Handout for New Engineers

Logic Design Workshop

Part I Basic Knowledge

Renesas Design Vietnam Co. Ltd.

Created by:	Dr. Keijiro Hayashi

Reviewed by:	Dr. Hai Pham

Throughout this text material

○ and ◎ mean good and recommended.

△ means not recommended but allowed in certain cases.

× means bad/wrong and must not be used.

In ver33r00, many pages have been changed and the revision marks formerly given on the upper right corner of revised pages were removed.

Your daily small effort makes you an excellent designer.

A small amount of your everyday's effort, when accumulated for weeks, months, and years, will bring you into a very much advanced technical world where those who spared such small effort can not join.

Help others and be helped by others.

Share your skill, knowledge, and information with others and help them improve themselves. You will be helped much by those people who have improved themselves with your help.

Part I Basic Knowledge

Index

Part I Basic Knowledge

Index

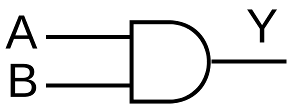
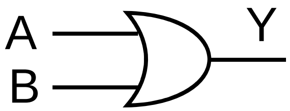
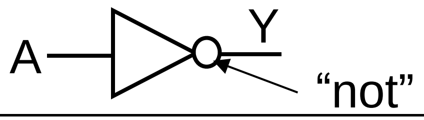
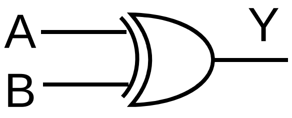
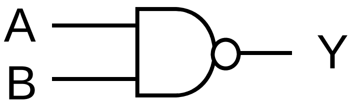
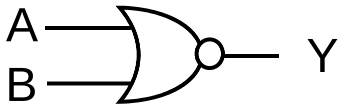
Part I Basic Knowledge

Index

1. Fundamentals

1.1 Various notation of logic gates

Fundamental logical gates and their operation

Name	Text book style	gate expression (MIL Logical notation)	operation
AND	$Y = A\,B$		Y is 1 only when both A and B are 1.
OR	$Y = A + B$		Y is 1 when A or B is 1.
NOT	$Y = \overline{A}$		Y is 1 only when A is 0.
EOR	$Y = \overline{A}\,B + A\,\overline{B}$		Y is 1 only when either one of A or B is 1 while the other is 0.
NAND	$Y = \overline{A\,B}$		Y is 1 when A or B is 0.
NOR	$Y = \overline{A + B}$		Y is 1 only when both A and B are 0.

Beside the MIL Logic notation and text book style notation, there are other notations commonly used in logic design.
Make yourself familiar with any type of logic notation so that you can freely use any style to take advantage of each notation.

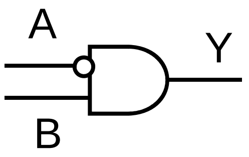
(1) Truth table

Suitable for small scale logic and can be mapped directly to RTL code.

2 inputs truth table example

A	B	Y
0	0	0
1	0	0
0	1	1
1	1	0

equivalent



3 inputs truth table example

a	b	c	y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	x	0	1
1	x	1	0

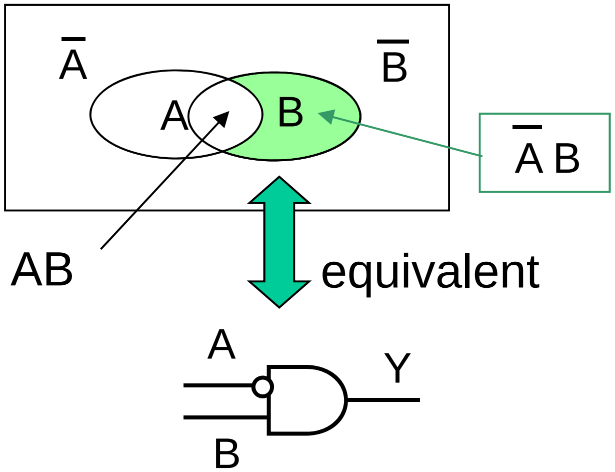
x: don't care

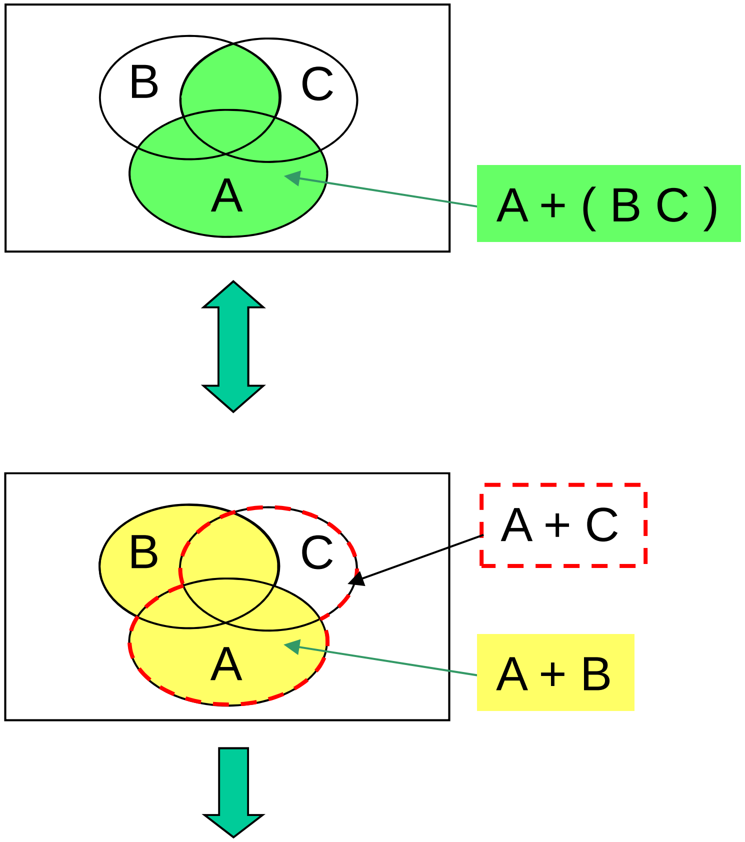
mapping to RTL code

```
always @(a or b or c)
begin
  casez ( {a,b,c} )
    3’b000 : y = 1 ;
    3’b001 : y = 0 ;
    3’b010 : y = 0 ;
    3’b011 : y = 1 ;
    3’b1?0 : y = 1 ;
    3’b1?1 : y = 0 ;
  endcase
end
```

(2) Venn diagram

Useful to illustrate simple set of relationships in logic.

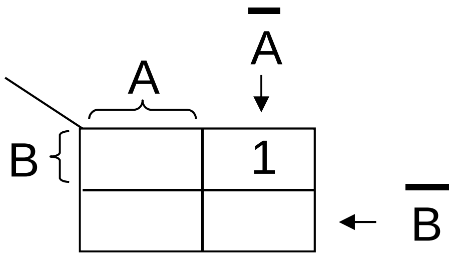




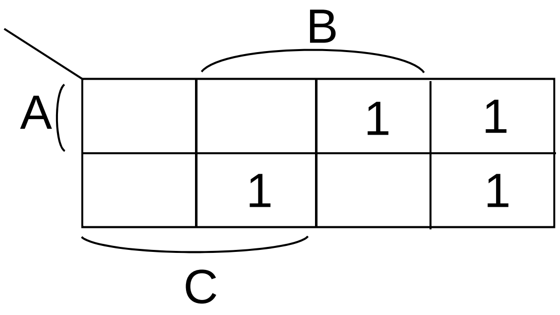
A + (BC) = (A+B)(A+C)

(3) Karnaugh map

Useful to hand-optimize logic gates.



Karnaugh map for $\overline{A}$ B



3 input Karnaugh Map example

same

BC A	01	11	10	00
1			1	1
0		1		1

Hand-optimize using Karnaugh map, example:

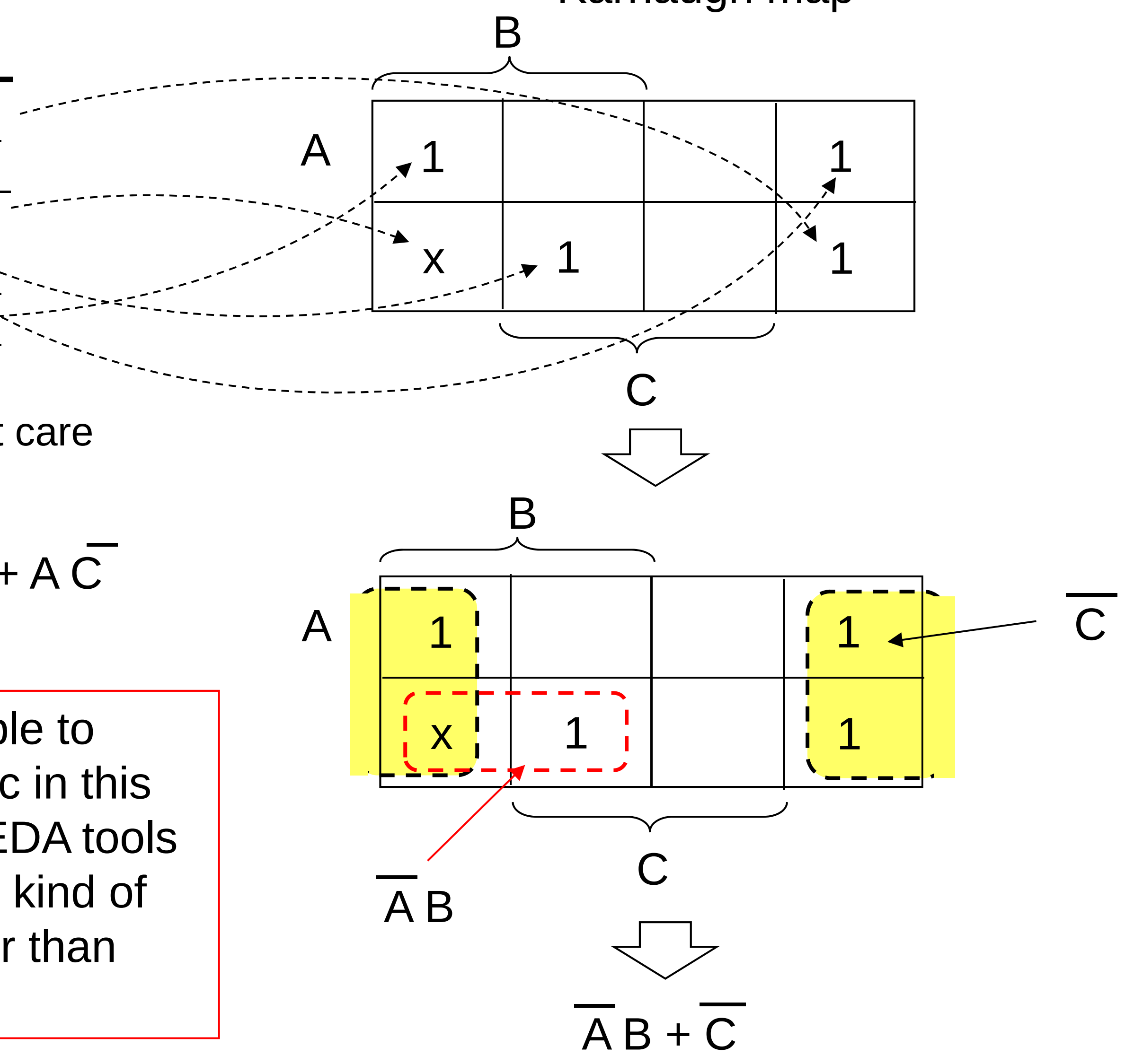
Truth table

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	x
0	1	1	1
1	x	0	1
1	x	1	0

x: don't care

$$Y = \overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,B\,C + A\,\overline{C}$$

Karnaugh map



$$\overline{A}\,B + \overline{C}$$

You have to be able to optimize your logic in this way, but today’s EDA tools can take care this kind of optimization better than human.

(4) Boolean algebra expression

$$C = \neg A \wedge B$$

$$Y1 = (A \wedge \neg C) \vee (\neg A \wedge \neg(\neg B \wedge C) \vee (B \wedge \neg C))$$

(5) Verilog RTL expression

```
assign C = (~A )& B ;

assign Y1 = ( A & (~C ) ) | ( (~A ) & ~( B ^ C ) ) ;
```

1.2 Basic theory

$$\left.\begin{aligned}&AB = BA\\&A{+}B = B{+}A\end{aligned}\right\}\ \text{commutativity}$$

$$\left.\begin{aligned}&A\,(B\,C) = (A\,B)\,C\\&A + (B + C) = (A + B) + C\end{aligned}\right\}\ \text{associativity}$$

$$\left.\begin{aligned}&A\,(B + C) = AB + AC\\&A + (B\,C) = (A{+}B)\,(A{+}C)\end{aligned}\right\}\ \text{distributivity}$$

$$\left.\begin{aligned}&A + A = A\\&AA = A\end{aligned}\right\}\ \text{idenpotency}$$

$$\left.\begin{aligned}&A + 1 = 1\\&A + 0 = A\\&A1 = A\\&A0 = 0\end{aligned}\right\}\ \text{boundedness}$$

$$\left.\begin{aligned}&A + \overline{A} = 1\\&A\,\overline{A} = 0\end{aligned}\right\}\ \text{complements}$$

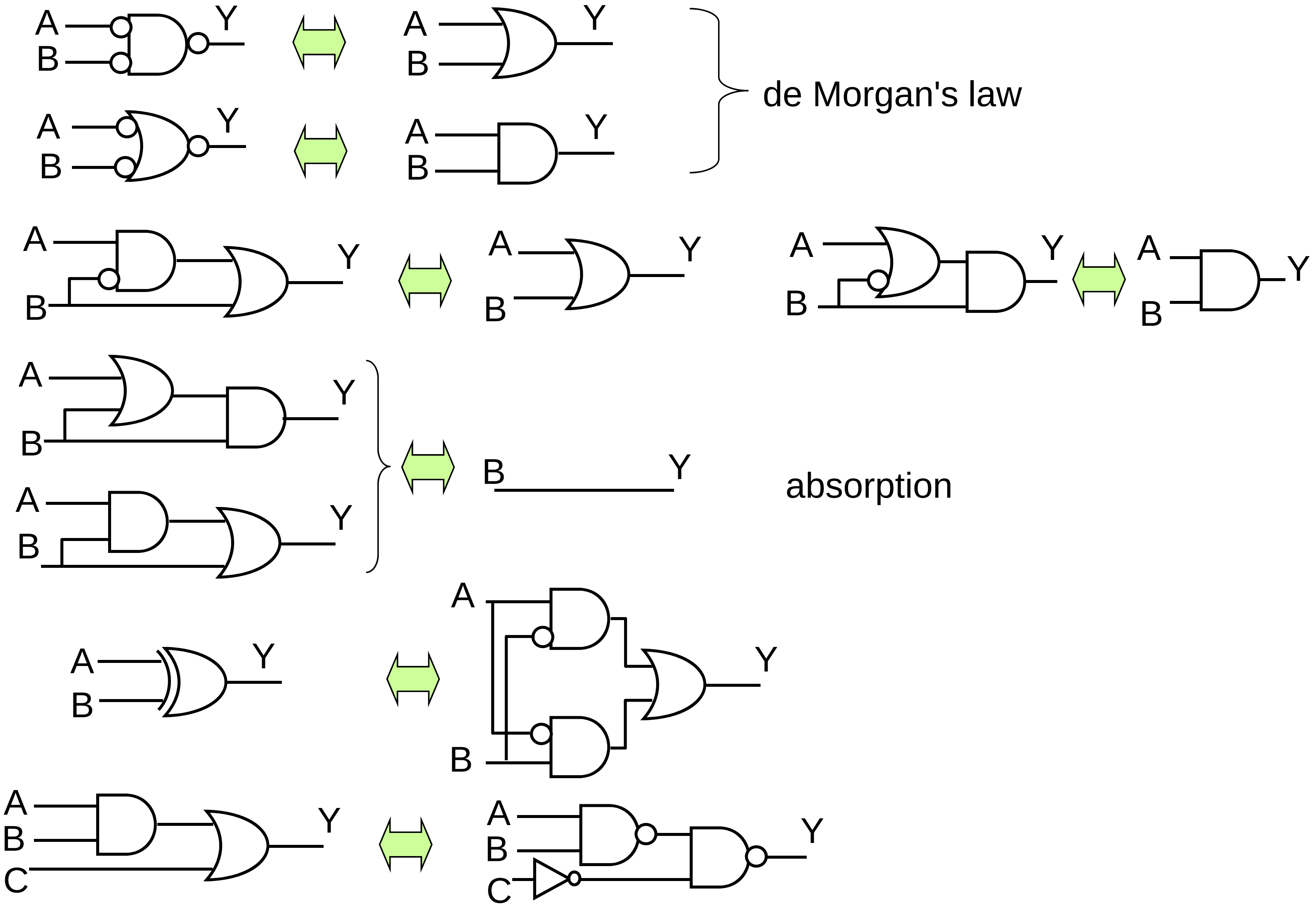
$$\left.\begin{aligned}&\overline{A\,B} = \overline{A} + \overline{B}\\&\overline{A}\,\overline{B} = \overline{A + B}\end{aligned}\right\}\ \text{de Morgan's law}$$

$$\left.\overline{\overline{A}} = A\ \right\}\ \text{involution}$$

$$\left.\begin{aligned}&\overline{1} = 0\\&\overline{0} = 1\end{aligned}\right\}\ \text{0 and 1 are complements}$$

$$\left.\begin{aligned}&A\,(A{+}B) = A\\&A + (A\,B) = A\end{aligned}\right\}\ \text{absorption}$$

Basic theory in gate level logic



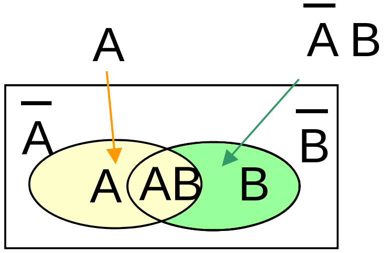
de Morgan's law

absorption

Example 1-1: Simplify the following logic

$A + \overline{A}\,B = ?$

(1) by using Venn diagram



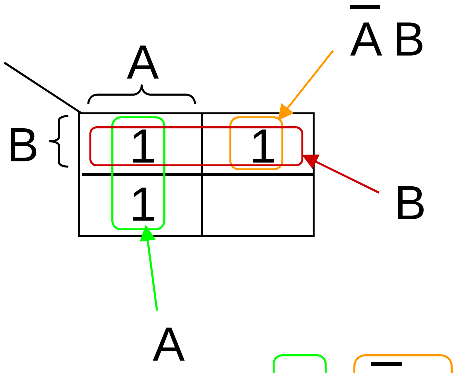
$A + \overline{A}\,B = A + B$

(2) by using Truth table

A	B	$\overline{A}\,B$	$A+\overline{A}\,B$
0	0	0	0
1	0	0	1
0	1	1	1
1	1	0	1

A + B

(3) by using Karnaugh map



$\boxed{A}+\boxed{\overline{A}\,B}=\boxed{A}+\boxed{B}$

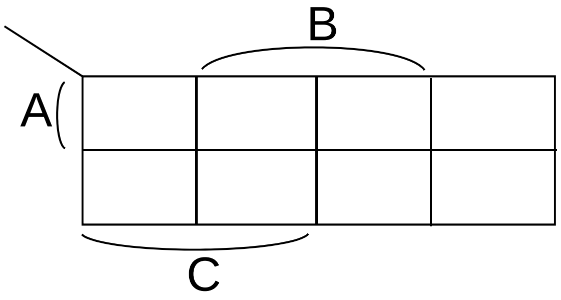
(4) by applying Logical theory

$$\begin{aligned}
A + \overline{A}\,B &= A\,1 + \overline{A}\,B \\
&= A\,\overbrace{(1 + B)} + \overline{A}\,B \\
&= A\ 1 + A\,B + \overline{A}\,B \\
&= A + \underbrace{(A + \overline{A})}\,B \\
&= A + 1\ B \\
&= A + B
\end{aligned}$$

Q 1.2-1: Simplify the following logic by using a Karnaugh map, truth table, and logic theory.

$$(\overline{A\,B}\,C + \overline{A\,C})(\overline{B}\,C + B\,\overline{C})(A + \overline{B}) =$$

by using Karnaugh map



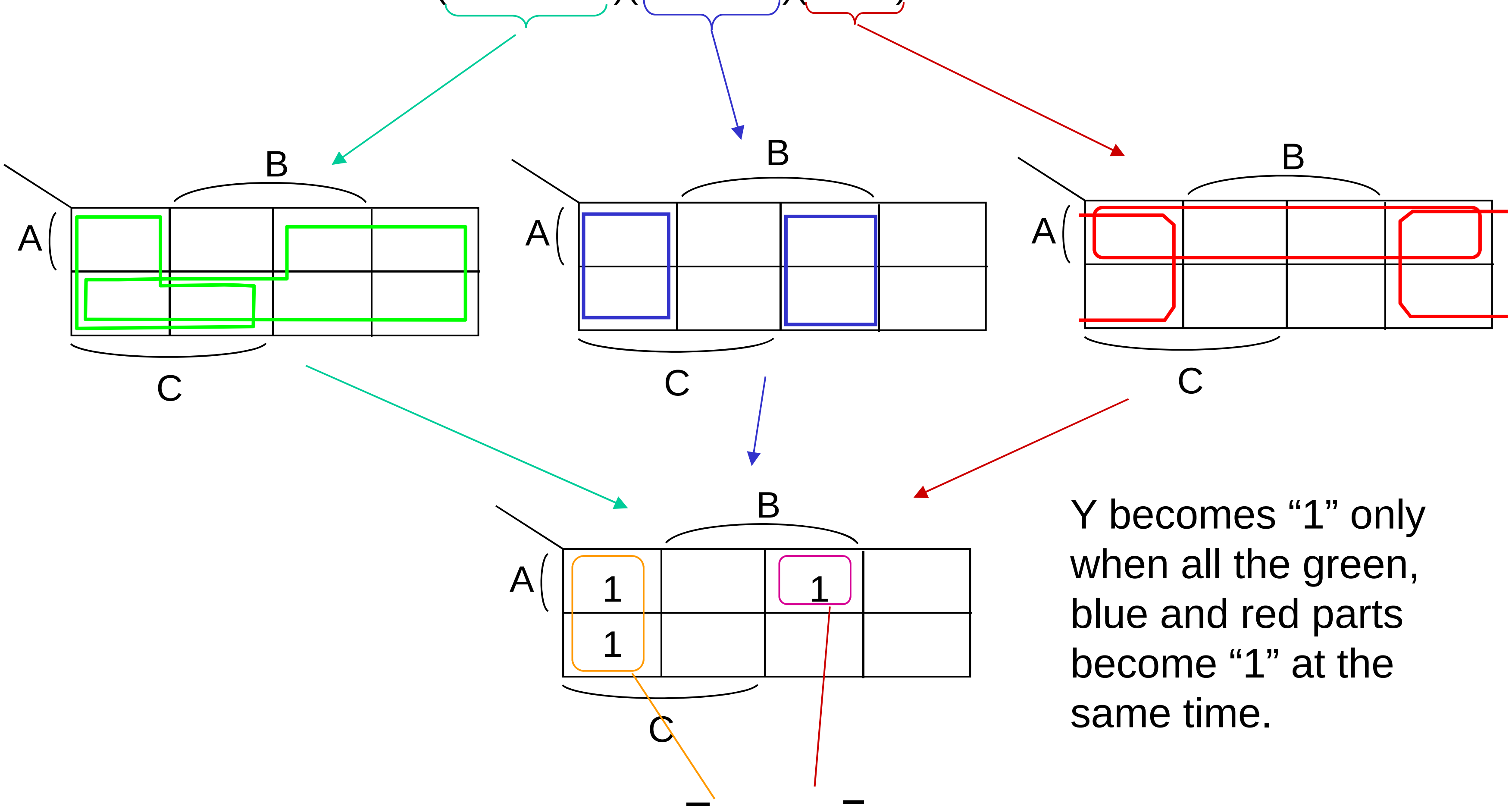
by using Truth table

A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Q1.2-1 A sample answer

(1) Karnaugh map

$$(\overline{AB}C + \overline{AC})(\overline{B}C + B\overline{C})(A + \overline{B}) = Y$$



$$= \overline{B}C + AB\overline{C}$$

Y becomes “1” only when all the green, blue and red parts become “1” at the same time.

Q1.2-1 A sample answer (Continued)

(2) truth table

$$\underbrace{(\overline{AB}C + \overline{AC})}\underbrace{(\overline{B}C + B\overline{C})}\underbrace{(A + \overline{B})} =$$

A	B	C	$\overline{AB}C$	$\overline{AC}$	$\overline{AB}C+\overline{AC}$	$\overline{B}C+B\overline{C}$	$A+\overline{B}$	Y
0	0	0	0	1	1	0	1	0
0	0	1	1	1	1	1	1	1
0	1	0	0	1	1	1	0	0
0	1	1	1	1	1	0	0	0
1	0	0	0	1	1	0	1	0
1	0	1	1	0	1	1	1	1
1	1	0	0	1	1	1	1	1
1	1	1	0	0	0	0	1	0

$$= \overline{B}C + AB\overline{C}$$

Q1.2-1 A sample answer (Continued)

(3) logic theory

$$(\overline{AB}C + \overline{AC})(\overline{B}C + B\overline{C})(A + \overline{B})$$

$$= ((\overline{A} + \overline{B})\,C + \overline{A} + \overline{C})(\overline{B}C + B\overline{C})(A + \overline{B})$$

$$= ((\overline{A}\,C + \overline{B}\,C + \overline{A} + \overline{C})(\overline{B}C + B\overline{C})(A + \overline{B})$$

$$= ((\overline{A} + \overline{B}\,C + \overline{C})\,(A + \overline{B})\,(\overline{B}C + B\overline{C})$$

$$= (\cancel{\overline{A}\,A} + A\,\overline{B}\,C + A\,\overline{C} + \overline{A}\,\overline{B} + \overline{B}\,\overline{B}\,C + \overline{B}\,\overline{C})\,(\overline{B}C + B\overline{C})$$

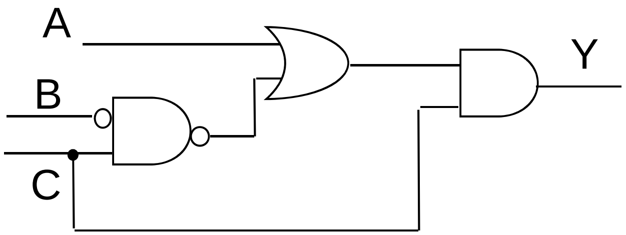
$$= (\overline{B}\,C + A\,\overline{C} + \overline{A}\,\overline{B} + \overline{B}\,\overline{C})\,(\overline{B}C + B\overline{C})$$

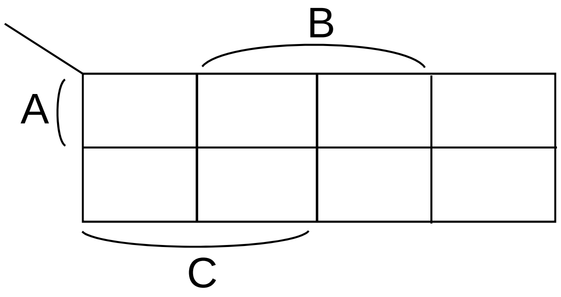
$$= \overline{B}\,C\,\overline{B}\,C + \cancel{A\,\overline{C}\,\overline{B}\,C} + \overline{A}\,\overline{B}\,\overline{B}\,C + \cancel{\overline{B}\,\overline{C}\,\overline{B}\,C} + \cancel{\overline{B}\,C\,B\,\overline{C}} + A\,\overline{C}\,B\,\overline{C} + \cancel{\overline{A}\,\overline{B}\,B\,\overline{C}} + \cancel{\overline{B}\,\overline{C}\,B\,\overline{C}}$$

$$= \overline{B}\,C + \overline{A}\,\overline{B}\,C + A\,B\,\overline{C}$$

$$= \overline{B}\,C + A\,B\,\overline{C}$$

Q 1.2-2: Simplify the following logic by using a Karnaugh map, truth table, and logic theory.

