

*Training Course
on*

*FPGA based Digital
Design using Verilog HDL*



By

NAUMAN MIR
(*HDL Designer*)

*** Organized by Skill Development Council,**
(*Ministry of Labour, Manpower and overseas Pakistani*)
Govt. of Pakistan.



Module

□ The Module Concept

- The module is the basic building block in Verilog
- Modules are:
 - Declared
 - Instantiated
- Modules declarations cannot be nested

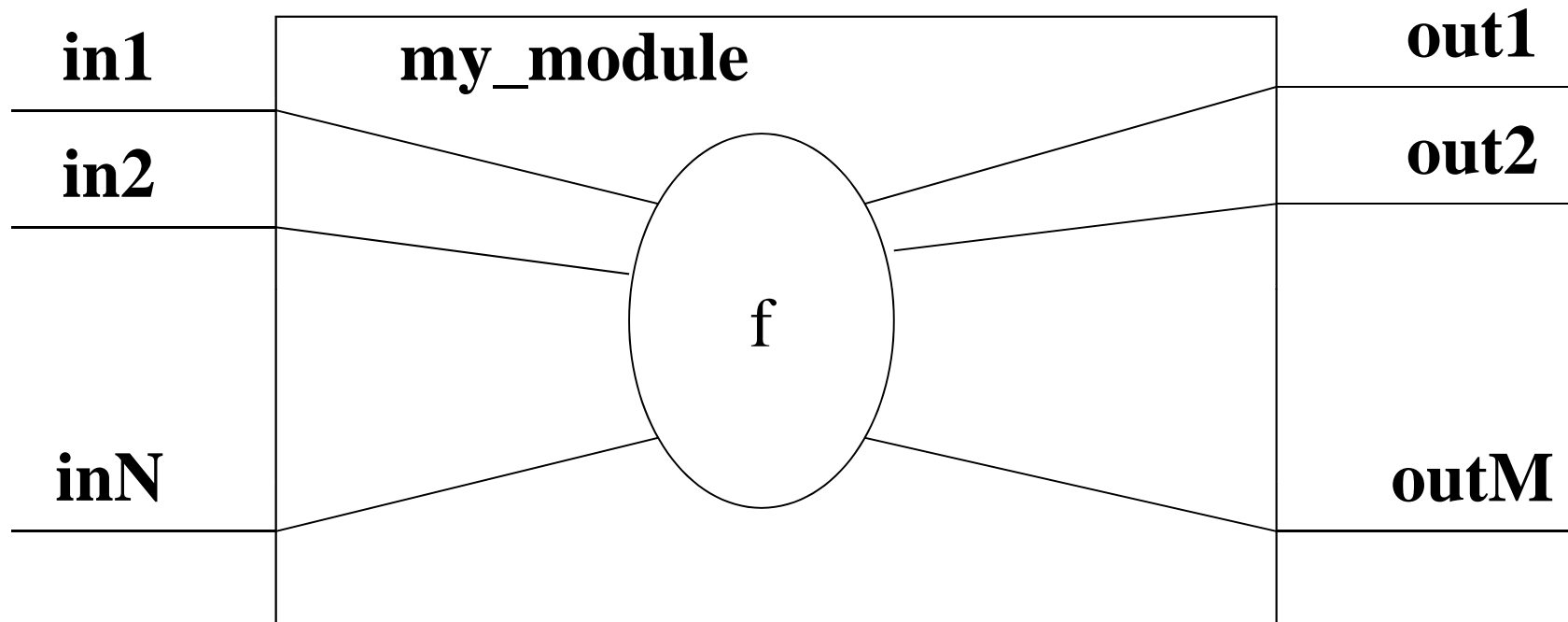


Module (contd)

- Modules can be interconnected to describe the structure of your digital system
- Modules start with keyword **module** and end with keyword **endmodule**
- Modules have ports for interconnection with other modules



Module (contd)



Everything you write in Verilog must be inside a module
exception: compiler directives

