Introduction to Verilog HDL

Geeth Karunarathne

Verilog HDL

- Invented by Philip Moorby in 1983/ 1984 at Gateway Design Automation
- Enables specification of a digital system at a range of levels of abstraction: switches, gates, RTL, and higher
- Initially developed in conjunction with the Verilog simulator

Verilog HDL

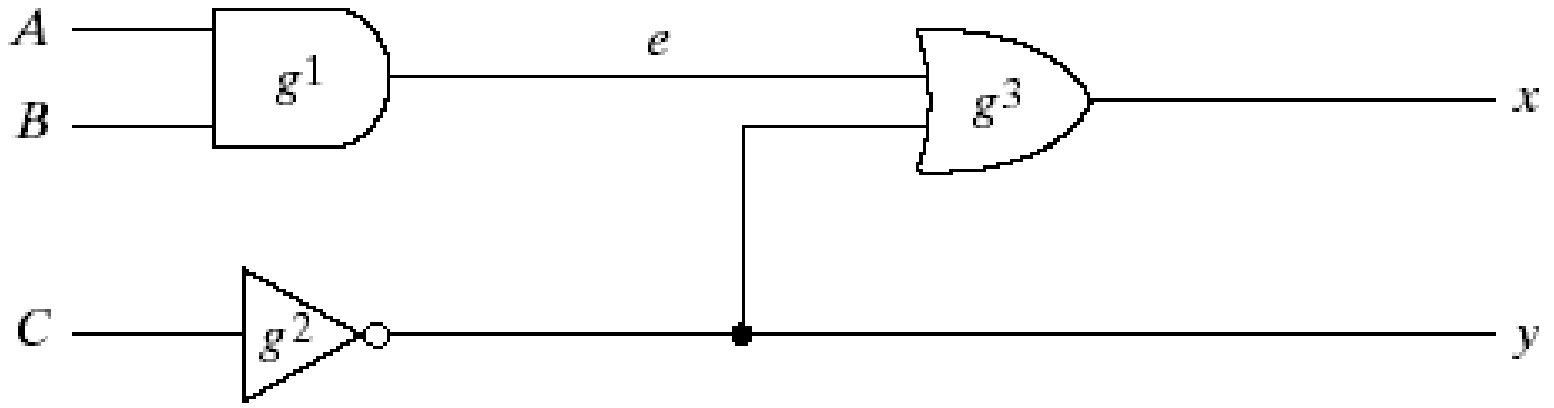
- Verilog- based synthesis tool introduced by Synopsys in 1987
- Gateway Design Automation bought by Cadence in 1989
- Verilog placed in public domain to compete with VHDL
  - Open Verilog International (OVI) IEEE 1364 -1995 and
    - revised version IEEE 1364 -2001
    - revised version IEEE 1364 -2005

Verilog

- Verilog HDL has a syntax that describes precisely the legal constructs that can be used in the language.
- It uses about 100 keywords pre-defined, lowercase, identifiers that define the language constructs.
- Example of keywords: *module, endmodule, input, output wire, and, or, not* , etc.,
- Any text between two slashes (//) and the end of line is interpreted as a comment.
- Blank spaces are ignored and names are case sensitive.

Module

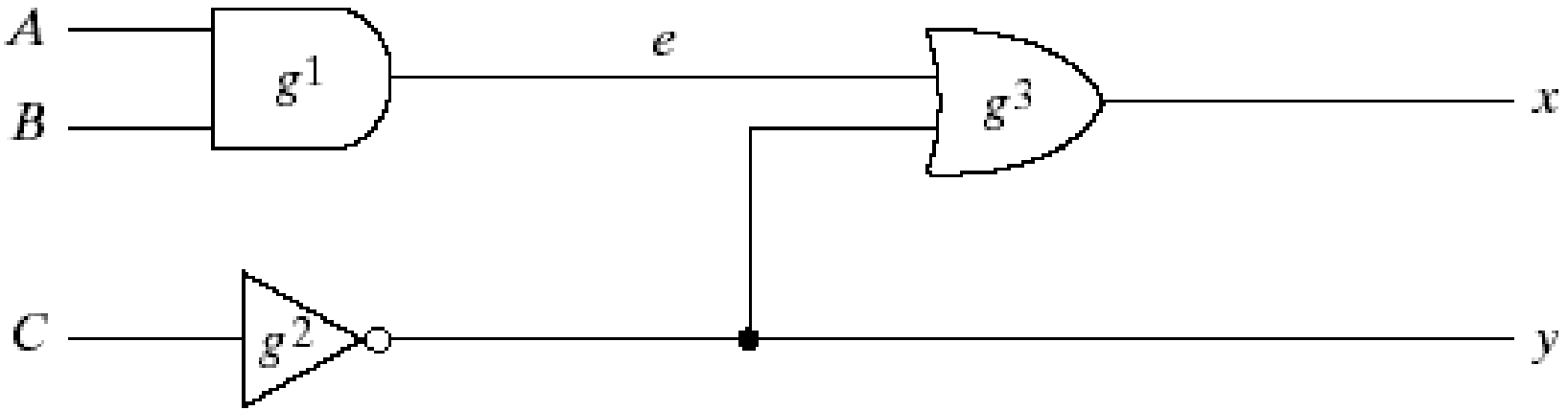
- A *module* is the building block in Verilog.
- It is declared by the keyword *module* and is always terminated by the keyword *endmodule*.
- Each statement is terminated with a semicolon, but there is no semi-colon after *endmodule*.



