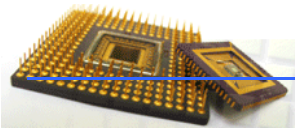
Verilog 簡介 (下)

陳泓烈(CIC)
Hotline : (03)5773693 ext.885
Hot mail : hotline@cic.narl.org.tw

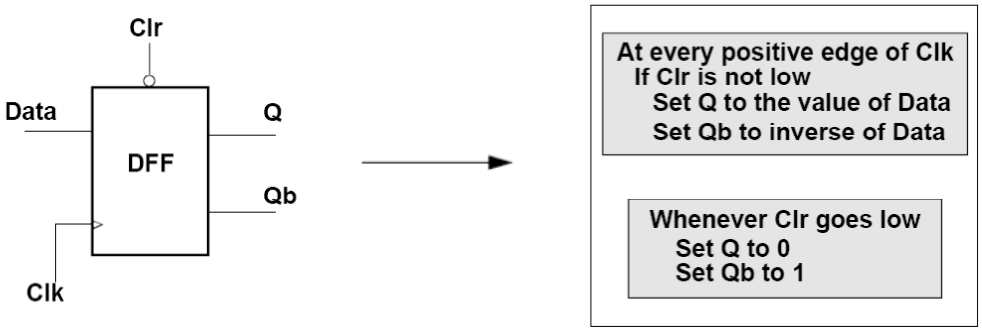


Outline

- ✓ Behavioral Modeling
  - ➢ Structured procedures
  - ➢ Block Statements
  - ➢ Procedural Timing Control
  - ➢ Procedural Assignments
  - ➢ Behavioral Control
- ✓ Continuous Assignment
- ✓ Synthesizable Verilog Code
- ✓ FSM introduction

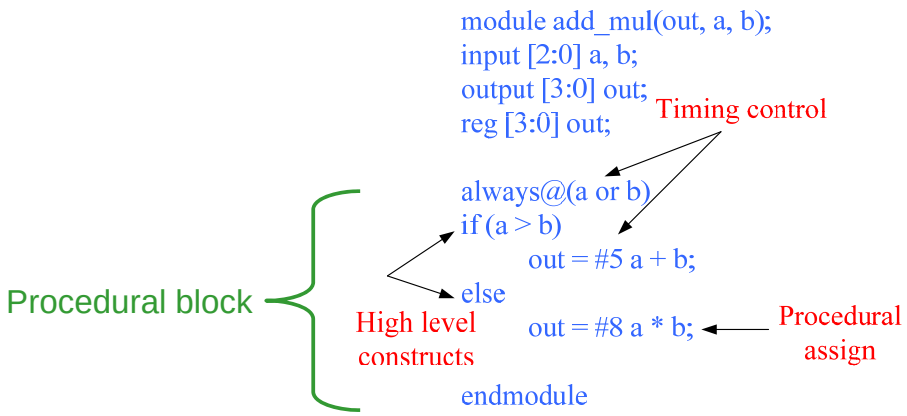
Behavioral Modeling

- ✓ 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.
- ✓ High-level programming language constructs are available in Verilog for behavioral modeling.
  - ➢ These include ***wait***, ***while***, ***if else***, ***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.



Behavioral Modeling - Structured procedures

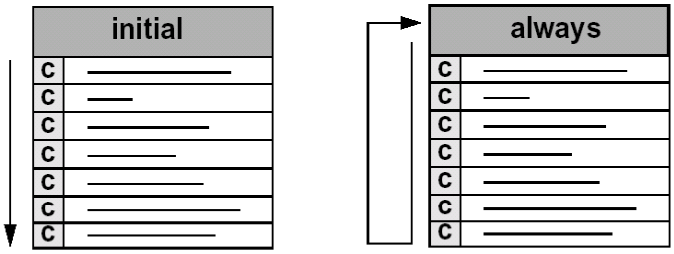
- ✓ Procedural blocks have the following components:
  - ➢ Procedural assignment statements to describe the data flow within the block
  - ➢ High-level constructs (loops, conditional statements) to describe the functional operation of the block
  - ➢ Timing controls to control the execution of the block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



Behavioral Modeling - Structured procedures

- ✓ Structured procedures (four types)
  - ➢ always statement
    - Each always statement repeats continuously throughout the whole simulation run.
    - Need timing control to prevent Deadlock condition.
      Ex : always areg = ~areg; // Deadlock!!
  - ➢ initial statement
    - An initial statement is executed only once.
    - Note : "initial" can't be synthesized!!
  - ➢ task, Function

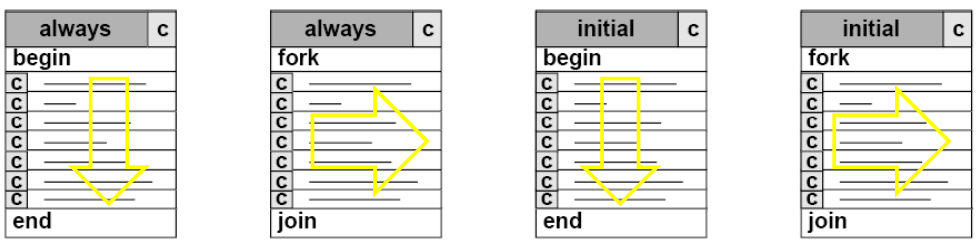
Note : each always statement and initial statement starts a separate activity flow, but all of flow are concurrent.



