Hardware Description Languages

HDL FOR COMBINATIONAL CIRCUITS

Chapters (3 and 4)

And For Sequential Circuits

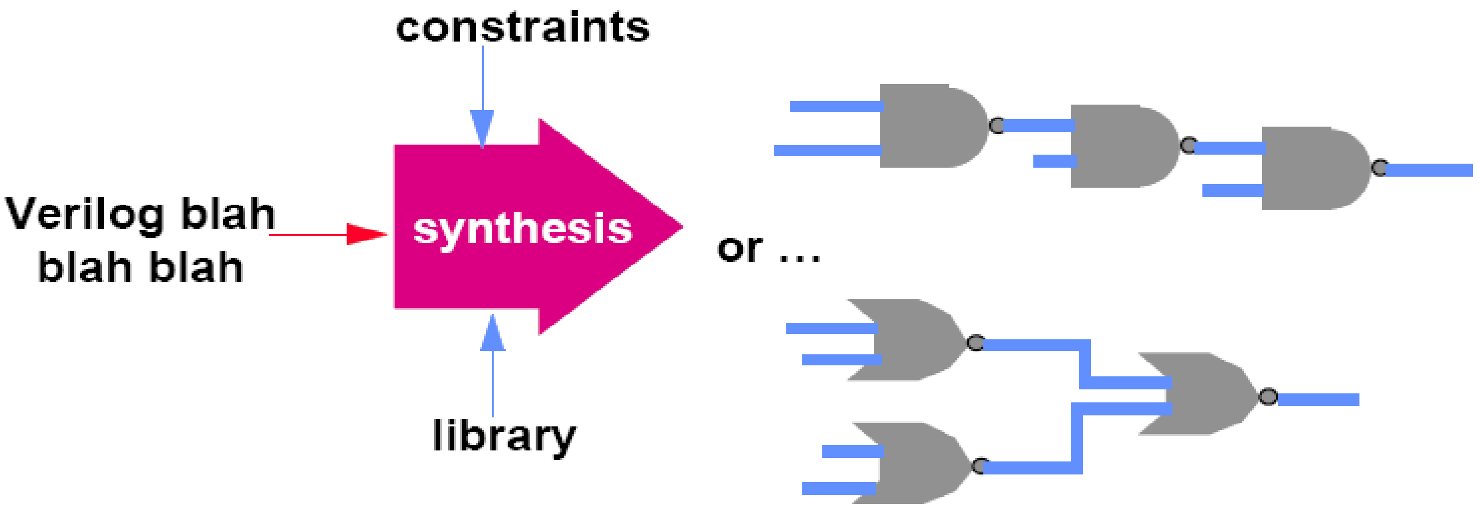
Chapter 5

Hardware Description Languages

- A hardware description language is a computer language that is used **to describe hardware**.
- Two HDLs are widely used
  - Verilog HDL
  - VHDL (Very High Speed Integrated Circuit Hardware Description Language)
- Write Verilog or VHDL code then make **Simulation** to test the circuit

Logic Synthesis

- Logic synthesis
  - A program that "designs" logic from abstract descriptions of the logic
    - takes constraints (e.g. size, speed)
    - uses a library (e.g. 3-input gates)
- How?
  - You write an "abstract" Verilog description of the logic
  - The synthesis tool provides alternative implementations



HDL VERILOG

- A module can be described in any one (or a combination)of the following modeling techniques:
- **Gate – level modeling** using instantiation of primitive gates and user- defined modules.
- **Data flow modeling** using continuous assignment statements with keyword **assign.**
- **Behavioral modeling** using procedural assignment statements with keyword **always**

Gate level (Structural) Representation

- Correspond to commonly used logic gates

- examples
  - AND gate → **and** (y, x1, x2);
  - OR gate → **or** (y, x1, x2, x3, x4);
  - NOT gate → **not** (y, x);

- Keywords: **and, or, not** are reserved

//HDL Example 3-1

```
//Description of the simple circuit of Fig. 3-37
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
```

port list

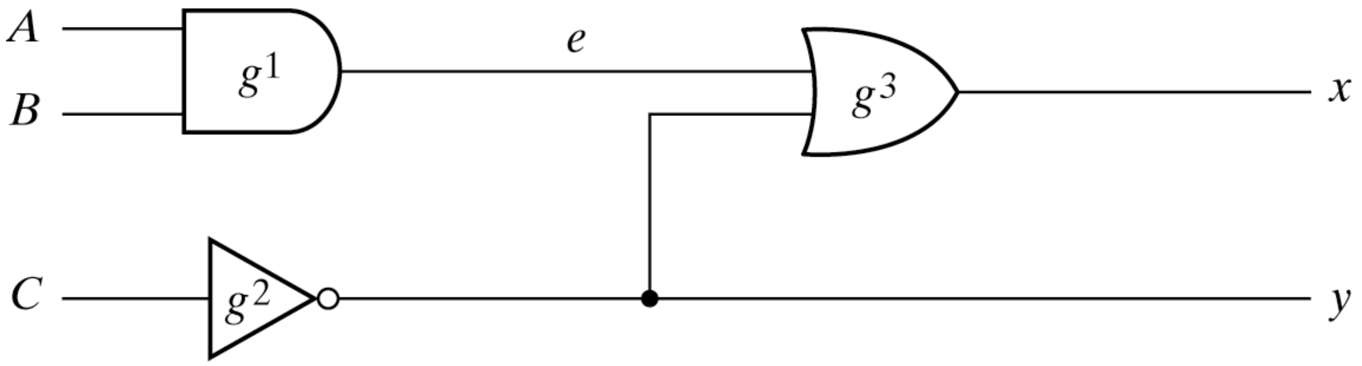


Fig. 3-37 Circuit to Demonstrate HDL

Boolean Algebra in HDL
Dataflow Modeling

$$x = A + BC + B'D$$

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

```
//HDL Example
//------------------------------
//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 & D);
   assign y = (~B & C) | (B & ~C & ~D);
endmodule
```

