

Unit-2

Gate Level Modeling

Introduction

Digital designers are normally familiar with all the common logic gates, their symbols, and their working. Flip-flops are built from the logic gates. All other functionally complex and more involved circuits can also be built using the basic gates. All the basic gates are available as “Primitives” in Verilog. Primitives are generalized modules that already exist in Verilog [IEEE]. They can be instantiated directly in other modules.

And Gate Primitive

The AND gate primitive in Verilog is instantiated with the following statement:

`and g1 (O, I1, I2, . . . , In);`

Here ‘and’ is the keyword signifying an AND gate. g1 is the name assigned to the specific instantiation. O is the gate output; I1, I2, etc., are the gate inputs. The following are noteworthy:

- The AND module has only one output. The first port in the argument list is the output port.
- An AND gate instantiation can take any number of inputs — the upper limit is compiler-specific.
- A name need not be necessarily assigned to the AND gate instantiation; this is true of all the gate primitives available in Verilog.

Truth Table of AND Gate Primitive

The truth table for a two-input AND gate is shown in Table below. It can be directly extended to AND gate instantiations with multiple inputs. The following observations are in order here:

Truth table of AND gate primitive

		Input 1			
		0	1	X	z
Input 2	0	0	0	0	0
	1	0	1	X	x
	x	0	x	X	x
	z	0	x	X	x

- If any one of the inputs to the AND gate instantiation is in the 0 state, its output is also in the 0 state. It is irrespective of whether the other inputs are at the 0, 1, x or z state.
- The output is at 1 state if and only if every one of the inputs is at 1 state.
- For all other cases the output is at the x state.
- Note that the output is never at the z state – the high impedance state. This is true of all other gate primitives as well.

Module Structure

In a general case a module can be more elaborate. A lot of flexibility is available in the definition of the body of the module. However, a few rules need to be followed:

- The first statement of a module starts with the keyword module; it may be followed by the name of the module and the port list if any.
- All the variables in the ports-list are to be identified as inputs, outputs, or inout. The corresponding declarations have the form shown below:

```
? Input a1, a2;
? Output b1, b2;
? Inout c1, c2;
```

The port-type declarations here follow the module declaration mentioned above.

- The ports and the other variables used within the body of the module are to be identified as nets or registers with specific types in each case. The respective declaration statements follow the port-type declaration statements.

Examples:

```
wire a1, a2, c;
reg b1, b2;
```

The type declaration must necessarily precede the first use of any variable or signal in the module.

- The executable body of the module follows the declaration indicated above.
- The last statement in any module definition is the keyword “endmodule”.
- Comments can appear anywhere in the module definition.

Other Gate Primitives

All other basic gates are also available as primitives in Verilog. Details of the facilities and instantiations in each case are given in Table below. The following points are noteworthy here:

- In all cases of instantiations, one need not necessarily assign a name to the instantiation. It need be done only when felt necessary – say for clarity of circuit description.
- In all the cases the output port(s) is (are) declared first and the input port(s) is (are) declared subsequently.
- The buffer and the inverter have only one input each. They can have any number of outputs; the upper limit is compiler-specific. All other gates have one output each but can have any number of inputs; the upper limit is again compiler-specific.

Table for Basic gate primitives in Verilog with details

Gate	Mode of instantiation	Output port(s)	Input port(s)
AND	and ga (o, i1, i2, . . . i8);	o	i1, i2, . .
OR	or gr (o, i1, i2, . . . i8);	o	i1, i2, . .
NAND	nand gna (o, i1, i2, . . . i8);	o	i1, i2, . .
NOR	nor gnr (o, i1, i2, . . . i8);	o	i1, i2, . .
XOR	xor gxr (o, i1, i2, . . . i8);	o	i1, i2, . .
XNOR	xnor gxn (o, i1, i2, . . . i8);	o	i1, i2, . .
BUF	buf gb (o1, o2, i);	o1, o2, o3, . .	i
NOT	not gn (o1, o2, o3, . . . i);	o1, o2, o3, . .	i

Example for a typical A-O-I gate circuit

The commonly used A-O-I gate is shown in Figure 1 for a simple case. The module and the test bench for the same are given in Figure 2. The circuit has been realized here by instantiating the AND and NOR gate primitives. The names of signals and gates used in the instantiations in the module of Figure 2 remain the same as those in the circuit of Figure 1. The module `aoi_gate` in the figure has input and output ports since it describes a circuit with signal inputs and an output. The module `aoi_st` is a stimulus module. It generates inputs to the `aoi_gate` module and gets its output. It has no input or output ports.

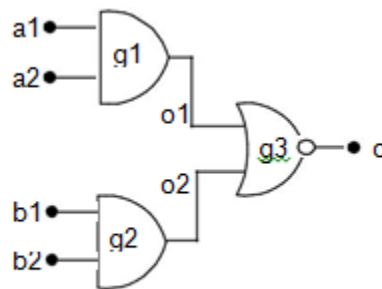


Figure for a typical A-O-I gate circuit.

```
/*module for the aoi-gate of figure 1 instantiating the gate primitives – fig 2*/
module aoi_gate(o,a1,a2,b1,b2);

input a1,a2,b1,b2;    // a1,a2,b1,b2 form the input //ports of the module

output o;            //o is the single output port of the module

wire o1,o2;          //o1 and o2 are intermediate signals //within the module

and g1(o1,a1,a2);    //The AND gate primitive has two and g2(o2,b1,b2);

                    // instantiations with assigned //names g1 & g2.

nor g3(o,o1,o2);    //The nor gate has one instantiation with assigned name g3.

endmodule

//Test-bench for the aoi_gate above
module aoi_st;
reg a1,a2,b1,b2;

//specific values will be assigned to a1,a2,b1, // and b2 and these connected
//to input ports of the gate insatntiations;
```

```

//hence these variables are declared as reg
wire o;
initial
begin
a1 = 0;
a2 = 0;
b1 = 0;
b2 = 0;
#3 a1 = 1;
#3 a2 = 1;
#3 b1 = 1;
#3 b2 = 0;
#3 a1 = 1;
#3 a2 = 0;
#3 b1 = 0;
end
initial #100 $stop;//the simulation ends after //running for 100 tu's.
initial $monitor($time , " o = %b , a1 = %b , a2 = %b , b1 = %b ,b2 = %b ",o,a1,a2,b1,b2);
aoi_gate gg(o,a1,a2,b1,b2);
endmodule

```

Tri-State Gates

Four types of tri-state buffers are available in Verilog as primitives. Their outputs can be turned ON or OFF by a control signal. The direct buffer is instantiated as
 Bufif1 nn (out, in, control);

The symbol of the buffer is shown in Figure

1. We have

- out as the single output variable
- in as the single input variable and
- control as the single control signal variable.

When
 control = 1,
 out = in.

When
 control = 0,
 out=tri-stated

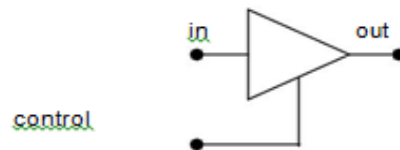
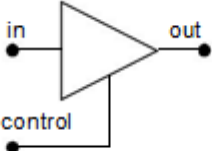
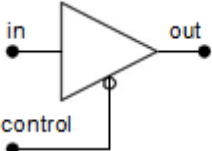
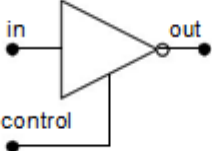
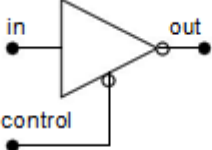


Figure 1 A tri-state buffer.

out is cut off from the input and tri-stated. The output, input and control signals should appear in the instantiation in the same order as above. Details of bufif1 as well as the other tri-state type primitives are shown in Table 1.

In all the cases shown in Table 1, out is the output; in is the input, and control, the control variable.

Table 1 Instantiation and functional details of tri-state buffer primitives

Typical instantiation	Functional representation	Functional description
<code>bufif1 (out, in, control);</code>		Out = in if control = 1; else out = z
<code>bufif0 (out, in, control);</code>		Out = in if control = 0; else out = z
<code>notif1 (out, in, control);</code>		Out = complement of in if control = 1; else out = z
<code>notif0 (out, in, control);</code>		Out = complement of in if control = 0; else out = z

Array of Instances of Primitives

The primitives available in Verilog can also be instantiated as arrays. A judicious use of such array instantiations often leads to compact design descriptions. A typical array instantiation has the form

```
and gate [7 : 4 ] (a, b, c);
```

where a, b, and c are to be 4 bit vectors. The above instantiation is equivalent to combining the following 4 instantiations:

```
and gate [7] (a[3], b[3], c[3]), gate [6] (a[2], b[2], c[2]), gate [5] (a[1], b[1], c[1]), gate [4] (a[0], b[0], c[0]);
```

The assignment of different bits of input vectors to respective gates is implicit in the basic declaration itself. A more general instantiation of array type has the form

```
and gate[mm : nn](a, b, c);
```

where mm and nn can be expressions involving previously defined parameters, integers and algebra with them. The range for the gate is $1+(mm-nn)$; mm and nn do not have restrictions of sign; either can be larger than the other.

Gate Delays

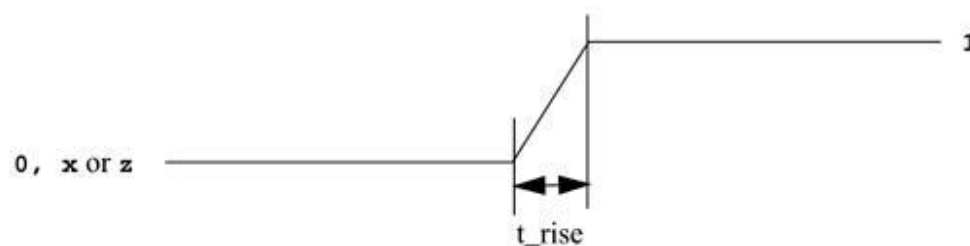
Until now, we described circuits without any delays (i.e., zero delay). In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog.

Rise, Fall, and Turn-off Delays

There are three types of delays from the inputs to the output of a primitive gate.

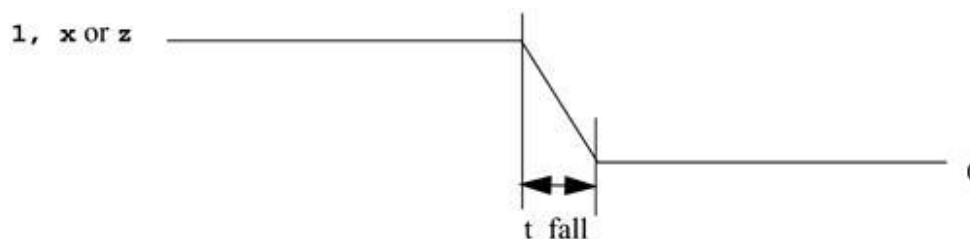
Rise delay

The rise delay is associated with a gate output transition to a 1 from another value.



Fall delay

The fall delay is associated with a gate output transition to a 0 from another value.



Turn-off delay

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value.

If the value changes to x, the minimum of the three delays is considered.

Three types of delay specifications are allowed. If only one delay is specified, this value is used for all transitions. If two delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays. If all three delays are specified, they refer to rise, fall, and turn-off delay values. If no delays are specified, the default value is zero.

