Verilog HDL

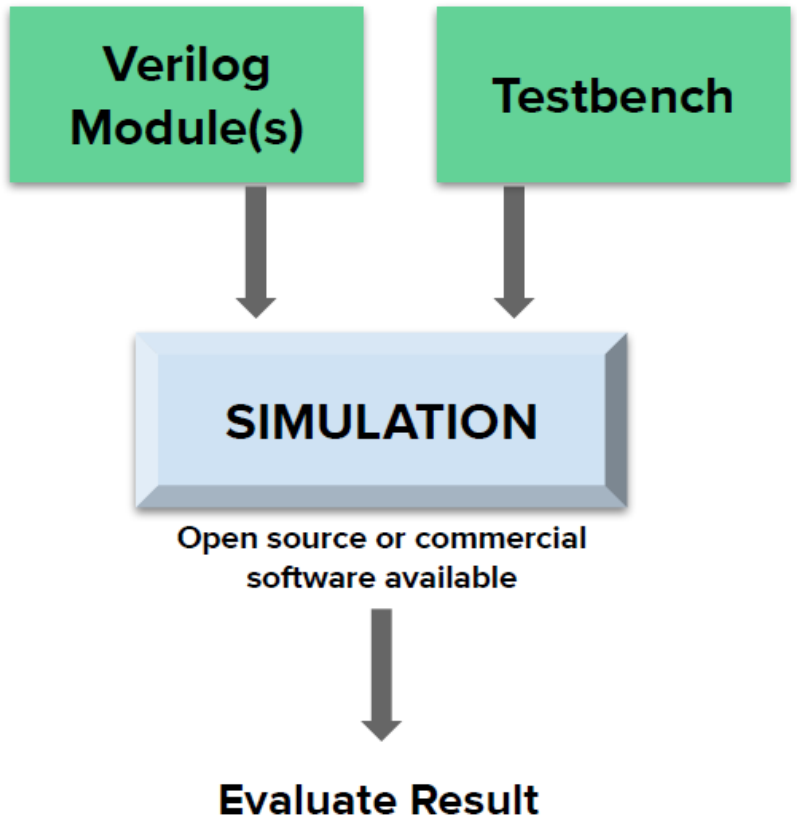
Topics

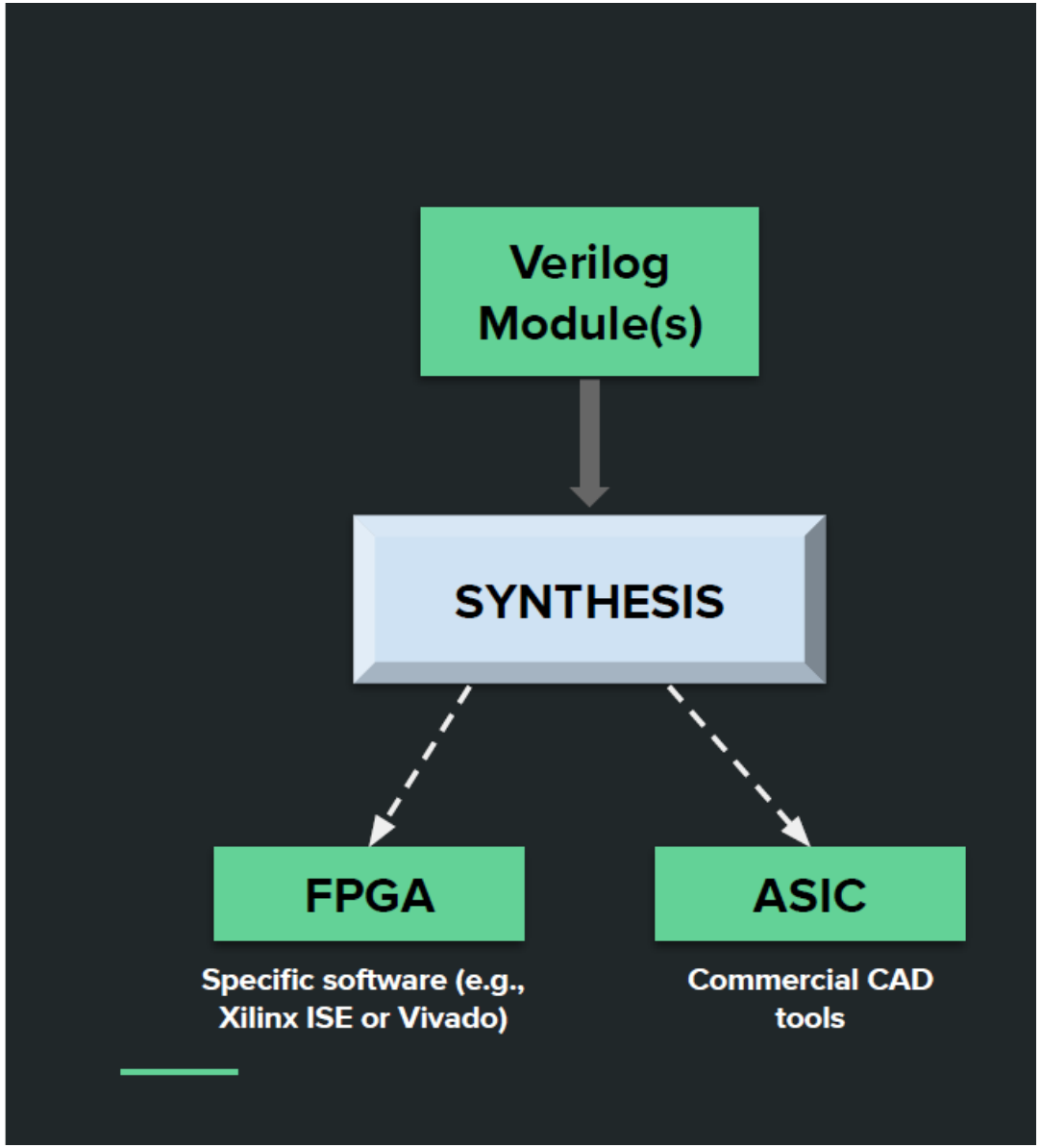
- Introduction to Verilog HDL
- Data types
- Operators
- Verilog code structures
- Verilog modelling styles and examples

Verilog evolution

- Verilog was designed in early 1984 by Gateway Design Automation. Initially the original language was used as a simulation and verification tool.
- Prabhu Goel, Phil Moorby, and Chi-Lai Huang invented Verilog between late 1983 and early 1984. Verilog is a hardware description language (HDL) that was originally developed as a simulation and verification tool. The name Verilog is a combination of the words "verification" and "logic".
- Cadence Design System acquired Gateway Design Automation. Since then, Cadence has been a strong force behind popularizing the Verilog hardware description language.
- In 1993, Verilog became the IEEE standard, IEEE Std. 1364-1995, in 1995.
- A new version of Verilog was approved by IEEE in 2001. This version that is referred to as Verilog-2001 is the present standard used by most users and tool developers

Development Process





Code format

- Verilog code is **case-sensitive**, i.e., identifiers using lowercase or uppercase characters are distinguished.
- The language uses certain keywords, all of which must use lowercase characters.
- Comments may appear anywhere in a Verilog source text. A comment

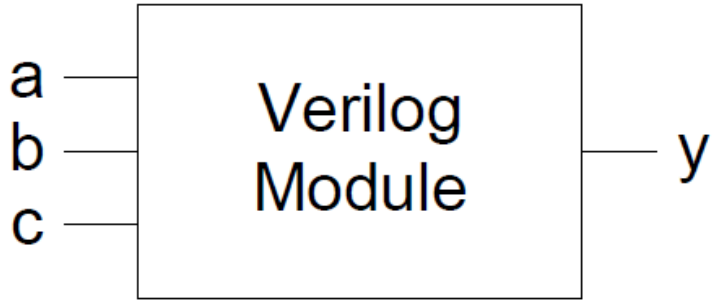
designator starting with // makes the rest of the line, up to a new-line character, a comment. The symbols /* and */ bracket a section of code as a comment and go across new-line characters.

Code format

- A **module** is the main structure for definition of hardware components and testbenches in verilog.
- Modules begin with the module keyword and end with **endmodule**. Immediately following the module keyword, port list of the module appears enclosed in parenthesis.
- Declaration of mode, type, and size of ports can either appear in the port list or as separate declarations.

```
module name (ports or ports and their declarations);
    port declarations if not in the header;
    other declarations;
    . . .
    statements
    . . .
endmodule
```

Example of a module




```
module example (a, b, c, y);
       input a;
       input b;
       input c;
       output y;

// here comes the circuit description

endmodule
```

Structure of a Module

```
module <mod name> (<port list>);
  <declarations>;  // input, output, inout
                   // wire, register, etc.
  <statements>;    // initial, begin, end, always
                   // dataflow statements
endmodule
```

```
module mux41 ( in0, in1, in2, in3, sel0, sel1, out );
```

→ Module Name & Ports List

```
input   sel0, sel1;
input   in0, in1, in2, in3;
output  out;
reg  op;
```

→ Declarations

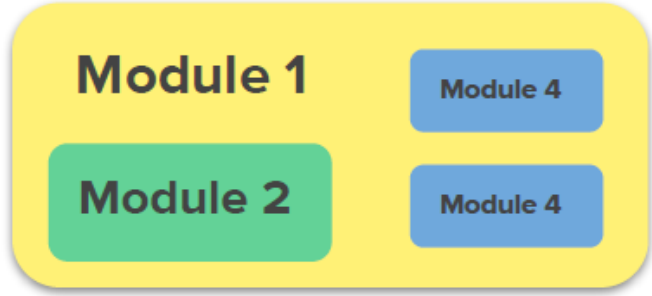
```
always @ ( in0 or in1 or in2 or in3 or sel0 or sel1 )
begin
    case ( { sel1, sel0 } )
            2'b00:  op  <= in0;
            2'b01:  op  <= in1;
            2'b10:  op  <= in2;
            2'b11:   op  <= in3;
          default:   op  <= op;
    endcase
end
assign  out = op;
```

→ Statements

```
endmodule
```

→ End of module

Verilog Language Features



Module - the basic unit of hardware in Verilog

- Cannot contain definitions of other modules,
- Can be ***instantiated*** within another module - **hierarchy of modules**.

Different than calling a function in programming lang.

Every instantiation adds to area!

```
module module_name (list_of_ports);
    input/output declarations
    Local net declarations   Temporary connections (wires)
    Parallel statements
endmodule
```

Why parallel?

