

Outline

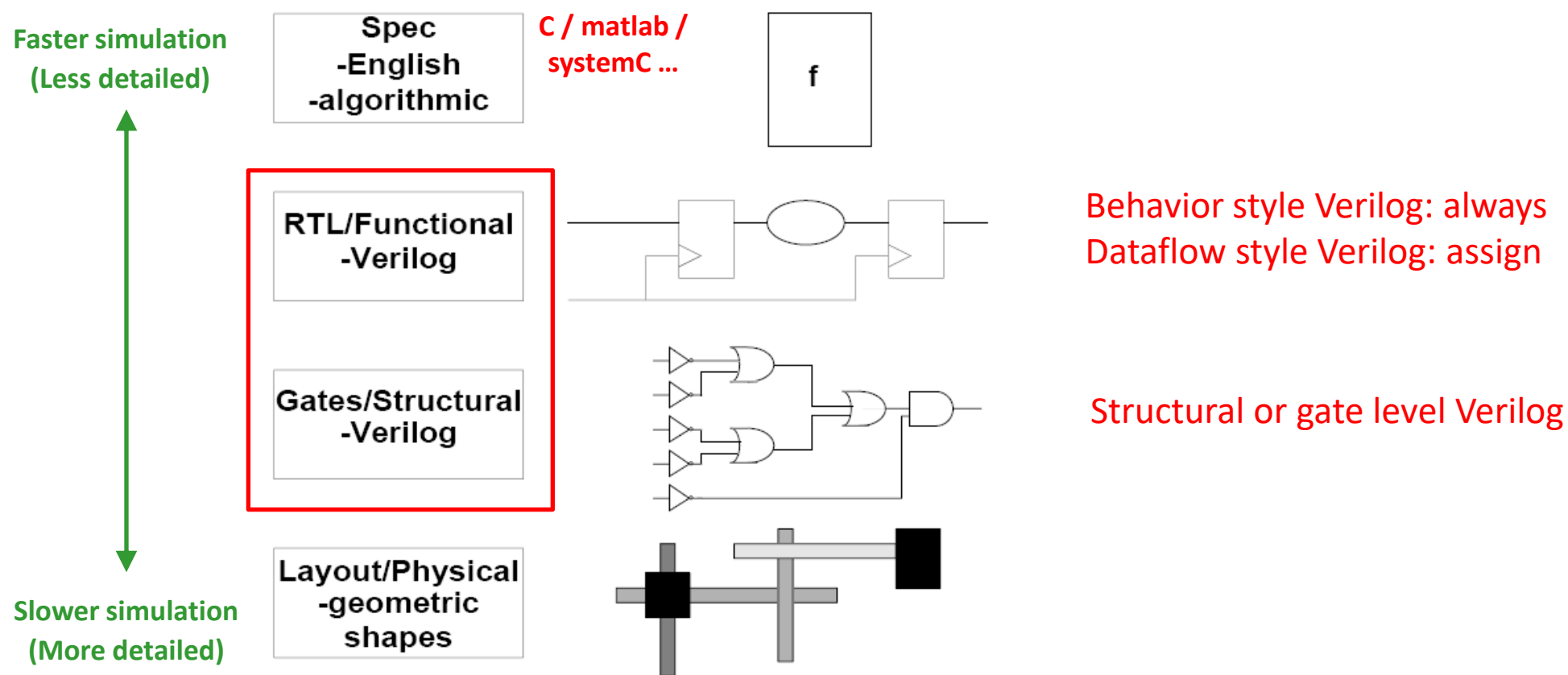
- Introduction to hardware description language (HDL)
 - Why it is better than schematic entry
- Verilog語法 1 (Functional viewpoints)
 - Modules and ports
 - Basic syntax rules
 - Structural Verilog
 - How to model combinational logic
 - How to model sequential logic
- Verilog語法 2 (Language viewpoints)
 - One language, many coding styles
 - Continuous v.s. procedural assignments
 - Blocking vs NonBlocking
 - For loop and parameterized design
- SystemVerilog
- Gotchas 正確的使用方法 <= (會踩到的陷阱)

FUNCTIONAL VERILOG

用功能性VERILOG 描述硬體設計

Levels of Abstraction

- **One design, many coding styles** to describe hardware functions



Review: Gate-level Verilog uses **structural Verilog** to connect primitive gates

```
module mux4( input  a, b, c, d, input [1:0] sel, output out );
```

