



# 數位IC設計

---

## *RTL Coding – Part II*



# Begin\_End Statements

## Begin-end block statements:

1. Block statements is a way of syntactically grouping several statements into a single statement.
2. Sequential blocks are delimited by the keywords begin and end. These begin...end pairs are commonly used in conjunction with if, case, and for statements to group several statements.
3. Functions and always blocks that contain more than one statement require a begin...end pair to group the statements.
4. Verilog provides a construct called a named block.

```
begin [: block_name  
  
    reg local_variable_1 ;  
  
    integer local_variable_2 ;  
  
    parameter local_variable_3 ;]  
  
    . . . statements . . .  
  
end
```

Verilog allows you to declare variables (reg, integer and parameter) locally within a named block but not in an unnamed block.



# Named Block (1/3)

```
module UNNAMED_BLOCK(A , B , E , Y);
parameter F= 12;
input [3 : 0]A;
input [3 : 0]B;
input E;
output [4 : 0]Y;

reg [4 : 0]Y;

always @(A or B or E)
begin
    if(E)
        Y = (A + B) & F;
    else
        Y = (A - B) & F;
end
endmodule
```

```
module NAMED_BLOCK(A , B , E , Y);
input [3 : 0]A; input [3 : 0]B;
input E; output [4 : 0]Y; reg [4 : 0] Y;
```

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

```
    if(E)
```

```
        begin: Add_And
```

```
        parameter F = 12; // in a named block
```

```
        Y = (A + B) & F;
```

```
        end
```

```
    else
```

```
        begin: Sub_And
```

```
        parameter F= 2; // in a named block
```

```
        Y = (A - B) & F;
```

```
        end
```

```
    end
```

```
endmodule
```

↗ block有命名

↘ 可以設 local variable

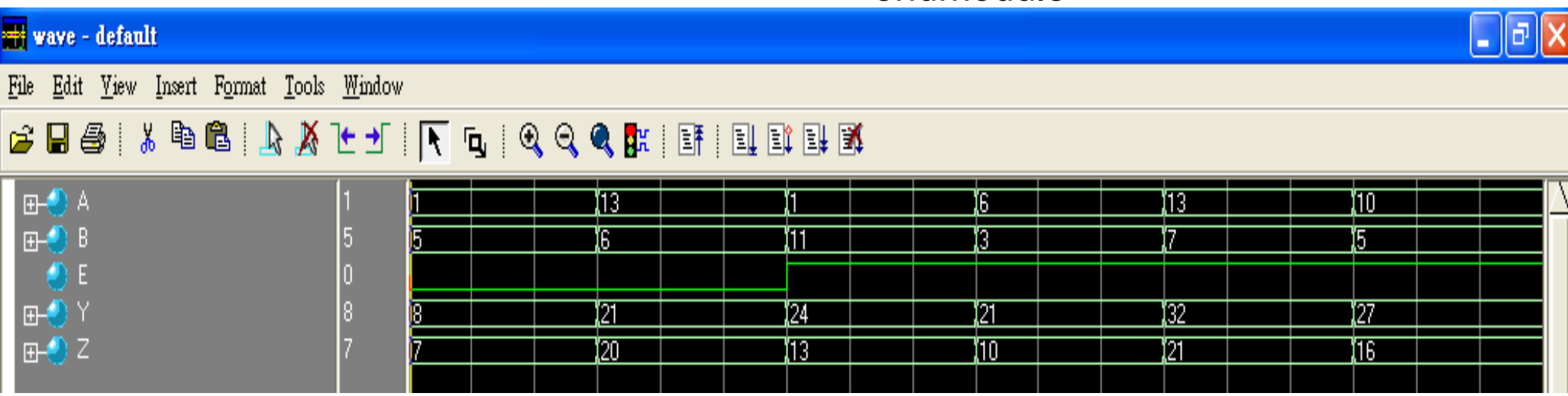
# Named Block (2/3)

```
module NAMED_BLOCK(A , B , E , Y);  
input [3 : 0]A; input [3 : 0]B;  
input E;    output [4 : 0]Y;  
reg [4 : 0]Y;
```

```
parameter F1 = 1;  
parameter F2 = 8;  
always @(A or B or E)  
begin
```

```
if(E)  
begin: Add_And  
    parameter F1 = 12;  
    Y = (A + B) + F1; //F1 is 12 not 1  
end  
else  
begin: Sub_And  
    parameter F2= 2;  
    Y = (A +B) + F2; //F2 is 2 not 8  
end end
```

```
assign Z = (A + B) + F1; //F1 is 1  
endmodule
```



# Named Block (3/3)

```
module LOCAL_GOL(IN_1 , IN_2 , OUT_1 , OUT_2);
input [3 : 0]IN_1;
input [3 : 0]IN_2;
output [4 : 0]OUT_1;
output [4 : 0]OUT_2;
reg [4 : 0]OUT_1;
reg [4 : 0]OUT_2;

always @(IN_1)
begin: Local_Value_1
    parameter X = 7;
    OUT_1 = IN_1 + X;
end

always @(IN_2)
begin: Local_Value_2
    parameter X = 8;
    OUT_2 = IN_2 + X;
end
endmodule
```

wave - default

File Edit View Insert Format Tools Window

IN\_1 IN\_2 OUT\_1 OUT\_2

0	1	13	1	6	13	10	0
2	5	6	11	3	7	5	2
7	8	20	8	13	20	17	7
10	13	14	19	11	15	13	10

# If-Else Statements (1/4)

1. The if statement is followed by a statement or block of statements enclosed by begin and end.
2. If the value of the expression is nonzero, the expression is true and the statement block that follows is executed. If the value of the expression is zero, the expression is false and the statement block following else is executed.

```
If (expression)
begin
... statements ...
end
[else
begin
... statements ...
end]
```

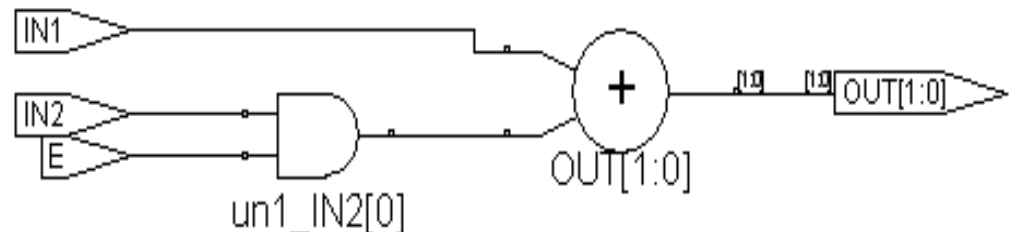
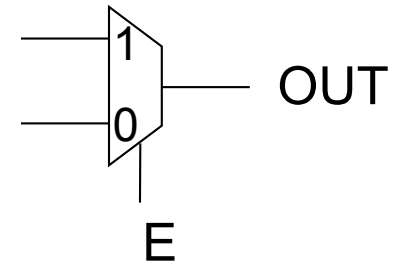


3. If..else statements can cause synthesis of latches.

不好控制  $\Rightarrow$  flip-flop 比較好

if 和 else 要同時寫

```
module IF_ELSE(IN1 , IN2 , E , OUT);
input IN1, IN2, E;
output [1 : 0] OUT; reg [1 : 0]OUT;
always @(IN1 or IN2 or E)
begin
    if(E == 1)
        OUT = IN1 + IN2;
    else
        OUT = IN1;    end
endmodule
```