Module cont..

Example code

```
module smpl_circuit(A,B,C,x,y);
  input  A,B,C;
  output x,y;
  wire e;
  and g1(e,A,B);
  not g2(y,C);
  or  g3(x,e,y);
endmodule
```



Gate Delays

- Sometimes it is necessary to specify the amount of delay from the input to the output of gates.
- In Verilog, the delay is specified in terms of time units and the symbol **#**.
- The association of a time unit with physical time is made using ***timescale*** compiler directive.
- Compiler directive starts with the “backquote (`)” symbol.

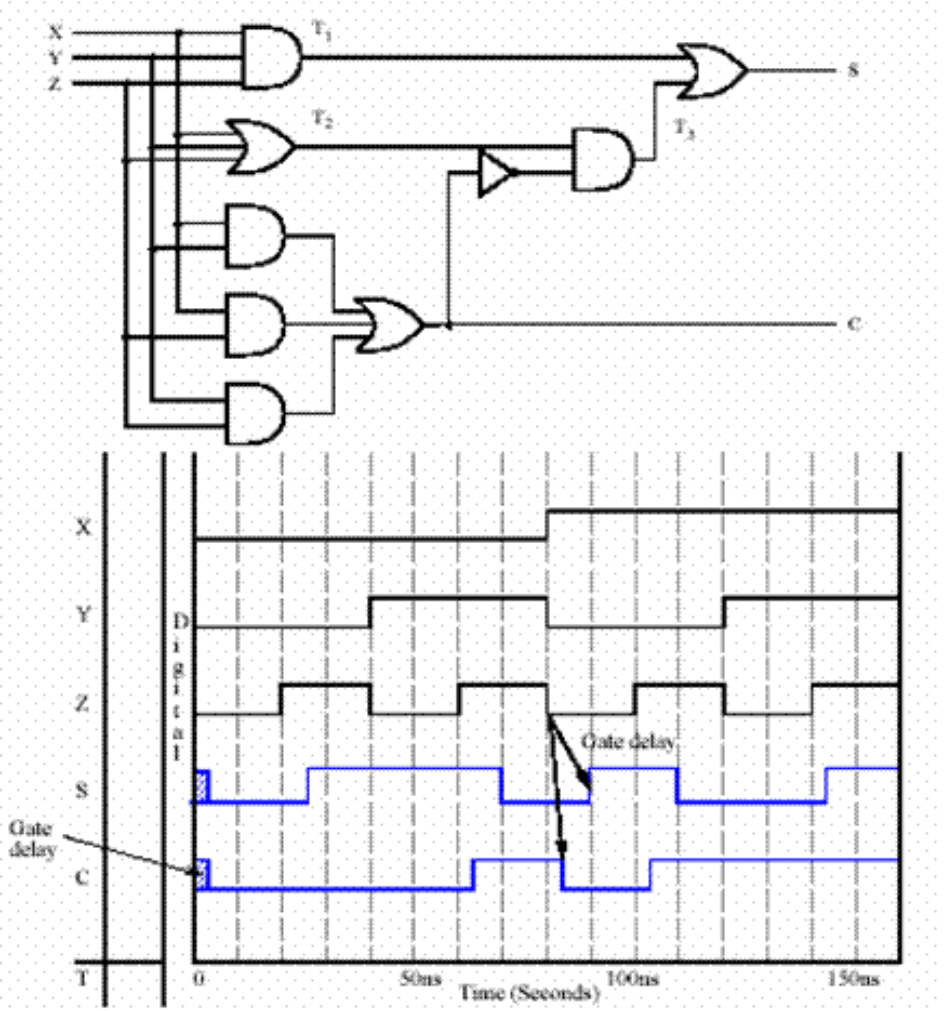
```
`timescale 1ns/100ps
```

- The first number specifies the *unit of measurement* for time delays.
- The second number specifies the *precision* for which the delays are rounded off, in this case to 0.1ns.

```
//Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
   input  A,B,C;
   output x,y;
   wire   e;
   and #(30) g1(e,A,B);
   or  #(20) g3(x,e,y);
   not #(10) g2(y,C);
endmodule
```

Logic Simulation

- Logic simulation is a fast, accurate method of analyzing a circuit to see its waveforms



Logic Simulation

- In order to simulate a circuit with verilog, it is necessary to apply inputs to the circuit for the simulator to generate an output response.
- A Verilog code that provides the stimulus to a design is called a **test bench**.
- The ***initial*** statement specifies inputs between the keyword ***begin*** and ***end***.
- Initially ABC=000 (A,B and C are each set to 1'b0 (one binary digit with a value 0).
- **$finish** is a *system task.*

Verilog Test Bench

```
module circuit_with_delay
(A,B,C,x,y);
   input A,B,C;
   output x,y;
   wire e;
   and #(30) g1(e,A,B);
   or  #(20) g3(x,e,y);
   not #(10) g2(y,C);
endmodule
```

```
//Stimulus for simple circuit
module stimcrct;
reg A,B,C;
wire x,y;
circuit_with_delay cwd(A,B,C,x,y);
initial
   begin
        A = 1'b0; B = 1'b0; C = 1'b0;
     #100
        A = 1'b1; B = 1'b1; C = 1'b1;
     #100  $finish;
   end
endmodule
```

