INTRODUCTION TO VERILOG

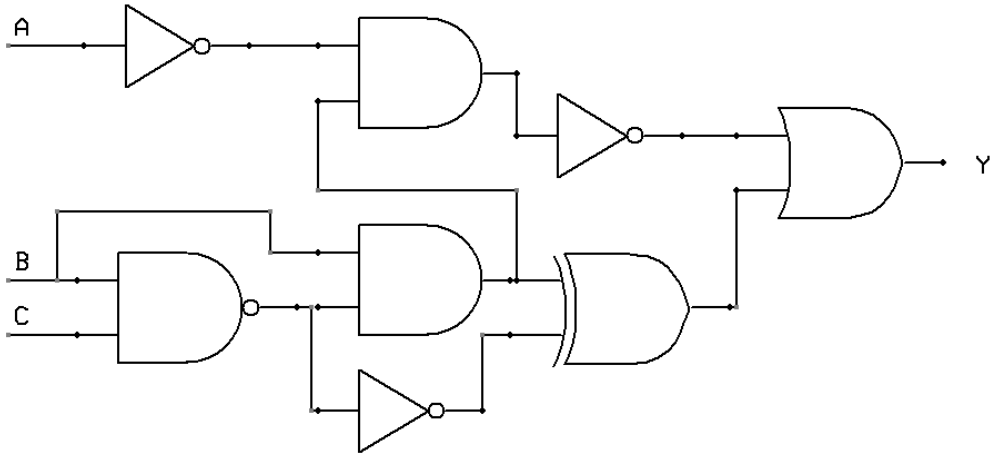
Mahmut Kamil Aslan

24.04.2017

CONVENTIONAL DESIGN

- Determine I/O & states
- Build truth table
- Optimization with K-map
- Obtain the logic eq.
- Draw the schematic

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



HDLs

- Hardware Description Language
- Describe any hardware (digital) in textual form
- Verilog and VHDL
- Simplifying the design representation

VHDL

```
library IEEE;
use IEEE.STD_Logic_1164, all;

entity LATCH_IF_ELSEIF is
  port (En1, En2, En3, A1, A2, A3: in std_logic;
        Y: out std_logic);
end entity LATCH_IF_ELSEIF;

architecture RTL of LATCH_IF_ELSEIF is
begin
   process (En1, En2, En3, A1, A2, A3)
   begin
      if (En1 = '1') then
        Y <= A1;
      elseif (En2 = '1') then
        Y <= A2;
      elseif (En3 = '1') then
        Y <= A3;
      end if;
   end process;

end architecture RTL;
```

Verilog

```
module LATCH_IF_ELSEIF  (En1, En2, En3, A1, A2, A3,Y);
    input En1, En2, En3, A1, A2, A3;
    output Y;

    reg Y;

    always @(En1 or En2 or En3 or A1 or A2 or A3)
       if (En1 == 1)
          Y =A1;
     else if (En2 == 1)
          Y = A2;
     else if (En3 == 1)
          Y = A3;

end module
```

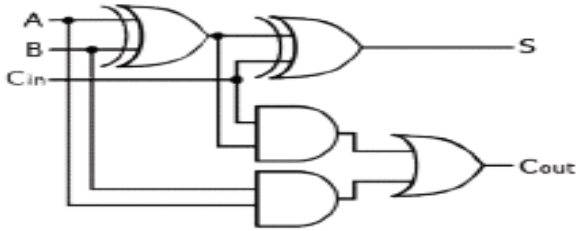
VERILOG SYNTAX

- Similar to **C**
- Reserve words with lowercase letters

  (module, wire, reg…)
- Case sensitive
- // line comment and /* */ block comment
- Every line ends with a semicolon **;**
- Different description levels (gate, RTL or behavioral level)

