

硬體描述語言

Chapter 03



# Behavioral Descriptions

Ren-Der Chen (陳仁德)

Department of Computer Science and  
Information Engineering

National Changhua University of Education

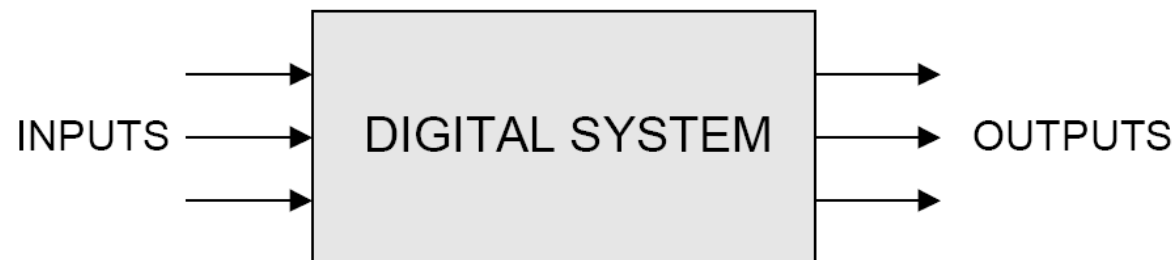
E-mail: [rdchen@cc.ncue.edu.tw](mailto:rdchen@cc.ncue.edu.tw)

Fall, 2024

# Behavioral Modeling (1/3)

---

- Behavioral description models input-output behavior, focusing on <sup>功能性</sup> functionality, not gate-level detail.
- The description is then <sup>合成</sup> synthesized into a physical technology, such as ASIC standard cell library, or a library of <sup>可程式化</sup> programmable parts (FPGAs).
- ***Not all descriptions can be synthesized, and not all synthesized descriptions are desirable.***



# Behavioral Modeling (2/3)

---

- Verilog accommodates mixed levels of abstraction: **structural** and **behavioral**.
- Three kinds of abstract (non-structural) behaviors

## Continuous assignment

- Implicit combinational logic by **static bindings** between expressions and target nets.

## initial behavior

- One-shot sequential activity flow

## always behavior

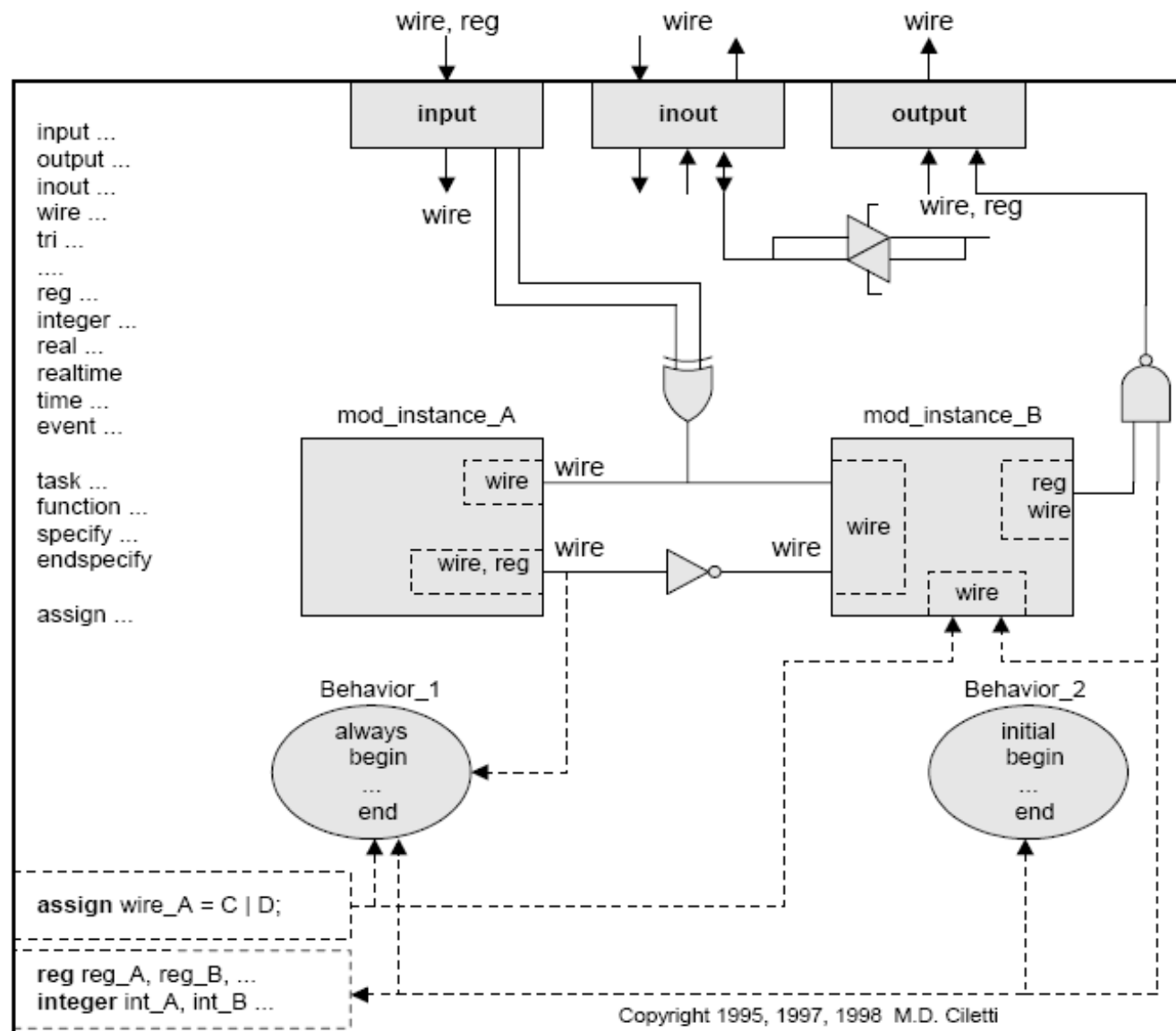
- Cyclic sequential activity flow

## Behavioral Modeling (3/3)

---

- Continuous assignments and primitives respond to inputs and produce outputs according to a functional specification, possible with propagation delay.
- When triggered by events at their inputs, they update the value of their output.
- A behavior only assigns value to a register variable when an assignment statement executes in the activity flow.
- A statement is not sensitive to the activity in the circuit.

# Module Elements



# Verilog Behaviors (1/5)

---

- Two kinds of behaviors: **initial** and **always**.
- An **initial** behavior describes a one-shot activity flow activated at  $t_{\text{sim}} = 0$ .
- It expires after all the procedural statements have executed.
- Designers use **initial** behavior to initialize a simulation and create stimulus waveforms for a simulation testbench.

```
module demo_1 (sig_a)
  output sig_a;
  reg    sig_a;

  initial
    sig_a = 0;
endmodule
```

## Verilog Behaviors (2/5)

---

- An **always** behavior is a cyclic activity flow activated at  $t_{\text{sim}} = 0$ .
- The procedural statements will **re-execute** after the last statement has executed.
- Re-execution continues **indefinitely** until the simulation is terminated.
- The procedural statements execute **sequentially** in the order they are listed, subject to **timing control** and **flow control constructs**.

## Verilog Behaviors (3/5)

---

- A behavior is declared by **initial** or **always**, followed by a single statement or a **begin ... end** block statement.
- Within a block statement, the individual procedural statements execute **sequentially**.
- Block statements may be nested within other blocks.



## Verilog Behaviors (4/5)

---

- Behaviors can be viewed as encapsulating **abstract activity-generators**.
- These activity generators exist **concurrently** with any gate instantiations in the same module.
- Gate effect changes only in the value of **net** variables.
- Behavioral statement primary effect changes in the value of **register** variables.

## Verilog Behaviors (5/5)

---

- A module may contain **any number** of behaviors, but they **may not** be nested.
- Behaviors describe hardware **implicitly**, having no explicit association with physical gates.
- Designers can focus on **functionality**, independent of hardware.
- In simulation, behaviors interact and execute **concurrently** with other behaviors and **instantiations** (e.g., primitives) within a module.

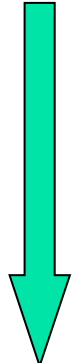
# One-Shot vs. Cyclic

---

- Single statement


 **initial**  
// Procedural statement;

- Block statement

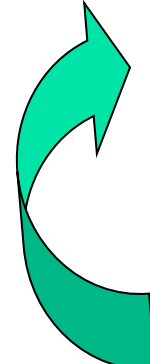
**initial**  
  
 **begin**  
// Procedural statements;  
  
...  
**end**

One-shot behavior

- Single statement

 **always** // @ (event\_ctrl\_exp)  
// Procedural statement;

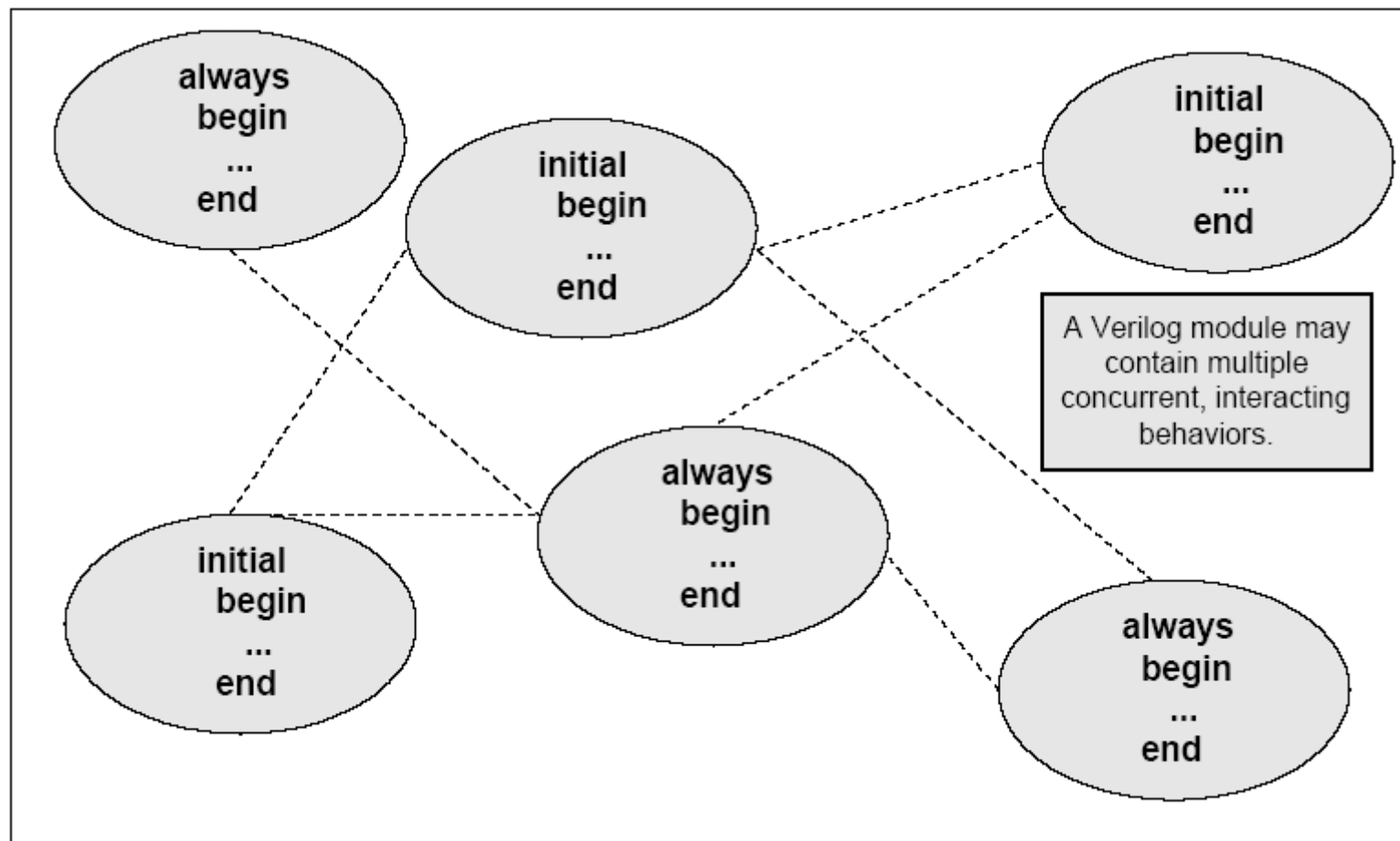
- Block statement

**always** // @ (event\_ctrl\_exp)  
  
 **begin**  
// Procedural statements;  
  
...  
**end**

Cyclic behavior

# Multiple Concurrent Processes

---



# Behavioral Statements (1/3)

---

```
module demo_2a (sig_a, ...);  
output sig_a, sig_b, sig_c, sig_d;  
reg    sig_a, sig_b, sig_c, sig_d;
```

**initial**

**begin**

sig\_a = 0;

sig\_b = 1;

sig\_c = 1;

sig\_d = 0;

**end**

**endmodule**

```
module demo_2b (sig_a, ...);  
output sig_a, sig_b, sig_c, sig_d;  
reg    sig_a, sig_b, sig_c, sig_d;  
// Needless overhead for simulation!
```

**initial**

sig\_a = 0;

**initial**

sig\_b = 1;

**initial**

sig\_c = 1;

**initial**

sig\_d = 0;

**endmodule**

## Behavioral Statements (2/3)

---

- **Procedural statements** support **sequential computations** that manipulate the values of data objects in **memory**.
- They also implicitly govern the activity flow of a simulation by **scheduling** the events in the event queue of the simulator.
- **Simulation time** does not correspond to real time.
- It is used to schedule and manage the **relative time** between events on data objects.

## Behavioral Statements (3/3)

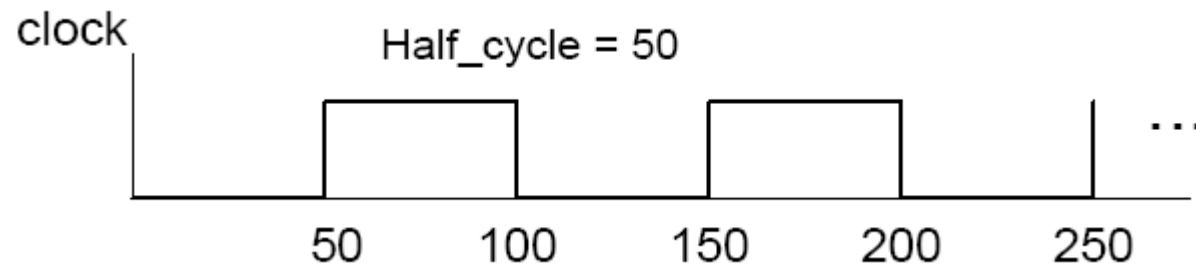
---

- When a procedural assignment statement executes, a value is *immediately* assigned to a register variable.
- In event-driven simulation, more than one assignment may be made *at the same simulation time*, ordered with respect to each other.

# Clock Generator

---

```
module clock_gen1 (clock);  
parameter      Half_cycle = 50;  
parameter      Max_time = 1000;  
output         clock;  
reg            clock;  
  
initial  
    clock = 0;  
  
always  
begin  
    # Half_cycle clock = ~clock;  
end  
  
initial  
    # Max_time $finish;  
endmodule
```





# Procedural Assignment (1/2)

---

- Procedural assignment
  - A statement that assigns value to a **register** variable.
- Two kinds of procedural assignments

## Blocking assignment

- Using operator “ = ”

無阻擋  
(同時執行)

## Non-blocking assignment

- Using operator “ <= ”

## Procedural Assignment (2/2)

---

- When the input to a primitive or continuous assignment statement changes, the output is **evaluated** and possibly **scheduled to change** in the future.
- Register variables can only get value when a procedural statement **executes**, i.e., when **control** is passed to it.
- When the statement executes, the value is assigned to the register variable **immediately**.

# Nets and Registers

---

- A **register** variable can be referenced anywhere in a module.
- A register variable can be assigned value only within a **procedural statement**, **task**, or **function**.
- A register variable cannot be an input or inout port of a module.
- A **net** variable can be referenced anywhere in a module.
- A net variable may not be assigned value within a behavior, task, or function.
- A net variable within a module must be driven by a **primitive**, **continuous assignment**, or **module port**.

# Procedural Continuous Assignment

---

- A *procedural continuous assignment (PCA)* is a continuous assignment made within a *behavior*.
- A continuous assignment establishes a static binding of an RHS expression to a LHS net variable.
- A PCA creates a dynamic binding to a variable.
- Two types of PCA
  - **assign ... deassign**: for register variable
  - **force ... release**: for register or net variable

## “assign ... deassign” PCA

---

- Similar to **continuous assignment** made to a **net**, but *its binding can be removed*.
- Used to model **level-sensitive** behavior of combinational logic, transparent latches, and asynchronous control of sequential parts.
- *While a PCA is in effect, it overrides all procedural assignments to the target variables.*
- *May not be supported by synthesis tools.*

# Multiplexer


```
module mux_PCA (a, b, c, d, select, y_out);
```

```
input  a, b, c, d;
```

```
input  [1:0] select;
```

```
output y_out;
```

```
reg    y_out;
```

 The binding created by a PCA remains effective until a subsequent binding overrides it, or until a “**deassign**” is executed.

```
always @ (select) 
```

```
    if (select == 0) assign y_out = a; else
```

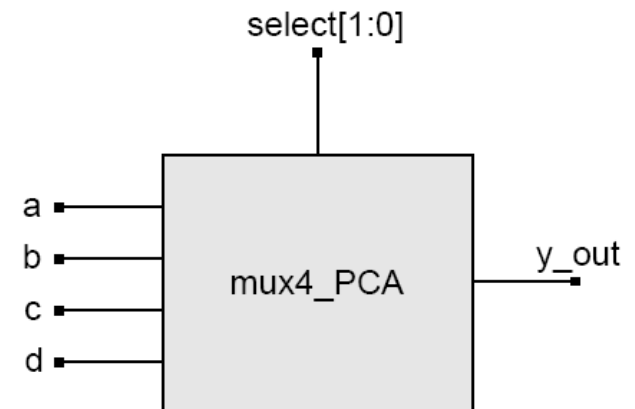
```
    if (select == 1) assign y_out = b; else
```

```
    if (select == 2) assign y_out = c; else
```

```
    if (select == 3) assign y_out = d; else
```

```
        assign y_out = 1'bx;
```

```
endmodule
```



# Asynchronous Override

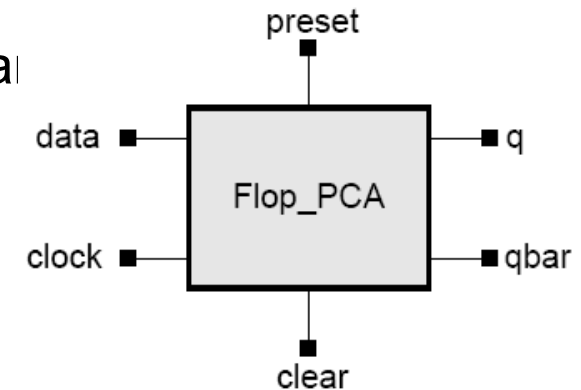
```
module flop_PCA (preset, clear, clock, data, q, qbar);
input  preset, clear, clock, data;
output q, qbar;
reg    q;
assign qbar = ~q;
always @ (negedge clock)
    q = data;
always @ (clear or preset)
begin
    if (!clear) assign q = 0;
    else if (!preset) assign q = 1;
    else deassign q;
end
endmodule
```

Continuous  
assignment

~~✎~~ While PCA is in effect, it overrides all procedural assignments to the target variable.

