# DIGITAL SYSTEM DESIGN USING VERILOG

- **Course code:** 18EC5DCDSV
- **Credits:** 03

**Course Objectives:**

- To study Verilog HDL in high-level synthesis of digital system designs.
- To learn Verilog HDL for modelling, simulating and synthesizing various digital modules.
- To study the additional features of Verilog HDL
- To be able to design and develop Test bench for Combinational and sequential modules
- To be able to design fixed and floating point arithmetic
- To be able to design and implement synchronous sequential circuits using FSM

# Course Outcomes

| | |
|---|---|
| **CO1** | **Describe & model digital blocks of computing systems using Verilog hardware description language..** |
| **CO2** | Apply hardware description language for writing test benches to check combinational & sequential modules. |
| **CO3** | Design clocked synchronous circuits and perform timing analysis |
| **CO4** | Analyze digital System and model using Verilog HDL. |
| **CO5** | Apply design Knowledge to FSM based digital Modules. |
| **CO6** | Synthesize and implement a given digital system. |

# Module 1

- **Computer-Aided Design , Hardware Description Languages, Module modelling styles(ming bo lin), Verilog Description of Combinational Circuits,**
- **Data flow modelling-dataflow modelling, operands, operators, (ming bo lin), Verilog Modules, Verilog Assignments, Procedural Assignments), Modeling Flip-Flops Using Always Block, Always Blocks Using Event Control Statements.[Text book 1 and 3] Delays in Verilog, Compilation, Simulation, and Synthesis of Verilog Code , Simple Synthesis Examples**
- TEXT BOOKS:Charles H Roth Jr., Lizy Kurian John, Byeong –kill-lee “Digital System Design using Verilog”, *publisher Cengage learning*
- Ming-Bo Lin, “Digital System designs and Practices using Verilog HDL and FPGAs”, *John Wiley & Sons*, 2008

# Introduction to verilog

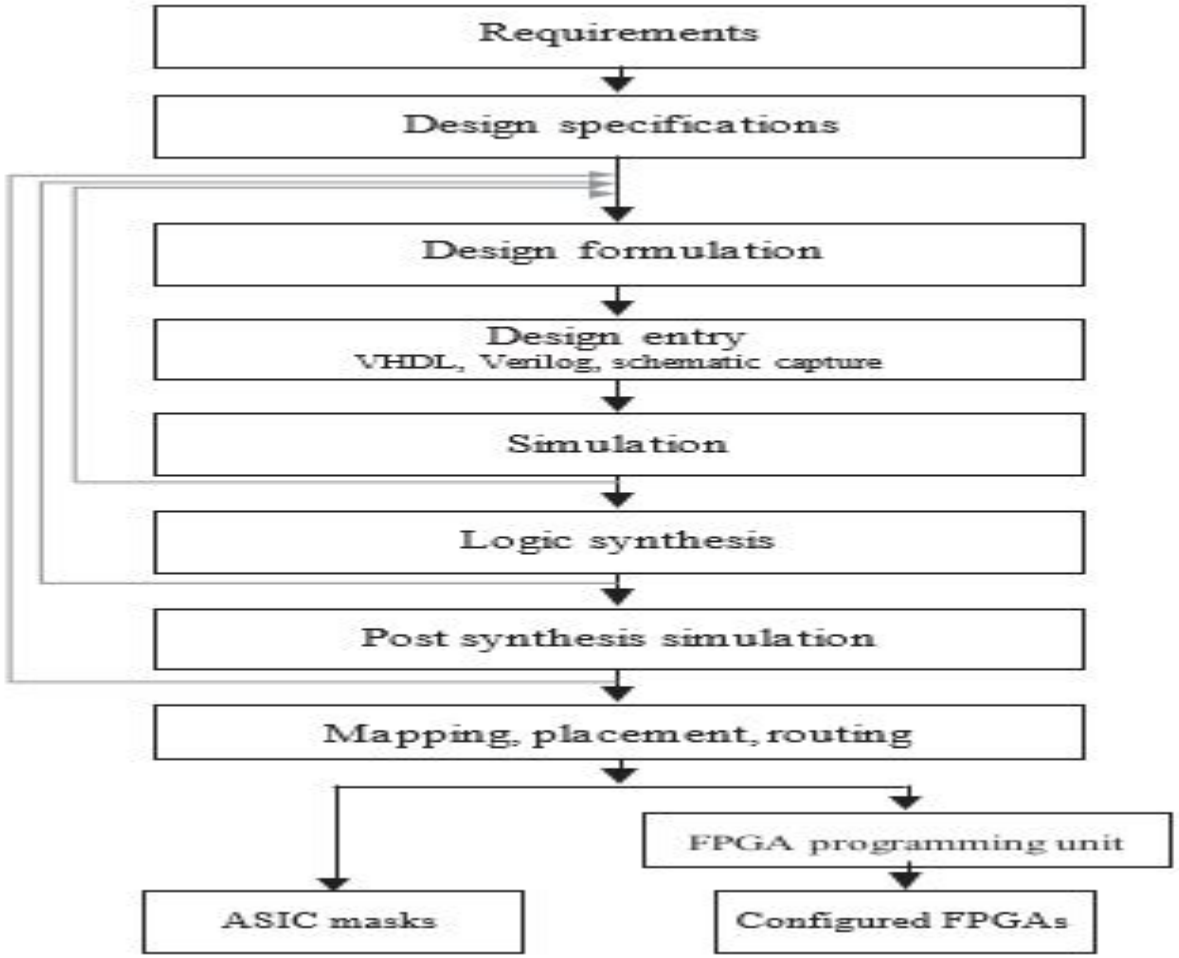
- Complexity
- Tedious and Time consuming
- Hand drawn Schematics,breadbords and wires
- More and more components on a chip,
- Improvements have advanced the **VLSI** (very large scale integration) field

