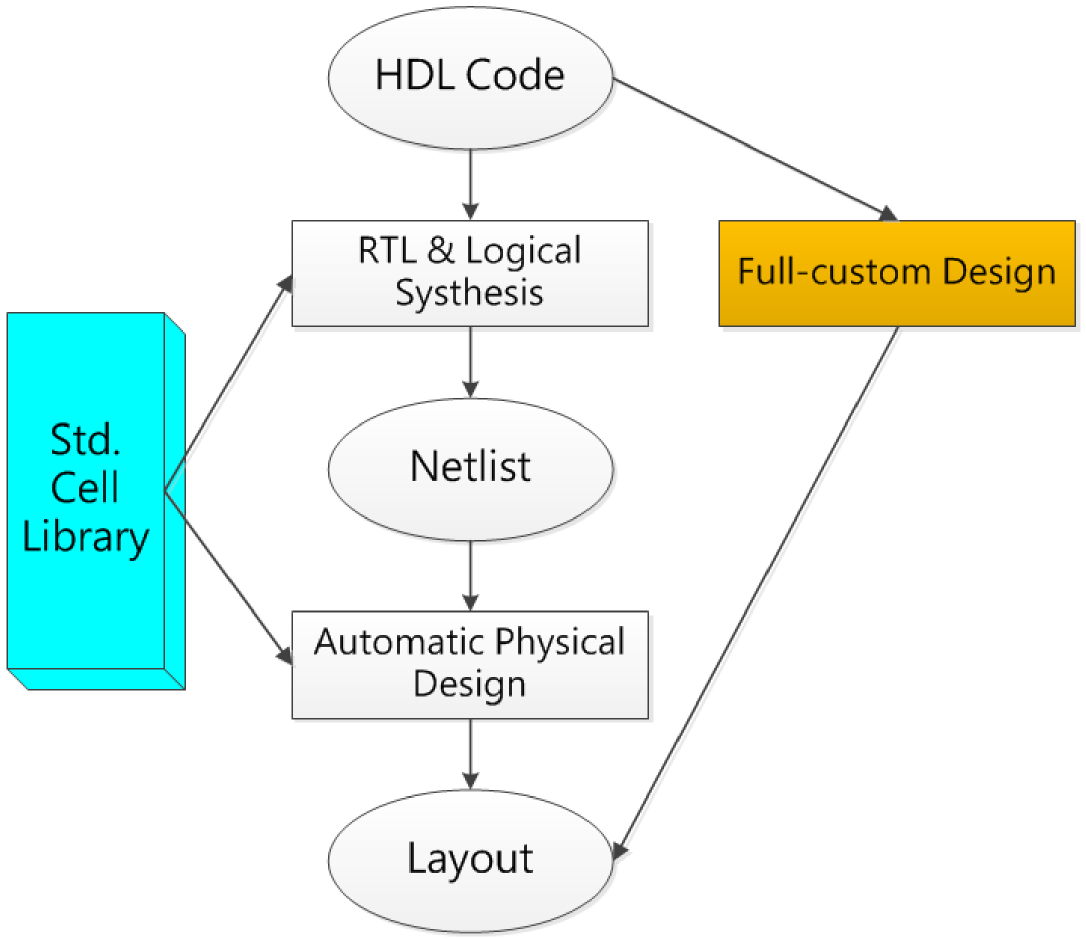
國家實驗研究院

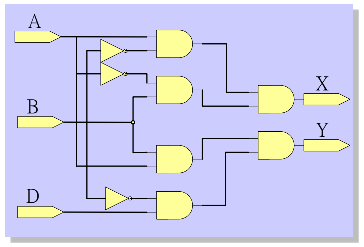
Verilog

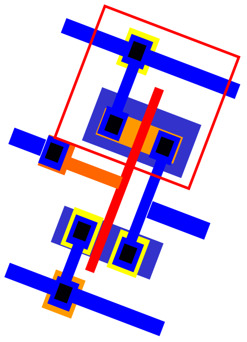
吳建明
Email : wucm@narlabs.org.tw

設計流程 (Design Flow)



```
always @(posedge clk) begin
  if (sel1) begin
    if (sel2)
      out=in1;
    else
      out=in2;
  end
end
```

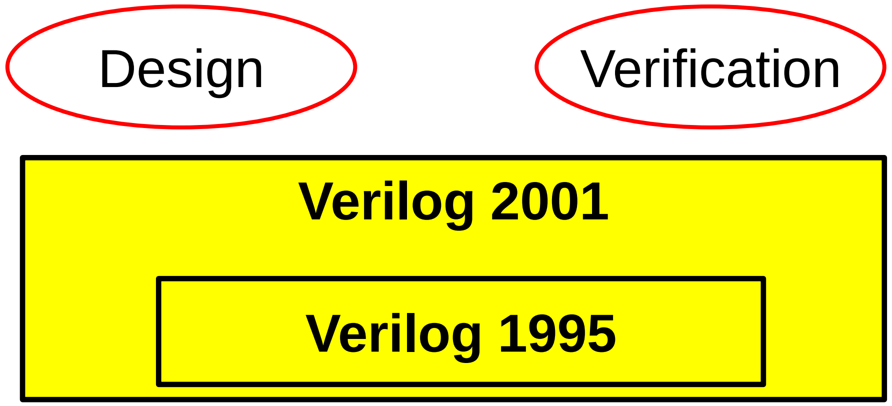




Verilog History (from Wikipedia)

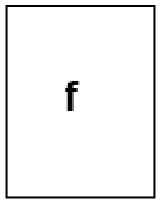
- Verilog HDL是一種硬體描述語言(hardware description language)，為了製作數位電路而用來描述ASICs和FPGA的設計之用。

- 1995年，Verilog被提交到IEEE併成為**IEEE 1364-1995**標準。我們通常稱這一標準為Verilog-95。
- 2001年，提交了一個改善Verilog-95標準缺陷的新的標準。這一擴展版本成為了**IEEE1364-2001**標準，也就是Verilog-2001。



Spec
-English
-algorithmic

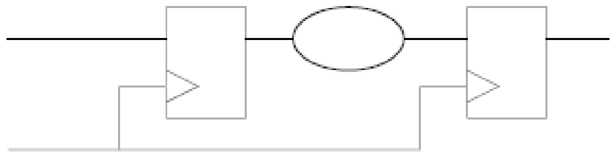
C / matlab / systemC …



Behavioral synthesis

↓

RTL/Functional
-Verilog



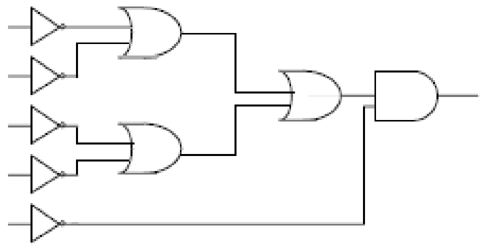
Pre-synthesis simulation

↓

Logic synthesis

↓

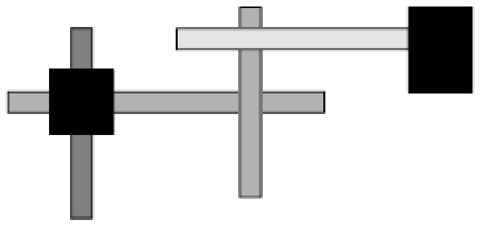
Gates/Structural
-Verilog



Post synthesis simulation

↓

Layout/Physical
-geometric
shapes

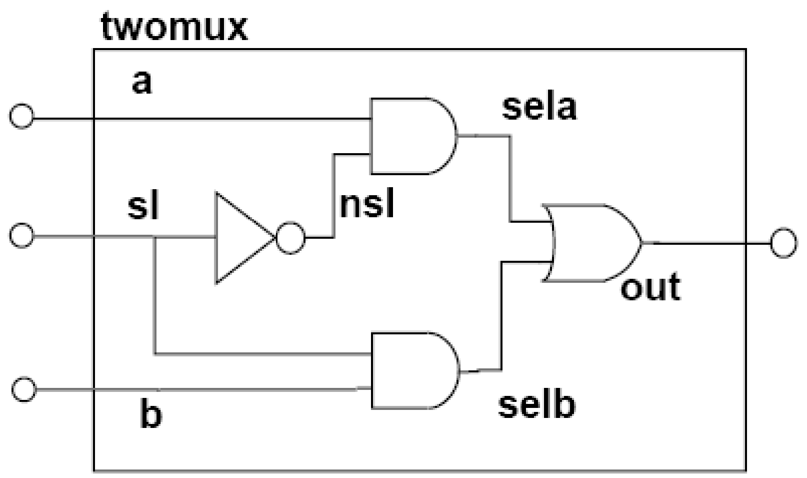


Layout

Gate (Structural)

- The structural level in Verilog is appropriate for small components such as ASIC and FPGA cells.
  - Verilog has built-in primitives, such as the and gate, that describe basic logic functions. (用內建的**邏輯閘(and, or …)**描述電路)
  - You can describe your own User Defined Primitives (UDPs).
  - The resulting netlist from synthesis is often purely structural, but you can also use structural modeling for glue logic.
- The function of the following structural model is represented in Verilog primitives, or gates. It also contains propagation delays:

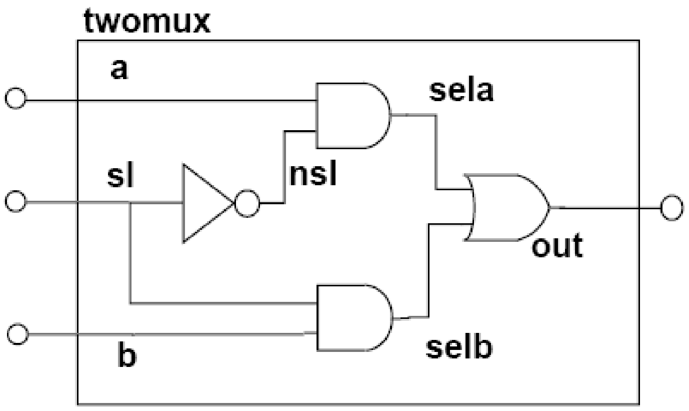
```
module twomux  (out, a,b,sl);

input a,b,sl;
output out;

not   u1 (nsl, sl );
and   #1 u2 (sela, a, nsl);
and   #1 u3 (selb, b, sl);
or    #2 u4 (out, sela, selb);

endmodule
```

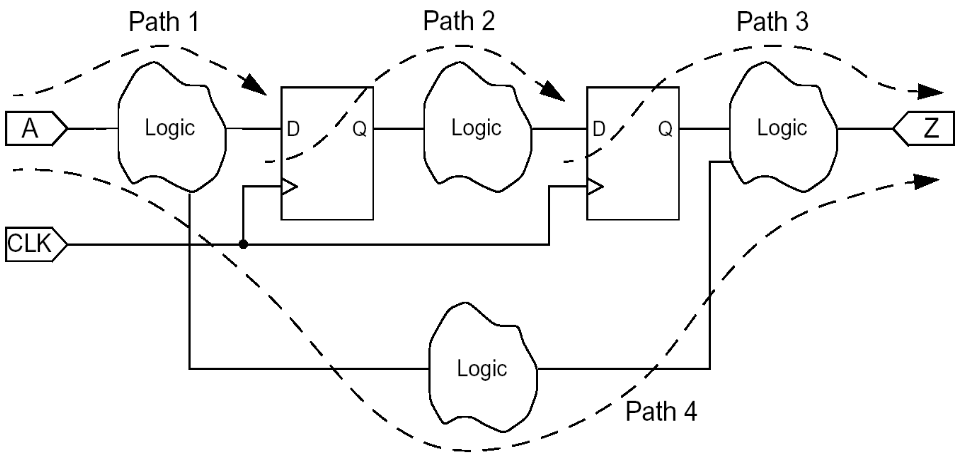


Dataflow

- **Describe the algorithm in terms of logical data flow. (描述電路中暫存器值資料流動方式)**

```
module mux2_1 (out, a, b, sel);
output out;
input a, b, sel;
     assign out = (a&~sel) | (b&sel);
    // or assign out = (sel == 0) ? a : b;
endmodule
```

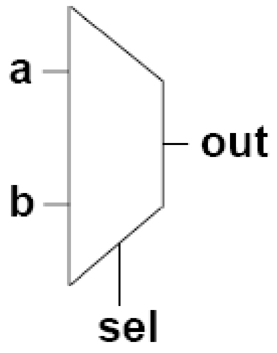




Behavioral

- **Describe the algorithm without concern for the actual logic required to implement it.**
- The behavior of the following MUX is : always, at any change in signals a or b or sl, if sl is 0 pass the value of a to out, else pass the value of b to out.

```
module muxtwo  (out, a, b, sl);
input a,b,sl;
output out; reg out;
always @(sl or a or b)
     if (!sl)
        out = a;
     else
        out = b;
endmodule
```



- In a behavioral model, the function of the logic is modeled using high level language constructs, such as @, while, wait, if and case.
- Test benches, or test fixtures, are typically modeled at the behavioral level. All behavioral constructs are legal for test benches.

DataFlow / Behavioral / Structural

Data Flow Modeling

```
module DF_AND (in1 , in2 , Out)

input in1, in2 ;
output Out ;
wrie  Out ;

assign  Out = in1 & in2 ;
and a1(Out, in1, in2);

endmodule
```

Gate/structural Modeling

Behavioral Modeling

```
module Beh_AND (in1 , in2 , Out)

input  in1 , in2 ;
output Out ;
reg   Out ;

always @ (in1 or in2)
begin
    Out = in1 & in2
end

endmodule
```