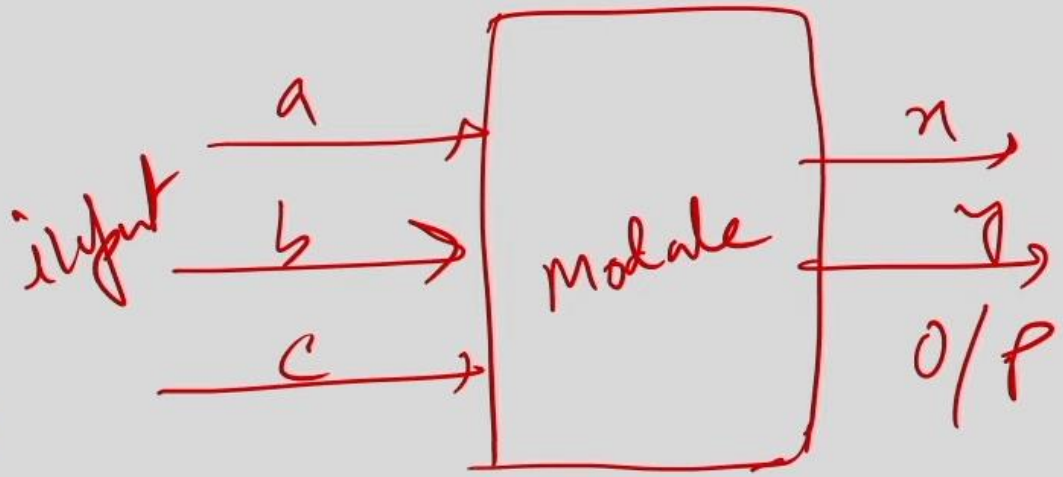
Ports

- Ports provide interface by which a module can communicate with it's environment.
  - By default ports are wire/net

input – Input port a, b, c

output – Output port x, y

inout – Bidirectional port



- Default type for input/output/inout is "**wire**".

  "**input wire in_1**" is same as "**input in_1**".

  "**output wire out_1**" is same as "**output out_1**".

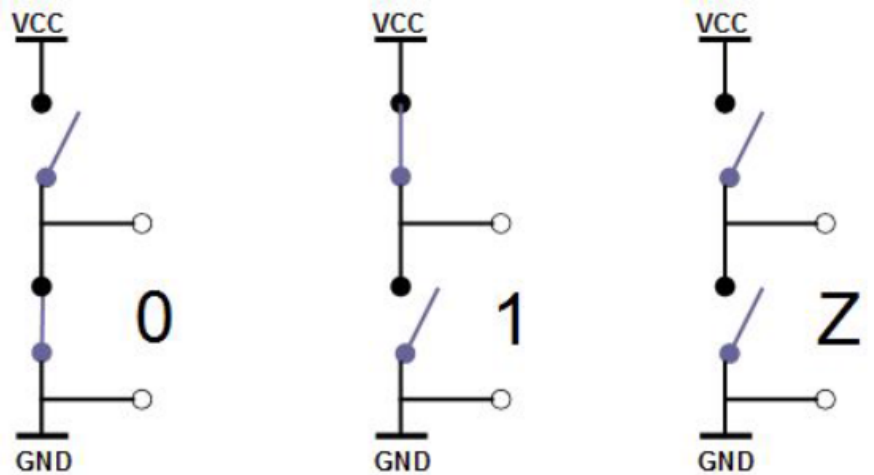
Verilog Language Features

Data Values

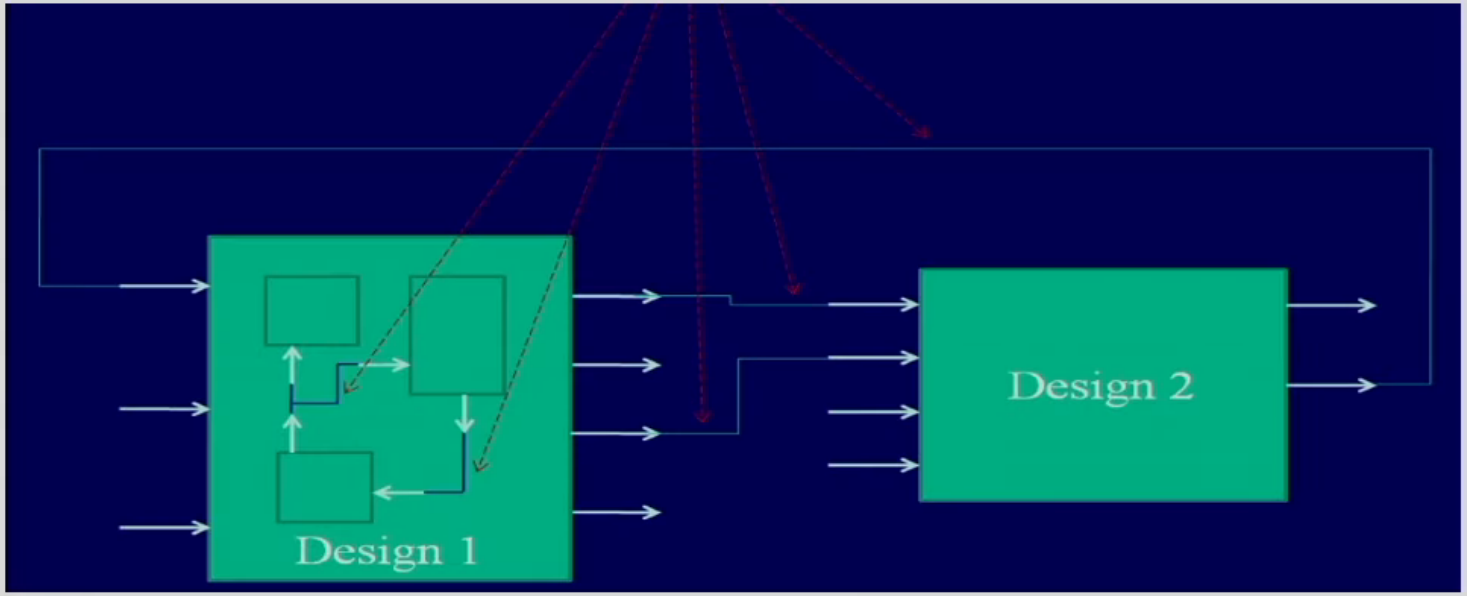
Verilog supports 4 value levels:

Value Level	Represents
0	Logic 0 state
1	Logic 1 state
x	Unknown logic state
z	High impedance state

- All unconnected nets are set to ‘z’.
- All register variables are set to ‘x’.



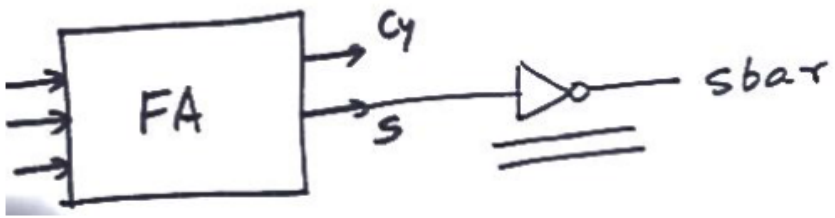
Nets



- Nets represent connections between hardware elements.
- They are always driven by some source.
- Declared by keyword “**wire**”.
- It can hold any one of the four values: 0, 1, x, z. Default value for any net type variable is ‘z’ (High impedance – hanging net).
- Always need a driver.

Verilog Language Features **Data Types**

```
wire sbar;
assign sbar = ~S;
```



A variable can be:

A. Net **wire, wor, wand, tri, supply0, supply1, etc.**

- Must be continuously driven,
- Cannot be used to store a value,
- Models connections between continuous assignments and instantiations,
- 1-bit values by default, unless declared as vectors explicitly.
- Default value of a *net* is “Z” - high impedance state.

B. Register **reg, integer, real, time**

- Retains the last value assigned to it,
- Usually used to represent storage elements (sometimes in combinational circuits),
- May or may not map to a HW register during synthesis.
- Default value of a *reg* data type is “X”.

Vectors

- Nets or reg data types can be declared as vectors (multiple bit widths).
- If bit width is not specified, the default is scalar (1-bit).

```
wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default
reg [0:40] virtual_addr; /* Vector register, virtual
address 41 bits wide */
```

Memories

- Often needs to model register files, RAMs, and ROMs.
- Memories are modeled in Verilog simply as a one-dimensional array of registers

reg [wordsize:0] memory [0:arraysize]

  - reg mem1bit[0:1023]; // Memory mem1bit with 1K 1-bit words
  - reg [7:0] membyte[0:1023]; // Memory membyte with 1K 8-bit words(bytes)

Parameters

- A parameter is defined by Verilog as a constant value declared within the module.

- Syntax:

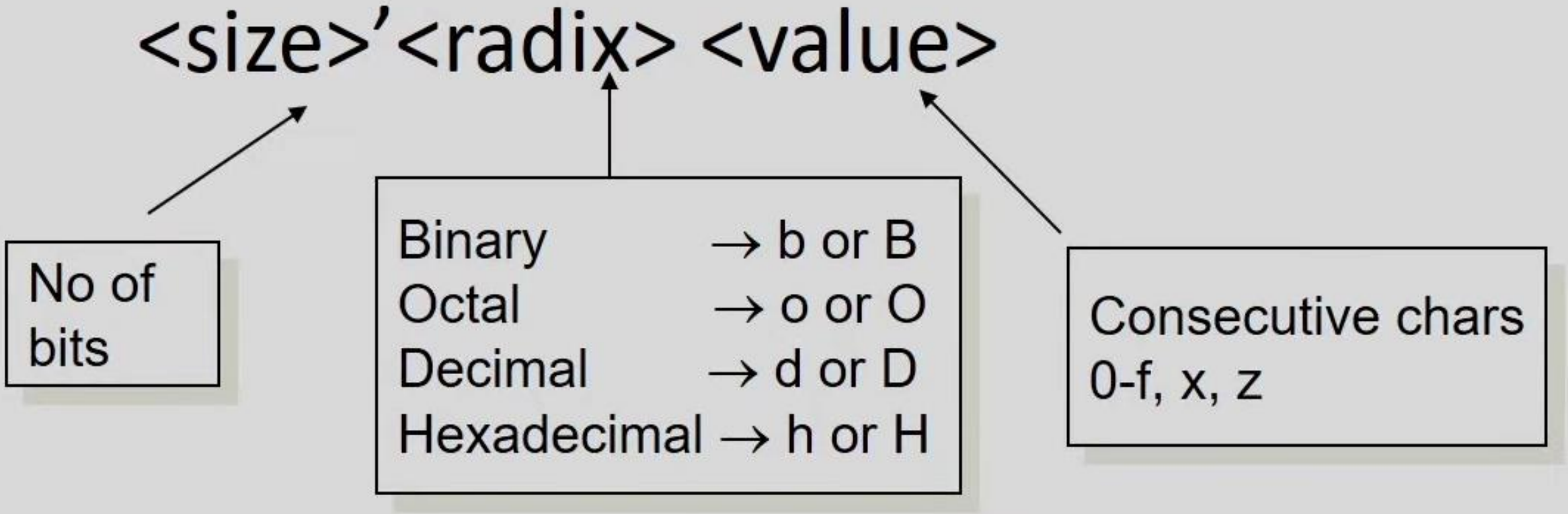
  parameter <identifier> = constant;

- Example:

  parameter byte_size = 8;

  reg[byte_size-1:0] A;

Number Representation in Verilog



Number Specification

- Numbers that are specified without a <base format> specification are decimal numbers by default
- Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).
- 23456 // This is a 32-bit decimal number by default
- 'hc3 // This is a 32-bit hexadecimal number
- 'o21 // This is a 32-bit octal number

