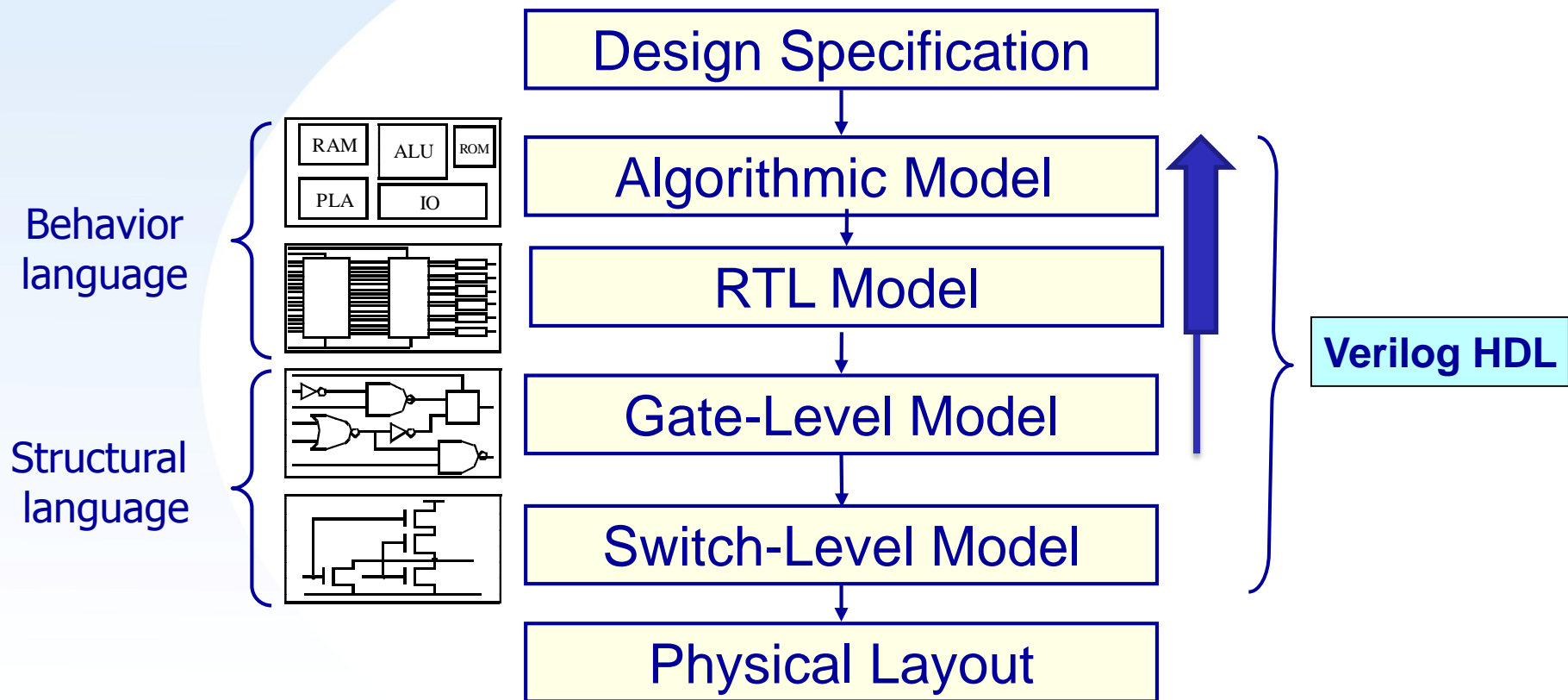


# Design Scope of Verilog

✓ **We use Verilog HDL to design**

— Typical design flow



# What is Verilog?

✓ **Hardware** Description Language

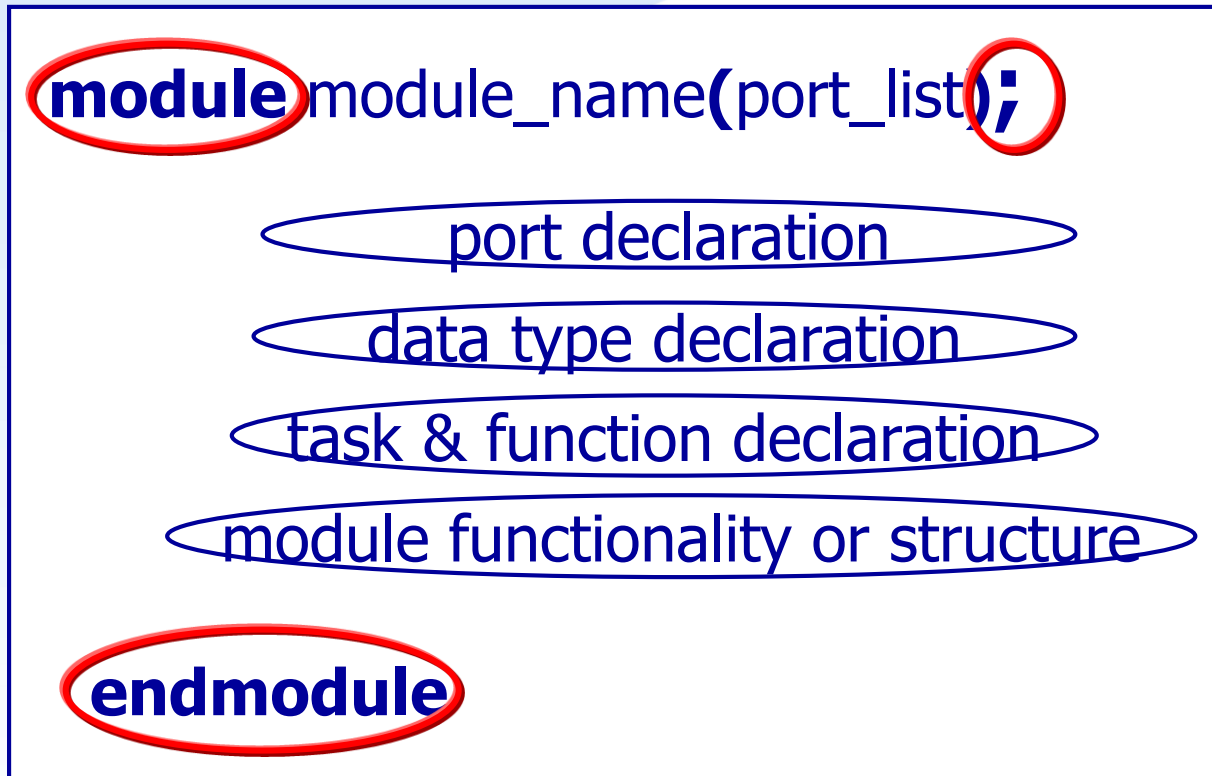
✓ **Hardware** Description Language

✓ **Hardware** Description Language



# Lexical Convention : A Module

- ✓ Encapsulate structural and functional details



```
module test ( Q,S,clk );  
output Q;  
input  S,clk;  
reg    Q;  
always@(S or clk)  
        Q<=(S&clk) ;  
endmodule
```

- ✓ All modules run **concurrently**
- ✓ Encapsulation makes the model available for instantiation in other modules



# Lexical Convention : Identifier and Comment

- ✓ Verilog is a **case sensitive** language
- ✓ Terminate lines with **semicolon ;**
- ✓ Identifiers
  - starts **only** with a letter or an **\_**(underscore), can be any sequence of letters, digits, \$, **\_**.
  - case-sensitive !
    - e.g. shiftreg\_a
    - \_bus3
    - n\$657
    - 12\_reg → **illegal !!!!**
- ✓ Comments
  - single line : **//**
  - multiple line : **/\* ... \*/**



# Lexical Convention : Number

## ✓ Number Specification

### – **<size>'<base><value>**

- <size> is the length of desired value in bits.
- <base> can be b(binary), o(octal), d(decimal) or h(hexadecimal).
- <value> is any legal number in the selected base.

[When <size> is *smaller than* <value>: left-most bits of <value> are truncated  
When <size> is *larger than* <value>, then left-most bits are filled based on the value of the left-most bit in <value>.]