Example--Types of Delay Specification

```
//Delay of delay_time for all transitions  
and #(delay_time) a1(out, i1, i2);
```

```
// Rise and Fall Delay Specification.
```

```
and #(rise_val, fall_val) a2(out, i1, i2);
```

```
// Rise, Fall, and Turn-off Delay Specification
```

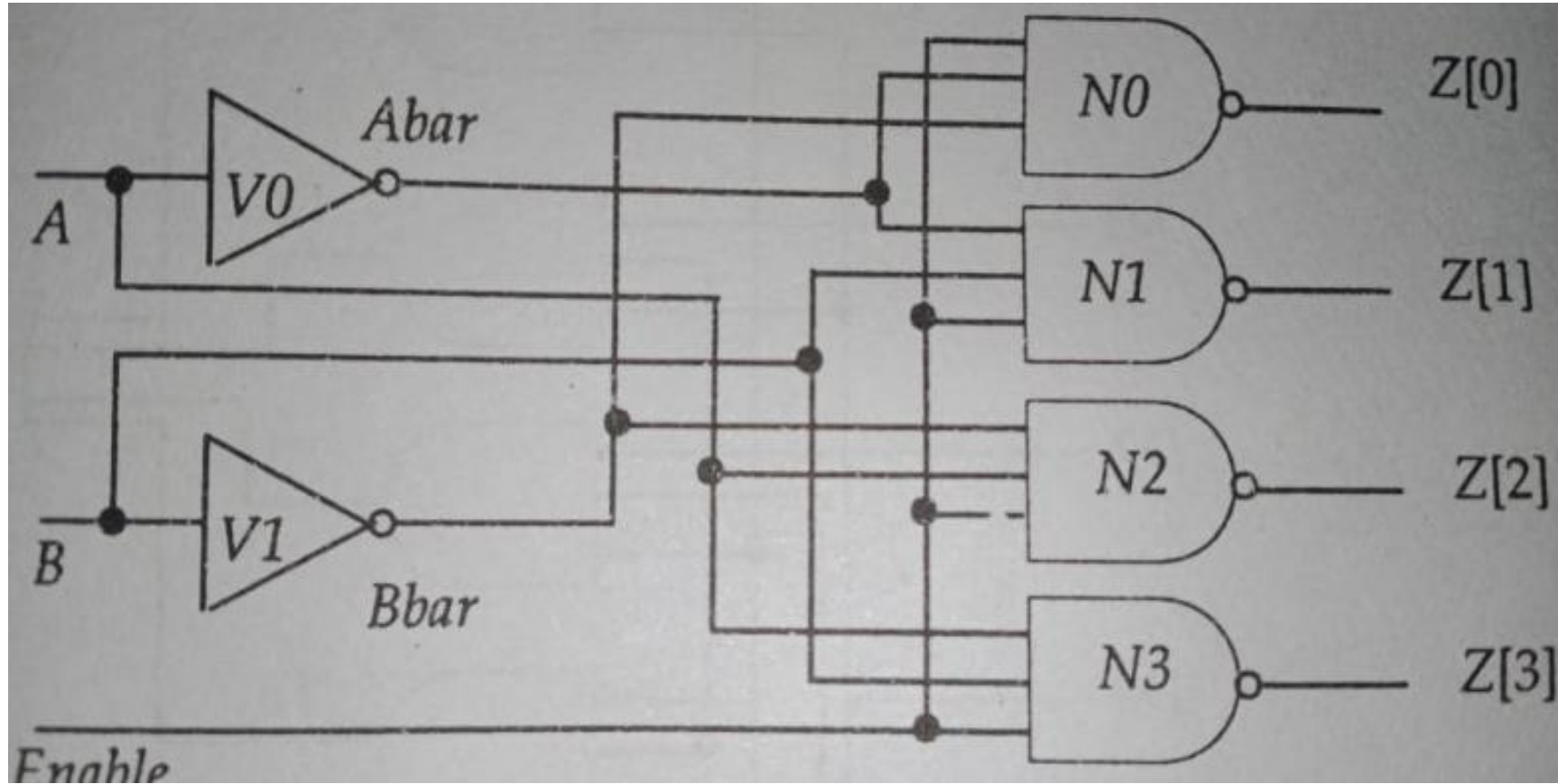
```
bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Examples of delay specification are shown below.

```
and #(5) a1(out, i1, i2); //Delay of 5 for all transitions and #(4,6) a2(out, i1, i2); // Rise  
= 4, Fall = 6
```

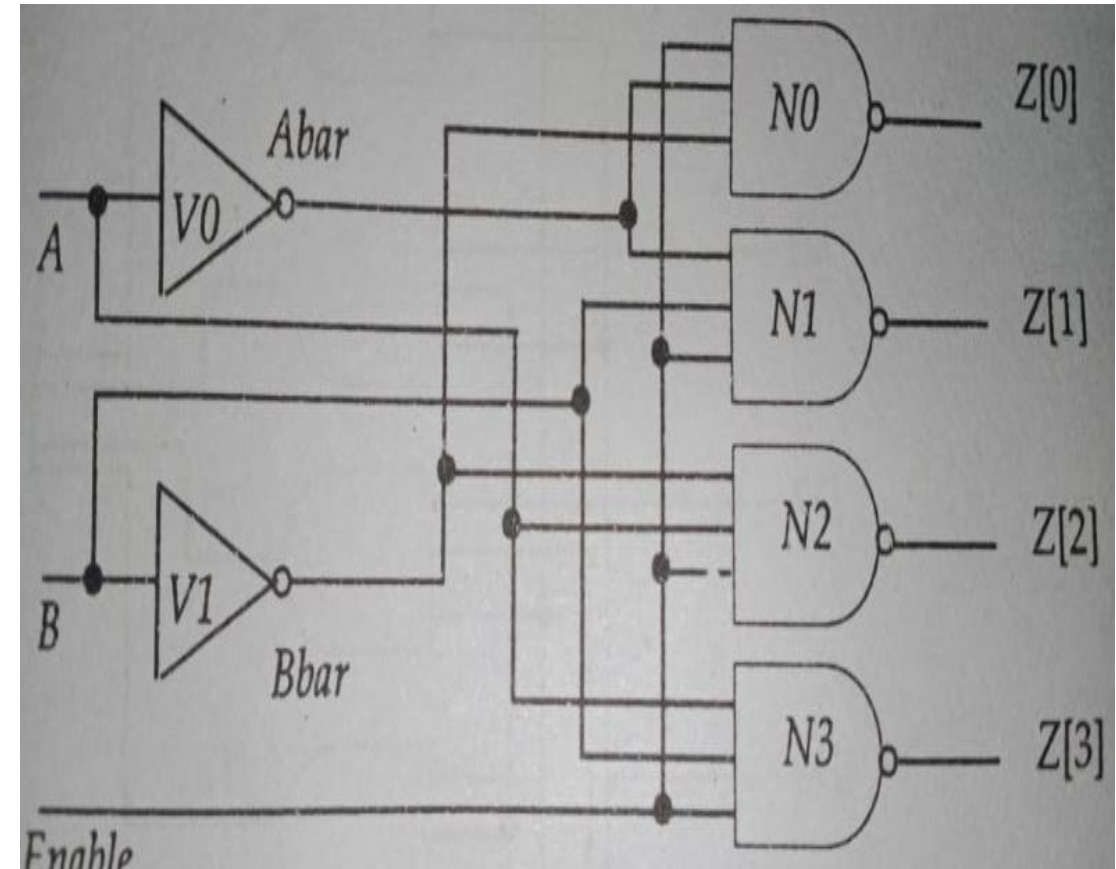
```
bufif0 #(3,4,5) b1 (out, in, control); // Rise = 3, Fall = 4, Turn-off = 5
```


2 to 4 Decoder



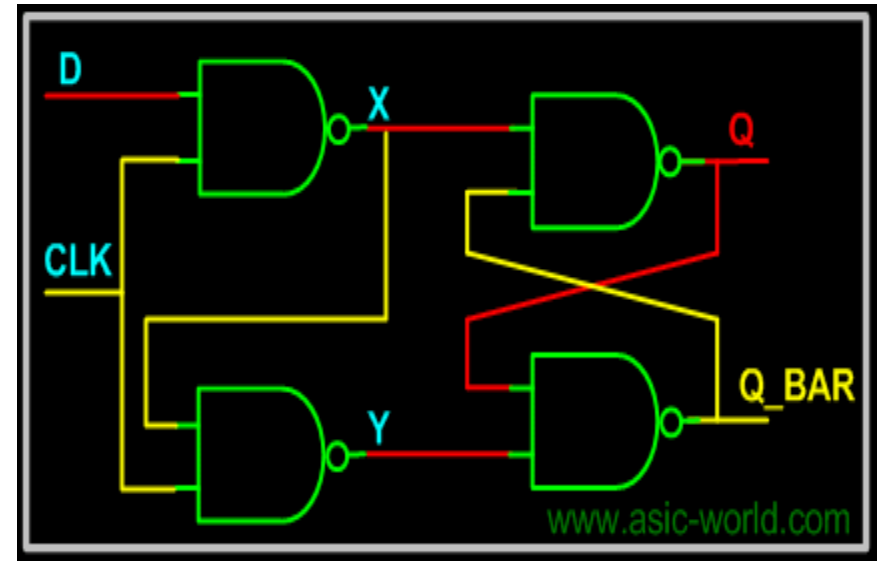
EXAMPLE :2 to 4 Decoder

```
module 2 to 4 dec (Z,A,B,Enable);  
input A,B,Enable;  
output [3:0] Z;  
wire Abar,Bbar;  
  
not V0(Abar,A);  
not V1(Bbar,B);  
  
