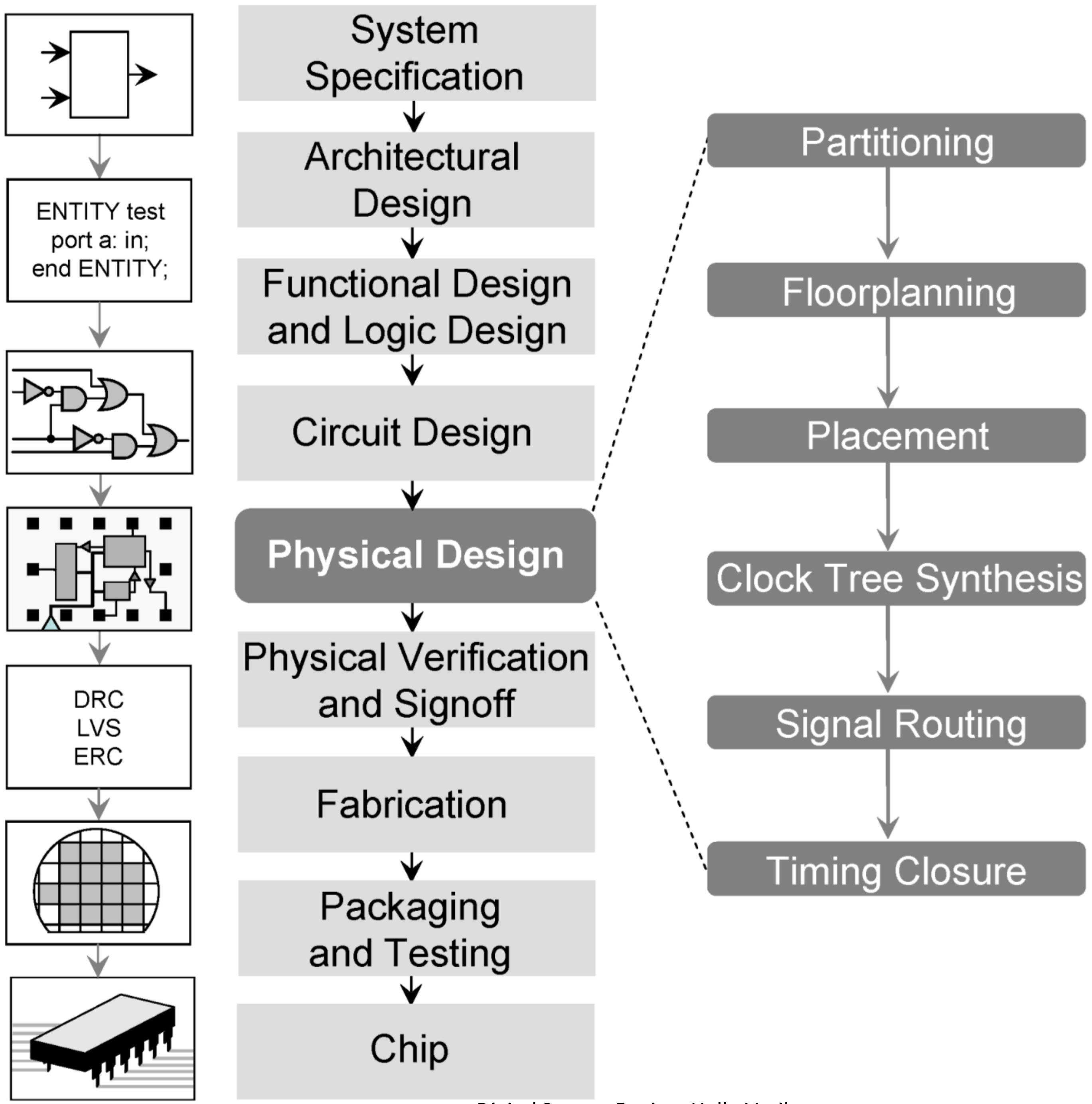
Digital System Design

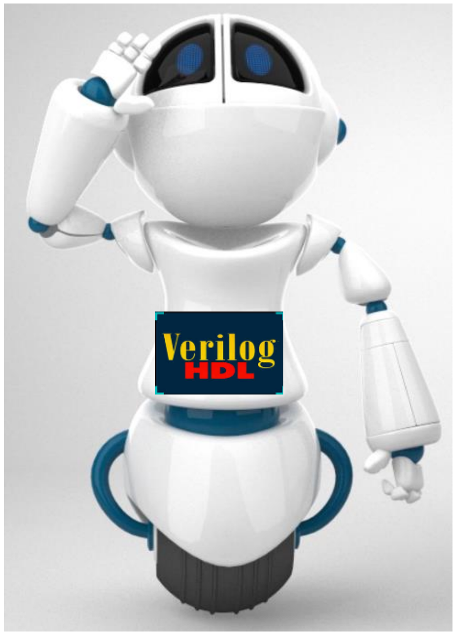
Hajar Falahati

hfalahati@ipm.ir
hfalahati@ce.sharif.edu


System Specification
Architectural Design
Functional Design and Logic Design
Circuit Design
Physical Design
Physical Verification and Signoff
Fabrication
Packaging and Testing
Chip
ENTITY test
port a: in;
end ENTITY;
DRC
LVS
ERC
Partitioning
Floorplanning
Placement
Clock Tree Synthesis
Signal Routing
Timing Closure

Outline

- Deeper on Verilog
  - Wires
  - Rules



Module Fundamentals

Let’s Start With Some Examples

- Implement an 8-bit adder

1-bit Full Adder

name of module

Port list and their declarations (inputs and outputs)

```verilog
module FA (intput a, b, cin, output s, cout);
       wire t1, t2, t3;
       xor (t1, a, b);
       xor (s, t1, cin);
       and (t2, a, b);
       and (t3, t1, cin);
       or (cout, t2, t3);
endmodule
```

8-bit Adder

```
module adder-8bit (input [7:0] a, b, output [7:0] s, output cout);
       wire [6:0] c;
         FA fa0 (a[0], b[0], 1’b0, s[0], c[0]);
         FA fa1 (a[1], b[1], c[0], s[1], c[1]);
         FA fa2 (a[2], b[2], c[1], s[2], c[2]);
         FA fa3 (a[3], b[3], c[2], s[3], c[3]);
         FA fa4 (a[4], b[4], c[3], s[4], c[4]);
         FA fa5 (a[5], b[5], c[4], s[5], c[5]);
         FA fa6 (a[6], b[6], c[5], s[6], c[6]);
         FA fa7 (a[7], b[7], c[6], s[7], cout);
endmodule

module FA (input a, b, cin, output s, cout);
       wire t1, t2, t3;
       xor (t1, a, b);
       xor (s, t1, cin);
       and (t2, a, b);
       and (t3, t1, cin);
       or (cout, t2, t3);
endmodule
```

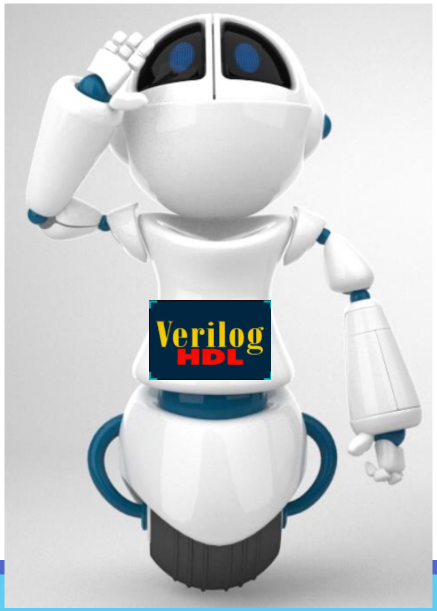
Important Module Rules

```
module adder-8bit (input [7:0] a,b, output [7:0] s, output cout);
        wire [6:0] c;
           FA fa0 (a[0], b[0], 1’b0, s[0], c[0]);
           FA fa1 (a[1], b[1], c[0], s[1], c[1]);
           FA fa2 (a[2], b[2], c[1], s[2], c[2]);
           FA fa3 (a[3], b[3], c[2], s[3], c[3]);
           FA fa4 (a[4], b[4], c[3], s[4], c[4]);
           FA fa5 (a[5], b[5], c[4], s[5], c[5]);
           FA fa6 (a[6], b[6], c[5], s[6], c[6]);
           FA fa7 (a[7], b[7], c[6], s[7], cout);





endmodule


module FA (input a, b, cin, output s, cout);
        wire t1, t2, t3;
        xor (t1, a, b);
        xor (s, t1, cin);
        and (t2, a, b);
        and (t3, t1, cin);
        or (cout, t2, t3);
endmodule
```

Important Module Rules

```
module adder-8bit (input [7:0] a,b, output [7:0] s, output cout);
        wire [6:0] c;
           FA fa0 (a[0], b[0], 1’b0, s[0], c[0]);
           FA fa1 (a[1], b[1], c[0], s[1], c[1]);
           FA fa2 (a[2], b[2], c[1], s[2], c[2]);
           FA fa3 (a[3], b[3], c[2], s[3], c[3]);
           FA fa4 (a[4], b[4], c[3], s[4], c[4]);
           FA fa5 (a[5], b[5], c[4], s[5], c[5]);
           FA fa6 (a[6], b[6], c[5], s[6], c[6]);
           FA fa7 (a[7], b[7], c[6], s[7], cout);





endmodule


module FA (input a, b, cin, output s, cout);
        wire t1, t2, t3;
        xor (t1, a, b);
        xor (s, t1, cin);
        and (t2, a, b);
        and (t3, t1, cin);
        or (cout, t2, t3);
endmodule
```

- Module instantiation
  - Creating objects (a.k.a., instance)
  - Nested ALLOWED

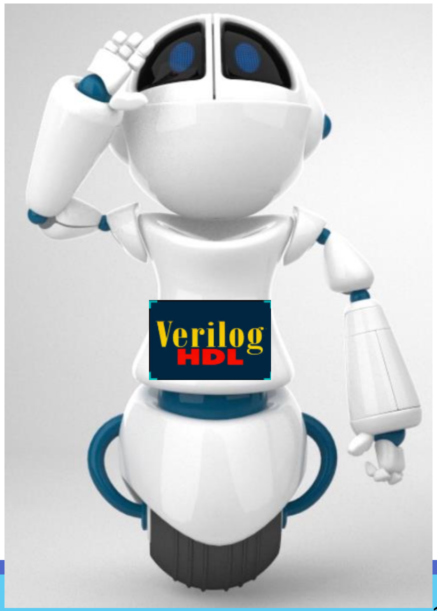


Important Module Rules

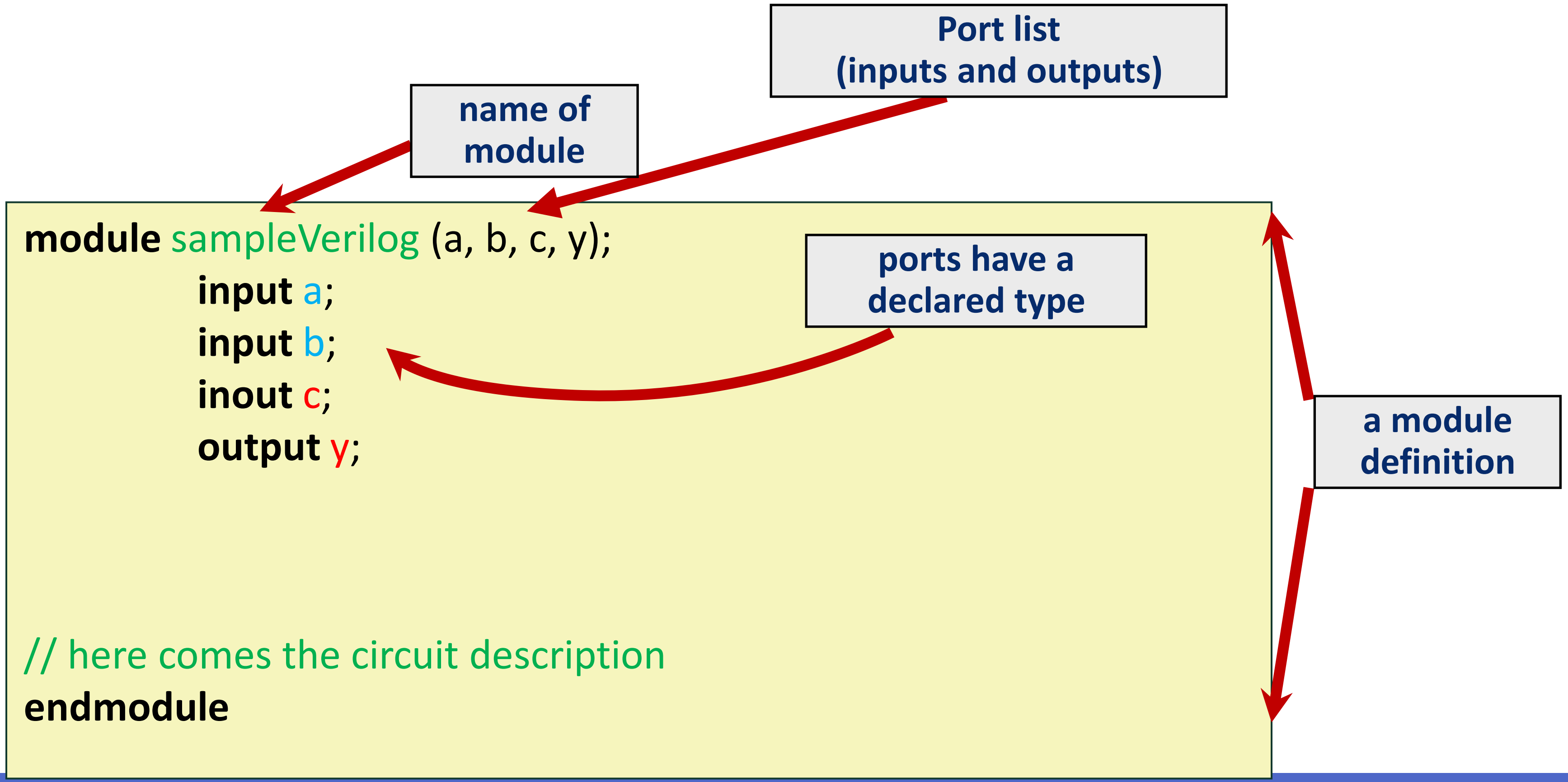
```
module adder-8bit (input [7:0] a,b, output [7:0] s, output cout);
        wire [6:0] c;
           FA fa0 (a[0], b[0], 1'b0, s[0], c[0]);
           FA fa1 (a[1], b[1], c[0], s[1], c[1]);
           FA fa2 (a[2], b[2], c[1], s[2], c[2]);
           FA fa3 (a[3], b[3], c[2], s[3], c[3]);
           FA fa4 (a[4], b[4], c[3], s[4], c[4]);
           FA fa5 (a[5], b[5], c[4], s[5], c[5]);
           FA fa6 (a[6], b[6], c[5], s[6], c[6]);
           FA fa7 (a[7], b[7], c[6], s[7], cout);
            module FA (input a, b, cin, output s, cout);
                  ....
               endmodule
endmodule


module FA (input a, b, cin, output s, cout);
        wire t1, t2, t3;
        xor (t1, a, b);
        xor (s, t1, cin);
        and (t2, a, b);
        and (t3, t1, cin);
        or (cout, t2, t3);
endmodule
```