Mixed Styles Modeling

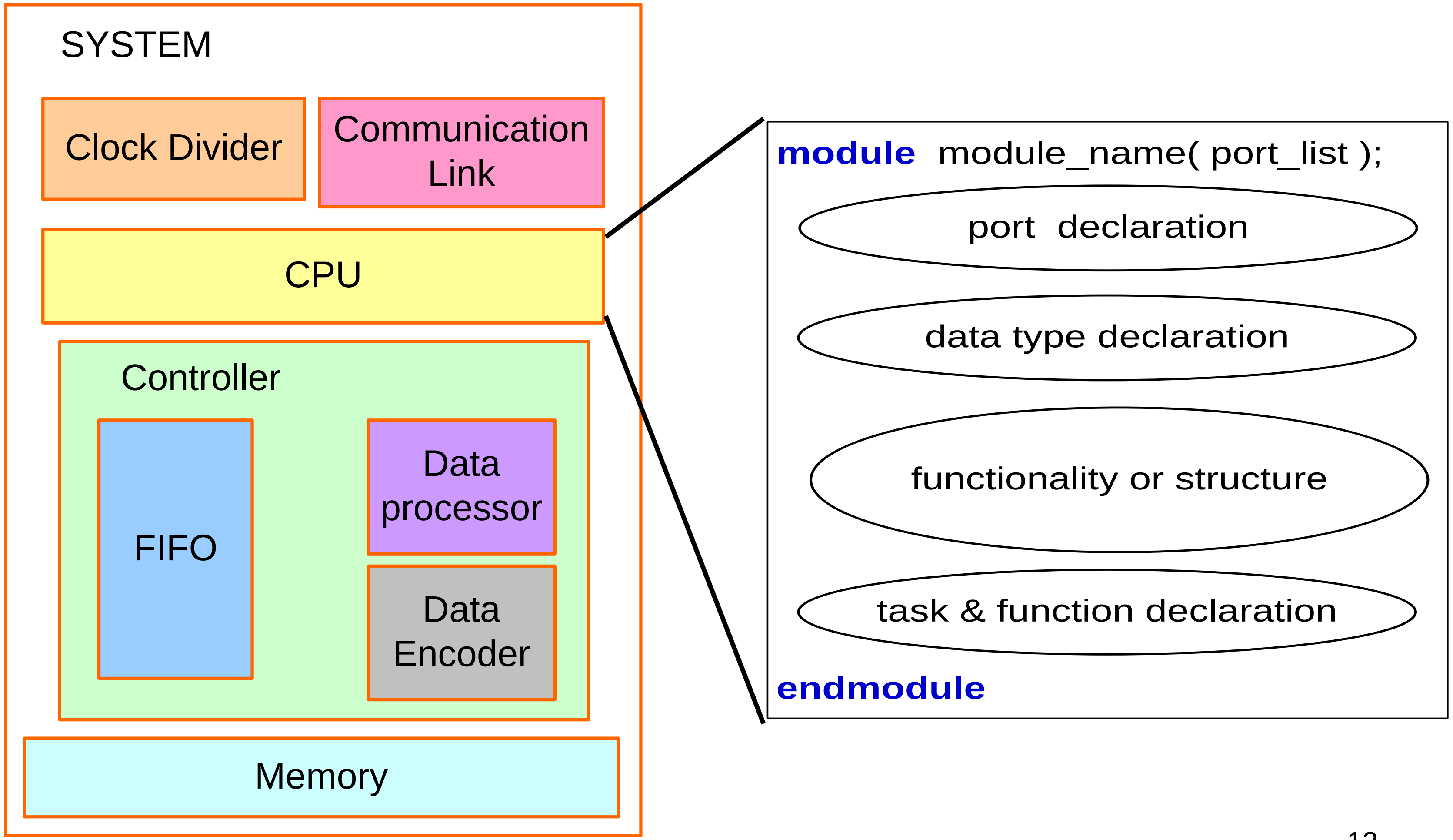
```
module FA_MIX(A, B, CIN, SUM, COUT);
   input A, B, CIN;
   output SUM, COUT;
   reg COUT;
   reg T1, T2, T3;

   wire S1;

   xor X1(S1, A, B); // gate instantiation        Structural

   always @ (A or B or CIN) // Always block
   begin
      T1 = A & CIN;
      T2 = B & CIN;
      T3 = A & B;                                  Behavioral
      COUT = (T1 | T2 | T3);
   end
   assign SUM = S1 ^ CIN; // Continuous assignment   Data Flow
endmodule
```

Lexical Conventions in Verilog

Key Language Features: Verilog Module (1/2)

(1) Verilog Module

◆ 模組(module)是組成一個電路的基本單位

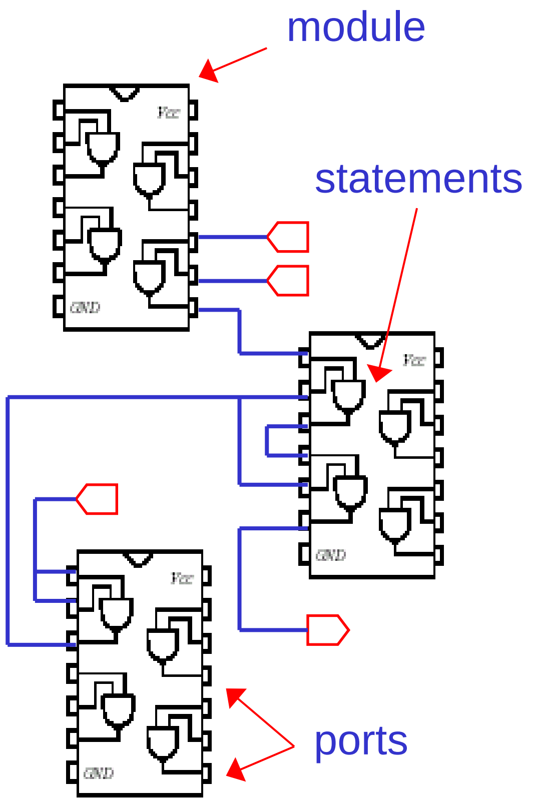


Key Language Features: Verilog Module (2/2)

(1) Verilog Module

- Modules are the basic building blocks in the design hierarchy.
- You place the descriptions of the logic being modeled inside modules.
- Modules can represent:
  - A physical block such as an IC or ASIC cell
  - A logical block such as the ALU portion of a CPU design
  - The complete system
- Every module description starts with the keyword ***module***, has a name (SN74LS74, DFF, ALU, etc.), and ends with the keyword ***endmodule***
- Modules define a new scope (level of hierarchy) in Verilog.
- Example is shown as below :

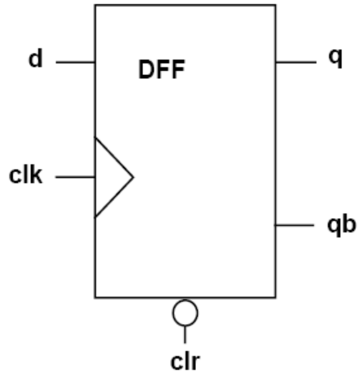
```
module  ALU
  ……         } Logic description(abstraction level)
     ……
endmodule
```



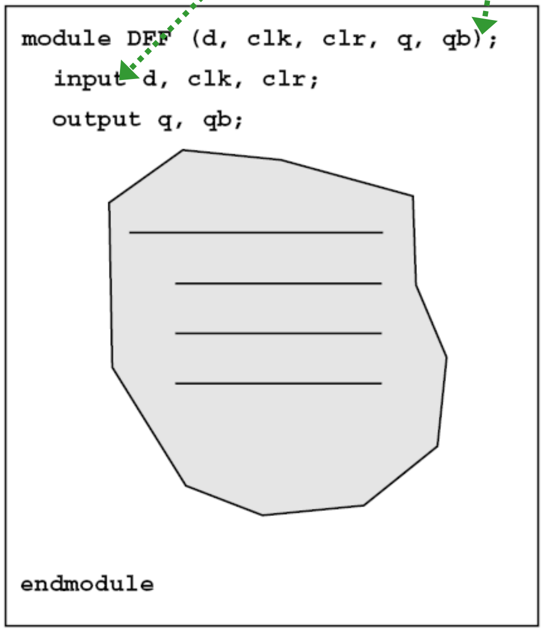
Key Language Features: Module Ports

(2) Module Ports

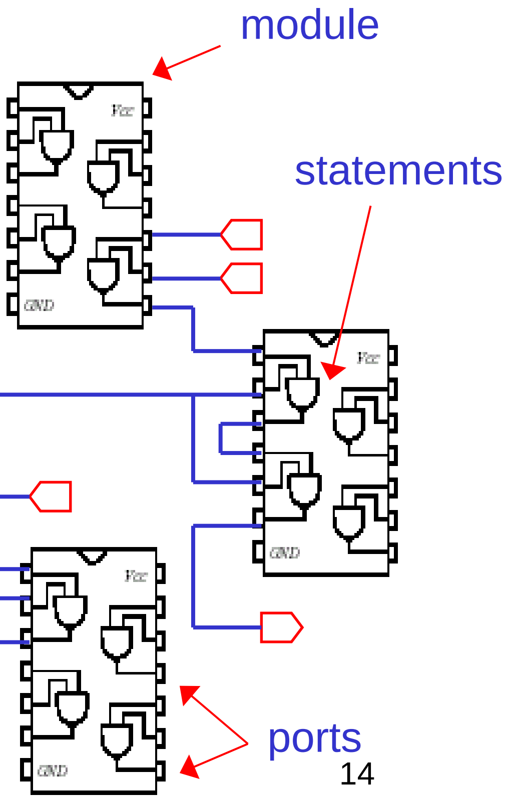
- Modules communicate with the outside world through ports.
- Module ports are equivalent to the pins in hardware.
- You list a module's ports in parentheses "()" after the module name.
- You declare ports to be *input*, *output*, or *inout* (bidirectional) in the module description.
- DFF Example:



input [3:0] c, d;

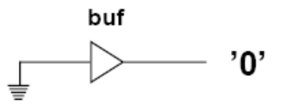
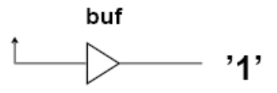
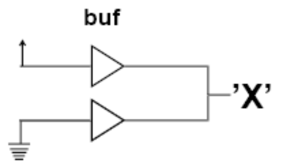
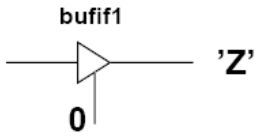


```
module DFF (d, clk, clr, q, qb);
   input d, clk, clr;
   output q, qb;

endmodule
```



4-Value Logic System in Verilog

- The Verilog HDL value set consists of four basic values :

	Zero, Low, False, Logic Low, Ground, VSS, Negative Assertion
	One, High, True, Logic High, Power, VDD, VCC, Positive Assertion
	X, Unknown: Occurs at Logical Conflict Which Cannot be Resolved
	HiZ, High Impedance, Tri-Stated, Disabled Driver (Unknown)

- The "unknown" logic value in Verilog is **not** the same as "don't care". It represents a situation where the value of a node cannot be predicted. In real hardware, this node will most be at either 1 or 0.
- When the Z value is present at the input of a gate, or when it is encountered in an expression, the effect is usually the same as an x value.

Integer Constants

In Verilog, constant values (literals) can be integers or reals.

- ***Integers*** can be sized or unsized. Sized integers are represented as

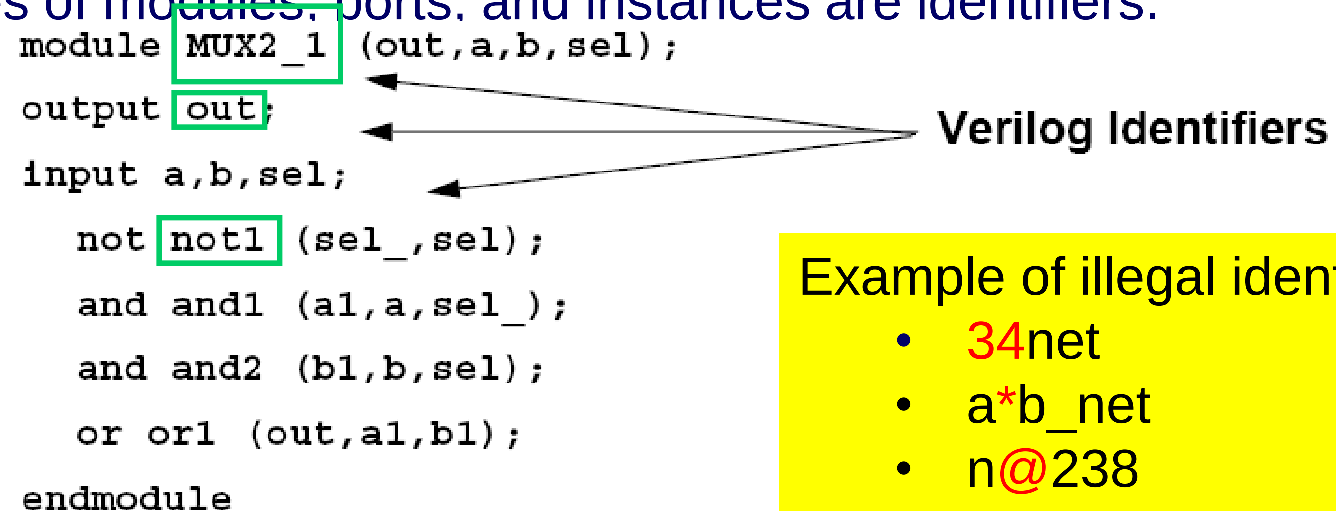
  <size>'<base><value>

  - **<size> is the size in bits.**
  - **<base> can be b(binary), o(octal), d(decimal), or h(hexadecimal).**
  - **<value> is any legal number in the selected base, including x and z bits.**
- Underscores(_) in number are ignored (4’b10_11 = 4’b1011)
  - Underscores(_) can be put anywhere in a constant number, except the beginning, to improve readability.
- Unsized integers default to 32bits (Ex: c=’ha5; This is a 32bits hexadecimal, c=a5)
- The base default to decimal
- The base and value fields are case insensitive (Ex: Decimal (d or D都可以))
- Example:

8’b1001_0011	**is a 8-bits binary number**
3’B01x	**is a 3-bits binary number with the LSB unknown**
659	**is a decimal number**
’h837FF	**is a hexadeciml number**

Identifiers

- An identifiers is used to given an object, such as a register or a module a **name** so that it can be referenced from other places in a description.
- An identifier shall be any sequence of letters (a-z, A-Z), digits (0-9), dollar signs ‘$’, and underscore characters ‘_’.
- The first character of an identifier shall be a letter or an underscore character.
- The first character must **not** be a digit or $
- Identifiers can be up to 1023 characters long.
- Names of modules, ports, and instances are identifiers.



```
module MUX2_1 (out,a,b,sel);
output out;
input a,b,sel;
   not not1 (sel_,sel);
   and and1 (a1,a,sel_);
   and and2 (b1,b,sel);
   or or1 (out,a1,b1);
endmodule
```

Example of illegal identifiers :

- 34net
- a*b_net
- n@238

Special Language Tokens: $

System Tasks and Functions

$<identifier>

- The '$' sign denotes Verilog system tasks and functions.
- A number of system tasks and functions are available to perform different operations, such as:
  - Finding the current simulation time ($time))
  - Displaying/monitoring the values of the signal ($display, $monitor)
  - Stopping the simulation ($stop)
  - Finishing the simulation ($finish)

```
$monitor ($time, "a = %b, b = %h", a, b);
```

Special Language Tokens:

Delay Specification

The pound sign (#) character denotes the **delay** specification for procedural statements, and for gate instances but not module instances.

- The # delay for gate instances is referred to by many names, including gate delay, propagation delay, intrinsic delay, and intra-object delay.

```
module MUX2_1 (out, a, b, sel) ;
output out ;
input   a, b, sel;
not   #1  not1(sel_,sel);
and  #2  and1(a1,a,sel_);
and  #2  and2(b1,b,sel);
or    #1  or1(out,a1,b1);
endmodule
```

Data Types

- Nets
  - Represent physical connection between devices (default = z)
  - Verilog automatically propagates a new value onto a net when the drivers on the net change value. (不斷被驅動 => 改變它的內含值)
  - Value is that of its drivers such as a continuous assignment (data flow modeling)or a gate

```
wire [MSB:LSB] DAT; // vector wire
wire RDY, START;
wire [2:0] ADDR;
```

- Registers
  - Represent a variable that can **hold a value**. (default = x). (reg表示資料儲存，除非給定新的數值，否則數值會一直維持。)
  - They are used in procedural blocks (behavioral modeling).

```
reg A, B, C;        // 1-bit scalars
reg [-3:3] STRANGE;
reg [0:7] QBUS;
```

Types of Nets

- Various net types are available for modeling design-specific and technology-specific functionality.

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

- Net types ***tri*** and ***wire*** are identical in functionality. A reason to declare a net as a **tri** is to indicate that this net can be driven to a high-impedance (z) state.
- Nets that are ***trireg*** act like wires, but store their last value when they are not driven.
- Nets that are not explicitly declared, default to a single-bit wire.

Types of Registers

- The register class consists of four data types.

Register Types	**Functionality**
reg	Unsigned integer variable of varying bit width (i.e. any size [**MSB:LSB**]).
integer	Signed integer variable, 32-bits wide.
Real (不可合成)	Signed floating-point variable, double precision (i.e. 64-bits wide).
Time (不可合成)	Unsigned integer variable, 64-bits wide. (Verilog-XL stores simulation time as a 64-bit positive value.)

