

# Introduction to Verilog

Design methodology

# Need for HDL (Verilog)

Hardware Description Languages (HDLs) like **Verilog** are essential for designing, simulating, and testing digital circuits and systems at various abstraction levels.

A standard in the industry for developing complex digital circuits, such as processors, memory systems, and custom ASIC designs.

**Challenge:** Digital circuits are complex and consist of millions of gates and components.

**Need:** Verilog provides a higher level of abstraction to describe the behavior of hardware rather than manually designing circuits at the gate level. This makes the design process more manageable and less error-prone.

**Challenge:** Manually implementing a hardware design at the gate or transistor level is highly inefficient for modern applications.

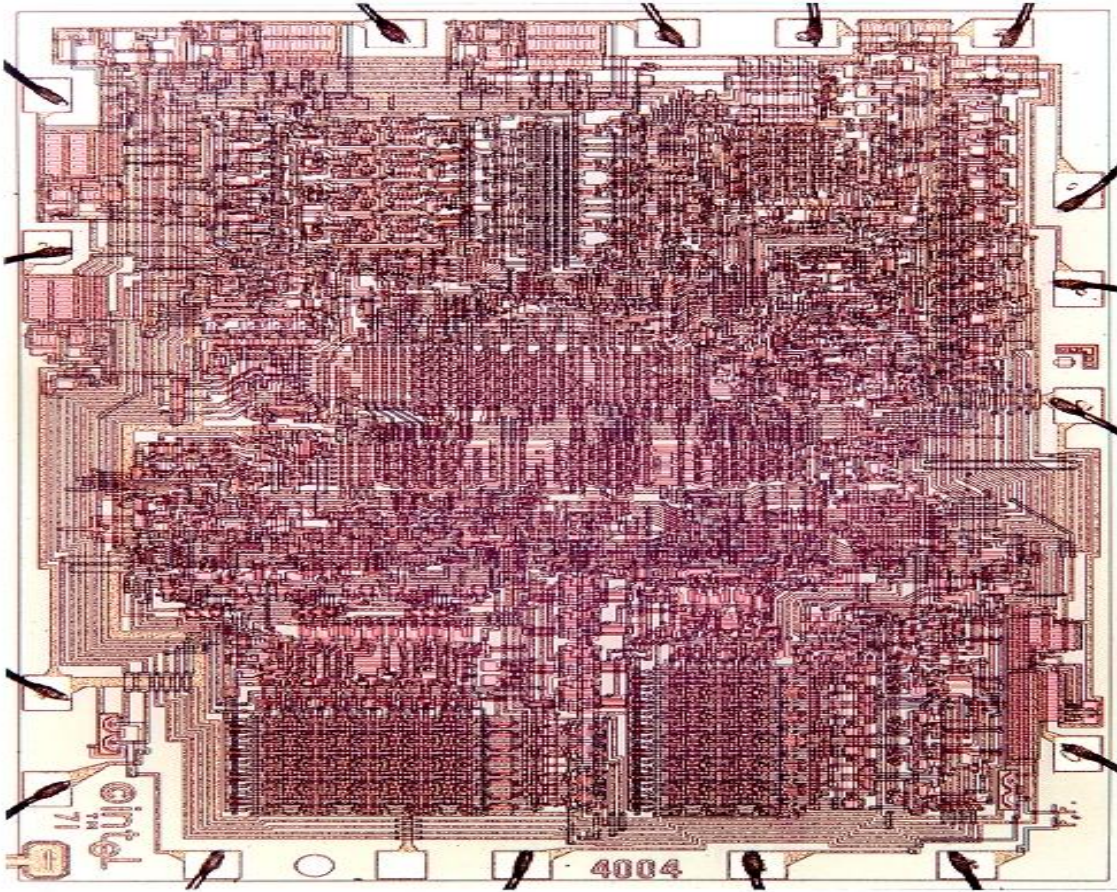
**Need:** Verilog can be synthesized into gate-level representations automatically using synthesis tools, saving time and effort. Designers focus on high-level design, while the tools handle low-level implementation..

**Challenge:** Hardware designs need to be portable across different tools and fabrication processes.

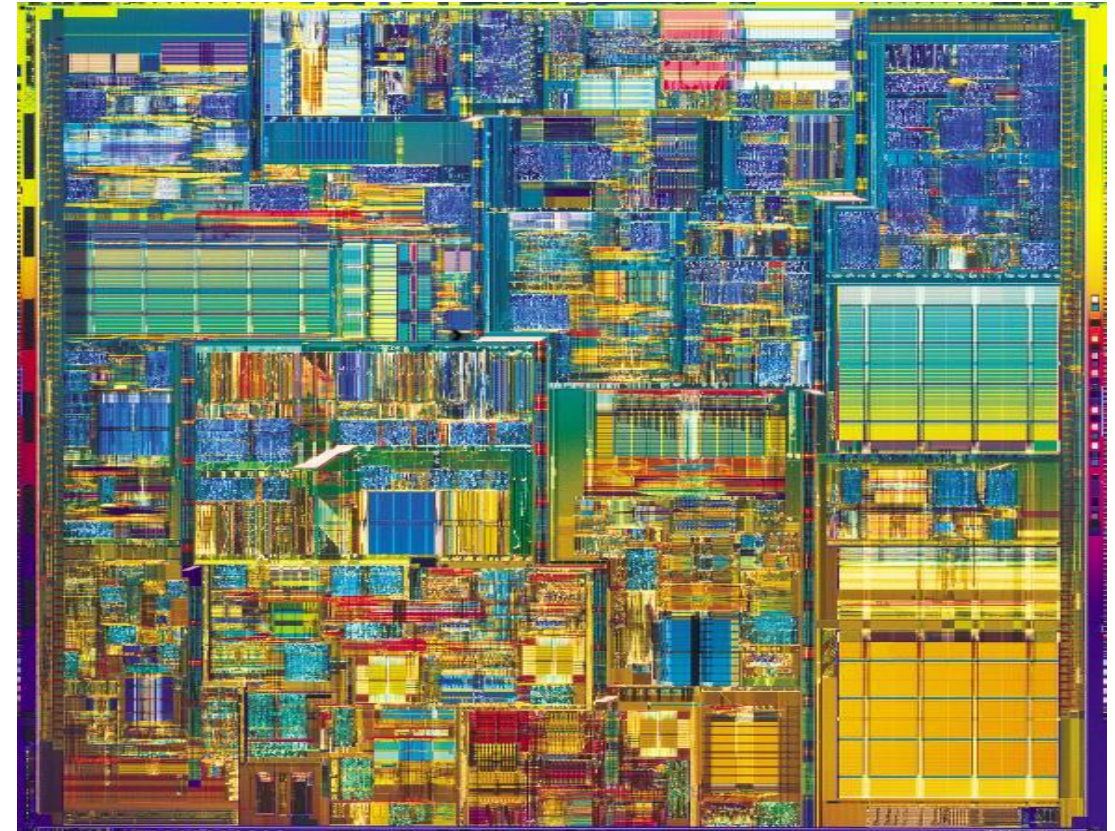
**Need:** Verilog is supported by a wide range of Electronic Design Automation (EDA) tools and can be used to target different FPGA and ASIC platforms. This standardization ensures that designs can be synthesized and verified across various environments.



# Advancements over the years



© Intel 4004 Processor Introduced in 1971 2300 Transistors 108 KHz Clock



- © Intel P4 Processor
- Introduced in 2000
- 40 Million Transistors
- 1.5GHz Clock



# What is Verilog?

- **Verilog is one of the two-major Hardware Description Languages (HDL) used by hardware designers in industry and academia. VHDL is the other one.**
- **Verilog allows a hardware designer to describe designs at a high level of abstraction such as at the architectural or behavioral level as well as the lower implementation levels (i. e., gate and switch levels) leading to Very Large Scale Integration (VLSI) Integrated Circuits (IC) layouts and chip fabrication.**
- **A primary use of HDLs is the simulation of designs before the designer must commit to fabrication.**

- Developed in 1984-85 by Philip Moorby
- In 1990 Cadence opened the language to the public
- Standardization of language by IEEE in 1995

- **Verilog is case sensitive**

- **Keywords are in lowercase**

Example: **initial, assign, module, always**

# Verilog Fundamentals

# HDL

Hardware Description Languages (HDLs) are absolutely foundational to modern digital design.