VERILOG DESCRIPTION LEVELS

- Structural (switch or gate) level design

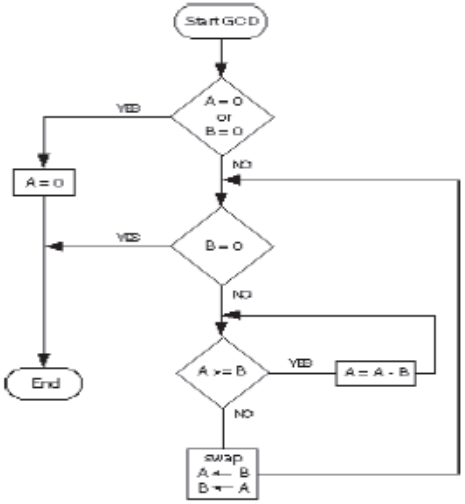


- RTL (Data Flow) level design

$$Y = \overline{A} \cdot B + A \cdot \overline{B} + A \cdot B$$

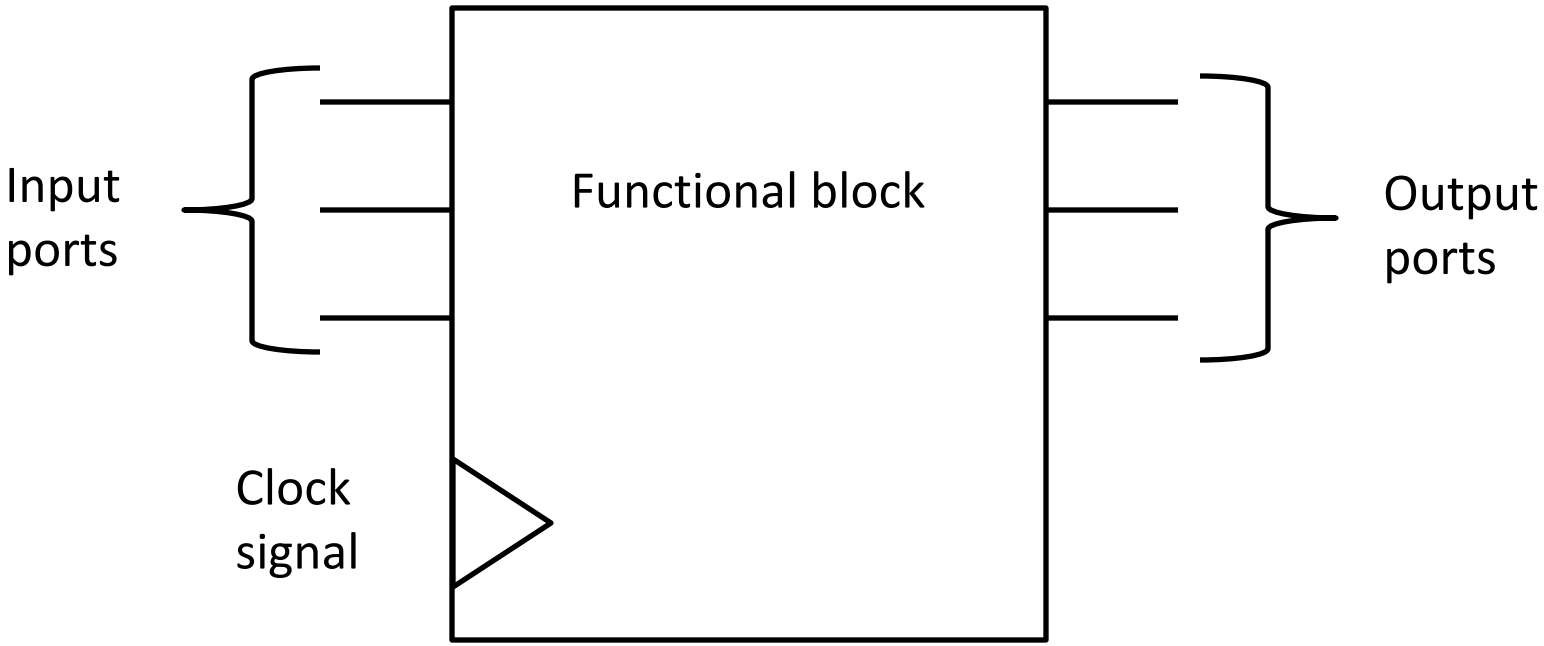
$$F = \overline{\overline{A} \cdot (B + \overline{C}) \cdot (A + \overline{B} + C) \cdot (\overline{A} \cdot \overline{B} \cdot \overline{C})}$$