Bitwise Operators

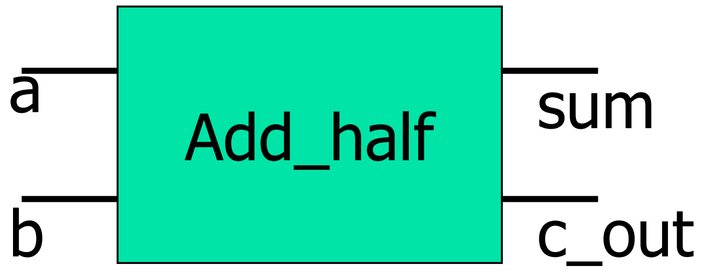
~(101011) = 010100

(010101) & (001100) = 000100

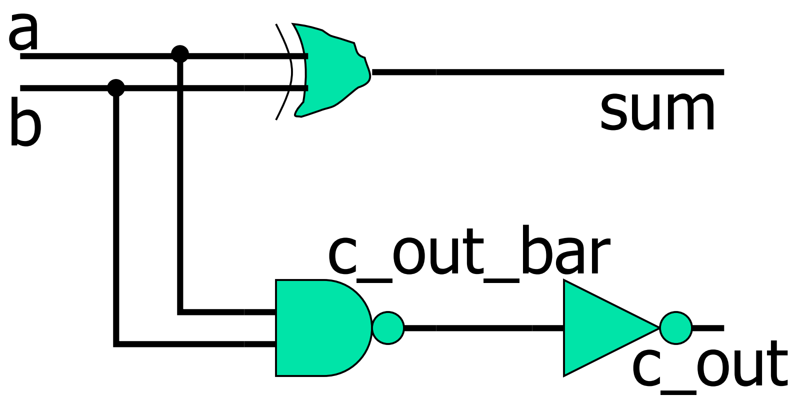
(010101) ^ (001100) = 011001

Symbol	Operator
~	Bitwise negation
&	Bitwise and
|	Bitwise inclusive or
^	Bitwise exclusive or
~^, ^~	Bitwise exclusive nor

Schematic Design



symbol



schematic

Input		output	
a	b	c_out	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$\rightarrow$

$$\text{sum} = a \oplus b$$

$$\text{c\_out} = a \bullet b$$

Boolean equations

Structure Description in Verilog

Module name *Module ports*

```
module Add_half ( sum, c_out, a, b );
    input       a, b;
    output      sum, c_out;
    wire        c_out_bar;

    xor Gate1 (sum, a, b);
    nand (c_out_bar, a, b);
    not (c_out, c_out_bar);
endmodule
```

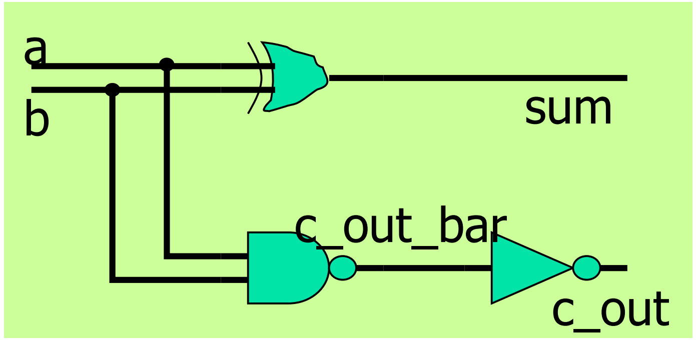
Declaration of port modes

Declaration of internal signal

Instance name

Instantiation of primitive gates

Verilog keywords



Using Vectored Signals

//4 bit adder; define A and B as a 4 bit vectors

```
module adder4 (carryin, X, Y, S, carryout);
    input carryin;
    input [3:0] X, Y;
    output [3:0] S;
    output carryout;
    wire [3:1] C;

    fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
    fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
    fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
    fulladd stage3 (C[3], X[3], Y[3], S[3], carryout);
    // note that fulladd must be implemented as a module
    // then, we can use four instants (copies) of it
endmodule
```

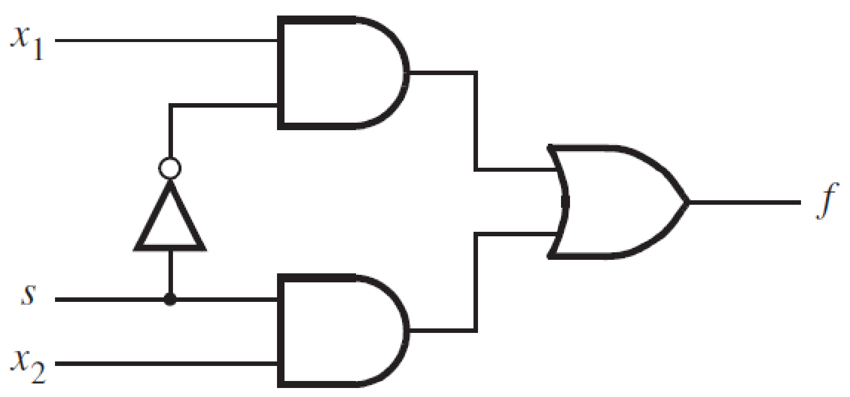
Vectored signals in port

Input / output declarations

Nodes inside the circuits

Access individual bits

Structural Approach



```
module  example1 (x1, x2, s, f);
    input  x1, x2, s;
    output  f;

    not (k, s);
    and (g, k, x1);
    and (h, s, x2);
    or (f, g, h);

endmodule
```

```
// Behavioral specification
module  example5 (input x1, x2, s, output reg f);

    always @(x1, x2, s)
        if (s == 0)
            f = x1;
        else
            f = x2;

endmodule
```

