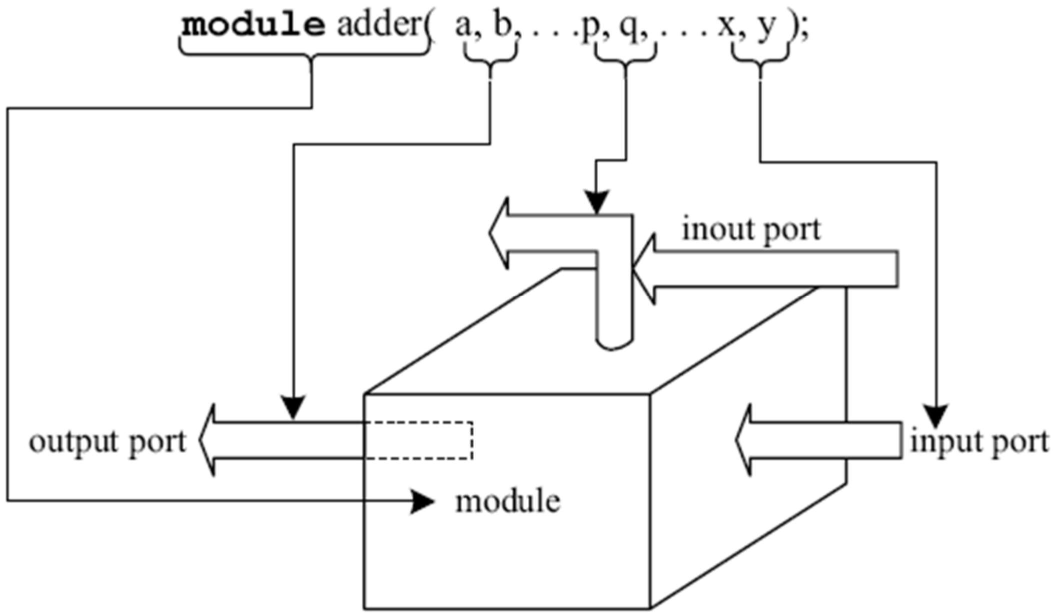
Module, Data Types and Ports

Dr.K.Sivasankaran
Associate Professor
VLSI Division
School of Electronics Engineering
VIT University