nand N0 (Z[0],Enable,Abar,Bbar);  
nand N1 (Z[1],Enable,Abar,B);  
nand N2(Z[2],Enable,A,Bbar);  
nand N3 (Z[3],Enable,A,B);  
end module
```

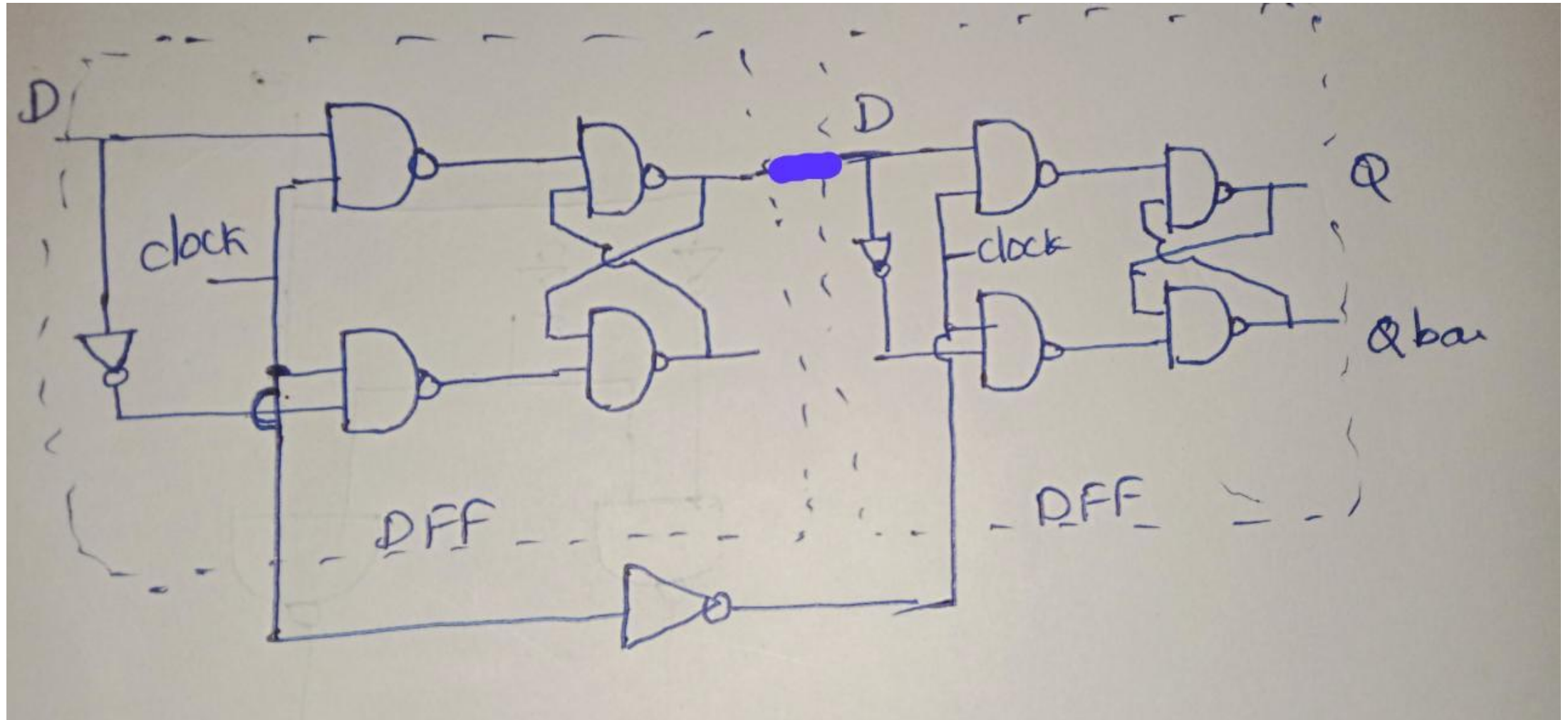


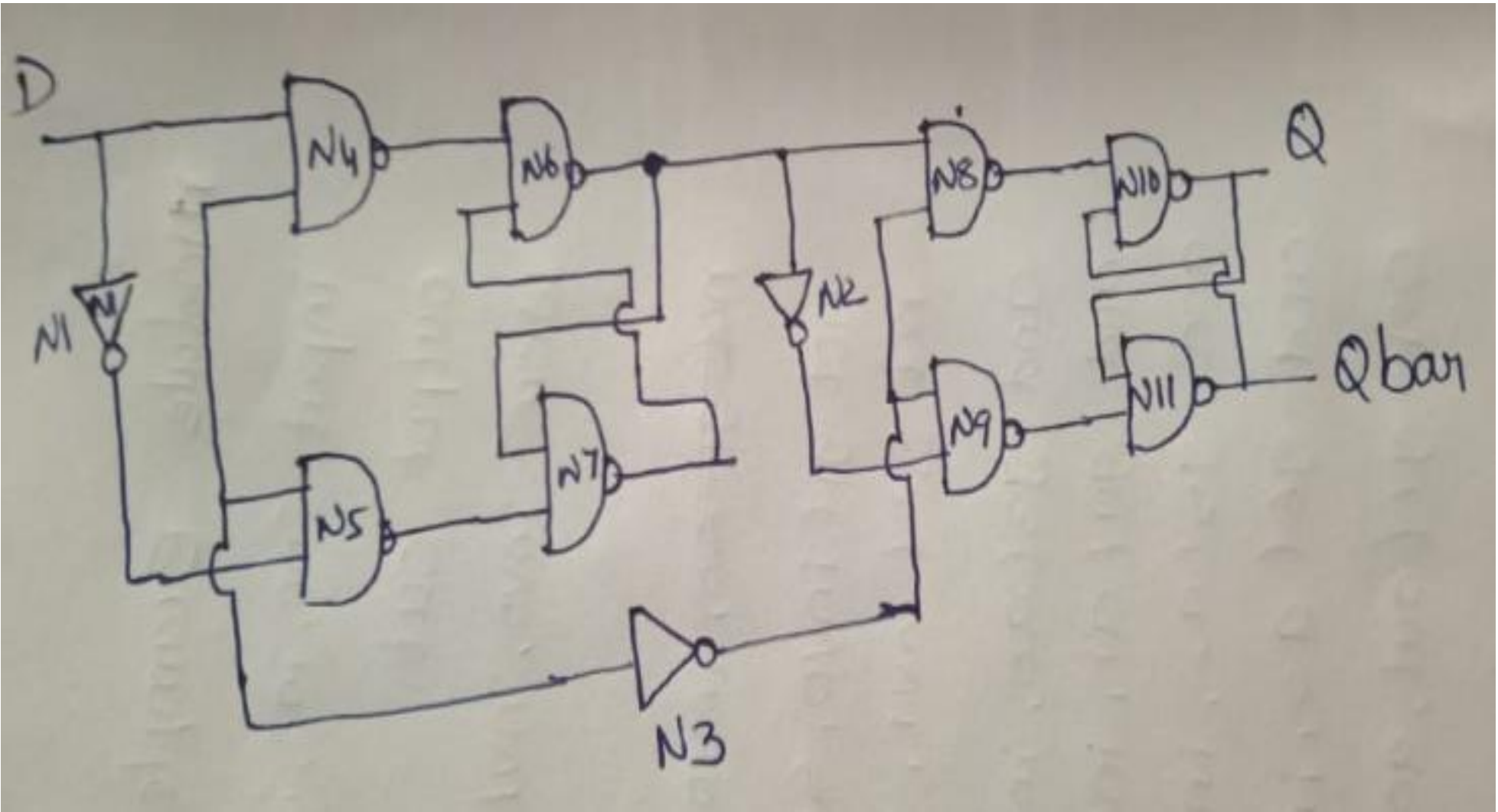
D Flip flop

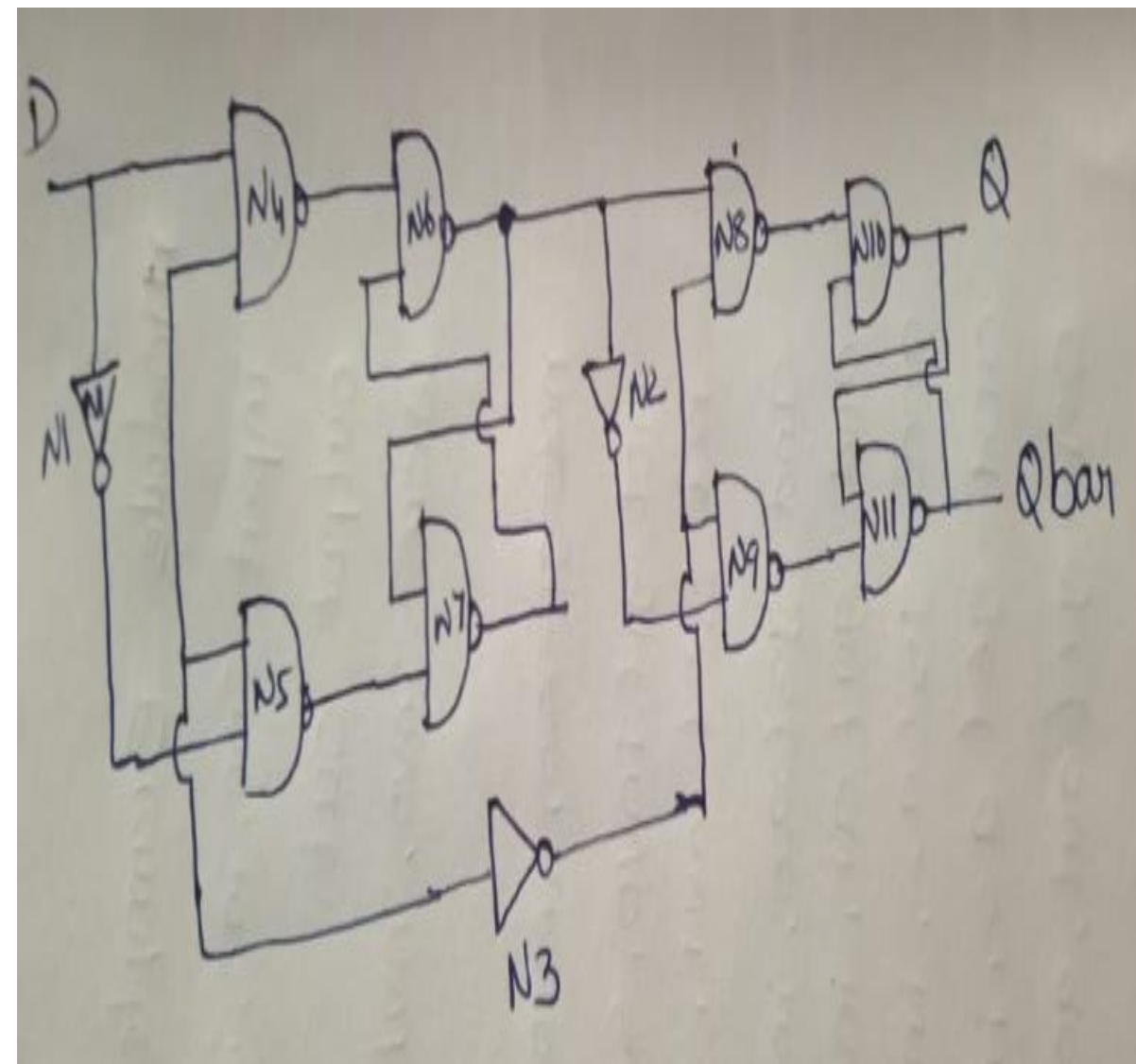
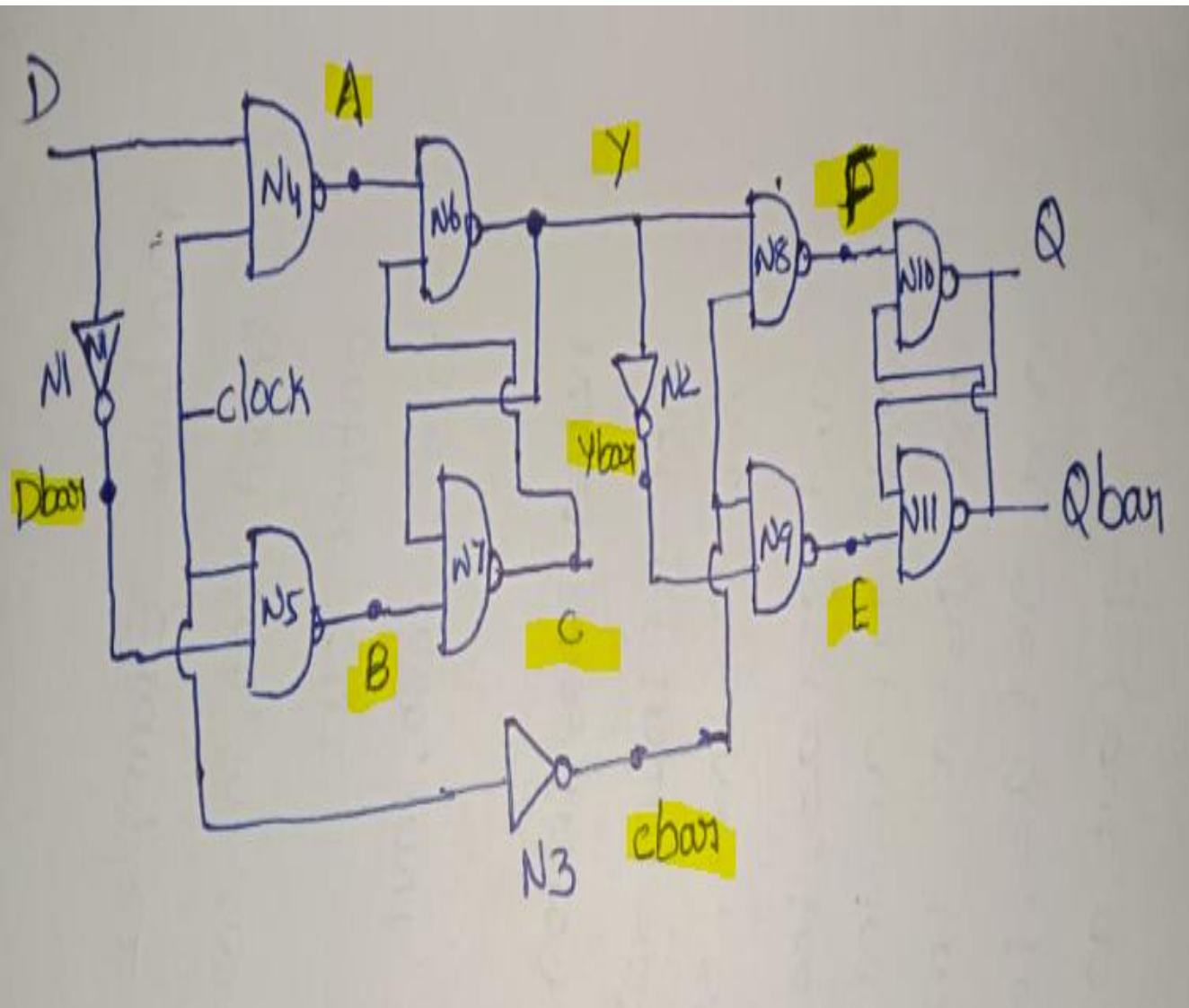
```
module dff_from_nand(Q,Q_BAR,D,CLK);  
input D,CLK;  
output Q,Q_BAR;  
wire X,Y;  
nand U1 (X,D,CLK) ;  
nand U2 (Y,X,CLK) ;  
nand U3 (Q,Q_BAR,X);  
nand U4 (Q_BAR,Q,Y);  
end module
```



MASTER SLAVE FLIP FLOP







