NATIONAL UNIVERSITY OF HO CHI MINH CITY
UNIVERSITY OF INFORMATION TECHNOLOGY
FACULTY OF COMPUTER ENGINEERING

CIRCUIT DESIGN WITH HDL

CHAPTER 2: VERILOG LANGUAGE CONCEPTS

Ho Ngoc Diem

Content

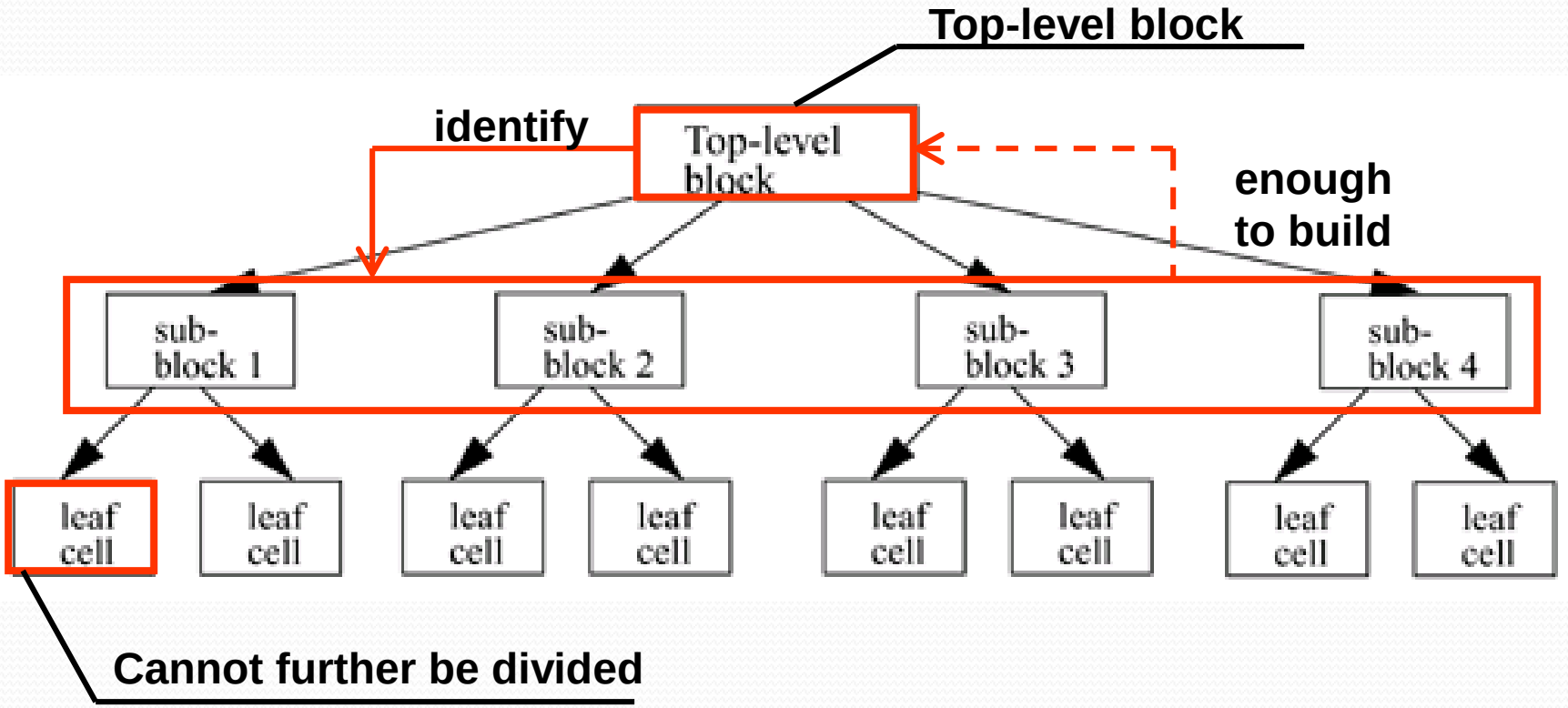
- Hierarchical structure
- Modules
- Instances
- Lexical convention
- Verilog data types

Hierarchical structure

- Modules to be embedded within other modules.
- Higher level modules create instances of lower level modules and communicate with them through input, output, and bidirectional ports.
- Module input/output (I/O) ports can be scalar or vector.