module- Definition

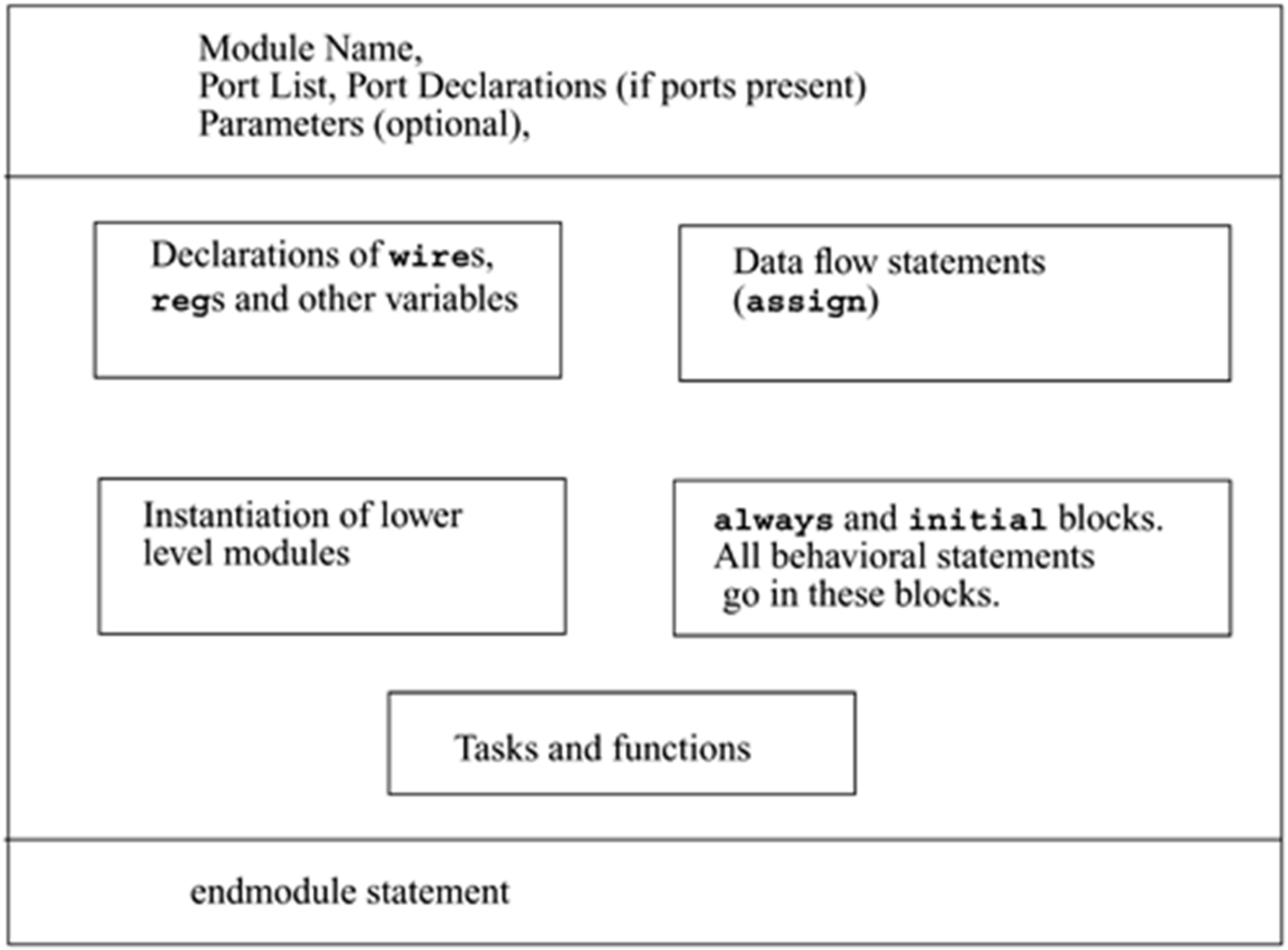
- Any Verilog program begins with a keyword – called a “**module.”**
- **A module is** the name given to any system considering it as a black box with input and output terminals



Concept of a ‘module’

- A module is a basic building block in Verilog and can specify a system of any complexity.
- The module definition is the same for a system of any complexity.
- Provides functionality through its port interface.
- It can be an element or a collection of lower-level design (macro or leaf cells or primitive cells) blocks.

Components of a Verilog Module


Module Name,
Port List, Port Declarations (if ports present)
Parameters (optional),
Declarations of wires, regs and other variables
Data flow statements (assign)
Instantiation of lower level modules
always and initial blocks. All behavioral statements go in these blocks.
Tasks and functions
endmodule statement

Components of Module

```
module <module_name> (<module_port_list>);
 <port type declaration>
 <output port data type declaration>
< parameter declaration(optional)>
  //module's body//
< data flow level modeling>
< Behavior level modeling>
< Structural / Gate level modeling>
< Switch level modeling>
<task and functions>
endmodule
```

How to declare a module ?

- A module in Verilog is declared using the keyword module and a corresponding keyword endmodule must appear at the end of the module.
- Each module must have a module name, which acts as an identifier.
- A module can have an optional port list which describes the input, output & inout terminals of the module.

Nesting of modules

In Verilog nesting of modules is not permitted i.e., one module definition cannot contain another module definition within the module and endmodule statements.