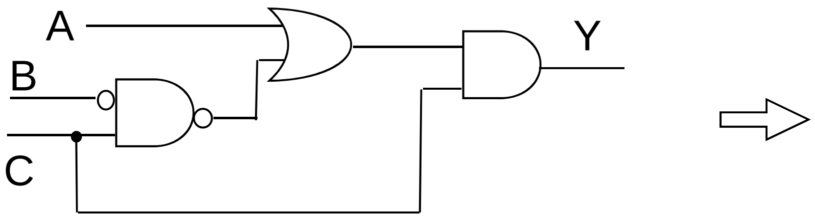




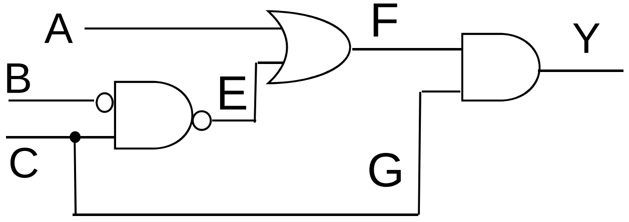
A	B	C	Y
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

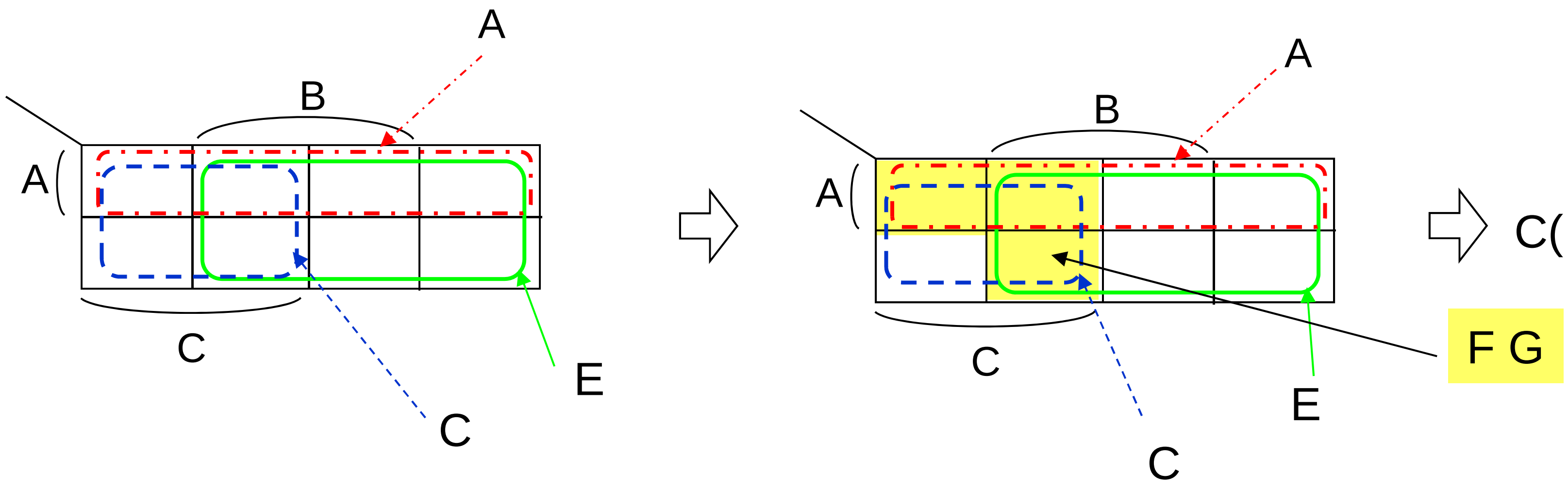
Q1.2-2 : A sample answer

(1) Karnaugh map



Define internal signals

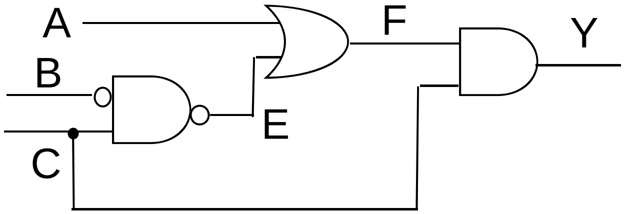




C(A + B)

Q1.2-2 : A sample answer (Continued)

(2) truth table



⇨

A	B	C	E	F	Y
0	0	0	1	1	0
0	0	1	0	0	0
0	1	0	1	1	0
0	1	1	1	1	1
1	0	0	1	1	0
1	0	1	0	1	1
1	1	0	1	1	0
1	1	1	1	1	1

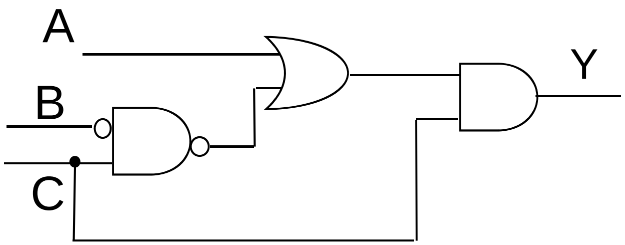
AC

BC

Y = AC + BC

Q1.2-2 : A sample answer (Continued)

(3) logic theory



$$Y = \left(A + \left(\overline{\overline{B}\ C} \right) \right) C$$

← distributivity

$$= A\,C + \overline{\overline{B}\,C}\ C$$

← de Morgan's law

$$= A\,C + (B + \overline{C})\,C$$

← distributivity

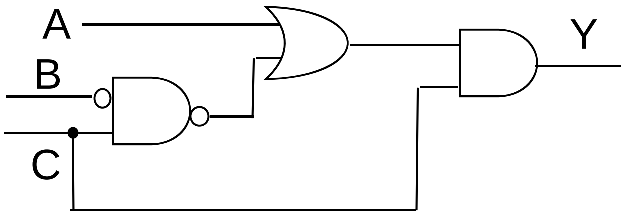
$$= A\,C + B\,C + \overline{C}\,C$$

← complements

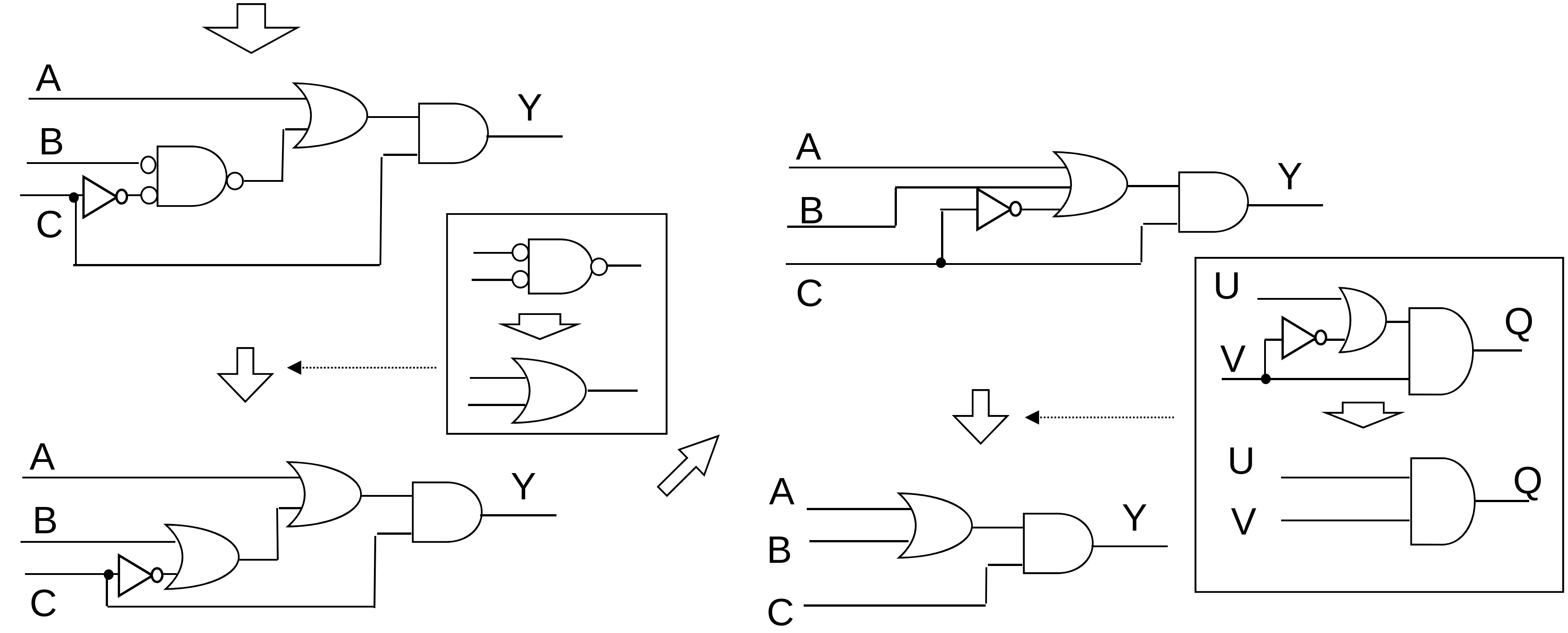
$$= A\,C + B\,C$$

Q1.2-2 : A sample answer (Continued)

(4) direct gate level conversion



Gate level conversion is not a practical method to optimize the logic, but you must make yourself so familiar with gate level drawings that you can do it manually.



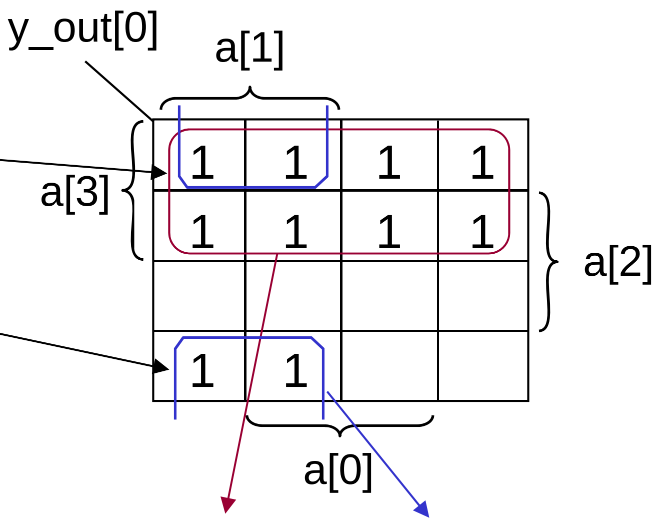
Q 1.2-3: Write equivalent gates for the following truth table.

a[3]	a[2]	a[1]	a[0]	y_out
1	x	x	x	3'b011
0	1	x	x	3'b010
0	0	1	x	3'b001
0	0	0	1	3'b000
0	0	0	0	3'b100

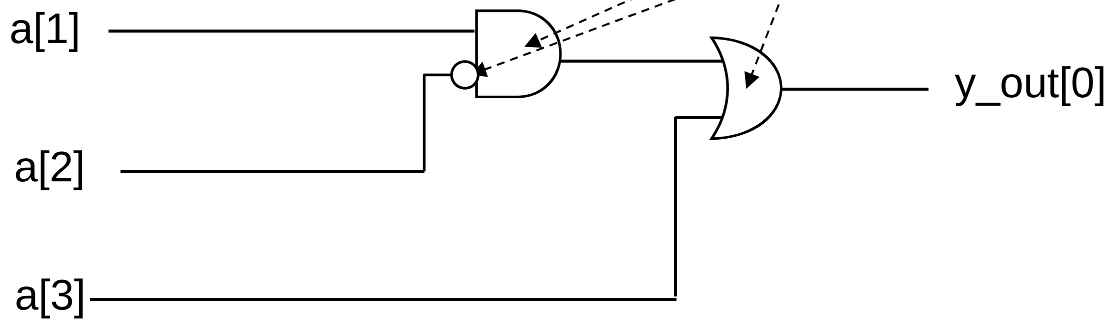
Q 1.2-3: A sample answer

First, focus on bit 0.

a[3]	a[2]	a[1]	a[0]	y_out
1	x	x	x	3'b011
0	1	x	x	3'b010
0	0	1	x	3'b001
0	0	0	1	3'b000
0	0	0	0	3'b100



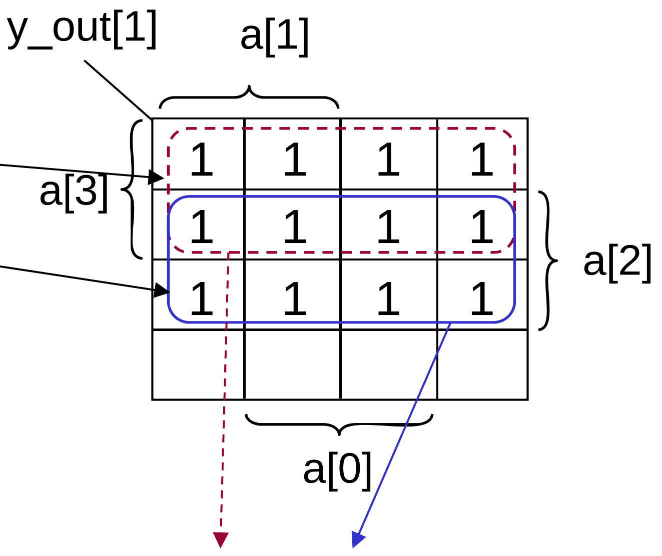
y_out[0] = a[3] | (a[1] & ~a[2])



Q1.2-3 : A sample answer. (continued)

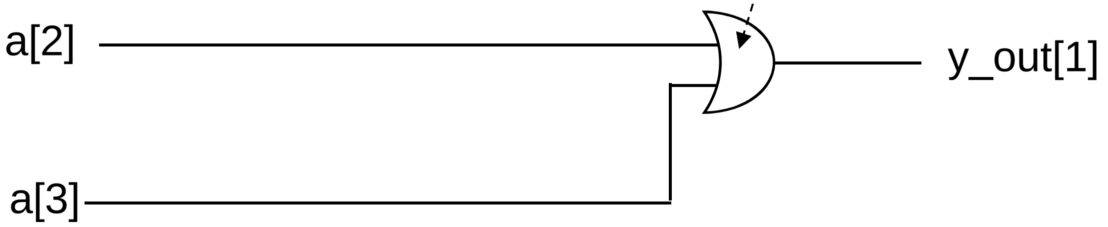
Next, focus on bit 1.

a[3]	a[2]	a[1]	a[0]	y_out
1	x	x	x	3'b011
0	1	x	x	3'b010
0	0	1	x	3'b001
0	0	0	1	3'b000
0	0	0	0	3'b100



1	1	1	1
1	1	1	1
1	1	1	1

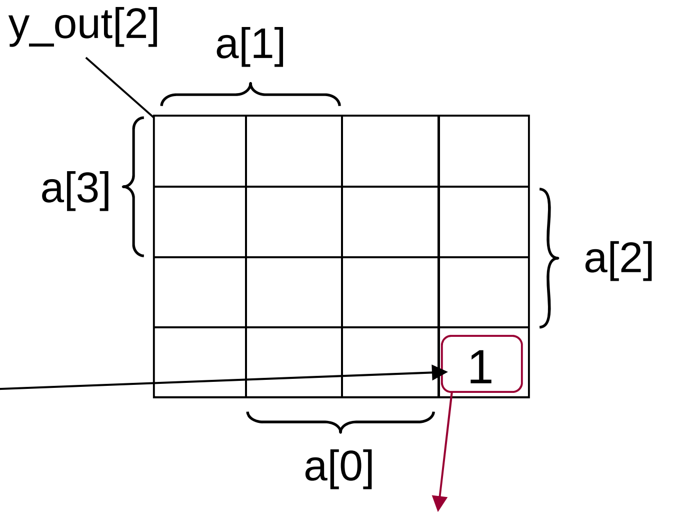
y_out[1] = a[3] | a[2]



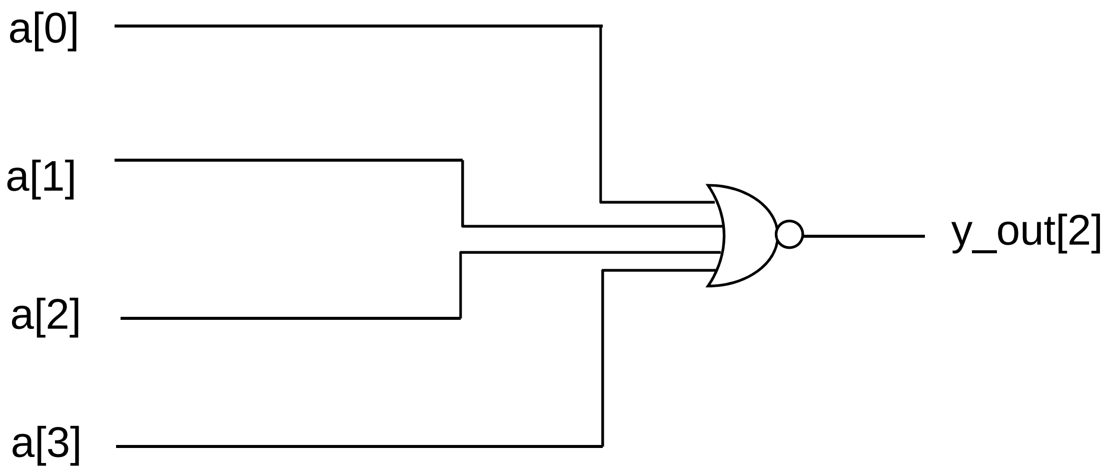
Q1.2-3 : A sample answer. (continued)

Lastly, check bit 2.

a[3]	a[2]	a[1]	a[0]	y_out
1	x	x	x	3'b011
0	1	x	x	3'b010
0	0	1	x	3'b001
0	0	0	1	3'b000
0	0	0	0	3'b100



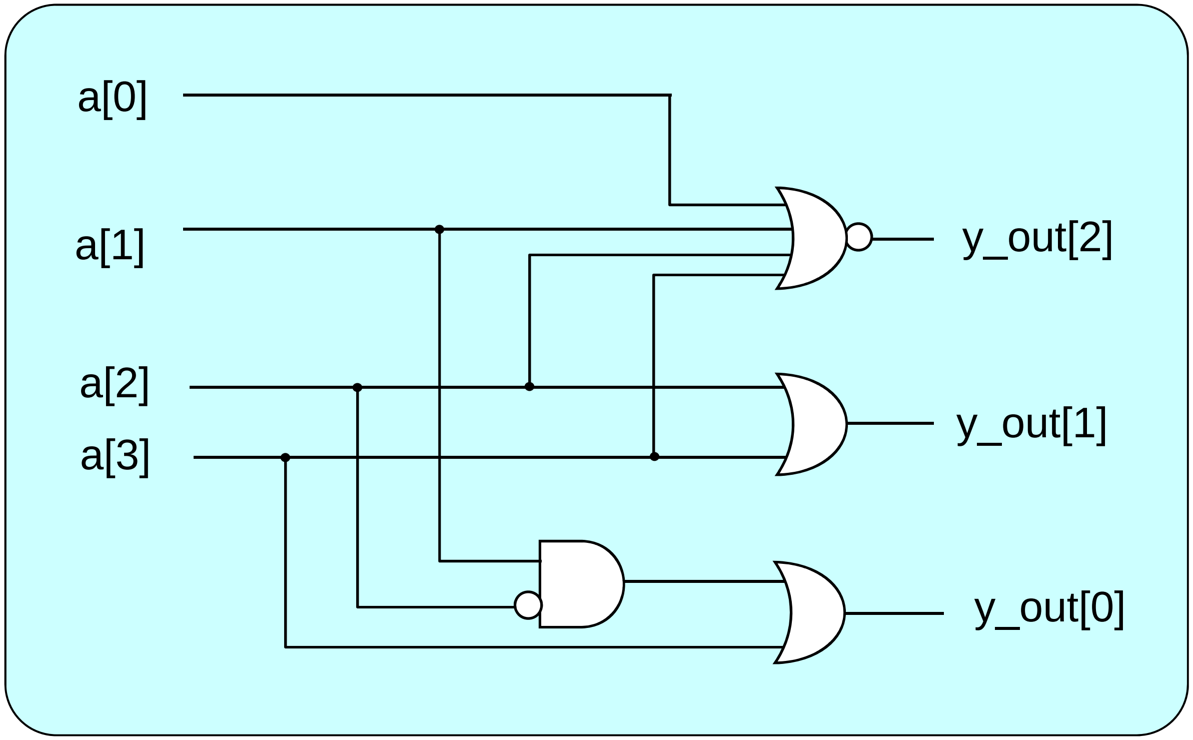
y_out[2] = ~(a[0] | a[1] | a[2] | a[3])



Q1.2-3 : A sample answer. (continued)

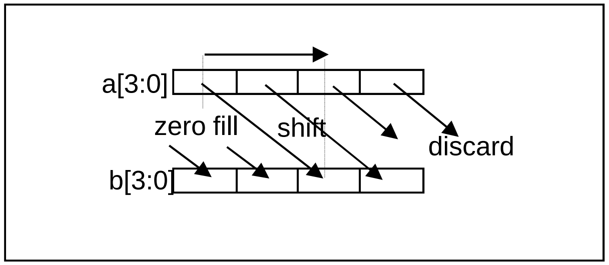
Then, we get the total structure as shown below.

a[3]	a[2]	a[1]	a[0]	y_out
1	x	x	x	3'b011
0	1	x	x	3'b010
0	0	1	x	3'b001
0	0	0	1	3'b000
0	0	0	0	3'b100

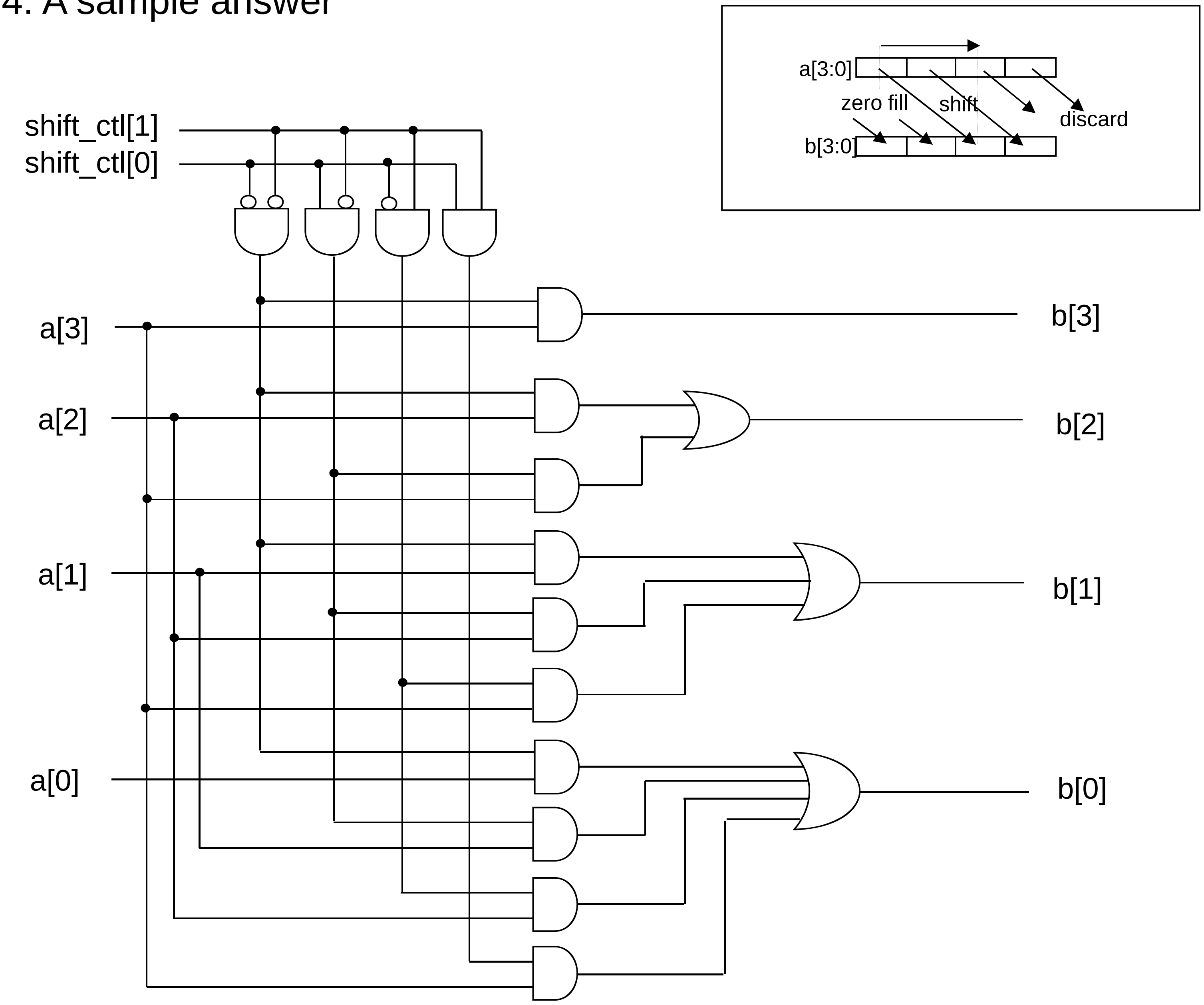


Q1.2-4: Create a gate diagram equivalent to the following truth table, shifter.

shift_ctl[1]	shift_ctl[0]	b[3]	b[2]	b[1]	b[0]
0	0	a[3]	a[2]	a[1]	a[0]
0	1	0	a[3]	a[2]	a[1]
1	0	0	0	a[3]	a[2]
1	1	0	0	0	a[3]



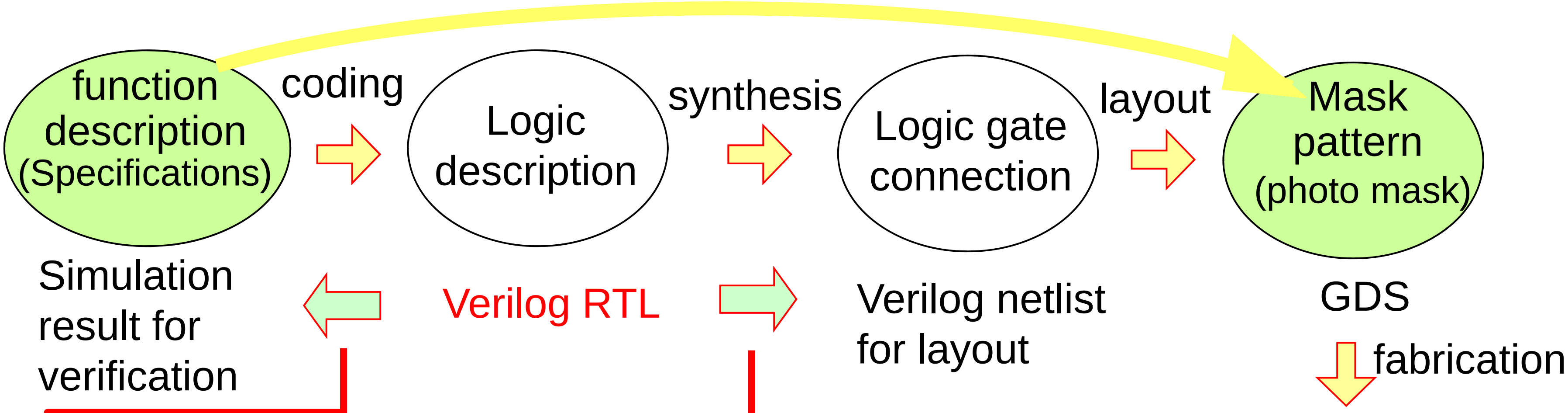
Q1.2-4: A sample answer



2. Hardware design and development process

2.1 Objective of hardware design

An objective of hardware design = Get photo masks, equivalent to specifications, for fabrication.



function description (Specifications) → coding → Logic description → synthesis → Logic gate connection → layout → Mask pattern (photo mask)

Simulation result for verification ← Verilog RTL → Verilog netlist for layout

GDS → fabrication → LSI chips to sell

RTL code must be "**simulatable**".

We have to write RTL code so that an EDA tool can simulate the code for verification.

RTL code must be "**synthesizable**".

We have to write RTL code so that an EDA tool can create a netlist from the code.

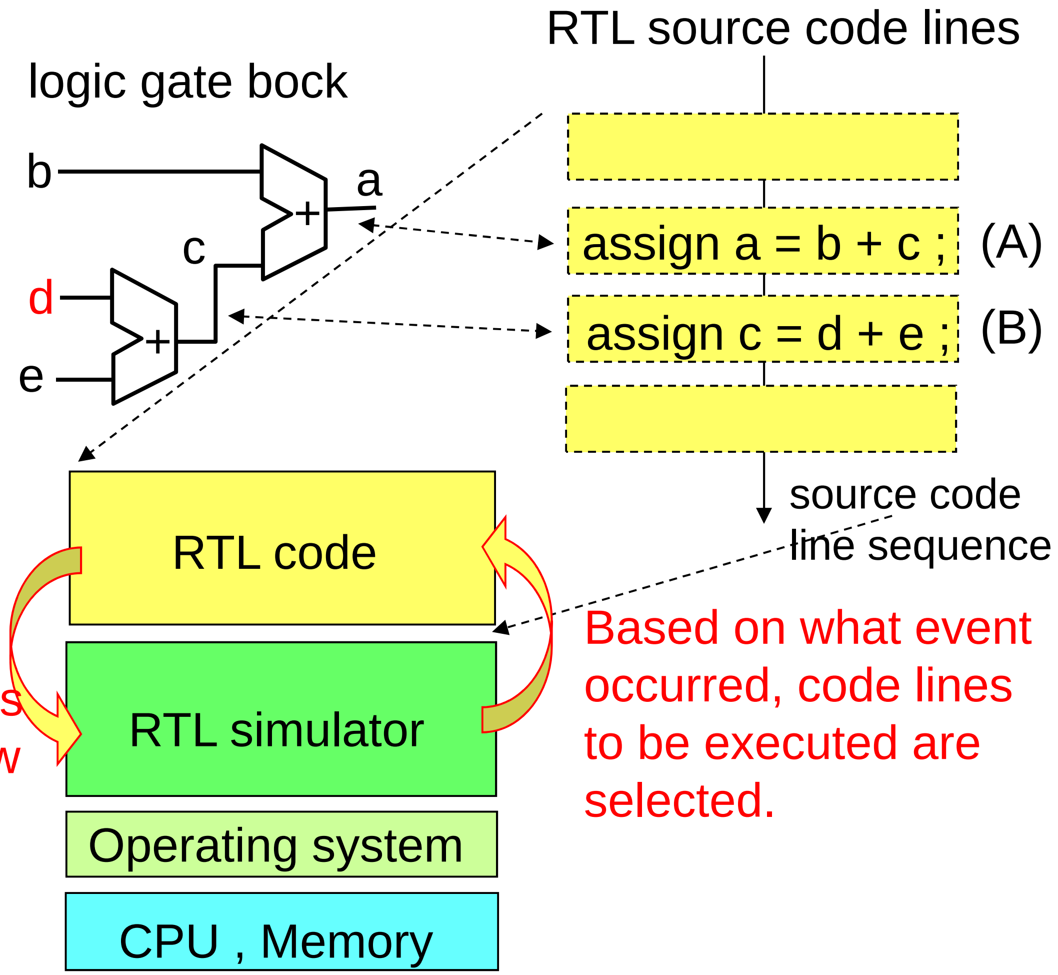
2.2 Logic simulation

An RTL simulator simulates how logic signals written in Verilog RTL language continue to change as time goes on.
Therefore, RTL source code lines are not executed in a sequential order they are written, but executed in a order of events such as changes of signal values.

When c and e do not change but d changes, (B) must be executed first to get the value of b, and then (A) must be executed using the value of b and c to get the value of a.

Which lines are to be executed is determined by what event occurred.

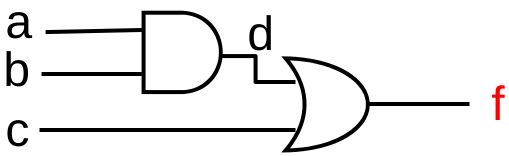
Execution of RTL code lines generates new events.



Based on what event occurred, code lines to be executed are selected.

Take another example to see how a simulator executes Verilog RTL code lines in simulation.

gate diagram

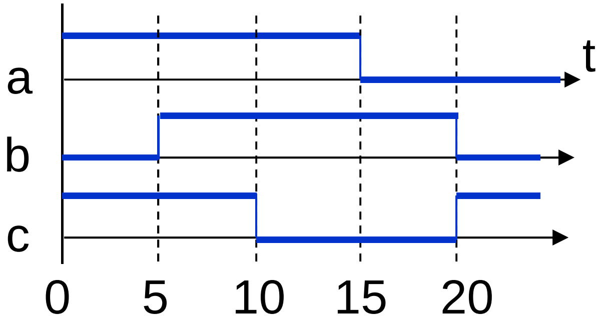


Verilog RTL source program

```
assign d = a & b ; // <1>
assign f = d | c ;   // <2>
```

logic description

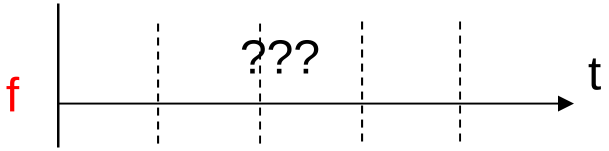
time chart



```
initial a = 1;
initial b =0 ;
initial c = 1;
initial    #5 b = 1 ;
initial  #10 c = 0 ;
initial  #15 a = 0 ;
initial  #20 b = 0 ;
initial  #20 c = 1 ;
```

initial values

simulation



See next page to find how to calculate the value of f ??

Verilog RTL source program

```
assign d = a & b ; // <1>
assign f = d | c ;   // <2>
```

<1> → If a or b does not change then d will not change.

<2> → If c or d does not change then f will not change.

```
initial a = 1;
initial b =0 ;
initial c = 1;
initial    #5 b = 1 ;
initial  #10 c = 0 ;
initial  #15 a = 0 ;
initial  #20 b = 0 ;
initial  #20 c = 1 ;
```

simulation process

At t=0, execute <1> and <2> to get the values of d (d=0) and f (f=1).

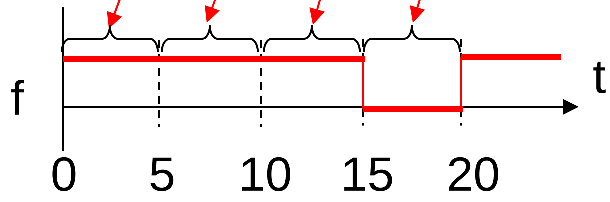
At t=5, execute <1> and <2> to get the values of d (d=1) and f (f=1).

At t=10, execute <2> to get the value of f (f=1). No need to execute <1> because only c changes.

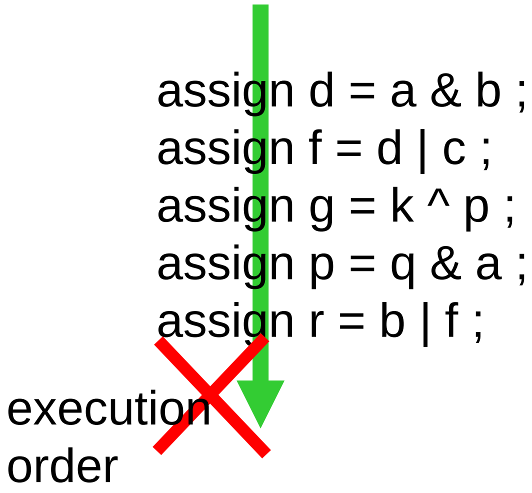
At t=15, execute <1> and <2> to get the values of d (d=0) and f (f=0).

