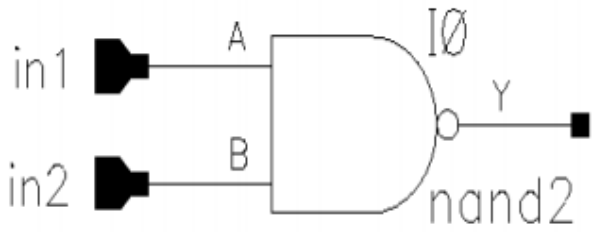
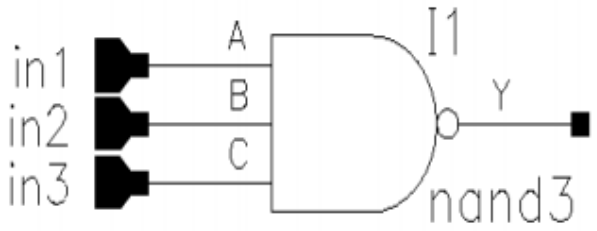
Lecture 2

Verilog HDL (Dataflow Level)

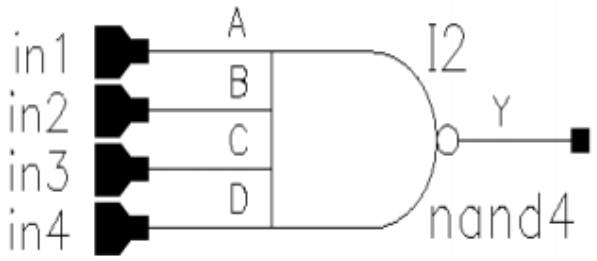
Gate level



nand (y, in1, in2) ;

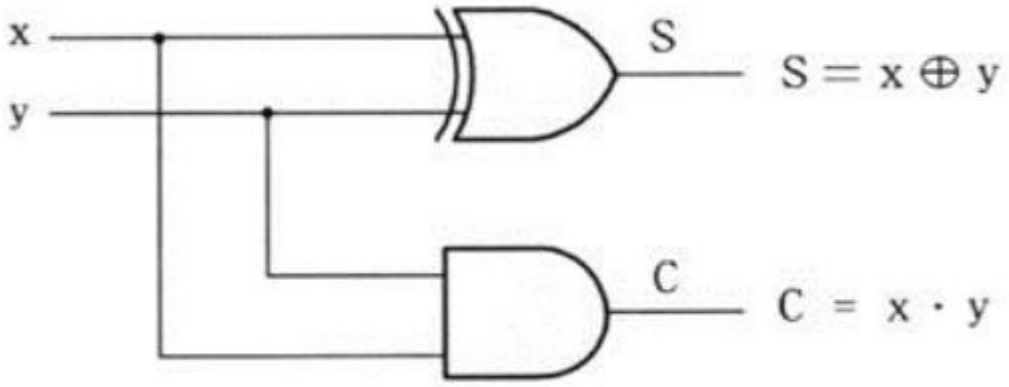


nand nand1(y, in1, in2, in3) ;



nand (y, in1, in2, in3, in4) ;

Gate level

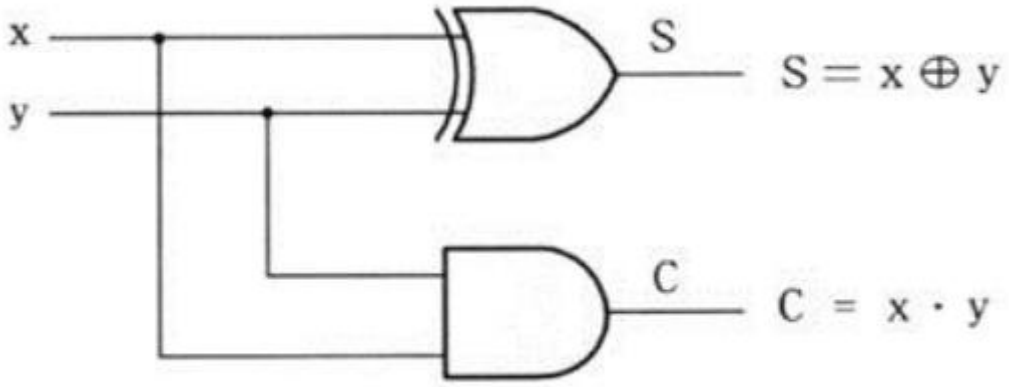



```
module ha( S, C, x, y) ;
    input x, y;
    output S, C;

    xor xor1( S, x, y);
    and and1( C, x, y);

endmodule
```