```

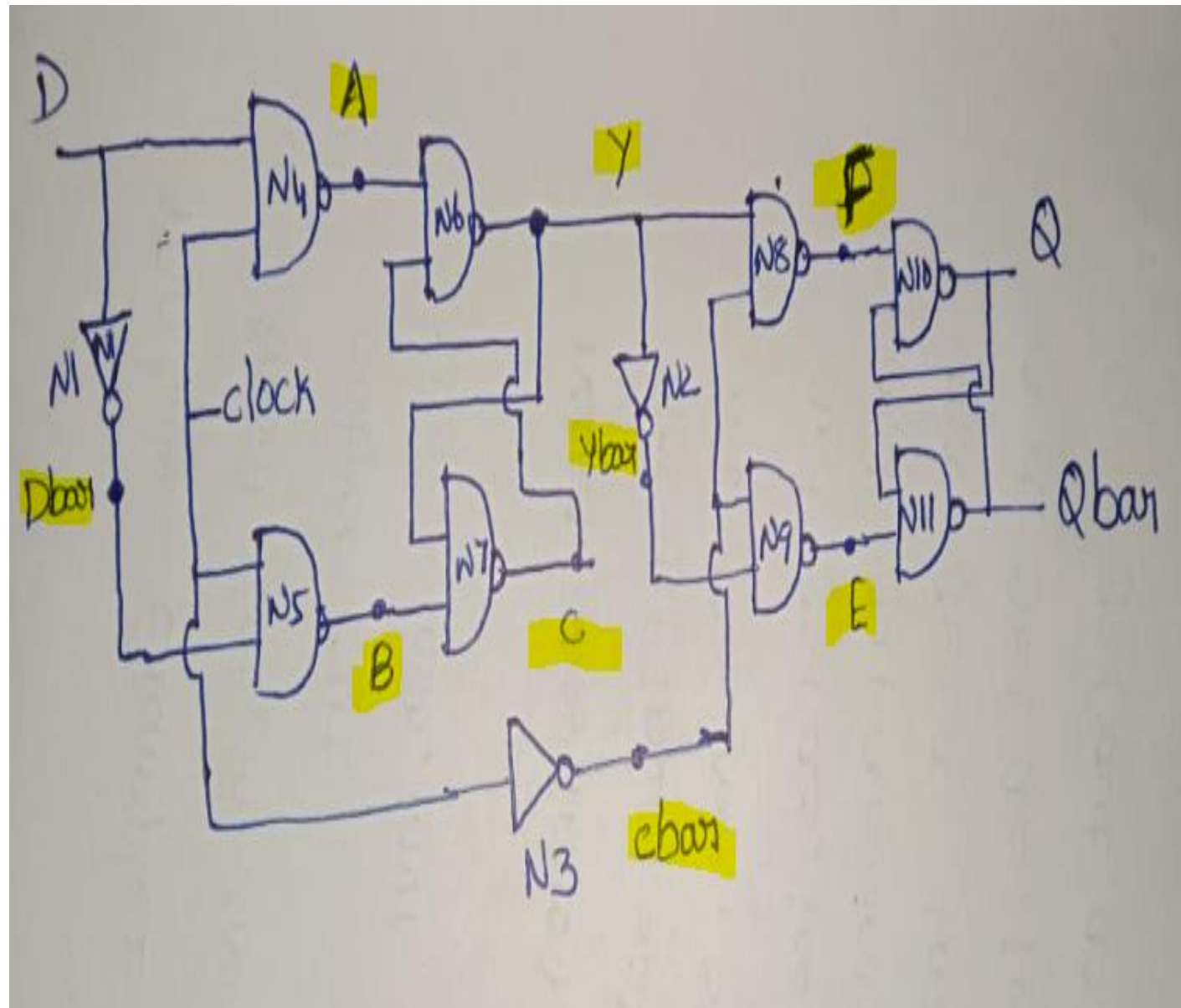
module MSFF ( Q, Qbar, D, clock);
output Q, Qbar;
input D, clock;
wire Dbar, cbar, ybar, A, B, C, E, F;

not N1 (Dbar, D);
not N2 (ybar, Y);
not N3 (cbar, clock);

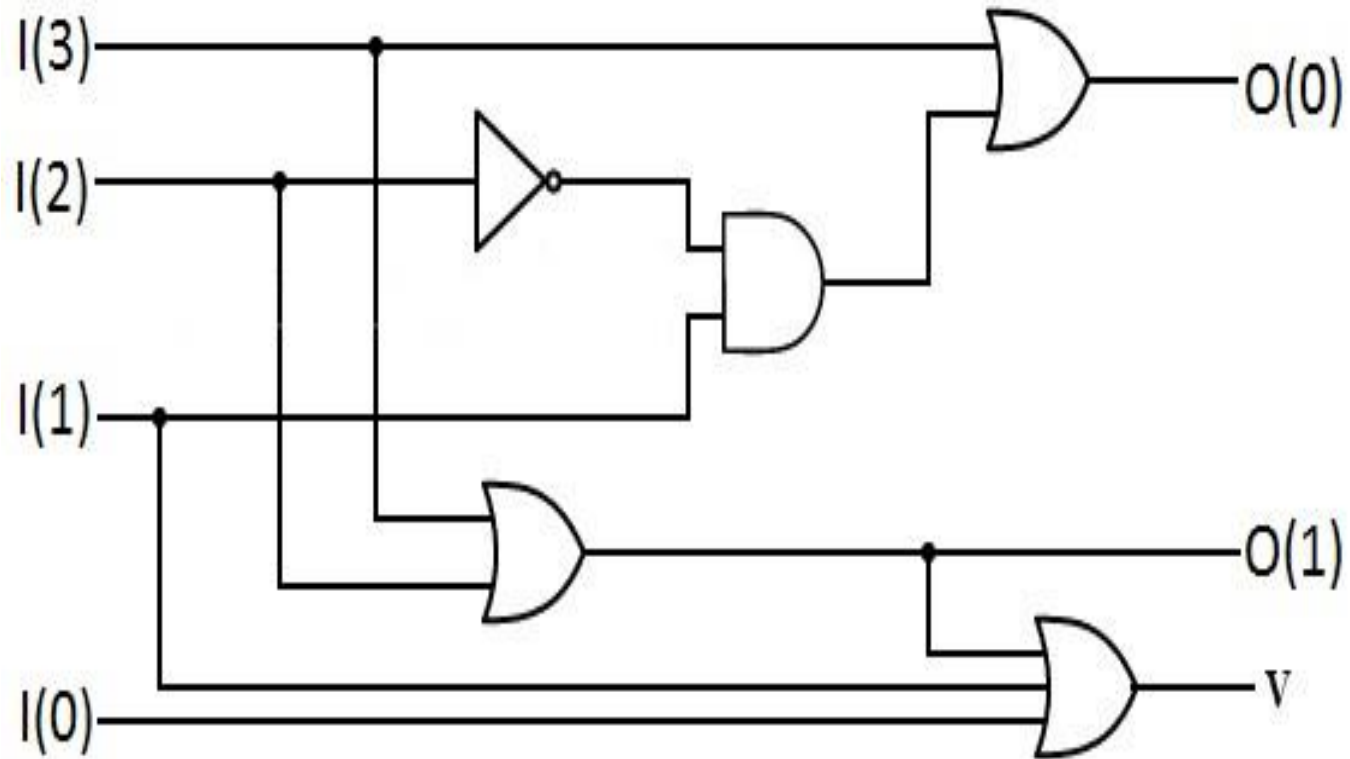
nand N4 (A, D, clock);
nand N5 (B, clock, Dbar);
nand N6 (Y, A, C);
nand N7 (C, B, Y);
nand N8 (F, Y, cbar), N9 (E, cbar, ybar);
nand N10 (Q, F, Qbar);
N11 (Qbar, E, Q);

