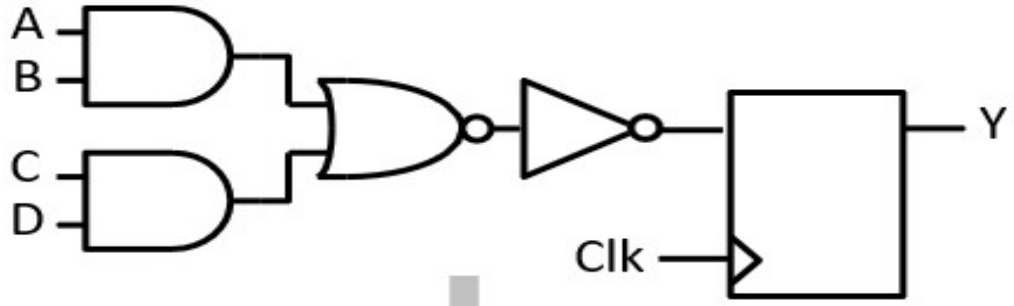
DIGITAL SYSTEMS DESIGN LAB

BECE102P

Introduction: Digital Logic Design

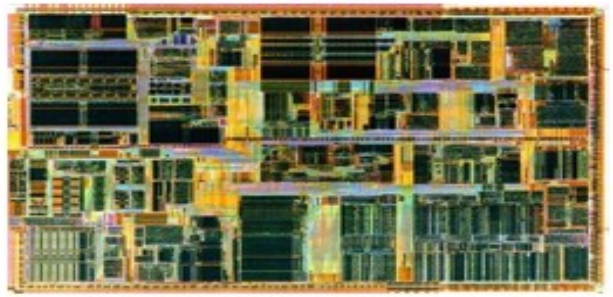
- Conventional Approach:
  - Schematic Entry → good for fairly small designs

    (Draw K-maps, optimize the Boolean logic, draw the schematic)



(10 gates)

  - Possible for large designs?
    - **NO!**



(10,000,000 gates)

Introduction: Why HDL ?

- Schematic entry not feasible for large designs:
  - Time consuming to draw the schematic for millions of gates
  - Prone to mistakes
  - Difficult design entry and sharing
  - Different design entry tools to learn
  - Tools not compatible (hard to convert the design from one to another)
  - Not easy to modify

Solution

- Describe the design in text Hardware Description Language (HDL)
- Just describe the design **“behavior”** not the detailed gate-level logic
- Gate-level logic is generated automatically by a **“synthesis”** tool

Advantages of HDL Coding

- Designer describes what the hardware should do without actually designing the hardware itself
- Designers develop an executable functional specification that documents the exact behavior of all the components and their interfaces
- Designers can make decisions about cost, performance, power, and area earlier in the design process

HDL Coding

- A Hardware Description Language is a high-level programming language that offers special constructs, used to model microelectronic circuits
- Two standard HDLs:
  - VHDL (**V**ery high-speed integrated circuit **HDL**)
  - Verilog
- Verilog:
  - Developed by Philip Moorby in 1985 as a proprietary language
  - Open to public by Cadence Design Systems in 1990
  - IEEE standard in 1995 and revised in 2001

Operators

Arithmetic Operators	+, -, *, /, %
Relational Operators	<, <=, >, >=
Logical Equality Operators	==, !=
Case Equality Operators	===, !==
Logical Operators	!, &&, ||
Bit-Wise Operators	~, &, |, ^(xor), ~^(xnor)
Unary Reduction Operators	&, ~&, |, ~|, ^, ~^
Shift Operators	>>, <<
Conditional Operators	? :
Concatenation Operator	{ }
Replication Operator	{ { } }

Different styles of modelling

1.Structural or gate-level modelling

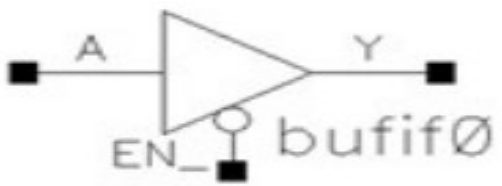
- instantiation of primitives and modules

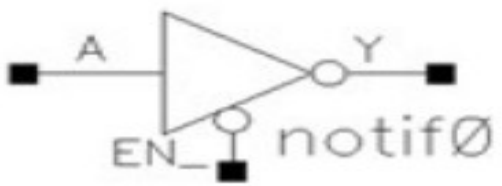
2.Dataflow-continuous assignments

3.Behavioral-procedural assignments

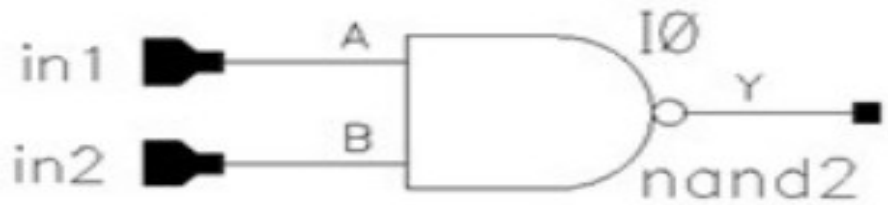
Structural or gate-level modelling
Verilog Primitives

- Basic logic gates only
  - and
  - or
  - not
  - buf
  - xor
  - nand
  - nor
  - xnor
  - bufif1, bufif0
  - notif1, notif0

