



數位IC設計

RTL Coding – Part I



Optimization issues for VLSI Design

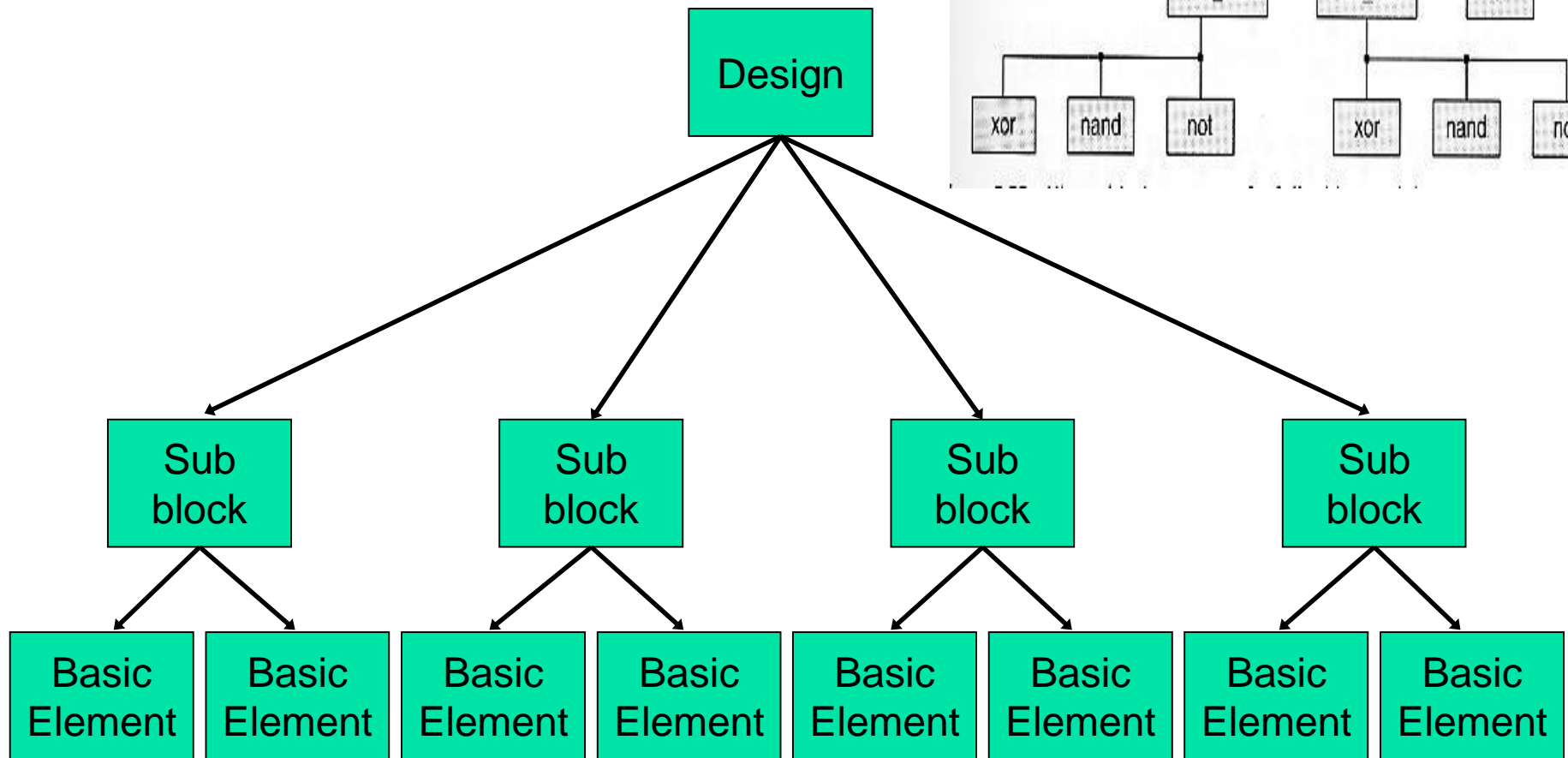
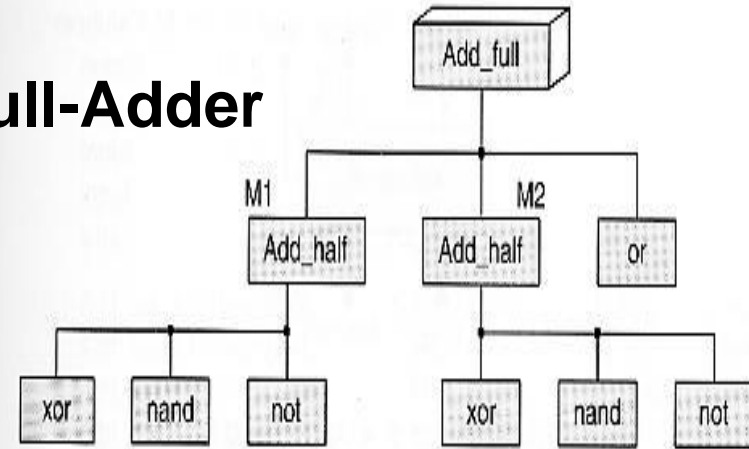
Five optimization issues:

1. Area (less silicon, less cost, high yield)
2. Speed (design constraint, better timing performance)
3. Power dissipation (cooling, battery)
4. Testability (minimize the time spent to test a single chip)
5. Design time (CAD tools)

Done by
designers
+ tools

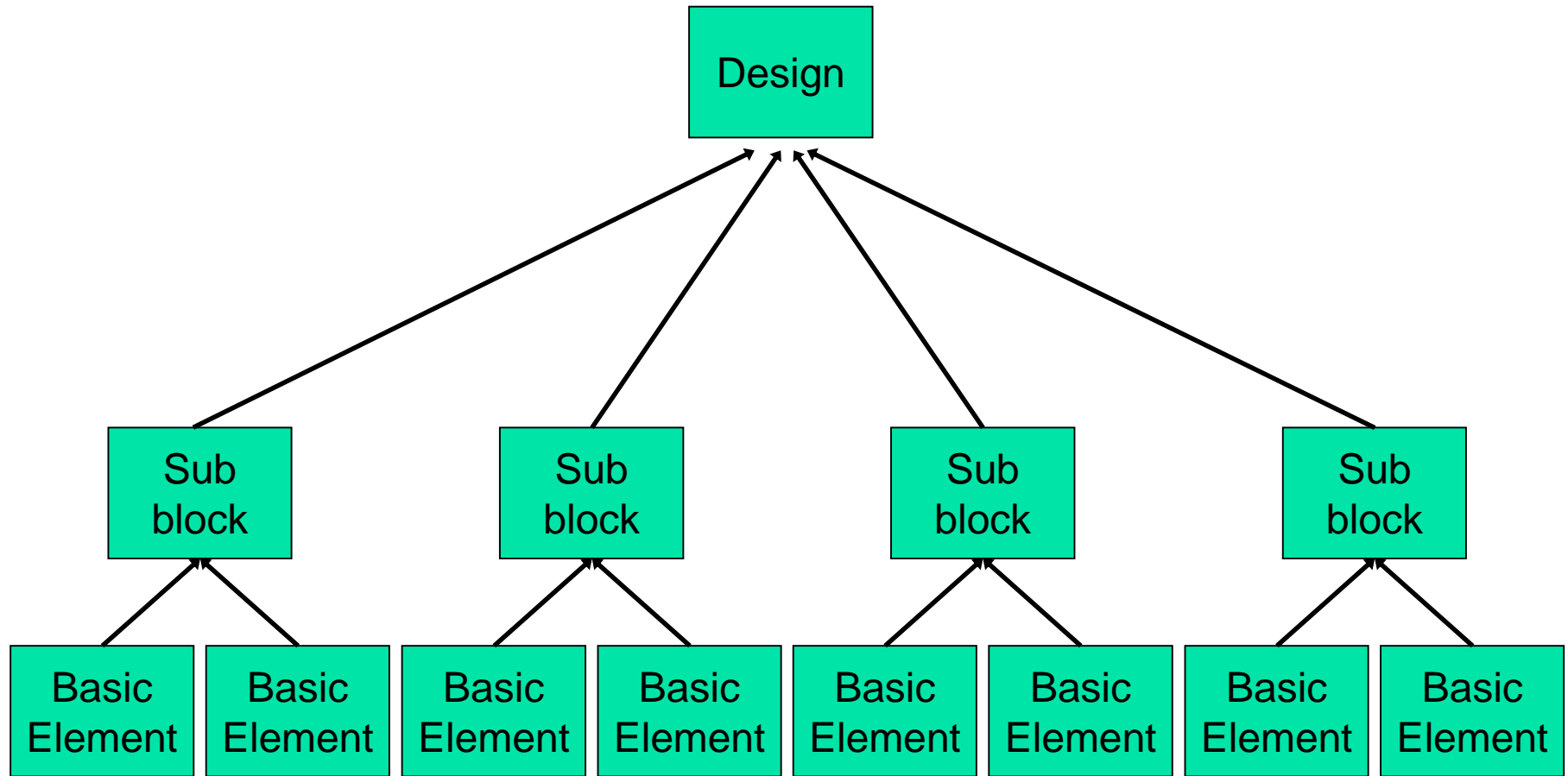
Top-Down Design Flow

Full-Adder





Bottom-Up Design Flow





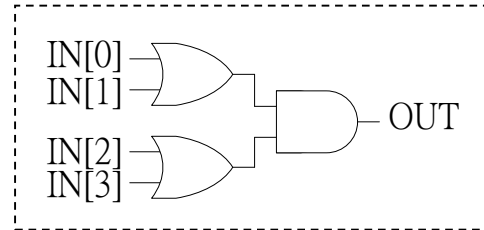
Design Entry for VLSI System

Choose the design entry method:

Schematic

Gate level design

Intuitive & easy to debug



HDL (Hardware Description Language)

Descriptive & portable

Easy to modify

```
always @(IN)
begin
    OUT = (IN[0] | IN[1]) &
          (IN[2] | IN[3]);
end
```

Mixed HDL & Schematic

...



Hardware Description Language (HDL)

- Hardware description language allows you to describe circuit at different levels of abstractions, and allows you to mix any level of abstraction in the design
- Two of the most popular HDLs
 - Verilog
 - VHDL
- HDLs can be used for both the cell-based synthesis and FPGA/CPLD implementation
- Only Verilog is introduced here



Why Verilog?

Verilog History

1. Verilog was written by gateway design automation in the early 1980
2. Cadence acquired gateway in 1990
3. Cadence released Verilog to the public domain in 1991
4. In 1995, the language was ratified as IEEE standard 1364

Why Verilog ?

1. Choice of many design teams
2. Most of us are familiar with C- like syntax/semantics



Verilog Features

Features:

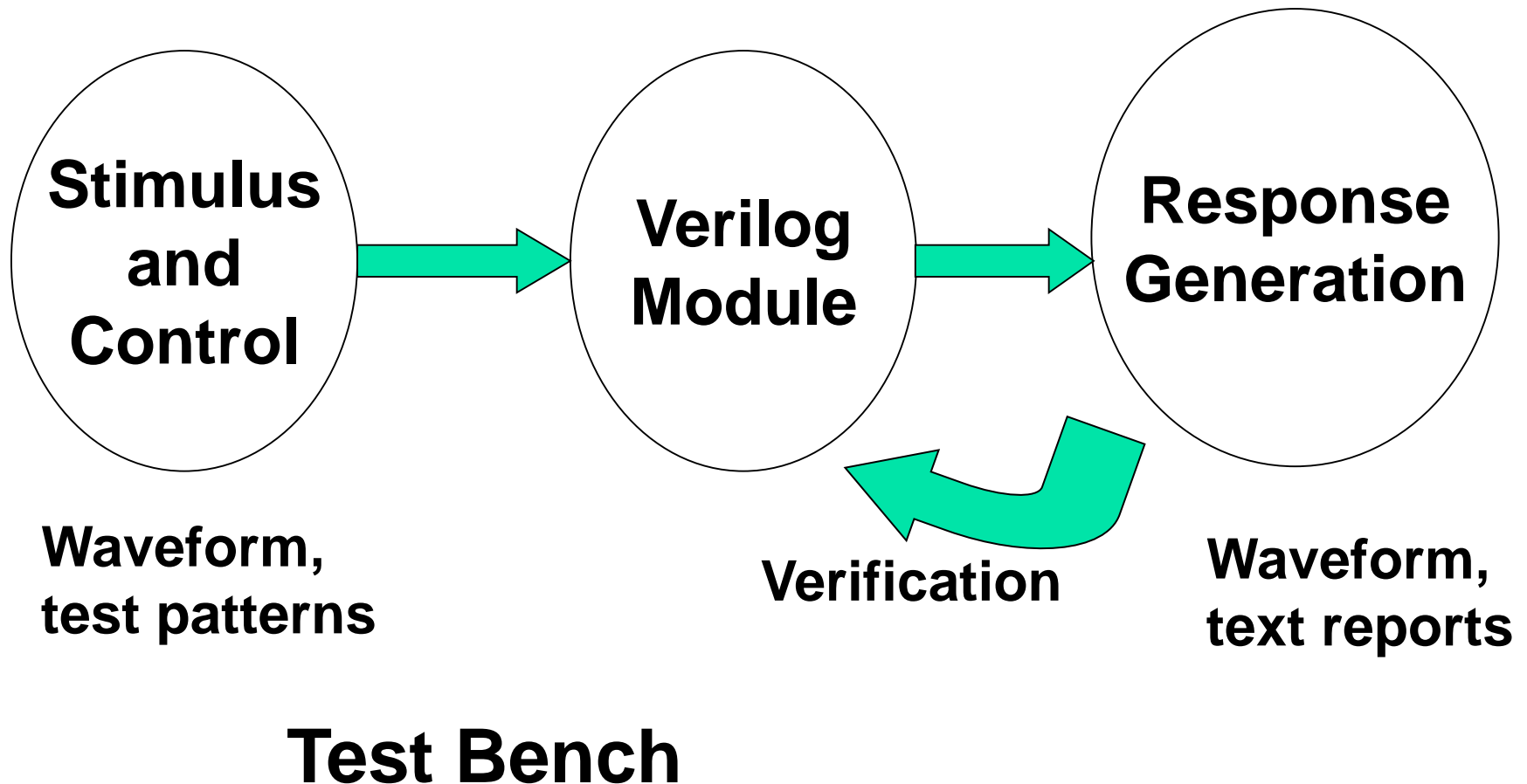
- Procedural constructs for conditional, if-else, case and looping operations
- Arithmetic, logical, bit-wise, and reduction operations for expression
- Timing control

Basics of Verilog Language:

- | | | |
|--------------------|--------------|------------|
| - Verilog Module | - Identifier | - Keyword |
| - Four Value Logic | - Data Types | - Numbers |
| - Port Mapping | - Operator | - Comments |



Verilog Module (1/3)



Verilog Module (2/3)

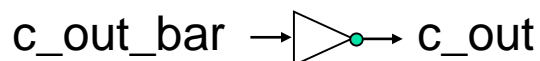
module module_name (port_name);

(1) port declaration

(2) data type declaration

(3) module functionality or structure

endmodule



```
module Add_half(sum, c_out, a, b);  
(1)  input      a, b;  
     output    sum, c_out;  
  
(2)  wire      c_out_bar;  
  
     xor        (sum, a, b);  
(3)  nand      (c_out_bar, a, b);  
     not        (c_out, c_out_bar);  
  
endmodule
```



Verilog Module (3/3)

Verilog Module: basic building block

```
module DFF
```

```
-----  
-----  
-----
```

```
  -  
  -  
  -
```

```
-----  
-----
```

```
endmodule
```

```
module ALU
```

```
-----  
-----  
-----
```

```
  -  
  -  
  -
```

```
-----  
-----
```

```
endmodule
```

```
module MUX
```

```
-----  
-----  
-----
```

```
  -  
  -  
  -
```

```
-----  
-----
```

```
endmodule
```

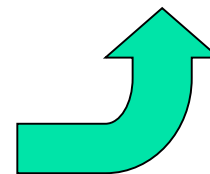
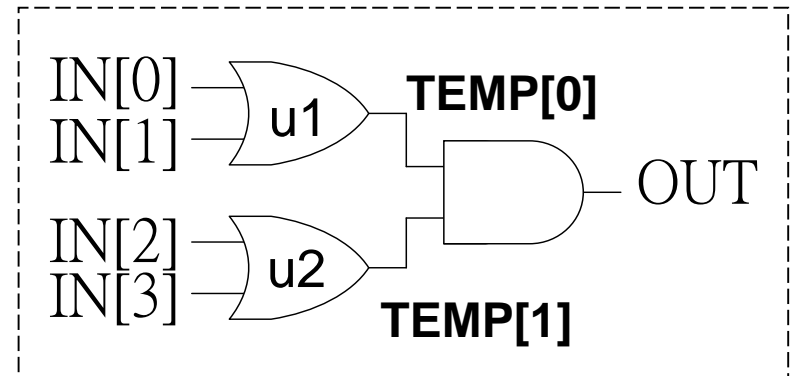
Structural Description

Verilog allows three kinds of descriptions for circuits:

- (1) Structural description
- (2) Data flow description
- (3) Behavioral description

Structural description:

```
1. module OR_AND_STRUCTURAL(IN,OUT);  
  
2.   input      [3:0]    IN;  
3.   output          OUT;  
4.   wire        [1:0]    TEMP;  
  
5.   or u1(TEMP[0], IN[0], IN[1]);  
6.   or u2(TEMP[1], IN[2], IN[3]);  
7.   and (OUT, TEMP[0], TEMP[1]);  
8. endmodule
```



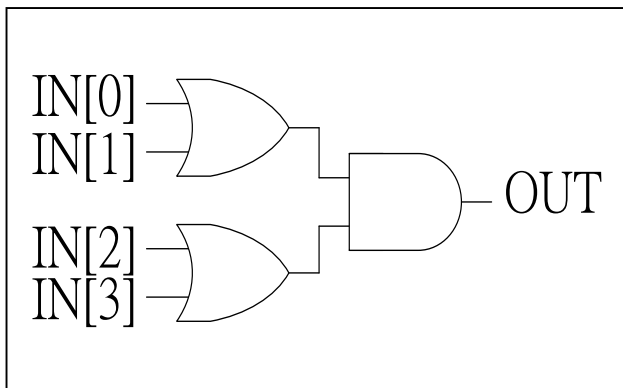
***Synthesized (synthesis) +
optimized by tools***

Data Flow Description

Data flow description

```
1. module OR_AND_DATA_FLOW(IN, OUT);  
2.   input      [3:0]      IN;  
3.   output                                OUT;  
  
4.   assign OUT = (IN[0] | IN[1]) & (IN[2] | IN[3]);  
  
endmodule
```

***Synthesized and
optimized by tools***



NOTE:

What is the difference between C and Verilog ?