end module

```

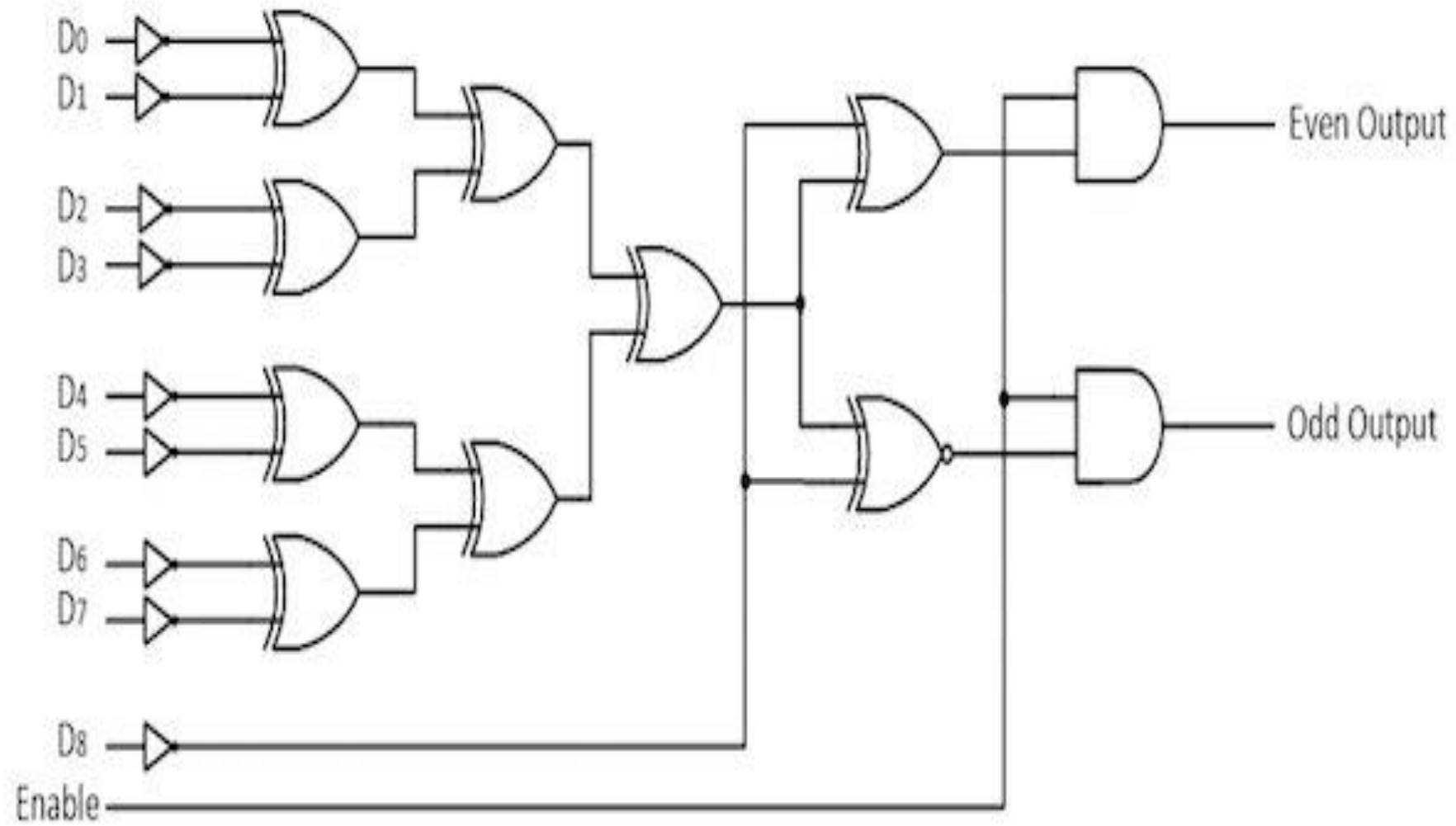


PRIORITY ENCODER



Inputs				Outputs		
D ₀	D ₁	D ₂	D ₃	Y ₁	Y ₀	V
0	0	0	0	×	×	0
1	0	0	0	0	0	1
×	1	0	0	0	1	1
×	×	1	0	1	0	1
×	×	×	1	1	1	1

PARITY GENERATOR



Implicit Nets

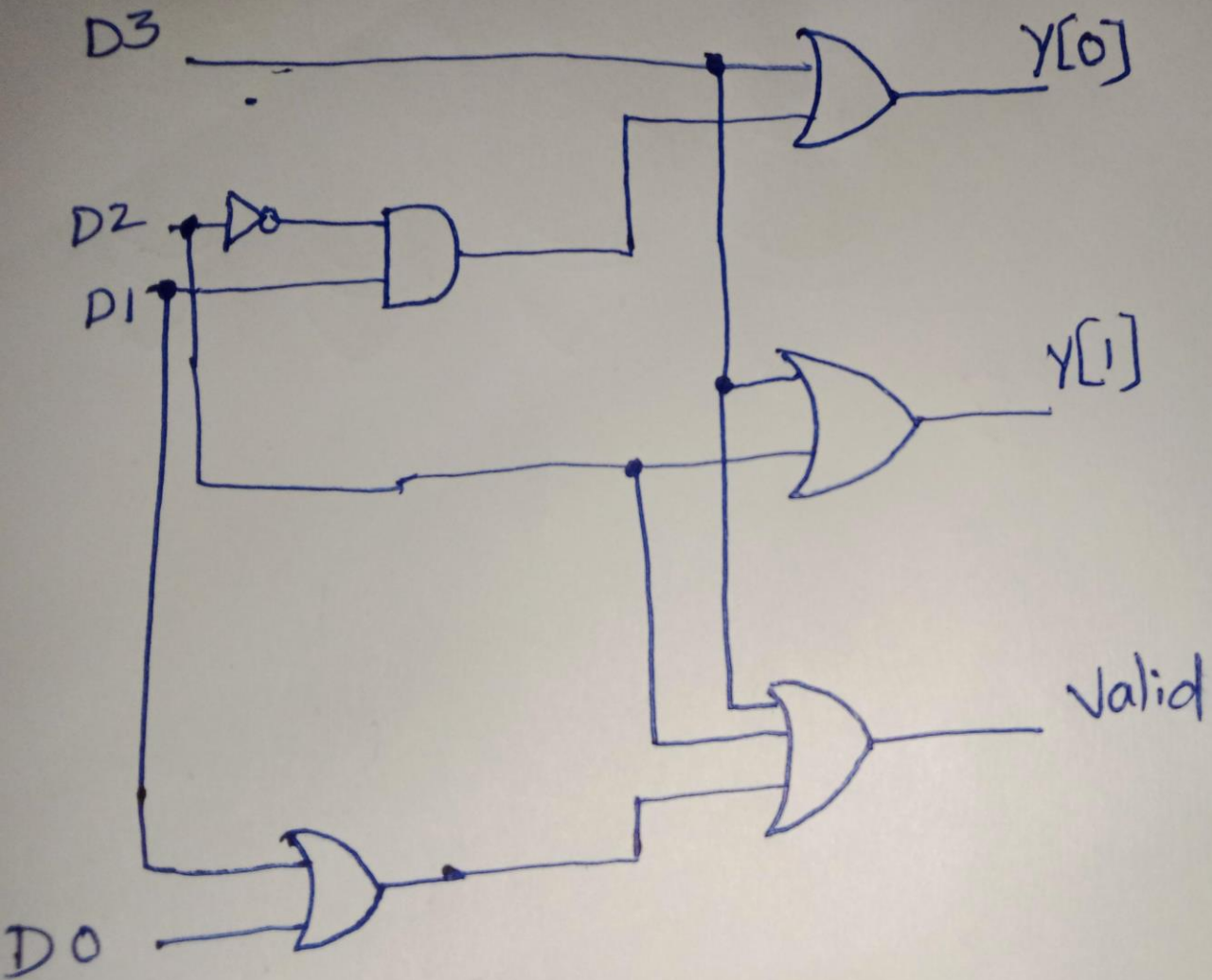
- If a net is not declared in a verilog model, by default it is implicitly declared as 1 bit wire.
- For this purpose we use compiler directive i.e `default_nettype`
- **Syntax**

```
default_nettype net_type;
```

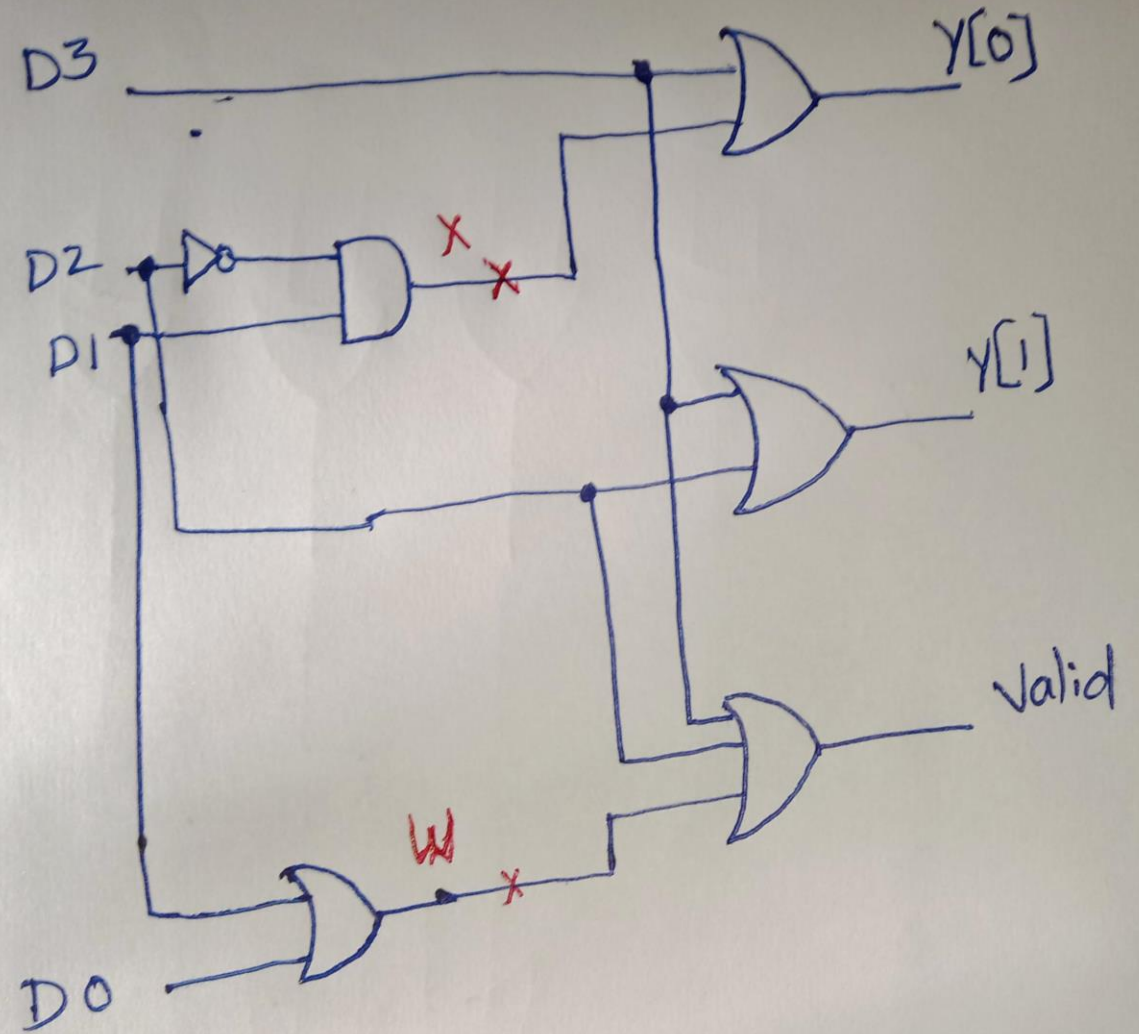
- Example

```
default_nettype wand;
```

- With the above declaration all uncledared nets are type wand.



II. Priority encoder



II. Priority encoder

```
module Pri_enc (Y, valid, D);
```

```
    output [1:0] Y;
```

```
    output valid;
```

```
    input [3:0] D;
```

```
    not g1 (D2bar, D[2]);
```

```
    and g2 (x, D2bar, D[1]);
```

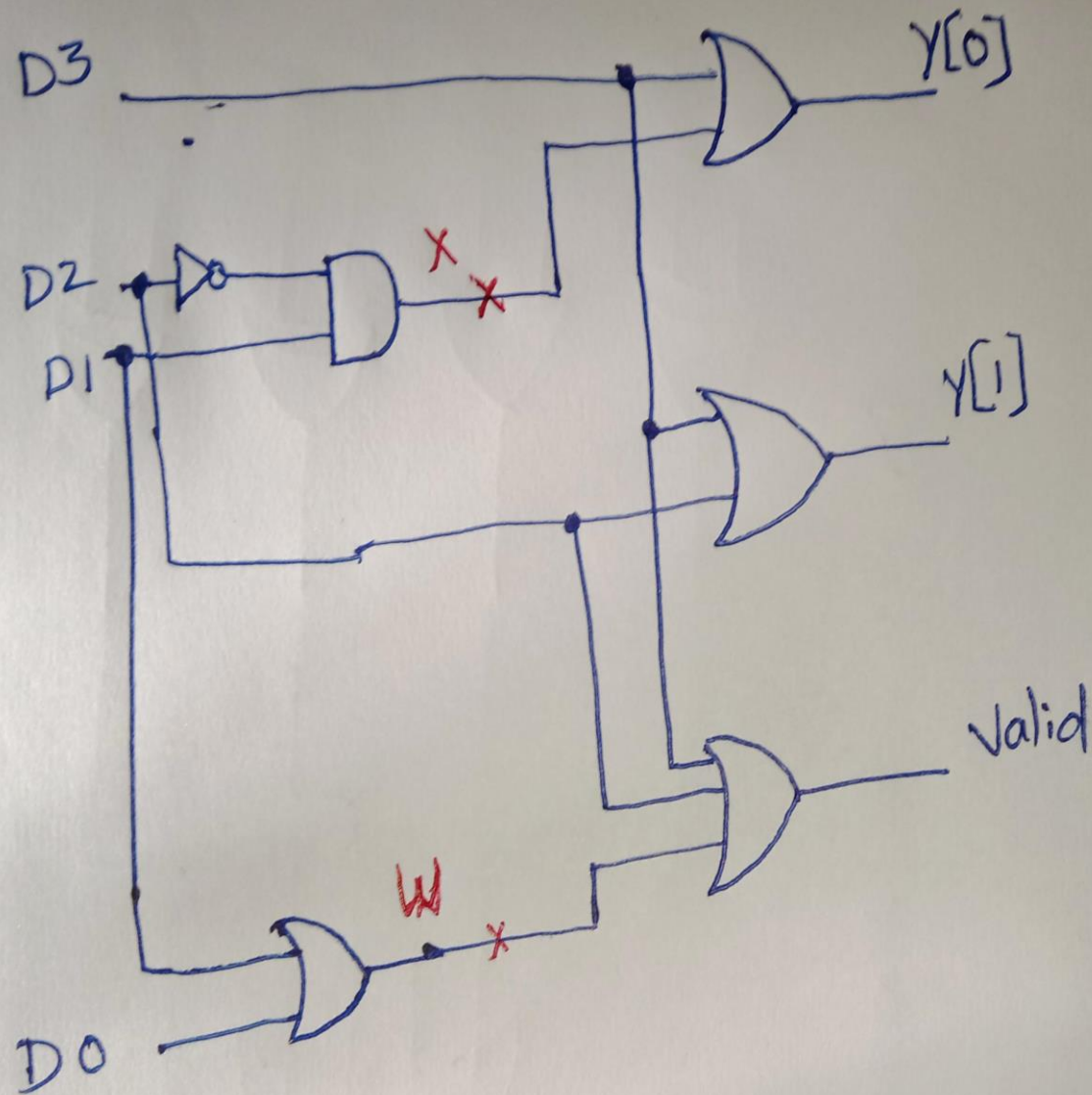
```
    or g3 (w, D[0], D[1]);
```