Synchronous procedural  
assignment

Asynchronous procedural continuous  
assignment overrides synchronous  
procedural assignments.



# “assign” vs. “assign ... deassign”

---

## ■ Continuous assignment (**assign**)

- Applied to net variables (made outside a behavior).
- Binding cannot be removed (static binding).
- Remains in effect for the duration of simulation.

## ■ Procedural continuous assignment (**assign ... deassign**)

- Applied to register variables (made within a behavior).
- Binding can be removed (dynamic binding).
- Remains in effect until another procedural continuous assignment is made to the same target register variable, or until a **deassign** statement is made.
- May not be supported by synthesis tools.



## “force ... release” PCA

---

- Applied to **net** and **register** variables.
- When the **force** is made to a net, the expression assigned to the target net overrides all other drivers of the net until the **release** is executed.
- It can override a **primitive** driver and **continuous assignment** driver of a net.
- It can also override a **procedural assignment** and an **assign ... deassign** PCA to a register.
- *It is used primarily with hierarchical de-referencing in testbenches, not supported by synthesis tools.*

# Example

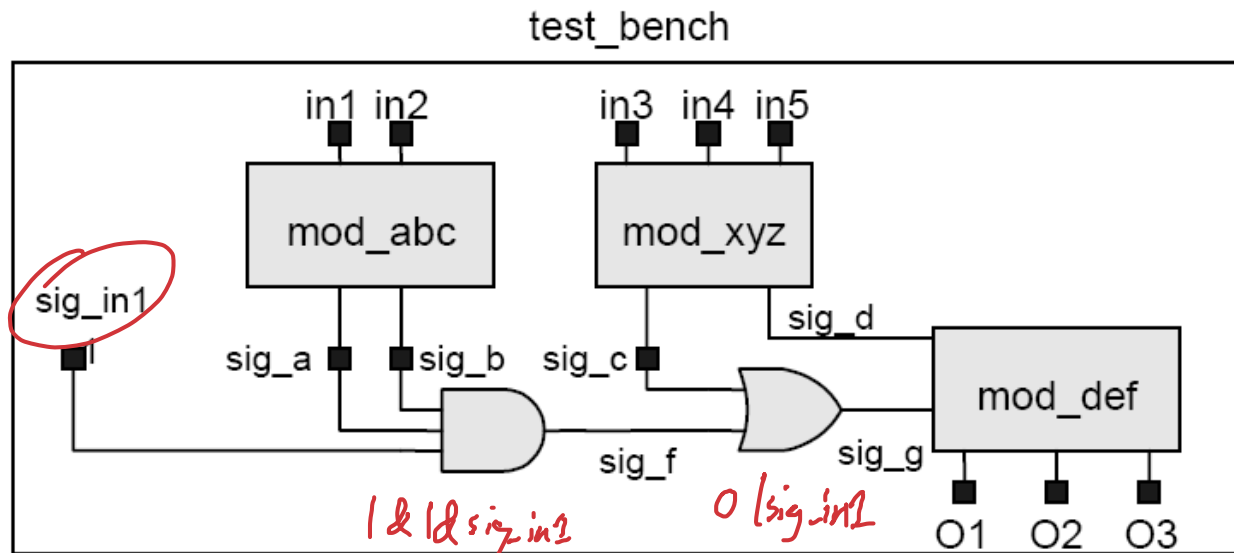
強制  
改值  
(內部信號)

```
force sig_a = 1;
force sig_b = 1;
force sig_c = 0;
sig_in1 = 0;
#5 sig_in1 = 1;
#5 sig_in1 = 0;
```


...

釋放

```
release sig_a;
release sig_b;
release sig_c;
```



這裏 sig\_in1 可以連入 mod\_def

 The **force** ... **release** PCA helps sensitize the path from sig\_in1 to sig\_g through the gates.

# Modes of Assignments to Net and Register Variables

---

Variable Type	Mode of Assignment				
	Output of Primitive	Continuous Assignment	Procedural Assignment	<b>assign ... deassign PCA</b>	<b>force ... release PCA</b>
Net	Yes	Yes	No	No	Yes
Register	Comb - No Seq - Yes	No	Yes	Yes	Yes

# Procedural Timing Controls and Synchronization

---

- Four mechanisms provide explicit control over the time of execution of a procedural statement
- Delay control #
- Event control @
- Named events *event*
- **wait** construct

## Delay Control Operator (#)

---

- Temporarily suspends the activity flow within a behavior by postponing the execution of a procedural statement.
- Two forms of delay control, depending on the **placement of the operator (#)** in a procedural assignment.
- If “#” precedes an assignment statement, it blocks execution of the statement that follows it, and *also affects all subsequent procedural statements*.
- A blocked statement must execute before the statements that follow it can execute.

# Example 1

```
#10 reg_sum = reg_a + reg_b;
```

```
#((delay1 + delay2)/2) reg_e = reg_m;
```

```
#a_register target = target + 1;
```

**initial**

**begin**

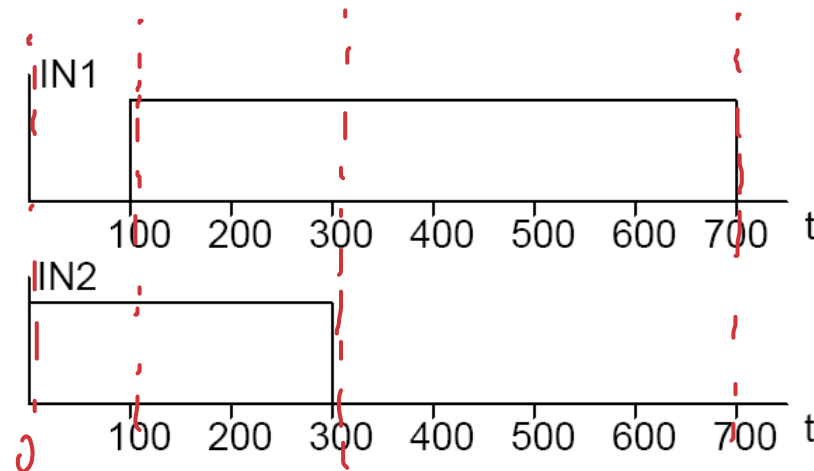
#0 IN1 = 0; IN2 = 1;


#100 IN1 = 1;

#200 IN2 = 0;

#400 IN1 = 0;

**end**



 The effect of using “#0” is to cause the statement to execute at the end of the current simulation cycle.

## Example 2

```
module simple_clock (clock_1, clock_2);  
output clock_1, clock_2;  
reg    clock_1, clock_2;
```

```
  always
```

```
begin
```

```
  #0    clock_1 = 0; clock_2 = 1;
```

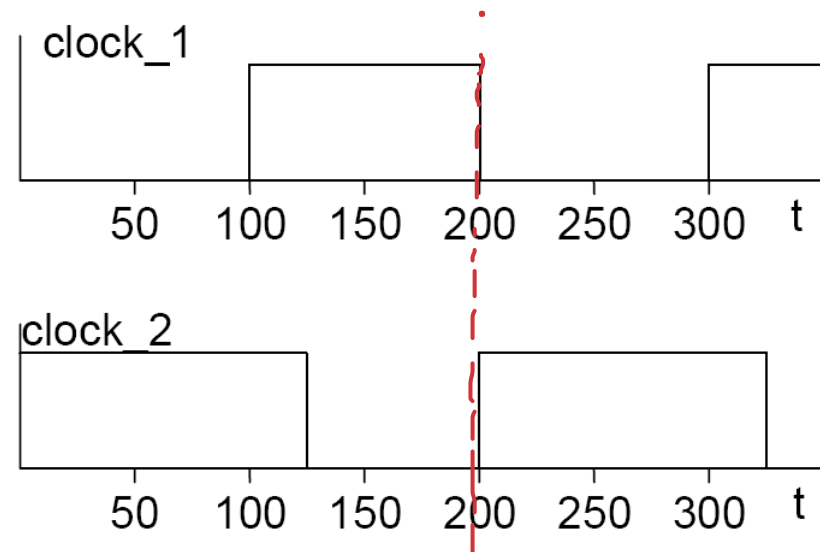
```
  #100  clock_1 = 1;
```

```
  #25   clock_2 = 0;
```

```
  #75;
```

```
end
```

```
endmodule
```

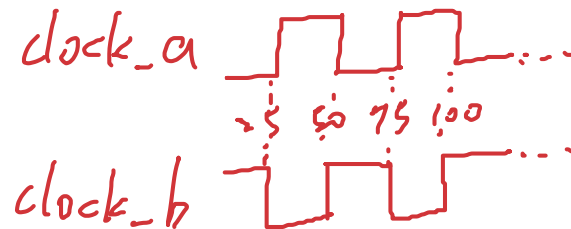


反  
器  
输  
入  
行

200

## Example 3

```
module clock_2phase (clock_a, clock_b);  
output clock_a, clock_b;  
reg          clock_a, clock_b;  
parameter    half_clock_period = 25;  
initial begin  
    clock_a = 0;  
    clock_b = 1;  
end  
always begin  
    #half_clock_period;  
    clock_a = ~clock_a;  
    clock_b = ~clock_b;  
end  
endmodule
```





# Two Types of Placement for Delay Operator (#)

---

先停再計算 ■ **#10** reg\_sum = reg\_a + reg\_b;

- After 10 time units, the sum of reg\_a and reg\_b is calculated and assigned to reg\_sum.

先計算再停 ■ reg\_sum = **#10** (reg\_a + reg\_b);

- 之後給值
- The sum of reg\_a and reg\_b is first calculated, and then after 10 time units, assigned to reg\_sum.

暫停(無時間)

## Event Control Operator (@) (1/2)

---

- Synchronize execution of a procedural statement (or a block of procedural statements) to a change in the value of either an **identifier** or an **expression**.
  - @ (event\_identifier) statement;
  - @ (event\_expression) statement;
- When the activity flow reaches “@”, the flow is **suspended** and the *event\_identifier* or *event\_expression* is monitored by the simulator to detect an **event** (a change in value).

事件發生

## Event Control Operator (@) (2/2)

- Given that event control suspends the host process until an event has occurred, the anticipated event must be caused by activity *in some process* other than the suspended one.

- **posedge** transitions

↑ 正源觸發 ●  $0 \rightarrow 1$ ,  $0 \rightarrow x$ , and  $x \rightarrow 1$

- **negedge** transitions

↓ 負 .. ●  $1 \rightarrow 0$ ,  $1 \rightarrow x$ , and  $x \rightarrow 0$

# Example 1

---

- If event\_A has occurred, but not event\_B, the behavior will be suspended at @ (event\_B).
- If event\_A occurs a second time while the process is suspended, *it will be ignored*.
- If the activity flow is suspended at @ (event\_A), the occurrence of event\_B *will be ignored*.

```
...  
@ (event_A) 停下, 事件A發生才繼續  
begin  
    ...  
    @ (event_B) 停下, 事件B...  
    begin  
        ...  
    end  
end
```

## Example 2

---

@ sig\_1 等待 sig\_1 發生變化,  $\uparrow$  or  $\downarrow$   
reg\_A = reg\_B;

always @ (posedge<sup>↑</sup>clock)

{cell\_B, cell\_A} = {cell\_A, cell\_B};

每一次都要等待!

等待  $\xrightarrow{\text{觸發}}$  執行  $\rightarrow$  等待  $\xrightarrow{\text{觸發}}$  執行  $\rightarrow$  等待...  
10ns 10ns

always @ (negedge clock)

cell\_G = data\_bus [3:0];

always @ (posedge clock)

#10 q = data\_in;

# D Flip-Flop

```
module df_behav (data, clk, set, reset, q, q_bar);
```

```
    input      data, clk, set, reset;
```

```
    output     q, q_bar;
```

```
    reg q;  内部給値
```

```
    assign    q_bar = ~q;
```

```
    always @ (posedge clk) begin
```

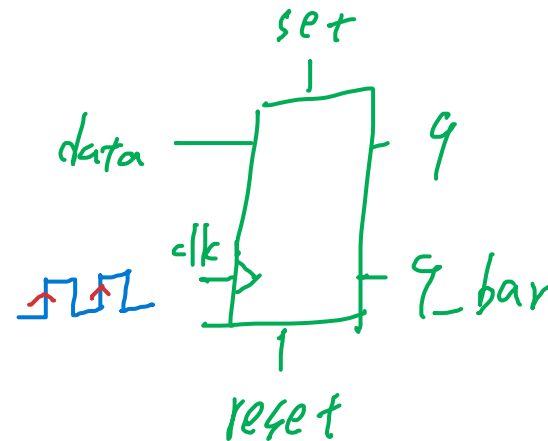
```
        if (reset == 0) q = 0;    // synchronous reset
```

```
        else if (set == 0) q = 1; // synchronous set
```

```
        else q = data;
```

```
    end
```

```
endmodule
```



同歩

# Event “or” (1/2)


```
module asynch_df_behav1 (data, clk, set, reset, q, q_bar);
```

```
  input      data, clk, set, reset;
```

```
  output     q, q_bar;
```

```
  reg        q;
```

```
  assign      q_bar = ~q;
```

 This model does **not** describe the functionality of a flip-flop correctly.

```
  always @ (set or reset or posedge clk) begin
```

```
    if (reset == 0) q = 0;
```

```
    else if (set == 0) q = 1;
```

```
    else if (clk == 1) q = data;
```

```
  end
```

```
endmodule
```

錯誤

set: 0 → 1  
reset: 0 → 1

## Event “or” (2/2)

---

```
module asynch_df_behav2 (data, clk, set, reset, q, q_bar);
  input      data, clk, set, reset;
  output     q, q_bar;
  reg        q;

  assign      q_bar = ~q;

  always @ (negedge set or negedge reset or posedge clk) begin
    if (reset == 0) q = 0;
    else if (set == 0) q = 1;
    else if (clk == 1) q = data;
  end
endmodule
```

*asynchronous*  
非同步



# Level-Sensitive Behavior

鎖存器  
(存資料)

```

module t_latch (enable, data, q_out);
  input          enable, data;
  output         q_out;
  reg            q_out;
  
```

```

always @ (enable or data) begin
  
```

```

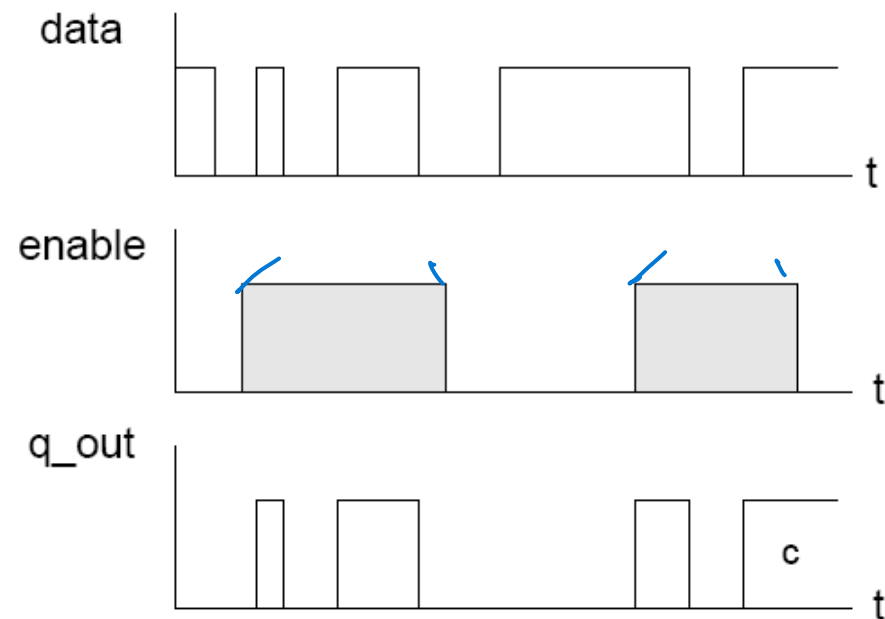
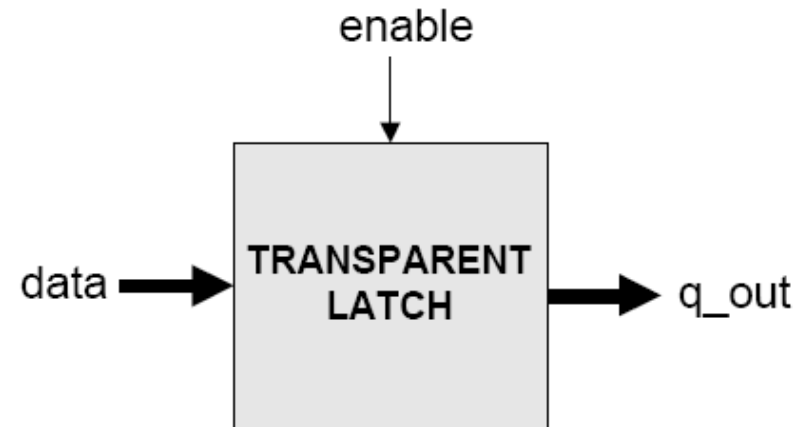
    if (enable) q_out = data;
  
```

```

end
  
```

```

endmodule
  
```



# Named Events

---

- A high-level mechanism of **communication** and **synchronization** within and between modules.
- Support **inter-process** communication without requiring details about physical implementation.
- Signals do not have to be passed between modules explicitly through **ports**.
- It can be referenced within that module directly or in other modules by **hierarchically de-referencing** the name of the event.

