# GATE LEVEL MODELING

# Levels of Design Description

- Verilog supports a design at many levels of abstraction.

The major four are –

- **Gate level (structural)**
- **Switch Level (circuit level)**
- **Behavioral level (algorithmic)**
- **Register-transfer level-RTL (data flow)**

## Introduction to gate level modeling

- Digital designers are normally familiar with all the common logic gates, their symbols, and their working.
- Any functionally complex and more involved circuits can also be built using the basic gates.
- All the basic gates are available as “Primitives” in Verilog.
- The gate-level modeling, describes the available built-in primitive gates and how these can be used to describe hardware.
- Primitives are some generalized modules that already exist in Verilog [IEEE].
- A gate (primitive) can be directly used in a design using a gate instantiation.

## Some built-in primitive are available in Verilog HDL

*i.* Multiple-input gates:
**and, nand, or, nor, xor, xnor**

*ii.* Multiple-output gates:
**buf, not**

*iii.* Tristate gates:
**bufif0, bufif1, notif0, notif1**

*iv.* Pull gates:
**pullup, pulldown**

*v.* MOS switches:
**cmos, nmos, pmos, rcmos, rnmos, rpmos**

*vi.* Bidirectional switches:
**tran, tranif0, tranif1, rtran, rtranif0, rtranif1**

## GATE LEVEL MODELING

- All the basic gates are available as “Primitives” in Verilog.

| 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 |

- A gate can be used in a design using a gate instantiation. Here is a simple format of a gate instantiation.

  ***gate_type [ instance_name ] ( term1 , term2 , . . . , termN***);

- Note that the instance_name is optional; gate_type is one the gates listed earlier.
- The terms specify the nets and registers connected to the terminals of the gate.
- Multiple instances of the same gate type can be specified in one construct. The syntax for this is the following.

  ***gate_type***

  ***[ instance_name1 ] ( term11 , term12 , . . . , term1N),***

  ***[ instance_name2 ] ( term21 , term22,. . . , term2N),***

  ***[ instance_nameM ] ( termM1 , termM2 , . . . , termMN);***

## 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.

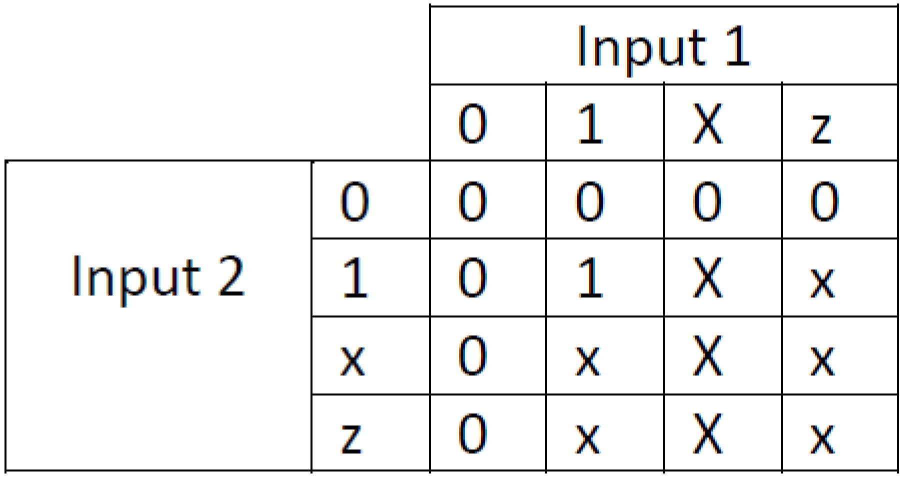
***The following are noteworthy:***

- 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:

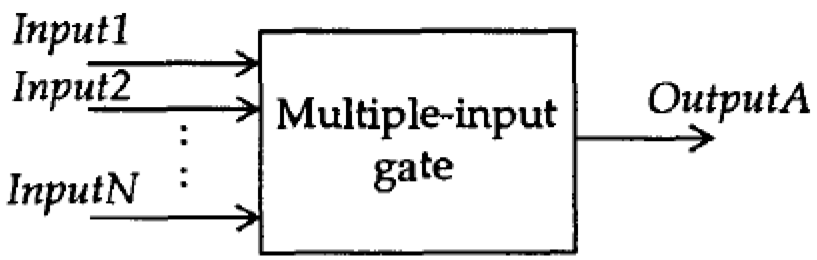
**Truth table of AND gate primitive**



| | | 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.

## 1. Multiple-input Gates



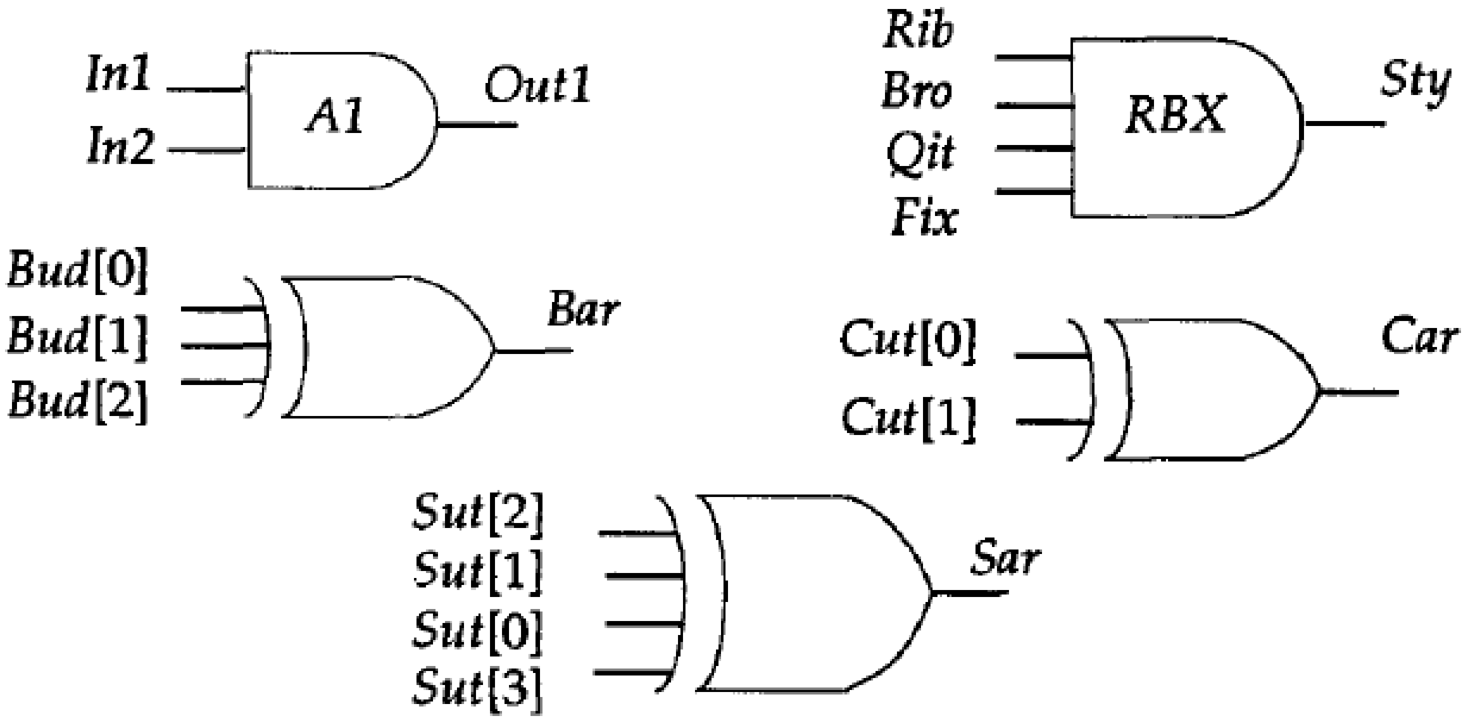
- The multiple-input built-in gates are:

  **and nand nor or xor xnor**

- These logic gateshave only one output and one or more inputs.
- Syntax:

```
multiple_input_gate_type
  [ instance_name ] ( OutputA , Input1 , Input2 ,. . ., InputN );
```

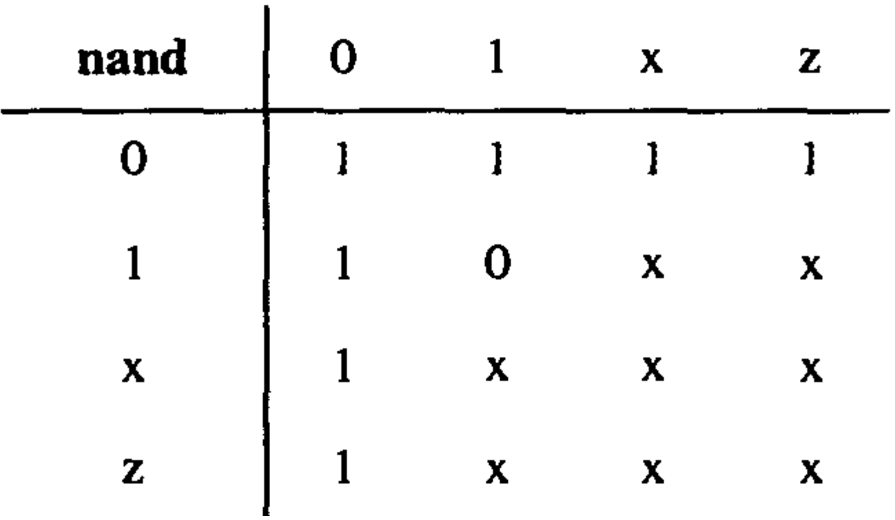
- The first terminal is the output and all others are the inputs



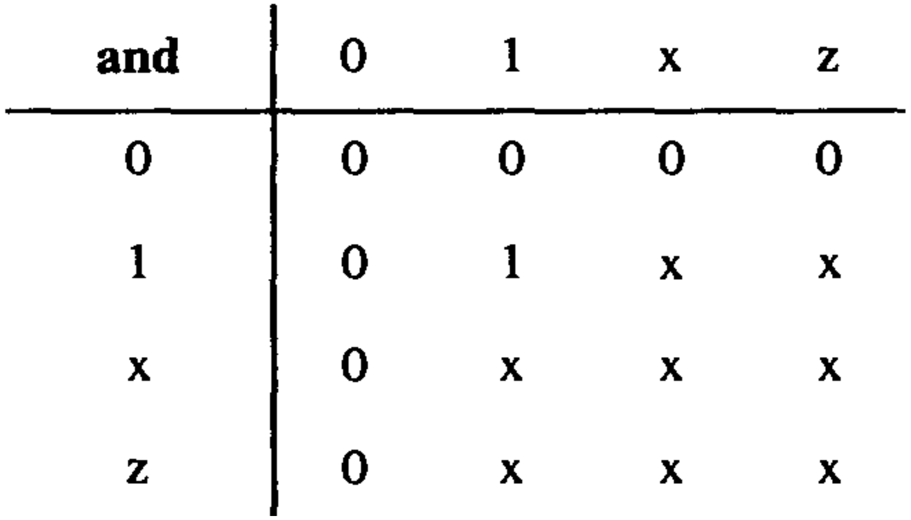
```
and A1 (Out1, In1, In2);

and RBX (Sty, Rib, Bro, Qit, Fix);

xor (Bar, Bud[0], Bud[1], Bud[2]),
    (Car, Cut[0], Cut[1]),
    (Sar, Sut[2], Sut[1], Sut[0], Sut[3]);
```

## The truth tables for logic gates

- Value z at an input is handled like an x; additionally, the output of a multiple-input gate can never be a z.



| nand | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | x | x |
| x | 1 | x | x | x |
| z | 1 | x | x | x |



| and | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x | x |
| x | 0 | x | x | x |
| z | 0 | x | x | x |

| **nand** | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | x | x |
| x | 1 | x | x | x |
| z | 1 | x | x | x |

| **and** | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x | x |
| x | 0 | x | x | x |
| z | 0 | x | x | x |

| **or** | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 0 | 1 | x | x |
| 1 | 1 | 1 | 1 | 1 |
| x | x | 1 | x | x |
| z | x | 1 | x | x |

| **nor** | 0 | 1 | x | z |
|---|---|---|---|---|
| 0 | 1 | 0 | x | x |
| 1 | 0 | 0 | 0 | 0 |
| x | x | 0 | x | x |
| z | x | 0 | x | x |

## 2. Multiple-output Gates

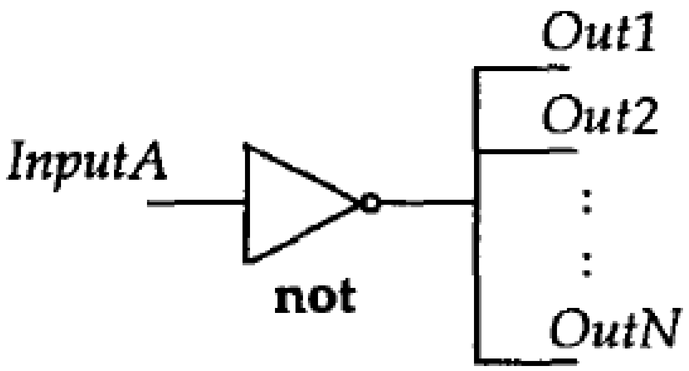
- The multiple-output gates are:

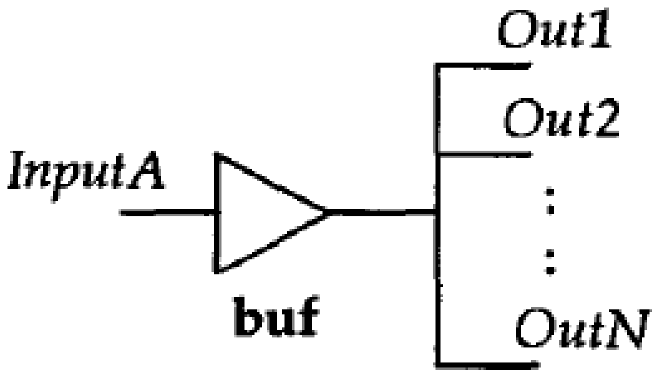
  **buf** **not**

- These gates have only one input and one or more outputs.
- The basic syntax for this gate instantiation is:

```
multiple_output_gate_type
          [ instance_name ] (Out1,Out2, . . . ,OutN,InputA);
```

- The last terminal is the input, all remaining terminals are the outputs.





```
buf  B1  (Fan[0], Fan[1], Fan[2], Fan[3], Clk);
not  N1  (PhA, PhB, Ready);
```