- Module instantiation
  - Creating objects (a.k.a., instance)
  - Nested ALLOWED

- Module definition
  - Nested NOT ALLOWED

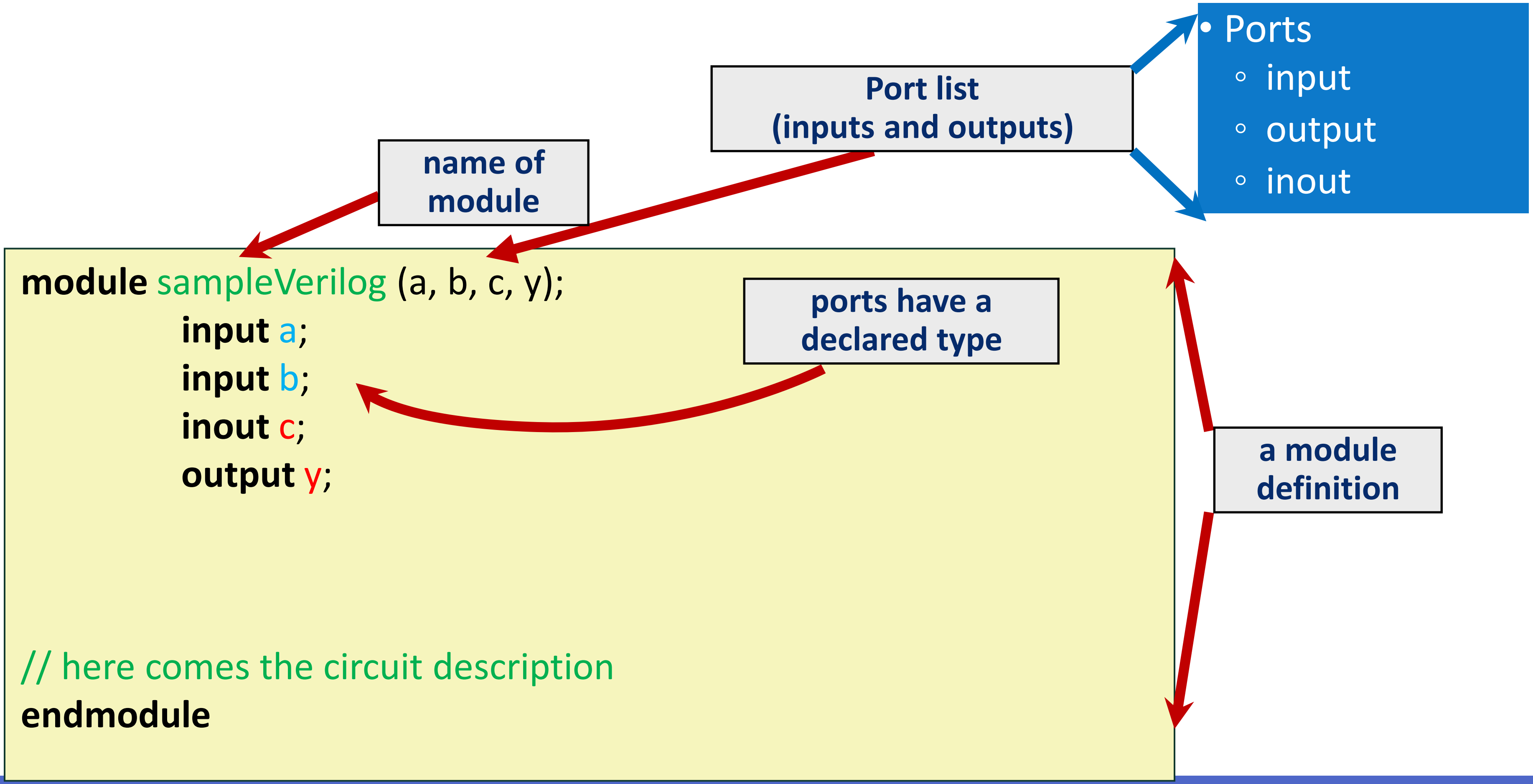


Module May Includes…



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



// here comes the circuit description
endmodule
```

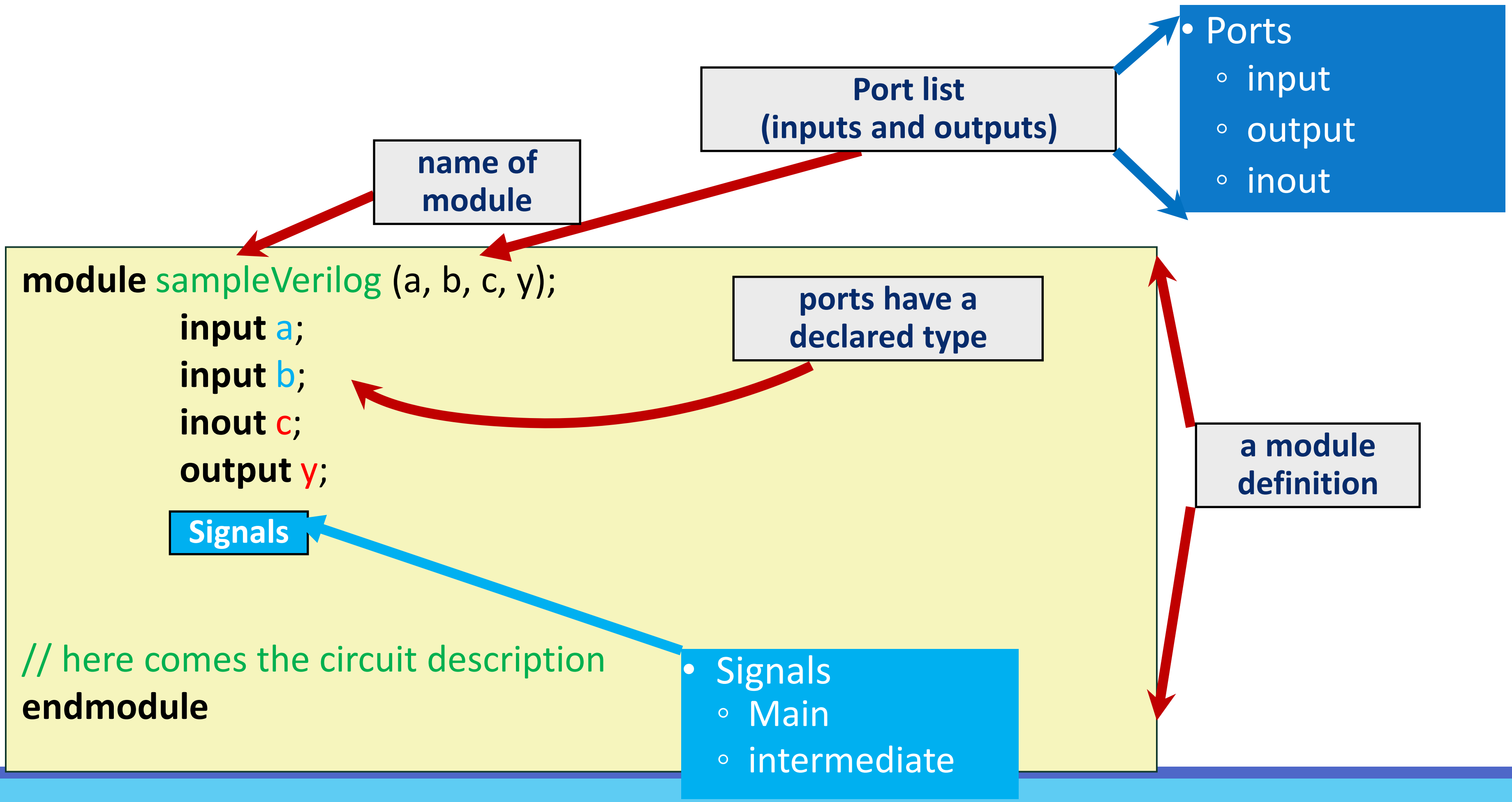
Module May Includes...



- Ports
  - input
  - output
  - inout

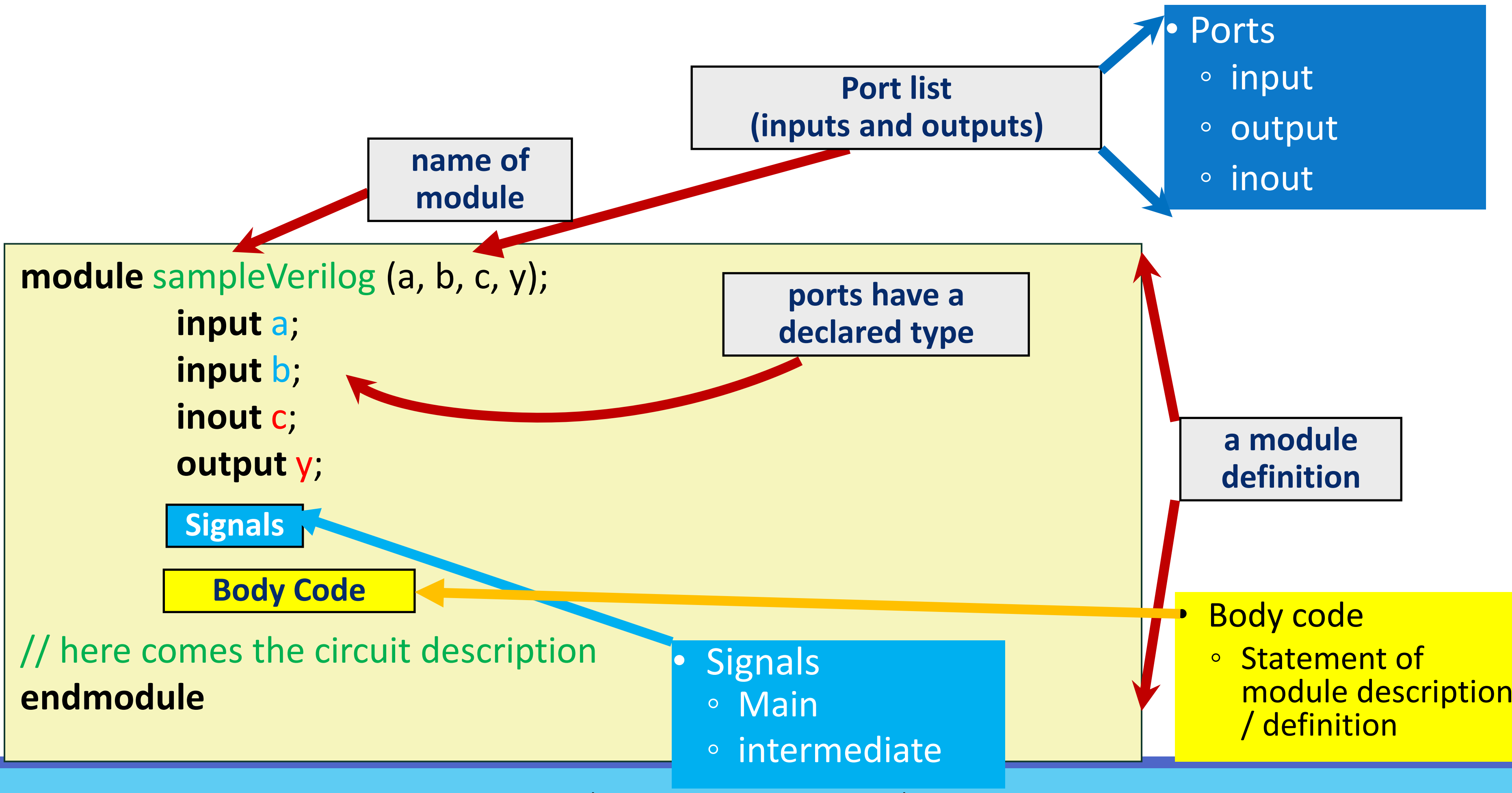
```
module sampleVerilog (a, b, c, y);
        input a;
        input b;
        inout c;
        output y;



// here comes the circuit description
endmodule
```

Module May Includes...



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

// here comes the circuit description
endmodule
```

Module May Includes…



```
module sampleVerilog (a, b, c, y);
            input a;
            input b;
            inout c;
            output y;
            Signals
            Body Code
// here comes the circuit description
endmodule
```