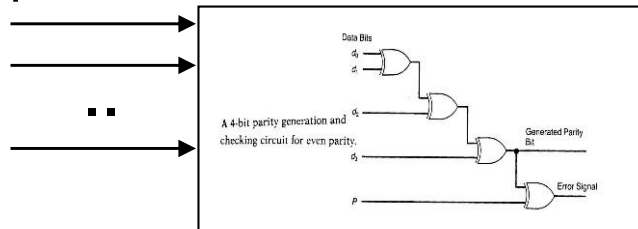


# Combinational vs. Sequential Circuit

**Combinational Circuit** 輸出只和輸入有關

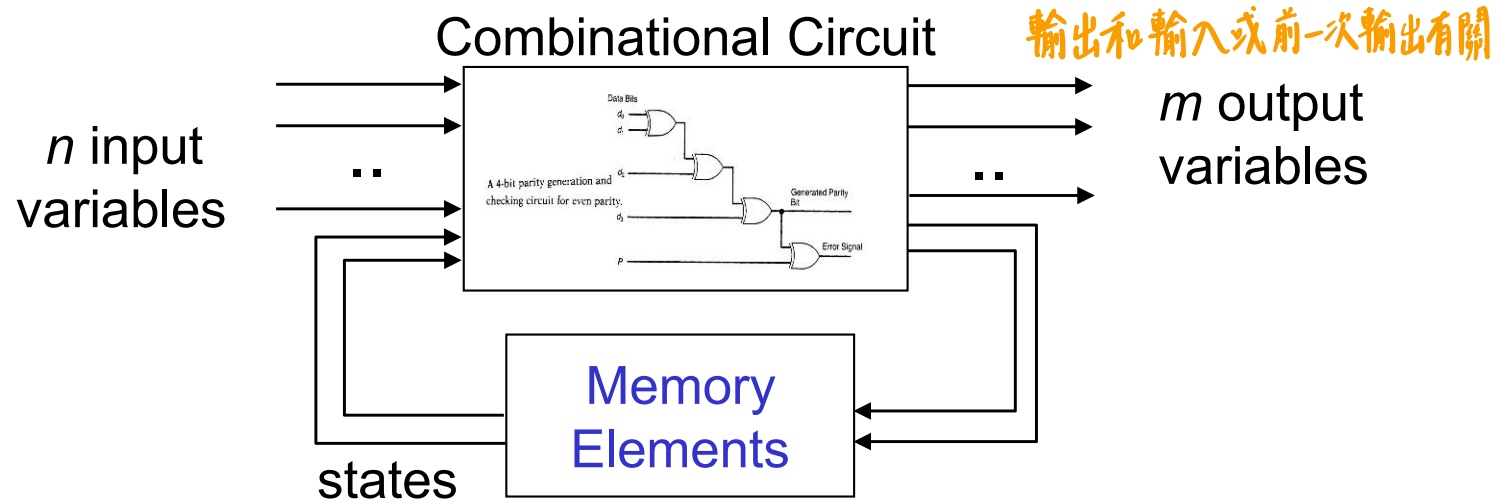
$n$  inputs

$m$  outputs



A **combinational circuit** consists of logic gates whose outputs at any time are determined directly from the present combination of inputs without regard to previous inputs.

A **sequential circuit** is a system whose outputs at any time are determined from the present combination of inputs and the previous inputs or outputs.



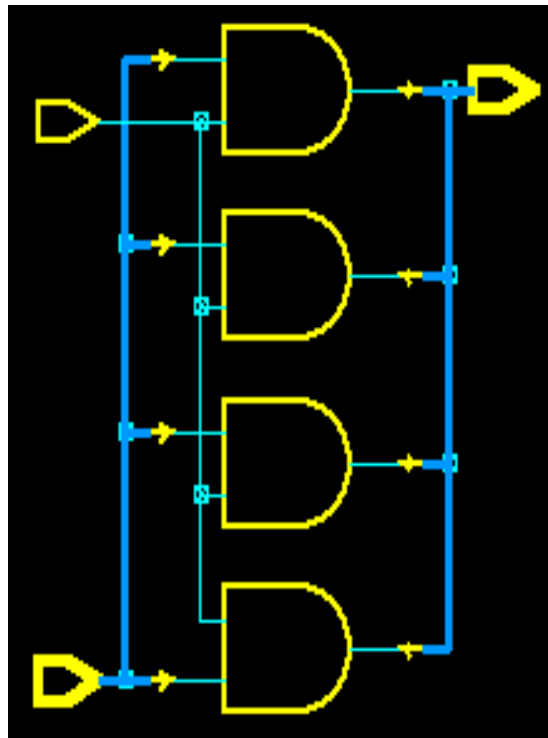
- Sequential components contain memory elements

- The output values of sequential components depend on the input values and the values stored in the memory elements

# If-Else Statements (2/4)

```
Module Latch(In, Enable, Out);  
input      Enable;  
Input  [3:0] In;  
output [3:0] Out;  
always @(In or Enable)  
begin  
    if(Enable)  
        Out=In;  
    else  
        Out=0;  
    end  
end  
endmodule
```

No latch inference



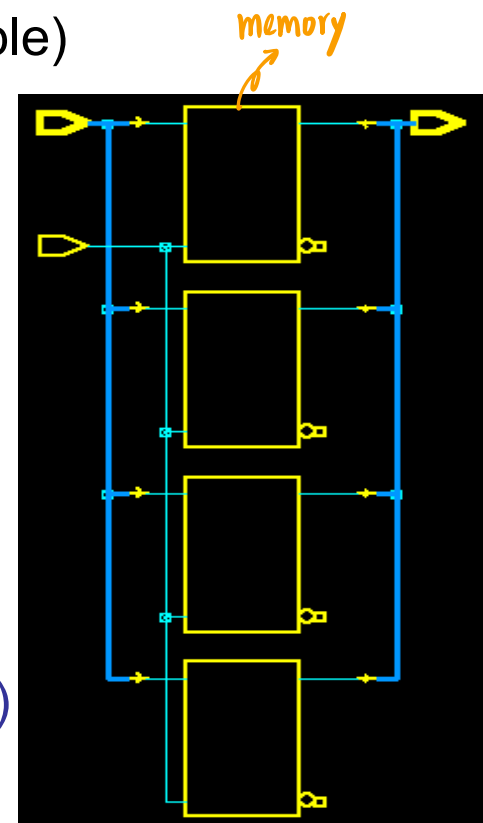
```
Always@ (In or  
Enable)  
begin  
    Out=0;  
    if(Enable)  
        Out=In;  
end // no latch
```

## Watch for unintentional Latches

```
Module Latch(In, Enable, Out);  
input      Enable;  
input  [3:0] In;  
output [3:0] Out;
```

```
always @(In or Enable)  
begin  
    if(Enable)  
        Out=In;  
    end  
end  
endmodule
```

If Enable == 1  
Out (new) = In  
If Enable == 0  
Out (new) = Out (old)





# If-Else Statements (3/4)

```
module IF_ELSE_VALUE(IN , OUT);
input [1 : 0]IN;
output [1 : 0]OUT;
reg [1 : 0]OUT;
always @(IN)
begin
    if(IN==1)
        OUT = 2'b00;
    else
        OUT = 2'b11;
end
endmodule
```