# Example 1

---

```
module Demo_mod_A ();  
    event something_happens; // Declaration of an abstract event  
    always begin  
        -> something_happens; // Triggering of an abstract event  
    end  
endmodule
```

```
module Demo_mod_B ();  
    always @ (Top_Module.Demo_mod_A.something_happens) begin  
        // Event monitor  
        // do something when something_happens in Demo_mod_A  
    end  
endmodule
```

## Example 2

```
module flop_event (clock, reset, data, q, q_bar);
```

```
  input      clock, reset, data;
```

```
  output     q, q_bar;
```

```
  reg        q;
```

```
  event
```

```
  up_edge;
```

信號

```
  assign     q_bar = ~q;
```

```
  always @ (posedge clock)
```

觸發

```
    -> up_edge;
```

發出信號

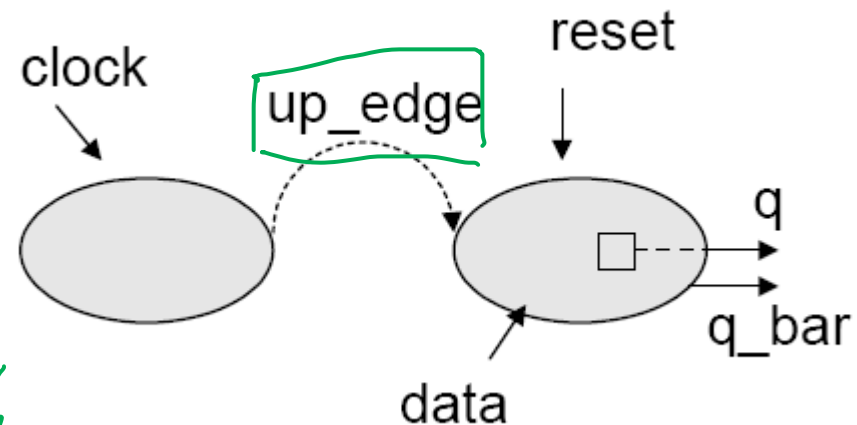
```
  always @ (up_edge or negedge reset) begin
```

```
    if (reset == 0) q = 0;
```

```
    else q = data;
```

```
  end
```

```
endmodule
```



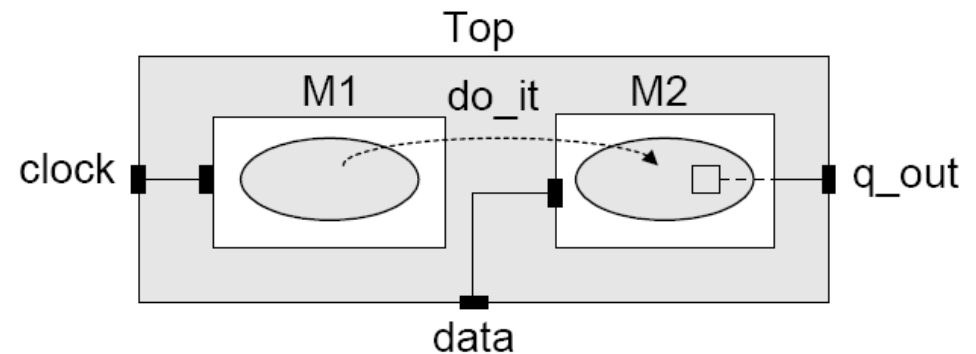
## Example 3

```
module Top (clock, data, q_out);  
    input clock, data;    output q_out;  
    Talker      M1 (clock);  
    Receiver    M2 (data, q_out);  
endmodule
```

```
module Talker (clock);  
    input      clock;  
    ✓ event    do_it;  
    always @ (posedge clock) -> do_it;  
endmodule
```

```
module Receiver (data, q_out);  
    input data;    output q_out;    reg q_out;  
    always @ (Top.M1.do_it) q_out = data;  
endmodule
```

不同module, 找路徑



# “wait” Construct

---

- Models **level-sensitive** behavior by suspending the activity flow in a behavior until an expression is “**TRUE**”.
- Other processes may execute while the process with the **wait** is suspended.
- If the expression evaluates to “TRUE” when the statement is encountered, *the execution is not suspended*.

# Example

---

```
module example_of_wait ( );
```

```
...
```

```
always
```

```
begin
```

等 enable 為 1 { 原本就是 1 → 直接做下去  
原本是 0 → 等到變成 1

```
...
```

```
wait (enable) reg_a = reg_b;
```

```
#10 reg_c = reg_d;
```

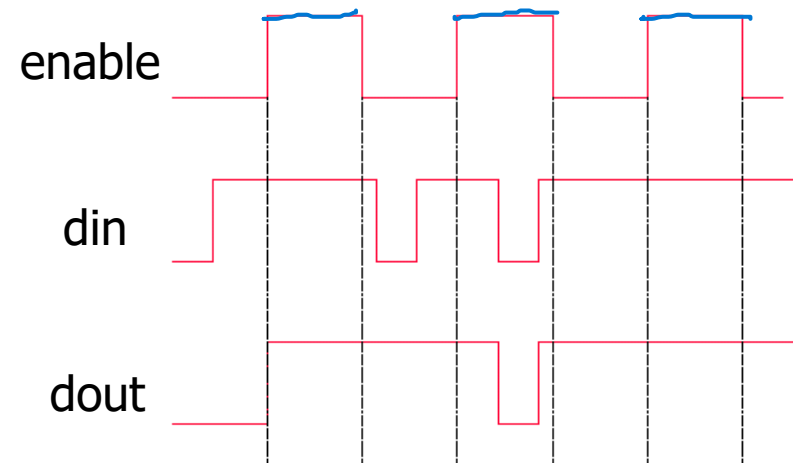
```
...
```

```
end
```

```
endmodule
```

# Level-Sensitive Latch (1/2)

```
module latch_level (din, enable, dout);  
    input          din, enable;  
    output         dout;  
    reg           dout;  
  
    always begin  
        wait (enable) assign dout = din;  
        wait (!enable) deassign dout;  
    end  
endmodule
```



斷開連接



## Level-Sensitive Latch (2/2)

---

```
module latch_level2 (din, enable, dout);  
    input          din, enable;  
    output         dout;  
    reg            dout;  
  
    always @ (din or enable) begin  
        if (enable == 1'b1)  
            dout = din;  
    end  
endmodule
```

# Intra-Assignment Delay for Blocked Assignments (1/2)

---

- When a timing control operator (# or @) appears in front of a **procedural statement**, the delay is referred to as a “**blocking**” delay.
- When it is placed between an assignment operator (“=”) and the RHS, the delay is called an “**intra-assignment**” delay.
- Ordinary delay control postpones *the execution of a statement*.

延迟在中间

## Intra-Assignment Delay for Blocked Assignments (2/2)

- Intra-assignment delay postpones *the occurrence of the assignment* that results from executing a statement.
- The value of the RHS expression when the statement is encountered determines the value assigned to the LHS after the period specified by the intra-assignment delay.
- Referencing and evaluation are separated in time from the actual **assignment** of value to the target register variable.

**#5** A = B + C;

**@ (a\_bus)** G = accum + 1;

1. wait 5ns  
2. B + C  
3. 给 A

A = **#5** B + C;

G = **@ (a\_bus)** accum + 1;

1. B + C  
2. wait 5ns  
3. 给 A

# Example 1

---

a = #10 b;



**begin**

tmp = b;

#10 a = tmp;

**end**

a = @(posedge clk) b;



**begin**

tmp = b;

@(posedge clk) a = tmp;

**end**

a = ~~\*~~repeat (2) @(posedge clk) b;



**begin**

tmp = b;

1. @(posedge clk);

2. @(posedge clk) a = tmp;

**end**

## Example 2

---

a = #2 1;

b = #3 0;

- The first statement is encountered at  $t_{\text{sim}} = 0$ ;
- At time  $t_{\text{sim}} = 2$ , “a” gets 1.
- At time  $t_{\text{sim}} = 2$ , the first statement is complete and the second is encountered.
- At time  $t_{\text{sim}} = 5$ , “b” gets 0.

## Example 3

---

**initial begin**

#0 a = 0;

#5 a = 1;

**end**

**initial begin**

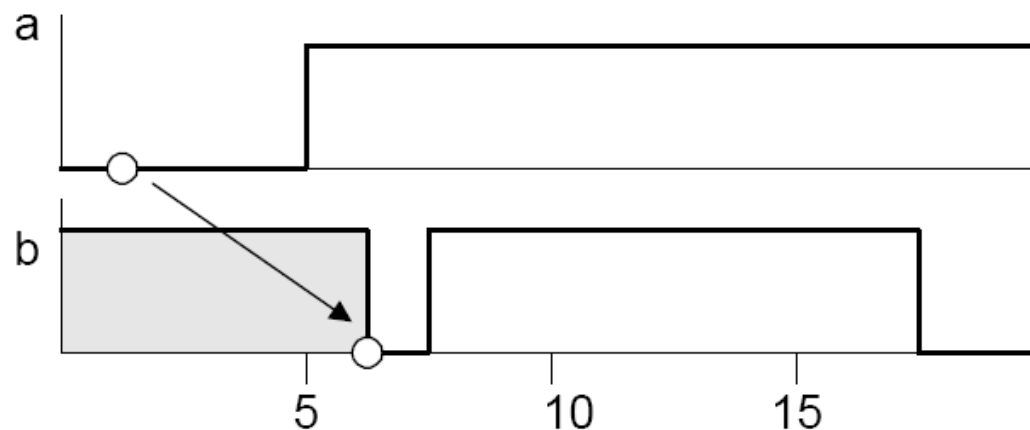
1 #1;

6 b = #5 a;

7 #1 b = 1;

17 #10 b = 0;

**end**

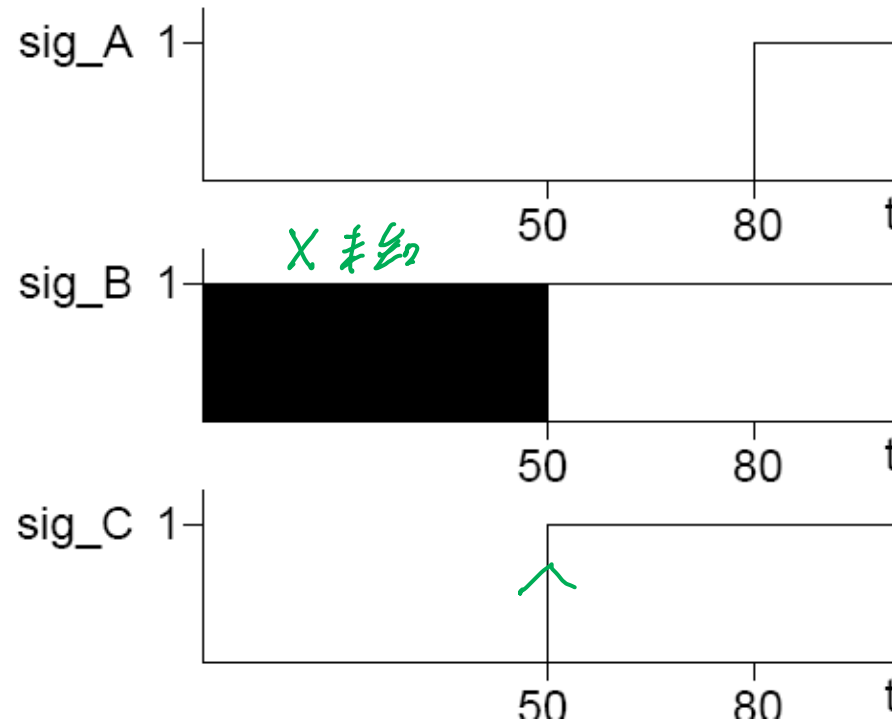


## Example 4

---

```
initial begin
    sig_C = 0;
    #50 sig_C = 1;
end
```

```
initial begin
    sig_A = 0;
    sig_B = @(posedge sig_C) 1;
    #30 sig_A = 1;
end
```



# Non-Blocking Assignments

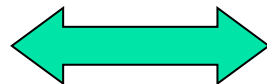
## ■ Blocking procedural assignments

- Use operator “=” and execute sequentially. 程式依序執行
- The statement that follows a procedural assignment cannot execute until the procedural assignment's execution is completed.

## ■ Non-blocking procedural assignments

- Use operator “<=” and execute concurrently. 同時執行
- The execution of the statement that follows a non-blocking assignment will **not** be blocked.

A <= B;  
B <= A;



**Equivalent**

A, B 交換

B <= A;  
A <= B;

A = B;  
B = A; ≠  
B

B = A;  
A = B;  
A



# Two Steps for Executing Non-Blocking Assignments

---

- The RHS expression is evaluated, and then
- The simulator schedules an assignment to the LHS at a time determined by an optional **intra-assignment delay** control or **event** control.

# Example


---

```
A = 1;  
B = 0;  
A = B;  // Uses B = 0  
B = A;  // Uses A = 0  
Result: A = 0, B = 0
```

```
A = 1;  
B = 0;  
B = A;  // Uses A = 1  
A = B;  // Uses B = 1  
Result: A = 1, B = 1
```

```
A = 1;  
B = 0;  
A <= B; // Uses B = 0  
B <= A; // Uses A = 1  
Result: A = 0, B = 1
```

```
A = 1;  
B = 0;  
B <= A; // Uses A = 1  
A <= B; // Uses B = 0  
Result: A = 0, B = 1
```

 The blocked assignments execute first because they are listed first in the order of the statements.

# Conclusions

---

- Non-blocking assignments are useful in modeling **concurrent data transfers** in synchronous circuits.
- The sampling of the RHS of a list of non-blocking assignments occurs **before** any assignments are made.
- The outcome does not depend on the order of the list, *with one exception*.
- If two or more statements assign value to the same register variable, the **last** statement will determine the final value.

X

```
B <= 0;  
B <= 1; // The final value of B is 1.
```

# Intra-Assignment Delay for Non-Blocking Assignment

---

- A delay in non-blocking assignments affects only the *scheduling* of the assignment, *not the completion of execution of the statements*.
- The statements complete execution concurrently at the same step at which they are encountered.

# Blocking Assignment

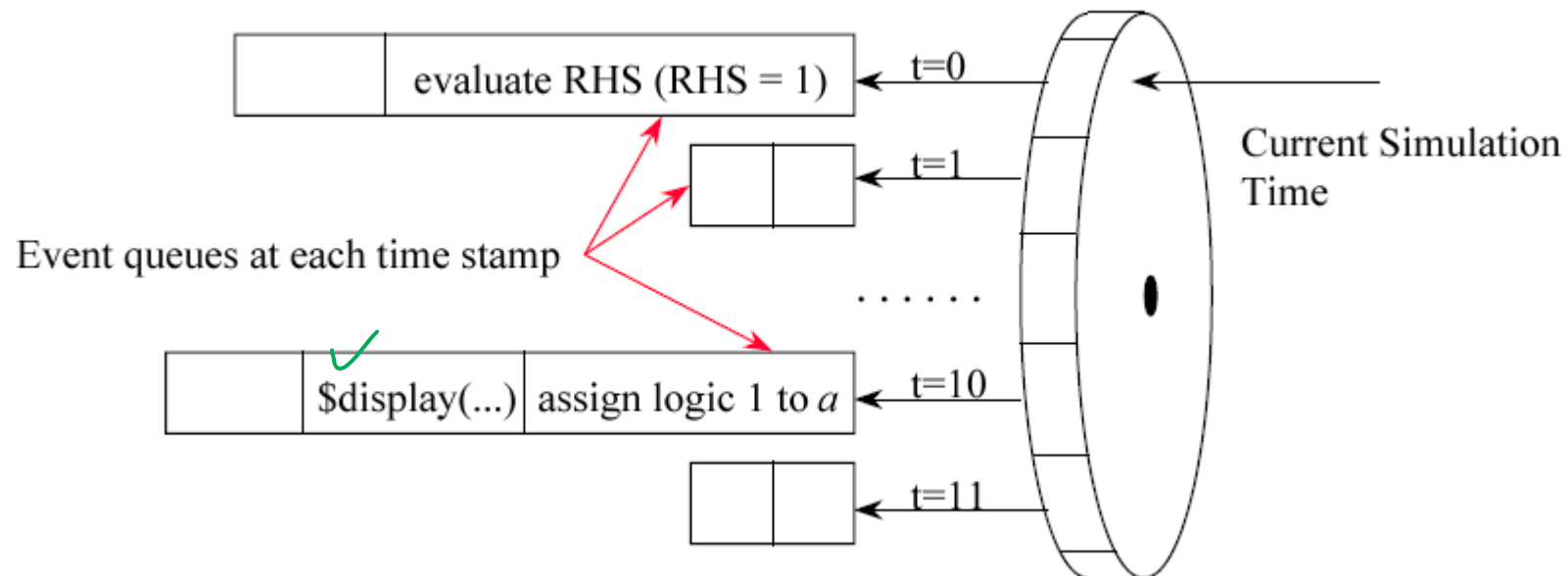
**initial begin**

**a = #10 1;**

**\$display** ("Current time = %t a = %b", \$time, a);

**end**

Result => Current time = 10, a = 1



# Non-Blocking Assignment

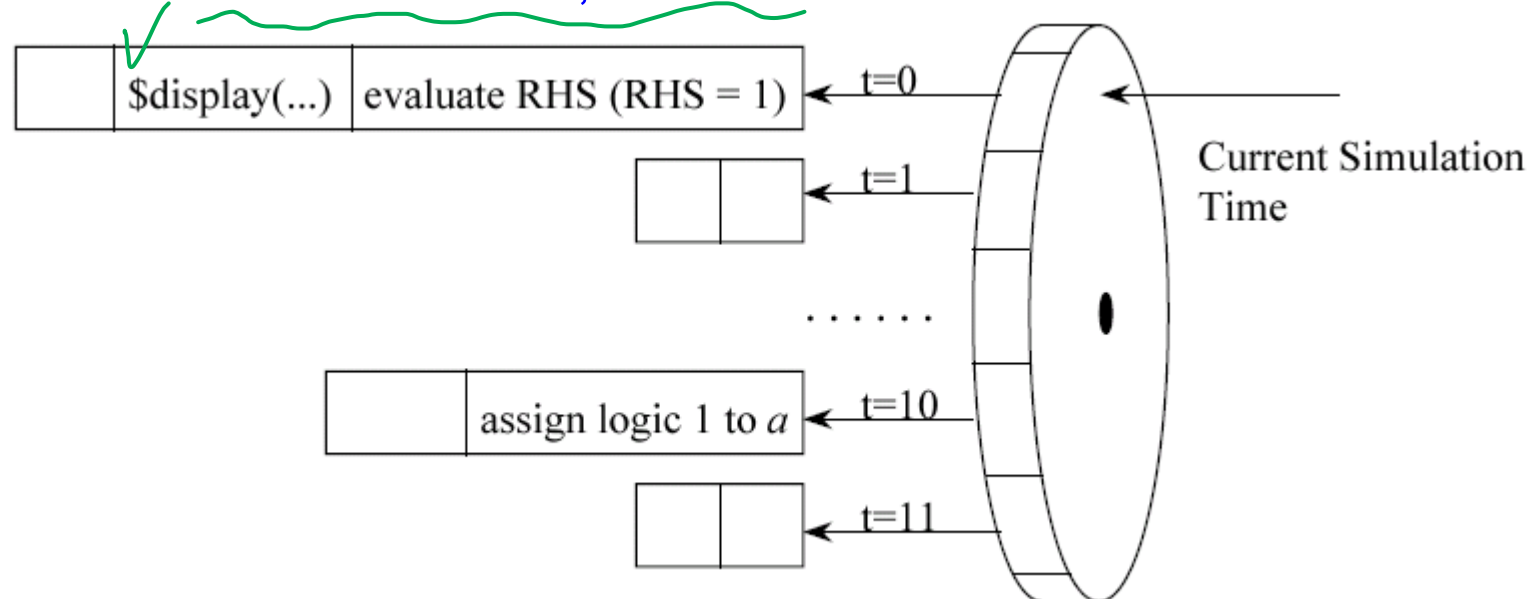
initial begin

`a <= #10 1;`

`$display ("Current time = %t a = %b", $time, a);`

end

Result => Current time = 0, a = x



# Example 1

---

**initial begin**

```
@ (posedge clock) begin
    G = @ (a_bus) accum;
    C = D;
```

**end**

**end**

1. Cycle cannot complete until a\_bus has an active edge.
2. The value that G gets is the value of accum when the statement is encountered in the activity flow. This may differ from the actual value of accum when a\_bus finally has activity.
3. When “posedge clock” occurs, the value of accum is sampled only when “a\_bus” has occurred for previous value of accum.

## Example 2

---

**initial begin**

```
@ (posedge clock) begin
    G <= @ (a_bus) accum;
    C <= D;
end
```

**end**

1. accum is sampled at the rising edge of clock.
2. D is sampled when accum is sampled.
3. C gets the value of D immediately.
4. G gets the value of accum when a\_bus has an activity.
5. Whenever “posedge clock” occurs, the value of accum is sampled.