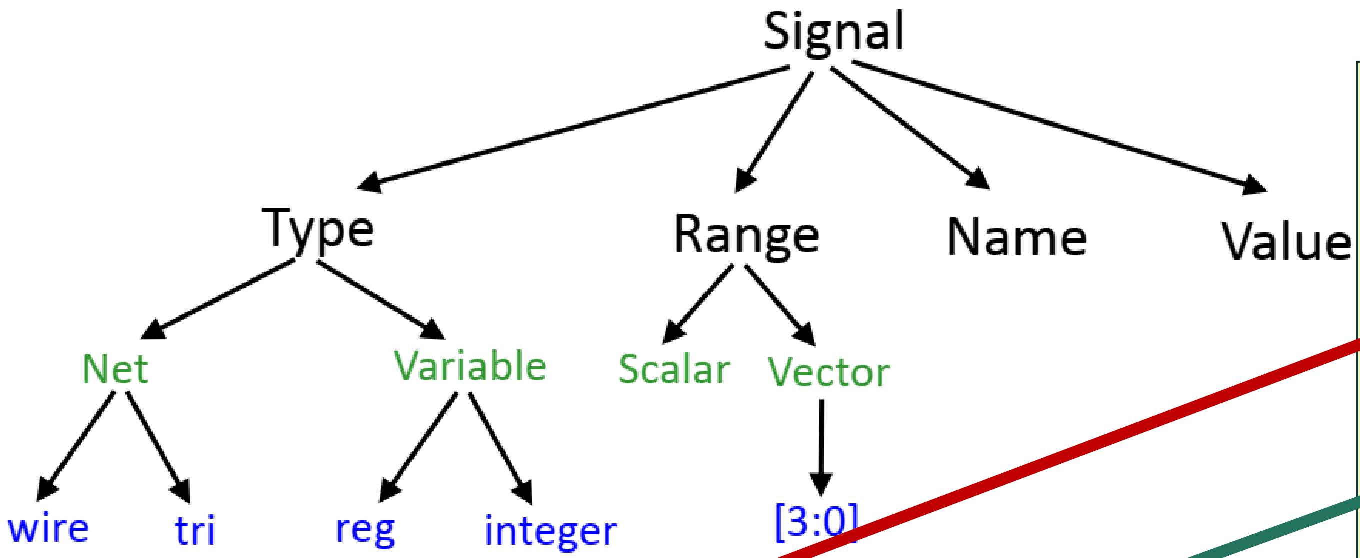
- Ports
  - input
  - output
  - inout

- Signals
  - Main
  - intermediate

- Body code
  - Statement of module description / definition

Signals

Signals



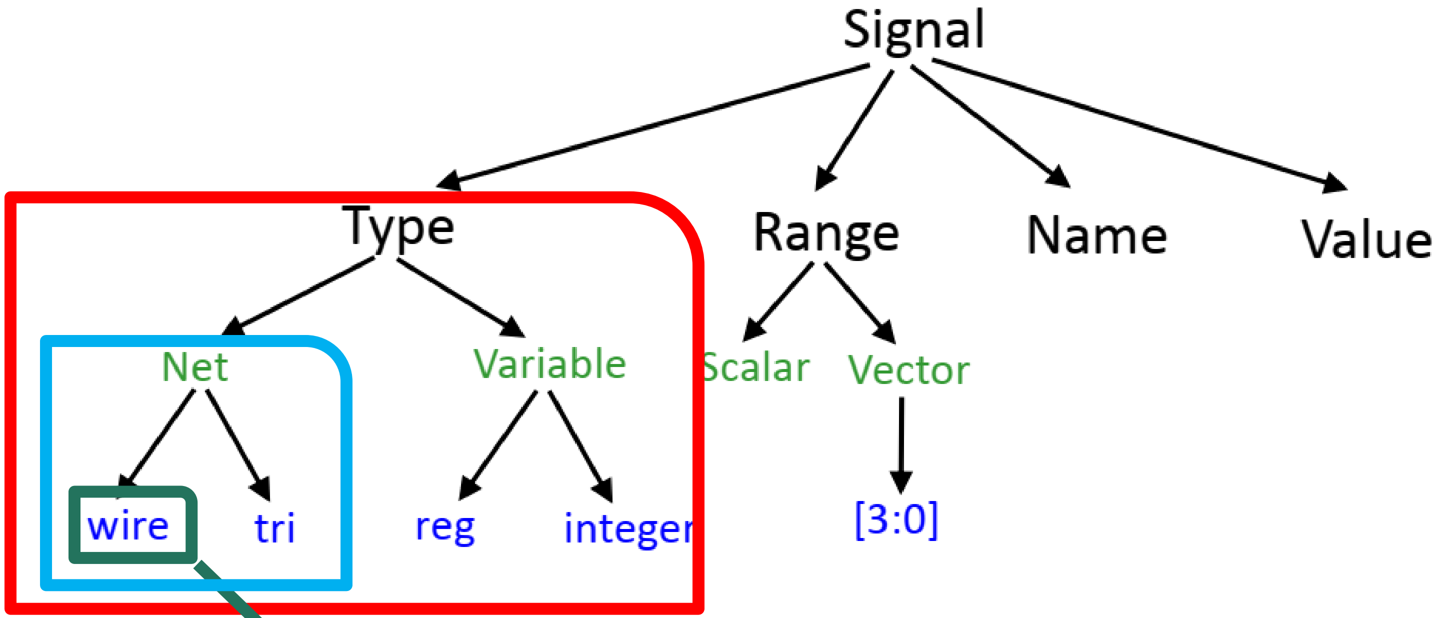
```
module Sample (a, y);
      input  a;
      output y;
      wire w;
      wire [2:0] tmp;
      reg y;

      tmp = 3'b 001
         ......

endmodule
```

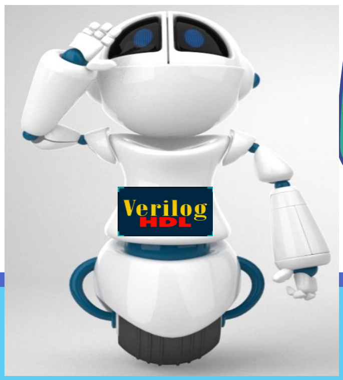
vector

Each element of a vector can be accessed

tmp[0] = 1
tmp[1] = 0
tmp[2] = 0

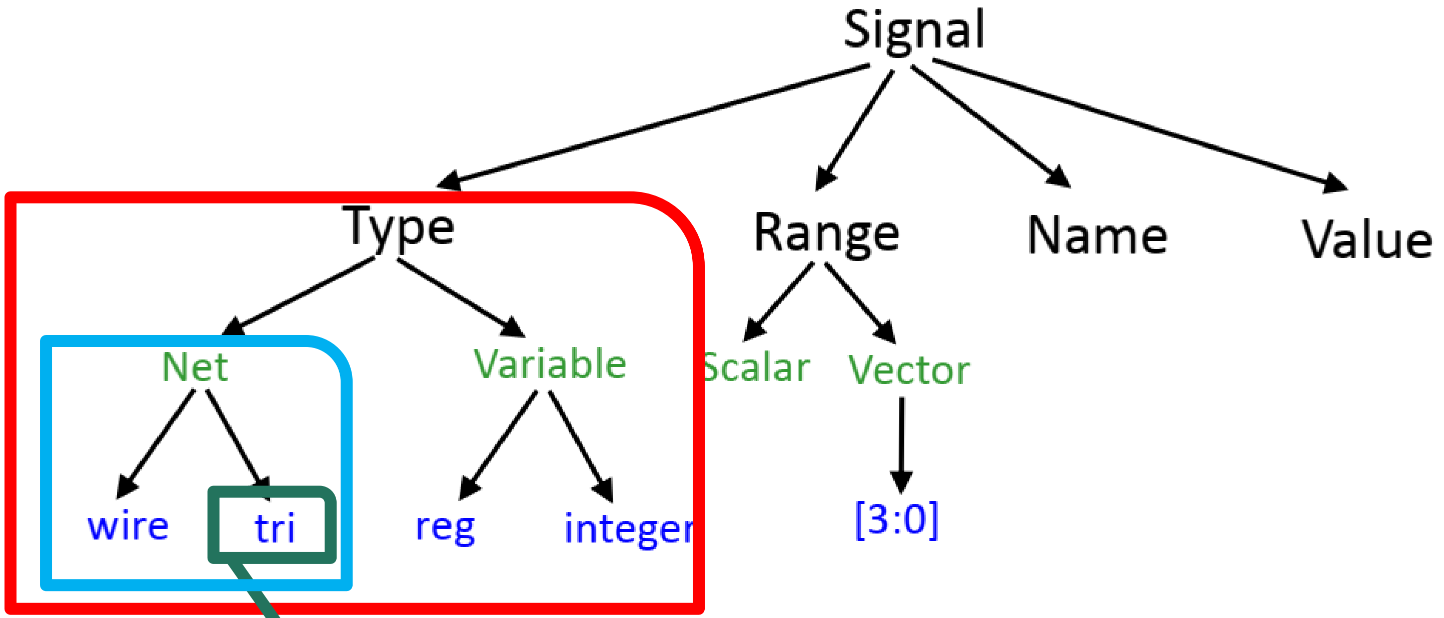
Signal Type





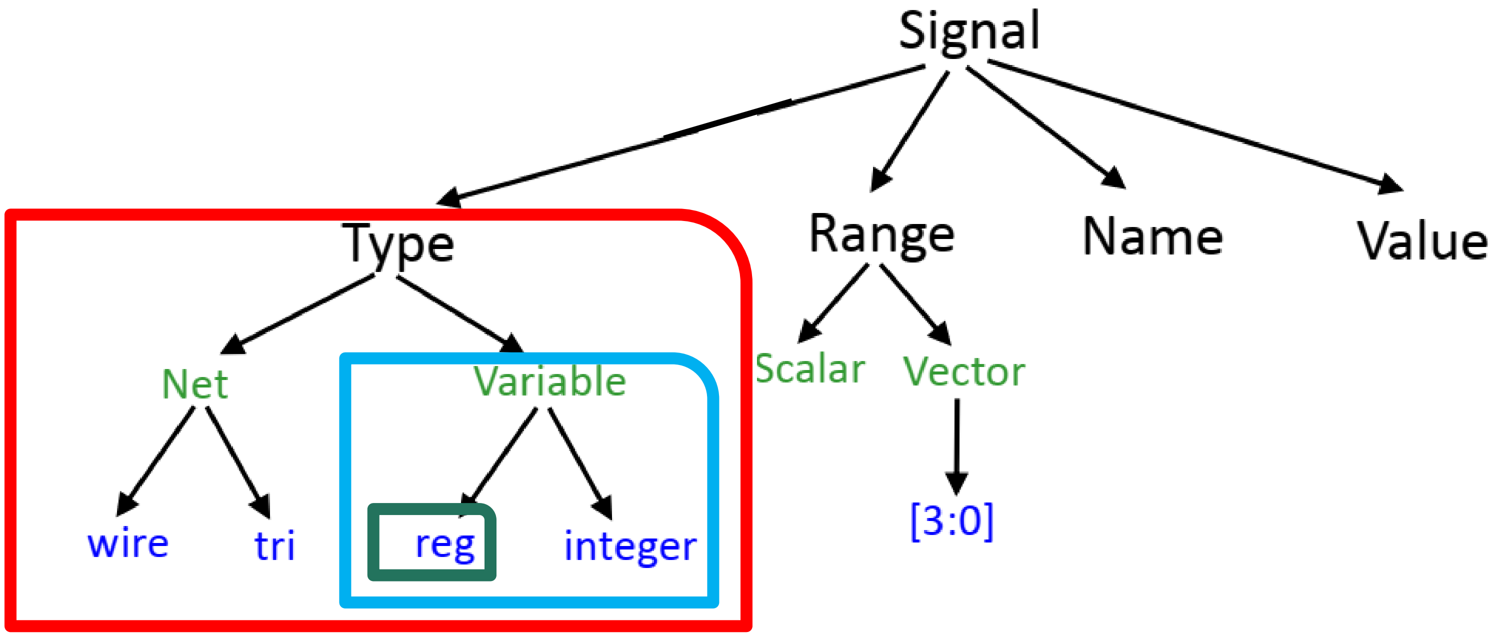
- For interconnecting logic elements (LEs)
- To connect an output of a logic element to the input of another LE

Signal Type



- Circuit nodes that are connected in a tri-state fashion

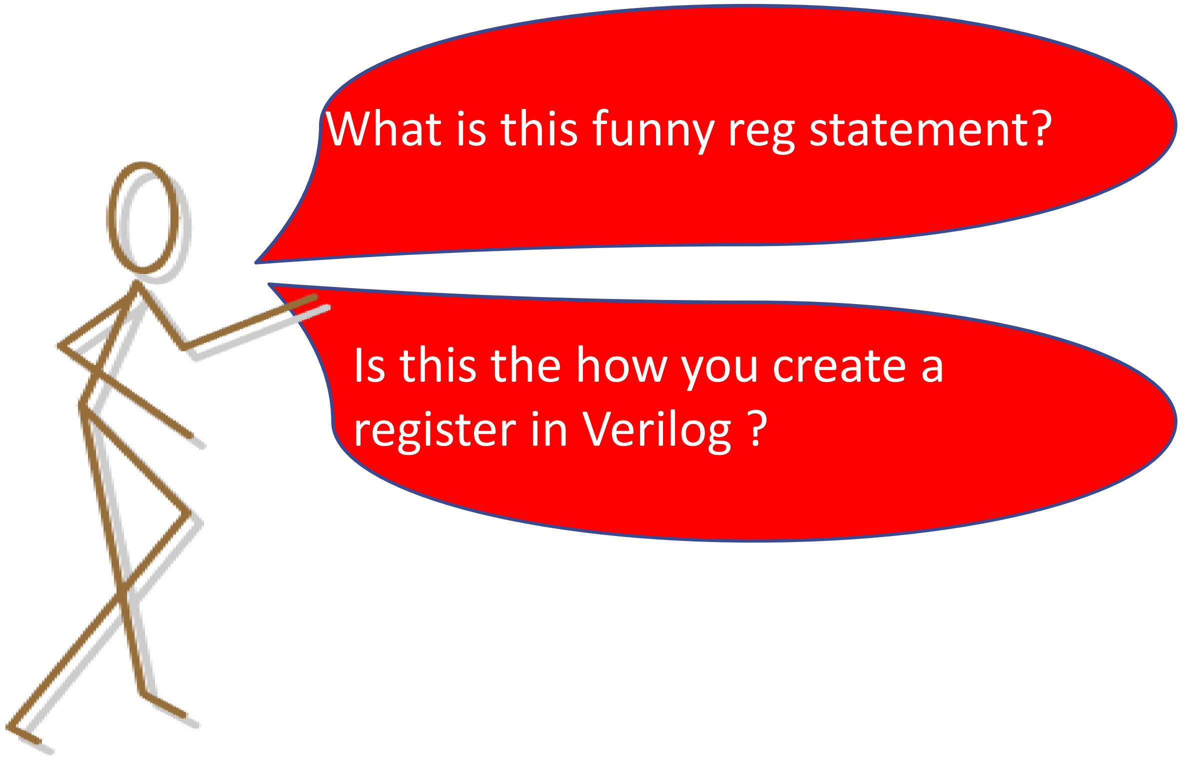
Signal Type


Signal
Type
Range
Name
Value
Net
Variable
Scalar
Vector
wire
tri
reg
integer
[3:0]

Verilog HDL

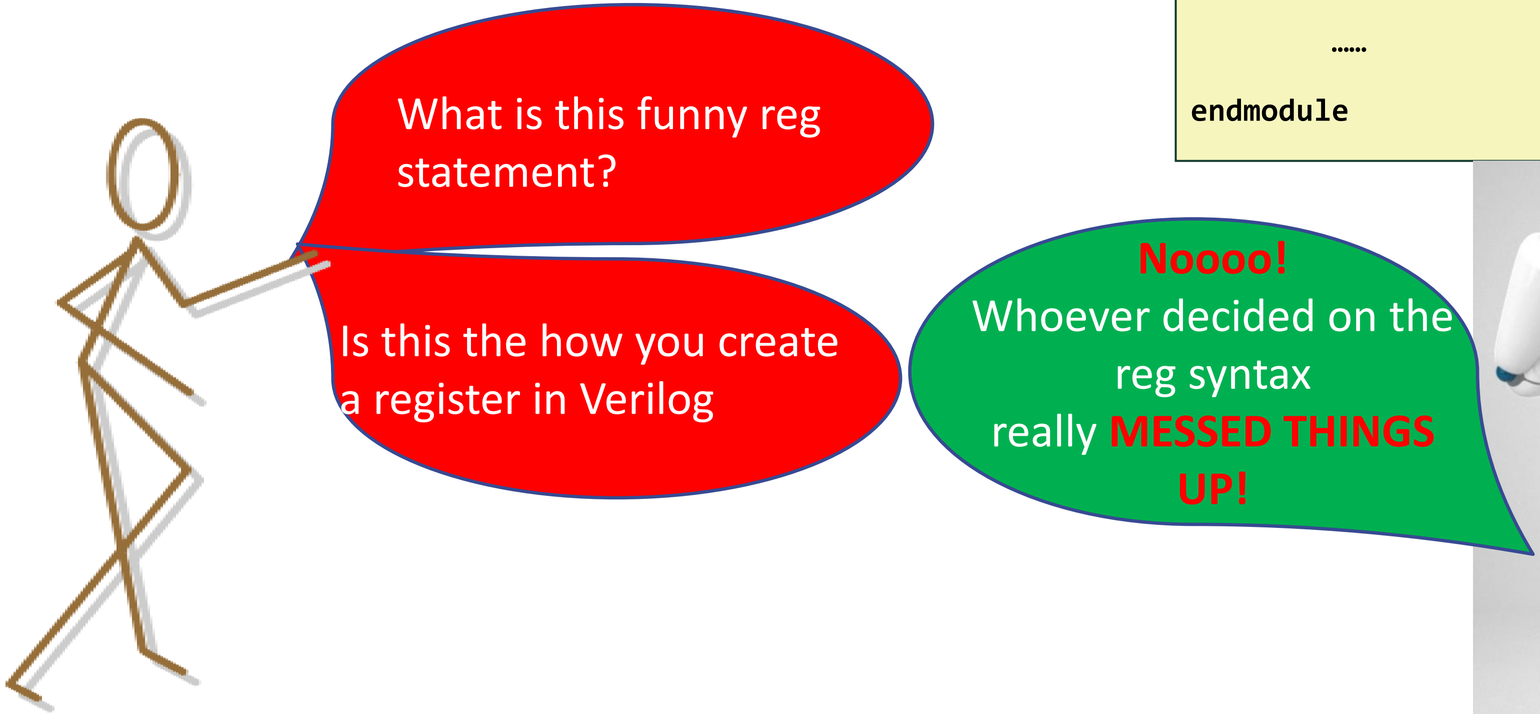
Reg?!

```
module Test-Module (a, y);
     input  a;
     output y;
     wire w;
     reg y;


        ……

endmodule
```


What is this funny reg statement?
Is this the how you create a register in Verilog ?