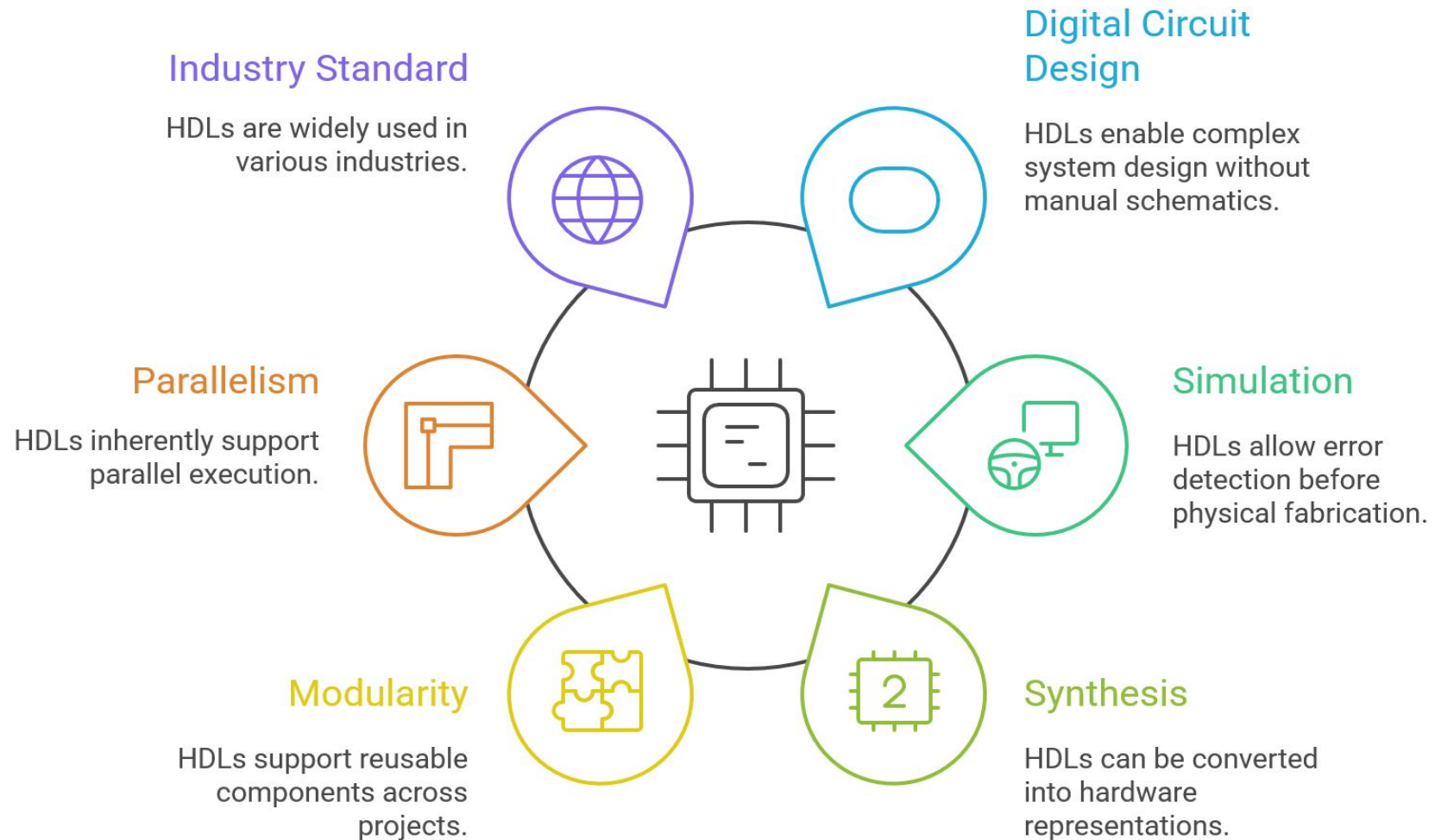
HDLs are specialized programming languages used to describe the structure, behavior, and timing of electronic circuits.

The two most widely used HDLs are

- 1.VHDL (VHSIC Hardware Description Language)**
- 2. Verilog**



## Advantages of Hardware Description Languages



# Significance of HDL

*Go, change the world<sup>®</sup>*

- **Digital Circuit Design**

HDLs allow engineers to design complex digital systems without manually drawing schematics.

- **Simulation Before Fabrication**

Before a chip is physically built, HDLs let designers simulate its behavior to catch errors early. This saves time, money, and headaches.

- **Synthesis to Hardware**

HDL code can be synthesized into gate-level representations, which are then used to fabricate actual hardware (like FPGAs or ASICs).

- **Modularity and Reusability**

Just like software, HDL designs can be modular. You can reuse components like adders, multiplexers, or memory blocks across multiple projects.

- **Parallelism**

Unlike traditional programming languages, HDLs inherently support parallel execution, which mirrors how hardware operates in real time.

- **Industry Standard**

HDLs are the backbone of chip design in industries ranging from consumer electronics to aerospace and defense.

# Verilog HDL

*Go, change the world<sup>®</sup>*

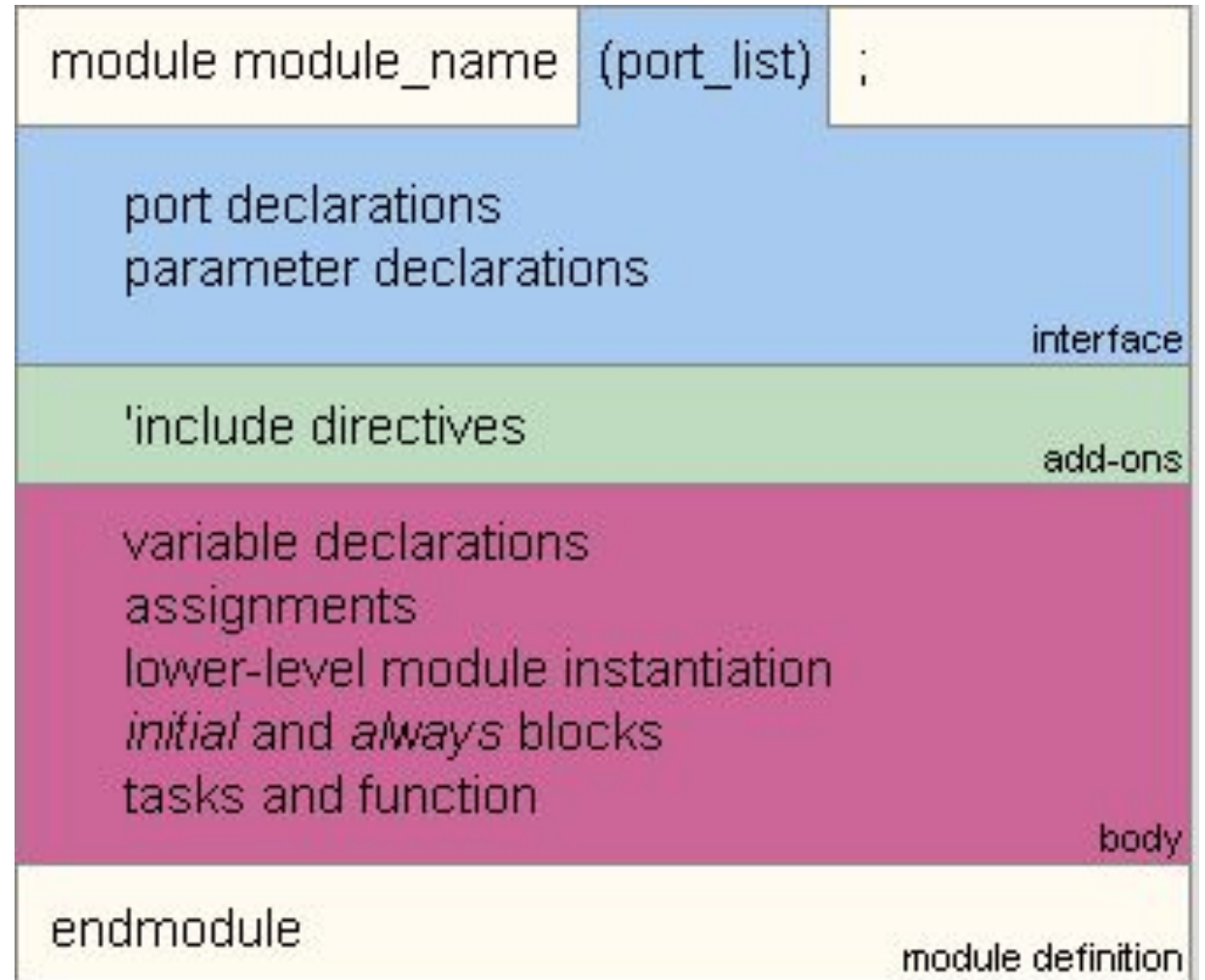
- Verilog HDL has a syntax that describes precisely the legal constructs that can be used in the language.
  - Syntax is pre-defined, lowercase, identifiers that define the language constructs.
  - Example : *module, endmodule, input, output wire, and, or, not* , etc.,
  - Any text between two slashes (//) and the end of line is interpreted as a comment.
  - Blank spaces are ignored and names are case sensitive.
- Comments
    - // The rest of the line is a comment
    - /\* Multiple line comment \*/
    - /\* Nesting /\* comments \*/ do **NOT** work \*/

# Definition of Module

**Interface:** port and parameter declaration

**Body:** Internal part of module

Add-ons (optional)



# Some points to remember

The name of Module

Comments in Verilog

One line comment (`// .....`)

Block Comment (`/* ..... */`)

Description of Module (optional but suggested)



# Description of Module

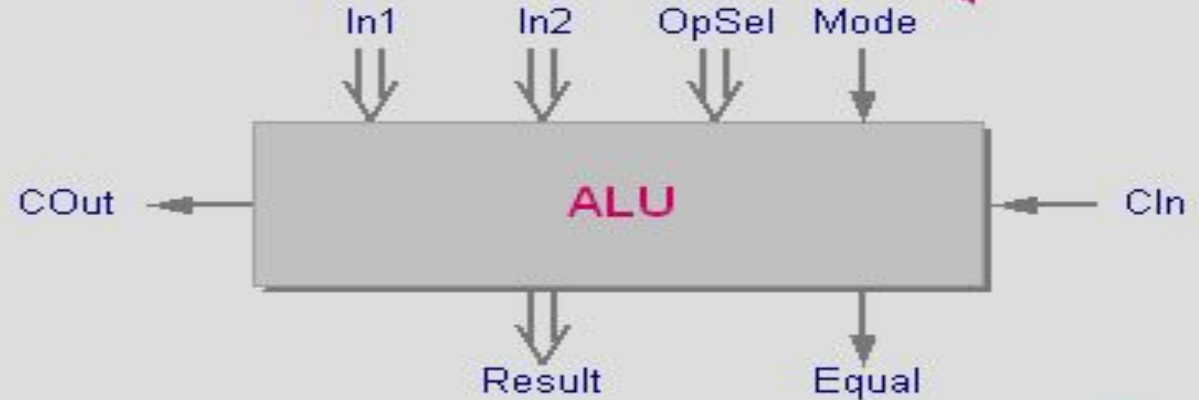
*Place the cursor on any description to see its purpose.*

```
// Design      : 8086 (RTL)
// File name   : 8086.v
// Purpose     : model of a 8086 microprocessor for the
//              design of "system-on-chip" embedded
//              modules. Fully compliant with the
//              specification by Intel.
//
// Note        : This model can be synthesized with
//              Active-CAD tools.
//
// Limitations  : A Clk frequency of 33 MHz is assumed.
//
// Errors       : None known.
//
// Include files : none.
//
// Author       : Evita Team
//              ALDEC Inc.
//              2230 Corporate Circle,
//              Henderson, Nevada 89014
//
// Simulator    : Active-CAD
// -----
// Revision list
// Version  Author   Date      Changes
// 1.0      ET       01 Jan 98   new version
```