◆ Left most '0' or '1' are filled with '0', 'Z' are filled with 'Z' and 'X' with 'X'

- Default size is 32-bits decimal number
- e.g. 4'd10 → 4-bit, 10, decimal
- e.g. 6'hca → 6-bit, store as 6'b001010 (truncated, not 11001010!)
- e.g. 6'ha → 6-bit, store as 6'b001010 (filled with 2-bit '0' on left!)
- Extension:
  - 12'hz → zzzz zzzz zzzz; 6'bx → xx xxxx; 8'b0 → 0000 0000; 8'b1 → 0000 0001

### – **Negative : -<size>'<base><value>**

- e.g. -8'd3 → legal      8'd-3 → illegal



# Lexical Convention : Operator

## ✓ Operators

### – Arithmetic Description

- $A = B + C;$
- $A = B - C;$
- $A = B * C;$
- $A = B / C;$
- $A = B \% C;$  modulus → some synthesis tools don't support this operator

### – Shift Operator (**bit-wise**)

- $A = B >> 2;$  → shift right 'B' by 2-bit
- $A = B << 2;$  → shift left 'B' by 2-bit

### – Shift Operator (**arithmetic**)

- $A = B >>> 2;$  ">>>>", "<<<" are used only for 'signed' data type in Verilog 2001
- $A = B <<< 2;$
- e.g.  $B = 4'b1000;$  ( $A = 4'b1110$ , which is 1000 shifted to the right two positions and sign-filled.)  
 $A = B >>> 2;$



# Lexical Convention : Operator

## ✓ Bit-wise Operator

- NOT:  $A = \sim B;$
- AND:  $A = B \& C;$
- OR:  $A = B | C;$
- XOR:  $A = B \wedge C;$
- e.g.  $4'b1001 | 4'b1100 \rightarrow 4'b1101$

## ✓ Logical Operators: return 1-bit true/false

- NOT:  $A = ! B;$
- AND:  $A = B \&\& C;$
- OR:  $A = B || C;$
- e.g.  $4'b1001 || 4'b1100 \rightarrow \text{true}, 1'b1$

## ✓ Conditional Description

- if else
- case endcase
- $? : \rightarrow c = \text{sel} ? a : b;$   

```
// if (sel==1'b1)
//   c = a;
// else
//   c = b;
```

## ✓ Relational and equality(conditional)

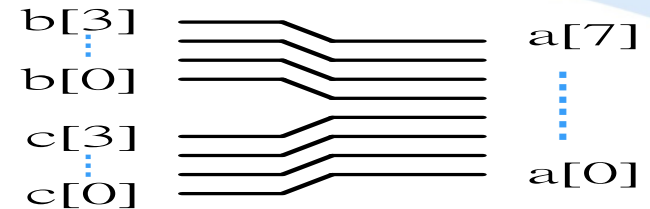
- $<=, <, >, >=, ==, !=$
- i.e.  $\text{if}((a <= b) \&\& (c == d) || (e > f))$



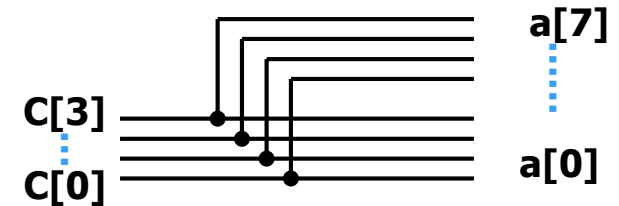
# Lexical Convention : Concatenation

## Concatenation

- $\{ \}$   $\rightarrow a = \{b, c\};$



- $\{\{ \}\}$   $\rightarrow a = \{2\{c\}\};$



- $a[4:0] = \{b[3:0], 1'b0\}; \Leftrightarrow a = b \ll 1;$





# Data Type

- ✓ 4-value logic system in Verilog: 0, 1, x or X, z or Z
- ✓ Declaration Syntax <data\_type>[<MSB> : <LSB>]<list\_of\_identifier>
  - **Nets** : represent physical connections between devices (**default=z**)
    - represent connections between things
    - Cannot be assigned in an initial or always block
  - **Register** : represent abstract data storage element (**default=x**)
    - represent data storage
    - Hold their value until explicitly assigned in an initial or always block
    - Can be used to model latches, flip-flops, etc., but do not correspond exactly

Register type	Attribute
reg	Unsigned value with Varying bit width
integer	32-bit <b>signed</b> (2's complement)
time	64-bit <b>unsigned</b>
real	Real number



# Data Type : Vector and Array

- Vectors : the wire and register can be represented as a vector

- wire [7:0] vec; → 8-bit bus
- reg [0:7] vec; → vec[0] is the MSB

- Arrays : <array\_name>[<subscript>]

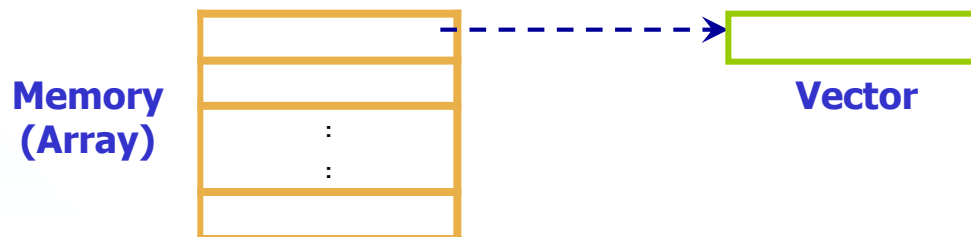
→ It isn't well for the backend verifications

- integer mem[0:7] → (8x32)-bit mem
- reg [7:0] mem[0:1023] → Memories!! (1k - 1byte)

For this reason, we do not use array as memory, Memory component will be introduced later

- What's difference between Vector and Array?

- **Vector** : single-element with multiple-bit
- **Array** : multiple-element with multiple-bit



# Port : Module Connection

## ✓ Port ordering

- one port per line with appropriate comments
- inputs first then outputs
- clocks, resets, enables, other controls, address bus, then data bus

## ✓ Modules connected by port order (implicit)

- Here order shall match correctly. Normally, it not a good idea to connect ports implicitly. It **could cause problem in debugging** when any new port is added or deleted.
- e.g. : FA U01( A, B, CIN, SUM, COUT );

## ✓ Modules connect by name (explicit)



Use this!!!

- Use **named** mapping instead of **positional** mapping
- name shall match correctly.
- e.g. : FA U01( .a(A), .b(B), .cin(CIN), .sum(SUM), .cout(COUT) );  
**foo u\_foo1(4'h2, 4'h5, 4'h8) ; X**



# Data Assignment

## ✓ Continuous Assignment → **for wire assignment**

- Imply that whenever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS.

- e.g. 

```
wire [3:0] a;  
assign a = b + c;           //continuous assignment
```

## ✓ Procedural Assignment → **for reg assignment**

- assignment to “**register**” data types may occur within *always*, *initial*, *task* and *function*. These expressions are controlled by triggers which cause the assignment to evaluate.

- e.g. 

```
reg a,clk;  
always #5 clk = ~clk;       //procedural assignment
```
- e.g. 

```
always @ (b)                //procedural assignment with triggers  
a = ~b;
```



# Data Assignment (cont.)

In the behavior-level modeling, Verilog code is just like C language. In the behavior level, it uses two essential statements.

## ✓ initial

- An initial block *starts at 0*, and executes once in a simulation.

## ✓ always

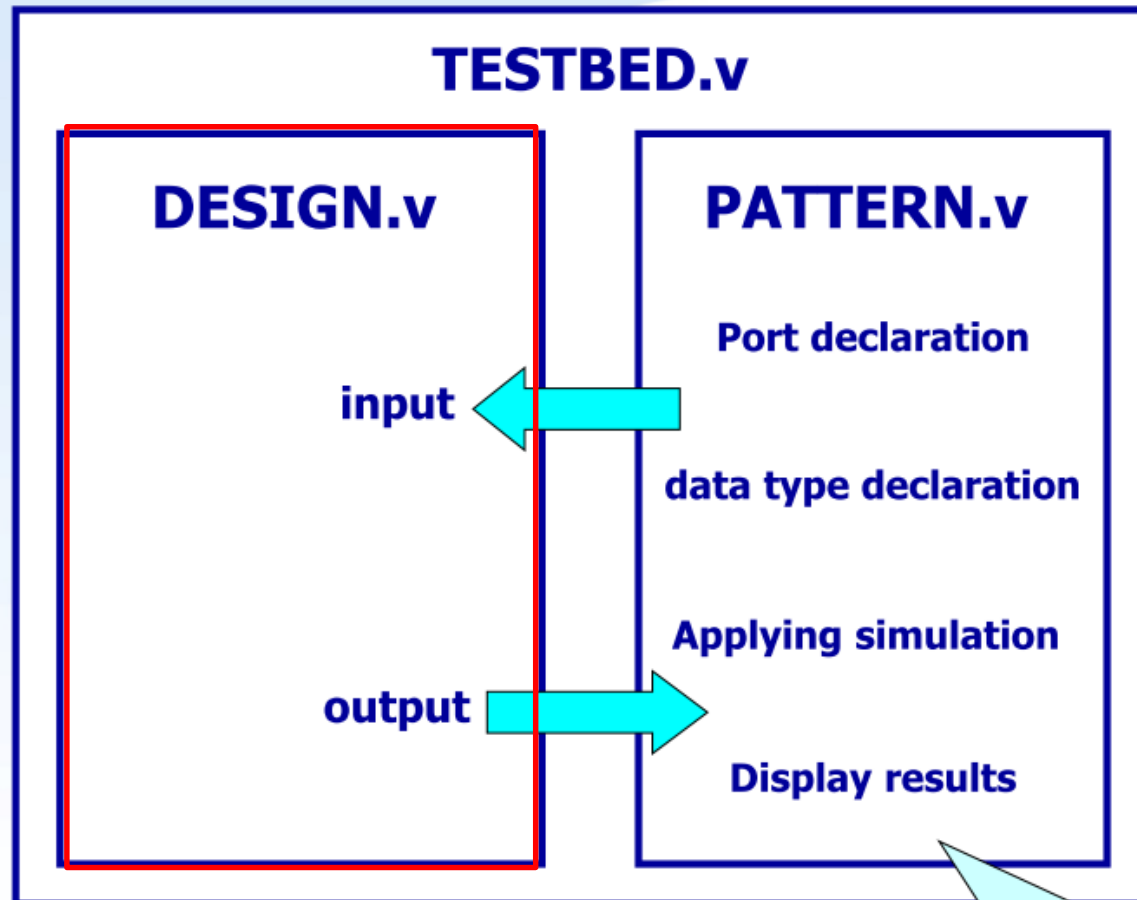
- An always block *starts at 0*, and executes repeatedly as a loop.

```
module example;
.
.
initial clk=1'b0;
always #10 clk=~clk;
initial //multiple statements uses
begin //begin-end to be grouped
    $display ("end");
    #1000 $finish ;
end
endmodule
```