Reg?

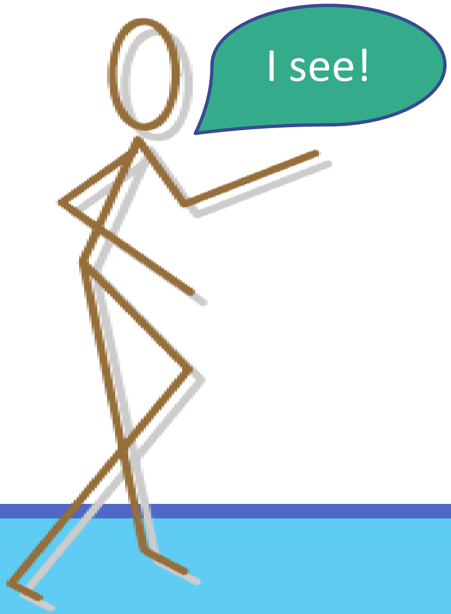
```
module Test-Module (a, y);
     input  a;
     output y;
     wire w;
     reg y;


        ……

endmodule
```

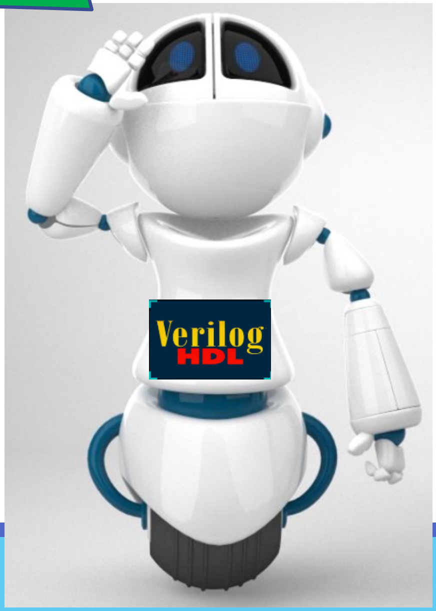

What is this funny reg statement?
Is this the how you create a register in Verilog
Noooo! Whoever decided on the reg syntax really MESSED THINGS UP!

Reg Type

- In Verilog a reg is just a variable
- When you see reg think variable not hardware register
- reg variable may or may not actually represent a hardware register

```
module Test-Module (a, y);
      input  a;
      output y;
      wire w;
      reg y;

         ......

endmodule
```

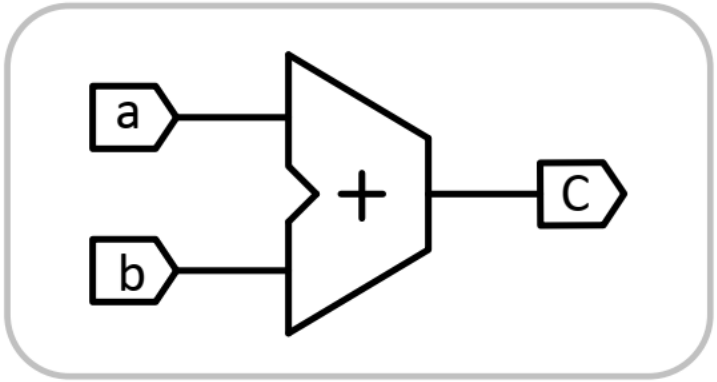




Reg Type: Example

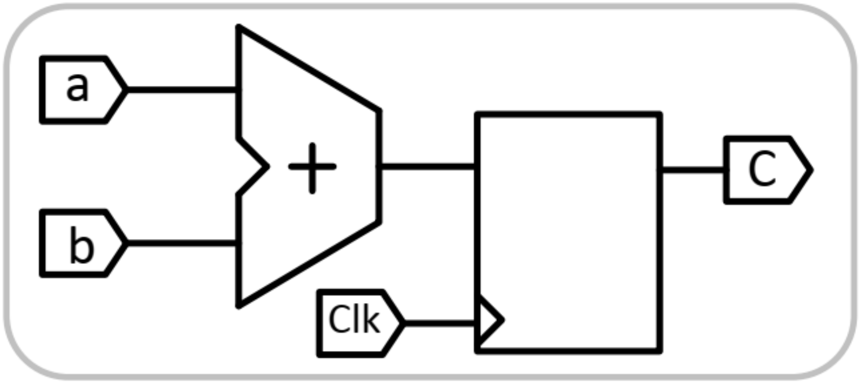
Not Register

```
reg  C;
    always @ (a,b)
        C = a+b;
```

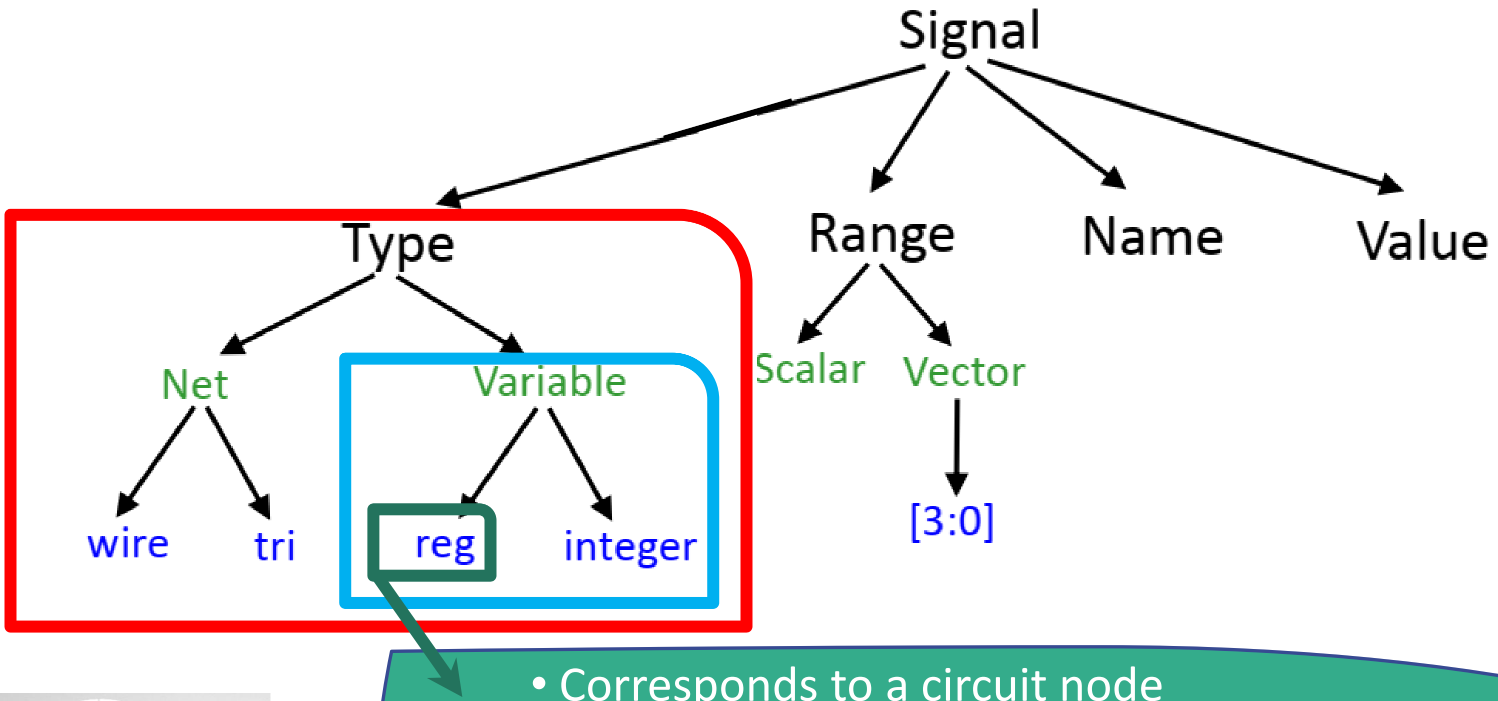


Register

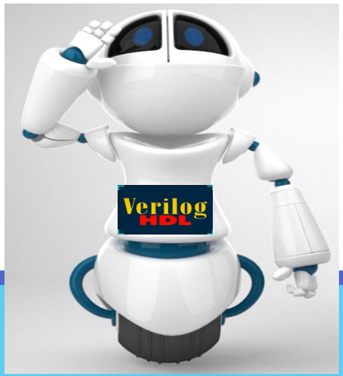
```
reg  C;
    always @ (posedge Clk)
        C <= a+b;
```



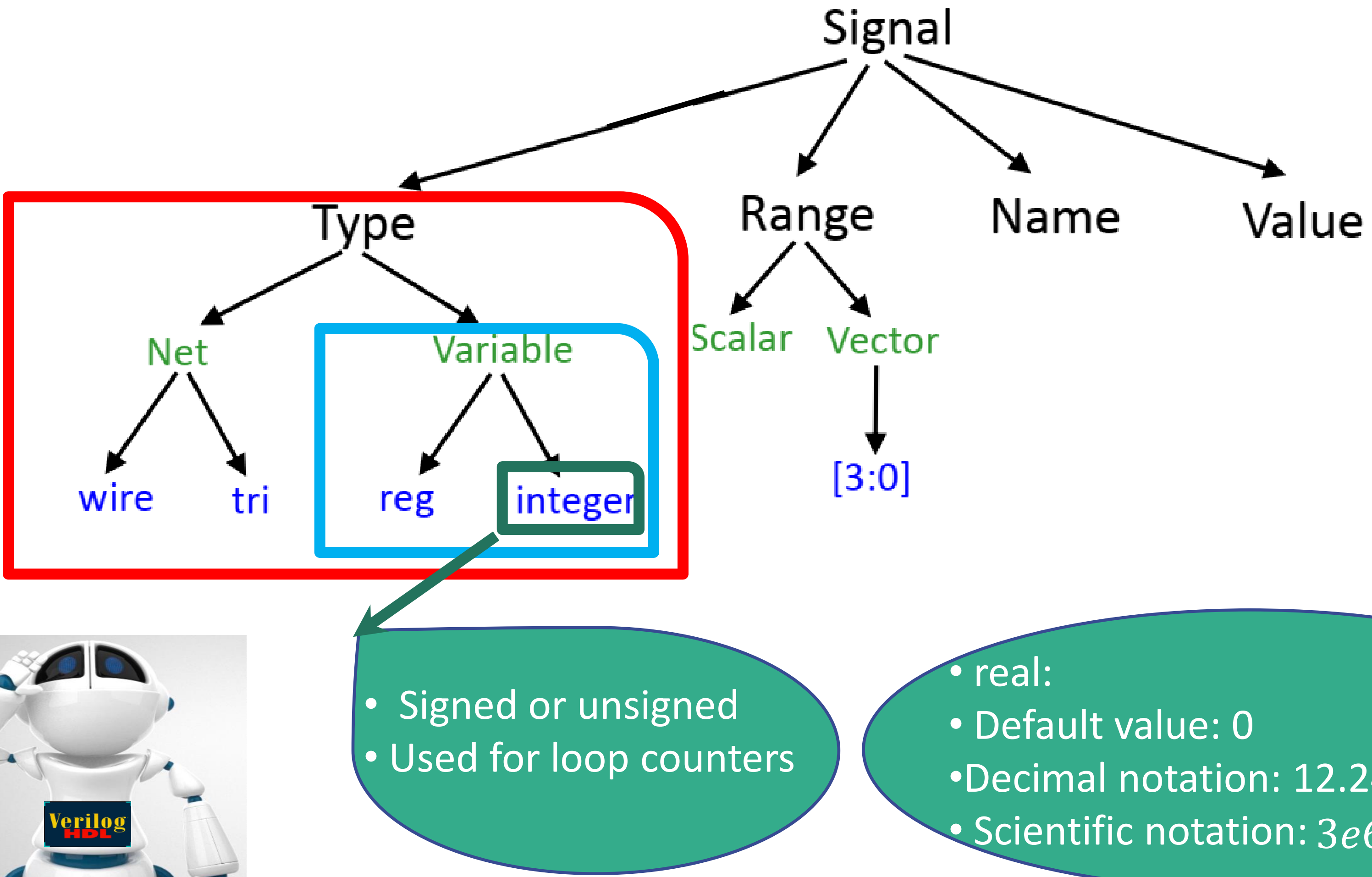
Signal Type



- Corresponds to a circuit node
  - Not necessarily a register!
- Allow a circuit to be described in terms of its behavior
- Retains its value until it is overwritten by a subsequent assignment