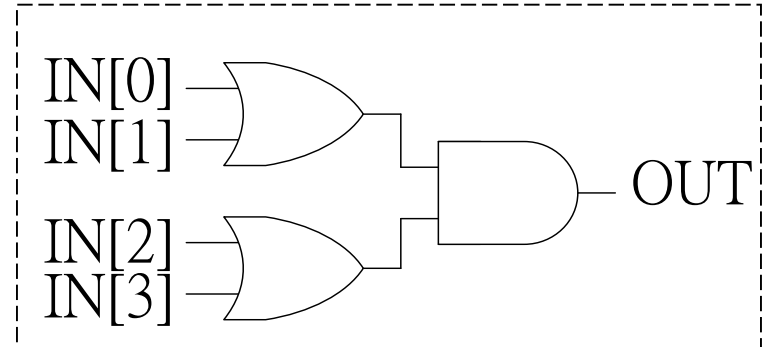
C : only one iteration (once) is implemented for assignment

Verilog : hard-wired circuit for assignment

Behavioral (RTL) Description (1/2)

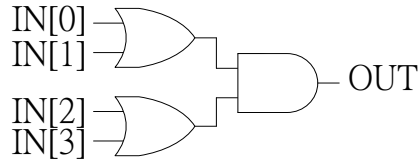
Behavioral description #1

```
1. module OR_AND_BEHAVIORAL(IN, OUT);  
2.   input [3:0]   IN;  
3.   output        OUT;  
4.   reg           OUT;  
  
5.   always @(IN)  
6.   begin  
7.     OUT = (IN[0] | IN[1]) & (IN[2] | IN[3]);  
8.   end  
9. endmodule
```



**Activate OUT while any voltage transition
(0→1 or 1→0) happens at signal IN**

Behavioral (RTL) Description (2/2)



Truth Table

IN[0]	IN[1]	IN[2]	IN[3]	OUT
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Behavioral description #2

```
module or_and(IN, OUT);
```

```
input [3:0] IN; output OUT; reg OUT; (Note)
```

```
always @(IN)
```

```
begin
```

```
case(IN)
```

```
4'b0000: OUT = 0; 4'b0001: OUT = 0;
```

```
4'b0010: OUT = 0; 4'b0011: OUT = 0;
```

```
4'b0100: OUT = 0; 4'b0101: OUT = 1;
```

```
4'b0110: OUT = 1; 4'b0111: OUT = 1;
```

```
4'b1000: OUT = 0; 4'b1001: OUT = 1;
```

```
4'b1010: OUT = 1; 4'b1011: OUT = 1;
```

```
4'b1100: OUT = 0; 4'b1101: OUT = 1;
```

```
4'b1110: OUT = 1; default: OUT = 1;
```

```
endcase
```

```
end
```

```
endmodule
```

***Synthesized and
optimized by tools***



Verilog Primitives

Combinational Logic	Three State	MOS Gates	CMOS Gates	Bi-Directional Gates	Pull Gates
and nand or nor xor xnor buf not	bufif0 bufif1 notif0 notif1	nmos pmos rnmos rpmos	cmos rcmos	tran tranif0 tranif1 rtran rtranif0 rtranif1	pullup pulldown

Note: all primitives are simulatable (可模擬)

but not all synthesizable (可合成)



Instance Name

- A module instance must have a name.

ex: OR_AND_STRUCTURAL

Note: naming skill is very important in Verilog

- The use of an instance name with a primitive is optional.

ex: u1, u2

```
or u1(TEMP[0], IN[0], IN[1]);
```

```
or u2(TEMP[1], IN[2], IN[3]);
```

```
and (OUT, TEMP[0], TEMP[1]);
```



Structural Description for Cell-Based Implementation

Structural description (cell-based):

```
module OR_AND_STRUCTURAL(IN,OUT);
```

```
input          [3:0]    IN;
```

```
output          OUT;
```

```
wire           [1:0]    TEMP;
```

```
orf203  u1(TEMP[0], IN[0], IN[1]);
```

```
orf203  u2(TEMP[1], IN[2], IN[3]);
```

```
andf201 (OUT, TEMP[0], TEMP[1]);
```

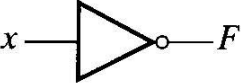
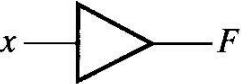
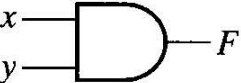
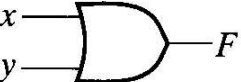
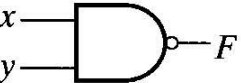



```
endmodule
```

Cell library from:
IBM, TSMC, UMC,...

This kind of design is not portable. Why ?

Bad design method !

Gate Delay (1/3)

NAME	GRAPHIC SYMBOL	FUNCTIONAL EXPRESSION	COST (NUMBER OF TRANSISTORS)	GATE DELAY (NS)
Inverter		$F = x'$	2	1
Driver		$F = x$	4	2
AND		$F = xy$	6	2.4
OR		$F = x + y$	6	2.4
NAND		$F = (xy)'$	4	1.4
NOR		$F = (x + y)'$	4	1.4
XOR		$F = x \oplus y$	14	4.2
XNOR		$F = x \odot y$	12	3.2

1. Cost (# of trans.)

2. Delay

Gate delay is dependent on the VLSI technology and the cell library.

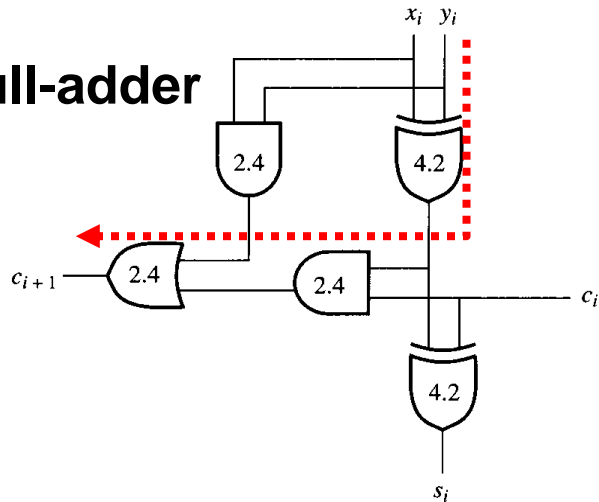
Technology Mapping:

Convert the expression $a'b' + c(a+b)$ into a logic schematic using gates defined in the cell library

Translation + Optimization

Gate Delay (2/3)

Full-adder

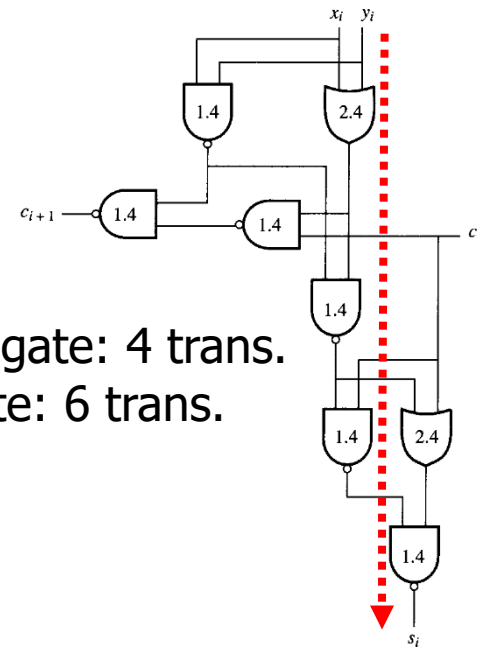


and gate: 6 trans. xor gate: 14 trans.

Totally, 46 transistors

INPUT/OUTPUT PATH	DELAY (ns)
c_i to c_{i+1}	4.8
c_i to s_i	4.2
x_i, y_i to c_{i+1}	9.0
x_i, y_i to s_i	8.4

Delay (critical path)= 9.0 ns



nand gate: 4 trans.
or gate: 6 trans.

Totally, 36 transistors

INPUT/OUTPUT PATH	DELAY (ns)
c_i to c_{i+1}	2.8
c_i to s_i	3.8
x_i, y_i to c_{i+1}	5.2
x_i, y_i to s_i	7.6

Delay (critical path)

=7.6 ns

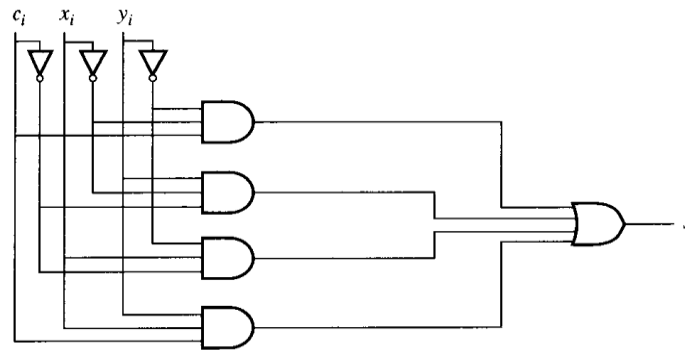
Critical Path: the longest path (delay) in a circuit

Gate Delay (3/3)

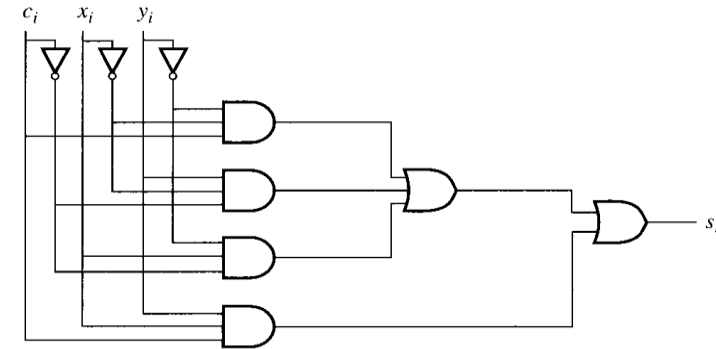
Technology Mapping (logic synthesis):
convert the expression into a logic
schematic using gates defined in the
cell library

$c_i \backslash x_i y_i$	0	01	11	10
0		1		1
1	1		1	

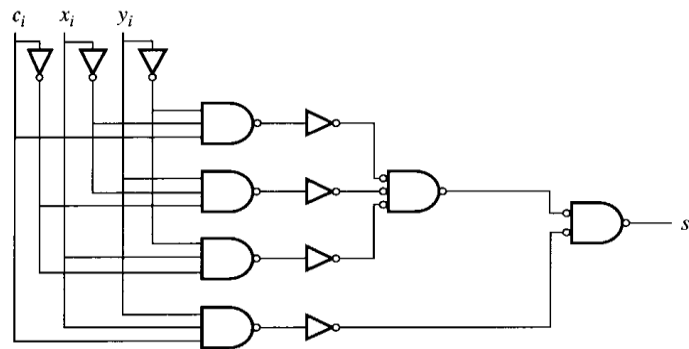
(a) Map representation



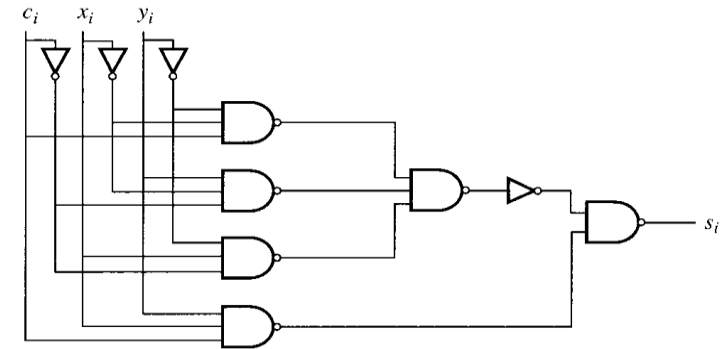
(b) AND-OR implementation



(c) OR gate decomposition



(d) Conversion to NAND network



(e) Optimized NAND network

Difference Between C and Verilog (1/2)

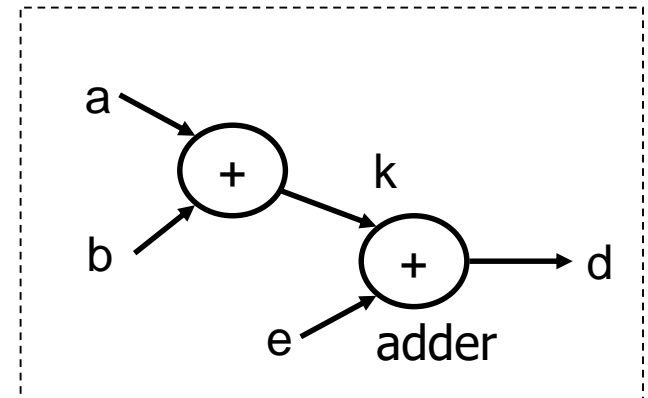
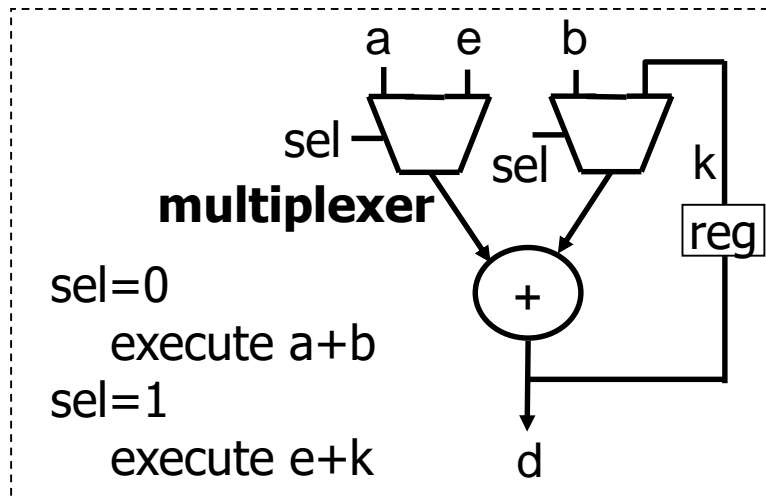
C language:

```
k=a+b;  
d=k+e;
```

Verilog language:

```
k=a+b;  
d=k+e;
```

or



Two adders
(higher cost, higher speed)

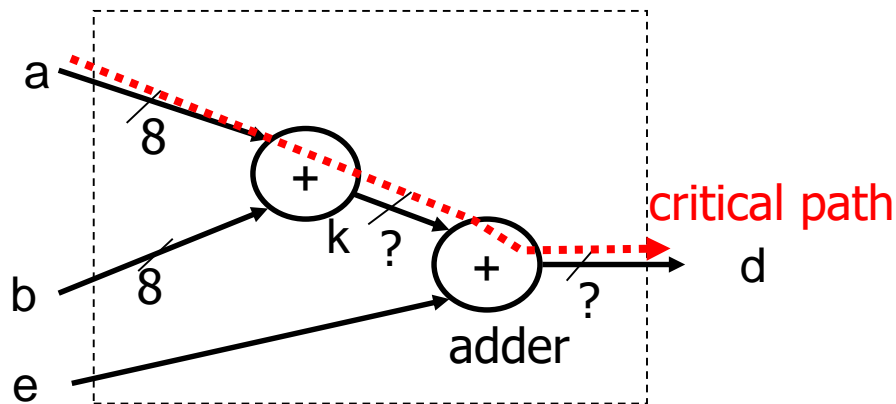
One adder
(lower cost, lower speed)
clock rate might be faster

C 語法中 變數k和d只被計算一次，欲多次計算需加上迴圈指令。

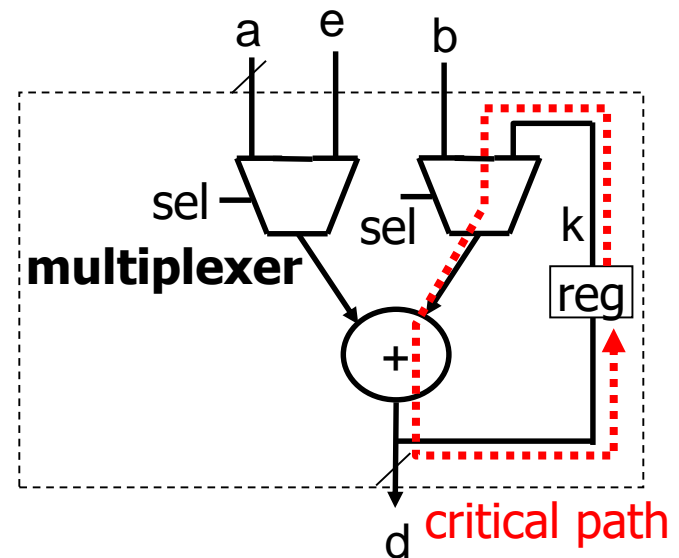
Verilog 語法中變數k和d需使用硬體元件來計算，感覺上該硬體永遠存在，只要輸入值有任何改變，相關聯的輸出會跟著改變（軟體指令為sequential process, 硬體則為一個彷彿永遠存在的實體）。

Difference Between C and Verilog (2/2)

1. 8-bit input wire, ?-bit adder, 2's complement
(The number of bits (pins) required in a hardware design)
2. How about the critical path ?



Critical path is longer, so the period is longer and the clock rate is slower



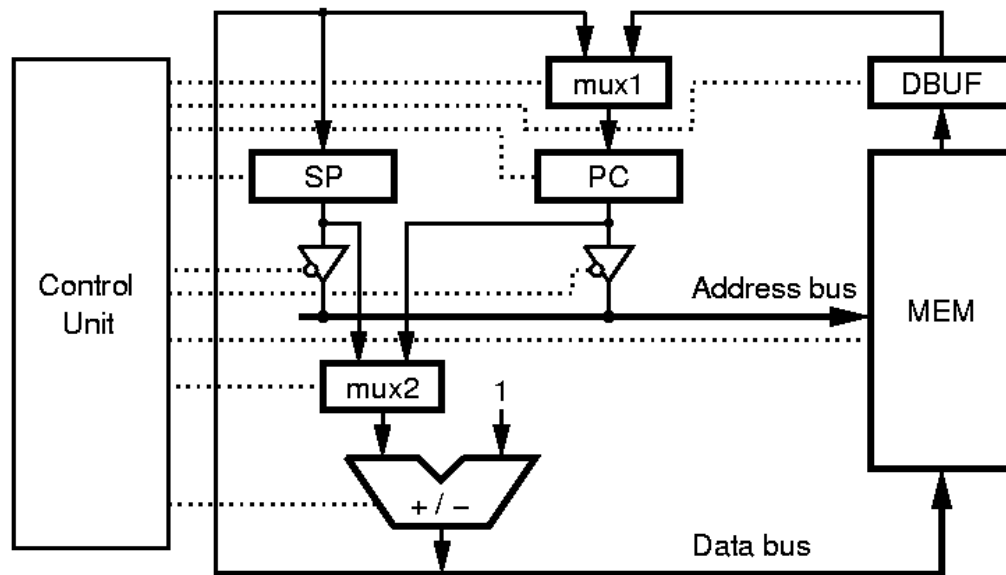
Critical path is shorter, so the period is shorter and the clock rate is faster

Clock rate = $1/\text{period}$ (If period is 15 ns, the clock rate is about 67 MHz)

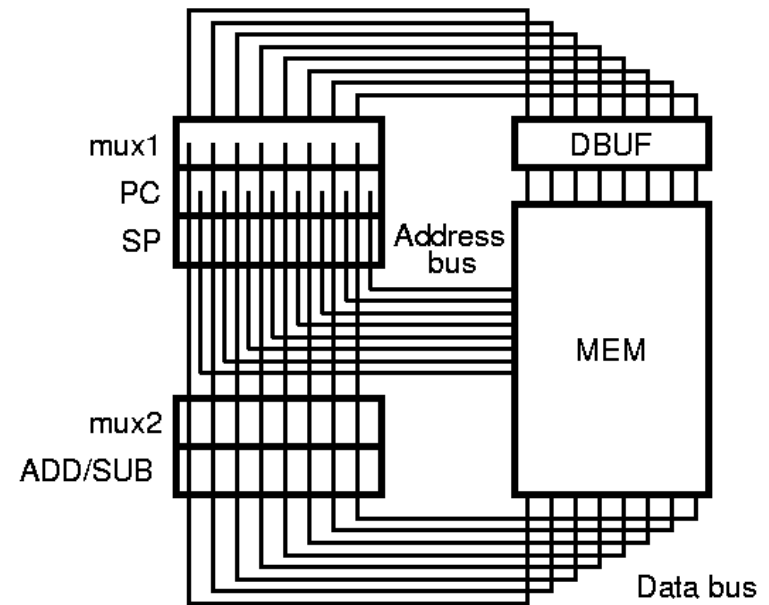
An Example

```
if IR(3) = '0' then
  PC      := PC + 1;
else
  DBUF    := MEM(PC);
  MEM(SP) := PC + 1;
  SP      := SP - 1;
  PC      := DBUF;
end if;
```

BEHAVIOR



STRUCTURE



FLOORPLAN



Identifier

- Identifiers are names given to Verilog objects
- Names of modules, ports and instances are all identifiers
- First character must use a letter, and other character can use letter, number or “_”
- Upper case and lower case letters are different
- How to determine a suitable name ???