IN=0,OUT=11  
IN=1,OUT=00  
IN=2,OUT=11  
IN=3,OUT=11

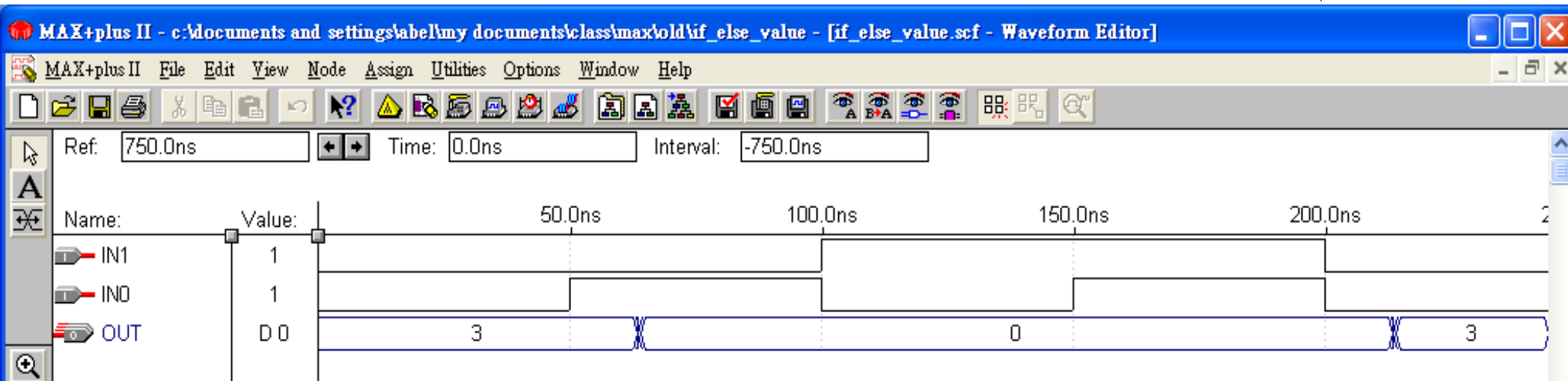
true  
false

```
module IF_ELSE_VALUE(IN , OUT);
input [1 : 0]IN;
output [1 : 0]OUT;
reg [1 : 0]OUT;
always @(IN)
begin
    if(IN)
        OUT = 2'b00;
    else
        OUT = 2'b11;
end
endmodule
```

IN=0,OUT=11  
IN=1,OUT=00  
IN=2,OUT=00  
IN=3,OUT=00

nonzero  $\Rightarrow$  true  
zero  $\Rightarrow$  false

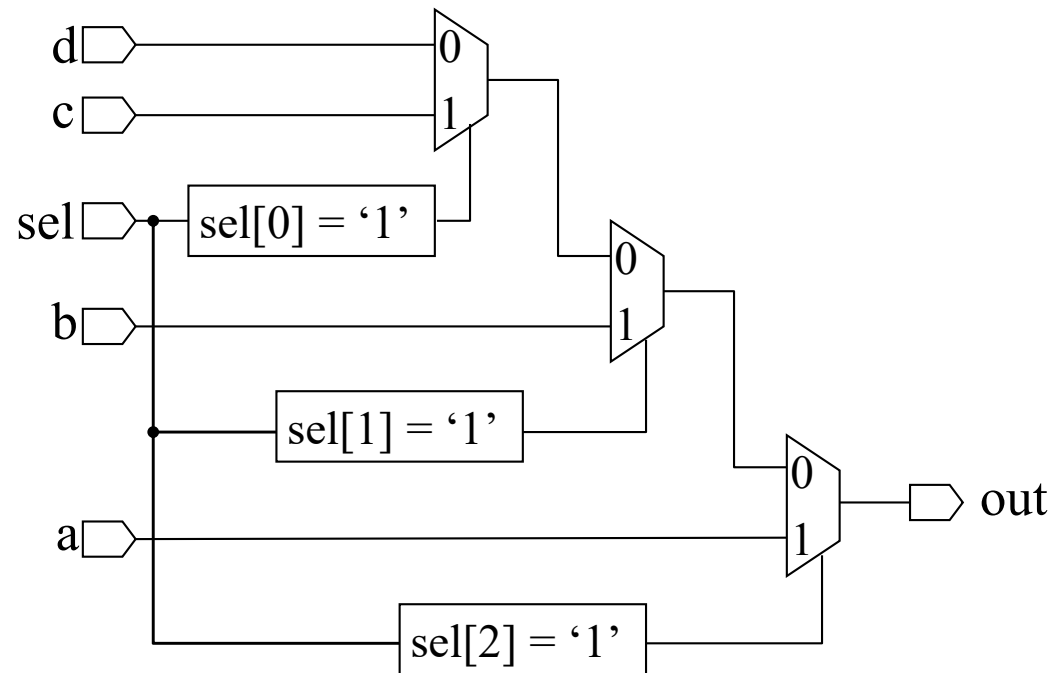
What is the difference between them?



# If-Else Statements (4/4)

*if-then-else* statement implies priority-encoded MUXs

```
always @(sel or a or b or c or d)
  if (sel[2] == 1'b1) 先看最高位元
    out = a;          //sel=1XX
  else if (sel[1] == 1'b1)
    out = b;          //sel=01X
  else if (sel[0] == 1'b1)
    out = c;          //sel=001
  else
    out = d;          //sel=000
```

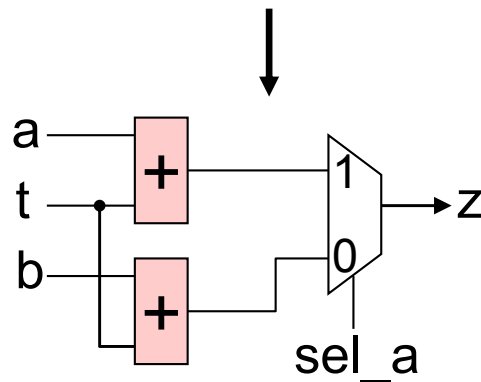


# Resource Sharing (1/2)

- Assign similar operations to a common netlist cell
  - **reduce hardware**
  - may degrade performance
  - resource sharing within the same **always block**
  - resource sharing **not done** for **conditional operator**

## Without resource sharing

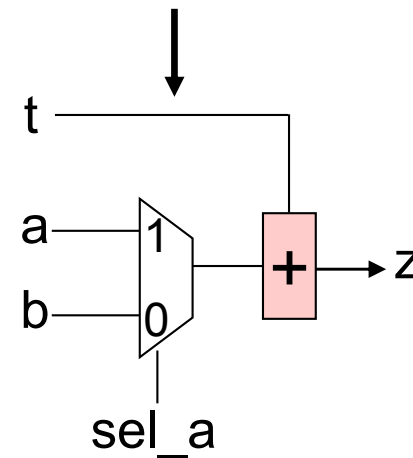
```
assign z = sel_a ? a+t : b+t;
```



Hence, in this case you better use if statement not conditional operator “?” to save a lot of cost and area