**not** n2 (Out1, Out2,….. OutN, *InputA);*
**buf b2 (**Out1, Out2,….. OutN, InputA);

## The truth table for these gates

| buf | 0 | 1 | x | z |
|---|---|---|---|---|
| (output) | 0 | 1 | x | x |

| not | 0 | 1 | x | z |
|---|---|---|---|---|
| (output) | 1 | 0 | x | x |

## 3.Tristate gates

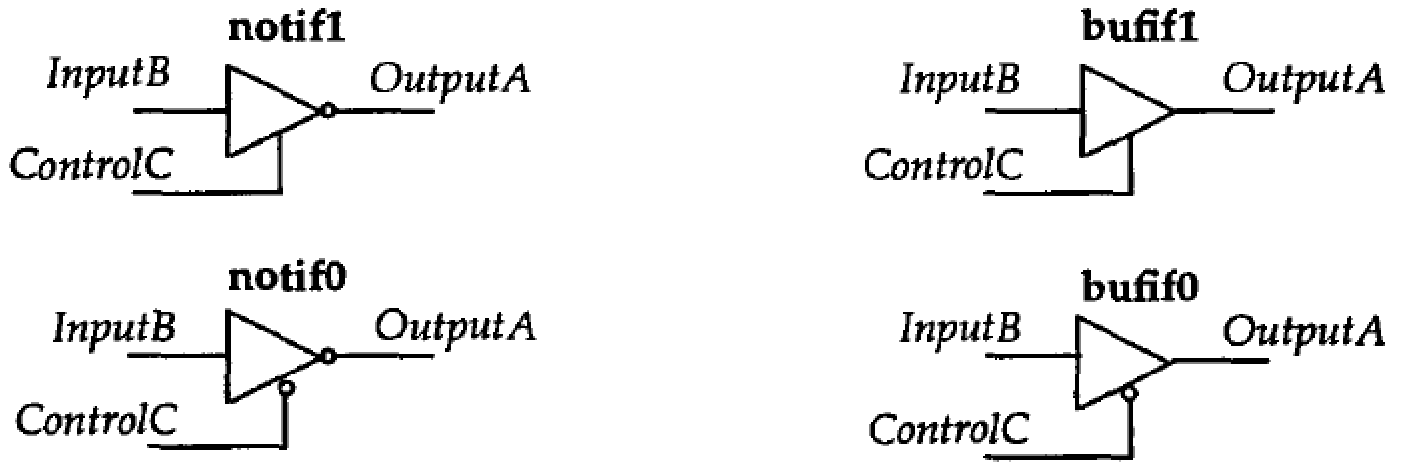
- The tristate gates are:

  **bufif0 bufif1 notif0 notif1**

- These gates model three-state drivers. These gates have one output, one data input and one control input.
- **B**asic syntax of a tristate gate instantiation.

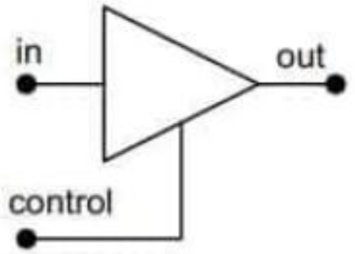
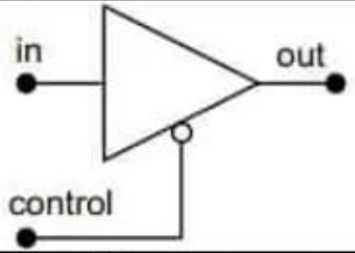
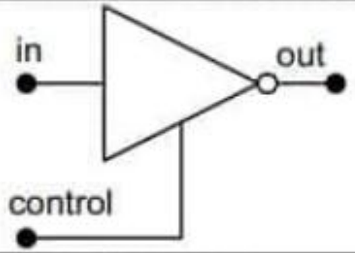
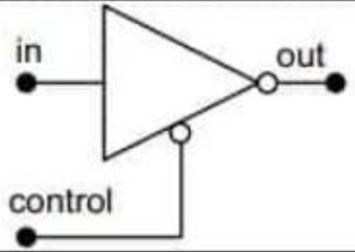
```
tristate_gate [ instance_name ] (OutputA,InputB,ControlC);
```

- The first terminal OutputA is the output, the second terminal InputB is the data input, and the control input is ControlC.
- Depending on the control input, the output can be driven to the high-impedance state, that is, to value z.



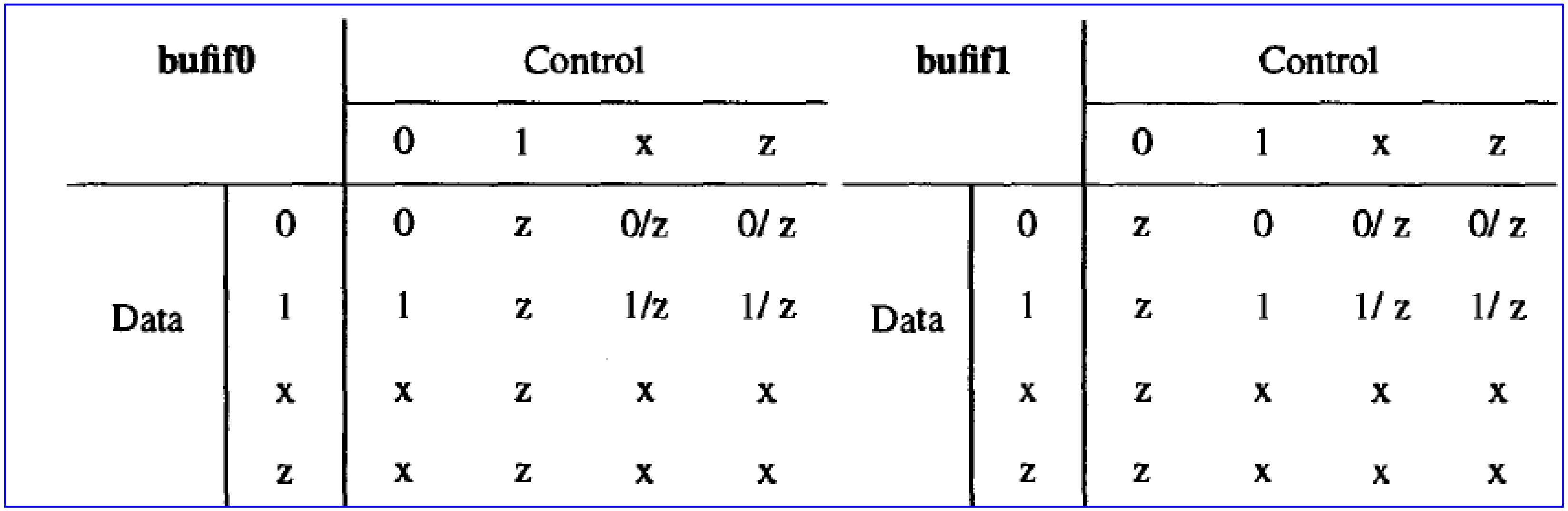
- For a bufif0 gate, the output is z if control is 1,else data is transferred to output.
- For a bufif1 gate, output is a z if control is 0.
- For a notif0 gate, output is at z if control is at 1 else output is the invert of the input data value.
- For notif1 gate, output is at z if control is at 0.