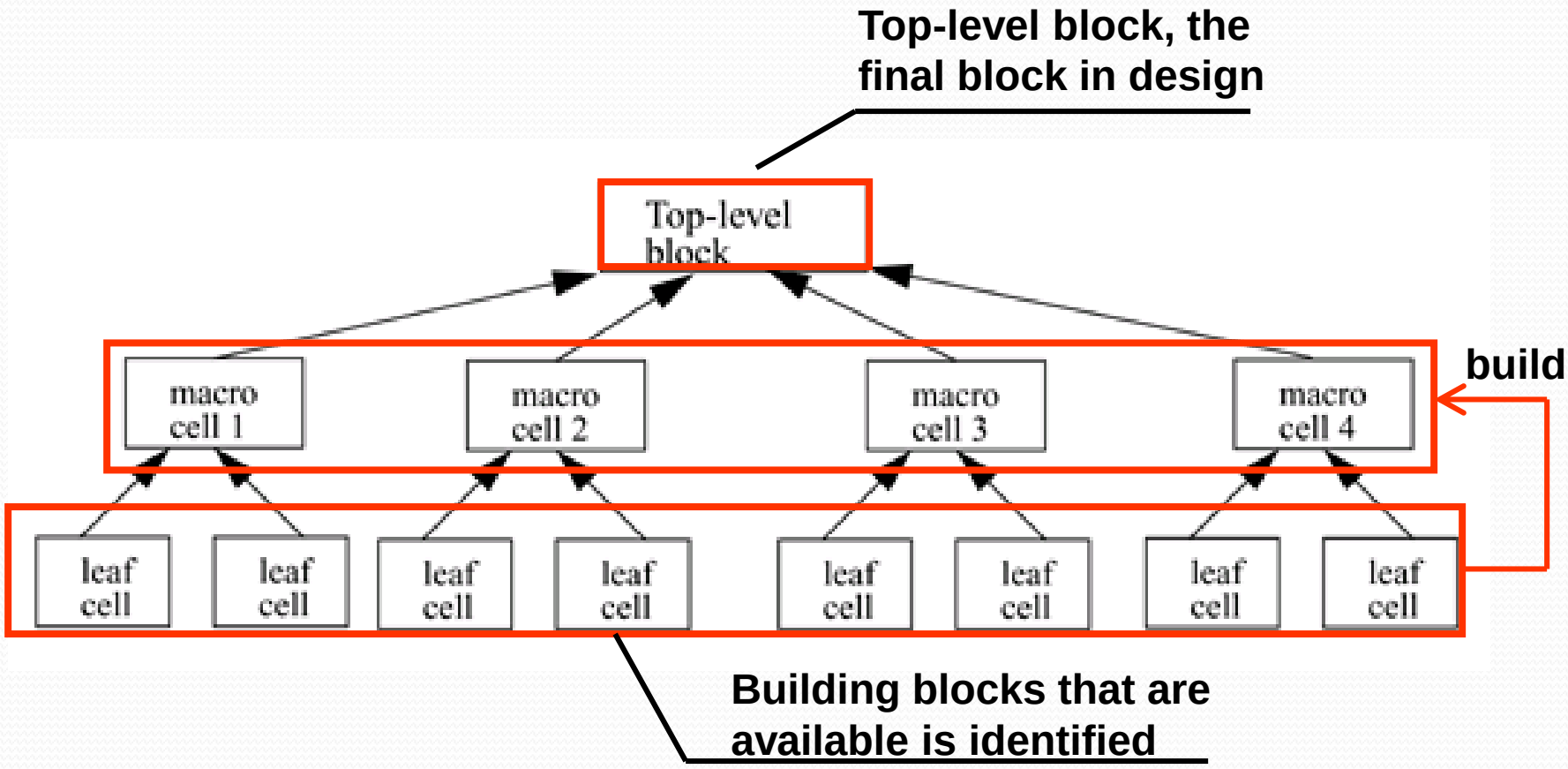
Hierarchical structure

- Top-down design methodology



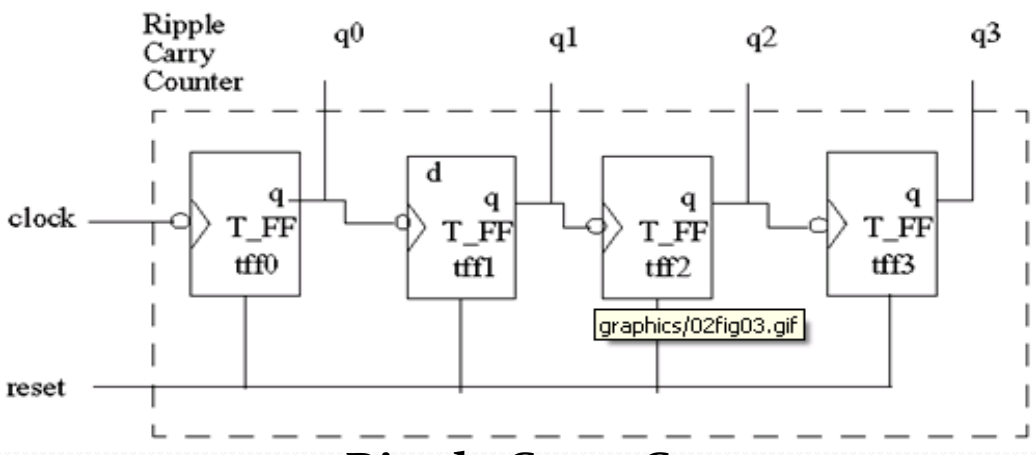
Hierarchical structure

- Bottom-up design methodology

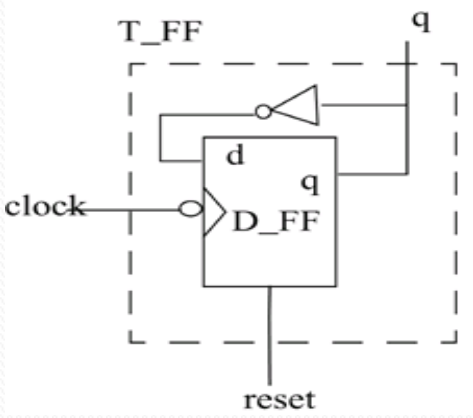


Hierarchical structure

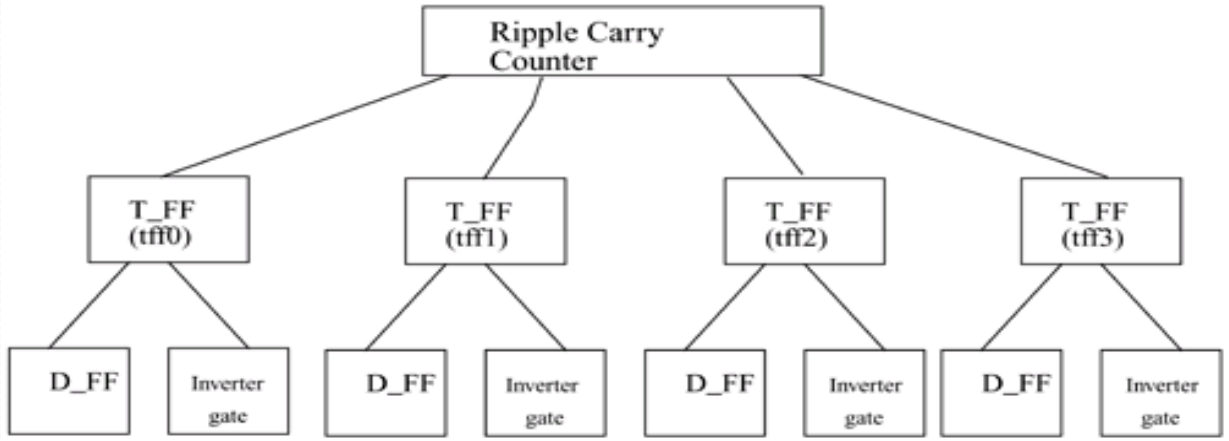
Example: 4-bit Ripple Carry Counter



Ripple Carry Counter



T-flipflop



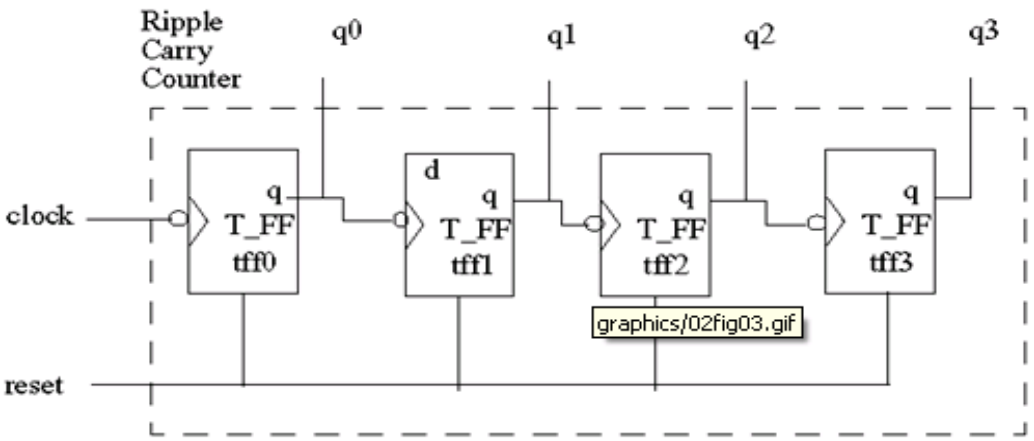
Design Hierarchy

Module

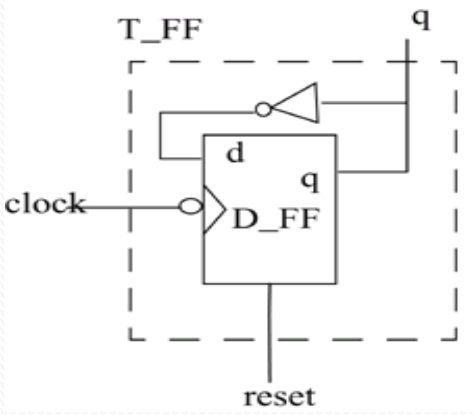
- A **module** is the basic building block in Verilog
  - Can be an element or a collection of lower-level design blocks
  - Provide functionality for higher-level block through its port interface
  - Hide internal implementation
  - Is used at many places in the design
  - Allows designers modify module internals without effecting the rest of design