**Don't forget the corresponding hardware when writing design!**



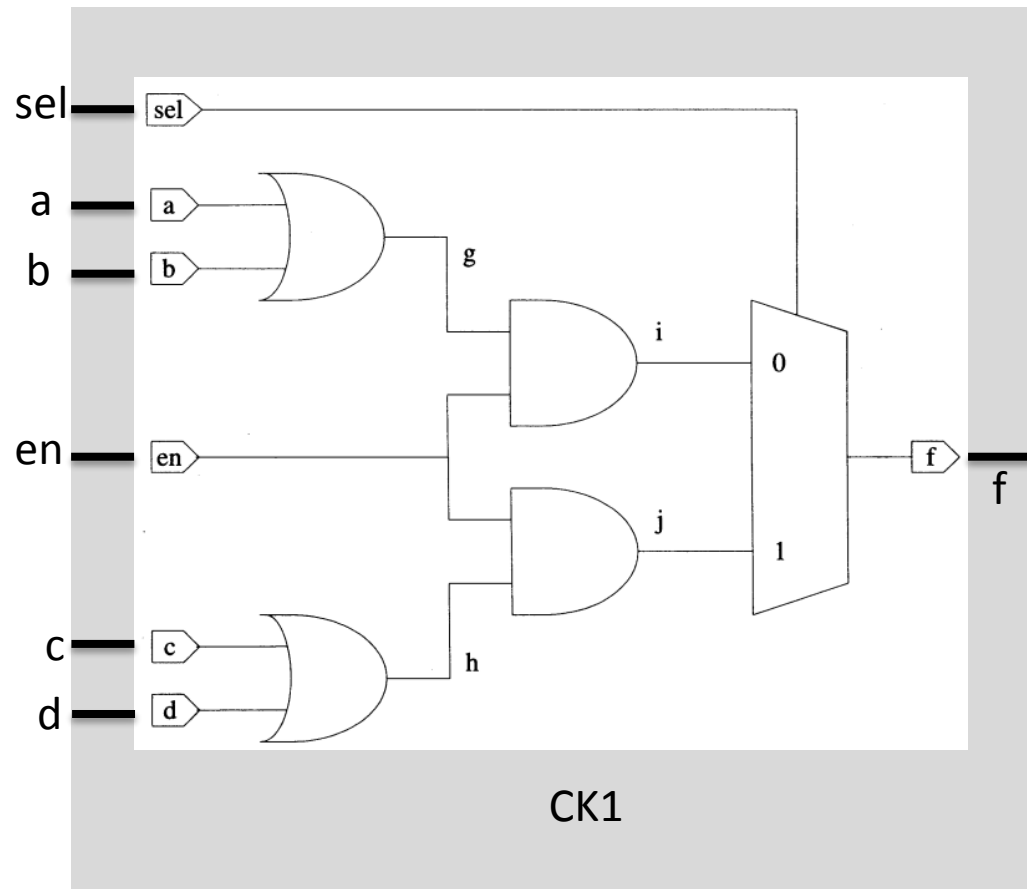
# Simulation Environment



Can use behavior-level



```
1 module
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22 endmodule
```

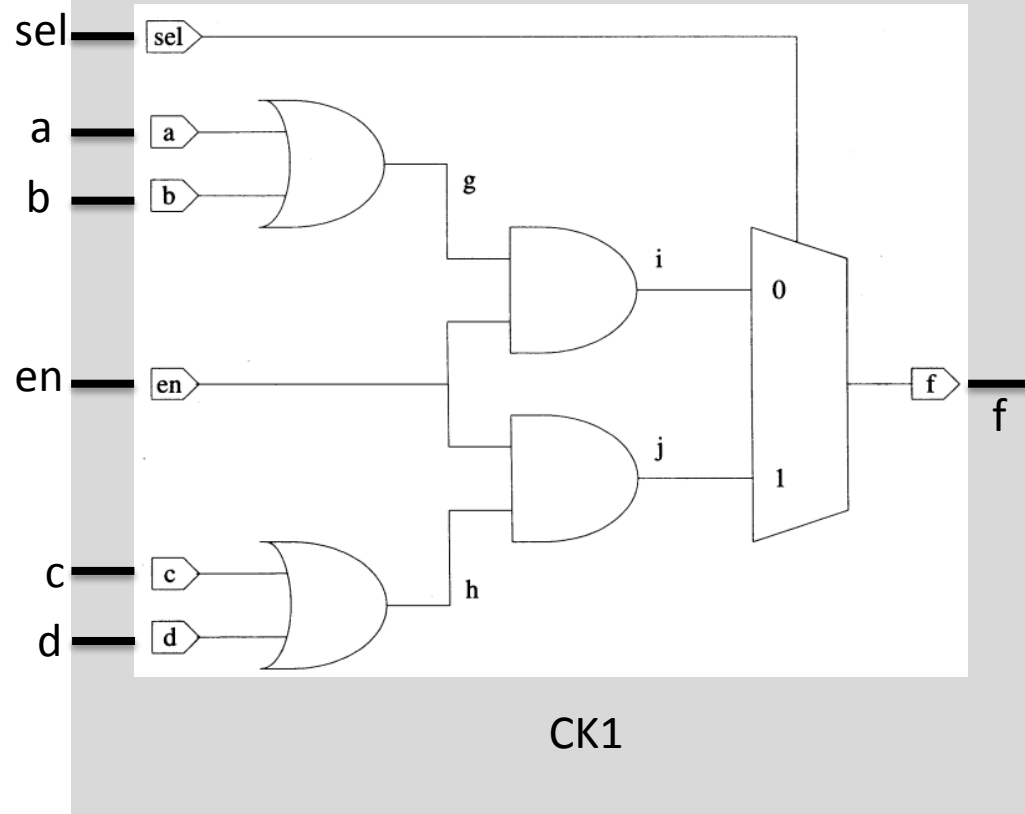


module name

module 對外的接線  
(in/out port)

分號結尾

```
1  module CK1(a, b, c, d, en, sel, f);
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22  endmodule
```