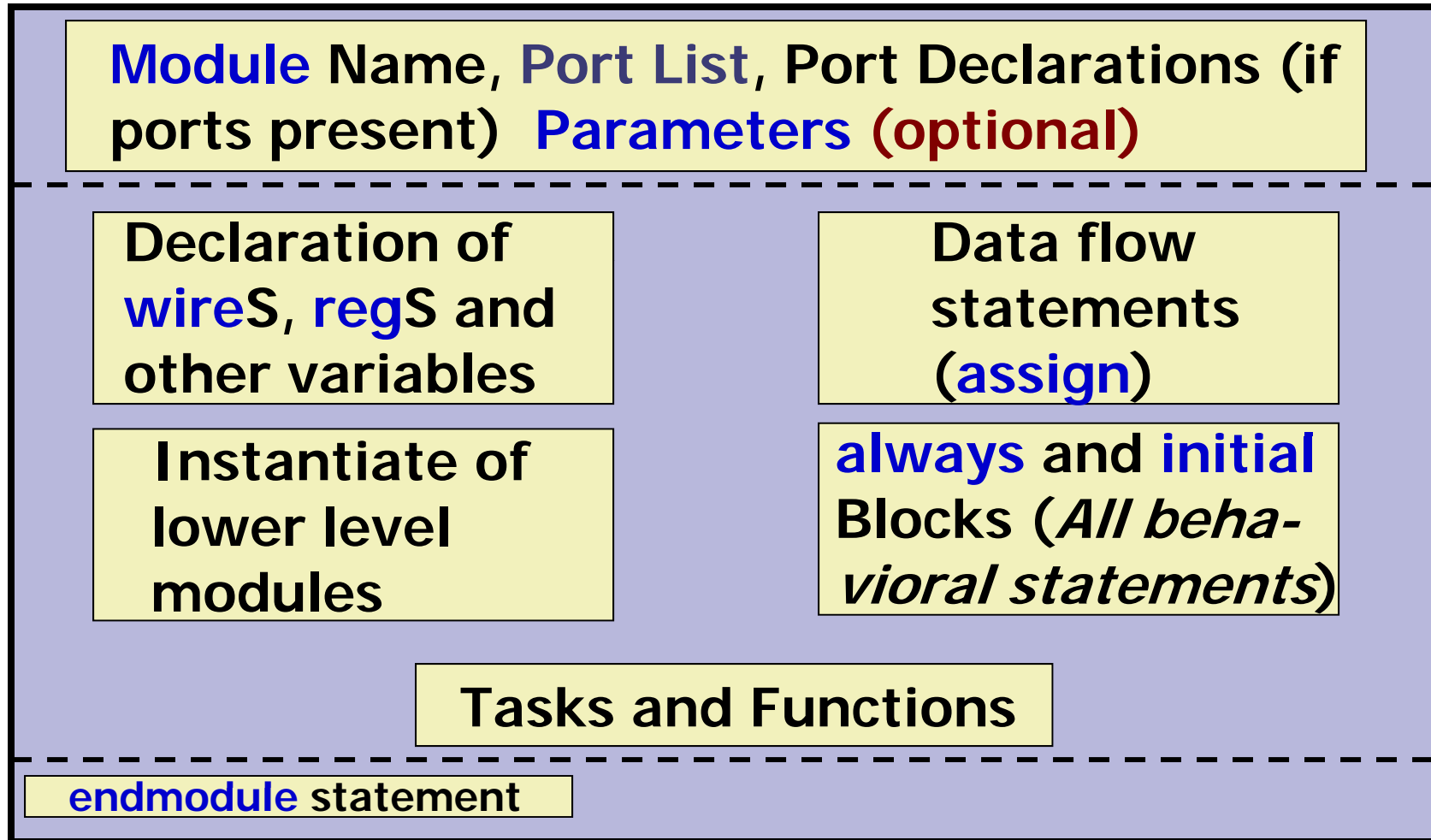


Module (contd)

```
module my_module(out1, .., inN);  
    output out1, .., outM;  
    input in1, .., inN;  
    .. // declarations  
    .. // description of f (maybe  
    .. // sequential)  
endmodule
```



Components of a Verilog Module



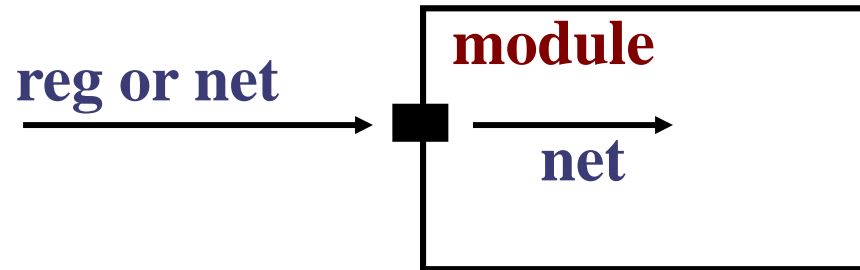


Module Ports

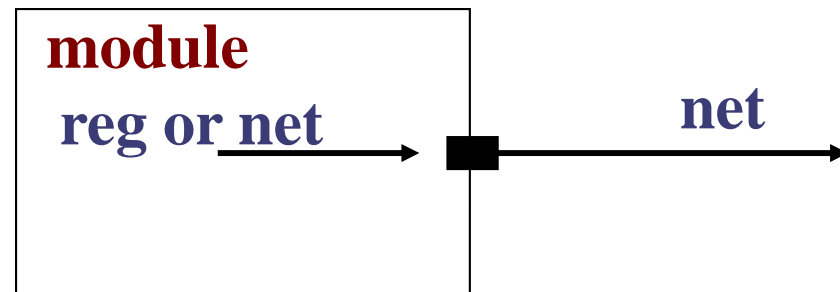
- **Similar to pins on a chip**
- **Provide a way to communicate with outside world**
- **All ports of a module must be declared in the module.**
- **Ports can be input, output or inout**