# The Module Interface

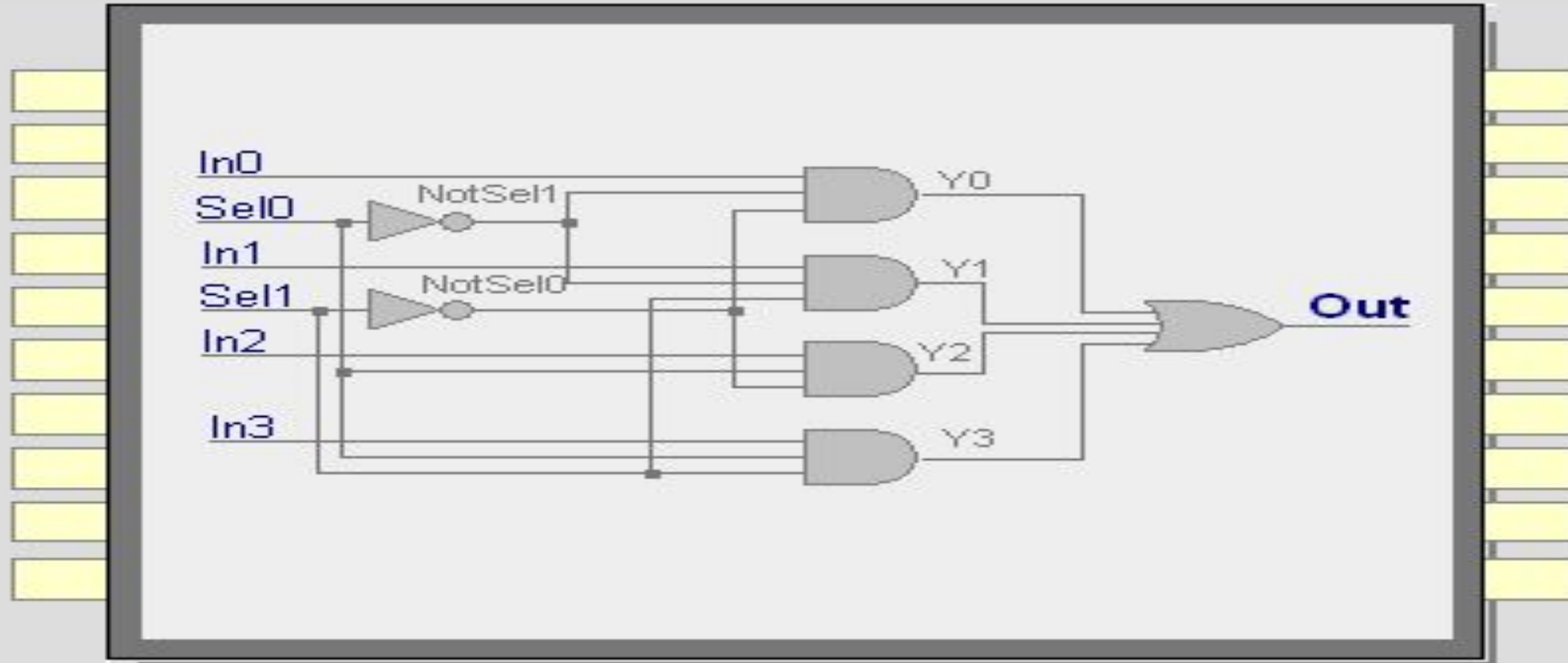
## Port List

## Port Declaration



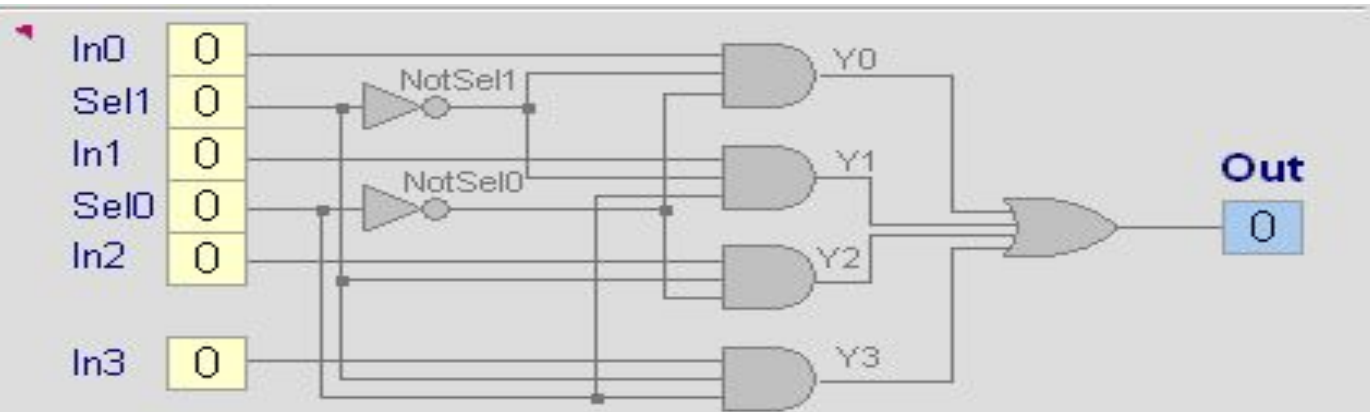
```
module ALU (Result, COut, Equal, In1, In2,  
             OpSel, CIn, Mode);  
  
  output [3:0] Result;    // operation result  
  output      COut;      // carry out  
  output      Equal;     // when 1, In1 = In2  
  input  [3:0] In1;      // first operand  
  input  [3:0] In2;      // second operand  
  input  [3:0] OpSel;     // operation select  
  input      CIn;        // carry in  
  input      Mode;       // mode arithm/logic;  
                        // arithm when 0  
  
  . . .  
  
endmodule
```

# One language, Many Coding Style



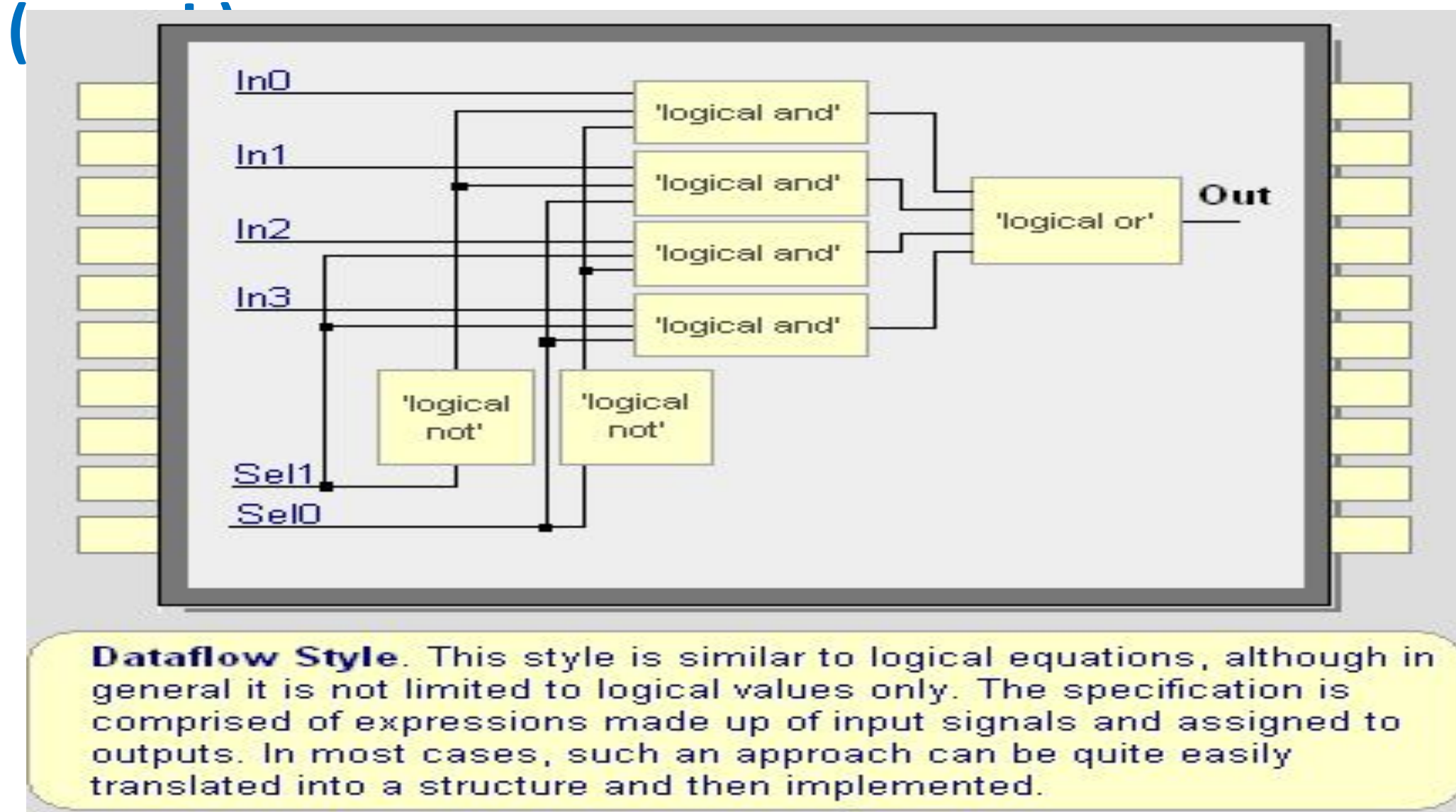
**Structural Style.** The circuit is specified in terms of instantiations of lower level components (in this case logic gates, which are Verilog primitives) connected with internal signals. The translation of such a specification into a physical circuit is straightforward.