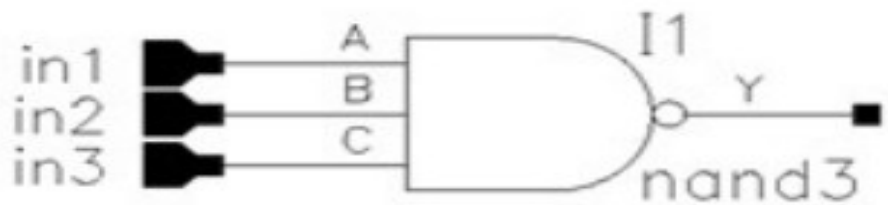




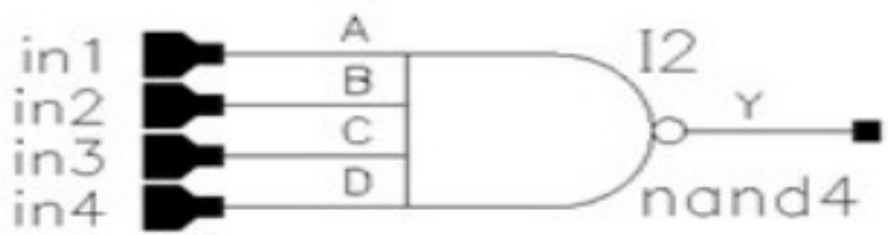
Primitive pins are Expandable




```
nand (y, in1, in2) ;
```




```
nand (y, in1, in2, in3) ;
```




```
nand (y, in1, in2, in3, in4) ;
```

Basic structure

module module_name(portlist);

declarations:

Port declarations(input,output,inout…);

datatype declarations(reg,wire,parameter…);

Statements:

Initial block

always block } Behavioral

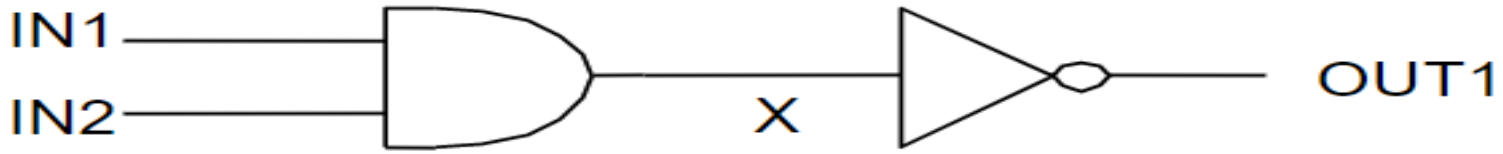
module instantiation

gate instantiation } Structural

Continuous assignment is used in data-flow

Gate Level Modeling

- Net-list description
  - built-in primitives gates



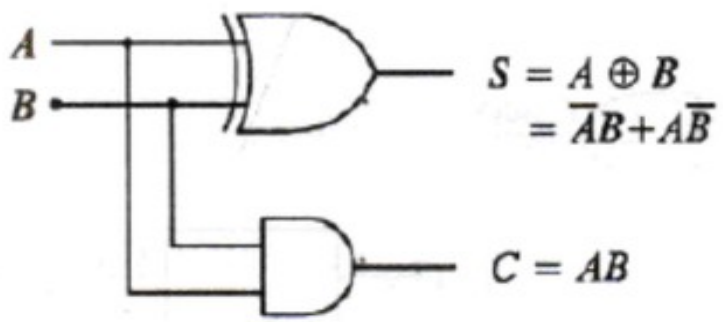

```
module  my_gate(OUT1, IN1, IN2);
    output  OUT1;
    input  IN1, IN2;
    wire  X;
    and   (X, IN1, IN2);
    not  (OUT1, X);
endmodule
```

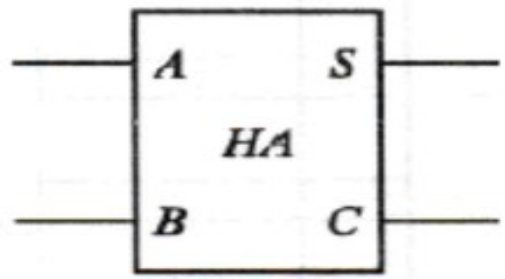
Any internal net must be defined as wire

Half-Adder:

- A combinational circuit which adds two one-bit binary numbers is called a half-adder.

Inputs		Outputs	
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

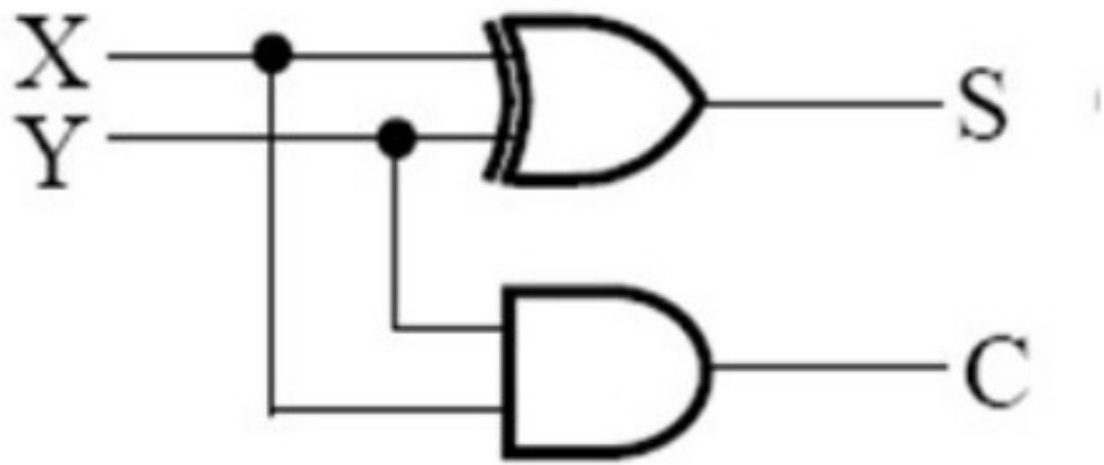




- The sum column resembles like an output of the XOR gate.
- The carry column resembles like an output of the AND gate.