Verilog – Simulation

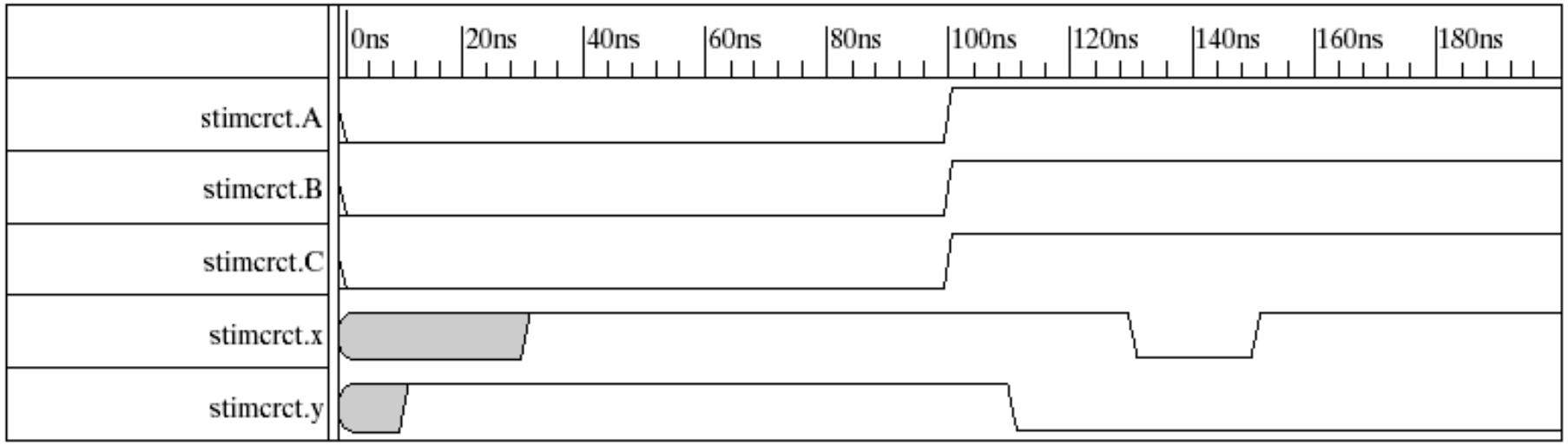


Fig. 3-38 Simulation Output of HDL Example 3-3

In the above example, **cwd** is declared as one instance **circuit_with_delay**.

Bitwise Operators

- Bitwise NOT : ~
- Bitwise AND: &
- Bitwise OR: |
- Bitwise XOR: ^
- Bitwise XNOR: ~^ or ^~

Boolean Expressions:

- These are specified in Verilog with a continuous assignment statement consisting of the keyword *assign* followed by a Boolean Expression.
- The earlier circuit can be specified using the statement:

```
assign x = (A&B)|~C)
```

E.g. $x = A + BC + B'D$

$y = B'C + BC'D'$

E.g. x = A + BC + B'D

y = B'C + BC'D'

```
//Circuit specified with Boolean equations
module circuit_bln (x,y,A,B,C,D);
   input A,B,C,D;
   output x,y;
   assign x = A | (B & C) | (~B & C);
   assign y = (~B & C) | (B & ~C & ~D);
endmodule
```

Types of Modelling in Verilog

- A Verilog code can be written in the following styles:

1. Structural style
   i. Gate Level Modeling
   ii. Module Instantiation
2. Dataflow style
3. Behavioral style
4. Mixed style

Gate-Level Modeling

- Circuit is specified by its **logic gates** and their **interconnections**.
- It provides a textual description of a schematic diagram.
- Verilog recognizes 12 basic gates as predefined primitives.
  - 4 primitive gates of 3-state type (bufif0, bufif1, notif0 and notif1).
  - Other 8 are: **`and`**, **`nand`**, **`or`**, **`nor`**, **`xor`**, **`xnor`**, **`not`**, **`buf`**
- When the gates are simulated, the system assigns a four-valued logic set to each gate – *0,1,unknown (x) and high impedance (z)*

Gate-level modeling

- When a primitive gate is incorporated into a module, we say it is *instantiated* in the module.
- statements that reference lower-level components in the design, essentially creating unique copies (or *instances*) of those components in the higher-level module.
- A module that uses a gate in its description is said to *instantiate* the gate.

Modeling with vector data (multiple bit widths)

- A vector is specified within square brackets and two numbers separated with a colon.

e.g. `output[0:3] D;` - This declares an output vector D with 4 bits, 0 through 3.

`wire[7:0] SUM;` – This declares a wire vector SUM with 8 bits numbered 7 through 0.

The first number listed is the most significant bit of the vector.