Dataflow level



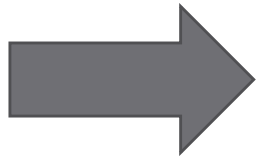

```
module ha( S, C, x, y) ;
    input x, y;
    output S, C;

    assign S = x ^ y;
    assign C = x & y;

endmodule
```

Dataflow level

A	B	C_{in}	C_{out}	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



$$Sum = C_{in} \oplus (A \oplus B)$$

$$C_{out} = C_{in}(A \oplus B) + AB$$

Continuous Assignments

- Used to drive a value onto a net
- The left hand side of an assignment must be a scalar or vector net(wire)
- Starts with the keyword **assign**

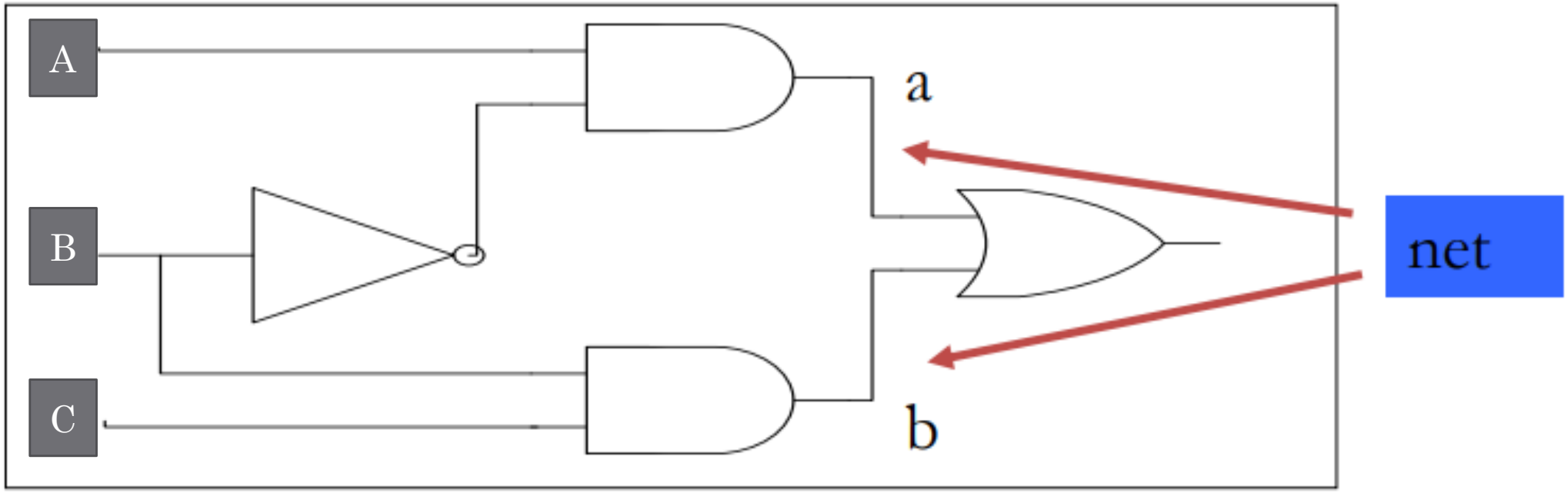
Syntax :

assign <left_hand_side>=<right_hand_side>

```
wire a, b, c ;
assign c = a & c;
```

Continuous Assignments

- Replaces gates in the description of a circuit
- assign b = B & C;



Continuous Assignments

- Delay values can be specified

```
assign      #10 out2= a & ( b | c );
```

Vectors

- Vectors
  - wire [0:31] w1; //a 32-bit wires with MSB=bit0
  - [2:0] vs [0:2]
- Scalar
  - 1-bit vector
  - wire a;

```
wire [2:0] a = 3'b110;
wire [0:2] b = 3'b110;
```

a[2]	a[1]	a[0]
1	1	0

b[2]	b[1]	b[0]
0	1	1

Number specifications

- <size> ’<base format><number>
  - <size> in decimal: # of bits
  - Legal base are decimal(‘d’ or ‘D’), hexadecimal(‘h’ or ‘H’), binary(‘b’ or ‘B’), and octal(‘o’ or ‘O’)

```
wire a = 1’b1;
```

Number specifications

- <size> ’<base format><number>
  - <size> in decimal: # of bits
  - Legal base are decimal(‘d’ or ‘D’), hexadecimal(‘h’ or ‘H’), binary(‘b’ or ‘B’), and octal(‘o’ or ‘O’)

```
wire [3:0] a =4’b0101;
```