## summary

| Typical instantiation | Functional representation | Functional description |
|---|---|---|
| **bufif1** (out, in, control); |  | Out = in if control = 1; else out = z |
| **bufif0** (out, in, control); |  | Out = in if control = 0; else out = z |
| **notif1** (out, in, control); |  | Out = complement of in if control = 1; else out = z |
| **notif0** (out, in, control); |  | Out = complement of in if control = 0; else out = z |

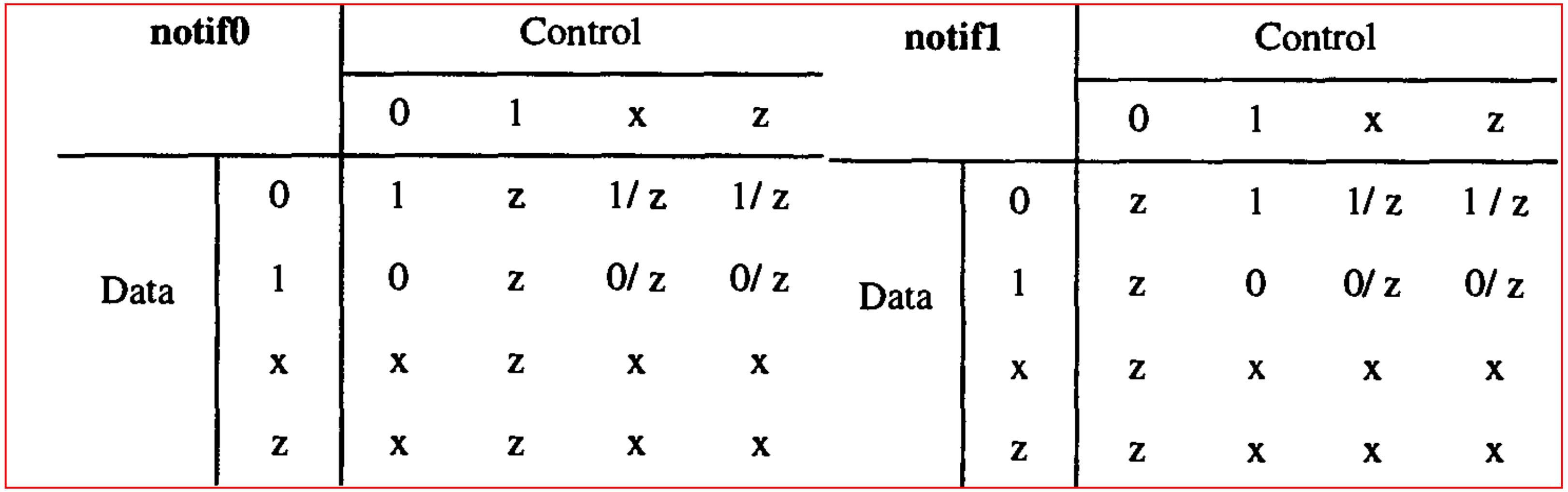
Instantiation of different cases of tristate buffer (bufif) primitives

The truth table for these gates are shown below. Some entries in the table indicate alternate entries. For example,0/z indicates that the output can either be a 0 or a z depending on the strengths of the data and control values



| **bufif0** | | Control | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| Data | 0 | 0 | z | 0/z | 0/ z |
| | 1 | 1 | z | 1/z | 1/ z |
| | x | x | z | x | x |
| | z | x | z | x | x |

| **bufif1** | | Control | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| Data | 0 | z | 0 | 0/ z | 0/ z |
| | 1 | z | 1 | 1/ z | 1/ z |
| | x | z | x | x | x |
| | z | z | x | x | x |



| **notif0** | | Control | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| Data | 0 | 1 | z | 1/ z | 1/ z |
| | 1 | 0 | z | 0/ z | 0/ z |
| | x | x | z | x | x |
| | z | x | z | x | x |

| **notif1** | | Control | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| Data | 0 | z | 1 | 1/ z | 1 / z |
| | 1 | z | 0 | 0/ z | 0/ z |
| | x | z | x | x | x |
| | z | z | x | x | x |

# 4. Pull Gates

- The pull gates are:

  **pullup**  **pulldown**

- These gates have only one output with no inputs.
- A pullup gate places a 1 on its output.
- A pulldown gate places a 0 on its output.
- A gate instantiation is of the form:

```
pull_gate [ instance_name ] ( OutputA );
```

- The terminal list of this gate instantiation contains only one output.
- Here is an example.

  ***pullup PUP {Pwr);***

- This pullup gate has instance name PUP with output Pwr tied to 1.

# 5. MOS Switches

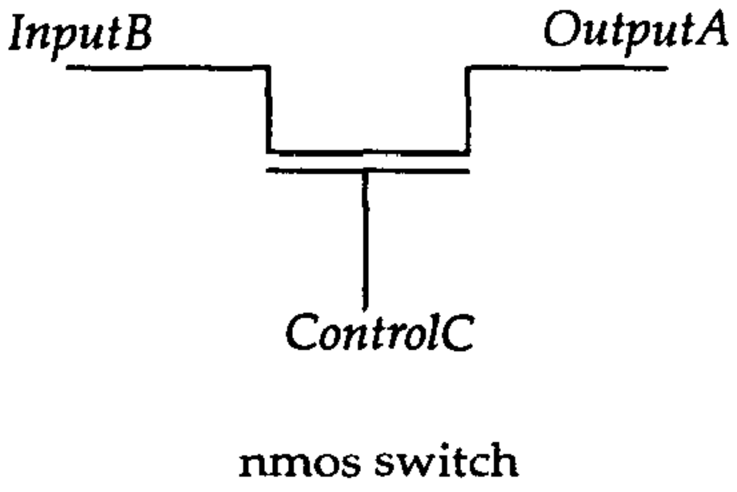
- The MOS switches are:

  **cmos   pmos   nmos   rcmos   rpmos   rnmos**

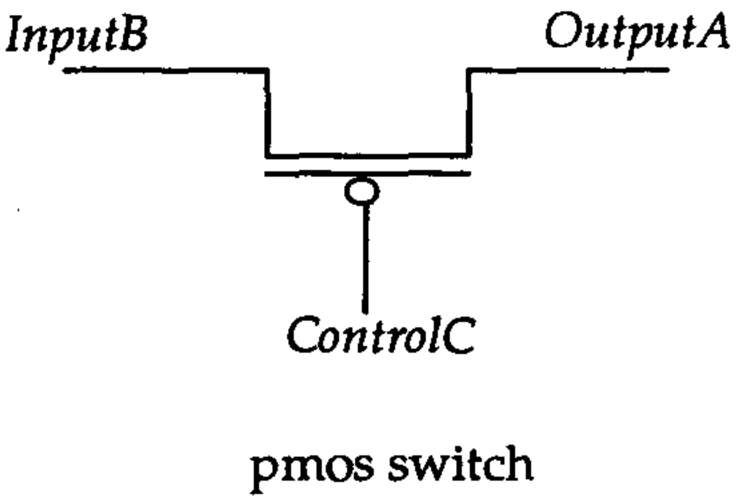
- These gates model unidirectional switches, that is, data flows from input to output and the data flow can be turned off by appropriately setting the control input(s).
- The pmos(p-type MOS transistor), nmos (n-type MOS transistor), rnmos ('r' stands for resistive) and rpmos switches have one output, one input and one control input.
- The basic syntax for an instantiation is:

```
gate_type  [ instance_name ] ( OutputA , InputB , ControlC );
```