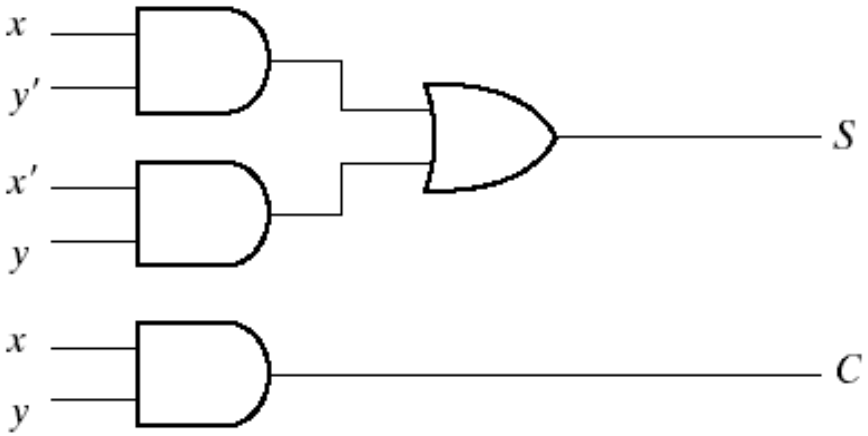
Gate-level Modeling

- Two or more modules can be combined to build a hierarchical description of a design.
- There are two basic types of design methodologies.
  - **Top down**: In top-down design, the top level block is defined and then sub-blocks necessary to build the top level block are identified.
  - **Bottom up**: Here the building blocks are first identified and then combine to build the top level block.
- In a top-down design, a 4-bit binary adder is defined as top-level block with 4 full adder blocks. Then we describe two half-adders that are required to create the full adder.
- In a bottom-up design, the half-adder is defined, then the full adder is constructed and the 4-bit adder is built from the full adders.

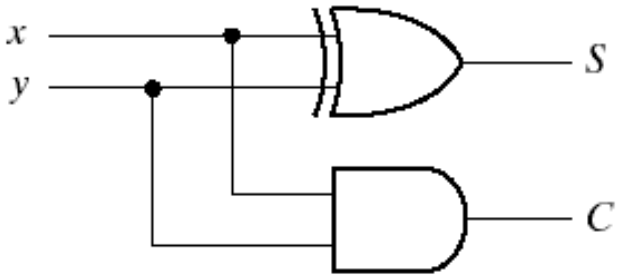
Gate-level Modeling

- A bottom-up hierarchical description of a 4-bit adder is described in Verilog as
  - Half adder: defined by instantiating primitive gates.
  - Then define the full adder by instantiating two half-adders.
  - Finally the third module describes 4-bit adder by instantiating 4 full adders.
- ***Note:*** In Verilog, one module definition cannot be placed within another module description.