## Example 3

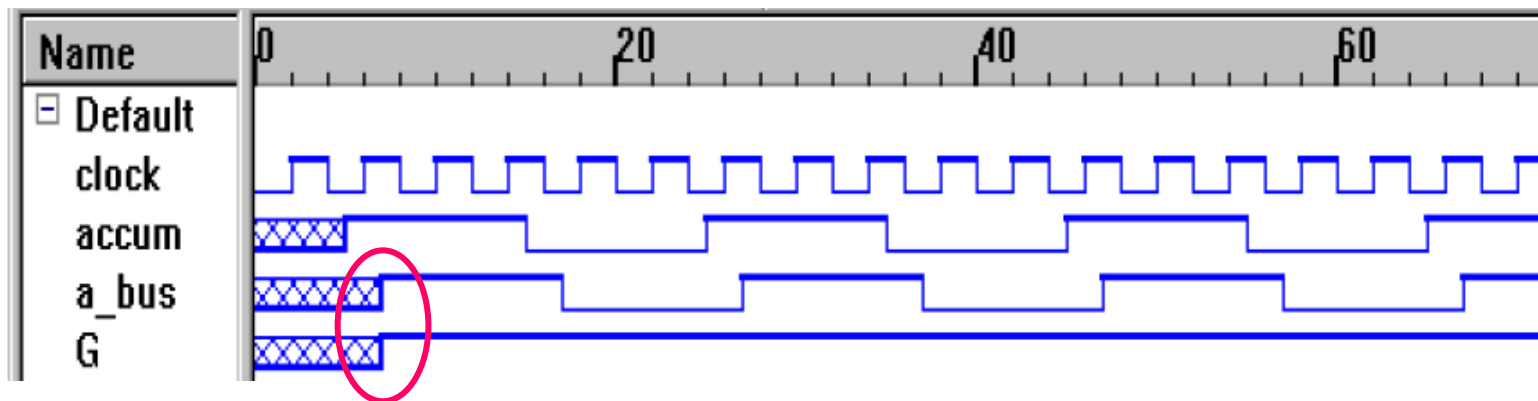
**always begin**

**@ (posedge clock)**

**G <= @ (a\_bus) accum;**

**end**

1. When a\_bus has an active edge, G gets the value that accum held when clock had its last active edge.
2. Whenever “posedge clock” occurs, the value of accum is sampled.



# Blocking vs. Non-Blocking

10  
12  
15 ↓

A = #10 1;  
B = #2 0;  
C = #3 1;

t	A	B	C
0	x	x	x
2	x	x	x
3	x	x	x
10	<b>1</b>	x	x
12	1	<b>0</b>	x
15	1	0	<b>1</b>

D <= #10 1;  
E <= #2 0;  
F <= #3 1;

t	D	E	F
0	x	x	x
2	x	<b>0</b>	x
3	x	0	<b>1</b>
10	<b>1</b>	0	1
12	1	0	1
15	1	0	1

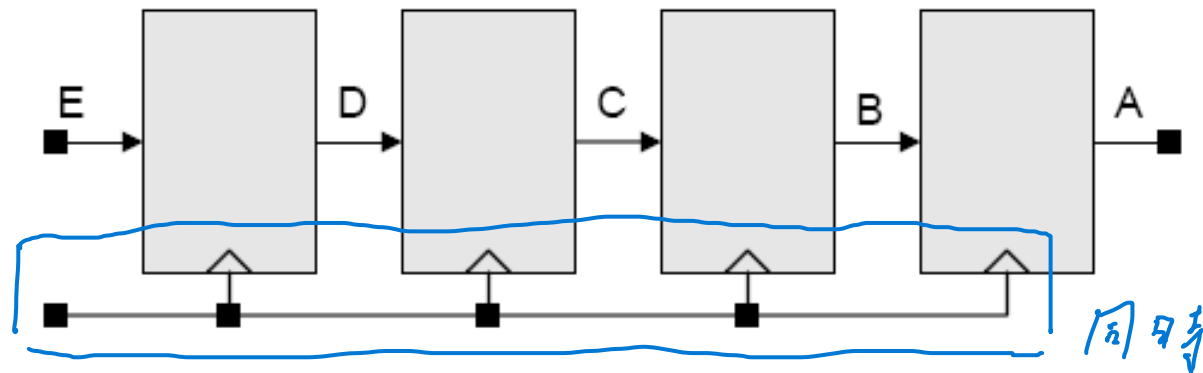
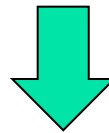
# Example 1

---

A = B;  
B = C;  
C = D;  
D = E;

A <= B;  
B <= C;  
C <= D;  
D <= E;

D <= E;  
C <= D;  
B <= C;  
A <= B;

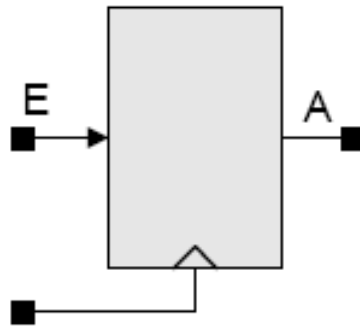
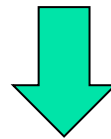


## Example 2

---

D = E;  
C = D;  
B = C;  
A = B;

$\Rightarrow A = E$



# Eight-Bit OR Gates (1/5)

---

*N2.2*

```
module or_gate1 (a, b, y1);  
    input        [7:0] a, b;  
    output       [7:0] y1;  
  
    assign       y1 = a | b;  
endmodule  
// continuous assignment
```

```
module or_gate2 (a, b, y2);  
    input        [7:0] a, b;  
    output       [7:0] y2;  
  
    wire         [7:0] y2 = a | b;  
endmodule  
// continuous assignment
```

## Eight-Bit OR Gates (2/5)

```
module or_gate3 (a, b, y3);
```

```
    input      [7:0] a, b;
```

```
    output     [7:0] y3;
```

```
    reg        [7:0] y3;
```

```
    initial begin
```

```
        assign y3 = a | b;
```

```
    end
```

```
endmodule
```

- 1. One-shot behavior.
- 2. The assignment remains in effect after the behavior expires.
- 3. Though a valid style, but not a preferred style, and will **not** be accepted by a synthesis tool.

```
No. 1 module or_gate4 (a, b, y4);
```

```
    input      [7:0] a, b;
```

```
    output     [7:0] y4;
```

```
    reg        [7:0] y4;
```

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

```
        y4 = a | b;
```

```
    end
```

```
endmodule
```

- 1. Cyclic behavior.
- 2. An acceptable style.

## Eight-Bit OR Gates (3/5)

```
module or_gate5 (a, b, y5);  
    input      [7:0] a, b;  
    output     [7:0] y5;  
    reg        [7:0] y5;  
    always @ (a or b) begin  
        #5 y5 = a | b;  
    end  
endmodule
```

1. The assignment to y is determined by old data.
2. While the delay control is blocking the procedural assignment to y, the event control expression **does not** respond to a or b.

```
module or_gate6 (a, b, y6);  
    input      [7:0] a, b;  
    output     [7:0] y6;  
    reg        [7:0] y6;  
    always @ (a or b) begin  
        y6 = #5 a | b;  
    end  
endmodule
```

1. The assignment is blocked, so the event control expression **cannot** respond to input events until the statement assigning value is executed.

## Eight-Bit OR Gates (4/5)

---

```
module or_gate7 (a, b, y7);
```

```
    input      [7:0] a, b;
```

```
    output     [7:0] y7;
```

```
    reg        [7:0] y7;
```

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

```
        y7 <= #5 a | b;
```

```
    end
```

```
endmodule
```

1. The non-blocking assignment executes and **returns control to the event control operator**, anticipating the next event on the data path, even though y has not yet received a new value from the original event.



# Eight-Bit OR Gates (5/5)



`#5 y5 = a | b; // Delayed sampling of the inputs, inertial delay`  
`y6 = #5 a | b; // Ignores inputs transitions`  
`* y7<=#5 a | b; // Correct, but no inertial delay effect in behaviors,`  
 等待时 always transport delay  
 持续暂判断

## Example (1/3)

```
module test;
```

```
    reg a, b, x1, x2, x3, x4, y1, y2, y3, y4;
```

```
initial begin
```

```
    a = 1;
```

```
    b = 0;
```

```
end
```

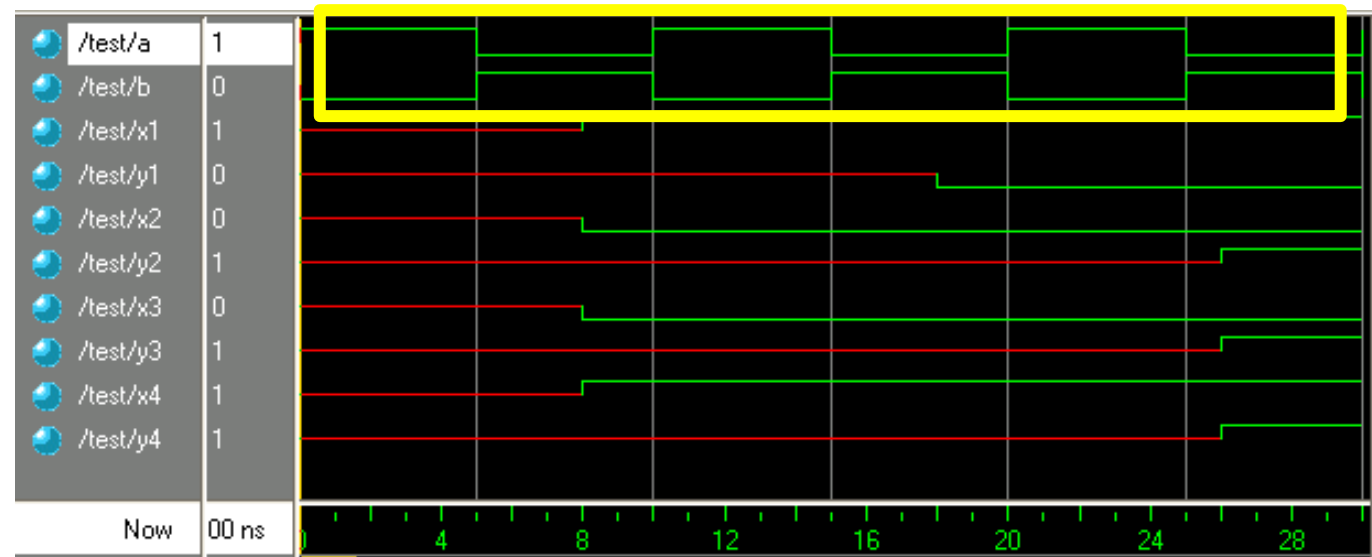
```
always begin
```

```
    #5;
```

```
    a = ~a;
```

```
    b = ~b;
```

```
end
```



## Example (2/3)

**initial begin** *At 0, get a=1* // **Non-blocking assignments**

*At 8, set x1=1* x1 <= #8 a; // x1(t\* + 8) gets a(t\*)

*At 18, set y1=0* y1 <= #18 b; // y1(t\* + 18) gets b(t\*)

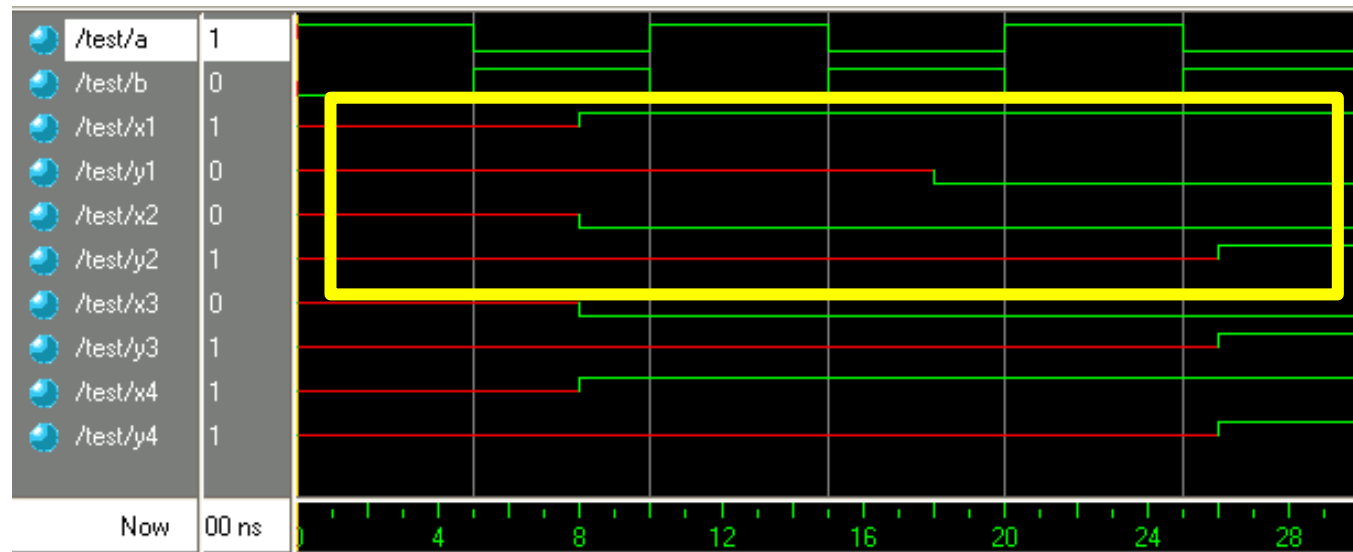
**end** *At 0, get b=0*

**initial begin** // **Blocking assignments**

*8* #8 x2 <= a; // x2(t\* + 8) gets a(t\* + 8)

*26* #18 y2 <= b; // y2(t\* + 26) gets b(t\* + 26)

**end**



## Example (3/3)

initial begin

// **Blocking assignments**

8  
26  
↓  
#8 x3 = a;  
#18 y3 = b;

// x3(t\* + 8) gets a(t\* + 8)

// y3(t\* + 26) gets b(t\* + 26)

end

initial begin

// **Blocking assignments**

At 8, set x4 = 1  
x4 = #8 a;

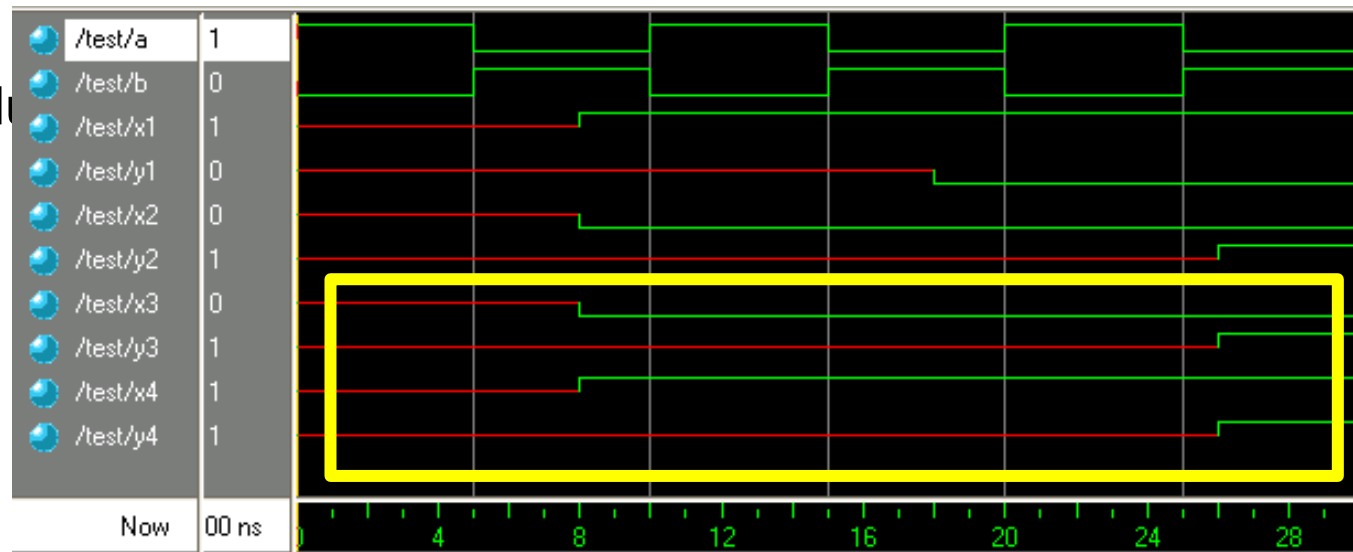
// x4(t\* + 8) gets a(t\*)

At 8, get b = 1  
y4 = #18 b;

// y4(t\* + 26) gets b(t\* + 8)

End

endmode



# Simulation of Simultaneous Procedural Assignments

---

- Multiple behaviors may assign value to the same register variable *at the same time step*.
- A simulator must determine the outcome of these multiple assignments.
- And must also distinguish between blocking and non-blocking assignments.

# Processing Steps

---

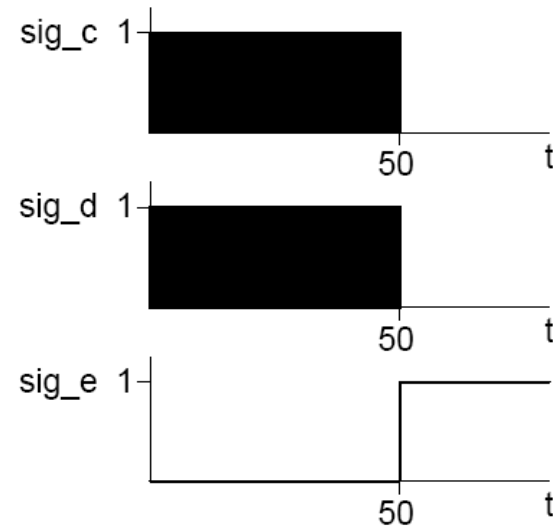
1. Evaluate the RHS expressions of **all the assignments** to register variables encountered *at the same time step*.
2. Execute **blocking assignments** to registers.
3. Execute **non-blocking assignments** that do not have intra-assignment timing controls.
4. Execute past procedural assignments whose timing controls have scheduled an assignment for the current simulator time.
5. Advance the simulator time.

# Example

```

module nb_b ();
    reg sig_c, sig_d, sig_e;
    initial begin
        swap { sig_c <= sig_d; // gets x
              sig_d <= sig_c; // gets x
              sig_c = 0;
              sig_d = 1;
        }
    end
    set { initial begin
          sig_e = 0;
          #50 sig_e = 1;
          sig_c = 0;
          sig_d = 0;
        }
    end
endmodule

```



1. The non-blocking statements are encountered first in the sequential activity flow, and **sample** the values of c and d (x).
2. The blocking assignments at the same time step then **execute** in order, setting c to 0 and d to 1.
3. Then the non-blocking assignments execute, setting c to x and d to x.
4. Thus, **the non-blocking assignments overwrite the blocked assignments at the same time step.**

# \$display vs. \$monitor

---

## ■ \$display

- Executes immediately when it is encountered in the sequential activity flow of a behavior.

## ■ \$monitor

- Executes automatically at the <sup>最後</sup>end of the current time step (i.e., after the non-blocking assignments have been updated).



# Example 1

---

**initial begin:** execute\_display

a = 1;

b = 0;

3. set a <= b; 1. get

b <= a;

2. print \$display ("display: a = %b b = %b", a, b);

**end**

Execution result: display: a = 1 b = 0

交换前

1. Assign value to a and b.
2. Sample the current RHS of a and b.
3. Display the current values of a and b.
4. Update a and b.

