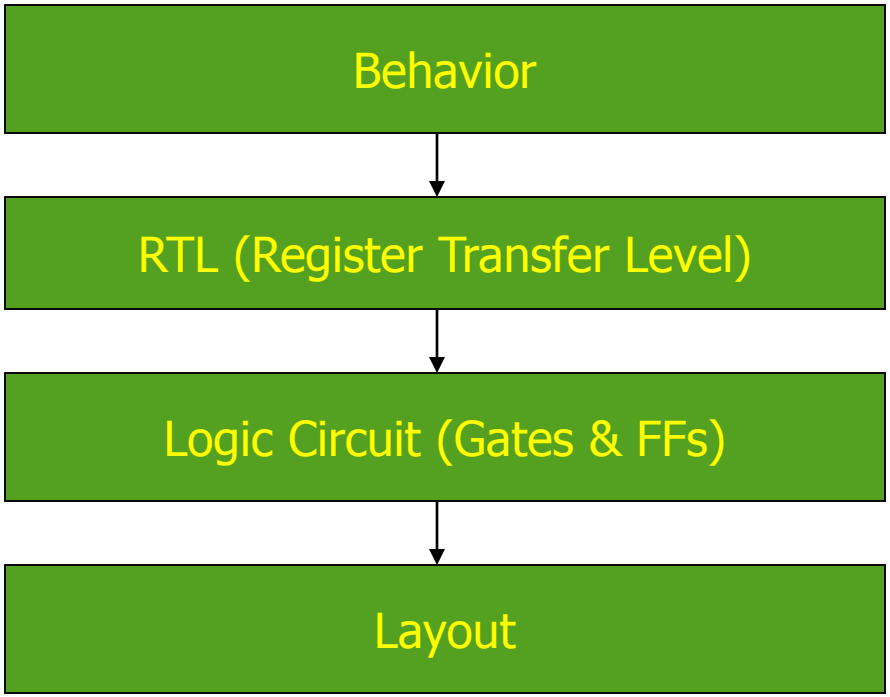
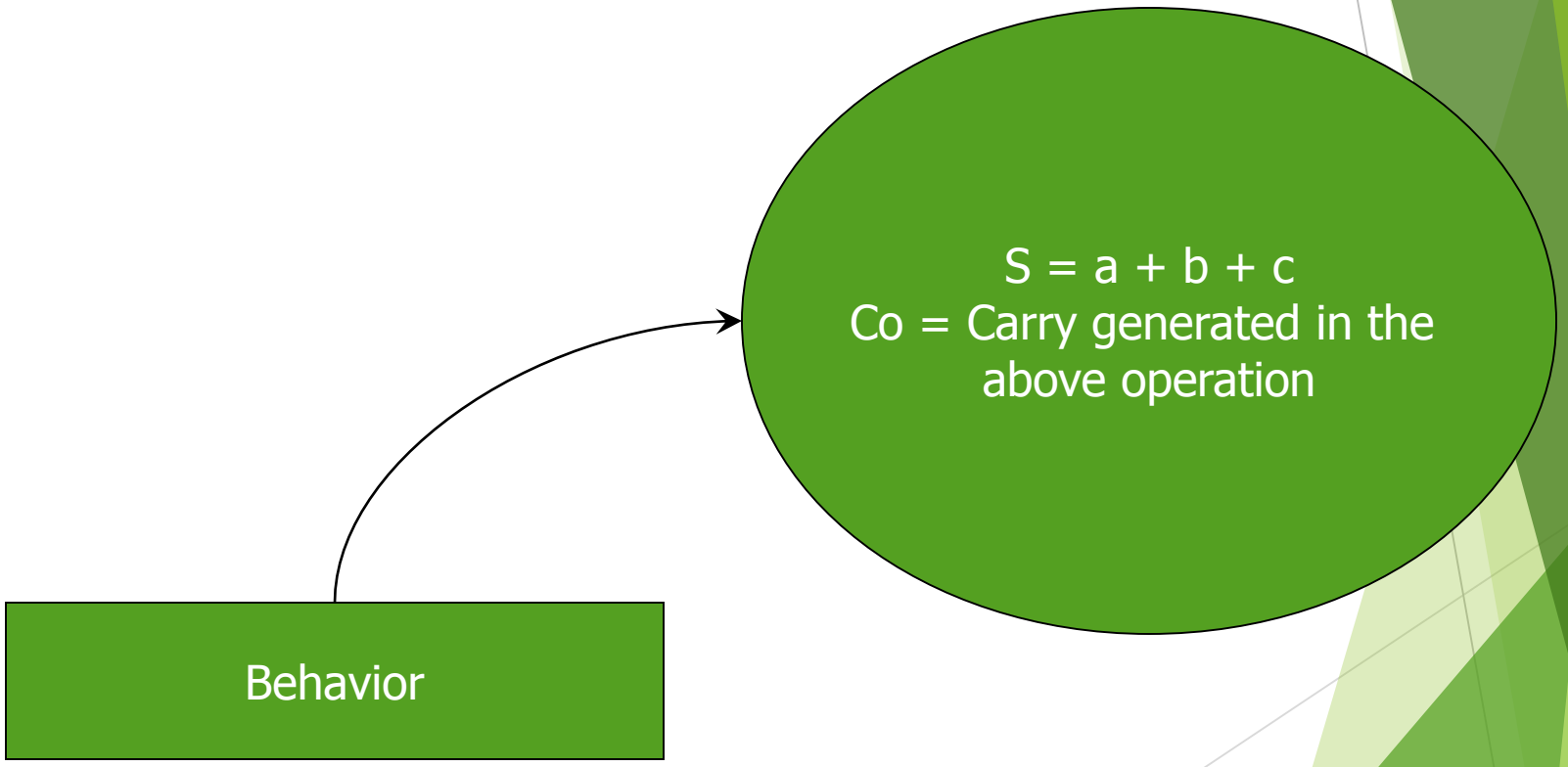
Hardware Description Language (HDL)