Verilog	Stored Number	Verilog	Stored Number
4’b1001	1001	4’d5	0101
8’b1001	0000 1001	12’hFA3	1111 1001 0011
8’b0000_1001	0000 1001	8’o12	00 001 010
8’bxX0X1zZ1	XX0X 1ZZ1	4’h7	0111
‘b01	0000 .. 0001	12’h0	0000 0000 0000

```
wire a ; //1-bit wire
wire [7:0] bus; //8-bit bus
wire [31:0] busA, busB, busC ; //3 buses of 32-bit width

reg a; //1-bit register
reg [31:0] b; //32-bit register
```

- **Memories** (array of registers)

  reg [wordsize:0] memory [0:arraysize]

```
reg [3 : 0] mem_a [63:0] ;  //sixty-four 4-bit registers
mem_a [address] = data_in ;
reg mem_b [0 : 4] // five 1-bit registers
```

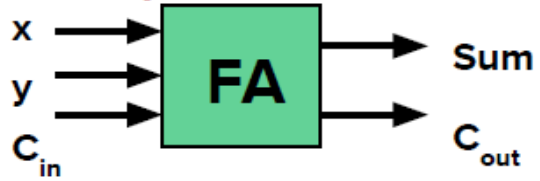
- **Number format**

  <size> ’ <base_format> <number_format>

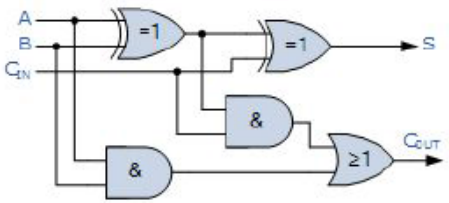
```
4’b1111   //This is a 4-bit  binary number
12’habc   //This is a 12-bit hexadecimal number
16’d255   //This is a 16-bit decimal number
23456     //This is a 32-bit decimal number by default
`hc3      //This is a 32-bit hexadecimal number
`o21      //This is a 32-bit octal number
```

Different Levels of Abstraction in Verilog HDL

Example: Full Adder



- **Gate-Level modeling**:
  - Verilog HDL supports built-in primitive gates modeling.




```
module full_adder(x,y,cin,sum,cout);
      input x,y,cin;
      output sum, cout;
      wire s1,c1,c2,c3;
      xor(s1,x,y);
      xor(sum,cin,s1);
      and(c1,x,y);
      and(c2,y,cin);
      and(c3,x,cin);
      or(cout,c1,c2,c3);
endmodule
```

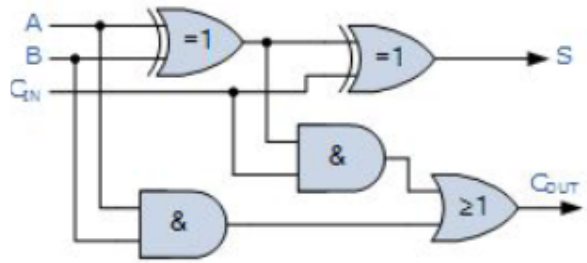
- **Dataflow modeling:**
  - Mainly used to describe combinational circuits. The basic mechanism used is the continuous assignment.

`assign [delay] LHS_net = RHS_expression;`

```
module full_adder(x,y,cin,sum,cout);
      input x,y,cin;
      output sum, cout;

      assign {cout, sum} = x + y + cin;

endmodule
```

- **Behavioral modeling**:
  - Used to describe complex circuits (primarily sequential). Mechanisms: **`initial`** and **`always`** statements.

```
module full_adder(x,y,cin,sum,cout);
      input x,y,cin;
      output reg sum, cout;

    always @ (x or y or cin) begin
        {cout, sum} = a + b + cin;
    end
endmodule
```

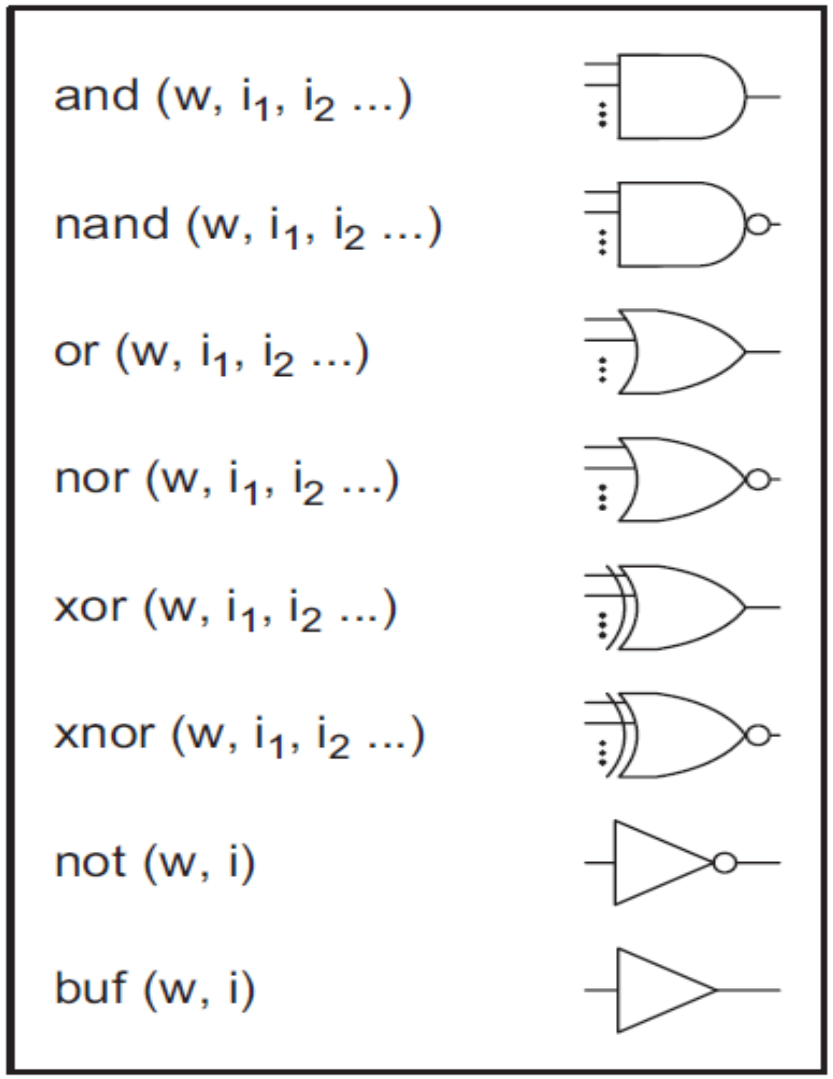



```
module full_adder(x,y,cin,sum,cout);
      input x,y,cin;
      output sum, cout;
      wire s1,c1,c2,c3;
      xor(s1,x,y);
      xor(sum,cin,s1);
      and(c1,x,y);
      and(c2,y,cin);
      and(c3,x,cin);
      or(cout,c1,c2,c3);
endmodule
```

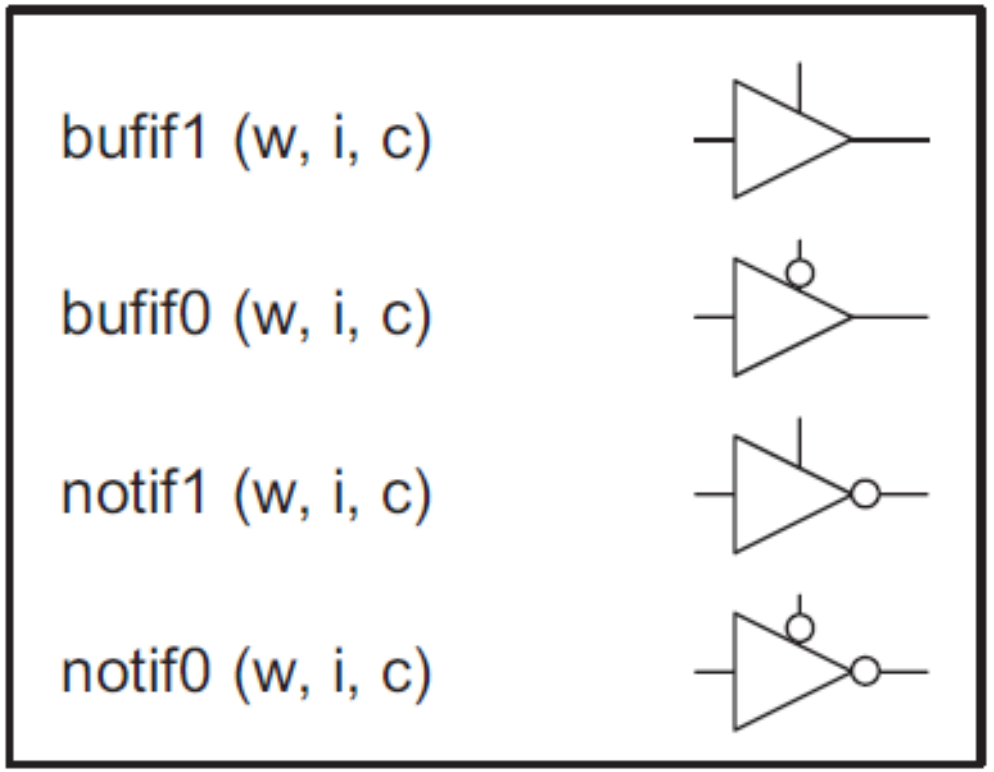
```
module full_adder(x,y,cin,sum,cout);
      input x,y,cin;
      output sum, cout;

      assign {cout, sum} = x + y + cin;

endmodule
```

```
module full_adder(x,y,cin,sum,cout);
      input x,y,cin;
      output reg sum, cout;

    always @ (x or y or cin) begin
         {cout, sum} = a + b + cin;
    end
endmodule
```

Verilog Gate Primitives



- Verilog provides primitive gates. One can instantiate these gates and use these gates to construct a logic circuit. This is also called Gate-level Structural Modelling.