Module

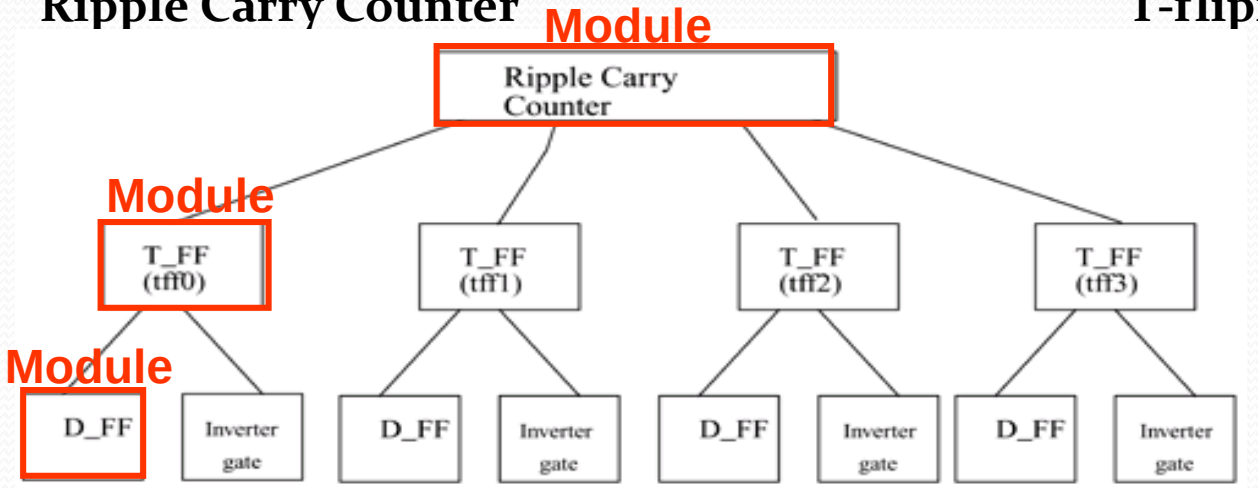
Example: 4-bit Ripple Carry Counter



Ripple Carry Counter



T-flipflop



Design Hierarchy

Module

- Typical Module Components Diagram

Module name, Port list (optional, if there are ports)
Port declaration
Data type declaration (wires, reg, integer etc.)
Instantiation of inner (lower-level) modules
Structural statements (i.e., assign and gates)
Procedural blocks (i.e., always and initial blocks)
Tasks and functions
endmodule declaration

Module

- **Module description**

module *module name* (*port name*, *port name*,…);

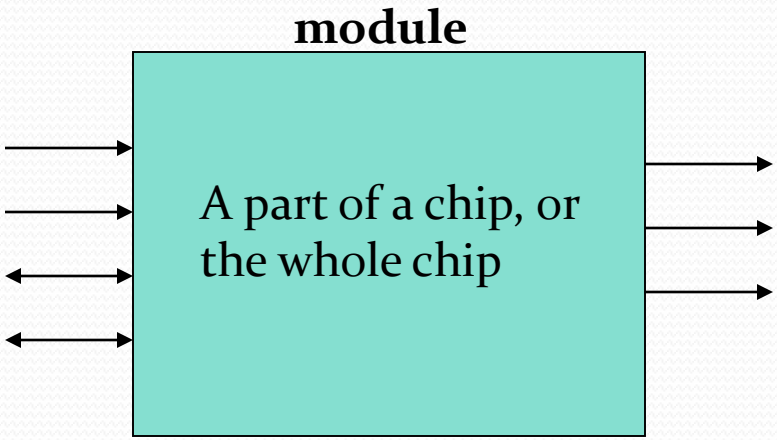
module_port declaration

data type declaration

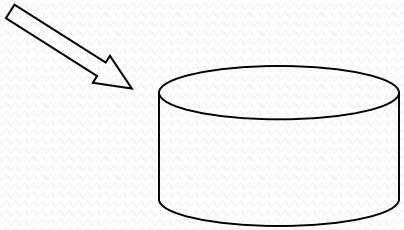
logic description part

endmodule

A module definition



The file name for RTL source must be
"*module name.v*"



Module

- **Module port declaration**

module *module name* (*port name*, *port name*,...);

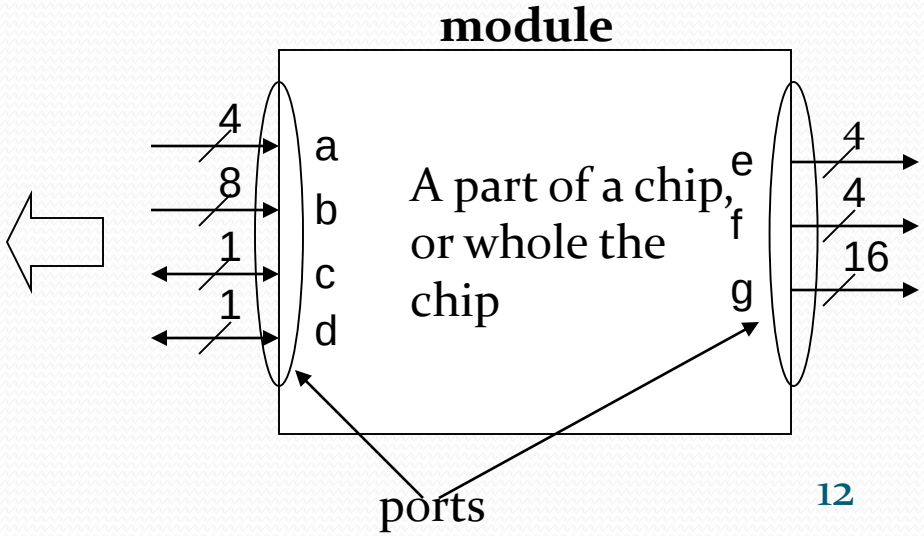
module_port declaration ← Declare whether the ports are input and/or output

input <port_size> *port name*, *port name*, ...;

output <port_size> *port name*, *port name*, ...;

inout <port_size> *port name*, *port name*, ...;

```
module data_conv ( a, b, ...);
input [3:0] a;
input [7:0] b;
output [3:0] e, f;
output [15:0] g;
inout c, d;
```