Number specifications

- <size> ’<base format><number>
  - <size> in decimal: # of bits
  - Legal base are decimal(‘d’ or ‘D’), hexadecimal(‘h’ or ‘H’), binary(‘b’ or ‘B’), and octal(‘o’ or ‘O’)

```
wire [11:0] a = 12’h9ac;
```

Operator types

Type	Operators
Concatenate, replicate	{} {{}}
Unary Reduction	! ~ & | ^
Arithmetic	* / % + -
Logical shift	<< >>
Relational	> < >= <=
Equality	== === != !==
Binary bit-wise	& | ^ ~^
Binary logical	&& ||
conditional	?:

Bitwise Operators

- A bit-by-bit operation

```
wire [1:0] a = 2'b10;
wire [1:0] b = ~a;
```

Symbol	Operation
~	Bitwise negation
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~^, ^~	Bitwise exclusive nor

Bitwise Operators

- A bit-by-bit operation

```
wire [1:0] a = 2'b10;
wire [1:0] b = 2'b11;
wire [1:0] c = a & b;
```

Symbol	Operation
~	Bitwise negation
&	Bitwise and
|	Bitwise or
^	Bitwise exclusive or
~^, ^~	Bitwise exclusive nor

Arithmetic Operators

- Negative numbers are represented as 2's complement

```
wire [1:0] a = 2'b10;
wire [1:0] b = 2'b01;
wire [1:0] c = a + b;
```

Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponent(Power)
%	Modulus

Concatenation/Replication Operators

- a = 1'b0; b = 1'b1;
- Concatenation operator
  - wire [1:0] x = { a , b };
  - // x = 2'b01;
- Replication operator
  - wire [2:0] y = { 3{ a } };
  - // y = 3'b000;

Symbol	Operation
{ , }	Concatenation
{const_expr{}}	Replication

Reduction Operators

- Perform only on one vector operand
- Yield a 1-bit result
- Work bit by bit from right to left

```
wire [3:0] a = 4'b0011;
wire [1:0] b = &a;
```

Symbol	Operation
&	Reduction and
~&	Reduction nand
|	Reduction or
~|	Reduction nor
^	Reduction exclusive or
~^	Reduction exclusive nor

Logical Operators

- The result is 1 if the expression is true and 0 if the expression is false

```
if ( a || b )
```

Symbol	Operation
!	Logical negation
&&	Logical and
||	Logical or

Relational Operators

- The result is 1 if the expression is true and 0 if the expression is false

```
if ( a > b )
```

Symbol	Operation
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

Equality Operators

- The result is 1 if the expression is true and 0 if the expression is false

```
wire [3:0] b = 4'b101x;

b == 4'b101x // unknown -x
b === 4'b101x // true - 1
```

Symbol	Operation
==	Logical equality
!=	Logical inequality
===	Case equality
!==	Case inequality

Shift Operators

- <<
  - Shift Left, Logical (fill with zero)
- >>
  - Shift Right, Logical (fill with zero)
- <<<
  - Shift Left, Arithmetic
- >>>
  - Shift Right, Arithmetic (keep sign)

Symbol	Operation
>>	Logical right shift
<<	Logical left shift
>>>	Arithmetic right shift
<<<	Arithmetic left shift

Shift Operators

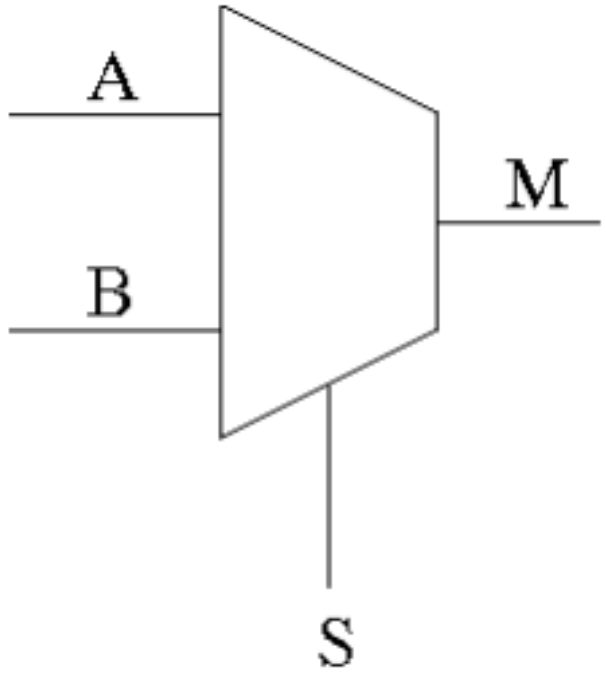
- reg [3:0] r_Shift1 = 4'b1000;
- reg signed [3:0] r_Shift2 = 4'b1000;