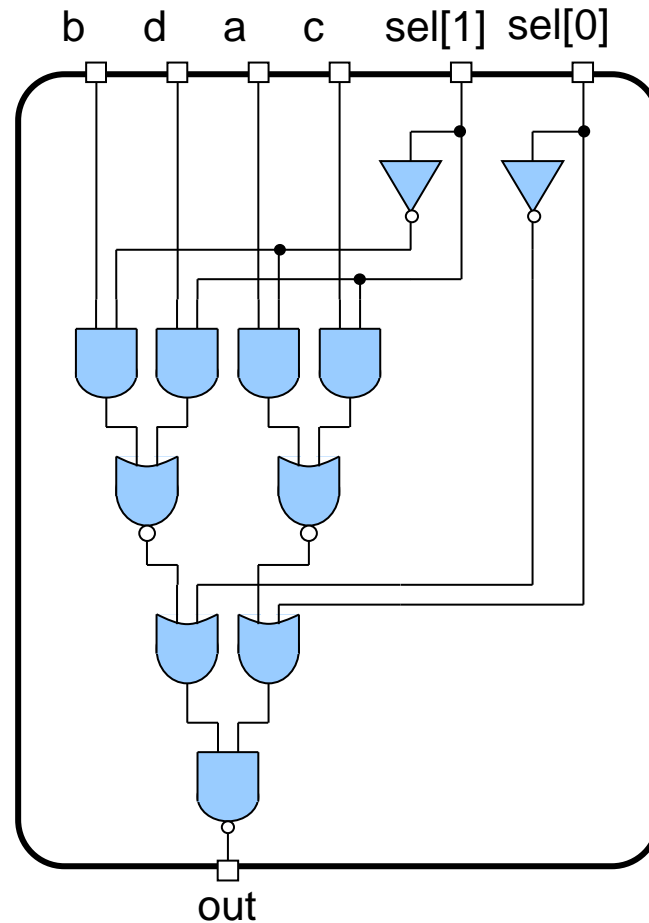
```
    wire [1:0] sel_b;
    not not0( sel_b[0], sel[0] );
    not not1( sel_b[1], sel[1] );
```

```
    wire n0, n1, n2, n3;
    and and0( n0, c, sel[1] );
    and and1( n1, a, sel_b[1] );
    and and2( n2, d, sel[1] );
    and and3( n3, b, sel_b[1] );
```

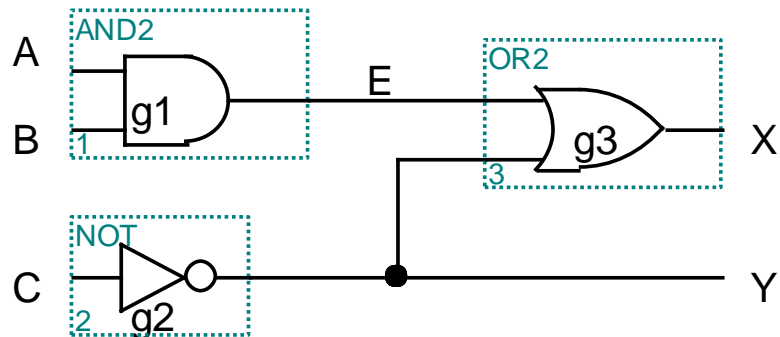
```
    wire x0, x1;
    nor nor0( x0, n0, n1 );
    nor nor1( x1, n2, n3 );
```

```
    wire y0, y1;
    or or0( y0, x0, sel[0] );
    or or1( y1, x1, sel_b[0] );
    nand nand0( out, y0, y1 );
```

```
endmodule
```



One language, Many Coding Styles



Model combinational logic

“Structural style”

```
wire E;
and g1(E,A,B);
not g2(Y,C);
or g3(X,E,Y);
```

Same as schematic

Dataflow style

Continuous assignment

```
wire E;
assign E = A & B;
assign Y = ~C;
assign X = E | Y;
```

Similar to logic equations
Use “assign”

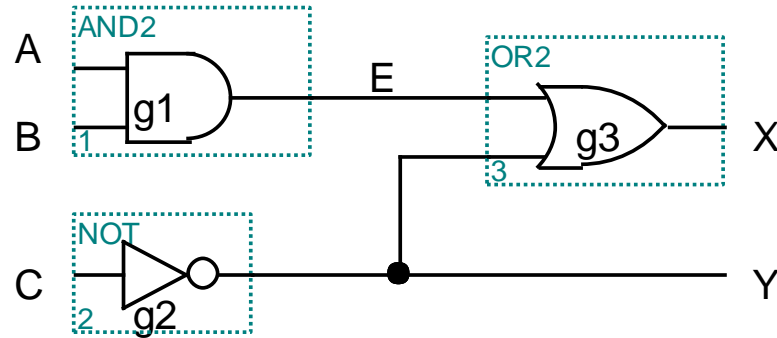
“Behavioral style”

Procedural assignment

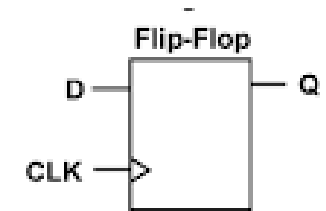
```
reg E, X, Y;
always @ (A or B or C)
begin
    E = A & B;
    Y = ~C;
    X = E | Y;
end
```

Similar to high level language
Use “always”

One language, Many Coding Styles



Model seq. logic



“Structural style”

```
wire E;
and g1(E,A,B);
not g2(Y,C);
or g3(X,E,Y);
```

Same as schematic

```
wire q;
dff d1(.d(d),
      .clk (clk))
```

Dataflow style

Continuous assignment

```
wire E;
assign E = A & B;
assign Y = ~C;
assign X = E | Y;
```

Similar to logic equations
Use “assign”

“Behavioral style”

Procedural assignment

```
reg E, X, Y;
always @ (A or B or C)
begin
    E = A & B;
    Y = ~C;
    X = E | Y;
end
```

Similar to high level language
Use “always”

```
reg q;
always @ (posedge clk)
    q <= d;
```