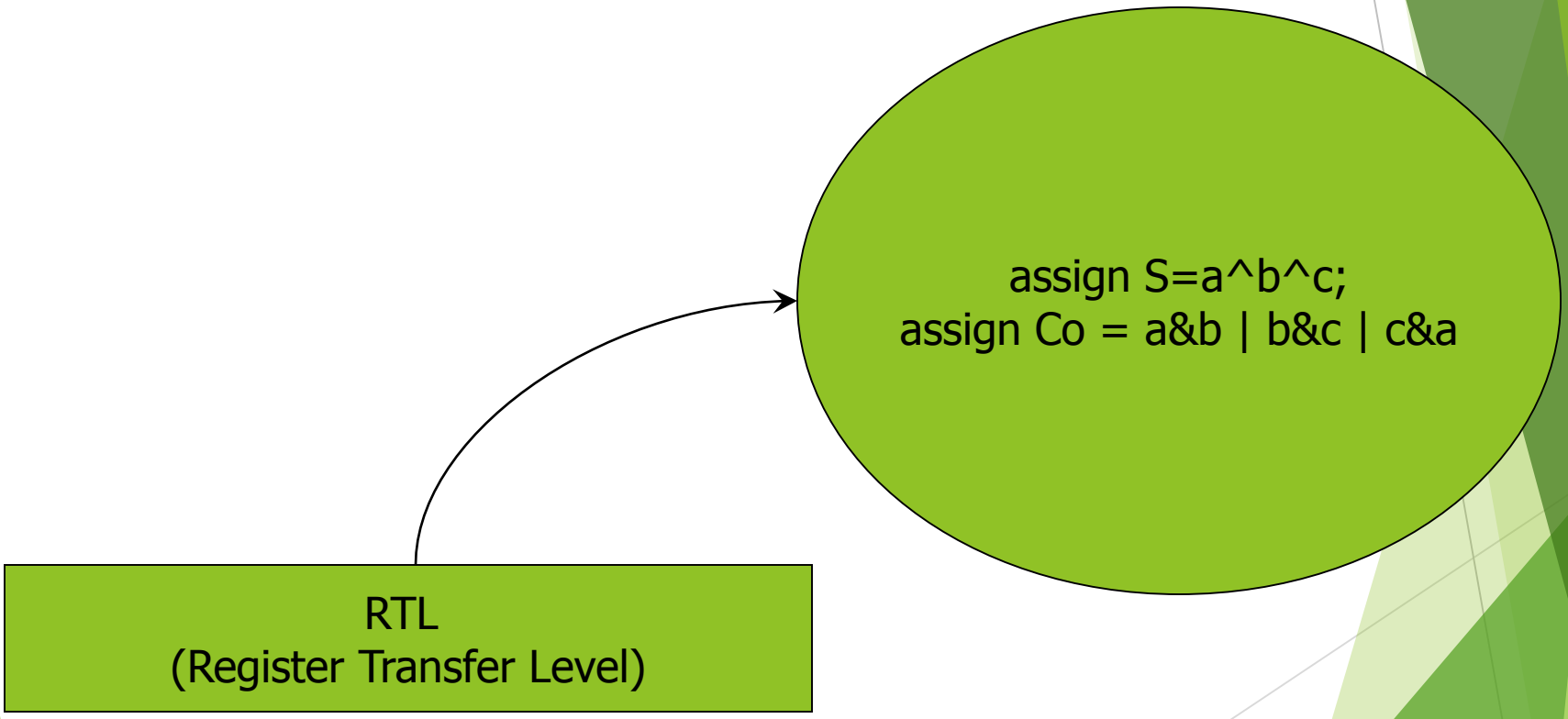
Digital Design Flow



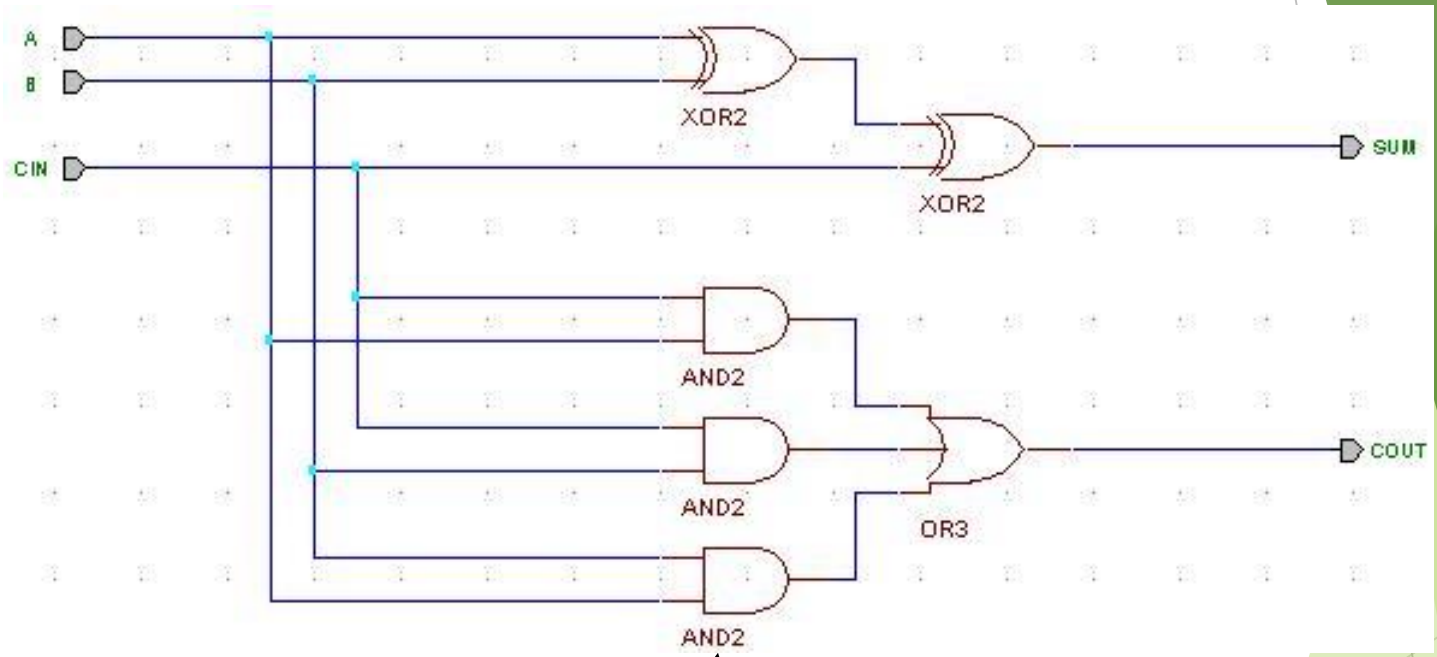
Abstractions in Digital IC Design – Example [1]



Abstractions in Digital IC Design – Example [2]



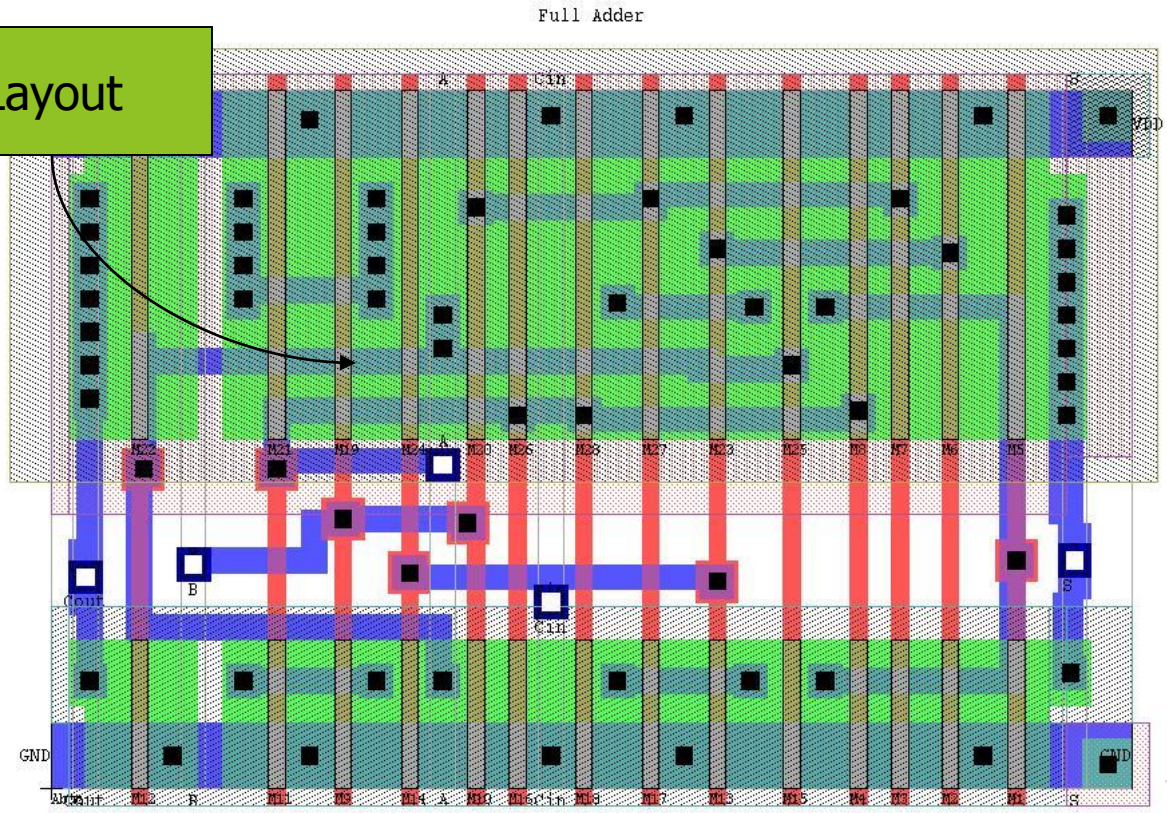
Abstractions in Digital IC Design – Example [3]



Logic Circuit
(Gates & FFs)

Abstractions in Digital IC Design

Layout



Hardware Description Language (HDL)

- A Programming language used to describe hardware
- It resembles a programming language, but is specifically oriented to describing hardware structures and behaviors.
- The main difference:
  - traditional programming languages→ mostly serial operations
  - HDLs→ extensive parallel operations
- The most common use of a HDL is to provide an alternative to schematics.

Hardware Description Language (HDL)

- When a language is used for the above purpose (i.e. to provide an alternative to schematics), it is referred to as a *structural description* in which the language describes an interconnection of components.

- Such a structural description can be used as input to logic simulation just as a schematic is used.