$display("%b", r_Shift1 << 1);	→	0000
$display("%b", $signed(r_Shift1) <<< 1);	→	0000
$display("%b", r_Shift2 <<< 1);	→	0000
$display("%b", r_Shift1 >> 2);	→	0010
$display("%b", $signed(r_Shift1) >>> 2);	→	1110
$display("%b", r_Shift2 >>> 2);	→	1110

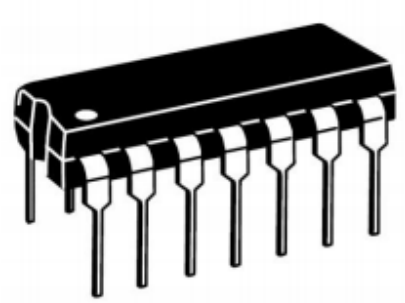
Conditional Operators

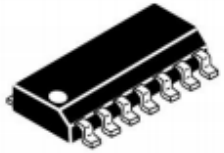
- Usage :
  - condition_expr ? true_expr : false_expr;

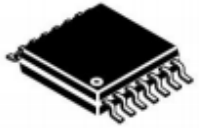
```
assign M = S ? A : B;
```

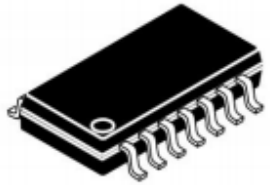


Module

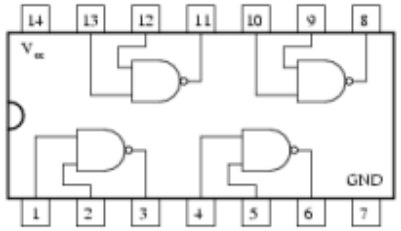




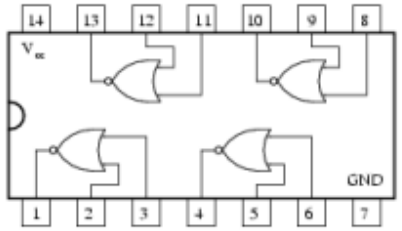