One language, Many Coding Styles

- Model combinational logic
 - Structural style
 - Continuous assignment
 - Procedural assignment
- Model sequential logic
 - Structural style
 - Procedural assignment

Continuous vs. Procedural Assignment

- Continuous assignment
 - Like logic equations
 - Single line Verilog statement
 - For simple logic
 - Each assignment statement is executed in parallel
 - Order does not matter
 - For combinational logic only
- Procedural assignment
 - a separate activity flow in Verilog
 - For combinational logic or
 - Single or multiple **blocking** Verilog statements (=)
 - Easy to describe complex logic
 - Order matters
 - sequential logic
 - **Nonblocking** Verilog statements (<=)
 - Executed in parallel
 - Order does not matter
- Different procedural executed in parallel

Concurrent blocks

- Blocks of code with no well-defined order relative to one another
 - Module instance is the most important concurrent block
 - Continuous assignments, and procedural blocks are concurrent within a module
 - Note. Hardware is concurrently executed

Continuous assignment statements assign one net to another or to a literal

Explicit continuous assignment

```
wire [15:0] netA;  
wire [15:0] netB;  
  
assign netA = 16'h3333;  
assign netB = netA;
```



Implicit continuous assignment

```
wire [15:0] netA = 16'h3333;  
wire [15:0] netB = netA;
```



- Implicit net declaration (not recommended)
 - If a signal name is used to the left of a continuous assignment, a implicit net declaration will be inferred

Using continuous assignments to implement an RTL four input multiplexer

```

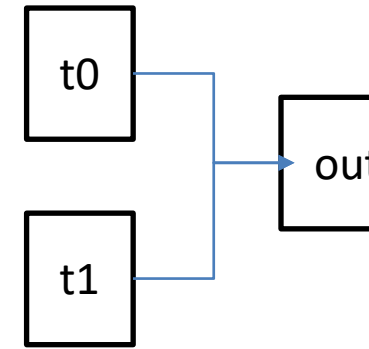
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  wire out, t0, t1;
  assign t0  = ~( (sel[1] & c) | (~sel[1] & a) );
  assign t1  = ~( (sel[1] & d) | (~sel[1] & b) );
  assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );

endmodule

```

Circuit topology



先後順序不重要

Same result if you write like this

```

assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
assign t1  = ~( (sel[1] & d) | (~sel[1] & b) );
assign t0  = ~( (sel[1] & c) | (~sel[1] & a) );

```

The **order** of these continuous assignment statements **does not matter**. They essentially happen in parallel!

But better to follow the circuit topological order

PROCEDURAL BLOCKS

Procedural blocks

- Each procedural block represent a separate activity flow in Verilog
- Procedural blocks
 - **always** blocks
 - To model a block of activity that is **repeated continuously**
 - **initial** blocks *simulation only*
 - To model a block of activity that is **executed at the beginning**
- Multiple behavioral statements can be grouped using keywords **begin** and **end**

Procedural assignments

- Procedural assignment changes the state of a reg
- Used for both **combinational and sequential logic inference**
- All procedural statements must be within `always` (or `initial`) block

```
reg A;  
always @ (B or C)  
begin  
    A = ~(B & C);  
end
```

Sensitivity list
Signal changes triggers
actions in the body

Output should use **reg** declaration for variable

Feel confusing?
Use **logic**

```
logic A;  
always @ (B or C)  
begin  
    A = ~(B & C);  
end
```

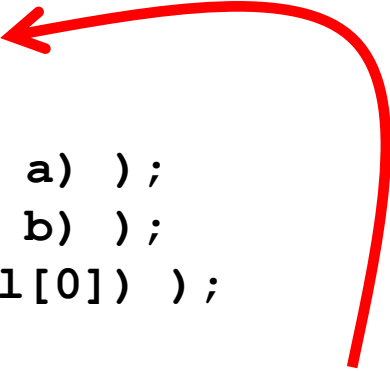
Always blocks have parallel inter-block and sequential intra-block semantics

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

    reg out, t0, t1;

    always @( a or b or c or d or sel )
    begin
        t0 = ~( (sel[1] & c) | (~sel[1] & a) );
        t1 = ~( (sel[1] & d) | (~sel[1] & b) );
        out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
    end

endmodule
```



ORDER MATTERS in procedural blocks
Executed sequentially from top to bottom
對blocking assignment 順序很重要

The always block **run once**
whenever a **signal in its**
sensitivity list changes

Sensitivity List in always

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

    reg out, t0, t1;

    always @( a or b or c or X or sel )
    begin
        t0 = ~( (sel[1] & c) | (~sel[1] & a) );
        t1 = ~( (sel[1] & d) | (~sel[1] & b) );
        out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
    end

endmodule
```

**What happens if we accidentally
forget a signal on the sensitivity list?**

Simulation will miss, but synthesis will ignore the sensitivity list
=> Simulation synthesis mismatch

Auto sensitivity list in always

```

module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );


    reg out, t0, t1;

    always @( * )
    begin
        t0 = ~( (sel[1] & c) | (~sel[1] & a) );
        t1 = ~( (sel[1] & d) | (~sel[1] & b) );
        out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
    end

endmodule

```

Feel confusing?
Use **always_comb**



```

always_comb
begin
    t0 = ~( (sel[1] & c) | (~sel[1] & a) );
    t1 = ~( (sel[1] & d) | (~sel[1] & b) );
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end