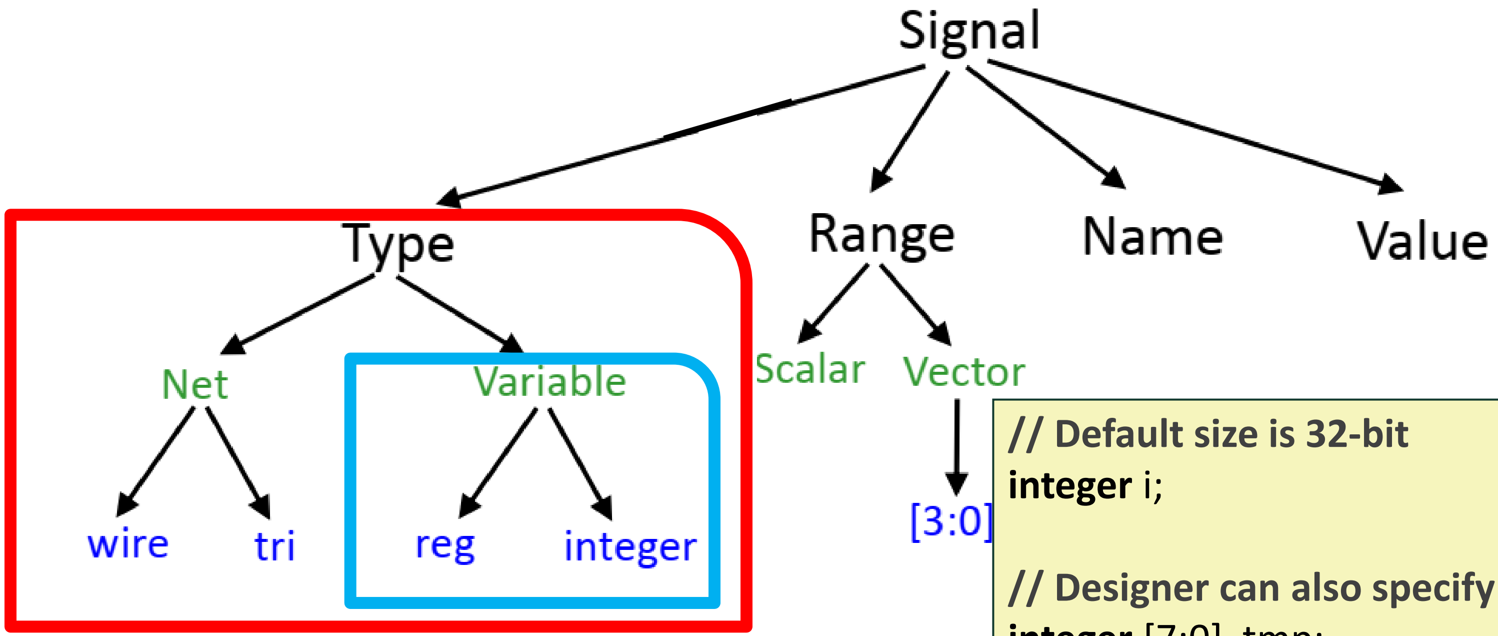
Signal Type



- Signed or unsigned
- Used for loop counters

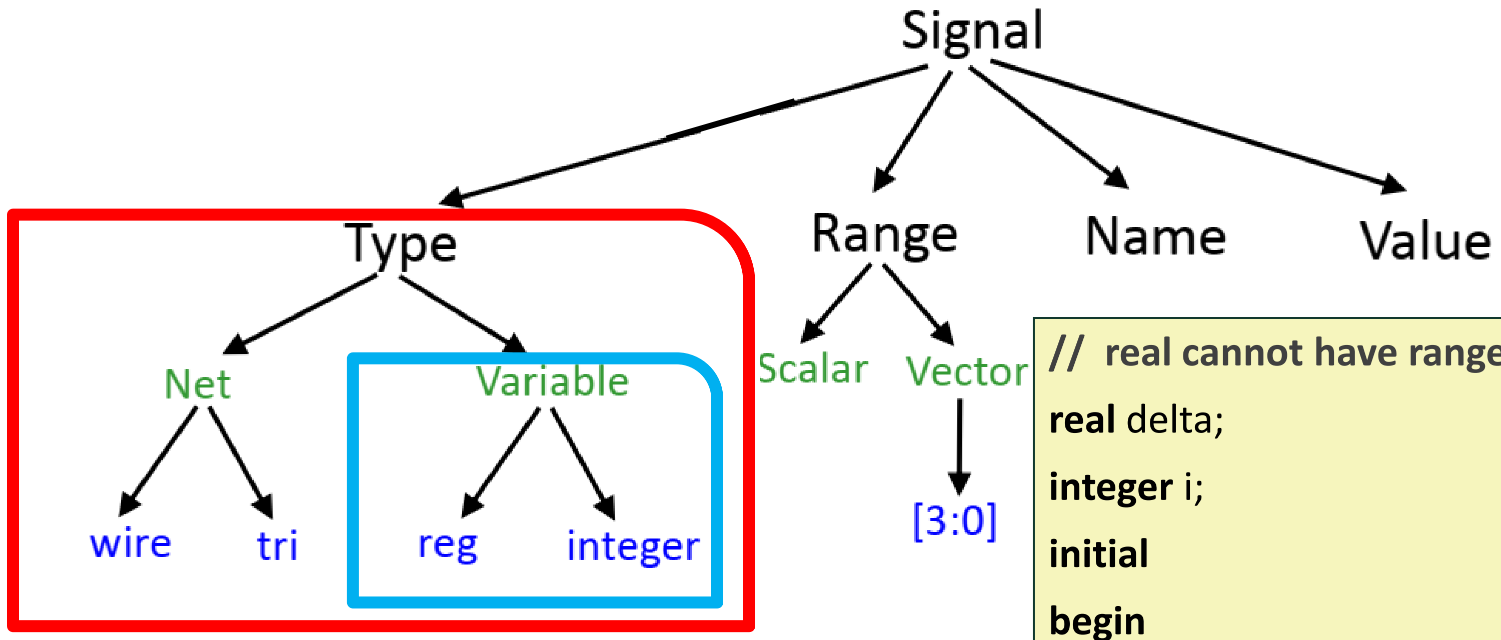
- real:
- Default value: 0
- Decimal notation: 12.24
- Scientific notation: $3e6$ $(= 3 \times 10^6)$

Signal Type: Example

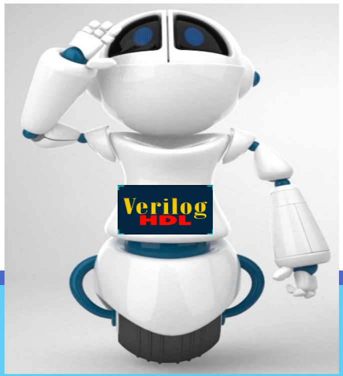


```
// Default size is 32-bit
integer i;

// Designer can also specify a width
integer [7:0]  tmp;

//reg vector is unsigned by default
// Unless specified as signed
reg signed [7:0]  temp-reg;
```

Signal Type: Example



```
//  real cannot have range declaration
real delta;
integer i;
initial
begin
   delta=4e10;   delta=2.13;
i = delta;
// i gets the value 2
// (rounded value of 2.13)
end
```

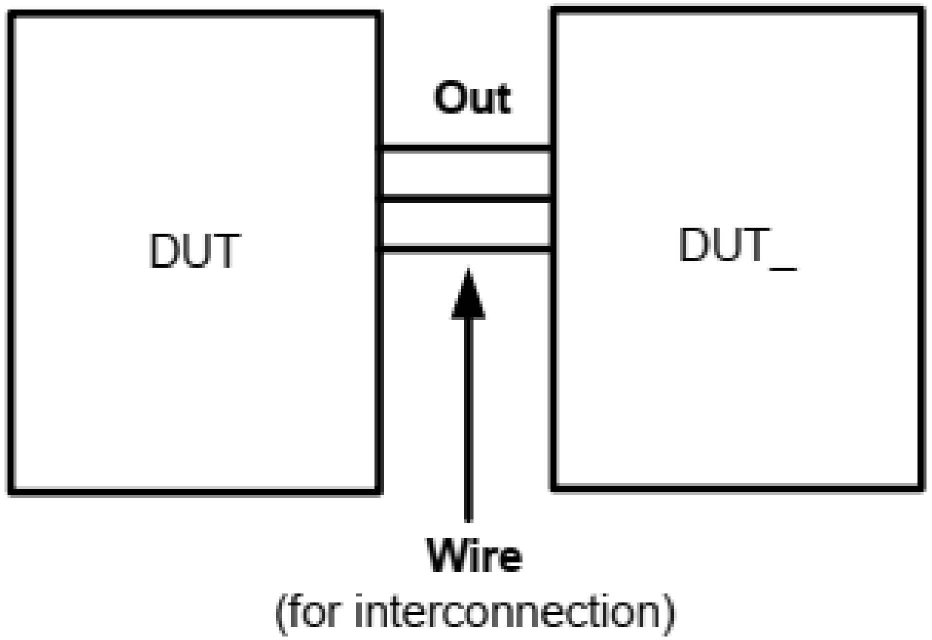


Signal Types

- wire
  - Declarations are not necessary as Verilog assumes that signals
- reg
  - Declarations is required

```
module Test-Module (a,
y);
     input  a;
     output y;
     wire w;


     wire a;

     reg y;
      ……

endmodule
```

Signal Types: Example



Signal Types: Example

Ports

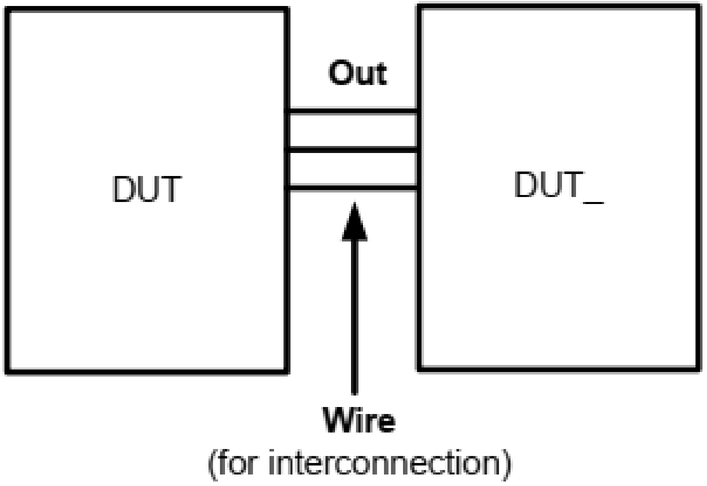
Signals

Code Body

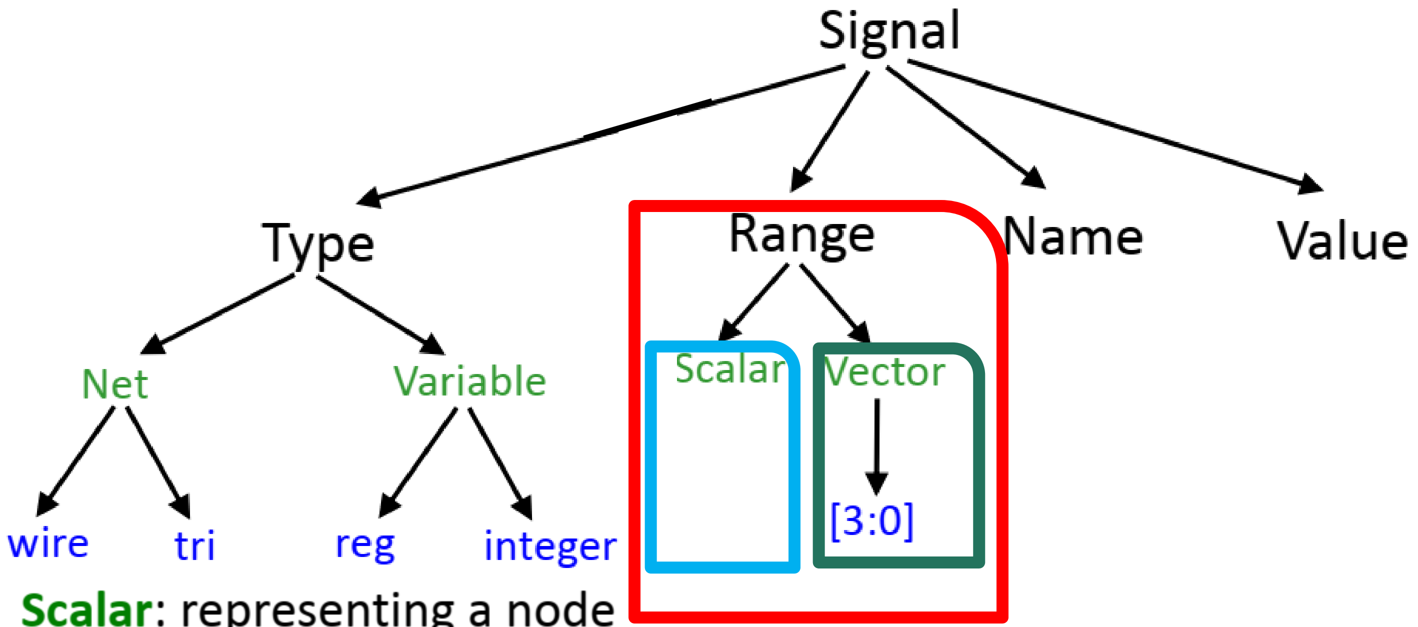
```
module  DUT (s, Out);
   input [3:0] s;
   output [2:0] Out;

   wire [2:0] Out;
   reg [2:0] Count;
   integer k;

   Count = 0;
   for (k=0; k<4; k=k+1)
      if (s[k])
         Count = Count + 1;
   assign Out = Count;

endmodule
```

Loop counter

";" at the end of each line



Signal Range



Scalar: representing a node

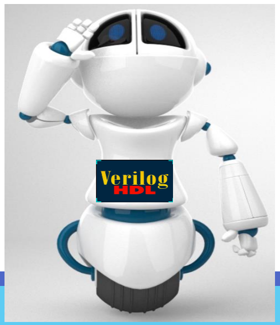
```
reg  C;
wire B;
```

Vector: representing a bus

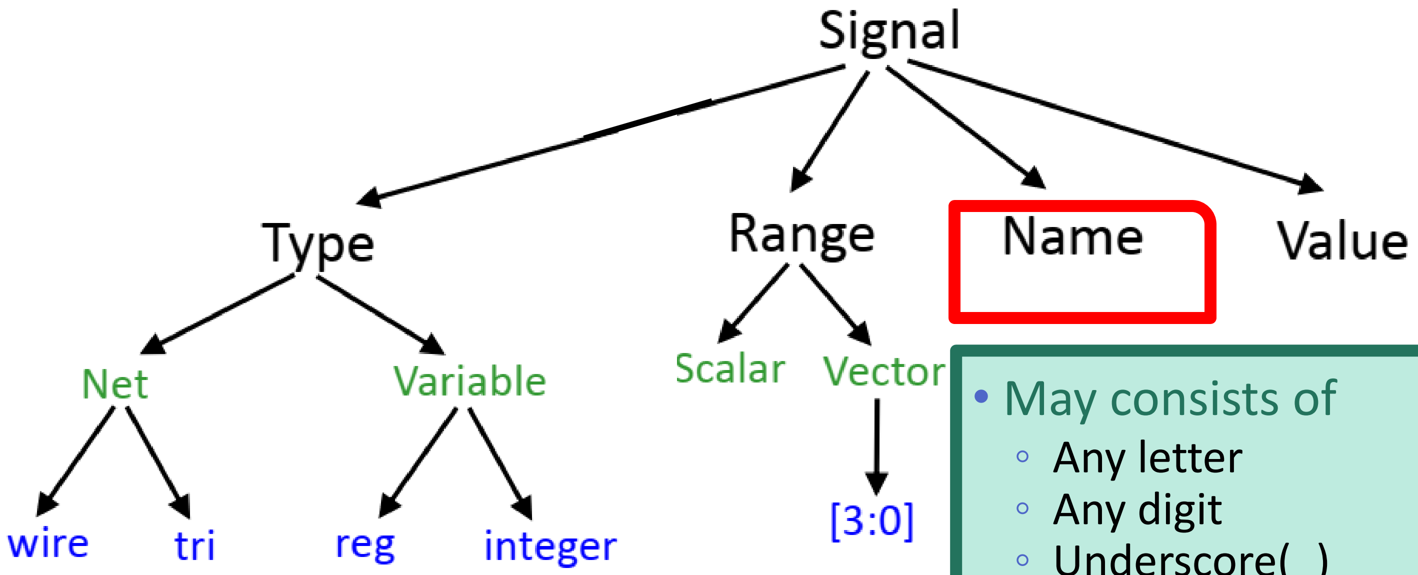
```
reg  [10:0] Data;
reg  [0:6] S;
wire [7:4] B;
```