## Example 2

---

**initial begin:** execute\_display

```
c = 1;  
d = 0;  
2. set c <= d; 1. get  
d <= c; 3. print  
$monitor ("monitor: c = %b d = %b", c, d);  
end
```

Execution result: monitor: c = 0 d = 1

交換後

1. Assign value to c and d.
2. Sample the current RHS of c and d.
3. Update c and d.
4. Display the current values of c and d.

# \$display() vs. \$monitor() (1/2)

```
module display;
```

```
    reg [2:0] a, b;
```

```
    initial begin
```

```
        // *** initial time step
```

```
        a = 3'd0;
```

```
        b = 3'd0;
```

2. mon  
1. dis

```
        $monitor("%0dns : \ $monitor 0: a=%d b=%d" , $stime, a, b);  
        $display("%0dns : \ $display 0: a=%d b=%d" , $stime, a, b);
```

```
    // ***
```

```
    #1;
```

```
    a = 3'd1;
```

```
    b = 3'd2;
```

3. set 1.9pt

```
    a <= b;
```

```
    b <= a;
```

4. mon

```
        $monitor("%0dns : \ $monitor 1: a=%d b=%d" , $stime, a, b);  
        $display("%0dns : \ $display 1: a=%d b=%d" , $stime, a, b);
```

2. dis

```
VSIM 12> run -all
```

```
# 0ns : $display 0: a=0 b=0
```

```
# 0ns : $monitor 0: a=0 b=0
```

```
# 1ns : $display 1: a=1 b=2
```

```
# 1ns : $monitor 1: a=2 b=1
```

```
# 2ns : $display 2: a=2 b=1
```

```
# 2ns : $monitor 2: a=2 b=1
```

```
# 3ns : $display 3: a=3 b=4
```

```
# 3ns : $monitor 3: a=1 b=2
```

# \$display() vs. \$monitor() (2/2)

```
VSIM 12> run -all
# 0ns :$display 0: a=0 b=0
# 0ns :$monitor 0: a=0 b=0
# 1ns :$display 1: a=1 b=2
# 1ns :$monitor 1: a=2 b=1
# 2ns :$display 2: a=2 b=1
# 2ns :$monitor 2: a=2 b=1
# 3ns :$display 3: a=3 b=4
# 3ns :$monitor 3: a=1 b=2
```

```
// ***
#1
$monitor("%0dns :\$monitor 2: a=%d b=%d" , $stime, a, b);
$display("%0dns :\$display 2: a=%d b=%d" , $stime, a, b);
```

```
// ***
#1;
a <= b;
b <= a;
a = 3'd3;
b = 3'd4;
```

```
$monitor("%0dns :\$monitor 3: a=%d b=%d" , $stime, a, b);
$display("%0dns :\$display 3: a=%d b=%d" , $stime, a, b);
```

end

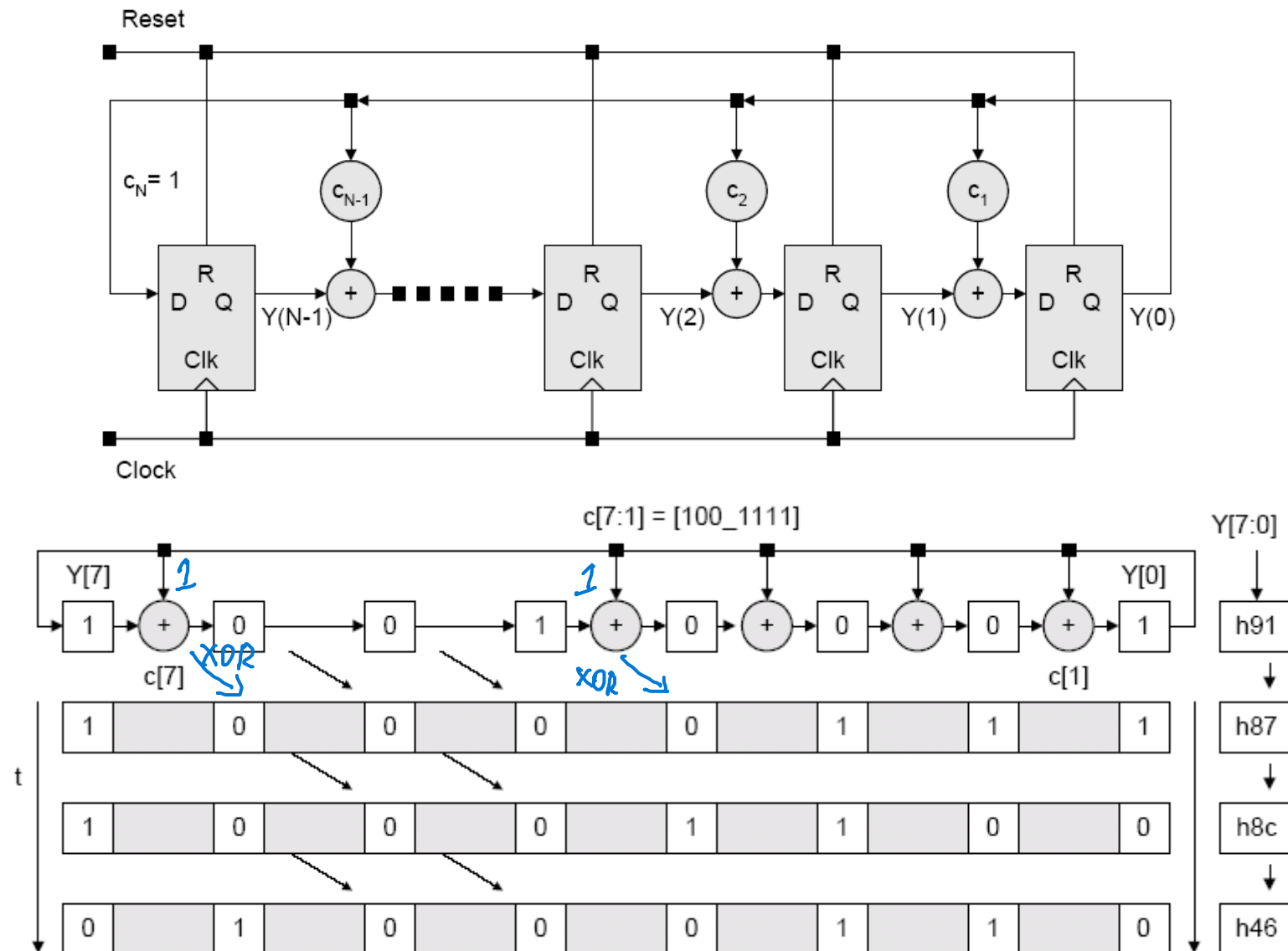
endmodule

4. set { a=1  
b=2 }  
2. set { a=3  
b=4 }

\$mon

\$dis

# Linear Feedback Shift Register (1/3)



# Linear Feedback Shift Register (2/3)

```
module LFSR_nb (Clock, Reset, Y);
```

```
    parameter Length = 8;
```

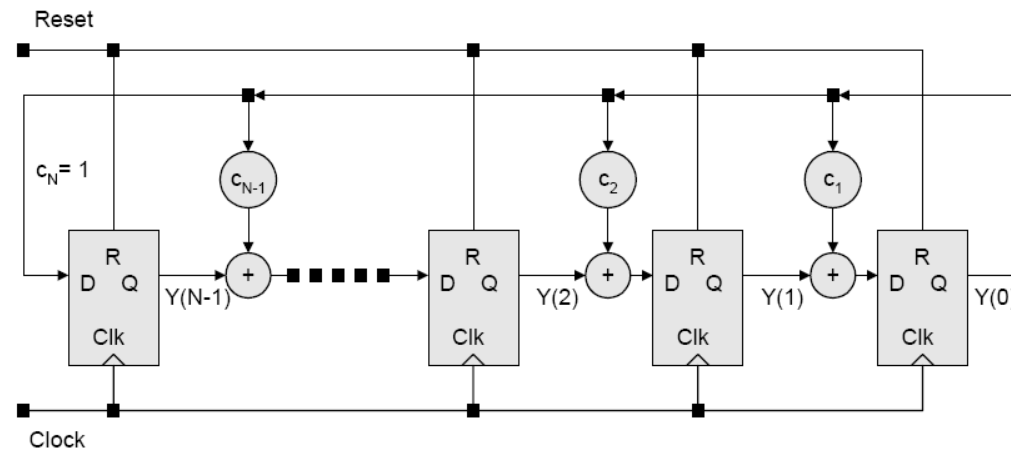
```
    parameter initial_state = 8'b1001_0001; 种子
```

```
    parameter [Length-2:0] Tap_Coefficient = 7'b100_1111; 加法器位置
```

```
    input Clock, Reset;
```

```
    output [Length-1: 0] Y;
```

```
    reg [Length-1: 0] Y;
```



# Linear Feedback Shift Register (3/3)

**always @ (posedge Clock or posedge Reset)**

**if (Reset)**  $Y \leq \text{initial\_state};$

**else begin**

加法(xor)

$Y[0] \leq \text{Tap\_Coefficient}[0] ? Y[1] \wedge Y[0] : Y[1];$

$Y[1] \leq \text{Tap\_Coefficient}[1] ? Y[2] \wedge Y[0] : Y[2];$

$Y[2] \leq \text{Tap\_Coefficient}[2] ? Y[3] \wedge Y[0] : Y[3];$

$Y[3] \leq \text{Tap\_Coefficient}[3] ? Y[4] \wedge Y[0] : Y[4];$

$Y[4] \leq \text{Tap\_Coefficient}[4] ? Y[5] \wedge Y[0] : Y[5];$

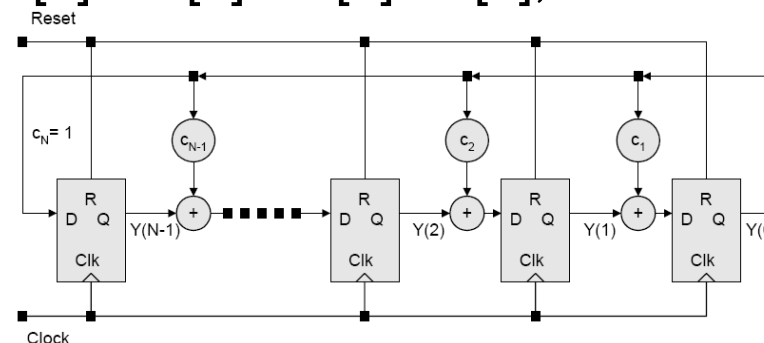
$Y[5] \leq \text{Tap\_Coefficient}[5] ? Y[6] \wedge Y[0] : Y[6];$

$Y[6] \leq \text{Tap\_Coefficient}[6] ? Y[7] \wedge Y[0] : Y[7];$

$Y[7] \leq Y[0];$

**end**

**endmodule**



# Repeated Intra-Assignment Delay

---

*1. set*  
reg\_a = **repeat** (5) @ (**negedge** clock) reg\_b;  
*2. run 5 times* *1. get*



**begin**

temp = reg\_b;

@ (**negedge** clock);

@ (**negedge** clock);

@ (**negedge** clock);

@ (**negedge** clock);

@ (**negedge** clock);

reg\_a = temp;

**end**



# Example

```
module repeater;  
  reg clock;  
  reg reg_a, reg_b;  
  initial  
    clock = 0;  
  initial begin  
    #5 reg_a = 1;  
    #10 reg_a = 0;  
    #5 reg_a = 1;  
    #20 reg_a = 0;  
  end  
  always  
    #5 clock = ~clock;
```

**initial**

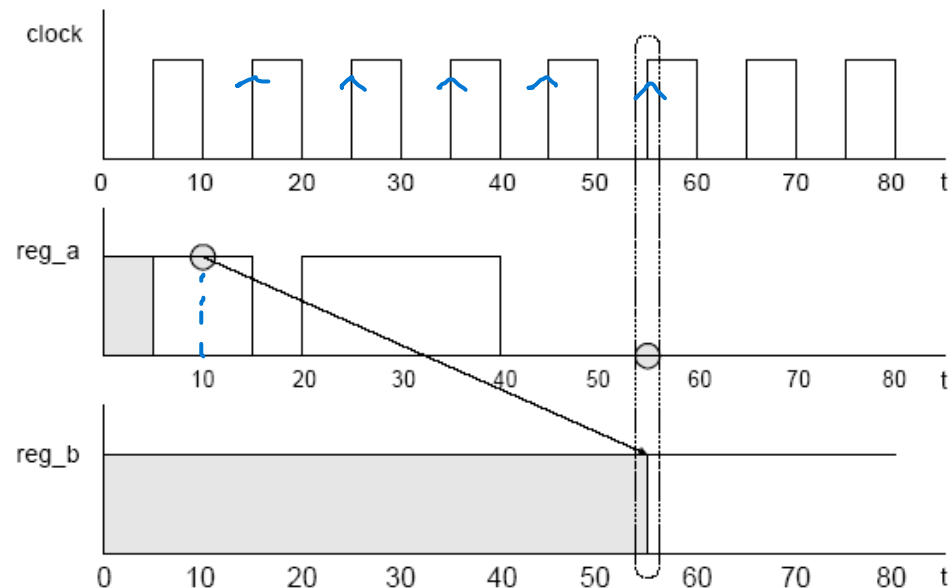
#100 \$finish;

**initial**

#10 reg\_b = repeat (5)

@ (posedge clock) reg\_a;

**endmodule**



# Indeterminate Assignments and Ambiguity

---

- Multiple **concurrent behaviors** may assign value to the same register in the same time step.
- When multiple assignments are made by different behaviors to the same target variable in the same time step, the order in which the assignments are made, and therefore the outcome, is **indeterminate**.

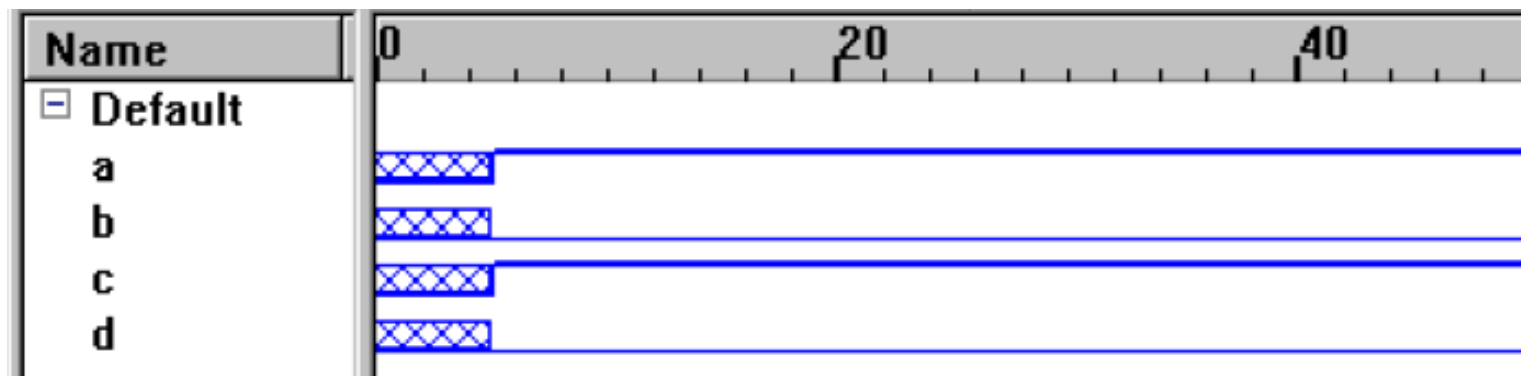
<b>always</b>	<b>always</b>
<code>a &lt;= 0;</code>	<code>a &lt;= 1;</code>

- The result is indeterminate and implementation-dependent.

# Example 1

```
module multi_assign ();  
  reg a, b, c, d;  
  initial #50 $finish;  
  
  initial begin  
    #5 a = 1; b = 0;  
  end
```

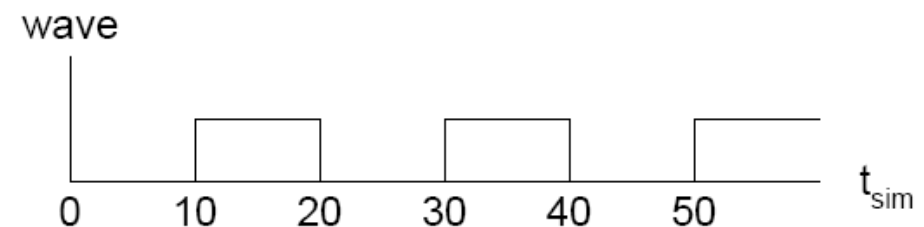
```
    always @ (posedge a) c = a;  
    always @ (posedge a) c = b;  
    always @ (posedge a) d = b;  
    always @ (posedge a) d = a;  
  
endmodule
```



## Example 2

```
module multi_nb1;  
    reg wave;  
    reg [2:0] i;  
  
    initial begin  
        wave = 0;  
        for (i = 0; i <= 5; i = i+1)  
            wave <= #(i*10) i[0];  
    end  
endmodule
```

i	i[2]	i[1]	i[0]	t <sub>sim</sub>
0	0	0	0	0
1	0	0	1	0 <sup>+</sup>
2	0	1	0	0 <sup>++</sup>
3	0	1	1	0 <sup>+++</sup>
4	1	0	0	0 <sup>++++</sup>
5	1	0	1	0 <sup>+++++</sup>



## Example 3

```
module multi_nb2;  
  reg wave1, wave2;
```

```
  initial begin
```

```
    #5    wave1 = 0;
```

```
    wave2 = 0;
```

```
    wave1 <= #5 1;    10
```

```
    wave2 <= #10 1;    15
```

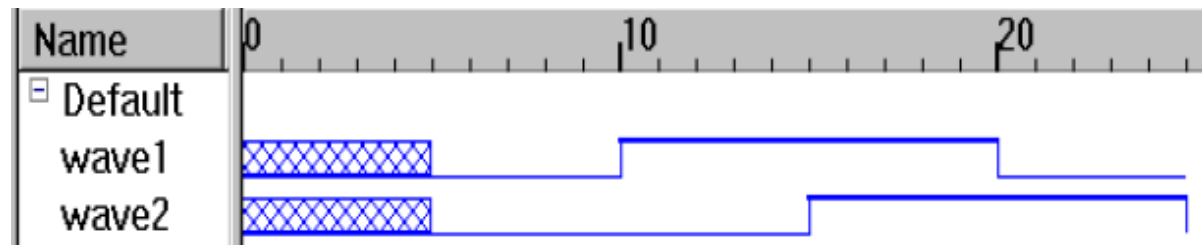
```
    wave2 <= #20 0;    25
```

```
    #10    wave1 = 1;
```

```
    wave1 <= #5 0;
```

```
  end
```

