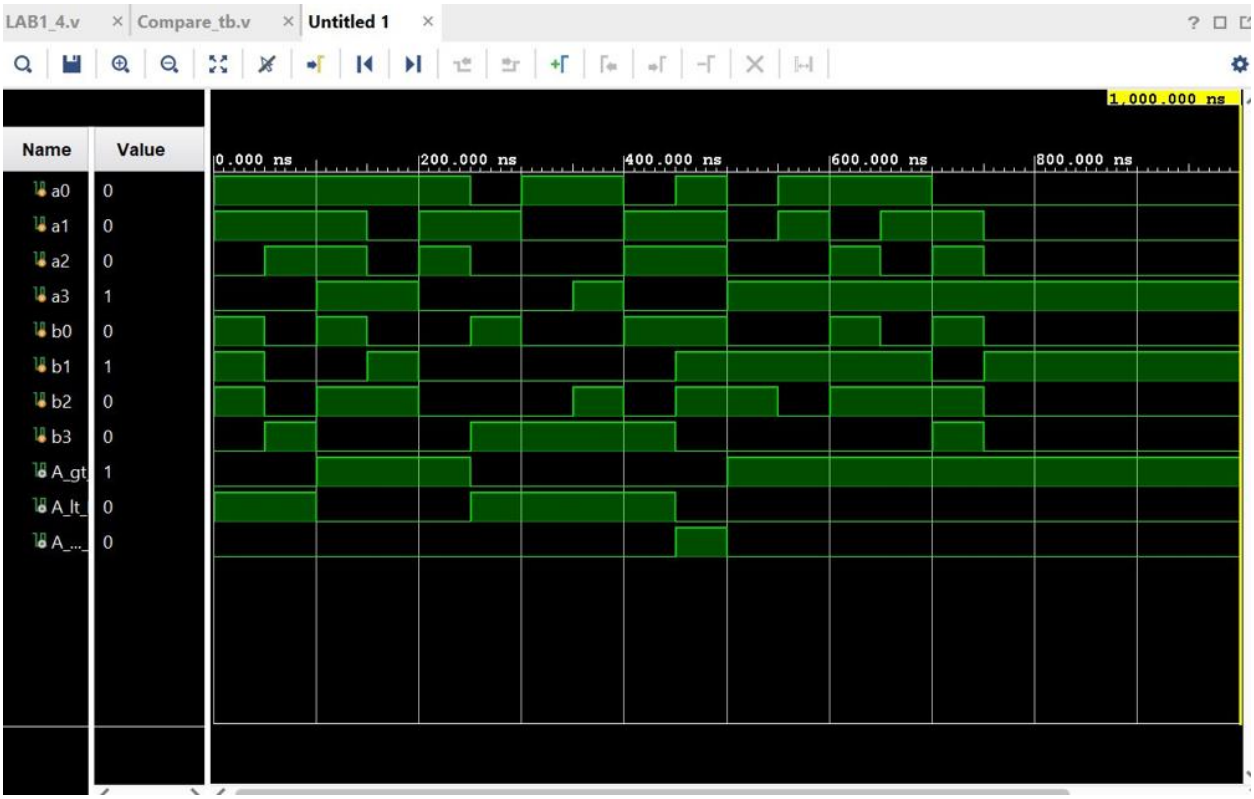
. • Write a test bench to simulate the implemented circuit.

Giải pháp:

Xem xét mạch so sánh 2 bit được đưa ra trong bài toán, nó bao gồm 2 đầu vào (A, B) của 2 bit mỗi (A1, A0 and B1, B0) và 3 đầu ra được xác định là:

- A_eq_B = 1 when A = B
- A_lt_B = 1 when A < B
- A_gt_B = 1 when A > B

```
module comparator4bit(A_gt_B, A_lt_B, A_eq_B, a0, a1, a2, a3, b0, b1, b2, b3);
   input a0, a1, a2, a3, b0, b1, b2, b3;
   output A_gt_B, A_lt_B, A_eq_B;

   wire Am_gt_Bm, Am_lt_Bm, Am_eq_Bm, Al_gt_Bl, Al_lt_Bl, Al_eq_Bl;

   comparator2bit comp1(Am_gt_Bm, Am_lt_Bm, Am_eq_Bm, a3, a2, b3, b2);
   comparator2bit comp2(Al_gt_Bl, Al_lt_Bl, Al_eq_Bl, a1, a0, b1, b0);

   assign A_gt_B = Am_gt_Bm | (Am_eq_Bm & Al_gt_Bl) ;
   assign A_lt_B = Am_lt_Bm | (Am_eq_Bm & Al_lt_Bl);
   assign A_eq_B = Am_eq_Bm & Al_eq_Bl;
endmodule

module comparator2bit(A_gt_B, A_lt_B, A_eq_B, a1, a0, b1, b0);
   input a1, a0, b1, b0;
   output A_gt_B, A_lt_B, A_eq_B;

   assign A_gt_B = ~(A_eq_B | A_lt_B);
   assign A_lt_B = (~a1 & b1) | (~a1 & ~a0 & b0) | (~a0 & b1 & b0);
   assign A_eq_B = (~(a1^b1)) & (~(a0^b0));
endmodule
```



Kết quả mô phỏng