$$f = \overline{x}_2 + \overline{x}_1 x_3 + x_1 \overline{x}_3$$

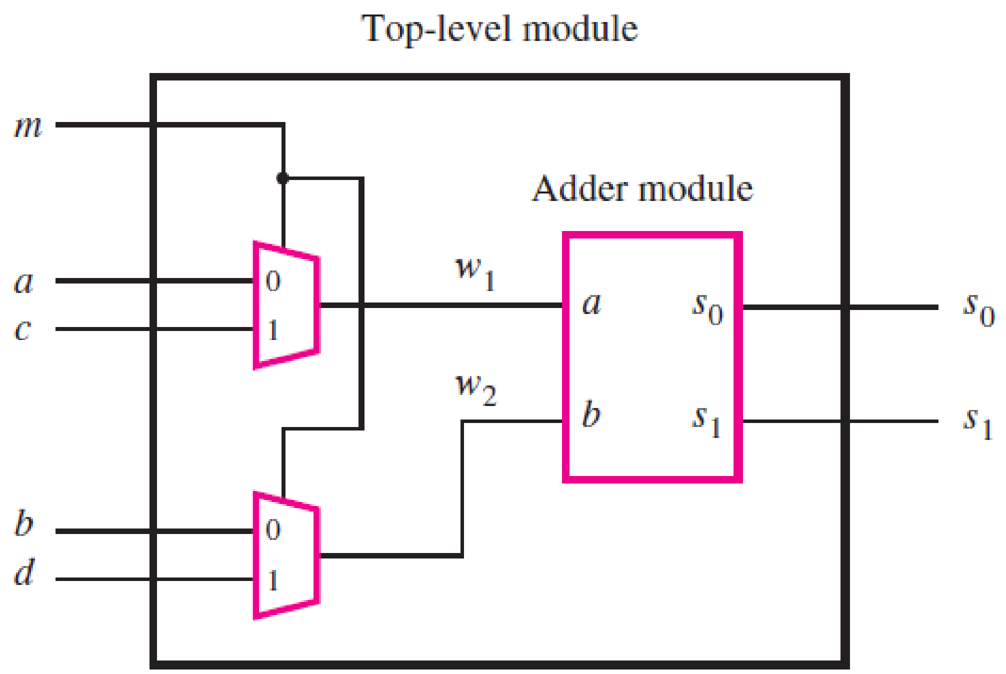
Verilog code that implements the circuit is

```
module prob2_49 (x1, x2, x3, f);
   input x1, x2, x3;
   output f;

   assign f = ~x2 | (~x1 & x3) | (x1 & ~x3);
endmodule
```

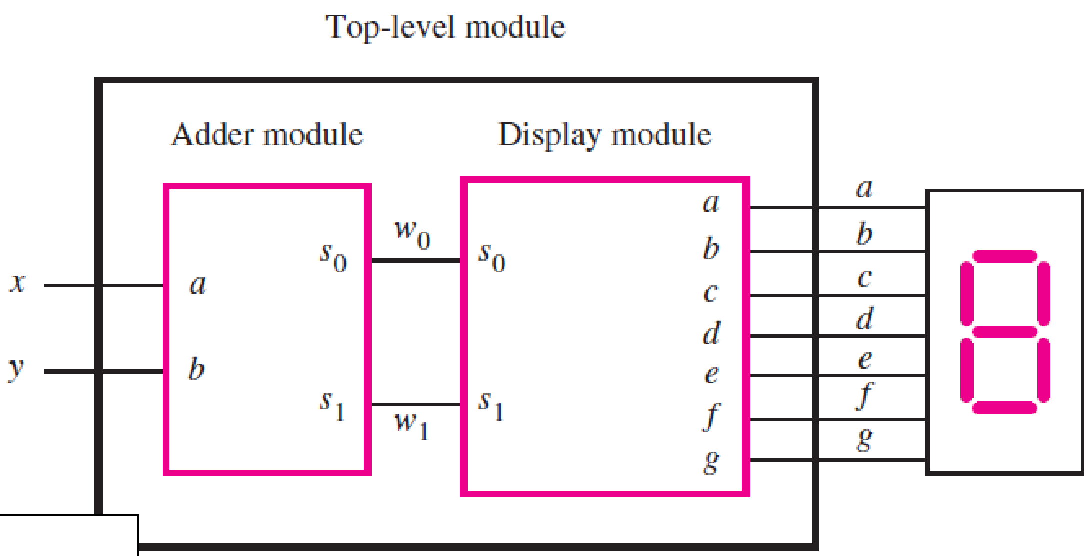
$$f = (x_1 + x_2 + x_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$$