- The first terminal is the output, the second terminal is the input and the last terminal is the control.
- For nmos and rnmos switches if control is 0 the switch is turned off, that is, output has value z; if control is 1,data at input passes to output.
- For pmos and rpmos switches, if control is 1 the switch is turned off, that is, output has value z; if control is 1,data at input passes to output.
- The resistive switches (rnmos and rpmos) have a higher impedance(resistance) between the input and output terminals as compared to the non-resistive switches (nmosand pmos).
- Thus when data passes from input to output, a reduction in strength occurs for resistive switches;



nmos switch



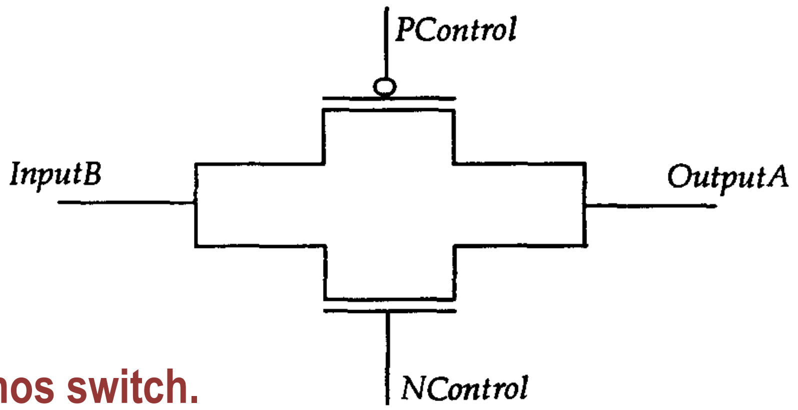
pmos switch

```
pmos P1 (BigBus, SmallBus, GateControl);
rnmos RN1 (ControlBit, ReadyBit, Hold);
```

*The first instance instantiates a pmosswitch with instance name Pl. The input to the switch is SmallBus and the output is BigBus and the control signal is GateControl*

```
(r)cmos  [ instance_name ]
                    ( OutputA , InputB , NControl , PControl );
```

- The cmos and rcmos (resistive version of cmos) switches have one data output, one data input and two control inputs.
- The first terminal is the output, the second is the input, the third is the n channel control input and the fourth terminal is the p-channel control input.
- A cmos (rcmos) switch behaves exactly like a combination of a pmos (rpmos) and an nmos (rnmos) switch with common outputs and common inputs;



(r)cmos switch.

- The truth tables for these switches are shown below. Some entries in the table indicate alternate entries. For example, 1/z indicates that the output can be either 1 or z depending on the input and control.

| **pmos** **rpmos** | | Control | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| Data | 0 | 0 | z | 0/z | 0/ z |
| | 1 | 1 | z | 1/z | 1/ z |
| | x | x | z | x | x |
| | z | z | z | z | z |

| **nmos** **rnmos** | | Control | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| Data | 0 | z | 0 | 0/ z | 0/ z |
| | 1 | z | 1 | 1/ z | 1/ z |
| | x | z | x | x | x |
| | z | z | z | z | z |

## 6. Bidirectional Switches

- The bidirectional switches are:

  **tran** **rtran** **tranif0** **rtranif0** **tranif1** **rtranif1**

- These switches are bidirectional, that is, data flows both ways and there is no delay when data propagates through the switches.
- The last four switches can be **turned off** by setting a **control signal** appropriately.
- The tran and rtran switches **cannot be turned off.**

- The syntax for instantiating a tran or a rtran (resistive version of tran) switch is:

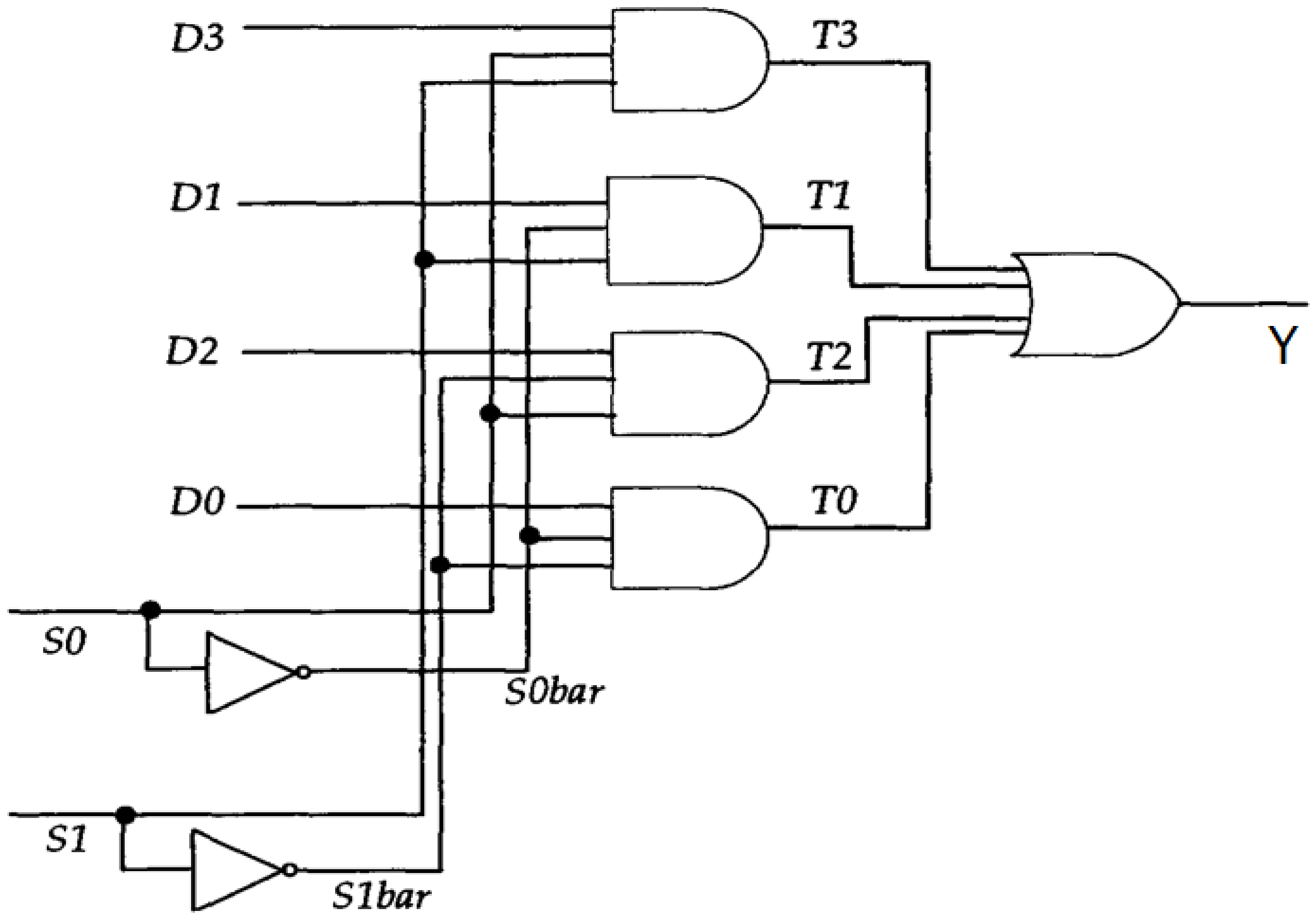
```
(r)tran [ instance_name ] ( SignalA , SignalB );
```

- The terminal list has only two terminals and data flows unconditionally both ways, that is, from SignalA to SignalB and vice versa.
- The syntax for instantiating the other bidirectional switches is:

```
gate_type [ instance_name ] ( SignalA , SignalB , ControlC );
```