- **SSI** (small-scale integration),
- **MSI** (medium-scale integration),
- **LSI** (large-scale integration)
- All depends on the density of integration.
- SSI referred to ICs with 1 to 20 gates
- MSI referred to ICs with 20 to 200 gates
- LSI referred to devices with 200 to a few thousand gates
- Building blocks such as adders, multiplexers, decoders, regis ters, and counters are available as MSI standard parts.
- Term **VLSI** was coined, devices with 10,000 gates were called VLSI chips. The boundaries between the different categories are fuzzy today. Many modern microprocessors contain more than 100 million transistors. Compared to what was referred to as VLSI in its initial days
- Modern integration capability could be described as **ULSI** (ultra large scale integration). Despite the changes in integration ability and the fuzzy defini- tion, the term VLSI remains popular.

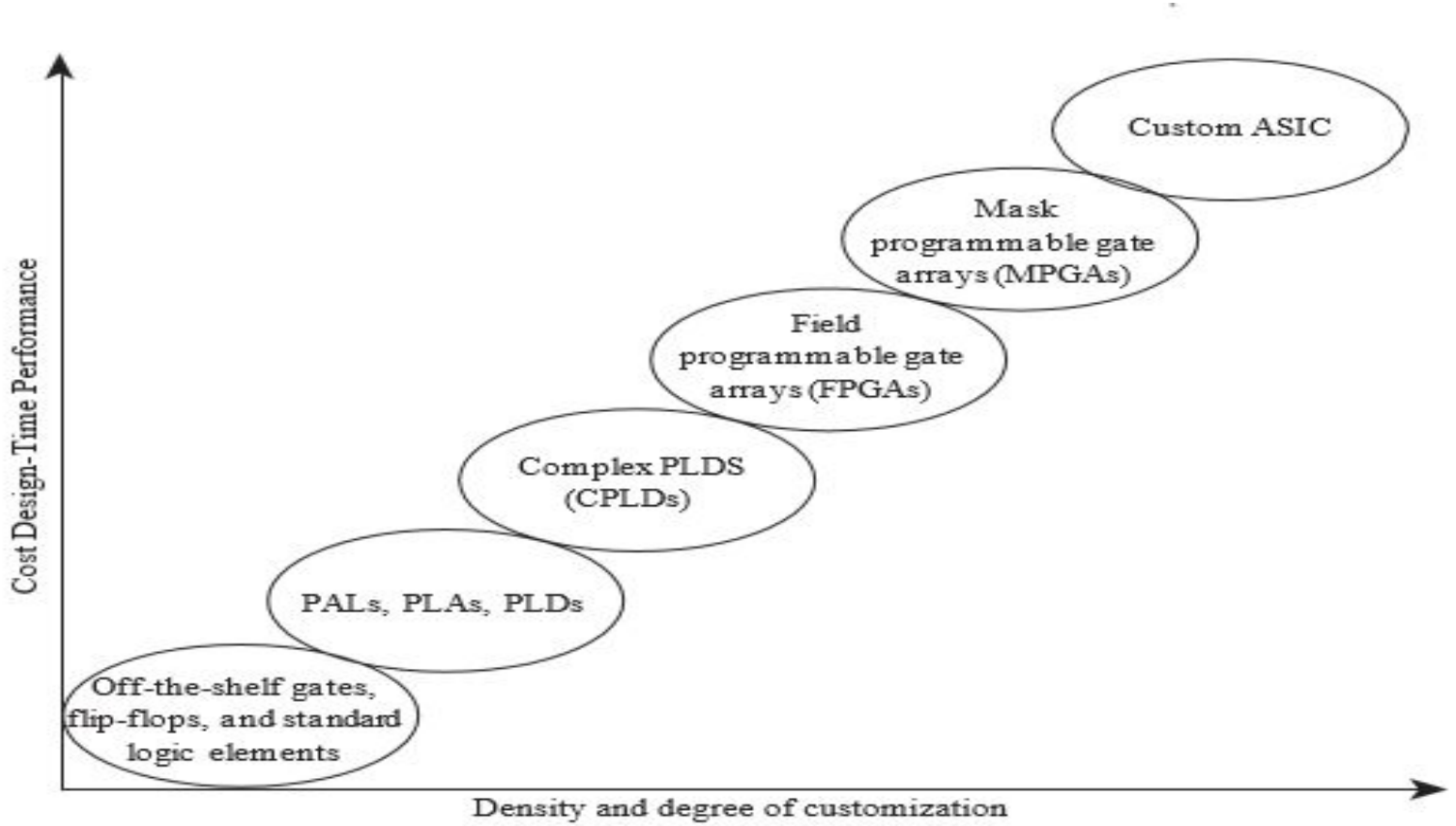
# Computer-aided design

- **(CAD)** tools have advanced significantly during the past decade, and nowadays digital design is performed using a variety of software tools.
- Prototypes or even final designs can be created without discrete components and interconnection wires.

# Design Flow in Modern Digital System Design



# Spectrum of Design Technologies



# Hardware Description Languages

- Hardware description languages (HDLs) are a popular mode of design entry for digital circuits and systems.
- There are two popular HDLs—VHDL and Verilog.
- Before the advent of HDLs, designers used graphical schematics and schematic capture tools to document and simulate digital circuits.
- A need was felt to create a textual method of documenting circuits and feeding them into simulators in the textual form as opposed to a graphic form.

