

UNIT - 2

GATE LEVEL MODELLING

AND GATE PRIMITIVE

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

```
and g1 (0, 11, 12, ----, 1n);
```

The AND module has only one output. The first port in the argument list is the output port.

The AND gate instantiation can take any number - 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.

MODULE STRUCTURE

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 of any. All the variables in the ports-list are to be identified as inputs, outputs, or inouts.

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.

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 element / statement in any module definition is the keyword "endmodule". Comments can appear anywhere in the module definition.

OTHER GATE PRIMITIVES.

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

Basic gate primitives in Verilog with details.

Gate	Mode of instantiation	Output port(s)	Input port(s)
AND	and ga (0, i ₁ , i ₂ , ---- i ₈);	0	i ₁ , i ₂ , ----
OR	or gr (0, i ₁ , i ₂ , ---- i ₈);	0	i ₁ , i ₂ , ----
NAND	nand gna (0, i ₁ , i ₂ , ---- i ₈);	0	i ₁ , i ₂ , ----
NOR	nor gnr (0, i ₁ , i ₂ , ---- i ₈);	0	i ₁ , i ₂ , ----
XOR	xor gxr (0, i ₁ , i ₂ , ---- i ₈);	0	i ₁ , i ₂ , ----
XNOR	xnor gxn (0, i ₁ , i ₂ , ---- i ₈);	0	i ₁ , i ₂ , ----
BUF	buf gb (0 ₁ , 0 ₂ , --- i);	0 ₁ , 0 ₂ , 0 ₃ , ---	i
NOT	not gn (0 ₁ , 0 ₂ , 0 ₃ , ---- i);	0 ₁ , 0 ₂ , 0 ₃ , ---	i

In all cases of instantiations, one need not necessarily assign a name to all instantiation. It need be done only when felt necessary - say for clarity of circuit description.

→ In all 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 input each. They 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.

Rules for deciding the output values of gate primitives for different input combinations.

(3)

Type of gate	0 Output state	1 Output state	x Output state
AND	Any one of the inputs is zero	All the inputs are at one	All other cases
NAND	All the inputs are at one.	Any one of the inputs is zero.	
OR	All the inputs are zero	Any one of the inputs is one.	
NOR	Any one of the inputs is one	All the inputs are at zero	
XOR	If every one of the inputs to defines at zero or one, the output is zero or one as decided by the XOR or XNOR function.		If any one of the inputs is at x or z state, the output is at x state.
XNOR			
BUF	If the only input is at 0 state	If the only input is at 1 state	All other cases of inputs
NOT	If the only input is at 1 state	If the only input is at 0 state	

Example programs

EX 2-to-4 Decoder

```
module dec2_4 (a, b, en);
```

```
    output [3:0] a; input [1:0] b; input en; wire [1:0] bb;
    not(bb[1], b[1]), (bb[0], b[0]); and(a[0], en, bb[1], bb[0]), (a[1], en, bb[1],
    b[0]), (a[2], en, b[1], bb[0]), (a[3], en, b[1], b[0]);
```

```
end module;
```

```
// test bench module
```

```
tst_dec2_4();
```

```
    wire [3:0] a;
```

```
    reg [1:0] b; reg en;
```

dec 2_4 dec(a, b, en);

2)

initial

begin

{b, en} = 3'b000;

#2 {b, en} = 3'b001;

#2 {b, en} = 3'b011;

#2 {b, en} = 3'b101;

#2 {b, en} = 3'b111;

end

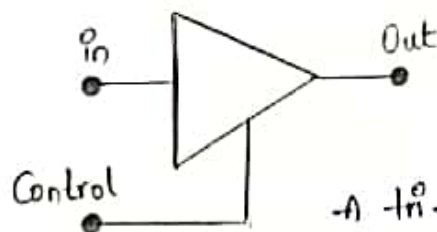
initial

\$monitor(\$time, "Output a = %b, Input b = %b", a, b);

endmodule

TRI-STATE GATES

-A tri-state buffer



-A tri-state buffer

four types of tri-state buffers are available in Verilog as primitive. Their outputs can be turned ON or OFF by a Control signal.

The direct buffer is instantiated as

Buf1 in (out, in, Control);

The symbol of the buffer is shown in figure

Out as the single output variable

in as the single input variable and

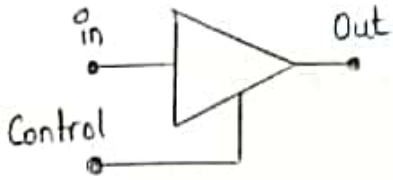
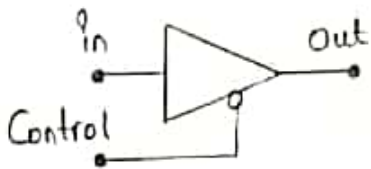
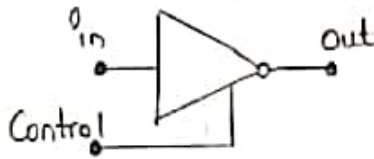
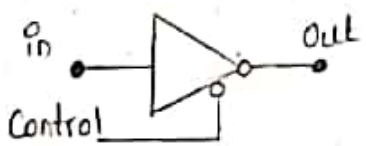
Control as the single control signal variable.

when Control = 1, Out = in

when Control = 0, Out 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 built-in as well as the other tri-state type primitives are shown in table. In all the cases in table, out is the output, in is the input, and control, the Control Variable.