Example:

```
module counter(q, clk, reset);
output [3:0]q;
input clk, reset;

   module T_FF(q, clock, reset)   // Illegal
   .
      endmodule
endmodule
```

Module declaration - Examples

```
module ex2(y,a,b,c,d);
output y;
input a,b,c,d;
wire f1,f2;
or o1(f1,a,b)
and a1(f2,c,d);
xor x1(y,f1,f2);
endmodule
```

Module name: ex2

No. of ports: 5

Structural Modeling

```
module <module_name>
(<module_port_list>);
<port type declaration>
<output port data type declaration>
< parameter declaration(optional)>
  //module's body//
< Structural / Gate level modeling>

endmodule
```

```
module half_adder (a,b,s,c);
 output  s,c;
 input    a,b;
wire  s,c;
xor  (s,a,b);
and (c,a,b);
endmodule
```

Data Flow level Modeling

```
module <module_name>
(<module_port_list>);
<port type declaration>
<output port data type
declaration>
< parameter
declaration(optional)>
  //module's body//
< data flow level modeling>

endmodule
```

```
module half_adder (a,b,s,c);
 output   s,c;
 input    a,b;
wire  s,c;
assign  s=a ^b;
assign c= a & b;
endmodule
```

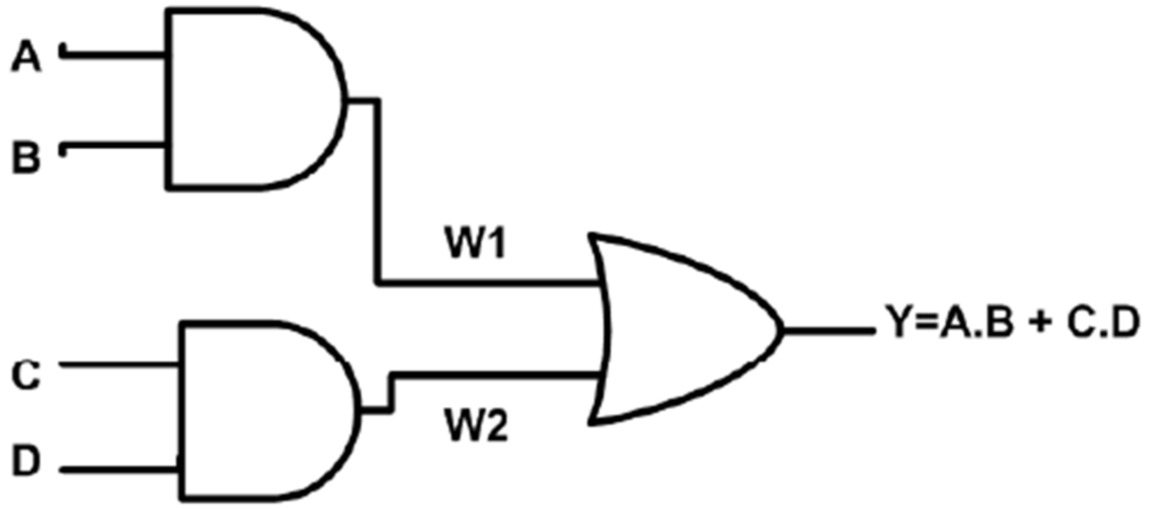
Behavioral Modeling

```
module <module_name>
(<module_port_list>);
<port type declaration>
<output port data type declaration>
< parameter declaration(optional)>
  //module's body//

< Behavior level modeling>
endmodule
```

```
module register  (a,b,s,c);
 output   s,c;
 input     a,b;
reg  s,c;
always @(a,b)
begin
if (a==0 && b==0)
begin
s=1'b0;
c=1'b0;
end
else if (a==1 && b==0 || a==0 && b==1)
begin
s=1'b1;
c=1'b0;
end
else
s=1'b0;
c=1'b1;
endmodule
```

Specification (Logic Diagram)