- Behavioral (algorithmic) level design

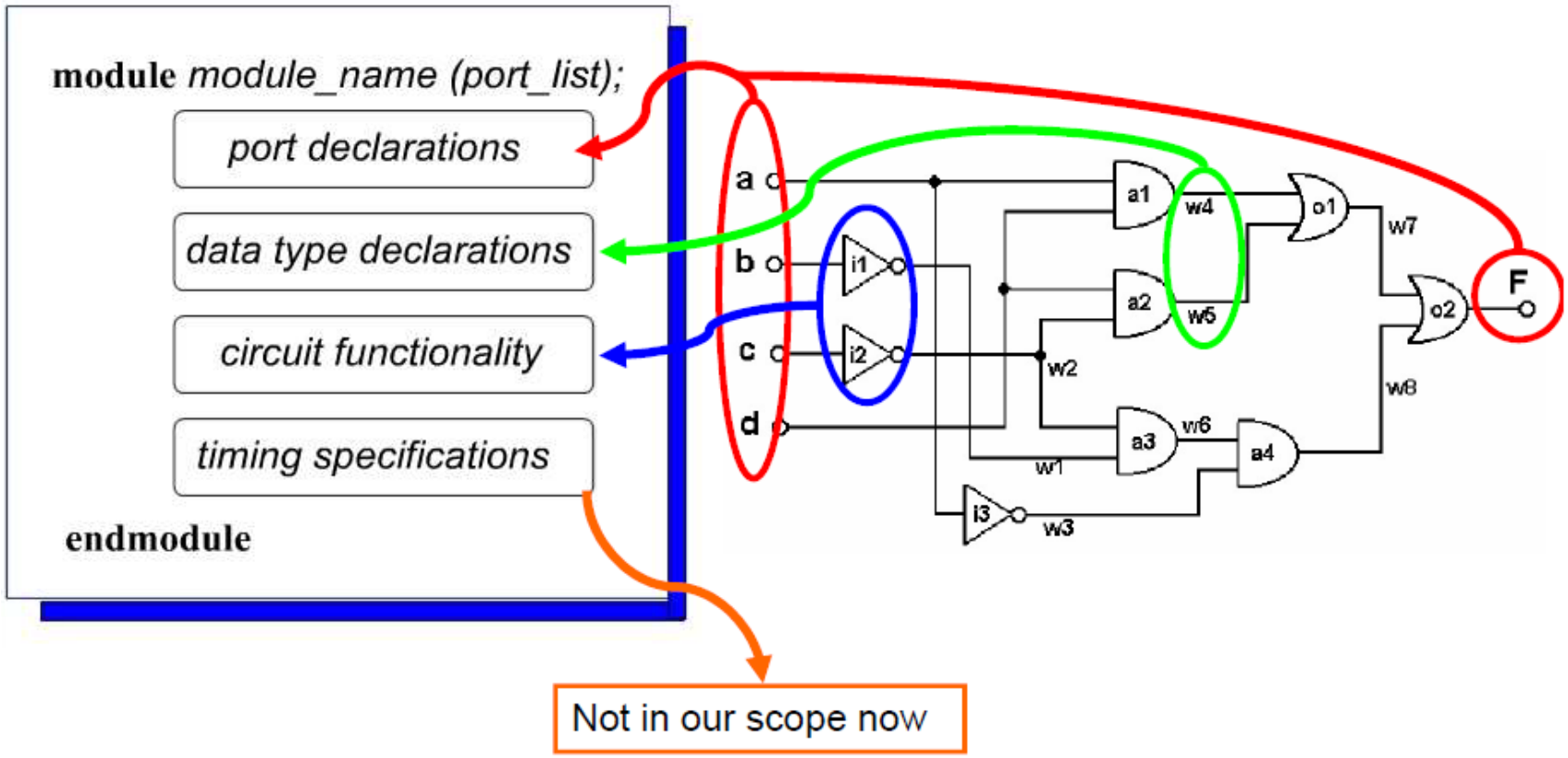


DESIGN WITH VERILOG

- **Module** design
- Determine description level
- Implement the Verilog code



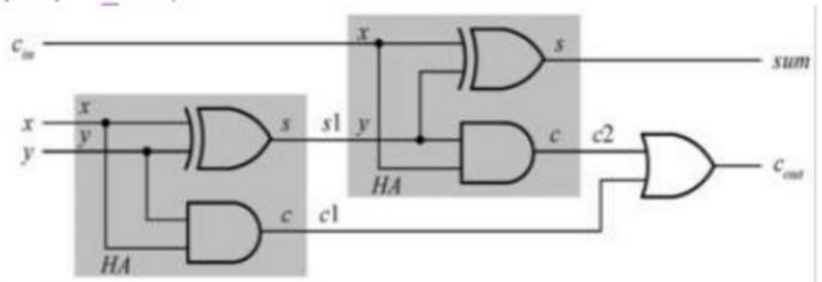
SAMPLE VERILOG CODE



EXAMPLE

- Design of a full adder with Verilog with different design levels.

```
module full_adder_mixed_style(x, y, c_in, s, c_out);
// I/O port declarations
input    x, y, c_in;
output   s, c_out;
reg      c_out;
wire     s1, c1, c2;
// structural modeling of the first half adder HA 1.
   xor xor_ha1(s1, x, y);
   and and_ha1(c1, x, y);
// dataflow modeling of the second half adder HA 2.
   assign s = c_in ^ s1;
   assign c2 = c_in & s1;
// behavioral modeling
always @(c1, c2)   // also can use always @(*)
     c_out = c1 | c2;
endmodule
```



VERILOG SYNTAX

- All statements are concurrent except **initial** or **always** blocks.

  assign sum=a+b+c;

  assign sum= a+b;

  assign sum=sum+c;

  sum value is **indeterminate**

- if, case, for, while loops should appear inside these blocks (**initial** or **always**).

VERILOG SYNTAX

- Logic types:

0 for logic ‘0’ or **false** condition

1 for logic ‘1’ or **true** condition

Z Output of an undriven tri-state driver – High-Z value

x when un-initialized or unknown logic value

VERILOG SYNTAX

- Net data types:

Format: <size>’<base_format><number>

d or D decimal (default if no base format given)