```

Verilog-2001 provides special syntax to
automatically create a sensitivity list for
all signals read in the always block

Always block – Event control @

- Always blocks model an activity that is repeated continuously
- @ can control the execution
 - posedge or negedge make sensitive to edge
 - @* / @(*), are sensitive to any signal that may be read in the statement group
 - Use “,”/or for multiple signals

Logic specific always in SystemVerilog

always_comb

always_ff

always_latch

Always block – Event control @

```
module M1 (input B, C, clk, rst, output reg X, Y,Z);  
  // controlled by any value change in B or C  
  always @ (B or C)  
    X = B & C;  
  
  // Controlled by positive edge of clk  
  always @(posedge clk)  
    Y <= B & C;  
  
  // Controlled by negative edge of clk or rst  
  always @(negedge clk, negedge rst)  
    if (!rst) Z <= B & C;  
    else      Z <= B & C;  
endmodule
```

Blocking assignment

NonBlocking assignment

Procedural blocks (summary)

- Blocks of code within a concurrent block which are read (simulated, executed) **in order**
- Procedural blocks may contain:
 - Blocking assignments
 - Nonblocking assignments
 - Procedural control statements (**if, for, case**)
 - function, or task calls
 - Event control (**'@'**)
 - procedural blocks enclosed in **begin ... end**

BLOCKING VS. NONBLOCKING

Procedural Assignment

- Assign value to registers
- **Blocking** procedural assignment (for combinational logic)
 - Use “=”
 - An assignment is completed before the next assignment starts
 - Use for **always@(*)**
- **Non-blocking** procedural assignment (for sequential logic)
 - Use “<=”
 - Assignments are executed in parallel
 - Use for **always@(posedge clock)**

Blocking v.s. NonBlocking

- Verilog has two different types of assignments: **blocking** & **nonblocking**.
- Blocking assignments **=** are executed in the order they appear, therefore they are done one after another. Therefore the first statement “blocks” the second until it is done, hence it is called blocking assignments.

```
a = b;
b = a;
// both a & b = b
```

blocking

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

```
begin
```

```
  x = a | b;
```

1. Evaluate $a | b$, assign result to x

```
  y = a ^ b ^ c;
```

2. Evaluate $a^b c$, assign result to y

```
  z = b & ~c;
```

3. Evaluate $b \& (\sim c)$, assign result to z

```
end
```

blocking

- Non-blocking assignments **<=** are executed in parallel. Therefore an earlier statement does not block the later statement. Note the subtle effect this has within **always** block:

```
a <= b;
b <= a;
// swap a and b
```

Non-blocking

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

```
begin
```

```
  x <= a | b;
```

1. Evaluate $a | b$ but defer assignment of x

```
  y <= a ^ b ^ c;
```

2. Evaluate $a^b c$ but defer assignment of y

```
  z <= b & ~c;
```

3. Evaluate $b \& (\sim c)$ but defer assignment of z

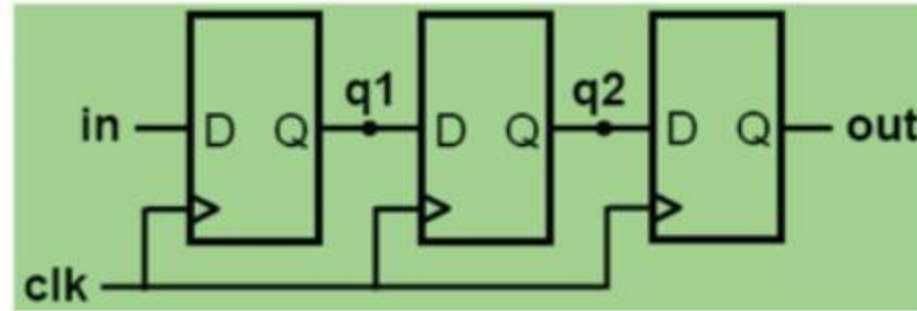
```
end
```

4. Assign x , y , and z with their new values

Non-blocking

Blocking vs NonBlocking

- ◆ Here are two versions of a 3-stage shift register consisting of 3 flipflops using blocking and nonblocking assignments.
- ◆ Will they give the same results?



```

module blocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;
  always @ (posedge clk)
  begin
    q1 = in;
    q2 = q1;
    out = q2;
  end
endmodule

```

```

module nonblocking(in, clk, out);
  input in, clk;
  output out;
  reg q1, q2, out;
  always @ (posedge clk)
  begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
  end
endmodule

```

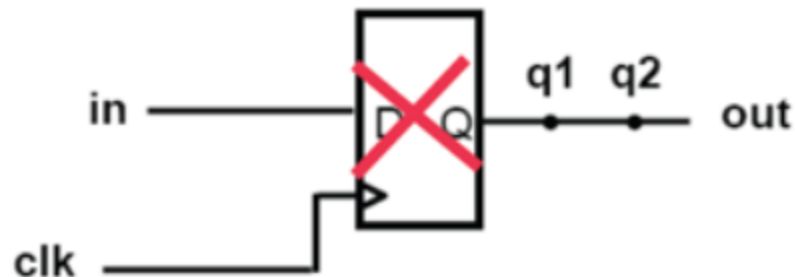


```

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

```

- At each rising clock edge:
 $q1 = in$,
then $q2 = q1 = in$,
then, $out = q2 = q1 = in$.
- Therefore $out = in$, which is NOT the intention.