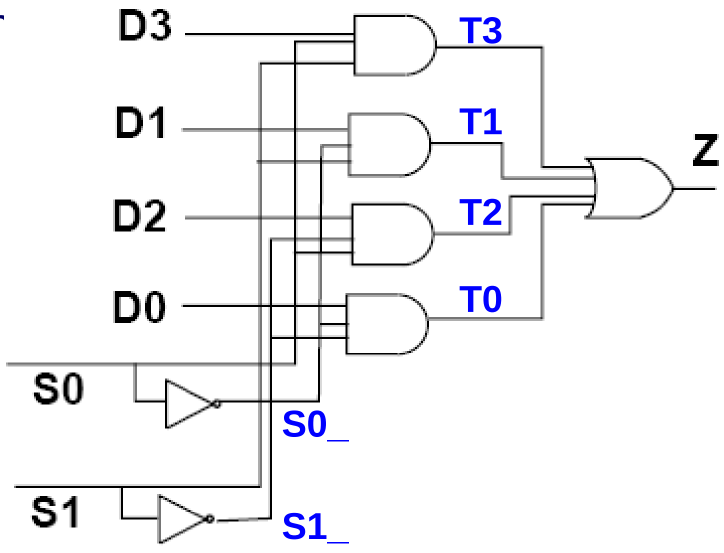
- ***reg*** is the most common register type. You can declare it to be scalar or vector. A ***reg*** is often associated with hardware.
- ***integer*** is used for manipulations of quantities. (not regarded as hardware)
- ***real*** has the same usage as ***integer***, except that you use it for real numbers (e.g. 3.14 ; 3e6 = 3×10^6). (not regarded as hardware)
- ***time*** is often used to store and manipulate integer simulation time quantities. (not regarded as hardware)

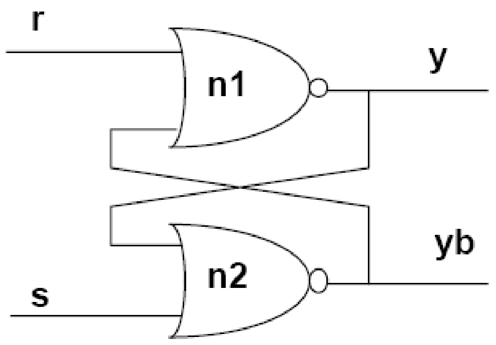
Structural Modeling

Structural Modeling

- A structural model in Verilog represents a schematic.
- A structural model is created using existing components.
- When you use components to create anoth
  these components.

```
module  MUX4x1(Z, D0, D1, D2, D3, S0, S1);
output  Z;
input    D0, D1, D2, D3, S0,S1;
and      (T0, D0, S0_, S1_),
         (T1, D1, S0_, S1),
         (T2, D2, S0, S1_),
         (T3, D3, S0, S1);
not      (S0_,S0),  (S1_,S1);
or       (Z, T0,  T1, T2, T3);
endmodule

module    rs_latch (y, yb, r, s);
output     y, yb;
input       r, s;
nor          n1(y, r, yb);
nor          n2(yb, s, y);
endmodule
```



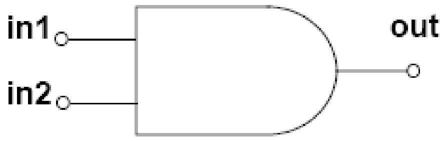


Primitive Pins Are Expandable

- The number of pins for a primitive gate is defined by the number of nets connected to it.
- All gates (except ***not*** and ***buf***) can have a variable number of inputs, but only one output.
- The ***not*** and ***buf*** gates can have a variable number of outputs, but only one input.

Note: The output and bidirectional terminals always come first in the terminal list, followed by the input terminals.

- Examples :
  - and (out, in1, in2);
  - and (out, in1, in2, in3, in4);
  - buf (out1, out2, in);
  - not (out, in);
  - nor (out, in1, in2, in3, in4);

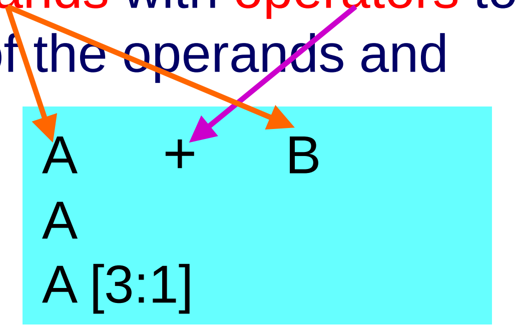


in1
in2
in3
in4
out

Verilog Expressions and Operators

Verilog Expressions

- An expression is a construct that combines operands with operators to produce a result that is a function of the values of the operands and the semantic meaning of the operator



- An operand can be one of the following :
  - Number (including real)
  - net, net bit-select, net part-select.
  - register, integer, time, register bit-select, register part-select.
  - memory element
  - A call to a user-defined function or system-defined function that returns any of the value.
- Example:
  - real a, b, c;
    c = a + b; // a and b are real operands
  - reg_out = reg1[3:0] + reg2[3:0] // reg1[3:0], reg2[3:0] are part-select register operands
  - ret_value = calculate_parity (A, B) // calculate_parity is user-defined function

Operator Types

- The following table shows the operators in Verilog, in order of precedence.
- Note that “and” operators always have higher precedence than “or” operators of the same type.

Type of Operators	Symbols
Concatenate & replicate (連接運算子)	{ } {{ }}
Unary (精簡運算子)	! ~ & | ^
Arithmetic (算數運算子)	** * / % + -
Logical shift (位移運算子)	<<< >>> << >>
Relational (關係運算子)	> < >= <=
Equality (等於運算子)	== === != !==
Binary bit-wise (位元邏輯運算子)	& | ^ ~^
Binary logical (邏輯運算子)	&& ||
Conditional (條件運算子)	? :

Highest ↑

Precedence

Lowest

```
a < b-1 && c != d || e == f              →  worse
( a < b-1 ) && ( c != d ) || ( e == f )  →  better
```

Arithmetic Operators

+	**add**
-	**subtract**
*****	**multiply**
/	**divide**
%	**modulus**
******	**exponent (power) (verilog-2001)**

- An assignment of a negative result to a ***reg*** or other unsigned variable uses the 2’s complement.
- If any bit of any operand is unknown(**x**) or tristate(**z**), the result is simply unknown(**x**).
- In integer division, the remainder is discarded.

```
module arithops ( );
integer ans, int;
parameter five = 5;
reg [3:0] rega, regb;
reg [3:0] num;
initial begin
        rega = 3;
        regb = 4'b1010;
        int = -3;
end
initial fork
        #10 ans = five * int;           // ans = -15
        #20 ans = (int + 5)/2;          // ans =  1
        #30 ans = five/int;             // ans = -1
        #40 num = rega + regb;          // num = 1101
        #50 num = rega + 1;             // num = 0100
        #60 num = int;                  // num = 1101
                                                   (2補數)
        #70 num = regb % rega;          // num = 1
        #80 ans = 2**3                  // ans = 8
        #90 $finish;
join
endmodule
```

Bit-Wise Operators

~	not
&	and
|	or
^	xor
~^	xnor
^~	xnor

- Bit-wise binary operators perform bit-wise manipulations on two operands. They compare each bit in one operand with its corresponding bit in the other operand to calculate each bit for the result.
- Because you can declare the operands to be of different sizes, the **smaller** operand is zero-extended to the size of the larger operand during the operation.

Note: Unknown bits in an operand do not necessarily lead to unknown bits in the result. For example, at time 50 above, the unknown bit in ***regc*** is ORed with a 1 in ***regb***, resulting in a 1.

```
module bitwise ();
reg [3:0] rega, regb, regc;
reg [3:0] num;

initial begin
      rega = 4'b1001;
      regb = 4'b1010;
      regc = 4'b11x0;
end

initial fork
#10 num = rega & 0;       // num = 0000
#20 num = rega & regb;  // num = 1000
#30 num = rega | regb;   // num = 1011
#40 num = regb & regc;  // num = 10x0
#50 num = regb | regc;   // num = 1110
#60 $finish;
join
endmodule
```

Logical Operators (Binary)

!	not
&&	and
||	or

- The result of a logical operation is always1'b0, 1'b1 or 1'bx.
- Logical **binary** operators operate on logic values. If an operand contains all zeroes, it is false (logic 0). If it contains any ones, it is true (logic 1). If it is unknown (contains only zeroes and/or unknown bits), its logical value is ambiguous.
- The logical negation operator reduces an operand to its logical inverse. For example, if an operand contains all zeroes, it is false (logic 0), so its inverse is true (logic 1). (後面投影片有範例)

```
module logical ();
parameter five = 5;
reg ans;
reg [3:0] rega, regb, regc;

Initial  begin
      rega = 4'b0011;
      regb = 4'b10xz;
      regc = 4'b0z0x;
end

initial fork
      #10 ans = rega && 0;       // ans = 0
      #20 ans = rega || 0;       // ans = 1
      #30 ans = rega && five;    // ans = 1
      #40 ans = regb && rega;    // ans = 1
      #50 ans = regc || 0;       // ans = X
      #60 $finish;
join
endmodule
```

Unary Reduction Operators

&	and
|	or
^	xor
~^	xnor
^~	xnor

- Reduction operators perform a bit-wise operation on all the bits of a single operand.
- The result is always 1'b1,1'b0 or 1'bx.

```
module reduction();
reg val;
reg [3:0] rega, regb;
initial begin
     rega = 4'b0100;
     regb = 4'b1111;
end
initial fork
     #10 val = &rega ;          // val = 0
     #20 val = |rega ;          // val = 1
     #30 val = &regb ;          // val = 1
     #40 val = |regb ;          // val = 1
     #50 val = ^rega ;          // val = 1
     #60 val = ^regb ;          // val = 0
     #70 val = ~|rega;     // (nor) val =  0
     #80 val = ~&rega;    // (nand) val = 1
     #90 val = ^rega && &regb; // val = 1
     #99 $finish;
join
endmodule
```

Note

a = 1011
b = 0010

bit-wise	unary reduction	logical
a | b = 1011	| a = 1	a || b = 1
a & b = 0010	& a = 0	a && b = 1

Logical Versus Bit-Wise Negation

!	**logical (not)**
~	**bit-wise (not)**

- The logical negation will return 1'b0, 1'b1, or 1'bx.
- Bit-wise negation returns a value with the same number of bits that are in the operand.

```
module negation();
reg [3:0] rega, regb;
reg [3:0] bit;
reg log;

initial begin
    rega = 4'b1011;
    regb = 4'b0000;
end

initial fork
    #10 bit = ~rega;    // num =  0100
    #20 bit = ~regb;    // num =  1111
    #30 log = !rega;    // num =  0
    #40 log = !regb;    // num =  1
    #50 $finish;
join
endmodule
```

Shift Operators

>> **shift right**

<< **shift left**

>>> **arithmetic shift right** (verilog-2001)

<<< **arithmetic shift left** (verilog-2001)

- a << n
  - a shift left n bits, and fills in zero bits
- b >> n
  - b shift right n bits, and fills in zero bits
- c <<< n
  - c shift left n bits, and fills in zero bits
- d >>> n
  - d shift right n bits, and fills in signed bits

- Shift operators perform left or right bit shifts to the first operand.
- The second operand is treated as unsigned.
- If the second operand has unknown or tristate bits, the result is unknown.

```
module shift ();
reg [5:0] numa, numb, numc, numd, nume;
reg [3:0] rega;
reg signed [3:0] regb;
initial   begin
     rega = 4'b1011;
     regb = 4'b1101;
end
initial begin
     #10 numa = rega << 2 ;  // rega =       1 0 1 1
                             // numa= 1 0 1 1 0 0

     #20 numb = rega >> 3;   // rega =       1 0 1 1
                             // numb= 0 0 0 0 0 1

     #30 numc = regb <<< 1; // regb =       1 1 0 1
                             // numc= 1 1 1 0 1 0

     #40 numd = regb >>> 2; // regb =       1 1 0 1
                             // numd= 1 1 1 1 1 1

     #50 nume = regb >> 2;   // regb =       1 1 0 1
                             // nume= 0 0 1 1 1 1
#300 $finish;
end
endmodule
```

Relational Operators

>	**greater than**
<	**less than**
>=	**greater than or equal**
<=	**less than or equal**

- The result is always 1'b1,1'b0 or 1'bx.
- ***Note***: relational operations have lower precedence than arithmetic operations.

```
a < size -1
a < (size -1)    } same expression

size – (1 < a)
size – 1 < a     } different expression
```

```
module relationals ();
reg [3:0] rega, regb, regc;
reg val;
initial begin
    rega = 4'b0011; // 3
    regb = 4'b1010; // 10
    regc = 4'b0x10;
end
initial fork
    #10 val = regc > rega;   // val = x
    #20 val = regb < rega;   // val = 0
    #30 val = regb >= rega; // val = 1
    #40 val = regb > regc;   // val = 1
    #50 $finish;
join
endmodule
```

Equality Operators (1/3)

==	**logical equality**
!=	**logical inequality**

- The result is always 1'b1,1'b0 or 1'bx

==	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

```
module equalities1();
reg [3:0] rega, regb, regc;
reg val;
initial begin
    rega = 4'b0011; // 3
    regb = 4'b1010; // 10
    regc = 4'b1x10;
end
initial fork
    #10 val = rega == regb ; // val = 0
    #20 val = rega != regc;   // val = 1
    #30 val = regb != regc;   // val = x
    #40 val = regc == regc;  // val = x
    #50 $finish;
join
endmodule
```

Equality Operators (2/3)

===   identity (case equality)
!==   nonidentity (case inequality)

- The result is always 1'b1 or 1'b0

===	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

```
module equalities2();
reg [3:0] rega, regb, regc;
reg val;
initial begin
    rega = 4'b0011; // 3
    regb = 4'b1010; // 10
    regc = 4'b1x10;
end
Initial  fork
    #10 val = rega === regb; // val = 0
    #20 val = rega !== regc;  // val = 1
    #30 val = regb !== regc;  // val = 1
    #40 val = regc === regc; // val = 1
    #50 $finish;
join
endmodule
```

Equality Operators (3/3)

- **=** is the assignment operator. It copies the value of the RHS of the expression to the LHS.
- **==** is the logical equality operator
  - Which the logical equality operator, an X in either of the operand is logicality unknown.
- **===** is the case equality operator
  - With the case equality operator the result can still evaluate to true (1) or false (0) when x or z values are present in the operands.

==	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

===	0	1	x	z
0	1	0	0	0
1	0	1	0	0
x	0	0	1	0
z	0	0	0	1

  - 2’b0x === 2’b1x evaluates to 0, because they are not equal.
  - 2’b1x == 2’b1x evaluates to 1’bx, because they may or may not be equal.

Conditional Operator (1/2)

- The syntax for the conditional operator is :
  - <LHS> = <condition> ? <true_expression> : <false_expression>;
- This can be read as :
  - If condition is TRUE,

    then LHS = true_expression, else LHS = false_expression.
- If the condition is unknown, and the true_expression and false_expression are not equal, the output is unknown.
  - if sel is 0 then out is set equal to a,

    if sel is 1 then out is set equal to b,

    if sel is unknown, a is 0, and b is 0, then out is set equal to 0,

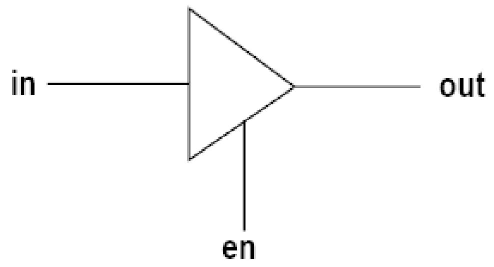
    if sel is unknown, a is 0, and b is 1, then out is unknown.