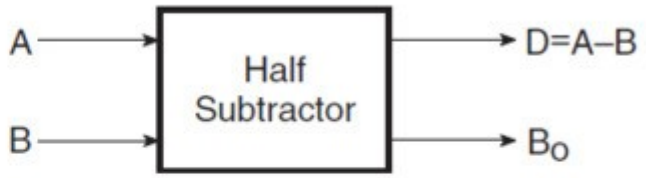
A Half Adder

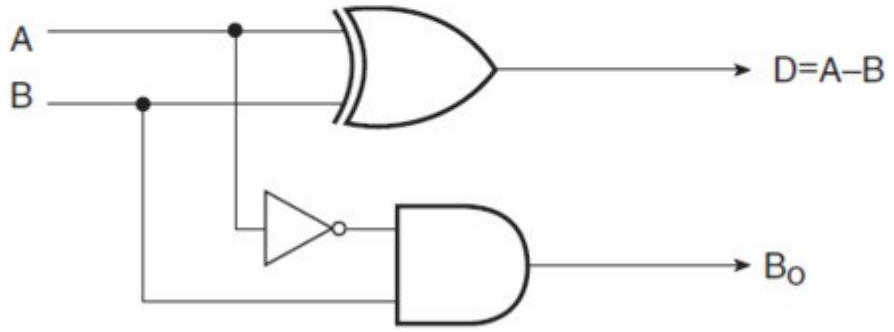
```
module hadd (S, C, X, Y);
    input X, Y;
    output S, C;

    xor (S, X, Y);
    and (C, X, Y);
endmodule
```



Half-Subtractor

A	B	D	B_O
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0





Dataflow Modelling

- Gate level modeling works best for circuits having a limited number of gates. It allows the designer to instantiate and connect each gate individually.

- But as the circuit becomes bigger, Gate level modeling starts to become tough.

- Thus, we shift to the next level of abstraction in Verilog, ***Dataflow modeling***.

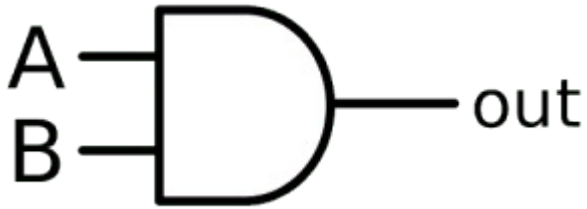
Dataflow Modelling

- Dataflow modeling provides the means of describing combinational circuits by their function rather than by their gate structure.

- Dataflow modeling uses a number of operators that act on operands to produce the desired results.

- The continuous assignment Statement is the main construct of dataflow modeling and is used to drive (assign) value to the net. It starts with the keyword **assign**

Definition of Net



- Nets are a datatype in Verilog that represent connections between hardware elements. Nets don’t store values.

- They have the values of the drivers. For instance, in the figure, the net **out** connected to the output is driven by the driving output value A&B

- Verilog has different net types, such as **wire**, **trior**, **wand**, **trireg,** etc.
- Nets need to be declared too.
- They are primarily declared using the keyword **wire.**

Verilog HDL Operators

Verilog HDL Operators

Symbol	Operation
+	binary addition
-	binary subtraction
&	bit-wise AND
|	bit-wise OR
^	bit-wise XOR
~	bit-wise NOT
==	equality
>	greater than
<	less than
{}	concatenation
? :	conditional

Basic structure

module module_name(portlist);

declarations:

Port declarations(input,output,inout…);

datatype declarations(reg,wire,parameter…);

Statements:

Initial block

always block } Behavioral

module instantiation

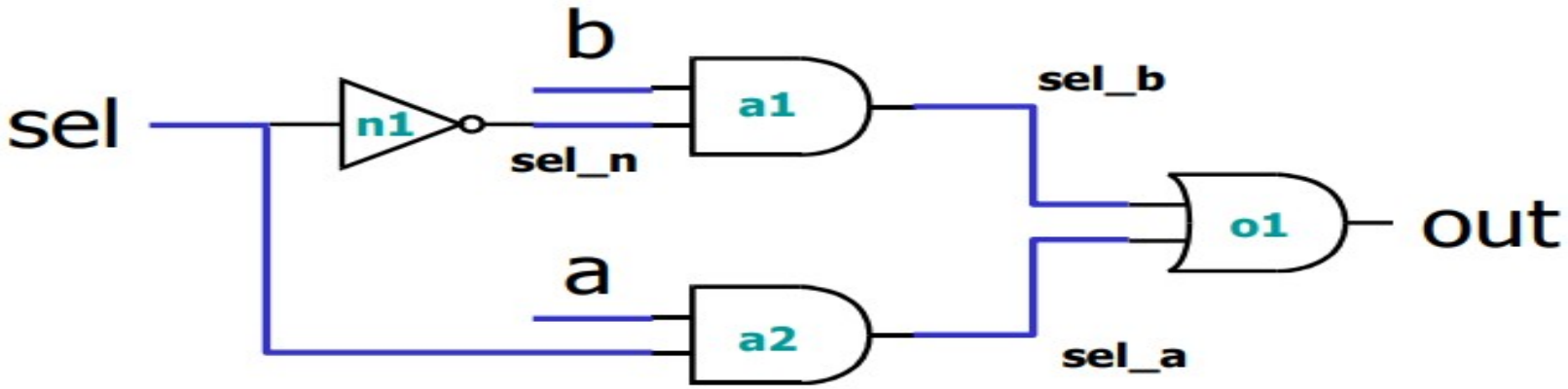
gate instantiation } Structural

Continuous assignment is used in data-flow

Example For ..?