## Resource sharing $\Rightarrow$ good

```
if (sel_a)  
    z = a + t;  
else  
    z = b + t;
```



# Resource Sharing (2/2)

```
module noshare(v, w, x, z, k);  
input [2:0]k,v,w;  
input x;  
output [3:0]z;  
wire [3:0]y;  
  
assign y=x?k+w:k+v;  
assign z=x?y+w:y+v;  
  
endmodule
```



Without resource sharing

With resource sharing



```
module share(v, w, z,k);  
input [2:0] k,v,w;  
input x; output [3:0]z;  
reg [3:0] y,z;  
  
always@(x or k or v or w)  
begin  
    if(x)  
        y=k+w;  
    else  
        y=k+v;  
end  
always@(y or x or w or v)  
begin  
    if(x)  
        z=y+w;  
    else  
        z=y+v;  
end  
endmodule
```



# Case Statements (1/4)

The case statement consists of the keyword case, followed by an expression in parentheses, followed by one or more case items (and associated statements to be executed), followed by the keyword endcase. A case item consists of an expression (usually a simple constant) or a list of expressions separated by commas, followed by a colon (:).

```
module FULL_CASE(IN , OUT);
input [1 : 0]IN;output [3 : 0] OUT;
reg [3 : 0]OUT;
always @(IN)
begin
    case(IN)
    2'b00: OUT = 4'b0001;
    2'b01: OUT = 4'b0010;
    2'b10: OUT = 4'b0100;
    2'b11: OUT = 4'b1000;
    endcase
end
endmodule
```

```
module FULL_CASE(IN , OUT);
input [1 : 0]IN;output [3 : 0] OUT;
reg [3 : 0]OUT;
always @(IN)
begin
    case(IN)
    2'b00: OUT = 4'b0001;
    2'b01: OUT = 4'b0010;
    2'b10: OUT = 4'b0100;
    endcase // not full-case, latches are inferred
end
endmodule
```

↳ case要寫滿

case (expression)

```
case_item 1:
begin
. statements . .
end
```

```
case_item 2:
begin
. statements. .
end
```

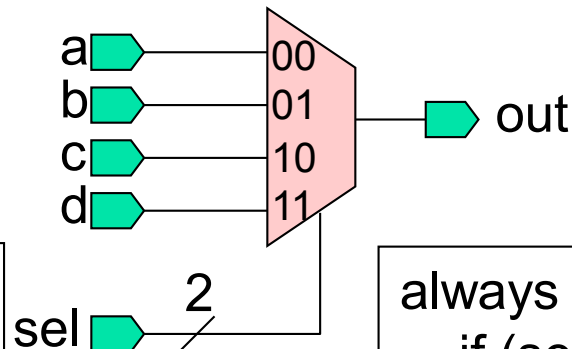
.....

```
default:
begin
. statements. .
end
```

endcase

# Case Statements (2/4)

If the conditional expression used is mutually exclusive (i.e. parallel) and the functional outputs are the same, then the hardware synthesized will be same.



```
always @ (sel or a or b or c or d)
begin
    case (sel)
        2'b00: out = a;
        2'b01: out = b;
        2'b10: out = c;
        default : out = d;
    endcase
end
```

```
always @(sel or a or b or c or d)
if (sel == 2'b00)
    out = a;
else if (sel == 2'b01)
    out = b;
else if (sel == 2'b10)
    out = c;
else
    out = d;
```



# Case Statements (3/4)

---

```
module FULL_CASE(IN , OUT);  
input  [1 : 0]IN; output [3 : 0] OUT;  
reg [3 : 0]OUT;  
always @(IN)  
begin  
    case(IN)  
        2'b00: OUT = 4'b0001;  
        2'b01: OUT = 4'b0010;  
        2'b10: OUT = 4'b0100;  
        2'b11: OUT = 4'b1000;  
    endcase  
end  
endmodule
```

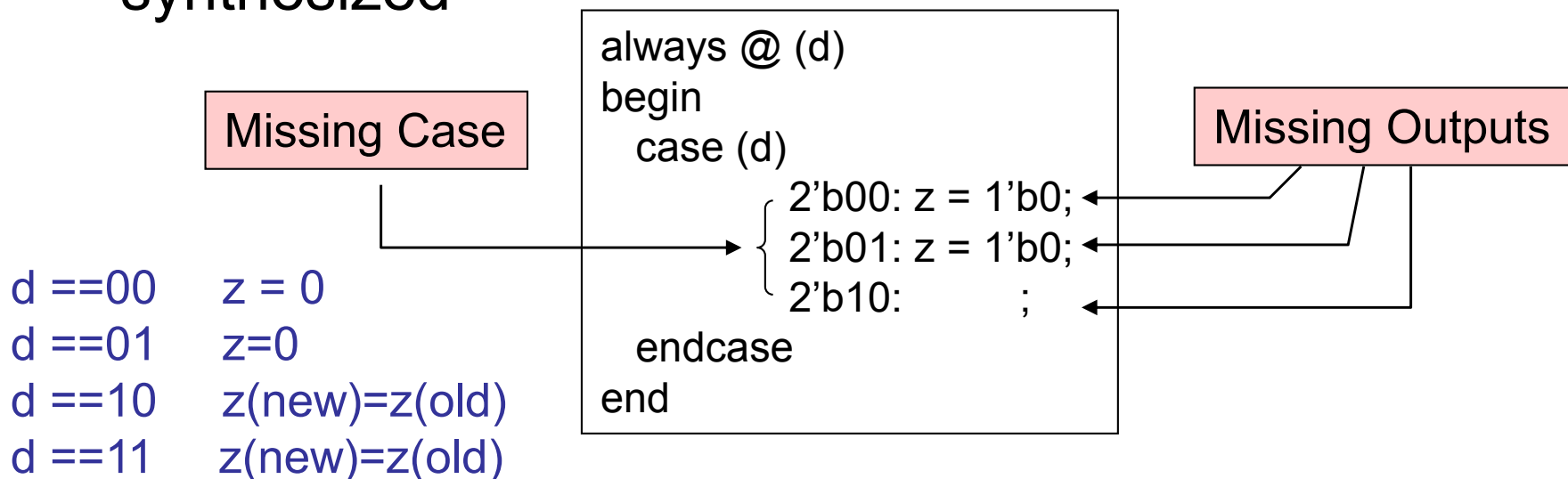
```
module FULL_CASE(IN , OUT);  
input  [1 : 0]IN; output [3 : 0] OUT;  
reg [3 : 0]OUT;  
always @(IN)  
begin  
    case(IN)  
        2'b00: OUT = 4'b0001;  
        2'b01: OUT = 4'b0010;  
        2'b10: OUT = 4'b0100;  
        default: OUT = 4'b1000;  
    endcase  
end  
endmodule
```

It is always a good idea to use default-case-item in all conditions to make sure no latch is inferred.

# Case Statements (4/4)

## Watch Out for Unintentional Latches → 不小心產生 latch

- Completely specify all clauses for every **case** and **if** statement
- Completely specify all output for every clause of each **case** or **if** statement
- Failure to do so will cause latches or flip-flops to be synthesized







# Casez and Casex Statements

casez (expression)

```
case_item 1:  
  begin  
    . statements . .  
  end
```

```
case_item 2:  
  begin  
    . statements. .  
  end
```

.....

```
default:  
  begin  
    . statements. .  
  end
```

endcase

casez:

The case item  
can use z or ?

casex:

The case item  
can use z, x,  
or ?

casex (expression)

```
case_item 1:  
  begin  
    . statements . .  
  end
```

```
case_item 2:  
  begin  
    . statements. .  
  end
```

.....