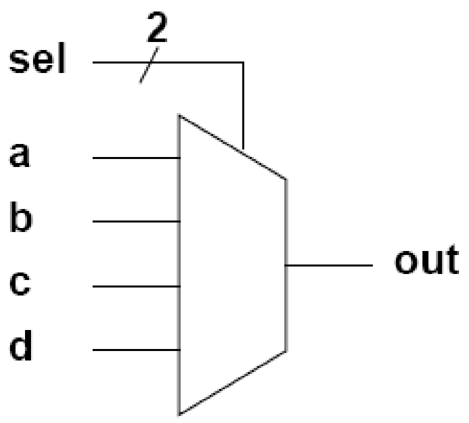
```
assign out = ( sel == 0 ) ? a : b;
```

Conditional Operator (2/2)

? :   conditional

- An unknown value on the condition will result in an unknown value on out.





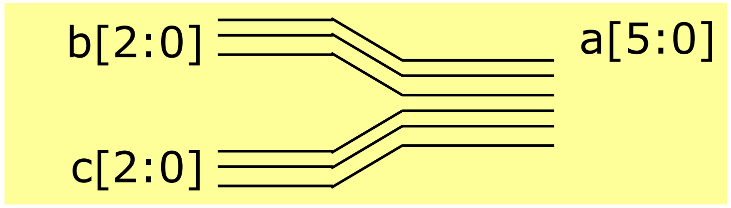
```
module likebufif (in,en,out);
input in;
input en;
output out;
     assign out = (en == 1) ? in : 'bz;
endmodule

module like4to1 (a,b,c,d,sel,out);
input a,b,c,d;
input [1:0] sel;
output out;
assign out = (sel == 2'b00) ? a :
                   (sel == 2'b01) ? b :
                   (sel == 2'b10) ? c : d;
endmodule
```

Concatenation and Replication Operators

- { } **concatenation**

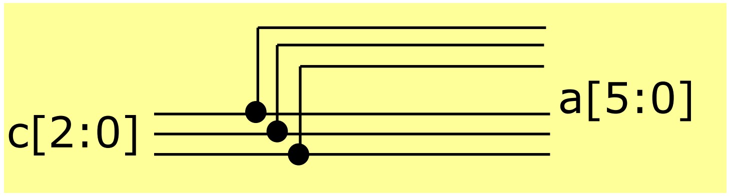
  Ex. a = {b, c};

  MSB LSB



- {{ }} **replication**

  Ex. a = {2{c}};



a[4:0] = {b[3:0], 1'b0};   a = b << 1;

{a, b[3:0], c, 2'b01} = {a, b[3], b[2], b[1], b[0], c, 1'b0, 1'b1}

{a, {3{a, b}}, b} = {a, a, b, a, b, a, b, b}

Concatenation Operator

{ }   concatenation

- Allows you to select bits from different vectors and join them into a new vector.
- Used for bit reorganization and vector construction.
- You must use sized quantities in concatenation and replication (下張投影片介紹). If you do not, an error message will be displayed. Here are some examples that fail to size their operators:
  - a[7:0]={4{'b10}}; //a[7:0]={4{2'b10}};
  - b[7:0]={2{5}};   //b[7:0]={2{4'd5}};
  - c[3:0]={3'b011,'b0};
    //c[3:0]={3'b011,1'b0};
- You can use concatenation on an unlimited number of operands. The operator symbol is { } with operands separated by commas. For example: {A, B, C, D, E}

```
module concatenation;
reg [7:0] rega,regb,regc,regd;
reg [7:0] new;
initial begin
    rega = 8'b00000011;
    regb = 8'b00000100;
    regc = 8'b00011000;
    regd = 8'b11100000;
end
initial fork
    #10 new = {regc[4:3], regd[7:5],
                    regb[2], rega[1:0]};
       // new = 8'b11111111
    #20 $finish;
join
endmodule
```

Replication Operator

{{ }}   replication

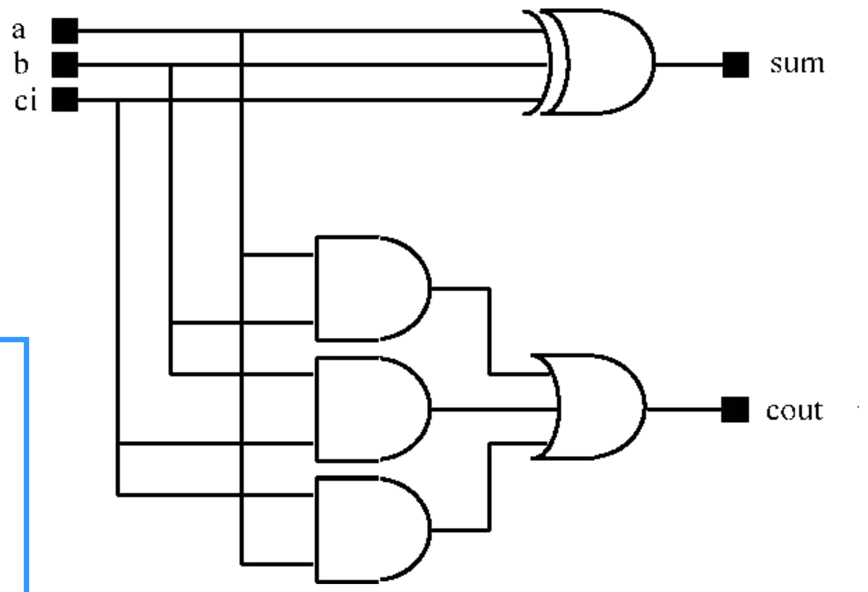
- Replication allows you to reproduce the variable or sized value inside the inner { }.
- Specify a positive integer number of repetitions between the two leading { characters.
- The braces around each replication at time 20 are required for correct syntax.

```
module replicate ();
reg [3:0] rega;
reg [1:0] regb,regc;
reg [7:0] bus;
initial begin
        rega = 4’b1001;
        regb = 2'b11;
        regc = 2'b00;
end
initial fork
        #10bus<={4{regb}};  //bus=11111111
        // regb is replicated 4 times.
        #20 bus <= { {2{regb}},  {2{regc}} };
        //bus=11110000. regc and regb are each
        //replicated, and the resulting vectors
        //are concatenated together
        #30 bus <= { {4{rega[0]}}, rega };
        //bus=11111001.regaissign-extended
        #40 $finish;
join
endmodule
```

Dataflow Modeling

A Full Adder Example

```
module fadder (sum,cout,a,b,ci);
   //port declaration
   output sum, cout;
   input  a, b, ci;

   //netlist declaration
   xor u0 (sum, a, b, ci);
   and u1 (net1, a, b);
   and u2 (net2, b, ci);
   and u3 (net3, ci, a);
   or  u4 (cout, net1, net2, net3);
endmodule
```



assign **{cout, sum} = a + b + ci;**

Continuous Assignments (1/4)

- You can model combinational logic with continuous assignments, instead of using gates and interconnect nets.
- Use continuous assignments **outside** of a procedural block.
- Use a continuous assignment to drive a value onto a **net**.
- The LHS is updated at any change in the RHS expression, after a specified delay.
  - Continuous assignments can only contain simple, left-hand side delay (i.e. limited to a # **delay**), but because of their continuous nature, @ timing control is unnecessary.
- Syntax for an explicit continuous assignment:
  - **<assign> [#delay] [strength] <net_name> = <expression>**
  - **<assign> [#delay] [strength] <net1_name> = <exp1>**

    [#delay] [strength] <net2_name> = <exp2>

    …

    [#delay] [strength] <netn_name> = <expn>

Continuous Assignments (2/4)

- The assignment is **always active** (continuous assignment)
  - Whenever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS.
- You can make continuous assignments explicit or implicit:

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

```
wire  [ 3:0 ]  a = b + c; // implicit
```

- It's not allowed which required a concatenation on the LHS.

```
wire  [ 7:0 ]  {co, sum} = a + b + ci;
assign  {co, sum} = a + b + ci;
```

$\longrightarrow$ ***Error !!***

$\longrightarrow$ ***OK !!***

- <assign> **[#delay] [strength]** <net_name> = <expression>
  - wire [7:0] **(strong1, weak0) #(3,5,2)** o1 = in;   // strength and delays

Continuous Assignments (3/4)

```
module assigns (o1, o2, eq, AND, OR, even, odd, one, SUM, COUT, a, b, in, sel, A, B, CIN);
output [7:0] o1, o2;
output [31:0] SUM;
output eq, AND, OR, even, odd, one, COUT;
input a, b, CIN;
input [1:0]sel;
input [7:0] in;
input [31:0] A, B;
    wire [7:0] #3 o2;                                  // No assignment yet, but a delay
    tri AND = a&b, OR = a|b;                           // two assignments
    wire #10 eq = (a == b);                            // implicit, with delay
    wire [7:0] (strong1, weak0) #(3,5,2) o1 = in;      // strength and delays

    assign o2 [7:4] = in [3:0], o2 [3:0] = in [7:4];   // part-select

    tri #5 even = ^in, odd = ~^in;                     // delay, two assignments

    wire one = 1'b1;                                   // Constant assignment
    assign {COUT, SUM} = A + B + CIN ;                 // Assignment to a concatenation

endmodule
```

Continuous Assignments (4/4)

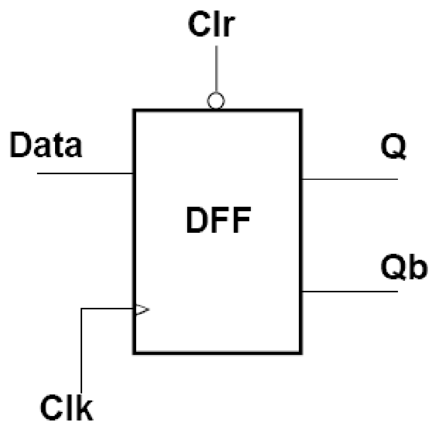
<assign> [#delay] [strength] <net_name> = <expression>

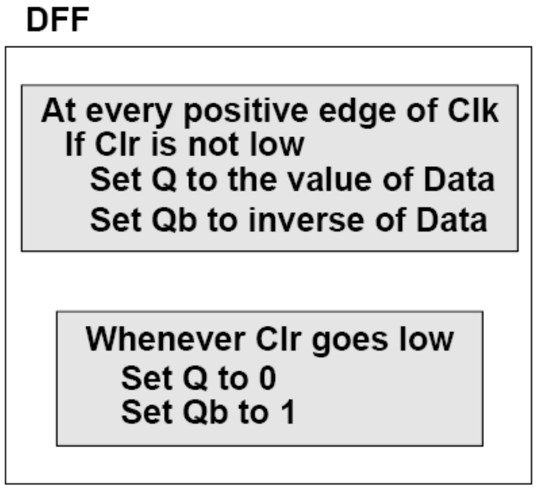
- LHS (left hand side)
  - To any net type
  - To bit- or part-selects of vectored nets // assign o2 [7:4] = 4'hc;
  - To several nets at once in a concatenation // assign {COUT, SUM} = A + B + CIN ;
- RHS (right hand side)
  - From any expression (composed of nets or registers or both), including a constant // assign w = ({a, b} & c) | r ; assign w = 1'b1;
  - With a propagation delay and strengths // wire [7:0] (strong1, weak0) #(3,5,2) o1 = in
  - From the return values of user-defined functions // assign w = f(…)

Behavioral Modeling

Behavioral Modeling

- Behavioral modeling enables you to describe the system at a high level of abstraction.
  - At this level of abstraction, implementation is not as important as the overall functionality of the system. **(ps. 並不是Behavioral Modeling就不能合成)**
- High-level programming language constructs are available in Verilog for behavioral modeling.
  - These include ***wait***, ***while***, ***if then***, ***case***, and ***forever***.
- Behavioral modeling in Verilog is described by specifying a set of **concurrently** active **procedural blocks** that together describe the operation of the system.




DFF

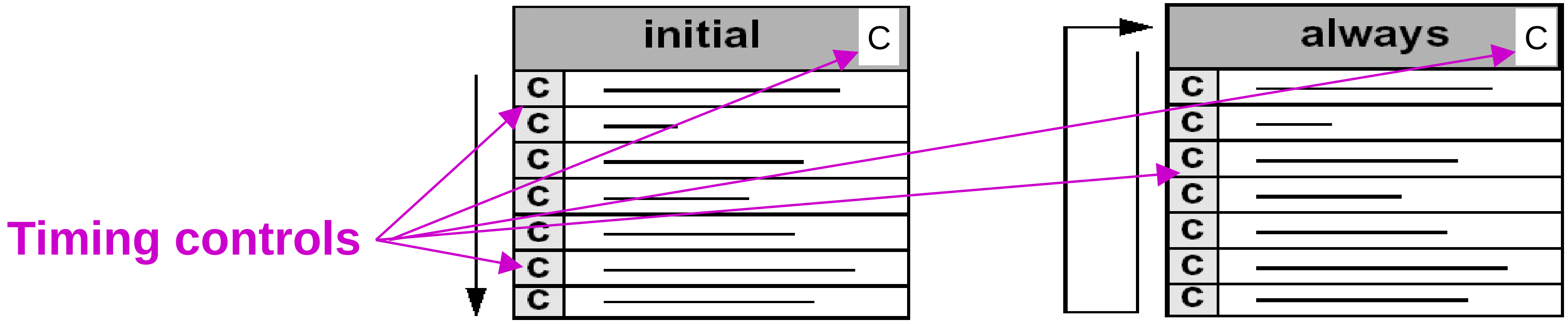
```
At every positive edge of Clk
  If Clr is not low
    Set Q to the value of Data
    Set Qb to inverse of Data
```

```
Whenever Clr goes low
  Set Q to 0
  Set Qb to 1
```

```
always @(posedge clk)
  begin
      if (clr == 0)
        begin
              q <= 1'b0;
              qb <= 1'b1;
        end
      else
        begin
              q <= d;
              qb <= ~d;
        end
end
```