Structural Modeling

```
module <module_name>
(<module_port_list>);
<port type declaration>
<output port data type declaration>
< parameter declaration(optional)>

 //module's body//
  < Structural / Gate level modeling>

endmodule
```

```
module example (a,b,c,d);
input a,b,c,d;
output y;
 wire y, w1,w2;
 and (w1,a,b);
  and (w2,c,d);
   or (y, w1,w2);
endmodule
```

Specification (Boolean Equation)

$$Y = (A.B) + (C.D)$$

Data Flow level Modeling

```
    module <module_name>
    (<module_port_list>);
    <port type declaration>
    <output port data type
    declaration>
    < parameter
    declaration(optional)>
     //module's body//
   < data flow level modeling>
endmodule
```

```
module example (a,b,c,d,y);
 input    a,b,c,d;
 output   y;
 wire  y;
assign  y= (a & b) |( b & d);
endmodule
```

Specification (Truth Table)

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Behavioral Modeling

```
module <module_name>
(<module_port_list>);
<port type declaration>
<output port data type declaration>
< parameter declaration(optional)>

  //module's body//
< Behavior level modeling>
endmodule
```

```
module example (a,b,y);
 input  a,b;
 output y;
reg y;
always @(a,b)
begin
if (a==1'b1 && b==1'b1)
   y=1'b1;
  else
   y=1'b0;
end
endmodule
```

Data Types

Data Types

- net
- register
- Integer

Data Types

A variable belongs to one of two data types

Net

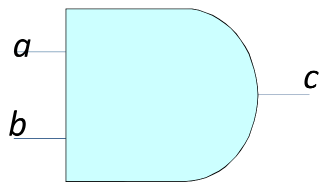
- Must be continuously driven
- Used to model connection between continuous assignments and Instantiations.

Register

- Retains the last value assigned to it.
- Often used to represent storage element.

Net Data Type

- Nets represent connections / physical wires between hardware elements.
- Nets will not store / hold any value.
- Different net types supported for synthesis:
  - **wire , wor, wand, tri , supply0, supply1**
  - **wire** and **tri** are equivalent ; when there are multiple drivers, driving them, the output of the drivers are shorted together.
  - **wor** / **wand** inserts an OR/AND gate at the connection.
  - **supply0** / **supply1** model power supply connections.
- Default Size : 1-bit / scalar
- Default Value : z



Net *c* continuously assumes the value computed at the output of the AND gate

Wire declaration - Examples

- **wire** a; // signal ‘a’ declared as wire
- **wire** out; // signal ‘out’ declared as wire

  Example: **assign** out = a | b;
  or o1(out, a, b);

- **wire** a, b; // signals ‘a’ & ‘b’ declared as wires
- **wire** d = 1’b0; /*net ‘b’ is fixed to logic value ‘0’ at declaration*/

Example for wire and wand – Net data type

```
module wired (a,b,f);
input a,b;
output f;
wire f;
assign f= a & b;
assign f= a | b;
endmodule
```

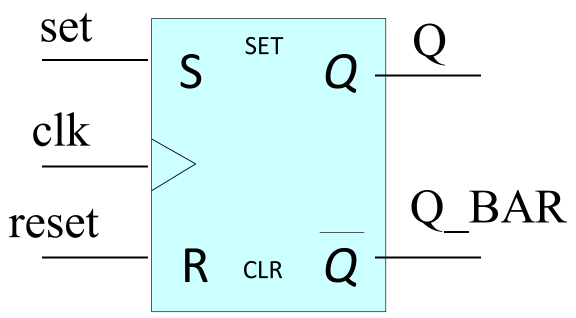
```
module wired_a (a,b,f);
input a,b;
output f;
wand f;
assign f= a & b;
assign f= a | b;
endmodule
```

Example for supply – Net data type

```
module supply_wire (a,b,c,f);
input a,b,c;
output f;
wire  f;
wire t1,t2;
supply0 gnd;
supply1 vdd;
nand G1(t1,vdd,a,b);
xor G2(t2,c,gnd);
and G3(f,t1,t2);
endmodule
```

Register Data Types (1/3)