Commercial HDLs…

- VHDL – VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
- Verilog HDL – **Veri**fying **Log**ic HDL
- System C, System Verilog – very popular now

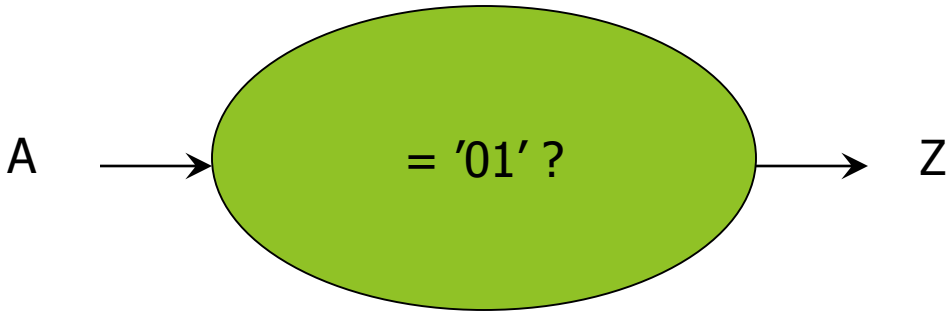
HDL Processing consists of two applications.

- - Logic Simulation
  - A simulator interprets the HDL description, produces readable outputs ex. Timing diagram. Simulation allows detection of functional errors in design Stimulus that tests the functionality of design is test bench

- Logic synthesis
  - Deriving a list of components and their interconnections from the HDL description of the digital system. The gate level netlist is used to fabricate the IC.

Using HDLs

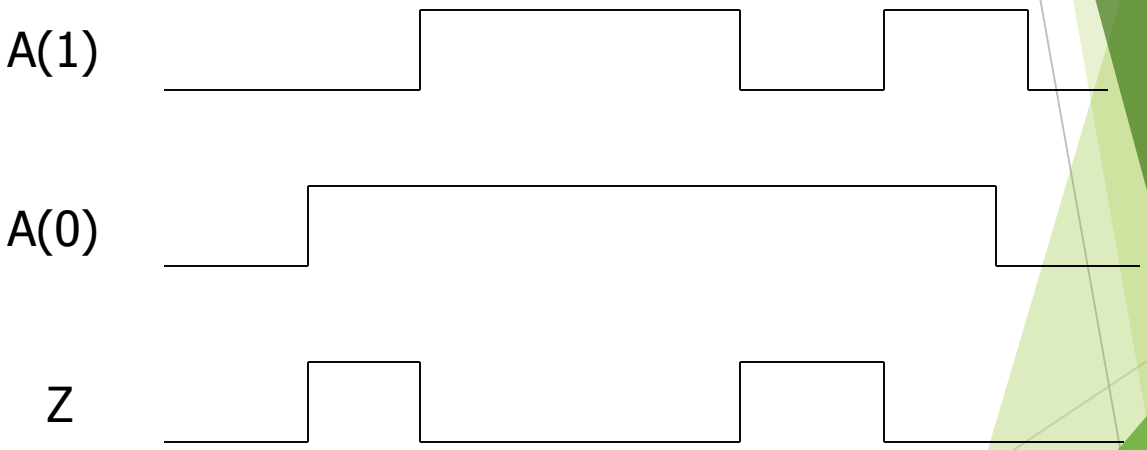
Modeling a specification



Z <= '1' when A = '01' else '0'

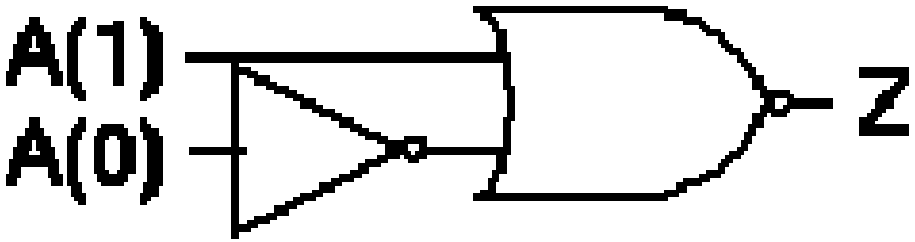
Using HDLs… (2)

Simulating a model



Using HDLs… (3)

Synthesizing a simulated model



Verilog HDL

- Verilog HDL is
  - C - like syntax
  - Initially developed as a simulation tool
  - With synthesis tools added later used for hardware implementation of the code.
- History
  - Developed as proprietary language in 1985
  - Opened as public domain spec in 1990
  - Became IEEE standard in 1995

Verilog HDL

- Verilog constructs are *keywords (lowercase)*
  - Examples: and, or, wire, input, output
- One important construct is the *module*
  - Modules have inputs and outputs
  - Modules can be built up of Verilog primitives or of user defined sub modules.

Module representation.

- A logic circuit is specified in the form of a **`module.`** (building block)
- The module is a text description of the circuit layout.
- Starts with module and ends with **`endmodule`**
- Module may have inputs, outputs referred to as ports

Module

- Basic building block of Verilog
- Describes any one of the three modelling techniques.
- Gate-level Modelling
  - using instantiation of primitive gates and user defined modules.
- Dataflow Modelling
  - using continuous assignment statements with keyword `assign`
- Behavioral Modelling
  - using procedural assignment statements with keyword `always`.

Rules for naming modules

Module Name :

- Can’t start with a number
- Should start only with a alphabet or underscore
- Can’t contain special characters except “_” (underscore)

Examples

Half_adder , _counter_10 , alu_4bit are valid module names

4x1_mux , my@and , alu 8bit are invalid module names

Note: Verilog is a case ***sensitive*** language.

Simple Circuit Notes

- The module starts with **`module`** keyword and finishes with **`endmodule`**.
- Internal signals are named with **`wire`**.
- Comments follow **`//`**
- **`Input`**, **`inout`** and **`output`** are ports. These are placed at the start of the module definition.
- Each statement ends with a semicolon, except **`endmodule`**.

Primitives

- Verilog includes gate level primitives

  A two input AND gate with inputs x1, x2 and output y is denoted as

  and (y,x1,x2)

- Verilog has all the standard gates
  - **and, nand**
  - **or, nor**
  - **xor, xnor**
  - **not, buf**

GATE LEVEL MODELLING

Example: Simple Circuit Diagram

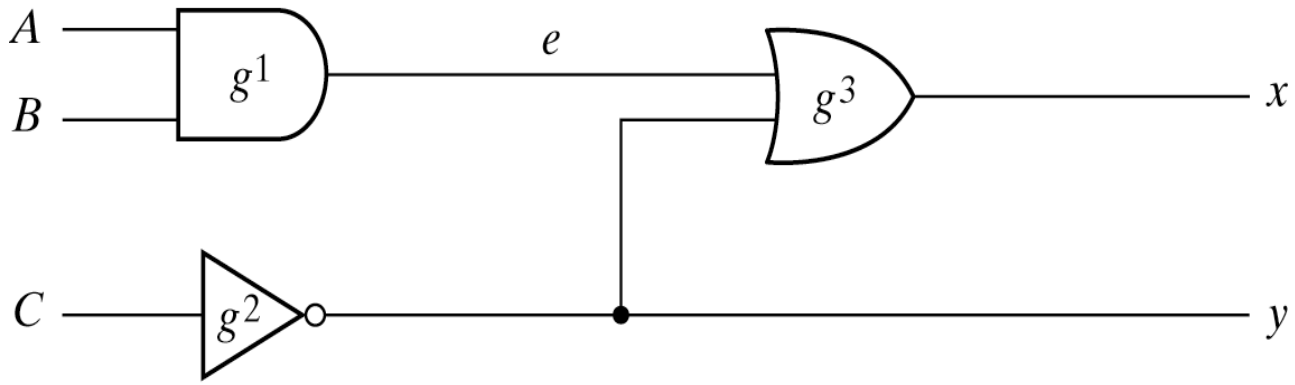


Fig. 3-37 Circuit to Demonstrate HDL

Source: Digital design: M. Mano, 3rd ed., Pearson

Example: Simple Circuit HDL

```
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
```

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

output [0:3] D;

wire [7:0] SUM;

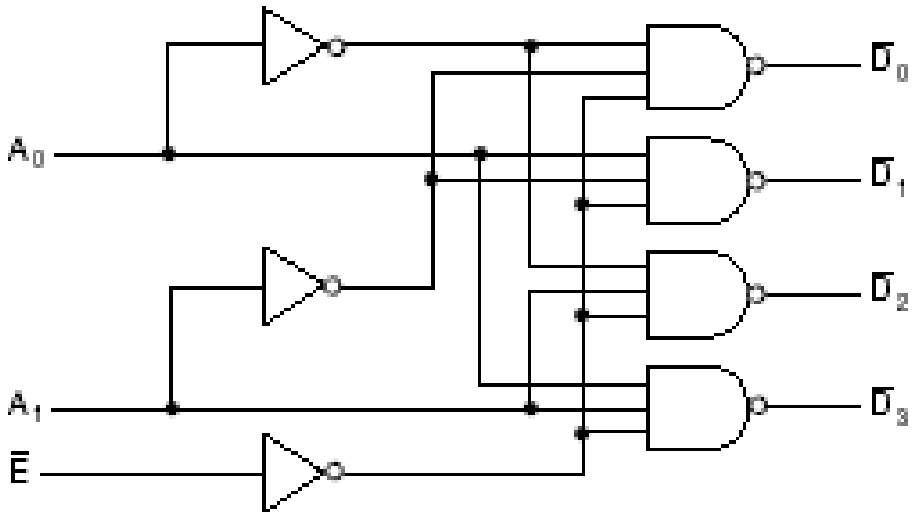
- output vector D with four bits 0 through 3

  First number is msb.