# Structural style: Verilog Code



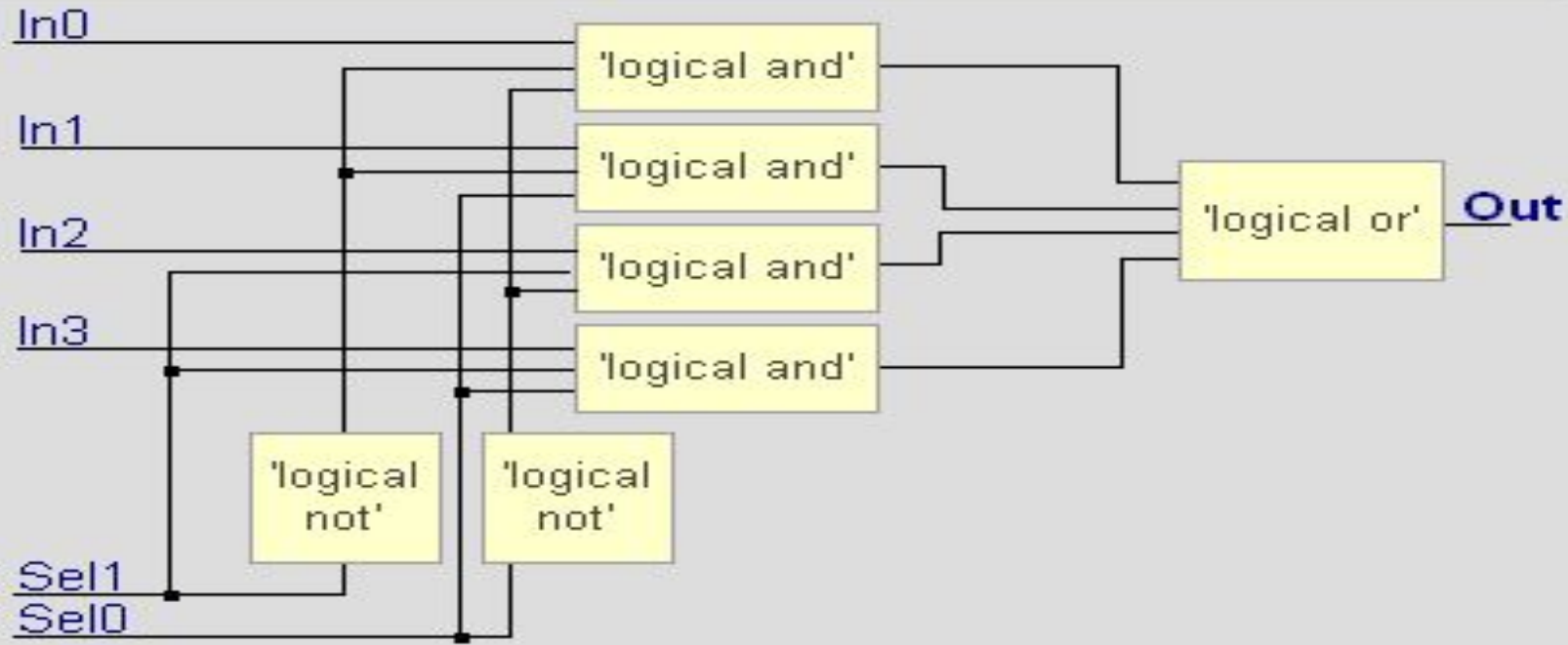
```
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);  
output Out;  
input In0, In1, In2, In3, Sel0, Sel1;  
  
wire NotSel0, NotSel1;  
wire Y0, Y1, Y2, Y3;  
  
not (NotSel0, Sel0);  
not (NotSel1, Sel1);  
and (Y0, In0, NotSel1, NotSel0);  
and (Y1, In1, NotSel1, Sel0);  
and (Y2, In2, Sel1, NotSel0);  
and (Y3, In3, Sel1, Sel0);  
or (Out, Y0, Y1, Y2, Y3);  
  
endmodule
```

# One language, Many Coding Style



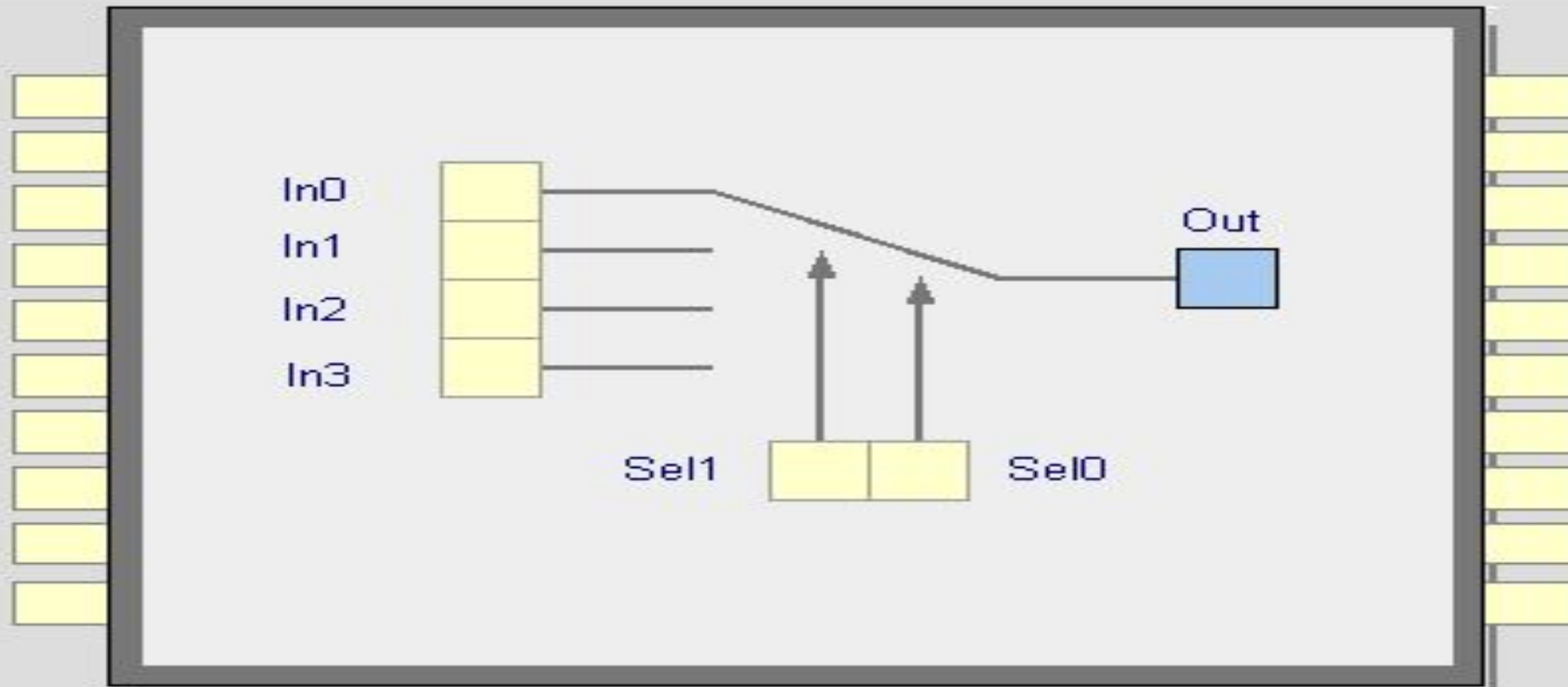


# Dataflow style: Verilog Code



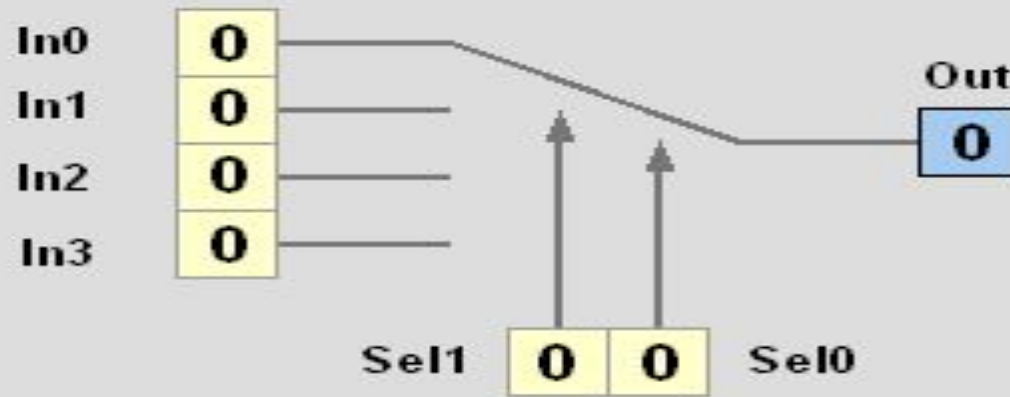
```
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);  
  
    output Out;  
    input In0, In1, In2, In3, Sel0, Sel1;  
  
    assign Out = (~Sel1 & ~Sel0 & In0) | (~Sel1 & Sel0 & In1)  
                | (Sel1 & ~Sel0 & In2) | (Sel1 & Sel0 & In3);  
  
endmodule
```