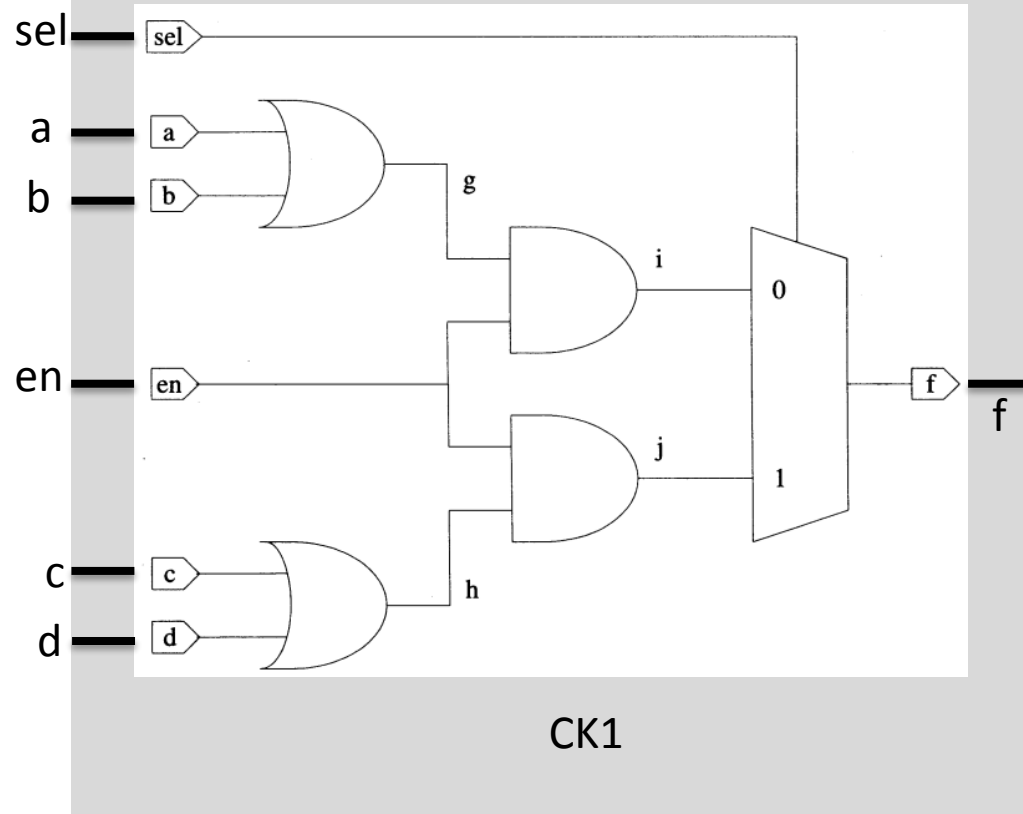


module name

module 對外的接腳  
(in/out port)

分號結尾

```
1 module CK1(a, b, c, d, en, sel, f);
2
3 input a, b, c, d, en, sel;
4 output f;
5 //針對對外的接腳, 定義 input/output
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22 endmodule
```

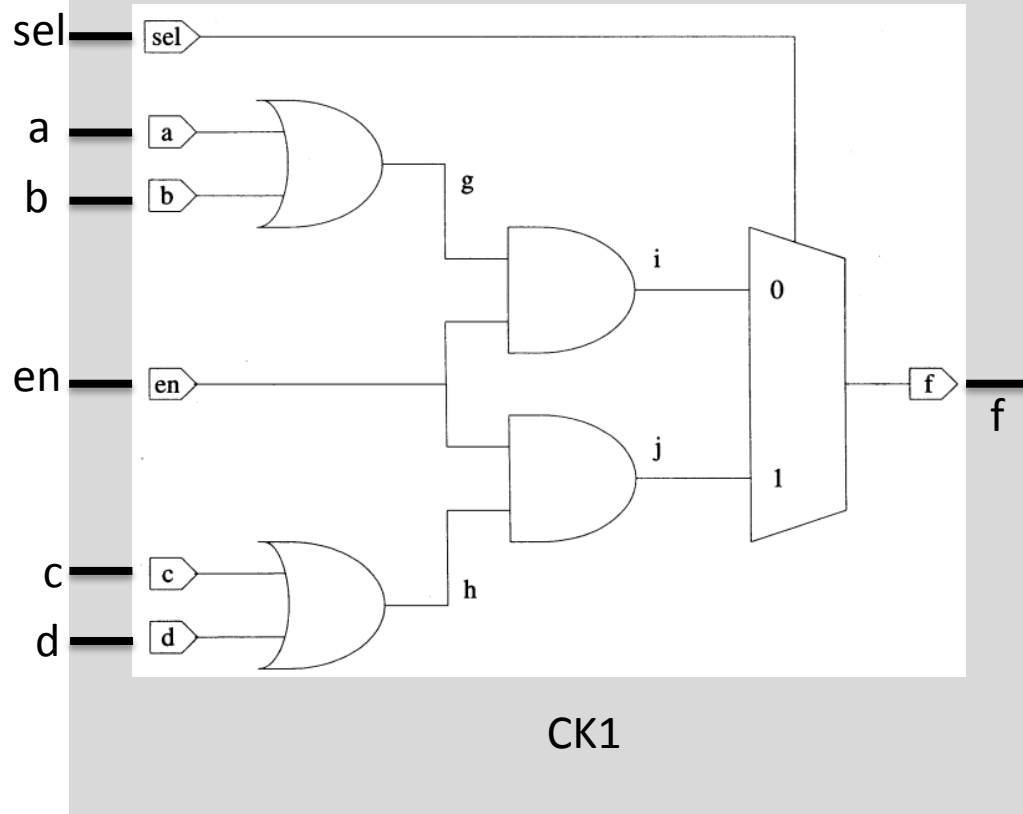


module name

module 對外的接腳  
(in/out port)

分號結尾

```
1  module CK1(a, b, c, d, en, sel, f);
2
3  input  a, b, c, d, en, sel;
4  output f;
5  //針對對外的接腳, 定義 input/output
6  wire  f;
7  //定義output腳位的data type
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22  endmodule
```

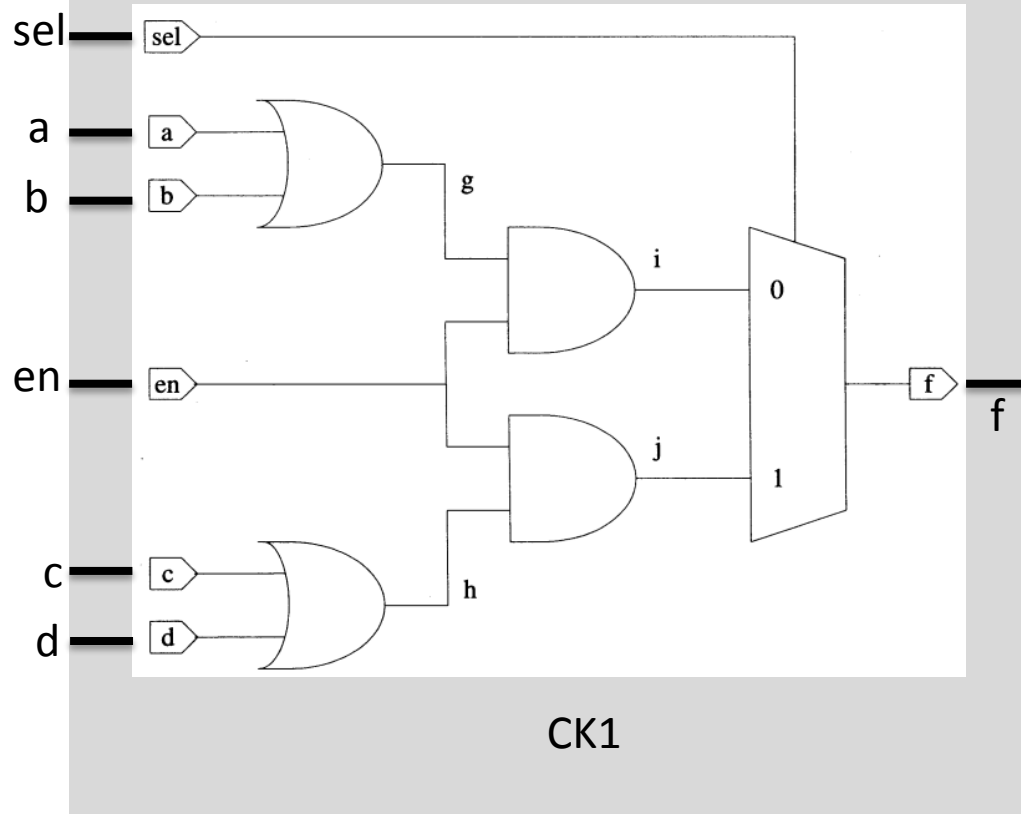


module name

module 對外的接腳  
(in/out port)

分號結尾

```
1  module CK1(a, b, c, d, en, sel, f);
2
3  input  a, b, c, d, en, sel;
4  output f;
5  //針對對外的接腳, 定義 input/output
6  wire  f;
7  //定義output腳位的data type
8
9  wire  g, h, i, j;
10 //定義內部接線的data type
11
12
13
14
15
16
17
18
19
20
21
22  endmodule
```