Block Statements

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

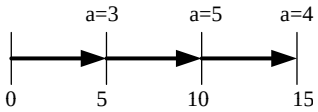
- ✓ Sequential block statements are enclosed between the key words ***begin*** and ***end***.
- ✓ Parallel block (concurrent block) statements are enclosed between the key words ***fork*** and ***join***.
  - ➢ Note that ***fork-join*** blocks are typically not synthesizable and result in inconsistent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



Block Statements (2/2)

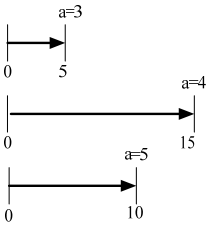
- ✓ In a sequential block, statements are evaluated and executed one after the other.

```
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;
  #15 a = 4;
  #10 a = 5;
join
```




```
fork
  a = 0;
  #5 a = 1;
  #5 a = 2;
join
```

time5 a = 2

" a " would be the last assigned value at the same time step

Procedural Timing Control (1/2)

You can specify procedural timing inside of procedural blocks, using three types of timing controls:

1. Simple delays, or pound delays: ***#(delay)***
   - Delays execution for a specific number of time steps.
     - Rise/fall and min/typ/max delay is valid in continuous assignment
       assign #(2:3:4, 3:4:5, 5:6:8) a = ~b;
     - You can only specify min/typ/max delay in procedural assignment
       always@(posedge CLK) #(3:4:6) a = ~b;
2. Edge-sensitive timing controls: **@(<signal>)**
   - Delays execution until an edge occurs on signal. You can specify the active edge of signal using ***posedge*** or ***negedge***. You can specify several signal arguments using the ***or*** keyword.
     - always@(posedge clk) a <= b;
     - always@(a or b) c = a + b;
3. Level-sensitive timing control: **wait(<expr>)**
   - Delays execution until <expr> evaluates TRUE (non-zero). If <expr> is already TRUE, the statement executes immediately.
     - wait(a == b) c = a;

Procedural Timing Control : Simple Delay

- ✓ Use simple delays (#delays) to delay stimulus in a test bench, or to approximate real-world delays in behavioral models.

```
module muxtwo  (out, a, b, sl);
input a,b,sl;
output out;reg out;
always @(sl or a or b)
  if (!sl)
    #10 out = a; // The delay from a to out is 10 time units
  else
    #12 out = b; // The delay from b to out is 12 time units
endmodule
```

- ✓ You can use module parameters to parameterize simple delays.

```
module clock_gen (clk);
output clk;
reg clk;
parameter cycle = 20;
initial clk = 0;
always
   #(cycle/2)  clk = ~clk;
endmodule
```


Procedural Timing Control : Edge-Sensitive Timing

- ✓ 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.
  - ➢ The **or** event control modifier has nothing to do with the **bitwise-OR** operator "|" or the **logical-OR** operator "||".

In Verilog2001, can use 『,』 or 『or』

```
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
```

Pay Attention ! Sensitivity List !!
Triggers the action in the body
In Verilog2001, can use 『』*

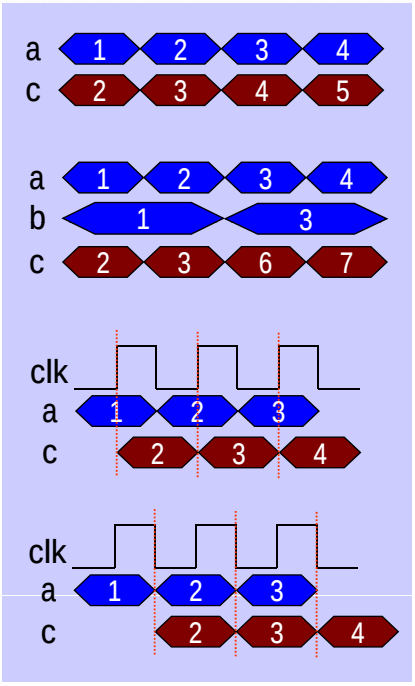
Combinational block

Sequential block

Event-Based Timing Control Example (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

- ✓ **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;



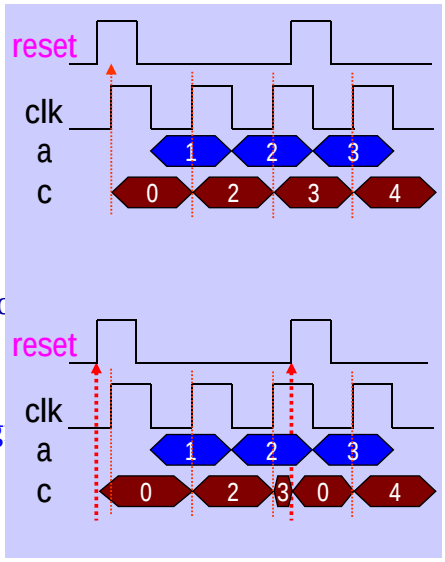
Event-Based Timing Control Example (2/2)

- ✓ Register with synchronous reset
  - ➢ @(posedge clk): for synchronous reset
    - Ex.

```
always @(posedge clk) begin
  if(reset) c <= 0;
  else c <= a+1;
end
```

- ✓ Register with asynchronous reset
  - ➢ @(posedge clk or posedge reset): for asynchronc reset
    - Ex.

```
always @(posedge clk or posedge reset) beg
  if(reset) c <= 0;
  else c <= a+1;
end
```