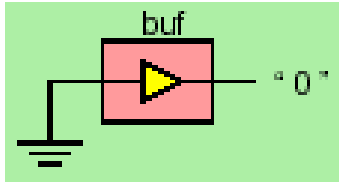


Keywords

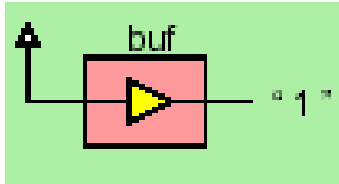
- Predefined identifiers to define the language constructs
- All keywords are defined in lower case
- Cannot be used as identifiers

Examples: module, initial, assign, always,
endmodule, ...

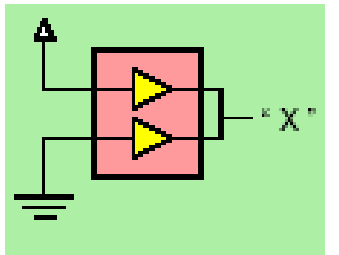
Four Value Logic



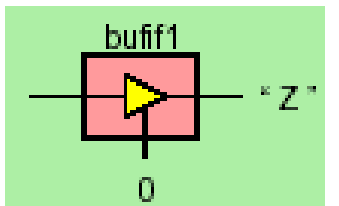
0: logic 0 / false



1: logic 1 / true



X: unknown logic value



Z: high-impedance



Four Value Logic: Example

- 6'hCA 001010 truncated, not 11001010
- 6'hA 001010 filled with two '0' on left
- 16'bZZ

Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z Z

filled with 16 Z's

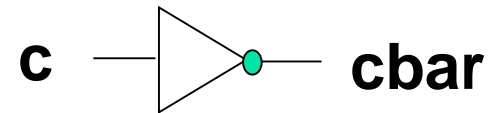
Timescale in Verilog

- The **'timescale** declares the time unit and its precision.

'timescale <time_unit> / <time_precision>

ex: **'timescale** 10 ns / 1 ns

delay= 20 ns



not #2 u1(cbar, c);

Delay=20 ns

or #2.54 u2(TEMP[1], IN[2], IN[3]);

Delay=25 ns

and # 3.55 (OUT, TEMP[0], TEMP[1]);

Delay=36 ns

Delay

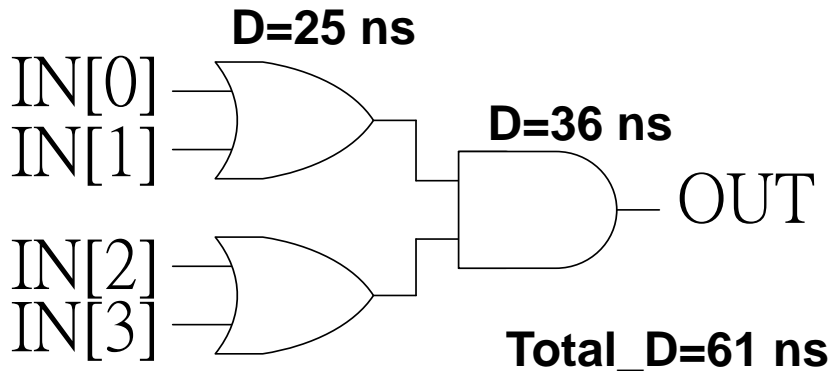
Simulation Delay

`'timescale 10 ns / 1 ns`

or `#2.54 u1(TEMP[0], IN[0], IN[1]);`

or `# 2.54 u2(TEMP[1], IN[2], IN[3]);`

and `#3.55 (OUT, TEMP[0], TEMP[1]);`



Physical Delay: *NOTE*

1. Physical delay can be acquired after synthesis process.
2. Physical delay is dependent on the VLSI technology and cell lib. (eg., 0.25, 0.18, 0.13,)
3. After synthesis, the instruction such as `# 3.55` becomes useless.
4. Timescale is used for simulation not for physical circuit.

If new `IN[3]` is activated at 10th ns, the simulated output `OUT` will be generated at 71th ns.

If new `IN[1]` is activated at 2th ns, the simulated output `OUT` will be generated at 63th ns.

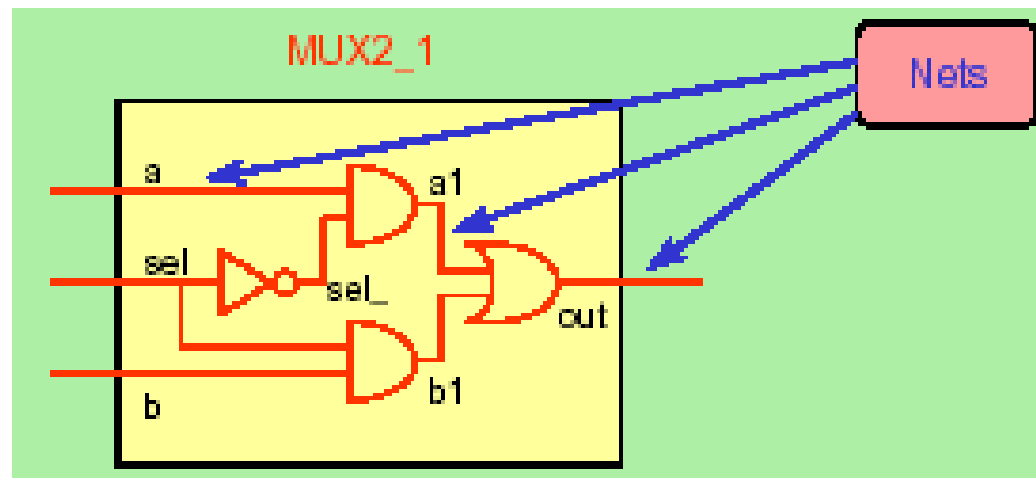


Data Types

- Nets
 - physical connection between devices
- Registers
 - abstract storage devices
- Parameters
 - run-time constants
- *The positions of three data types define whether they are global to a module or local to a particular always statement*
- *By default, net and register are one-bit wide (a scalar)
not multi-bit wide (a vector)*

NETs

- Connects between structural elements
- Must be continuously driven by
 - Continuous assignment
 - Module or gate instantiation
- Default initial value for a wire is “Z”





Types of Nets

■ Net declaration

`<nettype> <range>? <delay_spec>? <<net_name> <,net_name>*>`

Net Types	Functionality
wire, tri	for standard interconnection wires (default)
wor, trior	for multiple drivers that are Wire-ORed
wand, triand	for multiple drivers that are Wire-ANDed
triereg	for nets with capacitive storage
tri1	for nets which pull up when not driven
tri0	for nets which pull down when not driven
supply1	for power rails
supply0	for ground rails

Note: Some of those net types are un-synthesizable (不能電路合成的)

Nets (1/2)

wire, wand, wor, tri, supply0, supply1

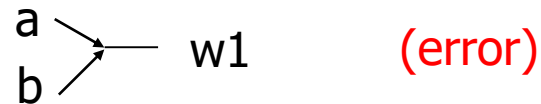
wire k; // single-bit wire

wire [0:31] w1, w2; // Two 32-bit wires

wire w1;

assign w1=a;

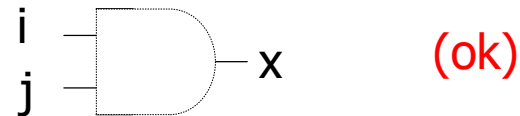
assign w1=b; (error)



Method 1:

wand x;

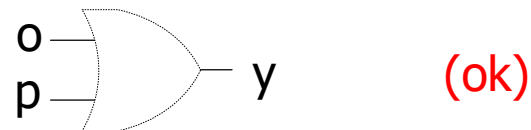
assign x=j; assign x=i;

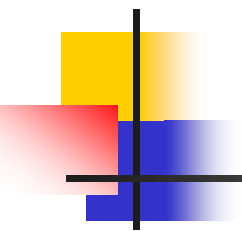


Method 2:

wor y;

assign y=o; assign y=p;





Nets (2/2)

tri: all variables that drive the tri must have a value of Z except one (ensured by the designer).

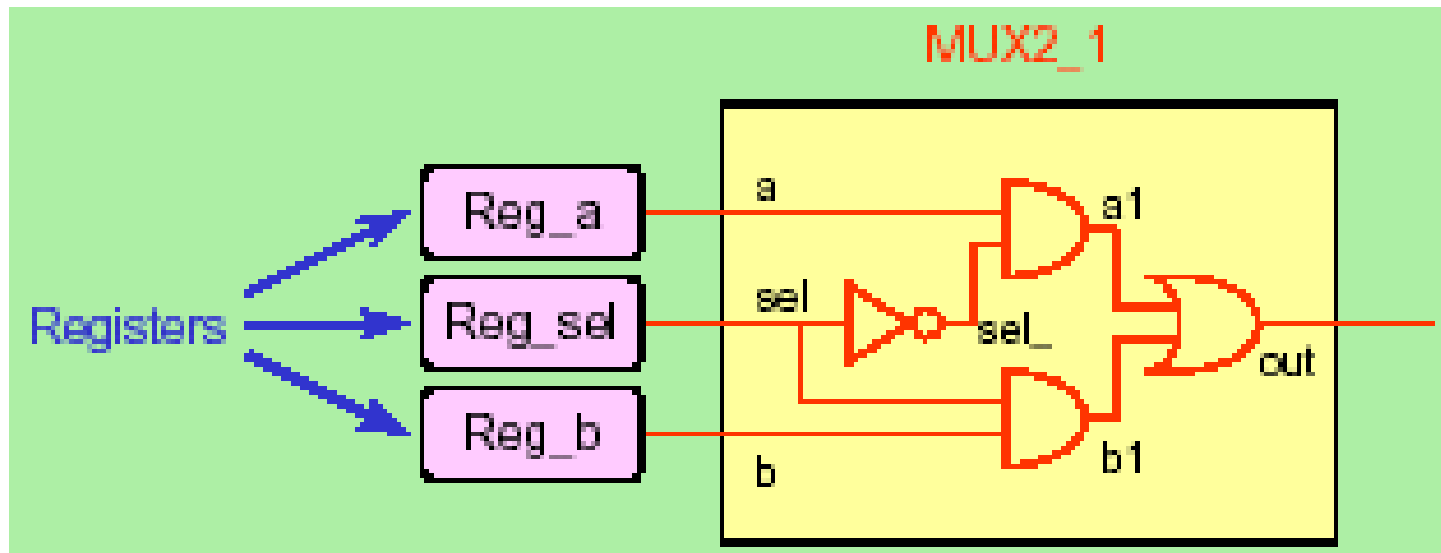
```
module tri-test(out, condition)
input [1:0] condition;  output out;
reg a, b, c;
tri out;
  always@(condition)
  begin
    a=1'bz; b=1'bz; c=1'bz;
    case (condition)
      2'b00: a=1'b1;
      2'b01: b=1'b0;
      2'b10: c=1'b1;
    endcase
  end
  assign out=a;  assign out=b; assign out=c;
endmodule
```

supply0 ➡ wires tied to logic 0 (ground)

supply1 ➡ wires tied to logic 1 (power)

Registers

- Represent abstract data storage elements
- Hold its value until a new value is assigned to it
- Registers are used extensively in behavioral modeling
- Default initial value for a register is “X”





Types Of Registers

- Register declaration

`<register_type> <range>? <<register_name> <,register_name>*>`

Register Types	Functionality
reg	Unsigned integer variable of varying bit width
integer	Signed integer variable, 32-bit wide. Arithmetic operations producing 2's complement results.
real	Signed floating-point variable, double precision
time	Unsigned integer variable, 64-bit wide

```
reg a;           // a scalar register
reg [3:0] b,c;   // two 4-bit vector registers
reg [7:0] byte_reg; // a 8-bit registers
reg [7:0] memory_block [255:0];
```

memory-block is an array of 256 registers, each of which is 8 bits width. You can access individual register, but you cannot access individual bits of register directly.

```
byte_reg=memory_block [120];
bit=byte_reg [7];           // wire bit;
```



Numbers (1/2)

- Numbers are integer or real constants.

Integer constants are written as

`<size>'<base format><number>`

- Real number can be represented in decimal or scientific format.
- A number may be **sized** or **unsized**



Numbers (2/2)

- The base format indicates the type of number
 - Decimal (d or D)
 - Hex (h or H)
 - Octal (o or O)
 - Binary (b or B)

ex: `usize`

`'h72ab`
base format number

size

`16'h72ab`
size base format number

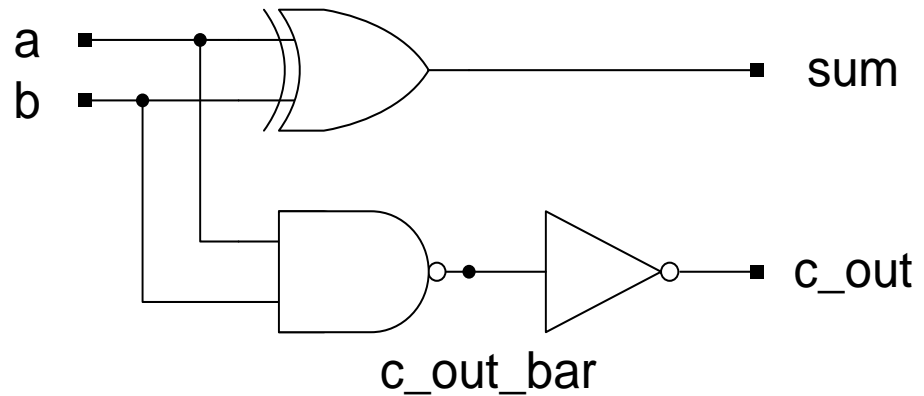
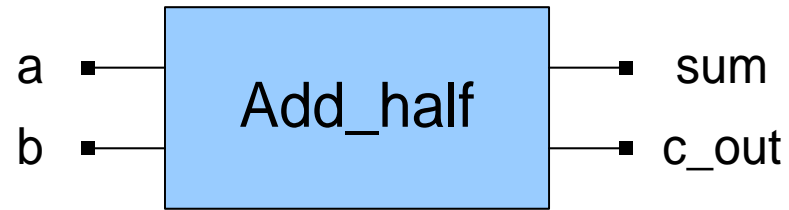
Half Adder (1/5)

a\b	0	1
0	0	1
1	1	0

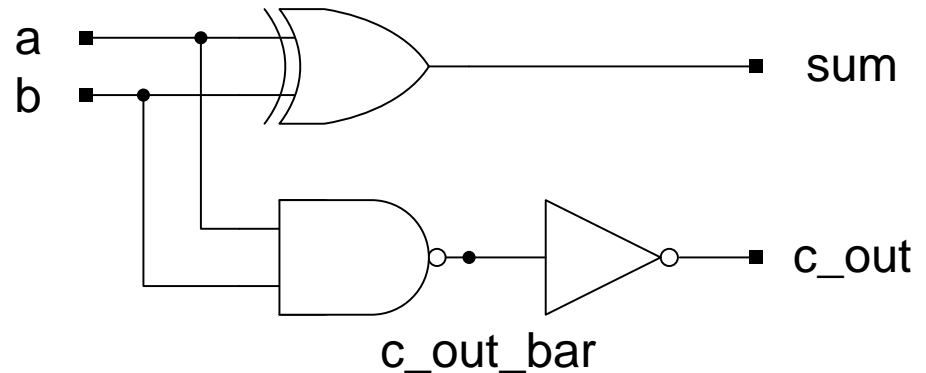
$$\text{sum} = a \oplus b$$

a\b	0	1
0	0	0
1	0	1

$$\text{c_out} = ab$$



Half Adder (2/5)



Structural description

```
module Add_half(sum, c_out, a, b);
```

```
  input      a, b;
```

```
  output    sum, c_out;
```

```
  wire      c_out_bar;
```

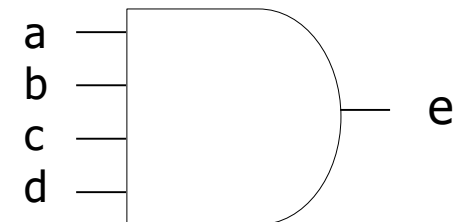
```
  xor       (sum, a, b);
```

```
  nand      (c_out_bar, a, b);
```

```
  not       (c_out, c_out_bar);
```

```
endmodule
```

```
and (e, a, b,c,d);
```

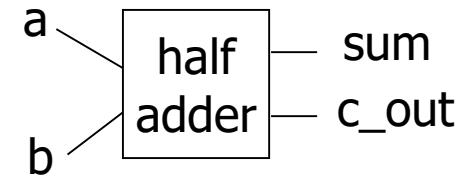


Half Adder (3/5)

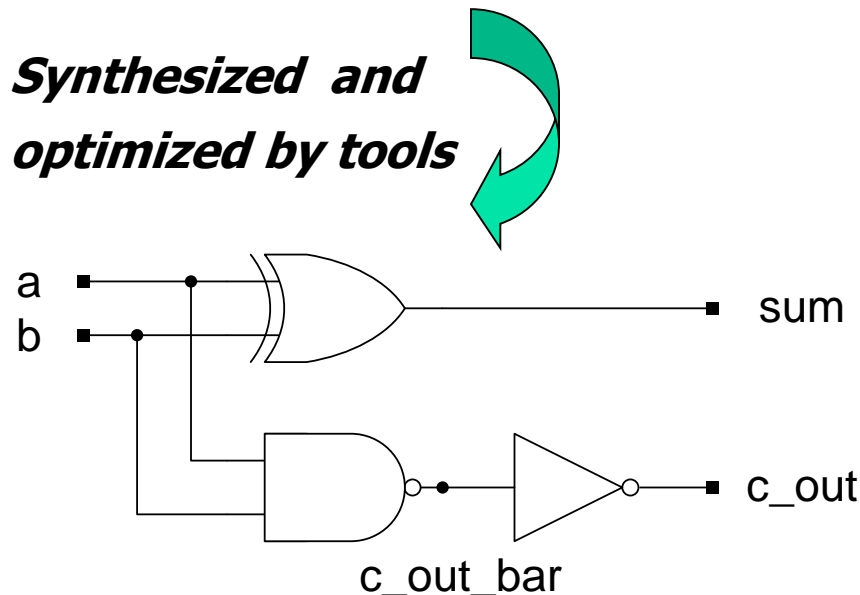
Data flow description

```
module Add_half(sum, c_out, a, b);  
  input    a, b;  
  output   sum, c_out;  
  
  assign   {c_out, sum} = a + b;  
endmodule
```

assign: continuous assignment



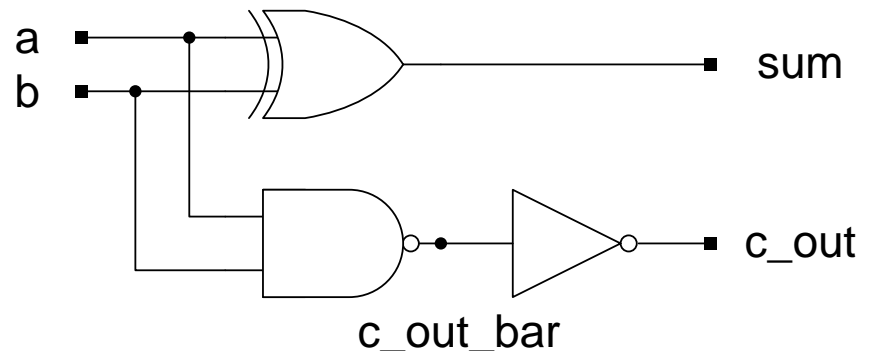
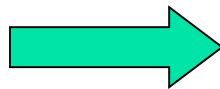
*Synthesized and
optimized by tools*



Half Adder (4/5)