```
    or g4 (Y[0], D[3], x);
```

```
    or g5 (Y[1], D[3], D[2]);
```

```
    or g6 (valid, D[3], D[2], w);
```

```
end module
```



II. Priority encoder

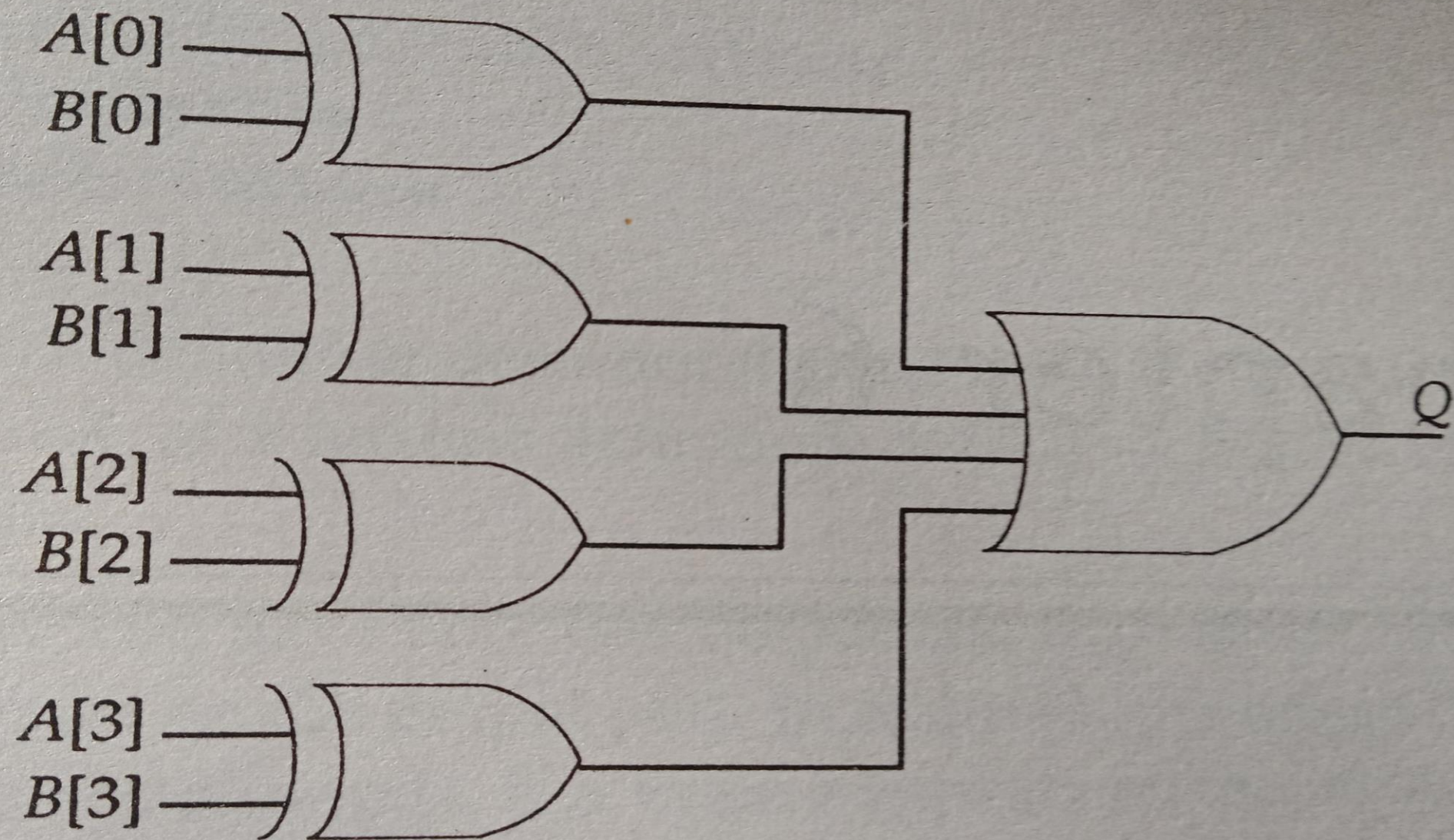


Figure 5-11 Logic for A not equals B .

Verilog code gate level

Module example (Q,A);

output Q;

input [3:0]A,B;

xor g1(w,A[0],B[0]);

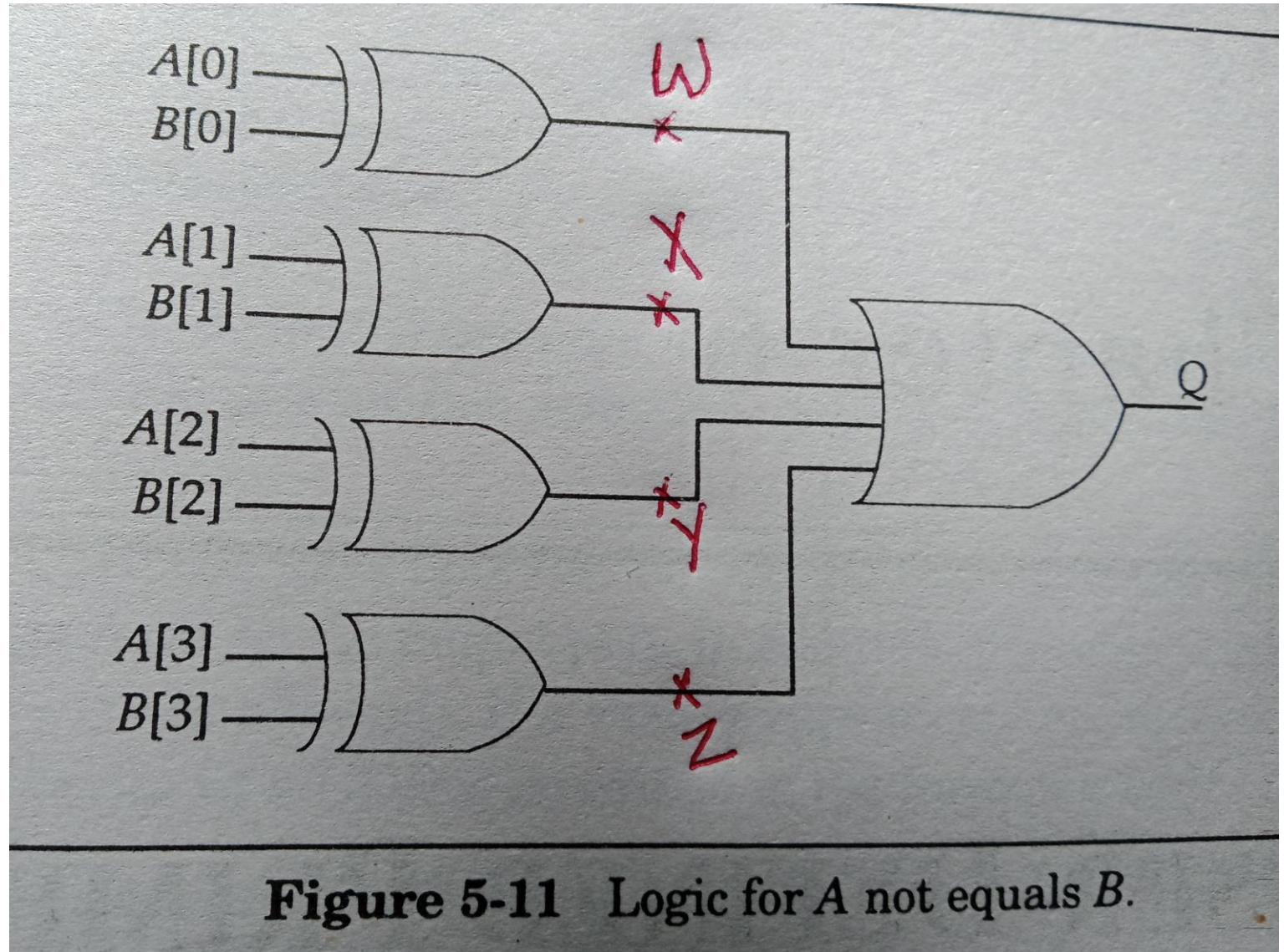
xor g2(X,A[1],B[1]);

xor g3(Y,A[2],B[2]);

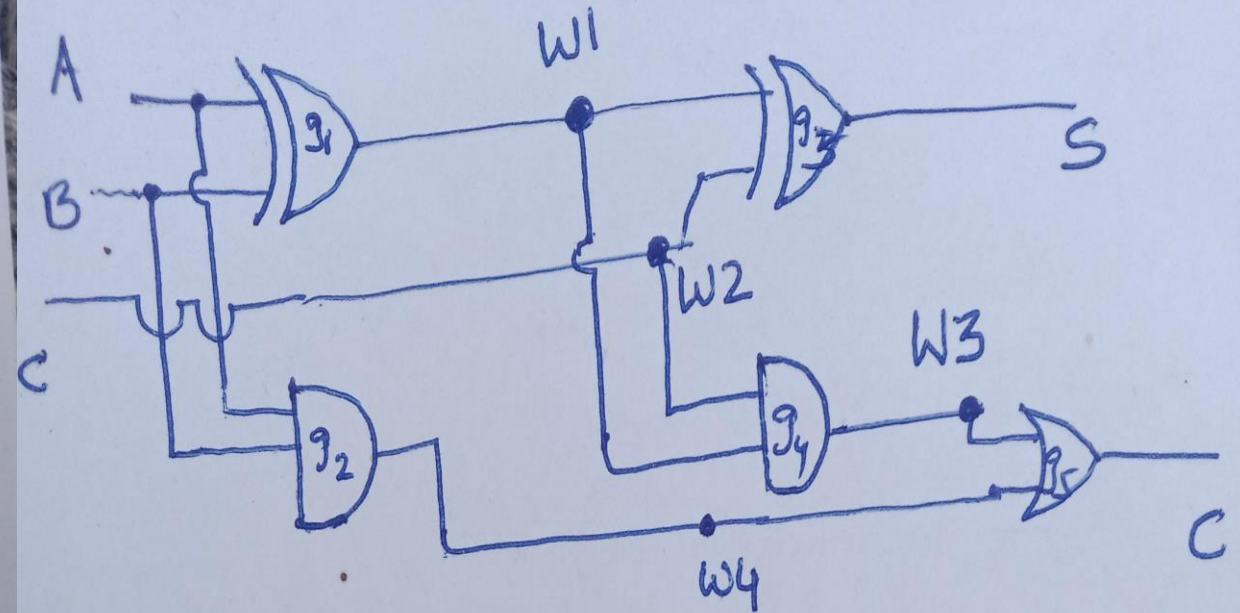
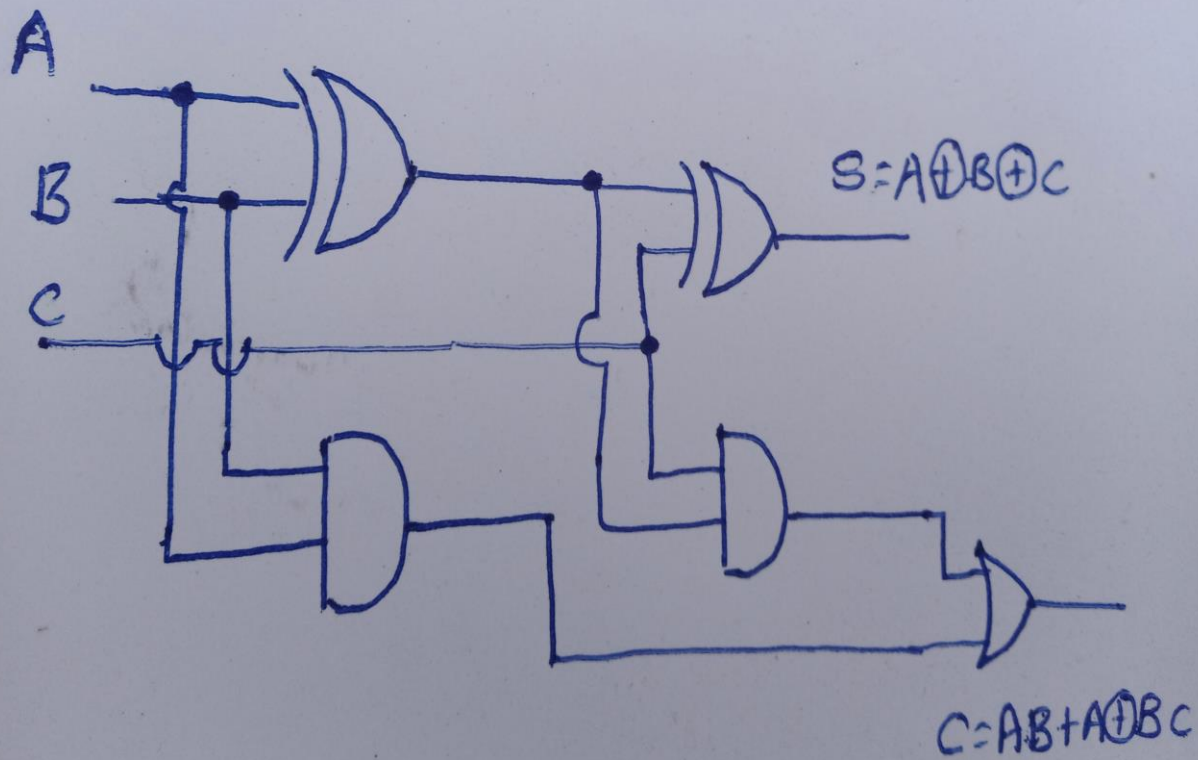
xor g4(Z,A[3],B[3]);

or g5(Q,W,X,Y,Z);

end module



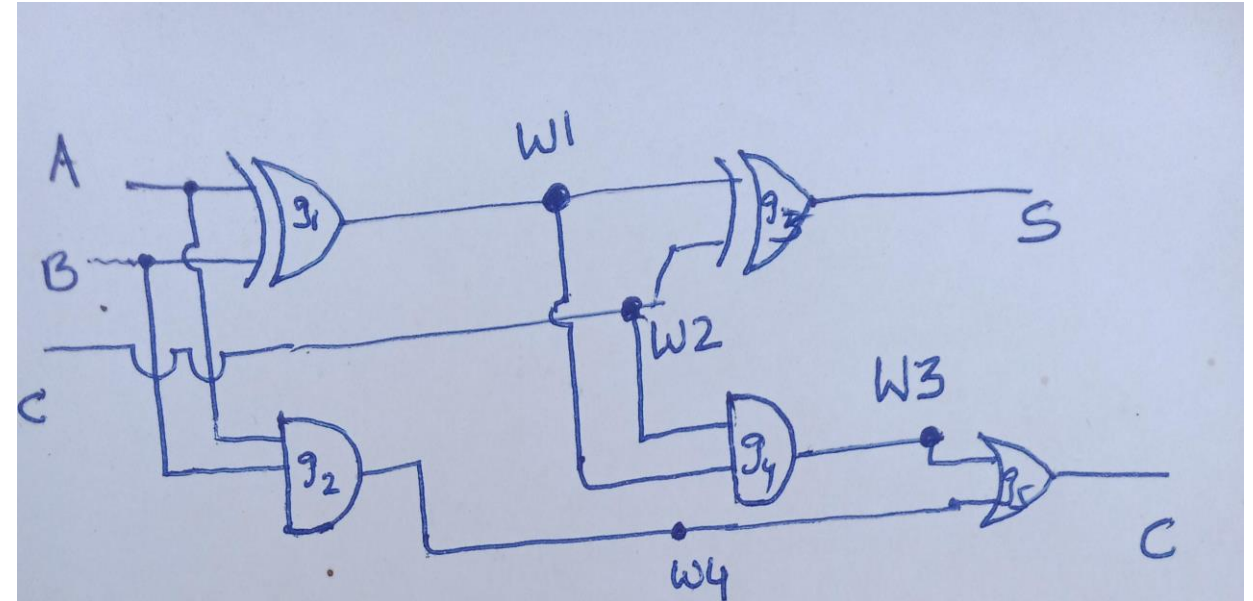
Full adder



⑥ FULL ADDER IN Gate level model

Verilog program