Edge-Sensitive Timing : Event control

- ✓ Event control can't be synthesized !!
- ✓ You can use changes on nets and register as event to trigger the execution of a statement.
- ✓ Syntax
  - ➢ @< event_expression > < statement_or _null >
    - Statement will wait for the data of <event_expression> changed, then execute statement.
- ✓ Example :
  - ➢ @(ee) rega = regb;
    // controlled by and value changes in the register ee;
  - ➢ @(posedge clk) rega = regb;
    // controlled by posedge on clk

```
initial  begin
         clk =  0;
         rst = 0;
         data_in = 0;
         #( cycle/2 ) rst = 1;
         @(  in_en ) data_in = 8'hff;
         @(  posedge clk ) data_in = 8'hff;
         for (i=0; i<=10; i=i+1) begin
                        @( posedge clk )
                        data_in = pattern[i];
         end
         ……
end
```


Level-sensitive Timing: The Wait Statement

- ✓ Use ***wait*** for level-sensitive timing control in behavioral code.
- ✓ The following behavioral model of an adder with a latched output illustrates edge-sensitive timing with the ***or*** keyword as well as level-sensitive timing with the ***wait*** statement.
- ✓ Note that wait is not synthesizable.

```
module latch_adder (out, a, b, enable);
input enable;
input [2:0]a,b;
output [3:0]out;
reg [3:0]out;
always @(a or b)
begin
        wait (!enable) // if enable is low, perform addition
        out = a + b;
end
endmodule
```

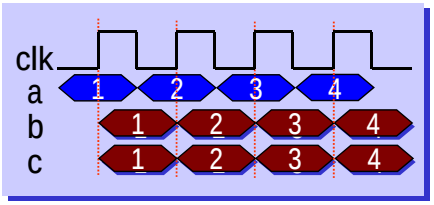
  - ➢ When still waiting, changes of a or b would be ignored.

Procedural Assignments

- ✓ **The Verilog HDL contains two types of procedural assignment**
  - ➢ **Blocking procedural assignment**
  - ➢ **Non-blocking procedural assignment**

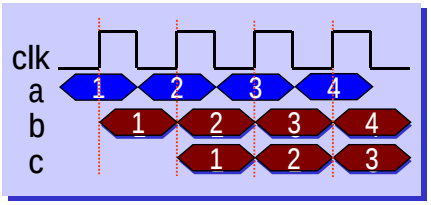
Blocking :

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



Non-blocking :

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



Behavioral Modeling – Register Description

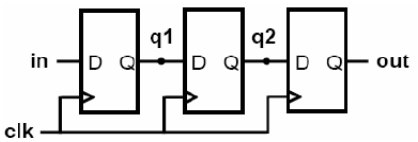
- ✓ Non-blocking assignment

Non-blocking

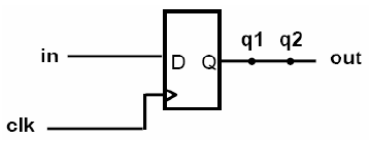
```
always @( posedge  clk)
begin
      q1 <=  in;
      q2 <=  q1;
      out <= q2;
end
```

Blocking

```
always @( posedge  clk)
begin
      q1  =  in;
      q2  =  q1;
      out  = q2;
end
```



Shift register behavior



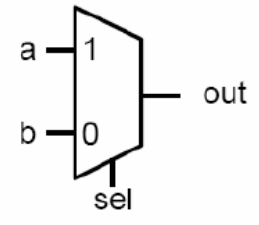
Single register behavior

Behavioral Modeling – Register Description (cont.)

✓ Comparison

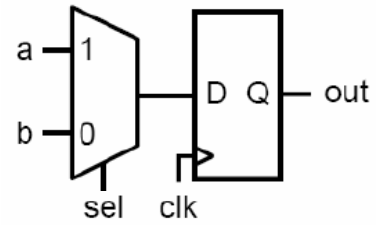
Blocking

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

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



Non-blocking

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

always @(posedge  clk )
begin
      if( sel )        out  <= a;
      else             out  <= b;
end
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!
               // half_carry is not declared,
               // and defaults to a 1-bit wire.
out = {half_carry, half_sum} ;
end
endmodule
```

Behavioral Control Statements

- ✓ **Conditional Statements**
  - ➢ **if**
  - ➢ **if-else**
  - ➢ **case**
- ✓ **Looping Statements**
  - ➢ **forever loop**
  - ➢ **repeat loop**
  - ➢ **while loop**
  - ➢ **for loop**

if and if-else Statements

- ✓ In nested ***if*** sequences, ***else*** is associated with the closest previous ***if*** (*to avoid synthesis tool to produce latch devices*).
- ✓ If condition 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.

```
if (expression)
  statement
else
  statement
```

```
if (expression)
  statement
else if (expression)
  statement
else
  statement
```

```
always@( posedge clock )  begin
 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");
end
```

Behavioral Modeling – Conditional Statements (cont.)

- ✓ Conditional Statements: The conditional statement is decide whether to execute a statement.
  - ➢ **if…,else if…,else…**: The most commonly used conditional statements. The statement occurs if the expressions controlling the if statement evaluates to be true.