Each element of a bus can be accessed

```
assign a = Data[8];
```

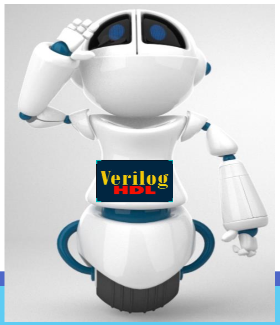


Signal Name



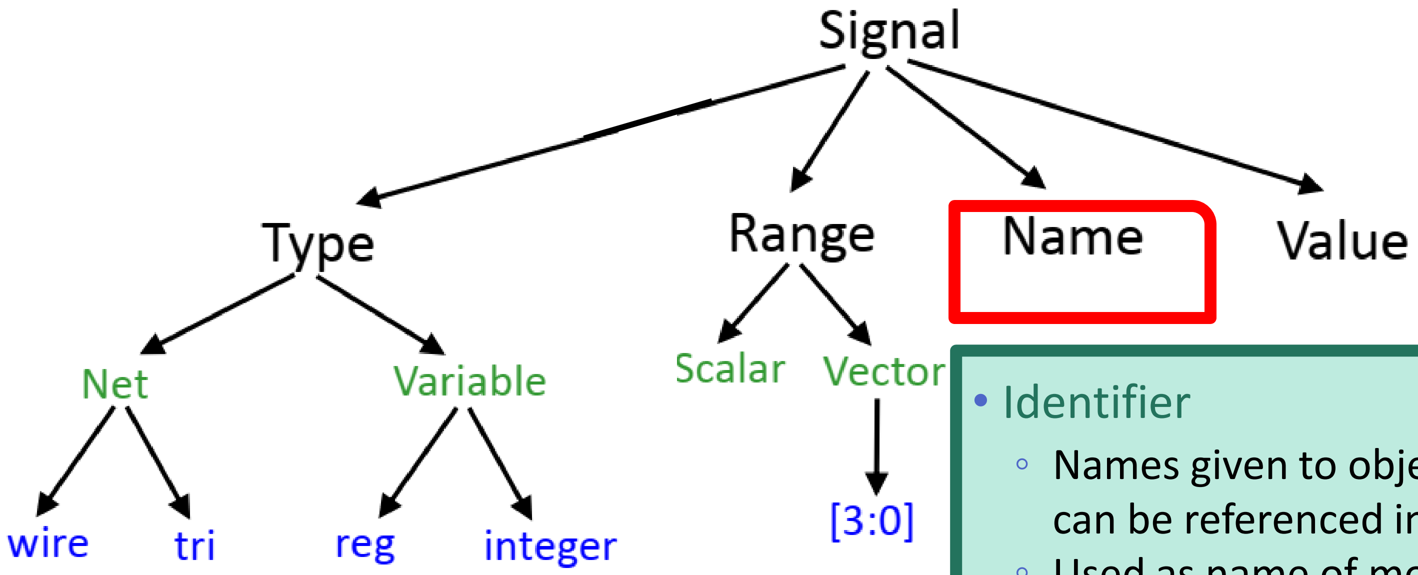
- May consists of
  - Any letter
  - Any digit
  - Underscore(_)
  - $ sign

- Do not
  - Should not start with a digit
  - Should not be a Verilog keyword
  - Should not be $



Legal	Illegal
A_m	1xb
B1_signal	wire
My$	R&z

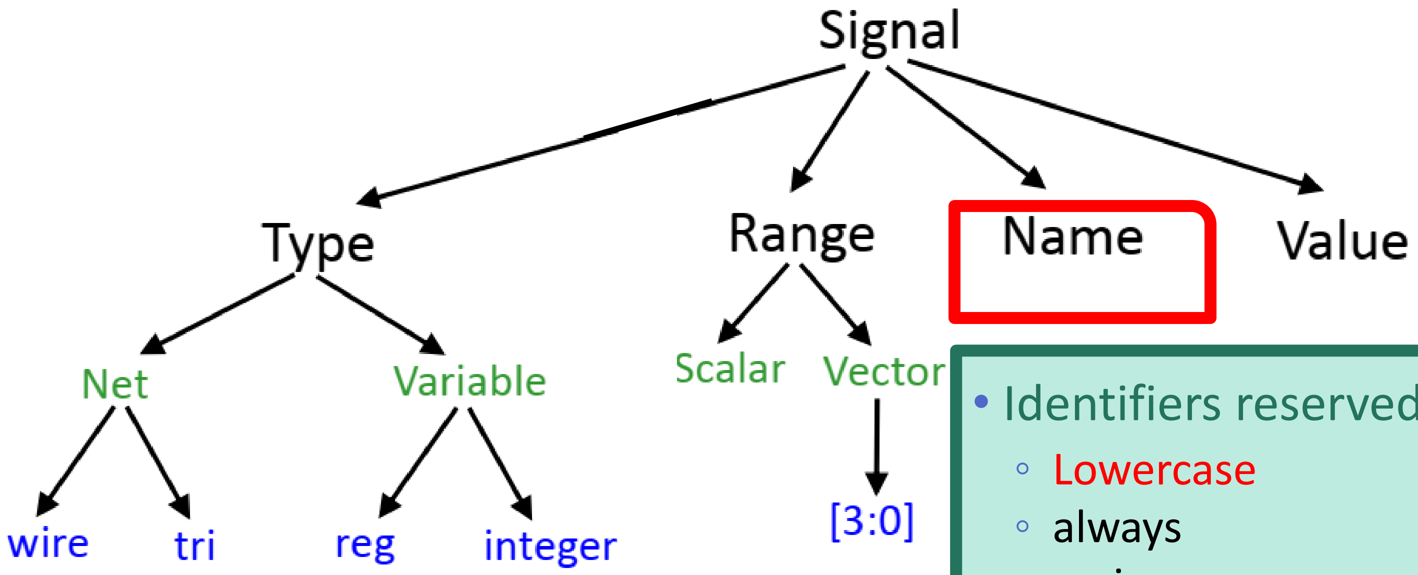
Signal Name: Identifier



```
\a+b+c          // a+b+c is the identifier
\**my_name**   // **my_name** is the identifier
```

- Identifier
  - Names given to objects so that they can be referenced in design
  - Used as name of modules
  - **Start with '\'**
  - **End with whitespace (space, tab, newline)**
  - Any character between start and end
  - The start and whitespace are not part of the identifier

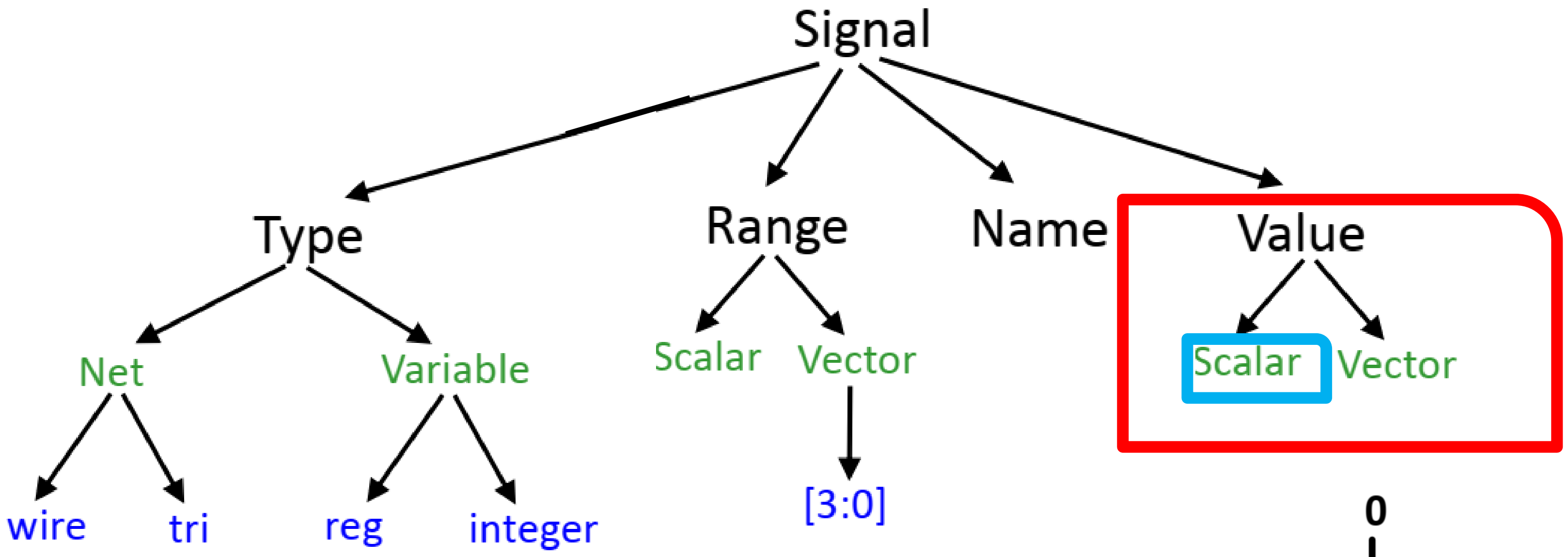
Signal Name: Keywords



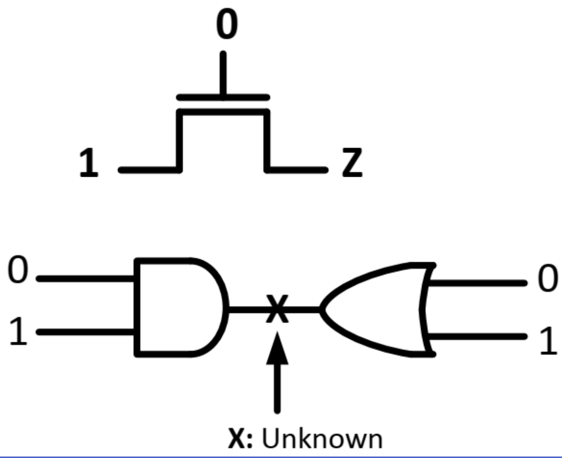
```
reg value;
input clk;
```

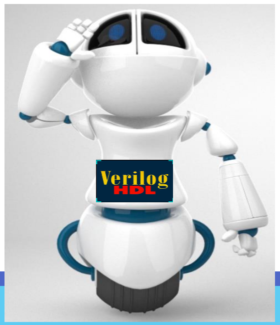
- Identifiers reserved by Verilog
  - Lowercase
  - always
  - assign
  - begin
  - end
  - reg, wire
  - input, output, inout
  - and, or, buf, bufif0, bufif1

Signal Value

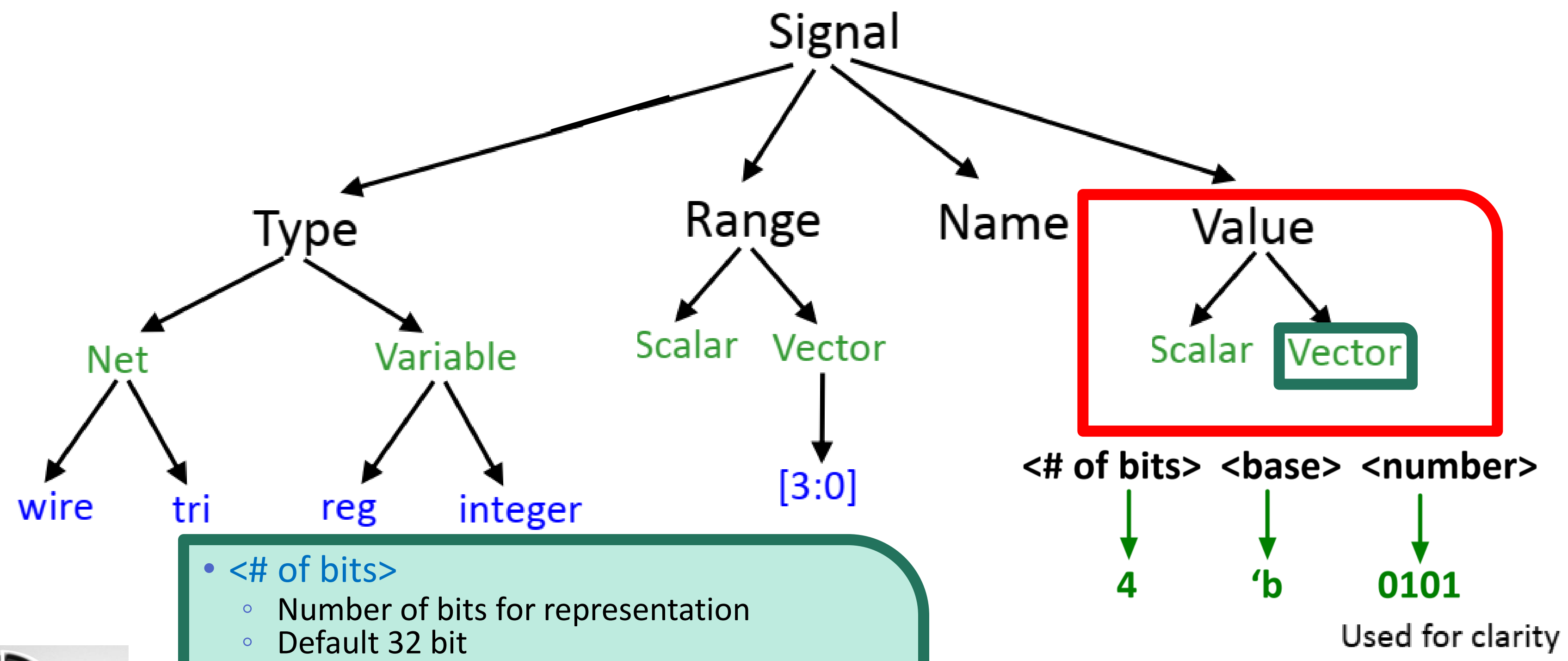


- Each **scalar** signal can have **four** possible values
  - 0: Logic value “0”
  - 1: Logic value “1”
  - Z(z): Tri-state (high impedance)
  - X(x): Unknown value

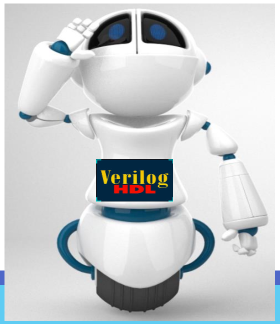




Signal Value



- <# of bits>
  - Number of bits for representation
  - Default 32 bit
- <base>
  - "d" : Decimal
  - "b" : Binary
  - "h" : Hexadecimal
  - "o" : Octal
- <number>
  - Signal value in base



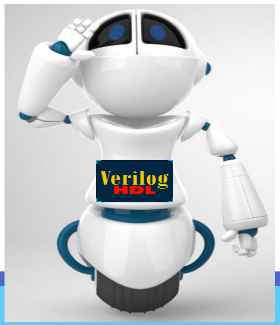
Example:

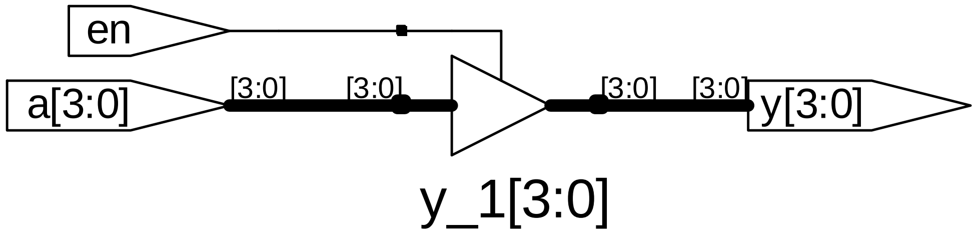
```
K = 8'ha9;      // K=1010_1001
C= 4'd3;        // C=0011
D= 4'b100;      // D=0100
F= 'b10x;       // F=10X
L = -6'b3       // L = 111101
```

Floating Signals

- Signal that is not driven by any circuit
  - Open circuit
  - Floating wire
  - High impedance
  - Hi-Z
  - Tri-state

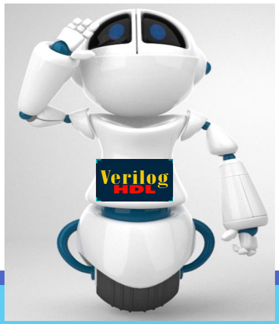
```
module tristate_buffer(input  [3:0] a,
                       input        en,
                       output [3:0] y);

        y = en ? a : 4'bz;

endmodule
```





Multiple Drivers!