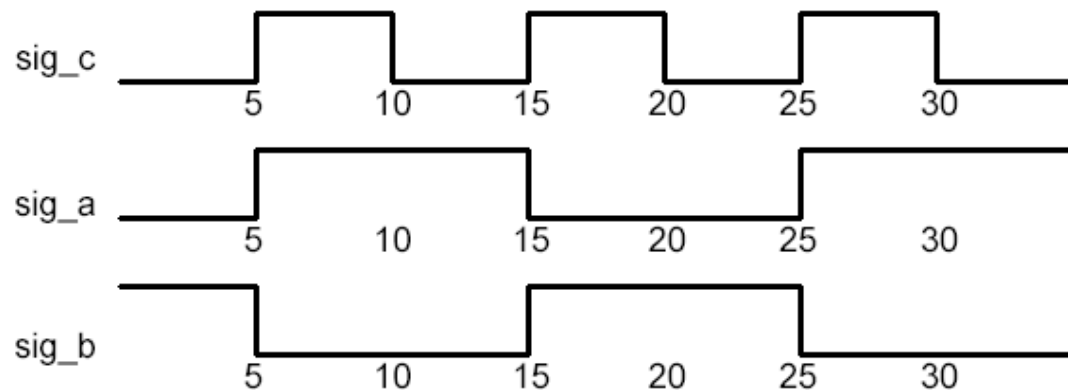
```
endmodule
```



## Example 4

---

```
module nb ();  
    reg sig_a, sig_b, sig_c;  
  
    initial begin  
        sig_a = 0; sig_b = 1; sig_c = 0;  
    end  
    always sig_c = #5 ~sig_c; clock  
    always @ (posedge sig_c) begin  
        swap ↗ sig_a <= sig_b;  
        ↘ sig_b <= sig_a;  
    end  
endmodule
```



# Constructs for Activity Flow Control

---

- Conditional operator (**? ... :**)
- The **case** statement (**case**, **casex**, **casez**)
- Conditional statement (**if ... else**)
- Loops (**repeat**, **for**, **while**, **forever**)
- The **disable** statement
- Parallel activity flow (**fork ... join**)

# Conditional Operator

---

- Example

- **assign** y\_out = (bus\_enabled) ? reg\_A : 16'bz;

- May appear in the following:

- An expression in a continuous assignment statement
- An expression in the body of a procedural statement



# Example

---

```
module mux_beh (clock, reset, sel, a, b, y_out);  
    input          clock, reset, sel;  
    input [15:0]    a, b;  
    output [15:0]    y_out;  
    reg [15:0]       y_out;  
  
    always @ (posedge clock or negedge reset)  
        if (reset == 0)  
            y_out = 0;  
        else  
            y_out = (sel) ? a+b : a-b;  
endmodule
```

*Handwritten green notes:*  
2  $\frac{1}{2}$  1  $\frac{1}{2}$  1  $\frac{1}{2}$   
1  $\frac{1}{2}$  1  $\frac{1}{2}$

## 4x1 Mux

---

```
module mux_cond1(a, b, c, d, sel, y);  
    input          a, b, c, d;  
    input  [1:0] sel;  
    output         y;  
  
    assign y =  
        (!sel[1] && !sel[0]) ? a :  
        (!sel[1] && sel[0]) ? b :  
        ( sel[1] && !sel[0]) ? c :  
        ( sel[1] && sel[0]) ? d : 1'bx;  
endmodule
```

```
module mux_cond2(a, b, c, d, sel, y);  
    input          a, b, c, d;  
    input  [1:0] sel;  
    output         y;  
  
    assign y =  
        (sel == 0) ? a :  
        (sel == 1) ? b :  
        (sel == 2) ? c :  
        (sel == 3) ? d : 1'bx;  
endmodule
```

# “case” Statement

## ■ case

- Complete bitwise match between expression and case\_item expression.

## ■ casex

- Treat ‘x’ and ‘z’ values in expression or case\_item as don't-cares.

## ■ casez

- Treat ‘z’ value in expression or case\_item as don't-cares.

Expression or case_item	case	casex	casez
0	0	0	0
1	1	1	1
x	x	0 1 x z	x
z	z	0 1 x z	0 1 x z
?	*	*	0 1 x z
default	0 1 x z	0 1 x z	0 1 x z

\* not applicable

# Example 1

---

```
// The case statement requires an exact bitwise match.  
reg [1:0] sig_A;  
...  
case (sig_A)  
    2'b00:                $display ("sig_A has no ones");  
    2'b01, 2'b10:         $display ("sig_A has a single one");  
    2'b11:                $display ("sig_A has two ones");  
    2'bx:                  $display ("sig_A is unknown value");  
    2'bz:                  $display ("sig_A is high impedance value");  
    default:             $display ("Value of sig_A is mixed");  
endcase
```

## Example 2

```
module casex_decoder;
```

```
  reg [7:0] r, mask;
```

```
  always begin
```

```
    mask = 8'bx0x0x0x0;
```

```
    casex (r ^ mask) // bitwise xor
```

```
      8'b001100xx: do_task_1;
```

```
      8'b1100xx00: do_task_2;
```

```
      // matched first, do_task_2 is executed!
```

```
      8'b00xx0011: do_task_3;
```

```
      8'bxx001100: do_task_4;
```

```
      // matched second, do_task_4 is NOT executed!
```

```
    endcase
```

```
  end
```

```
endmodule
```

x0x0x0x0	mask word
^01100110	r word
-----	
x1x0x1x0	case_expression
1100xx00	case item match

## 4x1 Mux

---

```
module mux_case1(a, b, c, d, sel, y);
  input      a, b, c, d;
  input      [1:0] sel;
  output     y;
  reg        y;

  always @ (sel)
    case (sel)
      0: assign y = a;
      1: assign y = b;
      2: assign y = c;
      3: assign y = d;
      default:
        assign y = 1'bx;
    endcase
endmodule
```

```
module mux_case2(a, b, c, d, sel, y);
  input      a, b, c, d;
  input      [1:0] sel;
  output     y;
  reg        y;

  always @ (a or b or c or d or sel)
    case (sel)
      0: y = a;
      1: y = b;
      2: y = c;
      3: y = d;
      default:
        y = 1'bx;
    endcase
endmodule
```

## “if ... else” Statement

---

- **if** (A < B) sum\_register = sum\_value + 1;
- **if** (C < D); // null statement
- **if** (k == 1) **begin**  
    ...  
**end**
- The value of the Boolean expression is treated as false if it has the numerical value of “0”, “x”, or “z”.
- A non-zero numeric expression will evaluate to “true”.

# Example

---

```
always @ (a or b) begin  
    if (a >= b) begin  
        neg_result = 0;  
        diff = a - b;  
    end  
    else begin  
        neg_result = 1;  
        diff = b - a;  
    end  
end
```

```
if (value_1 < value_2)  
    sum = sum + 1;  
else  
    sum = sum + 2;
```

---

```
if (A == 1)  
    sig_out = reg_a;  
else if (A == 2)  
    sig_out = reg_b;  
else if (A == 3)  
    sig_out = reg_c;
```



## 4x1 Mux

---

```
module mux_if1 (a, b, c, d, sel, y);  
  input      a, b, c, d;  
  input      [1:0] sel;  
  output     y;  
  reg        y;  
  
  always @ (sel)  
  begin  
    if (sel == 0)      assign y = a;  
    else if (sel == 1) assign y = b;  
    else if (sel == 2) assign y = c;  
    else if (sel == 3) assign y = d;  
    else               assign y = 1'bx;  
  
  end  
endmodule
```

```
module mux_if2 (a, b, c, d, sel, y);  
  input      a, b, c, d;  
  input      [1:0] sel;  
  output     y;  
  reg        y;  
  
  always @ (a or b or c or d or sel)  
  begin  
    if (sel == 0)      y = a;  
    else if (sel == 1) y = b;  
    else if (sel == 2) y = c;  
    else if (sel == 3) y = d;  
    else               y = 1'bx;  
  
  end  
endmodule
```

# “repeat” Loop

---

## repeat (count\_expression) statement

- The count\_expression determines the number of times the execution is repeated.
- If the count\_expression evaluates to “x” or “z”, the result will be treated as 0.
- The execution can end prematurely with the “**disable**” statement.

# Example 1

---

```
repeat (4) begin  
    sum_out[i] = acc[i];  
    i = i+1;  
end
```

```
repeat (reg_A) begin  
    sum_out[i] = acc[i];  
    i = i+1;  
end
```

```
initial begin  
    #start_up repeat (2)  
    #delay clock = ~clock;  
end
```

---

```
word_address = 0;  
repeat (memory_size) begin  
    memory [word_address] = 0;  
    word_address = word_address + 1;  
end
```

## Example 2

---

result = operand\_a \* operand\_b

```
repeat (operand_length) begin  
    #5 if (operand_b[0])  
        result = result + operand_a;  
    operand_a = operand_a << 1;  
    operand_b = operand_b >> 1;  
end
```

## “for” Loop

---

```
reg [15:0] demo_reg;  
integer k;  
...  
for (k = 4; k; k = k-1) begin  
    demo_reg [k+10] = 0; // demo_register[14] ~ demo_register[11] = 0  
    demo_reg [k+2] = 1;  // demo_register[6] ~ demo_register[3] = 1  
end
```

---

```
reg [3:0] k;  
for (k=0; k <= 15; k = k + 1) // Endless loop!  
...  
// k = 0, 1, 2, ..., 14, 15, 0, 1, 2, ...
```

# “while” Loop

---

## **while** (expression) statement

- Execution recycles as long the expression is TRUE.
- Skip execution if expression is FALSE when encountered.

```
for (K=16; K; K=K-1)  
begin  
    ...  
end
```

```
K = 16;  
while (K)  
begin  
    ...  
    K = K-1;  
end
```

# Example

---

```
begin: count_of_1s
    reg [7:0] temp_reg;
    count = 0;
    temp_reg = reg_a;

    while (temp_reg) begin
        if (temp_reg[0])
            count = count + 1;
        temp_reg = temp_reg >> 1;
    end
end
```

```
begin: count_of_1s
    reg [7:0] temp_reg;
    count = 0;
    temp_reg = reg_a;

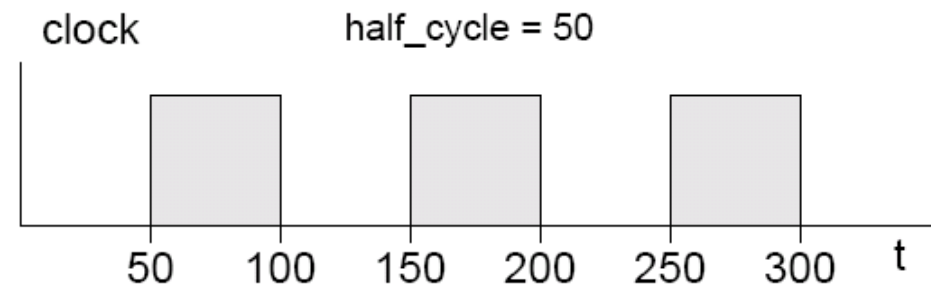
    while (temp_reg) begin
        count = count + temp_reg[0];
        temp_reg = temp_reg >> 1;
    end
end
```

# “forever” Loop

---

```
module microprocessor;  
always begin  
    power_on_initialization;  
  
    forever begin  
        fetch_instruction;  
        decode_instruction;  
        execute_instruction;  
    end  
end  
endmodule
```

```
Initial    // clock generator  
begin  
    clock = 0;  
  
    forever  
        #half_cycle clock = ~clock;  
end
```

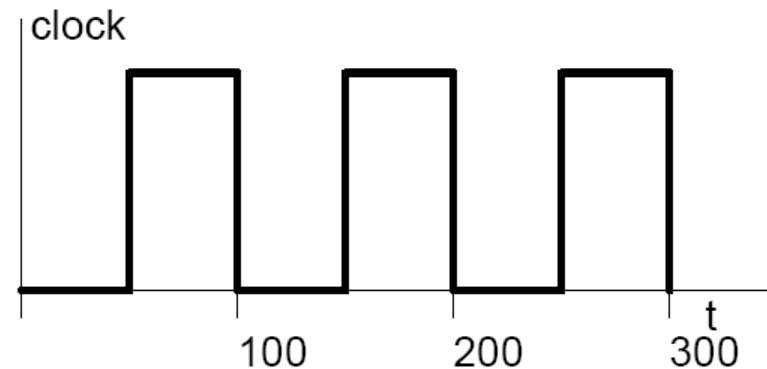




# Example 1

---

```
parameter half_cycle = 50;  
initial begin: clock_loop  
    clock = 0;  
    forever begin  
        #half_cycle clock = 1;  
        #half_cycle clock = 0;  
    end  
end  
  
initial  
    #350 disable clock_loop;
```

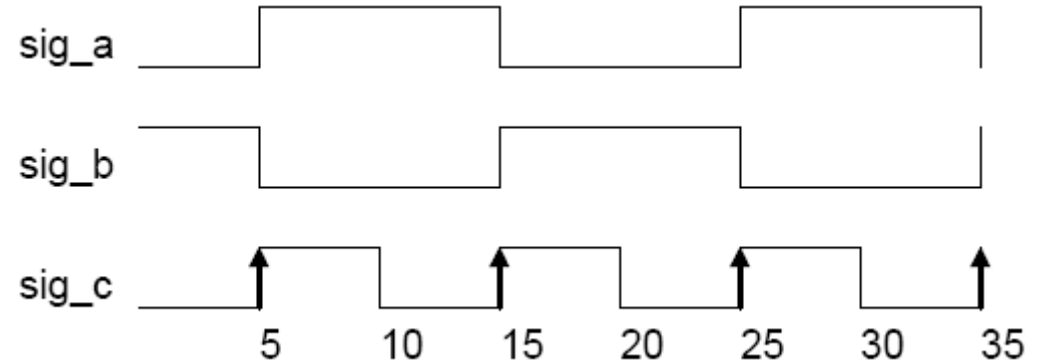


## Example 2

```
module nb2;
  reg    sig_a, sig_b, sig_c;

  initial begin
    sig_a = 0; sig_b = 1; sig_c = 0;
    forever
      sig_c = #5 ~sig_c;
  end

  always @ (posedge sig_c) begin
    sig_a <= sig_b;
    sig_b <= sig_a;
  end
endmodule
```



time	sig_a	sig_b	sig_c
0	0	1	0
5	1	0	1
10	1	0	0
15	0	1	1
20	0	1	0
25	1	0	1
30	1	0	0
35	0	1	1

# Comparison of Loops

---

## ■ repeat

- Expression determines a **constant** number of repetitions

## ■ for

- Loop variable determines number of repetitions

## ■ while

- Iterates while expression evaluates “true”

## ■ forever

- Iterates indefinitely

## ■ To terminate a loop: **“disable”**

*break*

# “always” vs. “forever”

---

## ■ **always** 總是行為 ⇒ 無巢狀

- Declares a concurrent behavior.
- May not be nested.
- Becomes active and can execute at the beginning of simulation.

## ■ **forever** 有巢狀 `while(true){...}`

- A computational activity flow, not necessarily concurrent with any other activity flow.
- Can be nested.
- Executes only when it is reached within a sequential activity flow.

## “disable” Statement

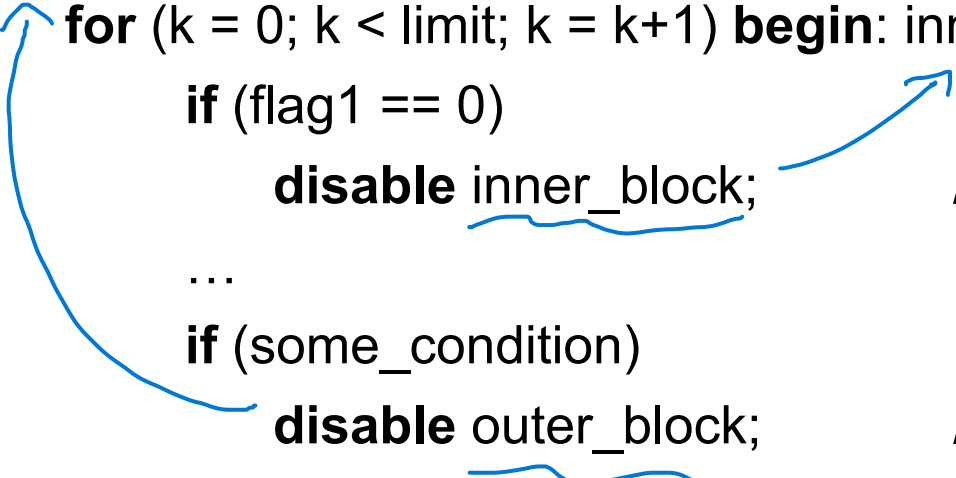
---

- “**disable**” terminates a named block or task.
- “**\$finish**” terminates simulation.
- Analogous to “**continue**” and “**break**” in C.
- Execution resumes with the statement following the block.

# Example 1

---

```
begin: outer_block
  for (k = 0; k < limit; k = k+1) begin: inner_block
    if (flag1 == 0)
      disable inner_block;           // aborts iteration
    ...
    if (some_condition)
      disable outer_block;         // aborts for loop
    ...
  end
end
```



## Example 2

---

```
module find_first_one (A_word, trigger, idx);  
    input          [15:0]  A_word;  
    input          trigger;  
    output        [4:0]   idx;  
    reg           [4:0]   idx;  
    // Find the location of the first “1” in a 16-bit word.  
    always @ trigger  
    begin: trigger_block  
        idx = 0;  
        for (idx=0; idx <= 15; idx = idx + 1)  
            if (A_word[idx] == 1)  
                disable trigger_block;  
    end  
endmodule
```

# “fork ... join” Statement

---

- Create **parallel threads of activity**, each executing concurrently with the others.

**fork**

```
statement_1;  
statement_2; ...
```

**join**

平行執行  
要等全部結束才能繼續執行

- **Not** supported by synthesis tools.
- Support waveform generation in testbenches.
- The statement following **fork ... join** cannot execute *until all the activity of the parallel threads is complete.*



# Example 1

**begin**

```
#50 r = 'h35;  
#100 r = 'h00;  
#150 r = 'hF7;  
#300 r = 'h00;
```

**end**

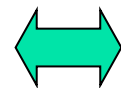


~~fork ... join~~ is not equivalent to non-blocking assignment because race is not possible in non-blocking assignment.

**fork**

```
#50 r = 'h35;  
#100 r = 'h00;  
#150 r = 'hF7;  
#300 r = 'h00;
```

**join**



**begin**

```
#50 r = 'h35;  
#50 r = 'h00;  
#50 r = 'hF7;  
#150 r = 'h00;
```

**end**



## Example 2

---

**fork**

**begin**

...

**end**

**begin**

**fork**

**begin**

...