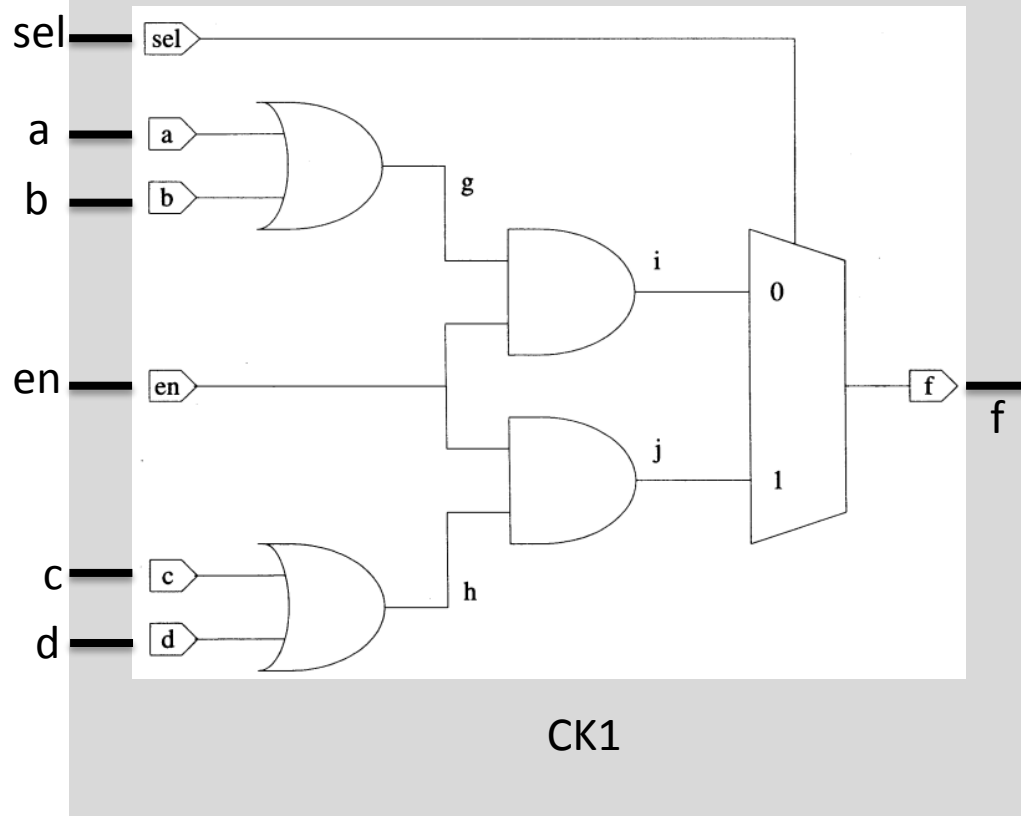


module name

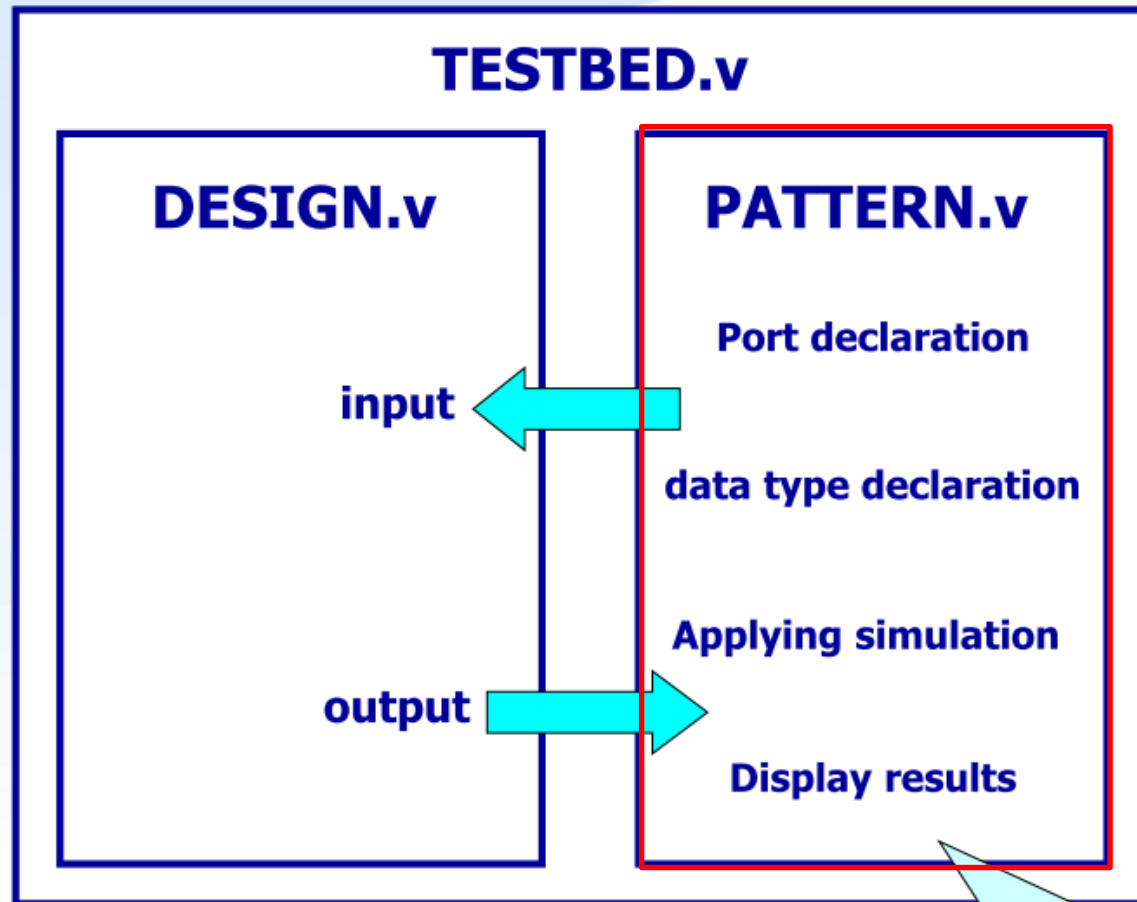
module 對外的接腳  
(in/out port)

分號結尾

```
1  module CK1(a, b, c, d, en, sel, f);
2
3  input  a, b, c, d, en, sel;
4  output f;
5  //針對對外的接腳, 定義 input/output
6  wire  f;
7  //定義output腳位的data type
8
9  wire  g, h, i, j;
10 //定義內部接線的data type
11 assign g = a|b;
12
13 assign i = g&en;
14
15 assign h = c|d;
16
17 assign j = h&en;
18
19 assign f = (sel==1'b0)?i:j;
20 //描述電路的行為
21
22 endmodule
```



# Simulation Environment



Can use behavior-level



```

module PATTERN(P_a, P_b, P_c, P_d, P_en, P_sel, P_f);

output reg P_a, P_b, P_c, P_d, P_en, P_sel;
input P_f;

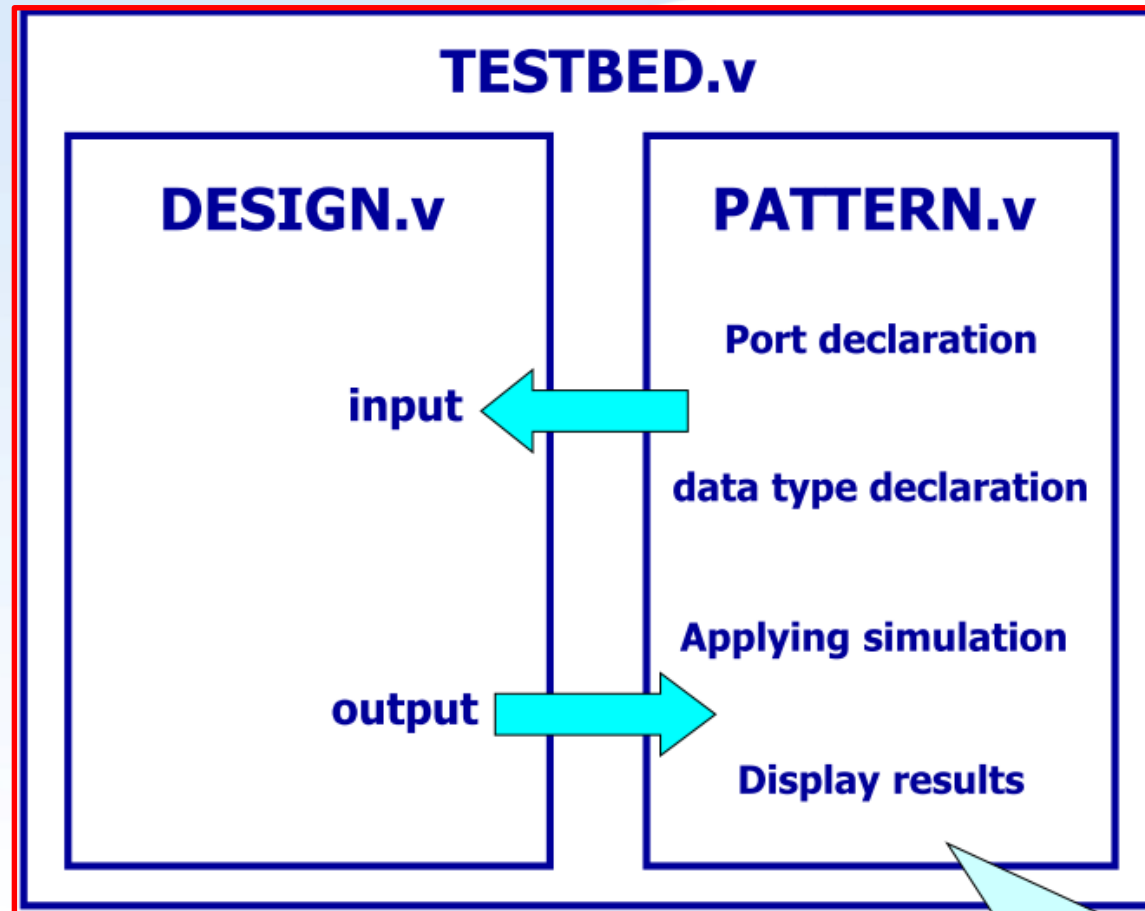
//-----
//          CLK , RESET control
//-----
parameter CYCLE = 4;
always #(CYCLE/2.0) CLK=~CLK;    //產生CLK

initial
begin
    P_a = 0;                      //設初值
    P_b = 0;
    P_c = 0;
    P_d = 0;
    P_en= 0;
    P_sel=0;

    @(negedge CLK);              //in_1
    P_a = 1;
    P_b = 0;
    P_c = 1;
    P_d = 0;
    P_en= 1;
    P_sel=1;
end