```
module MUX2_1(out,a,b,sel);
input       a,b,sel;
output      out;
reg         out;

//Procedural assignment
always @(a or b or sel)
begin
   if (sel==0)   out  = a;
   else          out  = b;
end
endmodule
```

Anything assigned in an "*always*" block must also be declared as *reg* type

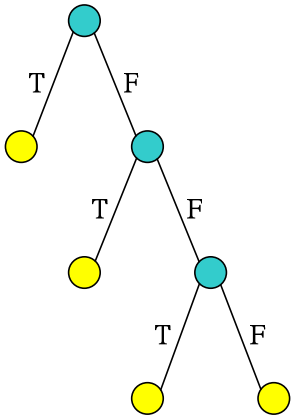
The "*always*" block runs once whenever a signal in the **sensitivity list** changes value

Behavioral Modeling – Conditional Statements (cont.)

if-else-if statements

- ✓ 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 statements.
- ✓ The last else part of the if-else-if construct handles the default case where none of the other conditions was satisfied.

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



Behavioral Modeling – Conditional Statements (cont.)

- **case** : (casex, casez) Be used for switching multiple selections.

Syntax:

```
case(expression)
   alternative1 : statement1;
    alternative2 : statement2;
    alternative3 : statement3;
     ...
    default : default statement;
  endcase
```

```
module MUX2_1(out,a,b,sel);
input       a,b,sel;
output      out;
reg         out;
//Procedural assignment
always @(a or b or sel) begin
  case(sel)
     1'b0:  out  = a;
     1'b1:  out  = b;
  endcase
end
endmodule
```

Conditional Statements (4/6)

```
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 , // specify multiple cases with the same result
              3'b100 : result = rega / regb;
              default : begin
                                result = 8'b0;
                                $display ("no match");
                        end
      endcase
endmodule
```

Conditional Statements (5/6)

case, casez and casex

- ✓ case does not allow don't-care values
- ✓ casez allow both "z" and "?" values to be treated as don't-care values
- ✓ casex allows "z", "x" and "?" to be treated as don't-care values
- ✓ "z" and "?" are always treated equal

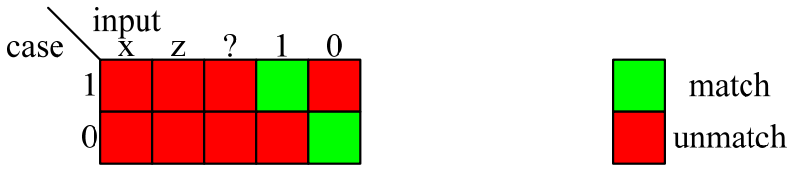
```
A[0] = 4'bx1x0;  A[1] = 4'bz1z0;   A[2] = 4'b?1?0;
for(c = 0; c < 3; c = c + 1)  begin
      casez(A[c])   // c = 0 -> 1 -> 2
           4'b1x0x: $display("casez: c=%d, statement1", c);
           4'b1z0z: $display("casez: c=%d, statement2", c);
           4'b1?0?: $display("casez: c=%d, statement3", c);
           4'b1100: $display("casez: c=%d, statement4", c);
           default  : $display("casez: c=%d, statement5", c);
      endcase
end
for(c = 0; c < 3; c = c + 1)  begin
      casex(A[c])
           4'b1x0x: $display("casex: c=%d, statement1", c);
           4'b1z0z: $display("casex: c=%d, statement2", c);
           4'b1?0?: $display("casex: c=%d, statement3", c);
           4'b1100: $display("casex: c=%d, statement4", c);
           default  : $display("casex: c=%d, statement5", c);
      endcase
end
```

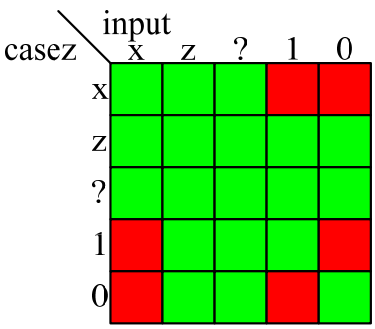
```
casez: c=    0, statement5
casez: c=    1, statement2
casez: c=    2, statement2
```

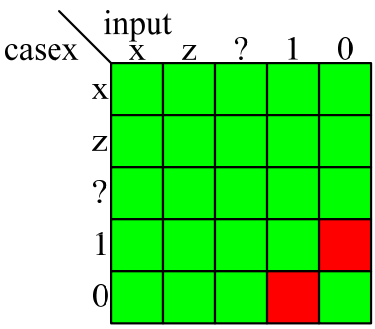
```
casex: c=    0, statement1
casex: c=    1, statement1
casex: c=    2, statement1
```


Conditional Statements (6/6)

- ✓ Equivalence Map







Behavioral Control Statements

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

- ✓ Four types
  - forever (can't synthesize)
    - Continuously executes a statement until to meet $finish or disable.
  - 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 statements by a three step process.

Note : The above is used to run simulation !!

Behavioral Modeling – Looping Statements (cont.)

- ✓ for-loop syntax
  - ➢ for (initial_assignment; condition; step_assignment)
    begin
     statement;
    end
  - ➢ Can use to initialize a memory

    EX.