# Verilog

- Verilog is a hardware description language used to describe the behavior and/ or structure of digital systems
- Verilog is a general-purpose hardware description language that can be used to describe and simulate the operation of a wide variety of digital systems, ranging in complexity from a few gates to an interconnection of many complex integrated circuits.
- VHDL was orig- inally developed under funding from the Department of Defense (DoD)
- Verilog was developed by the industry. It was initially developed as a proprietary language by a company called Gateway Design Automation around 1984.
- In 1990, Cadence acquired Gateway Design Automation and became the owner of Verilog. Cadence marketed it as a language and as a simulator, but it remained pro- prietary.
- At this time, Synopsis was promoting the concept of top-down design using Verilog. Cadence realized that it needed to make Verilog open in order to prevent the industry from shifting to VHDL and hence opened up the language.
- An organization called Open Verilog International (OVI) was formed, which helped to create a ven- dor-independent language specification for Verilog, clarifying many of the confusions around the proprietary specification. This was followed by an effort to create an IEEE standard for Verilog. The first Verilog IEEE Standard was created in 1995, which was revised in 2001 and 2005. Synopsis created synthesis tools for Verilog around 1988

# Module Modelling Styles

- Synthesis is process that converts HDL mdules into a structural description
- It s divided into logic synthesis and high level synthesis
- RTL description  into gate level netlist
- High level description into RTL results