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

The following Observations are Common to all the tri-state buffer primitives!

→ If the Control signal has a Value that Corresponds to the Control signal being on, two possibilities exist:

The output has the same Value as the input if the input is 0 or 1

The output is at x otherwise (i.e, if the input is x or z).

→ If the Control signal has a value that Corresponds to the Control signal being off, the Output is at z

state irrespective of the value of the input. (5)

→ if the control signal at x or z , three possibilities arise;

- if the input is at x or z , the output is at x .
- if the input is at 0 state, the output is L for buf1f1 and buf1f0 . It is at +1 for not1f1 and not1f0 .
- If the input is at 1 state, the output is +1 for buf1f1 and buf1f0 . It is at L for not1f1 and not1f0 .

Note that +1 corresponds to 1 or z state while L corresponds to 0 or z state.

ARRAY OF INSTANCES OF PRIMITIVES

The primitives available in verilog can also be instantiation 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:

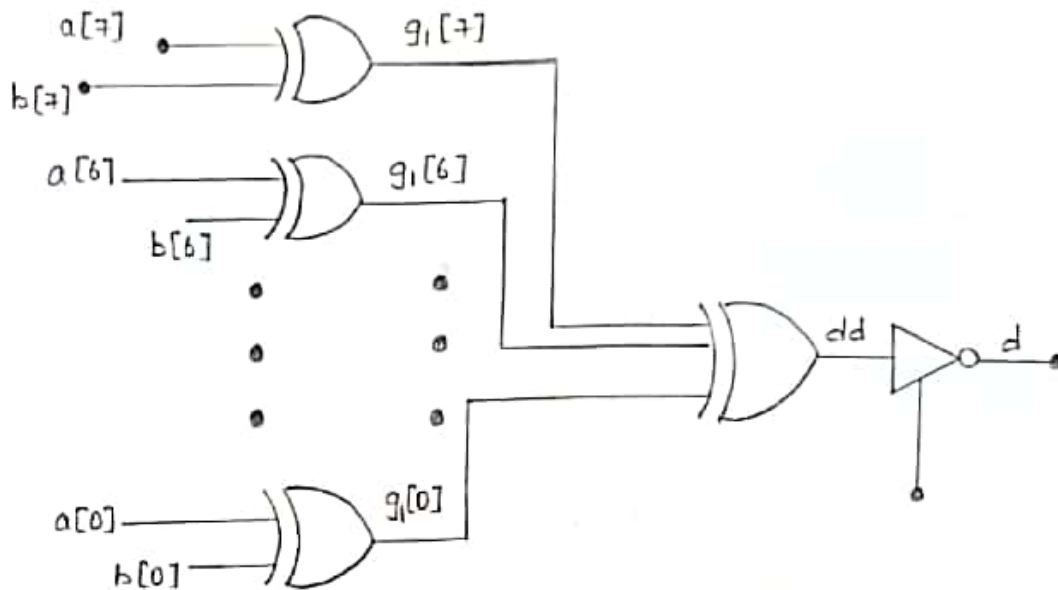
and gate [7] (a[3], b[3], c[3]), gate [6] (a[2], b[2], c[2]), gate [5] (a[1], b[1], c[1]), gate [4] (a[0], b[0], c[0]);

The assignment of different bits of input vectors to respective gate 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.

Example : Byte Comparator



```

module Comp (d, a, b, en);
    input en;
    input [7:0] a, b;
    output d;
    wire [7:0] c;
    wire dd;

    xor g1 [7:0] (c, b, a);
    or (dd, c);
    not #10 (d, dd, en);

endmodule

```

Test Bench for Comparator module

```

Comp_tb;
reg [7:0] a, b; reg en; Comp
gg (d, a, b, en);
initial begin a = 8'h00; b = 8'h00
en = 1'b0
and
always
#2 en = 1'b1;

```

(8)

```
#2b = b + 2'd2; end
```

```
initial $monitor($time, "en = %b, a = %b, b = %b,  
d = %b" en, a, b, d); initial #30 $
```

```
stop;  
end module
```

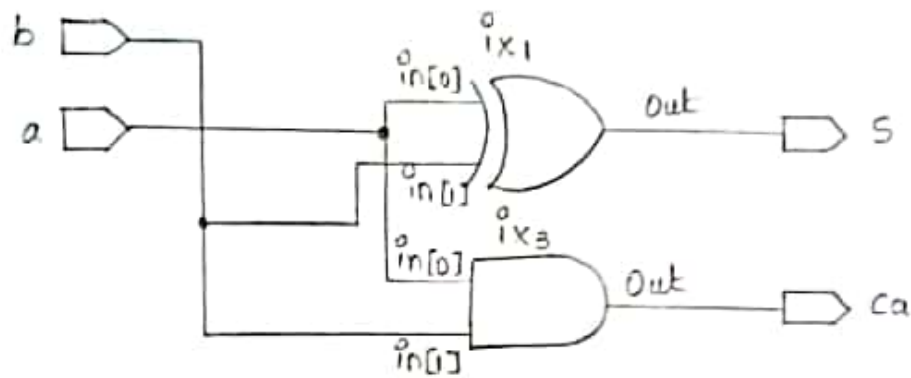
half adder

```
module ha(s, ca, a, b);  
input a, b;  
Output s, ca;  
xor(s, a, b); and(ca, a, b);  
end module
```

```
// test-bench  
module tstha();  
reg a, b; wire s, ca;  
ha hh(s, ca, a, b);  
initial begin  
a = 0; b = 0;  
end
```

```
always begin  
#2 a = 1; b = 0;  
#2 a = 0; b = 1;  
#2 a = 1; b = 1;  
#2 a = 0; b = 0; end
```

```
initial $monitor($time, " a = %b, b = %b, out carry = %b,  
outsum = %b", a, b, ca, s);  
initial #24 $stop  
end module.
```