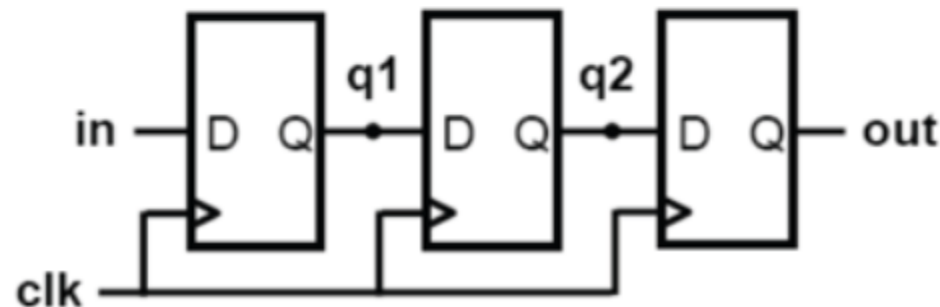


```

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

```

- At each rising clock edge, $q1$, $q2$ and out simultaneously receive the old values of in , $q1$, $q2$ respectively.

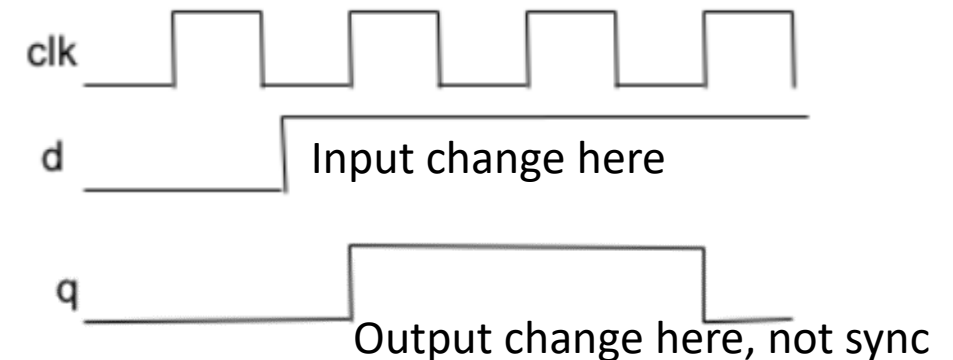


Six guidelines for using blocking and non-blocking assignment statements

- 1. **Flip-flops** should use non-blocking
- 2. **Latches** should use non-blocking
- 3. **Combinational logic should use blocking**
- 4. **Do not mix** combinational and sequential logic in the same always block
 - Hard to debug with waveform
- 5. Do not assign to the same variable from more than one always block
 - See [Multi-driver slide in the “Gotcha” part](#)

} Sequential logic

```
always @(posedge clk)
begin
    q <= d ? (a+1) : (d+f);
end
```



Six guidelines for using blocking and non-blocking assignment statements

- Do not mix blocking and non-blocking assignment

```
always @( posedge clk )  
begin  
    q <= d ? (a+1) : (d+f) ;  
    a = c + d;  
end
```

Continuous and procedural assignment statements are very different

Continuous assignments are for naming and thus we cannot have multiple assignments for the same wire

```
wire out, t0, t1;
assign t0 = ~( (sel[1] & c) | (~sel[1] & a) );
assign t1 = ~( (sel[1] & d) | (~sel[1] & b) );
assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
```

想像成有實際電路
同名，代表會被接再一起，
會無法決定真正的值該是多少

Procedural assignments hold a value semantically, but it is important to distinguish this from hardware state

```
reg out, t0, t1, temp;
always @( * )
begin
    temp = ~( (sel[1] & c) | (~sel[1] & a) );
    t0 = temp;
    temp = ~( (sel[1] & d) | (~sel[1] & b) );
    t1 = temp;
    out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
end
```

Illegal for this, multiple driver

```
assign temp = ~( (sel[1] & c) | (~sel[1] & a) );
assign t0 = temp;
assign temp = ~( (sel[1] & d) | (~sel[1] & b) );
assign t1 = temp;
assign out = ~( (t0 | sel[0]) & (t1 | ~sel[0]) );
```

在always中，只是文法上的姓名，不代表真的硬體

A COMPLETED EXAMPLE

Example

把sequential logic 和 combinational logic 分開
Sequential logic => 各種的DFF，沒有logic equation

```

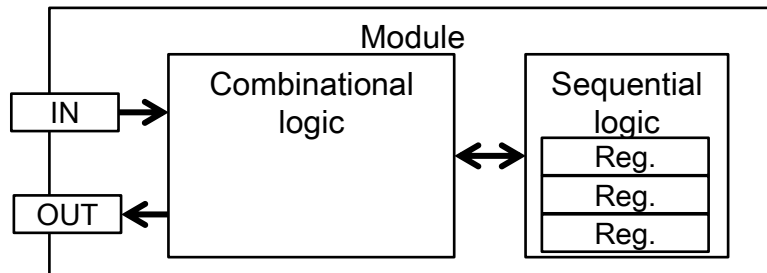
module design(
    // input
    input [7:0] data_in,
    // system
    input clk_p,
    input rst_n,
    // output
    output logic [7:0] data_out
);
    logic [7:0] value_w;