**end**

**join**

**end**

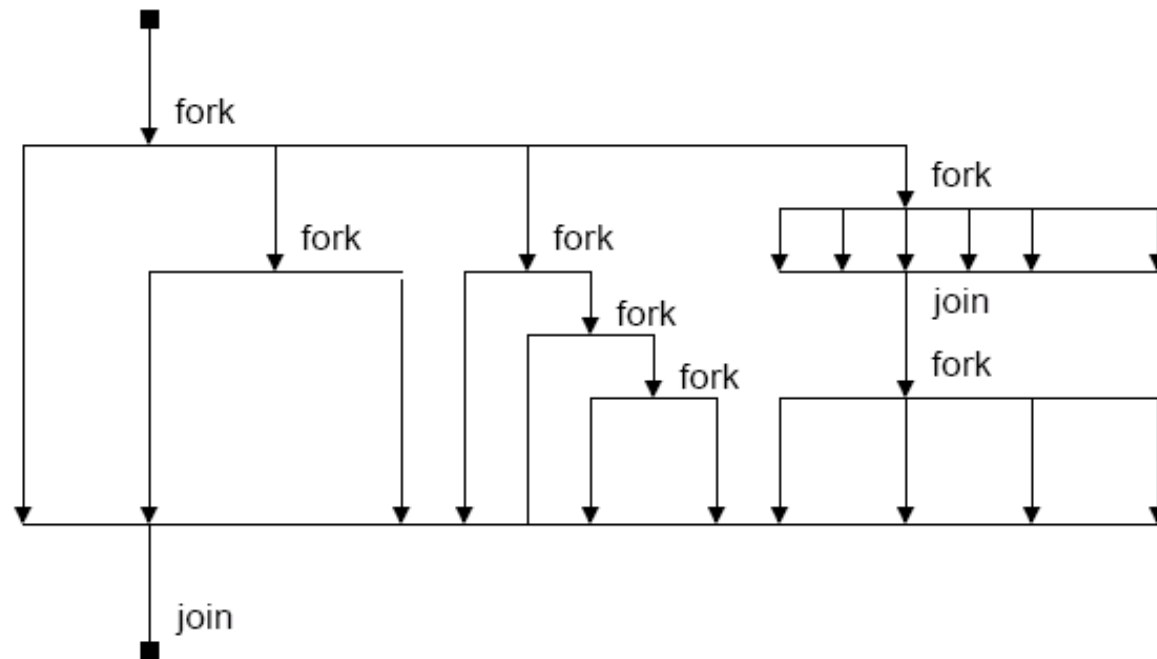
...

**begin**

...

**end**

**join**



確定行為，應避免

## Race Condition

---

- **Race** may occur when the same register variable is **assigned value** and **referenced** simultaneously in multiple activity threads in a parallel block.
- The outcome of simulation will be **indeterminate** (vendor-dependent and unpredictable).

# Example

---

**fork** // **Race!**

#150 **reg\_a** = reg\_b;

#150 reg\_c = **reg\_a**;

**join**

---

**fork** // No race!

reg\_a = #150 reg\_b;

reg\_c = #150 reg\_a;

**join**



reg\_a <= #150 reg\_b;

reg\_c <= #150 reg\_a;

1. Intra-assignment delay causes **immediate sampling** and delayed execution of a procedural assignment.
2. The equivalent non-blocking description does not race, with or without intra-assignment delay.

# Tasks and Functions

---


- Two types of sub-programs.
- Tasks <sup>void - without return</sup> create a hierarchical organization of the procedural statements.
- Functions <sup>at '=' right - have return</sup> substitute for an expression.
- Improve the readability, portability, and maintainability of code.

# Bit Counter by Task (1/2)

```
module bit_counter(data_word, bit_cnt);  
input    [7:0]    data_word;  
output   [3:0]    bit_cnt;  
reg      [3:0]    bit_cnt;
```

```
always @(data_word)  
    count_ones(data_word, bit_cnt);
```

task

 A task that receives a binary word (**data\_word**) and returns the number of ones in data\_word (by **bit\_cnt**).

## 執行結果

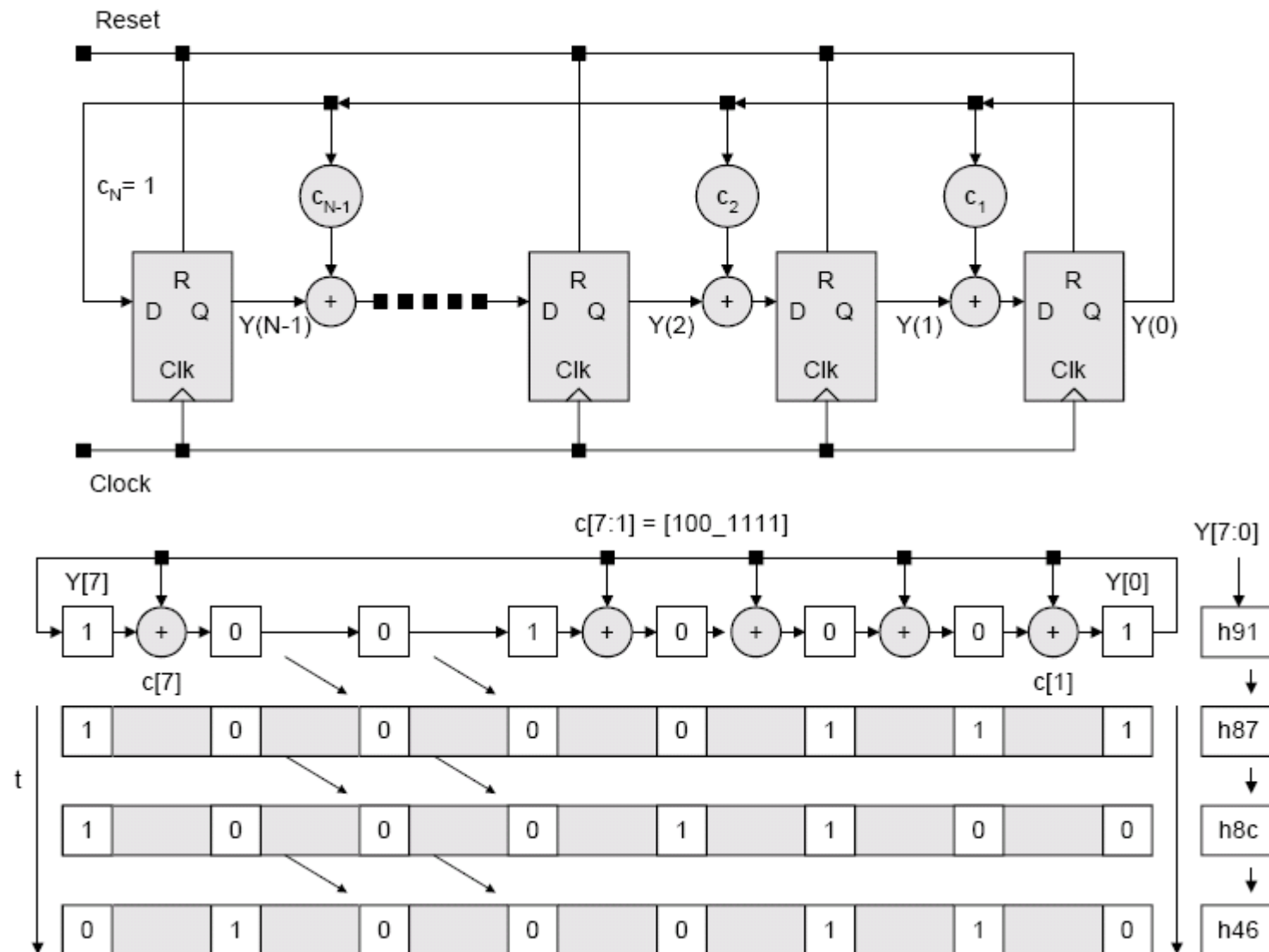
```
# Time: 0, data_word = 00000000, bit_cnt= 0  
# Time: 10, data_word = 01101001, bit_cnt= 4  
# Time: 20, data_word = 11000001, bit_cnt= 3  
# Time: 30, data_word = 00110111, bit_cnt= 5
```

## Bit Counter by Task (2/2)

---

```
task count_ones;  
  input    [7:0] reg_a;  
  output   [3:0] count;  
  reg      [3:0] count;  
  reg      [7:0] temp_reg;  
  
  begin  
    count = 0;  
    temp_reg = reg_a;  
    while (temp_reg) begin  
      count = count + temp_reg[0];  
      temp_reg = temp_reg >> 1;  
    end  
  end  
endtask  
  
endmodule
```

# Linear Feedback Shift Register (LFSR)





# LFSR by Task (1/2)

---

*global  
task*

```
module LFSR_task(clk, rst, Y);  
  parameter Length = 8;  
  parameter initial_state = 8'b1001_0001;  
  parameter [Length-2:0] Tap_Coefficient = 7'b100_1111;  
  
  input  clk, rst;  
  output [Length-1:0] Y;  
  reg    [Length-1:0] Y;  
  
  always @(posedge clk or posedge rst)  
    if (rst) Y = initial_state;  
    else Update_LFSR;
```

*call task*

## LFSR by Task (2/2)

```
task Update_LFSR;
    reg [Length-1:0] LFSR_state;
    integer i;
    begin
        LFSR_state = Y;
        for (i=0; i<=Length-2; i=i+1)
            if (Tap_Coefficient[i] == 1)
                Y[i] = LFSR_state[i+1] ^ LFSR_state[0];
            else
                Y[i] = LFSR_state[i+1]; // Y: global variable
        Y[Length-1] = LFSR_state[0];
    end
endtask
```

```
always @(posedge clk or posedge rst)
    if (rst)
        Y <= initial_state;
    else begin
        Y[0] <= Tap_Coefficient[0] ? Y[1] ^ Y[0] : Y[1];
        Y[1] <= Tap_Coefficient[1] ? Y[2] ^ Y[0] : Y[2];
        Y[2] <= Tap_Coefficient[2] ? Y[3] ^ Y[0] : Y[3];
        Y[3] <= Tap_Coefficient[3] ? Y[4] ^ Y[0] : Y[4];
        Y[4] <= Tap_Coefficient[4] ? Y[5] ^ Y[0] : Y[5];
        Y[5] <= Tap_Coefficient[5] ? Y[6] ^ Y[0] : Y[6];
        Y[6] <= Tap_Coefficient[6] ? Y[7] ^ Y[0] : Y[7];
        Y[7] <= Y[0];
    end
endmodule
```

# 4x1 Mux by Function

---

```
module mux_fc (a, b, c, d, sel, y);  
    input      a, b, c, d;  
    input      [1:0] sel;  
    output     y;  
  
    assign y = mux (a, b, c, d, sel);
```

```
function mux;  
    input a, b, c, d;  
    input [1:0] sel;  
        case (sel) return  
            2'b00: mux = a;  
            2'b01: mux = b;  
            2'b10: mux = c;  
            2'b11: mux = d;  
            default: mux = 1'bx;  
        endcase  
    endfunction  
endmodule
```

# LFSR by Function (1/2)

---

```
module LFSR_func(clk, rst, Y);
parameter    Length = 8;
parameter    initial_state = 8'b1001_0001;
parameter    [Length-2:0] Tap_Coefficient = 7'b100_1111;

input    clk, rst;
output   [Length-1:0] Y;
reg      [Length-1:0] Y;

always @(posedge clk or posedge rst)
    if (rst) Y = initial_state;
    else Y = LFSR_Value(Y);
```

# LFSR by Function (2/2)

```
function [Length-1:0] LFSR_Value;  
    input [Length-1:0] LFSR_state;  
    integer i;  
    begin  
        for (i=0; i<=Length-2; i=i+1)  
            if (Tap_Coefficient[i] == 1)  
                LFSR_Value[i] = LFSR_state[i+1] ^ LFSR_state[0];  
            else  
                LFSR_Value[i] = LFSR_state[i+1];  
        LFSR_Value[Length-1] = LFSR_state[0];  
    end  
endfunction
```

*all return or one bit return ?*

*1 bit*

*1 bit*

endmodule

```
always @(posedge clk or posedge rst)  
    if (rst)  
        Y <= initial_state;  
    else begin  
        Y[0] <= Tap_Coefficient[0] ? Y[1] ^ Y[0] : Y[1];  
        Y[1] <= Tap_Coefficient[1] ? Y[2] ^ Y[0] : Y[2];  
        Y[2] <= Tap_Coefficient[2] ? Y[3] ^ Y[0] : Y[3];  
        Y[3] <= Tap_Coefficient[3] ? Y[4] ^ Y[0] : Y[4];  
        Y[4] <= Tap_Coefficient[4] ? Y[5] ^ Y[0] : Y[5];  
        Y[5] <= Tap_Coefficient[5] ? Y[6] ^ Y[0] : Y[6];  
        Y[6] <= Tap_Coefficient[6] ? Y[7] ^ Y[0] : Y[7];  
        Y[7] <= Y[0];  
    end  
endmodule
```

'''  
'''  
'''

## Function Calls Function (1/2)

---

// 16-bit even-parity generator

**module** even\_parity\_16 (Din, Pout);

**input** [15:0] Din;   **output** Pout;

**reg** [7:0] High\_byte; // high byte of input data

**reg** [7:0] Low\_byte; // low byte of input data

**reg** High, Low;       // parities of high and low byte

**reg** Pout;           // parity output

**always** @(Din)

**begin**

        High\_byte = Din[15:8];           Low\_byte = Din[7:0];

        High = **even8** (High\_byte);      Low = **even8** (Low\_byte);

        Pout = High ^ Low; // bitwise xor

**end**

## Function Calls Function (2/2)

---

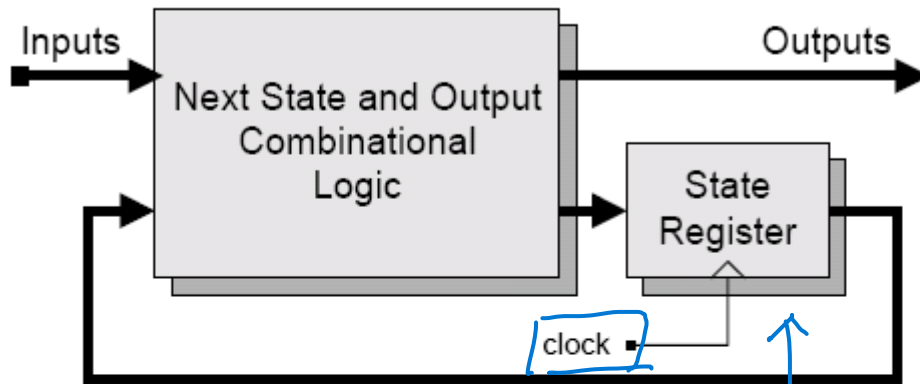
```
function even8;  
    input [7:0] I8;  
    even8 = even4 (I8[7:4]) ^ even4(I8[3:0]); // xor operation  
endfunction
```

```
function even4;  
    input [3:0] I4;  
    even4 = ^ I4; // reduction xor operation  
endfunction  
endmodule
```

# Behavioral Models of Finite State Machines (FSMs)

有限狀態機

## Mealy machine



~~The outputs of a Mealy machine are asynchronous, thus are subject to glitches in the inputs.~~

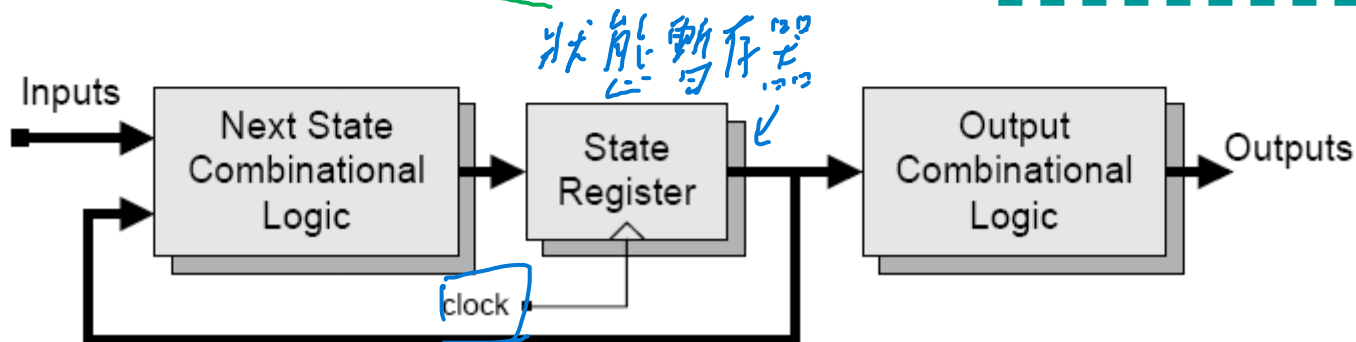
有雜訊  
不穩定

不同步

~~The outputs of a Moore machine are synchronous.~~

同步

## Moore machine



狀態寄存器



# Explicit Mealy Machine - Style 1

```
module FSM_style1 (...);  
    input ... ;    output ... ;  
    parameter size = ... ;  
    reg [size-1:0] state, next_state;
```

// Combinational logic for outputs and next state

```
assign outputs = ...           // a function of state and inputs
```

```
assign next_state = ...       // a function of state and inputs
```

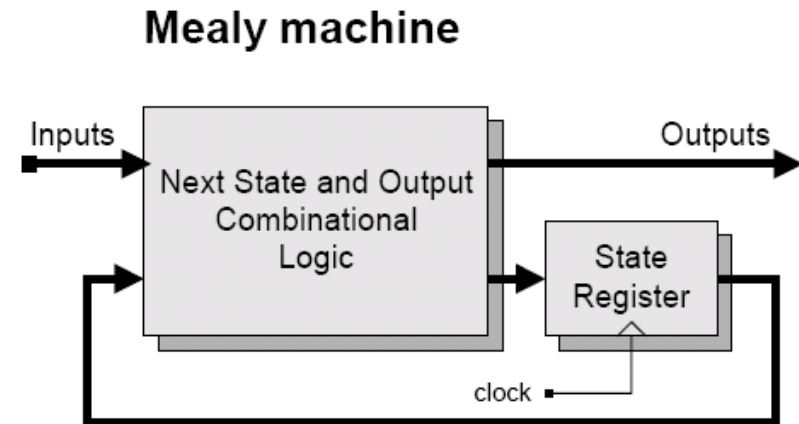
// State transitions

```
always @ (posedge clk or negedge reset)
```

```
    if (reset == 1'b0) state <= start_state;
```

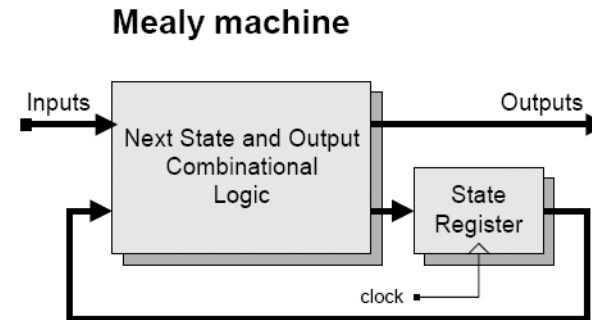
```
    else state <= next_state;
```

```
endmodule
```