FULL ADDER

module

-fa (sum, Cout, a, b, cin);

input a, b, cin

Output sum, Cout;

wire s, c1, c2;

ha ha1(s, c1, a, b), ha2(sum, c2, s, cin); or(Cout, c2, c1)

end module

// test-bench module

test_fa();

reg a, b, cin;

-fa ff (sum, Cout, a, b, cin);

initial begin a=0; b=0; cin=0;

end

always begin

#2 a=1; b=1; cin=0; #2 a=1; b=0; cin=1;

#2 a=1; b=1; cin=1; #2 a=1; b=0; cin=0;

#2 a=0; b=0; cin=0; #2 a=0; b=1; cin=0;

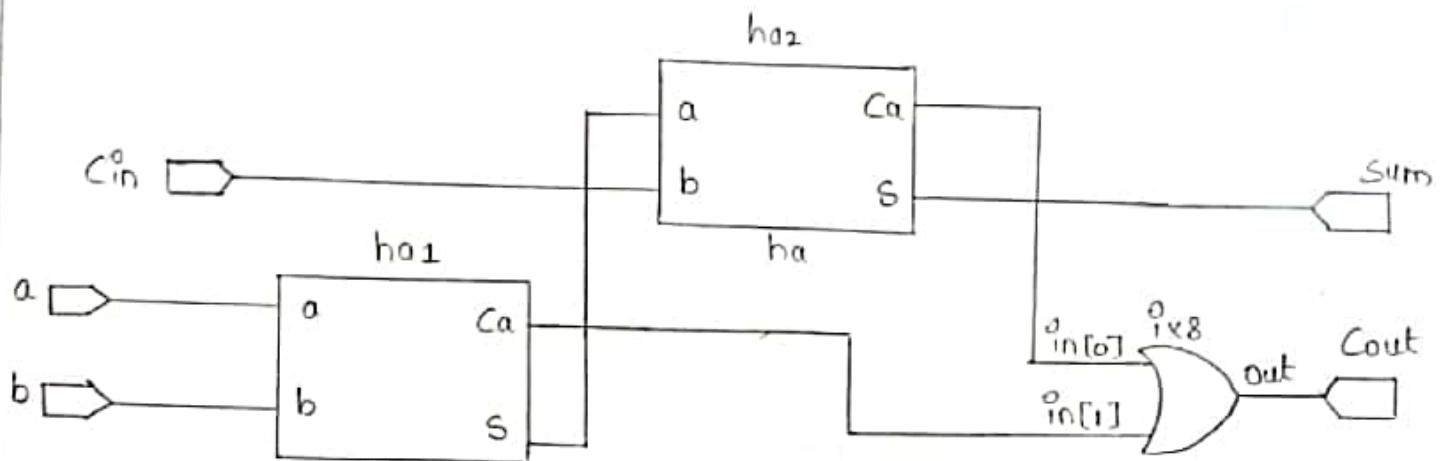
#2 a=0; b=0; cin=0; #2 a=0; b=1; cin=1;

#2 a=1; b=0; cin=0; #2 a=1; b=1; cin=0;

#2 a=0; b=1; cin=0; #2 a=1; b=1; cin=1;

end

initial \$ monitor (\$time, "a = %b, b = %b, Cin = %b, outsum = %b,
 outcar = %b", a, b, Cin, Sum, Cout); initial #30 \$ stop; (10)
 end module



Mux

```
module mux4_1 (y, i, s); input [3:0] i; input
[1:0] s; output y; wire [1:0] ss; wire
[3:0] yy; not
(ss[0], s[0]), (ss[1], s[1]); and
(yy[0], i[0], ss[0], ss[1]); and
(yy[1], i[1], s[0], ss[1]); and
(yy[2], i[2], ss[0], s[1]); and
(yy[3], i[3], s[0], s[1]); or
(y, yy[3], yy[2], yy[1], yy[0]);
endmodule
```

// test-bench module

tst_mux4_1 (); reg

[3:0] i; reg [1:0] s;

mux4_1 mm (y, i, s);

initial

begin

#2 {i, s} = 6'b 0000-00;

#2 {i, s} = 6'b 0001-00;

```

#2 {i,s} = 6'b 0010 - 01;
#2 {i,s} = 6'b 0100 - 10;
#2 {i,s} = 6'b 1000 - 11;
#2 {i,s} = 6'b 0001 - 00;

```

end

initial

```

$monitor ($time, " input s = %b, y = %b", s, y);

```

Endmodule.