Gates categorized as *n_input* gates are **and**, **nand**, **or**, **nor**, **xor**, and **xnor**. An *n_input* gate has one output, which is its left-most argument, and can have any number of inputs that may be listed as its argument separated by commas. These gates can have up to two delay parameters that can appear after the name of the gate in a set of parenthesis followed by a sharp sign. An example instantiation of a 4-input **nand** is shown here.

```
nand #(3, 5) gate1 (w, i1, i2, i3, i4);
```

In this example, t_{PLH} (low to high propagation) and t_{PHL} (high to low propagation) times are 3 and 5, respectively. Gate delays are optional,

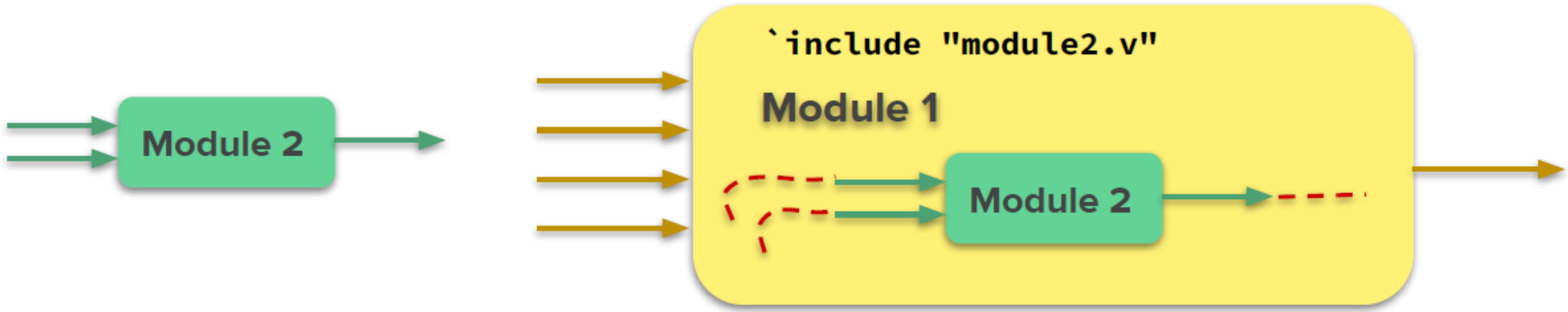
and if not included, 0 delay values are assumed

HA.v - D:/codes

```
1  module HA(a,b,s,c);
2  input a,b;
3  output s,c;
4  xor g1(s,a,b;
5  and g2(c,a,b);
6  endmodule
```

Verilog Language Features

Two Ways to Specify Connectivity During Module Instantiation

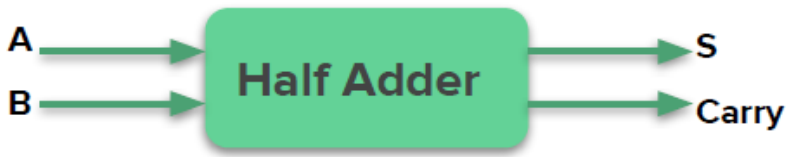


Connectivity of the signal lines between two modules can be specified in 2 ways:

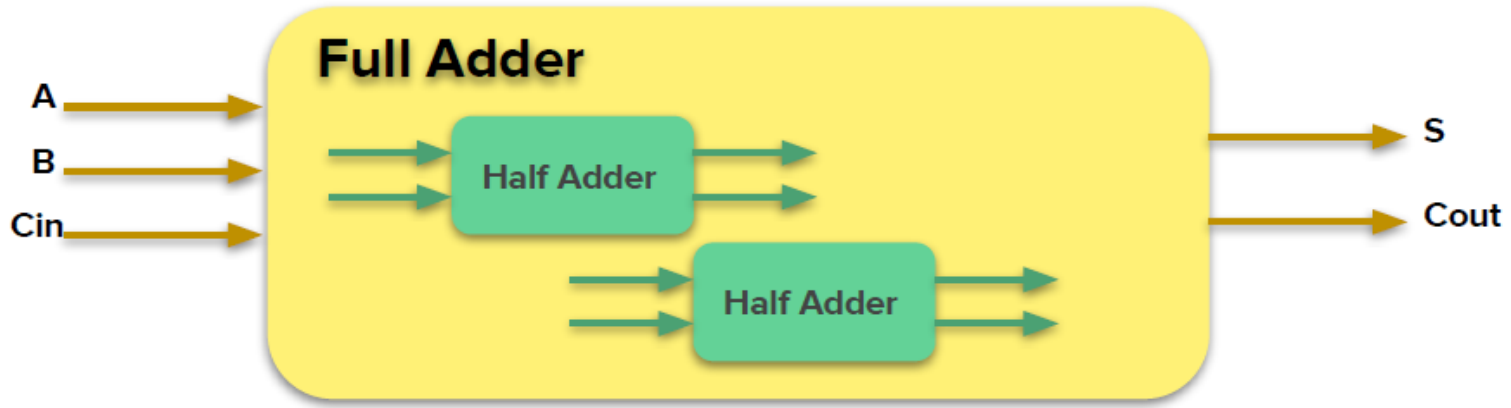
- **Positional Association (Implicit)**
  - Parameters listed **in the same order** as in the original module description.
- **Explicit Association**
  - Parameters **explicitly** listed **in arbitrary order**.

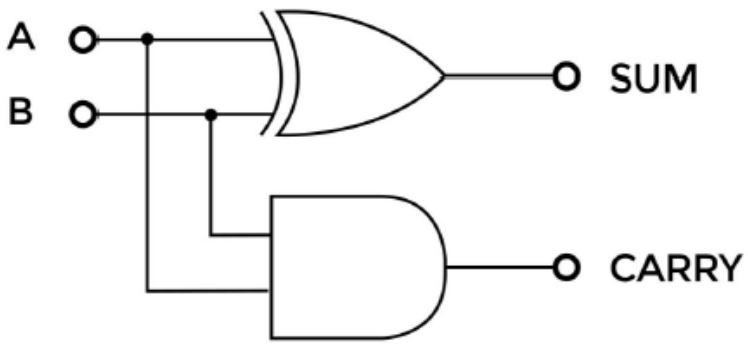
Verilog Language Features

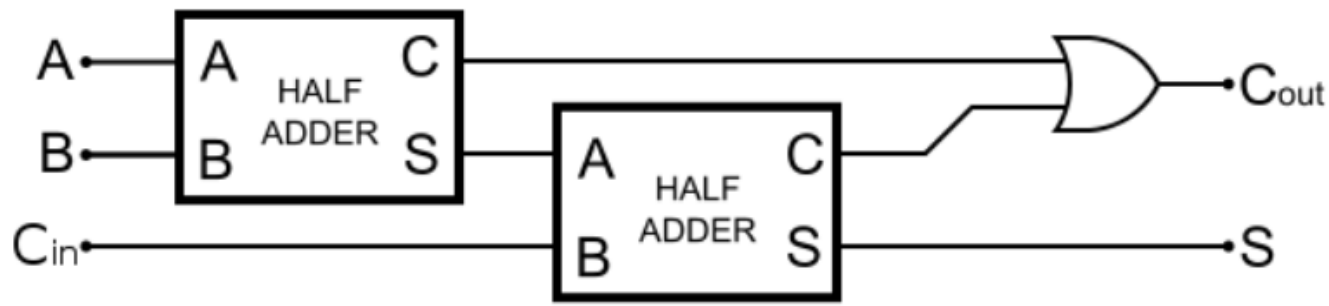
- **Positional Association Example - Full Adder Using Half Adder Module**



A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

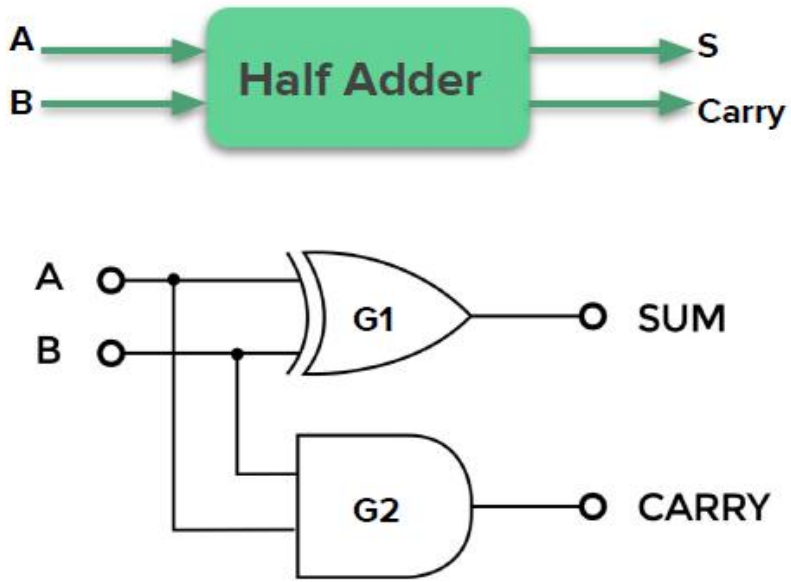






Verilog Language Features

- **Positional Association Example - Full Adder Using Half Adder Module**

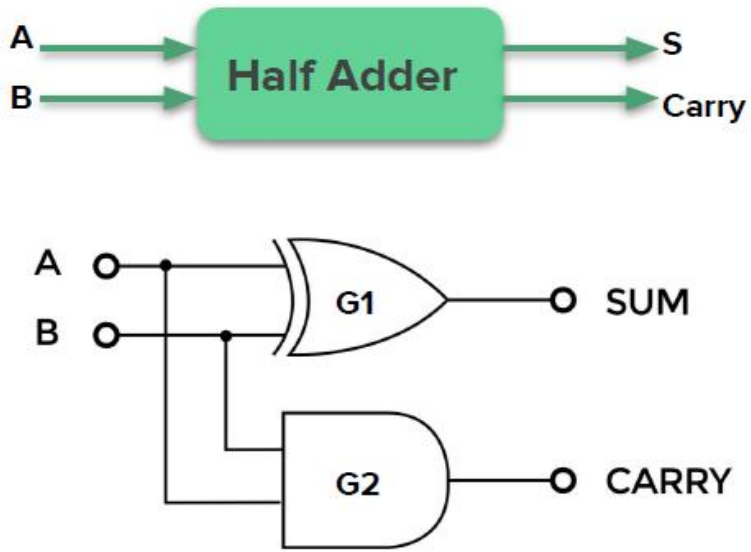



```
module half_adder (Sum, Carry, A, B);
    input A, B;
    output Carry, Sum;
    //structural description
    xor G1(Sum, A, B);
    and G2(Carry, A, B);
endmodule
```

What is the equivalent behavioral design?

Verilog Language Features

- **Positional Association Example - Full Adder Using Half Adder Module**



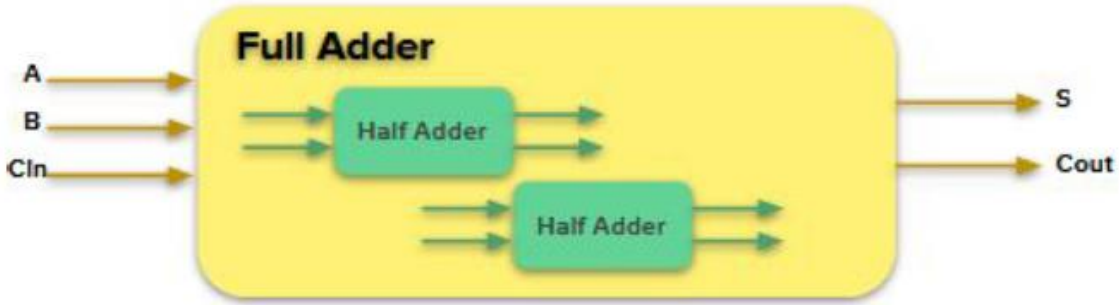

```
module half_adder (Sum, Carry, A, B);
    input A, B;
    output Carry, Sum;
    //structural description
    xor G1(Sum, A, B);
    and G2(Carry, A, B);
endmodule
```

