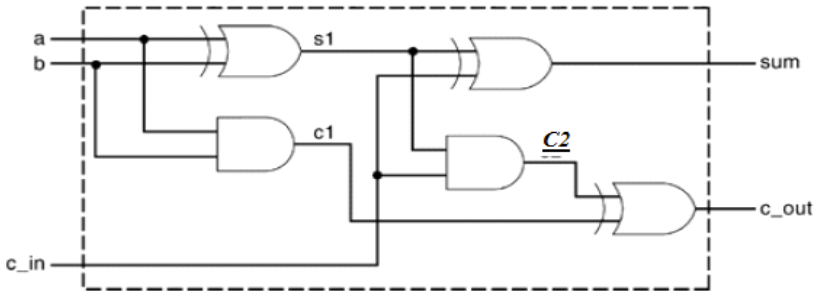
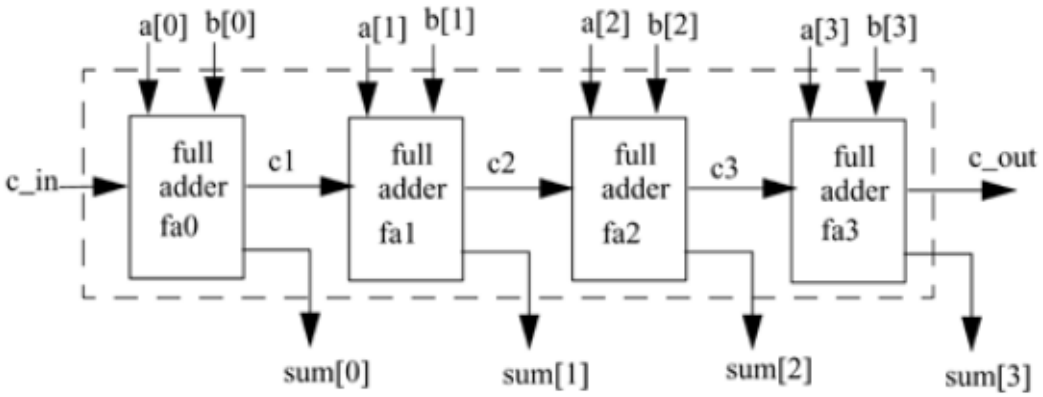
Verilog Program for Ripple Carry Adder using Gate Level Modeling

4-bit Ripple Carry Full Adder

Figure 5-6. 1-bit Full Adder




```
// 1-//bit full adder
module fulladd(sum, c_out, a, b, c_in);

// I/O port declarations
output sum, c_out;
input a, b, c_in;
wire s1, c1, c2;

// Instantiate logic gate primitives
xor (s1, a, b);
and (c1, a, b);
xor (sum, s1, c_in);
and (c2, s1, c_in);
xor (c_out, c2, c1);
endmodule
```



```
// Define a 4-bit full adder
module fulladd4(sum, c_out, a, b, c_in);
// I/O port declarations
output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in;
// Internal nets
wire c1, c2, c3;
// Instantiate four 1-bit full adders.
fulladd fa0(sum[0], c1, a[0], b[0], c_in);
fulladd fa1(sum[1], c2, a[1], b[1], c1);
fulladd fa2(sum[2], c3, a[2], b[2], c2);
fulladd fa3(sum[3], c_out, a[3], b[3], c3);
endmodule
```