⋮

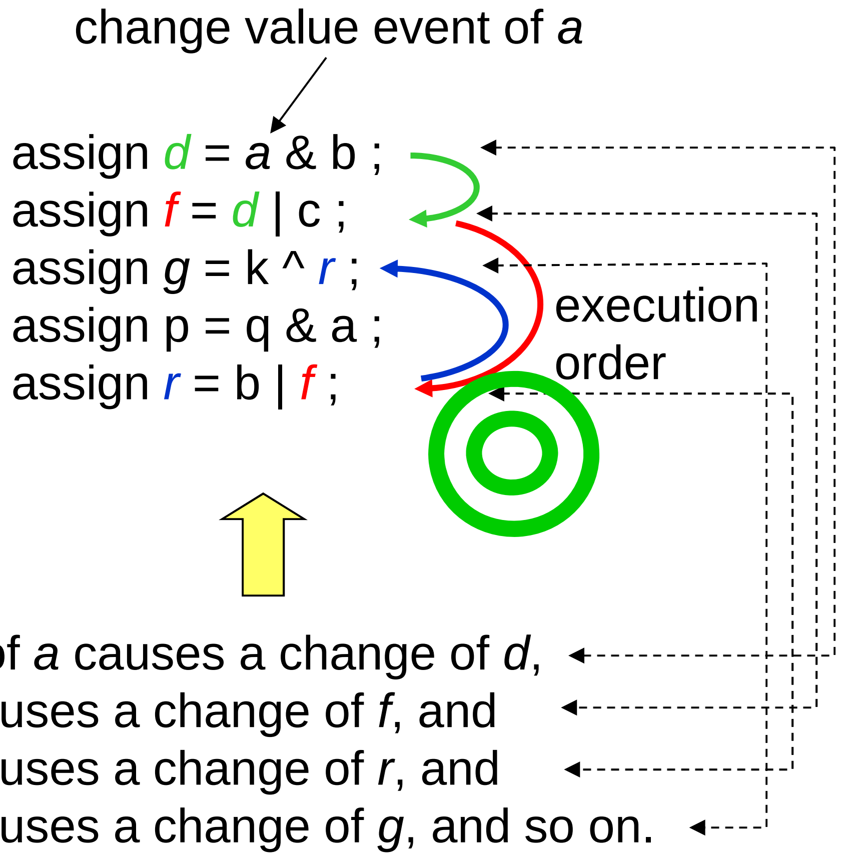
simulation result



As shown in the previous page, Verilog RTL code lines are not executed in the order they are written.
They are executed in the order of events as shown on the right of this page;



```
assign d = a & b ;
assign f = d | c ;
assign g = k ^ p ;
assign p = q & a ;
assign r = b | f ;
```



```
assign d = a & b ;
assign f = d | c ;
assign g = k ^ r ;
assign p = q & a ;
assign r = b | f ;
```

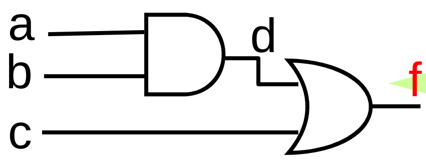
Update of *a* causes a change of *d*,
which causes a change of *f*, and
which causes a change of *r*, and
which causes a change of *g*, and so on.
Therefore, when *a* changes, *d* and *p* are calculated first, then *f*, and then *r*, and then *g*, and so on.

In the previous example, one RTL code line corresponds to one logic gate. It means that the logic gate structure is known to a designer. But this is not always the case.

In many cases, logic gate structure is not known to a designer while he/she is designing a sophisticated logic block. In Verilog RTL programming, he/she can write such a logic block by using procedures.

gate diagram

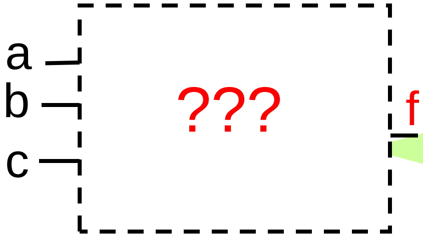
Verilog RTL source program



Simple case;
A designer can know logic gate structure.

```
assign d = a & b ;
assign f = d | c ;
```

logic gate description by continuous assign



function: Output b if a is true, otherwise output c onto f.

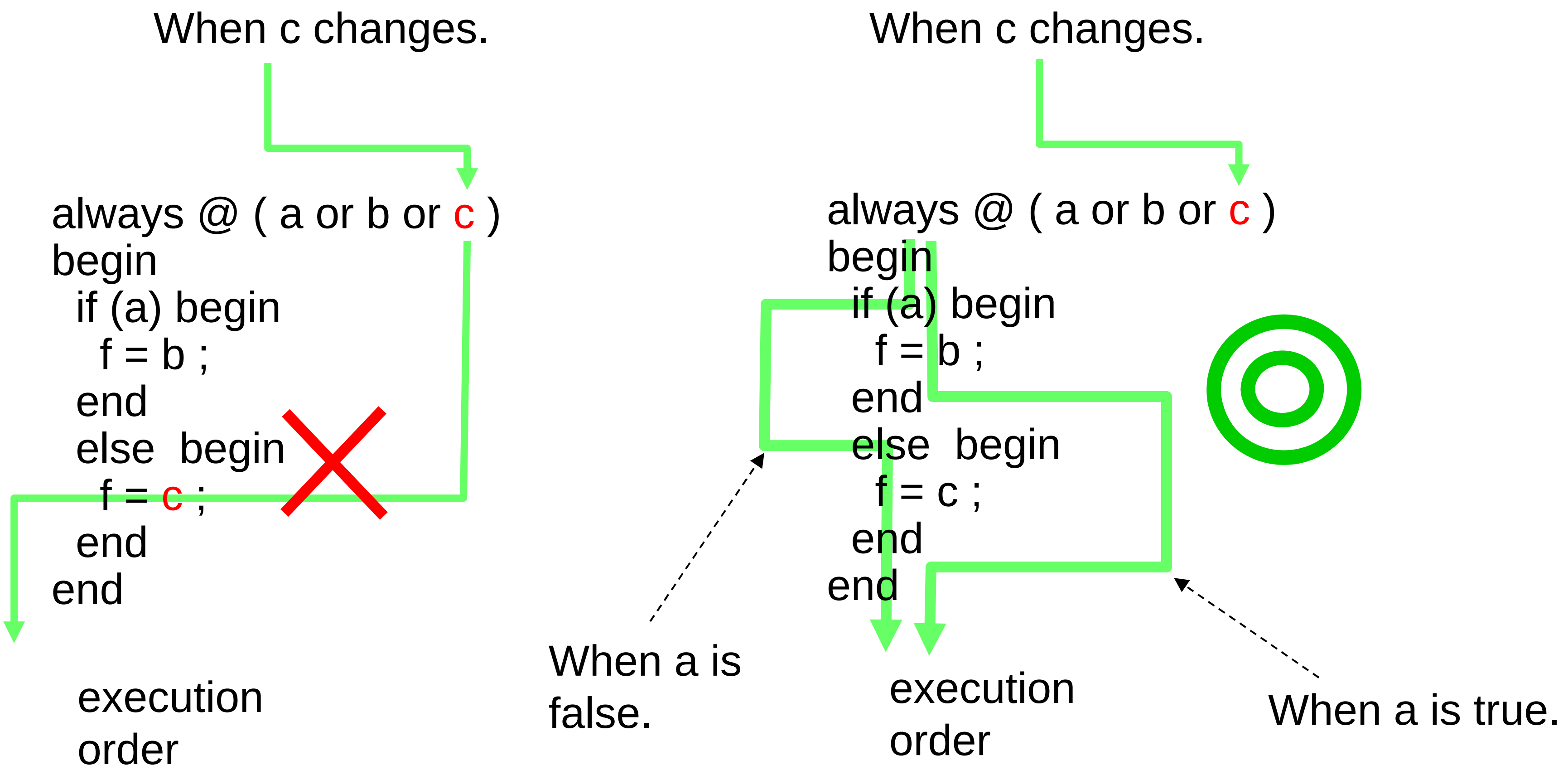
Sophisticated case;
A designer can not know logic gate structure.

```
always @ ( a or b or c )
begin
  if (a) begin f = b ; end
  else  begin f = c ; end
end
```

logic description by always construct (procedure)

While writing procedures, a designer can use programming techniques of procedural language.

In simulation, Verilog procedures are executed driven by events such as change value events.
While a procedure is executed, code lines in the procedure are executed line by line from top to bottom.



When c changes.

```
always @ ( a or b or c )
begin
  if (a) begin
    f = b ;
  end
  else  begin
    f = c ;
  end
end
```

execution order

When c changes.

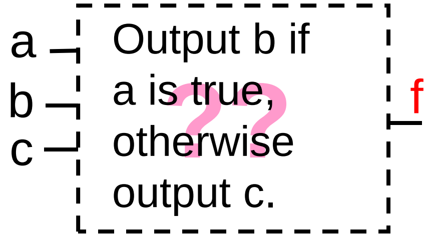
```
always @ ( a or b or c )
begin
  if (a) begin
    f = b ;
  end
  else  begin
    f = c ;
  end
end
```

When a is false.

execution order

When a is true.

gate diagram



Output b if a is true, otherwise output c.

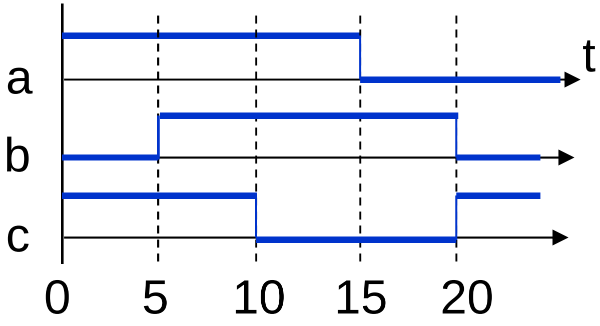
⟺

Verilog RTL source program

```
always @ ( a or b or c )
begin
  if (a) begin f = b ; end
  else  begin f = c ; end
end
```

logic description by always construct

time chart

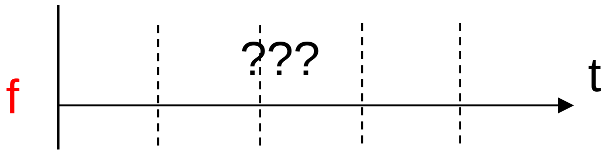


⟺

```
initial a = 1;
initial b =0 ;
initial c = 1;
initial    #5 b = 1 ;
initial  #10 c = 0 ;
initial  #15 a = 0 ;
initial  #20 b = 0 ;
initial  #20 c = 1 ;
```

initial values

simulation



See next page to find how to calculate the value of f ??

Verilog RTL source program

```
always @ ( a or b or c ) begin
  if (a) begin f = b ; end
  else  begin f = c ; end
end
initial a = 1;
initial b =0 ;
initial c = 1;
initial   #5 b = 1 ;
initial  #10 c = 0 ;
initial  #15 a = 0 ;
initial  #20 b = 0 ;
initial  #20 c = 1 ;
```

Execute this part only when a or b or c changes.

simulation process

At t=0, execute the procedure to get the value of f (f=0 because a=1 and b=0).

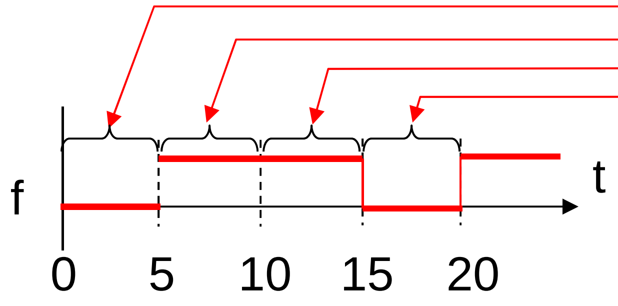
At t=5, execute the procedure to get the value of f (f=1 because a = 1 and b = 1).

At t=10, execute the procedure to get the value of f (f=1 because a=1 and b=1).

At t=15, execute the procedure to get the value of f (f=0 because a=0 and c=0).

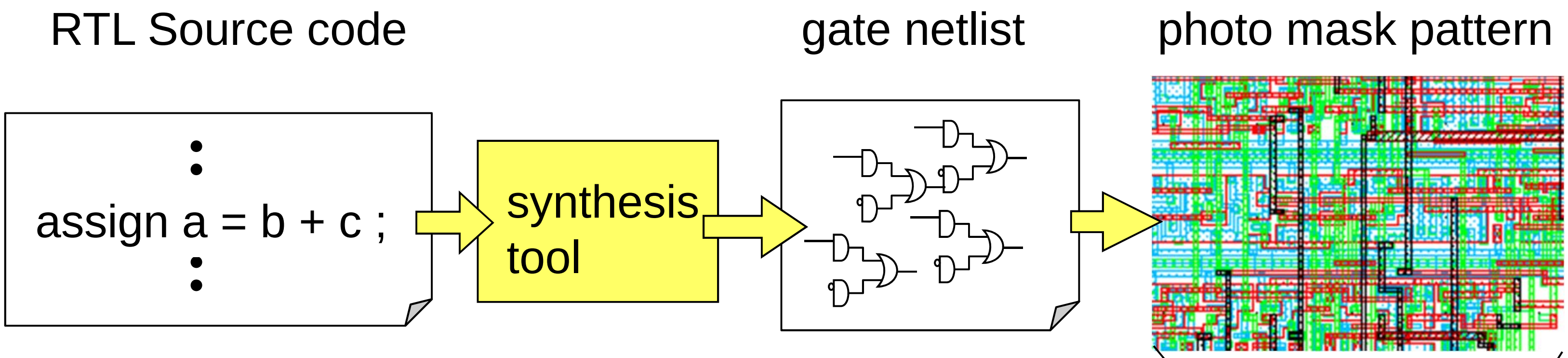
•
•
•

simulation result



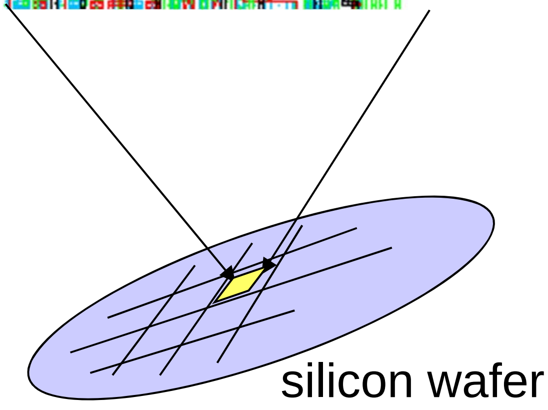
2.3 Logic synthesis

A synthesis tool uses a Verilog RTL source program as an input and generates a gate netlist which can be directly mapped to circuits on a silicon wafer.

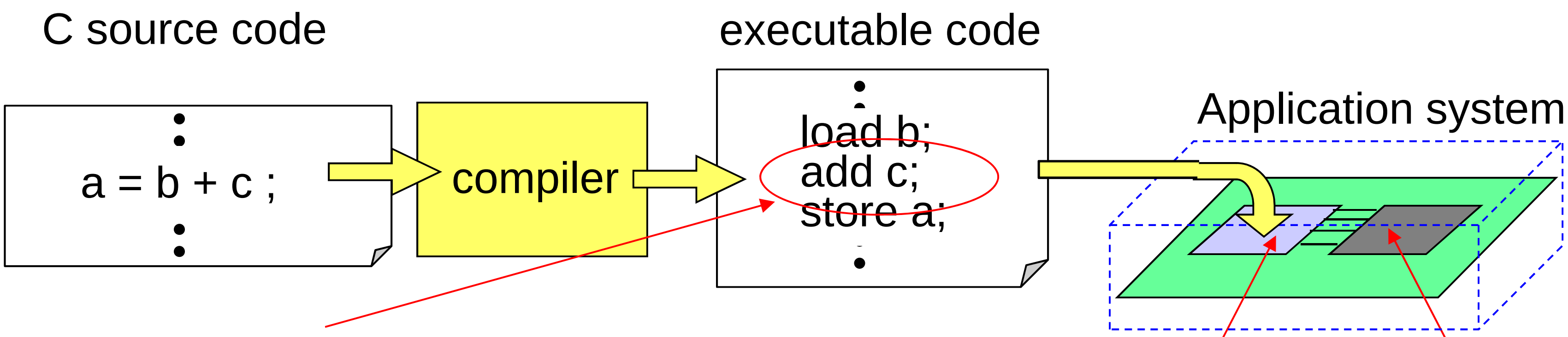


The tool will generate minimum set of logic gate blocks and will not generate gates which are not specified, explicitly or implicitly, in the RTL source code.

This means that a designer has to write all the logic blocks necessary to make them work properly themselves.



This is very much different from a case of software application program as shown on the next page.



C source code

```
:
a = b + c ;
:
```

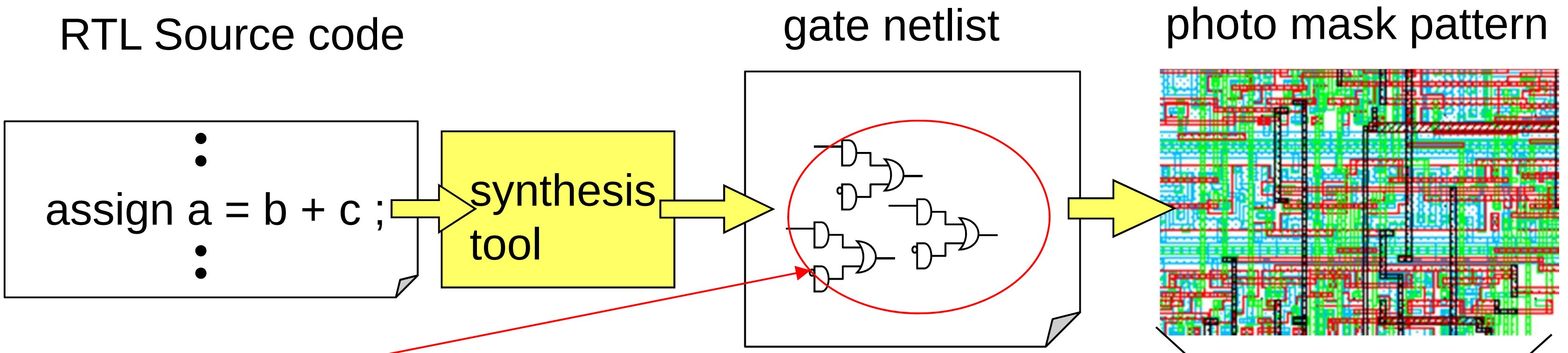
compiler

executable code

```
:
load b;
add c;
store a;
:
```

Application system

This add instruction can do add operation with a help of a CPU implemented in an application system. Without a CPU, it can not do anything.



RTL Source code

```
:
assign a = b + c ;
:
```

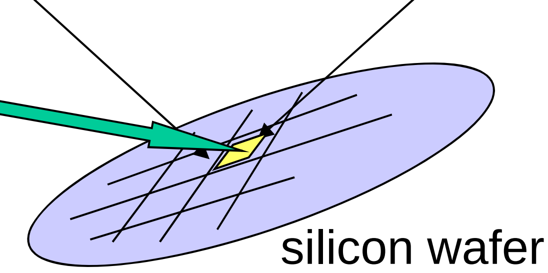
synthesis tool

gate netlist

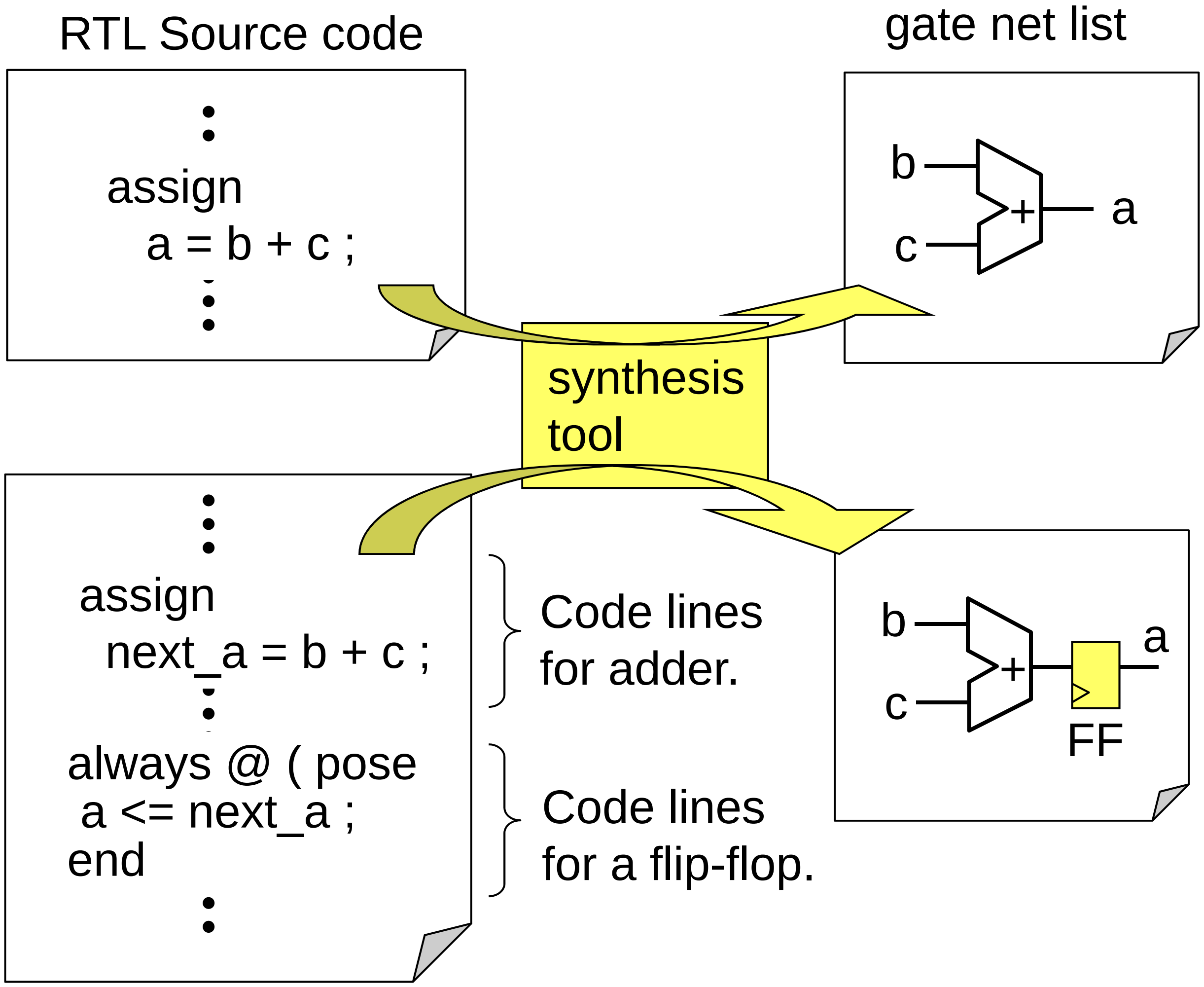
photo mask pattern

These logic gates have to be able to do add operation by themselves.

We sell this chip to our customers. If it needs additional CPUs to work properly, no customer will buy this chip.



“*A synthesis tool will not generate gates which are not specified, explicitly or implicitly, in the RTL source code*” means that a designer has to describe all the hardware resources such as memory elements in an RTL source program.



RTL Source code

```
⋮
assign
  a = b + c ;
⋮
```

```
⋮
assign
  next_a = b + c ;
⋮
always @ ( pose
 a <= next_a ;
end
⋮
```

Code lines for adder.

Code lines for a flip-flop.

We have to write RTL code lines explicitly which can generate flip-flops. Without such code lines, flip-flops are not generated by a synthesis tool.

⇕

Without a designer’s explicit coding, a synthesis tool can not tell which signals must be mapped into flip-flops.

Now, let's see some examples how a synthesis tool generate a gate netlist from Verilog RTL source code.

Verilog RTL code

```
assign d = a & b ; // <1>
assign f = d | c ;   // <2>
```

synthesis process

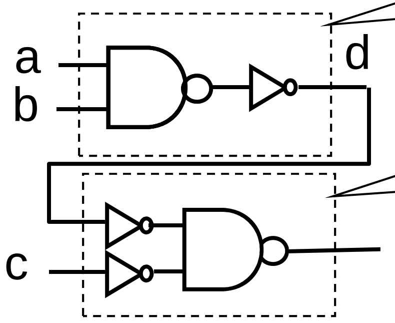
(1) Each assign statement is mapped to a gate logic block.

```
not01d1 INST1(.a1(E), .zn(D)) ;
nand02d1 INST2(.a1(A), .a2(B), .zn(E)) ;
```

connection

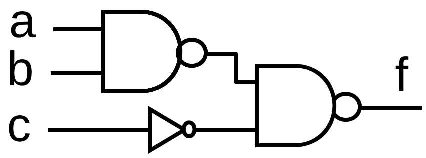
```
not01d1 INST3(.a1(C), .zn(G)) ;
not01d1 INST4(.a1(D), .zn(H)) ;
nand02d1 INST5(.a1(G), .a2(H), .zn(F)) ;
```

synthesis



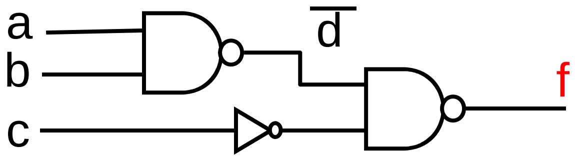
(2) Optimize and determine which circuit shall be used on a silicon.

synthesis
(optimization)



```
not01d1 INST1(.a1(C), .zn(G)) ;
nand02d1 INST2(.a1(A), .a2(B), .zn(E)) ;
nand02d1 INST3(.a1(E), .a2(G), .zn(F)) ;
```

Cell name



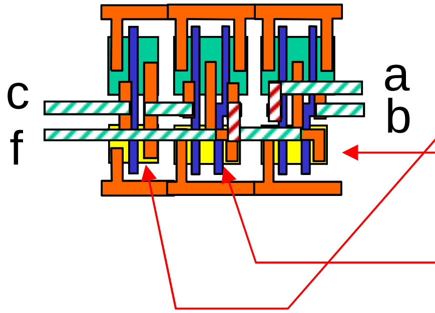
NAND gate is much smaller and efficient than AND and OR gates.

Cells are select from a standard cell library depending on a design constraint such as minimize area and/or power consumption, maximize speed, etc.

example:

If f has to drive four gates, then nand02d4 is used instead of nand02d1 as shown below.

Layout



```
not01d1 INST1(.a1(C), .zn(G)) ;
nand02d1 INST2(.a1(A), .a2(B), .zn(E)) ;
nand02d4 INST3(.a1(E), .a2(G), .zn(F)) ;
```

The drivability of this cell is 4.

Verilog gate level programming

While synthesizing, procedures are mapped into a logic gate blocks procedure by procedure base.

Verilog RTL code

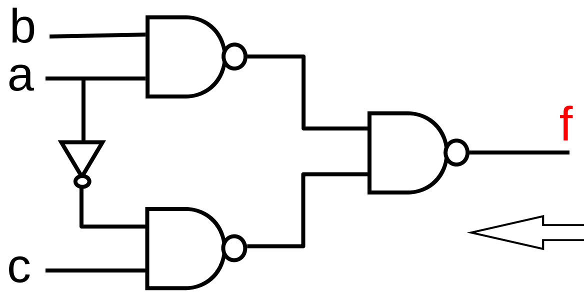
```
always @ ( a or b or c ) begin
  if (a) begin f = b ; end
  else  begin f = c ; end
end
```

synthesis process

(1) Each procedure is mapped to a logic gate block.

(2) Optimize and determine which circuit shall be used on a silicon.

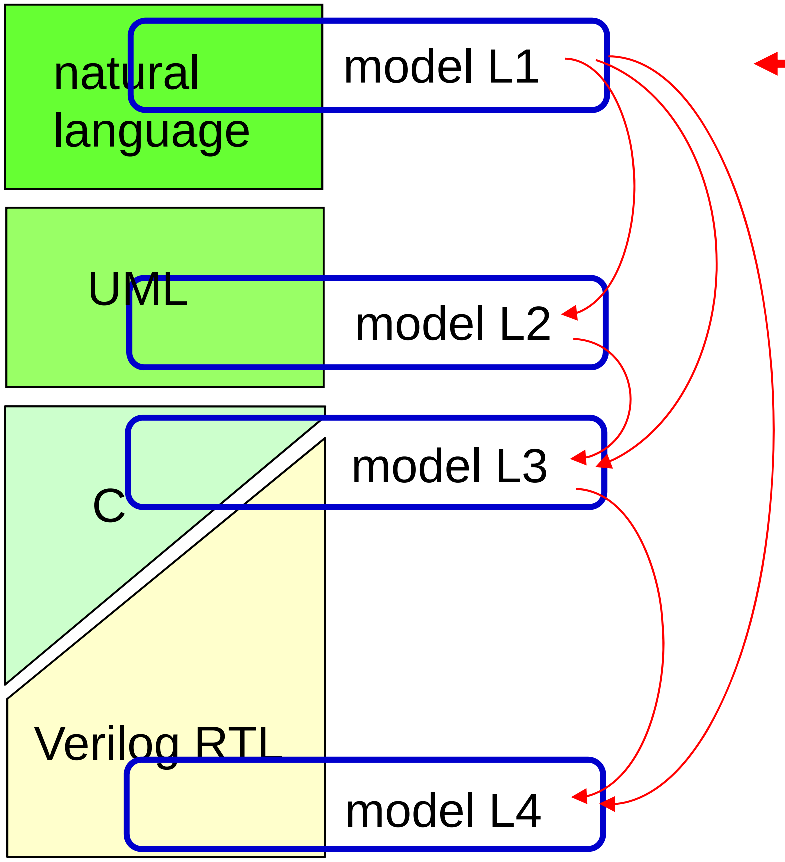
synthesis



```
not01d1 INST1(.a1(B), .zn(D)) ;
nand02d1 INST2(.a1(A), .a2(B), .zn(E)) ;
nand02d1 INST3(.a1(D), .a2(C), .zn(G)) ;
nand02d1 INST4(.a1(E), .a2(G), .zn(F)) ;
```

2.4 Development process

In general, development flow goes from high abstraction level to low abstraction level as shown below. Currently each model must be mapped into the next level manually. Some steps may be skipped, for example it is not always necessary to create UML or/and C models.



We usually make several levels of documents written in natural language, such as,

requirements specification,

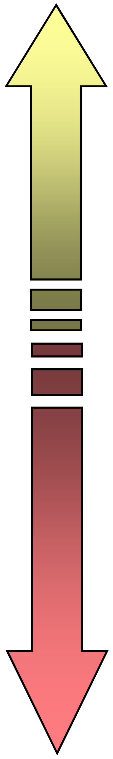
functional/interface specification,

internal design specification, and

etc.

abstraction level

Highest



Lowest

Description

Focusing on functionality only. Timing and hardware resource needed can be ignored.

Order of events such as signal change is taken into account. (Sequence accurate)

Processing time of signal change is taken into account. (Elapsed time accurate)

Precise timing is taken into account. (Cycle time accurate)

Not only functionality and timing, but also hardware resources needed are described.

difficulty to write the code

Easy

Just focus on functionality only.

(Short development time, less bugs)

Difficult

Many issues must be considered at the same time.

(Long development time, more bugs)

To get a synthesizable code, we have to go down to the lower abstraction level.

Currently higher level language such as C programming language is not fully synthesizable. Many developers are working on developing synthesizable high level languages such as “system C”.

We will be able to use such high level languages in the near future, but so far Verilog RTL is most widely used in LSI design.