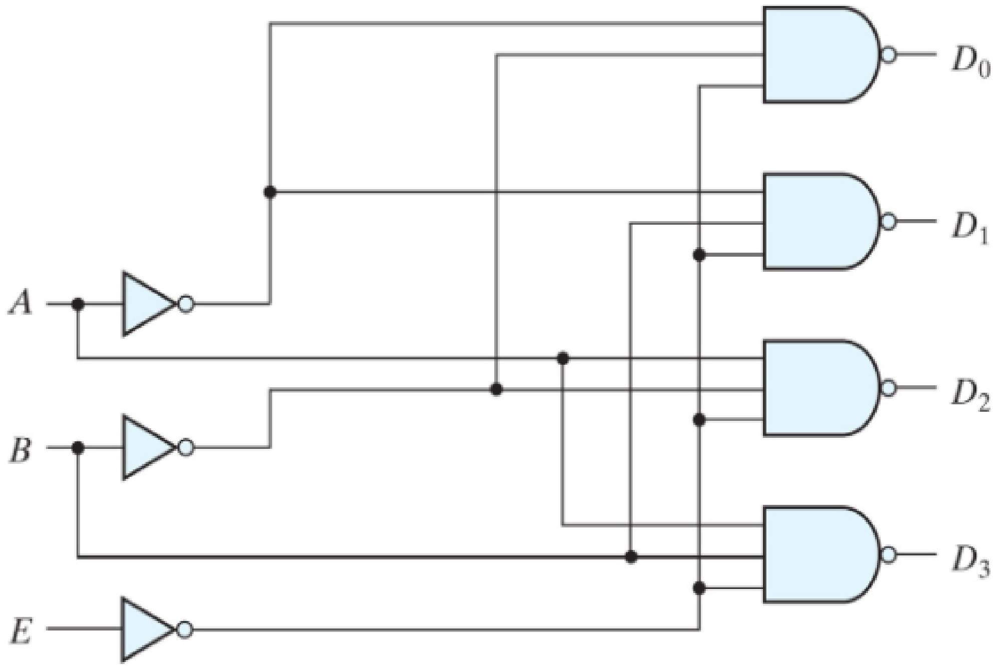
- The first two terminals are the bidirectional terminals, that is, data flows from SignalA to SignalB and vice versa.
- The third terminal is the control signal. If ControlC is 1 for tranif0 and rtranif0, and 0 for tranifl and rtranifl, the bidirectional data flow is disabled.
- For the resistive switches(rtran, rtranif0 and rtranifl), the strength of the signal reduces when it passes through the switch.

# Gate-level description of a 4:1 multiplexer circuit



```
module MUX4x1 (Y, D0, D1, D2, D3, S0, S1);
  output Y ;
  input D0, D1, D2, D3, S0, S1;

  and (T0, D0, S0bar, S1bar),
      (T1, D1, S0bar, S1),
      (T2, D2, S0, S1bar),
      (T3, D3, S0, S1);

  not (S0bar, S0),
      (S1bar, S1);

  or (Y, T0, T1, T2, T3);
endmodule
```

# A gate-level description of a 2-to-4 decoder



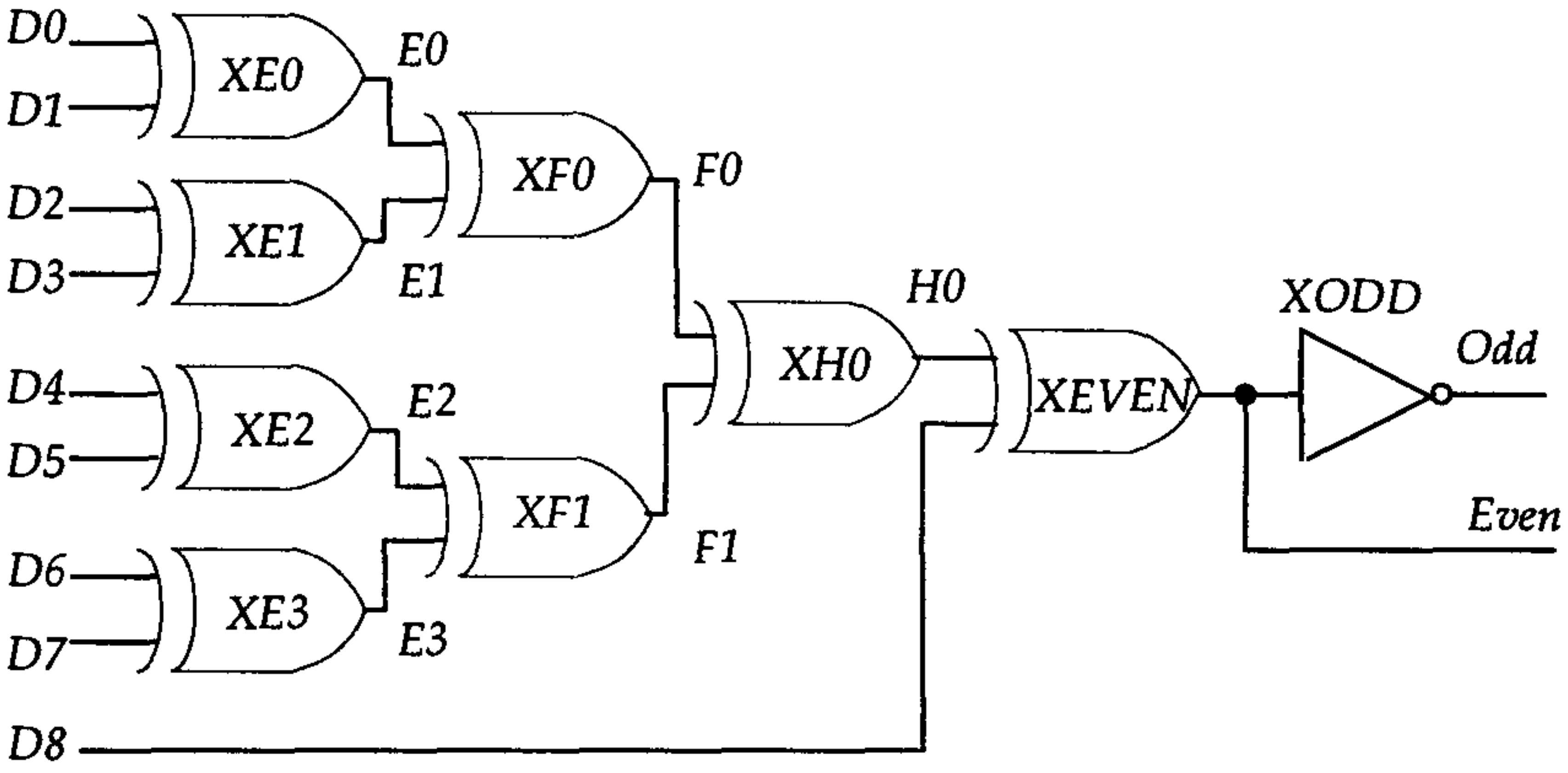
(a) Logic diagram

| $E$ | $A$ | $B$ | $D_0$ | $D_1$ | $D_2$ | $D_3$ |
|---|---|---|---|---|---|---|
| 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

```verilog
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
```

# A Parity Circuit



```
module Parity_9_Bit (D, Even, Odd);
  input [0:8] D;
  output Even, Odd;

  xor #(5, 4)                        //considering the rise delay is 5, the fall delay is 4
    XE0 (E0, D[0], D[1]),
    XE1 (E1, D[2], D[3]),
    XE2 (E2, D[4], D[5]),
    XE3 (E3, D[6], D[7]),
    XF0 (F0, E0, E1),
    XF1 (F1, E2, E3),
    XH0 (H0, F0, F1),
    XEVEN (Even, D[8], H0);

  not #2                             //considering both rise delay and fall delay are 2
    XODD (Odd, Even);
endmodule
```