Verilog code that implements the circuit is

```
module prob2_48 (x1, x2, x3, f);
   input x1, x2, x3;
   output f;

   or (g, x1, x2, x3);
   or (h, ~x1, ~x2, ~x3);
   and (f, g, h);

endmodule
```



```
module  shared (a, b, c, d, m, s1, s0);
    input a, b, c, d, m;
    output s1, s0;
    wire w1, w2;
    mux2to1 U1 (a, c, m, w1);
    mux2to1 U2 (b, d, m, w2);
    adder U3 (w1, w2, s1, s0);
endmodule

module  mux2to1 (x1, x2, s, f);
    input x1, x2, s;
    output f;
    assign f = (~s & x1) | (s & x2);
endmodule

module  adder (a, b, s1, s0);
    input a, b;
    output s1, s0;
    assign s1 = a & b;
    assign s0 = a ^ b;
endmodule
```



```
// An adder module
module  adder (a, b, s1, s0);
    input a, b;
    output s1, s0;

    assign s1 = a & b;
    assign s0 = a ^ b;

endmodule
```

```
// A module for driving a 7-segment display
module display (s1, s0, a, b, c, d, e, f, g);
    input s1, s0;
    output a, b, c, d, e, f, g;

    assign a = ~s0;
    assign b = 1;
    assign c = ~s1;
    assign d = ~s0;
    assign e = ~s0;
    assign f = ~s1 & ~s0;
    assign g = s1 & ~s0;

endmodule
```

```
module adder_display (x, y, a, b, c, d, e, f, g);
    input x, y;
    output a, b, c, d, e, f, g;
    wire w1, w0;

    adder U1 (x, y, w1, w0);
    display U2 (w1, w0, a, b, c, d, e, f, g);

endmodule
```

Representation of numbers

<size>’<base_format><number>

<size>: the number of bits

<base_format>: d (decimal), h (hexadecimal), o (octal), b (binary)

<number>: {0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f}

Number	*# Bits*	*Base*	*Decimal equiv.*	*stored*
2’b10	2	Binary	2	10
3’d5	3	Decimal	5	101
8’ha	8	Hex	10	00001010
12’h132	12	Hex	306	000100110010