Behavioral description #1

```
module Add_half(sum, c_out, a, b);  
    input    a, b;  
    output   sum, c_out;  
    reg      sum, c_out;  
  
    always @ (a or b)  
    begin  
        sum = a ^ b;  
        c_out = a & b;  
    end  
endmodule
```





Half Adder (5/5)

Behavioral description #2

```
module Add_half(sum, c_out, a, b);  
input  a, b;  
output sum, c_out;  
reg    sum, c_out;  
always @(a or b)  
begin  
    case({a,b})  
        2'b00:begin  
            sum = 0; c_out = 0;  
        end  
        2'b01:begin  
            sum = 1; c_out = 0;  
        end  
    endcase  
end  
endmodule
```

```
2'b10:begin  
    sum = 1; c_out = 0;  
end  
default:begin  
    sum = 0; c_out = 1;  
end  
endcase  
end  
endmodule
```

a\b	0	1	sum
0	0	1	
1	1	0	

a\b	0	1	c_out
0	0	0	
1	0	1	



Parameter

- Parameter declaration

`parameter<range>?<list_of_assignments>`

- You can use a parameter anywhere that you can use a literal.

ex: `module mod(ina, inb,out);`

`.....`

`parameter m1=8;`

`....`

`wire [m1:0] w1;`

`.....`

`endmodule`

w1 can be set as a (n+1)-bit wire if

we change m1 to n

(i.e., m1=10 ➡ w1 becomes a 11-bit wire

m1=4 ➡ w1 becomes a 5-bit wire)



Parameterized Design (1/2)

```
module PARAM(A , B , C);  
input [7 : 0] A , B;  
output [7 : 0] C;  
wire f;  
    or    o1(f,A,B);  
    test  #(4)  u1(A , f , C);  
endmodule
```

```
module test (a , b , c);  
parameter width = 8;  
input [width - 1 : 0] a, b;  
output [width - 1 : 0] c;  
  
    assign c = a & b;  
  
endmodule
```

Override the value of width when the test module is instantiated

Save the file as PARAM.v and compile (synthesis) it

 **the width value become 4**



Parameterized Design (2/2)

```
module PARAM_1(A , B , C , D);  
input [4 : 0] A;  
input [3 : 0] B;  
input [3 : 0] C;  
output [5 : 0] D;  
  
test_2  #(5 , 4 , 4)  u1(A , B , C , D);  
  
endmodule
```

```
module test_2(A , B , C , D);  
parameter width = 8;  
parameter height = 8;  
parameter length = 8;  
  
input [width - 1 : 0] A;  
input [height - 1 : 0] B;  
input [length- 1 : 0] C;  
output [width : 0] D;  
  
assign D = A + B + C;  
  
endmodule
```

Override those values of many parameters when the test_2 module is instantiated (width = 5; height = 4; length = 4)



Port Mapping

- In Order

```
Mux      Mux_1(Sel,x,y,Mux_Out);
```

```
Register8 Register8_1(Clock,Reset,Mux_Out,Reg_Out);
```

- By Name

```
Mux      Mux_1(.Sel(Sel),.x(x),.y(y),.out(Mux_Out));
```

```
Register8 Register8_1(.Clock(Clock), .Reset(Reset) ,.data(Mux_Out)  
                    ,.q(Reg_Out));
```




Port Mapping by Position Association

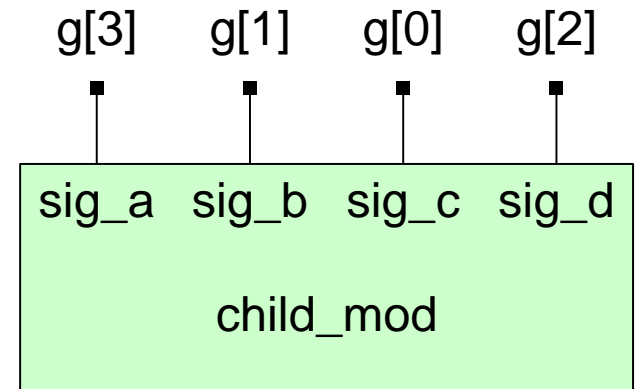
```
module parent_mod;  
  wire      [3:0] g;
```

```
  child_mod G1(g[3], g[1], g[0], g[2]);  
endmodule
```

```
module child_mod(sig_a, sig_b, sig_c, sig_d);  
  input      sig_c, sig_d;  
  output     sig_a, sig_b;
```

```
  module description  
endmodule
```

in-order port mapping





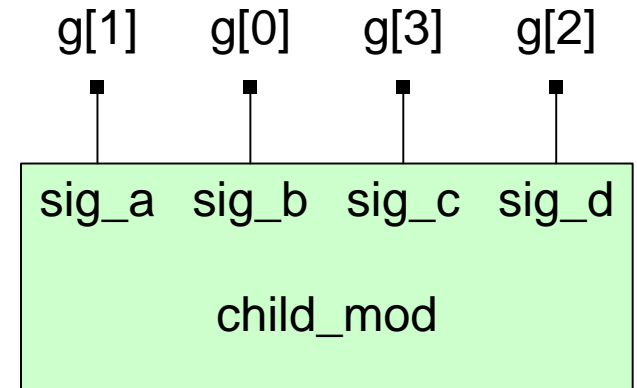
Port Mapping by Name Association

Name mapping is better, why ?

```
module parent_mod;  
  wire      [3:0] g;  
  
  child_mod G1(.sig_c(g[3]), .sig_d(g[2]), .sig_b(g[0]),  
               .sig_a(g[1]));  
  
endmodule
```

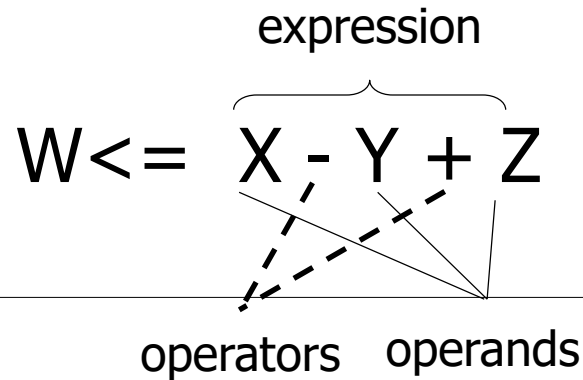
```
module child_mod(sig_a, sig_b, sig_c, sig_d);  
  input      sig_c, sig_d;  
  output     sig_a, sig_b;  
  
  module description  
  
endmodule
```

naming port mapping



Expressions

- An expression comprises of operators and operands, see Example, and are covered separately in the following two sections.
- Example





Verilog Operands

- Four data objects form the operands of an expression

Verilog Operands	
Identifiers	
Literals	
string (bit & character)	4'b 1101
numeric (integer, real ⁺)	34
Function Call	
Index & Slice Names	

Identifiers

- Verilog identifiers consists of letters, digits, underscores (_) and dollar sign (\$)
- Verilog is case sensitive, so upper and lower case identifier names are treated as different identifiers

Identifier and Literal Operands (1/2)

```
1. module LITERALS(A1, A2, B1, B2, Y1, Y2);  
2.     input  A1, A2, B1, B2;  
3.     output [7:0] Y1; output [5:0] Y2;  
4.     parameter CST = 4'b 1010, TF=25;  
5.     reg [7:0] Y1; reg [5:0] Y2;
```

```
6. always @(A1 or A2 or B1 or B2)
```

```
7. begin
```

```
8.     if (A1 == 1)
```

```
9.         Y1 = {CST, 4'b 0000};
```

```
10.    else if (A2 == 1)
```

```
11.        Y1 = {CST, 4'b 0101};
```

```
12.    else
```

```
13.        Y1 = {CST, 4'b 1111};
```

```
14.        if (B1 == 0) Y2 = 10;
```

```
15.        else if (B2 == 1) Y2 = 15;
```

```
16.        else Y2 = TF +10 +15;
```

```
17.    end
```

```
18. endmodule
```

Identifier

Bit string literals

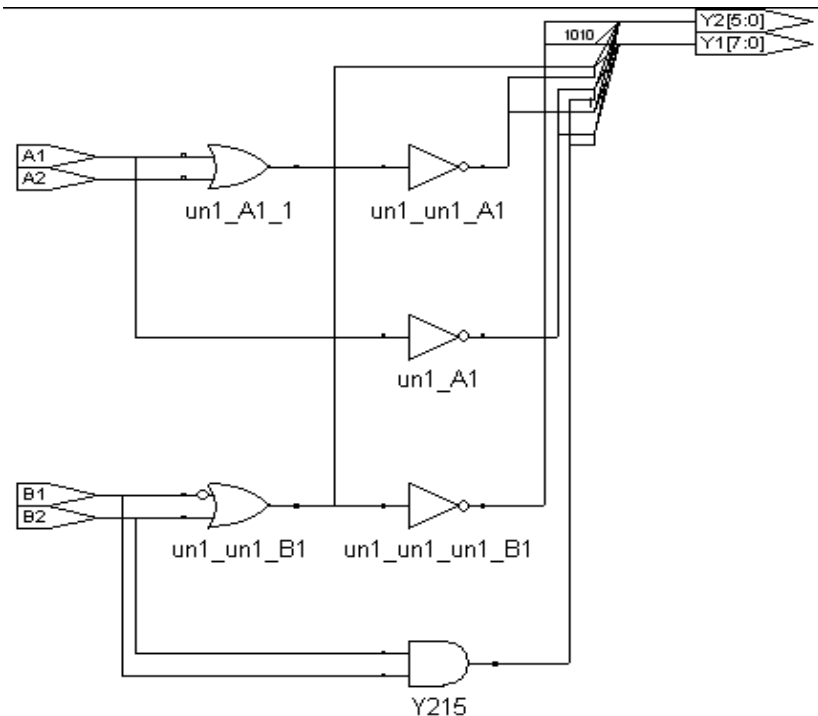
Identifier

Numeric (integer) literal

Identifier and Literal Operands (2/2)

```

parameter CST = 4'b 1010, TF=25;
if (A1 == 1)
    Y1 = {CST, 4'b 0000};
else if (A2 == 1)
    Y1 = {CST, 4'b 0101};
else
    Y1 = {CST, 4'b 1111};
if (B1 == 0) Y2 = 10;
else if (B2 == 1) Y2 = 15;
else Y2 = TF +10 +15;
    
```



A1	0	1	0	1
A2	0	0	1	1
B1	0	1	0	1
B2	0	0	1	1
Y1	<u>1010</u> 1111	<u>1010</u> 0000	<u>1010</u> 0101	<u>1010</u> 0000
Y2	001010 ₂ (10)	110010 ₂ (50)	001010 ₂ (10)	001111 ₂ (15)



Function Call Operands (1/4)

- Function calls, which must reside in an expression, are operands. The single value returned from a function is the operand value used in the expression.

```
1. module FUNCTION_CALLS (A1, A2, A3, A4, B1, B2, B3, B4, Y1, Y2);
2.     input    A1, A2, A3, A4, B1, B2, B3, B4; output [2:0] Y1, Y2;
3.     reg [2:0] Y1, Y2;

4.     function [2:0] Fn1;
5.         input    F1, F2, F3;
6.         begin
7.             Fn1 = F1+F2+F3;
8.         end
9.     endfunction

10.    always @(A1 or A2 or A3 or A4 or B1 or B2 or B3 or B4)
11.        begin
12.            Y1 = Fn1(A1, A2, A3)+A4;
13.            Y2 = Fn1(B1, B2, B3)-B4;
14.        end
15. endmodule
```

Function call operand

Function Call Operands (2/4)

```
function [2:0] Fn1;  
  input F1, F2, F3;  
  begin  
    Fn1 = F1+F2+F3;  
  end  
endfunction
```

```
always @(A1 or A2 or A3 or A4 or B1 or  
B2 or B3 or B4)  
  begin  
    Y1 = Fn1(A1, A2, A3)+A4;  
    Y2 = Fn1(B1, B2, B3)-B4;  
  end
```

A1	0	0	1	1
A2	0	0	1	1
A3	0	0	0	0
A4	0	1	0	1
B1	0	1	0	1
B2	0	0	0	0
B3	0	0	1	1
B4	0	0	1	1
Y1[2:0]	000	001	010	011
Y2[2:0]	000	001	000	001

2's complement is implemented: A-B

(1) A加上B的2補數

(2) a.若有進位(carry),代表結果為正, 去掉進位,剩下即為結果
b.若無進位(carry),代表結果為負, 取結果的2補數,在前面加上負號

1-1

2-1

001

+111

1000

010

+111

1001

Function Call Operands (3/4)

```

1.  module FUNCTION_CALLS (A1, A2, A3, A4, Y1, Y2);
2.      input  A1, A2, A3, A4;  output [4:0] Y1, Y2;
3.      reg [4:0] Y1, Y2;
4.      function [4:0] Fn1;
5.          input  F1, F2, F3, F4;
6.          begin
7.              Fn1 = F1+F2+F3+F4;
8.          end
9.      endfunction

```

```

10. function [4:0] Fn2;
11.     input  F1, F2, F3, F4;
12.     begin
13.         Fn2 = (F1+F2)+(F3+F4);
14.     end
15. endfunction

```

```

always @(A1 or A2 or A3 or A4)
begin
    Y1 = Fn1(A1, A2, A3, A4)+10;
    Y2 = Fn2(A1, A2, A3, A4)-5;
end
endmodule

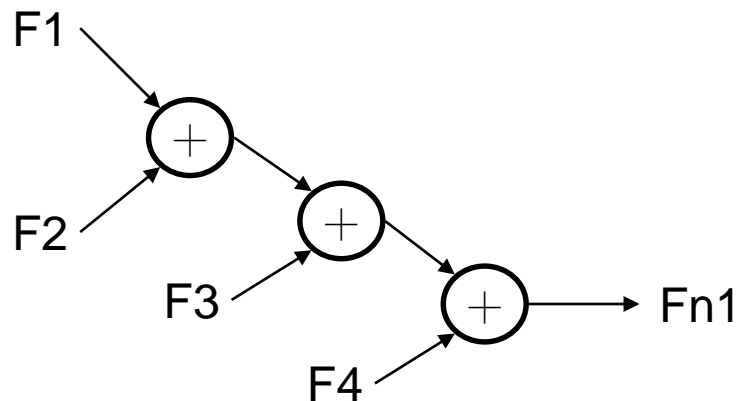
```

A1	0	0	0	1
A2	0	0	0	1
A3	0	0	1	0
A4	0	1	0	0
Y1[4:0]	01010	01011	01011	01100
Y2[4:0]	11011	11100	11100	11101

$00000 + 11011 (-5) = 11011 \rightarrow -00100 + 1 = -00101 (-5)$
 $00001 + 11011 (-5) = 11100 \rightarrow -00011 + 1 = -00100 (-4)$

Function Call Operands (4/4)

$$Fn1 = F1 + F2 + F3 + F4;$$

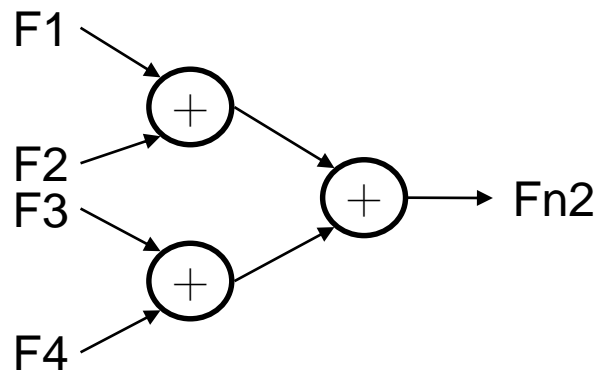


Longer delay (3 stages)

Why ?

What is the different ?

$$Fn2 = (F1 + F2) + (F3 + F4);$$



Shorter delay



Index and Slice Name Operands (1/2)

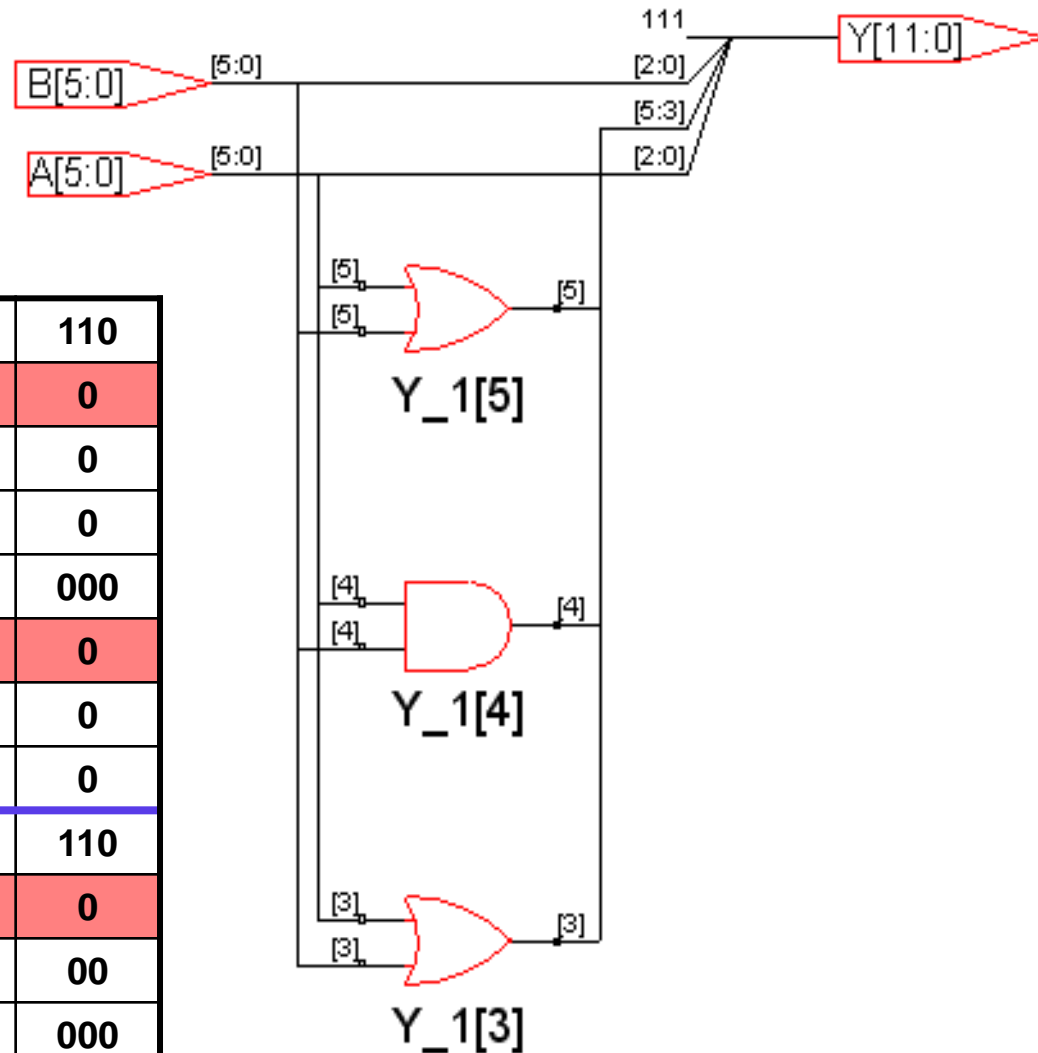
- Index operand specifies a single element of an array and slice operand specifies a sequence of elements within an array

```
1.  module INDEX_SLICE_NAME (A, B, Y);  
2.      input [5:0]      A, B;  
3.      output [11:0]Y;  
4.      parameter      C = 3'b111;  
5.      reg [11:0]      Y;  
  
6.      always @(A or B)  
7.      begin  
8.          Y[2:0]      = A[2:0];  
9.          Y[3]        = A[3] | B[3];  
10.         Y[5:4]      = {A[5] | B[5], A[4] & B[4]};  
11.         Y[8:6]      = B[2:0];  
12.         Y[11:9]     = C;  
13.      end  
14.  endmodule
```

Index and Slice Name Operands (2/2)

parameter $C = 3'b111$;
 $Y[2:0] = A[2:0];$ $Y[3] = A[3] \mid B[3];$
 $Y[5:4] = \{A[5] \mid B[5], A[4] \& B[4]\};$
 $Y[8:6] = B[2:0];$ $Y[11:9] = C;$

A[2:0]	000	010	100	101	110
A[3]	0	1	0	1	0
A[4]	0	1	0	1	0
A[5]	0	1	0	1	0
B[2:0]	110	111	101	001	000
B[3]	0	0	1	1	0
B[4]	0	0	1	1	0
B[5]	0	0	1	1	0
Y[0:2]	000	010	100	101	110
Y[3]	0	1	1	1	0
Y[5:4]	00	10	10	11	00
Y[8:6]	110	111	101	001	000
Y[11:9]	111	111	111	111	111





Operators (1/3)

- Operators perform an operation on one or more operands within an expression. An expression combines operands with appropriate operators to produce the desired function expression.