- wire vector SUM with eight bits numbered 7

  through 0

2-to-4 Line Decoder



(a) Logic diagram

E	A_1	A_0	$\bar{D}_0$	$\bar{D}_1$	$\bar{D}_2$	$\bar{D}_3$
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
1	X	X	1	1	1	1

(b) Truth table

$$\bar{D}_0 = \overline{E\,\bar{A}_1\,\bar{A}_0}$$
$$\bar{D}_1 = \overline{E\,\bar{A}_1\,A_0}$$
$$\bar{D}_2 = \overline{E\,A_1\,\bar{A}_0}$$
$$\bar{D}_3 = \overline{E\,A_1\,A_0}$$

(c) Logic Equations

Fig.3-14 A 2–to–4-Line Decoder

```
// Gate-level description of a 2-to-4 line decoder
module decoder_g1   ( A,B,E,D);
     input A,B,E;
     output [0:3] D;
     wire Anot, Bnot, Enot;
     not
          n1  (A1not, A1),
          n2  (A0not, A0),
          n3  (Enot, E);
    nand
        n4 (D[0], A1not, A0not, Enot),
        n5 (D[1], A1not, A0, Enot),
        n6 (D[2], A1, A0not, Enot),
        n4 (D[3], A1, A0, Enot);
endmodule
```

Adding Delays

- To simulate a circuits real world behavior it is important that propagation delays are included.
- The units of time for the simulation can be specified with **`timescale.**
- **Eg: `timescale 1us/100ns**
  - Default is 1ns with precision of 100ps
- Component delays are specified as #(delay)

Simple 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
```

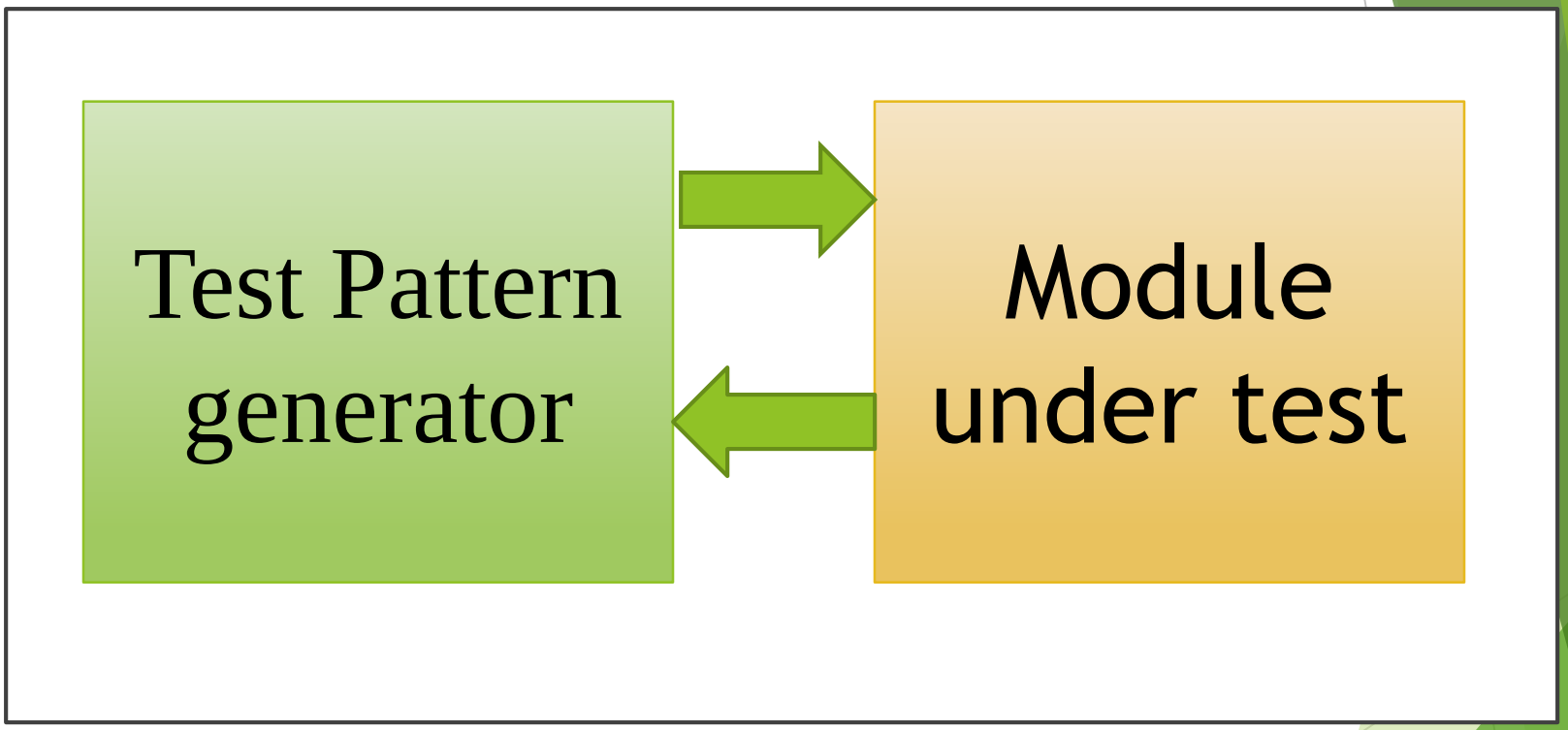
Input signals

- In order to simulate a circuit the input signals need to be known so as to generate an output signal.
- The input signals are often called the circuit *stimulus*.
- An HDL module is written to provide the circuit stimulus. This is known as a *testbench*.

Testbench

- The *testbench* module includes the module to be tested.
- There are no input or output ports for the testbench.
- The inputs to the test circuit are defined with **`reg`** and the outputs with **`wire`**.
- The input values are specified with the keyword **`initial`**
- A sequence of values can be specified between **`begin`** and **`end`**.

Test Bench



A Test Bench Block diagram representation

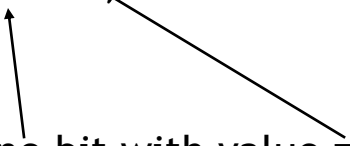
Test Bench

- A Test Bench is also a Verilog module whose purpose is to generate test patterns for another module.
- A Test bench doesn't have any input or output ports.
- A Test bench instantiates the module its going to test.
- The system tasks are used in test benches.
- ***$display , $monitor , $time , $stop, $finish***

Signal Notation

- In Verilog signals are generalised to support multi-bit values (e.g. for buses)
  - The notation

```
A =  1'b0;
```



  - means signal A is one bit with value zero.
- The end of the simulation is specified with **`$finish`**.

Stimulus module 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
```

Effect of delay

Time (ns)	Input A B C	Output y e x
-	0 0 0	1 0 1
-	1 1 1	1 0 1
10	1 1 1	0 0 1
20	1 1 1	0 0 1
30	1 1 1	0 1 0
40	1 1 1	0 1 0
50	1 1 1	0 1 1