Module

- **Data type declaration**

module *module name* (*port name*, *port name*,...);

module_port declaration

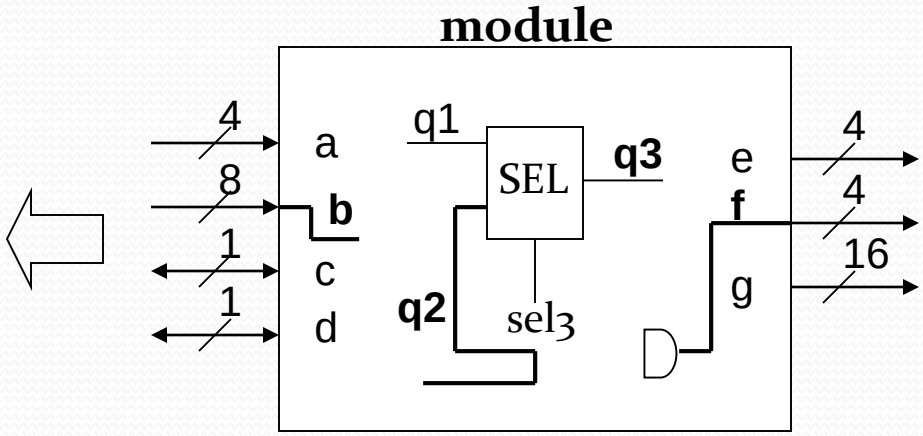
Data type declaration ← Declare characteristics of variables

for net data type

wire <size> *variable name*, *variable name*, ...;

wire <size> *variable name*, *variable name*, ...;

```
wire [3:0] a;
wire [7:0] b;
wire c, d;
wire [3:0] f;
wire [7:0] q1, q2, q3, q4;
wire sel3, ...;
....
```



Module

- **Logic description part**

module *module name* (*port name*, *port name*,...);

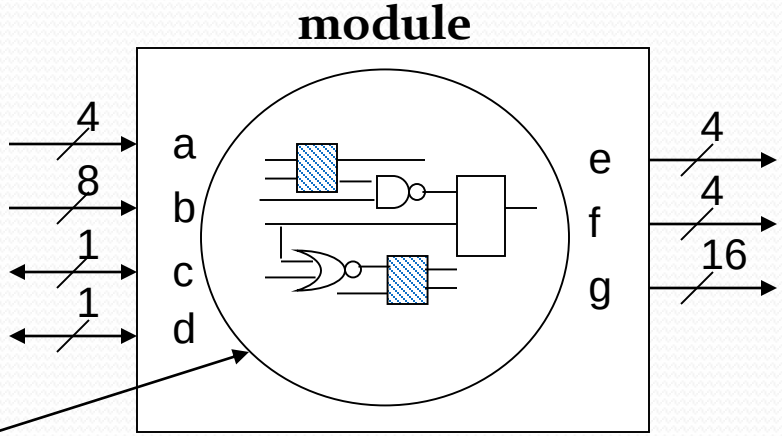
module_port declaration

Data type declaration

Logic description part ← The main part of logic is written here.

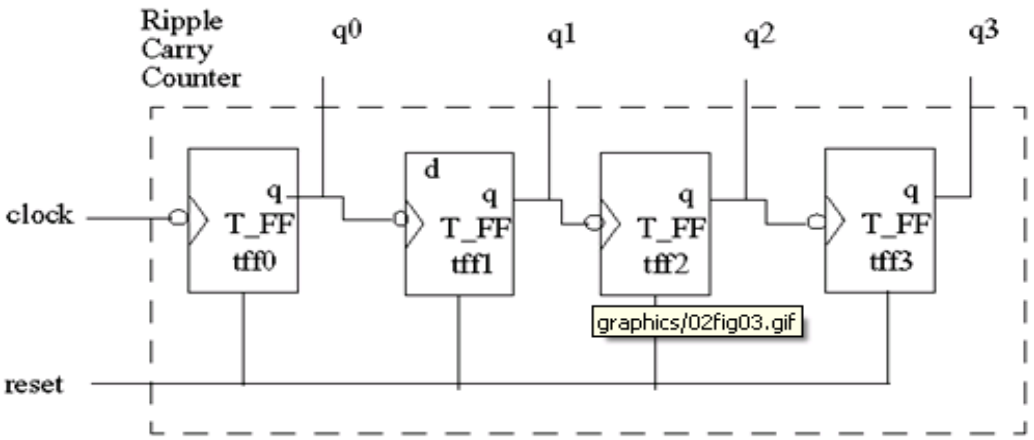
endmodule

Logic is coded in this part using various operator including connections to lower level blocks.

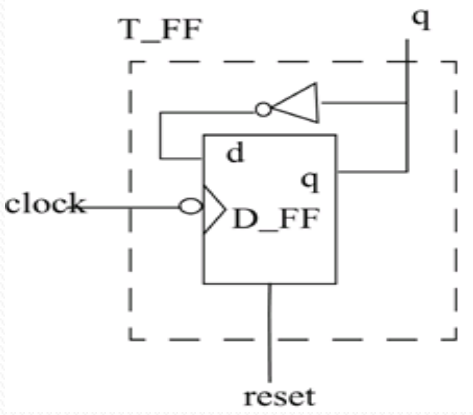


Instance

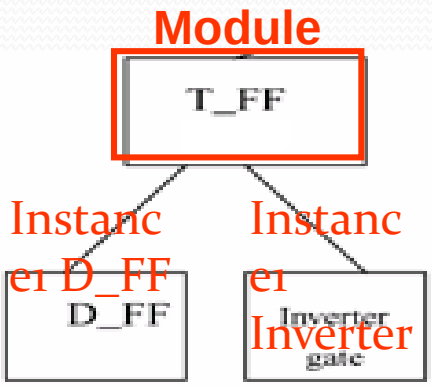
Example: 4-bit Ripple Carry Counter

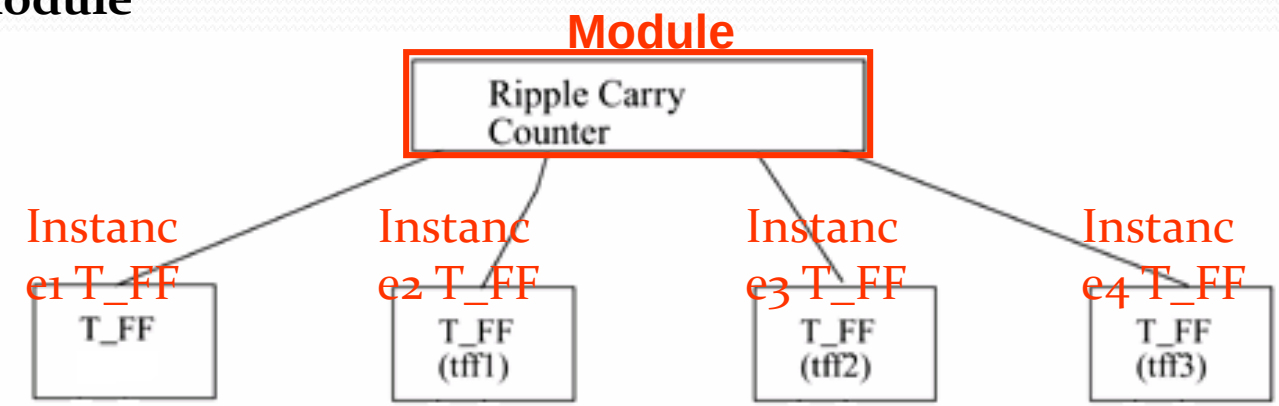


Ripple Carry Counter module



T-flipflop module





Instances

Example: 4-bit Ripple Carry Counter

```
module ripple_carry_counter(q, clk, reset);
output [3:0] q;
input clk, reset;
//4 instances of the module T_FF are created.
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
```

```
module T_FF(q, clk, reset);
output q;
input clk, reset;
wire d;
D_FF dff0(q, d, clk, reset);
not n1(d, q); // not is a Verilog-provided primitive.
endmodule
```

```
// module D_FF with synchronous reset
module D_FF(q, d, clk, reset);
output q;
input d, clk, reset;
reg q;
// Lots of new constructs. Ignore the
//functionality of the
// constructs.
// Concentrate on how the design block is
//built in a top-down fashion.
always @(posedge reset or negedge clk)
if (reset)
q <= 1'b0;
else
q <= d;
endmodule
```