DESIGN OF FLIPFLOPS WITH GATE PRIMITIVES

The basic RS latch can be designed using gate primitives. Two instantiations of NAND or NOR gates suffice here.

A Simple Latch

The design description of a simple latch formed with NAND gates as follows

```

module sbribff (sb,rb,q,qb); input sb,rb;
output q,qb; nand (q,sb,qb); nand(qb,rb,q);
end module

```

Test bench of simple latch

```

module tstsrbff; // test-bench

```

```

reg sb,rb;

```

```

wire q,qb

```

```

sbribff ff(sb,rb,q,qb)

```

```

initial begin sb=1'b1; rb=1'b0;

```

```

end

```

```

always begin

```

```

#2 sb=1'b1; rb=1'b1;

```

```

#2 sb=1'b0; rb=1'b1;

```

```

#2 sb=1'b1; rb=1'b1;

```

```

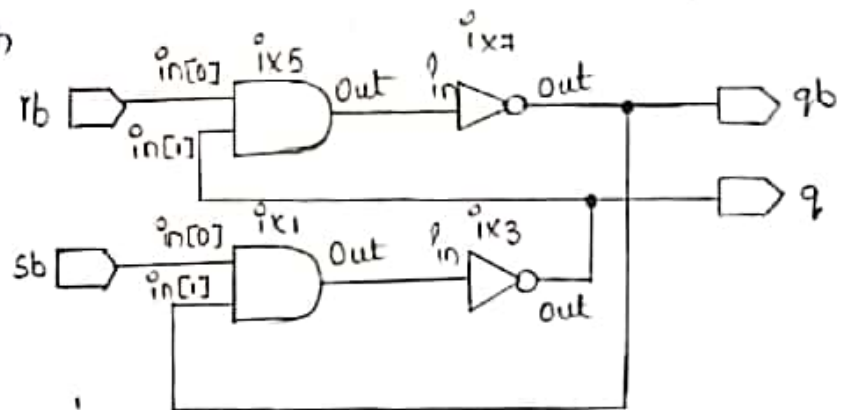
#2 sb=1'b1; rb=1'b0;

```

```

#2 sb=1'b1; qb=1'b1; end

```



initial \$monitor(\$time, "sb = %b, rb = %b, q = %b, qb = %b", sb, rb, q, qb);

initial #20 \$stop;

endmodule.

An RS Flip-flop

The design module of an RS flip-flop along with a test bench is module srff(s, r, q, qb); input s, r;

Output q, qb; wire ss, rr;

not(ss, s), (rr, r); nand(q, ss, qb);

endmodule.

Test bench

module tstsrff

reg s, r

wire q, qb;

srff ff(s, r, q, qb);

initial begin s = 1'b1; r = 1'b0;

end

always begin

#2 s = 1'b0; r = 1'b0;

#2 s = 1'b0; r = 1'b1;

#2 s = 1'b0; r = 1'b0;

#2 s = 1'b1; r = 1'b0;

#2 s = 1'b0; r = 1'b0; end

initial \$monitor(\$time, "s = %b, r = %b, q = %b, qb = %b", s, r, q, qb);

initial #20 \$stop;

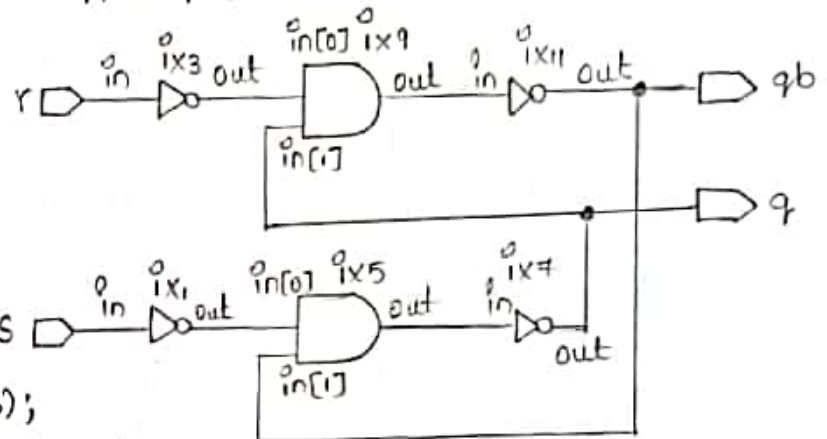
D - Latch

The design description of a D latch is module dlatch(en, d, q, qb); input d, en; Output q, qb; wire dd; wire s, r;

not n1(dd, d); nand(sb, d, en); nand(q2(rb, dd, en);

sbrrff ff(sb, rb, q, qb); // instantiation of the sbrrff

end module.