Timing Diagram

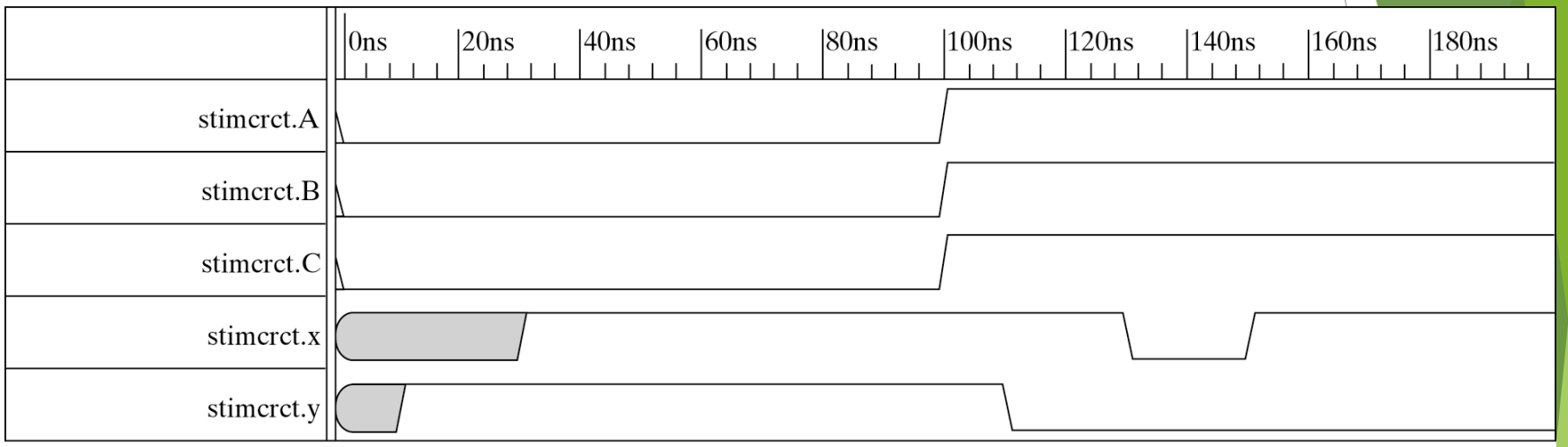


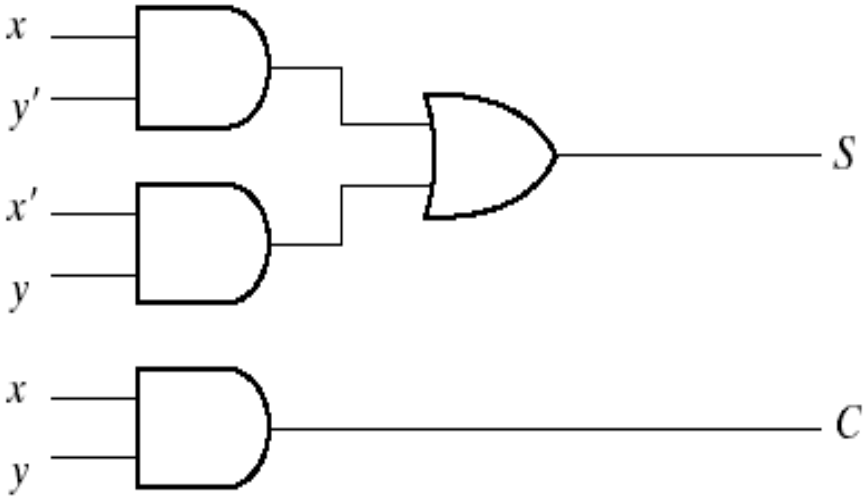
Fig. 3-38 Simulation Output of HDL Example 3-3

- 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.

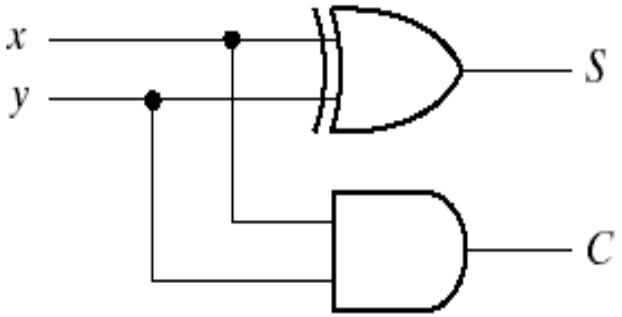
- In a top-down design, a 4-bit binary adder is defined as top-level block with 4 full adder blocks. Then we describe two half-adders that are required to create the full adder.
- In a bottom-up design, the half-adder is defined, then the full adder is constructed and the 4-bit adder is built from the full adders.

- A bottom-up hierarchical description of a 4-bit adder is described in Verilog as
  - Half adder is 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.**