```
integer  i;
always@(posedge clk or posedge reset)
begin
  if(reset)
  begin
     for(i=0; i<1024; i=i+1)
     begin
        memory[i]=0;
     end
  end
  else
   .
   .
   .
```

Behavioral Modeling – Looping Statements (cont.)

- ✓ Example
  - ➢ forever @(posedge clk) rega =~rega;
    - Need timing control to avoid dead-lock
      EX. forever rega = ~ rega // dead-lock !!
  - ➢ repeat(size) // if size = 5 then do loop five times
    begin
     statement
    end
  - ➢ While(temp) // do loop until temp=0 (false)
    begin
     statement
     temp >> 1;
    end

Note (for repeat) :

1. Number of iteration is a constant :It can synthesize, but it is inefficient. So not recommend to use.
2. If number of iterations is a variable : It can't be synthesized!!

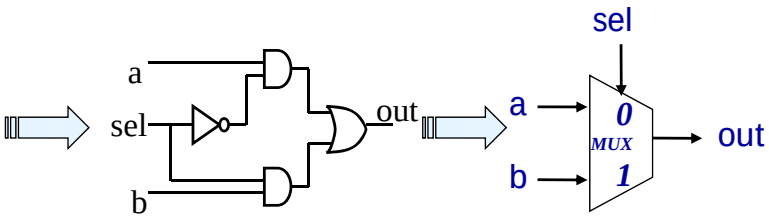
Assignments – Continuous Assignment

- ✓ Conditional Statements

  Syntax:

  assign <out_name> = (<expression>) ? true_statement : false_statement

```
module MUX2_1(out,a,b,sel);
input       a,b,sel;
output      out;
wire        out;

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

endmodule
```



Assignments – Continuous Assignment (cont.)

- ✓ The assignment is always active
  - ➢ Whenever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS.

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

- ✓ Net declaration assignment
  - ➢ An equivalent way of writing net assignment statement.
  - ➢ Can be declared once for a specific net.

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

- ✓ In the implicit continuous assignment statement, It's not allowed which required a concatenation on the LHS.

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

→ ***Error!!***

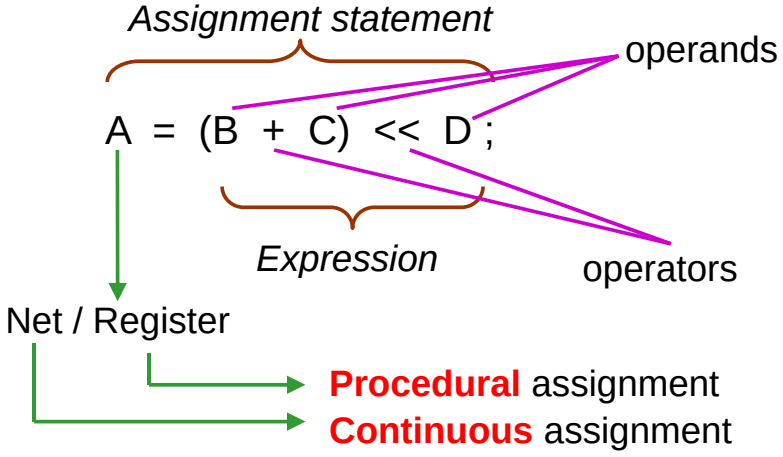
Assignments – Continuous Assignment (cont.)

- ✓ The example illustrates the power of using conditional operators in continuous assignments, and driving a net from multiple assignments.
- ✓ Only one assignment at any given time drives a non-tristate value onto *MUX2,* and if all drivers are tristate, the net defaults to a pull-strength 1.

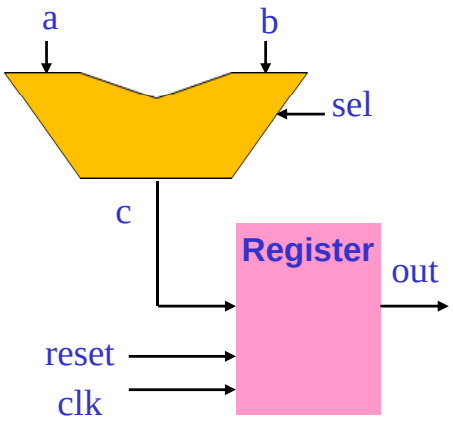
```
module cond_assigns (MUX1, MUX2, a, b, c, d, sel);
output      MUX1, MUX2;
input       a, b, c, d;
input  [ 1:0 ]  sel ;
tri1   MUX2;
   assign MUX1 = sel == 2'b00 ? a :
                 sel == 2'b01 ? b :
                 sel == 2'b10 ? c : d;
   assign MUX2 = sel == 0 ? a : 'bz,
          MUX2 = sel == 1 ? b : 'bz,
          MUX2 = sel == 2 ? c : 'bz,
          MUX2 = sel == 3 ? d : 'bz;
endmodule
```


Summary

- ✓ Update時間不同
  - ➢ 在Continuous assignment，LHS是隨RHS改變而改變，也就是assign的動作是連續不斷地在發生。而Procedural assignment，LHS則是在assign這個statement被執行到時(有timing control)，RHS的值才會被update到LHS。
- ✓ Assignment statement 放置的地方不同
  - ➢ Procedural assignment statement放在Procedural block 中，而Continuous assignment statement則不可以放在Procedural block中。



Behavioral Modeling 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
```

About reset

✓ D-flip-flop with synchronous set and reset example:

```
module dff(q, d, clk, set, rst);
    input d, clk, set, rst;
    output q;
    reg q;

    always @(posedge clk)
        if (rst)
            q <= 1'b0;
        else if (set)
            q <= 1'b1;
        else
            q <= d;
endmodule
```

This gives priority to reset over set and set over d.