```

```

    // combinational circuit
    always_comb
    begin
        value_w = data_in + 7'd1;
    end
    // sequential circuit
    always_ff@(posedge clk_p, negedge rst_n)
    begin
        if(~rst_n)begin
            data_out <= 7'd0;
        end else begin
            data_out <= value_w;
        end
    end
end
endmodule

```



Simple testbench

```
`timescale 1ns/1ps //time resolution
```

```
`include "design.v"
```

```
`include "pattern.v"
```

```
module testbench;
// inter connection wire
wire      clk_p;
wire      rst_n;
wire [7:0] data_in;
wire [7:0] data_out;
```

```
// connection, test pattern
pattern U_pattern(
    .clk_p(clk_p),
    .rst_n(rst_n),
    .data_in(data_in),
    .data_out(data_out));
```

```
//my design
design U_design(
    .clk_p(clk_p),
    .rst_n(rst_n),
    .data_in(data_in),
    .data_out(data_out));
```

Test pattern generation
and response check

DUT

```
// dumping waveform
```

```
initial begin
```

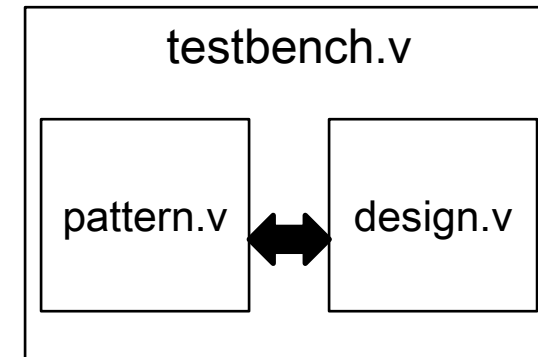
```
    $fsdbDumpfile("waveform.fsdb");
```

```
    $fsdbDumpvars;
```

```
end
```

```
endmodule
```

← Save waveforms of all variables



Simple testbench (SystemVerilog Version)

```
`timescale 1ns/1ps //time resolution
```



```
`include "design.v"
```

```
`include "pattern.v"
```

```
module testbench;
// inter connection wire
wire      clk_p;
wire      rst_n;
wire [7:0] data_in;
wire [7:0] data_out;
// connect test pattern
pattern U_pattern(
    .*);
//my test
design U_design(
    .*));
// dumping waveform
initial begin
    $fsdbDumpfile("waveform.fsdb");
    $fsdbDumpvars;
end
endmodule
```

Defines the time units and simulation precision (smallest increment)

```
`timescale <reference_time_unit> / <time_precision>
```

- reference_time_unit: simulation time unit
- time_precision: unit for rounding 四捨五入的單位

The precision unit must be less than or equal to the time unit

```
`ifdef RTL
```

```
    `timescale 1ns/100ps
```

```
`endif
```

```
//finer resolution for gate level
```

```
`ifdef GATE
```

```
    `timescale 1ns/10ps
```

```
`endif
```

```
//global parameters for all modules
```

```
`define CLK_PERIOD 30.0
```


Simple test pattern

```

module pattern(
    clk_p,
    rst_n,
    data_in,
    data_out
);
// parameter setting
parameter cycle = 10;
parameter Th = 2;
// I/O declaration
output      clk_p;
output      rst_n;
output [7:0] data_in;
input  [7:0] data_out;
reg       clk_p;
reg       rst_n;
reg  [7:0] data_in;
// clock
always begin
    #(cycle/2.0) clk_p = 1'b1;
    #(cycle/2.0) clk_p = 1'b0;
end

```

```

initial begin
    // reset
    rst_n = 1'b1;
    wait(clk_p!=1'bx);
    @(negedge clk_p);
    rst_n = 1'b0;
    @(negedge clk_p);
    rst_n = 1'b1;
    // stimulus generation
    @(posedge clk_p);
    #(Th)
    data_in = 8'd8; // 1st cycle
    @(posedge clk_p);
    #(Th)
    data_in = 8'd2; // 2nd cycle
    .....
    $finish;
end
endmodule

```

← Finish simulation

FOR LOOP IN VERILOG DESIGN **(ONLY WITHIN PROCEDURAL BLOCKS)**

- Provide a shorter way to express a series of statements.
- Loop index variables must be integer type.
- Step, start & end value must be **constant**.
- In synthesis, for loops are “**unrolled**”, and then synthesized.

```
always@( a or b )begin
    for( i=0; i<4; i=i+1 ) c[i] = a[i] & b[i];
end
```

```
always@( a or b ) begin
    c[0] = a[0] & b[0];
    c[1] = a[1] & b[1];
    c[2] = a[2] & b[2];
    c[3] = a[3] & b[3];
end
```


FOR Loop in Hardware

- Equivalent to repeat multiple hardware units

```
int matrix[9];  
for(idx=0;idx<9;idx++){  
    matrix[idx] = 0;  
} //C code
```

```
reg[31:0]matrix[8:0];  
always@(posedge clk)begin  
    if(reset)  
        for(idx=0; idx <9; idx = idx +1)begin  
            matrix[idx] <= 0;  
        end  
end //Verilog code
```

