

Gate Level Modelling

Problem Statement

Objective:

Design and implement a **2×2 Binary Multiplier** using **primitive logic gates** and **half adders** in Verilog.

Description:

Let the binary inputs be

$$A = A_1A_0, \quad B = B_1B_0$$

The multiplication result is a 4-bit output:

$$P = P_3P_2P_1P_0 = A \times B$$

What Are Verilog Primitive Gates?

Primitive gates are **built-in logic elements** provided by Verilog.

They are **not user-defined modules** — they are **keywords** that directly represent **basic hardware logic gates**.

👉 You don't write their internal code;
you just *instantiate* them like ready-made components.

Why Called “Primitive”?

Because they are the **lowest-level building blocks** —

the smallest hardware elements from which larger circuits (like adders, multipliers, etc.) are built.

Verilog Gate Primitives

- Verilog provides primitive gates. One can instantiate these gates and use these gates to construct a logic circuit. This is also called Gate-level Structural Modelling.

and (w, i₁, i₂ ...)



nand (w, i₁, i₂ ...)



or (w, i₁, i₂ ...)



nor (w, i₁, i₂ ...)



xor (w, i₁, i₂ ...)



xnor (w, i₁, i₂ ...)



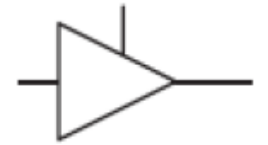
not (w, i)



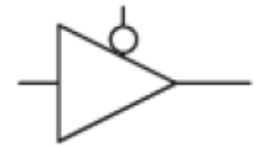
buf (w, i)



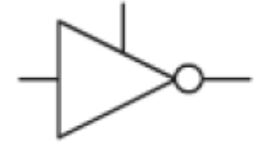
bufif1 (w, i, c)



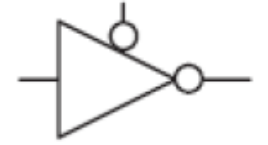
bufif0 (w, i, c)



notif1 (w, i, c)



notif0 (w, i, c)



HA.v - D:/codes

```
1 module HA(a,b,s,c);  
2   input a,b;  
3   output s,c;  
4   xor g1(s,a,b);  
5   and g2(c,a,b);  
6 endmodule
```

FA.v - D:/codes

```
1 module FA(a,b,cin,s,cout);  
2   input a,b,cin;  
3   output s,cout;  
4   wire s1,c1,c2;  
5   // instantiate HAs  
6   HA m1(a,b,s1,c1);  
7   HA m2(s1,cin,s,c2);  
8   or o1(c,c1,c2);  
9 endmodule
```

4bit_RCA.v - D:/codes

```
1 module 4bit_RCA(x,y,s,co);  
2   input [3:0] x,y; // Two 4-bit inputs  
3   output [3:0] s;  
4   output co;  
5   wire w1,w2,w3;  
6   // instantiating 4 1-bit full adders in Verilog  
7   FA u1(x[0],y[0],1'b0,s[0],w1);  
8   FA u2(x[1],y[1],w1,s[1],w2);  
9   FA u3(x[2],y[2],w2,s[2],w3);  
10  FA u4(x[3],y[3],w3,s[3],co);  
11 endmodule
```

```
module half_adder(output sum, carry, input a, b);  
    xor (sum, a, b);  
    and (carry, a, b);  
endmodule
```

```

module multiplier_2x2_HA (
    output [3:0] P,
    input  [1:0] A, B

);

wire a0b0, a1b0, a0b1, a1b1;
wire sum1, carry1, carry2;

// Generate partial products
and (a0b0, A[0], B[0]);
and (a1b0, A[1], B[0]);
and (a0b1, A[0], B[1]);
and (a1b1, A[1], B[1]);

// First bit
assign P[0] = a0b0;

// Middle bits using half adders
half_adder HA1 (sum1, carry1, a1b0, a0b1);    // Add (A1B0 + A0B1)
half_adder HA2 (P[2], carry2, a1b1, carry1);  // Add (A1B1 + carry1)

// Connect outputs
assign P[1] = sum1;
assign P[3] = carry2;

endmodule

```

Problem Statement

Design a **4-bit Binary to Gray code converter** using **primitive logic gates** (`xor` , `buf`) in Verilog.

The output Gray code should follow the rule:

$$G_i = B_{i+1} \oplus B_i$$

and

$$G_{n-1} = B_{n-1}$$

```
//=====
// 4-bit Binary to Gray Code Converter
// Using Primitive Gates
//=====
module binary_to_gray (
    output [3:0] G,
    input  [3:0] B
);
    // G3 = B3
    buf (G[3], B[3]);

    // G2 = B3 XOR B2
    xor (G[2], B[3], B[2]);

    // G1 = B2 XOR B1
    xor (G[1], B[2], B[1]);

    // G0 = B1 XOR B0
    xor (G[0], B[1], B[0]);
endmodule
```

Objective:

Design and implement a **4-bit Gray Code Counter** in Verilog that counts in the Gray code sequence (only one bit changes between successive states) using **primitive gates and flip-flops**.

Concept Overview

- A **Gray counter** can be derived from a **binary counter** using the relation:

$$G = B \oplus (B \gg 1)$$

where B is the binary count and G is the Gray output.

- Alternatively, it can be directly designed using **combinational logic and D flip-flops**.

Each clock pulse advances the counter to the next Gray code value.

```
//=====
// 4-bit Gray Code Counter
//=====
module gray_counter (
    input clk, rst,
    output [3:0] G
);
    wire [3:0] B;

    // Instantiate binary counter
    binary_counter_4bit BC (clk, rst, B);

    // Convert to Gray code
    binary_to_gray BG (G, B);
endmodule
```