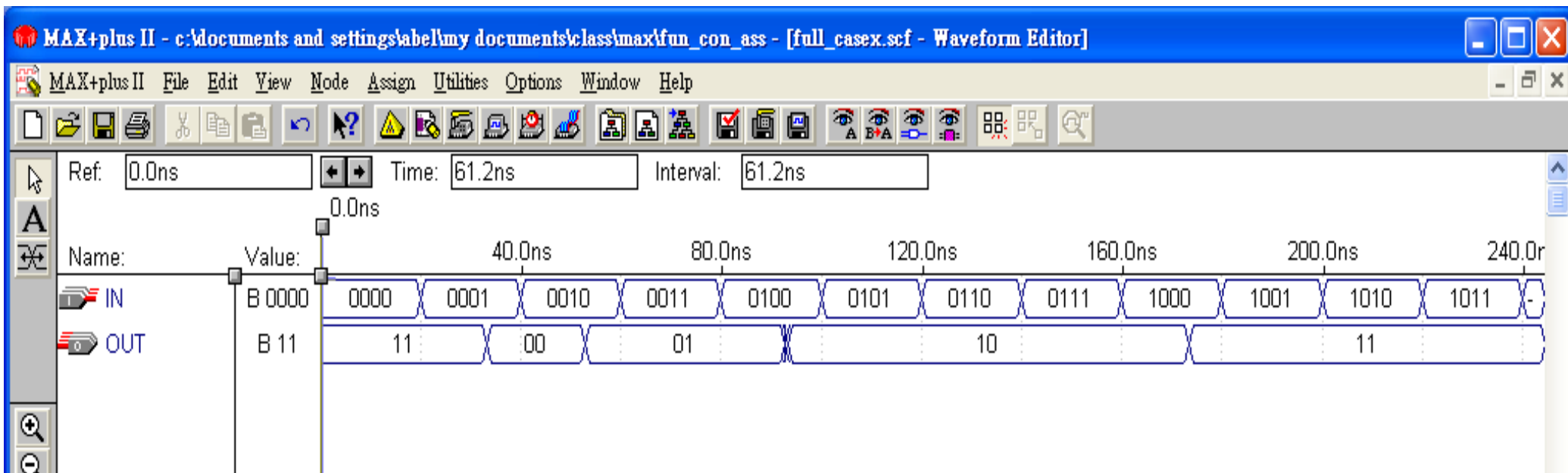
```
default:  
  begin  
    . statements. .  
  end
```

endcase

casez is one of the conditions in casex

# Case Statement

```
module FULL_CASEX(IN , OUT);  
  input  [3 : 0] IN;  
  output [1 : 0] OUT;  
  
  reg [1 : 0] OUT;  
  
  always @(IN)  
  begin  
    casex(IN)  
      4'b0001: OUT = 2'b00;  
      4'b001?: OUT = 2'b01;  
      4'b01???: OUT = 2'b10;  
      default: OUT = 2'b11;  
    endcase  
  end  
endmodule
```





# For Loop Statements (1/2)

The for loop repeatedly executes a single statement or block of statements. The repetitions are performed over a range determined by the range expressions assigned to an index. Two range expressions appear in each for loop: `low_range` and `high_range`. In the syntax lines that follow, `high_range` is greater than or equal to `low_range`. HDL Compiler recognizes incrementing as well as decrementing loops. The statement to be duplicated is surrounded by `begin` and `end` statements.

```
for (index = low_range; index < high_range; index = index + step)
for (index = high_range; index > low_range; index = index - step)
for (index = low_range; index <= high_range; index = index + step)
for (index = high_range; index >= low_range; index = index - step)
```

```
for (i = 0; i <= 31; i = i + 1)
```

```
begin
```

```
    s[i] = a[i] ^ b[i] ^ carry;
```

```
    carry = (a[i] & b[i]) | (a[i] & carry) | (b[i] & carry);
```

```
end
```

**For statement: unrolling the logic**

↳ 全部展開 i.e. 生成32份硬體

# For Loop Statements (2/2)

*for* loop are “unrolled”, and then synthesized.

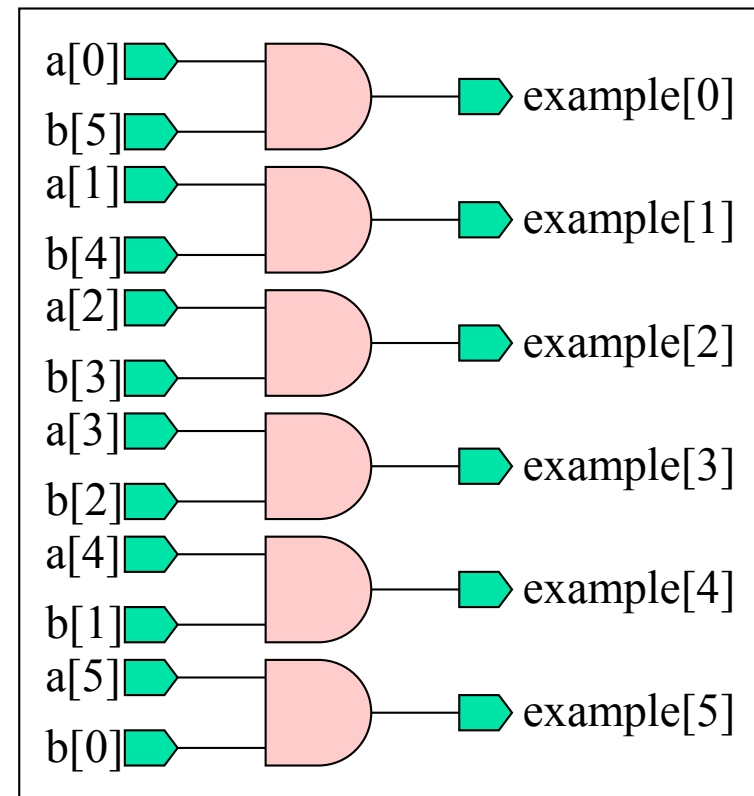
```
integer i;  
always @(a r b)  
begin  
    for (i = 0; i < 6; i = i+1)  
        example[i] <= a[i] & b[5-i];  
end
```

**Verilog for loop**



```
example [0] <= a[0] and b[5];  
example [1] <= a[1] and b[4];  
example [2] <= a[2] and b[3];  
example [3] <= a[3] and b[2];  
example [4] <= a[4] and b[1];  
example [5] <= a[5] and b[0];
```

**for loop unrolled**



**for loop synthesized to gates**



# Sorting Problem (1/2)

Bubble Sort: (four inputs)

1. (0)?(1) (compare temp[0] and temp [1],  
then the bigger value is stored in temp[1])  
(1)?(2) (compare temp[1] and temp [2],...)  
(2)?(3) (compare temp[2] and temp [3],...)
2. (0)?(1), (1)?(2)
3. (0)?(1)