Name	Operator
bit-select or part-select	[]
parenthesis	()
Arithmetic Operators	
multiplication	*
division	/
addition	+
subtraction	-
modulus	%
Sign Operators	
identity	+
negation	-



Operators (2/3)

Name	Operator
Relational Operators less than less than or equal to greater than greater than or equal to	< <= > >=
Equality Operators logic equality logic inequality case equality case inequality	== != === !==
Logical Comparison Operators NOT AND OR	! &&
Logical Bit-Wise Operators unary negation NOT binary AND binary OR binary XOR binary XNOR	~ & ^ ^~ or ~^



Operators (3/3)

Name	Operator
Shift Operators logical shift left logical shift right	<< >>
Concatenation & Replication Operators concatenation replication	{ } {{ }}
Reduction Operators AND OR NAND NOR XOR XNOR	& ~& ~ ^ ^~ or ~^
Conditional Operator conditional	?:

Arithmetic Operators

```
1. module ARITHMETIC(A, B, Y1, Y2, Y3);
2.     input [2:0]      A, B;
3.     output [3:0]     Y1;
4.     output [4:0]     Y3;
5.     output [3:0]     Y2;
6.     reg [3:0] Y1;
7.     reg [4:0] Y3;
8.     reg [3:0] Y2;
```

```
9.
10. always @(A or B)
11. begin
```

```
12.     Y1 = A + B;
13.     Y2 = A - B;
14.     Y3 = A * B;
15.     Y4 = A / B;
16.     Y5 = A % B; } (不建議)
```

```
17. end
18. endmodule
```

Arithmetic operators:

(1) +

(2) -

(3) *

(4) / (++)non-syn.

(5) % (++)non-syn.

A[2:0]	000	001	101	110
B[2:0]	101	111	100	001
Y1[3:0]	0101	1000	1001	0111
Y2[3:0]	1011 (-5)	1010(-6)	0001(+1)	0101(+5)
Y3[4:0]	00000	00111	10100	00110

Sign Operators

```
1. module SIGN(A, B, Y1, Y2);
2.     input [2:0]    A;
3.     input [2:0]    B;
4.     output [3:0]   Y1;
5.     output [3:0]   Y2;
6.     reg [3:0]      Y1;
7.     reg [3:0]      Y2;
```

Sign operators:
(1) +
(2) -

```
8.     always @(A or B)
9.     begin
10.         Y1 = A + -B;
11.         Y2 = -A + B;
12.     end
13. endmodule
```

A[2:0]	000	001	100	101	110	111	000
B[2:0]	110	101	100	010	000	001	101
Y1[3:0]	1010 (-6)	1100 (-4)	0000 (0)	0011 (3)	0110 (6)	0110 (6)	1011 (-5)
Y2[3:0]	0110 (6)	0100 (4)	0000 (0)	1101 (-3)	1010 (-6)	1010 (-6)	0101 (5)

Relational Operators

```
1.  module RELATIONAL_OPERATORS(A, B, Y);
2.      input [2:0]      A;
3.      input [2:0]      B;
4.      output [3:0]     Y;
5.      reg [3:0] Y;

6.      always @(A or B)
7.      begin
8.          Y[0] = A > B;
9.          Y[1] = A >= B;
10.         Y[2] = A < B;
11.         if ( A <= B)
12.             Y[3] = 1;
13.         else
14.             Y[3] = 0;
15.     end
16. endmodule
```

Relational operators:

(1) <

(2) <=

(3) >=

(4) >

A[2:0]	1	2	3	4	5	6	7	3	0
B[2:0]	3	5	6	7	2	1	0	3	6
Y[3]	1	1	1	1	0	0	0	1	1
Y[2]	1	1	1	1	0	0	0	0	1
Y[1]	0	0	0	0	1	1	1	1	0
Y[0]	0	0	0	0	1	1	1	0	0

Equality & Inequality Operators

```
1. module EQUALITY_OPERATORS(A, B, Y);
2.     input [2:0]      A;
3.     input [2:0]      B;
4.     output [4:0]     Y;
5.     reg [4:0] Y;

6.     always @(A or B)
7.     begin
8.         Y[0] = A != B;
9.         Y[1] = A == B;
10.        if ( A == B)
11.            Y[4:2] = A;
12.        else
13.            Y[4:2] = B;
14.    end

15. endmodule
```

Equality operators:

(1) == (等於)

(2) != (不等於)

(3) === (++) non-syn.

(4) !== (++) non-syn.

When comparison is true, the result is 1

When comparison is false, the result is 0

A[2:0]	0	7	4	1	0
B[2:0]	3	7	3	4	5
Y[0]	1	0	1	1	1
Y[1]	0	1	0	0	0
Y[4:2]	3	7	3	4	5



Logical Comparison Operators (1/2)

```
1.  module COMPARISON(A, B, C1,C2,C3);  
2.      input  [2:0]  A,B;  
3.      output [2:0]  C1, C2, C3;  
4.      reg     [2:0]  C1, C2, C3;
```

```
5.  always @(A or B)  
6.  begin  
7.      if (( A == 1) && ( B>2 ) )  
8.          C1= 2'b 00;  else  C1= 2'b 11;  
9.      if (( A>3) || ( B>1 ) )  
10.          C2= 2'b 00;  else  C2= 2'b 11;  
11.      if (!A)  
12.          C3= 2'b 00;  else  C3= 2'b 11;  
13.  end  
14. endmodule
```

Logic comparison operators:

(1) ! (NOT)

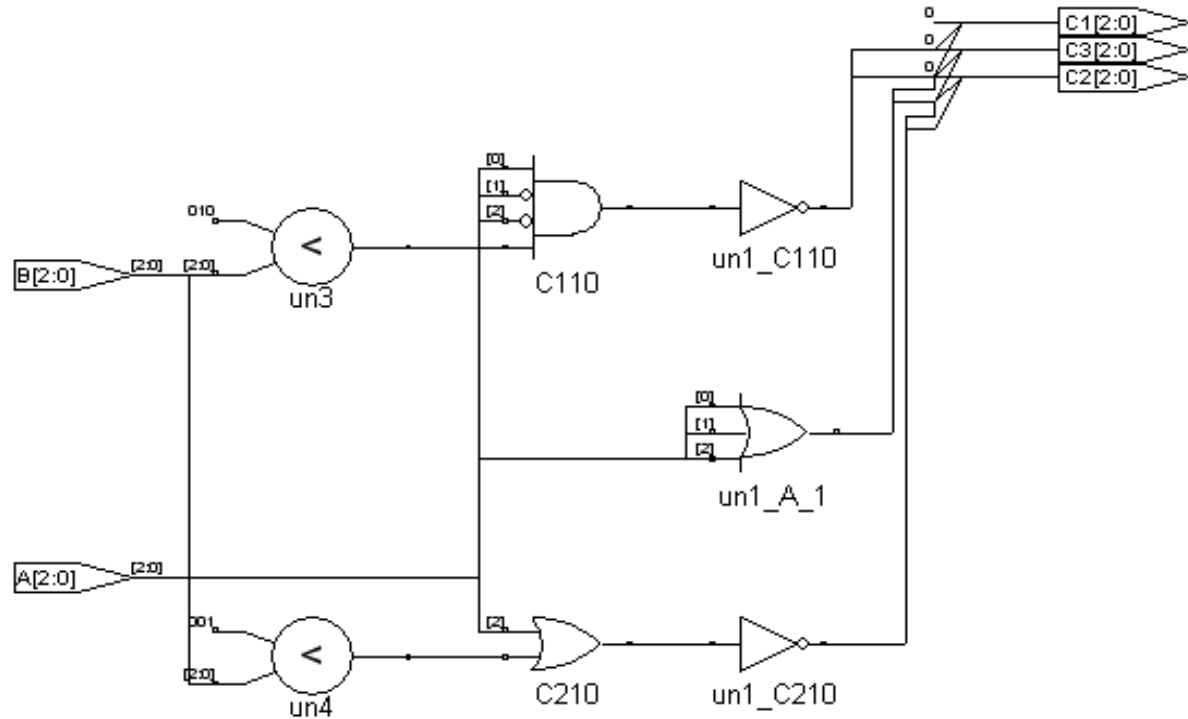
(2) && (AND)

(3) || (OR)

Logical Comparison Operators (2/2)

```

always @(A or B)
begin
  if (( A == 1) && ( B>2))
    C1= 2'b 00;
    else C1= 2'b 11;
  if (( A>3) || ( B>1 ))
    C2= 2'b 00;
    else C2= 2'b 11;
  if (!A)
    C3= 2'b 00;
    else C3= 2'b 11;
end
  
```



A[2:0]	000	001	010	101
B[2:0]	000	010	001	010
C1[2:0]	011	011	011	011
C2[2:0]	011	000	011	000
C3[2:0]	000	011	011	011

Logical Bit-Wise Operators

```
module BITWISE(A, B, Y1, Y2, Y3, Y4, Y5);
```

```
    input [3:0]  A;
```

```
    input [3:0]  B;
```

```
    output [3:0] Y1, Y2, Y3, Y4, Y5;
```

```
    reg [3:0]    Y1, Y2, Y3, Y4, Y5;
```

Logic bit-wise operators:

(1) ~ (unary NOT)

(2) & (binary AND)

(3) | (binary OR)

(4) ^ (binary XOR)

(5) ^~ or ~^ (binary XNOR)

```
always @(A or B)
```

```
begin
```

```
    Y1 = ~A;
```

```
    Y2 = A & B;
```

```
    Y3 = A | B;
```

```
    Y4 = A ^ B;
```

```
    Y5 = A ^ ~ B;
```

```
end
```

```
endmodule
```

0110

& 0011

0010

0110

| 0011

0111

A[3:0]	0000	0001	0010	0110
B[3:0]	0000	0000	0001	0011
Y1[3:0]	1111	1110	1101	1001
Y2[3:0]	0000	0000	0000	0010
Y3[3:0]	0000	0001	0011	0111
Y4[3:0]	0000	0001	0011	0101
Y5[3:0]	1111	1110	1100	1010

Shift Operators

```
1. module SHIFT(A, Y1, Y2);
2.     input [7:0]    A;
3.     output [7:0]   Y1;
4.     output [7:0]   Y2;

5.     reg [7:0]      Y1;
6.     reg [7:0]      Y2;

7.     parameter      B=3;

8.     always @(A)
9.     begin
10.         Y1 = A << B;
11.         Y2 = A >> 2;

12.     end

13. endmodule
```

Shift operators:

(1) << (left shift)

(2) >> (right shift)

A[7:0]	00000000	00000001	00000011	00000100
Y1[7:0]	00000000	00001000	00011000	00100000
Y2[7:0]	00000000	00000000	00000000	00000001

Concatenation & Replication Operators

1. **module** CONCATENATION (A, B, Y);

2. **input** [2:0] A;

3. **input** [2:0] B;

4. **output** [14:0] Y;

5. **reg** [14:0] Y;

6. **parameter** C=3'b011;

7. **always** @(A)

8. **begin**

9. Y = { B, A, { 2 { C } }, 3'b 001};

10. **end**

11. **endmodule** 3+3+2*3+3=15

Concatenation { }

Replication { { } }

A[2:0]	000	001	010	011
B[2:0]	000	010	100	110
Y[14:0]	000	010	100	110
	000	001	010	011
	<u>011</u>	<u>011</u>	<u>011</u>	<u>011</u>
	<u>011</u>	<u>011</u>	<u>011</u>	<u>011</u>
	001	001	001	001

Reduction Operators

```
module REDUCTION (A, Y);  
    input [3:0] A;  
    output [5:0] Y;  
  
    reg [5:0] Y;
```

Reduction operators:

(1) & (2) |
(3) ~& (4) ~|
(5) ^ (6) ~^

```
always @(A)  
begin  
    Y[0] = & A;  
    Y[1] = | A;  
    Y[2] = ~& A;  
    Y[3] = ~| A;  
    Y[4] = ^ A;    // XOR, 奇同位  
    Y[5] = ~^ A;   //XNOR, 偶同位  
end  
endmodule
```

& A \Rightarrow A[0] & A[1] & A[2] & A[3]

| A \Rightarrow A[0] | A[1] | A[2] | A[3]

^ A \Rightarrow A[0] ^ A[1] ^ A[2] ^ A[3]

A[3]	0	0	0	0
A[2]	0	0	0	0
A[1]	0	0	1	1
A[0]	0	1	0	1
Y[5:0]	101100	010110	010110	100110

Conditional Operators

```
1. module ADD_SUB (A, B, SEL, Y);
2.   input [7:0] A;   input [7:0] B;
3.   input           SEL;
4.   output [8:0]     Y1,Y2;
5.   reg [8:0]       Y2,Y1;
6.   always @( A or B)
7.   begin
8.     Y1 = ( SEL == 1 ) ? A + B : A - B ;
9.     Y2 = (!SEL) ? A : B;
10.  end
```

Conditional operators:

? :

SEL	0	0	1	1
A[7:0]	00000100	00000001	00000100	00000001
B[7:0]	00000001	00000010	00000001	00000010
Y1[8:0]	00000011	11111111	00000101	00000011
Y2[8:0]	00000100	00000001	00000001	00000010

Full Adder (1/6)

Reduction with K-map or Boolean Algebra.

ab\c_in	0	1
00	0	1
01	1	0
11	0	1
10	1	0

sum

$$\begin{aligned} &= \overline{a}\overline{b}c_{in} + a\overline{b}c_{in} + \overline{a}bc_{in} + abc_{in} \\ &= (\overline{a} + a)\overline{b}c_{in} + (\overline{a} + a)bc_{in} \\ &= (a \oplus b)\overline{c}_{in} + (a \oplus b)c_{in} \\ &= (a \oplus b) \oplus c_{in} \end{aligned}$$

ab\c_in	0	1
00	0	0
01	0	1
11	1	1
10	0	1

c_out

$$\begin{aligned} &= ab + \overline{a}\overline{b}c_{in} + \overline{a}bc_{in} \\ &= ab + (\overline{a}\overline{b} + \overline{a}b)c_{in} \\ &= ab + (a \oplus b)c_{in} \end{aligned}$$

Reduction is done by user or tool.



Full Adder (2/6)

```
module Add_full(c_in, sum, c_out, a, b);
```

```
input  a, b, c_in;
```

```
output sum, c_out;
```

$$\text{sum} = (a \oplus b) \oplus c_{\text{in}}$$

$$c_{\text{out}} = ab + (a \oplus b)c_{\text{in}}$$

```
reg  sum, c_out;
```

```
always @(a or b or c_in)
```

```
begin
```

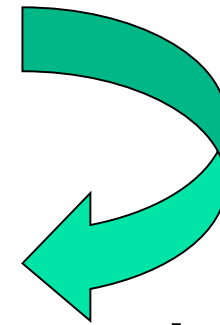
```
    sum = (a ^ b) ^ c_in;
```

```
    c_out = (a&b) | ((a ^ b) &c_in);
```

```
end
```

```
endmodule
```

structural description



or

```
assign {c_out, sum} = a + b + c_in;
```

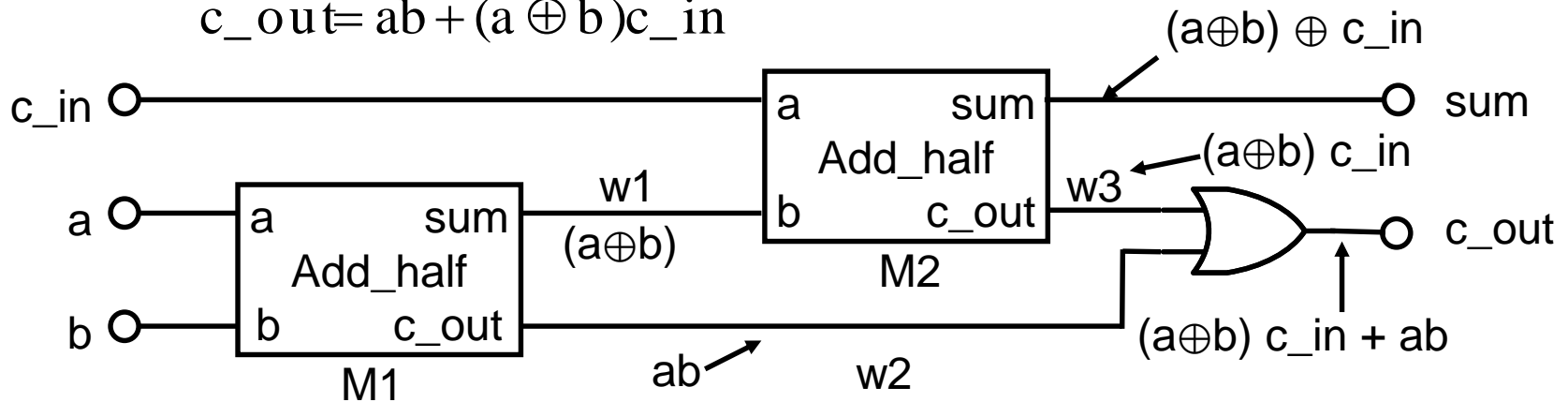
data flow description

Save file as Add_full.v and Synthesize it

Full Adder (3/6)

$$\text{sum} = (a \oplus b) \oplus c_{\text{in}}$$

$$c_{\text{out}} = ab + (a \oplus b)c_{\text{in}}$$



structural description

```
module Add_full(sum, c_out, a, b, c_in);
```

```
  input      a, b, c_in;
```

```
  output     sum, c_out;
```

```
  wire       w1, w2, w3;
```

```
  Add_half   M1(w1, w2, a, b);
```

```
  Add_half   M2(sum, w3, c_in, w1);
```

```
  or        (c_out, w2, w3);
```

```
endmodule
```

in-order port mapping

Hierarchical Description

data flow description

```
module Add_half(sum, c_out, a, b);
```

```
  input  a, b;
```

```
  output sum, c_out;
```

```
  assign {c_out, sum} = a + b;
```

```
endmodule
```



Full Adder (4/6)

Implementation Issue:

```
module Add_full(sum, c_out, a, b, c_in);  
  input      a, b, c_in;  
  output     sum, c_out;  
  wire       w1, w2, w3;
```

```
    Add_half    M1(w1, w2, a, b);  
    Add_half    M2(sum, w3, c_in, w1);  
  or           (c_out, w2, w3);
```

```
endmodule
```

```
module Add_half(sum, c_out, a, b);  
  input  a, b;  
  output sum, c_out;
```

```
  assign {c_out, sum} = a + b;  
endmodule
```

Add_full.v

If both modules (Add_full and Add_half) are saved in the same file, then name and save the file as Add_full.v (top module)

Compile Add_full.v and synthesize it



Full Adder (5/6)

```
`include "Add_half.v"
```

```
module Add_full(sum, c_out, a, b, c_in);
```

```
    input          a, b, c_in;
```

```
    output         sum, c_out;
```

```
    wire           w1, w2, w3;
```

```
    Add_half      M1(w1, w2, a, b);
```

```
    Add_half      M2(sum, w3, c_in, w1);
```

```
    or            (c_out, w2, w3);
```

```
endmodule
```

Add_full.v

```
module Add_half(sum, c_out, a, b);
```

```
    input  a, b;
```

```
    output sum, c_out;
```

```
    assign {c_out, sum} = a + b;
```

```
endmodule
```

Add_half.v

If modules **Add_full** and **Add_half** are saved in the distinguish files (**Add_full.v** and **Add_half** respectively), then the include command is necessary (otherwise ?..)