# One language, Many Coding Style (contd.)



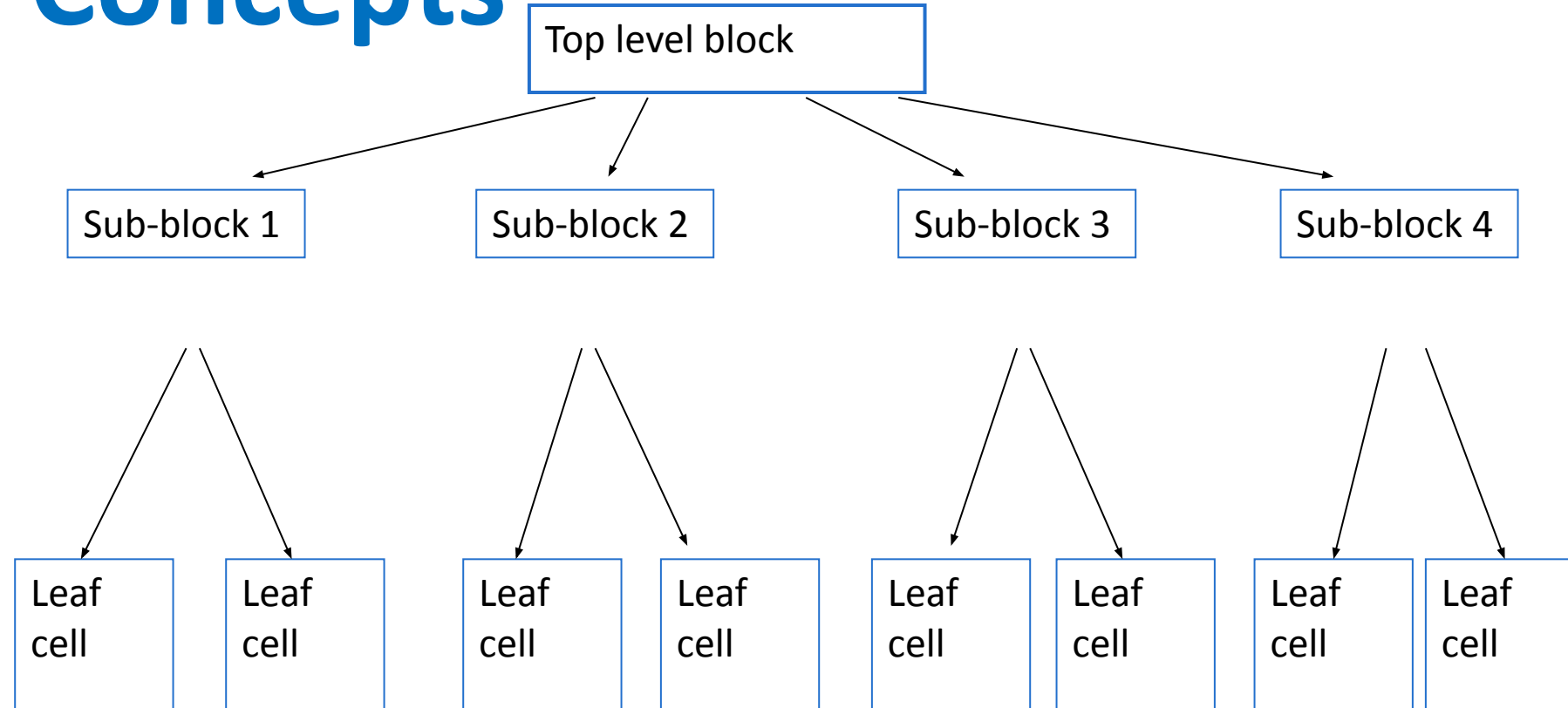
**Behavioral Style.** It specifies the circuit in terms of its expected behavior. It is the closest to a natural language description of the circuit functionality, but also the most difficult to synthesize.

# Behavioral style: Verilog Code

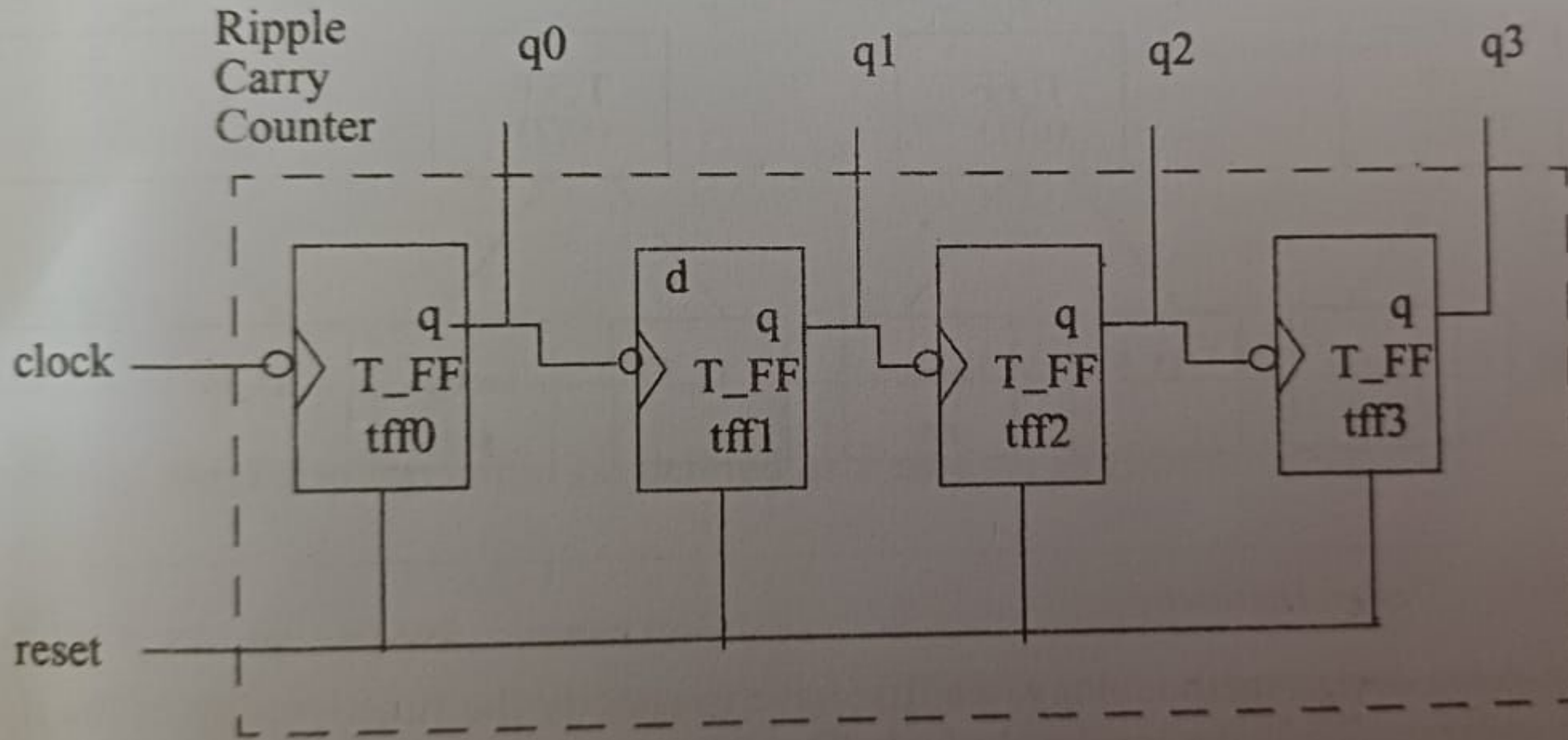


```
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);  
  output Out;  
  input In0, In1, In2, In3, Sel0, Sel1;  
  reg Out;  
  
  always @(Sel1 or Sel0 or In0 or In1 or In2 or In3)  
  begin  
    case ({Sel1, Sel0})  
      2'b00 : Out = In0;  
      2'b01 : Out = In1;  
      2'b10 : Out = In2;  
      2'b11 : Out = In3;  
      default : Out = 1'bx;  
    endcase  
  end  
  
endmodule
```

# Hierarchical Modeling Concepts



## 2.2 4-bit Ripple Carry Counter





# Hierarchical Modeling Concepts

```
Module ripple_carry_counter(q,clk, reset);
```

```
Output [3:0] q;
```

```
Input clk, reset;
```

```
T_FF tff0(q[0], clk, reset);
```

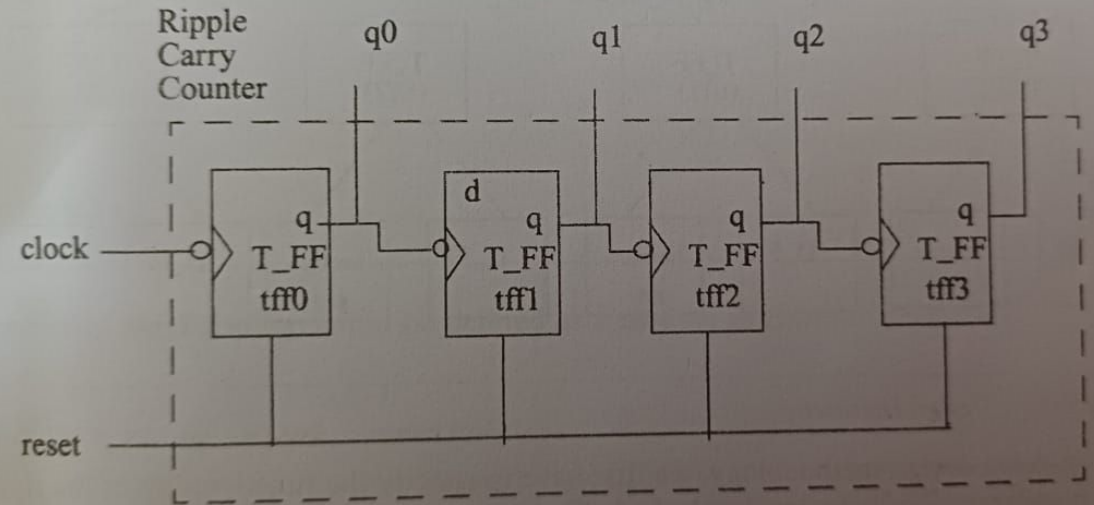
```
T_FF tff1(q[1], q[0], reset);
```

```
T_FF tff2(q[2], q[1], reset);
```

```
T_FF tff3(q[3], q[2], reset);
```

```
endmodule
```