Totally, six comparators are needed  
for parallel comparisons of 4 inputs

```
module for_loop(a,b,c,d,out);  
input  [3:0]a,b,c,d;  
output [3:0]out;  
reg    [3:0] temp [3:0];  
reg    [3:0] buffer,out;  
integer i,j;
```

always@(a or b or c or d)  
begin

```
temp[0]=a;  
temp[1]=b;  
temp[2]=c;  
temp[3]=d;
```

```
for(i=2;i>=0;i=i-1)  
  for(j=0;j<=i;j=j+1)
```

```
    if(temp[j]>temp[j+1])  
      begin
```

```
        buffer=temp[j+1];  
        temp[j+1]=temp[j];  
        temp[j]=buffer;
```

```
      end
```

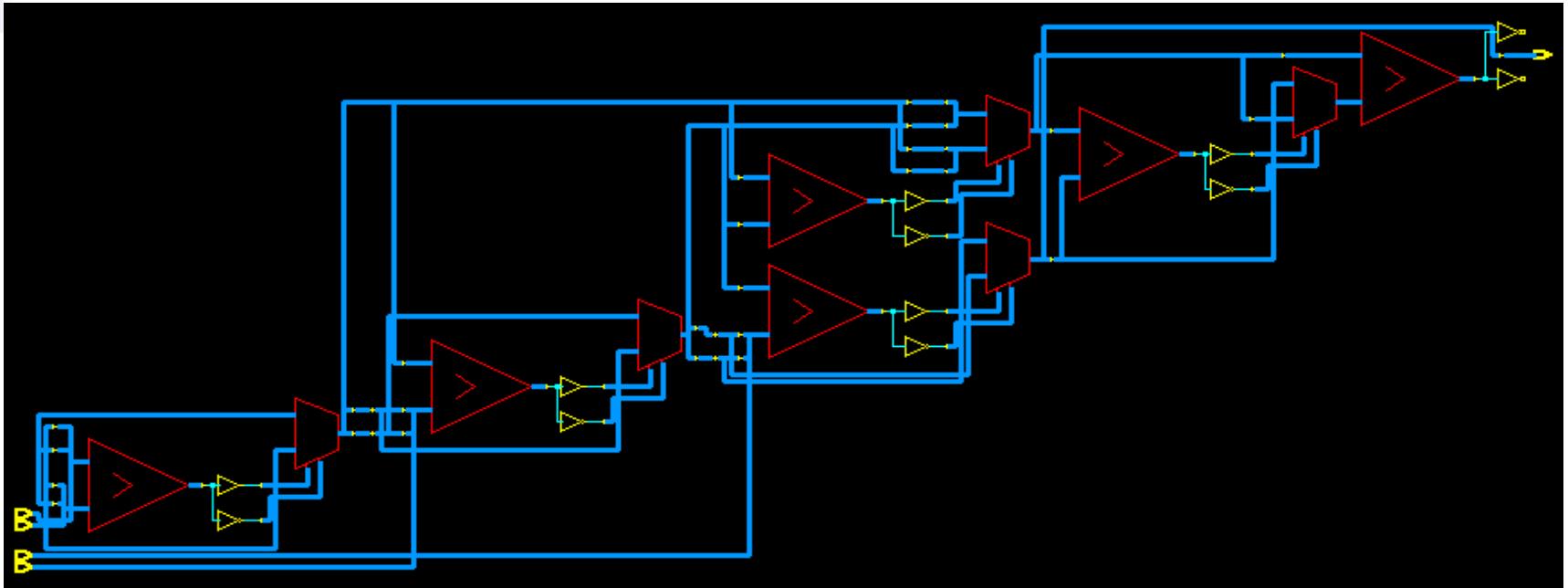
```
    out=temp[3];
```

```
end
```

```
endmodule
```

↳浪費,要依序比完才是有意義的  
其他都是無效比較

# Sorting Problem (2/2)



Bubble Sort: (four inputs)

(0)?(1), (1)?(2), (2)?(3)

(0)?(1), (1)?(2)

(0)?(1)      **critical path=6.01 ns**

Totally, 6 comparators are needed  
for parallel comparisons

Bubble Sort: (five inputs)

(0)?(1), (1)?(2), (2)?(3), (3)?(4)

(0)?(1), (1)?(2), (2)?(3)

(0)?(1), (1)?(2)

(0)?(1)      **critical path=8.05 ns**

Totally, 10 comparators are needed

**The more inputs, the longer delay** ➡ **use one comparator and a suitable FSM**



# Always Block (1/3)

An always block can imply latches or flip-flops, or it can specify purely combinational logic. An always block can contain logic triggered in response to a change in a level (asynchronous triggers) or the rising or falling edge of a signal (synchronous triggers).

The syntax of an always block is

```
always @ (event-expression )  
begin  
    ... statements ...  
end (combinational circuit)
```

**Completely specify sensitivity lists to avoid error**

## Asynchronous triggers

```
always@ (a or b or c)  
begin  
    x=a | b | c;  
end
```

x is recalculated as soon as any input (a or b or c) has a level transition (0 to 1 or 1 to 0).

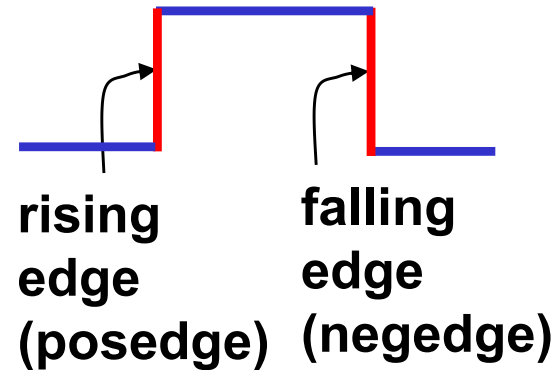
# Always Block (2/3)

always @ ( [posedge or negedge] event)

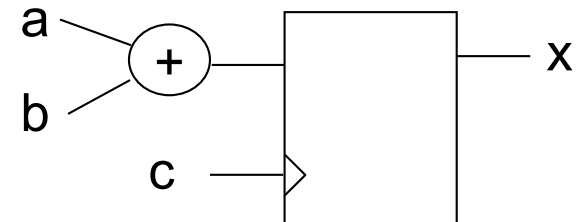
begin

... statements ...

end (sequential circuit)



Rising edge or positive edge (posedge)  
Falling edge or negative edge (negedge)



## Synchronous triggers

always@ (posedge c)

begin

x=a +b;