- Consider a node that is connected to multiple drivers
- What is the value of node?

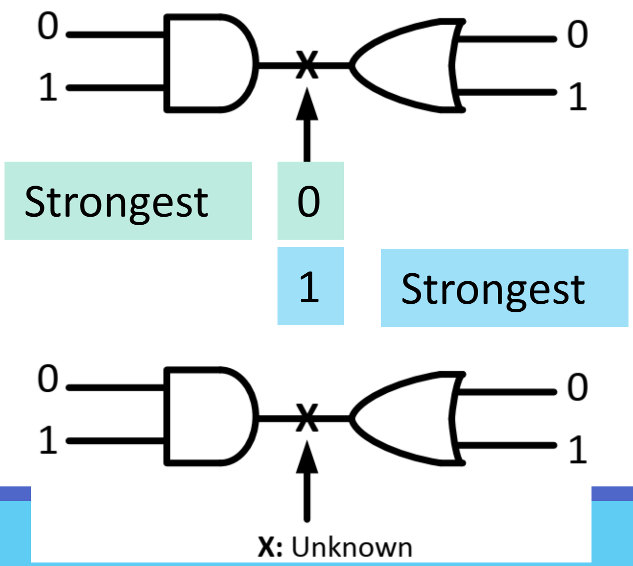


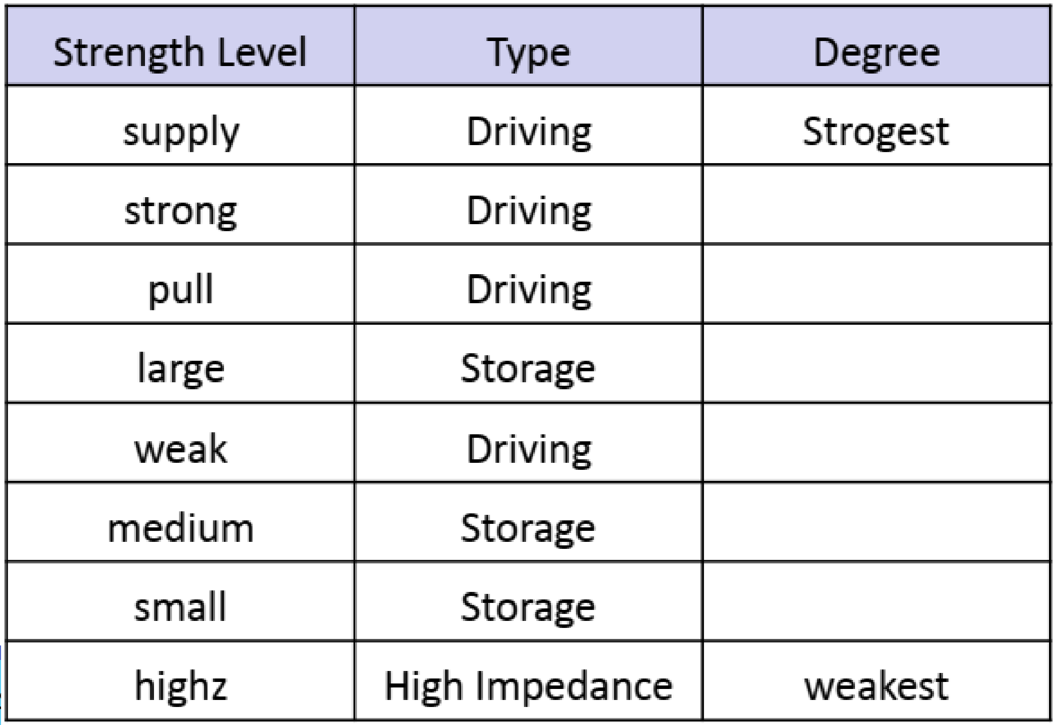
The Value in Multiple Drivers?

- Consider a node that is connected to multiple drivers
- What is the value of node?
  - Signal strength will help you ☺

Signal Strength

- Resolving the contention among different drivers
- Multiple drivers with different strength level and different values
  - The driver with stronger strength level determines the value
- Multiple drivers with the same strength level and different values
  - The value is unknown



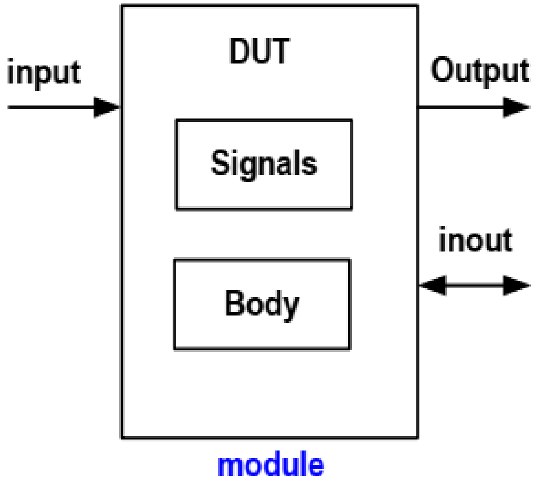


Strength Level	Type	Degree
supply	Driving	Strogest
strong	Driving	
pull	Driving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	
highz	High Impedance	weakest

Ports

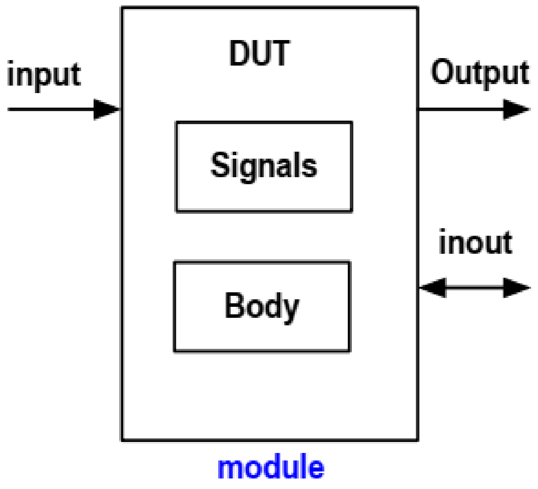
Remember

- Internals of the module are not visible to the environment
- Internals can be changed



Port

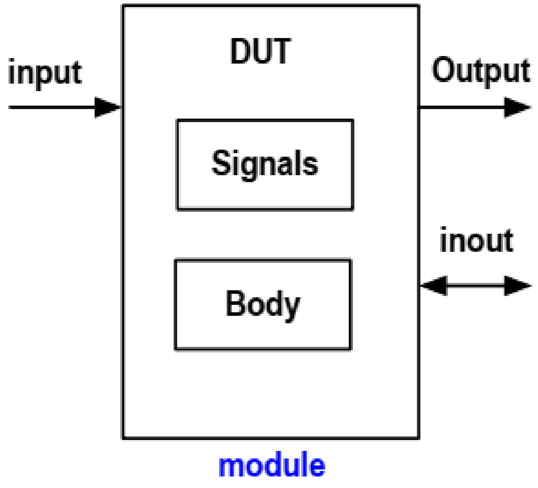
- Interface
- Ports are visible to the environment
- The interface (ports) is not changed
- Port definition
  - Port list
  - Port mode
  - Port type



Port List

Port list
(inputs and outputs)

```
module sampleVerilog (a, b, c, y);


// here comes the circuit description
endmodule


module top;
    ...
    <module internals>
    ...
endmodule
```

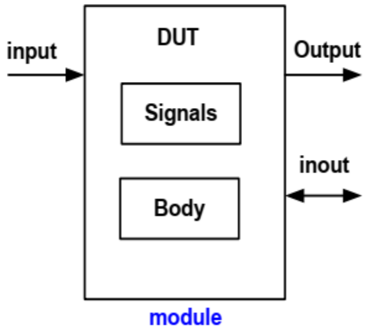


Port Mode

- **input**
  - Input port
- **output**
  - Output port
- **inout**
  - Bidirectional port

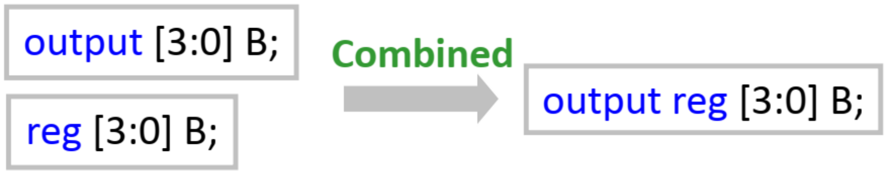
```
module sampleVerilog (a, b, c, y);
            input a;
            input b;
            inout c;
            output y;
            .....
endmodule
```

```
module fullAdder4 (sum, c_out, a, b, c_in);
            output [3:0] sum;
            output c_out;
            input [3:0] a, b;
            input c_in;
             .... .
endmodule
```



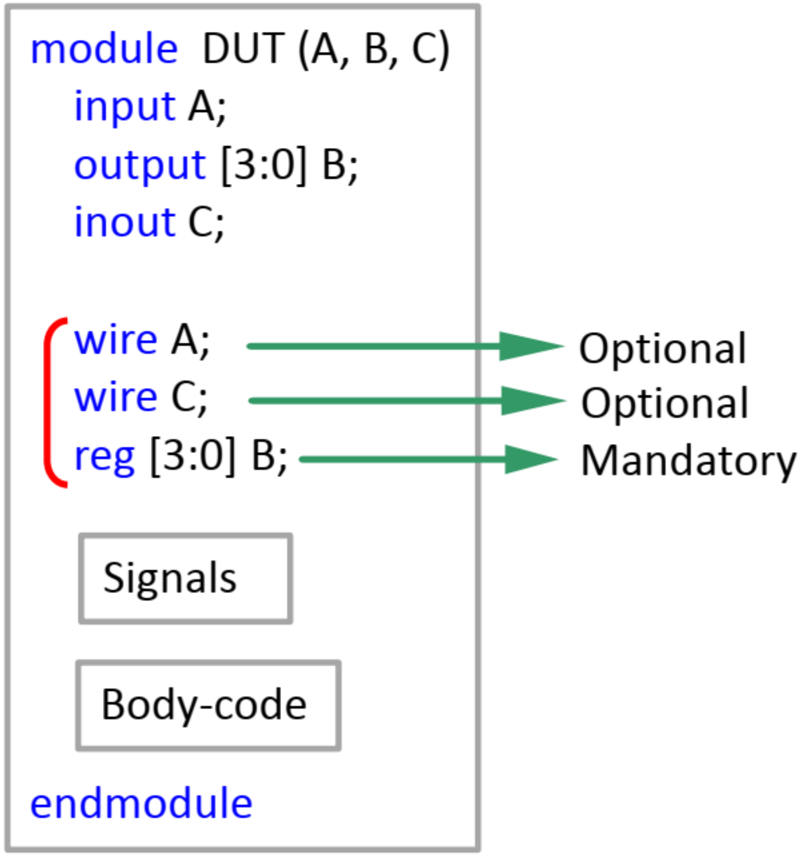
Port Type

- All ports are wire by default
- No need to declare it again as wire
- Need to declare it again as **reg**
  - Only valid for output ports



```
output [3:0] B;
reg [3:0] B;
```

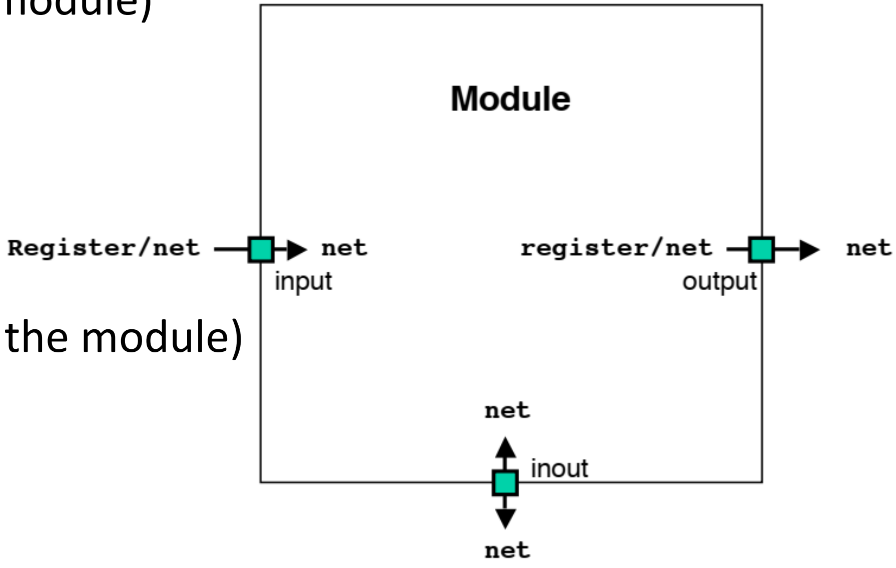
```
output reg [3:0] B;
```



```
module  DUT (A, B, C)
   input A;
   output [3:0] B;
   inout C;

   wire A;
   wire C;
   reg [3:0] B;

   Signals

   Body-code

endmodule
```

Port Parts

- Internally
  - Internal to module (when defining the module)
  - Input port → type ***net***
  - Output port → type ***reg*** or ***net***
  - Inout port → type ***net***
- Externally
  - External to module (when instantiating the module)
  - Input port → type ***reg*** or ***net***
  - Output port → type ***net***
  - Inout port → type ***net***



Port Mapping

- Connecting by order list
  - More intuitive for beginners
  - Few ports

```
module FA(sum, cout, a, b, c_in);
  …
endmodule
```

```
module Top;
        reg [3:0] A, B;
        reg C_IN;
        wire [3:0] SUM;
        wire C_OUT;

// instantiate the design block
FA fa0(SUM, C_OUT, A, B, C_IN);
endmodule
```

Port Mapping

```
module FA(sum, cout, a, b, c_in);
   …
endmodule
```

- Connecting by name
  - More ports
  - Independent from order of ports

```
module Top;
       reg [3:0] A, B;
       reg C_IN;
       wire [3:0] SUM;
       wire C_OUT;

// instantiate the design block
FA fa0(.cout(C_OUT), .sum(SUM), .b(B), .a(A));
endmodule
```

Port Mapping

- Connecting by order list
  - More intuitive for beginners
  - Few ports
- Connecting by name
  - More ports
  - Independent from order of ports

```
module FA(sum, cout, a, b, c_in);
   …
endmodule
```

```
module Top;
       reg [3:0] A, B;
       reg C_IN;
       wire [3:0] SUM;
       wire C_OUT;

// instantiate the design block
FA fa0(SUM, C_OUT, A, B, C_IN);
endmodule
```

```
module Top;
       reg [3:0] A, B;
       reg C_IN;
       wire [3:0] SUM;
       wire C_OUT;

// instantiate the design block
FA fa0(.cout(C_OUT), .sum(SUM), .b(B), .a(A));
endmodule
```