## 2.2 4-bit Ripple Carry Counter



# Hierarchical Modeling

## Concepts

```
module T_FF(q, clk, reset);
```

```
output q;
```

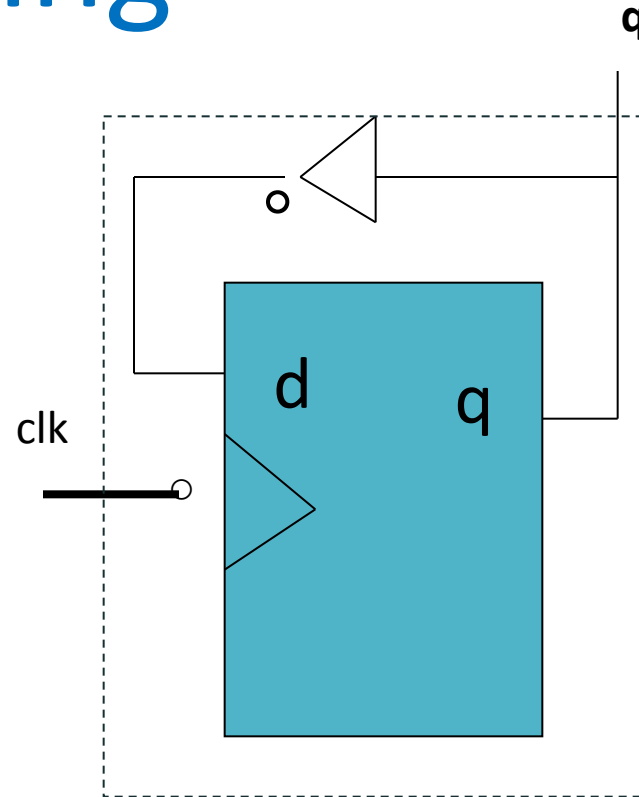
```
input clk, reset;
```

```
wire d;
```

```
D_FF dff0(q, d, clk, reset);
```

```
not na(d, q);
```

```
endmodule
```

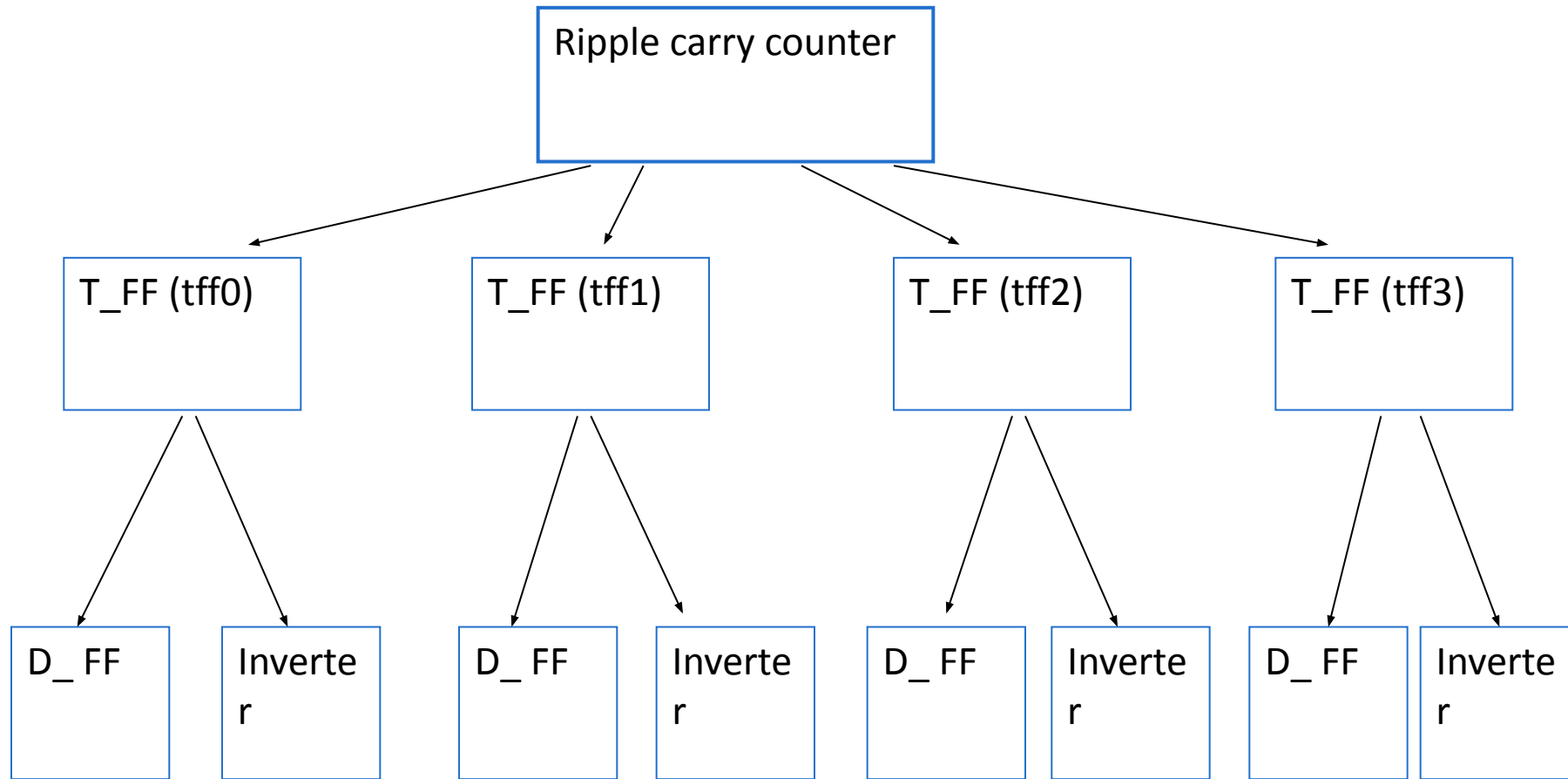


# Hierarchical Modeling

## Concepts

```
module D_FF(q, d, clk, reset);  
    output q;  
    input d, clk, reset;  
    reg q;  
    always @(posedge reset or negedge clk)  
        if (reset)  
            q=1'b0;  
        else  
            q=d;  
    endmodule
```

# 4-bits Ripple Carry Counter



# Verilog- Basic Concepts

## Number Specifcaton:

### Sized :

syntax: ‘

Size --> no. of bits in the number; only decimal

Base format --> for decimal : ‘d or ‘D

Hex : ‘h or ‘H

Binary : ‘b or ‘B

Octal : ‘o or ‘O

Eg:

4'b1011 //4 bit binary no.

12'hafb //12 bit hex no.

16'd255 //16 bit decimal no.

Number -> 0 to 9(if deci)

0 to F(if hex)

0 to 8(if oct)



## Unsize Numbers:

Numbers that are specified without are decimal numbers by default.

Eg:

23456 //32 bit decimal no. by default

'hc3 //32 bit hex no. by default

'o21 //32 bit oct no. by default

## X or Z values:

X ---> denotes unknown value

Z ---> denotes a high impedance value

Eg:

12'h13X	//12 bit hex no.;4 LSBs are unknown
6'hx	//6 bit hex no.;all 6 bits are unknown
32'bz	//32 bit high impedance number

## Identifiers and Keywords:

- Special reserved identifiers
- Are in lowercase

Eg:

`reg value;`      *// 'reg' is keyword; 'value' is an identifier*

`input clk;`      *// 'input' is a keyword; 'clk' is an identifier*

## Strings:

- Should be enclosed by double quotes
- Must be in a single line without carriage return
- Treated as a sequence of one byte ASCII values

"This is an example" //is a string

# Data Types in Verilog

- A storage format having a specific range or type is called data type.

## Logic values

1

## Description

Logic one, true condition

0

Logic zero, false condition

x

Unknown value

z

High impedance

# Nets

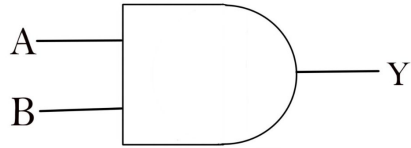
- Nets are used to connect between hardware entities like logic gates and hence do not store any value.
- The net variables represent the physical connection between structural entities such as logic gates.
- These variables do not store values.
- Nets are declared with keyword **wire**



**Note:**

- 1.The net and wire terms are interchangeably used.
- 2.Usually, the default value of the net is z.

*3.1 bit value by default, unless it is declared as vector explicitly.*



```
wire a; //one-bit value as a single net.  
wire [5:0] a; //net as a vector
```

**Examples:**

```
wire Y;    //Declare net Y for above ckt
```

```
wire A, B; //Declare 2 wires B,,C for above ckt
```

```
wire d=1'b0; // Net d is fixed to logic 0 at declaration
```

# Registers

- The registers represent data storage elements.
- It retains value till it is overridden.
- Keyword used to declare a register data type is *reg*.

```
reg a; // single bit register  
reg [5:0] a; // 6 bit register as a vector
```

## Examples:

```
reg reset;  
initial  
begin  
    reset=1'b0;  
#100 reset=1'b1;  
end
```

# Vectors

- The nets or registers can be declared as vectors to represent multiple bit widths.
- If bit width is not specified, it is a scalar.
- Vectors can be declared at [high #: low#] or [low #: high#], left number is always the MSB.