Port Connection Rules

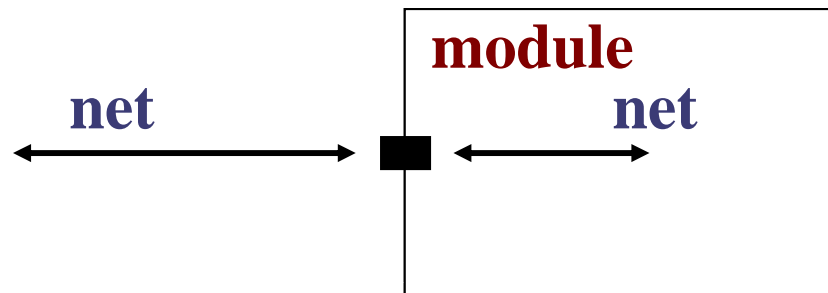
■ Inputs



■ Outputs



■ Inouts





Port Connection Rules (contd)

■ Inputs

Internally, input ports must always be of the type net. Externally, the inputs can be connected to a variable which is a reg or a net.

■ Outputs

Internally, outputs ports can be of the type reg or net. Externally, outputs must always be connected to a net. They cannot be connected to a reg.

■ Inouts

Internally, inout ports must always be of the type net. Externally, inout ports must always be connected to a net.



Port Connection Rules (contd)

■ Width matching

It is legal to connect internal and external items of different sizes when making intermodule port connections. However, a warning is typically issued that the widths do not match.

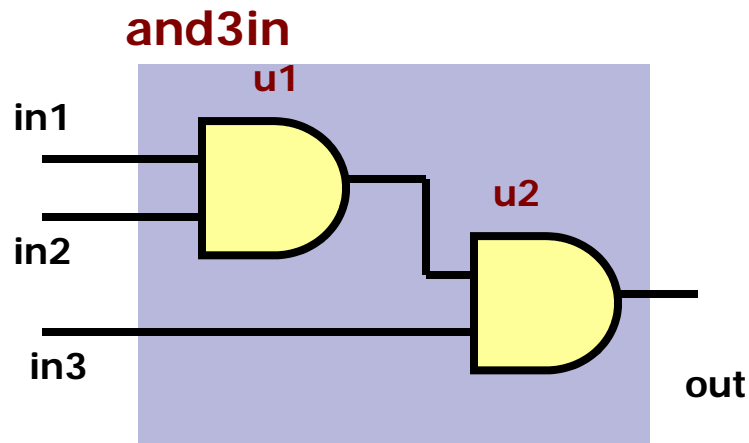
■ Unconnected ports

Verilog allows ports to remain unconnected. For example, certain output ports might be simply for debugging, and you might not be interested in connecting them to the external signals. You can let a port remain unconnected by instantiating a module as shown below.

```
fulladd4 fa0(SUM, , A, B, C_IN); // Output port c_out  
                                is unconnected
```

Module Instances

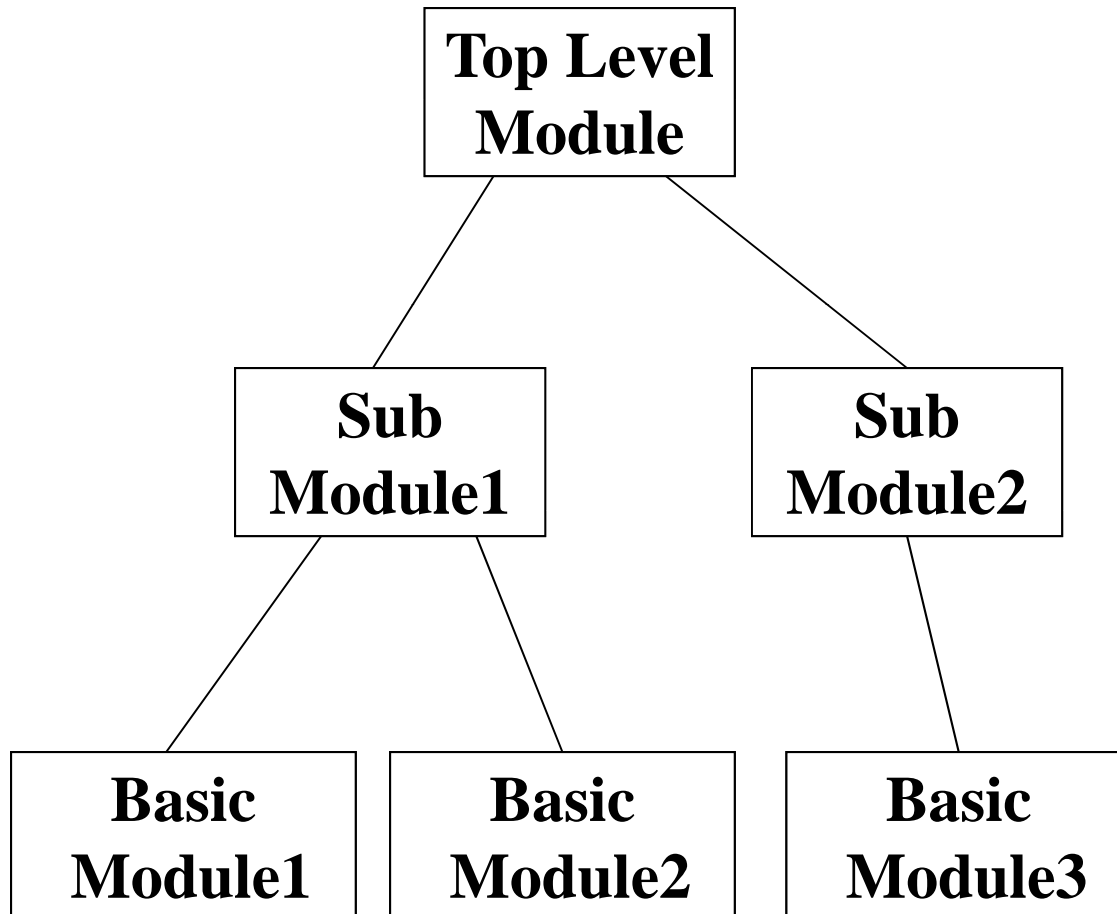
- Verilog models consist of a hierarchy of module *instances*
- In C++ speak: modules are classes and instances are objects