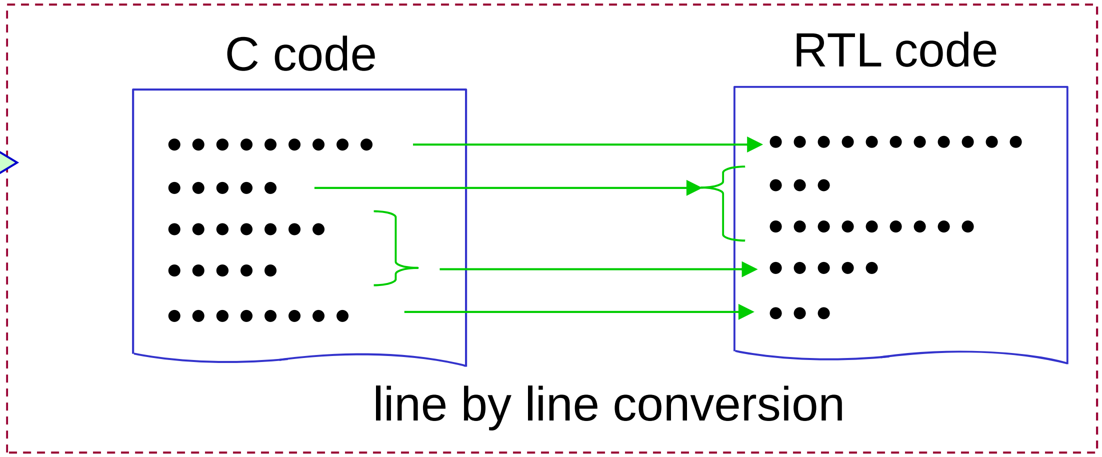
Although C code is not synthesizable, C code may have great value because it is

easy to write (shorter development time than that of RTL code),
efficient to describe certain logic,
able to eliminate ambiguity, and
usable as an golden model to check functionality.

C program does not have to care about hardware resource description, while in RTL, we have to describe hardware resources needed for implementing the logic into silicon.
C code is much simpler and shorter than a code written in RTL.
“Simple and short” means less bugs. Therefore C code can be used as a golden model.

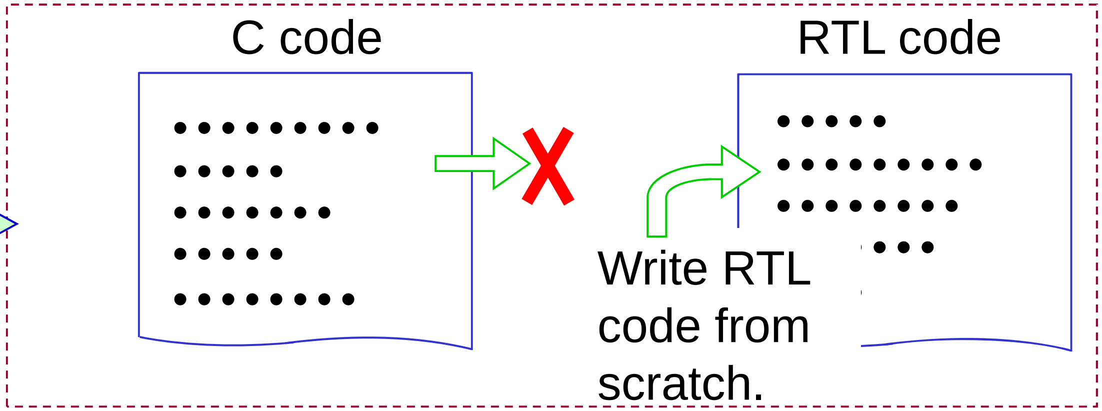
If you write a C model, it can be mapped into RTL code line by line. But line by line mapping may not always be a good method to write RTL.
C program structure may not be suitable to represent high speed data path logic.
In such cases, use C model as reference model and create RTL code from scratch.

Some times, line by line conversion from C code to RTL code may work.

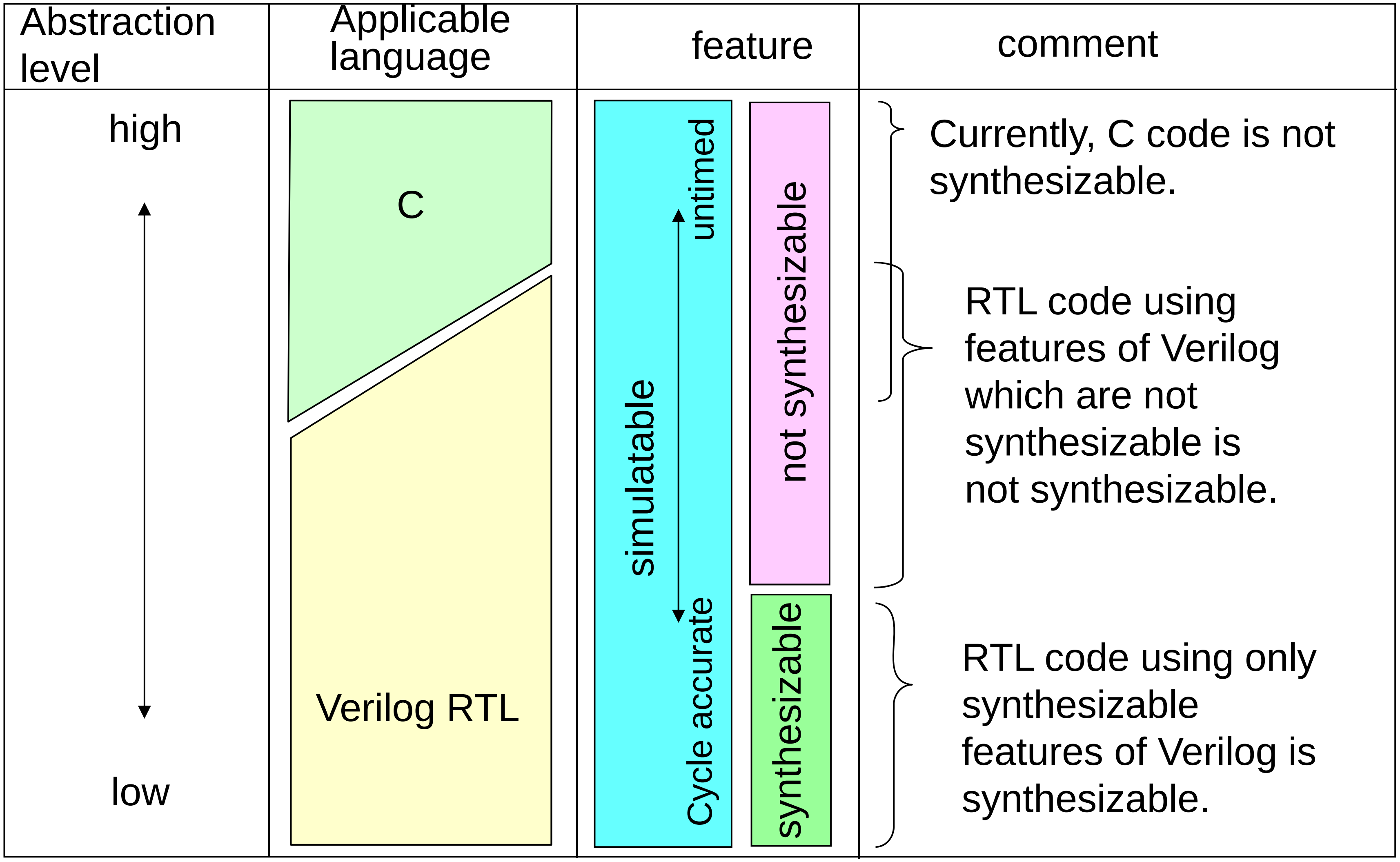


However, C code may have improper logic structure for parallel processing.

In such cases, write the code from scratch. Line by line conversion may result in poor RTL code.



As a summary, we must select a suitable abstraction level depending on the purpose of modeling the target logic.



Abstraction level

Applicable language

feature

comment

high

low

C

Verilog RTL

simulatable

untimed

Cycle accurate

not synthesizable

synthesizable

Currently, C code is not synthesizable.

RTL code using features of Verilog which are not synthesizable is not synthesizable.

RTL code using only synthesizable features of Verilog is synthesizable.

3. Verilog fundamentals

In actual job, always check manuals to see if your understanding is correct and applicable to specific version or specific vendor's tool.

In verilog reference manual many features are written.

However, many of them are just written for compatibility with previous generations. For example, wired-or connections are explained in manual and features related to this connection are explained, but do not use them. Today, we do not use wired-or connection in logic design any more.

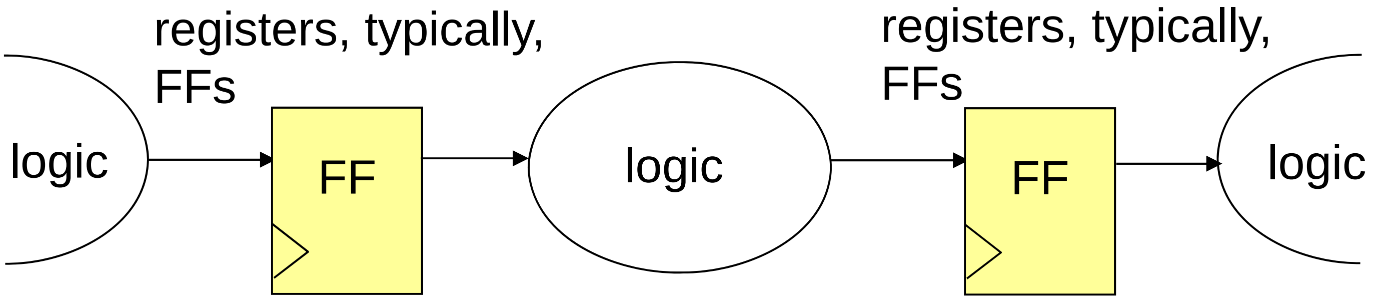
Do not use features just because a manual says “you can do it”.

When using features not explained in this text material, always ask your supervisor if you can use them or not.

What is recommended may change when a new standard or new methodology becomes available. Be aware of evolution of tools.

3.1. Verilog RTL

There are several levels of description in Verilog language. For front end design, we generally use Register Transfer Level, RTL, which can describe how signals go into registers, memory elements, as shown below.
It is suitable for synchronous design.



Verilog RTL can be used to write a "behavioral model" of the target system. The term "behavioral model" can encompass any model other than a gate level model, since there is no concrete definition for the term.

We can write a "behavioral model" in RTL just to see how it behaves. Such RTL code may not be hardware resource accurate. It can be incrementally optimized to get the desired implementation satisfying timing and area constraints. That is, earlier versions of RTL code may not be synthesizable. But it does not mean “behavioral model is not synthesizable.”

RTL code

Verilog HDL source text files shall be a stream of lexical tokens. A lexical token shall consist of one or more characters.
The layout of tokens in a source file shall be free format; that is, spaces and newlines shall not be syntactically significant other than being token separators, except for escaped identifiers.

The types of lexical tokens in Verilog HDL

Type	description	comment
White space	Characters for spaces, tabs, newlines, and formfeeds. These characters shall be ignored except when they serve to separate other lexical tokens.	Blanks and tabs shall be considered significant characters in strings.
Comment	A one-line comment shall start with the two characters // and end with a newline. A block comment shall start with /* and end with */. Block comments shall not be nested.	Do not use a block comment.
Operator	Perform specific manipulation over the operand(s).	Operators are single-, double-, or triple-character sequences.
Number	Constant numbers can be specified as integer constants or real constants.	See 3.3 of this material.

The types of lexical tokens in Verilog HDL (Continued)

Type	description	comment
String	A string is a sequence of characters enclosed by double quotes (“ ") and contained on a single line.	Strings used as operands shall be treated as unsigned integer constants represented by a sequence of 8-bit ASCII values.
Identifier	An identifier is used to give an object a unique name so it can be referenced. An identifier is either a simple identifier or an escaped identifier. A simple identifier shall be any sequence of letters, digits, dollar signs ($), and underscore characters (_).	The first character of a simple identifier shall not be a digit or $; it can be a letter or an underscore. Identifiers shall be case sensitive.
Keyword	Keywords are predefined nonescaped identifiers that are used to define the language constructs.	A Verilog HDL keyword preceded by an escape character is not interpreted as a keyword.

Verilog is free format

```
module abc ( a, b ) ;
input a ;
output [3:0] b ;
wire a ;
reg [3:0] b ;
always @ ( a ) begin
   if ( a == 1'b1 ) b <= 4'b0011;
  else                    b <= 4'b1010 ;
end
endmodule
```

Same

```
module abc (
     a,
     b             ) ;
input a ; output [3
                         :0] b ; wire a ;
reg [3:0] b ;
always
  @
  (
        a )
      begin   if ( a == 1'b1 )
         b
              <=
                4'b0_0_1_1
            ;
  else                    b  <= 4'b1010 ;
                       end
endmodule
```

The code on the right is just for explanation.
Never write RTL code in this manner.

Escape character

Backslash (＼) can be used as an escape character to give a special meaning to characters that follow it when it is used in a string.

Escape string	Character produced by escaped string
＼n	Newline character
＼t	Tab character
＼＼	＼character
＼“	“ character
＼ddd	A character specified in 1-3 octal digits (0≦d≦7)

Example:
$display(“\141b”);
will display a character string ab on a terminal, because octal 141 is hex 61. Hex 61 is ASCII code for the character a.

Escape character (continued)

Backslash (＼) can be used to define escaped identifier.
Escaped identifiers shall start with backslash and end with white space (space, tab, newline).

They provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal).

Neither the leading backslash character nor the terminating white space is considered to be part of the identifier.
Therefore, an escaped identifier ＼abcdef is treated the same as a non-escaped identifier abcdef.

By using backslash we can use an identifier such as \a-b^c for a variable name or a task name.
But such naming is not recommended. Do not use the backslash character in identifiers.

Q3.1-1 Answer if the following statements are correct or wrong for Verilog RTL programming.

(1) The following two pieces of RTL code are equivalent.

```
8'b00000000      ⟷      8'b_0000_0000
```

(2) The following two pieces of RTL code are equivalent.

```
abcdef0001       ⟷      abcdef000_1
```

(3) The following two pieces of RTL code are equivalent.

```
a == b ;         ⟷      a =
                           = b;
```

(4) The following two pieces of RTL code are equivalent.

```
a = b+c ;        ⟷      a=b
                           +
                           c
                            ;
```

Q3.1-1 A sample answer

(1) The following two pieces of RTL code are equivalent. ⇦ Correct, they are equivalent.

```
8‘b00000000   ⟷   8‘b_0000_0000
```

_ can be placed at any position except at the beginning in numbers.

(2) The following two pieces of RTL code are equivalent. ⇦ Wrong, they are different.

```
abcdef0001   ⟷   abcdef000_1
```

(3) The following two pieces of RTL code are equivalent. ⇦ Wrong, they are different.

```
a == b ;   ⟷   a =
                  = b;
```

== is a Verilog equality operator.

(4) The following two pieces of RTL code are equivalent. ⇦ Correct, they are equivalent.

```
a = b+c ;   ⟷   a=b
                  +
                  c
                   ;
```

Q3.1-2 Answer if the following statements are correct or wrong for Verilog RTL programming.

(1) The following two pieces of RTL code are equivalent.

module ↔ mod ule

(2) The following two pieces of RTL code are equivalent.

```
abc  def        abc
                  def
```

(3) The following piece of RTL code may not be illegal.

```
\abc+def = abc + def ;
```

(4) The following piece of RTL code is illegal.

```
\abc + def = abc + def ;
```

(5) The following piece of RTL code is illegal.

```
\abc+def= abc + def ;
```

Q3.1-2 A sample answer.

(1) The following two pieces of RTL code are equivalent. ⇐ Wrong, they are different.

module ⟷ mod ule

(2) The following two pieces of RTL code are equivalent. ⇐ Correct, they are equivalent.

abc def ⟷ abc
  def

(3) The following piece of RTL code may not be illegal. ⇐ Correct, LHS is an escaped identifier.

\abc+def = abc + def ;

(4) The following piece of RTL code is illegal. ⇐ Correct, LHS consists of two identifiers and one operator.

\abc + def = abc + def ;

(5) The following piece of RTL code is illegal. ⇐ Correct, the expression consists of three identifiers and one operator.

$\underbrace{\backslash\text{abc+def=}}\ \underbrace{\text{abc}} + \underbrace{\text{def}}\ ;$

3.2. Name space and naming rules

3.2.1 Name space

Name space

global name space

(1) Module and primitive names
(2) Text macro names

→ A name used to define a module can not be used to define other modules.

local name space

Names defined in
(1) blocks (named block, function, task),
(2) modules,

See Verilog standard for other local name spaces.

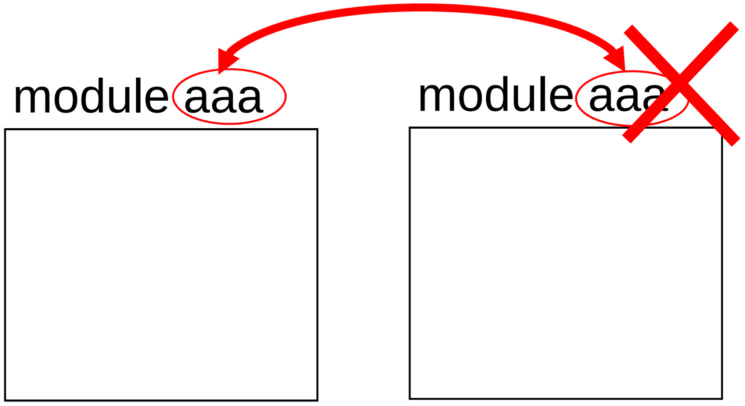
Note that the rule says “defined”, not “used”.

→ A name defined in a module can be used to define other objects in other modules.

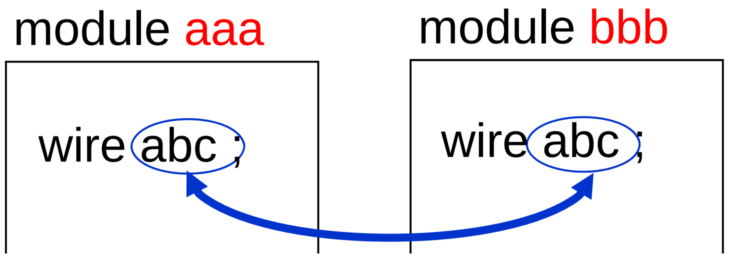
Until you understand well about the name space, just remember “module names are global and the names defined in modules are local”.

- Module name is global

Can not use the same name for modules.

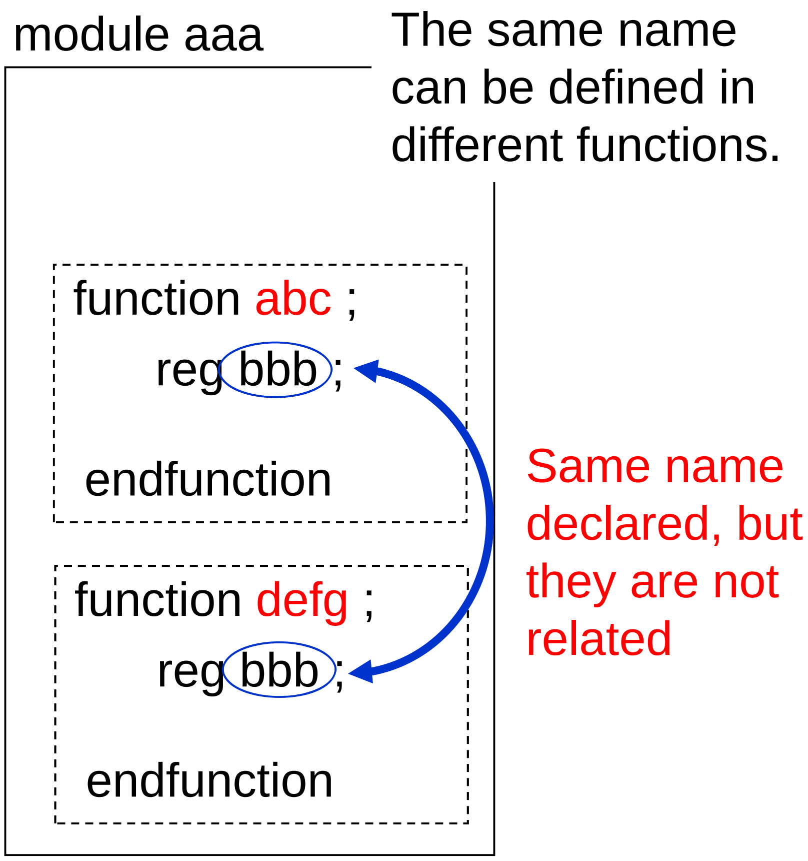


- Names defined in a module are local



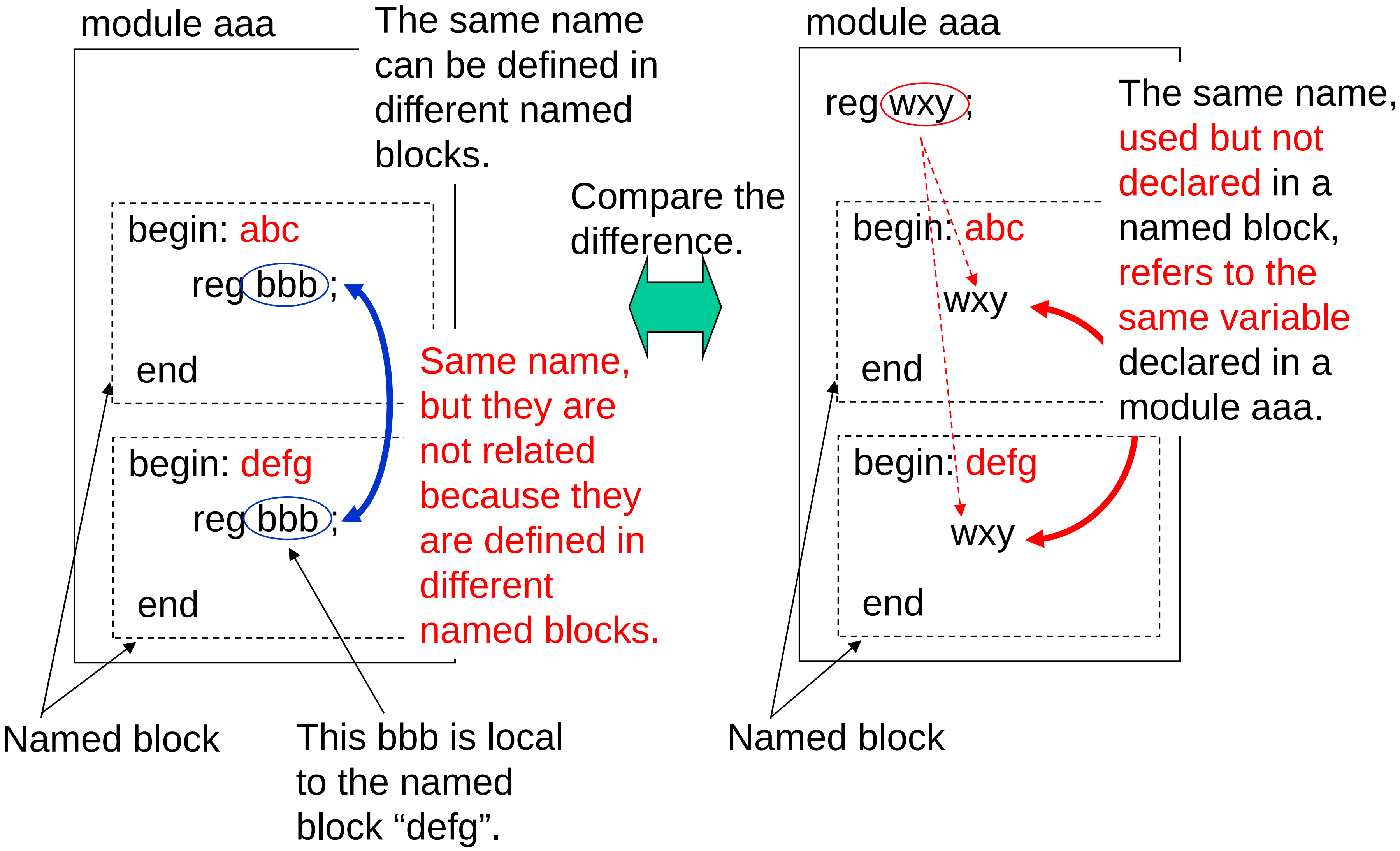
The same name declared to define different signals.
The above two signals are unrelated although they have the same name.

- Names defined in a function are local


module aaa
The same name can be defined in different functions.
function abc ;
reg bbb;
endfunction
function defg ;
reg bbb;
endfunction
Same name declared, but they are not related

Avoid using the same name even in a different name space unless using the same name makes the logic easy to understand.

- Names defined in a named block are local



```
module aaa
    begin: abc
        reg bbb;
    end
    begin: defg
        reg bbb;
    end
```

The same name can be defined in different named blocks.

Same name, but they are not related because they are defined in different named blocks.

Named block

This bbb is local to the named block “defg”.

Compare the difference.

```
module aaa
    reg wxy;
    begin: abc
        wxy
    end
    begin: defg
        wxy
    end
```

The same name, used but not declared in a named block, refers to the same variable declared in a module aaa.

Named block

3.2.2. Naming rules

Give names to identify the objects in Verilog based on the following rules.

(1) Do not use single character names, use at least two or more characters.
Do not use more than 32 characters. ⇨ Use 5 to 16 characters.

- Use a meaningful name. Meaningful in English, not in Vietnamese.
- Use only English letters, numbers, and underscore (_)
- Use only English letters at the beginning of a name,
- Don't use underscore at the end.
- Don't assume that tools are case sensitive. Do not use “ABC” if you used “abc” somewhere else. Depending on the tool, ABC and abc make no difference.

Tools assign names for deliverable signals/information based on the name you gave. Therefore if you use a long name, names created by the tools may become too long to handle conveniently.

⇨ In this text book, one character names are used to save spaces, however, you must follow the rules above in your work and exercises.

(2) Do not mix uppercase and lowercase characters.

In software designing world uppercase and lowercase characters are mixed to enhance readability. But in RTL coding, do not mix them.

- Use only lowercase characters, numbers, and underscore (_) for module, block, function, task, and signal names.
- Use only uppercase characters, numbers and underscore (_) only for parameter, define and constant.

```
wire Dmac_Rd_Strb           ✗   ⇨   wire dmac_rd_strb            ◎

parameter sig_a_Bit_Width   ✗   ⇨   parameter SIG_A_BIT_WIDTH    ◎
```

(3) Name reset and clock signal so that it is clear to anyone that they are reset and clock.

The reset signal must begin with "rst ".

⇨ rst_timer, rst_dmac, and so on

The clock signal must begin with "clk " or "ck ".

⇨ clk_sysbus, and so on

(4) Append _n to the active low signal name.

Name the active low reset signal “rst_n”, and so on.

This is relatively a new recommendation proposed
in STARC modeling guide.
Depending on job groups, appending “_x” or other
notation has been used for active low signals.

STARC: Semiconductor Technology Academic Research Center

Note on polarity of signals (1)

In general, use active high signal which takes logic value 1 when it is asserted and takes logic value 0 when it is negated.

door_open → active high

When this signal is 1, it means the door is open.
If it is 0, it means the door is not open.

door_open_n → active low

When this signal is 0, it means the door is open.
If it is 1, it means the door is not open.

When defining a signal, always make it clear that the signal is active high or active low.

Active low signals are not recommended to be used.
Use active high signals unless active low is needed.

Note on polarity of signals (2)

When using active low signals, be careful about operators to apply to them.

When there are three active los signals door_1_open_n, door_2_open_n, and both_door_open_n (both door_1 **and** door_2 are open), the relation between these three signals must be

```
both_door_open_n = door_1_open_n | door_2_open_n ;
```

Verilog OR operator. ◎

Care must be taken that it must not be

```
both_door_open_n = door_1_open_n & door_2_open_n ;
```

Verilog AND operator. ✗

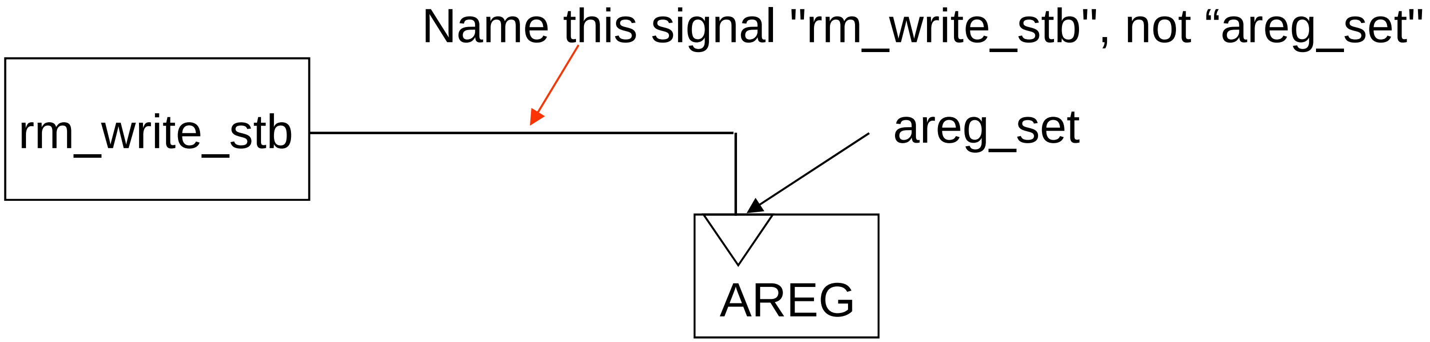
(5) Don't use names confusingly similar to reserved words.

"clock", "all", "vcc", "vdd", "gnd", "pwr", etc. are Verilog keywords, that are used for special purpose by tools.

⇨ Do not use names such as, "pw1", "al1",,,,,

Avoiding using general terms such as “stop”, “start”, or “enable”, etc. alone even if currently they are not defined as keywords. Use them mixed with other words in a way such as “start_dma”, or “enable_counting”.

(6) Use output terminal name for a signal name.



You must check your RTL code by "SpyGlass" to confirm it follows the naming rule.

Always write your RTL code following the rule.
However, when reading the other's code,
always beware that it may not follow the rule.

Never use any tricky programming technique even if it reduces code lines !

Your code has to be easy to read, to understand, to use , and to modify by others and must work properly when used and/or modified by others.

Never show off your skill or talents by making complicated, tricky, loophole passing design or logic.

An excellent engineer is a person who can make a sophisticated function with simple logic.

Always make everything simple.

Q3.2-1 : Among the names shown below which do you think better naming?

sysmem_rd_strobe sysmem_wt_strobe	⇔	sysmem_read_stb sysmem_write_stb
temp1 temp2 temp3	⇔	tmp_1 tmp_2 tmp_3
l1rd_data l2rd_data l3rd_data	⇔	rd_d_line_1 rd_d_line_2 rd_d_line_3

Q3.2-1 : A sample answer

△		○
sysmem_rd_strobe sysmem_wt_strobe	⇔	sysmem_read_stb sysmem_write_stb
2/16 characters to identify		4/15 characters to identify
2 miss-types will result in another name.		Even 3 miss-types will not result in another name.

worst is like sysmemrstrobe / sysmemwstrobe

×		△
temp1 temp2 temp3	⇔	tmp_1 tmp_2 tmp_3
Using numbers like 1,2,3 as an identifier is not a good idea.		Better readability for human.

×		△
l1rd_data l2rd_data l3rd_data	⇔	rd_d_line_1 rd_d_line_2 rd_d_line_3
putting 1 and l side by side is very dangerous		

Q3.2-2 : The followings are examples of names not recommended to use in our code to name signals we use in our design. Answer why it is better not to use these naming.

al1	event1	time0
BUF	vce	gmd
vddd	pwt	INITIAL1

Q3.2-2 : A sample answer

Because they are confusingly similar to Verilog keywords.

al1 ⇧ all	event1 ⇧ event	time0 ⇧ time
BUF ⇧ buf	vce ⇧ vcc	gmd ⇧ gnd
vddd ⇧ vdd	pwt ⇧ pwr	INITIAL1 ⇧ initial

Q3.2-3 : Correct the following naming and give the reason why they are not good.

```
(1) wire EnableDataRead ;
```

```
(2) `define ST1  2'b00  // initial state
    `define ST2  2'b01  // idling
    `define ST3  2'b10  // moving up
    `define ST4  2'b11  // moving down
```

```
(3) parameter Bit_Width = 8 ;
```

```
(4) wire [7:0] sum1,sum2,sum3 ; // a1+a2=sum1, a3+a4=sum2, sum1+sum2=sum3
```

Q3.2-3 : A sample answer

(1) wire ~~EnableDataRead~~ ;

Do not mix uppercase characters and lowercase characters. Use lowercase characters only for signal names.