Compile Add_full.v and synthesize it

Full Adder (6/6)

4-bit Adder

```
module Adder_4_RTL(sum, c_out, a, b, c_in);
```

```
    input    [3:0]  a, b;
```

```
    input          c_in;
```

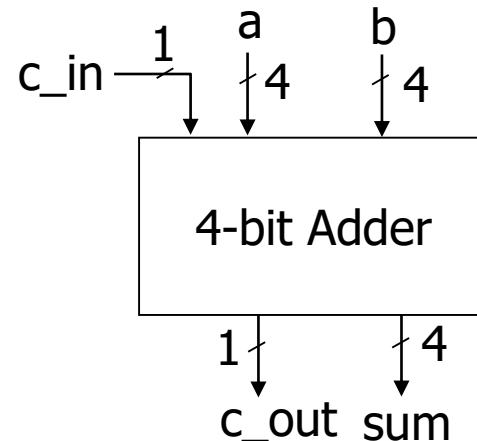
```
    output    [3:0]  sum;
```

```
    output          c_out;
```

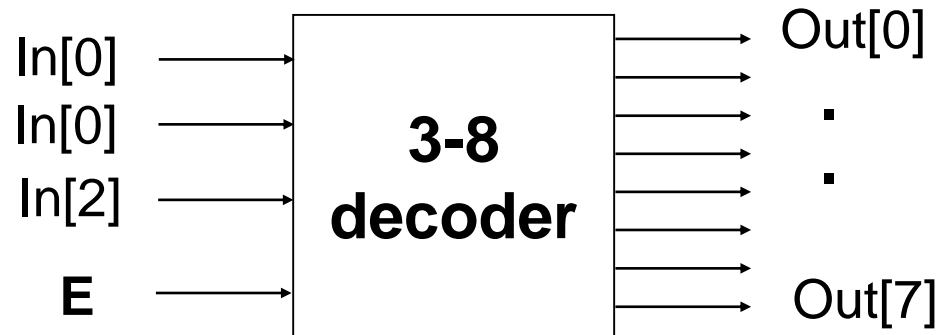
```
    assign {c_out, sum} = a + b + c_in;
```

```
endmodule
```

The synthesized circuit is dependent on the tool you use (might be ripple-carry Adder or other Adders).

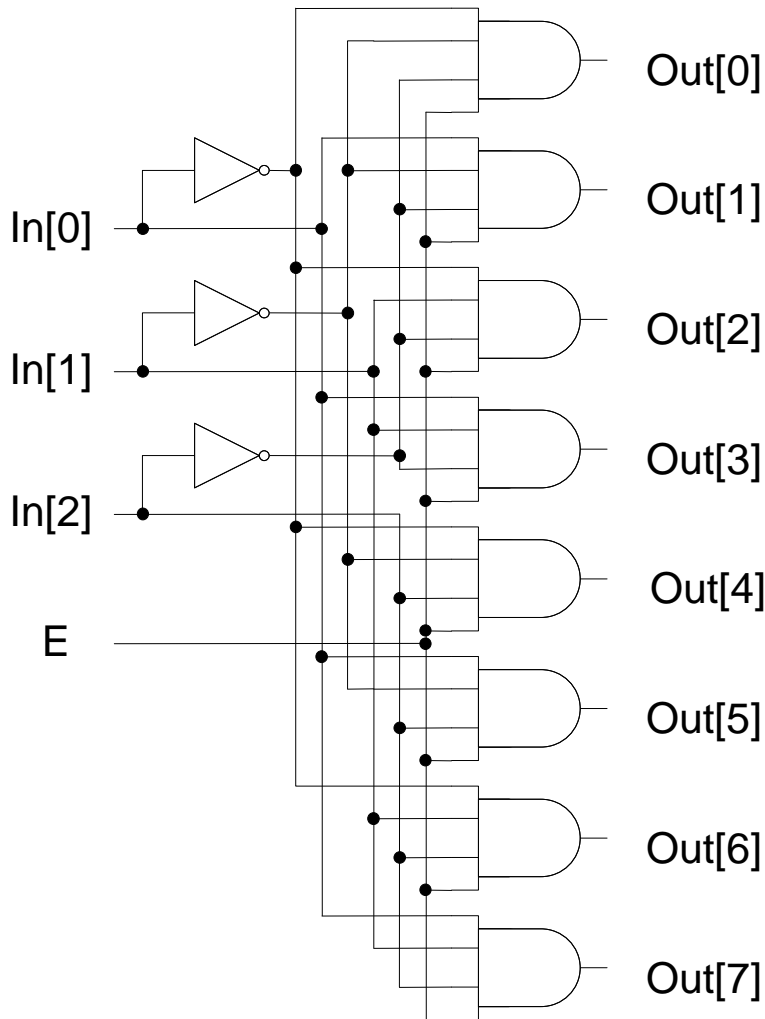


3 to 8 Decoder (1/4)



E	In[2]	In[1]	In[0]	Out[7]	Out[6]	Out[5]	Out[4]	Out[3]	Out[2]	Out[1]	Out[0]
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

3 to 8 Decoder (2/4)



Structural description

```
1.  module decoder (E , In , Out);
2.      input E;  input [2:0] In;
3.      output [7:0] Out;
4.      wire [7:0] Out;
5.      wire tmp0 , tmp1 , tmp2;

6.      not not1(tmp0,In[0]); not not2(tmp1,In[1]);
7.      not not3(tmp2,In[2]);

8.      and and0(Out[0] , E , tmp0 , tmp1 , tmp2);
9.      and and1(Out[1] , E , In[0] , tmp1 , tmp2);
10.     and and2(Out[2] , E , tmp0 , In[1] , tmp2);
11.     and and3(Out[3] , E , In[0] , In[1] , tmp2);
12.     and and4(Out[4] , E , tmp0 , tmp1 , In[2]);
13.     and and5(Out[5] , E , In[0] , tmp1 , In[2]);
14.     and and6(Out[6] , E , tmp0 , In[1] , In[2]);
15.     and and7(Out[7] , E , In[0] , In[1] , In[2]);
16. endmodule
```

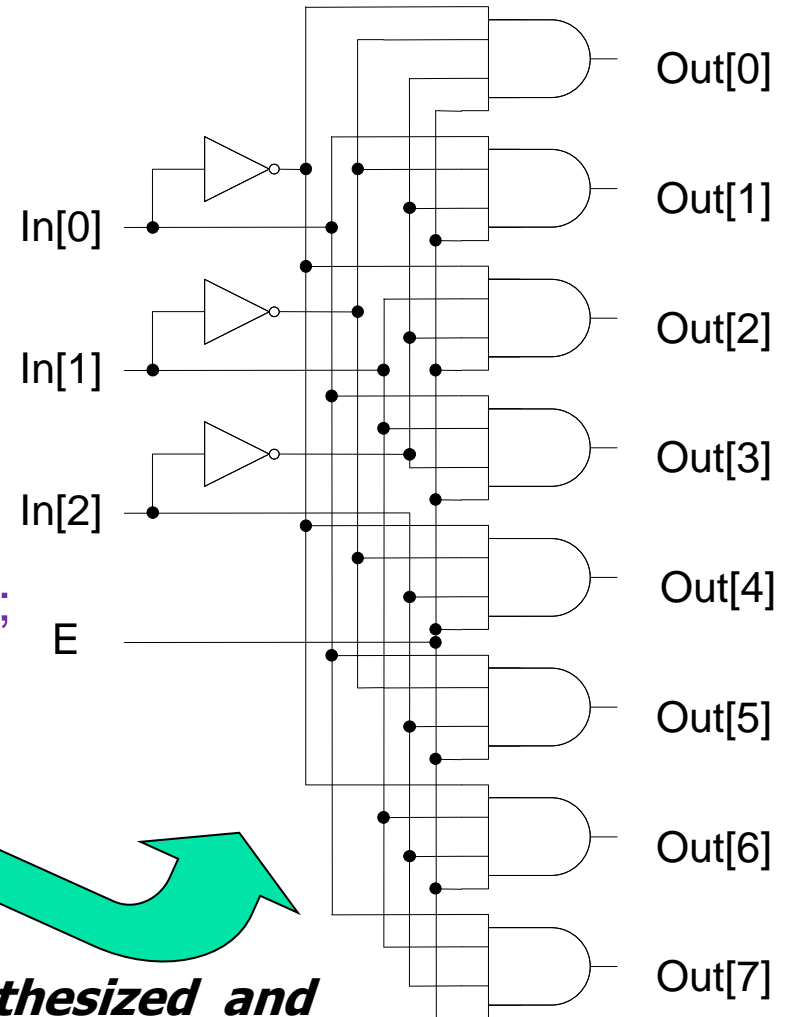
3 to 8 Decoder (3/4)

Data flow description

```
1. module decoder(E , In , Out);
2.   input E;
3.   input [2:0]In;
4.   output [7:0]Out;
5.   wire [7:0]Out;
6.   assign Out = E ? 1'b1 << In : 8'h0;
7. endmodule
```

true false

E	In			Out							
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	1	0	0	0
....											



***Synthesized and
optimized by tools***

3 to 8 Decoder (4/4)

Behavioral description

```
module Decoder_Behavioral(E, In, Out);
```

```
    Input          E;
```

```
    input  [2:0]    In;
```

```
    output [7:0]    Out;
```

```
    reg     [7:0]    Out;
```

```
    always @(E or In)
```

```
    begin
```

```
        if(!E)
```

```
            Out = 8'h00;
```

```
        else
```

```
            begin
```

```
                case(In)
```

```
                    3'b000: Out = 8'h01;
```

```
                    3'b001: Out = 8'h02;
```

```
                    3'b010: Out = 8'h04;
```

```
                    3'b011: Out = 8'h08;
```

```
                    3'b100: Out = 8'h10;
```

```
                    3'b101: Out = 8'h20;
```

```
                    3'b110: Out = 8'h40;
```

```
                    default: Out = 8'h80;
```

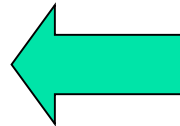
```
                endcase
```

```
            end
```

```
        end
```

```
    endmodule
```

E	In			Out							
0	X	X	X	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0



```

module decoder_2_4(E , In , Out);
input E;          input [1:0] In;
output [3:0]Out;  wire [3:0] Out;

assign Out = E ? 1'b1 << In : 4'h0;

endmodule

```

2 to 4 decoder

```
endmodule
```

endmodule



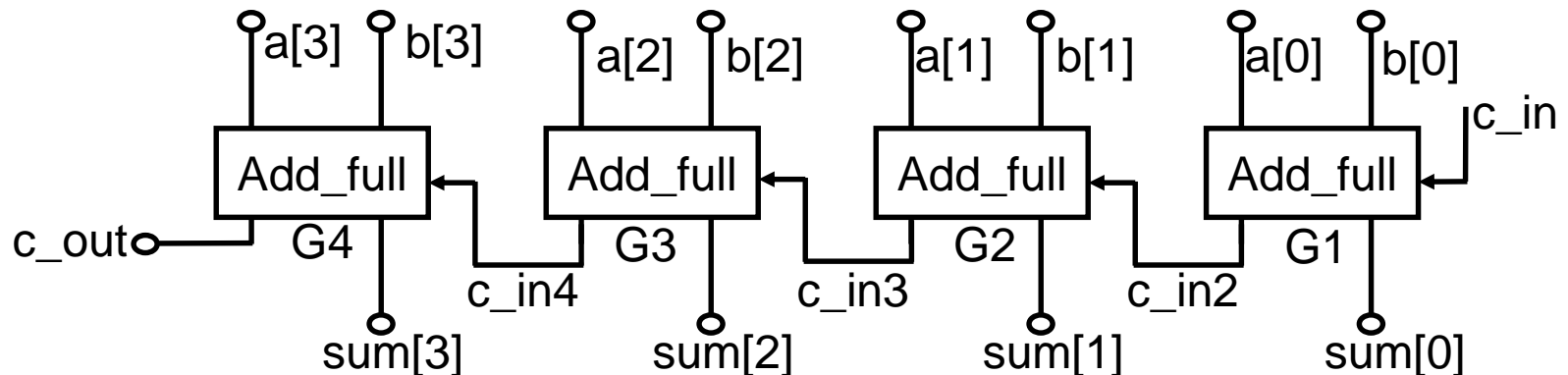
Hierarchical Description of Circuit

Ripple-Carry Adder

```
module Add_rca_4(sum, c_out, a, b, c_in);  
    input    [3:0]    a, b;  
    input    c_in;  
    output   [3:0]    sum;  
    output   c_out;  
    wire     c_in2, c_in3, c_in4;
```

```
    Add_full G1(sum[0], c_in2, a[0], b[0], c_in);  
    Add_full G2(sum[1], c_in3, a[1], b[1], c_in2);  
    Add_full G3(sum[2], c_in4, a[2], b[2], c_in3);  
    Add_full G4(sum[3], c_out, a[3], b[3], c_in4);
```

```
endmodule
```



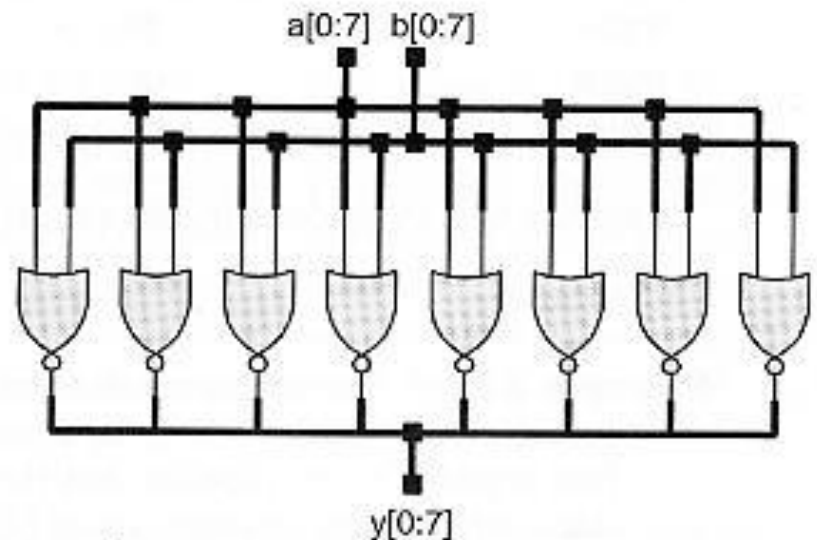
Array of Instances

```
module array_of_nor(y, a, b);  
  input      [0:7]  a, b;  
  output    [0:7]  y;
```

8 {

```
  nor        (y[0], a[0], b[0]);  
  nor        (y[1], a[1], b[1]);  
  nor        (y[2], a[2], b[2]);  
  nor        (y[3], a[3], b[3]);  
  nor        (y[4], a[4], b[4]);  
  nor        (y[5], a[5], b[5]);  
  nor        (y[6], a[6], b[6]);  
  nor        (y[7], a[7], b[7]);  
endmodule
```

```
module array_of_nor(y, a, b);  
  input      [0:7]  a, b;  
  output    [0:7]  y;  
  
  nor        [0:7]  (y, a, b);  
endmodule
```



Two alternatives for Continuous Assignment

```
module bit_or8_gate1(y, a, b);
```

```
    input      [7:0] a, b;
```

```
    output    [7:0] y;
```

```
    assign    y = a | b;
```

```
endmodule
```

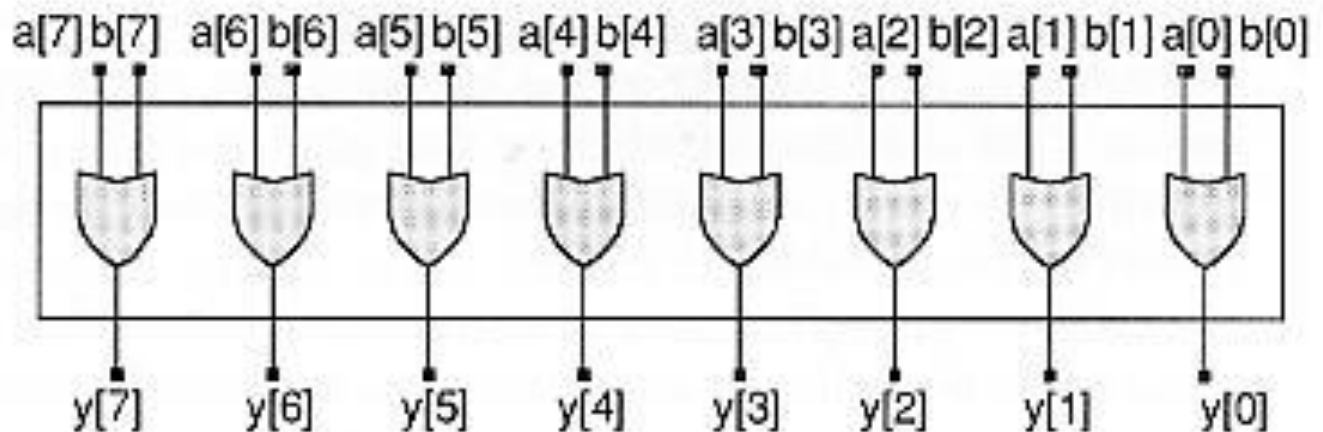
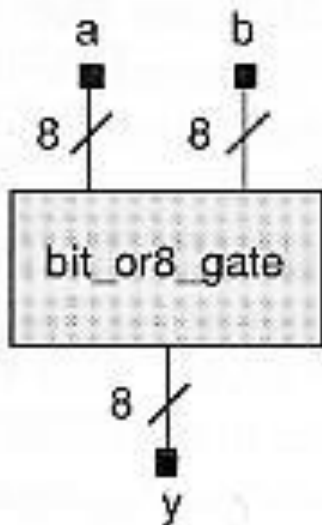
```
module bit_or8_gate2(y, a, b);
```

```
    input      [7:0] a, b;
```

```
    output    [7:0] y;
```

```
    wire      [7:0] y = a | b;
```

```
endmodule
```