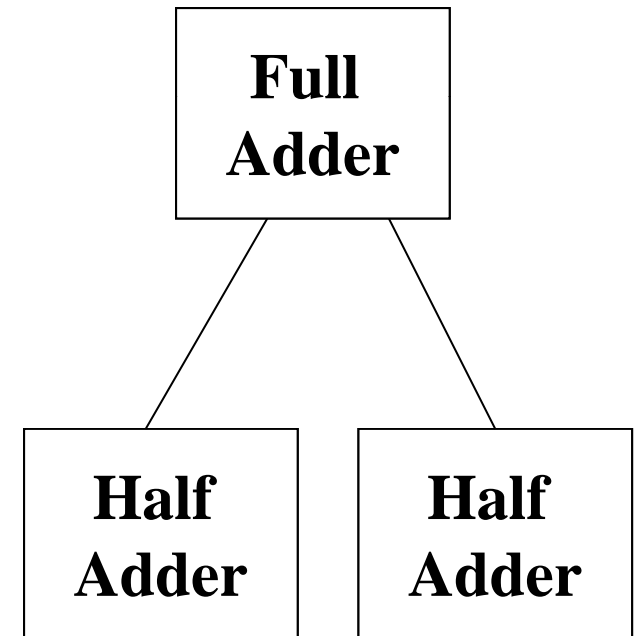
```
module and3in(in1, in2, in3, out);  
  input in1, in2, in3 ;  
  output out ;  
  wire and1_o ;  
  and u1(and1_o, in1, in2) ;  
  and u2(out, in3, and1_o) ;  
endmodule
```



Hierarchical Design



E.g.





Abstraction Levels

There are four levels of abstraction

- **Switch level**
- **Gate level**
- **Dataflow level**
- **Behavioral or algorithmic level**



Modeling Techniques

➤ Switch-Level Modeling

This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches and the interconnections between them. Design at this level requires knowledge of switch-level implementation details. *(It's a rarely use)*

➤ Gate Level Modeling

The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level logic diagram. *It is feasible for small circuits.*



Modeling Techniques (contd)

➤ **Dataflow Modeling**

At this level, the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.

➤ **Behavioral Modeling**

This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.



Gate Level Modeling

➤ Gate Level Modeling

- The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level logic diagram. *It is feasible for small circuits.*
- Verilog supports basic logic gates as predefined primitives. These primitives are instantiated like modules except that they are predefined in Verilog and do not need a module definition.

Gate Level Modeling (contd)

- **Built-in gate primitives:**

`and`, `nand`, `nor`, `or`, `xor`, `xnor`, `buf`,
`not`, `bufif0`, `bufif1`, `notif0`, `notif1`

- **Usage:**

`nand` `n1(out, in1, in2);`

2-input NAND without delay

`and` `#2 u1(out, in1, in2, in3);`

3-input AND with 2 t.u. delay

`not` `#1 N1(out, in);`

NOT with 1 t.u. delay and instance name

`xor` `X1(out, in1, in2);`

2-input XOR with instance name

- **Write them inside module, outside procedures**



Gate Level Modeling (contd)

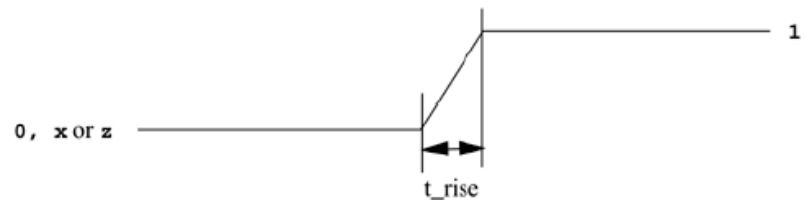
□ Gate Delays

- In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits.
- There are three types of delays from the inputs to the output of a primitive gate.
 - Rise delay
 - Fall delay
 - Turn-off delay