Test - bench

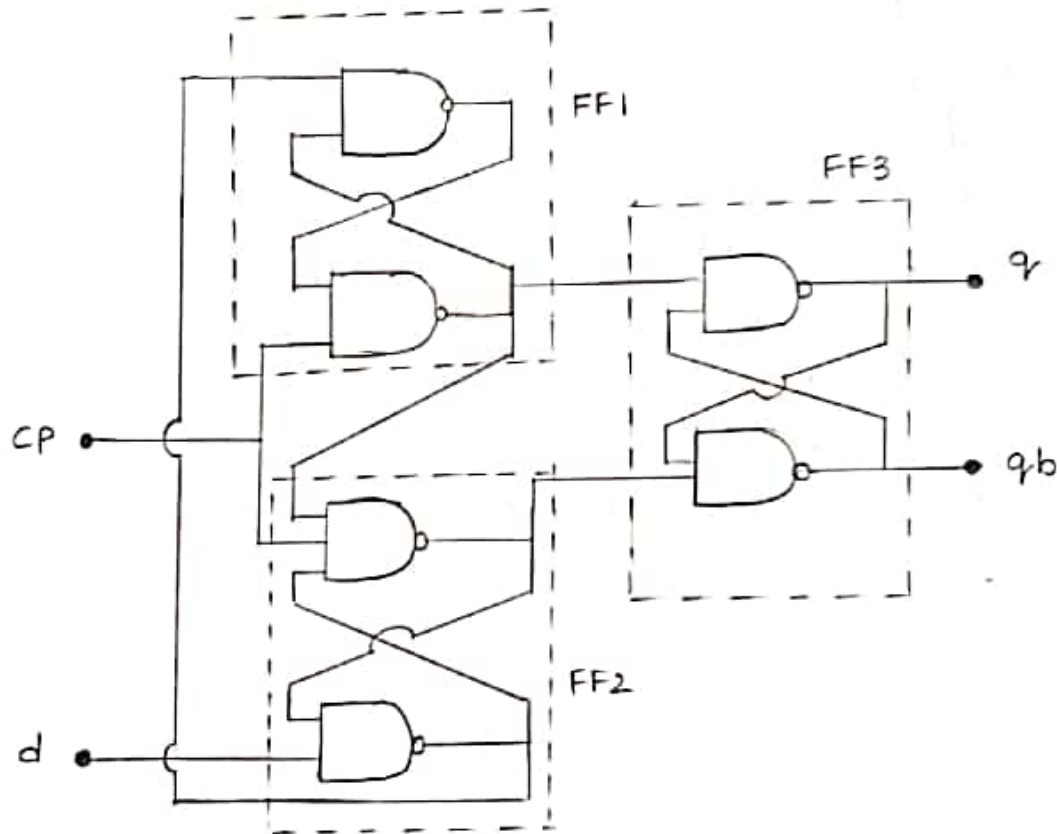
12

```

module tstlatch; req d, en; wire q, qb;
    latch ff (en, d, q, qb);
    initial begin d = 1'b0; en = 1'b0;
    end
    always #4 en = ~en; always #8 d = ~d;
    initial $monitor ($time, " en = %.b, d = %.b, q = %.b, qb = %.b", en,
d, q, qb); initial #10 $stop;
    endmodule.

```

An Edge-Triggered Flip-Flop



```

module dffgatnew1 (cp, d, q, qb);
    input d, cp; output q, qb;
    wire sb, rb; wire s, r;

    sbrbtdff ff1 (rb, cp, s);
    sbrbtf1 ff2 (s, d, cp, r, rb);
    sbrbtf ff3 (s, r, q, qb);
endmodule

```

Test-Bench

```

module test_dffqatnew1;
    reg d, cp; wire q, qb;

    dffqatnew1 #1 (cp, d, q, qb);
    initial begin d = 1'b0; cp = 1'b0;

    #2 cp = 1'b1; #2 cp = 1'b0; #2 cp = 1'b1; #2 cp = 1'b0;
    #2 cp = 1'b1; #2 cp = 1'b0; #2 cp = 1'b1; #2 cp = 1'b0; end

    initial
    begin
        #3 d = 1'b1; #2 d = 1'b1; #2 d = 1'b0; #3 d = 1'b0; #3 d = 1'b1;
        End

        initial $monitor ($time, " cp = %b, d = %b, q = %b, qb = %b",
        cp, d, q, qb); initial #40 $stop;

    endmodule

```

```

module sbrbtdff (sb, rb, qb);
    input sb, rb; output
    qb; wire q;
    nand (q, sb, qb);
    nand (qb, rb, q);
endmodule

```

Test-Bench

```

module sbrtdff (sb, rb, cp, q, qb);
    input sb, rb, cp; output
    q, qb
    nand (q, sb, cp, qb);
    nand (qb, rb, q);
endmodule.

```

Verilog has the facility to account for different types of propagation delays of Circuits elements. Any Connection Can Cause a delay due to the distributed nature of its resistance and Capacity. Similar delays are present in gates too. These manifest as propagation delays in the 0 to 1 transitions and 1 to 0 transitions from input to the output. Such propagation delays can differ for the two types of transitions.