Instance

- **Connecting module instance port by ordered list**

  The port expressions listed for the module instance shall be in the same order as the ports listed in the module declaration.

```
module topmod;
wire [4:0] v;
wire a,b,c,w;
modB b1 (v[0], v[3], w, v[4]);
endmodule

module modB (wa, wb, c, d);
inout wa, wb;
input c, d;
tranif1 g1 (wa, wb, cinvert);
not #(2, 6) n1 (cinvert, int);
and #(6, 5) g2 (int, c, d);
endmodule
```

Instance

- **Connecting module instance port by name**

Connections are made by name, the order in which they appear is irrelevant.

```
module topmod;
wire [4:0] v;
wire a,b,c,w;
modB b1 (.wb(v[3]),.wa(v[0]),.d(v[4]),.c(w));
endmodule

module modB(wa, wb, c, d);
inout wa, wb;
input c, d;
tranif1 g1(wa, wb, cinvert);
not #(6, 2) n1(cinvert, int);
and #(5, 6) g2(int, c, d);
endmodule
```

Content

- Hierarchical structure
- Modules
- Instances
- Lexical convention
- Verilog data types

Lexical convention

- Verilog is a case-sensitive language.
- Whitespace: **\b** blank, **\t** tab, **\n** new line
- Comment: **/*..*/** or **//……..**
- Operator: unary, binary, ternary (more detail in chapter 5)

  a = ~ b; // ~ is a unary operator. b is the operand

  a = b && c; // && is a binary operator. b and c are operands

  a = b ? c : d; // ?: is a ternary operator. b, c and d are operands

- Keywords: **module**, **end module**, **inout**, **reg**…

Lexical convention

- **Number**

  - Two forms to express numbers:

    * 37 : 32 bit decimal 37, or
    * **<size>'<base_format><number>**

  Ex:

10'hFA	10 bits hexadecimal number FA (00_1111_1010)
1'b0	1 bit binary number 0 (0)
6'd30	6 bits decimal number (011110), decimal 30
15'o10752	15 bits octal number (001,000,111,101,010), decimal 4586
	4'b0 is equal to 4'b0000
4'b1	is equal to 4'b0001
4'bz	is equal to 4'bzzzz
4'bx	is equal to 4'bxxxx
-8 'd 6	The two's complement of 6, held in 8 bits

Lexical convention

- **Strings** are stored as a sequence of 8 bit ASCII values
- Strings are variables of ***reg*** type

```
module string_test;
reg [8*14:1] stringvar;
initial begin
    stringvar = "Hello world";
    $display("%s is stored as %h", stringvar,stringvar);
    stringvar = {stringvar,"!!!"};
    $display("%s is stored as %h", stringvar,stringvar);
end
endmodule
```

The output is as follows:

```
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

Lexical convention

Keyword

- Keywords are used to define the language constructs. There are a lot of keywords in Verilog HDL. (Refer to Verilog books)
- All keywords are defined in lower case
- Do not use keywords as user's definition.

Examples:

module, endmodule	fork, join
input, output, inout	specific, endspecific
reg, integer, real, time	timescale
not, and, nand, or, nor, xor	include
parameter	undef
begin, end	nmos, pmos,…

Lexical convention

System task & function

- Standard system tasks for certain routine operations
- Appear in form: **$<keyword>**
- Introduce most useful system tasks, more are listed in IEEE specification
- Functions: Displaying on screen, monitor values on net…
- **$display**, **$monitor**, **$stop**, **$finish**

Lexical convention

Compiler directive

- `define: define text macro in Verilog (similar to #define in C)

```
//define a text macro that defines default word size
//Used as 'WORD_SIZE in the code
'define WORD_SIZE 32

//define an alias. A $stop will be substituted wherever 'S appears
'define S $stop;

//define a frequently used text string
'define WORD_REG reg [31:0]
// you can then define a 32-bit register as 'WORD_REG reg32;
```

- `include: include content of a Verilog source file in another Verilog file
- `ifdef, `timescale

Data types

- **Value set**: Four basic values:

0 – represents a logic zero, logic low, ground or false condition.

1 – represents a logic one, logic high, power or true condition.

x – represents an unknown logic value.

z – represents a high-impedance, unconnected, tri-state.

Data types

- 2 main groups: **Net** and **Variable**
- The net data types represent physical connections between structural entities, such as gates.
- **Net**: does not store a value, except “trireg” net
- Default value is z (trireg default value is x)
- Net types:

wire	tri	tri0	supply0
wand	triand	tri1	supply1
wor	trior	trireg	uwire

(Ref. Verilog IEEE book for detail of net types)

Ex:

- wire w1, w2; // declare 2 wires
- wand w;

Data types

- **Net**

- Nets present the physical connections between devices. A net does not store a value, it must be driven by a gate or a continuous assignment. If a net variable has no driver, then it has a high-impedance value (z).
- Verilog automatically propagates new values onto a net when the drivers change value.
- Cannot be assigned in an **initial** or **always** block

Data types

- **Variable**: hold a value until another value assigned to it

- Used in behavioral description (procedural assignment)

reg : unsigned integer variables of varying bit width

integer : 32-bit signed integer

time : 64-bit unsigned integer

real : signed floating-point

realtime: store time as a real value

- **reg**, **time**, **integer** (default value is x),

real , **realtime** (default value is 0)

Data types

- **Vector**: Multiple ***net*** or ***reg*** data types are declared by specifying a range, is known as a vector