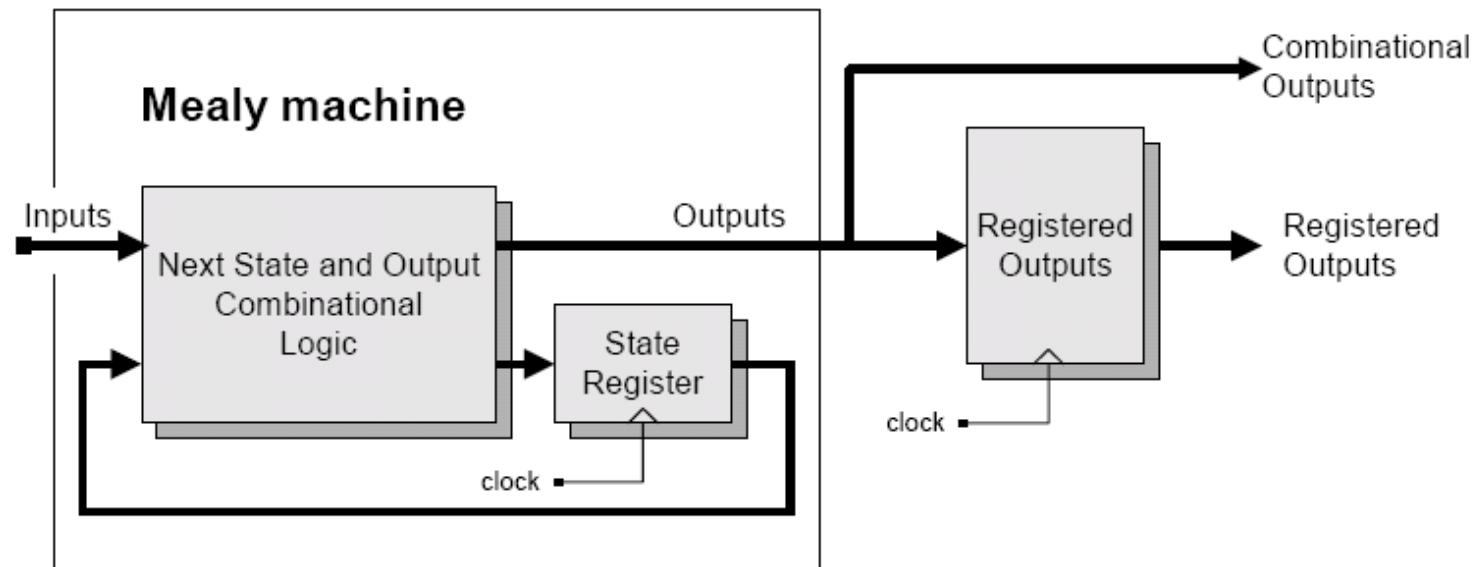


# Explicit Mealy Machine - Style 2

```
module FSM_style2 (...);  
    input ... ;    output ... ;  
    parameter size = ... ;  
    reg [size-1:0] state, next_state;  
    // Combinational logic for outputs and next state  
    assign outputs = ... // a function of state and inputs  
    always @ (state or inputs) begin  
        next_state = ... ;  
    end  
    // State transitions  
    always @ (posedge clk or negedge reset)  
        if (reset == 1'b0) state <= start_state;  
        else state <= next_state;  
endmodule
```

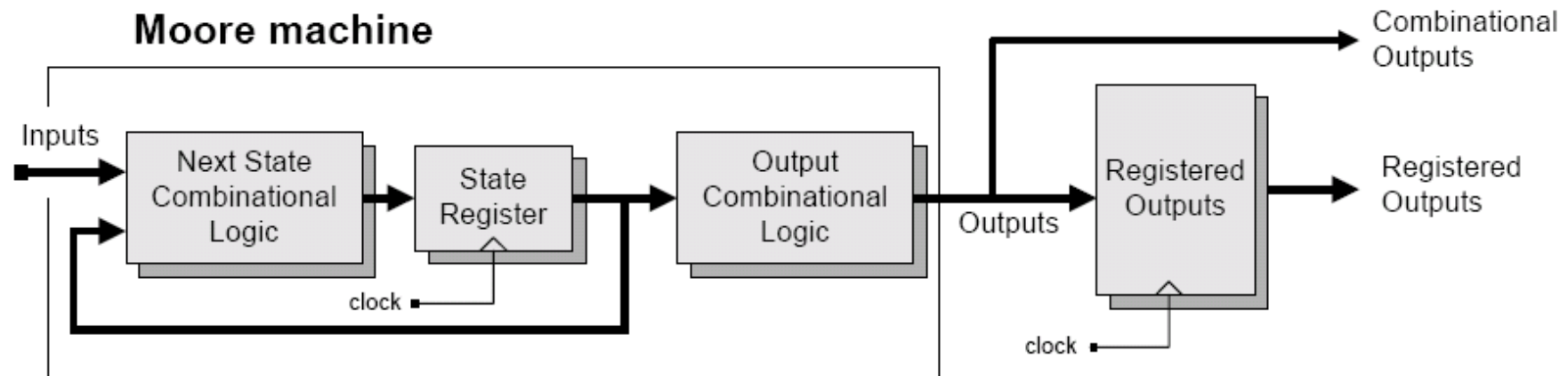


# FSMs with Registered Outputs



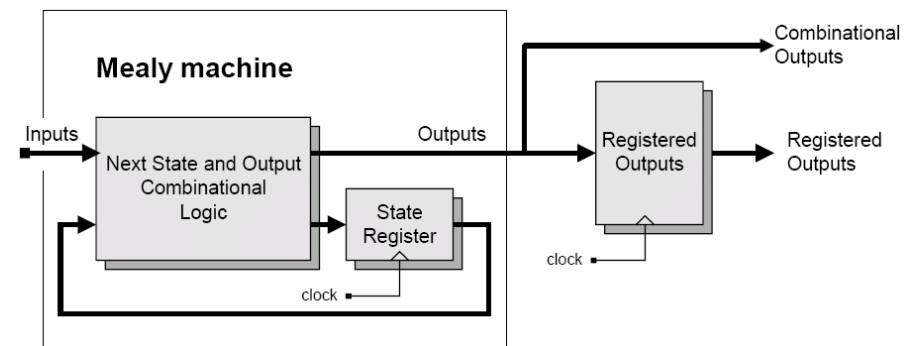
即時  
① 不穩定 - 雜訊

② 穩定  
非即時



# Mealy Machine with Registered Outputs

```
module FSM_style3 (...);  
input ... ;      output ... ;  
parameter size = ... ;  
reg [size-1:0]  state, next_state;  
// Combinational logic for next state  
always @ (state or inputs) begin  
    next_state = ... ;  
end  
// State transitions and registered outputs  
always @ (posedge clk or negedge reset)  
    if (reset == 1'b0) state <= start_state;  
    else begin  
        state <= next_state; outputs <= some_value (inputs, state);  
    end  
endmodule
```



# Up-Down Counter (1/3)

---

```
module up_down_explicit (clock, reset, up_dwn, count);
```

```
    input                clock, reset;
```

```
    input        [1:0]    up_dwn;
```

```
    output       [2:0]    count;
```

```
    reg          [2:0]    count, next_count;
```

```
    always @ (negedge clock or negedge reset)
```

```
        if (reset == 0)
```

```
            count = 3'b0;
```

```
        else
```

```
            count = next_count;
```

## Up-Down Counter (2/3)

---

```
always @ (up_dwn or count) begin  
  case (count)  
    0: case (up_dwn)  
      0, 3: next_count = 0;  
      1:   next_count = 1;  
      2:   next_count = 7;  
      default: next_count = 0;  
    endcase  
    1: case (up_dwn)  
      0, 3: next_count = 1;  
      1:   next_count = 2;  
      2:   next_count = 0;  
      default: next_count = 1;  
    endcase
```

```
    2: case (up_dwn)  
      0, 3: next_count = 2;  
      1:   next_count = 3;  
      2:   next_count = 1;  
      default: next_count = 2;  
    endcase  
    3: case (up_dwn)  
      0, 3: next_count = 3;  
      1:   next_count = 4;  
      2:   next_count = 2;  
      default: next_count = 3;  
    endcase
```

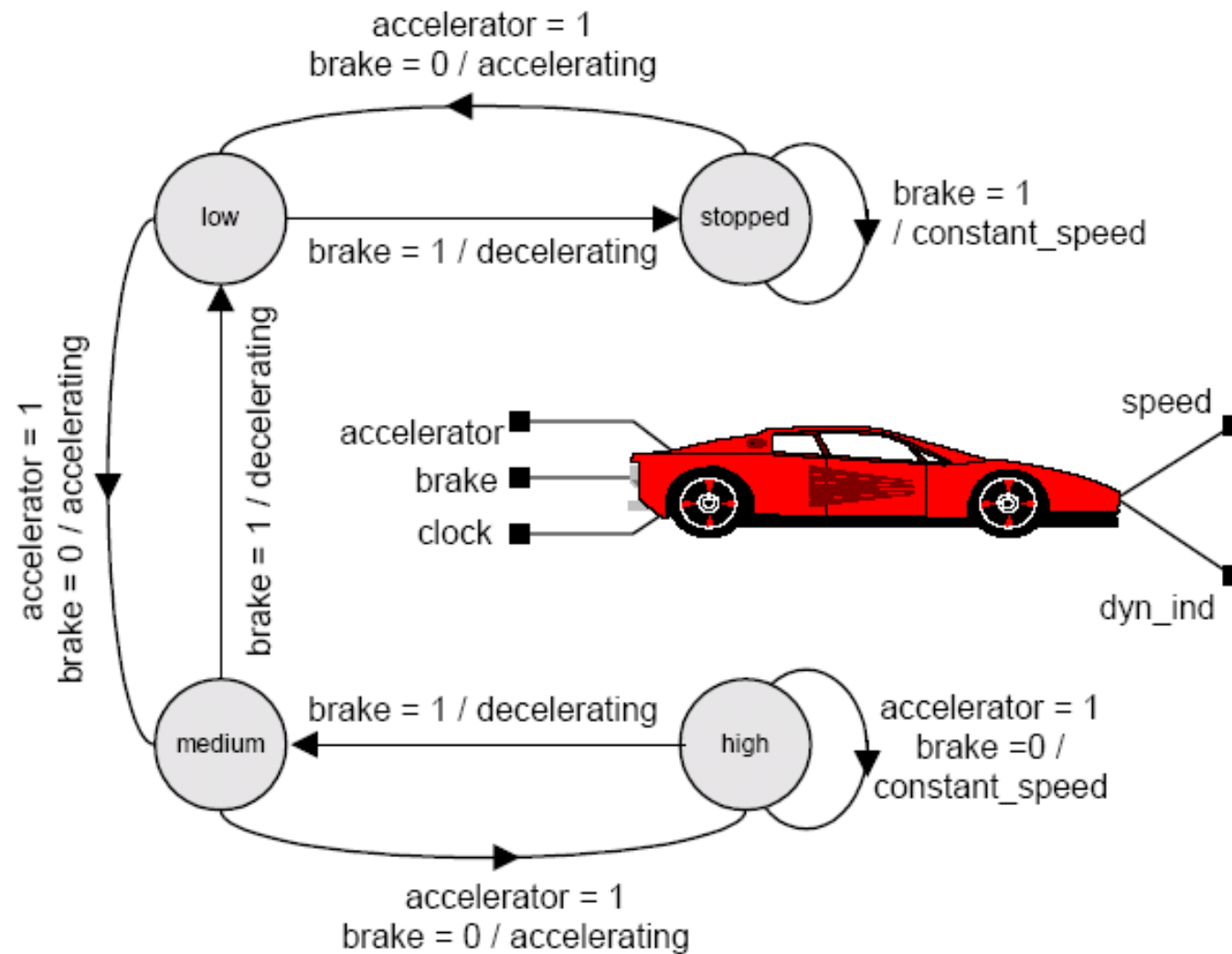
## Up-Down Counter (3/3)

---

4, 5, 6, 7:

```
    if (up_dwn == 0 || up_dwn == 3) next_count = count;
    else if (up_dwn == 1) next_count = count + 1;
    else if (up_dwn == 2) next_count = count - 1;
    else next_count = 0;
  endcase
end
endmodule
```

# Speed Machine (1/6)





## Speed Machine (2/6)

```
module speed_1 (clock, accelerator, brake, speed, dyn_ind);
```

```
    input          clock, accelerator, brake;
```

```
    output [1:0]    speed, dyn_ind;
```

```
    reg [1:0]       state, next_state;
```

```
    parameter stopped          = 2'b00;
```

```
    parameter S_low            = 2'b01, decelerating = 0;
```

```
    parameter S_medium        = 2'b10, constant_speed = 1;
```

```
    parameter S_high           = 2'b11, accelerating = 2;
```

```
// combinational part for output
```

```
assign speed = state;
```

輸出 輸入 clk 無關  $\Rightarrow$  Mealy Machine

## Speed Machine (3/6)

---

**assign** dyn\_ind =

brake ? (state == stopped) ? constant\_speed : decelerating:  
accelerator ? (state == S\_high) ? constant\_speed : accelerating :  
constant\_speed;

// registered part

**always** @ (posedge clock)

state = next\_state;

## Speed Machine (4/6)

---

```
// combinational part for next_state
always @ (state or accelerator or brake)
    if (brake == 1'b1)
        case (state)
            stopped:      next_state = stopped;
            S_low:        next_state = stopped;
            S_medium:     next_state = S_low;
            S_high:       next_state = S_medium;
            default:      next_state = stopped;
        endcase
```

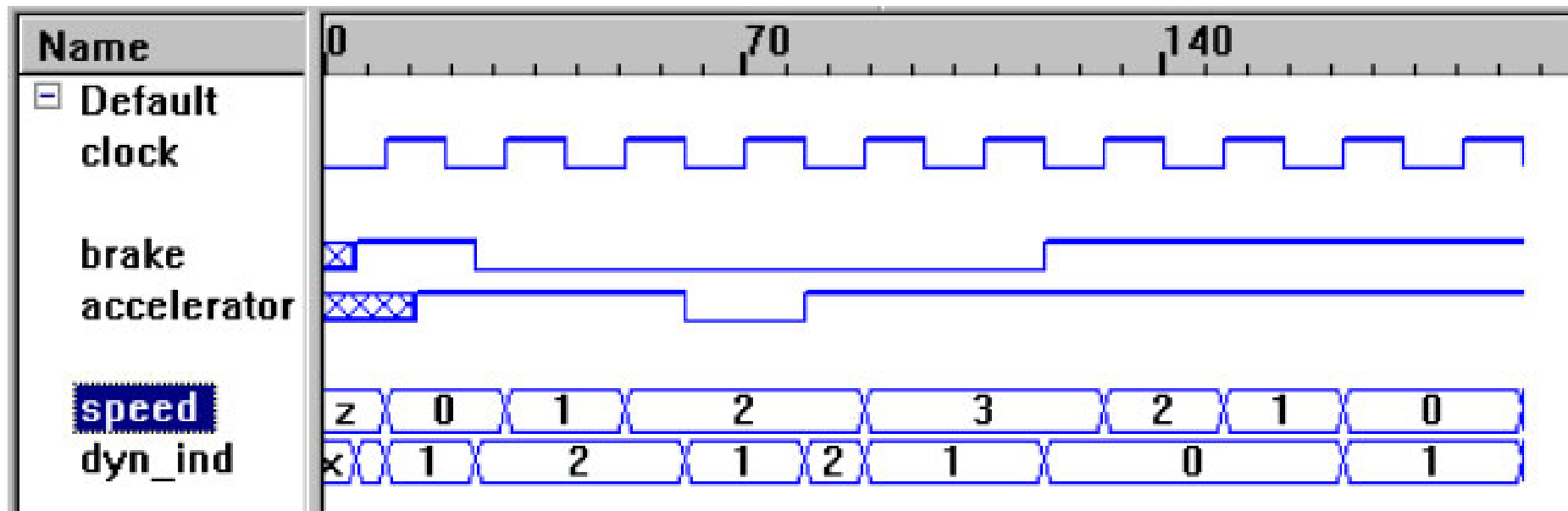
## Speed Machine (5/6)

---

```
    else if (accelerator == 1'b1)
        case (state)
            stopped:    next_state = S_low;
            S_low:      next_state = S_medium;
            S_medium:   next_state = S_high;
            S_high:     next_state = S_high;
            default:    next_state = stopped;
        endcase
    else
        next_state = state;
    endmodule
```

## Speed Machine (6/6)

---



# Implicit FSM

---

- Does not contain explicitly-declared **state registers**.
- The state is implied by the evolution of the **activity flow**.
- More abstract than the explicit state machine.
- Requires less code.
- Suitable only for machines in which *a given state can be reached from only one other state*.

# Up-Down Counter – Implicit 1

---

```
module up_down_implicit1 (clock, reset, up_dwn, count);  
    input          clock, reset;    input  [1:0]    up_dwn;  
    output         [2:0]    count; reg    [2:0]    count;  
  
    always @ (negedge clock or negedge reset)  
        if (reset == 0) count = 3'b0;  
        else if (up_dwn == 2'b00 || up_dwn == 2'b11) count = count;  
        else if (up_dwn == 2'b01) count = count + 1;  
        else if (up_dwn == 2'b10) count = count - 1;  
        else count = count;  
  
endmodule
```

## Up-Down Counter – Implicit 2

---

```
module up_down_implicit2 (clock, reset, up_dwn, count);  
    input          clock, reset;    input  [1:0]    up_dwn;  
    output        [2:0]    count; reg    [2:0]    count, next_count;  
  
    always @ (negedge clock or negedge reset)  
        if (reset == 0) count = 3'b0;  
        else count = next_count;  
    always @ (up_dwn or count) begin  
        if (up_dwn == 2'b00 || up_dwn == 2'b11) next_count = count;  
        else if (up_dwn == 2'b01) next_count = count + 1;  
        else if (up_dwn == 2'b10) next_count = count - 1;  
        else next_count = count;  
    end  
endmodule
```



# Speed Machine – Implicit 1 (1/3)

---

```
module speed_2 (clock, accelerator, brake, speed);
```

```
    input                clock, accelerator, brake;
```

```
    output      [1:0]    speed;
```

```
    reg         [1:0]    speed;
```

```
    parameter stopped      = 2'b00;
```

```
    parameter S_low        = 2'b01;
```

```
    parameter S_medium     = 2'b10;
```

```
    parameter S_high       = 2'b11;
```

## Speed Machine – Implicit 1 (2/3)

---

```
always @ (posedge clock)
    if (brake == 1'b1)
        case (speed)
            stopped:      speed <= stopped;
            S_low:        speed <= stopped;
            S_medium:     speed <= S_low;
            S_high:       speed <= S_medium;
            default:      speed <= stopped;
        endcase
```

## Speed Machine – Implicit 1 (3/3)

---

```
    else if (accelerator == 1'b1)
        case (speed)
            stopped:      speed <= S_low;
            S_low:        speed <= S_medium;
            S_medium:     speed <= S_high;
            S_high:       speed <= S_high;
            default:      speed <= stopped;
        endcase
    else
        speed <= speed;
    endmodule
```

## Speed Machine – Implicit 2 (1/2)

---

```
module speed_3 (clock, accelerator, brake, speed);
```

```
    input                clock, accelerator, brake;
```

```
    output      [1:0]    speed;
```

```
    reg         [1:0]    speed;
```

```
    parameter stopped      = 2'b00;
```

```
    parameter S_low        = 2'b01;
```

```
    parameter S_medium     = 2'b10;
```

```
    parameter S_high       = 2'b11;
```

## Speed Machine – Implicit 2 (2/2)

---

```
always @ (posedge clock)  
    case (speed)  
        stopped: if (brake == 1'b1)                speed <= stopped;  
                else if (accelerator == 1'b1)        speed <= S_low;  
        S_low: if (brake == 1'b1)                    speed <= stopped;  
                else if (accelerator == 1'b1)        speed <= S_medium;  
        S_medium: if (brake == 1'b1)                 speed <= S_low;  
                else if (accelerator == 1'b1)        speed <= S_high;  
        S_high: if (brake == 1'b1)                  speed <= S_medium;  
                else if (accelerator == 1'b1)        speed <= S_high;  
        default:                                    speed <= stopped;  
    endcase  
endmodule
```

# Reference

---

1. 教師自製
2. Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL, Michael D. Ciletti, ISBN: 0139773983, Prentice Hall, 1999.