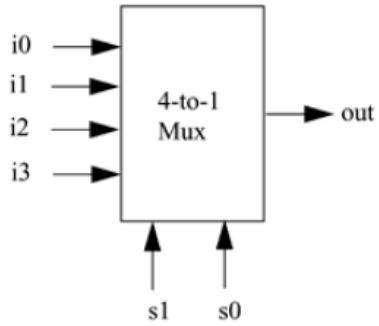
stimulus oor testbench program

```
// Define the stimulus (top level module)
module stimulus;

// Set up variables
reg [3:0] A, B;
reg C_IN;
wire [3:0] SUM;
wire C_OUT;

// Instantiate the 4-bit full adder. call it FA1_4
fulladd4 FA1_4(SUM, C_OUT, A, B, C_IN);


// Set up the monitoring for the signal values
initial
begin
  $monitor($time," A= %b, B=%b, C_IN= %b, --- C_OUT= %b, SUM= %b\n",
                          A, B, C_IN, C_OUT, SUM);
end

// Stimulate inputs
initial
begin
  A = 4'd0; B = 4'd0; C_IN = 1'b0;

  #5 A = 4'd3; B = 4'd4;

  #5 A = 4'd2; B = 4'd5;

  #5 A = 4'd9; B = 4'd9;

  #5 A = 4'd10; B = 4'd15;

  #5 A = 4'd10; B = 4'd5; C_IN = 1'b1;
end

endmodule
```

The output of the simulation is shown below.

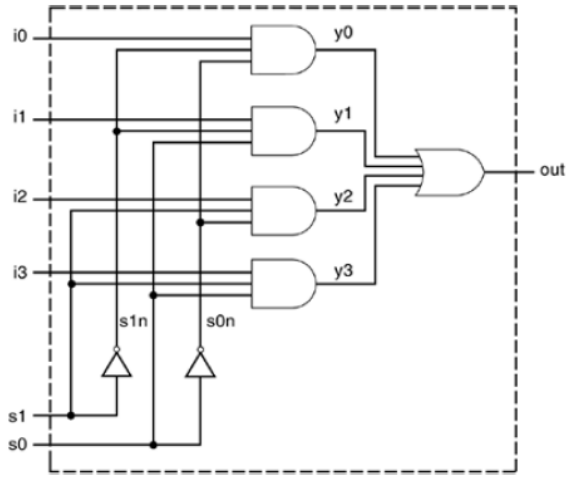
```
0 A= 0000, B=0000, C_IN= 0, --- C_OUT= 0, SUM= 0000
5 A= 0011, B=0100, C_IN= 0, --- C_OUT= 0, SUM= 0111
10 A= 0010, B=0101, C_IN= 0, --- C_OUT= 0, SUM= 0111
15 A= 1001, B=1001, C_IN= 0, --- C_OUT= 1, SUM= 0010
20 A= 1010, B=1111, C_IN= 0, --- C_OUT= 1, SUM= 1001
25 A= 1010, B=0101, C_IN= 1,, C_OUT= 1, SUM= 0000
```

4:1 Mux based on gatelevel modelling



s1	s0	out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

4:1 Mux




```
// Module 4-to-1 multiplexer. Port list is taken exactly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;

// Gate instantiations

// Create s1n and s0n signals.
not (s1n, s1);
not (s0n, s0);

// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);

// 4-input or gate instantiated
or (out, y0, y1, y2, y3);

endmodule
```