```
module full-adder (S, c, A, B, C);  
  input A, B, C;  
  output S, c;  
  wire w1, w2, w3, w4;  
  xor g1 (w1, A, B);  
  and g2 (w2, A, B);  
  xor g3 (S, w1, w2);  
  and g4 (w3, w2, C);  
  or (c, w3, w4);  
end module;
```



⑥ FULL ADDER IN Gate level model


```

module Example2 (out, in0, in1, in2, in3, s0, s1);
    input in0, in1, in2, in3, s0, s1;
    output out;
    wire inv0, inv1, a0, a1, a2, a3;

```

```

not g1(s0, inv0);

```

```

not g1(inv0, s0);

```

```

not g2(inv1, s1);

```

```

and g3(a0, in0, inv0, inv1);

```

```

and g4(a1, in1, inv0, s1);

```

```

and g5(a2, in2, s0, inv1);

```

```

and g6(a3, in3, s0, s1);

```

```

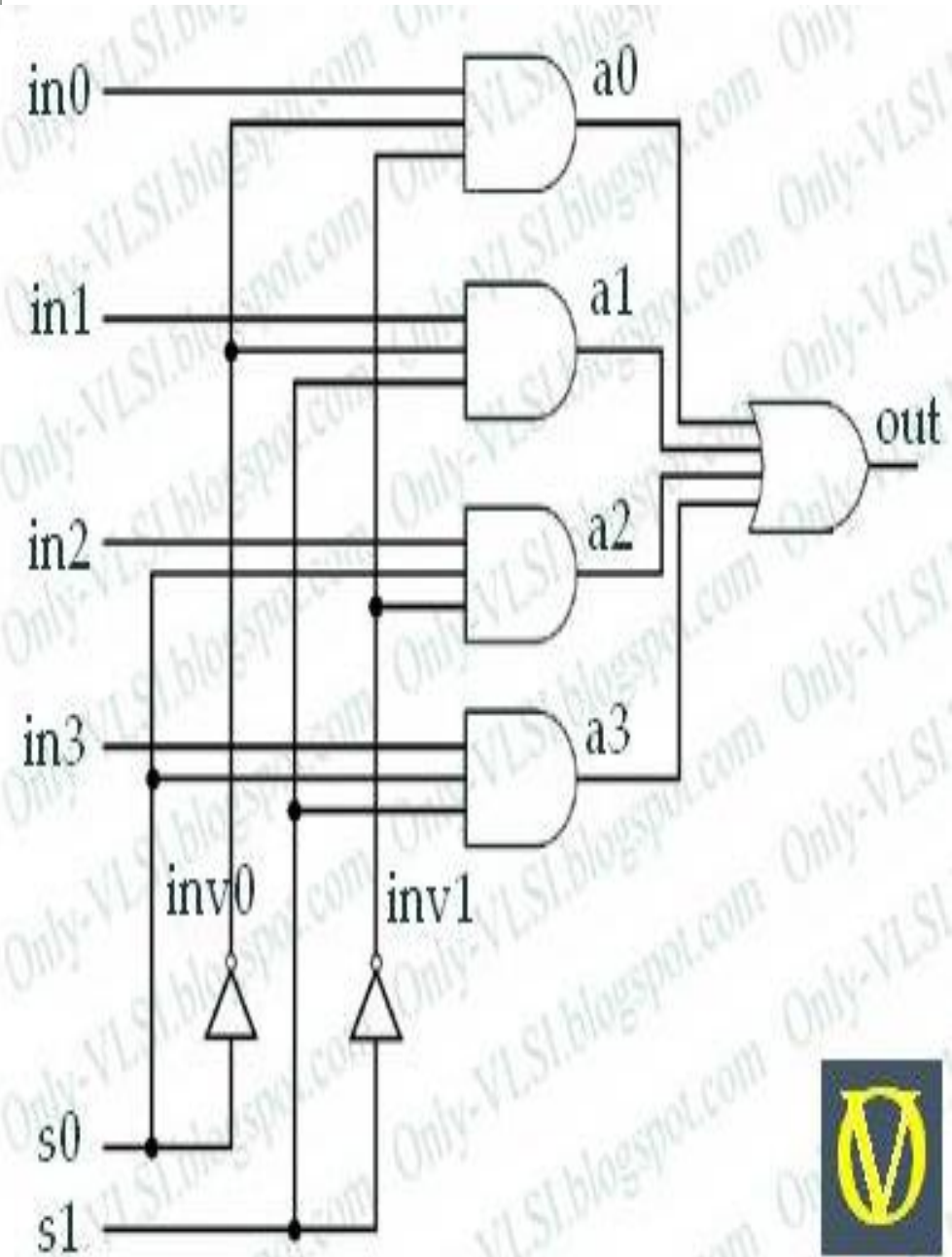
or g7(out, a0, a1, a2, a3);

```

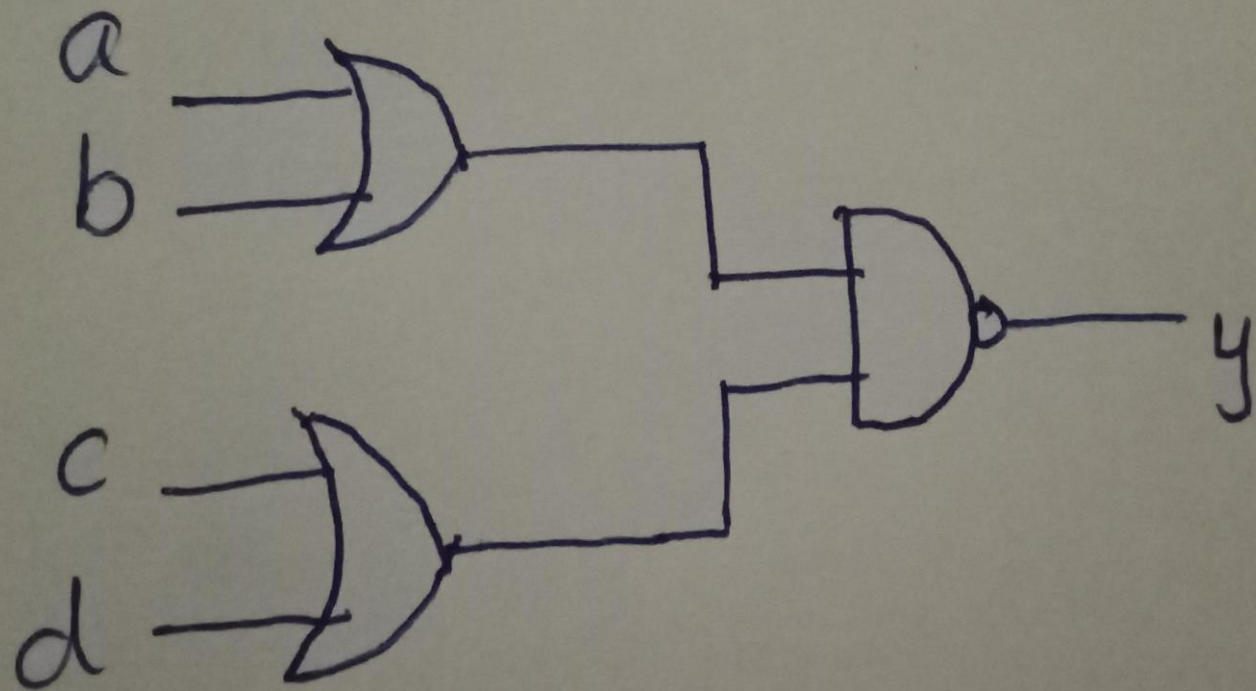
```