⇨ wire enable_data_read ;

(2) `define ~~ST1~~ 2'b00 // initial state
`define ~~ST2~~ 2'b01 // idling
`define ~~ST3~~ 2'b10 // moving up
`define ~~ST4~~ 2'b11 // moving down

⇨

```
`define INTL      2'b00  // initial stat
`define IDL       2'b01  // idling
`define MVUP      2'b10  // moving up
`define MVDWN     2'b11  // moving down
```

Use meaningful names.

(3) parameter ~~Bit_Width~~ = 8 ;

Use uppercase characters for parameter and define.

⇨ parameter BIT_WIDTH = 8 ;

(4) wire [7:0] ~~sum1,sum2,sum3~~ ; // a1+a2=sum1, a3+a4=sum2, sum1+sum2=sum3

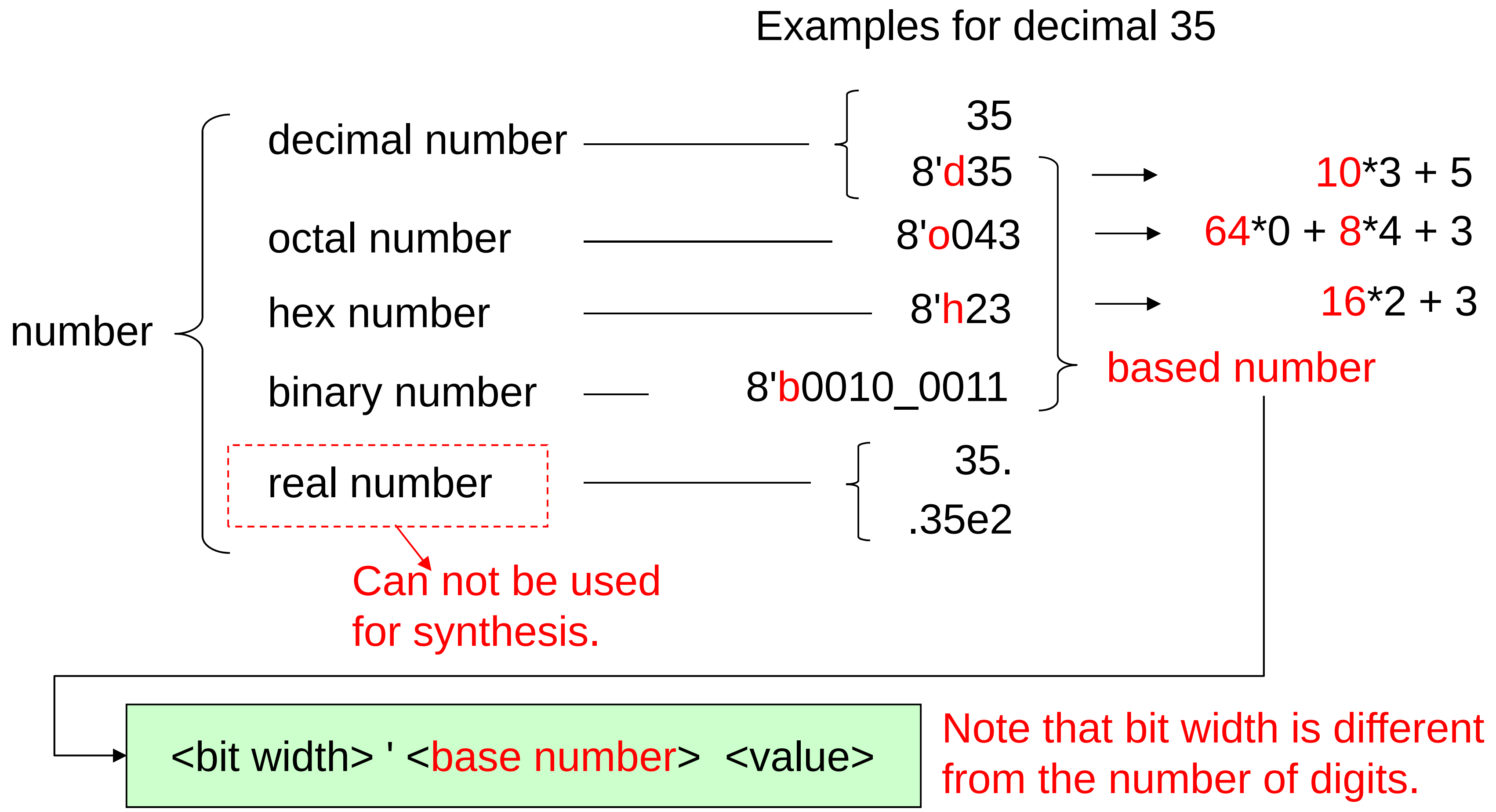
Use meaningful names.

```
wire [7:0] sum12,sum34,sum1234 ;
                    // a1+a2=sum12, a3+a4=sum34, sum12+sum34=sum1234
```

Keywords of Verilog2001

always, and, assign, automatic,
begin, buf, bufif0, bufif1,
case, casex, casez, cell, cmos, config,
deassign, default, defparam, design, disable,
edge, else, end, endcase, endconfig, endfunction, endgenerate, endmodule,
endprimitive, endspecify, endtable, endtask, event,
for, force, forever, fork, function,
generate, genvar,
highz0, highz1,
if, ifnone, incdir, include, initial, inout, input, instance, integer,
join, large,
liblist, library, localparam,
macromodule, medium, module,
nand, negedge, nmos, nor, noshowcancelled, not, notif0, notif1,
or, output,
parameter, pmos, posedge, primitive, pull0, pull1, pulldown, pullup,
pulsestyle_onevent, pulsestyle_ondetect,
rcmos, real, realtime, reg, release, repeat, rnmos, rpmos, rtran, rtranif0, rtranif1,
scalared, showcancelled, signed, small, specify, specparam,
strong0, strong1, supply0, supply1,
table, task, time, tran, tranif0, tranif1, tri, tri0, tri1, triand, trior, trireg,
unsigned, use, uwire,
vectored,
wait, wand, weak0, weak1, while, wire, wor,
xnor, xor

3.3 Numbers



Examples for decimal 35

number

decimal number — 35, 8'd35 → 10*3 + 5

octal number — 8'o043 → 64*0 + 8*4 + 3

hex number — 8'h23 → 16*2 + 3

binary number — 8'b0010_0011

based number

real number — 35., .35e2

Can not be used for synthesis.

<bit width> ' <base number> <value>

Note that bit width is different from the number of digits.

Based numbers are unsigned, except where s notation is used in the base number. (s is introduced in Verilog2001.)

Decimal (size not given) can be more than 32 bits depending on implementation. The standard says "at least 32 bits". Real is 64-bit.

Examples of based number

10'hFA 10 bits hexadecimal number FA (00_1111_1010), decimal 250

10'sh3FA 10 bits signed hexadecimal number 3FA (11_1111_1010)
The MSB is on, therefore, this means decimal -6,
not a positive value, decimal 1018.

1'b0 1 bit binary number 0 (0)

5'd30 5 bits decimal number (11110), decimal 30

5'sd30 5 bits decimal number (11110), decimal -2

note the difference

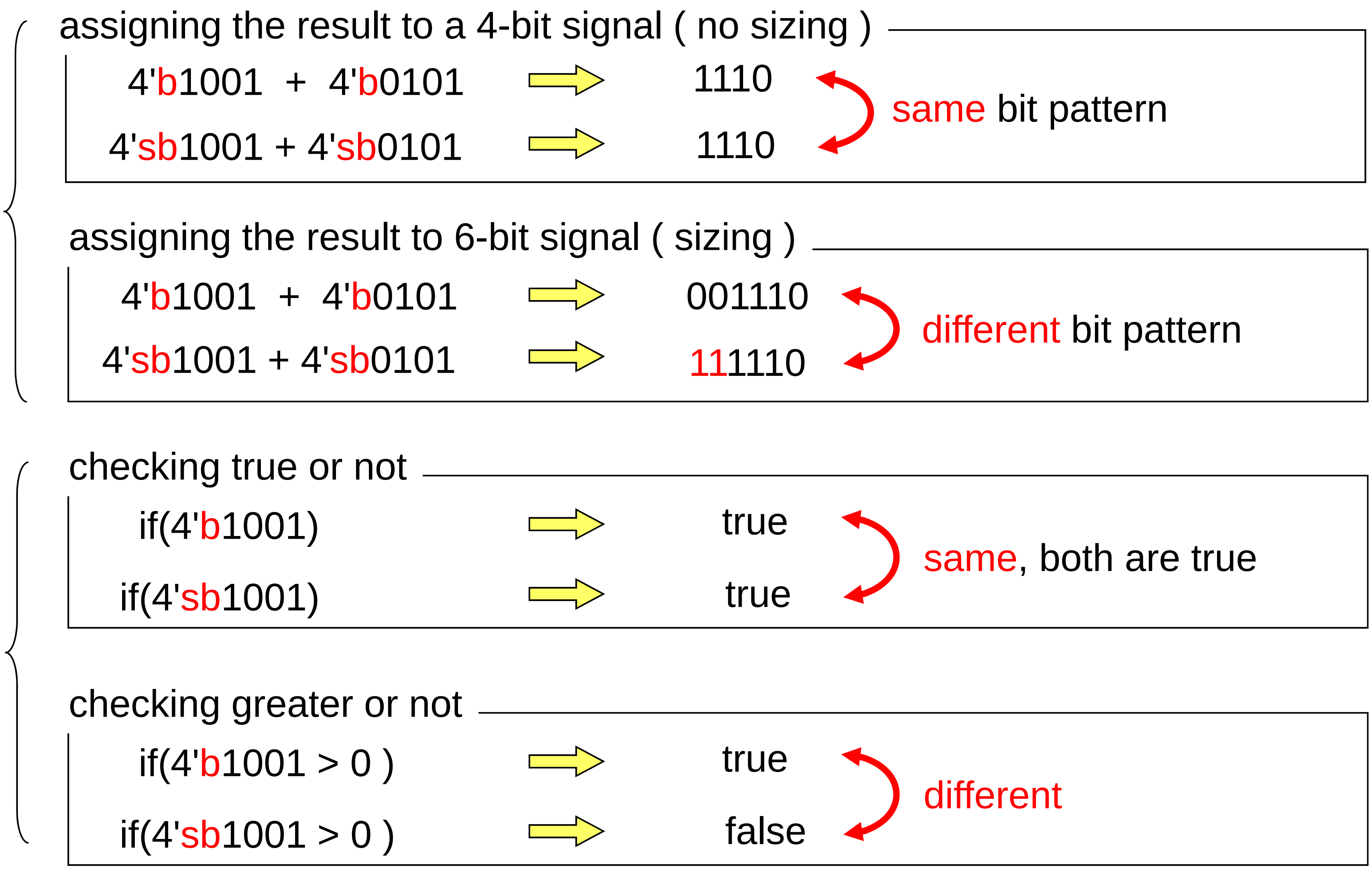
15'o10752 15 bits octal number (001_000_111_101_010), decimal 4586

37 32 bits decimal 37, no bit width given means 32-bit,
no base number given means decimal and signed.

In Verilog 1995, there is no signed value. In Verilog 2001 signed value is introduced. To specify signed value, we must use "s" in base specifier.

"s" in base number	negative value if MSB is 1, positive if MSB is 0 (can be negative depending on MSB)
no "s" in base number	positive value even if MSB is 1 (can not be negative regardless of MSB)

Bit patterns such as 1001 can be positive or negative depending on their declaration and the context they are used. Therefore, operation results can be different depending on the context they are used as shown below.



assigning the result to a 4-bit signal (no sizing)

4'b1001 + 4'b0101 → 1110

4'sb1001 + 4'sb0101 → 1110

same bit pattern

assigning the result to 6-bit signal (sizing)

4'b1001 + 4'b0101 → 001110

4'sb1001 + 4'sb0101 → 111110

different bit pattern

checking true or not

if(4'b1001) → true

if(4'sb1001) → true

same, both are true

checking greater or not

if(4'b1001 > 0) → true

if(4'sb1001 > 0) → false

different

Use Verilog 2001, when you have to handle negative values.
If you are not allowed to use Verilog 2001, you have to write additional code lines to handle negative values.

Verilog1995 code

```
if ( sig_y & 8'b1000_000) begin
   flg = 1'b1 ; // if minus, smaller than 56
end
else begin  // sig_y is positive or 0
  if ( sig_y < 56 ) begin
     flg = 1'b1 ;
  end
  else begin
     flg = 1'b0 ;
  end
end
```

Verilog2001 code

```
if ( sig_y < 56 ) begin
   flg = 1'b1 ;
end
else begin
   flg = 1'b0 ;
end
```

A piece of RTL code to check if 8-bit sig_y is smaller than 56, when sig_y can be negative.

Octal, binary, and hex based numbers can have x and z in their digit.

12'h3xa — 12 bits hexadecimal number having 4 bits of x,
The bit pattern is 0011_xxxx_1010.

8'b10x0_0zz1 — 8 bits binary number having 1 bit of x and 2 bits of z,

12'o2z75 — 12 bits octal number having 3 bits of z,
The bit pattern is 010_zzz_111_101.

When the number of digits of the value given is smaller than the given bit width, 0 is padded to the left.

examples

padding to the left →

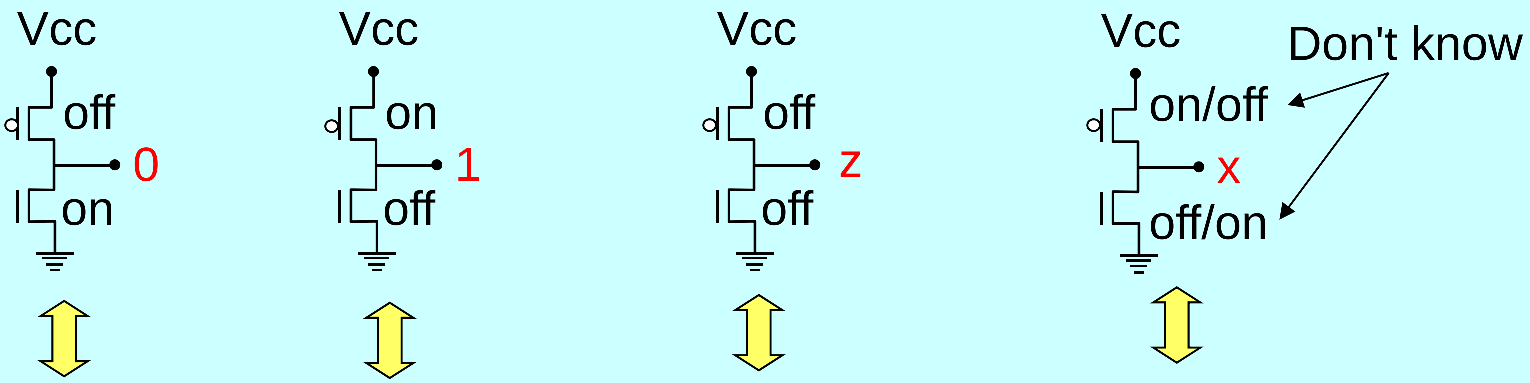
4'b0 is equal to 4'b0000

4'b1 is equal to 4'b0001

4'bz is equal to 4'bzzzz ← z padded instead of 0

4'bx is equal to 4'bxxxx ← x padded instead of 0

0, 1, x, and z represent the following logic values.



a logic 0, or a false condition

a logic 1, or a true condition

a high-impedance state

an unknown logic value

x can not be implemented into silicon. On silicon a signal must be 0, 1, or z. It can not take the value x.
That is, x has meaning only in a simulator.

Assignment such as y = 1'bx will be neglected by synthesis tool.
(x means “don’t care in synthesis.)

In Verilog “?” can be used as a shorthand for “z”.

Q3.3-1; Answer the following questions.

(1) What is the bit pattern of 15 in 4-bit?

(2) What the range, Max and Min, of 8-bit value if MSB is not a sign-bit?

(3) What the range , Max and Min, of 8-bit value if MSB is a sign-bit?

(4) What the bit pattern of minus value of 8-bit data 0010_1011 if MSB is a sign-bit?

(5) Which is larger 8-bit 1111_1000 and 1000_0111 if their MSB are sign-bit?

Q3.3-1; A sample answer.

(1) What is the bit pattern of 15 in 4-bit?

1	1	1	1

8+ 4+ 2+ 1 = 15

(2) What the range, Max and Min, of 8-bit value if MSB is not a sign-bit?

Max: 1111_1111 = 256 - 1 = 255

Min : 0000_0000 = 0

(3) What the range , Max and Min, of 8-bit value if MSB is a sign-bit?

Max: 0111_1111 = 128 - 1 = 127

Min : 1000_0000 = -128

(4) What the bit pattern of minus value of 8-bit data 0010_1011 if MSB is a sign-bit?

Invert the bit-pattern and add 1 ➡ 1101_0100 + 1 = 1101_0101

(5) Which is larger 8-bit 1111_1000 and 1000_0111 if their MSB are sign-bit?

1111_1000 = 1111_1111 - 0000_0111 = (-1) - (7) = -8

1000_0111 = 1000_000 + 0000_0111 = (-128) + (7) = -121

➡ 1111_1000 > 1000_0111

Q3.3-2; Answer if the following statements are correct or not.

(1) 4'b1 is equal to 4-bit unsigned decimal 1 because 0 is padded to the left.

(2) 4'sb1 is equal to 4-bit signed decimal –1 because sign-bit must be extended.

(3) The bit pattern of -4'sd12 is 0100.

(4) The bit pattern of -5'sd12 is 10100.

(5) If 16'hx3 is assigned to 16-bit signal y, y’s bit pattern shall become xxxx_xxxx_xxxx_0011.

Q3.3-2; A sample answer:

(1) 4'b1 is equal to 4-bit unsigned decimal 1 because 0 is padded to the left.

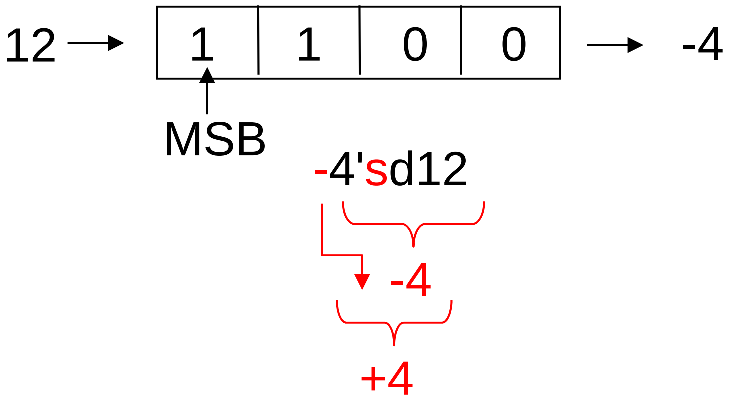
This is correct. The resulting bit pattern is 0001.

(2) 4'sb1 is equal to 4-bit signed decimal –1 because sign-bit must be extended.

This is wrong. For 4'sb1, and for 4'b1, zero will be filled for the remaining 3 bits, bit 1, 2, and 3. The resulting bit pattern is 0001 which can not be -1 of which bit pattern is 1111.

(3) The bit pattern of -4'sd12 is 0100.

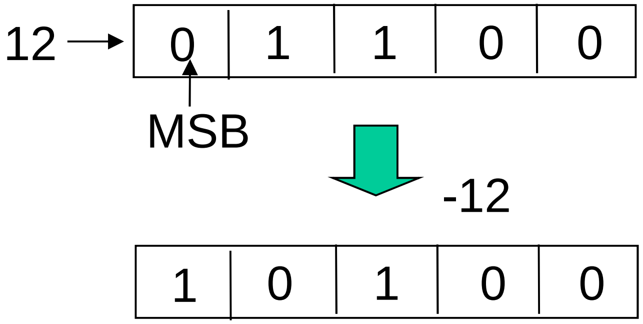
This is correct.
4-bit pattern of 12 is 1100.
Sign-bit is on, therefore, this is not 12 but -4.
-(-4) is 4.



Q3.3-2; A sample answer (continued):

(4) The bit pattern of -5'sd12 is 10100.

This is correct.
The bit pattern of 5'sd12
is 01100.
5-bit -(0110) is 10100.



0	1	1	0	0
1	0	1	0	0

(5) If 16'hx3 is assigned to 16-bit signal y, y’s bit pattern shall become xxxx_xxxx_xxxx_0011.

This is correct.
x is padded to the left.

Q3.3-3; Answer if the following statements are correct or not.

(1) In simulation, if 3-bit variable y is given a 3-bit value “1x0”, then y will become 3-bit “xxx”.

(2) In synthesis, a logic block assigning 3-bit constant value “1x0” to a 3-bit signal y will generate a circuit assigning 1, x, and 0 to bit-2, bit-1, and bit-0 of y respectively.

Q3.3-3; A sample answer

(1) In simulation, if 3-bit variable y is given a 3-bit value “1x0”, then y will become 3-bit “xxx”.

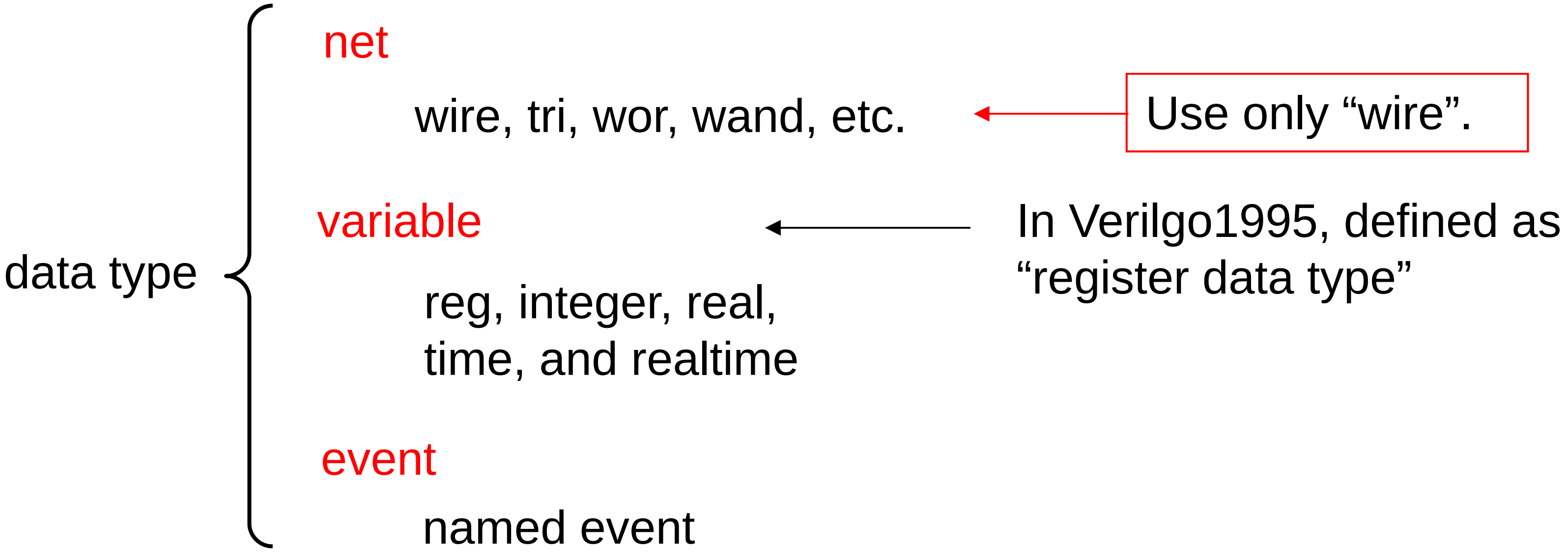
This statement is wrong.
If such a piece of code is executed in a simulation, y will become “1x0” and only the bit-1 of y will take the unknown value x.

(2) In synthesis, a logic block assigning 3-bit constant value “1x0” to a 3-bit signal y will generate a circuit assigning 1, x, and 0 to bit-2, bit-1, and bit-0 of y respectively.

This statement is wrong.
If such logic is synthesized, a circuit assigning 1 to bit-2 of y and 0 to bit-0 of y will be generated.
Assigning constant x is neglected by a synthesis tool and no circuit is generated to assign x. (There is no circuit in the world that can assign x to a signal.)

3.4 Data type

There are two main groups of data type in Verilog, nets and variables;



data type

net

wire, tri, wor, wand, etc.

Use only “wire”.

variable

In Verilgo1995, defined as “register data type”

reg, integer, real, time, and realtime

event

named event

[1] Multibit reg and net data type shall be declared by specifying a range, which is known as a vector.
[2] A net or reg declaration without range specification shall be considered to have 1 bit width, and is known as scalar.

Note: Range declaration is allowed only for nets and reg data types. (Range declaration for integer, real, realtime, and time data types are illegal.)

3.4.1 Net

- Net data types are used to model physical connections.

- They do not store values (in simulation).

- The net data types have the value of their drivers. If a net has no driver, then it has a high-impedance value (z).

⇨ Whenever the driver changes its value, net data type shall be given the updated value by a simulator.

⇨ Nets must not appear on LHS in procedures, initial construct, always construct, function, and task.

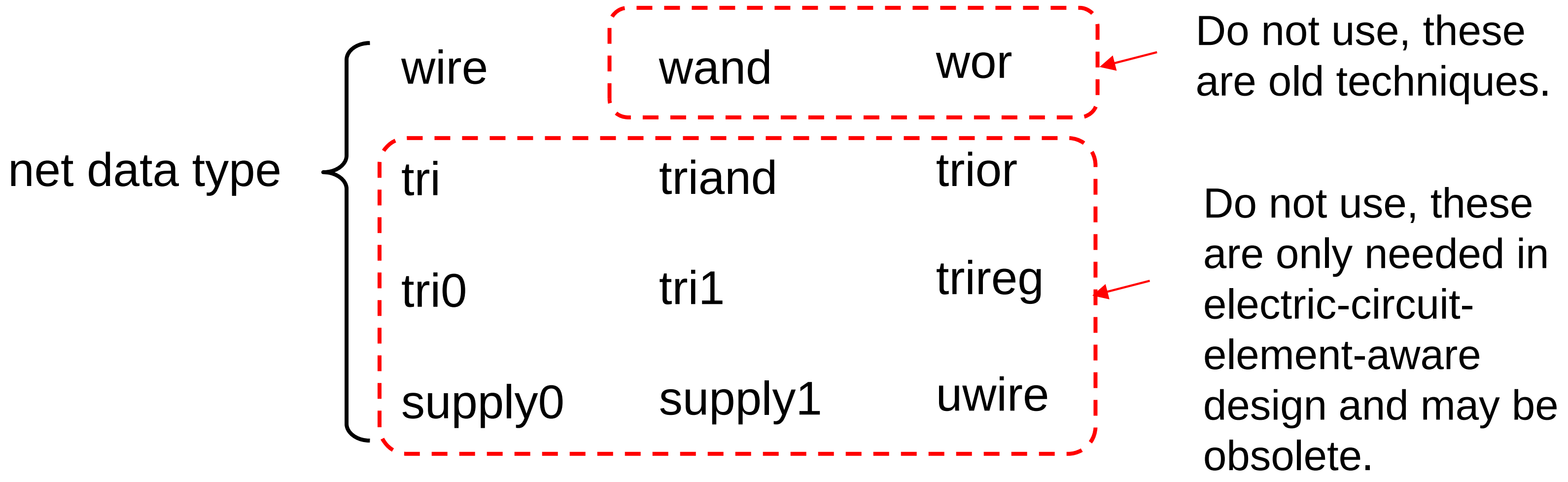
syntax

wire [range specification] *name*, *name*, ,,, ;

wire signed [range specification] *name*, *name*, ,,, ;

A net is unsigned if it is not declared *signed*.

keywords to declare net.



net data type

wire wand wor

tri triand trior

tri0 tri1 trireg

supply0 supply1 uwire

Do not use, these are old techniques.

Do not use, these are only needed in electric-circuit-element-aware design and may be obsolete.

Use only *wire* keywords.

3.4.2 Variable

- Variable data types can hold values between assignments.

- Their values are updated only when the procedure reaches the assignment statement.

⇨ Variable data type can be written on LHS only in procedures, initial construct, always construct, function, and task.

syntax

reg [range specification] *name*, *name*, ,,, ;

reg signed [range specification] *name*, *name*, ,,, ;

integer *name*, *name*, ,,, ;

real *name*, *name*, ,,, ;

time *name*, *name*, ,,, ;

realtime *name*, *name*, ,,, ;

Variable data types

data type	description	note
reg data type	Used to model storage elements, flip-flops and latches, and combinational logic. Initial value is x.	Signed only if declared *signed*.
integer data type	32-bit signed integer variable. Initial value is x.	
real data type	64-bit floating-point variable. Initial value is 0.	Not synthesizable. Can be used only in simulation.
time data type	64-bit unsigned integer to store simulation time and to check timing dependence. Initial value is 0.	
realtime data type	Realtime data type is used to store simulation time in 64-bit floating-point variable. Realtime declaration can be used interchangeably with real declaration.	

Reg data type can hold a value between assignments and can have arbitrary range, bit width, therefore, it can be used to model hardware registers, flip-flops, and latches.

Also it can model combinational logic.

→ Write your declaration of reg separately for memory elements and for combinational logic.
Give comments to show they must be FF or non-FF.

```
reg ff1, ff2, sig1,ff3, sig2 ;
```

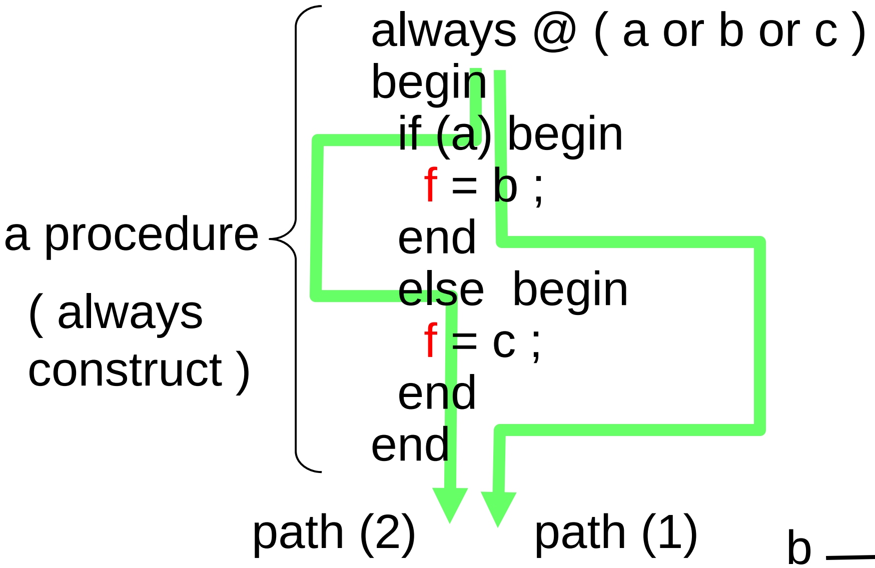
➡

```
reg ff1, ff2, ff3 ;  // FF
reg sig1,  sig2 ;  // non-FF
```

⇨ This is to make it easy for you to check if unintentional latches are generated by a synthesis tool because of your poor coding style.

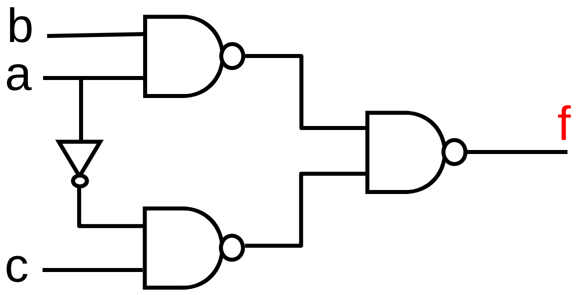
"Reg data type can hold a value between assignments" does not mean that reg data type is equal to memories.