4 bit Full Adder



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



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

Fig. 4-5 Implementation of Half-Adder

4 bit Full Adder

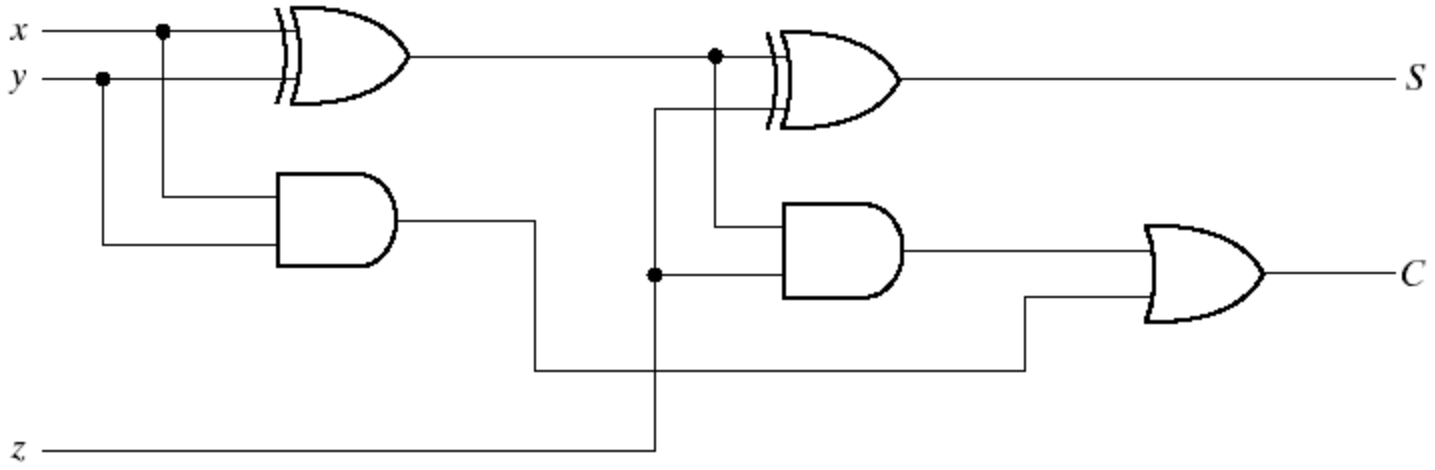


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

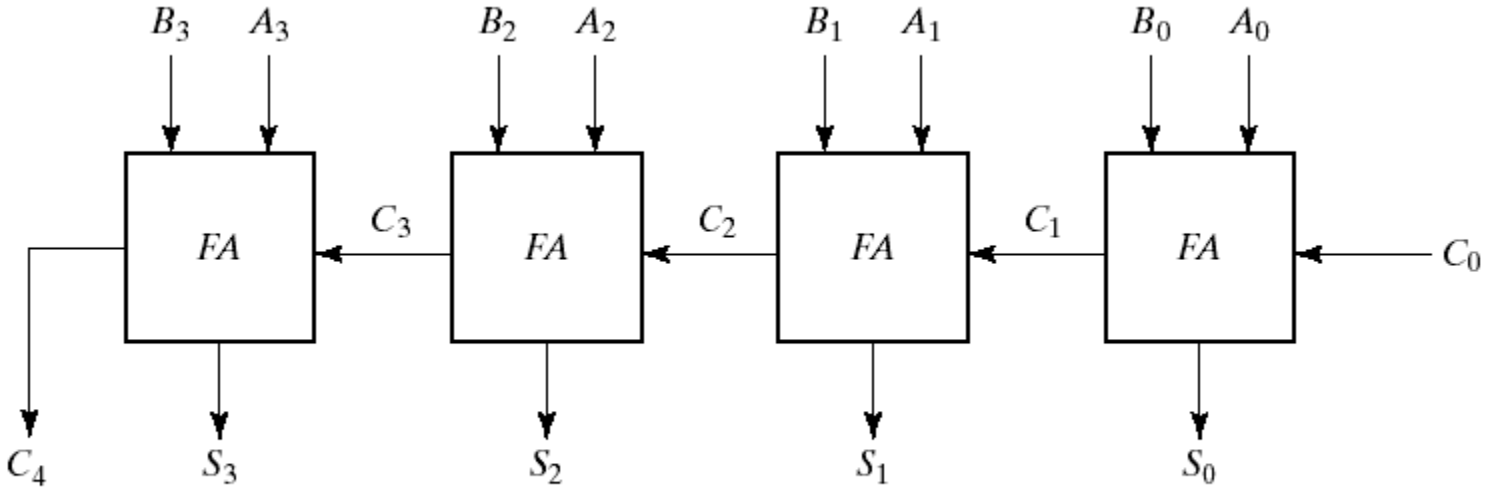


Fig. 4-9 4-Bit Adder

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

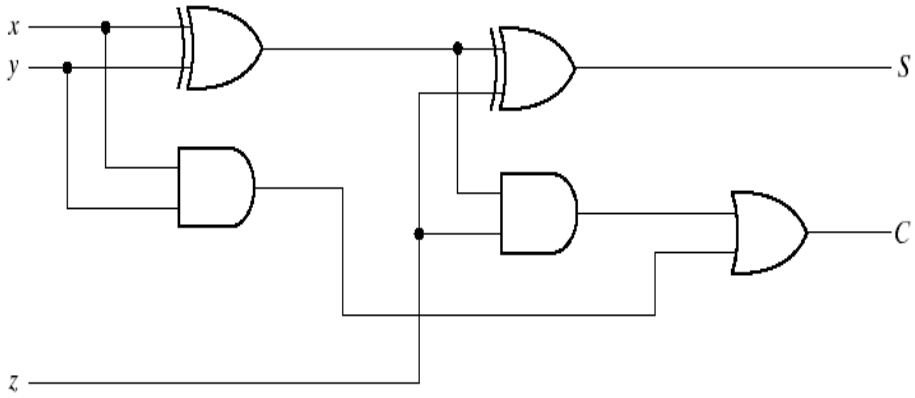
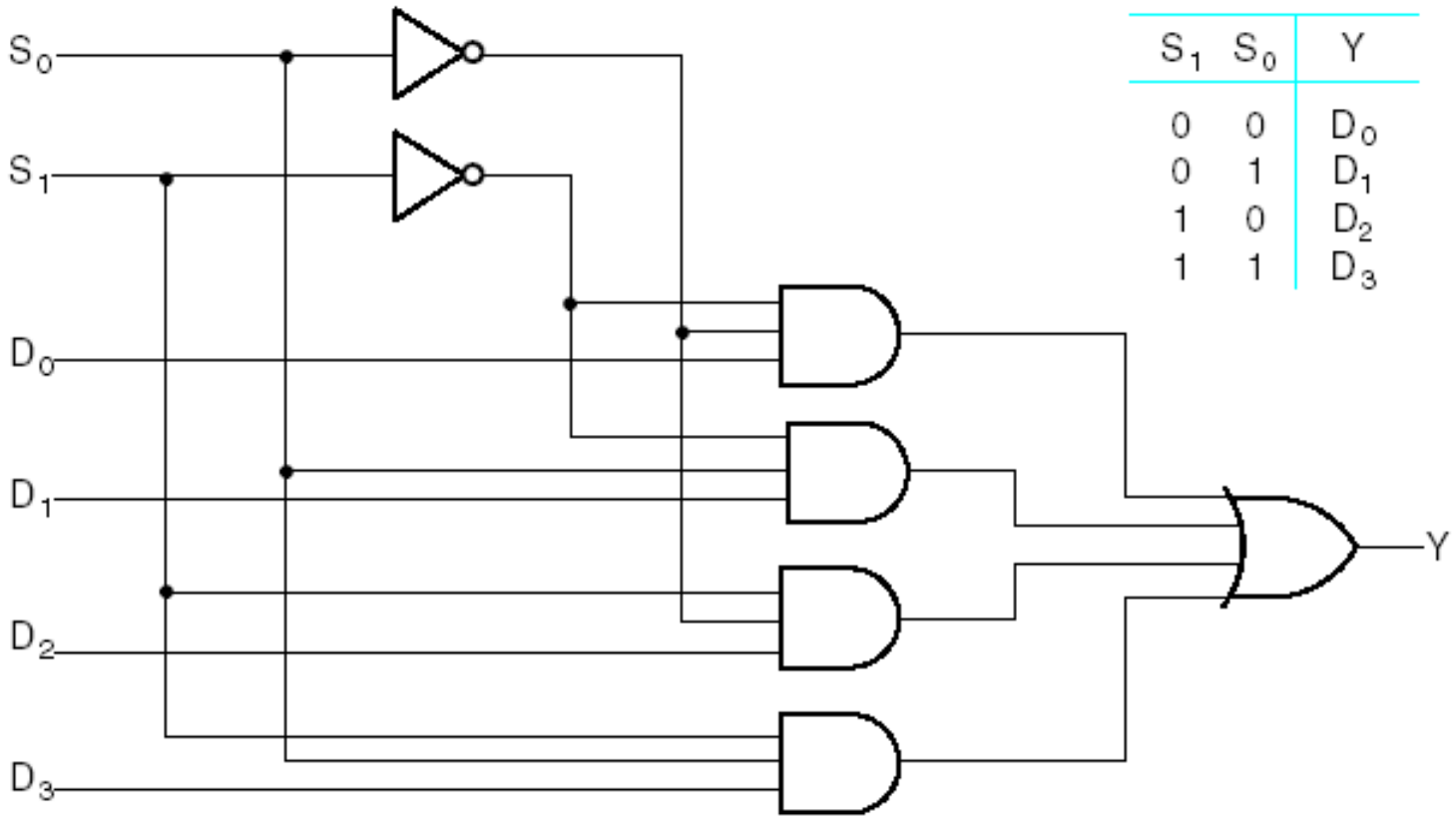


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

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

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 Line Multiplexer: Structural Verilog Description     //  1
// (See Figure 3-19 for logic diagram)                         //  2
module multiplexer_4_to_1_st_v(S, D, Y);                       //  3
   input [1:0] S;                                              //  4
   input [3:0] D;                                              //  5
   output Y;                                                   //  6
                                                               //  7
   wire [1:0] not_S;                                           //  8
   wire [0:3] N;                                               //  9
                                                               // 10
not                                                            // 11
    gn0(not_S[0], S[0]),                                       // 12
    gn1(not_S[1], S[1]);                                       // 13
                                                               // 14
and                                                            // 15
   g0(N[0], not_S[1], not_S[0], D[0]),                         // 16
   g1(N[1], not_S[1], S[0], D[1]),                             // 17
   g2(N[2], S[1], not_S[0], D[2]),                             // 18
   g3(N[3], S[1], S[0], D[3]);                                 // 19
                                                               // 20
or go(Y, N[0], N[1], N[2], N[3]);                              // 21
                                                               // 22
endmodule                                                      // 23
```

Fig. 3-45 Structural Verilog Description of a 4-to-1 Line Multiplexer

DATA FLOW MODELLING

Dataflow modelling

- Another level of abstraction is to model dataflow.
- Dataflow modeling uses a number of operators that act on operands to produce desired results.
- Verilog HDL provides about 30 operator types.
- In dataflow models, signals are continuously assigned values using the **`assign`** keyword.

Dataflow Modeling

- 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.
- For ex: assign Y = (A &B) | (C &D)
- **`assign`** can be used with Boolean expressions.
  - **Verilog uses `&` (and), `|` (or), `^` (xor) and `~` (not)**
- Logic expressions and binary arithmetic are also possible.

Simple Circuit Boolean Expression

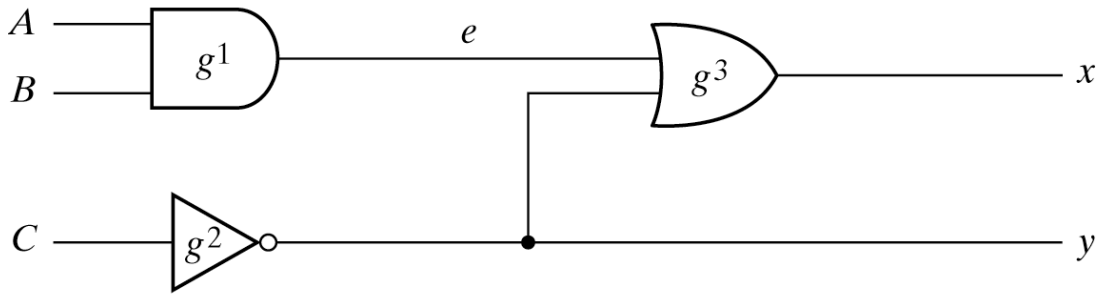


Fig. 3-37 Circuit to Demonstrate HDL

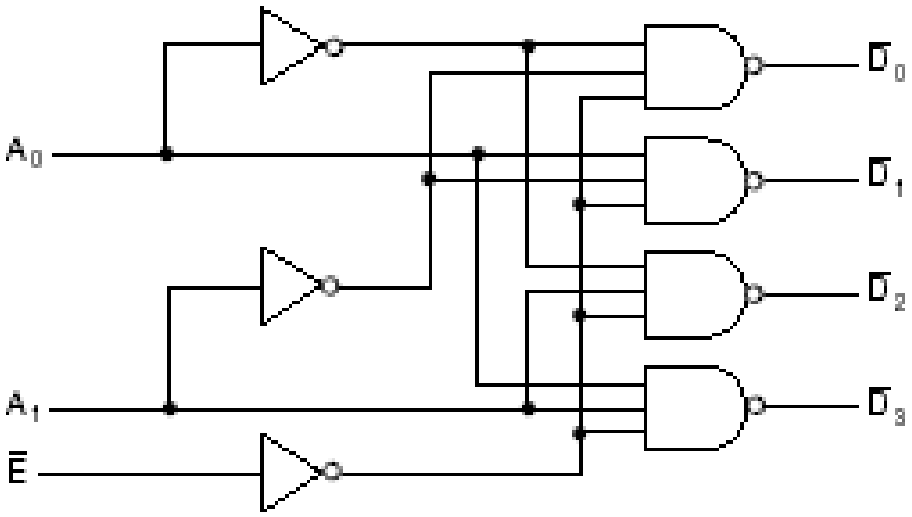
$x = A.B + C$

$y = C$

Boolean Expressions

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

2-to-4 Line Decoder



(a) Logic diagram

E	A_1	A_0	$\overline{D}_0$	$\overline{D}_1$	$\overline{D}_2$	$\overline{D}_3$
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
1	X	X	1	1	1	1

(b) Truth table

$$\overline{D}_0 = \overline{E\ \overline{A}_1\ \overline{A}_0}$$
$$\overline{D}_1 = \overline{E\ \overline{A}_1\ A_0}$$
$$\overline{D}_2 = \overline{E\ A_1\ \overline{A}_0}$$
$$\overline{D}_3 = \overline{E\ A_1\ A_0}$$

(c) Logic Equations

Fig.3-14 A 2–to–4-Line Decoder

Dataflow Modeling