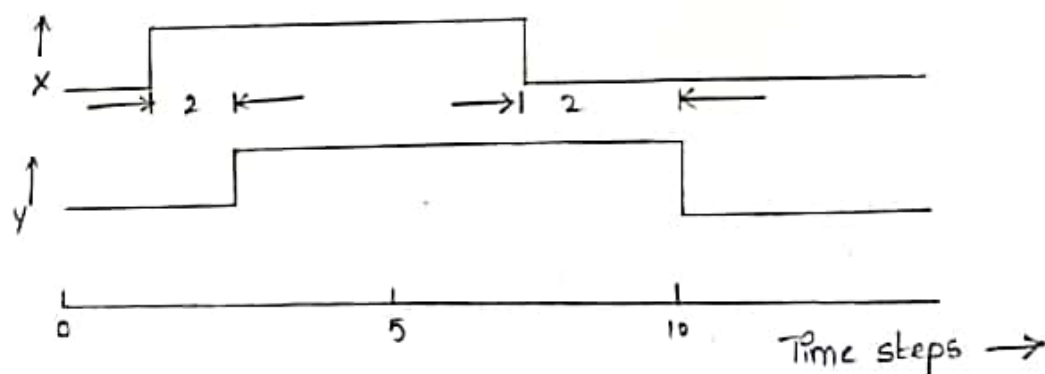
NET Delay

One of the simplest delays in that of direct Connection - a net

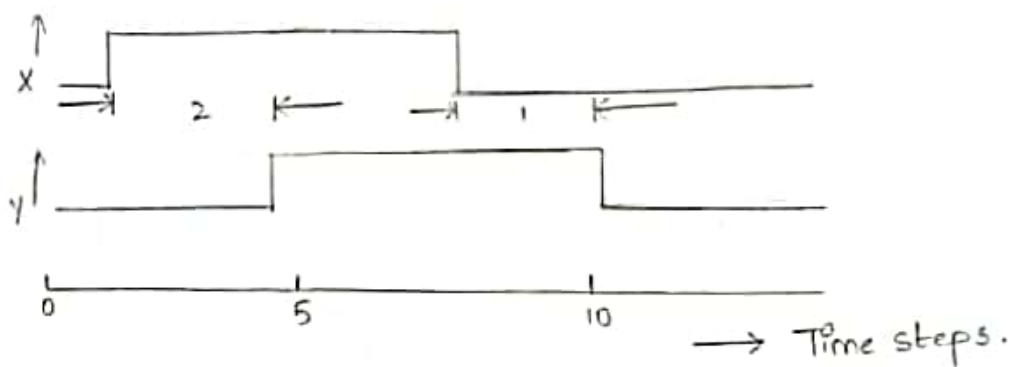
It can be part of the declaration statement wire

#2nn; a propagation delay of 2 time steps.

Here nn is declared as a net with an associated propagation delay of 2 time steps. The delay is the same for the positive as well as the negative transitions. Similar delays can be assigned to other types of nets as well. Whenever a variable or a signal is defined as a net and no delay is specified for it, the associated delay is taken as zero.



The impedance Connected as well as the type of loading can differ for the two transitions. The propagation delay too can differ accordingly. Two such delays can be specified as follows: wire #(2,1)nn; here nn is declared as a net with two distinct propagation delays; the positive (0 to 1) transition has a delay of 2 time steps associated as a net with it. The negative (1 to 0) transition has a delay of 1 time step.

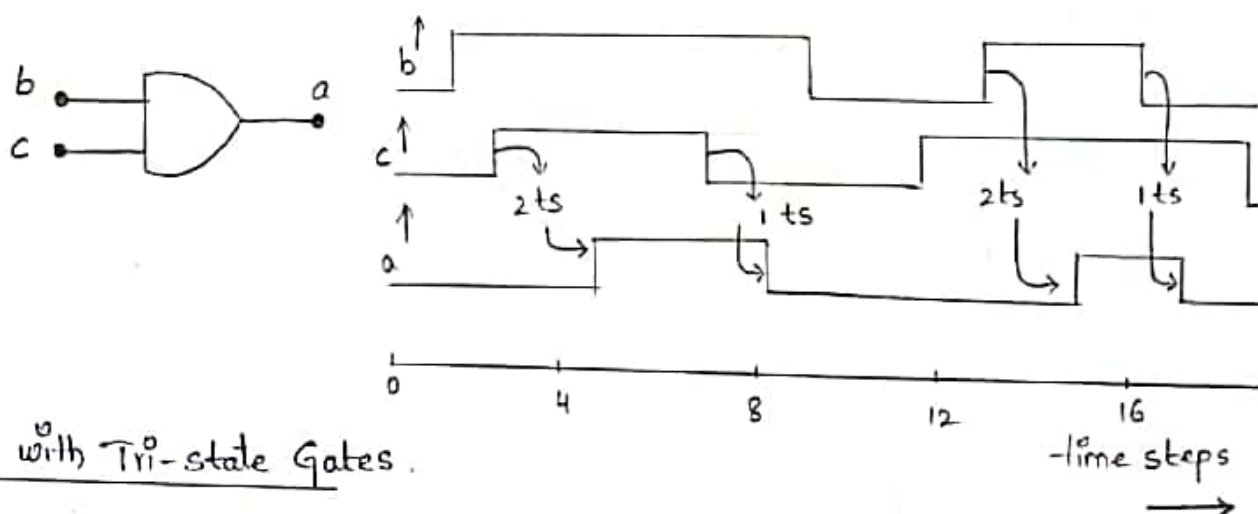


Gate delay

Gates delays can be specified as part of the instantiation itself, and `#3q(a,b,c);`

The above represents as AND gate description with a uniform delay of 3 ns for all transitions from input to output. A more detailed description can be as follows: `and # (2,1) (a,b,c);`

with the above statement the positive (0 to 1) transition at the output has a delay of 2 time steps while the negative (1 to 0) transition has a delay of 1 time step.