```

# Simulation Environment



Can use behavior-level



```
`timescale 1ns/10ps
`include "PIE.v"
`include "PATTERN.v"

module TESTBED;
wire T_a, T_b, T_c, T_d, T_en, T_sel, T_f;

    CK1 T_CK(
        .a(T_a),
        .b(T_b),
        .c(T_c),
        .d(T_d),
        .en(T_en),
        .sel(T_sel),
        .f(T_f));

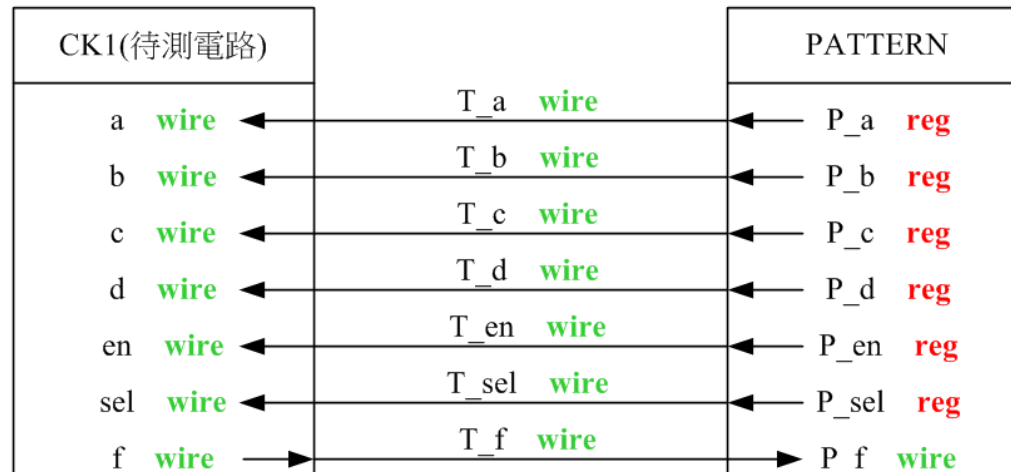
    PATTERN T_PATTERN(
        .P_a(T_a),
        .P_b(T_b),
        .P_c(T_c),
        .P_d(T_d),
        .P_en(T_en),
        .P_sel(T_sel),
        .P_f(T_f));

initial begin
    $dumpfile("PIE.fsdb");
    $dumpvars;
end

endmodule
```



## TESTBED



```
module CK1(a, b, c, d,
           en, sel, f);

input  a, b, c, d;
input  en, sel;

output f;
```

```
module TESTBED;

wire T_a, T_b, T_c, T_d, T_en, T_sel, T_f;

CK1 T_CK(
    .a(T_a),
    .b(T_b),
    .c(T_c),
    .d(T_d),
    .en(T_en),
    .sel(T_sel),
    .f(T_f));

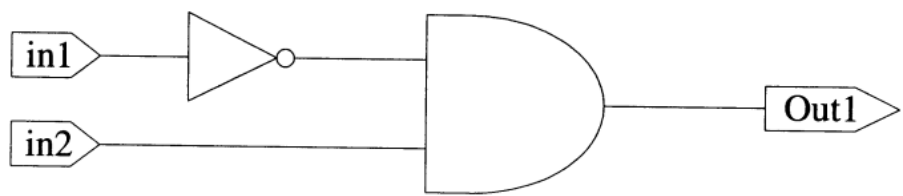
PATTERN T_PATTERN(
    .P_a(T_a),
    .P_b(T_b),
    .P_c(T_c),
    .P_d(T_d),
    .P_en(T_en),
    .P_sel(T_sel),
    .P_f(T_f));
```

```
module PATTERN(P_a, P_b, P_c, P_d,
               P_en, P_sel, P_f);

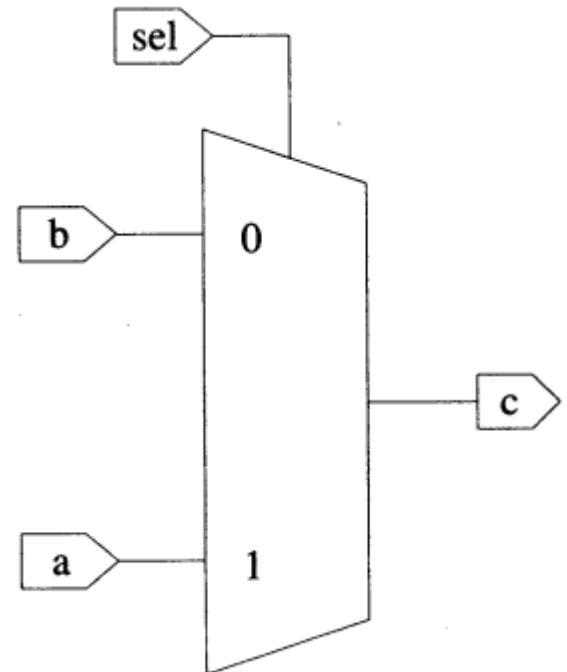
output reg P_a, P_b, P_c, P_d;
output reg P_en, P_sel;

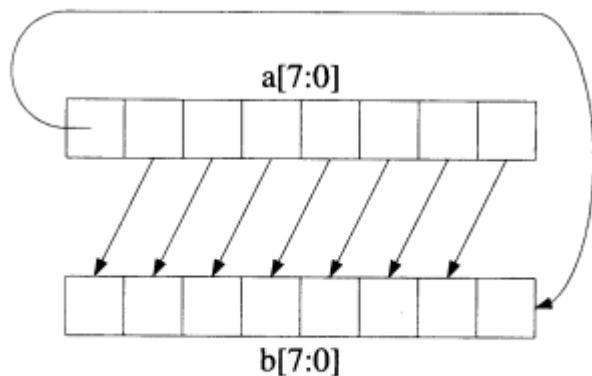
input P_f;
```

```
1  module EX2(in1, in2, out1);
2
3  input  in1, in2;
4  output out1;
5
6  wire out1;
7
8
9  assign out1 = (~in1) & in2;
10
11 endmodule
```

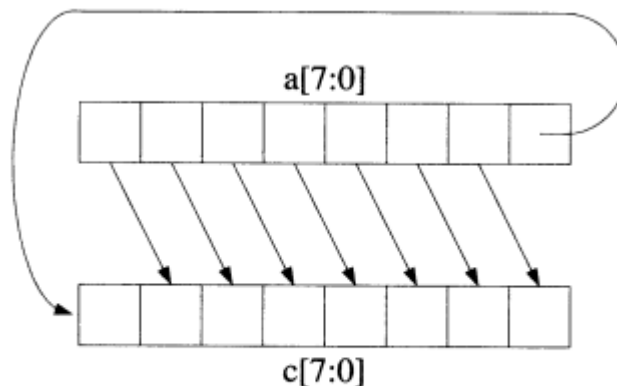


```
1  module EX4(a, b, c, sel);  
2  
3  input    a, b, sel;  
4  output   c;  
5  
6  wire     c;  
7  
8  assign   c = (sel==1'b1)? a : b;  
9  
10 endmodule
```