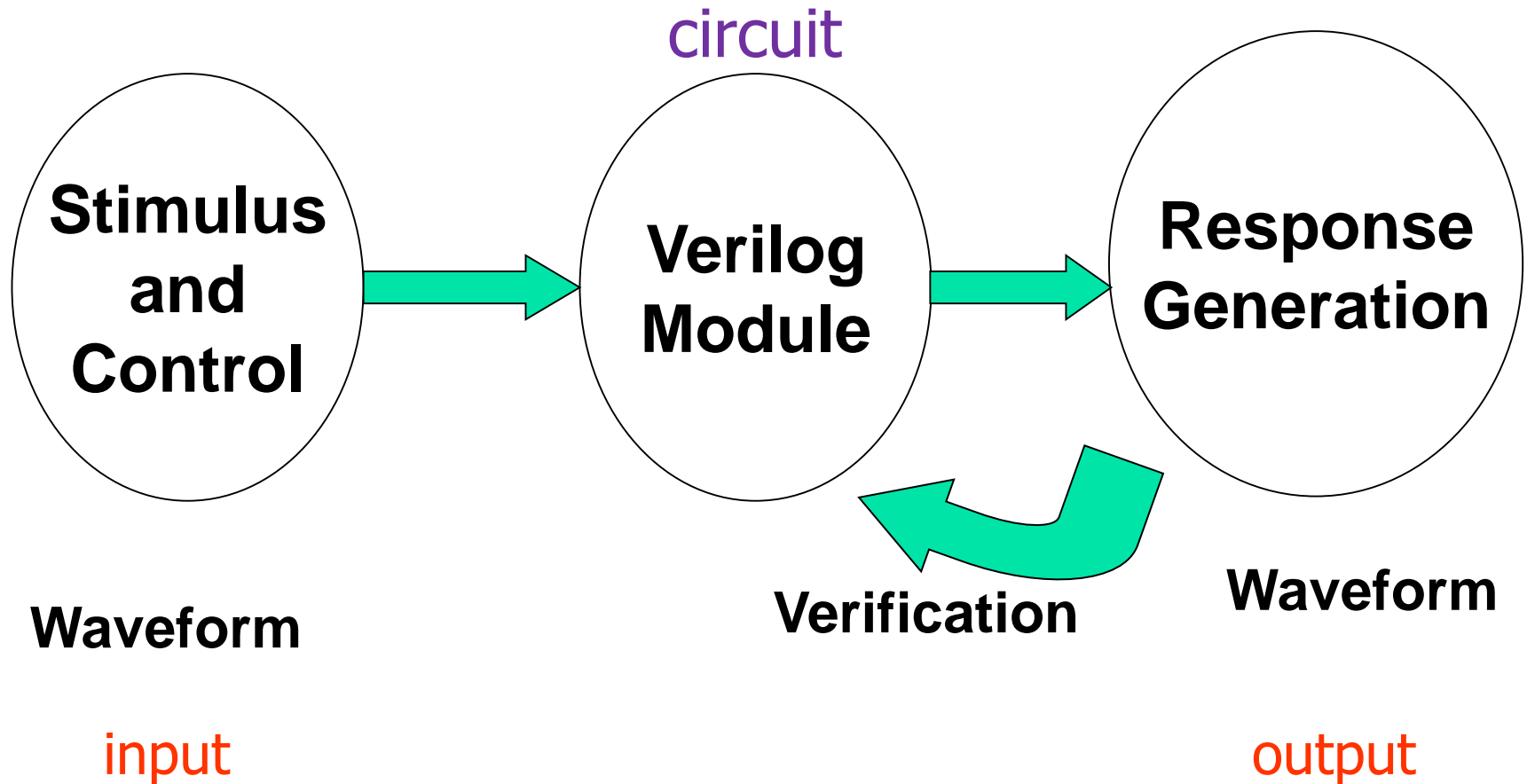
Multiple Instantiations and Assignments

```
module Multiple_Gates(y1, y2, y3, a1, a2, a3, a4);  
  input      a1, a2, a3, a4;  
  output     y1, y2, y3;
```

```
  nand #1 G1(y1, a1, a2, a3), (y2, a2, a3, a4), (y3, a1, a4);  
endmodule
```

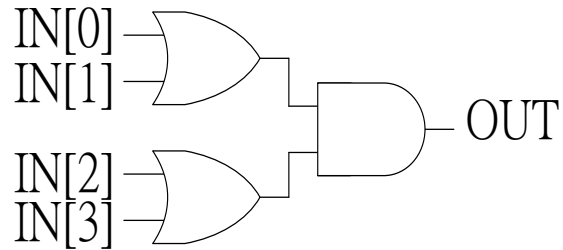
```
module Multiple_Assigns(y1, y2, y3, a1, a2, a3, a4);  
  input      a1, a2, a3, a4;  
  output     y1, y2, y3;  
  
  assign #1   y1 = a1 ^ a2, y2 = a2 | a3, y3 = a1 + a2;  
endmodule
```

Simulation for a circuit



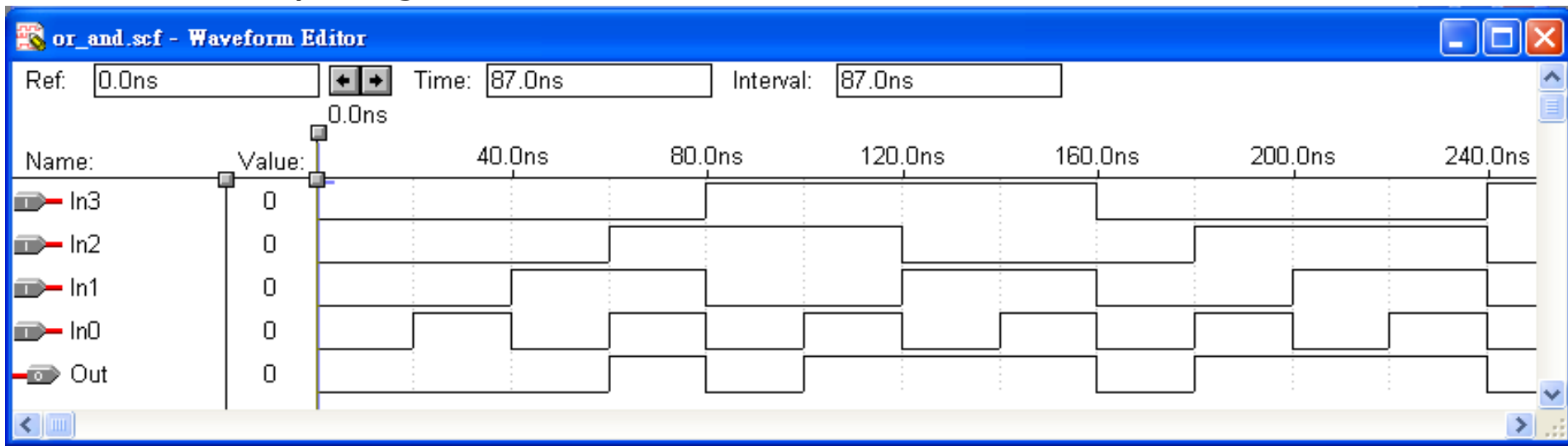
Waveform Simulation (1/3)

Most EDA tools support waveform simulation



```
module OR_AND_DATA_FLOW(in, out);  
  input      [3:0]    in;  
  output          out;  
  
  assign out = (in[0] | in[1]) & (in[2] | in[3]);  
  
endmodule
```

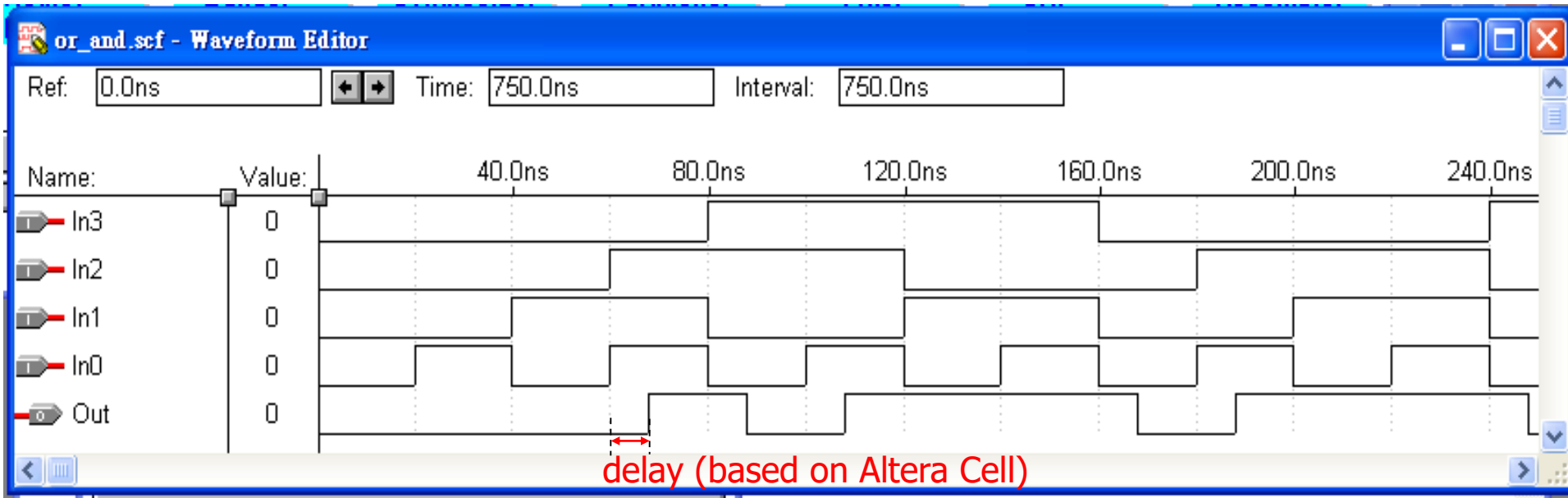
The results by using Altera **functional simulation**



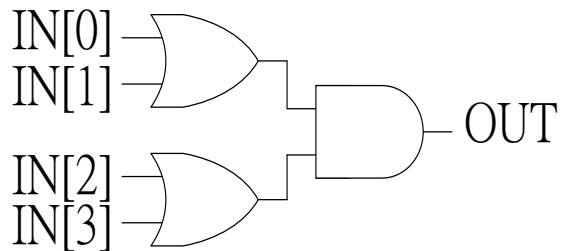
No delay is introduced if only functional simulation is used

Waveform Simulation (2/3)

The results by using Altera functional simulation + timing simulation

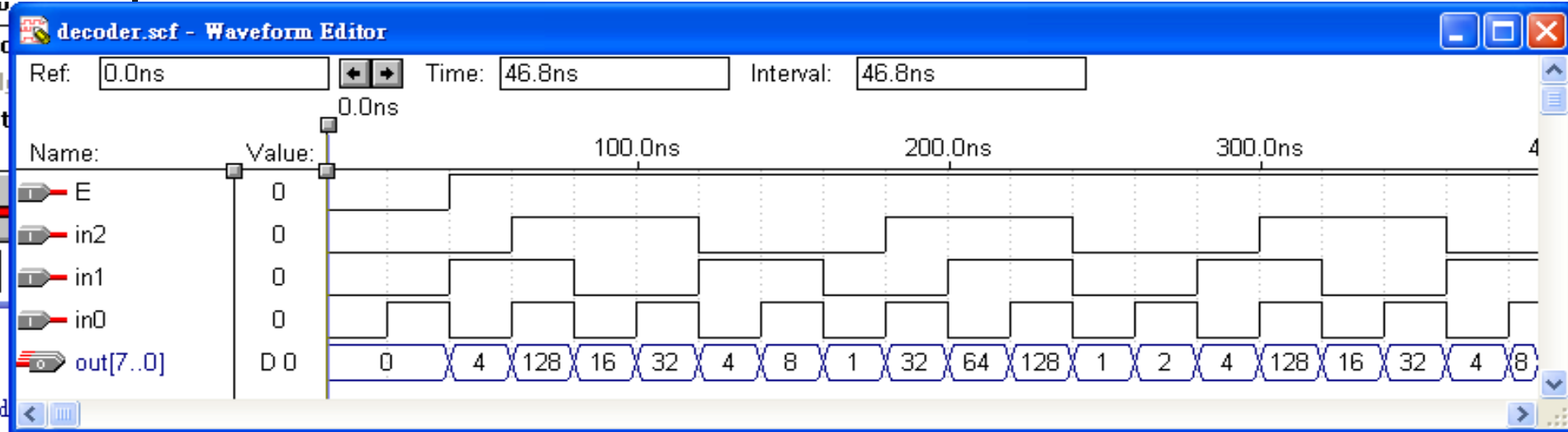


No delay is introduced if only functional simulation is used

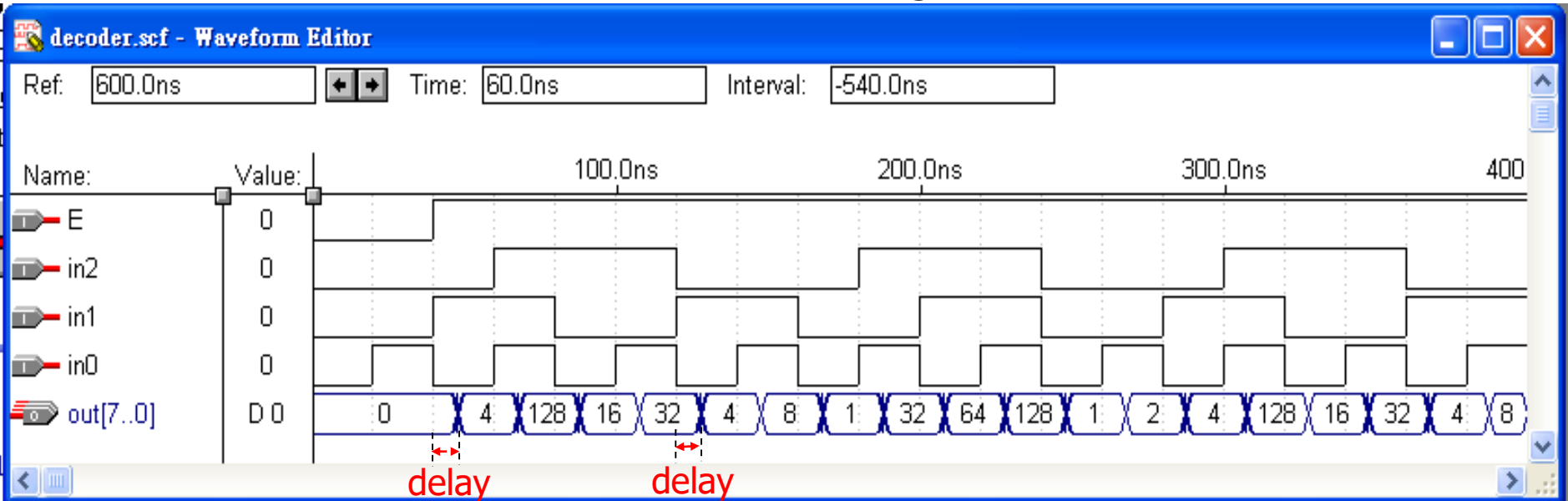


Waveform Simulation (3/3)

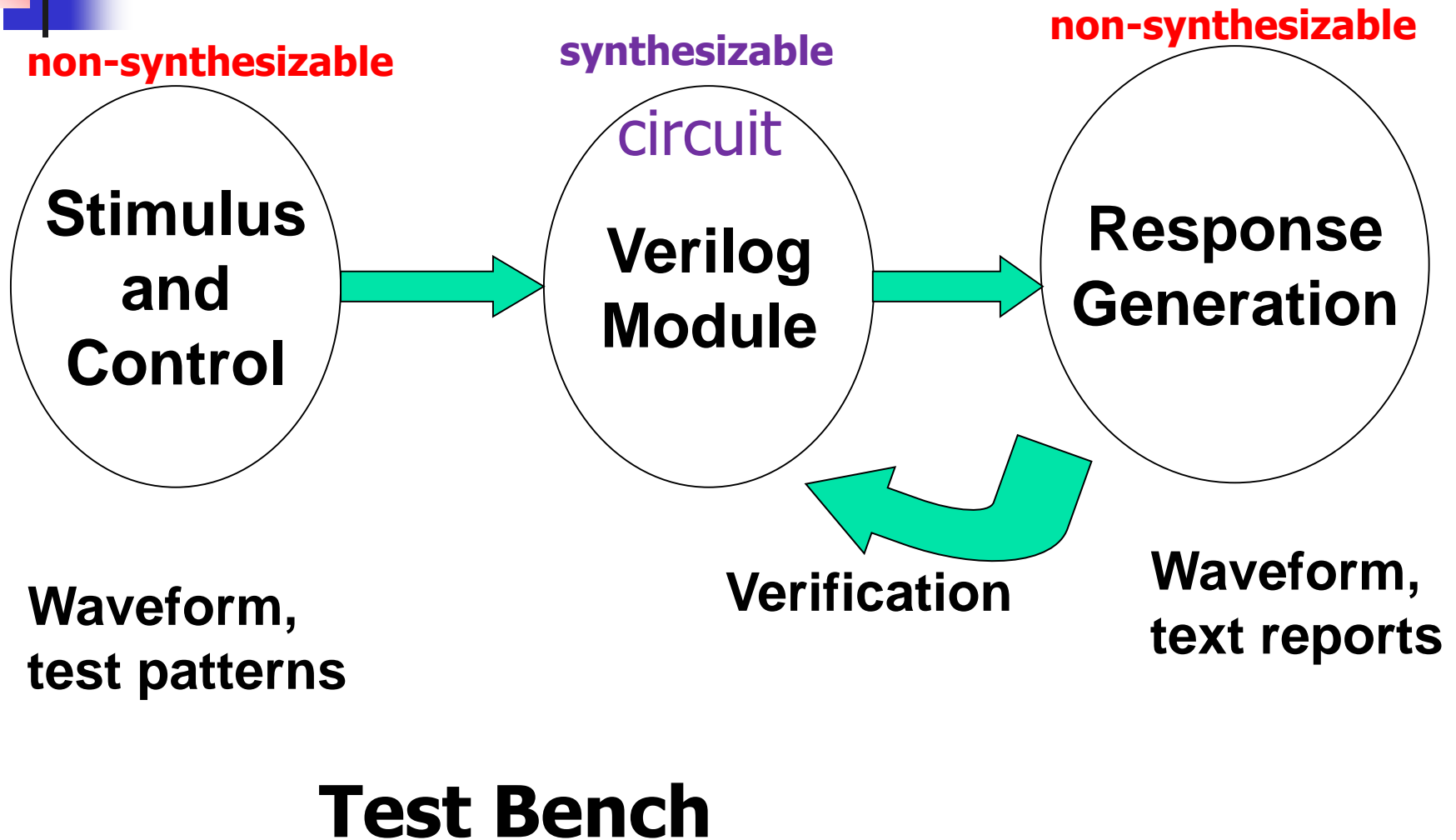
functional simulation for 3 to 8 decoder



functional simulation + timing simulation



Simulation for a circuit





ModelSim Test_Bench Simulation (1/4)

Test_bench

```
module or_and_tb;
reg in1, in2, in3, in4;
wire out;
or_and ok(.in1(in1), .in2(in2),
          .in3(in3), .in4(in4), .out(out));
initial
begin
#0   in1=0; in2=0; in3=0; in4=0;
#10  in1=0; in2=0; in3=0; in4=1;
#10  in1=0; in2=0; in3=1; in4=0;
#10  in1=0; in2=0; in3=1; in4=1;
#10  in1=0; in2=1; in3=0; in4=0;
#10  in1=0; in2=1; in3=0; in4=1;
#10  in1=0; in2=1; in3=1; in4=0;
#10  in1=0; in2=1; in3=1; in4=1;
```

non-synthesizable

```
#10 in1 = 1; in2 = 0; in3 = 0; in4 = 0;
#10 in1 = 1; in2 = 0; in3 = 0; in4 = 1;
#10 in1 = 1; in2 = 0; in3 = 1; in4 = 0;
..
end
endmodule
```

synthesizable

```
module or_and (in, out);
input[3:0]      in;
output          out;

assign out = (in[0] | in[1]) & (in[2] | in[3]);

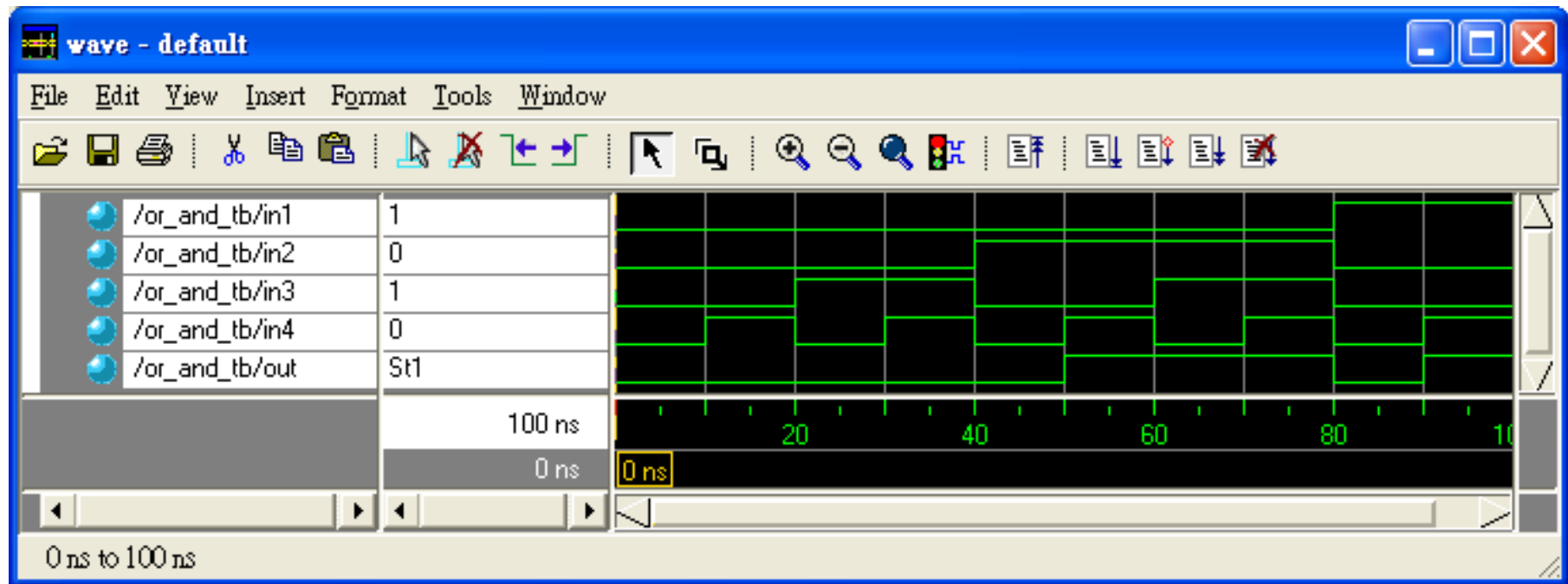
endmodule
```

Altera does not support this kind of simulation (inputs is activated by commands in the test bench file).