Delays with Tri-state Gates.

For tri-state gates the delays associated with the control signals can be different from those of the input as well as the input.

These time delays values are specified:

1. The first number represents the delay associated with the positive (0 to 1) transition of the output.
2. The second number represents the delay associated with the negative (1 to 0) transition of the output.

3. The third number represents the delay for the output to go to the hi-Z state as the control signal changes from 1 to 0 (i.e., ON to OFF Command).

bufif0 @ (1, 2, 3) bi(a0, a1, c);

Delay of the 0 to 1 transition of a0

Delay for the 1 to 0 transition of a0.

Delay for the output to go to the hi-Z state as c changes from 1 to 0.

Delays for the other hi-Z state buffers - namely bufif0, notif0 and notif1 - may be specified in a similar manner.

The following are noteworthy here:

- * Delays and storage times can be specified on the gate primitives and the nets but not on regs.

- * All three time values are separately specified in the most versatile case.

If only two time values are specified, these are interpreted by Verilog as the rise (0 to 1) and fall (1 to 0) time, respectively.

- * The turn-off time (delay) is taken as the smaller of these two.

- * If only one time value is specified, it is taken as the rise time, the fall time, and the turn-off time.

- * If no time value is specified, the rise and fall times at the output are taken as zero. The turn-off is taken as instantiations.

It is customary for manufacturers to specify delays and their range in the following manner:

→ Max delay: The maximum value of the delay in a batch; that is, the delay encountered in practice is guaranteed to be less than this in the worse case.

→ Min delay: Minimum Value to delay in a batch; that is, the Specified signal is guaranteed to be available only after a minimum of time specified.

→ Typ. delay: Typical or representative value of the delay.

Each of the delays in a gate primitive or for a net can be specified in terms of these three values.

for example

and #2(2;3;4) g1(a0,a1,a2);

Can instantiate an AND gate with the following time delay specifications:

The 0 to 1 rise-time and 1 to 0 fall-time are equal

⇒ The minimum value of either is 2 time steps. Typical value is 3 time steps and the maximum value is 4 time steps.

⇒ Note that the colon that separates the number signifies that the timings specified are the minimum, typical, and maximum values.

⇒ If the same is not specified, the simulation is carried out with the typical delay value.

The group of minimum, typical, and maximum delay values for the propagation delays can be specified separately for any gate primitive.

Thus, an AND gate primitive can be specified as and #2(1;2;3, 2;4;6) g2(b0,b1,b2)

Here for the 0 to 1 transition of the output (rise-time) the gate has a minimum delay value of 1ns, a typical value of 2ns, and a maximum value of 3ns. Similarly, for the 1 to 0 transition (fall-time) the gate has a minimum delay value of 2ns, a typical delay 4ns, and a maximum delay value of 6ns.

Such delay specifications can be associated with nets as well as tri-state type gates also.

wire #1(1;2;3) a;

buf1 #1(1;2;3, 2;4;6, 3;6;9) g3(a0,b0,c0);

The different delay values for the buffer are as follows: (19)

The Output rise time (0 to 1 transition) has a minimum value of 1 ns, a typical value of 2 ns and a maximum value of 3 ns.

The Output fall time (1 to 0 transition) has a minimum value of 2 ns, a typical value of 4 ns and a maximum value of 6 ns.

The Output turn-off time (1 to 0) has a minimum value of 3 ns, a typical of 6 ns, and a maximum value of 9 ns.

The following general observations are in order regarding the Overall, through the Circuit:

→ A Normal design can have a (many) gates and nets in its signal paths. The delays through any path for a signal depends on the path and the type of transition at each stage.

→ The cumulative delay for a signal in a path puts an upper limit on the maximum operating frequency via a signal.

→ A signal may go through multiple paths in a design to arrive at one gate. It is necessary to match the delays within specified tolerances for reliable operation of the device.

→ In larger device(s) designs, one has to identify the longest signal path.

→ This puts an upper limit on the operating frequency apart from causing malfunction in a worst case scenario.

One of the practices in designs is to reroute selected signals or redo selected design segments to reduce critical path delays.

In practical situations, outputs of logic gates and signals on nets in a circuit have associated source impedances. When the outputs of two gates are joined together, the signal level is decided by the relative magnitude of the source impedances.