end

↳ C 訊號正緣來時,  $x = a + b$

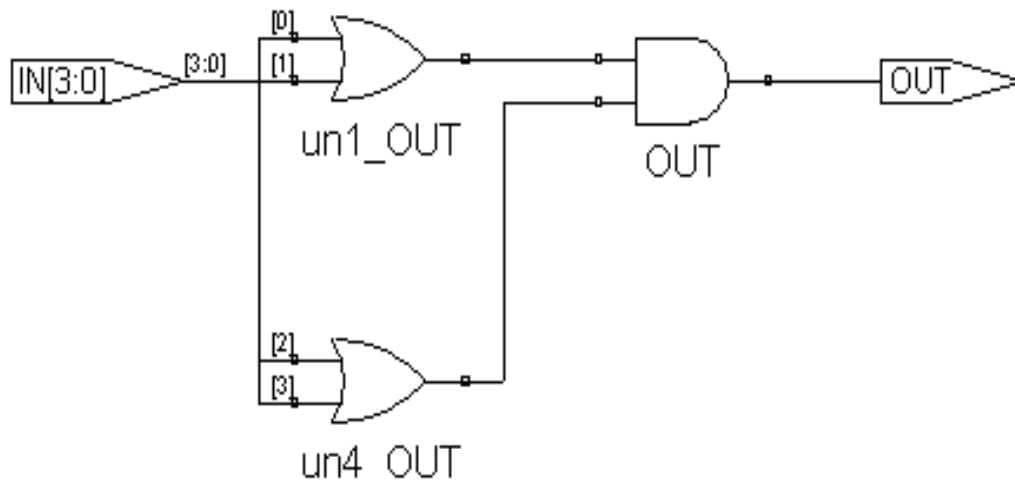
storage unit

x is recalculated as soon as  
c changes from 0 to 1

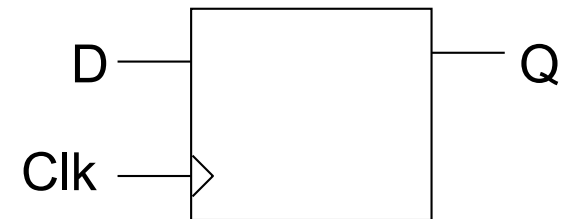


# Always Block (3/3)

```
module ALWAYS_BLOCK(IN , OUT);  
input [3 : 0]IN; output OUT; reg OUT;  
  
always @(IN)  
begin  
    OUT = (IN[0] | IN[1]) & (IN[2] | IN[3]);  
end  
endmodule
```



```
module D_FF(Clk, D, Q);  
input Clk, D;  
output Q;  
Reg Q;  
  
always @(posedge Clk)  
begin  
    Q=D;  
end  
endmodule
```



At every positive edge of signal Clk, Q is set as D.



# Variables

Nets:

```
wire a; // 1-bit wire
wire [3:0] b; // 4-bit wire
```

## Three types of common variables in Verilog:

- (1) register (default width is 1 bit)
- (2) integer (default width is 32 bit) ⇒ 成本較高
- (3) parameter (default width is 1 bit)

```
reg a; // 1-bit register
reg [3:0] b; // 4-bit register
integer c; // single 32-bit integer
parameter d=4, e=6;
```

```
parameter [range] identifier=expression,
        identifier=expression;
```

Three variables (register, integer and parameter) are declared globally at the module level, or locally at the function level and begin-end block.

Verilog allows you to assign the value of a reg variable only within a function or an always block.



# Function

---

```
function [range] name_of_function;  
    function declaration  
    statement  
endfunction
```

1. Begin with function and end with endfunction
2. [Range] defines the width of the return value of the function (default is 1 bit)  
Contain one or more statements (enclosed inside a begin-end pair)
3. You can call function in a continuous assignment, always block or other functions

Function declaration:

**Input declaration:** specify the input signals for a function

**Output:** The output from a function is assigned to the function name.

Use concatenation operation to bundle several values for multi-outputs

# Function Statements (1/4)

Procedure assignments are assignment statements used inside a function.  
(It is similar to C language, Note: it cannot be used in module)

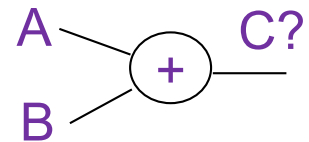
They are similar to the continuous assignments, except that the left side of a procedural assignment can contain only reg and integer variables.

```
module FUN_STATE(A , B , C1 , C2 , C3 , C4 , C5);  
input [3 : 0]A; input [3 : 0]B;  
output [6 : 0]C1;           //0  
output [2 : 0]C2;           //discard  
output [4 : 0]C3;           //always  
output [4 : 0]C4;           //assign  
reg [6 : 0]C1; reg [2 : 0]C2;  
reg [4 : 0]C3; reg [4 : 0]C4;
```

```
function [4 : 0]Fn1;  
input [3 : 0]A; input [3 : 0]B;  
    Fn1 = A + B;           // like C  
endfunction
```

每次多呼叫一次仍會多生成一份硬體

```
always @(A or B)  
begin  
    C1 = Fn1(A , B);  
end  
always @(A or B)  
begin  
    C2 = Fn1(A , B);  
end
```



```
always @(A or B)  
begin  
    C3 = A + B;  
end
```

```
assign C4 = A + B;
```

Three different ways to implement addition

```
endmodule
```

# Function Statements (2/4)

```
input [3 : 0]A;      input [3 : 0]B;      // A and B are 4-bit values
output [6 : 0]C1;    // C1 is 7-bit values
output [2 : 0]C2;    // C2 is 3-bit values
```

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

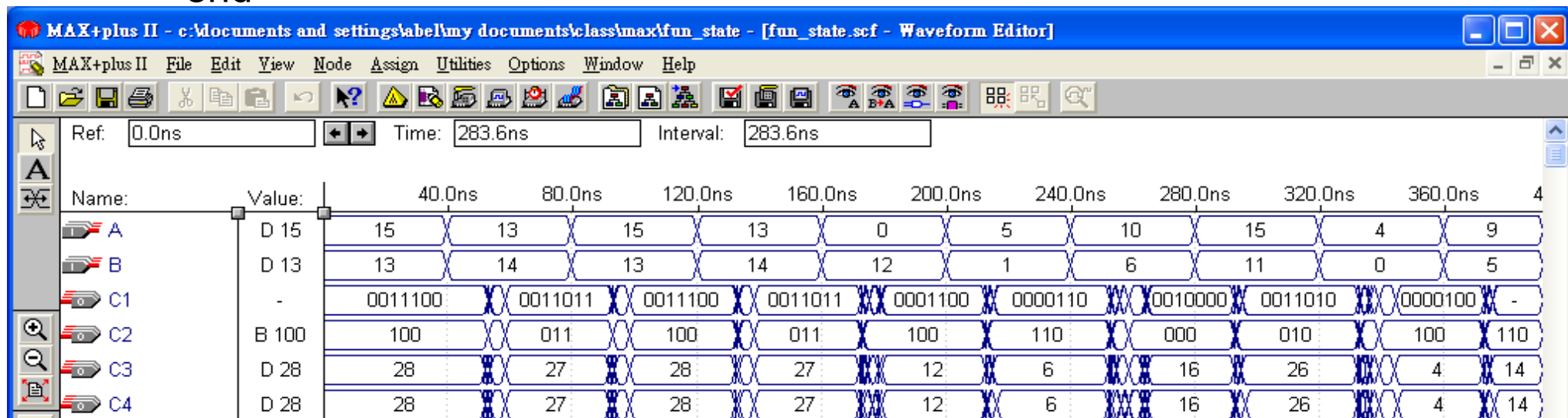
```
    C1 = A + B;      // insert "0" in the first two bits
```

```
end
```

```
begin
```

```
    C2 = A + B ;    // discard the first two bits
```

```
end
```