Reset a matrix to zero




```
always@(posedge clk)begin  
    matrix[0] <= 0;  
    matrix[1] <= 0;  
    matrix[2] <= 0;  
    matrix[3] <= 0;  
    matrix[4] <= 0;  
    matrix[5] <= 0;  
    matrix[6] <= 0;  
    matrix[7] <= 0;  
    matrix[8] <= 0;  
end
```

Add 1 to 10 with for loop

- A wrong example
 - Iteration in a Verilog “for” loop is not what you think as in C
 - Equivalent to expand the for loop

```
reg [7:0] temp;  
always@(posedge clk)begin  
    for(idx=0;idx<10;idx=idx+1)begin  
        temp <= temp+idx;  
    end  
end
```



```
always@(posedge clk)begin  
    temp <= temp+0;  
    temp <= temp+1;  
    temp <= temp+2;  
    temp <= temp+3;  
    .  
    .  
    .  
    temp <= temp+9;  
end
```

在verilog使用for loop時，他會把你的for做展開，

Add 1 to 10 with for loop

- Use counter instead
- Note.
 - When you use for loop
 - Try to expand it to see if it matches what you think or not

要用for loop 請先展開看看合不合

```
reg[3:0]counter, temp;
```

```
assign counter_nxt = (counter == 10) ?  
counter : counter + 1;
```

```
always@(posedge clk)begin  
    if(reset)  
        counter <= 0;  
    else  
        counter <= counter_nxt;  
end
```

```
assign temp_nxt = temp + counter;  
always@(posedge clk)begin  
    if(reset)  
        temp <= 0;  
    else if(counter<10)  
        temp <= temp_nxt;  
end
```

- Converting A Software-Style For Loop to VHDL/Verilog

```
// Example Software Code:
```

```
For (int i=0; i<10; i++)  
    data[i] = data[i] + 1;
```

```
//equivalent code in Verilog:
```

```
always @(posedge clock)  
    begin  
        if (index < 10)  
            begin  
                data[index] <= data[index] + 1;  
                index      <= index + 1;  
            end  
        end  
    end
```

```

module for_loop_synthesis (i_Clock);
    input i_Clock;
    integer ii=0;
    reg [3:0] r_Shift_With_For = 4'h1;
    reg [3:0] r_Shift_Regular = 4'h1;

    // Performs a shift left using a for loop
    always @(posedge i_Clock)
        begin
            for(ii=0; ii<3; ii=ii+1)
                r_Shift_With_For[ii+1] <= r_Shift_With_For[ii];
            end

    // Performs a shift left using regular statements
    always @(posedge i_Clock)
        begin
            r_Shift_Regular[1] <= r_Shift_Regular[0];
            r_Shift_Regular[2] <= r_Shift_Regular[1];
            r_Shift_Regular[3] <= r_Shift_Regular[2];
        end
    endmodule

```

```

module for_loop_synthesis_tb ();    //
Testbench
    reg r_Clock = 1'b0;
    // Instantiate the Unit Under Test (UUT)
    for_loop_synthesis UUT
    (.i_Clock(r_Clock));
    always
        #10 r_Clock = !r_Clock;
endmodule

```


Bit Reversal

- Given a 100-bit input vector [99:0], reverse its bit ordering

```
module top_module (  
    input [99:0] in,  
    output reg [99:0] out  
);  
  
    always @(*) begin  
        for (int i=0;i<$bits(out);i++) // $bits() is a system function that returns the width of a signal  
            out[i] = in[$bits(out)-i-1]; // $bits(out) is 100 because out is 100 bits wide.  
        end  
    endmodule
```

population count

- A "population count" circuit counts the number of '1's in an input vector. Build a population count circuit for a 255-bit input vector

```
module top_module (  
    input [254:0] in,  
    output reg [7:0] out  
);  
  
    always @(*) begin // Combinational always block  
        out = 0;  
        for (int i=0;i<255;i++)  
            out = out + in[i];  
    end  
  
endmodule
```

Not synthesizable, combinational feedback loop

Improper Loop Use (1/2)

```
input ['N-1:0] a;  
output ['N-1:0] b;  
integer i;  
reg ['N-1:0] b;  
  
always @ (a) begin  
    for (i=0; i<='N-1; i=i+1)  
        b[i] = ~a[i];  
end
```

BAD
bit visit
loop overhead

```
input ['N-1:0] a;  
output ['N-1:0] b;  
  
assign b = ~a;
```

Good
bus visit
parallel evaluation

Improper Loop Use (2/2)

Bus Reversal

```
input [15:0] a;  
output [15:0] b;  
integer i;  
reg [15:0] b;  
  
always @ (a) begin  
    for (i=0; i<=15; i=i+1)  
        b[15 - i] = a[i];  
end
```

BAD
loop overhead

```
input [15:0] a;  
output [15:0] b;  
  
assign b = { a[0], a[1],  
a[2], a[3], a[4], a[5],  
a[6], a[7], a[8], a[9],  
a[10], a[11], a[12],  
a[13], a[14], a[15] };
```