```
wand w;                          // a scalar net of type "wand"
tri [15:0] busa;                 // a three-state 16-bit bus
trireg (small) storeit;          // a charge storage node of strength small
reg a;                           // a scalar reg
reg[3:0] v;                      // a 4-bit vector reg made up of (from most to
                                 // least significant)v[3], v[2], v[1], and v[0]
reg signed [3:0] signed_reg;        // a 4-bit vector in range -8 to 7
reg [-1:4] b;                    // a 6-bit vector reg
wire w1, w2;                     // declares two wires
reg [4:0] x, y, z;               // declares three 5-bit regs
```

Data types

- **Array** declaration for a net or variable that is either scalar or vector

Declaration	Element type
reg x[11:0];	scalar **reg**
wire [0:7] y[5:0];	8-bit-wide vector wire indexed from 0 to 7
reg [31:0] x [127:0];	32-bit-wide **reg**

- **Memory**: is one-dimension array with element of type **reg**. These memories can be used to model ROM, RAM, or reg files

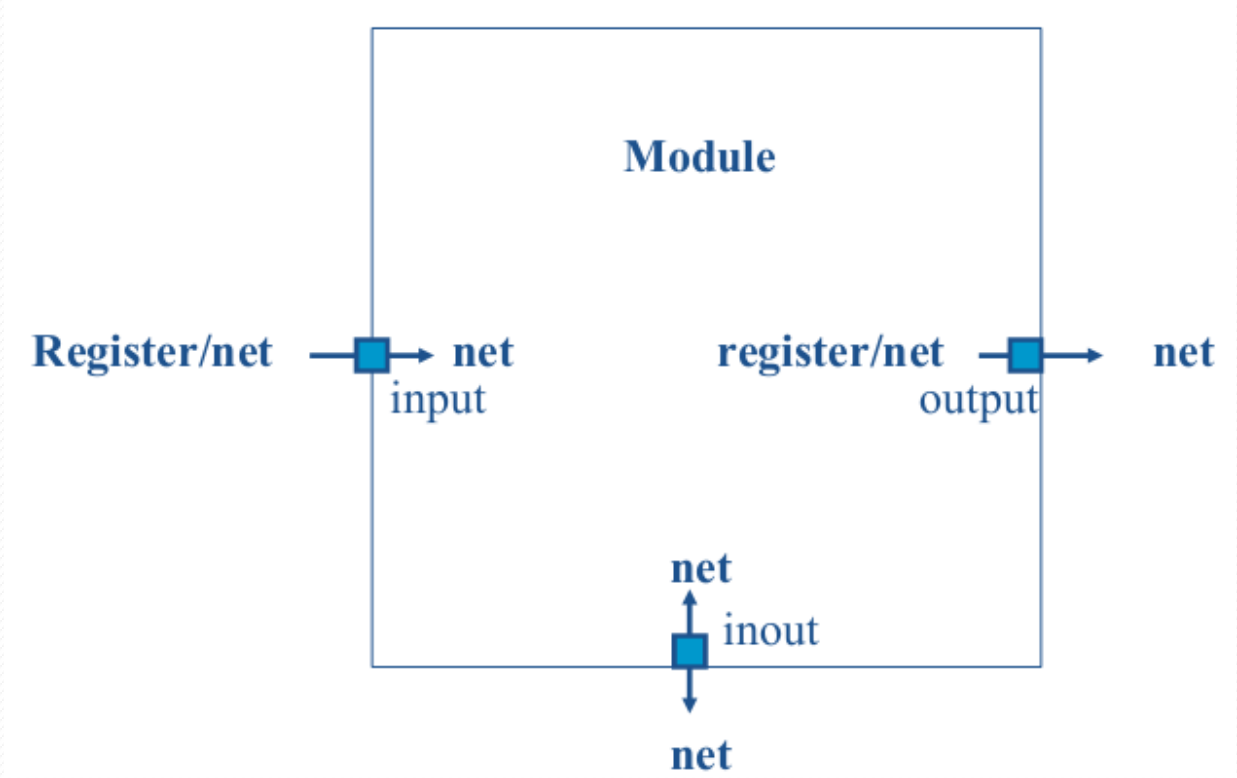
Data types

- **Parameters** do not belong to net or variable. Parameters are constants
- 2 types: **module parameter** (defparam) and
  specify parameter (specparam)

```
parameter  e = 25, f = 9;
module RAM16GEN (output [7:0] DOUT, input [7:0] DIN, input [5:0] ADR,
    input WE, CE);
specparam dhold = 1.0;
specparam ddly = 1.0;
parameter width = 1;
parameter regsize = dhold + 1.0;   // Illegal - cannot assign
                                   // specparams to parameters
endmodule
```

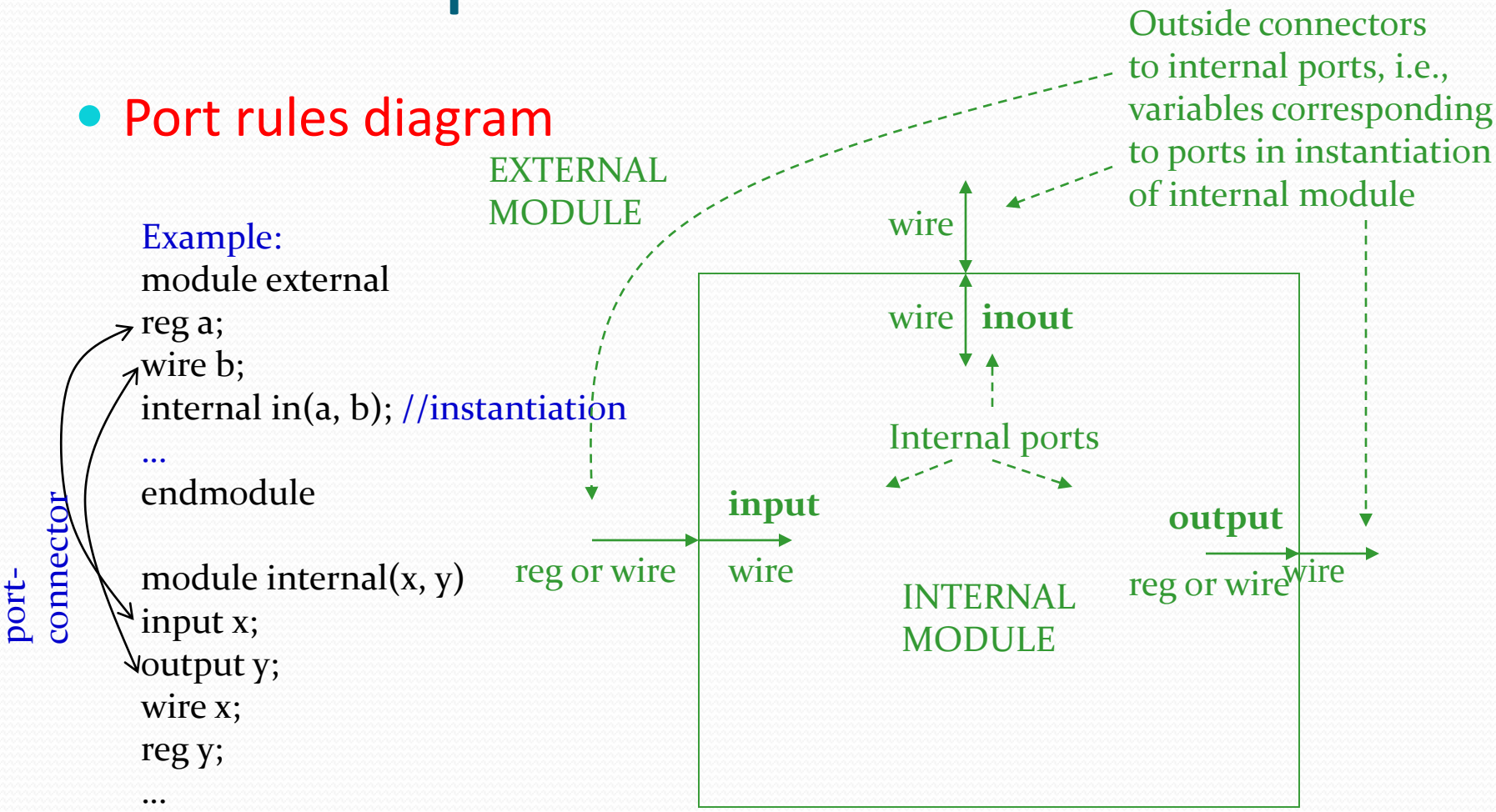
Data types in Module

- Correct data types for ports



Module port rules

- Port rules diagram



Example:

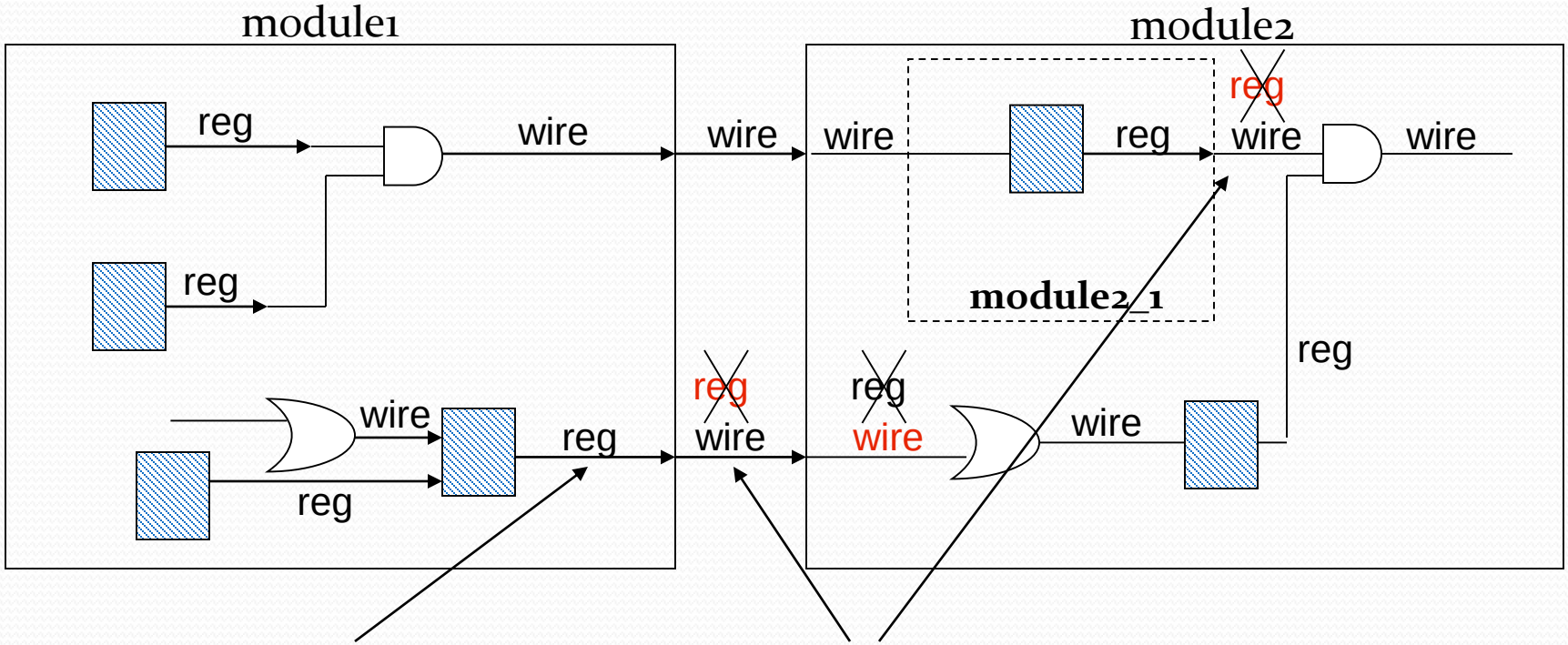
```
module external
reg a;
wire b;
internal in(a, b); //instantiation
...
endmodule