end module

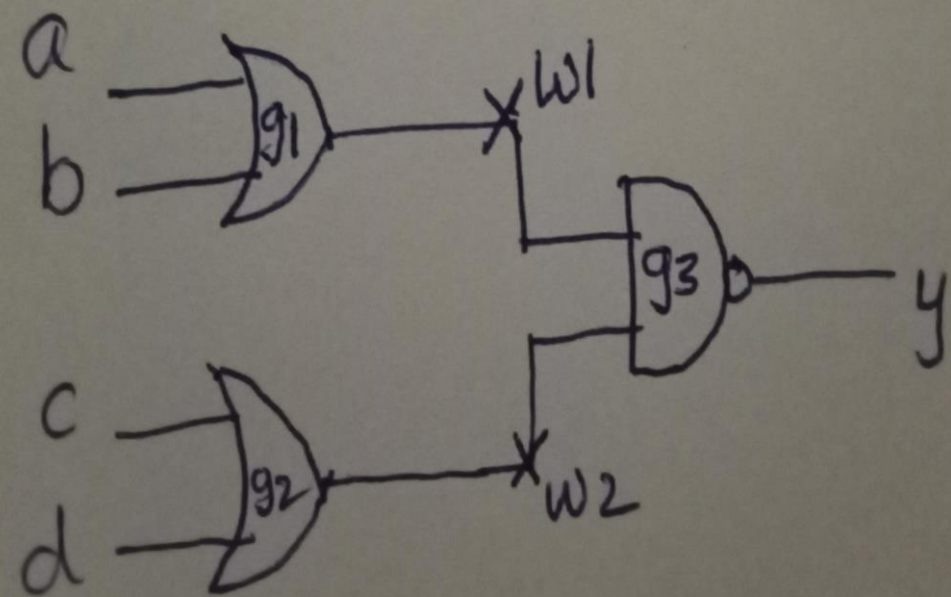
```



OR - AND - Invert



OR - AND - Invert




```
module OAIgate (y,a,b,c,d)
```

```
output y;
```

```
input a,b,c,d;
```

```
Wire w1,w2;
```

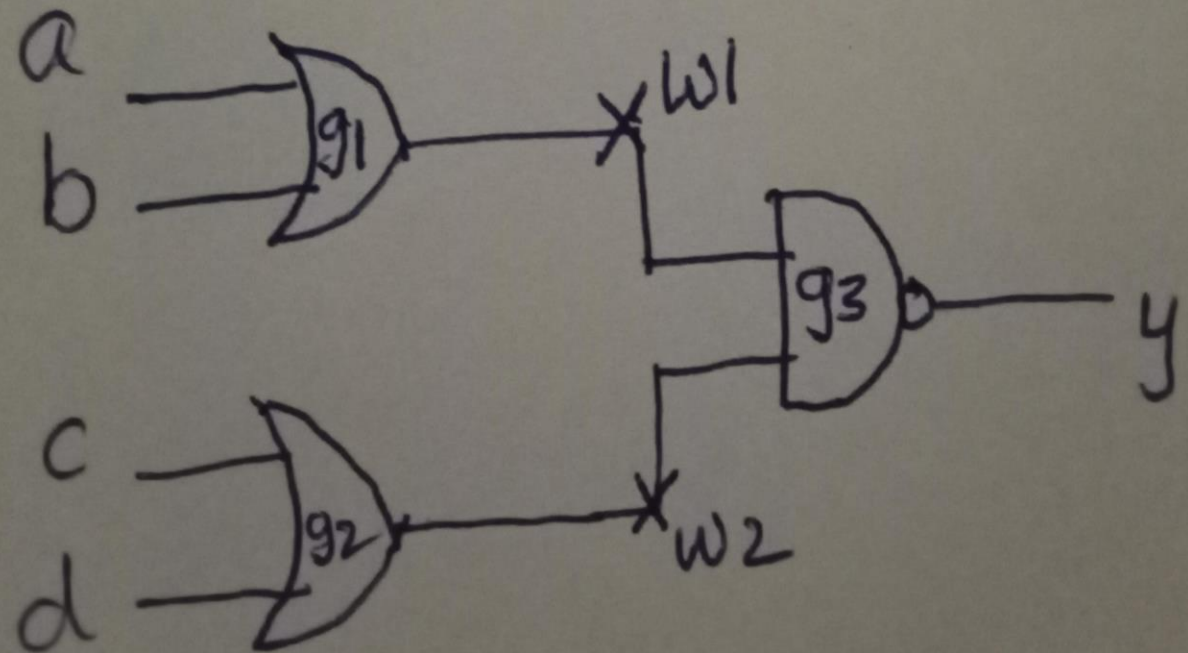
```
or g1 (w1,a,b);
```

```
or g2 (w2,c,d);
```

```
nand g3 (y,w1,w2);
```

```
end module;
```

OR-AND-Invert



Example aoi gate

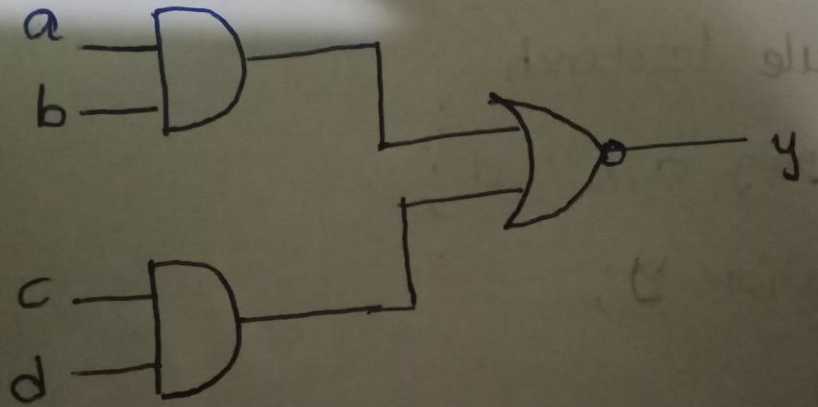
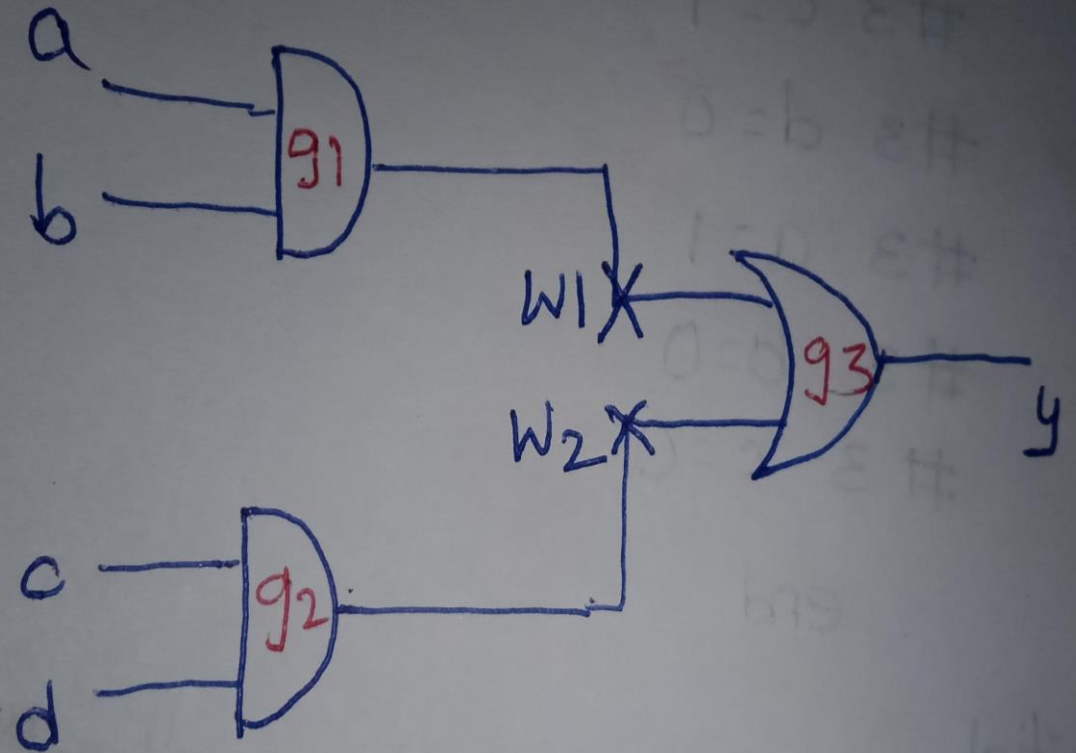


fig: a typical AOI gate



Verilog code

// Verilog code for A-O-I logic circuit

```
module aoigate (y,a,b,c,d);
```

```
input a,b,c,d;
```

```
output y;
```

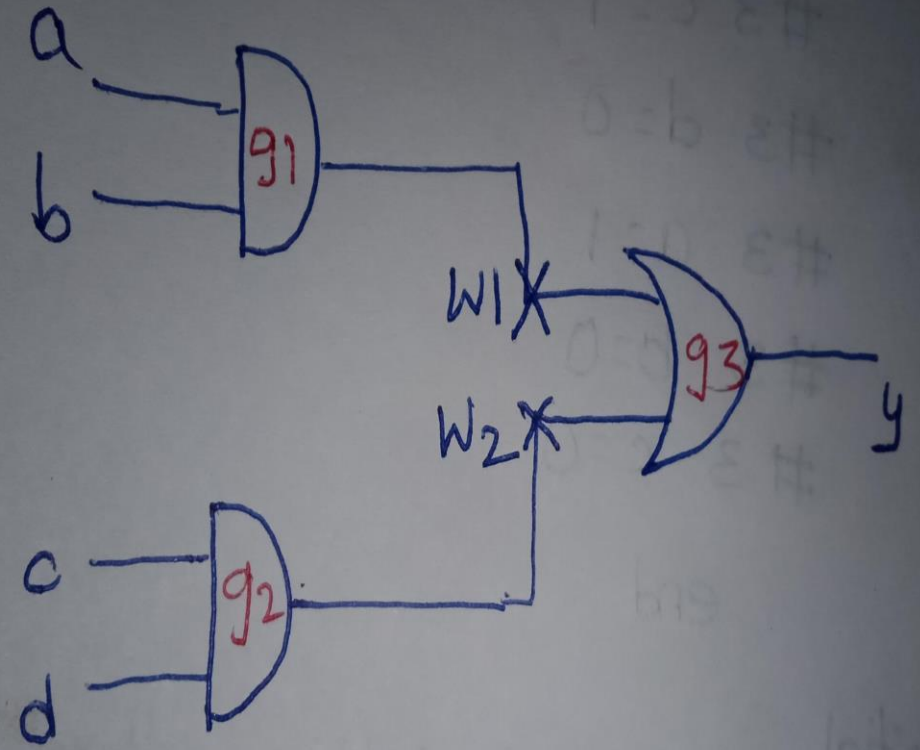
```
wire w1,w2;
```

```
and g1(w1,a,b);
```

```
and g2(w2,c,d);
```

```
nor g3(y,w1,w2);
```

```
end module;
```



Test bench for aoi gate

```
// Test bench code for aoi gate

module testbench
    reg a,b,c,d;
    wire y;

    initial
        begin
            a=0; b=0; c=0; d=0;
            #3 a=1
            #3 b=1
            #3 c=1
            #3 d=0
            #3 a=1
            #3 b=0
            #3 c=0
        end

    initial
        #100 $monitor($time, "y=%b, a=%b, b=%b, c=%b, d=%b", y,a,b,c,d);

    aoi gate gg(y,a,b,c,d);
end module
```

	a	b	c	d
	0	0	0	0
#3	1	0	0	0
#3	1	1	0	0
	1	1	1	0
	1	1	1	1
	1	1	1	0
	1	0	1	1
	1	0	0	