Good
concatenation

For Loop

- It simulates slow
 - from 10X to > 1000X slower than non-for loop versions
- It synthesizes slow
- Memory clear
 - legitimate for loop use in chip design
- Avoid using the for loop whenever possible

loop

- The for loops are supported, with two constraints:
 - The loop index range must be globally **static**, and the loop body **must not contain a wait** statement.
 - Latches are also synthesized whenever a for loop statement does not assign a variable for all possible executions of the for loop and when a variable assigned inside the for loop is not assigned a value before entering the enclosing for loop.
- The while loops are supported, but the loop body must contain at least **one wait** statement.
 - Combinational while loops are supported if the iterative bound is statically determinable. The loop statements with no iteration scheme (infinite loops) are supported, but the loop body must contain at least one wait statement.
- Repeat is not supported in Synopsys Design Compiler

HDL Compiler allows four syntax forms for a for loop

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

A Simple for Loop

```
for (i = 0; i <= 31; i = i + 1) begin
    s[i] = a[i] ^ b[i] ^ carry;
    carry = a[i] & b[i] | a[i] & carry | b[i] & carry;
end
```

Nested for Loop

```
for (i = 6; i >= 0; i = i - 1)
    for (j = 0; j <= i; j = j + 1)
        if (value[j] > value[j+1]) begin
            temp = value[j+1];
            value[j+1] = value[j];
            value[j] = temp;
        end
```

PARAMETERIZED DESIGN

Generate Example (1/2)

```
generate
  genvar i;

  for(i = 0; i <= 7; i = i + 1)
  begin: u
    adder8 add(sum[(i*8)+:8], co[i+1],
              a[(i*8)+:8], b[(i*8)+:8], ci[i]);
  end
endgenerate
```

u[0].add, u[1].add, ..., u[7].add are generated

Verilog-2001

Generate Example (2/2)

```
module multiplier (a, b, product);  
    parameter a_width = 8, b_width = 8;  
    localparam product_width = a_width + b_width;  
    input [a_width-1:0] a;  
    input [b_width-1:0] b;  
    output [product_width-1:0] product;  
  
    generate  
        if ((a_width < 8) || (b_width < 8))  
            CLA_multiplier #(a_width, b_width) u1 (a, b, product);  
        else  
            WALLACE_multiplier #(a_width, b_width) u1 (a, b, product);  
        endgenerate  
  
endmodule
```

Verilog-2001

Ripple Adder Generator

Parameters give us a way to generalize our designs. A module becomes a "generator" for different variations. Enables design/module reuse. Can simplify testing.

```
module Adder(A, B, R);
```

Declare a parameter with default value.

```
parameter N = 4;
```

Note: this is not a port. Acts like a "synthesis-time" constant.

```
input [N-1:0] A;
```

```
input [N-1:0] B;
```

```
output [N:0] R;
```

```
wire [N:0] C;
```

Replace all occurrences of "4" with "N".

variable exists only in the specification - not in the final circuit.

Keyword that denotes synthesis-time operations

```
genvar i;
```

For-loop creates instances (with unique names)

```
generate
```

```
for (i=0; i<N; i=i+1) begin:bit
```

```
FullAdder add(.a(A[i]), .b(B[i]), .ci(C[i]), .co(C[i+1]), .r(R[i]));
```

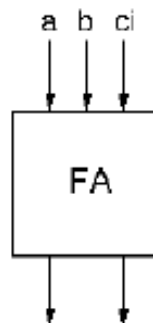
```
end
```

```
endgenerate
```

```
assign C[0] = 1'b0;
```

```
assign R[N] = C[N];
```

```
endmodule
```



```
Adder adder4 ( ... );
```

```
Adder #(.N(64))  
adder64 ( ... );
```

Overwrite parameter N at instantiation.

Generate Loop

Permits variable declarations, modules, user defined primitives, gate primitives, continuous assignments, initial blocks and always blocks to be instantiated multiple times using a for-loop.

```
//Gray-code to binary-code converter
```

```
module gray2bin1 (bin, gray);
```

```
    parameter SIZE = 8;
```

```
    output [SIZE-1:0] bin;
```

```
    input  [SIZE-1:0] gray;
```

```
    genvar i;
```

```
    generate for (i=0; i<SIZE; i=i+1) begin:bit
```

```
        assign bin[i] = ^gray[SIZE-1:i];
```

```
    end endgenerate
```

```
endmodule
```

variable exists only in
the specification - not in
the final circuit.

Keywords that denotes
synthesis-time operations

For-loop creates instances
of assignments
Loop must have
constant bounds

generate if-else-if based on an expression that is deterministic at the time the design is synthesized.

generate case : selecting case expression must be deterministic at the time the design is synthesized.

Generate

- Use **for** loops to generate any number of instances of:
 - modules, primitives, procedures, continuous assignments, tasks, functions, variables, nets
- Use **if-else** and **case** decisions to control what instances are generated
 - provides greater control than the VHDL generate
- New reserved words added:
 - **generate, endgenerate, genvar**

Exercise

- <https://hdlbits.01xz.net/wiki/Vector100r>

▼ More Verilog Features

- ☐ Conditional ternary operator
- ☐ Reduction operators
- ☐ Reduction: Even wider gates
- ☐ **Combinational for-loop:
Vector reversal 2**
- ☐ Combinational for-loop: 255-bit population count
- ☐ Generate for-loop: 100-bit binary adder 2
- ☐ Generate for-loop: 100-digit BCD adder