Gate Level Modeling (contd)

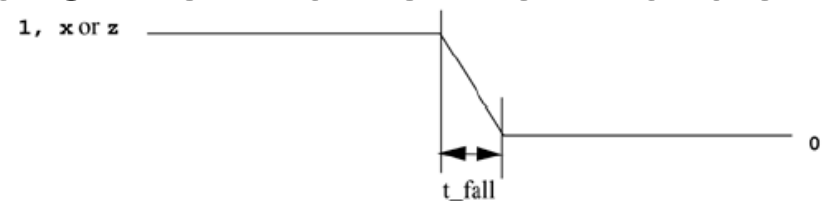
➤ Rise Delay

The rise delay is associated with a gate output transition to a 1 from another value.



➤ Fall Delay

The fall delay is associated with a gate output transition to a 0 from another value.



➤ Turn-off Delay

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value.

Gate Level Modeling (contd)

Example: Types of Delay Specification

```
// Delay of delay_time for all transitions
and #(delay_time) a1(out, i1, i2);

// Rise and Fall Delay Specification.
and #(rise_val, fall_val) a2(out, i1, i2);

// Rise, Fall, and Turn-off Delay Specification
bufif0 #(rise_val, fall_val, turnoff_val)
        b1(out,in,control);
```

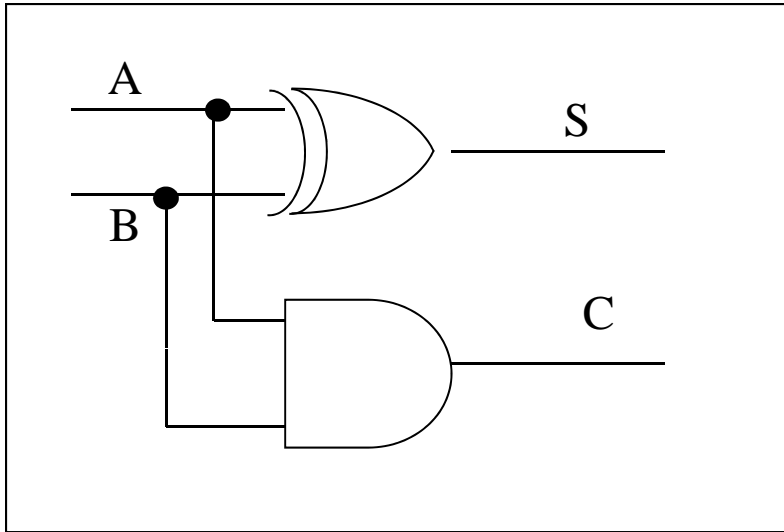
Examples of delay specification are shown below.

```
and #(5) a1(out,i1,i2); // Delay of 5 for all
                        transitions

and #(4,6) a2(out, i1, i2); // Rise = 4, Fall = 6

bufif0 #(3,4,5) b1(out, in, control);
        // Rise = 3, Fall = 4, Turn-off = 5
```

Example: Half Adder



Assuming:

- **XOR: 2 t.u. delay**
- **AND: 1 t.u. delay**

```
module half_adder(S, C, A, B);  
  output S, C;  
  input A, B;  
  
  xor #2 u1(S, A, B);  
  and #1 u2(C, A, B);  
  
endmodule
```



Dataflow Modeling

□ Dataflow Modeling

At this level, the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.

NOTE: In the digital design community, the term **RTL** (*Register Transfer Level*) design is commonly used for a combination of dataflow modeling and behavioral modeling.



Dataflow Modeling (contd)

□ Continuous Assignment

- A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net.
- This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction.
- The assignment statement starts with the keyword assign.
- The syntax of an assign statement is as follows.

Continuous Assignments a closer look

➤ Syntax:

`assign #del <id> = <expr>;`

optional

➤ Where to write them:

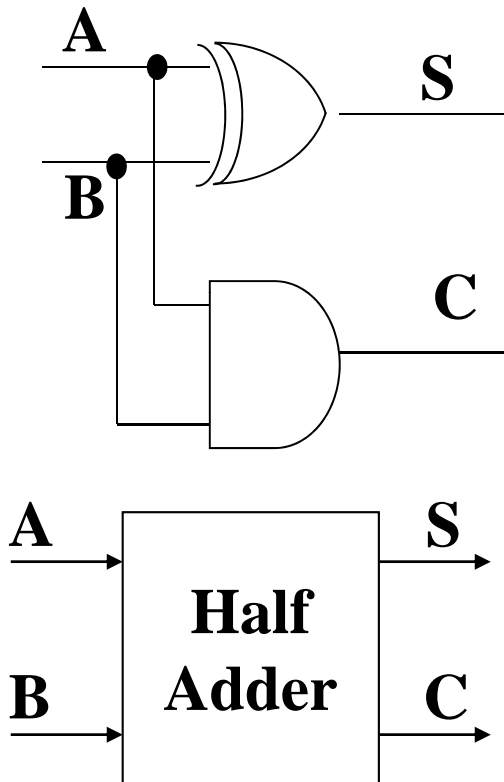
- inside a module
- outside procedures

net type

➤ Properties:

- they all execute in parallel
- are order independent
- are continuously active

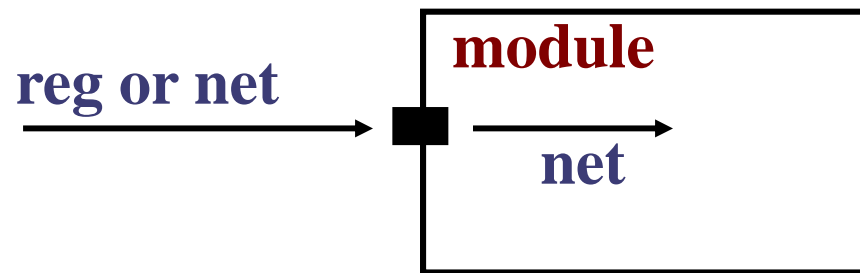
Example: Half Adder



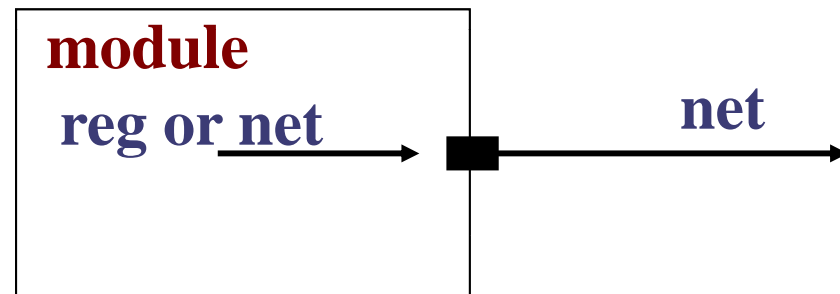
```
module half_adder(S,C,A,B);  
    output S, C;  
    input A, B;  
  
    assign S = A ^ B;  
    assign C = A & B;  
  
endmodule
```

Port Assignments

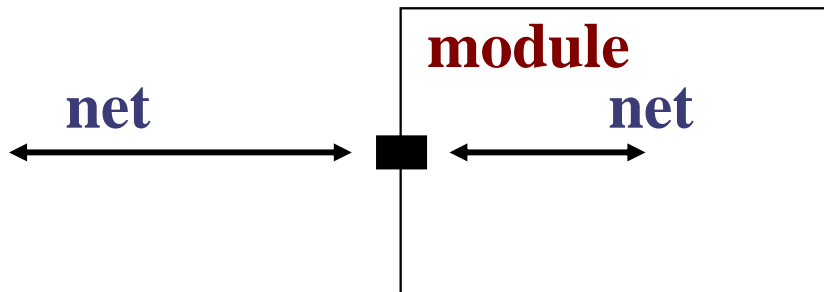
■ Inputs



■ Outputs



■ Inouts





Connecting Ports to External Signals

□ There are two methods of making connections b/w signals specified in the module instantiation and the ports in a module definition.

- **Connecting by ordered list**

The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition.



Connecting Ports to External Signals (contd)

Example:

```
module Top;
```

```
fulladder fa_ordered(SUM, C_OUT, A, B, C_IN);
```

```
endmodule
```

- **Connecting ports by name**

Here you can specify the port connections in any order as long as the port name in the module definition correctly matches the external signal.



Connecting Ports to External Signals (contd)