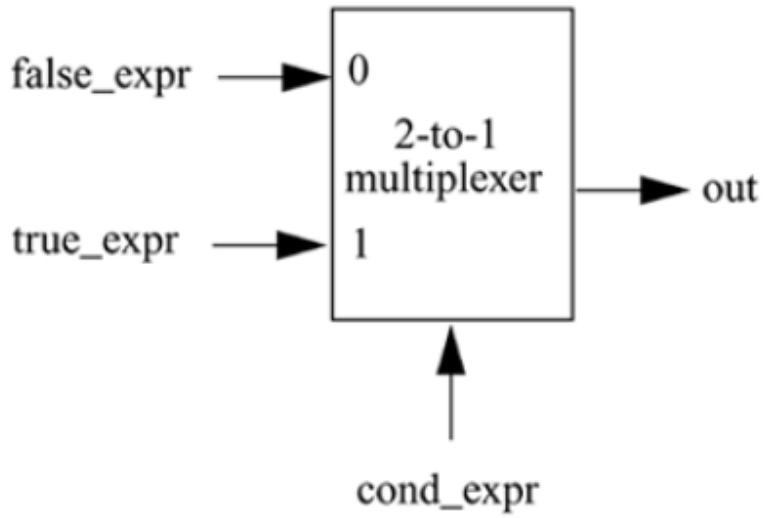
```
// Define the stimulus module (no ports)
module stimulus;

// Declare variables to be connected
// to inputs
reg IN0, IN1, IN2, IN3;

reg S1, S0;

// Declare output wire
wire OUTPUT;

// Instantiate the multiplexer
mux4_to_1 mymux(OUTPUT, IN0, IN1, IN2, IN3, S1, S0);

// Stimulate the inputs
// Define the stimulus module (no ports)
initial
begin
  // set input lines
  IN0 = 1; IN1 = 0; IN2 = 1; IN3 = 0;
  #1 $display("IN0= %b, IN1= %b, IN2= %b, IN3= %b\n",IN0,IN1,IN2,IN3);

  // choose IN0
  S1 = 0; S0 = 0;
  #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

  // choose IN1
  S1 = 0; S0 = 1;
  #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

  // choose IN2
  S1 = 1; S0 = 0;
  #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);

  // choose IN3
  S1 = 1; S0 = 1;
  #1 $display("S1 = %b, S0 = %b, OUTPUT = %b \n", S1, S0, OUTPUT);
end

endmodule
```

U can use $monitor in the place $display

The output of the simulation is shown below. Each combination of the select signals is tested.

```
IN0= 1, IN1= 0, IN2= 1, IN3= 0

S1 = 0, S0 = 0, OUTPUT = 1

S1 = 0, S0 = 1, OUTPUT = 0

S1 = 1, S0 = 0, OUTPUT = 1

S1 = 1, S0 = 1, OUTPUT = 0
```

2:1 mux based on condition operator



Conditional operators are frequently used in dataflow modeling to model conditional assignments. The conditional expression acts as a switching control.

```
//model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'bz;

//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;


endmodule
```

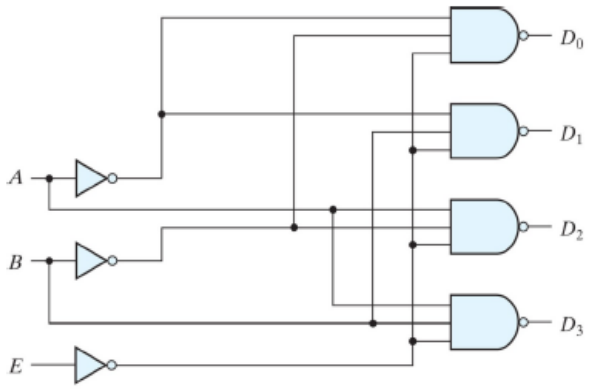
4:1 mux based on condition operator

```
// Module 4-to-1 multiplexer using data flow. Conditional operator.
// Compare to gate-level model
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

// Use nested conditional operator
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;

endmodule
```

4:1 mux based on Data flow modelling

```
// Module 4-to-1 multiplexer using data flow. logic equation
// Compare to gate-level model
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;

//Logic equation for out
assign out =    (~s1 & ~s0 & i0)|
                (~s1 & s0 & i1) |
                (s1 & ~s0 & i2) |
                (s1 & s0 & i3) ;

endmodule
```

<u>Decoder based on Gate level modeling</u>



(a) Logic diagram

E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

```
// HDL example 4.1
// Gate-level description of 2-to-4 decoder

module decoder( D, A, B, enable );
  output [0:3] D;            // vector of 4 bits
  input  A, B;
  input  enable;
  wire   Anot, Bnot, enableNot;

  not
    G1 (Anot, A),            // note syntax:  list of gates
    G2 (Bnot, B),            // separated by ,
    G3 (enableNot, enable);

  nand
    G4 (D[0], Anot, Bnot, enableNot ),
    G5 (D[1], Anot, B,    enableNot ),
    G6 (D[2], A,    Bnot, enableNot ),
    G7 (D[3], A,    B,    enableNot );
endmodule
```