- ✓ In Verilog registers represent data storage elements.
- ✓ Used to model hardware memory elements / registers.
- ✓ Registers can hold / store a value.
- ✓ Declared by the keyword **reg , integer**
- ✓ Default Size : 1-bit / scalar
- ✓ Default Value : x



Register Data Type (2/3)

- Different "register" types supported for synthesis
  - reg,integer
- ➢ The "reg" declaration explicitly specifies the size
  - reg x,y //single-bit register variable
  - reg [15:0] bus; // 16-bit bus
- ➢ For "integer", it takes the default size, usually 32-bits
- ➢ Synthesizer tries to determine the size

Registers Data Types (3/3)

In arithmetic expression,

- ➢ An integer is treated as a 2’s complement signed integer
- ➢ A reg is treated as an unsigned quantity

General Rule

- ➢ “**reg**” is used to model actual hardware registers such as counters, accumulators etc.,
- ➢ “**integer**” is used for situation like loop counting

- ❖ The reg declaration explicitly specifies the size either in scalar or vector quantity.
- ❖ For integer it takes default size, usually 32-bits

integers

- When 'integer' is used, the synthesis system often carries out a data flow analysis of the model to determine its actual size.

  Example:

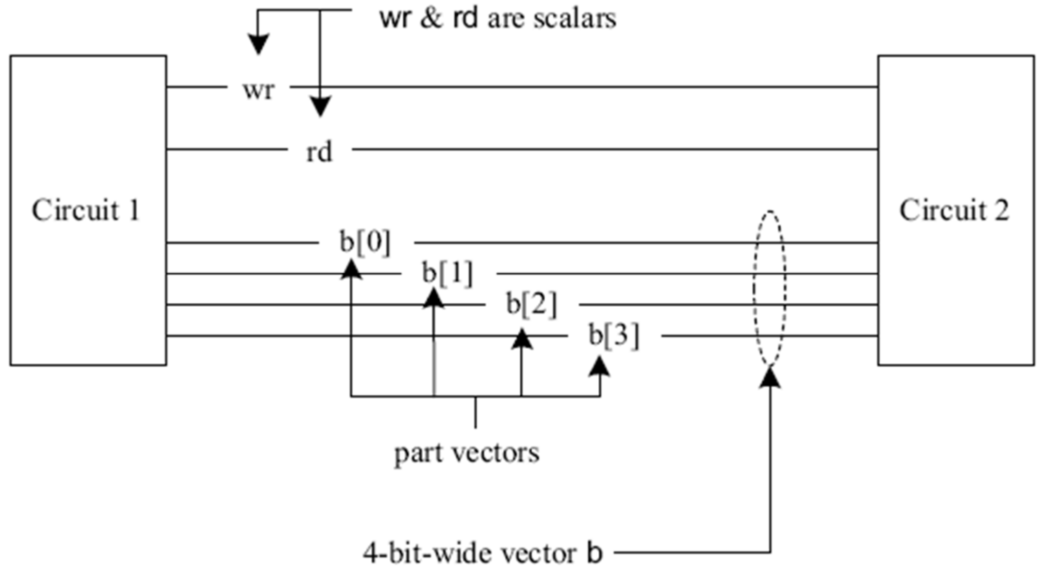
  ```
  wire [0:9] A,B;
  integer C;
  C=A+B;
  ```

- The size of C can be determined to be equal to 11 (10 plus a carry)

Scalar and Vectors

Entities representing single bits — whether the bit is stored, changed, or transferred — are called “scalars.”

Similarly, a group of **regs stores a value,** which may be assigned, changed, and handled together. The collection here is treated as a “vector.”



Vectors

Nets or **register** data types can be declared as vectors (more no. of bits).