A value of reg data type is memorized in a simulator, not in an actual logic circuits on silicon.



```
reg f ;

always @ ( a or b or c )
begin
  if (a) begin
    f = b ;
  end
  else  begin
    f = c ;
  end
end
```

synthesis

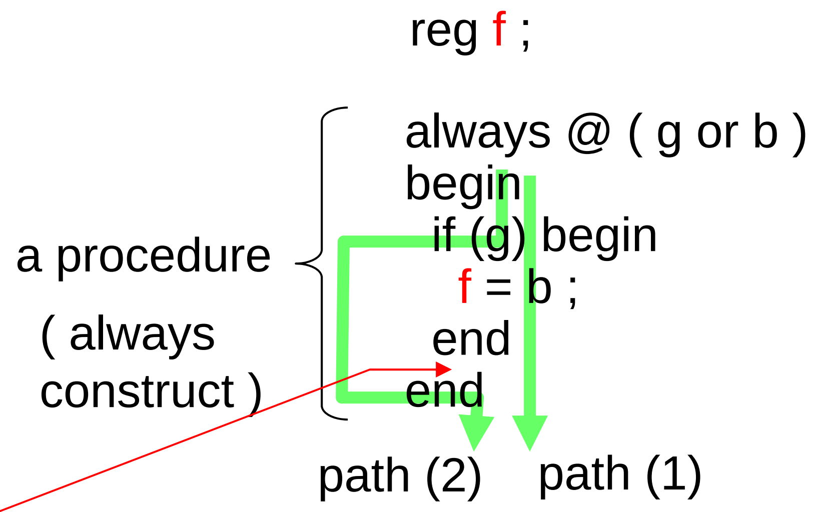


When a procedure on the left is executed in path (1) and f is given the value of b, f holds the same value until the procedure is executed again and f is given a new value in path (1) or (2) depending on a.

A synthesis tool will generate a combinational logic block shown below. It will not generate any memory elements for the procedure on the left.

Note that if the procedure is executed, f is always given the value, b or c.

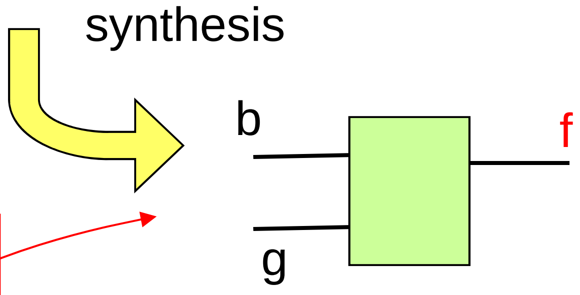
Take another example shown below.



```
reg f ;

always @ ( g or b )
begin
  if (g) begin
    f = b ;
  end
end
```

Suppose b is assigned to f in path (1) and after the assignment if g becomes false then f will not be given a new value any more because path (1) will not be executed. The processing path shall be path (2) while g is false.

And a simulator is supposed to keep the same value of f until the new assignment done. This means that if g is false, f will not change even if b changes. In actual circuits, this is possible only when f is implemented as a memory element such as a latch.



latch

(a kind of memory element)

"else part" is missing.

Note that even if the procedure is executed, f is not given a new value in path (2).

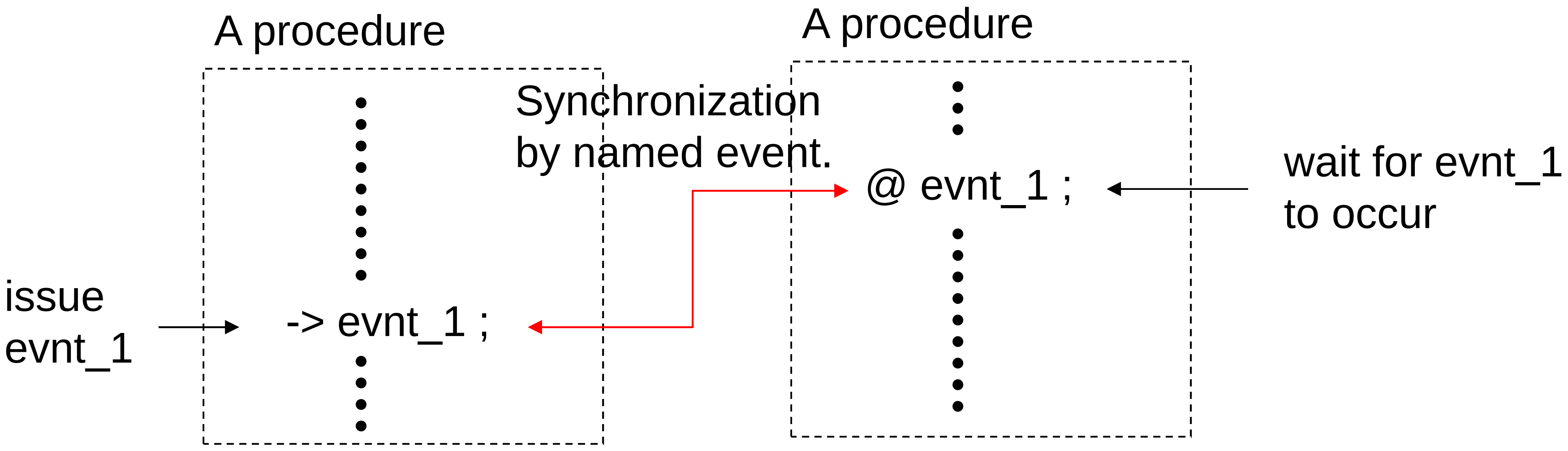
3.4.3 Event

- The execution of a procedural statement can be synchronized with events.
- An event, named event, can be triggered by a Verilog operator "->".
- An event can not hold any value.

syntax

event *name*, *name*, ,,, ;

Named event is available only in simulation. It is not synthesizable.



A procedure

A procedure

Synchronization by named event.

issue evnt_1

-> evnt_1 ;

@ evnt_1 ;

wait for evnt_1 to occur

There are implicit events as described in the table below.

Implicit event

implicit event		description
change direction	posedge	Change to 1 from other than 1, i.e. 0, x, and z.
	negedge	Change to 0 from other than 0, i.e. 1, x, and z.
change value		An expression changes its value.

In case of a vector, only LSB matters for the direction.

Examples;

@ posedge a ; ← Wait for a changes from 0 to 1.

@ (a & b) ; ← Wait for a changes of the value of the expression a & b.

A change from a=0 and b=1 to a=1 and b=0 is not detected as an event.

Q 3.4-1 Answer if the following explanation is correct or not.

(1) If an 1-bit signal changes from 1'bx to 1'b1, it will cause posedge event.

(2) If 1-bit signals, a and b, change their values from 0 to 1 and from 1 to 0 respectively at the same time, it will cause a change value event of a | b, where | is a Verilog OR operator.

(3) If 1-bit signal changes from 1'b1 to 1'bz, it will cause a negedge event.

(4) If a 3-bit signal changes from 110 to 001, it will cause a negedge event, because their value decreased from 6 to 1.

Q 3.4-1 A sample answer.

(1) If an 1-bit signal changes from 1'bx to 1'b1, it will cause posedge event.

Correct: Any change from non-1 to 1 is a posedge event. Therefore, If we set a clock to 1 at time 0, it will cause a clock rise at time 0.

(2) If 1-bit signals, a and b, changes their values from 0 to 1 and from 1 to 0 respectively at the same time, it will cause a change value event of a | b, where | is a verilog OR operator.

Wrong: This will not cause any event because the value of a|b is always 1.

(3) If 1-bit signal changes from 1'b1 to 1'bz, it will cause a negedge event.

Wrong: A negedge event is a change to 0 from other than 0.
1 to z change is a change value event.

(4) If a 3-bit signal changes from 110 to 001, it will cause a negedge event, because their value decreased from 6 to 1.

In case of multi-bit signal, only LSB matters. LSB changes from 0 to 1.
This is a posedge event.

3.4.4 Vector and range specification

Vector is multibit net or reg data type with range specification.

```
for nets;
    wire [MSB:LSB] sig_y ;  // sig_y is unsigned, can not be negative
    wire signed [MSB:LSB] sig_w ; // sig_w can be negative

for reg;
    reg [MSB:LSB] sig_y ;  // sig_y is unsigned, can not be negative
    reg signed [MSB:LSB] sig_w ; // sig_w can be negative
```

"signed" keyword was introduced in Verilog2001.

Implementation may set a limit on maximum length of a vector, but it shall be at least 2**16 bits.

[1] Each bit of a vector can independently take any value of 0, 1, x, and z.

[2] A part of a vector is accessible by using expressions shown on the next page.

Note: Range declaration is allowed only for nets and reg data types. (Range declaration for integer, real, realtime, and time data types are illegal.)

integer [31:0] k ; ← illegal

integer k ;

y = k[5:2] ; // k is integer ← OK

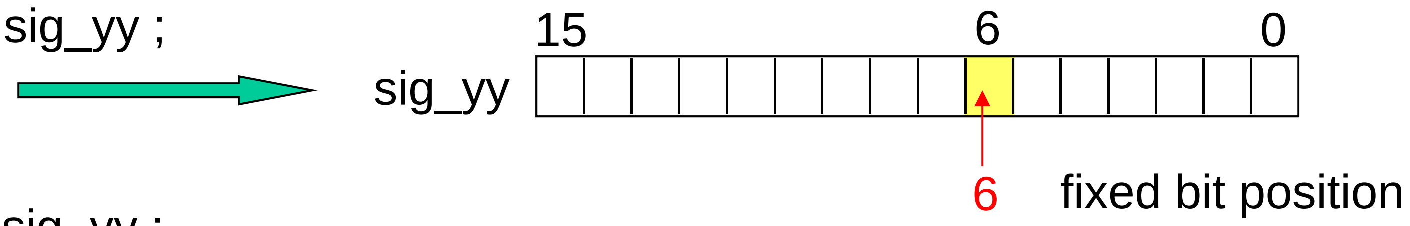
Part selection of vector

type		expression example	meaning
bit select	constant	wire [15:0] sig_yy ; sig_yy [6]	bit-6 of 16-bit variable sig_yy
	indexed	wire [15:0] sig_yy ; wire [3:0] idx_yy ; sig_yy [idx_yy]	A bit specified by idx_yy. If idx_yy is 5, bit-5 of 16-bit variable sig_yy
part select	constant	wire [15:0] sig_yy ; sig_yy [10 : 6]	bit-10, bit-9, bit-8, bit-7, and bit-6, of 16-bit variable sig_yy
	indexed *1	wire [15:0] sig_yy ; integer k ; parameter B_WD=4 ; sig_yy [k + : B_WD]	B_WD bits of sig_yy, staring from bit-k toward MSB (if specified k **-:** B_WD, the toward LSB). k must be integer data type. B_WD must be a constant. If k=7, and BW_D=4, bit-10, 9, 8, and 7 of sig_yy. (if k **-:** B_WD, bit 7, 6, 5, and 4) *2

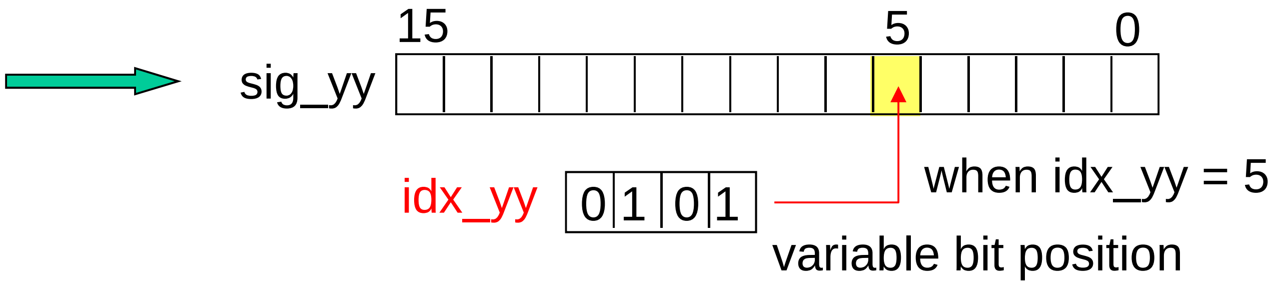
*1: Indexed part select is available in Verilog2001.

*2 : If MSB<LSB, then k + : B_WD means bit-7, 8, 9, and 10.

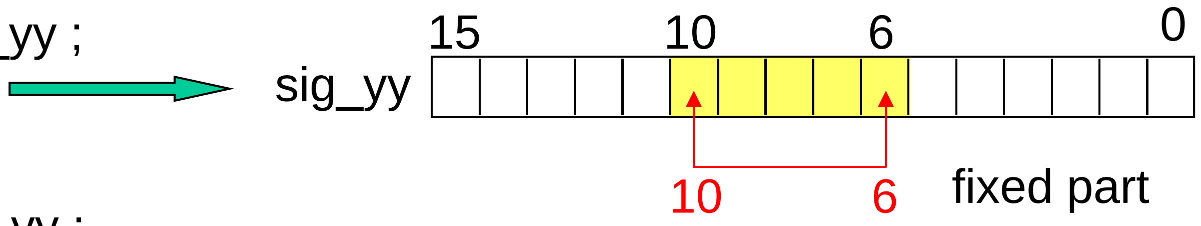
```
wire [15:0 ] sig_yy ;
sig_yy [ 6 ]
```



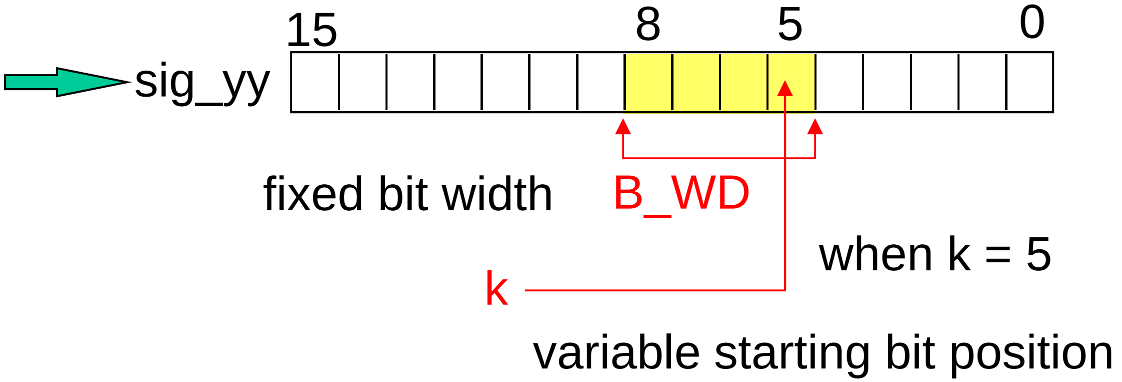
```
wire [15:0 ] sig_yy ;
wire [3:0] idx_yy ;
sig_yy [ idx_yy ]
```



```
wire [15:0 ] sig_yy ;
sig_yy [ 10 : 6 ]
```



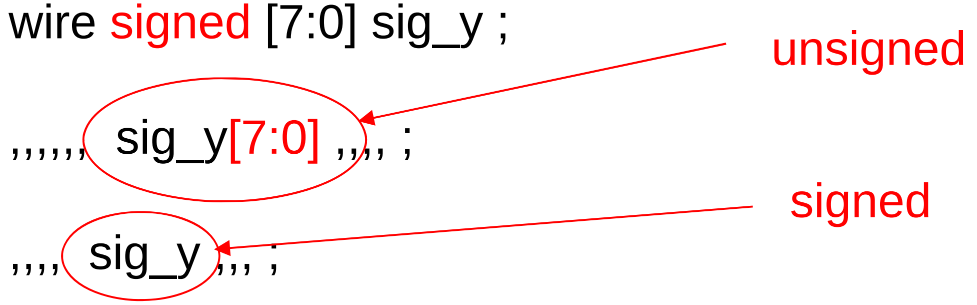
```
wire [15:0 ] sig_yy ;
integer k ;
parameter B_WD=4 ;
sig_yy [ k + : B_WD ]
```



Important

Part-select and bit select results are unsigned regardless of the operands even if part-select specifies entire vector.

```
wire signed [7:0] sig_y ;
```



```
,,,,,, sig_y[7:0] ,,,,, ;
,,,, sig_y ,,,, ;
```

Do not use entire bit specification such as sig_y[7:0] for 8-bit vector data sig_y.
Use just sig_y only.

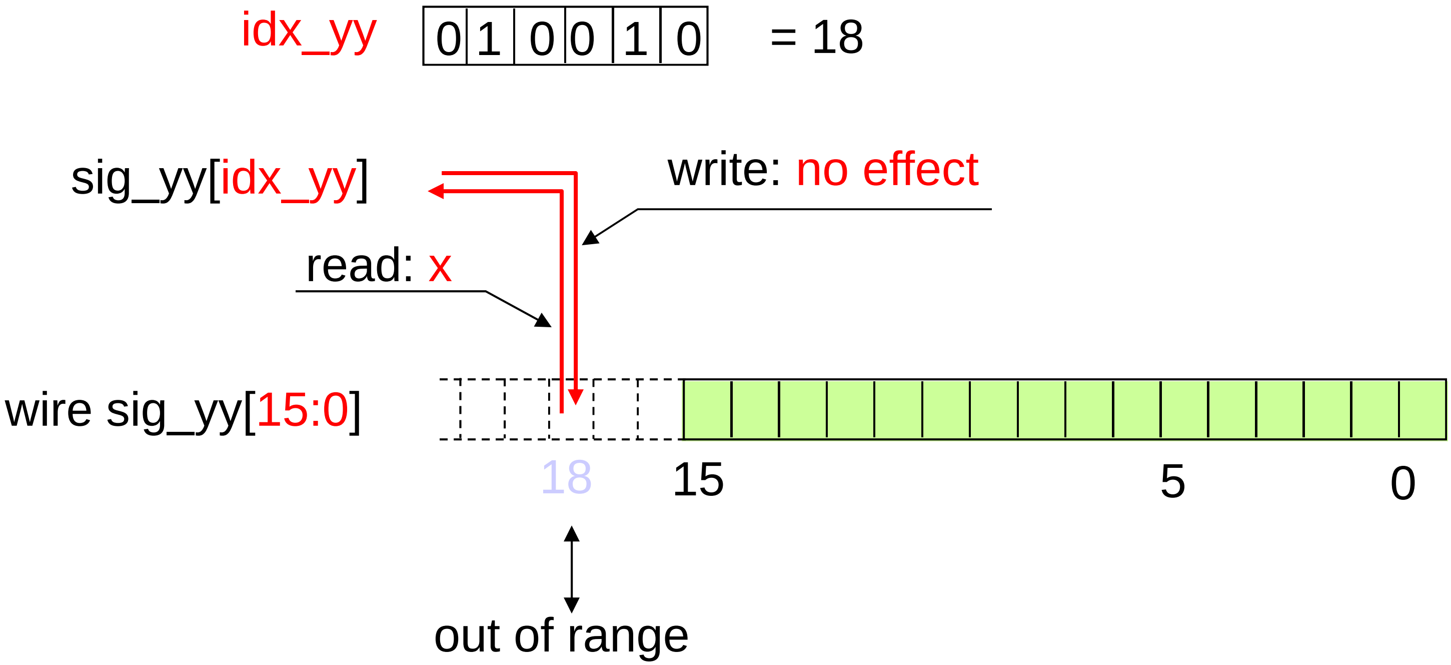
Part select and bit select is not permitted for vectors declared with "vectored" keyword.

```
wire vectored [15:0] a ;

        assign a[7:0] = abc ;
```

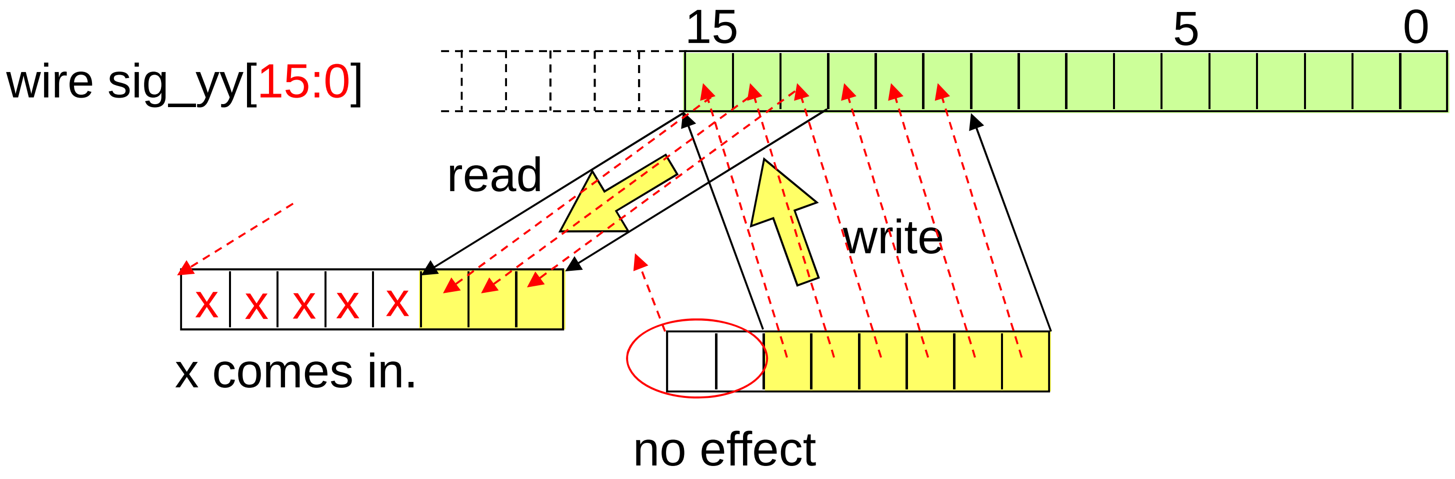
Part select can not be used in L value (Left-hand side value) because a is declared with vectored keyword.

Do not use "vectored" or "scalared" keyword in data type specification to avoid implementation dependency.

A part-select of any type that addresses a range of bits that are completely out of the address bounds of the net, reg, integer, time variable, or parameter or a part-select that is x or z shall yield the value x when read and shall have no effect on the data stored when written.



Part-selects that are partially out of range shall, when read, return x for the bits that are out of range and shall, when written, only affect the bits that are in range. (see next page)



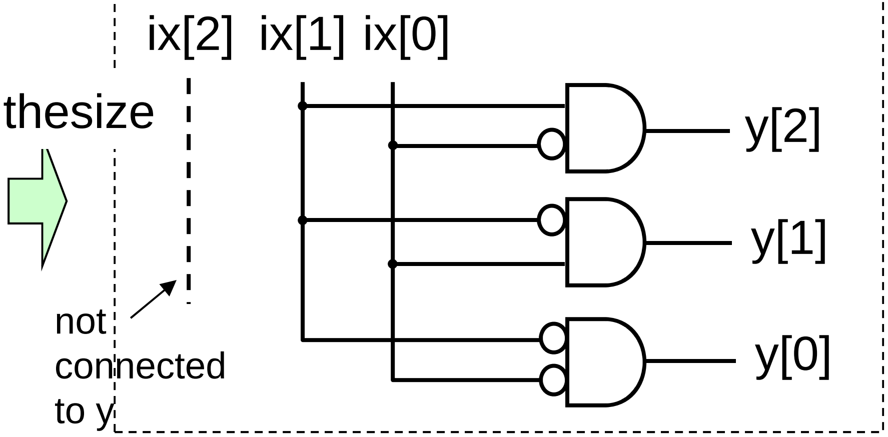
However, if out of range happens in gate level simulation or in actual device, some data must be written into the target signal for write operation and some data, 0 or 1, will come out from the target signal for read operation.

RTL code

```
reg [2:0] ix ; // can be 0 to 7
reg [2:0] y ; // 3 bits data

// logic to assign 1 to only 1 bit
y = 0 ; // clear all bits to 0
y[ix] = 1'b1; //assign 1 to bit ix
```

synthesize

Gates generated



If ix becomes 5, then y will be 0 in simulation.

In circuit, y will become binary 010 if ix becomes 5.

Bit numbering and MSB/LSB

In Verilog language, there is no limitation for MSB and LSB such as MSB must be larger than LSB. Therefore, the following declaration is legal.

wire [0:7] sig_w ; ← MSB, 0, is smaller than LSB, 7.

By using [0:15] style, reversing bit order is possible. However, as shown on the next page, it does not affect the value itself.

⇨ Do not use [0:n] style vector declaration. It will cause many bugs. It can only be used in a very limited part of the code such as changing big endian to/from little endian system.

```
wire [3:0] sig_y ;
wire [0:3] sig_w ;

assign sig_y = 4'b1100 ;
assign sig_w = sig_y ;
```

The result will be;

sig_w[0] = 1, ← MSB
sig_w[1] = 1,
sig_w[2] = 0,
sig_w[3] = 0 ← LSB

As a numerical value, sig_y is 12 (4'b1100) and sig_w is also 12.
sig_w can not become 3 (4'b0011) by the above assignment.

```
wire [15:0] sig_y ;
wire [0:15] sig_w ;
wire [15:0] sig_sum ;

assign sig_y [15:0] = 16'd94 ;
assign sig_w [0:15] = 16'd35 ;
assign sig_sum [15:0] =
            sig_y [15:0] + sig_w [0:15] ;
```

Even if the bit order is reversed, the numerical value does not change at all, therefore in the example shown on the left, sig_sum is 94+35=129.

If we write ;

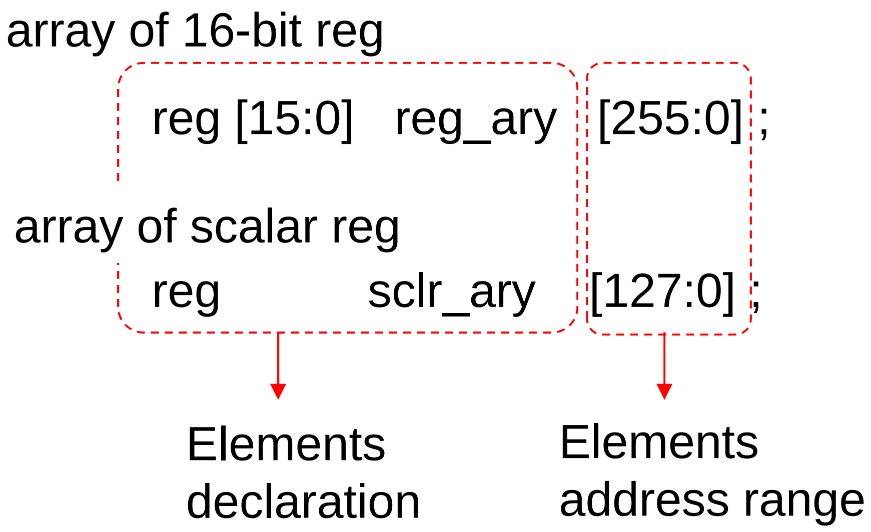
    sig_y [7:0] + sig_w [7:0]

with vector declaration wire [0:7] sig_w
, it will cause an error.

3.4.5 Array declaration

(1) array

Arrays can be used to group elements of a data type into multidimensional objects.
Arrays shall be declared by specifying the element address range(s) after the declared identifier.
An expression that specifies the indices of the array shall be a constant integer expression.



array of 16-bit reg

```
reg [15:0]   reg_ary   [255:0] ;
```

array of scalar reg

```
reg          sclr_ary  [127:0] ;
```

Elements declaration

Elements address range

Only one entire element of an array is accessible.

reg_ary[3:0][228] is illegal.

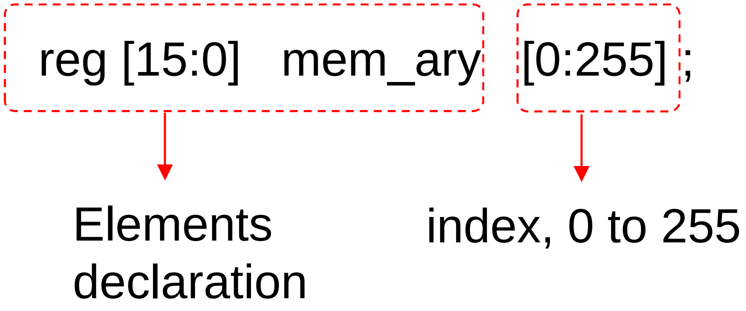
reg_ary [96] is OK, meaning 96th element of reg_ary

Do not use an array unless it makes the code simple and easy to understand.

Array size may be implementation dependent, but it must be able to define at least 2**24 elements.

(2) Memory

Memory is an one dimensional array with elements of reg data type.



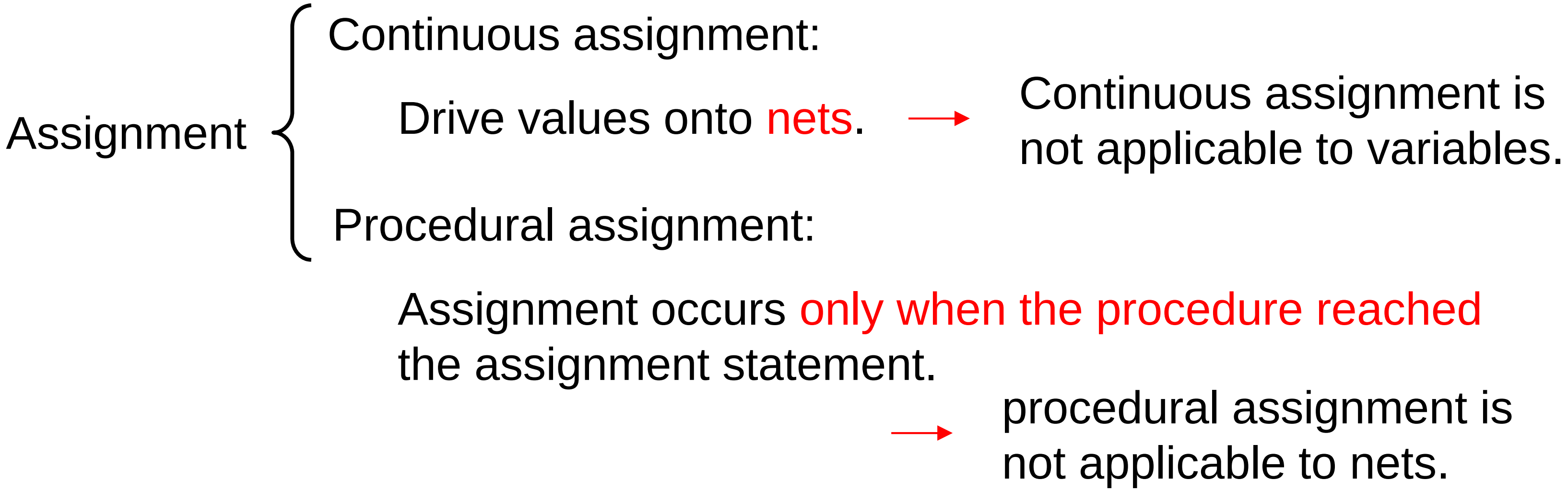
```
reg [15:0]   mem_ary [0:255] ;
```

Elements declaration

index, 0 to 255

Note that memory can not be mapped to an actual memory logic circuit such as DRAM by a synthesis tool, it can just model such memory elements.

3.5. Assignment



Assignment

Continuous assignment:

Drive values onto nets. → Continuous assignment is not applicable to variables.

Procedural assignment:

Assignment occurs only when the procedure reached the assignment statement.

→ procedural assignment is not applicable to nets.

Assignment	Constraints on LHS	Assignment timing
Continuous assignment	LHS must be net data type	Assignment occurs whenever the value of Right-Hand Side, RHS, changes.
Procedural assignment	LHS must be variable data type	Assignment occurs only when the control flow of the procedure reached the assignment statement.

Sizing in assignment;

bit width	Sizing
LHS < RHS	MSBs of the RHS will always be discarded. Truncating the sign bit of a signed expression may change the sign of the result.
LHS = RHS	No sizing occurs.
LHS > RHS	Sizing to LHS occurs as shown in the table below.

Bit size extension rule

LHS \ RHS	signed	unsigned
signed	RHS's sign-bit (MSB) is extended	0 fill
unsigned	⇧	⇧

Note that signed or unsigned is determined only by RHS itself.
LHS has no effect on determining the type of the expression of RHS.