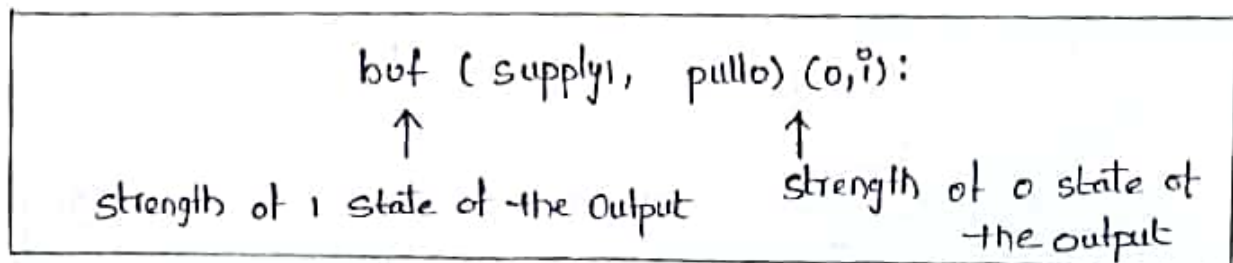
Strength Contention in Gate primitives.

When two signals of opposite polarity and differing strengths drive a line, the output status is decided by the stronger signal. However, if the signals are of equal strength, the output is indeterminate. Different contention possibilities arise here whenever there is a contention, the logic value of the output is decided by the stronger signal.

Net charges. Strengths of gate primitives.

Table gives the names associated with strength, respective abbreviations, and their order by weight.

Name	Supply	Strong	Pull	Weak	High impedance
Abbreviations	SV1	SH	PU1	WE1	HIZ1
	SV0	ST0	PU0	WE0	HIZ0
Strength	Strongest			Weakest	



Net charges

Whenever a net is driven by a signal, it takes the logic value of the signal. When the signal source is tri-stated, the net too gets tri-stated. In practice the net can have a capacitor associated with it, which can store the signal level

even after the signal are of equal strength, source dries up. To account for this situation, a charge storage capacity is associated with the net. Such nets are declared with the keyword `trireq`.

A `trireq` net can be in one of two possible states only

Driven state: when driven by a source or multiple sources, the net assumes the strength of the source. It can be any of strengths except the high impedance value.

Capacitive state: when the driven source (sources) is (are) tri-stated, the net retains the last value it was in - by virtue of the Capacitive (or) Capacitance associated with it.

The value can be in one of two possible states only,

can be 0, 1 or x (but not the high impedance value)

when the capacitive state, a net can have a storage strength associated with it. Three such storage

strengths are possible - namely large, medium, and small.

when a storage strength is not specified, it is assigned the default value - medium.

NET TYPES

wire is possibly the simplest type of net declaration.
wand and war types of Nets.

DESIGN OF BASIC CIRCUITS

Example -ALU

The -ALU Considered Carries out four functions;

Addition of two 4-bit numbers

Complementing all the bits of a 4-bit Vector

Bit-by-bit AND operation of two nibbles.

Bit-by-bit XOR operation of two nibbles.

module add4g (Sum, Carry, a, b, cin);

input [3:0] a, b; input cin; output [3:0] Sum;

Output Carry; wire [2:0] cc; -la

a0(Sum[0], cc[0], a[0], b[0], cin); -la

a1(Sum[1], cc[1], a[1], b[1], cc[0]); -la

a2(Sum[2], cc[2], a[2], b[2], cc[1]); -la

a3(Sum[3], Carry, a[3], b[3], cc[2]);

end module

module andg4(c, a, b);

input [3:0] a, b;

Output [3:0] c;

and (c[0], a[0], b[0]);

and (c[1], a[1], b[1]);

and (c[2], a[2], b[2]);

and (c[3], a[3], b[3]);

endmodule

module xorg (c, a, b); input [3:0] a, b;

Output [3:0] c; wire [3:0] cc

xor x0(c[0], a[0], b[0]); xor

x1(c[1], a[1], b[1]); xor

x2(c[2], a[2], b[2]); xor

x3(c[3], a[3], b[3]);

end module

module Compl(c, a);

input [3:0] a;

Output [3:0] c;

not (c[0], a[0]);

not (c[1], a[1]);

not (c[2], a[2]);

not (c[3], a[3]);

end module

```

module dec2_4 (a, b, en);
    output [3:0] a; input [1:0] b; input en; wire [1:0] bb;
    not(bb[1], b[1]); not(bb[0], b[0]); and(a[0], en, bb[1], bb[0]), (a[1], en, bb[1],
    b[0]), (a[2], en, b[1], bb[0]), (a[3], en, b[1], b[0]);
endmodule

```

4-to-1 mux module

```

module mux4_1alu (y, i, e); input
[3:0] i; input e; output [3:0] y; bufif1
g1(y[3], i[3], e); bufif1
g2(y[2], i[2], e); bufif1
g3(y[1], i[1], e); bufif1
g4(y[0], i[0], e);
endmodule

```

```

module alu_4g(a, b, c, Carry, Cin, cen, s); input [3:0] a, b;
input [1:0] s; input cen, Cin;
    output [3:0] c; output Carry; wire [3:0]
    data0, data1, data2, data3, e; wire Carry1;
    dec2_4 m5(e, s, cen); add4g
    m1(data0, Carry1, a, b, Cin); Comp1
    m2(data1, a); Xor4 m3(data2, a, b); and4g
    m4(data3, a, b); bufif1
    g5(Carry, Carry1, cen); mux4_1alu
    m6(c, data0, e[0]); mux4_1alu
    m7(c, data1, e[1]); mux4_1alu
    m8(c, data2, e[2]); mux4_1alu
    m9(c, data3, e[3]);
endmodule.

```