Procedural Blocks

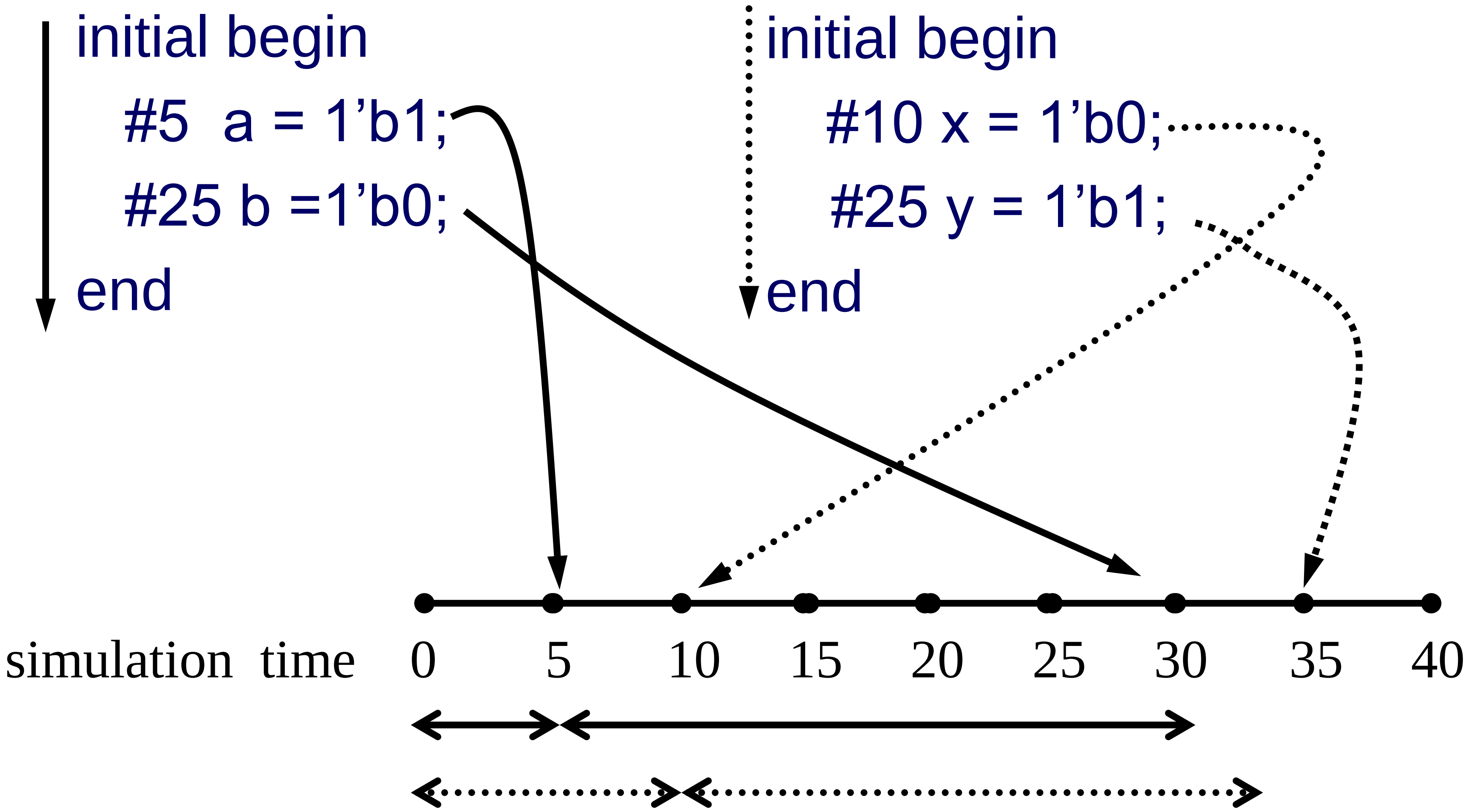
- ***Procedural blocks*** are the basis for behavioral modeling.
- Procedural blocks are of two types:
  - ***initial*** procedural blocks, which **execute only once**. (initialization and waveform generation)
  - ***always*** procedural blocks, which **execute in a loop**.
- The ***initial*** and ***always*** constructs are enabled at the beginning of a simulation.
- Any number of ***initial*** and ***always*** statements may appear within a module
- ***initial*** and ***always*** statements all execute in parallel. (no imply order)



Example of *initial* Statements

```
initial begin
    #5  a = 1'b1;
    #25 b =1'b0;
end
```

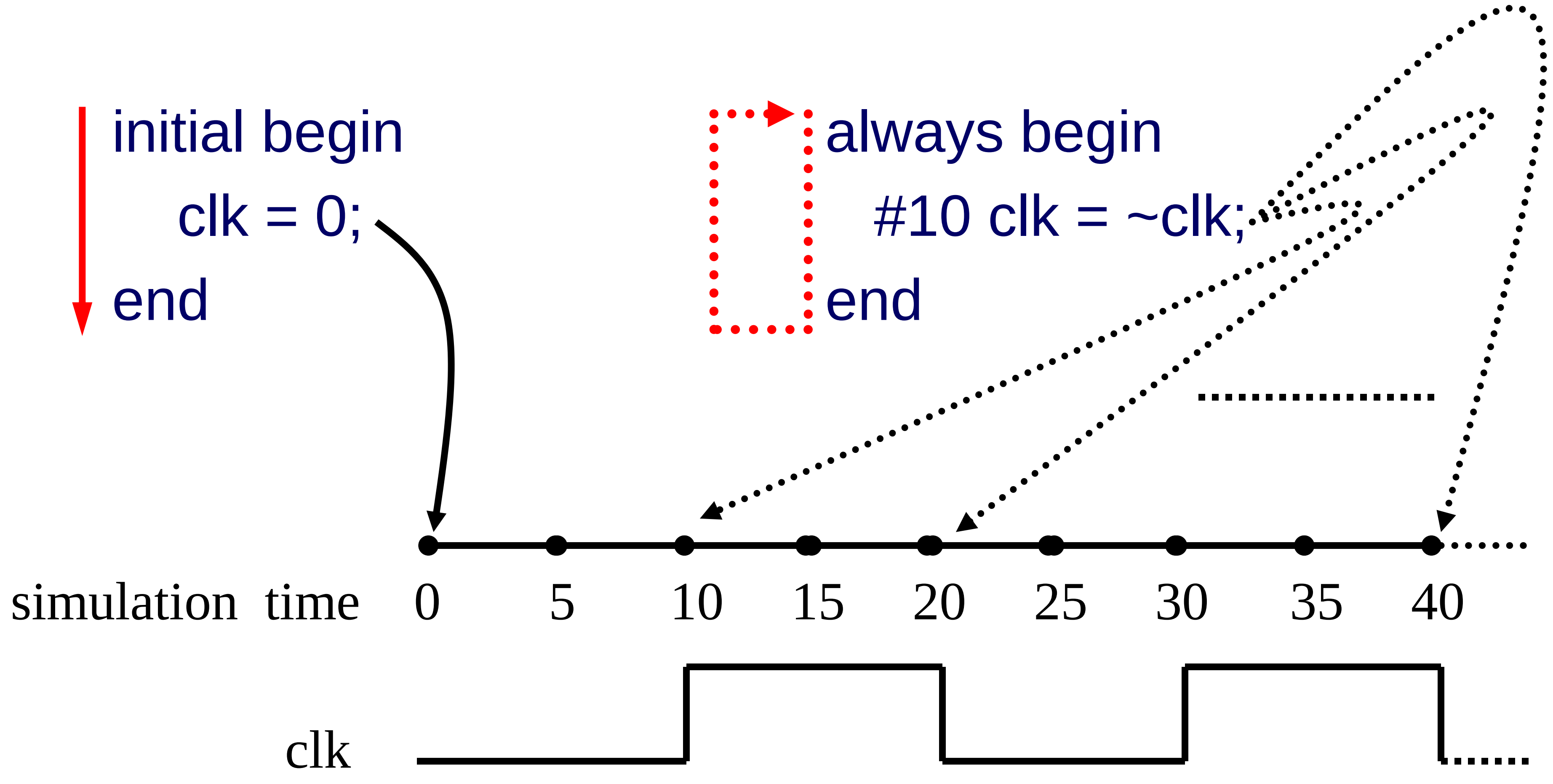
```
initial begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end
```



Example of *always* Statements

```
initial begin
    clk = 0;
end
```

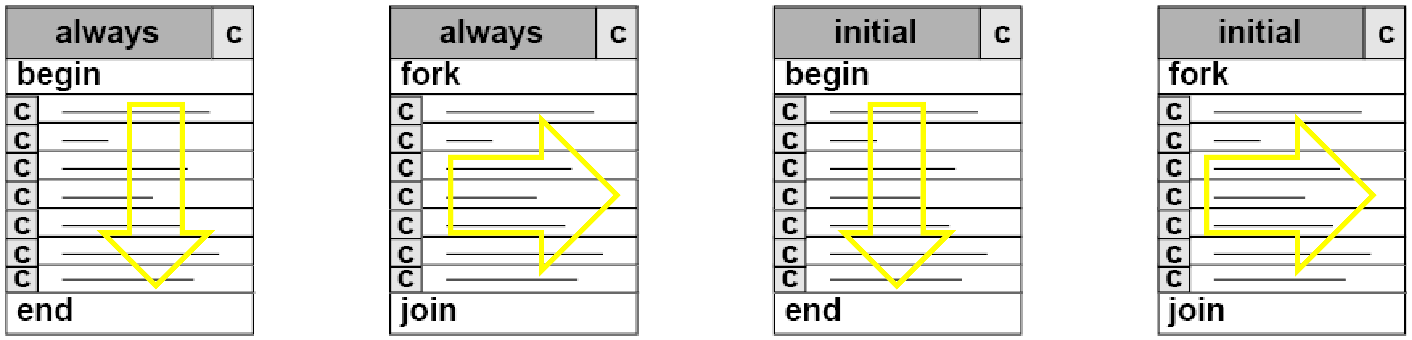
```
always begin
    #10 clk = ~clk;
end
```



Block Statements

Block statements are used to group two or more statements together.

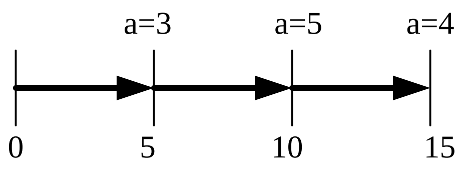
- Sequential block statements are enclosed between the key words ***begin*** and ***end***.
- Parallel block (concurrent block) statements are enclosed between the key words ***fork*** and ***join***.
  - Note that ***fork-join*** blocks are typically not synthesizable and result in inconsistent synthesis result, it is also handled inefficiently by some simulators.



Block Statements: *Examples*

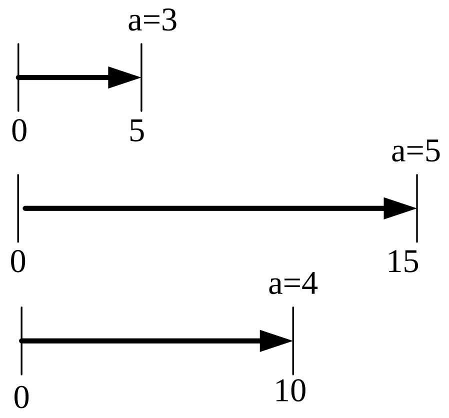
- In a sequential block, statements are evaluated and executed one after the other.

```
begin
    #5 a = 3;    // #5
    #5 a = 5;    // #10
    #5 a = 4;    // #15
end
```



- In a concurrent block, all statements are immediately scheduled to be evaluated and executed after their respective delays.

```
fork
    #5   a = 3;   // #5
    #15 a = 5;    // #15
    #10 a = 4;    // #10
join
```



Components in Procedural Blocks

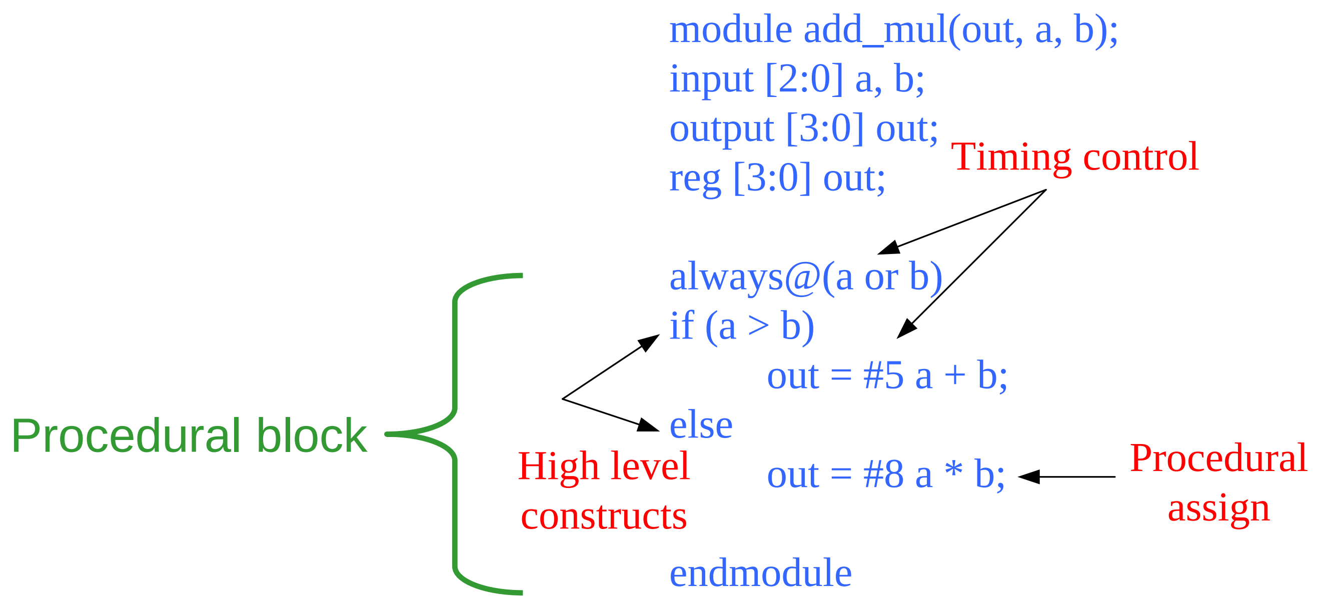
- Procedural blocks have the following components:

描述電路行為

- ➢ Procedural assignment statements to describe the data flow within the block
- ➢ High-level constructs (loops, conditional statements) to describe the functional operation of the block

控制電路何時/何種條件動作

- ➢ Timing controls to control the execution of the block and the statements in the block



```
module add_mul(out, a, b);
input [2:0] a, b;
output [3:0] out;
reg [3:0] out;

always@(a or b)
if (a > b)
        out = #5 a + b;
else
        out = #8 a * b;

endmodule
```

Edge-sensitive Timing Control (Even-Based)

Edge-sensitive (Even-based) timing controls: **@(<signal>)**

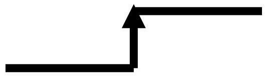
- ➢ **Delays execution until an edge occurs on signal**.
- ➢ Syntax of <event>

  @(<sensitivity>), for a single variable // always @(**a**)

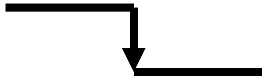
  @(<sensitivity_list>), for several variables // always @(**a or b or c**)

Sensitivity List !!

- ◆ A regular event is that signal <*sensitivity*> has
  - ➢ a change
    - ⅄ E.g. @(a) b = ~a;
  - ➢ a positive edge transition
    - ⅄ E.g. @(posedeg clk) q = d;



  - ➢ a negative edge transition
    - ⅄ E.g. @(negedge clk) q = d;



- ◆ The keywords **posedge** and **negedge** will imply a **FF**

Edge-sensitive Timing Control (Even-Based)

- Use the @ timing control for combinational and sequential models at the RTL and behavioral levels.
- You can qualify signal sensitivity with the ***negedge*** and ***posedge*** keywords, and you can wait for changes on multiple signals by using the **or** keyword.