- MODULES

 Two parts

- Interface
- Internal

Modeling the Internal of a  Module

- 1.Structural Style
  - -Gate Level
  - -Switch Level

2.Dataflow Style

-Module is specified as a set of continuous assignments statements

3.Behavioral or algorithmic style

4.Mixed style

-modelling Large Designs

# Port Declaration

- Interface signals of any Verilog HDL module can be of three types
  - Input
  - output
  - Inout
- The complete interface of a module is to divide it into three parts
  - Port list
  - Port Declaration
  - Data type declaration of each type

# Port List style

```
module adder(x,y,c_in,sum,c_out)
   Input [3:0] x,y;
   Input c_in;
   output [3:0] sum;
   output c_out;
   reg [3:0] sum;
   reg c_out;
```

This style of port declaration is known as port list style

```
module adder(x,y,c_in,sum,c_out)
   Input [3:0] x,y;
   Input c_in;
   output  reg [3:0] sum ;
   output  reg c_out;

module adder(input [3:0]x,y, input c_in, output  reg [3:0] sum
   output  reg c_out );
```

# Port Connection Rules

- Named association
- Positional association

*Coding Styles*

- A module cannot be declared within another module
- A module can instantiate other modules
- A module instantiation must have a module identifier (instance name) except built in primitives,gate and switch primitives and user defined primitives

Named association at the top level modules to avoid confusion

# Port connection Rules

```
module half adder(x,y,s,c);
Input x,y;
Output ,c;
//half adder body//
//instantiate primitive gates
xor xor1(s,x,y);          ( s,x,y)-Positional association
and and1 (c,x,y);      and1 (instance name is optional)
endmodule
```

```
module full _adder(x,y,cin,s,cout);
Input x,y,cin;
output s,cout;
wire s1,c1,c2;
// full adder body//
// instantiate the half adder
half adder ha_1(x,y,s1,c1);          // (x,y,s1,c1)-Positional  association
half adder ha_2(.x(cin), .y(s1), .s(s), .c(c2)); //ha_2(instance name is necessary)
Or (cout,c1,c2);                     //(.x(cin), .y(s1), .s(s), .c(c2))-Named association
endmodule
```