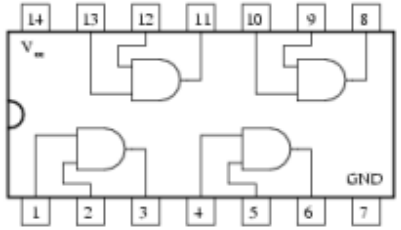
5400/7400
Quad NAND gate



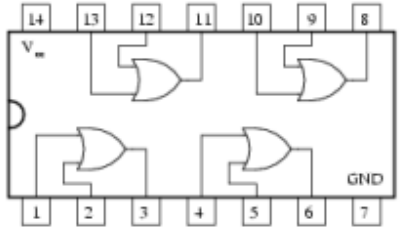
5402/7402
Quad NOR gate



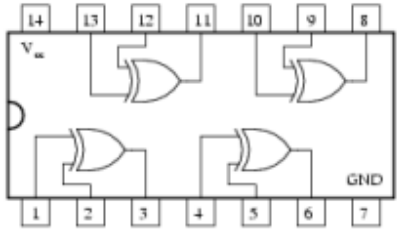
5408/7408
Quad AND gate



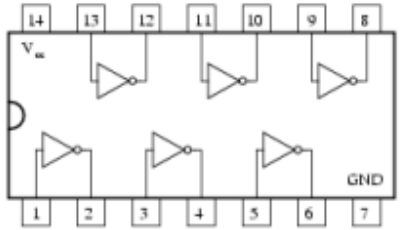
5432/7432
Quad OR gate



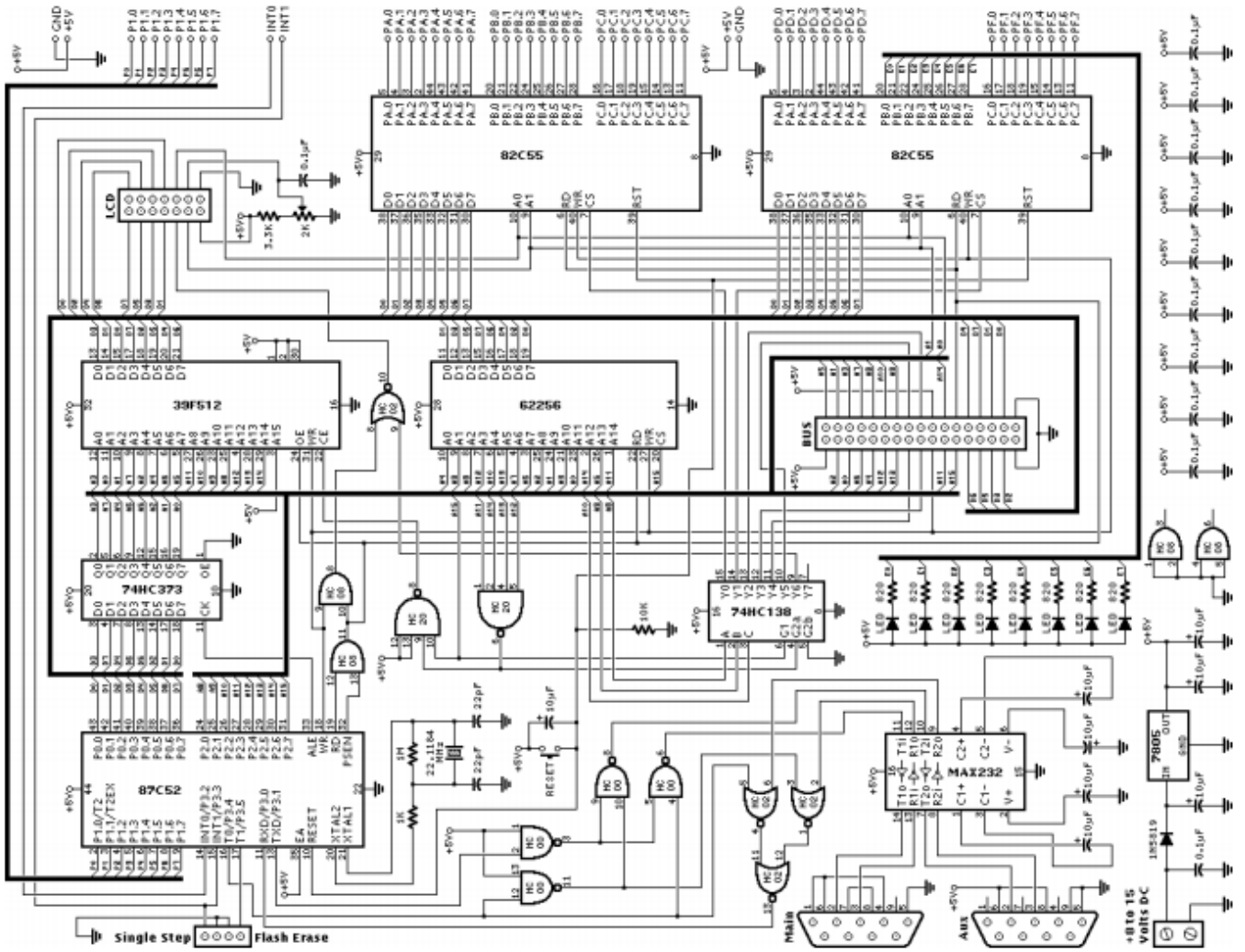
5486/7486
Quad XOR gate



5404/7404
Hex inverter

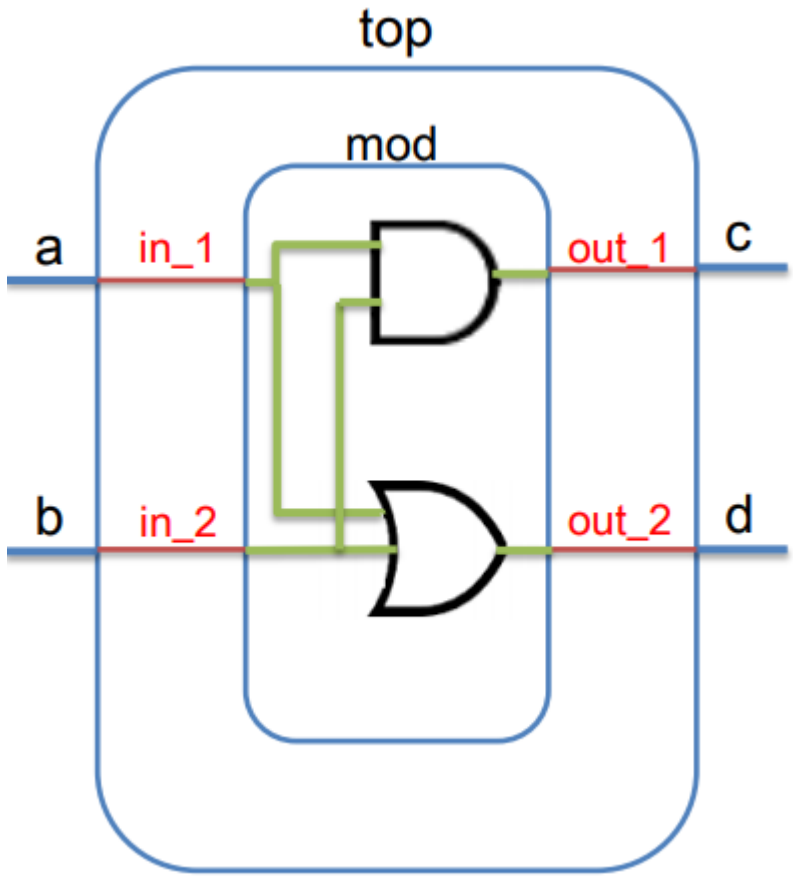


Many Modules



Module Instantiation

- Positional mapping
  - port order follows the module declaration.
- Name mapping
  - port order is independent of the position.



Positional Mapping