h or H : hexadecimal

o or O : octal

b or B : binary

Ex: 8’b1111_0000

8’hF0

(‘_’ is used for grouping digits increases readibility)

DATA TYPES

- Two groups of types, **"net data types"** and **"variable data types"**
- "net data type" means that it must be driven. The value changes when the driver changes value. **wire, supply0, supply1, tri, triand, trior, tri0, tri1**
- "variable data type" means that it changes value upon assignment and holds its value until another assignment. **integer**, **real**, **realtime**, **reg**, **time**.

DATA TYPES

- Arrays:

integer a[1:10];

reg[31:0] mem[0:4095]; 4096 x 32 bits

mem[2] the third word in memory

mem[2][31] is the most significant bit of the third word in memory.

wire[7:0] mem[1:3][799:0][599:0];

Wire vs Reg

- **wire:**

  Represents a wire

  Cannot store value without being driven

  Used to model combinational logic

- **reg:**

  Represents a register

  Can store value

  Used to model sequential logic

CONTINUOUS ASSIGNMENTS

- Drives values into the nets.
- **assign** operation

Continuous assignments to wire

assign variable = exp;

wire out;
assign out = In_A & In_B ;

Results in combinational logic

PROCEDURAL ASSIGNMENTS

- Enable updating registers
- Can be used only within the structured procedures (always, initial, task, function)
- Procedural assignments statements:

  blocking and **non-blocking**

‘Continuous Procedural Assignment’