Half Adder



(a) $S = xy' + x'y$
$C = xy$



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

4-bit Full Adder

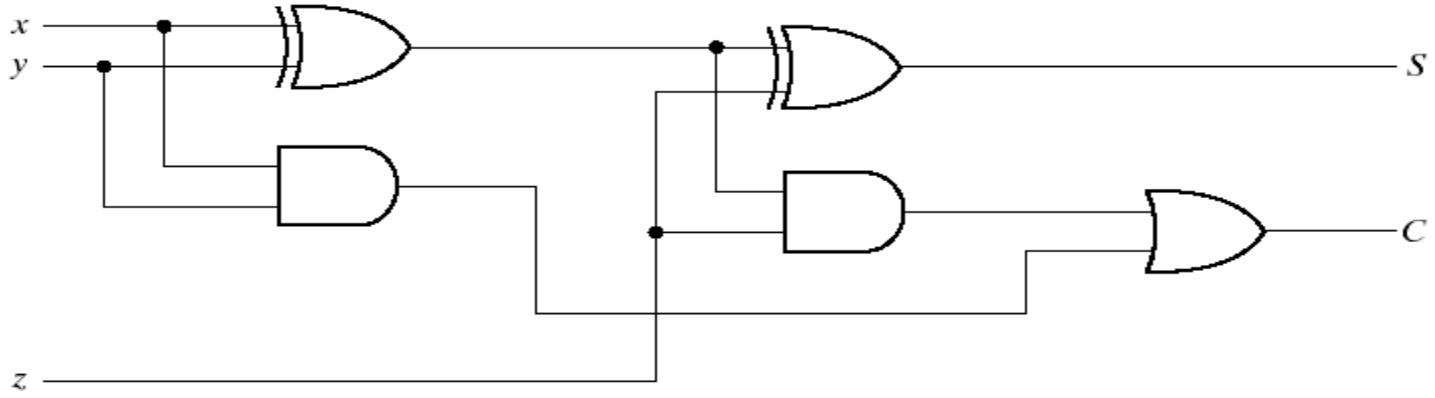


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

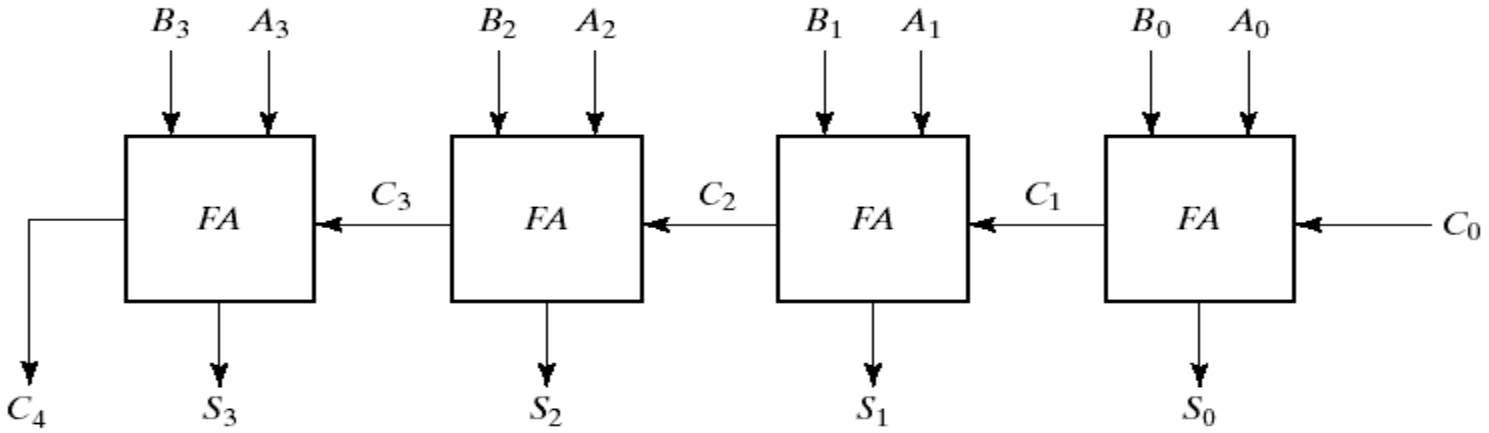


Fig. 4-9 4-Bit Adder

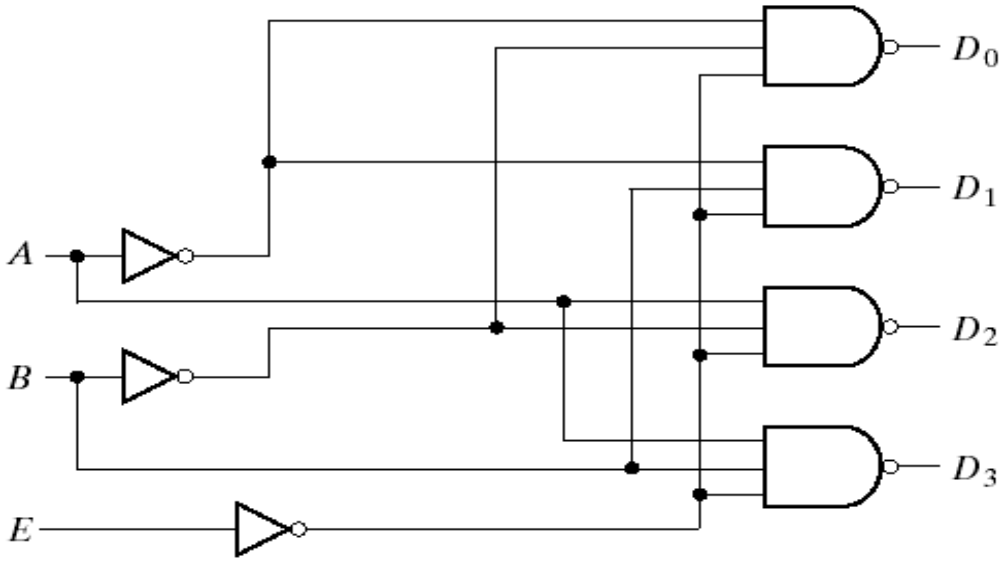
4-bit Full Adder

```
//Gate-level hierarchical description of 4-bit adder
module halfadder (S,C,x,y);
    input x,y;
    output S,C;
    //Instantiate primitive gates
    xor (S,x,y);
    and (C,x,y);
endmodule

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 half adders
     halfadder HA1(S1,D1,x,y), HA2(S,D2,S1,z);
     or g1(C,D2,D1);
endmodule
```

4-bit Full Adder

```
module 4bit_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 full adder
    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
```

2 to 4 Decoder



(a) Logic diagram

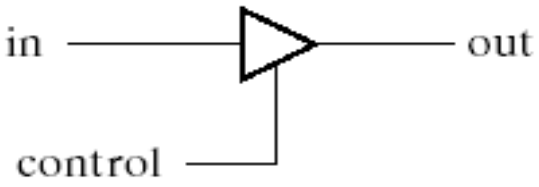
E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

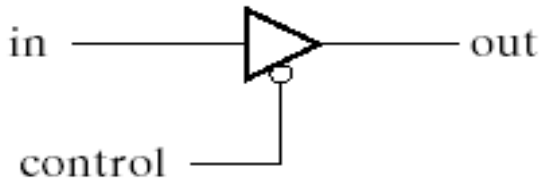
2 to 4 Decoder

```
//Gate-level description of a 2-to-4-line decoder
module decoder_gl (A,B,E,D);
   input   A,B,E;
   output[0:3]D;
   wire    Anot,Bnot,Enot;
   not
      n1 (Anot,A),
      n2 (Bnot,B),
      n3 (Enot,E);
   nand
      n4 (D[0],Anot,Bnot,Enot),
      n5 (D[1],Anot,B,Enot),
      n6 (D[2],A,Bnot,Enot),
      n7 (D[3],A,B,Enot);
endmodule
```