```
//Dataflow description of a 2-to-4-line decoder
//See Fig.4-19
module decoder_df (A,B,E,D);
   input A,B,E;
   output [0:3] D;
   assign D[0] = ~(~A1 & ~A0 & ~E),
          D[1] = ~(~A1 & A0 & ~E),
          D[2] = ~(A1 & ~ A0 & ~E),
          D[3] = ~(A1 & A0 & ~E);
endmodule
```

4 bit Full Adder

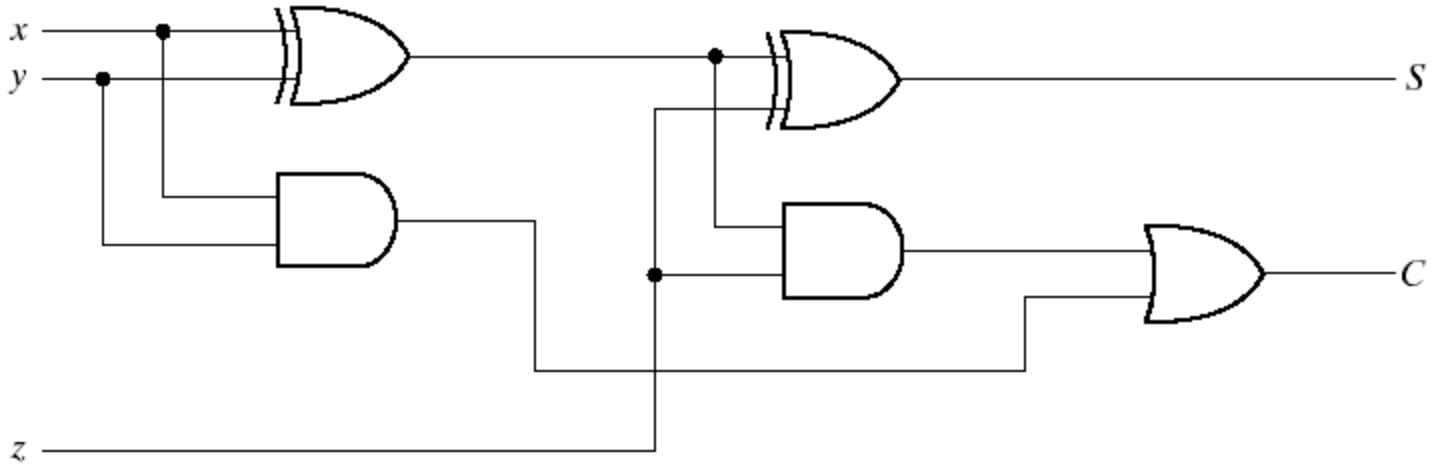


Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

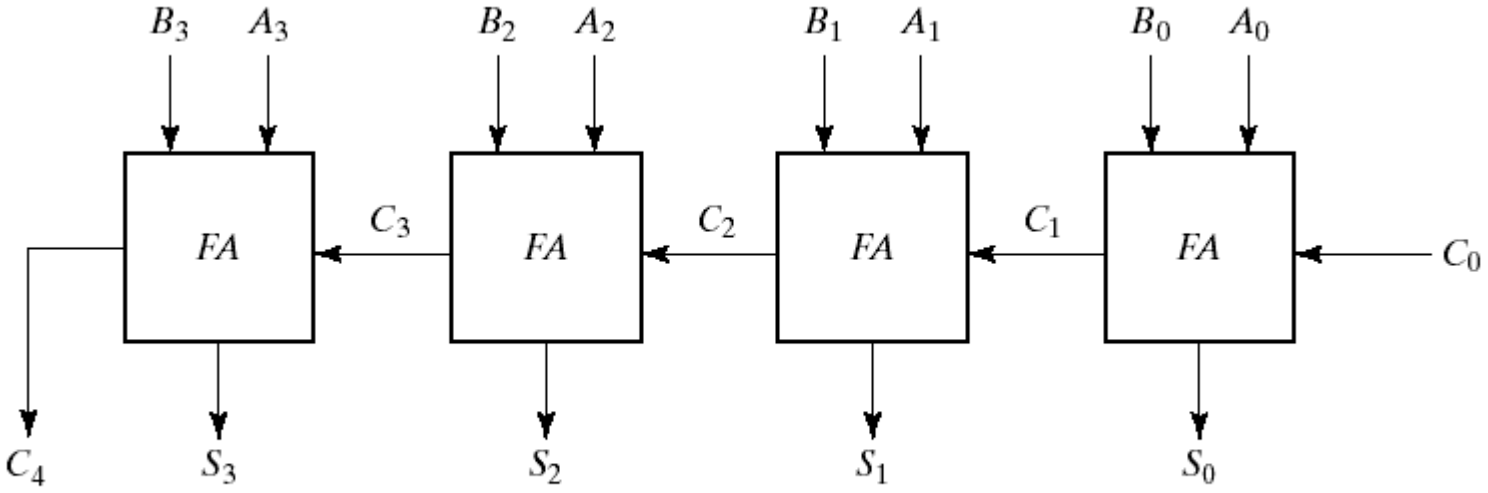


Fig. 4-9 4-Bit Adder

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 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.

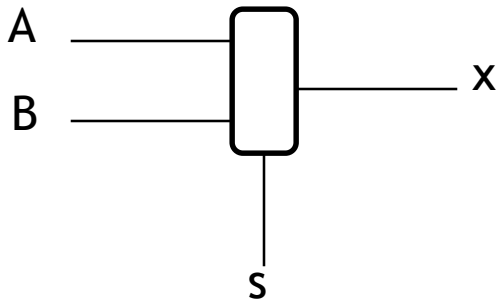
- Dataflow modelling provides means of describing combinational circuit by function rather than gate structure.
- A verilog HDL Synthesis tool can accept this module as input and can provide a netlist of a circuit equivalent.

Dataflow Modeling