# Function Statements (3/4)

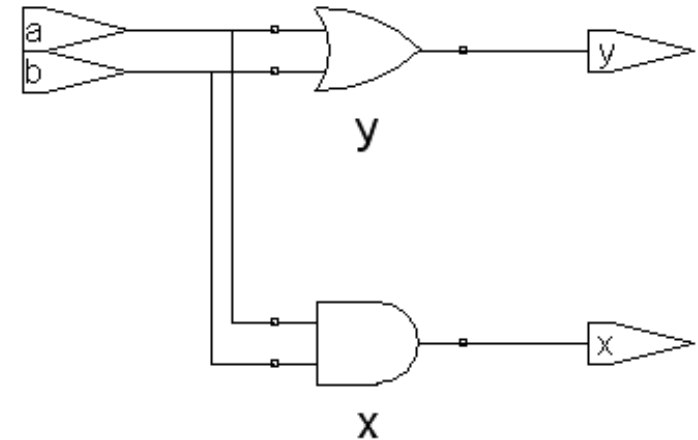
```
module test_n(a, b, x, y);
input  a, b; output x, y;
reg    x, y;
function Fn1;
  input a, b;
  Fn1 = a & b;
  /* begin-end is required
  for more statements */
endfunction
```

```
function Fn2;
  input a, b;
  Fn2 = a | b;
endfunction
```

```
always @(a or b)
begin
  x = Fn1(a, b);
  y = Fn2(a, b);
end
endmodule
```

```
module test_n(a, b, x, y);
input  a, b;
output x, y;
assign x=a & b;
assign y=a | b;
endmodule
```

```
module test_n(a, b, x, y);
input  a, b;
output x, y;
reg    x, y;
always @(a or b)
begin
  x = a & b;
  y = a | b;
end
endmodule
```



## C language

```
x = a&b;
y = a | b;
```



# Function Statements (4/4)

```
module test_n(a1, a, b, x, y);  
input [7:0] a1;  
input      a, b;  
output     x, y;  
reg        x, y;
```

```
function Fn1;  
input [width-1:0] a1;  
parameter width=8;//error message  
    input a, b;  
    Fn1 = a & b;  
endfunction
```

```
always @(a or b)  
begin  
    x = Fn1(a1, a, b);  
end  
endmodule
```

```
module test_n(a1, a, b, x, y);  
input [7:0] a1;  
input      a, b;  
output     x, y;  
reg        x, y;
```

```
function Fn1;  
parameter width=8;  
input [width-1:0] a1; // OK  
    input a, b;  
    Fn1 = a & b;  
endfunction
```

```
always @(a or b)  
begin  
    x = Fn1(a1, a, b);  
end  
endmodule
```

Can we input width with scanf or input ??  
Why?

# Example-Function

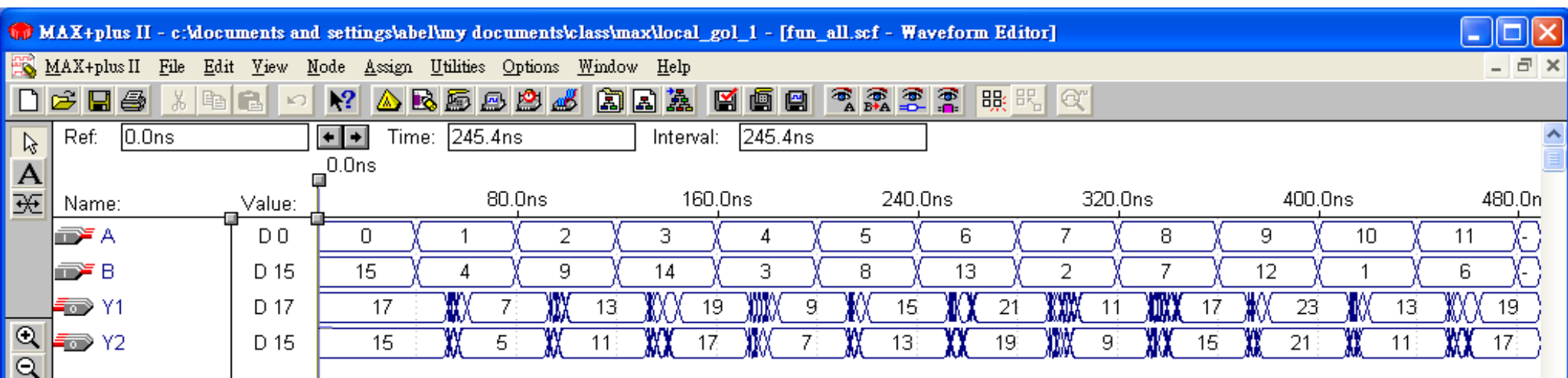
```
module FUN_ALL(A , B , Y1 , Y2);
input [3 : 0] A,B;output [4 : 0] Y1, Y2;
reg [4 : 0] Y1, Y2;
```

```
function [4 : 0] Fn2;
input [3 : 0] F1 , F2;
begin
    Fn2 = F1 + F2;
end
endfunction
```

```
function [4 : 0] Fn1;
input [3 : 0]F1 , F2;
begin
    Fn1 = Fn2(F1 , F2) + 2;
end
endfunction
```

```
always @(A or B)
begin
    Y1 = Fn1(A , B);
    Y2 = Fn2(A , B);
end
endmodule
```

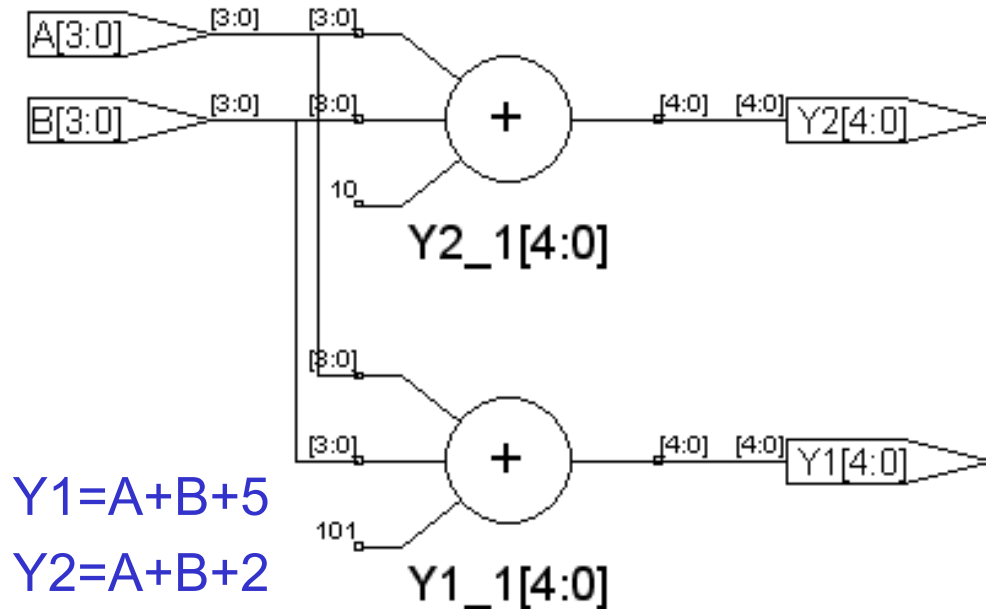
$Y1 = A + B + 2$   
 $Y2 = A + B$





# Example-Function with Many Outputs

```
function [9 : 0]Fn1;
input [3 : 0] F1 , F2;
reg [4 : 0] Y1_1; reg [4 : 0] Y2_1;
begin
    Y1_1 = F1 + F2 + 5;
    Y2_1 = F1 + F2 + 2;
    Fn1 = {Y1_1 , Y2_1};
end
endfunction
assign {Y1 , Y2} = Fn1(A , B);
endmodule
```



$$Y1 = A + B + 5$$

$$Y2 = A + B + 2$$

$Y1\_1$  and  $Y2\_1$  can be declared as reg or integer, not wire