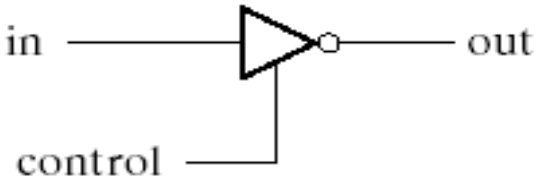
Three-State Gates



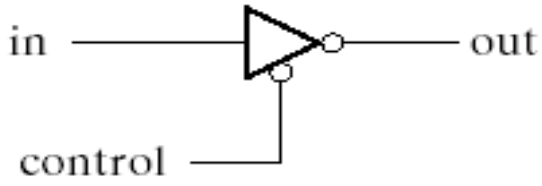
bufifl



bufif0



notifl

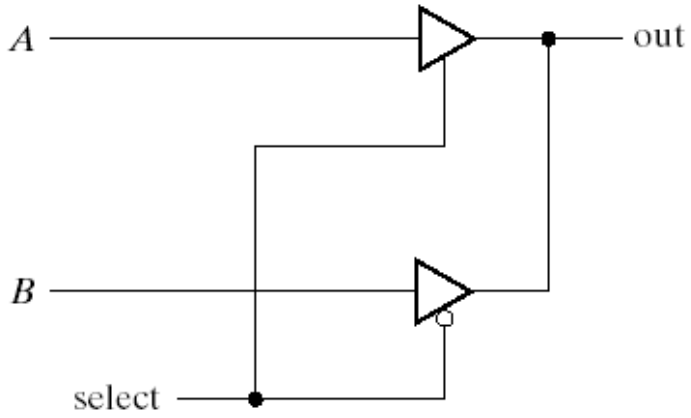


notif0

Three-State Gates

- Three-state gates have a control input that can place the gate into a high-impedance state. (symbolized by **`z`** in verilog).
- The **`bufif1`** gate behaves like a normal buffer if `control=1`. The output goes to a high-impedance state **`z`** when `control=0`.
- **`bufif0`** gate behaves in a similar way except that the high-impedance state occurs when `control=1`
- Two **`not`** gates operate in a similar manner except that the o/p is the complement of the input when the gate is not in a high impedance state.
- The gates are instantiated with the statement
  - `gate name (output, input, control);`

Three-State Gates



The output of 3-state gates can be connected together to form a common output line. To identify such connections, verilog uses the keyword tri (for tri-state) to indicate that the output has multiple drivers.

```
module muxtri(A,B,sel,out);
   input      A,B,sel;
   output     OUT;
   tri        OUT;
   bufif1     (OUT,A,sel);
   bufif0     (OUT,B,sel);
endmodule
```

Dataflow Modeling

- Dataflow modeling uses a number of operators that act on operands to produce desired results.
- Verilog provides about 30 operator types.
- Dataflow modeling uses continuous assignments and the keyword **`assign`**.
- A continuous assignment is a statement that assigns a value to a net.
- The value assigned to the net is specified by an expression that uses operands and operators.

Dataflow Modeling

```
//Dataflow description of a 2-to-4-line decoder
module decoder_df (A,B,E,D);
    input A,B,E;
    output [0:3] D;
    assign D[0] = ~(~A & ~B & ~E),
           D[1] = ~(~A & B & ~E),
           D[2] = ~(A & ~B & ~E),
           D[3] = ~(A & B & ~E);
endmodule
```

A 2-to-1 line multiplexer with data inputs A and B, select input S, and output Y is described with the continuous assignment

```
assign Y = (A & S) | (B & ~S)
```

Dataflow Modeling

```
//Dataflow description of 4-bit adder
module binary_adder (A,B,Cin,SUM,Cout);
    input [3:0] A,B;
    input Cin;
    output [3:0] SUM;
    output Cout;
    assign {Cout,SUM} = A + B + Cin;
endmodule
```

```
//Dataflow description of a 4-bit comparator.
module magcomp (A,B,ALTB,AGTB,AEQB);
    input [3:0] A,B;
    output ALTB,AGTB,AEQB;
    assign ALTB = (A < B),
           AGTB = (A > B),
           AEQB = (A == B);
endmodule
```

Dataflow Modeling

- The addition logic of 4 bit adder is described by a single statement using the operators of addition and concatenation.
- The plus symbol (+) specifies the binary addition of the 4 bits of **`A`** with the 4 bits of **`B`** and the one bit of **`Cin`**.
- The target output is the concatenation of the output carry **`Cout`** and the four bits of **`SUM`**.
- Concatenation of operands is expressed within braces and a comma separating the operands. Thus, {**`Cout,SUM`**} represents the 5-bit result of the addition operation.

Dataflow Modeling

- Dataflow Modeling provides the means of describing combinational circuits by their function rather than by their gate structure.
- **Conditional operator (?:)**

  condition ? true-expression : false-expression;
- A 2-to-1 line multiplexer

```
assign OUT = select ? A : B;
```

```
//Dataflow description of 2-to-1-line mux
module mux2x1_df (A,B,select,OUT);
    input A,B,select;
    output OUT;
    assign OUT = select ? A : B;
endmodule
```

Behavioral Modeling

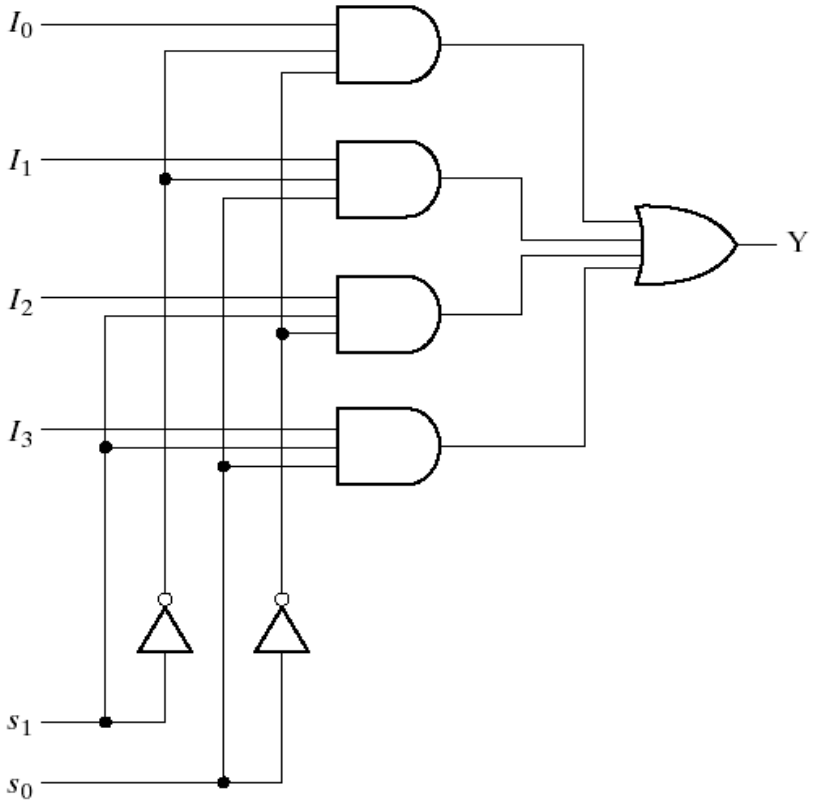
- Behavioral modeling represents digital circuits at a functional and algorithmic level.
- It is used mostly to describe sequential circuits but can also be used to describe combinational circuits.
- Behavioral descriptions use the keyword **`always`** followed by a list of procedural assignment statements.
- The target output of procedural assignment statements must be of the **`reg`** data type.
- A **`reg`** data type retains its value until a new value is assigned.