Synthesizable Verilog Code

- ✓ Four data type can be synthesized
  - ➢ Input, output, reg, wire
- ✓ 1-D data type is convenient for synthesis
  - ➢ EX. reg [7:0] a;
    reg [7:0] a[3:0]; →It isn't well for the backend verifications
- ✓ Examples of synthesizable register description
  - ➢ always@(posedge clk)
  - ➢ always@(negedge clk)
  - ➢ always@(posedge clk or posedge reset) if(reset).. else..
  - ➢ always@(negedge clk or posedge reset) if(reset).. else..
  - ➢ always@(posedge clk or negedge reset) if(~reset).. else..
  - ➢ always@(negedge clk or negedge reset or negedge set) if(~set).. else if (~reset).. else..
- ✓ Example of not synthesizable register description
  - ➢ always@(posedge clk or negedge clk)

Synthesizable Verilog Code (cont.)

- ✓ Use below description for synthesis
  - ➢ assign
  - ➢ always block
  - ➢ Called sub-module
- ✓ Not use below description for synthesis
  - ➢ initial block
  - ➢ task
  - ➢ function
- ✓ Data bit order can't vary

Wrong!!

```
initial
begin
  a=b;
  b=c;
end
```

Wrong!!

```
wire  [15:0]  a;
reg   [3:0]   b;

assign a[b] = 0;
```

Synthesizable Verilog Code (cont.)

✓ Data has to be described in one always block

```
always@(posedge clk)
        out <= out + 1;
always@(posedge clk)
        out <= a;
```

Wrong!!

✓ Data has to be described by either blocking assignment or non-blocking assignment.

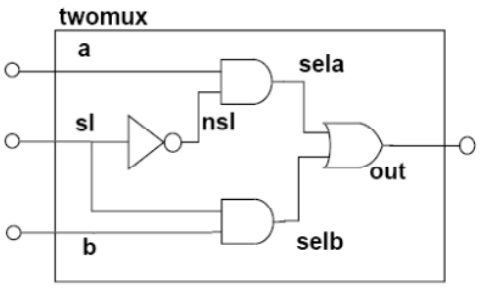
```
always@(posedge clk or posedge reset)
        if(reset) out = 0;
else    out <= out + in;
```

Wrong!!

Multiplexor Example (OPTIONAL)

◆ 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);
endmodule
```

◆ Behavioral modeling

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

4-1 Multiplexor Example (OPTIONAL)

◆ Dataflow modeling

```
module mux4_1 (out, in0, in1, in2,
in3, sel);
    output out;
    input in0, in1, in2, in3;
    input [1:0] sel;

    assign out =  (sel == 2'b00) ? in0 :
                  (sel == 2'b01) ? in1 :
                  (sel == 2'b10) ? in2 :
                  (sel == 2'b11) ? in3 : 1'bx;