# Ripple-carry Adder

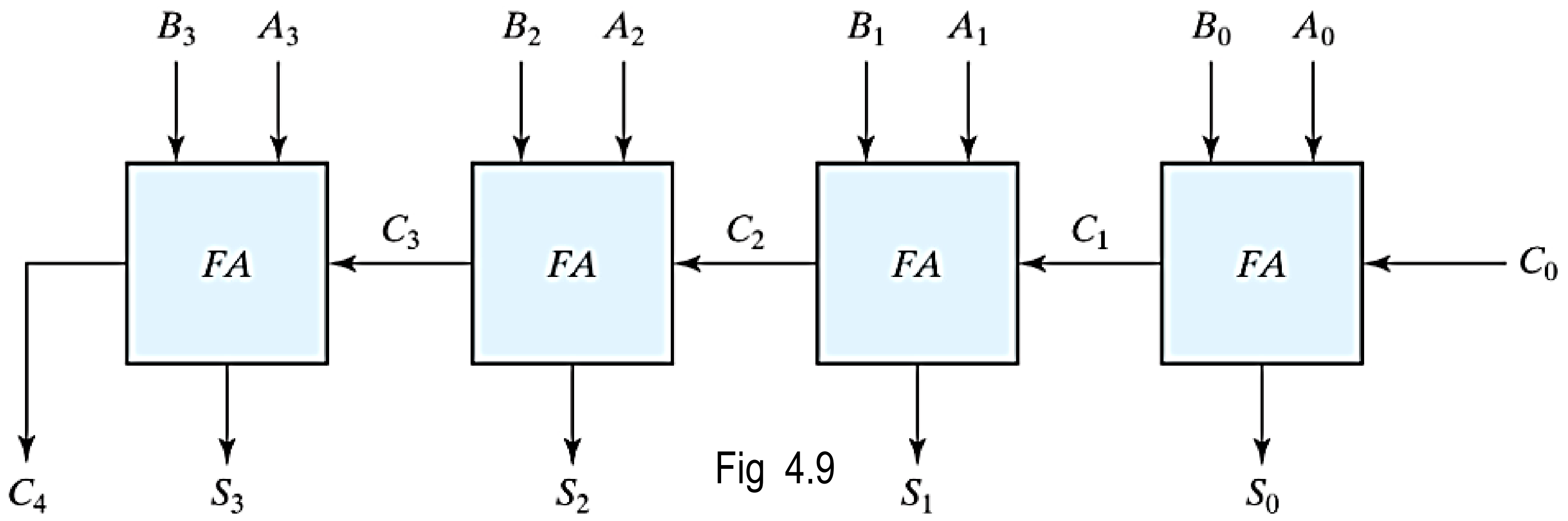


Fig 4.9

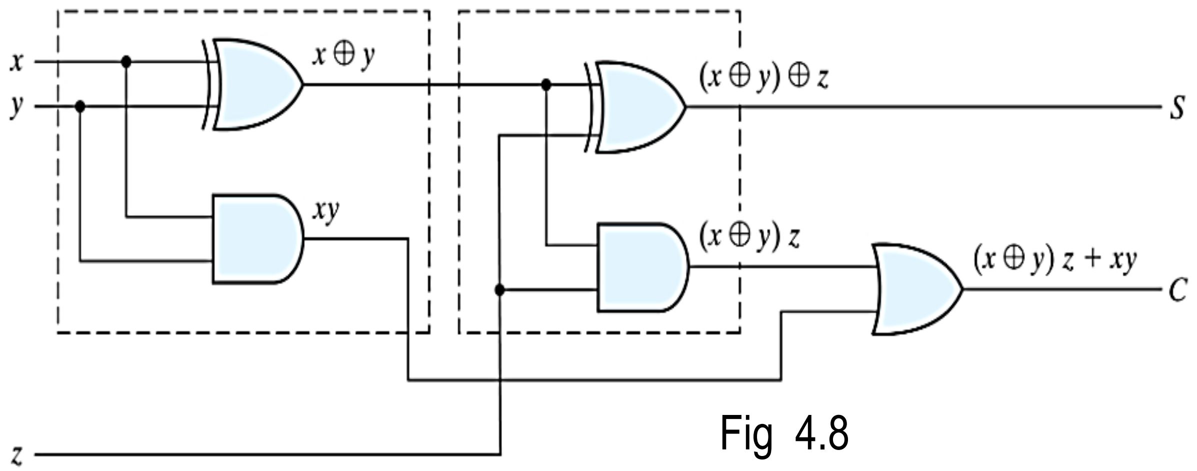
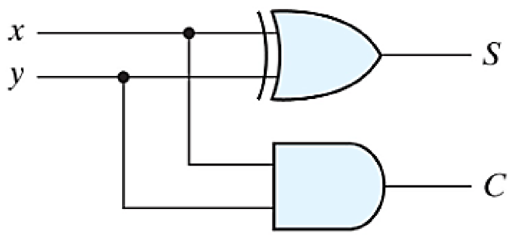