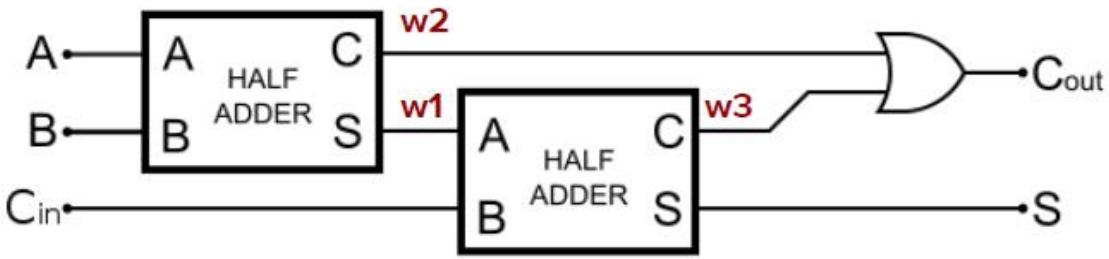
=

```
//behavioral description
assign Sum = A ^ B;
assign Carry = A & B;
```

Verilog Language Features

- **Positional Association Example - Full Adder Using Half Adder Module**





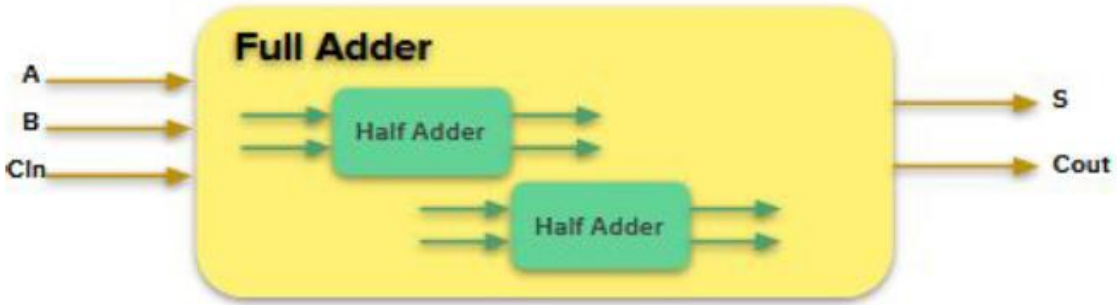
Note the port order

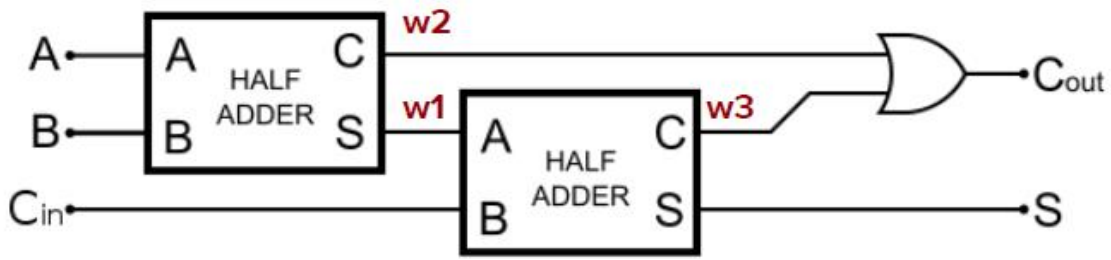
```
module half_adder (Sum, Carry, A, B);
```

```
module full_adder (Sum, Cout, A, B, Cin);
    input A, B, Cin;
    output Cout, Sum;
    wire w1, w2, w3;

    How do we use the half_adder
    module to implement a full adder?

endmodule
```

Verilog Language Features

- **Positional Association Example - Full Adder Using Half Adder Module**





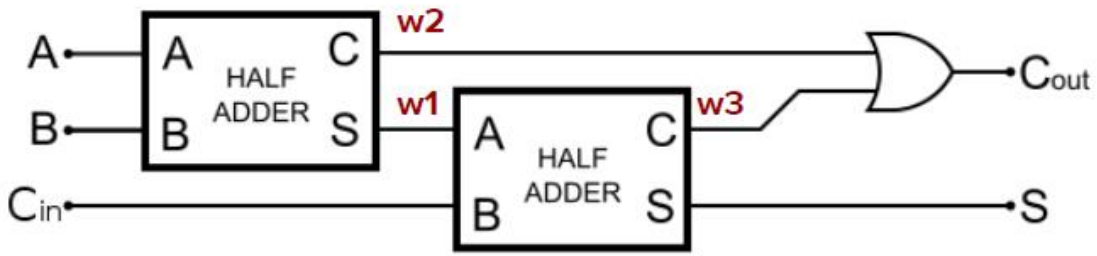
Note the port order

```
module half_adder (Sum, Carry, A, B);
```

```
module full_adder (Sum, Cout, A, B, Cin);
    input A, B, Cin;
    output Cout, Sum;
    wire w1, w2, w3;
    half_adder HA1 (w1, w2, A, B);
    half_adder HA2 (Sum, w3, Cin, w1);
    or (Cout, w2, w3);
endmodule
```

Verilog Language Features

- **Explicit Association Example - Full Adder Using Half Adder Module**




```
module half_adder (Sum, Carry, A, B);
```

```
module full_adder (Sum, Cout, A, B, Cin);
    input A, B, Cin;
    output Cout, Sum;
    wire w1, w2, w3;
    half_adder HA1 (.A(A), .B(B), .Sum(w1), .Carry(w2));
    half_adder HA2 (.Sum(Sum), .Carry(w3), .B(Cin), .A(w1));
    or (Cout, w2, w3);
endmodule
```

Ports are explicitly specified - order is not important

Less chance for errors

Wires and variables

Verilog has two main data types, net and reg. A net represents a wire driven by a hardware structure or output of a gate. A reg represents a variable that can be assigned values in a behavioral description of a component in a Verilog procedural block.

Verilog Operators

Bitwise Operators	&	|	^	~	~^	^~	
Reduction Operators	&	~&	|	~|	^	~^	^~
Arithmetic Operators	+	-	*	/	%		
Logical Operators	&&	||	!				
Compare Operators	<	>	<=	>=	==		
Shift Operators	>>	<<					
Concatenation Operators	{ }	{ n{ } }					
Conditional Operator	?:						

Arithmetic Operators

- * → multiply
- / → divide
- + → add
- - → subtract
- % → modulus

Some Examples

```
5 + 10 = 15
5 - 10 = -5
10 - 5 = 5
10 * 5 = 50
10 / 5 = 2
10 / -5 = -2

10 % 3 = 1
```

Logical Operators

- `&&` → logical AND
- `||` → logical OR
- `!` → logical NOT
- Operands evaluated to ONE bit value: *0*, *1* or *x*
- Result is ONE bit value: *0*, *1* or *x*

```
A = 1;          A && B → 1 && 0 → 0
B = 0;          A || !B → 1 || 1 → 1
C = x;          C || B → x || 0 → x
```

but C&&B=0

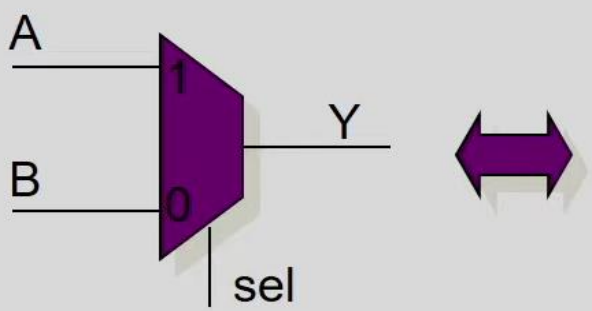
Shift, Conditional Operator

- >> → shift right
- << → shift left

- a = 4'b1010;

d = a >> 2;// d = 0010,c = a << 1;// c = 0100

- cond_expr ? true_expr : false_expr




```
Y = (sel)? A : B;
```

Summary: Bitwise Operators

```
module gates(input  [3:0]  a, b,
             output [3:0] y1, y2, y3, y4, y5);

   /* Five different two-input logic
      gates acting on 4 bit busses */

   assign y1 = a & b;     // AND
   assign y2 = a | b;     // OR
   assign y3 = a ^ b;     // XOR
   assign y4 = ~(a & b); // NAND
   assign y5 = ~(a | b); // NOR

endmodule
```

Keywords

- Note : All keywords are defined in lower case
- Some examples of Keywords are:
- module, endmodule
- input, output, inout
- reg
- parameter
- begin, end

```
module memory (“ports”);
input clock;
input address;
output reg read_enable;
inout data;
parameter DATA_WIDTH = 8 ;
parameter ADDR_WIDTH = 8 ;
parameter RAM_DEPTH = 1 << ADDR_WIDTH;
endmodule
```

Multiple statements must be enclosed

mux2_1.v - D:/codes

```
1  // Example of a verilog code
2
3  module mux2_1(i0,i1,s,y);
4  input [3:0] i0,i1;
5  input s;
6  output [3:0] y;
7  assign y=s?i1:i0;
8  endmodule
```

Verilog Hardware Modelling Styles

- Structural → Gate-level or Switch-level (Transistor)
- Dataflow
- Behavioral

majority_ckt1.v * - D:/codes

```
1  // Majority circuit using wire
2
3  module majority_ckt1(input a,b,c,output y);
4  wire s1,s2,s3,s4;
5  assign s1=a&b;
6  assign s2=b&c;
7  assign s3=c&a;
8  assign s4=s1|s2;
9  assign y=s3|s4;
10 endmodule
```

majority_ckt2.v - D:/codes

```
1  // Majority circuit using reg
2
3  module majority_ckt2(input a,b,c,output reg y);
4  always @(a or b or c) begin
5  y=(a&b)|(b&c)|(c&a);
6  end
7  endmodule
```

halfadder_dataflow.v * - D:/codes

```
1  // Dataflow design
2
3  module halfadder_dataflow(a,b,sum,carry);
4  input a,b;
5  output sum,carry;
6  assign sum=a^b;
7  assign carry=a&b;
8  endmodule
```

halfadder_structural.v * - D:/codes

```
1  // Structural design
2
3  module halfadder_structural(a,b,sum,carry);
4  input a,b;
5  output sum,carry;
6  xor g1(sum,a,b);
7  and g2(carry,a,b);
8  endmodule
```

halfadder_behavioral.v - D:/codes

```
1  // Behavioral design
2
3  module halfadder_behavioral(a,b,sum,carry);
4  input a,b;
5  output sum,carry;
6  assign {carry,sum}=a+b;
7  endmodule
```

HA.v * - D:/codes

```
1  module HA(a,b,s,c);
2  input a,b;
3  output s,c;
4  xor g1(s,a,b;
5  and g2(c,a,b);
6  endmodule
```

FA.v - D:/codes

```
1  module FA(a,b,cin,s,cout);
2  input a,b,cin;
3  output s,cout;
4  wire s1,c1,c2;
5  // instatiate HAs
6  HA m1(a,b,s1,c1,);
7  HA m2(s1,cin,s,c2,);
8  or o1(c,c1,c2);
9  endmodule
```

4_bit_RCA.v * - D:/codes

```
1  module 4bit_RCA(x, y, s, co);
2  input [3:0] x, y;// Two 4-bit inputs
3  output [3:0] s;
4  output co;
5  wire w1, w2, w3;
6   // instantiating 4 1-bit full adders in Verilog
7  FA u1(x[0],y[0], 1'b0, s[0], w1);
8  FA u2(x[1],y[1], w1, s[1], w2);
9  FA u3(x[2],y[2], w2, s[2], w3);
10 FA u4(x[3],y[3], w3, s[3], co);
11 endmodule
```

Verilog

Part-2

Recap

- Verilog is case sensitive
- Verilog supports inherently concurrent execution
- Synthesizable code writing is mandatory for FPGA implementation
- Specification → HDL → Synthesis may fail (Avoid direct coding from psudo code)
- Specification → Digital VLSI architecture design → HDL → Synthesis
- Event-driven simulation

Boolean level behaviour of Half adder

```
module half_adder(a, b, sum, carry);


input  a;
input  b;
output sum;
output carry;

// combinational half-adder logic
assign sum   = a ^ b;   // xor
assign carry = a & b;   // and

endmodule
```

🔹 What is a Continuous Assignment?