endmodule
```

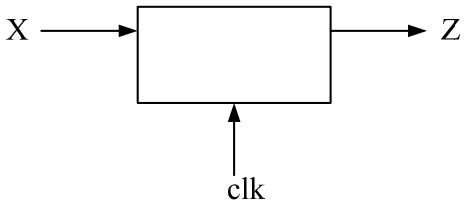
◆ Behavioral modeling

```
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
```


FSM introduction

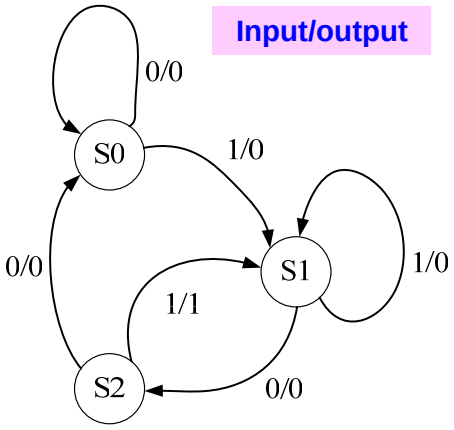
What is FSM?

- ✓ A desired relationship between the input and the output sequences
- ✓ Modeling sequencer circuits
  - ➢ Let the sequences pass through specific states to generate the required output sequences
  - ➢ The state are a predetermined sequential manner
- ✓ Example
  - ➢ Z gives 1 when X is 101
    - ➢ X = 001011010110011
    - ➢ Z = 000010010100000
- ✓ FSM structures
  - ➢ Mealy-machine
  - ➢ Moore-machine



Mealy Machine

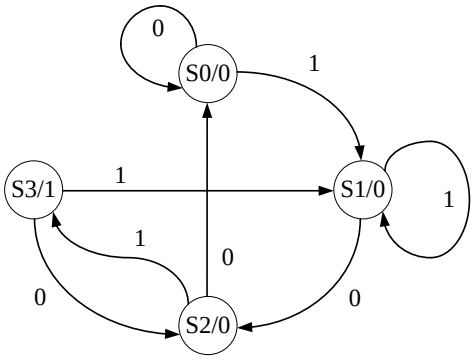
- ✓ The outputs depends on the **present state and the incoming input**
- ✓ The output may change if the inputs change during a clock cycle
- ✓ Derivation of state tables
  - ➢ Z gives 1 when X is 101
    - ⮝ X = 001011010110011
    - ⮝ Z = 000010010100000
  - ➢ Start from a reset state S0



Present state	Next state		Present Output	
	X = 0	X = 1	X = 0	X = 1
S0	S0	S1	0	0
S1	S2	S1	0	0
S2	S0	S1	0	1

Moore Machine

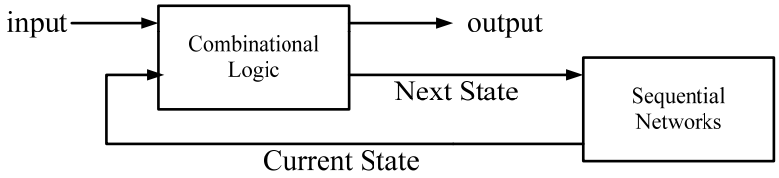
- ✓ The outputs of the machine depends on the **present state only**
- ✓ The output are synchronized with the clock (因為input不影響輸出結果)
- ✓ Derivation of state tables
  - ➢ Z gives 1 when X is 101
    - ⮝ X = 001011010110011
    - ⮝ Z = 000010010100000
  - ➢ Start at S0, Z = 0



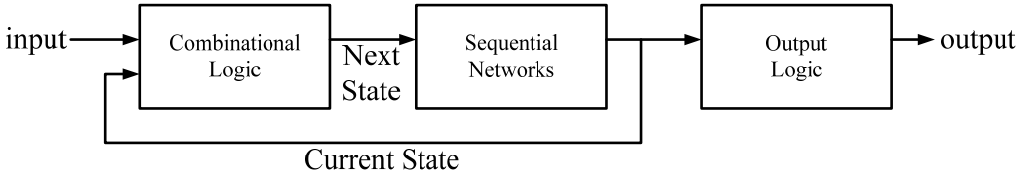
Present state AB	Next state X = 0	X = 1	Z
S0(00)	S0(00)	S1(01)	0
S1(01)	S2(11)	S1(01)	0
S2(11)	S0(00)	S3(10)	0
S3(10)	S2(11)	S1(01)	1

FSM structures

- ✓ Mealy machine
  - ➢ Contains sequential networks and combinational logic.
  - ➢ The sequential network is a set of Flip-Flops.
  - ➢ The output follows the input and current state.

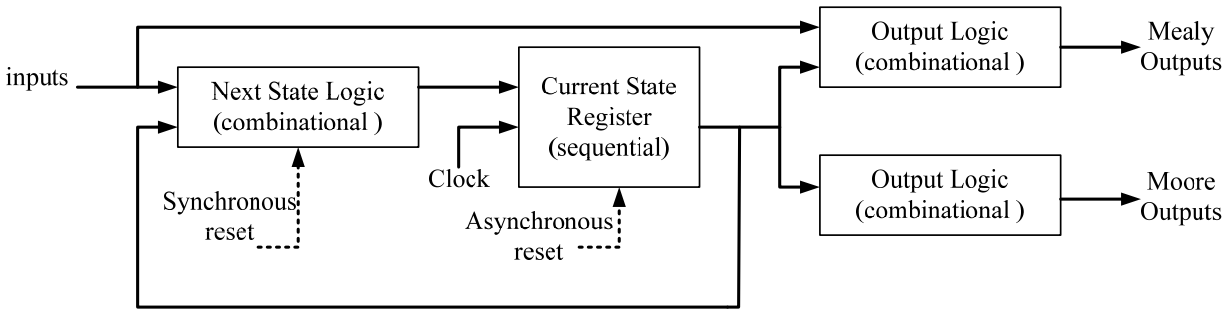


- ✓ Moore machine
  - ➢ Contains sequential networks and combinational logic.
  - ➢ Sequential network is a set of Flip-Flops.
  - ➢ The output depends on the current state only.



FSM HDL Code Model

- ✓ HDL code model
  - ➢ Current state register (CS) -- sequential
  - ➢ Next state logic (NS) -- combinational
  - ➢ Output logic (OL) -- combinational
- ✓ A state machine can only be in one state at a given time
- ✓ Each active transition of clock causes it to change from its current state to the next state as defined by the NS.



Asynchronous and Synchronous Reset

- ✓ Use an asynchronous reset
  - ➢ Always initialized to know valid state.
  - ➢ No need to decode unused current state values.
  - ➢ Minimizing the next state logic.

- ✓ No reset or use the synchronous reset
  - ➢ May power up and permanently stuck in an unused state
  - ➢ All 2^n binary values must be decoded in the next logic
  - ➢ Using smaller flip-flops without an asynchronous reset input

- ✓ An asynchronous reset must be included in the event list of the always statement with the posedge or negedge clause
  - ➢ Eg. always@(posedge clk or negedge reset)

State Encoding

- ✓ To assign binary numbers to states
- ✓ Commonly used encoding formats
  - ➢ Sequential
  - ➢ Gray
  - ➢ Johnson
  - ➢ One-Hot
  - ➢ Define your own

No.	Sequential	Gray	Johnson	One-Hot
0	0000	0000	00000000	0000000000000001
1	0001	0001	00000001	0000000000000010
2	0010	0011	00000011	0000000000000100
3	0011	0010	00000111	0000000000001000
4	0100	0110	00001111	0000000000010000
5	0101	0111	00011111	0000000000100000
6	0110	0101	00111111	0000000001000000
7	0111	0100	01111111	0000000010000000
8	1000	1100	11111111	0000000100000000
9	1001	1101	11111110	0000001000000000
10	1010	1111	11111100	0000010000000000
11	1011	1110	11111000	0000100000000000
12	1100	1010	11110000	0001000000000000
13	1101	1011	11100000	0010000000000000
14	1110	1001	11000000	0100000000000000
15	1111	1000	10000000	1000000000000000

FSM HDL Coding Styles (1/4)

- ✓ The three blocks may be combined or separated within a Verilog model as
  - ➢ Separate CS, NS and OL
  - ➢ Combine CS and NS, separate OL.
  - ➢ Combine NS and OL, separate CS.
- ✓ Advantage of the model
  - ➢ Easier for synthesis tools to recognize the FSM
  - ➢ To get a better structure optimization
- ✓ The NS is best modeled by using the case statement
  - ➢ To avoid explicitly defining all 2^n values that are not part of the state machine.
    - ➢ A state machine with "N" state FF has 2^n possible binary numbers that can be used to represent states. Often, not all 2^n numbers are needed, so the unused ones should be designed no to occur during normal operation.

FSM HDL coding Style (2/4)

- ✓ Separate CS, NS, and OL