Sensitivity List !!

```
module reg_adder (out, a, b, clk);
            input clk;
            input [2:0]a,b;
            output [3:0]out;
            reg [3:0] out;
            reg [3:0] sum;
            always @(a or b) // When any change occurs on a or b
                    #5 sum = a + b;
            always @(negedge clk) // at every negative edge of clk
                    out = sum;
            endmodule
```

Combinational block

Sequential block

Note

- Verilog-2001中，可以用逗號來代替or隔開敏感信號

  Verilog-1995:

  ```
  always @(a or b or c or d or sel)
  always @ (posedge clk or posedge reset)
  ```

  Verilog-2001：

  ```
  always @(a, b, c, d, sel)
  always @ (posedge clk, posedge reset)
  ```

- **Verilog-2001**中，使用 * 來包含所有敏感信號

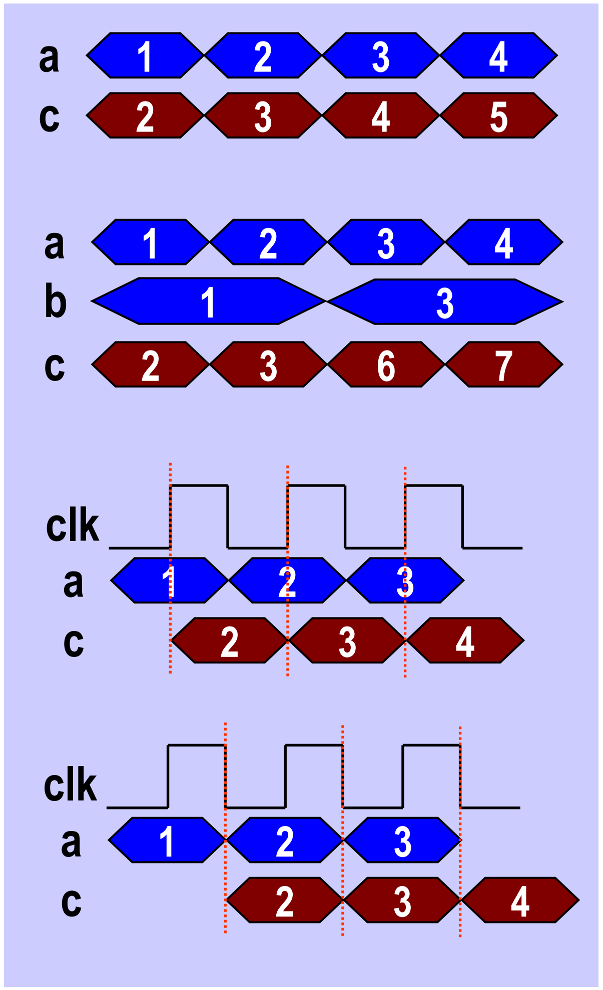
  Verilog-2001:

  ```
  always @*
  ```

Edge-sensitive Timing Control Examples (1/2)

- **Combinational circuit**
  - @ (a): act if signal ‘a’ changes.
    - Ex. always @ (a) c <= a + 1;
  - @ (a or b): act if signal ‘a’ or ‘b’ changes.
    - Ex. always @ (a or b) c <= a + b;
  - The sensitivity list must include all inputs

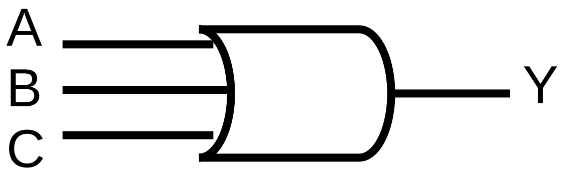
- **Sequential circuit (Register)**
  - @ (posedge clk): act at the rising edge of clk signal.
    - Ex. always @ (posedge clk) c <= a + 1;
  - @ (negedge clk): act at the falling edge of clk signal.
    - Ex. always @ (negedge clk) c <= a + 1;



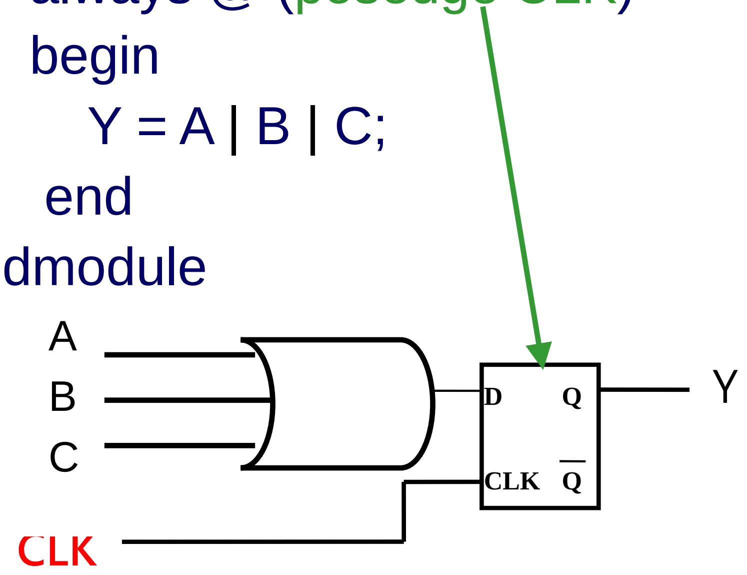
Edge-sensitive Timing Control Examples (2/2)

Register Inference

```
module Comb (A, B, C, Y);
    input A, B, C;
    output Y;
    reg Y;

    always @ (A or B or C)
    begin
        Y = A | B | C;
        end
    endmodule
```



```
module SEQ (CLK, A, B, C, Y);
    input CLK, A, B, C;
    output Y;
    reg Y;

    always @ (posedge CLK)
    begin
        Y = A | B | C;
     end
endmodule
```



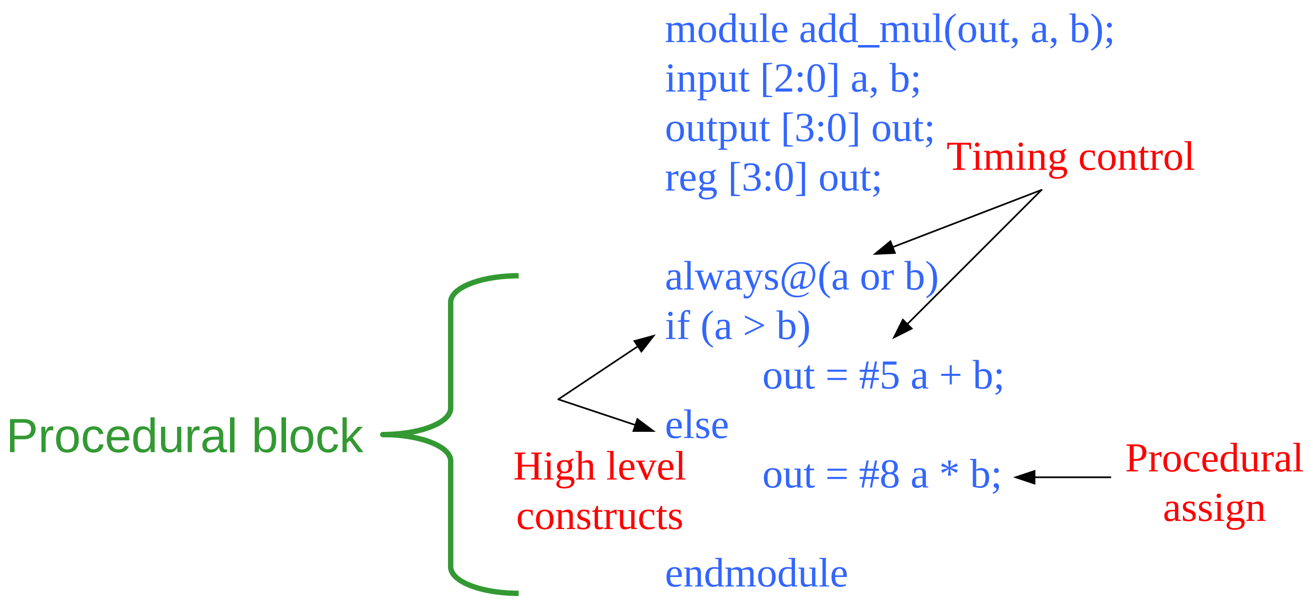
所以，宣告reg，並不一定會合成一個hardware register

Components in Procedural Blocks (remind)

- Procedural blocks have the following components:
  - Procedural assignment statements to describe the data flow within the block
  - High-level constructs (loops, conditional statements) to describe the functional operation of the block
  - Timing controls to control the execution of the block and the statements in the block

描述電路行為

控制電路何時/何種條件件動作



```
module add_mul(out, a, b);
input [2:0] a, b;
output [3:0] out;
reg [3:0] out;

always@(a or b)
if (a > b)
        out = #5 a + b;
else
        out = #8 a * b;

endmodule
```

Procedural Assignments

- Assignments made inside procedural blocks are called **procedural assignments**.
- All signals on the left-hand side **must** be a **register data type** (such as type ***reg***).
- The right-hand side of a procedural assignment can be any valid expression. The data types used here are **not restricted**.
- If you forget to declare a signal, it defaults to type ***wire***. If you make a procedural assignment to a ***wire***, it is an ***ERROR***.

```
module adder (out, a, b, cin);
input a, b, cin;
output [1:0] out;
wire a, b, cin;
reg half_sum;
reg [1:0] out;

always @(a or b or cin)
begin
half_sum = a ^ b ^ cin ; // OK
half_carry = a & b | a & !b & cin | !a & b & cin ;
                // ERROR! (必須宣告為reg)
                // half_carry is not declared,
                // and defaults to a 1-bit wire.
out = {half_carry, half_sum} ;
end
endmodule
```

Procedural Assignments

Continuous Assignment (remind)

Data Flow Modeling

- You can model combinational logic with continuous assignments, instead of using gates and interconnect nets.
- Use continuous assignments **outside** of a procedural block.
- Use a continuous assignment to drive a value onto a **net**.
- The LHS is updated at any change in the RHS expression, after a specified delay.
  - Continuous assignments can only contain simple, left-hand side delay (i.e. limited to a # **delay**), but because of their continuous nature, @ timing control is unnecessary.
- Syntax for an explicit continuous assignment:
  - **<assign> [#delay] [strength] <net_name> = <expression>**

Dataflow vs. Behavior

Any dataflow statements

assign <*variable*> = <*expression*>;

Data type: net

can be re-expressed as

always @(<*sensitivity_list*>) begin

<*variable*> = <*expression*>;

end

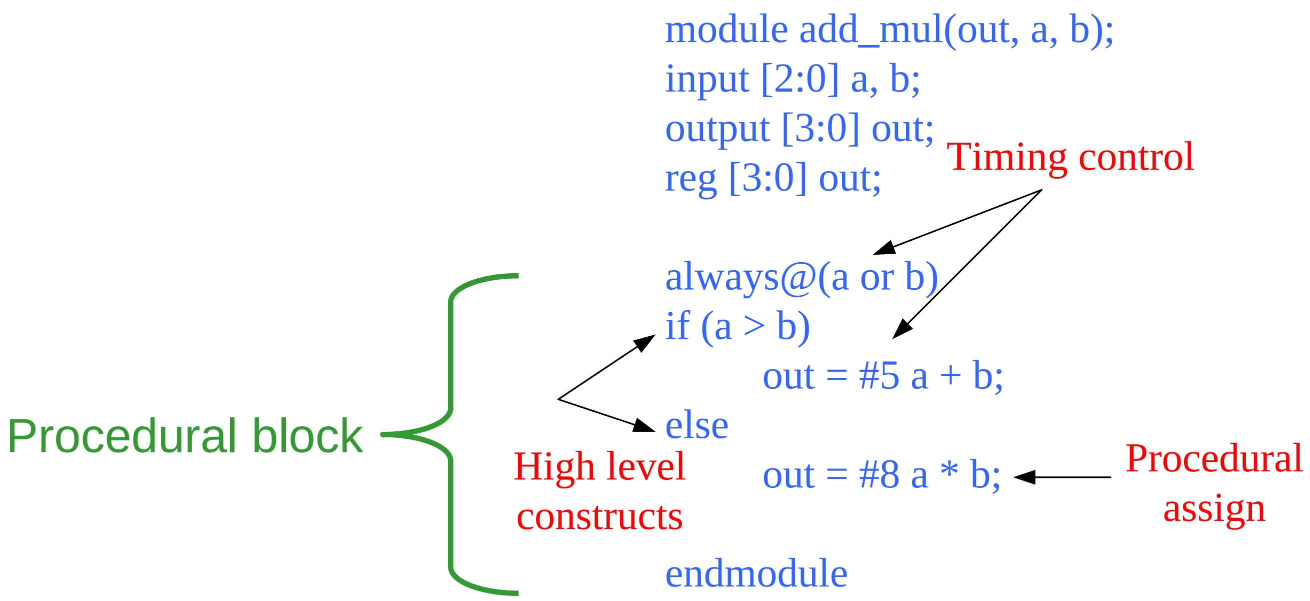
Data type: reg

Components in Procedural Blocks (remind)

- Procedural blocks have the following components:
  - Procedural assignment statements to describe the data flow within the block
  - High-level constructs (loops, conditional statements) to describe the functional operation of the block
  - Timing controls to control the execution of the block and the statements in the block

描述電路行為

控制電路何時/何種條件件動作



```
module add_mul(out, a, b);
input [2:0] a, b;
output [3:0] out;
reg [3:0] out;

always@(a or b)
if (a > b)
        out = #5 a + b;
else
        out = #8 a * b;

endmodule
```

Behavioral Control Statements

- **Conditional Statements**
    - **If; If-else**
    - **case**
- **Looping Statements**
    - **forever loop**
    - **repeat loop**
    - **while loop**
    - **for loop**

Conditional Statements: *if* (1/4)

- If true (1), the true_statement is executed. If **false (0) or ambiguous (x)**, the false_statement is executed
- To ensure proper readability and proper association, use ***begin...end*** block statements.

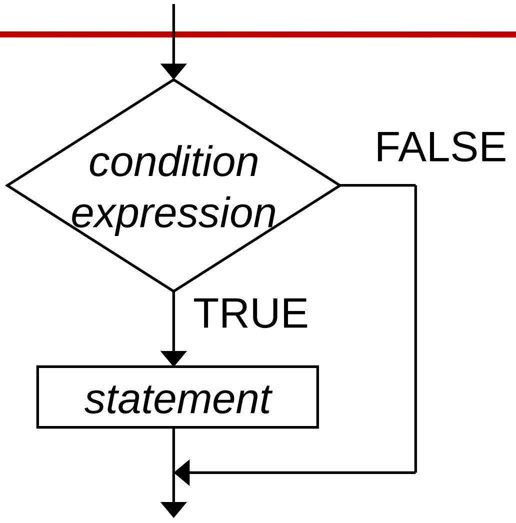
- Types of Conditional Statements
  - **Type 1**: no **else** statement
  - **Type 2**: one **else** statement
  - **Type 3**: nested **if-else-if** statement

Conditional Statements: *if* (2/4)

Type 1: no **else** statement

Syntax:

if <*condition_expression*> <*statement*>

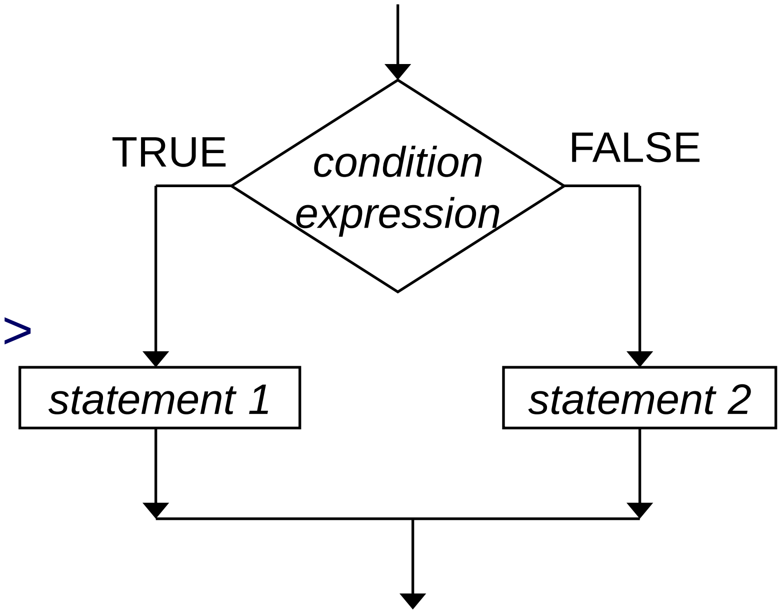


Type 2: one **else** statement

Syntax:

if <*condition_expression*> <*statement 1*>

else <*statement 2*>



Conditional Statements: *if* (3/4)

Example

```
always #20
if (index > 0) // Beginning of outer if
    if (rega > regb) // Beginning of the 1st inner if
            result = rega;
    else
            result = 0; // End of the 1st inner if
else
    if (index == 0)
      begin
            $display("Note : Index is zero");
            result = regb;
      end
    else
            $display("Note : Index is negative");
```

Conditional Statements: *if* (4/4)

Type 3: nested **if-else-if** statement

Syntax:

if *<condition_expression1> <statement1>*

else if *<condition_expression2> <statement2>*

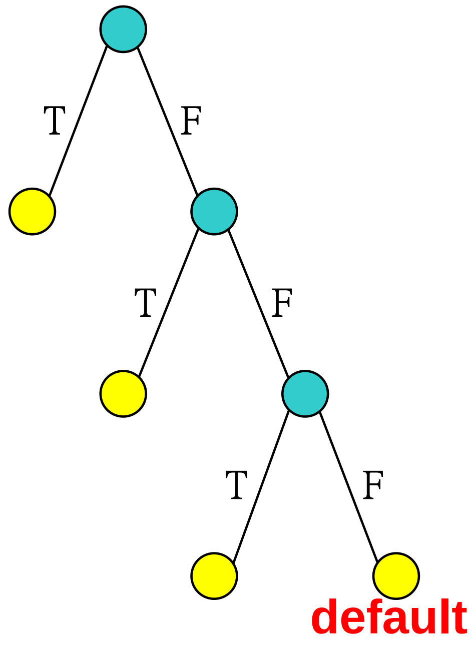
……

else if *<condition_expressionN> <statementN>*

else <***default***_*statement*>

```
always
  if (index < stage1)
      result = a + b;
  else if (index < stage2)
      result = a – b;
  else
      result = a;
```

- The expressions are evaluated in order; if any expression is true, the statement associated with it is executed, and this terminates the whole chain.
- Each statement is either a single statement or a block (***begin...end***) statements.
- The last else part of the if-else-if construct handles the default case where none of the other conditions was satisfied.