- A **continuous assignment** in Verilog is written using the `assign` keyword.
- It drives a **wire-type net** *continuously* with some logic expression.
- Meaning: whenever the right-hand side expression changes, the left-hand side net updates automatically.

It is used to model **combinational logic** (no clock, no memory).

🔹 Syntax

verilog Copy

```
assign <net_name> = <expression>;
```

- `<net_name>` → must be a **wire** (or something that defaults to wire, like an output without reg).
- `<expression>` → any logic, arithmetic, bitwise operation, concatenation, etc.

In Verilog, if a signal is **not declared with a data type** and is used in a **continuous assignment** (`assign`) or as a **module port**, it is **implicitly of type** `wire` .

```
module half_adder(a, b, sum, carry);

    // Port declarations (inside the module)
    input  a;
    input  b;
    output sum;
    output carry;

    // Logic
    assign sum   = a ^ b;
    assign carry = a & b;

endmodule
```

Can we describe more high level behaviour and ask synthesis tool to generate gate-level netlist?

```
module one_bit_adder (a, b, sum);

    // Port declarations inside the body
    input  a;           // 1-bit input
    input  b;           // 1-bit input
    output [1:0] sum;  // 2-bit output (carry + sum)

    // Continuous assignment
    assign sum = a + b;

endmodule
```

HA → one-bit-adder

```
module one_bit_adder (
    input  a,
    input  b,
    output [1:0] sum
);
    assign sum = a + b;    // continuous assignment
endmodule
```

Problem Statement

Design a Verilog module that performs a multiply–add operation.

- The module takes **two 8-bit inputs** `a` and `b`.
- It also takes a **16-bit input** `c`.
- The module should first **multiply** `a` and `b` to generate a 16-bit product.
- Then it should **add** this product to `c`.
- The final **output** `y` **is 17 bits wide**, since the addition may cause an overflow.

Problem: Design of Arithmetic Circuit

Design a **combinational** circuit to compute:

$$y = a \times b + c$$

Specifications

- `a` : 8-bit unsigned input
- `b` : 8-bit unsigned input
- `c` : 16-bit unsigned input
- `y` : 17-bit unsigned output

Implement in Two Ways

Method 1: Single-Module (Dataflow)

- Write **one Verilog module only**.
- Use only `assign` **statements**.
- Ensure correct **bit-width/zero-extension** so the result maps to **17 bits**.

Method 2: Structural Design

- Write two separate modules:
  1. **8×8 multiplier** → output **16-bit product**
  2. **17-bit adder** → adds (extended product) and (extended `c`)
- Write a **top module** that instantiates and connects multiplier + adder (structural coding), including all necessary bit extensions.

```
module mul_add (
    input  [7:0]  a,   // first input
    input  [7:0]  b,   // second input
    input  [15:0] c,   // third input
    output [16:0] y    // final result
);

    wire [15:0] product;   // output of multiplier
    wire [16:0] sum;       // result of adder

    // Step 1: multiply
    assign product = a * b;

    // Step 2: add product with input c
    assign sum = product + c;

    // Step 3: final output
    assign y = sum;

endmodule
```

```
//------------------------------
// Top Module (Structural Design)
// y = a*b + c
// a,b : 8-bit
// c   : 16-bit
// y   : 17-bit
//------------------------------
module top_design(a, b, c, y);
    input  [7:0]  a;
    input  [7:0]  b;
    input  [15:0] c;
    output [16:0] y;

    wire [15:0] prod;

    // Multiplier instance
    multiplier_8bit U1 (
        .a(a),
        .b(b),
        .p(prod)
    );

    // Adder instance
    adder_16bit U2 (
        .x(prod),
        .z(c),
        .s(y)
    );

endmodule
```

1. Design Full-adder using Half adder

2. Design 4- bit Ripple carry adder using Full-adder

HA.v - D:/codes

```
1  module HA(a,b,s,c);
2  input a,b;
3  output s,c;
4  xor g1(s,a,b;
5  and g2(c,a,b);
6  endmodule
```

FA.v - D:/codes

```
1  module FA(a,b,cin,s,cout);
2  input a,b,cin;
3  output s,cout;
4  wire s1,c1,c2;
5  // instatiate HAs
6  HA m1(a,b,s1,c1,);
7  HA m2(s1,cin,s,c2,);
8  or o1(c,c1,c2);
9  endmodule
```

4_bit_RCA.v - D:/codes

```
1  module 4bit_RCA(x, y, s, co);
2  input [3:0] x, y;// Two 4-bit inputs
3  output [3:0] s;
4  output co;
5  wire w1, w2, w3;
6   // instantiating 4 1-bit full adders in Verilog
7  FA u1(x[0],y[0], 1'b0, s[0], w1);
8  FA u2(x[1],y[1], w1, s[1], w2);
9  FA u3(x[2],y[2], w2, s[2], w3);
10 FA u4(x[3],y[3], w3, s[3], co);
11 endmodule
```

Can we describe a FA based on truth table?

```
 1  module full_adder_case(a, b, cin, sum, cout);
 2
 3      input  a;
 4      input  b;
 5      input  cin;
 6      output sum;
 7      output cout;
 8
 9      reg sum;
10      reg cout;
11
12      always @(a,b,cin) begin
13          case ({a, b, cin})
14              3'b000: begin sum = 1'b0; cout = 1'b0; end
15              3'b001: begin sum = 1'b1; cout = 1'b0; end
16              3'b010: begin sum = 1'b1; cout = 1'b0; end
17              3'b011: begin sum = 1'b0; cout = 1'b1; end
18              3'b100: begin sum = 1'b1; cout = 1'b0; end
19              3'b101: begin sum = 1'b0; cout = 1'b1; end
20              3'b110: begin sum = 1'b0; cout = 1'b1; end
21              3'b111: begin sum = 1'b1; cout = 1'b1; end
22              default: begin sum = 1'b0; cout = 1'b0; end
23          endcase
24      end
25
26  endmodule
27
```

Recap

- Continuous assignments
- Structural modelling → Module instantiation
- Sequential execution in HDL is also possible

In Verilog, numbers (constants/literals) are written in a radix format:

```
<bit_width>'<base><number>
```

1. `bit_width` → number of bits to represent the value
2. `base` → radix of the number system
   - `b` → binary
   - `o` → octal
   - `d` → decimal
   - `h` → hexadecimal
3. `number` → actual value in that base

```
8'b10110011   // 8-bit binary: 10110011 (decimal 179)
4'd9          // 4-bit decimal: 1001
12'hABC       // 12-bit hex: 1010_1011_1100
6'o27         // 6-bit octal: decimal 23
```

Model 2:1 MUX using HDL

- In Verilog, **procedural statements** are those **inside an** `always` **or** `initial` **block**.
- They describe hardware in a **step-by-step / sequential style** (like software instructions), but still represent parallel hardware when synthesized.

- An `always` **block** is a block of code in Verilog that **executes whenever a specified event occurs** on its sensitivity list.
- It is like saying: *"Whenever this signal changes, do these operations."*

```verilog
always @(sensitivity_list) begin
   // sequential statements
end
```

  - `sensitivity_list` : Signals or events that trigger execution.

```
module mux2to1(a, b, s, y);

    input  a;
    input  b;
    input  s;
    output y;

    reg y;

    always @(*) begin
        case (s)
            1'b0: y = a;
            1'b1: y = b;
        endcase
    end

endmodule
```

```
module mux2to1(a, b, s, y);

    input  a;
    input  b;
    input  s;
    output y;

    reg y;

    always @(*) begin
        if (s == 1'b0)
            y = a;
        else if (s == 1'b1)
            y = b;
    end

endmodule
```

// ❌ no default case

```
module mux2to1(a, b, s, y);

    input  a;
    input  b;
    input  s;
    output y;

    reg y;

    always @(*) begin
        case (s)
            1'b0: y = a;
            1'b1: y = b;
        endcase
    end

endmodule
```

// ❌ no final else

```
module mux2to1(a, b, s, y);

    input  a;
    input  b;
    input  s;
    output y;

    reg y;

    always @(*) begin
        if (s == 1'b0)
            y = a;
        else if (s == 1'b1)
            y = b;
    end

endmodule
```

Consequence → Dangerous for a purely combinational circuit

Missing 'else/default case' means the output isn't assigned for some inputs, so it retains its previous value, causing latch inference in synthesis

What happens?

1. **Output holds its previous value**

- When none of the conditions/case-items match, y is **not updated.**
- So, y keeps whatever value it had before.

2. **A latch is inferred in hardware**

- "Holding previous value" requires memory.
- Synthesis implements that memory as a **level-sensitive latch** (unwanted in a pure combinational MUX).

```
module mux2to1 (
    input  [3:0] a, b,
    input        sel,
    output reg [3:0] y
);
    always @(*) begin
        if (sel == 1'b0)
            y = a;
        else if (sel == 1'b1)
            y = b;
        else
            y = 4'bx;   // unknown when sel is not 0 or 1
    end
endmodule
```

```
59 module mux2to1 (
60     input  wire [3:0] a,
61     input  wire [3:0] b,
62     input  wire       sel,
63     output reg  [3:0] y
64 );
65 
66     always @(*) begin
67         case (sel)
68             1'b0: y = a;
69             1'b1: y = b;
70             default: y = 4'bxxxx;  // unknown if sel is floating/invalid
71         endcase
72     end
73 endmodule
74 
```

Describe 4:1 multiplexer in Verilog

A 4:1 multiplexer selects one of four inputs based on two select lines and forwards it to the output.

mux4to1.v - D:/codes/behavioral

```
1  module mux4to1(y,a,b,c,d,s);
2  input [3:0] a,b,c,d;
3  input [1:0] s;
4  output reg [3:0] y;
5  always @(a,b,c,d,s)
6  begin
7  case(s)
8  2'b00: y=a;
9  2'b01: y=b;
10 2'b10: y=c;
11 2'b11: y=d;
12 default:y=4'bxxxx;
13 endcase;
14 end
15 endmodule
```

🔹 Types of always blocks

1. **Combinational logic** (pure logic, no memory)

```verilog
always @(*) begin
   y = a & b;  // whenever a or b changes, this block runs
end
```

- `@(*)` means "sensitive to all signals used inside automatically."
- Used for describing **logic gates, multiplexers, decoders**, etc.