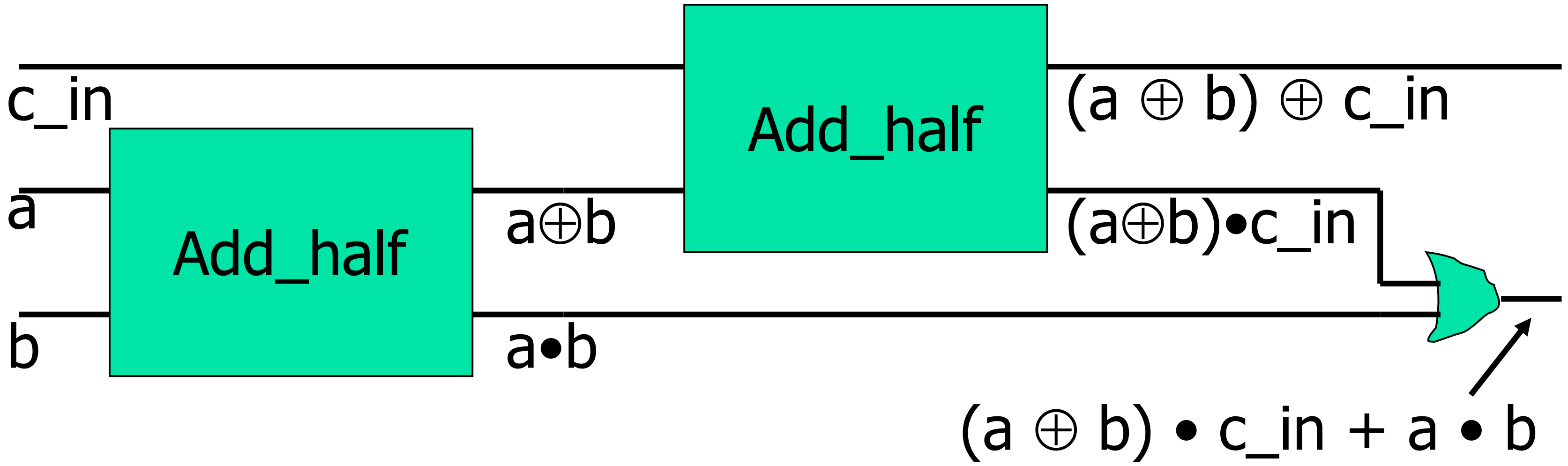
Full Adder ≈ 2 Half Adders

$\mathbf{sum_{HA} = a \oplus b}$

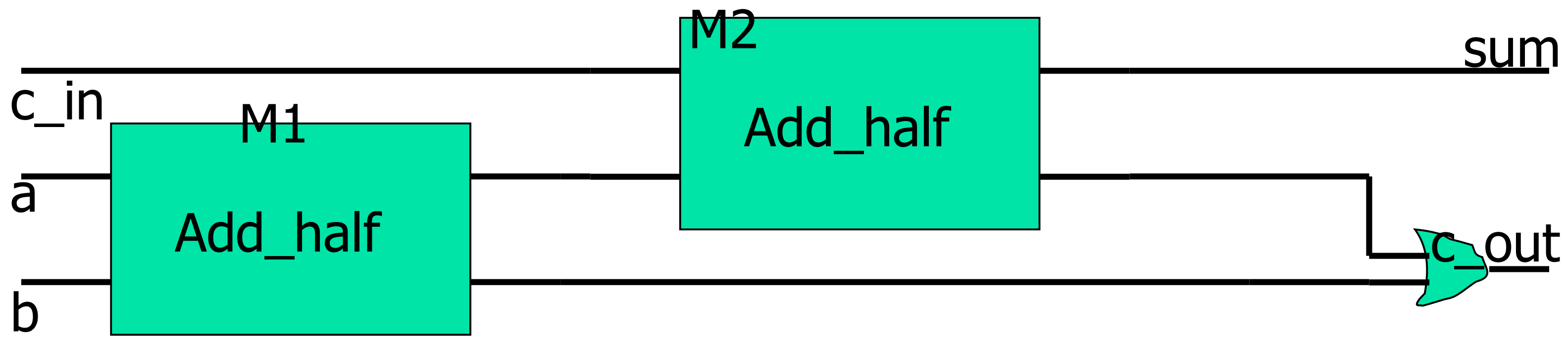
$\mathbf{c\_out_{HA} = a \bullet b}$

$\mathbf{sum_{FA} = (a \oplus b) \oplus c\_in}$

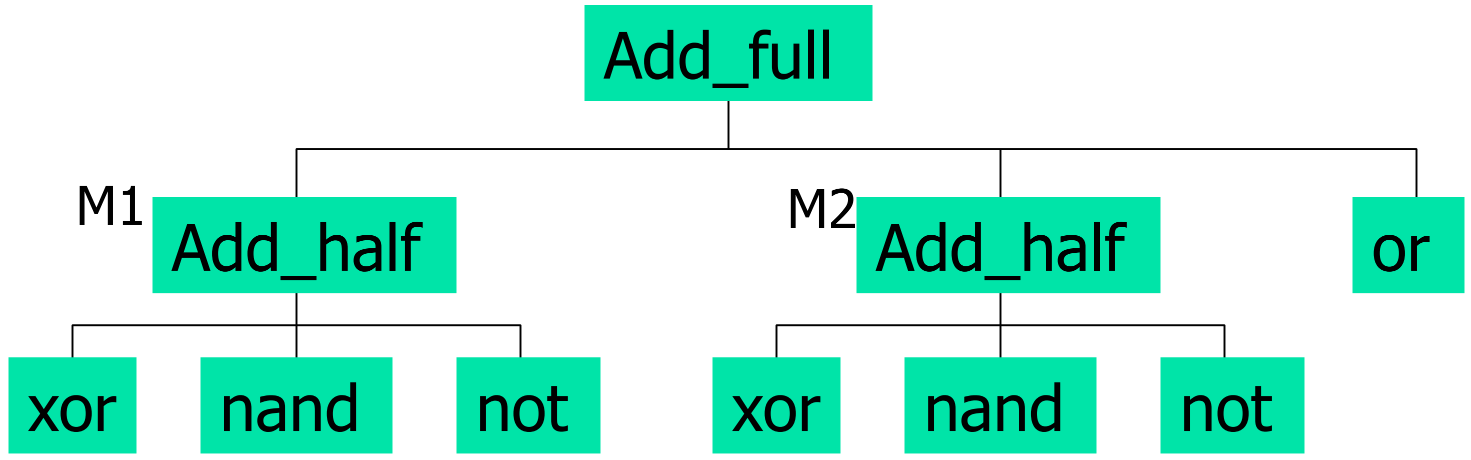
$\mathbf{c\_out_{FA} = (a \oplus b) \bullet c\_in + a \bullet b}$