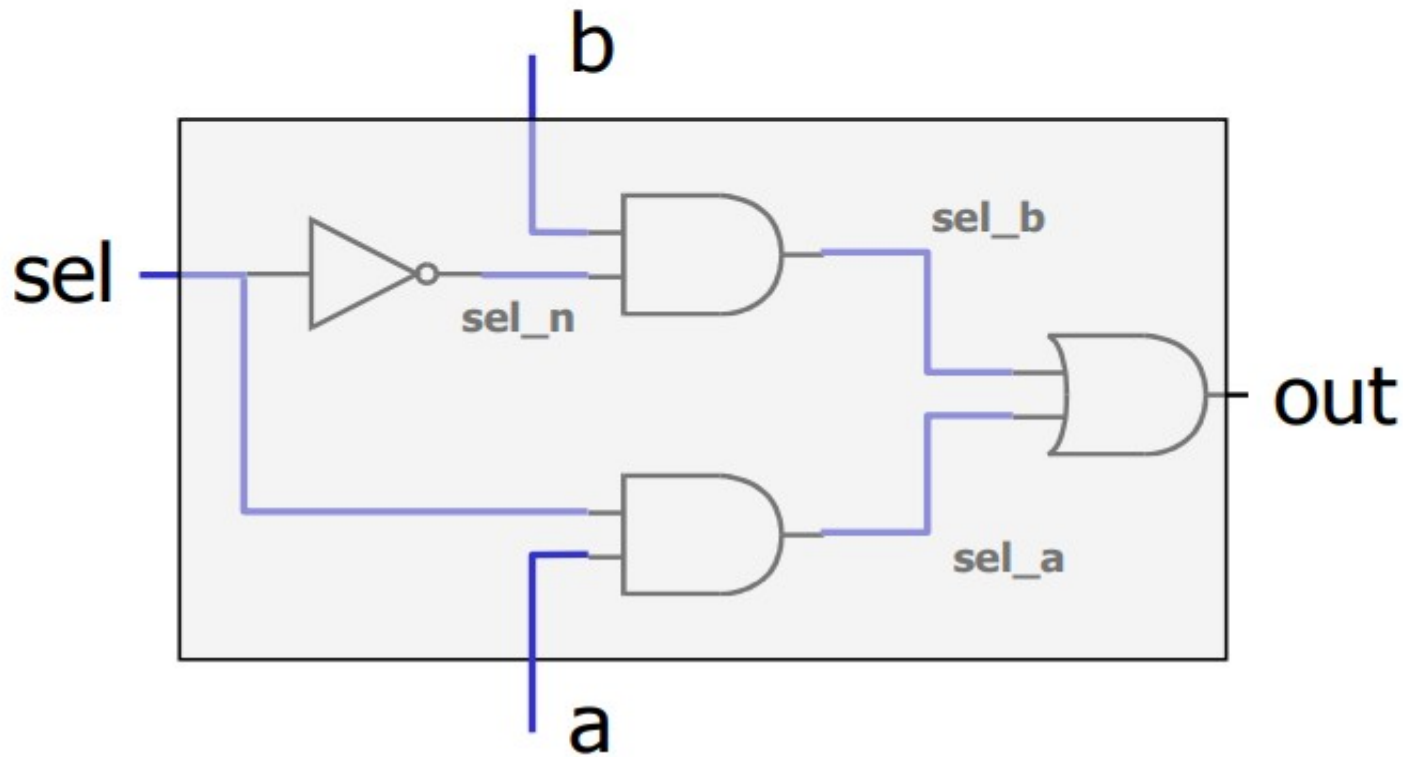
- Example:

```
not  n1(sel_n, sel);
and a1(sel_b, b, sel_b);
and a2(sel_a, a, sel);
or   o1(out, sel_b, sel_a);
```



- **Dataflow**: Specify output signals in terms of input signals
- Example:

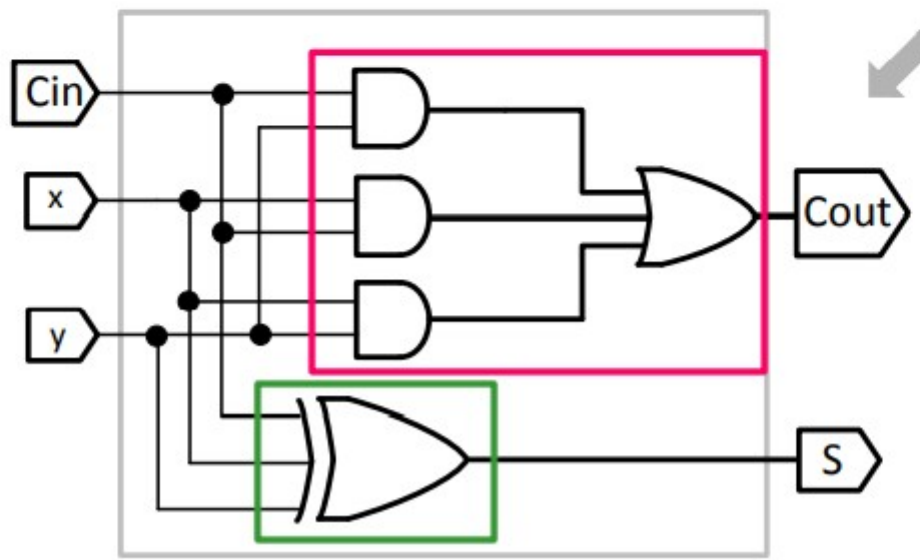
```
assign out = (sel & a) | (~sel & b);
```