```
//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
```

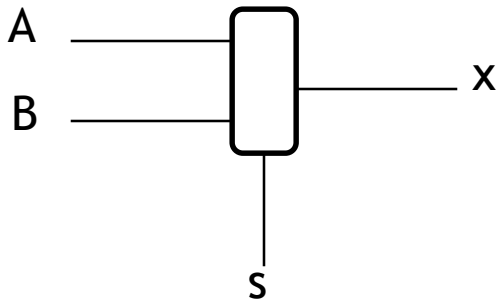
Multiplexer

- Multiplexer is a combinational circuit where an input is chosen by a select signal.
  - Two input mux
  - output =A if select =1
  - output= B if select =0



Two Input Multiplexor

- A two-input mux is actually a three input device.



$x = A.s + B.\bar{s}$

s	A	B	x
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Dataflow description of 2-input Mux

- Conditional operator `?:` takes three operands:

```
condition? true_expression : false_expression
```

```
module mux2x1_df (A,B,select,OUT);
    input A,B,select;
    output OUT;
    assign OUT = select ? A : B;
endmodule
```

Behavioral modelling

Behavioral Modeling

- Behavioral modeling represents digital circuits at a functional and algorithmic level.
- mostly to describe sequential circuits, but can also be used to describe combinational circuits.
- uses 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 Modelling

- Behavioral models place procedural statements in a block after the **`always`** keyword.
- The **`always`** keyword takes a list of variables. The block of statements is executed whenever one of the variables changes.
- The target variables are of type **`reg`**. This type retains its value until a new value is assigned.

Behavioral Modeling

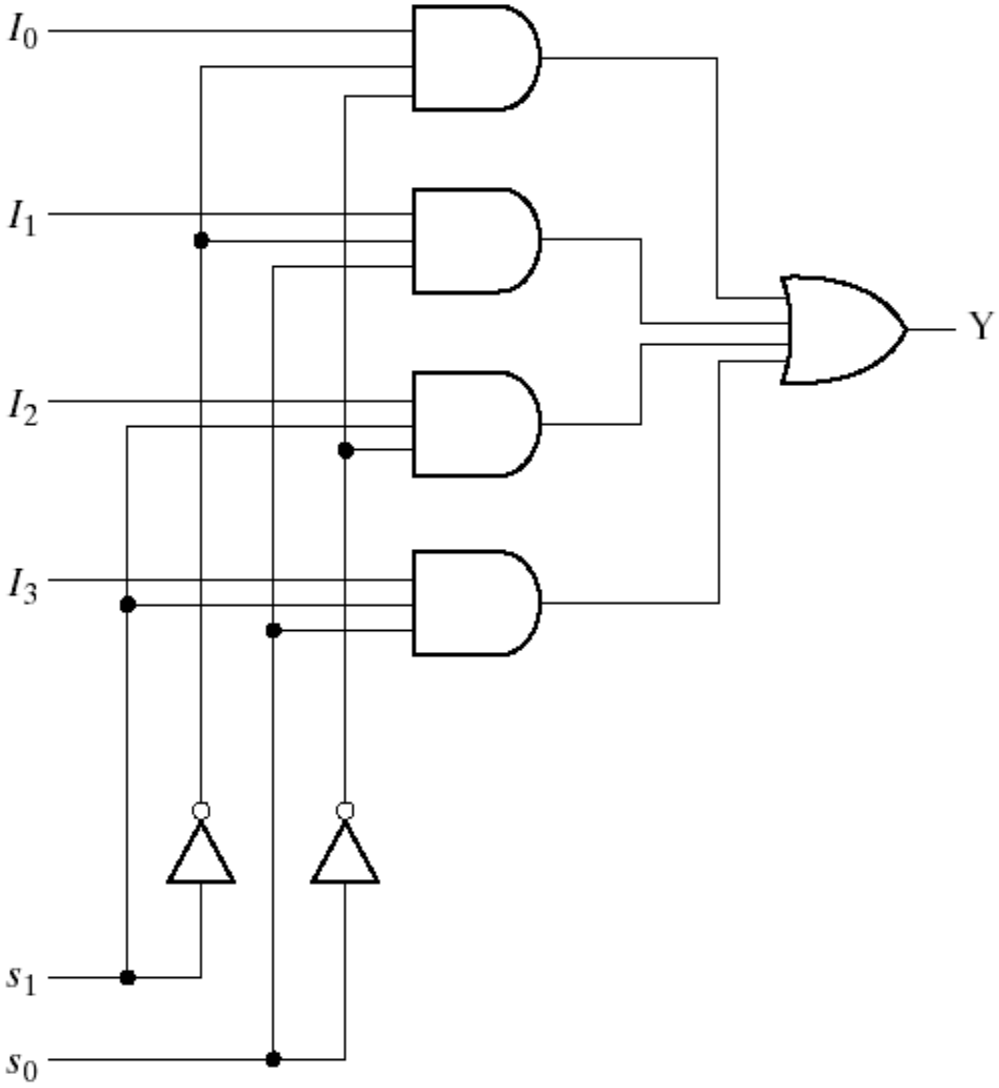
- The procedural assignment statements inside the **always** block are executed every time there is a change in any of the variable listed after the @ symbol. (Note that there is no “;” at the end of **always** statement)

Behavioral description of 2-input mux

```
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 Modelling

4-to-1 line multiplexer



(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

Behavioural Modelling Example

```
module mux_4X1 (y, s, d ) ;
input  [3:0] d  ;  // four input lines
modeled as a bus
Input  [1:0] s  ;  // select lines bundled as
a bus
output  reg  y ;
always  @ (  *  )
begin
case (s)
2`b00 : y = d[0] ;
2`b01 : y = d[1] ;
2`b10 : y = d[2] ;
2`b11 : y = d[3] ;
default : y = 1`bz ;
endcase
end
endmodule
```

Behavioural Modelling: Case statement

- **case** statement can have one ***default*** statement.
- **casex** will treat both x and z as don't care conditions
- **casez** will treat only z as don't care condition.
- A ? Can also be used instead of z
- If any of the bits in the case expression or case item expression is a don't-care value then that bit position will be ignored.

Behavioral Modeling

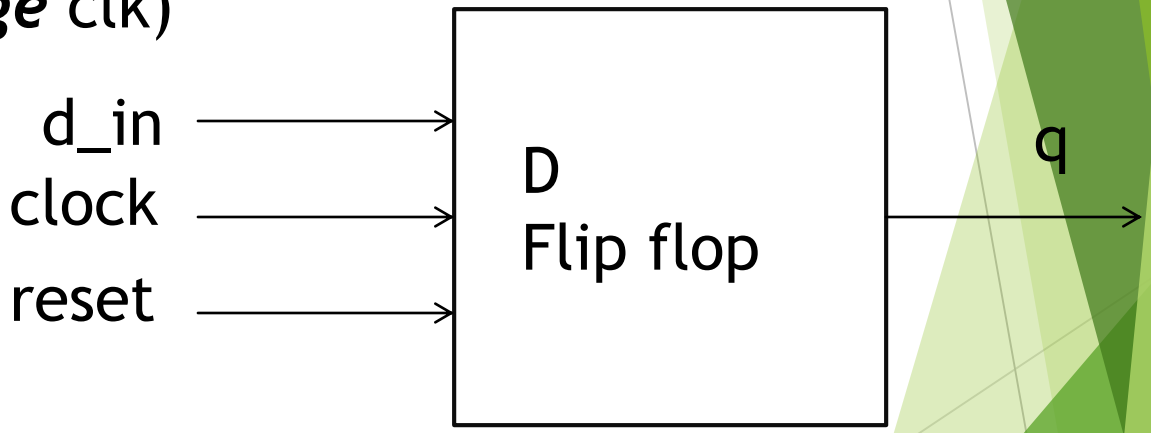
- In 4-to-1 line multiplexer, the select input is defined as a 2-bit vector and output y is declared as a reg data.
- The always block has a sequential block enclosed between the keywords case and endcase.
- The block is executed whenever any of the inputs listed after the @ symbol changes in value.

Clocked circuits

- In case of sequential design which are sensitive to edge of the clocks we use **posedge** or **negedge**
- **Posedge=** positive edge triggered
- **negedge=** negative edge triggered

Behavioural Modelling Example

```
module  dff_ (q,clk,reset,d);
input clk,reset,d ;
output  reg q ;
always @(posedge clk)
begin
if(reset)
q <= 0 ;
else
q <= d ;
end
endmodule
```



TRY IT OUT

- Can you specify edge sensitive and level sensitive input in a single sensitivity list statement in a single always block? i.e. can we have a statement
always @(d or posedge (clk))

HDL Summary

- Hardware Description Languages allow fast design and verification of digital circuits.
- Accurate simulation and testing requires delays and inputs to be specified.
- There are three different levels of abstraction for modelling circuits.