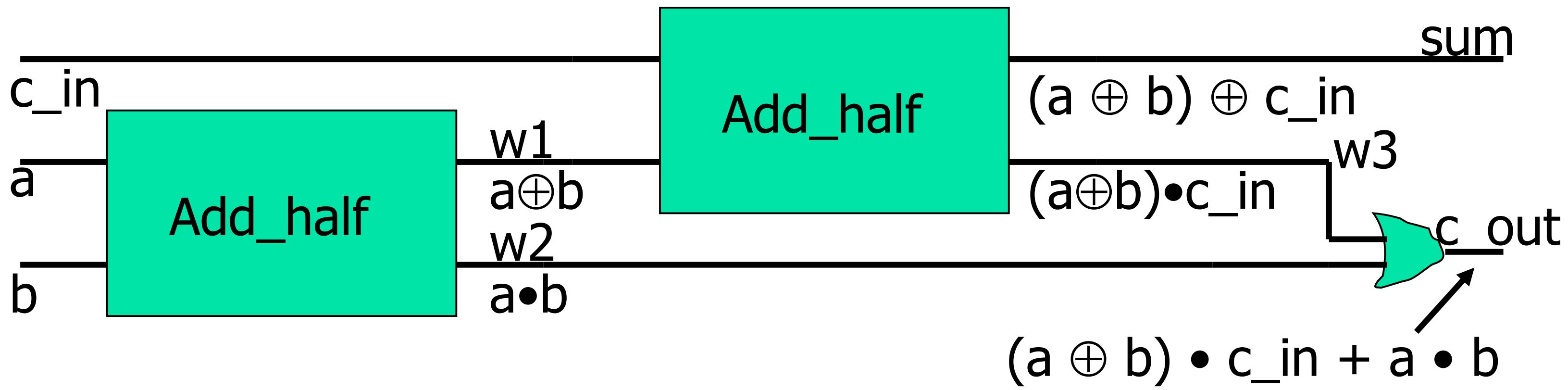
Hierarchical Description



Nested module instantiation to arbitrary depth



Full Adder in Verilog

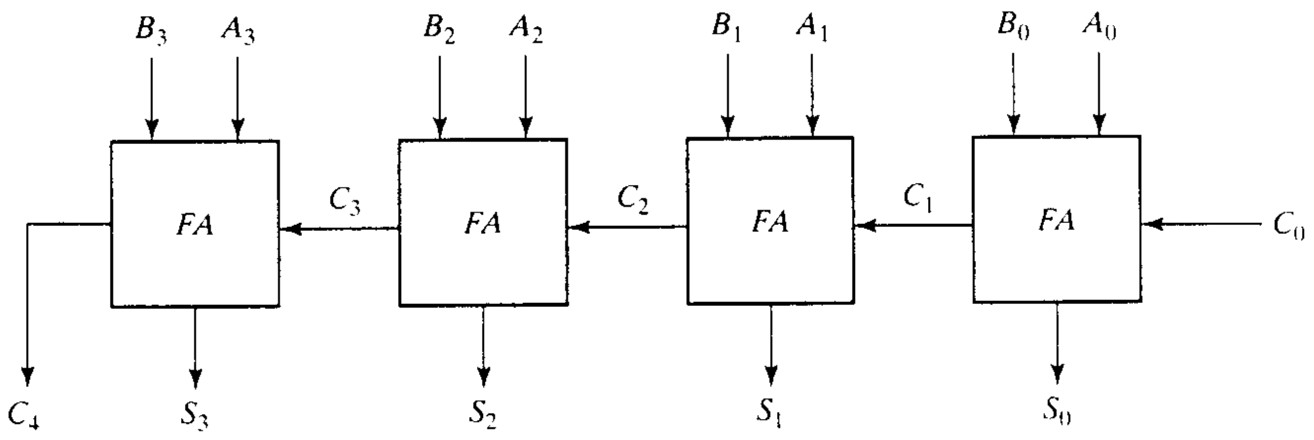


```
Module Add_full ( sum, c_out, a, b, c_in );     // parent module
    input a, b, c_in;
    output c_out, sum;
    wire w1, w2, w3;
    Add_half M1 ( w1, w2, a, b );
    Add_half M2 ( sum, w3, w1, c_in );          // child module
    or ( c_out, w2, w3 );                       // primitive instantiation
endmodule
```

Module instance name

Binary adder

Subscript i:	**3**	**2**	**1**	**0**	
Input carry	0	1	1	0	C_i
Augend	1	0	1	1	A_i
Addend	0	0	1	1	B_i
Sum	1	1	1	0	S_i
Output carry	0	0	1	1	C_{i+1}



Gate-level description of 4-bit adder

```
// Gate-level description of 4-bit
//ripple-carry adder

//Description of half adder
module half_adder (S,C,x,y);
output   S,C;
input    x,y;
xor (S,x,y);
and (C,x,y);
Endmodule
/…………………………………………………….
// description of full adder
module full_adder (S,C,x,y,z);
output   S,C;
input    x,y,z;
wire     S1, D1, D2;
// Instantiate half adders
half_adder HA1 (S1,D1,x,y);
half_adder HA2(S,D2,S1,z);
or G1 (C, D2, D1)
endmodule
```

Continue Description of 4-bit adder

```
// Description of 4-bit adder (see Fig 4-9)

module ripple_carry_4_bit_adder (sum, C4, A, B, CO);
output [3:0] Sum;
output   C4;
input [3:0] A,B;
input     CO;
wire      C1,C2,C3;// Intermediate carries
// Instantiate chain of full adders
full_adder FAO(Sum[0],C1,A[0], B[0], CO),
             FA1 (Sum[1],C2,A[1],B[1], C1),
             FA2(Sum[2],C3,A[2],B[2],C2,
             FA3(Sum[3], C4,A[3],B[3],C3);
endmodule
```

Dataflow Modeling

- 1)Dataflow modeling provides the means of describing combinational circuits by their function rather than by their gate structure.
- 2)Dataflow modeling uses a number of operators that act on operands to produce desired results
- 3)Dataflow modeling uses continuous assignments and the keyword assign.

4)A continuous assignments is a statement that assign a value to a net.

5)The value assigned to the net is specified by an expression that uses poerands and operators.

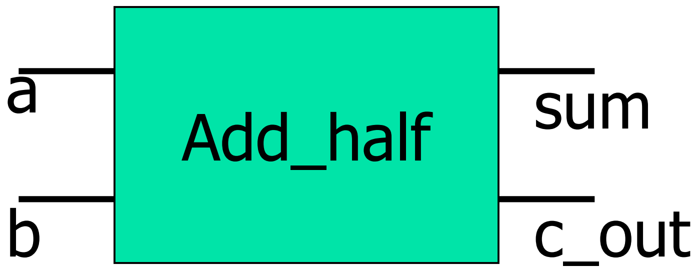
Arithmetic Operators

Symbol	Operator
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Continuous assignment model

```
module Add_half ( sum, c_out, a, b );
   input       a, b;
   output      sum, c_out;

   assign { c_out, sum } = a + b; // Continuous assignment

endmodule
```

Concatenation



Dataflow description of 4-bit adder

```
//Dataflow description of 4-bit adder

module adder _4_bit_df (A,B,C_in,SUM,C_out)
output [3:0] SUM;
output C_out;
input [3:0] A,B;
input C_in;
assign {C_out, Sum} = A+B+C_in;
endmodule
```

Behavioral modeling represents digital circuits

- It is used mostly to describe sequential circuits .
- Behavioral descriptions use the keyword always followed by a list of procedural assignments
- The target output must be of the type reg
- 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.
- The conditional statement if-else provides a decision based upon the value of the select input.

if-else examples

```
module mux2to1 (w0, w1, s, f);
    input w0, w1, s;
    output f;
    reg f;

    always @(w0 or w1 or s)
        if (s= =0)
            f = w0;
        else
            f = w1;

endmodule
```

reg data type

Include all input signals

Case statement

- Format

```
case(expression)
        alternative1: statement;
        alternative2: statement;
        …
        alternativej: statement;
        [default: statement;]
endcase
```

1) Many possible alternatives
2) Expression and each alternative are compared bit by bit.
3) If there is a match, the statement is executed
4) If the alternatives do not cover all possibilities, default should be included.