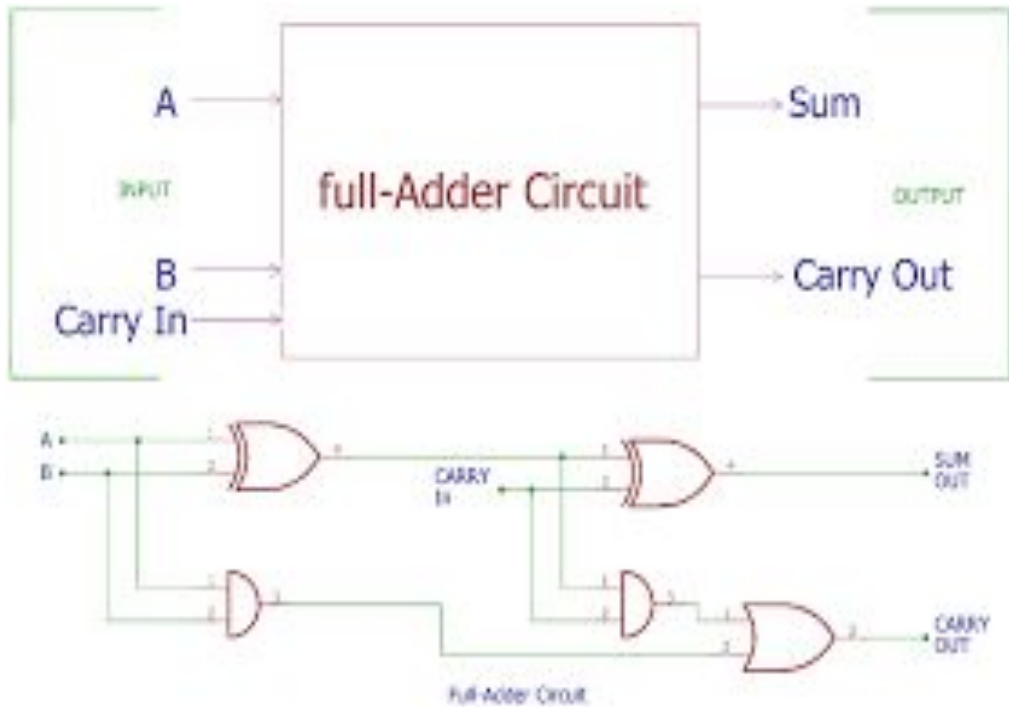
# Structural Modelling

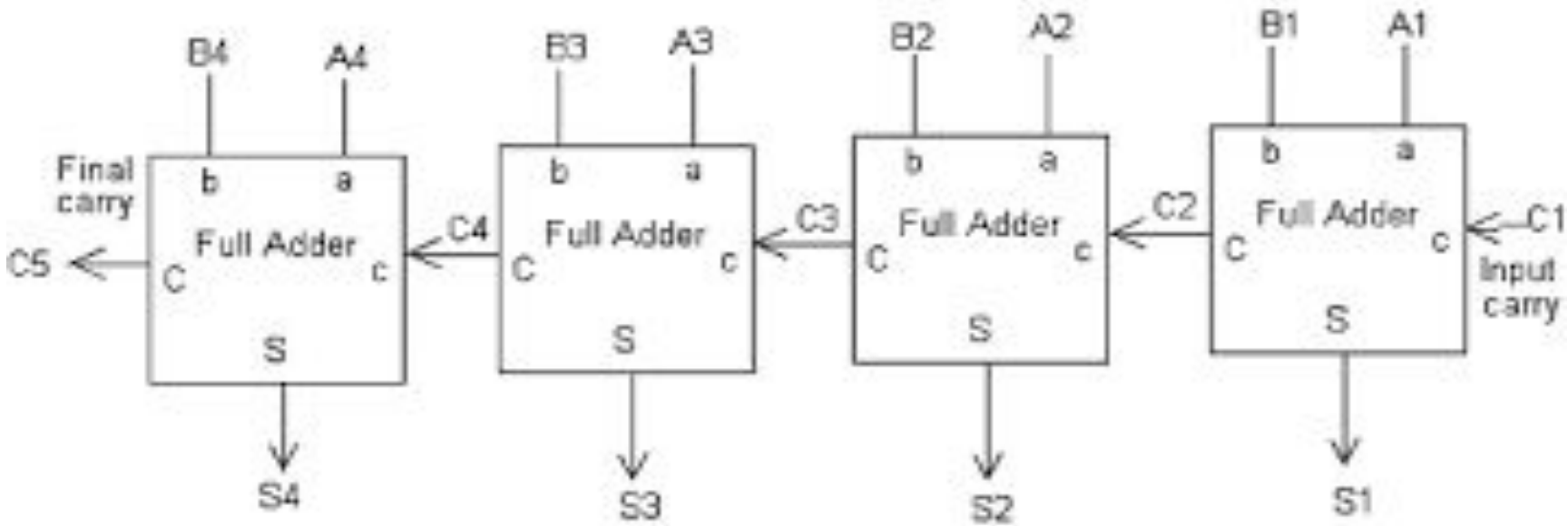
- Structural Modelling at gate level

The Half adder instantiates two gate primitives

The Full adder instantiates two half modules and one gate primitive

The Four bit adder is constructed by four full adders instances


A
B
Carry In
INPUT
full-Adder Circuit
Sum
Carry Out
OUTPUT
A
B
CARRY IN
SUM OUT
CARRY OUT
Full-Adder Circuit


B4
A4
B3
A3
B2
A2
B1
A1
Final carry
C5
b
a
Full Adder
C
c
S
C4
C3
C2
C1
Input carry
S4
S3
S2
S1

# Gate level description of half adder

```
module half adder(x,y,s,c);
Input x,y;
Output s,c;
//half adder body
// instantiate primitive gates
 xor(s,x,y);
and(c,x,y);
endmodule
```

# Gate level description of Full adder

```
module full adder(x,y,cin,s,cout);
input x,y ,cin;
output s,cout;
Wire s1,c1,c2;
//full adder body
//instantiate the half adder
half _adder ha_1(x,y,s1,c1);
half_adder ha_2(cin,s1,s,c2);
Or(cout,c1,c2);
endmodule
```