Conditional Statements: *case* (1/4)

- The ***case*** statement is a special multiway conditional statement that tests whether the expression matches one of a number of other expressions and branches accordingly.
  - The ***case*** statement does a bit-by-bit comparison for an **exact match** (including x and z)
  - The ***default*** statement is optional. It is executed when none of the statements match the case expression. If it is not specified, Verilog takes no action.
  - Use of **multiple *default*** statements is illegal.
- It is a good programming practice to always use the ***default*** statement, especially to check for x and z (**To avoid synthesis tool to produce Latch device**).

Conditional Statements: *case* (2/4)

Syntax

```
case (<expression>)
        <alternative 1> : <statement 1>
        <alternative 2> : <statement 2>
        ……
        <alternative N> : <statement N>
        default : <default_statement>
endcase
```

Conditional Statements: *case* (3/4)

- **case** is easier to read than a long string of **if...else** statements

```
module mux_2_to_1(a, b, out, sel);
input a, b, sel;
output out;
reg out;

always @ (a or b or sel)
begin
  if (sel) out = a;
  else out = b;
end

endmodule
```

=

```
case (sel)
   1'b1: out = a;
   1'b0: out = b;
endcase
```

Conditional Statements: *case* (4/4)

```
module compute (result, rega, regb, opcode);
    input [7:0] rega, regb;
    input [2:0] opcode;
    output [7:0] result;
    reg [7:0] result;
    always @(rega or regb or opcode)
        case (opcode)
                3'b000 : result = rega + regb;
                3'b001 : result = rega - regb;
                …
                3‘b010 : result = rega * regb;
                3'b100 : result = rega / regb;
                default : begin
                        result = 8’b0;
                        $display ("no match");
                    end
        endcase
endmodule
```

Behavioral Control Statements

- **Conditional Statements**
  - **If; if-else**
  - **case**
- **Looping Statements**
  - **forever loop**
  - **repeat loop**
  - **while loop**
  - **for loop**

Looping Statements

- There are four types of looping statements. They provide a means of controlling the execution of a statement zero, one, or more times.
  - forever continuously executes a statement.
  - repeat executes a statement a fixed number of times.
  - while executes a statement until an expression becomes false. If the expression starts out false, the statement is not executed at all.
  - for controls execution of its associated statement by a three-step process, as follows :
    1. initialize a variable
    2. evaluates an expression
    3. modify the value of the loop-control variable

Looping Statements: *forever*

- The following style of behavioral clock is very flexible (you can control the start time and duty cycle) and simulates very efficiently.
- A **forever** loop executes a statement (or block of statements) until the simulation ends.
- A **forever** loop should be the last item in a procedural **begin/end** block, as any statement that followed it would never be executed.
- **forever** loops are not synthesizable. They are generally implemented in test benches only.

```
...
reg clk;
initial
begin
    clk = 0;
    forever
        begin
                clk = 1;
        end
end
...
```

Looping Statements: *repeat*

- A **repeat** loop executes a block of statements a fixed number of times.
- The value of loop count variable is determined once at the beginning of the execution of the loop. It's not possible to exit loop by changing the loop count variable. (see EX2)
- The **repeat** is not efficient for synthesis, it could be used in testbench modules only.

Ex1:

```
parameter  wordlength = 16;
reg   CRC_valid;
initial   begin
            repeat ( wordlength-1)  begin
            CRC_valid = check ^ datai;
            end
end
```

Ex2:

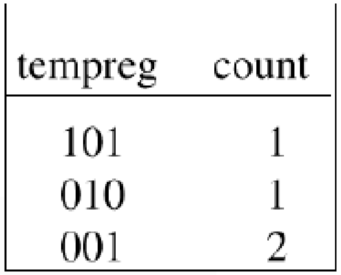
```
initial   begin
            count = 0;
            NUM = 10;
            #30  NUM = 30;
end
initial   begin
            repeat (NUM)  begin
                #10       count = count+1;    // count = 10
            end
end
```

Changing loop count variable can not interrupt the repeat loops

Looping Statements: *while*

- A **while** loop executes a statement (or block of statements) as long as its expression is true (or nonzero).
- If the expression is initially false, the statements are not executed.
- The **while loop** is not efficient for synthesis, it could be used in testbench modules only.

```
reg  [7:0]  tempreg;
reg  [3:0]  count;
. . .
count = 0;
while (tempreg)   // Count the ones in tempreg
          begin
            if  (tempreg[0])  count  =  count + 1;
            tempreg = tempreg >> 1;    // Shift right
          end
. . .
```



tempreg	count
101	1
010	1
001	2

Looping Statements: *for*

- **Syntax: for (<initialization>; <condition>; <operation>)**
  1. The ***initialization*** is performed on the loop index.
  2. The loop executes as long as the ***condition*** evaluates to TRUE.
  3. After each time the loop executes, the ***operation*** is performed.
- A simple comparison to zero often suffices in a **for** loop, and is usually handled much faster. However, this type of comparison may not be accepted by your synthesis tool.

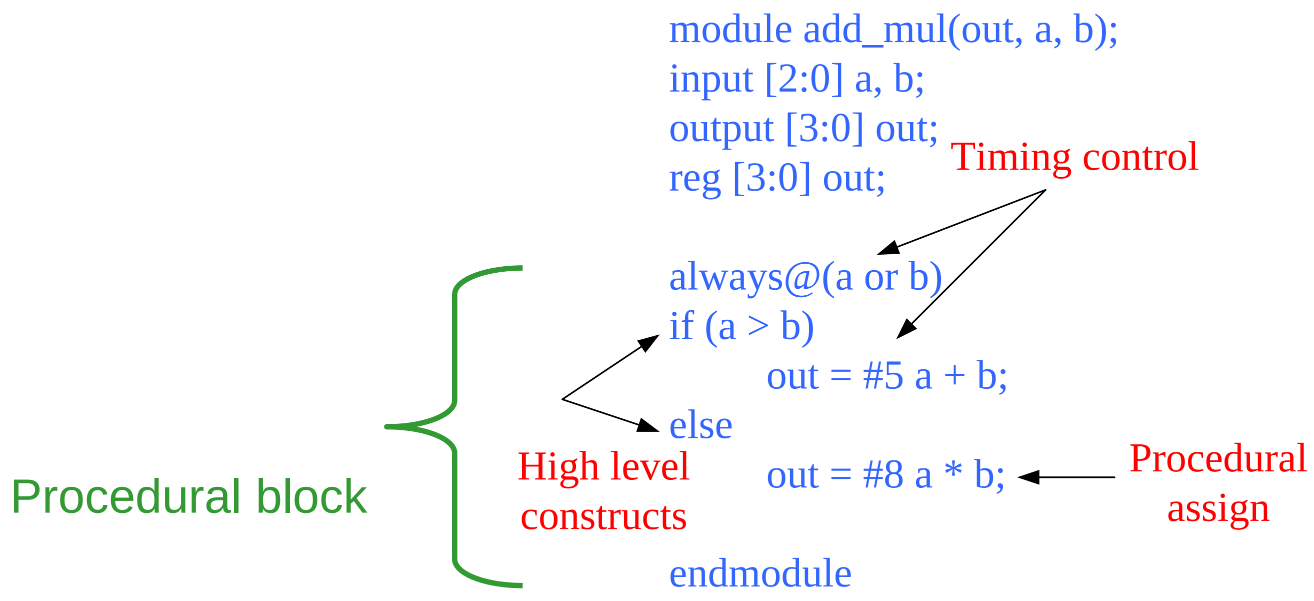
```
// X detection
            for (index = 0; index < size; index = index + 1)
               if (val[index] === 1'bx)
                     $display ("found an X");

// Memory load; "!= 0" is simulated efficiently
            for (i = size; i != 0; i = i - 1)
                memory[i-1] = 0;

// Factorial sequence
            factorial = 1;
                for (j = num; j != 0; j = j - 1)
                      factorial = factorial * j;
```

Components in Procedural Blocks (remind)

- Procedural blocks have the following components:
  - Procedural assignment statements to describe the data flow within the block
  - High-level constructs (loops, conditional statements) to describe the functional operation of the block
  - Timing controls to control the execution of the block and the statements in the block

描述電路行為

控制電路何時/何種條件動作



```
module add_mul(out, a, b);
input [2:0] a, b;
output [3:0] out;
reg [3:0] out;

always@(a or b)
if (a > b)
        out = #5 a + b;
else
        out = #8 a * b;

endmodule
```

Procedural Assignments

- The Verilog HDL contains two types of procedural assignment
  - Blocking procedural assignment //循序式的方式執行程式
  - Non-blocking procedural assignment //平行式的方式執行程式

Blocking :

```
a = 0;
b = 0;
```

Non-blocking :

```
a <= 0;
b <= 0;
```

Blocking vs. Nonblocking

- A blocking procedural assignment is executed (assign) **before** the next statement in the sequential block is scheduled (read).

- A nonblocking assignment does not block the procedural flow, so as soon as the assignment is read by the simulator and scheduled, the next assignment can be read. Then, they are assigned (execute) **concurrently**.
  - When all assignments in a procedural block are nonblocking, the assignments happen in two steps:
    1. The simulator **evaluates all the RHS expressions**, stores the resulting values, and schedules the assignments to take place at the time specified by timing control.
    2. After each **delay has expired**, the simulator **executes the assignment** by assigning the stored values to the LHS expression.

Examples (1/3)

```
module non_block1;
  reg a, b, c, d, e, f;
  initial    begin  //blocking assignments
     a = #10 1; // time 10
     b = #2 0;   // time 12
     c = #4 1;   // time 16
  end
  initial begin   //non-blocking assignments
     d <= #10 1;   // time 10
     e <= #2 0  ;   // time 2
     f <= #4 1   ;   // time 4
  end
  initial begin
     $monitor($time,,"a= %b b= %b c= %b d= %b e= %b f= %b", a, b, c, d, e, f);
     #100 $finish;
  end
endmodule
```

- The simulation result :

```
0    a=x  b=x  c=x  d=x  e=x  f=x
2    a=x  b=x  c=x  d=x  e=0  f=x
4    a=x  b=x  c=x  d=x  e=0  f=1
10   a=1  b=x  c=x  d=1  e=0  f=1
12   a=1  b=0  c=x  d=1  e=0  f=1
16   a=1  b=0  c=1  d=1  e=0  f=1
```

Examples (2/3)

```
module swap_vals;
	reg a, b, clk;
	initial
	  begin
		a = 0;
		b = 1;
		clk = 0;
	  end
	always   #5   clk = ~clk;
	always @(posedge clk)
	  begin
		a = b;   // a = 1
		b = a;   // b = a = 1
	  end
endmodule
```

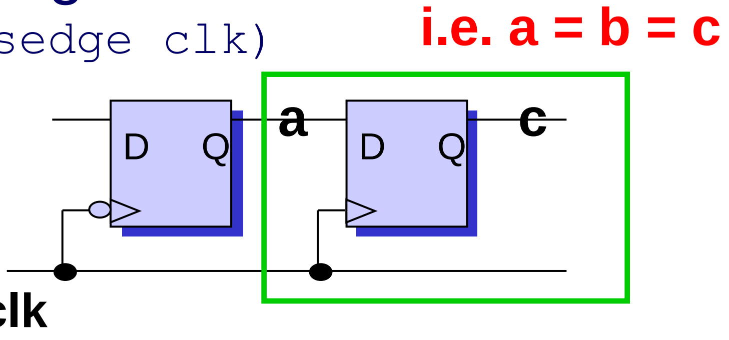
```
always @(posedge clk)
    begin
        a <= b;  // a = 1
        b <= a;  // b = 0
    end
```

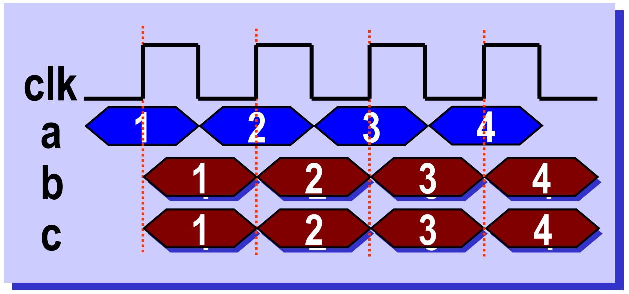
- **swaps** the values of a and b.