Multiplexer using case

```
module mux4to1 (W, S, f);
    input [0:3] W;
    input [1:0] S;
    output f;
    reg f;

    always @(W or S)
        case (S)
            0: f = W[0];
            1: f = W[1];
            2: f = W[2];
            3: f = W[3];
        endcase

endmodule
```

or binary numbers

Decoder using case

```
module dec2to4 (W, Y, En);
      input [1:0] W;
      input En;
      output [0:3] Y;
      reg [0:3] Y;

      always @(W or En)
            case ({En, W})
                  3'b100: Y = 4'b1000;
                  3'b101: Y = 4'b0100;
                  3'b110: Y = 4'b0010;
                  3'b111: Y = 4'b0001;
                  default: Y = 4'b0000;
            endcase

endmodule
```

Concatenate operator
Default for En=0

HDL for Sequential Circuits

- Behavioral Modeling
- Structural Modeling

always statement

- The always statements can be controlled by delays that wait for certain time or for certain condition to become true or by events to occur

always @ (event control expression)

procedural assignment statement.

The event control expression specifies the condition that must occur to activate the execution of the procedural statements

The variables in the procedural statement must be of the *reg* type and must be declared as such

Types of events

Two kinds of events:

- Level sensitive : (latches)

  always @ (A *or* B *or* Reset)

- Edge triggered events : (FF)

always @ (*posedge* clock *or* *negedge* reset)

This will cause the execution of the procedural statement if the clock goes through a positive transition or reset goes through negative transition

FF and Latches

- D Latch ➔ has control input (Fig 5.6)

```
module D_Latch (Q,D,control);
  output Q;
  input D, control;
  reg Q;
  always @ (control or D)
  if (control) Q=D;  //same as: if (control==1)
endmodule
```

FF and Latches

- D FF ➔ has clock

```
module D_FF (Q,D, CLK);
  output Q;
  input D, CLK;
  reg Q;
  always @ (posedge CLK )
     Q=D;
endmodule
```

- D FF ➔ has clock and Reset

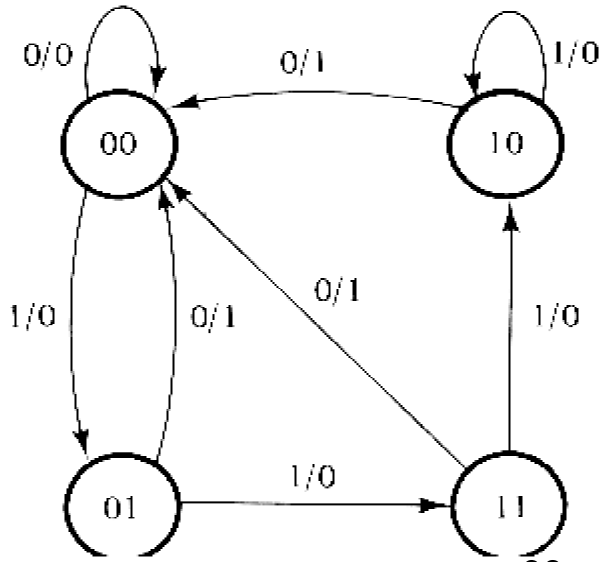
```
module D_FF (Q,D, CLK,RST);
  output Q;
  input D, CLK,RST;
  reg Q;
  always @ (posedge CLK or
                    negedge RST )
 if (~RST)  Q=1'b0;
else  D=Q;
endmodule
```

State Diagram (Fig 5.16)

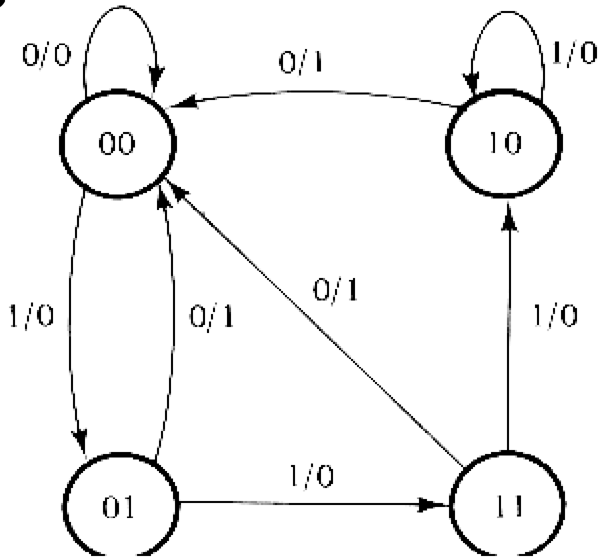
/* any change that occurs in the Nxtstate is transferred to Prstate as a result of a posedge event */

/*state binary assignment is done using *parameter* */

//* Verilog allows constants to be defined in a module by the keyword parameter */