❖Example: Full Adder, same circuit, two descriptions:

```
module Adder (Cin, x, y, S, Cout)
   input  x, y, Cin;
   output  S, Cout;
   wire S, Cout;

   assign S = x ^ y ^ Cin;

   assign Cout = (x & y)|(x & Cin)|(y & Cin);

endmodule
```

```
module Adder (Cin, x, y, S, Cout)
   input  x, y, Cin;
   output  S, Cout;
   wire S, Cout;


   assign {Cout, S} = x + y + Cin;

endmodule
```



x	y	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

- Uses continuous assignment statement
  - Format: assign [delay] net = expression;
  - Example: assign sum = a ^ b;
- **Delay**: Time duration between assignment from RHS to LHS
- All continuous assignment statements execute concurrently
- Order of the statement does not impact the design

- Delay can be introduced
  - Example: assign #2 sum = a ^ b;
  - "#2" indicates 2 time-units
  - No delay specified : 0 (default)
- Associate time-unit with physical time
  - `timescale time-unit/time-precision
  - Example: `timescale 1ns/100 ps
- Timescale

  `timescale 1ns/100ps
  - 1 Time unit = 1 ns
  - Time precision is 100ps (0.1 ns)

- Example:

```
`timescale 1ns/100ps
module HalfAdder (A, B, Sum, Carry);
  input A, B;
  output Sum, Carry;
  assign #3 Sum = A ^ B;
  assign #6 Carry = A & B;
endmodule
```


0
9
18
27
36
45
54
63
72
81
A
B
Sum
Carry
3
ns
6 ns

Data Type Declaration for Ports & Variables

Verilog Modelling	Input	Output	InOut	Variables	
				LHS	RHS
Gate Level /Structured Level	Wire	Wire	Wire	Wire	Wire/Reg
Data Flow/ RTL	Wire	Wire	Wire	Wire	Wire/Reg
Behaviour	Wire	Reg LHS	Wire	Reg	Wire/Reg

Behavioral Modeling

- Behavioral modeling represents digital circuits at a functional and algorithmic level. It is used mostly to describe sequential circuits, but can be used to describe combinational circuits. Here the behavioral modeling concept will be presented for combinational circuits.

- Behavioral description use the keyword always @ followed by a list of procedural assignment statements. The target output of procedural assignment statement must be of the reg data type

$$
\begin{aligned}
\text{Carry} &= A\,B + A\,C_{in} + B\,C_{in} \\
&= A\,B + A\,C_{in}(B + \bar{B}) + B\,C_{in}(A + \bar{A}) \\
&= A\,B + A\,B\,C_{in} + A\,\bar{B}\,C_{in} + A\,B\,C_{in} + \bar{A}\,B\,C_{in} \\
&= A\,B(1 + C_{in} + C_{in}) + A\,\bar{B}\,C_{in} + \bar{A}\,B\,C_{in} \\
&= A\,B + A\,\bar{B}\,C_{in} + \bar{A}\,B\,C_{in} \\
&= A\,B + C_{in}(A\,\bar{B} + \bar{A}\,B) \\
&= A\,B + C_{in}(A \oplus B)
\end{aligned}
$$

$A + A = A$

$1 + A = 1$

Behavioral Modeling

The construct of Verilog behavioral modeling consists of three main parts:

1.Module declaration

2.Port list

3.Procedural statements

There are three different ways we can proceed with Verilog coding for full adder:

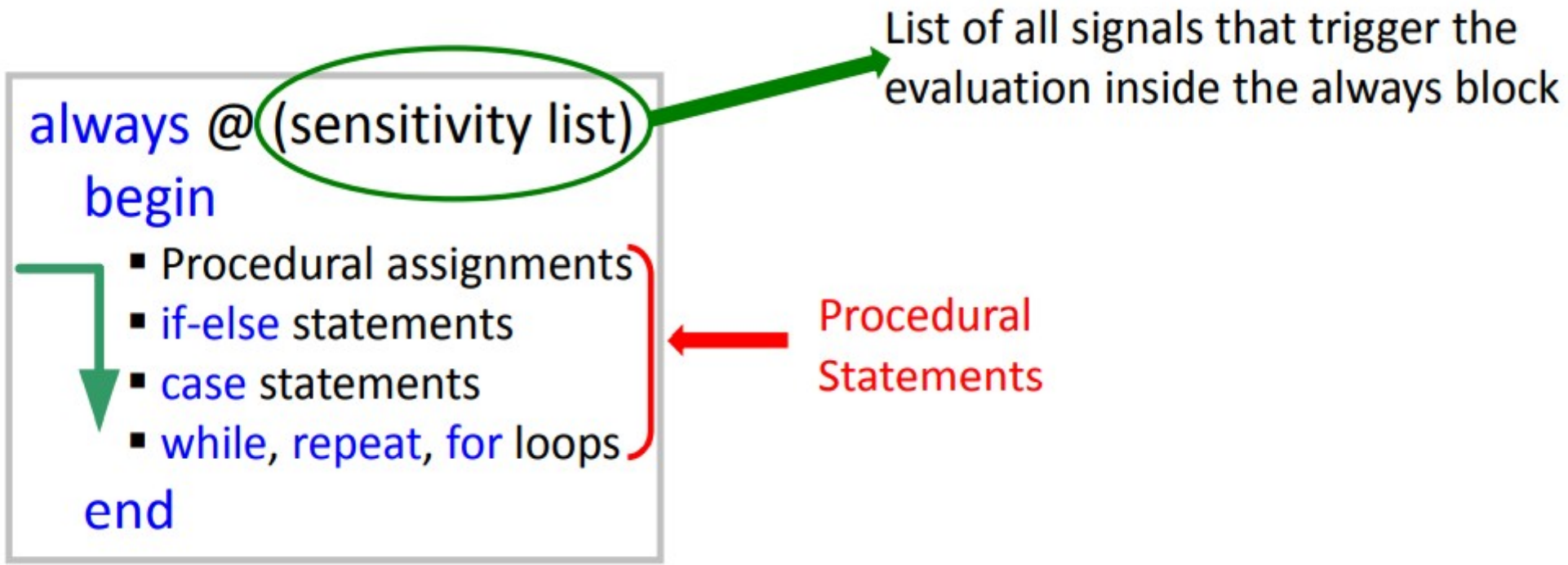
4.Using an always statement

5.Case statements

6.If-else statements

Procedural Statements

- Evaluated in the order in which they appear in the code (sequential)
- Should be inside an "**always**" block
- An "**always**" block contains one or more procedural statements