(continuously drive a value into a register or a net)

BLOCKING vs NON-BLOCKING

- Blocking:

<variable> = <statement>

Similar to C code

Next statement will be executed after the execution of the current statement

Used for combinational logic

BLOCKING vs NON-BLOCKING

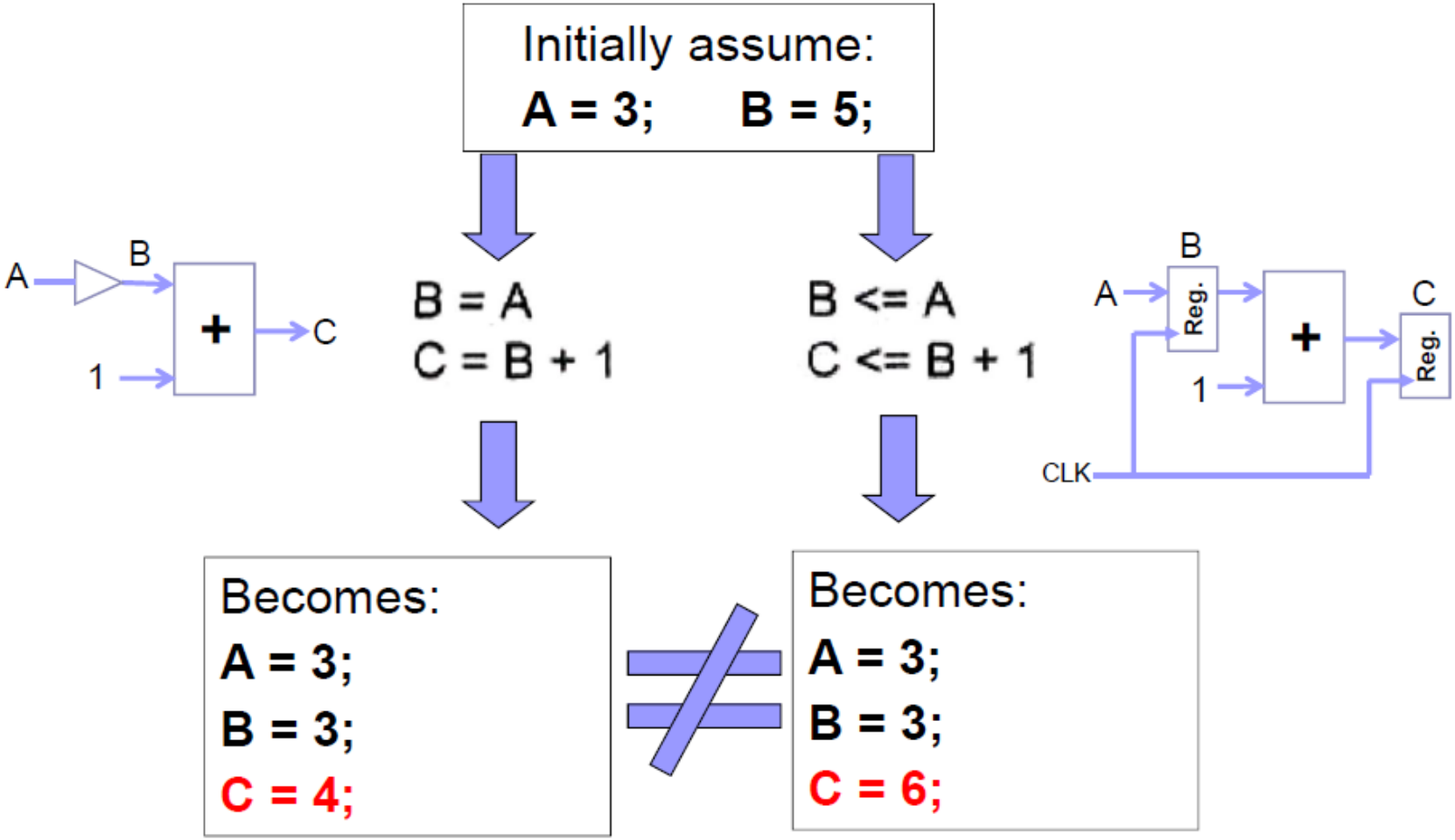
- Non- Blocking:

<variable> <= <statement>

Next statement will be executed at the same time

Used in sequential logic

BLOCKING vs NON-BLOCKING



Initial Block

- Executed once at beginning

```
initial
begin
    a = 0;
    b = 0;
    c = 0; // Use Blocking Statements
end
```

Always Block

- ‘always’ block executes always

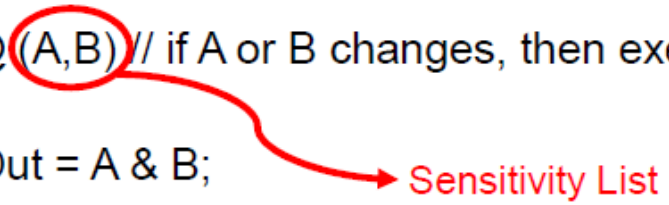
The @ symbol indicates that the block will be triggered "at" the condition in parenthesis.

```
// C analogy:
while(1)
{
      if(condition)
            do operation
}
```

```
//Demonstrates the use of “always” block:
module always_and_gate(Out,A,B);

always @(A,B) // if A or B changes, then execute the following:
begin
        Out = A & B;
end

endmodule
```



RELATIONAL OPERATIONS

- A<B, A>B A<=B, A>=B, A==B, A!=B

The result is 0 if the relation is false, 1 if the relation is true, X if either of the operands has any X's in the number

A===B, A!==B

These require an exact match of numbers, X's and Z's included

data=52'bx

data==52'bx (the result is x so false '0')

data===52'bx (the result is 1)

RELATIONAL OPERATIONS

- !, &&, ||

  Logical NOT, AND, OR of expressions

  e.g. if (!(A == 5) && (B != 2))
- ~, &, |, ^

  bitwise NOT, AND, OR, and XOR

  e.g. Y = ~(A & B);
- {a, b[3:0]} // example of concatenation

IF-ELSE

- **if - else**

```
if (a)
begin
counter = counter + 1;
data_out = counter;
end
else
data_out = 8'bz;
```

CASE

- **case**()
  <case:1>:<statement1>
  <case:1>:<statement1>
  <case:1>:<statement1>
  endcase

LOOPS

- **for, while, forever**

```
t = 0;
while (t < MSB)
  begin
    //Initializes vector elelments
    Vector[t] = 1'b0;
    t = t + 1;
  end
for (index=0; index < 10; index = index + 2)
mem[index] = index;
```

4 TO 1 MUX EXAMPLE

HDL Example 4.8

```
// Behavioral description of four-to-one line multiplexer

// Verilog 2001, 2005 port syntax

module mux_4x1_beh
( output reg  m_out,
  input       in_0, in_1, in_2, in_3,
  input [1: 0] select
);
  always @ (in_0, in_1, in_2, in_3, select)        // Verilog 2001, 2005 syntax
    case (select)
      2'b00:          m_out = in_0;
      2'b01:          m_out = in_1;
      2'b10:          m_out = in_2;
      2'b11:          m_out = in_3;
    endcase
  endmodule
```


HIERARCHIAL DESIGN

```
module bottom1(a, b, c);
input a, b;
output c;
reg c;
always
begin
c<=a & b;
end
endmodule
```

HIERARCHIAL DESIGN

```
module bottom2(l, m, n);
input l, m;
output n;
reg n;
always
begin
n<=l | m;
end
endmodule
```

HIERARCHIAL DESIGN

```
module top_ver (q, p, r, out);
input q, p, r;
output out;
reg out, intsig;
bottom1 u1(.a(q), .b(p), .c(intsig));
bottom2 u2(.l(intsig), .m(r), .n(out));
endmodule
```