Always make sure that bit width of LHS and RHS are equal to avoid troubles.

```
wire signed [7:0] sig_a, sig_b ;
```

```
assign sig_a = 4'sbx100 ;
```

sig_a	x	x	x	x	x	1	0	0

sign bit “x” is extended.

```
assign sig_b = 4'bx100 ;
```

sig_b	0	0	0	0	x	1	0	0

3.5.1 Continuous assignment

Continuous assignment

- Continuous assignment statement
- Net declaration assignment

- In simulation, “continuous assignment” is continuously executed. Whenever an input operand of the expression, Right-Hand Side (RHS), changes value, the new updated value is assigned to Left-Hand Side (LHS).

- In synthesis, “continuous assignment” is synthesized line by line. A continuous assignment will be mapped to a wire connection to LHS.

Restrictions

[1] Continuous assignment can not drive variables, therefore LHS must be net data type.

LHS must be net (vector or scalar), constant bit-select of a vector net, constant part-select (Indexed part select not allowed) of a vector net, constant indexed part-select of a vector net, or concatenation or nested concatenation of any of the above left-hand side.

[2] Continuous assignment must not appear in procedures.

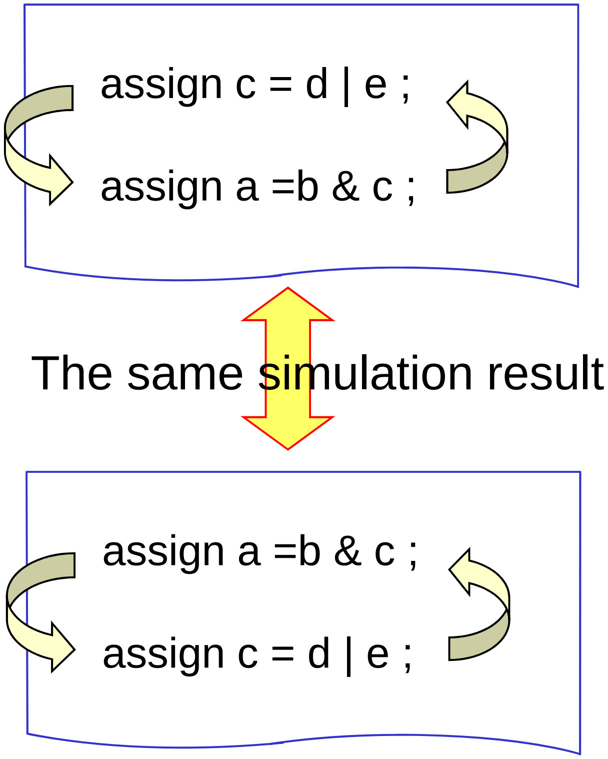
syntax of continuous assignment

	Continuous assignment statement	Net declaration assignment
syntax	assign y = expression ; y must be a net data type.	wire y = expression ; Range specification is allowed.
example	assign y = a + b ; assign w = c & d ;	wire [15:0] wd_a ; wire [7:0] up_byt_a = wd_a[15:8]; wire [7:0] lw_byt_a = wd_a[7:0] ;

note	The following is allowed, but, do not use it to avoid possible mistakes.
	assign
	y = a + b ,
	w= c & d ,
	q = e – f ;
		It is not recommended to use "net declaration assignment" for a complex expression, such as "wire y = a & ((b^c)|(e|f)) ;". It can be used only for defining a signal itself as shown in the example above.

In simulation, continuous assignments are executed always whenever values of the operands are updated.

Therefore, exchanging the lines of continuous assignment has no influence over the simulation result.



```
assign c = d | e ;

assign a =b & c ;
```

The same simulation result

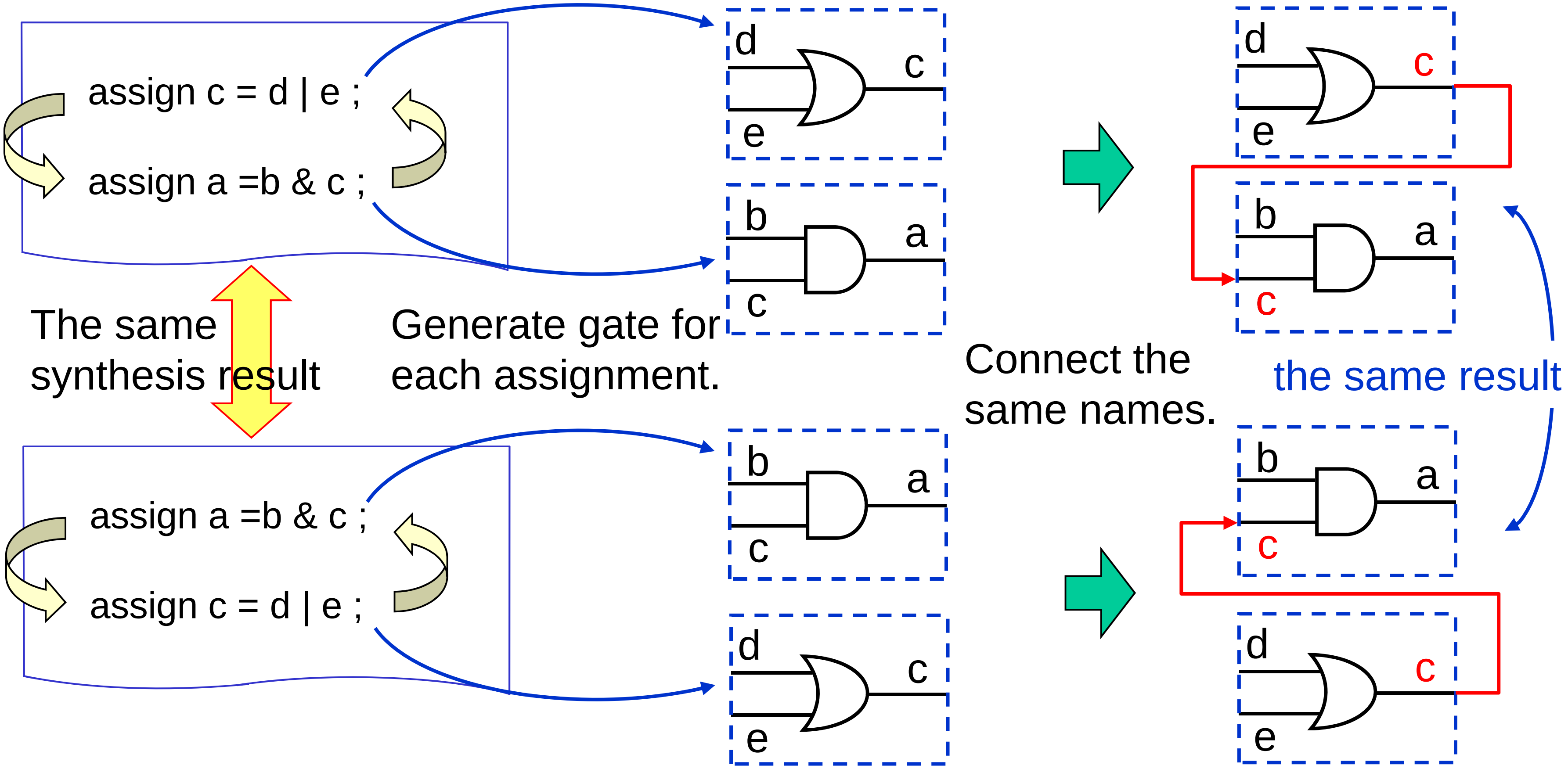
```
assign a =b & c ;

assign c = d | e ;
```

If d is updated in simulation, c will change its value.

And c’s change will cause the change of the expression b & c, therefore a will be given the new value based on the updated c.

In synthesis, a tool will create a wire connection to LHS for a continuous assignment, and connects signals together having the same name.

Therefore, exchanging the lines of continuous assignment has no influence over the synthesis result.



```
assign c = d | e ;

assign a =b & c ;
```

The same synthesis result

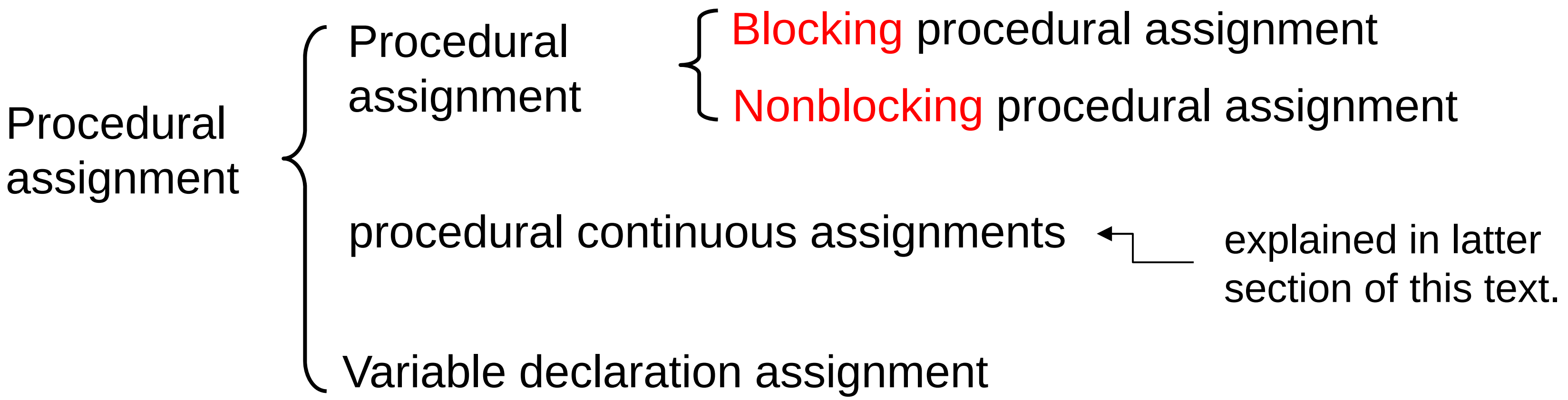
```
assign a =b & c ;

assign c = d | e ;
```

Generate gate for each assignment.

Connect the same names.

the same result

3.5.2 Procedural assignment



Procedural assignment

Procedural assignment

Blocking procedural assignment

Nonblocking procedural assignment

procedural continuous assignments

explained in latter section of this text.

Variable declaration assignment

- In simulation, “procedural assignment” is executed only when the flow of execution in the simulation reaches the assignments.

- In synthesis, procedural assignments are not synthesized "sentence by sentence" basis, but are synthesized "procedure by procedure" basis.

Restrictions

[1] Procedural assignment can not drive nets, therefore LHS must be variable data types.

[2] Procedural assignment is allowed only in procedures. But blocking and nonblocking procedural assignments must not coexist in a procedure. If coexisting, the synthesis tool will report an error.

(In simulation, coexistence may not cause error.)

[3] For LHS, only the following are allowed;

reg, integer, real, realtime, or time data type;
Bit-select and part-select of a reg, integer, or time data type;
Memory word;
Concatenation or nested concatenation of any of the above.

syntax of procedural assignment

type	procedural assignment		variable declaration assignment *1
	Blocking	Nonblocking	
syntax	yy = expression ; yy must be variable data type.	yy <= expression ;	reg yy = constant expression ; Range specification is allowed for reg data type.

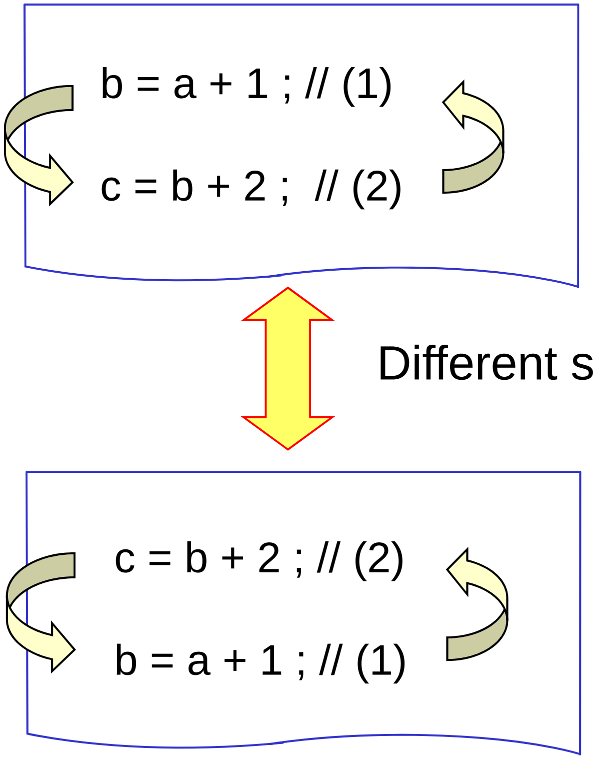
Exa-mple	b = a + 1 ;
	w = b | c ;
		b <= a + 1 ;
		w <= b | c ;
			reg [3:0] yy = 4'b1010 ;
			integer k = 1 ;

mea-ning	w=b|c is executed after b is updated by b=a+1. It blocks the execution of the next statement. If LHS requires an evaluation, it shall be evaluated at the time specified by the intra-assignment timing control.	b|c is evaluated before b is updated by b<=a+1. It does not block the execution of the next statement.	The above code is equivalent to reg [3:0] yy ; integer k ; initial yy = 4'b1010 ; initial k = 1 ; Do not use variable declaration assignment. It is not synthesizable.

*1: Introduced in Verilog2001. Applicable to other variables.

In simulation, procedural assignments are executed only when the flow of execution in the simulation reaches the assignments. Therefore, exchanging the lines of procedural assignment has influence over the simulation result when blocking procedural assignment is used.

Suppose a=1 and b=1 before executing the following two lines;



```
b = a + 1 ; // (1)

c = b + 2 ;  // (2)
```

Different simulation result

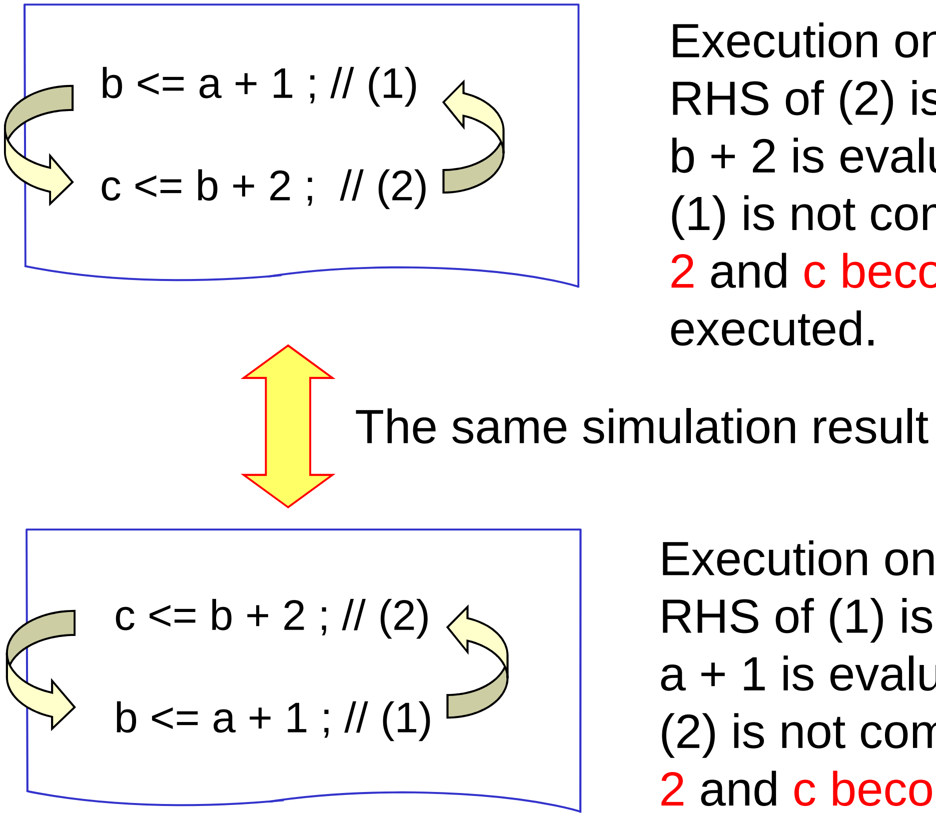
```
c = b + 2 ; // (2)

b = a + 1 ; // (1)
```

Execution on (2) is blocked by (1).
(2) is executed after (1), assignment to b, completed.
Therefore, b becomes 2 by (1), and then c becomes 4 by (2), eg. 2 + 2.

Execution on (1) is blocked by (2).
(1) is executed after (2), assignment to c, completed.
Therefore, c becomes 3 by (2), eg. 1 + 2.
And then b becomes 2 by (1), eg. 1 + 1.

However, if we use nonblocking procedural assignment, then the results are different from the results shown on the previous page.

Suppose a=1 and b=1 before executing the following two lines;



```
b <= a + 1 ; // (1)

c <= b + 2 ;  // (2)
```

Execution on (2) is not blocked by (1).
RHS of (2) is evaluated before (1) completed.
b + 2 is evaluated to be 3, eg. 1 + 2, because (1) is not completed yet. Therefore, b becomes 2 and c becomes 3 after these two lines are executed.

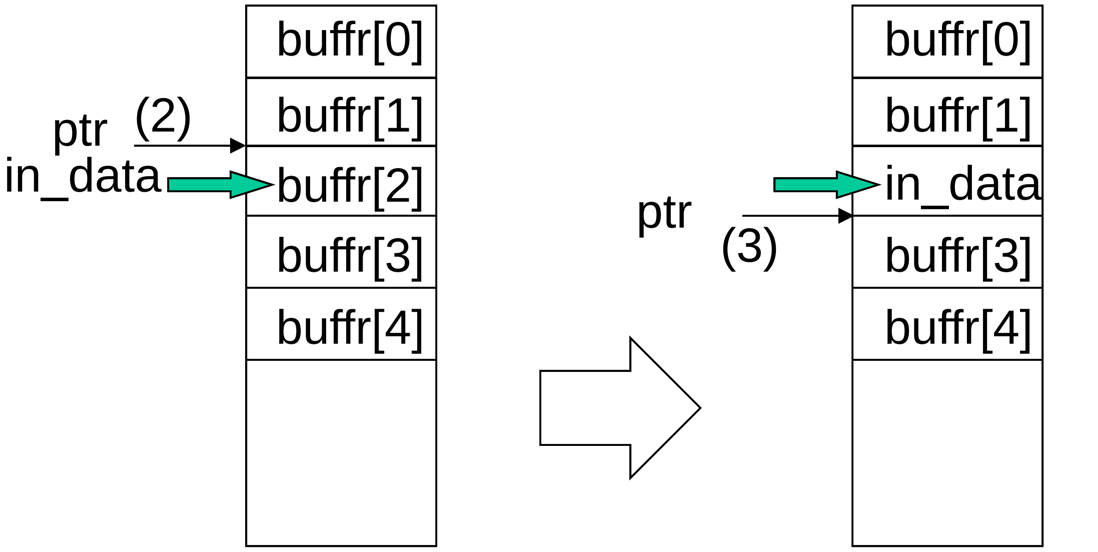
The same simulation result

```
c <= b + 2 ; // (2)

b <= a + 1 ; // (1)
```

Execution on (1) is not blocked by (2).
RHS of (1) is evaluated before (2) completed.
a + 1 is evaluated to be 2, eg. 1 + 1, because (2) is not completed yet. Therefore, b becomes 2 and c becomes 3 after these two lines are executed.

Nonblocking procedural assignment causes parallel processing.

Take an example of the following code. This code is to save data to buffr using a pointer ptr. ptr must be updated by one after in_data is stored to buffr pointed by ptr.



before execution of the procedure.

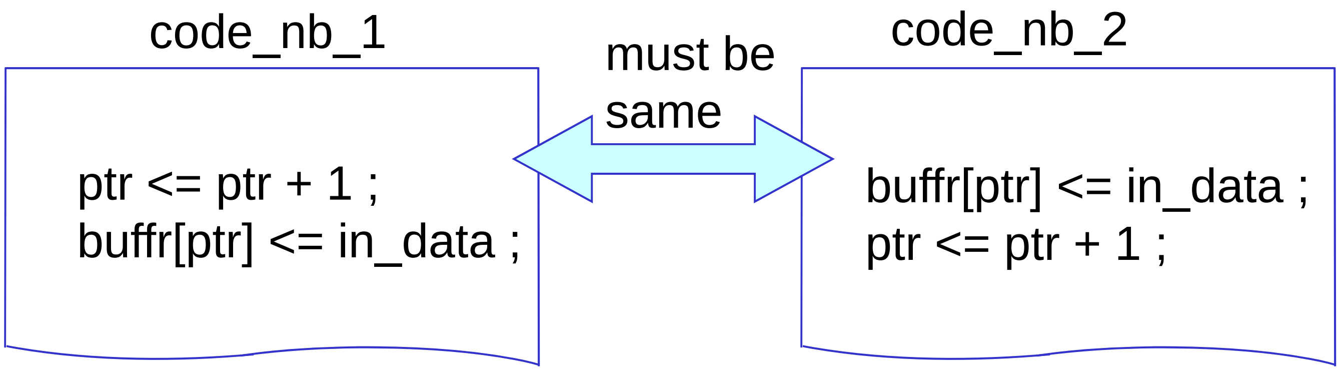
after execution of the procedure.

code_nb_1

```
ptr <= ptr + 1 ;
buffr[ptr] <= in_data ;
```

This code may look strange but the assignments used are nonblocking procedural assignment.

Because code_nb_1 uses nonblocking procedural assignment, it is equal to code_nb_2. This means both code can work correctly.



code_nb_1

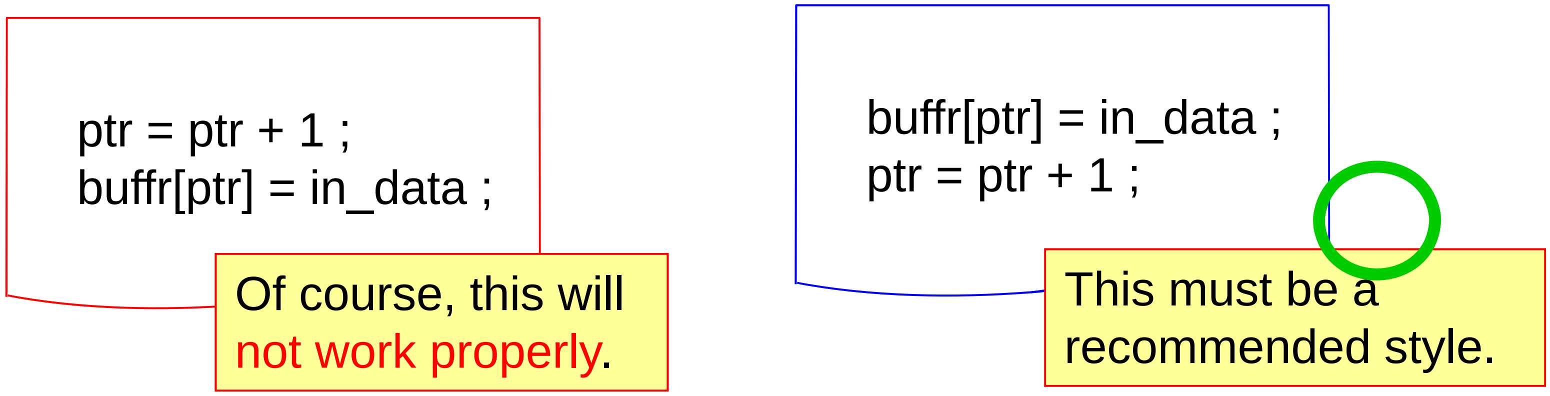
```
ptr <= ptr + 1 ;
buffr[ptr] <= in_data ;
```

must be same

code_nb_2

```
buffr[ptr] <= in_data ;
ptr <= ptr + 1 ;
```

However, code_nb_1 is not a good programming style because it looks strange to a person who is accustomed to serial execution.
Use the coding style shown on the bottom right unless using nonblocking procedural assignment is really needed.



```
ptr = ptr + 1 ;
buffr[ptr] = in_data ;
```

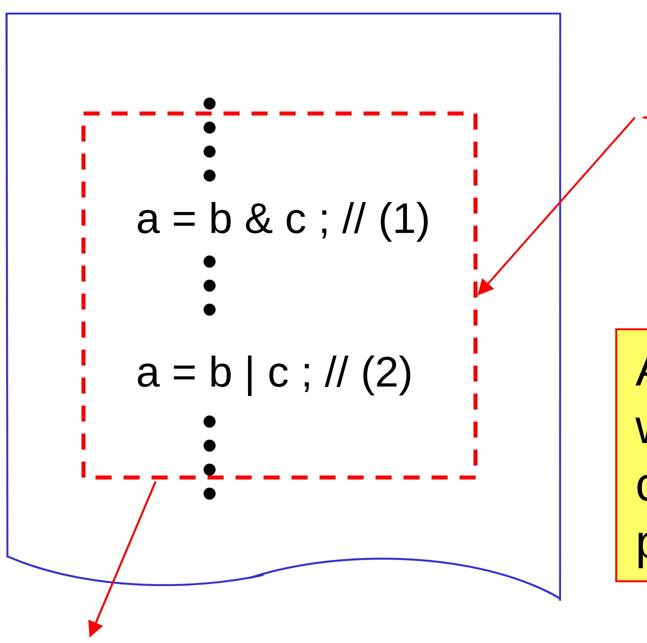
Of course, this will not work properly.

```
buffr[ptr] = in_data ;
ptr = ptr + 1 ;
```

This must be a recommended style.

A synthesis tool generates a logic block on a "procedure by procedure" basis.

It must not be expected that the tool will generate gates corresponding to every procedural assign statement.



```
⋮
a = b & c ; // (1)
⋮
a = b | c ; // (2)
⋮
```

a procedure

Even if a designer uses AND operator in (1) and OR operator in (2), a synthesis tool may not generate AND gate nor OR gate for (1) and (2) respectively.

A tool is not responsible for using logic gates written in RTL, but is responsible for generating circuit blocks which are logically equivalent to procedures.

A designer can not control what gates shall be used by controlling RTL source code description.

⟺

A synthesis tool will generate the optimal logic gates on behalf of a designer.

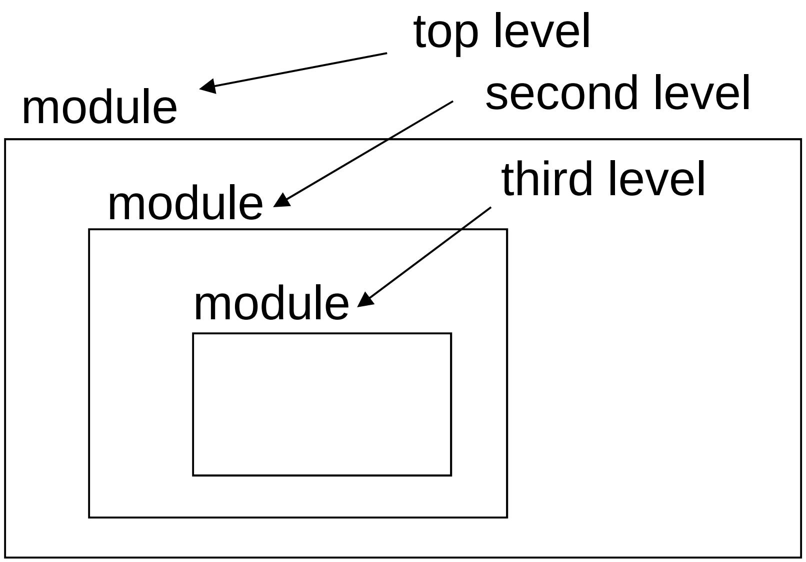
3.6. Module

A module is a basic entity in Verilog that can be simulated and synthesized.

- A module can have input and/or output.
  A module without any input or output is allowed. But to be able to communicate with other modules, it must have input and/or output.

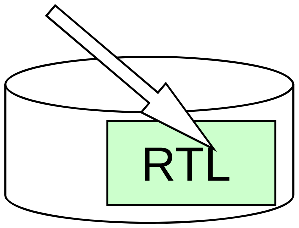
- It begins with keyword “module” and ends with “endmodule”.

- It can incorporate other modules by instantiating them and can have a hierarchical structure as shown below.

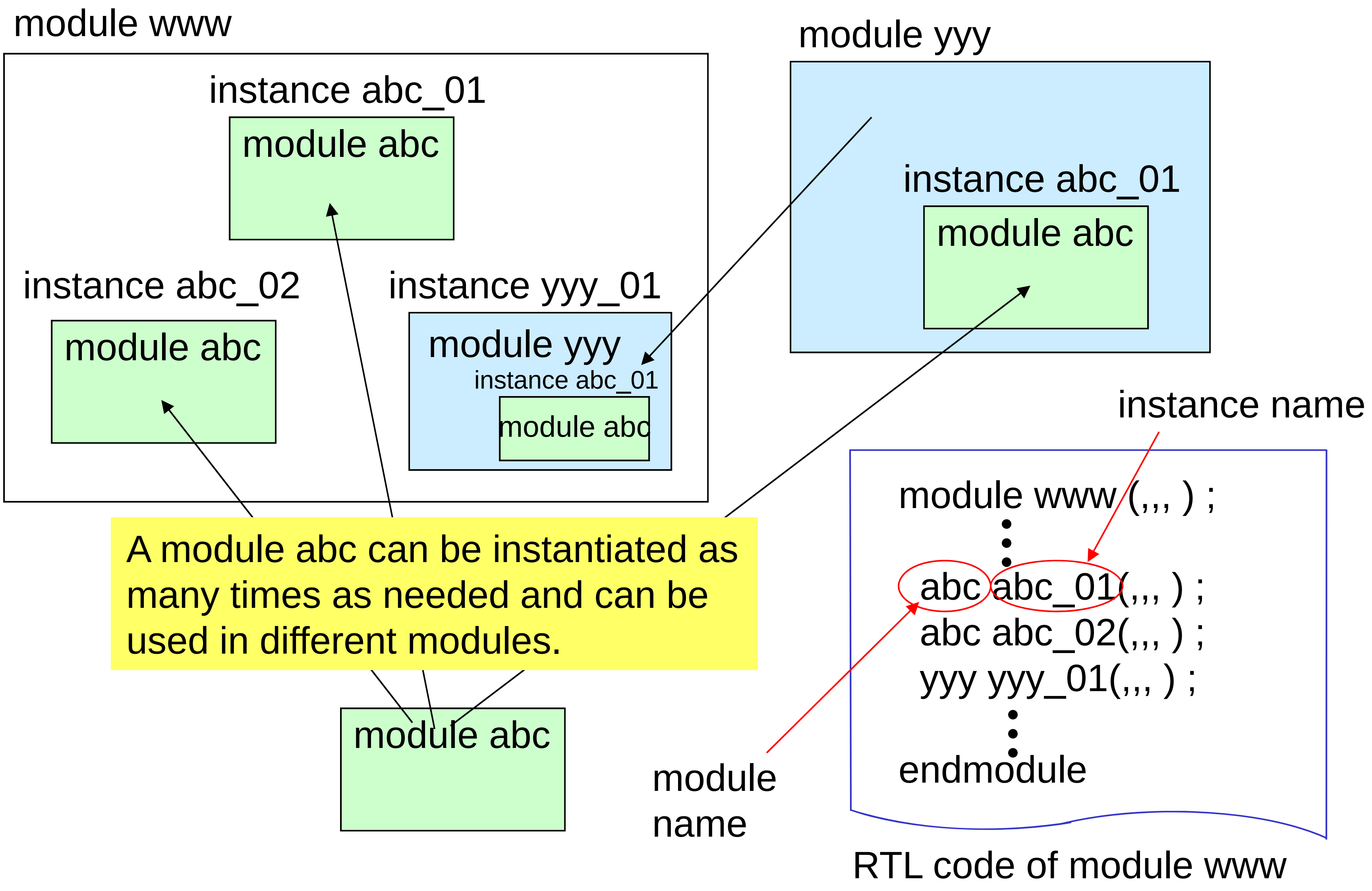


Hierarchical structure of modules

It is recommended to name a file for RTL source "*module_name* .v".