Procedural Statements: Half-Adder

❖Example:

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Type: "reg"

```
module Adder (x, y, S, C)
   input x,y;
   output S,C;
   reg S, C;
   always @ (x, y)
      begin
         S = x ^ y;
         C = x & y;
      end
endmodule
```

If either x or y changes, the statements inside the always block are evaluated.

```
module Adder (x, y, S, C)
   input x,y;
   output S,C;
   wire S, C;
   assign S = x ^ y;
   assign C = x & y;
endmodule
```

Type: "wire"

❑ Anything on the RHS should be on the sensitivity list

➢ always @(*) ➡ Automatically considers all signals on the RHS in the sensitivity list

Verilog code for full adder – Using always statement

```
full_adder( A, B, Cin, S, Cout);

 input wire A, B, Cin;
 output reg S, Cout;

 always @(A or B or Cin)
  begin
   S = A ^ B ^ Cin;
   Cout = A&B | (A^B) & Cin;
  end
endmodule
```

Verilog code for full adder – Using case statement

```
full_adder(input wire A, B, Cin, output reg S, output reg Cout);

 always @(A or B or Cin)
  begin

    case (A | B | Cin)
      3'b000: begin S = 0; Cout = 0; end
      3'b001: begin S = 1; Cout = 0; end
      3'b010: begin S = 1; Cout = 0; end
      3'b011: begin S = 0; Cout = 1; end
      3'b100: begin S = 1; Cout = 0; end
      3'b101: begin S = 0; Cout = 1; end
      3'b110: begin S = 0; Cout = 1; end
      3'b111: begin S = 1; Cout = 1; end
    endcase

  end

endmodule
```

Verilog code for full adder – Using if-else statement

```
module full_adder( A, B, Cin, S, Cout);

input wire A, B, Cin;
output reg S, Cout;

always @(A or B or Cin)
begin
 if(A==0 && B==0 && Cin==0)
  begin
   S=0;
   Cout=0;
  end

 else if(A==0 && B==0 && Cin==1)
  begin
   S=1;
   Cout=0;
  end
```

```
else if(A==0 && B==1 && Cin==1)
 begin
  S=0;
  Cout=1;
 end

else if(A==1 && B==0 && Cin==0)
 begin
  S=1;
  Cout=0;
 end

else if(A==1 && B==0 && Cin==1)
 begin
  S=0;
  Cout=1;
 end
```

```
 else if(A==1 && B==1 && Cin==0)
  begin
   S=0;
   Cout=1;
  end

 else if(A==1 && B==1 && Cin==1)
  begin
   S=1;
   Cout=1;
  end

end

endmodule
```

The Gray code:-

It is non weighted code in which each number differs from previous number by a single bit.

Decimal	Binary	Gray code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101

Decimal digit	BCD Code		
	8 4 2 1	4 2 2 1	5 4 2 1
0	0 0 0 0	0 0 0 0	0 0 0 0
1	0 0 0 1	0 0 0 1	0 0 0 1
2	0 0 1 0	0 0 1 0	0 0 1 0
3	0 0 1 1	0 0 1 1	0 0 1 1
4	0 1 0 0	1 0 0 0	0 1 0 0
5	0 1 0 1	0 0 1 1	1 0 0 1
6	0 1 1 0	1 1 0 0	1 0 1 0
7	0 1 1 1	1 1 0 1	
8	1 0 0 0	1 1 1 0	1 0 1 1
9	1 0 0 1	1 1 1 1	1 1 0 0

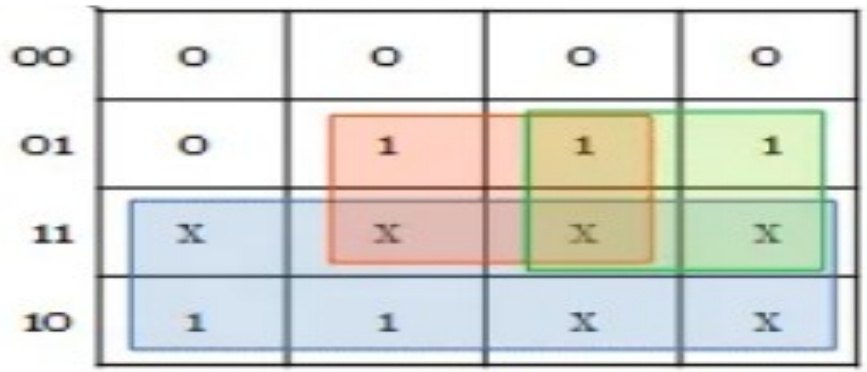
Decimal	Binay (BCD) 8 4 2 1	8 4 -2 -1
0	0 0 0 0	0000
1	0 0 0 1	0111
2	0 0 1 0	0110
3	0 0 1 1	0101
4	0 1 0 0	0100
5	0 1 0 1	1011
6	0 1 1 0	1010
7	0 1 1 1	1001
8	1 0 0 0	1000
9	1 0 0 1	1111

- Design a 4 bit **BCD to Excess-3** code converter and obtain its logic diagram

Decimal Number	BCD code (Input)				Excess-3 code (Output)			
	D_3	D_2	D_1	D_0	E_3	E_2	E_1	E_0
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

- From the truth table, the minterms are obtained for each outputs(E_3, E_2, E_1, E_0).
- $E_3 = \sum m(5, 6, 7, 8, 9)$,
- $E_2 = \sum m(1, 2, 3, 4, 9)$,
- $E_1 = \sum m(0, 3, 4, 7, 8)$,
- $E_0 = \sum m(0, 2, 4, 6, 8,)$

The minterms of each output in plotted in k-map and simplified expression is obtained.