- If bit width is not specified then the default value is 1-bit (scalar).

```
wire a;                          // default scalar net value
wire [7:0] bus;                  // 8-bit bus
wire [31:0] busA, busB, busC;    //32- bit bus
reg clock;                       // scalar register(default)
reg [0:40] virtual_addr;         //virtual address 41 bits
wire[-2:2]d;                     /*d is a 5-bit vector with individual bits
                                  designated as d[-2],d[-1],d[0],d[1],d[2] */
```

Normally vectors –nets or **regs** are treated as unsigned quantities. They have to be specifically declared as "signed" if so desired.

Example: **wire signed** [4:0] num;

reg signed[3:0] num_1;

Addressing Vectors

```
wire [15:0]busA;
busA[9];  // bit # 9 or 10th bit of vector busA from LSB

 wire [0:15]busB;
busB[9];  // bit # 9 or 7th bit of vector busB from LSB

 reg [31:0]cnt_out;
cnt_out[14:7]; // group of 8 bits of a vector register
cnt_out[7:14]; // is illegal addressing
```

Ports

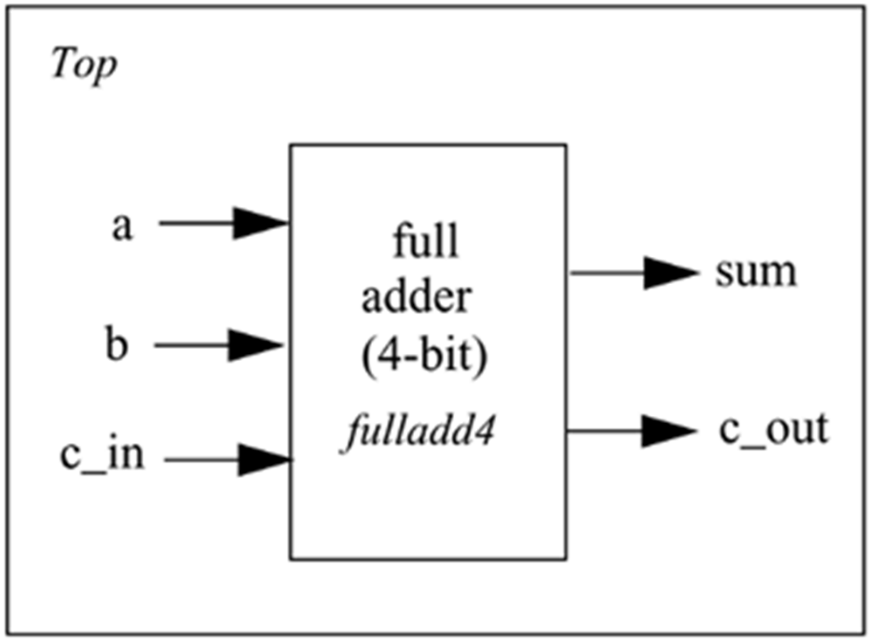
- Ports provide the interface by which a module can communicate with its environment.
- For example, the input/output pins of an IC chip are its ports.
- The environment can interact with the module only through its ports.
- Ports are also referred to as terminals.

Ports

All ports in the list of ports must be declared in the module. Ports can be declared as follows:

Verilog Keyword	Type of Port
input	input port
output	output port
inout	bidirectional