# Gate level description of 4 bit adder

```
module 4_bit_adder(x,y,c_in,sum,c_out);
Input [3:0] x,y;
Input c_in;
output [3:0] sum;
output c_out;
Wire c1,c2,c3;
//4_bit adder body
// instantiate the full adder
```

```
full_adder fa_1(x[0],y[0],c_in,sum[0],c1);
full_adder fa_2 (x[1],y[1],c1,sum[1],c2);
full _adder fa_3(x[2],y[2],c2,sum[2],c3);
full _adder fa_4(x[3],y[3],c3,sum[3],c_out);
endmodule
```

# Dataflow Modelling

- Continuous assignment
- Keyword used”assign”

syntax

assign[delay] 1_value=expression;

✔ Expression is evaluated and result is assigned to 1_value after the specified delay

✔ Default –zero delay

```
module full_adder_dataflow(x,y,c_in,sum,c_out);
//  I/O port declarations
Input x,y,c_in;
output sum,c_out;
//specify the function of full adder

assign #5 {c_out,sum} =x+y+c_in;
endmodule
```

# Behavioral Modelling

- Two procedural constructs

 Initial

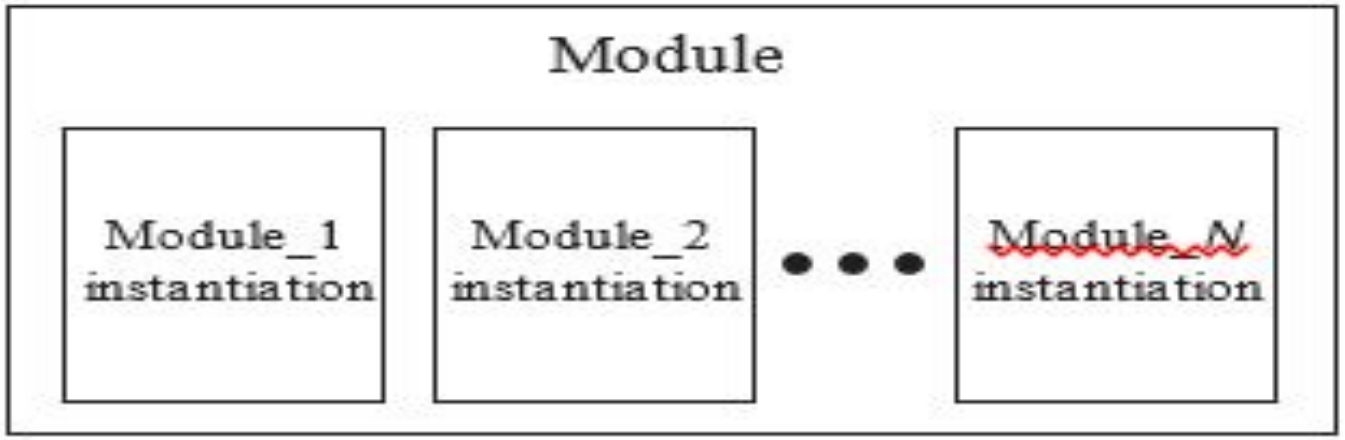
 Always

Initial  statement used to set up initial values of variable data types

Always are used to model combinational and sequential logic

```
module full_adder_behavioral(x,y,c_in,sum,c_out);
Input x,y,c_in;
output sum,c_out;
reg sum,c_out;
always @(x,y,c_in)
#5 {c-out,sum}=x+y+c_in;
endmodule
```

# Verilog program structure




Each module declaration includes a list of interface signals that can be used to connect to other modules or to the outside world.

# Module Declaration

**module** module-name (module interface list);
[list-of-interface-ports]
...
[port-declarations]
...
[functional-specification-of-module]
...
**endmodule**

The items enclosed in square brackets are optional. The list-of-interface- ports normally has the following form:

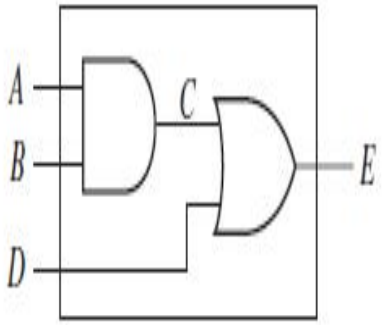
type-of-port list-of-interface-signals

{; type-of-port list-of-interface-signals};

- The curly brackets indicate zero or more repetitions of the enclosed clause. Type-of-port indicates the direction of information; whether information is flowing into the port or out of it.
- Input port signals are of keyword **input**, output port signals are of keyword **output**, and bidirectional signals are of keyword **inout**. Also, list-of- ports can be combined with the module interface list.