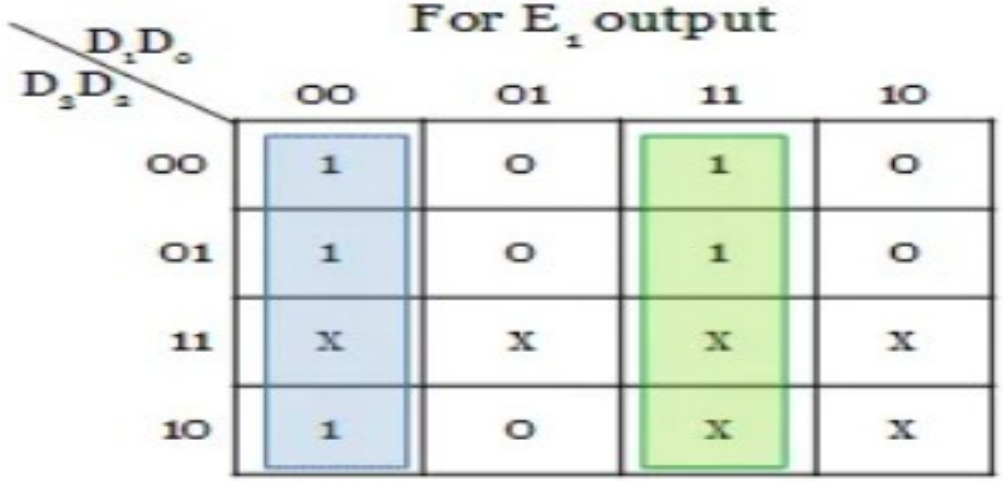
00	0	0	0	0
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

$$E_3 = D_3 + D_2D_0 + D_2D_1$$

00	0	1	1	1
01	1	0	0	0
11	X	X	X	X
10	0	1	X	X

$$E_2 = D_2\overline{D}_1\overline{D}_0 + \overline{D}_2D_0 + \overline{D}_2D_1$$

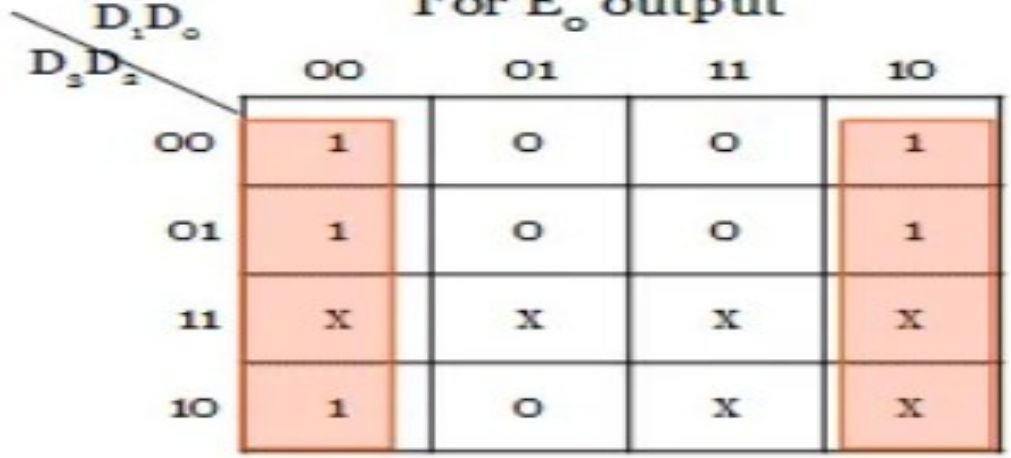
For E_1 output



D_3D_2 \ D_1D_0	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	X	X	X	X
10	1	0	X	X