Behavioral Modeling

- The procedural assignment statements inside the **always** block are executed every time there is a change in any of the variable listed after the @ symbol. (Note that there is no ";" at the end of **always** statement)

```
//Behavioral description of 2-to-1-line multiplexer
module mux2x1_bh(A,B,select,OUT);
    input A,B,select;
    output OUT;
    reg OUT;
    always @(select or A or B)
          if (select == 1) OUT = A;
          else OUT = B;
endmodule
```

Behavioral Modeling



(a) Logic diagram

s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

4-to-1 line multiplexer

Behavioral Modeling

```
//Behavioral description of 4-to-1 line mux

module mux4x1_bh (i0,i1,i2,i3,select,y);
    input i0,i1,i2,i3;
    input [1:0] select;
    output y;
    reg y;
    always @(i0 or i1 or i2 or i3 or select)
              case (select)
                 2'b00: y = i0;
                 2'b01: y = i1;
                 2'b10: y = i2;
                 2'b11: y = i3;
              endcase
endmodule
```

Behavioral Modeling

- In 4-to-1 line multiplexer, the select input is defined as a 2-bit vector and output y is declared as a reg data.
- The always block has a sequential block enclosed between the keywords **`case`** and **`endcase`**.
- The block is executed whenever any of the inputs listed after the @ symbol changes in value.

Structural code

Ex1:

```
module mynand (x, a, b);
  input a, b;
  output x;
  wire y;

  and A1 (y,a,b);
  not A2 (x,y);

endmodule
```

Data Flow code

Ex2:

```
module mynand (x, a, b);
  input a, b;
  output x;

assign y = (a==1 & b==1)?1:0;
assign x = (y==0)?1:0;

endmodule
```

Writing a Test Bench

- A test bench is an verilog program used for applying stimulus to an verilog design in order to test it and observe its response during simulation.
- In addition to the always statement, test benches use the **`initial`** statement to provide a stimulus to the circuit under test.
- The **`always`** statement executes repeatedly in a loop. The initial statement executes only once starting from simulation time=0 and may continue with any operations that are delayed by a given number of units as specified by the symbol #.

Writing a Test Bench

```
initial begin
    A=0; B=0;
    #10 A=1;
    #20 A=0; B=1;
end
```

- The block is enclosed between **begin** and **end**. At time=0, A and B are set to 0. 10 time units later, A is changed to 1. 20 time units later (at t=30) a is changed to 0 and B to 1.

Writing a Test Bench

- Inputs to a 3-bit truth table can be generated with the initial block

```
initial begin
     D = 3'b000;
     repeat (7);
     #10 D = D + 3'b001;
 end
```

- The 3-bit vector D is initialized to 000 at time=0. The keyword **`repeat`** specifies looping statement: one is added to D seven times, once every 10 time units.

Writing a Test-Bench

- A stimulus module is an verilog program that has the following form.

  module *testname*

  Declare local ***reg*** *and* ***wire*** *identifiers*

  Instantiate the design module under test.

  Generate stimulus using ***initial*** *and* ***always*** *statements*

  Display the output response.

  endmodule

- A test module typically has no inputs or outputs.
- The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local reg data type.
- The outputs of the design module that are displayed for testing are declared in the stimulus model as local wire data type.
- The module under test is then instantiated using the local identifiers.

Writing a Test-Bench

- The response to the stimulus generated by the **initial** and **always** blocks will appear at the output of the simulator as timing diagrams.
- It is also possible to display numerical outputs using Verilog *system tasks*.
  - **$display** – display one-time value of variables or strings with end-of-line return,
  - **$write** – same $display but without going to next line.
  - **$monitor** – display variables whenever a value changes during simulation run.
  - **$time** – displays simulation time
  - **$finish** – terminates the simulation
- The syntax for **$display,$write** and **$monitor** is of the form

  Task-name (format-specification, argument list);

  E.g. **$display**(%d %b %b, C,A,B);

  $display**(“time = %0d A = %b B=%b”,**$time,A,B);