```
always@( posedge clk )
        current_state <= next_state;
```

```
always@( current_state or In )    begin
        case (current_state)
                state_0 : case (In)
                        In0 : next_state <= state_value1;
                        In1 : next_state <= state_value2;
                                ……
                        endcase
                ……
                state_n :  ……
        endcase
end
```

CS

NS

Mealy machine

Moore machine

```
always@( current_state or In )
        Z <= values;
```

```
always@( current_state )
        Z <= values;
```

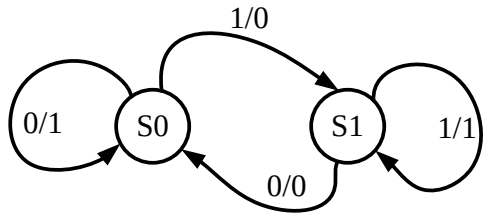
OL

FSM HDL Coding Styles: Example (1/3)

✓ Separate CS, NS, and OL

```
module FSM(in, out, clk, reset);
input in, clk, reset;
output out;

reg out, current_state, next_state;

parameter state0 = 0, state1 = 1;
```

```
always@(posedge clk or posedge reset)
begin  // CS
    if (reset) current_state <= state0;
    else current_state <= next_state;
end
```




```
always@(current_state or in )
begin  // NS
    case (current_state )
        state0 : next_state = (in == 0)? state0 : state1;
        state1 : next_state = (in == 0)? state0 : state1;
    endcase
end
```

```
always@(current_state or in)
begin  // OL
    case (current_state )
        state0 : out = (in == 0)? 1 : 0;
        state1 : out = (in == 0)? 0 : 1;
    endcase
end

endmodule
```


FSM HDL coding Style (3/4)

✓ Combined CS and NS, separate OL

```
always@( posedge clk )   begin
      case (current_state)
            state_0 : case (In)
                  In0 : current_state <= state_value1;
                  In1 : current_state <= state_value2;
                        ……
                  endcase
            ……
            state_n : ……
      endcase
end
```

CS & NS

Mealy machine

```
always@( current_state or In )
      Z <= values;
```

Moore machine

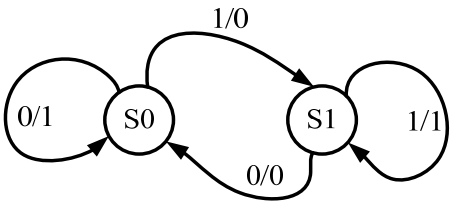
```
always@( current_state )
      Z <= values;
```

OL

FSM HDL Coding Styles: Example (2/3)

✓ Combined CS and NS, separate OL

```
module FSM(in, out, clk, reset);
input in, clk, reset;
output out;

reg out, current_state;

parameter state0 = 0, state1 = 1;

always@(posedge clk or posedge reset)
begin  // CS and NS
    if (reset) current_state <= state0;
    else begin
        case (current_state )
            state0 : current_state = (in == 0)? state0 : state1;
            state1 : current_state = (in == 0)? state0 : state1;
        endcase
    end
end

always@( current_state or in)
begin  // OL
    case (current_state )
            state0 : out = (in == 0)? 1 : 0;
            state1 : out = (in == 0)? 0 : 1;
    endcase
end

endmodule
```



FSM HDL coding Style (4/4)

✓ Combine NS and OL, separate CS

```
always@( posedge clk )
        current_state <= next_state;

always@( current_state or In )    begin
        case (current_state)
                state_0 : case (In)
                        In0 : next_state <= state_value1;
                        In1 : next_state <= state_value2;
                                ……
                        Z <= values;    // Mealy machine
                        endcase
                ……
                state_n :  ……
                Z <= values;            // Moore machine
        endcase
end
```

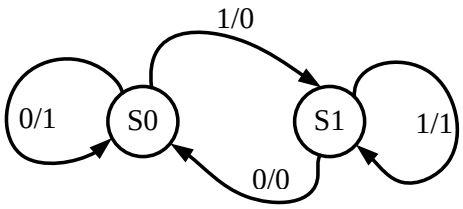
CS

NS & OL

FSM HDL Coding Styles: Example (3/3)

✓ Combine NS and OL, separate CS

```
module FSM(in, out, clk, reset);
input in, clk, reset;
output out;

reg out, current_state, next_state;

parameter state0 = 0, state1 = 1;

always@(posedge clk or posedge reset)
begin   // CS
      if (reset) current_state <= state0;
      else current_state <= next_state;
end
```




```
always@( current_state or in)
begin   // NS and OL
  case (current_state )
    state0 : begin
      case (in)
        0 : begin  next_state = state0; out = 1;  end
        1 : begin  next_state = state1; out = 0;  end
      endcase
      end
    state1 : begin
      case (in)
        0 : begin  next_state = state0; out = 0;  end
        1 : begin  next_state = state1; out = 1;  end
      endcase
      end
  endcase
end

endmodule
```