2. **Sequential logic** (flip-flops, registers → memory elements)

```verilog
always @(posedge clk or posedge rst) begin
   if (rst)
      q <= 0;        // asynchronous reset
   else
      q <= d;        // register stores data on clock edge
end
```

- `posedge clk` → execute on **rising edge** of clock.
- Describes **registers, counters, state machines**, etc.
- Non-blocking assignment (`<=`) is used to avoid race conditions.

🔹 What is `reg` in Verilog?

- `reg` (short for **register**) is a **data type** used to represent a variable that can **store a value until it is explicitly updated**.
- It does **not** necessarily mean a hardware register/flip-flop!

In **Verilog-1995/2001**, the rule is:

- A variable that is **assigned inside an `always` block** must be declared as `reg`.
- It doesn't matter whether the block is sequential (`posedge clk`) or purely combinational (`always @(*)`).

So, even if it's just combinational logic (no memory), you must still write `reg` — because the signal is updated procedurally, not by a continuous assignment.

Functional simulation of FA

Full adder code

```
1  `timescale 1ns / 1ps
2  module fa(
3       input a,
4       input b,
5       input cin,
6       output sum,
7       output cout
8       );
9  wire s1,c1,c2;
10 ha m1(a,b,s1,c1);
11 ha m2(s1,cin,sum,c2);
12 or g1(cout,c1,c2);
13 endmodule
```

Test Bench code

```
1  `timescale 1ns / 1ps
2  module fa_tb;
3      // Inputs
4      reg a;
5      reg b;
6      reg cin;
7      // Outputs
8      wire sum;
9      wire cout;
10     // Instantiate the Design Under Test (DUT)
11     fa dut (.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));
12     initial begin
13         // Initialize Inputs
14         a = 1'b0; b = 1'b0; cin = 1'b0;
15         // Add stimulus here
16         #100 a = 1'b0; b = 1'b0; cin = 1'b1;
17         #100 a = 1'b0; b = 1'b1; cin = 1'b0;
18         #100 a = 1'b0; b = 1'b1; cin = 1'b1;
19         #100 a = 1'b1; b = 1'b0; cin = 1'b0;
20         #100 a = 1'b1; b = 1'b0; cin = 1'b1;
21         #100 a = 1'b1; b = 1'b1; cin = 1'b0;
22         #100 a = 1'b1; b = 1'b1; cin = 1'b1;
23     end
24     initial begin
25     $monitor($time," a=%b,b=%b,cin=%b,sum=%b,cout=%b",a,b,cin,sum,cout);
26     #10000 $finish;
27     end
28 endmodule
```

```v
// File: adder.v
module adder (
    input  [7:0] a,       // 8-bit input A
    input  [7:0] b,       // 8-bit input B
    output [8:0] sum      // 9-bit output (to hold carry)
);
    assign sum = a + b;  // simple combinational addition
endmodule
```

```v
// File: tb_adder.v
`timescale 1ns/1ps
module tb_adder;

  // Declare signals to connect to DUT
  reg  [7:0] a, b;
  wire [8:0] sum;

  // Instantiate the DUT
  adder uut (
    .a(a),
    .b(b),
    .sum(sum)
  );

  // Stimulus block
  initial begin
    // Apply a few test cases
    a = 8'd5;   b = 8'd3;    #10;

    a = 8'd10;  b = 8'd7;    #10;

    a = 8'd100; b = 8'd55;   #10;

    a = 8'd255; b = 8'd1;    #10;

    // Finish simulation
    $finish;
  end
```

```
`timescale <time_unit>/<time_precision>
```

- **time_unit** = the unit used by `#` delays.
- **time_precision** = the resolution at which the simulator rounds.

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

1. Time unit (`1ns`)

- This defines the **base unit of time** used in your simulation delays (`#`).
- Example:

```verilog
#10 clk = ~clk;
```

With `timescale 1ns/1ps`, this means `#10 = 10 ns`.

2. Time precision (`1ps`)

- This sets the **resolution** or smallest step the simulator can distinguish when rounding delays.
- In `1ns/1ps` :
  - All time values are expressed in **nanoseconds**.
  - The simulator keeps track of delays with a **precision of 1 picosecond**.

`timescale 1ns/1ps` means:

- **1 ns = 1 simulation time unit**
- **Delays are tracked with accuracy of 1 ps**

```
#0.5 clk = ~clk;  // 0.5 ns delay
```

This works because 0.5 ns = 500 ps, and precision is 1 ps.

If your precision was coarser (e.g. `1ns/1ns`), then `#0.5` would be rounded off to 1 ns.

```
`timescale 1ns/0.5ns
#8
#7
```

1. Meaning of `1ns/0.5ns`

- **Time unit = 1 ns** → all `#` delays are expressed in nanoseconds.
- **Time precision = 0.5 ns** → simulator rounds to the nearest **0.5 ns step**.

So the smallest distinguishable delay is 0.5 ns.

2. Case `#8`

- Requested delay = **8 ns**
- In precision steps:

$$8\ \text{ns} / 0.5\ \text{ns} = 16\ \text{steps}\ (\text{exact})$$

- No rounding → Effective delay = **8 ns**

- `#7.25` ns → 7.25 / 0.5 = 14.5 steps → rounds to 15 steps = 7.5 ns
- `#7.1` ns → 7.1 / 0.5 = 14.2 steps → rounds to 14 steps = 7.0 ns
- `#7.3` ns → 7.3 / 0.5 = 14.6 steps → rounds to 15 steps = 7.5 ns

So **all delays are snapped to the nearest 0.5 ns.**

System task

- A **system task** is a **built-in command provided by the simulator**, not synthesized into hardware.
- They **control or observe the simulation**, but **do not describe real hardware**.
- All system tasks start with a **$ symbol** (dollar sign) —
- They are part of the **simulation system**, not the circuit.

Most useful system tasks in Verilog TB

`$display` — print *once immediately*

Purpose:

Print text or variable values **immediately** when the simulator executes it.

What it does:

- Prints the message **right now** (at the current simulation time).
- Works like `printf()` in C.
- Does **not** automatically repeat.

`$display` format basics

`$display` lets you print variables in a formatted way:

```verilog
$display("a=%d b=%d sum=%d", a, b, sum);
```

Here:

- `%d` → print as **decimal**
- `%b` → print as **binary**
- `%h` → print as **hexadecimal**
- `%t` → print **simulation time**
- `%0t` or `%4t` → print time with **field width formatting**

```
$display("a=%d b=%d sum=%d", a, b, sum);
```

```
a=5 b=3 sum=8
```

`$monitor` — print *automatically on any signal change*

Keep watching one or more signals, and print **every time** any of them changes.

What it does:

- Prints a line **whenever** any listed signal changes value.
- Runs **continuously** until simulation ends (or you turn it off).

```
$monitor("t=%0t clk=%b rst=%b a=%d b=%d sum=%d",
         $time, clk, rst, a, b, sum);
```

```
t=0 clk=0 rst=1 a=0 b=0 sum=0
t=15 clk=1 rst=0 a=0 b=0 sum=0
t=25 clk=1 rst=0 a=5 b=3 sum=8
t=35 clk=1 rst=0 a=10 b=7 sum=17
```

What it does:

- Prints a line **whenever** any listed signal changes value.
- Runs **continuously** until simulation ends (or you turn it off).
- You only write it **once** — no loops or delays needed.

`$finish` — end the simulation

Purpose:

Stops simulation completely and closes all output files or waveforms.

Example:

```verilog
$finish;
```

What it does:

- Tells the simulator "I'm done — stop now."
- Used at the end of your testbench (after all vectors applied).

```
initial begin
  $display("Simulation started at t=%0t", $time);
  $monitor("t=%0t a=%0d b=%0d sum=%0d", $time, a, b, sum);

  a=5; b=3;
  #10 a=10; b=7;
  #10 a=255; b=1;

  #10;
  $display("Simulation finished at t=%0t", $time);
  $finish;
end
```

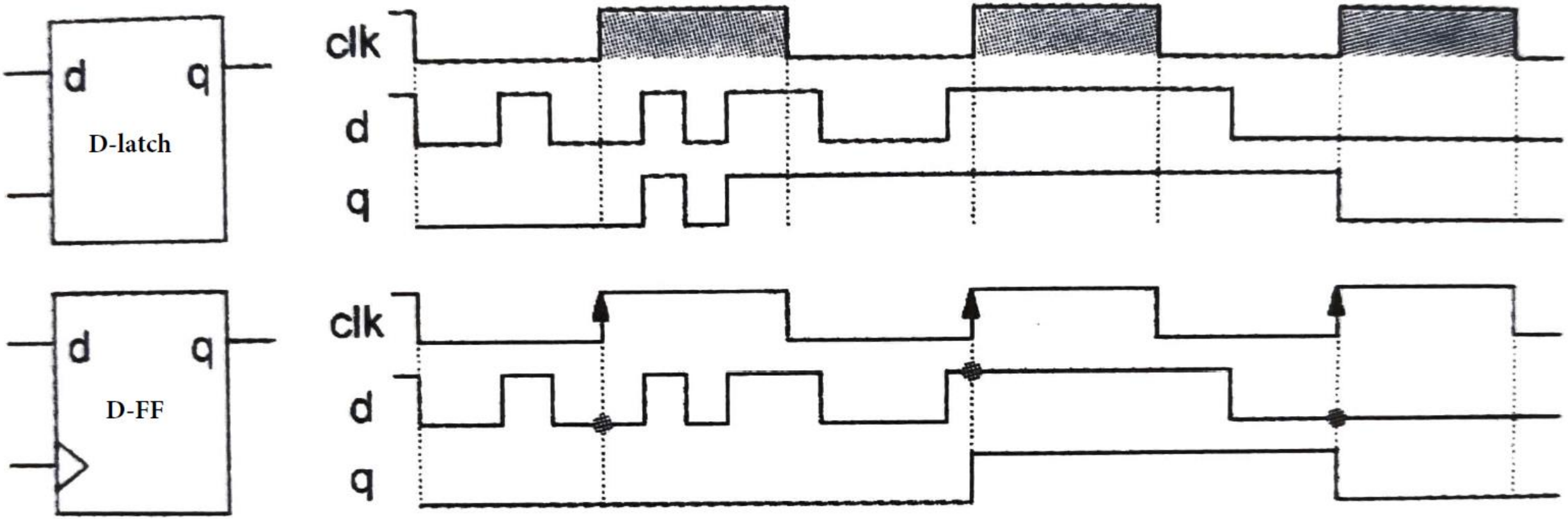
Console Output:

```
Simulation started at t=0
t=0 a=5 b=3 sum=8
t=10 a=10 b=7 sum=17
t=20 a=255 b=1 sum=256
Simulation finished at t=30
```

```
initial begin
  a = 0; b = 0;
  #10 a = 5; b = 3;  $display("Time=%0t a=%0d b=%0d sum=%0d", $time,a,b,sum);
  #10 a = 10; b = 7; $display("Time=%0t a=%0d b=%0d sum=%0d", $time,a,b,sum);
  #10 a = 255; b = 1;$display("Time=%0t a=%0d b=%0d sum=%0d", $time,a,b,sum);
end
```

```
initial begin
  $monitor("Time=%0t a=%0d b=%0d sum=%0d", $time, a, b, sum);
  a = 0; b = 0;
  #10 a = 5; b = 3;
  #10 a = 10; b = 7;
  #10 a = 255; b = 1;
  #10 $finish;
end
```

Write the behaviour of memory elements in Verilog



```
module d_latch (
    input  wire d,
    input  wire en,
    output reg  q
);
    always @(d or en) begin
        if (en)
            q <= d;
    end
endmodule
```

Why “no else” creates a latch

A latch is basically **a memory element** that **holds its last value** when it’s not enabled.