ModelSim is a PC-based tool.
(workstation-like)

ModelSim Test_Bench Simulation (2/4)

Functional simulation only (no delay is introduced)



In ModelSim, Input: test patterns (using comds.) Output: waveform

In Altera, Input: waveform Output: waveform



ModelSim Test_Bench Simulation (3/4)

Test_bench

```
module decoder_3_8_tb;
```

```
    reg      E;  
    reg [2:0] in;
```

```
    wire [7:0] out;
```

```
    decoder  
        ok(.E(E), .in(in), .out(out));
```

```
    initial  
    begin
```

```
        #0  E = 0; in = 3'b000;
```

```
        #10 E = 1; in = 3'b000;
```

```
        #10 E = 1; in = 3'b001;
```

```
        #10 E = 1; in = 3'b010;
```

```
        #10 E = 1; in = 3'b011;
```

```
        #10 E = 1; in = 3'b100;
```

```
        #10 E = 1; in = 3'b101;
```

```
        #10 E = 1; in = 3'b110;
```

```
        #10 E = 1; in = 3'b111;
```

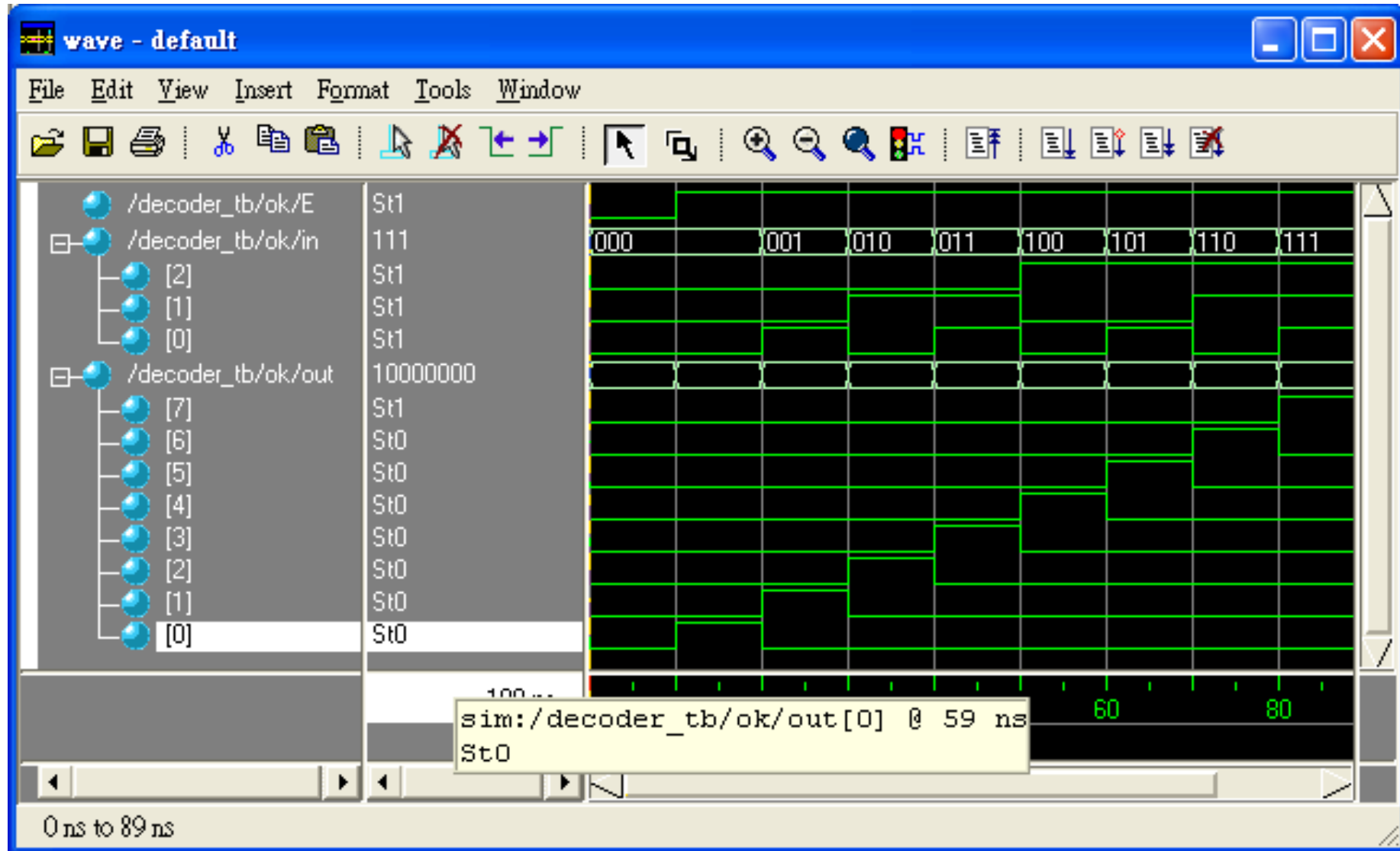
```
    end
```

```
endmodule
```

Exhaustive test or partial test for functional simulation in

test_bench ? How about chip test ? Exhaustive or partial testing?

ModelSim Test_Bench Simulation (4/4)





Signalscan Simulation in WS (1/4)

```
Test_bench
module decoder_3_8_tb;

reg      E;
reg [2:0] in;

wire [7:0] out;

decoder ok(.E(E), .in(in), .out(out));

initial
begin
#0  E = 0; in = 3'b000;
#10 E = 1; in = 3'b000;
#10 E = 1; in = 3'b001;
#10 E = 1; in = 3'b010;
#10 E = 1; in = 3'b011;
#10 E = 1; in = 3'b100;
```

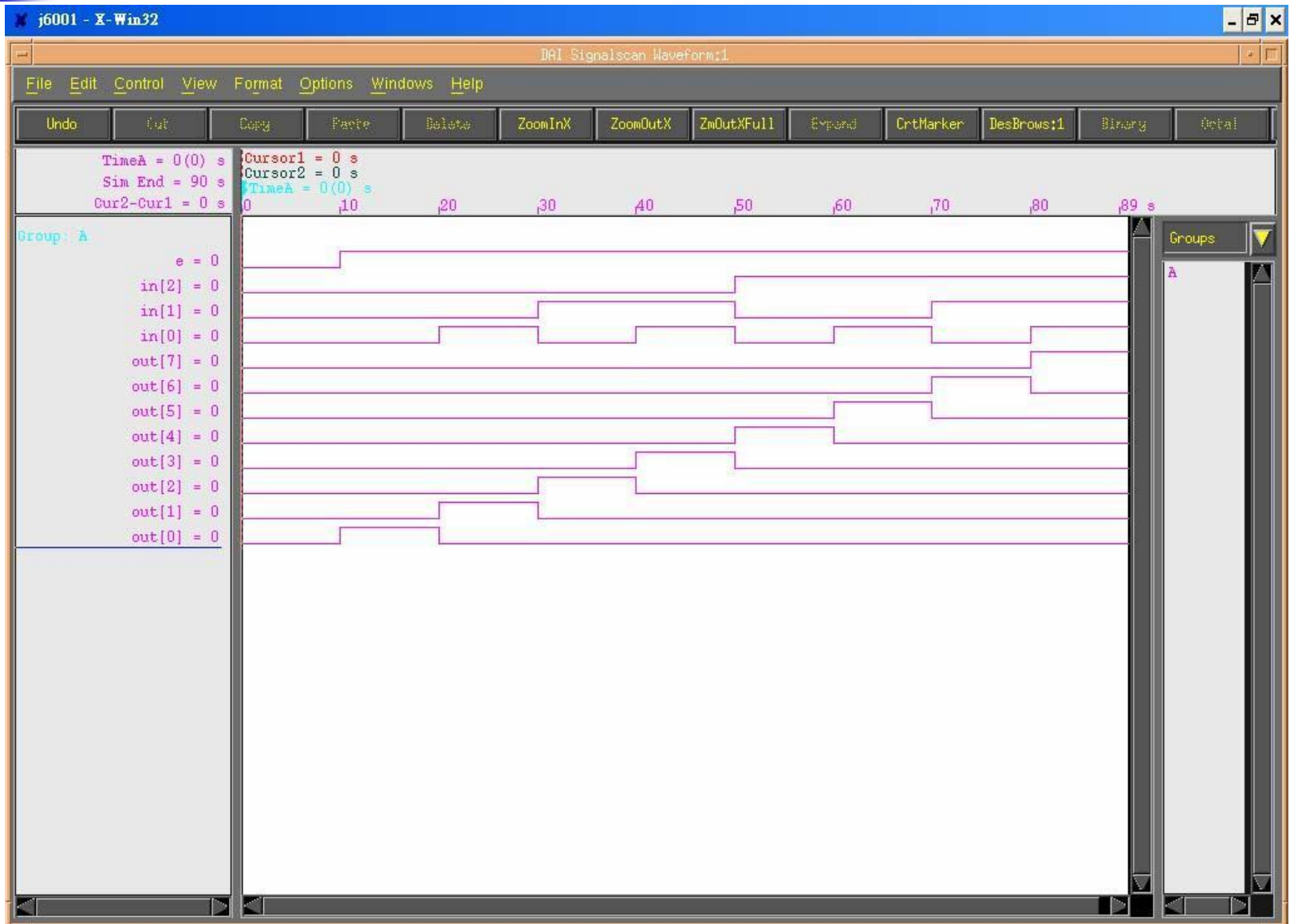
```
#10 E = 1; in = 3'b101;

#10 E = 1; in = 3'b110;
#10 E = 1; in = 3'b111;

End
Initial
begin
    $dumpfile("decoder.fsdb");
    $dumpvars(0 , decoder_tb);
    $shm_open("ok");
    $shm_probe("AS");
end

endmodule
```

Signalscan Simulation in WS (2/4)





Signalscan Simulation in WS (3/4)

```
module decoder_tb;
```

```
    reg          E;  
    reg [2:0] in;  
    reg [3:0] i;  
    wire [7:0] out;  
    integer decoder_1;
```

```
    decoder ok(.E(E), .in(in), .out(out));
```

```
    initial      //Open file_decoder  
    begin  
        decoder_1 = $fopen("decoder_out.txt");  
    end
```

```
    initial
```

```
    begin
```

```
    for(i=0; i<8; i = i + 1)
```

```
        begin
```

```
            #10 E = 1 ; in[2:0] = i[3:0];
```

```
            #1 $fdisplay(decoder_1," E = %d
```

```
in[2] =%d in[1] = %d in[0] = %d
```

```
out = %b",E,in[2],in[1],in[0],out[7:0]);
```

```
$monitor($time , " E = %d in[2] = %d
```

```
in[1] = %d in[0] = %d out = %b", E,
```

```
in[2], in[1], in[0], out[7:0]);
```

```
        end
```

```
    $dumpfile("sim_decoder.dump");
```

```
    $dumpvars(1,decoder_tb);
```

```
    $shm_open("sim_decoder");
```

```
    $shm_probe("AC");
```

```
end
```

```
endmodule
```



Signalscan Simulation in WS (4/4)

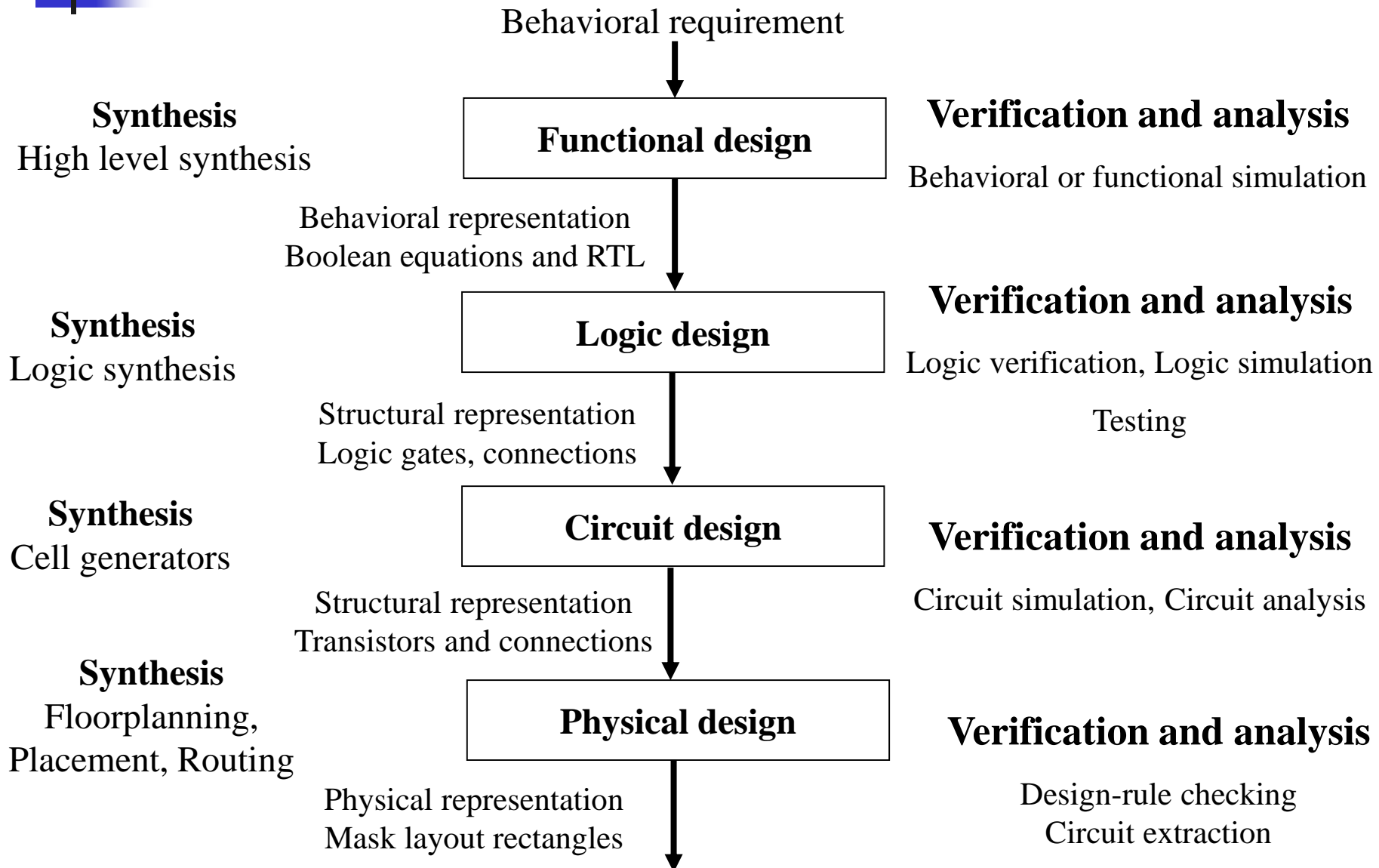
decoder_out.txt

E = 1 in[2] = 0 in[1] = 0 in[0] = 0 out = 00000001
E = 1 in[2] = 0 in[1] = 0 in[0] = 1 out = 00000010
E = 1 in[2] = 0 in[1] = 1 in[0] = 0 out = 00000100
E = 1 in[2] = 0 in[1] = 1 in[0] = 1 out = 00001000
E = 1 in[2] = 1 in[1] = 0 in[0] = 0 out = 00010000
E = 1 in[2] = 1 in[1] = 0 in[0] = 1 out = 00100000
E = 1 in[2] = 1 in[1] = 1 in[0] = 0 out = 01000000
E = 1 in[2] = 1 in[1] = 1 in[0] = 1 out = 10000000

What should we do after HDL simulation ?
FPGA synthesis or ASIC synthesis



Synthesis Flow of VLSI design (1/3)





Synthesis Flow of VLSI design (2/3)

Synthesis:

1. A transition from a single domain to another
2. Add detail to the current state of the design
3. Perform fully automatically by some synthesis tool or manually by the designer

Verification tool:

Check whether a synthesis step has left the specification intact

Analysis tool:

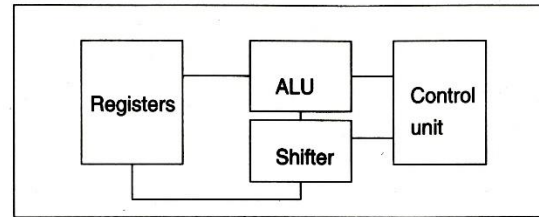
Provide data on the quality of the design (speed, area,...)

Synthesis Flow of VLSI design (3/3)

**Behavioral
description**

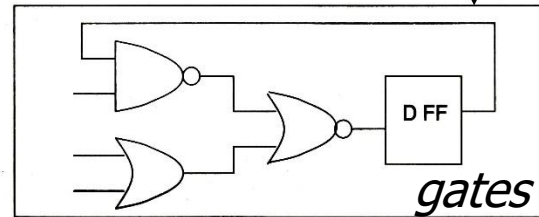
→ Functional design

High level synthesis



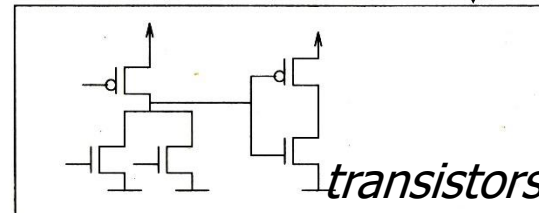
↓ **logic synthesis**

Logic design



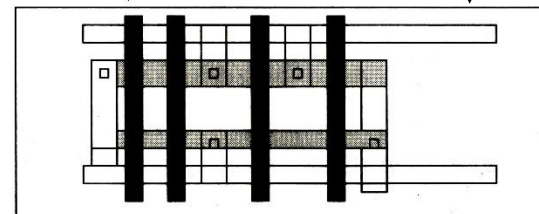
↓ **circuits synthesis**

Circuit design



↓ **physical synthesis**

Physical design



RTL
Code

Physical synthesis:

- Floorplanning & Placement**
Fix the relative positions of the subblocks
- Routing**
Generate the interconnection wires between blocks