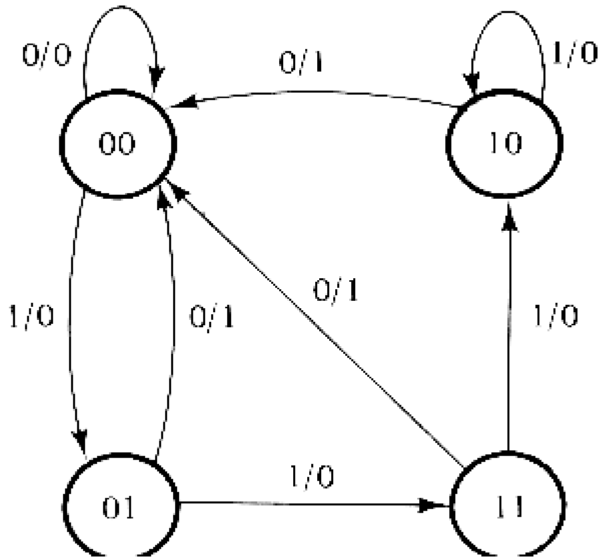
// 3 always block : 1) reset and clocked operatio



```
module Mealy_mdl (x,y,CLK,RST);
    input x, CLK,RST;
    output y;
    reg y;
    reg [1:0]  Prstate, Nxtstate;
    parameter S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;
      always @ (posedge CLK or negedge RST)

        If (~RST) Prstate =S0;      //initialize to S0

        else Prstate=Nxtstate ;     //clock operations
```

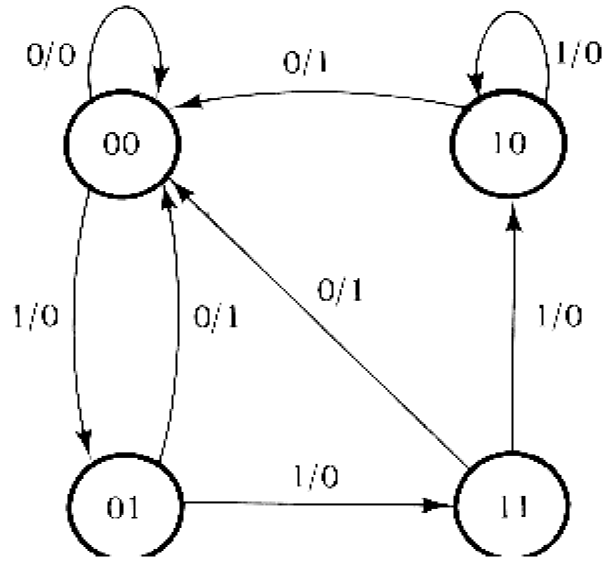
/* 2) second always determines next state transition as a function of present state and input*/

```
always @ (Prstate or x)
        case (Prstate)
         S0:  if (x) Nxtstate = S1;
              else  Nxtstate = S0;
         S1:  if (x)  Nxtstate = S3;
              else  Nxtstate = S0;
         S2:  if (~x) Nxtstate = S0;
              else  Nxtstate = S2;
         S3:  if (x)  Nxtstate = S2;
              else  Nxtstate = S0;
        endcase
```



/ 3-rd always evaluates the output and listed separately for clarity, but can be combined with 2-nd always */*

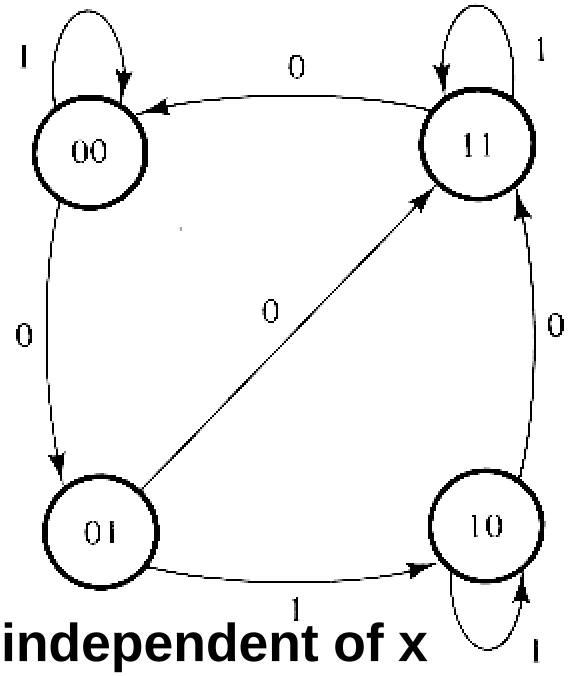
```
always @ (Prstate or x) // Evaluate output
        case (Prstate)
          S0:  y=0;
          S1:  if (x)  y=1'b0; else y=1'b1;
          S2:  if (x)  y=1'b0; else y=1'b1;
          S3: if (x)  y=1'b0; else y=1'b1;
      endcase
endmodule
```



*/*note that y may change if x changes while the circuit is at any state*/*

Moore Machine Fig 5.19

```
module moore_mdl (x,AB,CLK,RST); // output is a function of Pres.state
  input x, CLK,RST;                          // only
  output  [1:0] AB;
 reg [1:0] state; // PS is identified by the variable state
 parameter S0=2'b00, S1=2'b01, S2=2'b10, S3=2'b11;
   always @ (posedge CLK or negedge RST)
      if (~RST) state =S0; //initialize to S0
      else
      case (state)
        S0: if (~x) state =S1; else state=S0;
        S1: if (x) state =S2; else state=S3;
        S2: if (~x) state =S3; else state=S2;
        S3: if (~x) state =S0; else state=S3;
      endcase
  assign AB=state;       // output  of flip-flops is independent of x
endmodule
```



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