- When `en = 1`, the statement `q <= d;` executes, so **q follows d** (transparent).
- When `en = 0`, the `if (en)` condition is false, and because there is **no** `else`, **q is not assigned** in that execution of the always block.

In hardware inference terms:

- If a signal (`q`) is **not assigned for some input conditions**, the synthesizer must create storage so that `q` can **keep its old value**.
- That storage element is inferred as a **level-sensitive latch** controlled by `en`.

So “**missing else**” ⇒ “**hold previous value**” ⇒ “**latch inferred**”.

If `d` is removed from the sensitivity list

Example:

```verilog

always @(en) begin
    if (en)
        q <= d;
end
```

- The `always` block **triggers only when** `en` **changes**
- If `en = 1` and `d` **changes**, the block **does NOT execute**
- So `q` will **NOT update** when `d` changes
- This causes **simulation–hardware mismatch**

```
 1 module d_ff (
 2     input  wire clk,
 3     input  wire d,
 4     output reg  q
 5 );
 6     always @(posedge clk) begin
 7         q <= d;
 8     end
 9 endmodule
10
```

A **synchronous reset** resets the register **only on the active clock edge**.

- Reset is **checked inside** the clocked `always` block
- Reset takes effect **only when the clock arrives**

An **asynchronous reset** resets the register **immediately**, independent of the clock.

- Reset is included in the sensitivity list
- Reset acts **as soon as it is asserted**

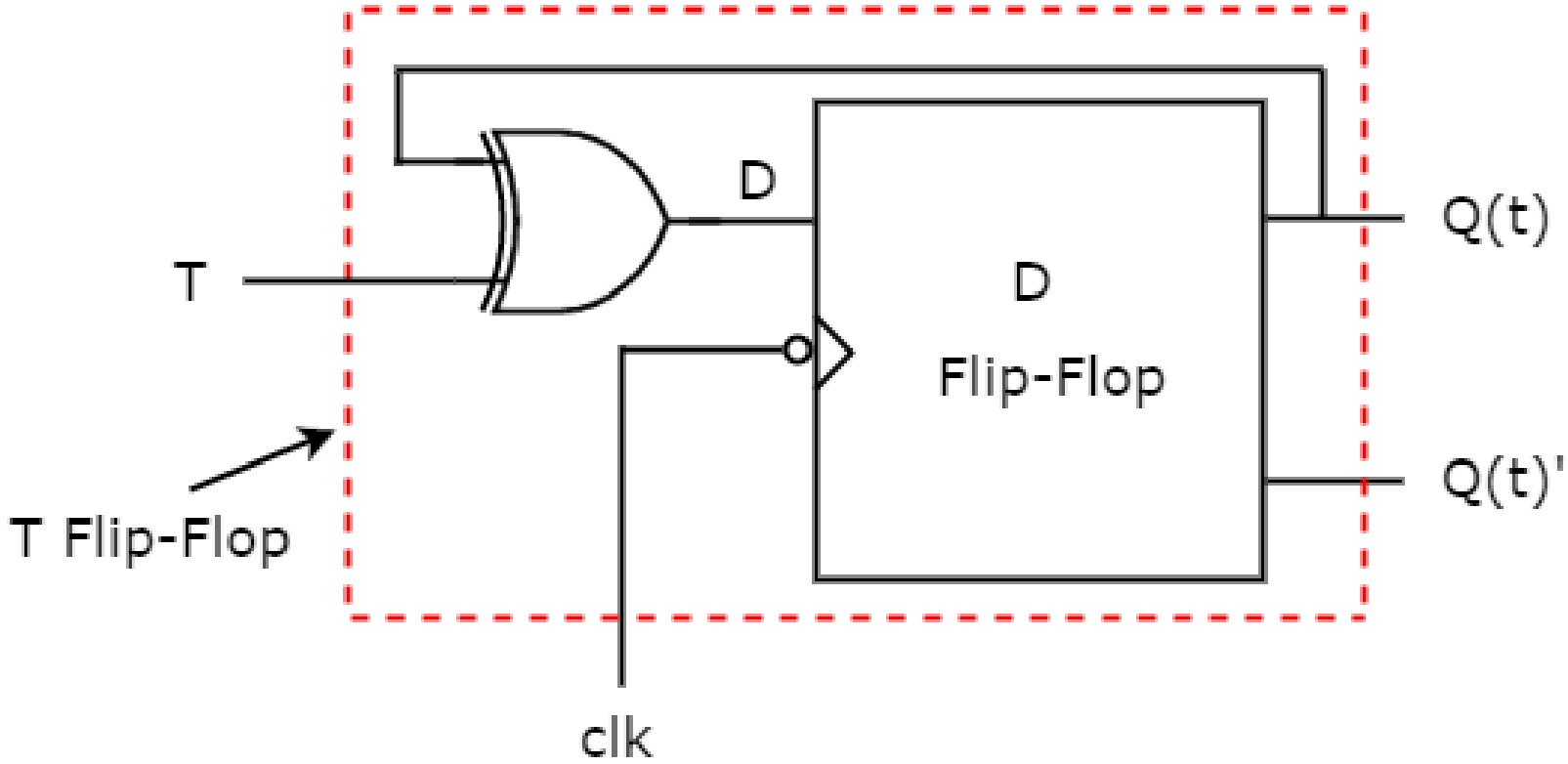
Conceptual meaning:

> *Reset directly controls the register, bypassing the clock.*

D-flip-flop with asynchronous reset

```
module d_ff_async_reset (
    input  wire clk,
    input  wire rst,
    input  wire d,
    output reg  q
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            q <= 1'b0;
        else
            q <= d;
    end
endmodule
```

```
module d_ff_sync_reset (
    input  wire clk,
    input  wire rst,
    input  wire d,
    output reg  q
);
    always @(posedge clk) begin
        if (rst)
            q <= 1'b0;
        else
            q <= d;
    end
endmodule
```


T
D
D
Flip-Flop
Q(t)
Q(t)'
clk
T Flip-Flop

A mixture of assign (combinational) and always (sequential/register) is the standard way to write LOGIC that matches real hardware concurrency.

```
module t_ff_using_dff (
    input  wire clk,
    input  wire rst,
    input  wire t,
    output reg  q
);
    wire d;

    assign d = q ^ t;

    always @(posedge clk or posedge rst) begin
        if (rst)
            q <= 1'b0;
        else
            q <= d;
    end
endmodule
```

All assign statements and all always blocks execute concurrently in Verilog.

Meaning in hardware terms

- `assign` models **combinational hardware** that is always active
- `always` models **independent hardware processes** (registers or combinational logic)
- Every `assign` and every `always` block represents a **separate hardware block**
- All these blocks exist and operate **in parallel**, just like real circuits

You can convert an assign (continuous assignment) into a combinational always block by following this rule:

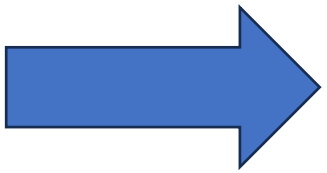
Rule

`assign y = expr;` ⟶ `always @(signals_in_expr) y = expr;`

Using `assign`:

verilog

```
wire y;
assign y = a & b;
```



Using `always` (combinational):

verilog

```
reg y;
always @(a or b) begin
    y = a & b;
end
```

```
module t_ff_two_always (
    input  wire clk,
    input  wire rst,
    input  wire t,
    output reg  q
);
    reg d;

    always @(q or t) begin
        d = q ^ t;
    end

    always @(posedge clk or posedge rst) begin
        if (rst)
            q <= 1'b0;
        else
            q <= d;
    end
endmodule
```

1. What concurrency means in hardware

In **real hardware**:

- All logic blocks operate **at the same time**
- Flip-flops, adders, multiplexers, FSM logic all work **in parallel**
- There is **no concept of sequential execution** like in software

> **Hardware is inherently concurrent.**

2. How Verilog models hardware concurrency

Verilog preserves hardware concurrency using:

- **Multiple `always` blocks**
- **Continuous assignments (`assign`)**

Each `always` block represents an **independent hardware process**.

> **Every `always` block runs in parallel with every other `always` block.**

In Verilog, this is written as:

```verilog
always @(posedge clk)        // Register 1
always @(a or b or c)        // Combinational logic
always @(posedge clk)        // Register 2
```

Hardware interpretation

- Block-1 models **input flip-flops**
- Block-2 models **combinational logic**
- Block-3 models **output flip-flops**

All three blocks **exist and operate simultaneously in hardware**.

Multiple always blocks preserve concurrency in Verilog by modeling independent hardware processes that operate in parallel, matching the inherently concurrent nature of digital hardware and enabling correct RTL synthesis.

```
 1 module counter_4bit (
 2     input  wire clk,
 3     input  wire rst,
 4     output reg  [3:0] q
 5 );
 6     reg  [3:0] d;
 7
 8     always @(q) begin
 9         d = q + 4'b0001;
10     end
11
12     always @(posedge clk or posedge rst) begin
13         if (rst)
14             q <= 4'b0000;
15         else
16             q <= d;
17     end
18 endmodule
19
```

```
module pipo_loadable (
    input  wire       clk,
    input  wire       rst,
    input  wire       load,
    input  wire [3:0] d,
    output reg  [3:0] q
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            q <= 4'b0000;
        else if (load)
            q <= d;
    end
endmodule
```

Shift-left register (4-bit)

verilog

```
module shift_left_reg (
    input  wire       clk,
    input  wire       rst,
    input  wire       en,
    input  wire       sin,
    output reg  [3:0] q
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            q <= 4'b0000;
        else if (en)
            q <= {q[2:0], sin};
    end
endmodule
```

Shift-right register (4-bit)

verilog

```
module shift_right_reg (
    input  wire       clk,
    input  wire       rst,
    input  wire       en,
    input  wire       sin,
    output reg  [3:0] q
);
    always @(posedge clk or posedge rst) begin
        if (rst)
            q <= 4'b0000;
        else if (en)
            q <= {sin, q[3:1]};
    end
endmodule
```

```
 1 module universal_shift_reg (
 2     input  wire       clk,
 3     input  wire       rst,
 4     input  wire       en,
 5     input  wire       load,
 6     input  wire       dir,
 7     input  wire       sin,
 8     input  wire [3:0] d,
 9     output reg  [3:0] q
10 );
11     always @(posedge clk or posedge rst) begin
12         if (rst)
13             q <= 4'b0000;
14         else if (en) begin
15             if (load)
16                 q <= d;
17             else if (dir)
18                 q <= {q[2:0], sin};
19             else
20                 q <= {sin, q[3:1]};
21         end
22     end
23 endmodule
```