Fig 4.8



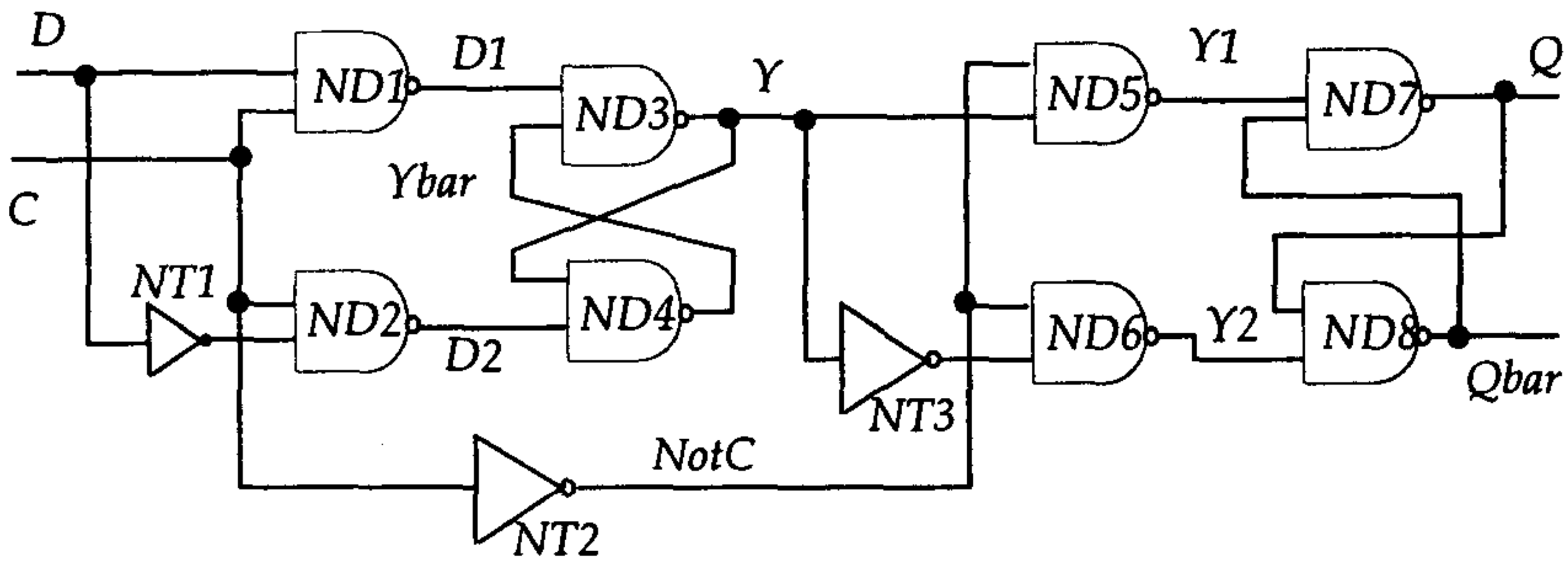
(b) $S = x \oplus y$
$C = xy$

Fig 4.5

```
// Description of half adder (see Fig 4-5b)
module halfadder (S, C, x, y);
   input x, y;
   output S, C;

// Instantiate primitive gates
   xor (S, x, y);
   and (C, x, y);
endmodule

// Description of full adder (see Fig 4-8)
module fulladder (S, C, x, y, z);
   input x,y,z;
   output S,C;
   wire S1,D1,D2; //Outputs of first XOR and two AND gates

//Instantiate the halfadders
    halfadder HA1 (S1,D1,x, y),
              HA2 (S, D2,S1,z);
    or g1(C,D2,D1);
endmodule
```

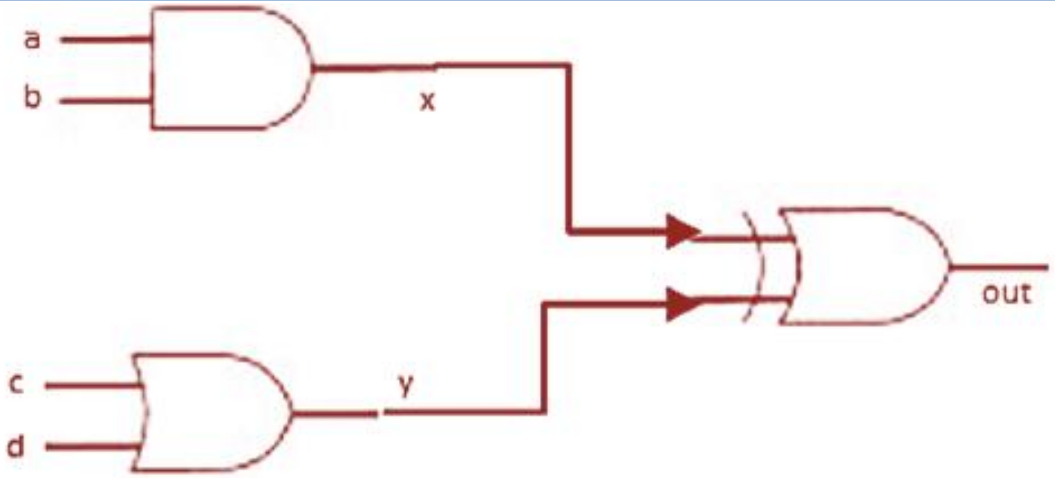
```
// Description of 4-bit adder (see Fig 4-9)
module Four_bit_adder (S,C4,A,B,C0);
   input  [3:0] A,B;
   input  C0;
   output [3:0] S;
   output C4;
   wire   C1,C2,C3;  //Intermediate carries

// Instantiate the fulladder
   fulladder  FA0 (S[0],C1,A[0],B[0],C0),
              FA1 (S[1],C2,A[1],B[1],C1),
              FA2 (S[2],C3,A[2],B[2],C2),
              FA3 (S[3],C4,A[3],B[3],C3);
endmodule
```

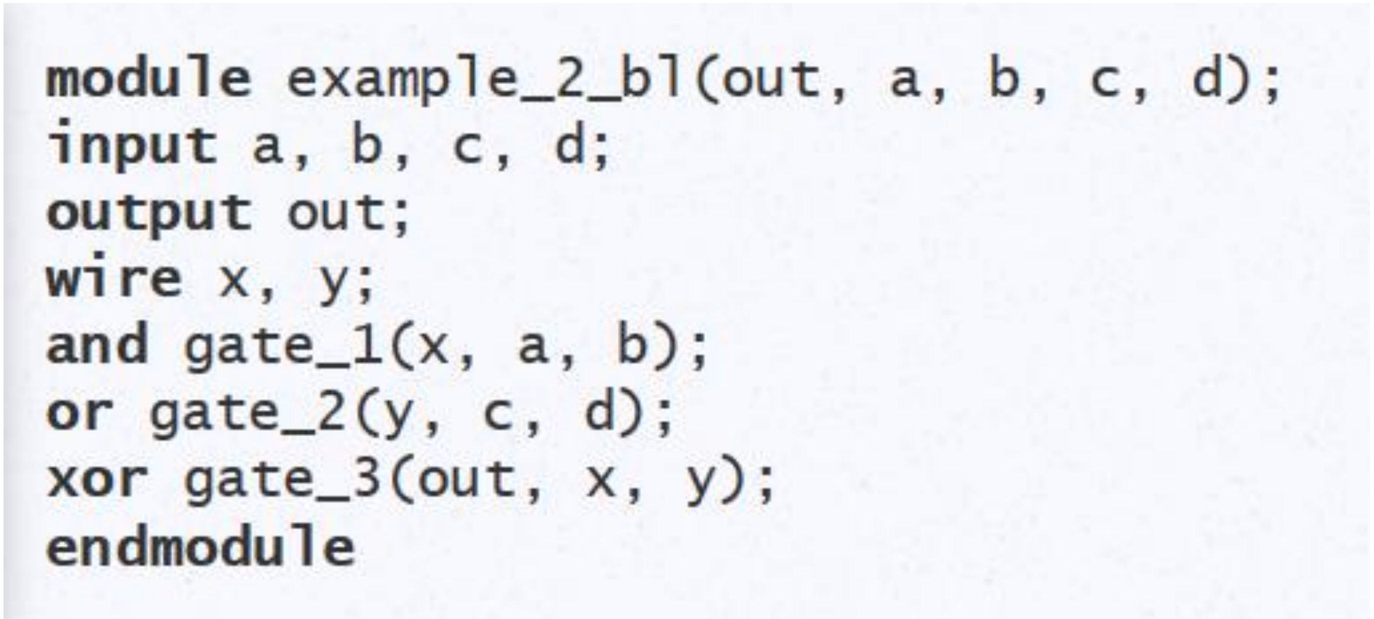
# Master-slave D-type flip-flop



```
  module MSDFF (D, C, Q, Qbar);
    input D, C;
    output Q, Qbar;

    not
      NT1 (NotD, D),
      NT2 (NotC, C),
      NT3 (NotY, Y);

   nand
     ND1 (D1, D, C),
     ND2 (D2, C, NotD),
     ND3 (Y, D1, Ybar),
     ND4 (Ybar, Y, D2),
     ND5 (Y1, Y, NotC),
     ND6 (Y2, NotY, NotC),
     ND7 (Q, Qbar, Y1),
     ND8 (Qbar, Y2, Q);
endmodule
```

## Q. Write the Verilog (gate modeling) code for the circuit below





```
module example_2_bl(out, a, b, c, d);
input a, b, c, d;
output out;
wire x, y;
and gate_1(x, a, b);
or gate_2(y, c, d);
xor gate_3(out, x, y);
endmodule
```

# Thank You