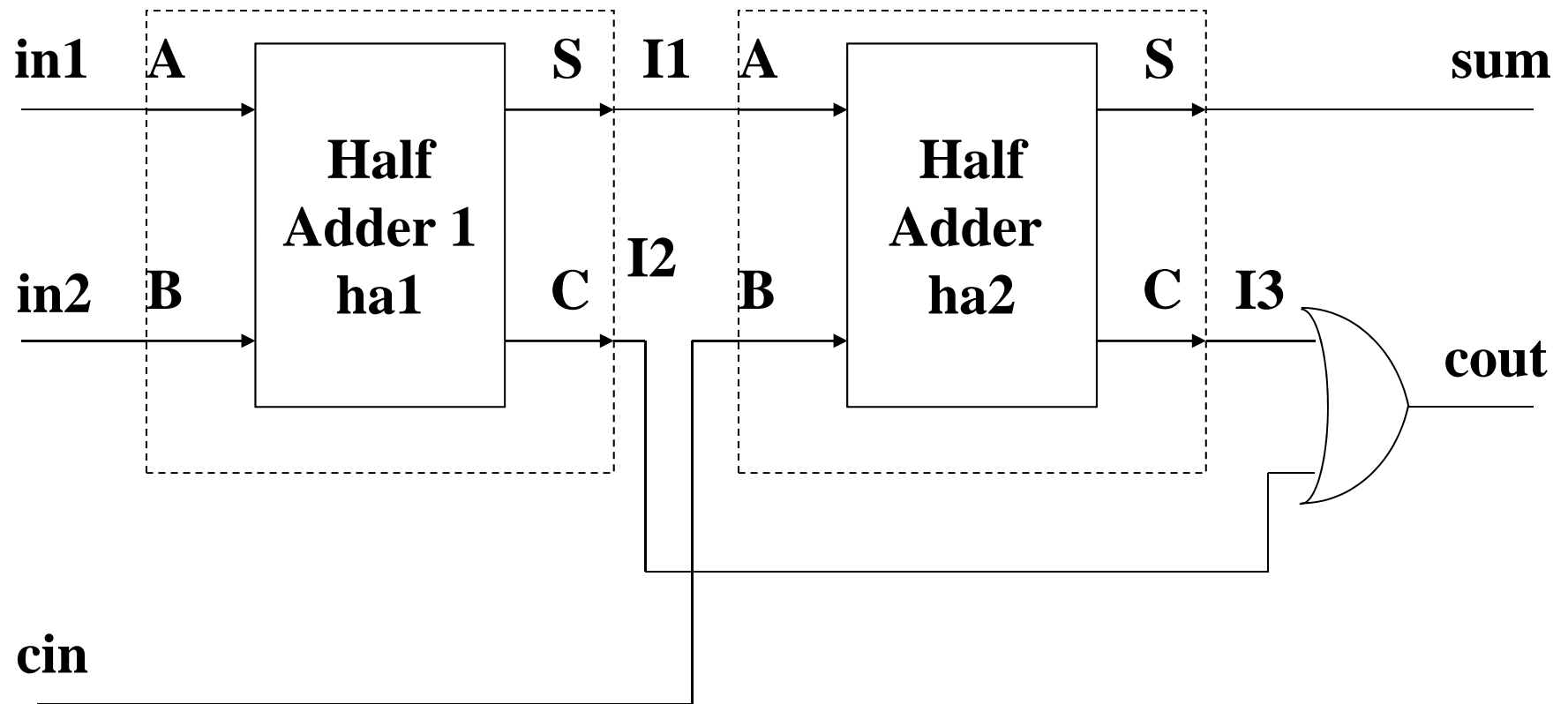
Example:

```
module Top;
```

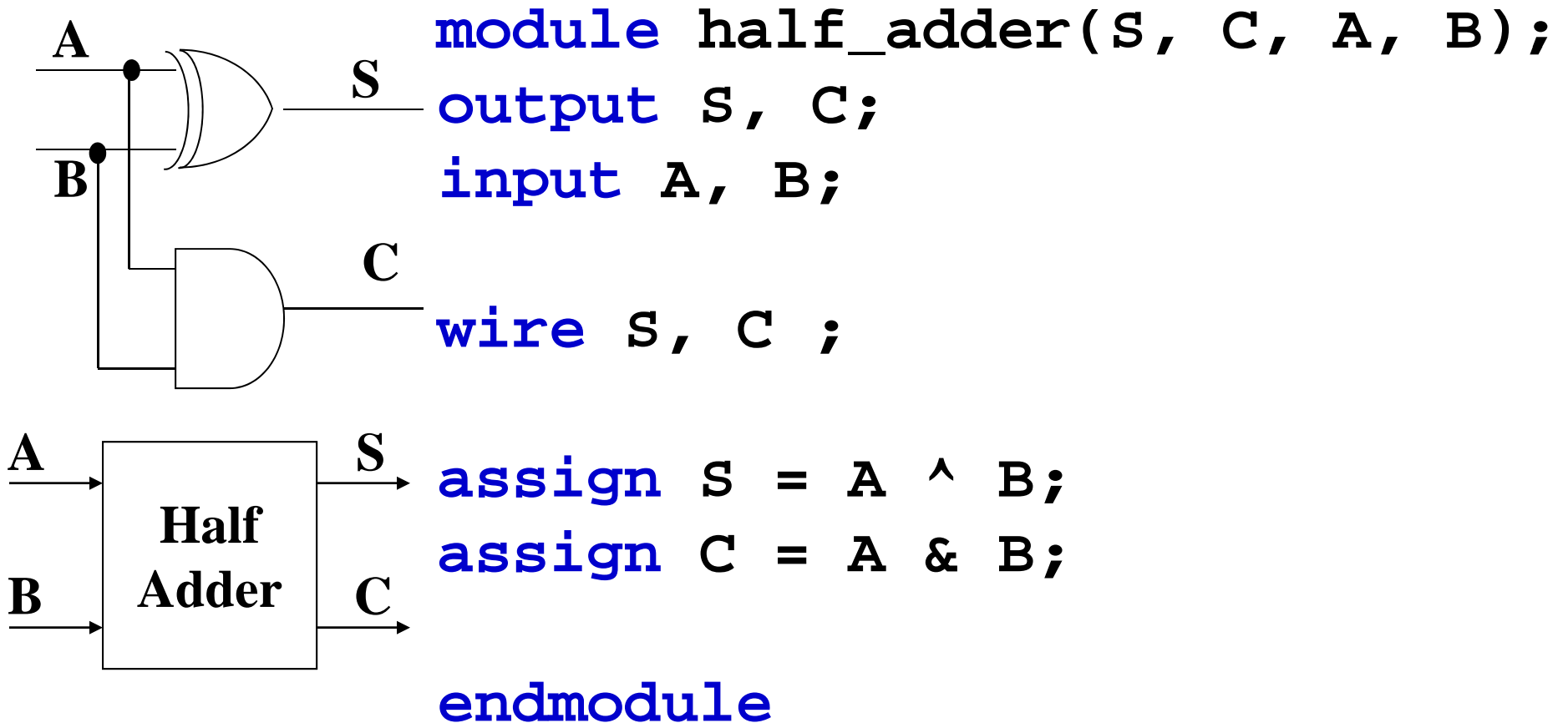
```
    fulladder fa_byname(.c_out    (C_OUT),  
                        .sum      (SUM  ),  
                        .in2      (B    ),  
                        .c_in     (C_IN ),  
                        .in1      (A    )  
    );
```

```
endmodule
```