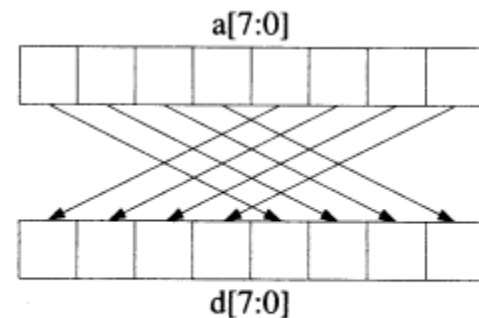




(a) 位元向左移位旋轉 (rotate left)



(b) 位元向右移位旋轉 (rotate right)



(c) nibble 位元交換 (nibble swap)

```

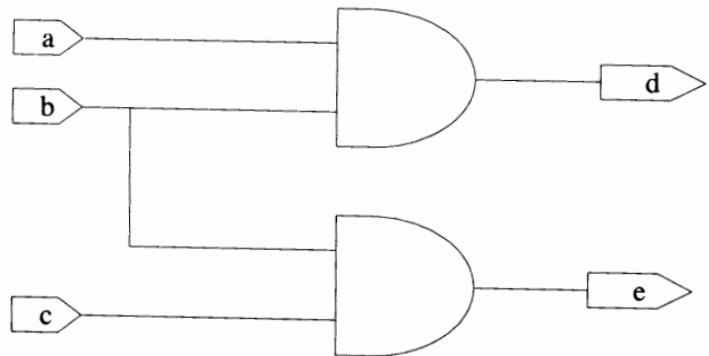
1  module EX5(a, b, c, d);
2
3  input  [7:0] a;
4  output [7:0] b, c, d;
5
6  wire  [7:0] b, c, d;
7
8  assign b = {a[6:0], a[7]};
9
10 assign c = {a[0], a[7:1]};
11
12 assign d = {a[3:0], a[7:4]};
13
14 endmodule

```

## □ 學習重點

1. 學習以 **always** 行為模式設計一般的 **組合電路** (combinational circuit)
2. 學習以 **always** 行為模式搭配 **if-else**、**case** 的整合描述
3. 了解如何避免合成後產生 **latch** 電路

# □ 以 **always** behavior model 設計電路



	begin-end 之內	begin-end 之外
data type	<u>reg</u>	wire
assign	X	Yes

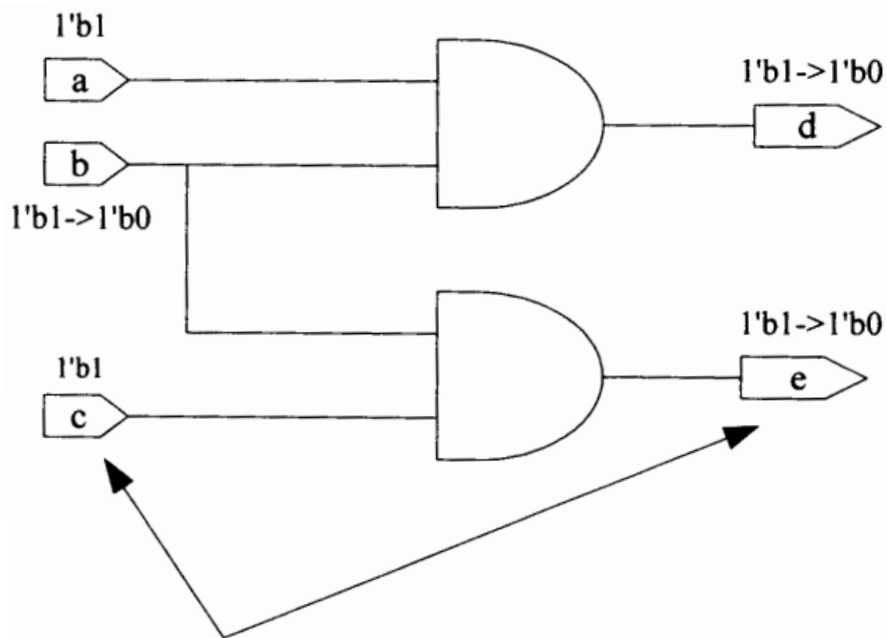
## behavior module 描述電路

```
1  module CK2_1(a, b, c, d, e);
2
3  input  a, b, c;
4  output d, e;
5
6  reg    d, e;
7
8  always @(a or b or c)
9  begin
10
11      d = a & b;
12      e = b & c;
13
14  end
15
16  endmodule
```

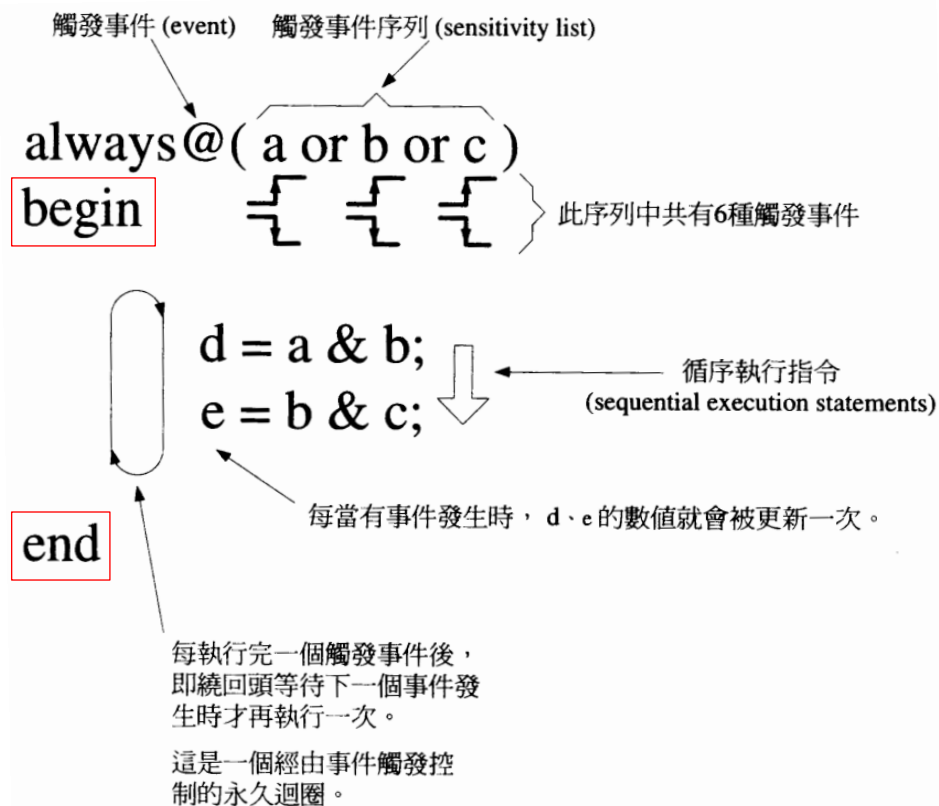
## data flow 的方式描述電路

```
1  module CK2_1(a, b, c, d, e);
2
3  input  a, b, c;
4  output d, e;
5
6  wire   d, e;
7
8
9
10
11  assign d = a & b;
12  assign e = b & c;
13
14
15
16  endmodule
```