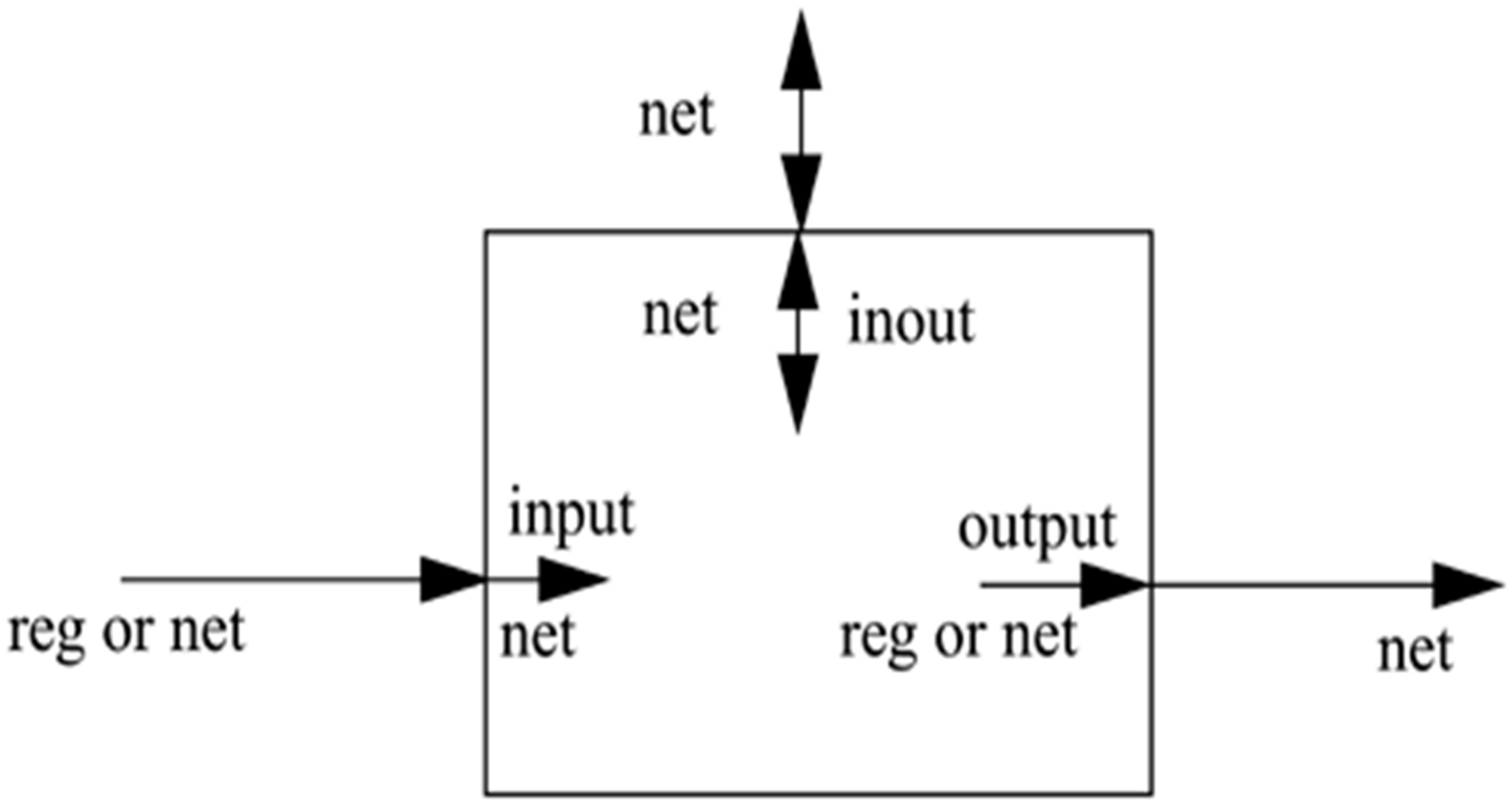
List of Ports



Example

```
module fulladd4(sum, c_out, a, b, c_in); //Module with a list of ports
 module Top; // No list of ports, top-level module in simulation
```

Port Connection Rules



Port Connection Rules

- Ports provide the interface by which a module can communicate with its environment.
- Port declarations for DFF

```
module DFF (dout,din,clk,resetn);
output  dout;
input    din,clk,resetn;

reg dout;  // As the output of D Flip-Flop holds value it is declared as reg

always @(posedge clk or negedge resetn)
if (~resetn) dout<=0;
else dout<=din;
     endmodule
```

- *input* or *inout* ports can not be of type *reg*, because *reg* variables store values; and *input* ports *should only reflect the changes of the external signals* they are connected to.