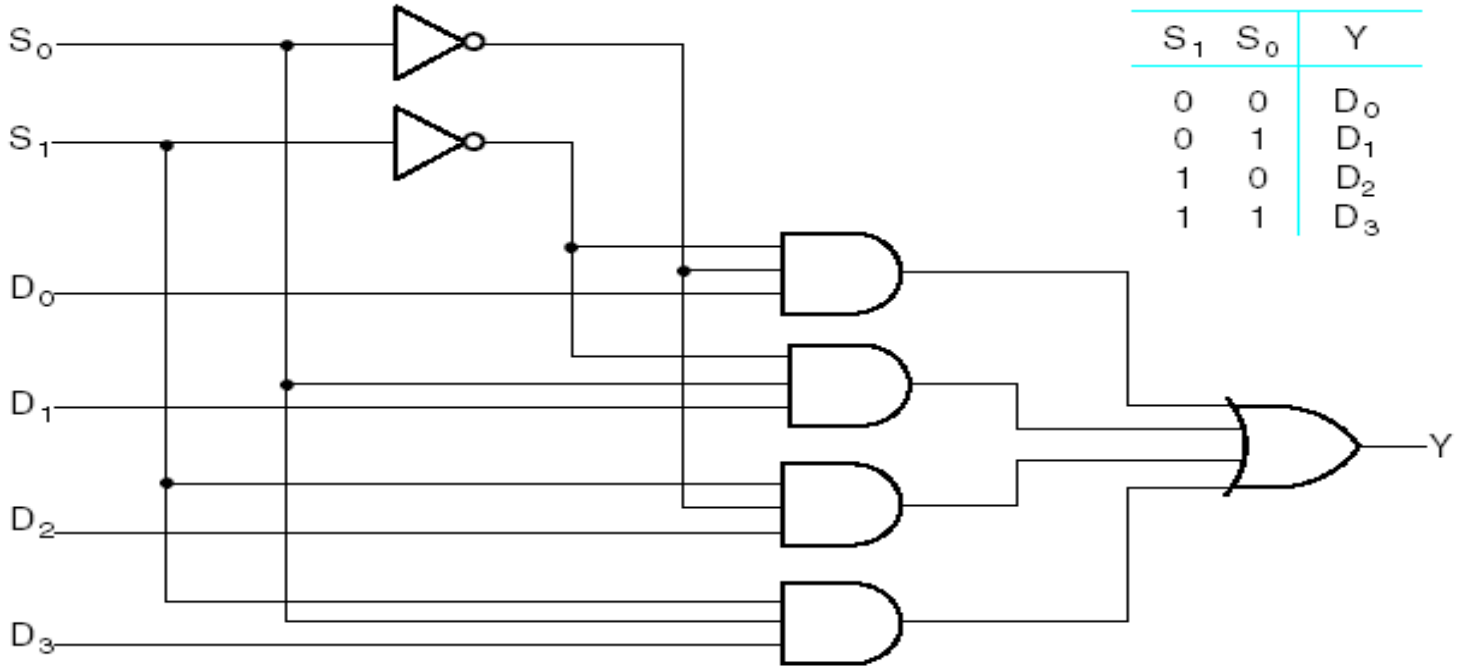
Writing a Test-Bench

```
//Dataflow description of 2-to-1-line multiplexer
module mux2x1_df (A,B,select,OUT);
    input A,B,select;
    output OUT;
    assign OUT = select ? A : B;
endmodule
```

Writing a Test-Bench

```
//Stimulus for mux2x1_df
module testmux;
  reg TA,TB,TS;  //inputs for mux
  wire Y;        //output from mux
  mux2x1_df mx (TA,TB,TS,Y);  // instantiate mux
  initial begin
     $monitor(”select=%b A=%b B=%b OUT=%b",TS,TA,TB,Y);
     TS = 1; TA = 0; TB = 1;
     #10 TA = 1; TB = 0;
     #10 TS = 0;
     #10 TA = 0; TB = 1;
   end
endmodule
```

4-to-1 Multiplexer



Function table

S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Fig. 3-19 4-to-1-Line Multiplexer

4-to-1 Multiplexer

```
//4-to-1 Mux: Structural Verilog
module mux_4_to_1_st_v(S,D,Y);
  input [1:0]S;
  input [3:0]D;
  output Y;
  wire [1:0]not_s;
  wire [0:3]N;
  not g0(not_s[0],S[0]),g1(not_s[1],S[1]);
  and g2(N[0],not_s[0],not_s[1],D[0]),
      g3(N[1],S[0],not_s[1],D[0]),
      g4(N[2],not_s[0],S[1],D[0]),
      g5(N[3],S[0],S[1],D[0]);
  or g5(Y,N[0],N[1],N[2],N[3]);
endmodule
```

4-to-1 Multiplexer – Data Flow

```
//4-to-1 Mux: Dataflow description
module mux_4_to_1(S,D,Y);
  input [1:0]S;
  input [3:0]D;
  output Y;
  assign Y = (~S[1]&~S[0]&D[0])|(~S[1]&S[0]&D[1])
      |(S[1]&~S[0]&D[2])|(S[1]&S[0]&D[3]);
endmodule
```

```
//4-to-1 Mux: Conditional Dataflow description
module mux_4_to_1(S,D,Y);
  input [1:0]S;
  input [3:0]D;
  output Y;
  assign Y = (S==2'b00)?D[0] : (S==2'b01)?D[1] :
  (S==2'b10)?D[2] : (S==2'b11)?D[3]:1'bx;;
endmodule
```

4-to-1 Multiplexer

```
//4-to-1 Mux: Dataflow Verilog Description
module mux_4_to_1(S,D,Y);
 input [1:0]S;
 input [3:0]D;
 output Y;
 assign Y=S[1]?(S[0]?D[3]:D[2]):(S[0]?D[1]:D[0]);
endmodule
```