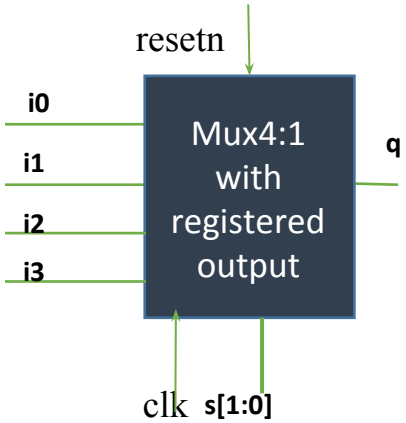
# Verilog Modules

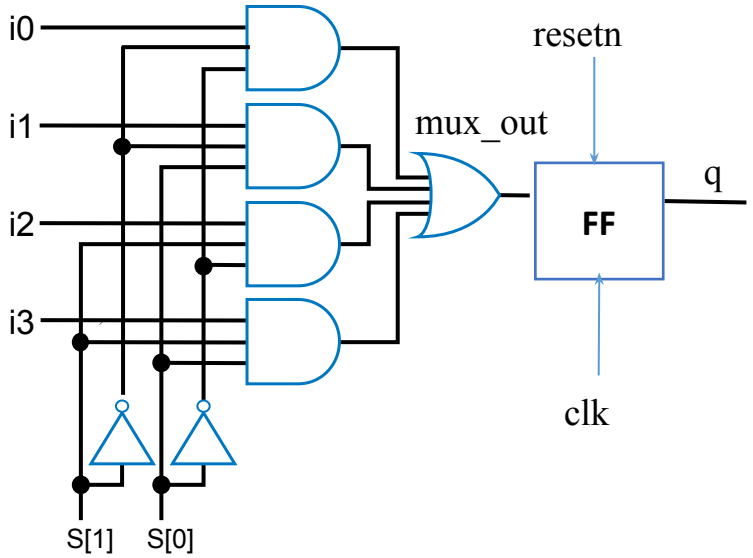
- The general structure of a Verilog code is a module description.
- A module is a basic building block that declares the input and output signals and specifies the internal operation of the module
- The **module** declara- tion has the name two_gates and specifies the inputs and outputs. *A*, *B*, and *D* are input signals, and *E* is an output signal.
- The signal *C* is declared within the module as a **wire** since it is an internal signal.
- The two concurrent statements that describe the gates are placed and the module ends with **endmodule**. All the input and output signals are listed in the module statement without specifying whether they are input or output.




```
module two_gates (A, B, D, E);
output E;

input A, B, D;

wire C;

  assign C = A && B; // concurrent
  assign E = C || D; // statements
endmodule
```