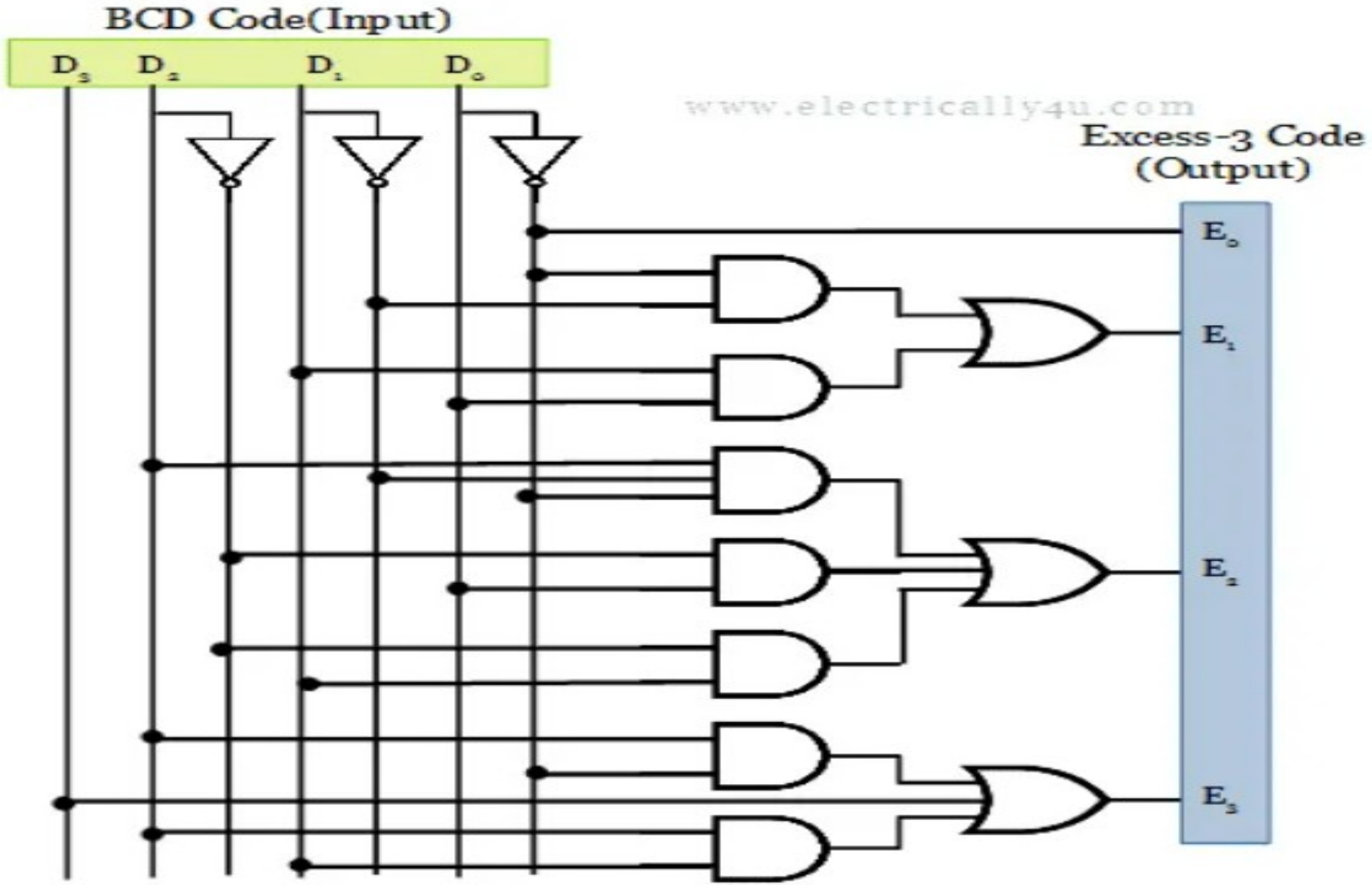
$$E_1 = \overline{D}_1\overline{D}_0 + D_1D_0$$

For E_0 output



D_3D_2 \ D_1D_0	00	01	11	10
00	1	0	0	1
01	1	0	0	1
11	X	X	X	X
10	1	0	X	X

$$E_0 = \overline{D}_0$$


BCD Code(Input)
D3 D2 D1 D0
www.electrically4u.com
Excess-3 Code
(Output)
E0
E1
E2
E3

```
module combex (exout, k, exip);
  output [1:0] exout;
  output [1:0] k ;
  input [2:0] exip;

  assign exout[0] = (exip[0] | (exip[1]  & exip[2]));
  assign exout[1]  = ~ exip[2];
  assign k = { exout[1], exout[0] };
endmodule
```

Verilog to Logic Diagram

```
module combex (exout, k, exip);
output [1:0] exout;
output [1:0] k ;
 input [2:0] exip;
assign exout[0] = (exip[0] | (exip[1]  & exip[2]));
assign exout[1]  = ~ exip[2];
assign k = { exout[1], exout[0] };
endmodule
```

What is a Testbench?

- Once the design of a chip is completed its correctness must be verified or checked.
- It means the designer must convince himself that the chip designed by him/her is functioning according to the specs.
- So, to verify or check the correctness of a chip, a code using any HDL is developed, called the “Test Bench”
- A test bench is defined as a HDL code to check the correctness of a chip, that provides the stimulus (input vectors) to the design

What is a Testbench?

- Define inputs as reg and output as wire.
- As we have to initialize the DUT inputs inside the procedural block(s),typically ‘initial’ where only ‘reg’ type variables can be assigned.
- This initial procedural block executes only once.
- But always block can also be used to generate some test inputs ,like a clock signal.

Test Bench Ex-Half adder

First let us write the design code using Verilog.

```
• module  half-adder(S,C,A,B);
   input  A,B;
   output S,C;
   assign  S= A^ B;  // data flow model
    assign C= A&B;
Endmodule
```

Let us now writ the testbench

```
module   tb_ half-adder;
    reg  A,B;
```

contd

```
wire S,C;
half-adder(S,C,A,B);//Instantiate the design
Initial
      begin
      #5   A = 0 ; B= 0;
      #5   A = 0 ; B= 1;
      #5   A = 1 ; B= 0;
      #5   A = 1 ; B= 1;
end
endmodule
```

Ex: Full adder

First let us write the design code using Verilog.

```
• module  full-adder(S,C0,A,B,C);
    input  A,B,C;
    output S,C0;
    assign  S= A^ B^C;  // data flow model
     assign C0= (A&B)| (B&C)|(C&A);
endmodule
```

Let us now write the test bench

```
module   tb_ full-adder;
     reg  A,B,C;
```

contd

```
  wire  S, C0 ;
   full_adder(S,C0,A,B,C); //Instantiation of the design
   initial
   begin
     # 5   A =0;B=0;C=O;
     # 5   A =1;B=0;C=O;
     # 5   A =1;B=1;C=1;
     # 5   A =0;B=1;C=1;
     # 5   A =1;B=0;C=1;
     # 5   A =0;B=0;C=1;
   end
endmodule
```

Test Bench

- In order to test a circuit, a test bench code is to be written which is commonly called stimulus
- It displays the output of the design based on the inputs

Primitive gates

- and,or,nand,nor,xor,xnor

  -First terminal is output followed by inputs

- not,buf

  -one or more outputs first followed by one input