Example: Full Adder



Full Adder (contd)






Full Adder (contd)

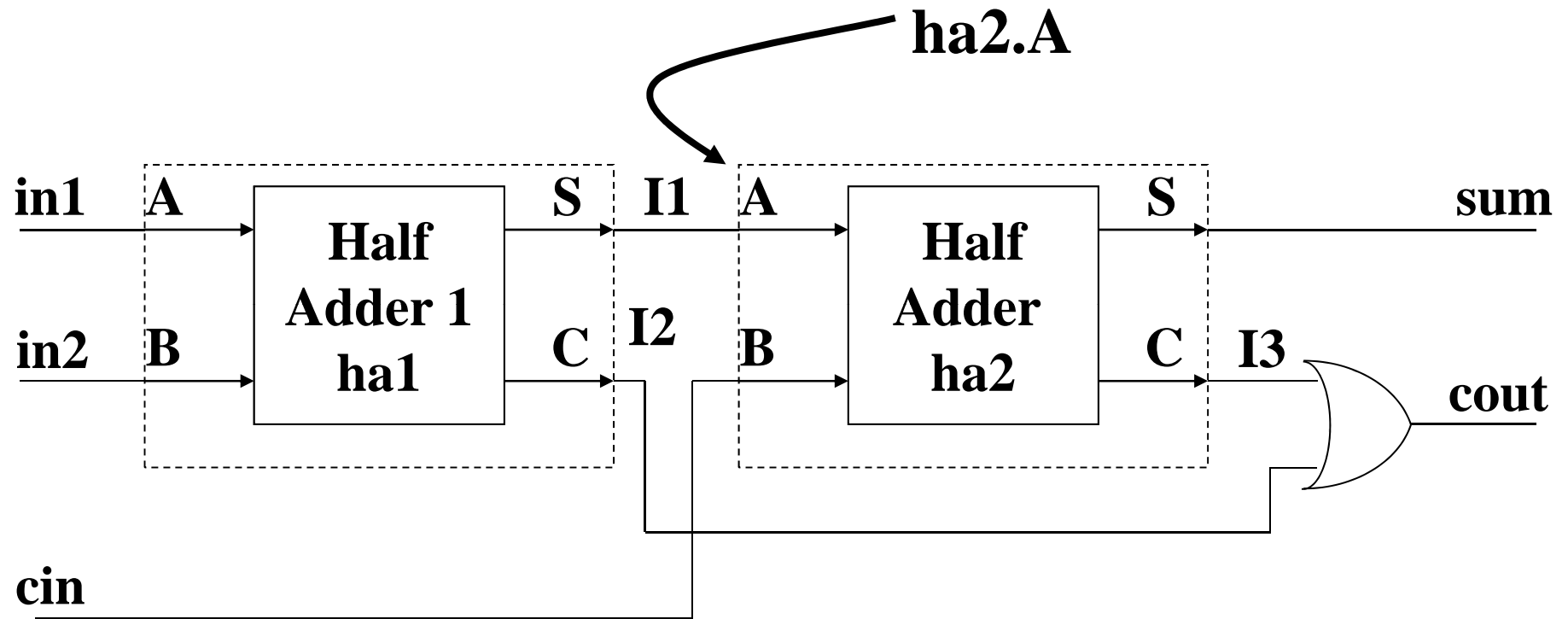
```
module full_adder(sum, cout, in1, in2, cin);  
output sum, cout;  
input in1, in2, cin;
```

```
    wire sum, cout;  
    wire I1, I2, I3;  
Module name  
    half_adder ha1(I1, I2, in1, in2);  
    half_adder ha2(sum, I3, I1, cin);  
    assign cout = I2 | I3;  
  
endmodule
```

Instance name



Hierarchical Names



**Remember to use instance names,
not module names**

2-Bit Full Adder Schematic View