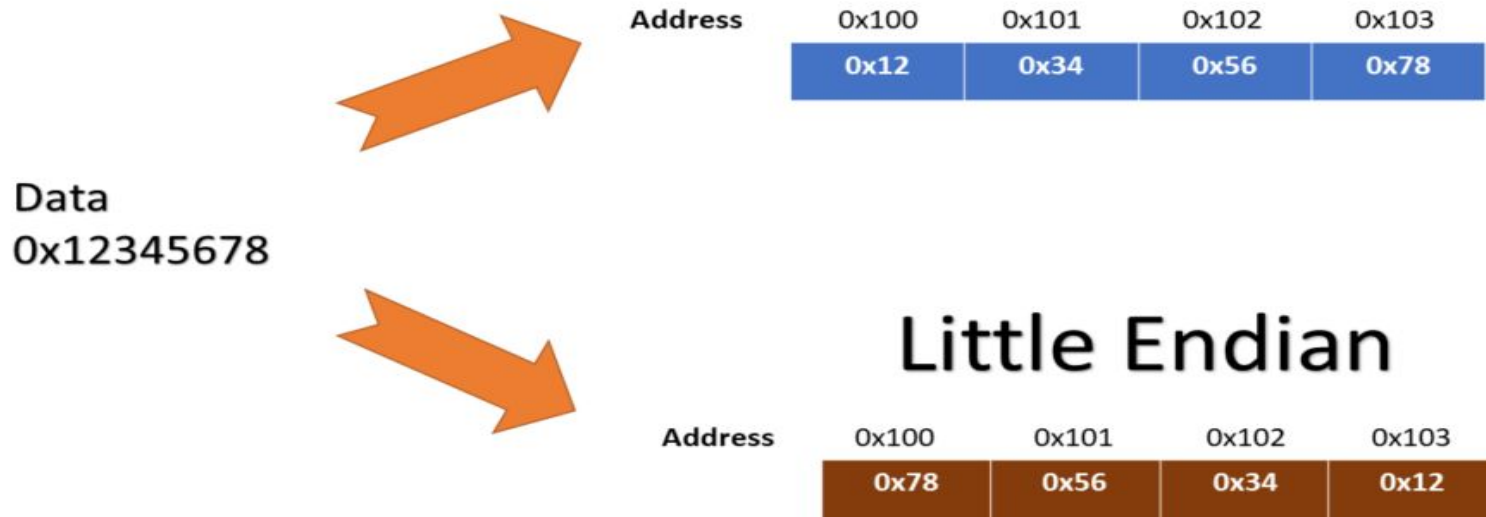
```
wire [5:0] a;  
reg [5:0] a;
```

## Examples:

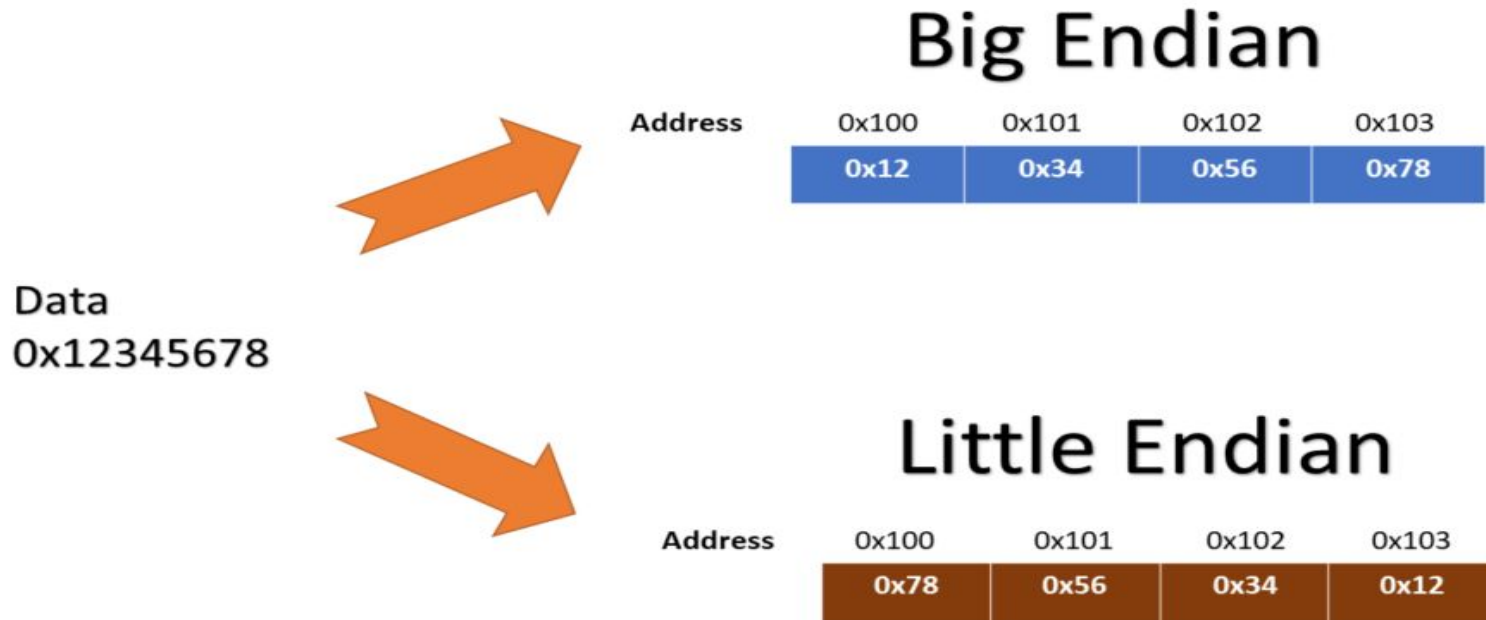
```
wire [7:0] bus; // 8 bit bus  
wire [31:0] A,B,C; // 3 buses of 32 bit width  
reg [0:40] addr; // vector register 41 bits wide
```

## Examples:

```
reg [15:0]; data1 // little endian notation  
reg [0:15]; data2 // big endian notation
```



Endianness is about *byte address order*. Little endian means the lower significant bytes get the lower addresses. Big endian means, the most significant bit of a word is stored in the byte with the lower address.



# Vector part selection

## Examples:

```
reg[7:0] bus; // 8 bit bus
reg[31:0] A,B,C; // 3 buses of 32 bit width, little Endian notation
reg [0:40] addr; // vector register 41 bits wide, big endian
notation
```

```
//Vector part selection
A[7] //bit # 7 of vector A
bus[2:0] // three LSB bits of vector bus
```

**bus[0:2] ———what is the result???**

**Two MSB bits of vector addr??**

# Vector part selection

## Examples:

```
wire [7:0] bus; // 8 bit bus
wire [31:0] A,B,C; // 3 buses of 32 bit width
reg [0:40] addr; // vector register 41 bits wide
```

```
//Vector part selection
A[7] //bit # 7 of vector A
bus[2:0] // three LSB bits of vector bus
```

**bus[0:2] ———is it valid???**

**Two MSB bits of vector addr??**

```
Addr[0:1] // Two MSB bits of vector addr
```



```

module jdoodle;
  reg [7:0]bus;
  initial begin
    bus=15;
    //3 LSB bits of bus
    $display ("3 LSB bits:%b",bus[2:0]);
    $finish;
  end
endmodule

```

```

3 LSB bits:111
jdoodle.v:17: $finish called at 0 (1s)

```

```

module jdoodle;
  reg [7:0]bus;
  initial begin
    bus=15;
    //3 LSB bits of bus
    $display ("3 LSB bits:%b",bus[2:0]);
    $display ("%b",bus[0:2]);
    $finish;
  end
endmodule

```

```

jdoodle.v:17: error: part select bus[0:2] is out of order.
1 error(s) during elaboration.

```

```

module jdoodle;
  reg [3:0]data;
  initial begin

    $display ("Welcome to JDoodle!!!");
    data=4'b1011;
    $display ("%b",data[3:1]);
    $finish;
  end
endmodule


```

Output    Generated Files

```

Welcome to JDoodle!!!
101
jdoodle.v:18: $finish called at 0 (1s)
|

```

 Compiled and executed in 0.776 sec(s)

```

module jdoodle;
  reg [0:3]data;
  initial begin

    $display ("Welcome to JDoodle!!!");
    data=4'b1011;
    $display ("%b",data[1:3]);
    $finish;
  end
endmodule

```

Output    Generated Files

```

Welcome to JDoodle!!!
011
jdoodle.v:18: $finish called at 0 (1s)
|

```

# Variable Vector part selection

## Syntax:

`[<starting bit>+:width]` // part select increments from starting bits

`[<starting bit>-:width]` // part select decrements from starting bits

## Examples:

```
reg [255:0]; data1 // little endian notation
reg [0:255]; data2 // big endian notation
reg [7:0] byte;
```

//Variable part selection

```
byte= data1[31 -:8]; // ⇒ data1[31:24]
byte=data1[24+:8]; // ⇐ data1[31:24]
```

```
byte=data2[31 -:8]; // ????
```

```
byte=data2[24+:8]; // ?????
```

# Variable Vector part selection

## Syntax:

`[<starting bit>+:width]` // part select increments from starting bits

`[<starting bit>-:width]` // part select decrements from starting bits

## Examples:

```
reg [255:0]; data1 // little endian notation
reg [0:255]; data2 // big endian notation
reg [7:0] byte;
```

//Variable part selection

`byte= data1[31 -:8];` //  $\Rightarrow$  `data1[31:24]`

`byte=data1[24+:8];` //  $\Rightarrow$  `data1[31:24]`

`byte=data2[31 -:8];` //  $\Rightarrow$  `data2[24:31]`

`byte=data2[24+:8];` //  $\Rightarrow$  `data2[24:31]`

# Integer, Real, Time data types

## Integer:

It's a general purpose register used for manipulating quantities.