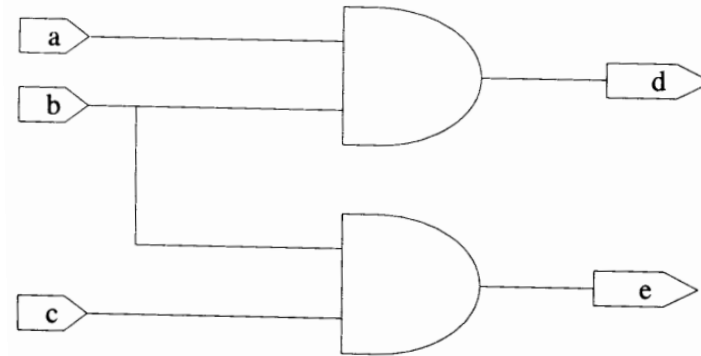
# □ **always** behavior model




當 a、b、c 有任何數值發生變化，即造成 d、e 數值在邏輯上的更新。



# □ 以 **always** behavior model 設計電路



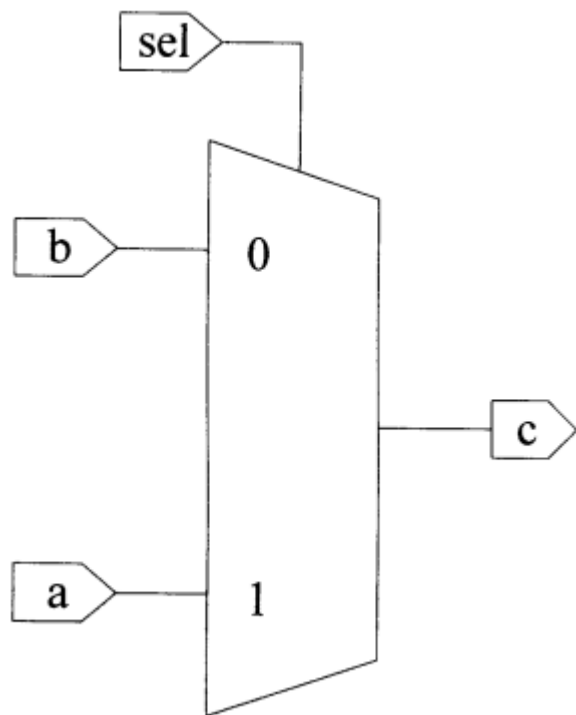
```
1  module CK2_1(a, b, c, d, e);  
2  
3  input  a, b, c;  
4  output d, e;  
5  
6  reg    d, e;  
7  
8  always @(a or b or c)  
9  begin  
10  
11      d = a & b;  
12      e = b & c;  
13  
14  end  
15  
16  endmodule
```



```
always @(*)  
begin  
    d = a & b;  
    e = b & c;  
end
```

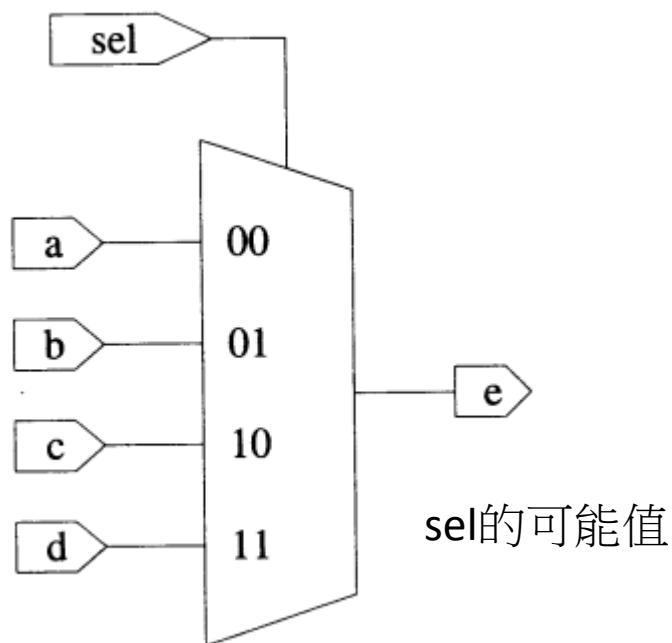


## □ 以 **always** + **if-else** 設計多工器



```
1  module CK2_2(a, b, sel, c);  
2  
3  input  a, b, sel;  
4  output c;  
5  
6  reg    c;  
7  
8  always @(*)  
9  begin  
10  
11      if(sel==1'b1)  
12          c = a;  
13      else  
14          c = b;  
15  
16  end  
17  
18  endmodule
```

# □ 以 **always** + **case** 設計多工器



```
1  module CK2_3(a, b, c, d, sel, e);
2
3  input  [7:0]  a, b, c, d;
4  input  [1:0]  sel;
5  output [7:0]  e;
6
7  reg  [7:0]  e;
8
9  always @(*)
10 begin
11
12     case (sel)
13         2'b00    : e = a;
14         2'b01    : e = b;
15         2'b10    : e = c;
16         default  : e = d;
17     endcase
18
19 end
20
21 endmodule
```

select line

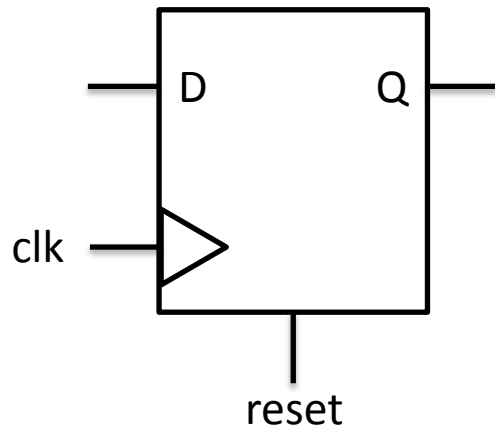
對應的運算

default代表sel的其他情況  
都會落入default值  
執行對應的運算

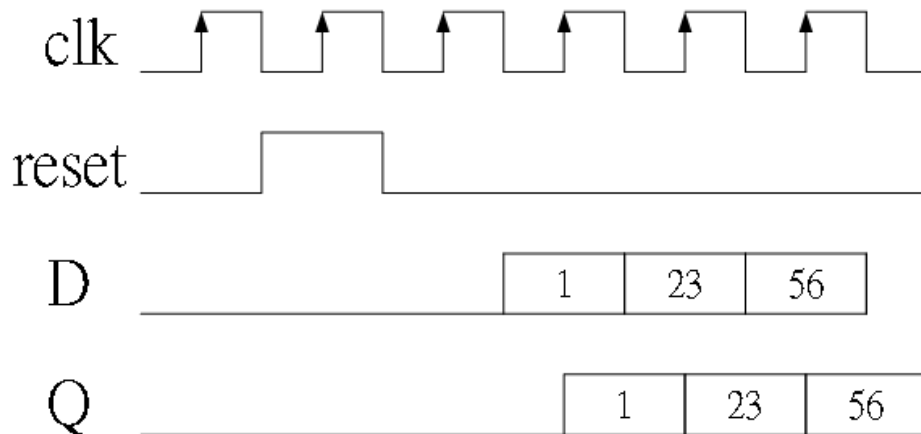
## □ 學習重點

1. 學習以**always** 行為模式設計一般的**循序電路**(sequential circuit)
2. 學習 **Finite State Machine (FSM)** 的設計方式

# □ register (DFF) 的操作原理和描述方式



```
always @(posedge clk) begin
    if(reset)
        Q <= 1'b0;
    else
        Q <= D;
end
```



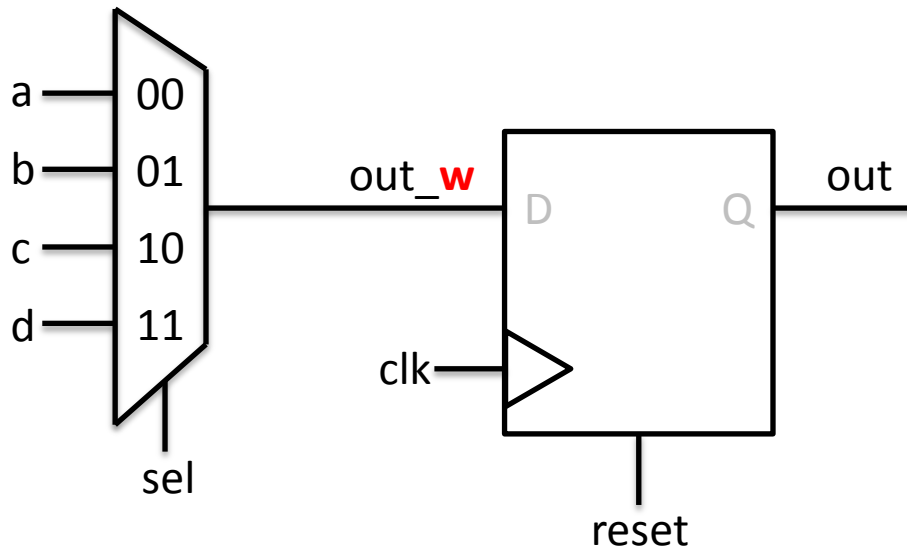
差別 1：觸發條件

改成 **posedge** clk / **negedge** clk

差別 2：non-blocking assignment

改成 **<=**

# □ 設計sequential電路的技巧



1. 命名技巧：DFF前面 input data 的接線命名成\_w

2. 將 register 和 combinational 分開寫

```
input  clk, reset;
input  a, b, c, d, sel;

output reg out;

reg out_w;

// sequential circuit
always@(posedge clk) begin

    if(reset)
        out <= 1'b0;
    else
        out <= out_w;

end

// combinational circuit
always@(*) begin

    case(sel)
        2'b00 : out_w = a;
        2'b01 : out_w = b;
        2'b10 : out_w = c;
        default : out_w = d;
    endcase

end
```