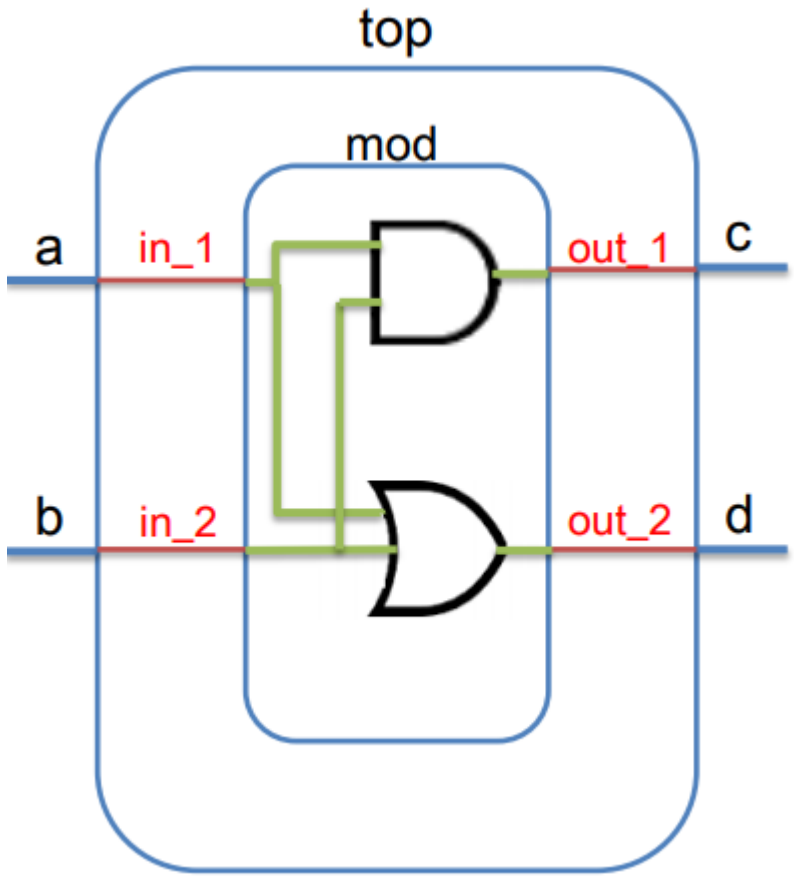
```
Module m1(in_1, in_2, out_1, out_2);
   input    in_1, in_2;
   output  out_1, out_2;

   and  and1(out_1, in_1, in_2);
   or    or1(out_2, in_1, in_2);
endmodule

Module top(a, b, c, d);
   input    a, b;
   output  c, d;

m1  mod(a, b, c, d);

endmodule
```



Name Mapping

```
Module m1(in_1, in_2, out_1, out_2);
   input    in_1, in_2;
   output  out_1, out_2;

   and  and1(out_1, in_1, in_2);
   or    or1(out_2, in_1, in_2);
endmodule

Module top(a, b, c, d);
   input    a, b;
   output  c, d;

m1  mod(.out_1(c), .out_2(d), .in_1(a), .in_2(b));

endmodule
```