Its 32 bits.

Keyword : *integer*

Difference between **reg** and **integer** type??

Reg is unsigned where as integer is signed.

## Example:

```
integer count;  
Count=-1;
```

# Real

Real number constants real registers are declared as real.

Can be specified as decimal(3.142) or scientific notation. (2e6)

Default value is 0;

Keyword : *real*

**Difference between *reg* and *integer* type??**

Reg is unsigned where as integer is signed.

**Example:**

```
real delta;
```

```
delta=4e10;
```

```
delta=2.134;
```

# Time

Used to store the simulation time.

Its 64 bits.

Keyword : *time*

**\$time** is used to get the current simulation time.

**Example:**

```
time sim_time;  
Sim_time=$time;
```



# Arrays

Arrays are allowed with reg, real, integer, time.  
Multidimensional arrays can be declared.

## Example:

```
integer count[7:0]; // integer array  
reg bool[31:0]; // 32 one bit Boolean array  
time chk_point[1:100]; // array of 100 check points  
integer matrix[4:0][0:255]; // 2 dimensional
```

# Parameter

- Verilog allows constants to be declared as parameter.
- This allows module instances to be customized.
- It's a way to define constants that can be modified when the module is instantiated without changing the original design.
- values that can be adjusted for different design needs.

Keyword is *parameter*

## Example:

```
parameter port_id=5;  
parameter cache_line_width=256;  
parameter signed[15:0] WIDTH;
```

# System tasks

Verilog provides standard system tasks for certain routine operations.

Appear in the form \$<keyword>

## Example:

```
$display  
$time  
$monitor  
$stop // to stop the simulation  
$finish // terminates the simulation
```

Format specifiers	Description
%d or %D	To display variables in decimal
%b or %B	To display variables in binary
%h or %H	To display variables in hexadecimal
%o or %O	To display variables in octal
%c or %C	To display ASCII character
%s or %S	To display string
%t or %T	To display the current time
%f or %F	To display real numbers in decimal format. (Ex. 3.14)
%e or %E	To display real numbers in scientific format. (Ex. 2e20)

```
module jdoodle;  
  reg [0:3]data;  
  initial begin  
    data=15;  
    $display ("Data is (in Binary):%b",data);  
    $display ("Data is (in Hexadecimal):%h",data);  
    $display ("Data is (in Octal):%o",data);  
    $finish;  
  end  
endmodule
```

```
Data is (in Binary):1111  
Data is (in Hexadecimal):f  
Data is (in Octal):17  
jdoodle.v:18: $finish called at 0 (1s)
```

```
module jdoodle;
  reg [0:3]data;
  initial begin

    $display ("Welcome to JDoodle!!!");
    $display ("%t", $time);
    #100
    $display ("%t", $time);
    $finish;
  end
endmodule
```

```
Welcome to JDoodle!!!
           0
          100
jdoodle.v:22: $finish called at 100 (1s)
```

# Modules & Ports

In Verilog, modules and ports are fundamental concepts that **enable the communication of data and signals between different parts of a digital circuit.**

## Modules:

Self-contained blocks that **represent hardware components or functions.** Modules can be nested, and they promote code reusability and modular design.

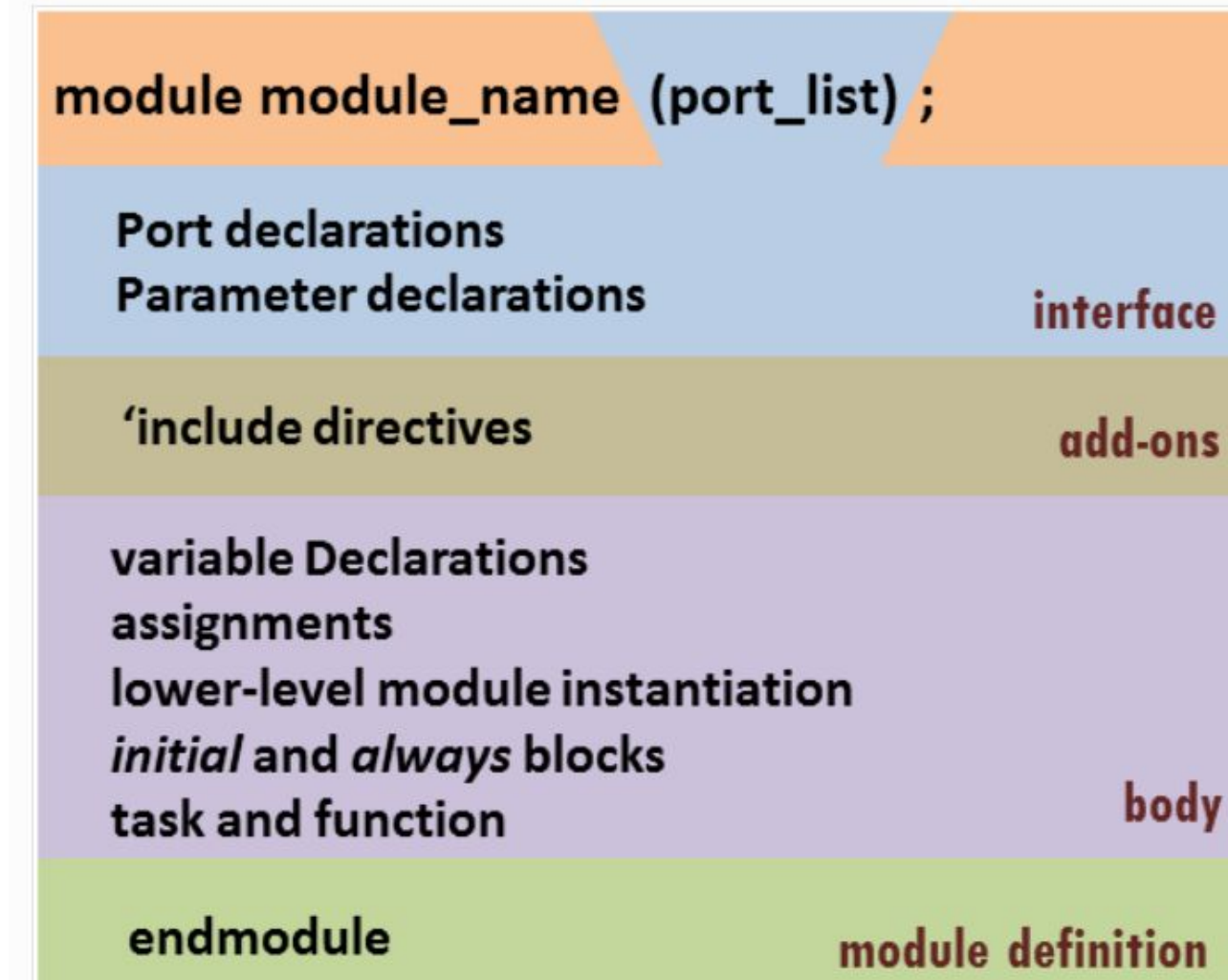
## Ports:

Define the **interface of a module, specifying how data and signals can enter or exit it.** There are three types of ports:

- **Input ports:** Bring data or signals into a module
- **Output ports:** Send data or signals out of a module
- **Bidirectional(inout) ports:** Allow data to flow both into and out of a module



# Components of Verilog Module



# Verilog Ports

- Port is an essential component of the Verilog module. Ports are used to communicate for a module with the external world through input and output.
- It communicates with the chip through its pins because of a module as a fabricated chip placed on a PCB.
- Every port in the port list must be declared as *input*, *output* or *inout*. All ports declared as one of them is assumed to be wire by default to declare it, or else it is necessary to declare it again.

```
module fulladd4(sum, c_out, a, b, c_in);
```

```
//Begin port declarations section
```

```
output [3 : 0] sum;
```

```
output c_out;
```

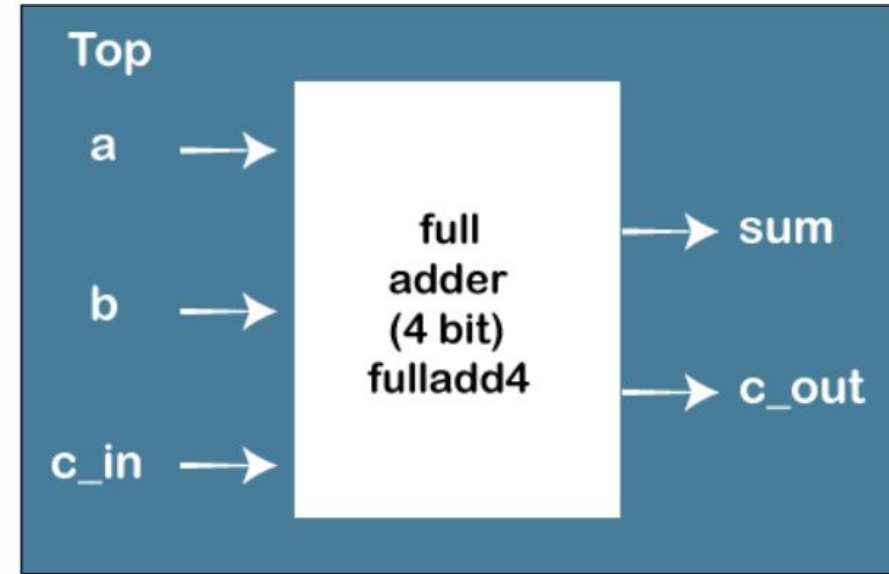
```
input [3:0] a, b;
```

```
input c_in;
```

```
//End port declarations section
```

```
<module internals>
```

```
endmodule
```



I/O Ports for Top and Full Adder



NOTE: By convention, outputs of the module are always first in the port list. This convention is also used in the predefined modules in Verilog.



NOTE: Ports of the type **input** and **inout** cannot be declared as reg.



# Components of Verilog Test Bench