<top-module-name>.<instance-name>.<identifier>

Port Connection Rule

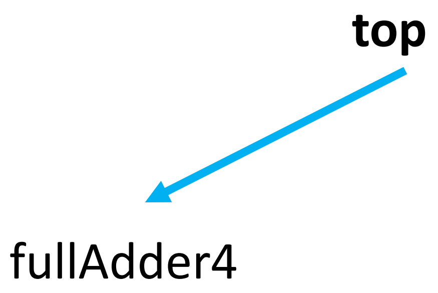
- Width matching
  - Legal to connect items of different sizes
  - Verilog simulator may issue a warning
- Unconnected ports
  - Allowed in Verilog

```
module adder-8bit(input [7:0] a,b, output [7:0] s, output cout);
          wire [6:0] c;
          FA fa0 (a[0], b[0], 1'b0, s[0], c[0]);
           FA fa1 (a[1], b[1], c[0], s[1], c[1]);
           FA fa2 (a[2], b[2], c[1], s[2], c[2]);
           FA fa3 (a[3], b[3], c[2], s[3], c[3]);
           FA fa4 (a[4], b[4], c[3], s[4], c[4]);
           FA fa5 (a[5], b[5], c[4], s[5], c[5]);
           FA fa6 (a[6], b[6], c[5], s[6], c[6]);
           FA fa7 (a[7],    , c[6], s[7], cout);
endmodule

module FA (int a, b, cin, output s, cout);
        wire t1, t2, t3;
        xor (t1, a, b);
        xor (s, t1, cin);
        and (t2, a, b);
        and (t3, t1, cin);
        or (cout, t2, t3);
endmodule
```

Hierarchical Identifier

- Every signal, variable, or module instance has a unique identifier
- Same identifier at different hierarchy levels
- Unique identifier at each hierarchy levels

<top-module-name>.<instance-name>.<identifier>



Hierarchical Identifier: Example

```
module fullAdder4(output reg [3:0] sum,
                  output reg c_out,
                  input [3:0] a, b,
                  input c_in);

   …
endmodule
```

```
module top;
     reg [3:0] A, B;
     reg c_in;
     reg [3:0] sum;
     wire c_out;
//instantiate the design block
fullAdder4 fa0(sum, c_out, a, b, c_in);
… .
endmodule
```

$$\begin{tikzcd} \textbf{top} \arrow[d] \\ \text{fullAdder4} \end{tikzcd}$$

top.sum

top.fa0.sum

Are These Codes Correct?

```
module fullAdder4(output reg [3:0] sum,
                  output reg c_out,
                  input [3:0] a, b,
                  input c_in);
  ...
  <module internals>
  ...
endmodule
```

```
module top;
    reg [3:0] A, B;
    reg c_in;
    reg [3:0] sum;
    wire c_out;

  //instantiate the design block
  fullAdder4 fa0(sum, c_out, a, b, c_in);

  … .
endmodule
```

Are These Codes Correct?
Illegal

```
module fullAdder4(output reg [3:0] sum,
                  output reg c_out,
                  input [3:0] a, b,
                  input c_in);
  ...
  <module internals>
  ...
endmodule
```

```
module top;
    reg [3:0] A, B;
    reg c_in;
    reg [3:0] sum;
    wire c_out;

  //instantiate the design block
  fullAdder4 fa0(sum, c_out, a, b, c_in);

  .... 
endmodule
```

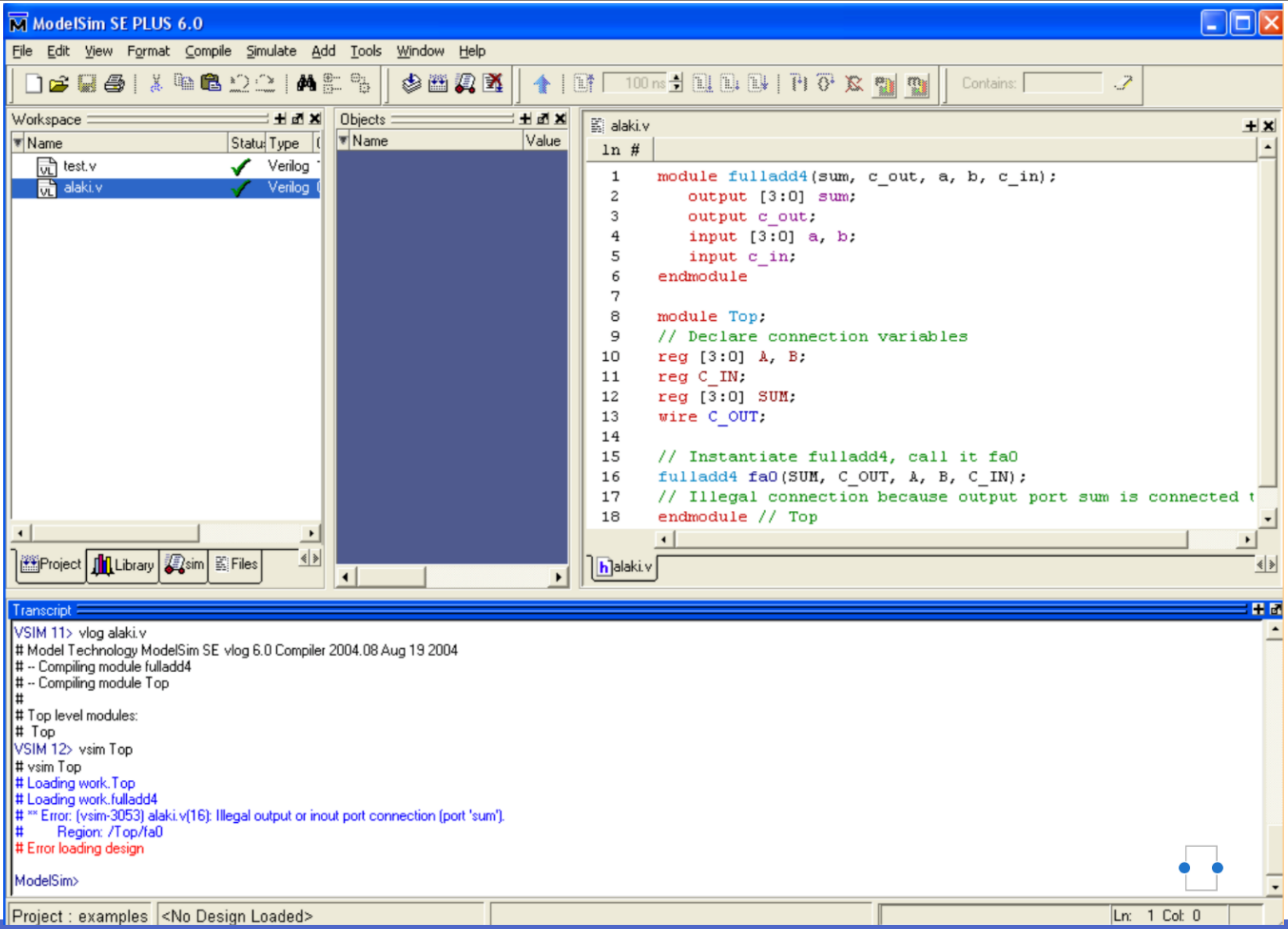
Are These Codes Correct?

Illegal

```
module fullAdder4(output reg [3:0] sum,
                  output reg c_out,
                  input [3:0] a, b,
                  input c_in);

  …
  <module internals>
  …
endmodule
```

```
module top;
    reg [3:0] A, B;
    reg c_in;
    wire [3:0] sum;
    wire c_out;

  //instantiate the design block
  fullAdder4 fa0(sum, c_out, A, B, c_in);

  … .
endmodule
```

Port Connection Rule (cont'd)



ModelSim SE PLUS 6.0
File Edit View Format Compile Simulate Add Tools Window Help
Workspace
Name Status Type
test.v Verilog
alaki.v Verilog
Objects
Name Value
alaki.v
Project Library sim Files
Project : examples <No Design Loaded>
Ln: 1 Col: 0

```
module fulladd4(sum, c_out, a, b, c_in);
   output [3:0] sum;
   output c_out;
   input [3:0] a, b;
   input c_in;
endmodule

module Top;
// Declare connection variables
reg [3:0] A, B;
reg C_IN;
reg [3:0] SUM;
wire C_OUT;

// Instantiate fulladd4, call it fa0
fulladd4 fa0(SUM, C_OUT, A, B, C_IN);
// Illegal connection because output port sum is connected t
endmodule // Top
```

```
Transcript
VSIM 11> vlog alaki.v
# Model Technology ModelSim SE vlog 6.0 Compiler 2004.08 Aug 19 2004
# -- Compiling module fulladd4
# -- Compiling module Top
#
# Top level modules:
#  Top
VSIM 12> vsim Top
# vsim Top
# Loading work.Top
# Loading work.fulladd4
# ** Error: (vsim-3053) alaki.v(16): Illegal output or inout port connection (port 'sum').
#        Region: /Top/fa0
# Error loading design

ModelSim>
```

Parameters

Writing More Reusable Verilog Code

- We have a module that can compare two 4-bit numbers

- What if in the overall design we need to compare:
  - **5**-bit numbers?
  - **6**-bit numbers?
  - …
  - **N**-bit numbers?
  - Writing code for each case looks tedious

- What could be a better way?

Parameterized Modules

- Define module parameters

```
module mux2 #(parameter width = 8)  // name and default value
            (input  [width-1:0] d0, d1,
             input              s,
             output [width-1:0] y);

             y = s ? d1 : d0;
endmodule
```

Parameters

- Used as a “constant” to facilitate coding
- Can be overridden for each module at compile-time
- Can be changed from outside of the module
- Provide flexible hardware programming
- Syntax:
  - **parameter** “param” = “value”;

```
parameter  port_id = 5;
parameter  cache_line_width = 256;
parameter  bus_width = 8;

wire         [bus_width – 1: 0] bus;
```

```
module  DUT (s, Out)

  parameter n = 3;
  parameter S0 = 4’b1010;

  input [n-1:0] s;
  output [n:0] Out;

  wire [n:0] Out;
  assign Out = S0;

endmodule
```

Example

```
module test-param();
        parameter id_num = 0;
       …
endmodule


module top();

   test-param   tp1();
   test-param   tp2();
   test-param   tp2();
   defparam tp1.id_num = 1, tp2.id_num = 2;

endmodule
```

```
module test-param #(parameter id_num = 0)();
        …
endmodule


module top();

     test-param #(1)             tp1();
     test-param #(.id_num (2)) tp2();
     test-param                  tp3();


endmodule
```

Local Parameters

- Defined inside a module
- Cannot be changed from outside of the module
- Syntax:
  - **localparam** “param” = “value”;

```
localparam     state1 = 4’b0001;
               state2 = 4’b0010;
               state3 = 4’b0100;
               state4 = 4’b1000;
```

Operations for HDL Simulation

- Compilation/Parsing
- Elaboration
  - Binding modules to instances
  - Build hierarchy
  - Compute parameter values
  - Establish net connectivity
- Simulation

Generate Block

- Dynamically generate Verilog code at elaboration time
- Variants:
  - Generate loop
  - Generate conditional
  - Generate case
- Generate blocks can be nested!
- Nested loops cannot use the same **genvar** variable

Generate Loop

```
module bitwise_xor (output [N-1:0] out, input [N-1:0] i0, i1);
          parameter N = 32;

          genvar j;// This variable does not exist during simulation

          generate for (j=0; j<N; j=j+1)
               begin: xor_loop
                    xor g1 (out[j], i0[j], i1[j]);
               end
          endgenerate//end of the generate block

endmodule
```

Generate Conditional

```
module multiplier (
                    output [product_width-1:0] product,
                    input [a0_width-1:0] a0,
                    input [a1_width-1:0] a1);


          parameter a0_width = 8;
          parameter a1_width = 8;

          Localparam product_width = a0_width + a1_width;

          generate
             if(a0_width < 8) || (a1_width < 8)
                 multiplier_type1 #(a0_width , a1_width ) m0 (product, a0, a1);
             else
                 multiplier_type2 #(a0_width , a1_width ) m0 (product, a0, a1);
          endgenerate


endmodule
```

Generate Case

```
module adder (output co, output [N-1:0] sum, input [N-1:0] a0, a1, input ci);

         parameter N = 8;

         generate
         case (N)
              1: adder_1bit adder1(c0, sum, a0, a1, ci);
              2: adder_2bit adder2(c0, sum, a0, a1, ci);
              default: adder_cla #(N) adder3(c0, sum, a0, a1, ci);
         endcase
         endgenerate

endmodule
```

Example: Ripple Carry Adder

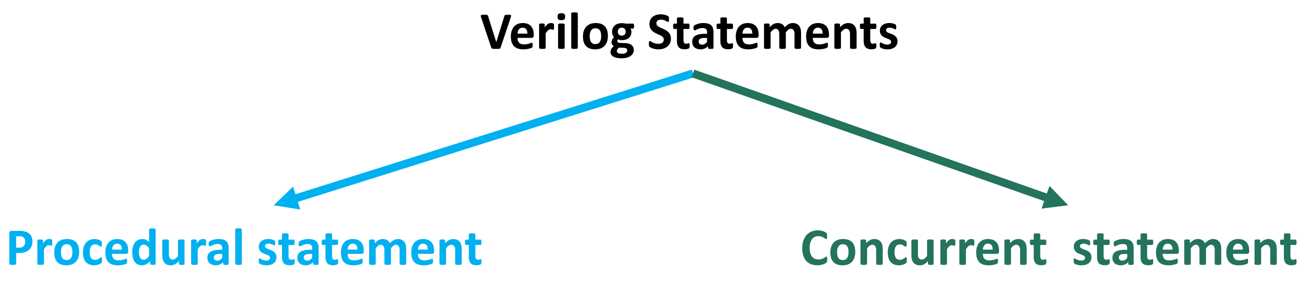
```
module ripple_adder (input [N-1:0] a,b, input cin, output [N-1:0] sum, output cout);

         parameter N = 4;
         wire [N:0] carry;

         buf b1 (carry[0], cin);
         buf b2 (cout, carry[N]);

         genvar i;
         generate for (i = 0; i < N; i = i +1)
              begin: r_loop
                  wire t1, t2, t3;
                  xor g1 (t1, a0[i], a1[i]);
                  xor g2 (sum[i], t1, carry[i]);
                  and g3 (t2, a0[i], a1[i]);
                  and g4 (t3, t1, carry[i]);
                  or  g5 (carry[i+1], t2, t3);
              end
         endgenerate
endmodule
```

Body-Code

Body-Code

- Consists of some statements
- Statements describe the circuit/module functionality
- In High-Level Language (HLL) all statement are sequential
  - Statements evaluated in the order and one-by-one
- Verilog Statement?

Verilog statement



Evaluated sequentially
Order IS important

Evaluated in parallel
Order is NOT important

```
always @ (x, y)
  begin
    s = x ^ y;
    c = x & y;
  end
```

```
assign  s = x ^ y;
assign c = x & y;
assign  out = x | y;
```

Thank You