Examples (3/3)

- Blocking assignment

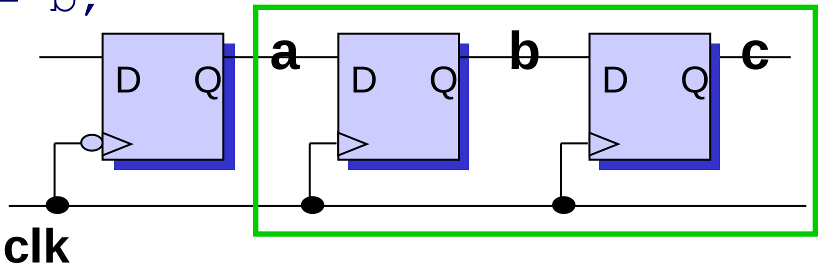
```
always @(posedge clk)
begin
  b = a;
  c = b;
end
```

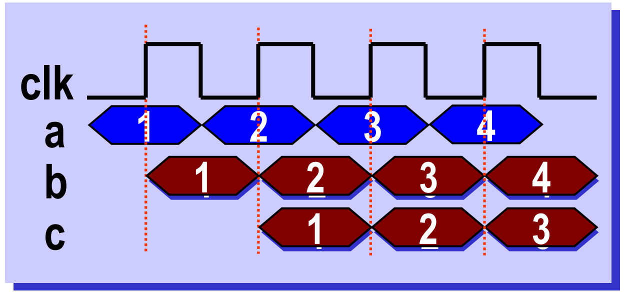




- Nonblocking assignment

```
always @(posedge clk)
begin
  b <= a;
  c <= b;
end
```



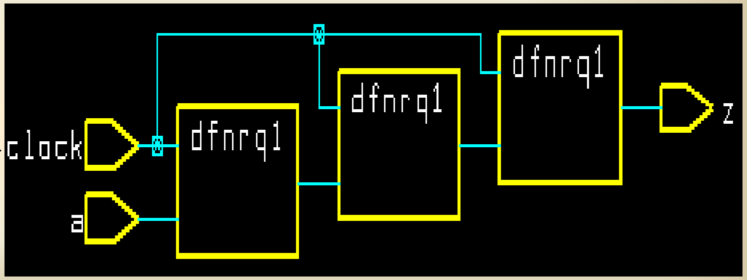


Coding Style for Synthesis (1/2)

- Use *non-blocking* assignments within sequential always block.
- Example:

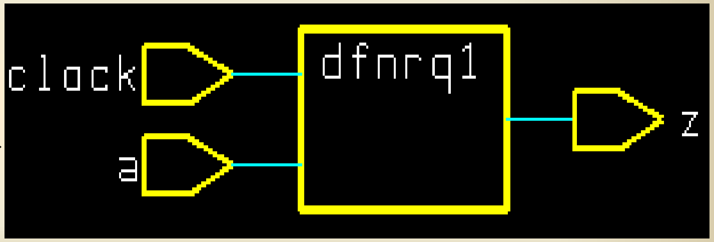
```
always @(posedge clock) begin
    x <= a;
    y <= x;
    z <= y;
end
```

Usually you expect



```
always @(posedge clock) begin
    x = a;
    y = x;
    z = y;
end
```

May not you expect

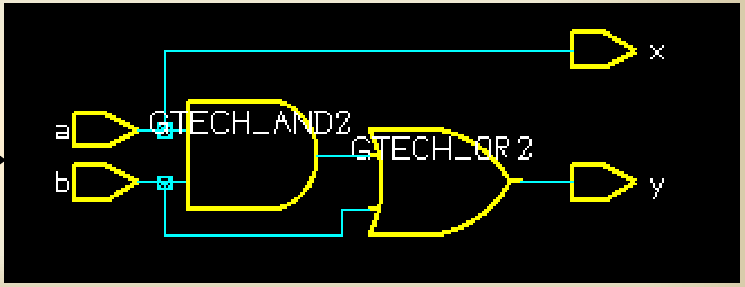


Coding Style for Synthesis (2/2)

- Use *blocking* assignments within combinational always block.
- Example:

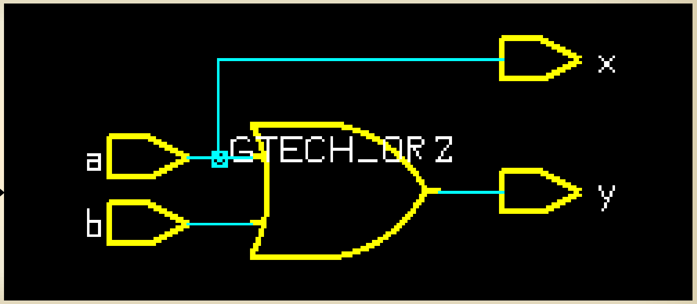
```
always @(a or b or x) begin
    x = a & b;
    y = x | b;
    x = a;
end
```

Usually you expect



```
always @(a or b or x) begin
    x <= a & b;
    y <= x | b;
    x <= a;
end
```

May not you expect



Put Things in the Right Place

```
module adder (…);

always @(…)
begin
    ~~and (a, b, c );~~
    ~~assign sum = a + b;~~
end

endmodule
```

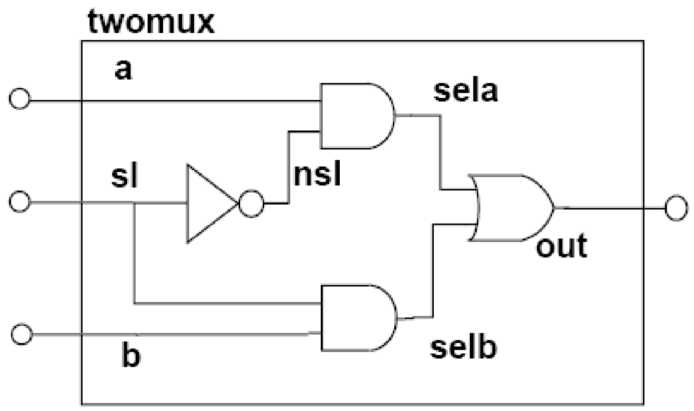
```
module adder (…);

~~sum = a + b ;~~

always @(…)
begin
    ….
end

endmodule
```

```
module adder (…);

~~If (sl==1);~~
    ~~sum = a+b;~~

always @(…)
begin
    ….
end

endmodule
```

Example: 2-1 Multiplexor

- Gate-level modeling

```
module mux2_1 (out, a, b, sel);
output out;
input a, b, sel;
    not (nsel, sel);
    and (sela, a, nsel);
    and (selb, b, sel);
    or (out, sela, selb);
endmodule
```



- Dataflow modeling

```
module mux2_1 (out, a, b, sel);
output out;
input a, b, sel;
   assign out = (a&~sel) | (b&sel);
  // or assign out = (sel == 0) ? a : b;
endmodule
```

- Behavioral modeling

```
module mux2_1 (out, a, b, sel);
output out;
reg out;
input a, b, sel;
   always @ (a or b or sel)
     if (sel)
                out = b;
     else
                out = a;
endmodule
```

Example: 4-1 Multiplexor

- Dataflow modeling

```
module mux4_1 (out, in, sel);
    output out;
    input [3:0] in;
    input [1:0] sel;

    assign out =  (sel == 2’b00) ? In[0] :
                  (sel == 2’b01) ? In[1] :
                  (sel == 2’b10) ? In[2] :
                  (sel == 2’b11) ? In[3] :
                   1’bx;
endmodule
```

同樣的電路，可以有不同的描述方法

- Behavioral modeling

```
module mux4_1 (out, in, sel);
    output out;
    input [3:0] in;
    input [1:0] sel;
    reg out;

    always @ (sel or in) begin
        case (sel)
          2’d0: out = in[0];
          2’d1: out = in[1];
          2’d2: out = in[2];
          2’d3: out = in[3];
          default: out = 1’bx;
        endcase
    end
endmodule
```

Example: 4-to-2 Encoder

- if ... else structure

```
always @(in) begin
   if (in == 4'b0001) y = 0; else
   if (in == 4'b0010) y = 1; else
   if (in == 4'b0100) y = 2; else
   if (in == 4'b1000) y = 3; else
                              y = 2'bx; end
```

- case structure

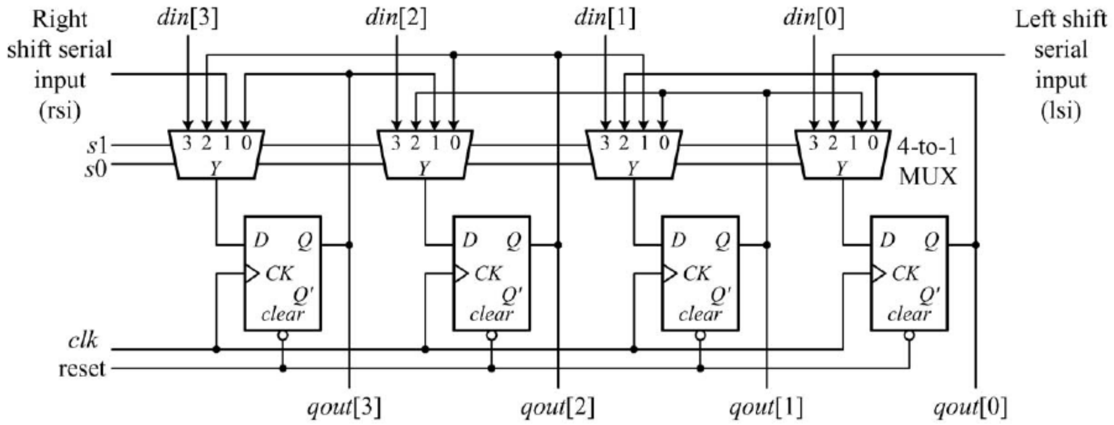
```
always @(in)
   case (in)
      4'b0001 : y = 0;
      4'b0010 : y = 1;
      4'b0100 : y = 2;
      4'b1000 : y = 3;
      default : y = 2'bx;
    endcase
```

I_3	I_2	I_1	I_0	Y_1	Y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

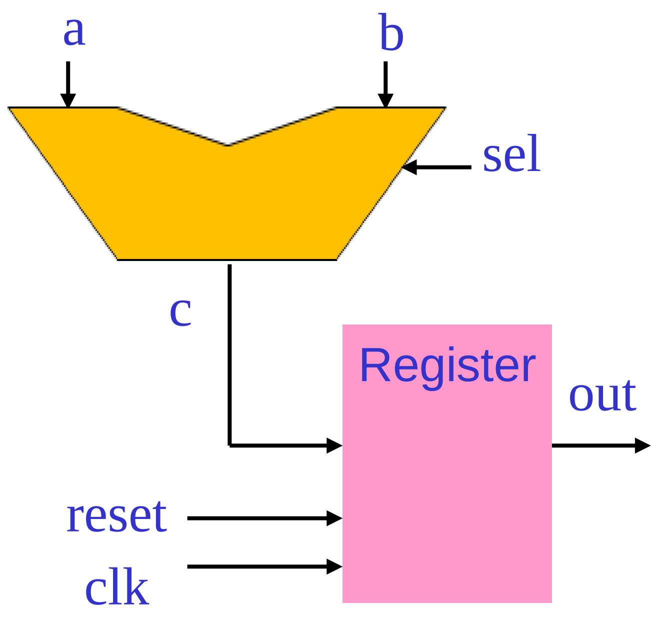
Example: Universal Shift Registers

```
module universal_shift_register (clk, reset_n, s1, s0, …);
parameter N = 4; // define the default size
…
always @(posedge clk or negedge reset_n)
    if (!reset_n) qout <= {N{1'b0}};
    else case ({s1,s0})
            2'b00: ; // qout <= qout;            // No change
            2'b01: qout <= {lsi, qout[N-1:1]}; // Shift right
            2'b10: qout <= {qout[N-2:0], rsi}; // Shift left
            2'b11: qout <= din;                // Parallel load
    endcase
```

$s1$	$s0$	Function
0	0	No change
0	1	Right shift
1	0	Left shift
1	1	Load data



Example



```
module MUX2_1(out,a,b,sel,clk,reset);
input       sel,clk,reset;
input       [7:0]      a,b;
output      [7:0]      out;
wire        [7:0]      c;
reg         [7:0]      out;

//Continuous assignment
assign  c = (sel==0)?a:b;

//Procedural assignment
always @(posedge clk or posedge reset)
begin
            if(reset==1) out <=  0;
            else out <=  c;
end
endmodule
```

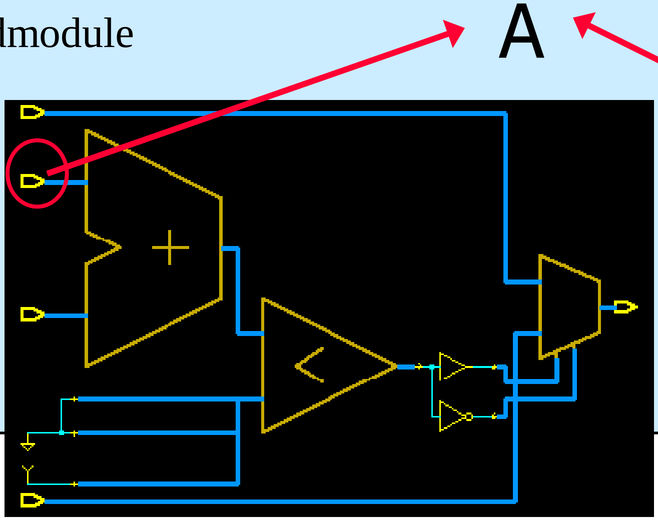
Example: Operator in if

- We assume that signal "A" is latest arrival signal

Before_Improved

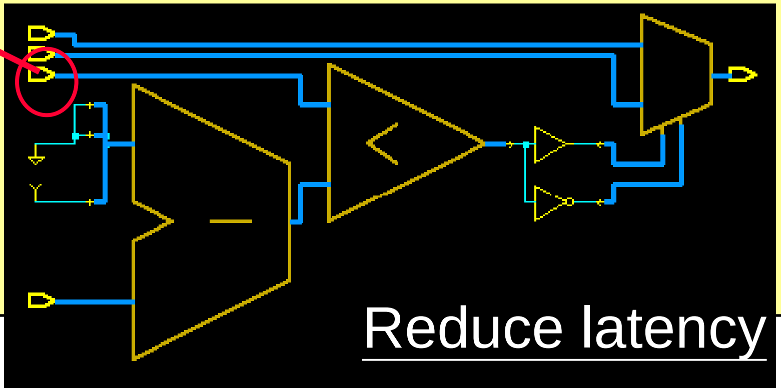
```
always @(A or B or C or D) begin
if (A + B < 24) // A is late arriving
  Z <= C;
else
  Z <= D;
end
endmodule
```

A



After_Improved

```
always @(A or B or C or D) begin
if (A < 24 - B)   // A is late arriving
  Z <= C;
else
  Z <= D;
end
endmodule
```

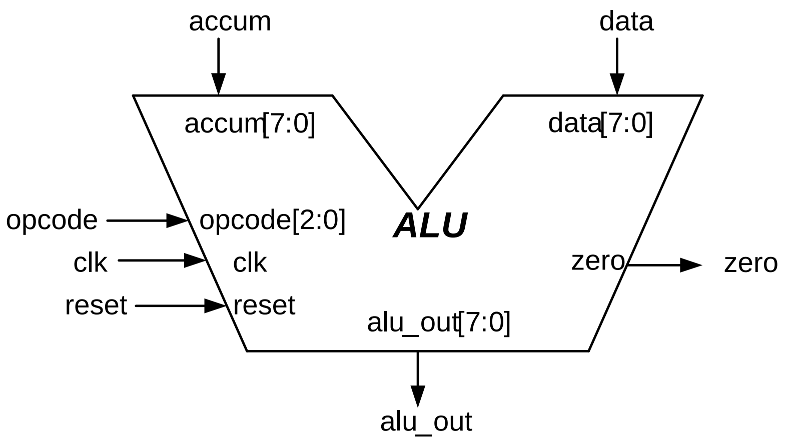


References

- Cadencd, “Verilog-XL Reference”, Product version 3.1, August 2000.
- Synopsys, “HDL Compiler for Verilog: Reference Manual”, Version 1999.05, May 1999.
- Samir Palnitkar, “Verilog HDL: A Guide to Digital Design and Synthesis”, Prentice Hall.
- Michael D.Ciletti, “Advanced Digital Design with the VERILOG HDL”, Prentice Hall.
- 教育部「超大型積體電路與系統設計」教育改進計畫課程資料
- 黃俊銘, “Cell-Based IC Design Concepts”, CIC訓練課程
- 王旭昇, “Logic Synthesis with Design Compiler”, CIC訓練課程
- Verilog教學網站

Lab: ALU

- 所有輸入及輸出（**除了「zero」訊號以外**）均需同步於clock的正緣（rising edge）。
- 同步reset架構。當reset為1時表示reset啟動，此時alu_out訊號輸出為0。
- accum、data及alu_out訊號的數值使用2補數表示。
- 當accum輸入為0時，zero訊號輸出為1；反之當accum輸入不為0時，zero訊號輸出為0。並且zero訊號不需理會reset訊號的動作。
- **當opcode輸入為X(unknow)時，其alu_out訊號輸出為0。**



opcode	ALU operation	
000	Pass accum	
001	accum + data	(add)
010	accum – data	(subtraction)
011	accum AND data	(bit-wise AND)
100	accum XOR data	(bit-wise XOR)
101	ABS(accum)	(absolute value)
110	NEG(accum)	(negate value)
111	Pass data	

1. absolute value時，使用accum[7]當作signed bit
2. negate value時，accum做2補數(反向+1)