# Multiplexer Design

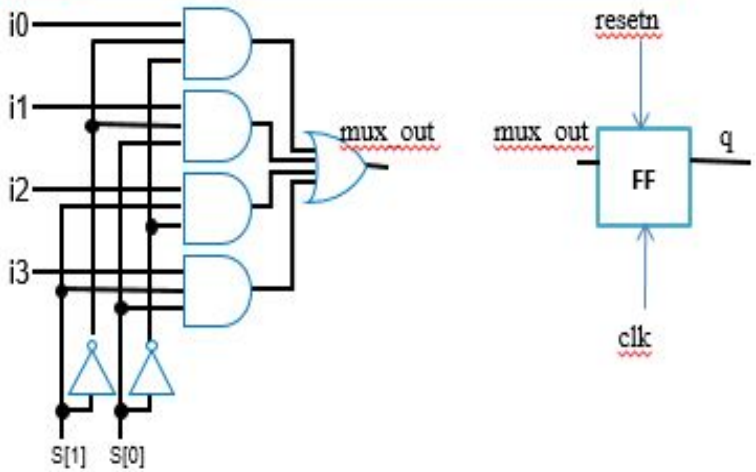



| Sel[1:0] | mux_out |
|---|---|
| 00 | i0 |
| 01 | i1 |
| 10 | i2 |
| 11 | i3 |

- A multiplexer of $2^n$ inputs has n select lines, are used to select which input line to send to the output.
- There is only one output in the multiplexer
- Simple 4: 1 Multiplexer with a registered output
- Based on the select input Sel[1:0], mux_out shall be any one of i0, i1, i2, i3 as shown in truth table above
- Multiplexer output is registered in the last stage

```
module <module Name> (ports);
    Inputs, outputs, Inout ports; parameters
    Assign statements (concurrent)
    Initial statements (procedural)
    Always statements (procedural)
endmodule
```




```
// Multiplexer in Data flow model
module mux_ex ( i0, i1, i2,i3,clk, resetn, sel,q);
 input clk, resetn;
 input  i0, i1,i2,i3;
 input [1:0] sel;
 output q;

 reg  q_out;
 wire mux_out;

// concurrent assignments
assign mux_out = sel[1] ? ( sel[0] ? i3: i2) : ( sel[0] ? i1: i0);

always @(posedge clk or resetn)
  begin
   if (!resetn)
    q_out <= 0;
   else
    q_out <= mux_out;
  end
assign q = q_out;

endmodule
```

ports

Module

Module I/Os

Wire, reg declarations

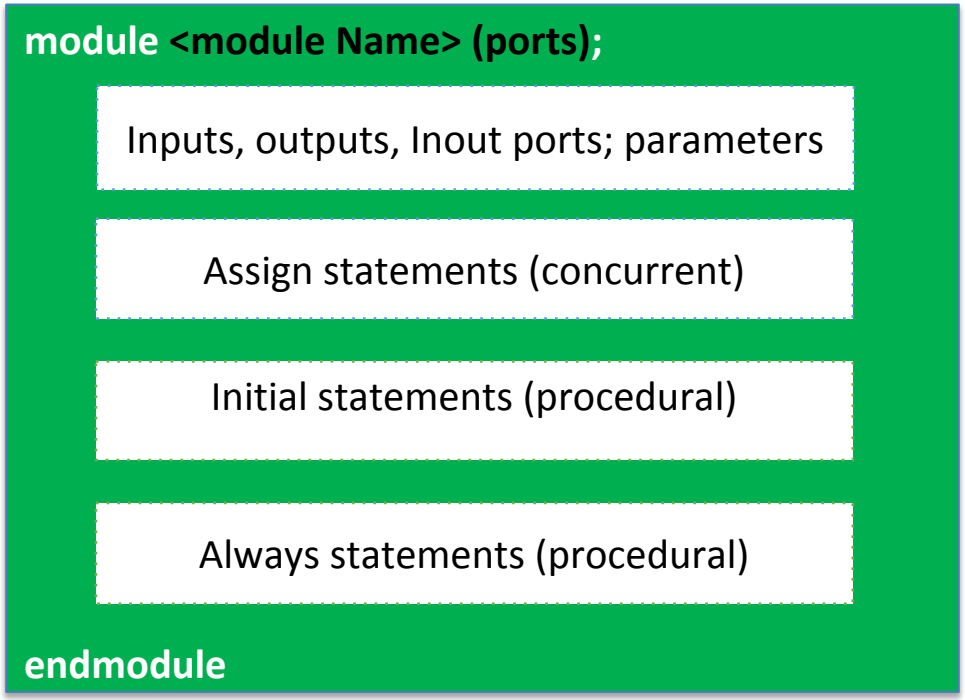
Concurrent assignments

Procedural block

Sensitivity list

Sequential statements

End module




```
// Multiplexer in Data flow model
module mux_ex ( i0, i1, i2,i3,clk, resetn, sel,q);
 input clk, resetn;
 input  i0, i1,i2,i3;
 input [1:0] sel;
 output q;

 reg  q_out;
 wire mux_out;

 // concurrent assignments
 assign mux_out = sel[1] ? ( sel[0] ? i3: i2) : ( sel[0] ? i1: i0);



always @(posedge clk or resetn)
   begin
     if (!resetn)
       q_out <= 0;
     else
       q_out <= mux_out;
   end
 assign q = q_out;

endmodule
```