module internal(x, y)
input x;
output y;
wire x;
reg y;
...
endmodule
```

General rule (with few exceptions) Ports in all modules except for the stimulus module should be wire. Stimulus module has registers to set data for internal modules and wire ports only to read data from internal modules.

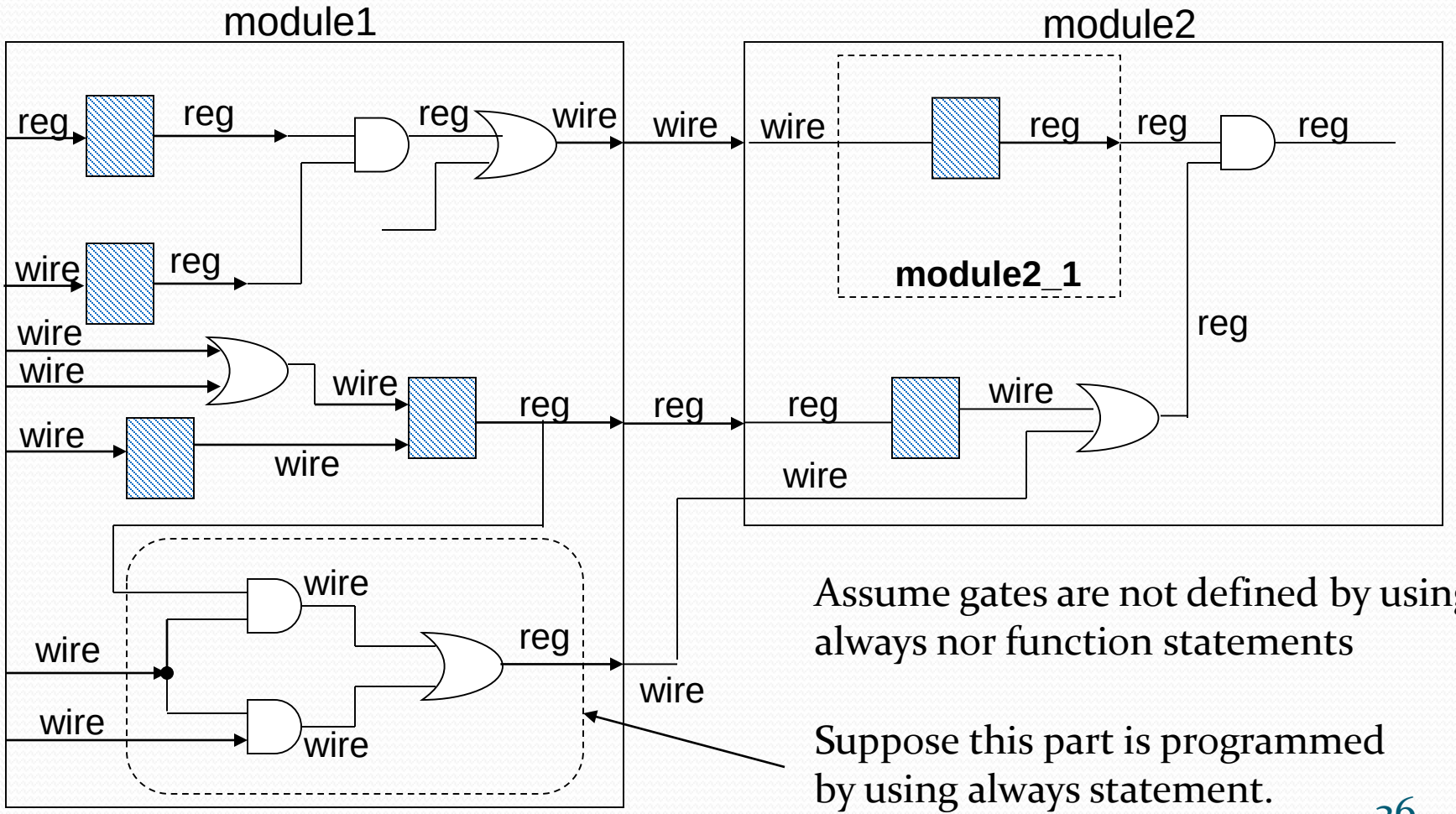
Module port rules

- Mistakes and correct on register and net data type



Module port rules

- Mistakes and correct on register and net data type

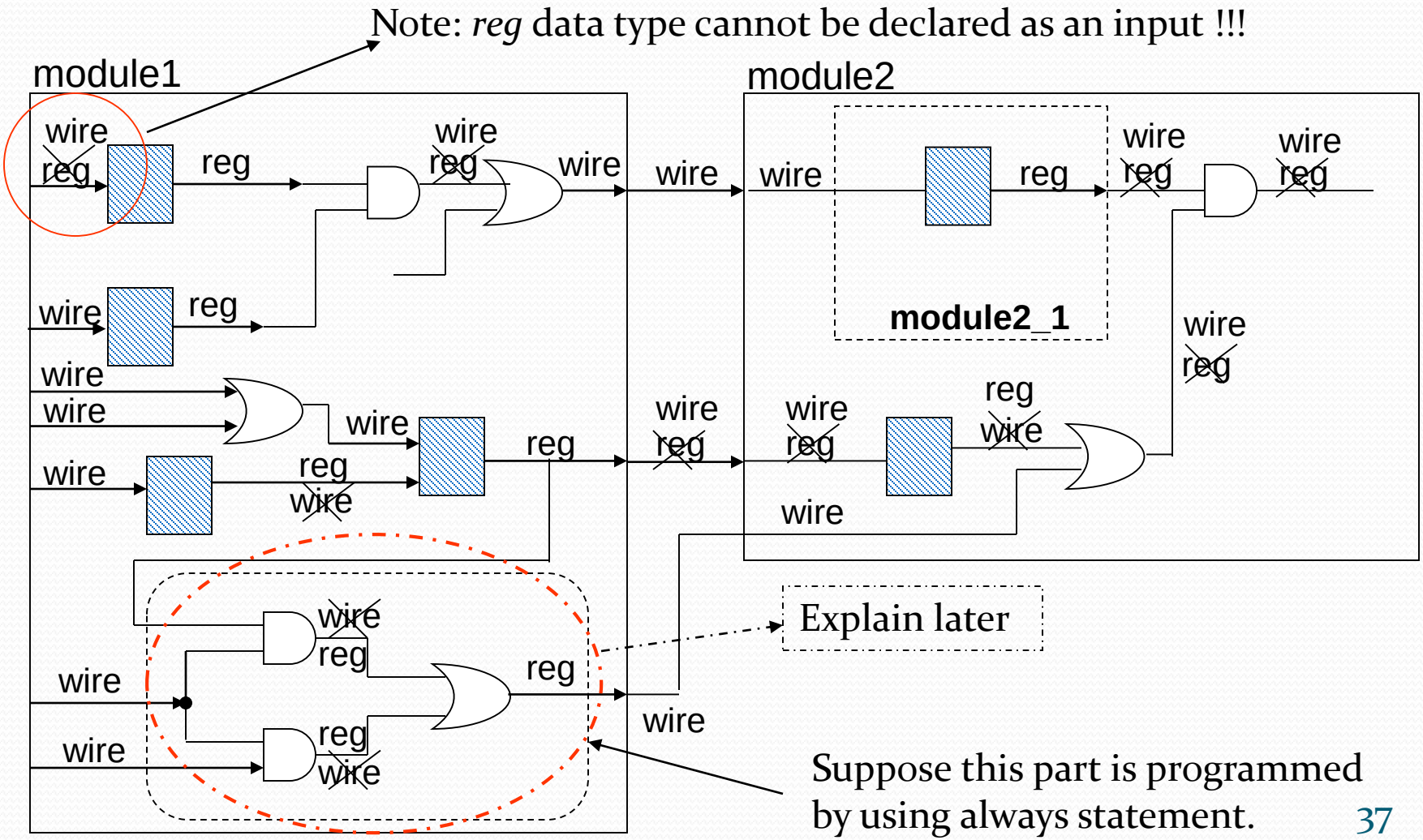


Assume gates are not defined by using always nor function statements

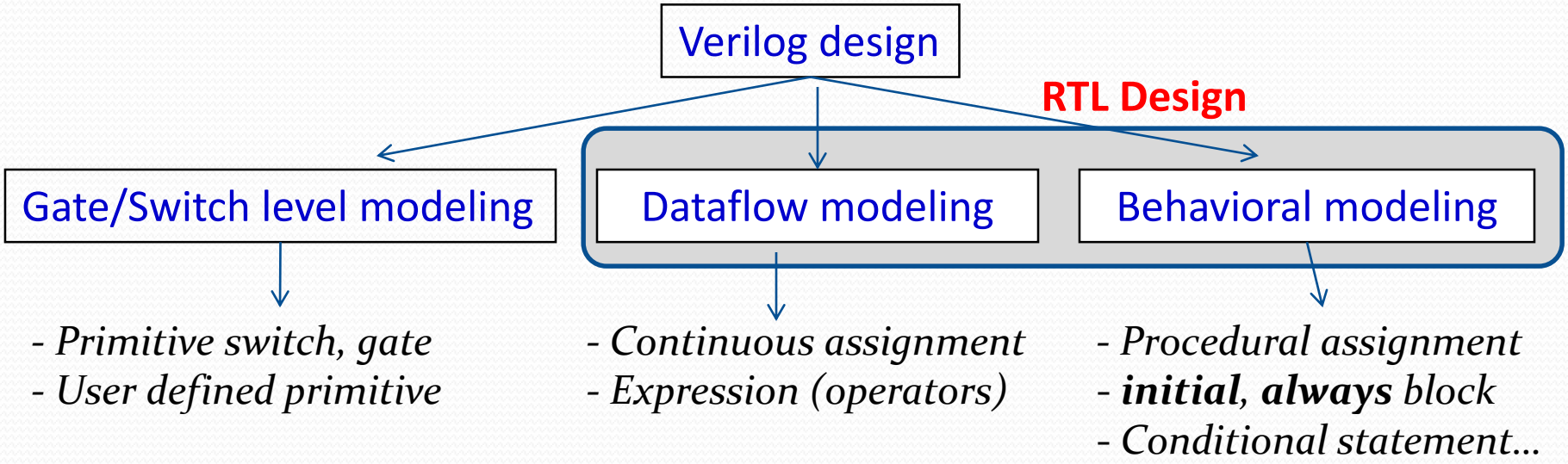
Suppose this part is programmed by using always statement.

Module port rules

- Mistakes and correct on register and net data type



Verilog model for hardware design



- There are different ways of modeling a hardware design. Choose an appropriate model to design Combinational or Sequential Circuit.
- Some books do not classify Dataflow modeling is a separate modeling type, but belongs to Behavioral modeling.