A module must be instantiated when it is incorporated into the other modules.

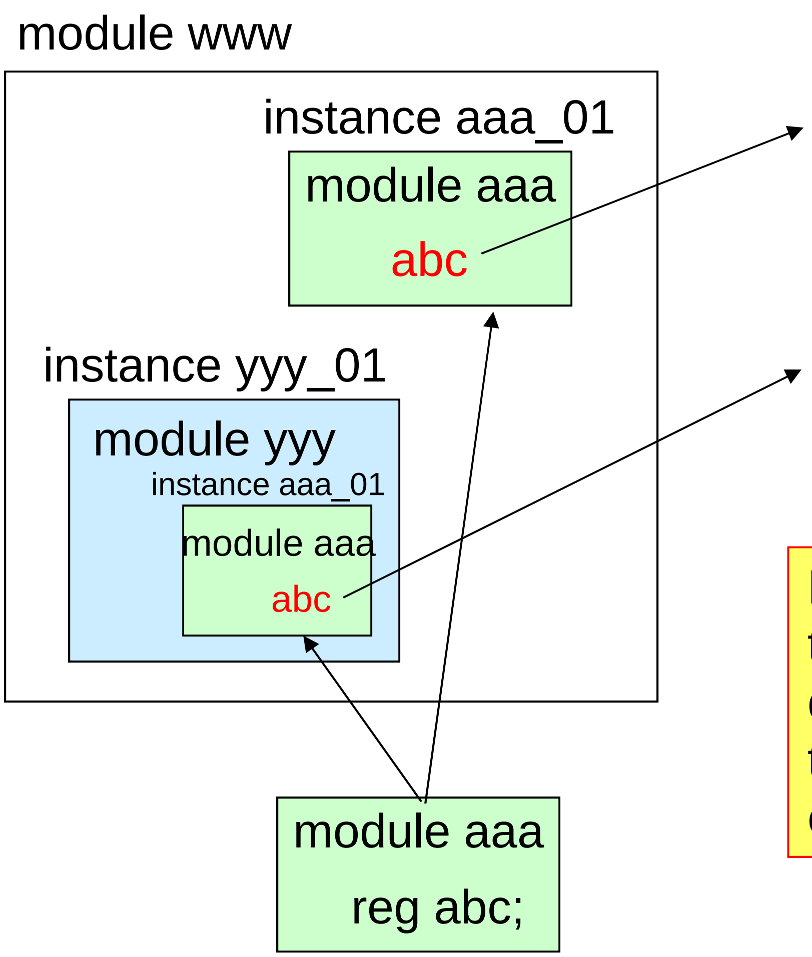


A module abc can be instantiated as many times as needed and can be used in different modules.

```
module www (,,, ) ;
        :
  abc abc_01(,,, ) ;
  abc abc_02(,,, ) ;
  yyy yyy_01(,,, ) ;
        :
endmodule
```

Hierarchical name

A variable abc defined in module aaa can be referenced in module www by using "hierarchical name".

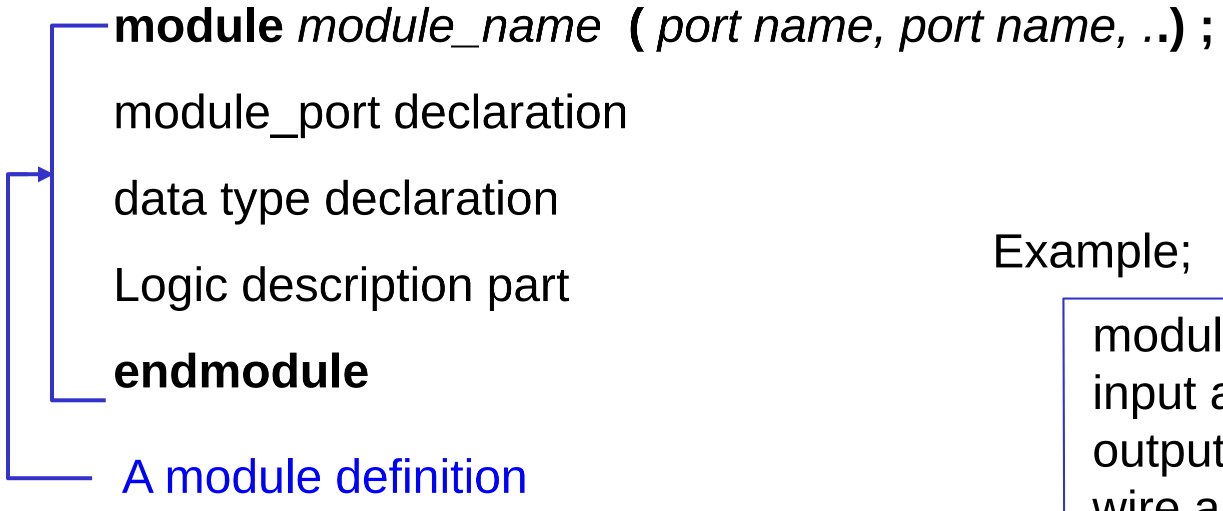


This variable is accessible by a name aaa_01.abc.

This variable is accessible by a name yyy_01.aaa_01.abc.

Do not use hierarchical name to refer to lower level Verilog items except for debugging, i.e. checking variables/nets that are not declared as ports (explicit connection to upper level modules).

A syntax for a module is shown below;



module *module_name* **(** *port name, port name, ..* **) ;**

module_port declaration

data type declaration

Logic description part

endmodule

A module definition

Example;

```
module and_logic ( a, b, c ) ;
input a, b ;
output c ;
wire a, b ;
wire c;
assign c = a & b ;
endmodule
```

List up port in the sequence of input, output, and inout port.
For inputs, list up in the sequence of clock signal, reset signal, and then other input signals.

3.6.1 Module port declaration

module *module_name* **(** *port name, port name,* **..) ;**

module_port declaration ← Declare whether the ports are input and/or output.

input <port size> *port name, port name*, ... ;
output <port size> *port name, port name*, ... ;
inout <port size> *port name, port name*, ... ;

Restrictions

[1] An input or inout port shall be of type net.

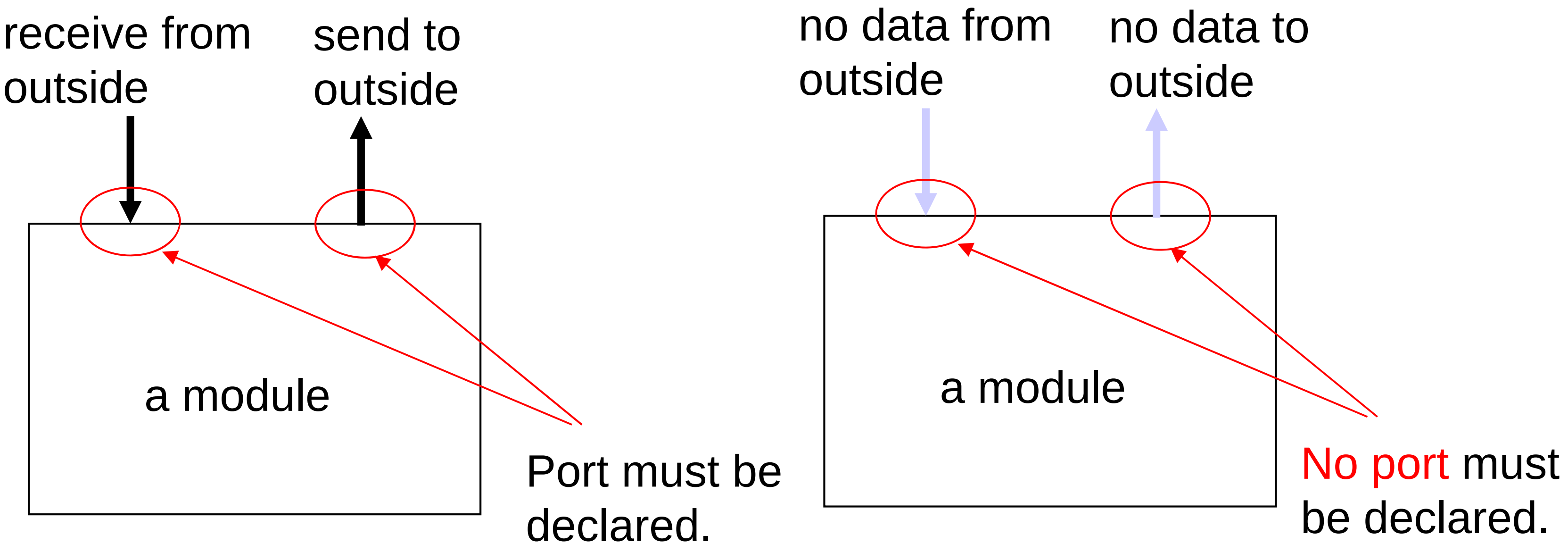
[2] Variables external to a module can not be connected to output port or inout port of the module.

[3] Port can be
a simple identifier or escaped identifier,
a bit-select of a vector declared within the module,
a part-select of a vector declared within the module, or
a concatenation of any of the above.

[4] Size specification must be identical to the range declaration used in data type declaration if the size is specified.

The keyword "signed" can be used to declare a signed port.

Port declaration is needed only when a module has to receive data from outside the module or to send data to outside the module.
If a module does not have to receive data from outside nor to send data to outside, we must not declare a port for such modules.



receive from outside

send to outside

a module

Port must be declared.

no data from outside

no data to outside

a module

No port must be declared.

Port declaration together with data type declaration is allowed as below;

```
input wire [7:0] in_a ;
output reg [15:0] out_b ;
```

○

But if the above style is used, then declaring data type again for those ports is illegal.

```
input wire [7:0] in_a ;
output reg [15:0] out_b ;

wire [7:0] in_a ;
reg [15:0] out_b ;
```

✕

Do not use this style.
Use the following style,
separately declare port and data type.

```
input     [7:0] in_a ;
output [15:0] out_b ;

wire [7:0] in_a ;
reg [15:0] out_b ;
```

◎

List_of_port_declarations

Ports can be listed entirely in module port list as shown left below;

An example of list_of_port_declarations

```
module abc (
    input wire [3:0] a , b,
    output reg [7:0] c
                   ) ;
```

It is illegal to declare port and/or data type again in the module.

Conventional style

```
module abc ( a, b, c ) ;
  input [3:0] a, b ;
  output [3:0] c ;
  wire [3:0] a, b ;
  reg [7:0] c ;
```

Although the code on the left is allowed, use the style above to keep the same style recommended in current company standard.

Port name different from internal net/variable.

Port name can be assigned different from internal name as shown below;

```
module abc ( a, b,  .p_name(intnl_name), ,,, ) ;
 input ,,,, ;
 ,,
 input intnl_name ;
 ,,,
```

This port is accessible from outside the module by a name "p_name"

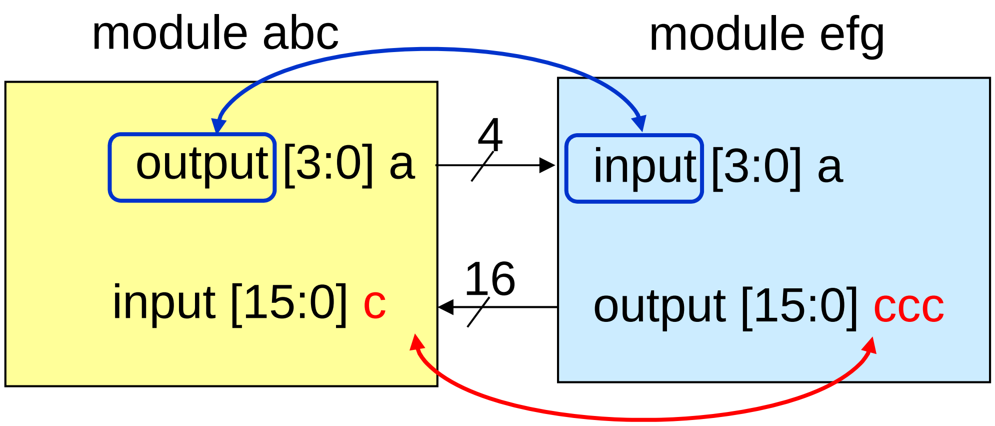
This can be useful to name the port such as { a, b } as below;

```
module abc ( .fg( { a, b } ), c ) ;
 input [7:0] a, b ;
 output [7:0] c ;
 wire [7:0] a, b ;
 wire [7:0] c ;
 assign c = a + b ;
endmodule
```

This module can receive a 2-byte signal as input and return the sum of an upper byte and a lower byte. The input port is accessible by the name "fg".

However, do not use port name different from internal names unless using a different name is really needed.

Port connection



module abc

output [3:0] a

input [15:0] c

4

16

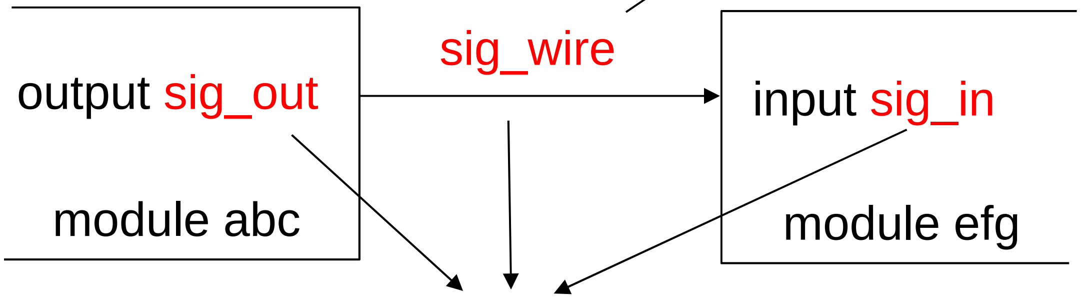
module efg

input [3:0] a

output [15:0] ccc

Using “sig_out”
is recommended.

When connecting the ports among modules, input port must be connected to other module’s output port.
Those ports connected together may have different names.



output sig_out

module abc

sig_wire

input sig_in

module efg

Using different names is OK.
For a connecting net's name, using output port name is often better.

port connection

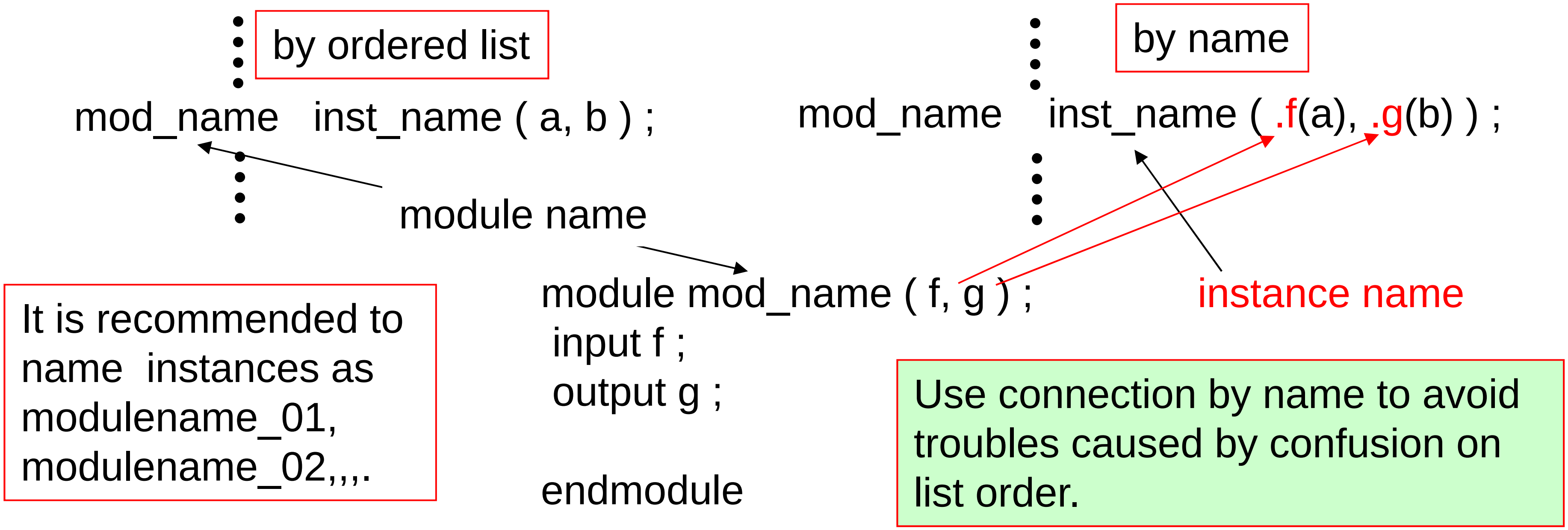
- Port connection by ordered list

  Port expressions listed for a module instance shall be in the same order as the port listed in the module declaration.

- Port connection by name

  Names in the module port declaration are used to identify the connections.

In the following examples, a and b are connected to f and g of module mod_name respectively.



by ordered list

```
:
mod_name   inst_name ( a, b ) ;
:
```

module name

by name

```
:
mod_name   inst_name ( .f(a), .g(b) ) ;
:
```

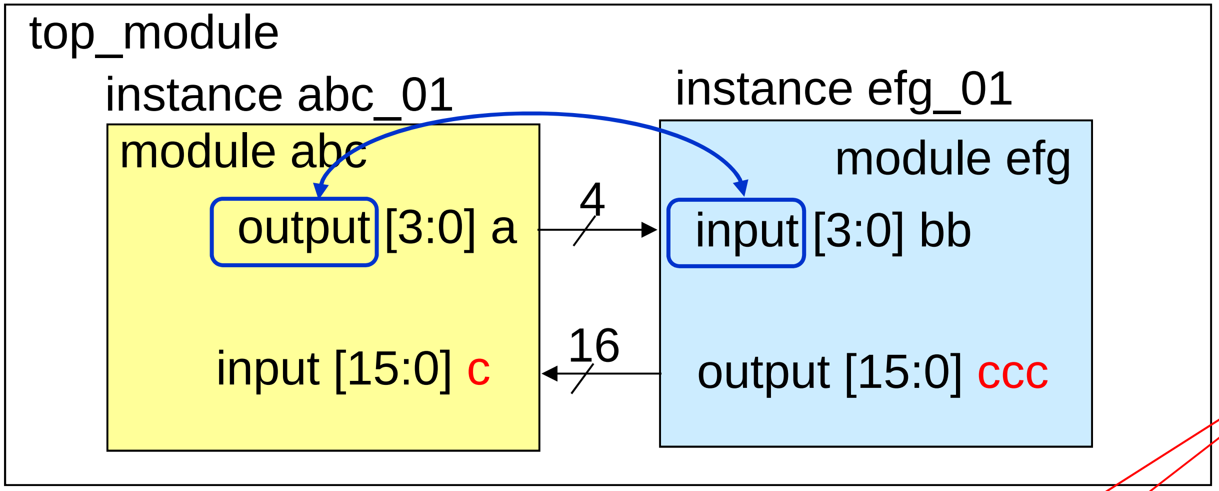
instance name

```
module mod_name ( f, g ) ;
 input f ;
 output g ;

endmodule
```

It is recommended to name instances as modulename_01, modulename_02,,,.

Use connection by name to avoid troubles caused by confusion on list order.

Port connection by name example;



It is recommended to use output port names for module connecting wire signals.

```
module top_module ;
wire [3:0] a ;
wire [15:0] ccc ;
        ⋮
abc abc_01(.c(ccc), .a(a) ) ;
efg efg_01 ( .bb(a), .ccc(ccc) ) ;
        ⋮
endmodule
```

module name

instance name

```
module abc ( c, a ) ;
input [15:0] c ;
output [3:0] a ;
        ⋮
endmodule
```

```
module efg ( bb, ccc ) ;
input [3:0] bb ;
output [15:0] ccc ;
        ⋮
endmodule
```

Q3.6-1: Answer if the following statements are correct or not.

(1) The following port declaration is illegal because bit width in the module port list and port declaration bit width are different.

```
module abc ( aa[15:8] ) ; // upper byte of aa is a port
  input [15:0] aa ;
  wire [15:0] aa ;
```

(2) The following port declaration is illegal because real data type can not be a port.

```
module abc ( aa ) ;
  input aa ;
  real aa ; // aa is real data type
```

(3) a net aa of module instance abc_01 is connected to bb of module efg by the following code.

```
module efg ;
  wire aa, bb ;
       ⋮
  abc abc_01( bb, aa ) ;
```

```
module abc ( aa, bb ) ;
  input aa, bb ;
  wire aa, bb ;
```

Q3.6-1: A sample answer;

(1) The following port declaration is illegal because bit width in the module port list and port declaration bit width are different.

```
module abc ( aa[15:8] ) ; // upper byte of aa is a port
  input [15:0] aa ;
  wire [15:0] aa ;
```

This statement is wrong. A port can be a part-select of a vector declared within the module.

```
module abc ( aa[15:8] ) ;
  input [15:0] aa ;
  wire [15:0] aa ;
```

Port connection by name is not applicable because part select is used.

The range specification must be identical.

No range declaration

```
input   aa ;
wire [15:0] aa ;
```

This is OK.

The declaration on the left is OK, but not recommended. Use the following style.

```
module abc ( u_aa ) ;
  input [7:0] u_aa ;
  wire [7:0] u_aa ;
  wire [15:0] aa ;
        •
        •
        •
assign aa[15:8] = u_aa;
        •
        •
        •
```

Q3.6-1: A sample answer (continued);

(2) The following port declaration is illegal because real data type can not be a port.

```
module abc ( aa ) ;
  input aa ;
  real aa ; // aa is real data type
```

⇦ This statement is correct.
Real data type can not be a port.

(3) a net aa of module instance abc_01 is connected to bb of module efg by the following code.

```
module efg ;
  wire aa, bb ;
          ⋮
  abc abc_01( bb, aa ) ;
```

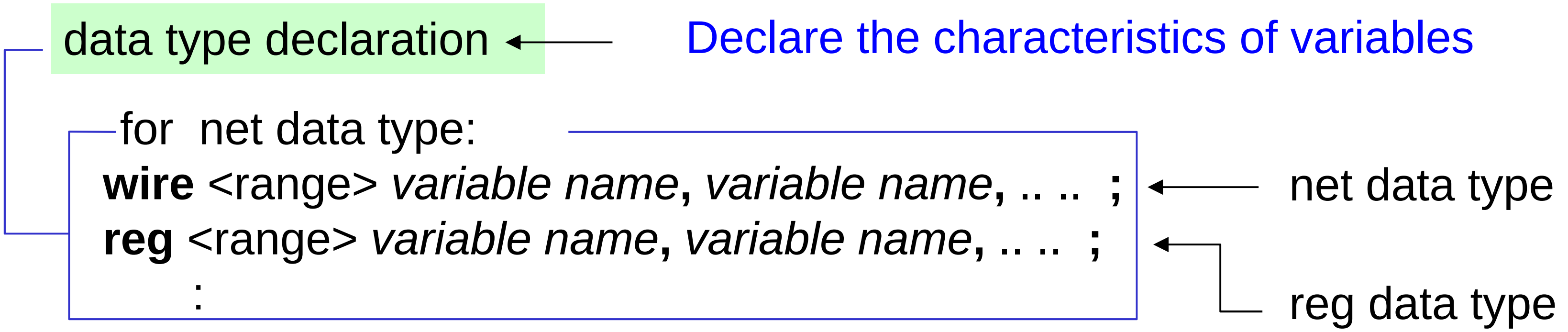
```
module abc ( aa, bb ) ;
  input aa, bb ;
  wire aa, bb ;
```

⇦ This statement is correct.
Using connection by ordered list is not recommended.

3.6.2 Data type declaration

module *module name* **(***port name***,** *port name***, ..) ;**

module_port declaration



data type declaration

Declare the characteristics of variables

for net data type:

wire <range> *variable name***,** *variable name***, ... ;**

reg <range> *variable name***,** *variable name***, ... ;**

:

net data type

reg data type

Restrictions

[1] Range specification must be identical to the size specification in port declaration if the size is specified in port declaration.

[2] LHS of continuous assignment must be declared as nets. LHS in structured procedure must be declared as variables.

Use data type declaration explained in 3.4.

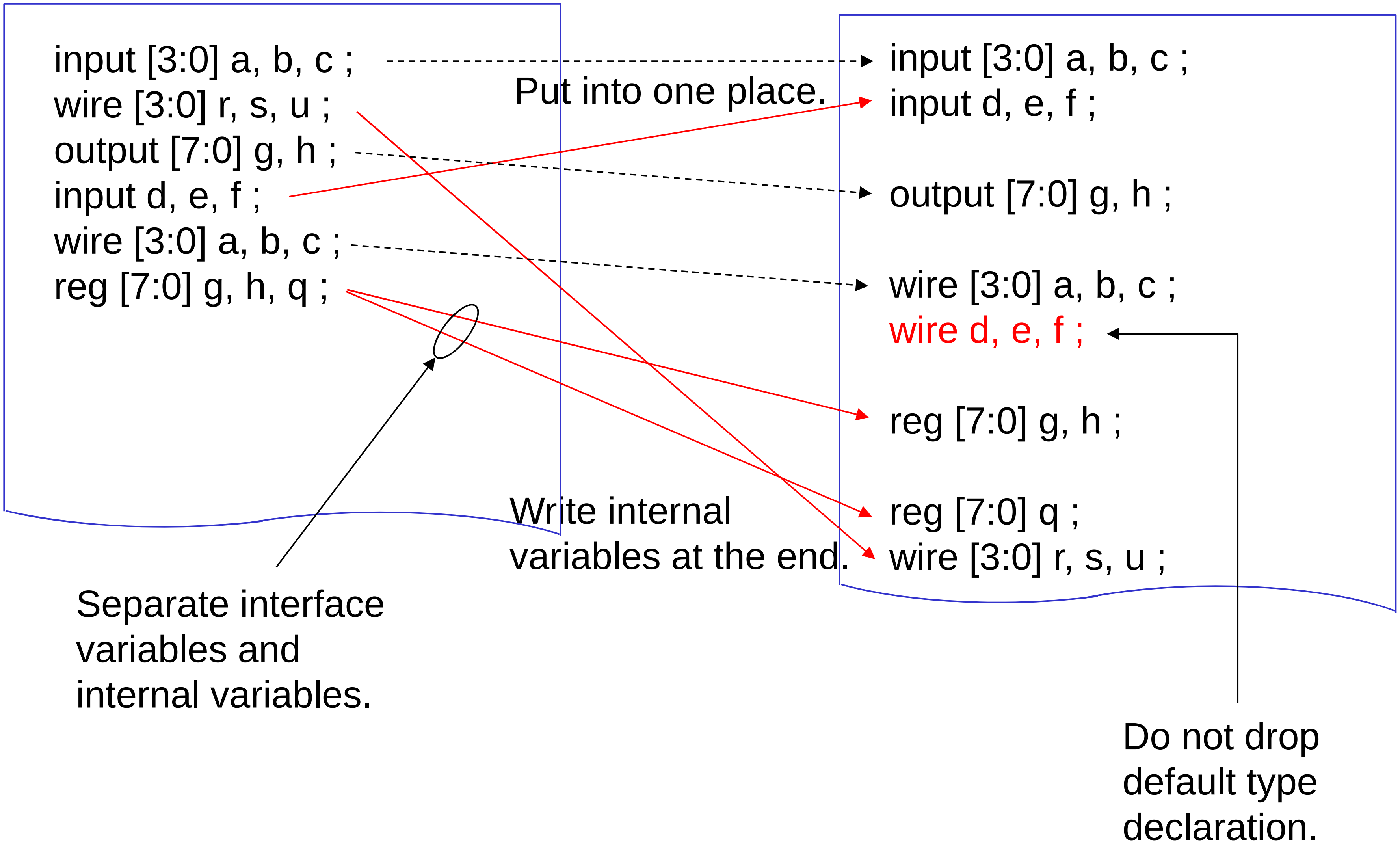
Q3.6-2: Rewrite the following part of RTL code into a better style.

(Do not change variable name or type.)

```
input [3:0] a, b, c ;
wire [3:0] r, s, u ;
output [7:0] g, h ;
input d, e, f ;
wire [3:0] a, b, c ;
reg [7:0] g, h, q ;
```

Always try to write a beautiful code.

Q3.6-2: A sample answer.



```
input [3:0] a, b, c ;
wire [3:0] r, s, u ;
output [7:0] g, h ;
input d, e, f ;
wire [3:0] a, b, c ;
reg [7:0] g, h, q ;
```

```
input [3:0] a, b, c ;
input d, e, f ;

output [7:0] g, h ;

wire [3:0] a, b, c ;
wire d, e, f ;

reg [7:0] g, h ;

reg [7:0] q ;
wire [3:0] r, s, u ;
```

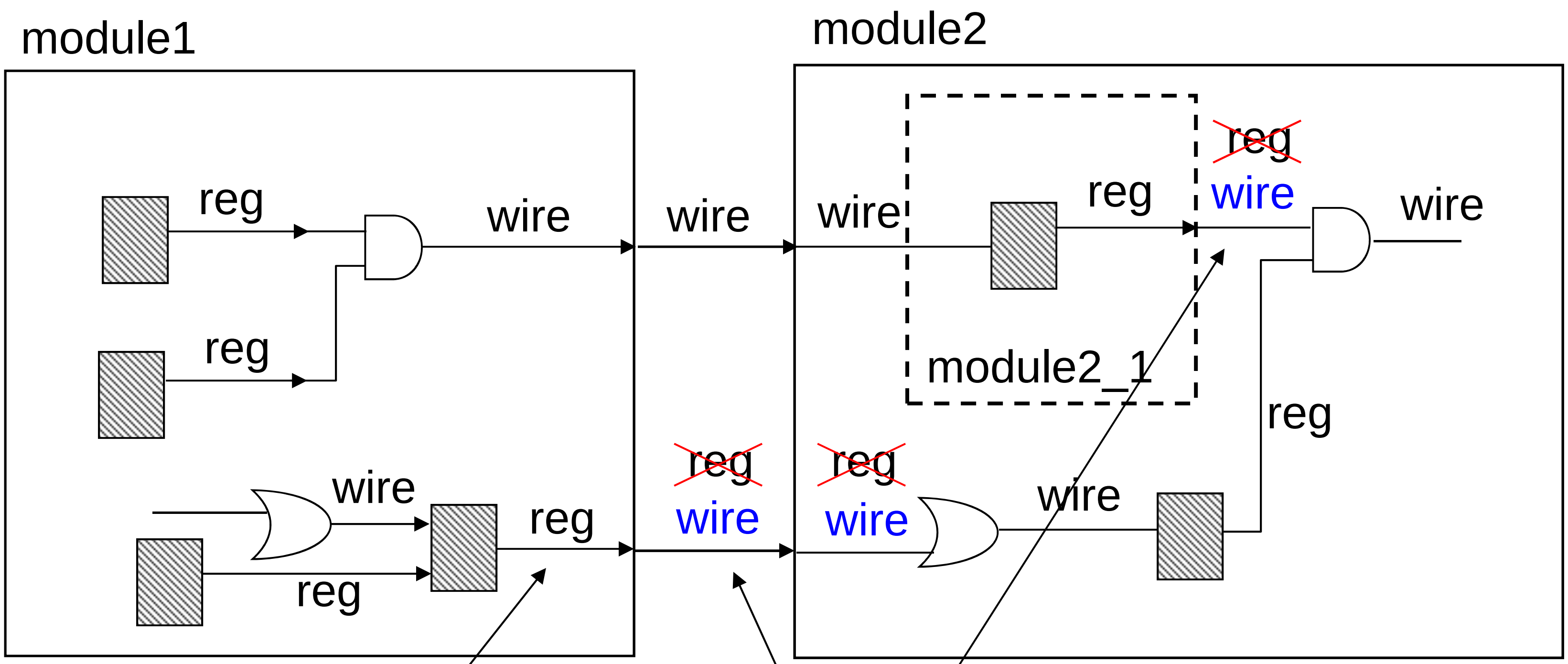
Put into one place.

Write internal variables at the end.

Separate interface variables and internal variables.

Do not drop default type declaration.

Note on variable and net data type.

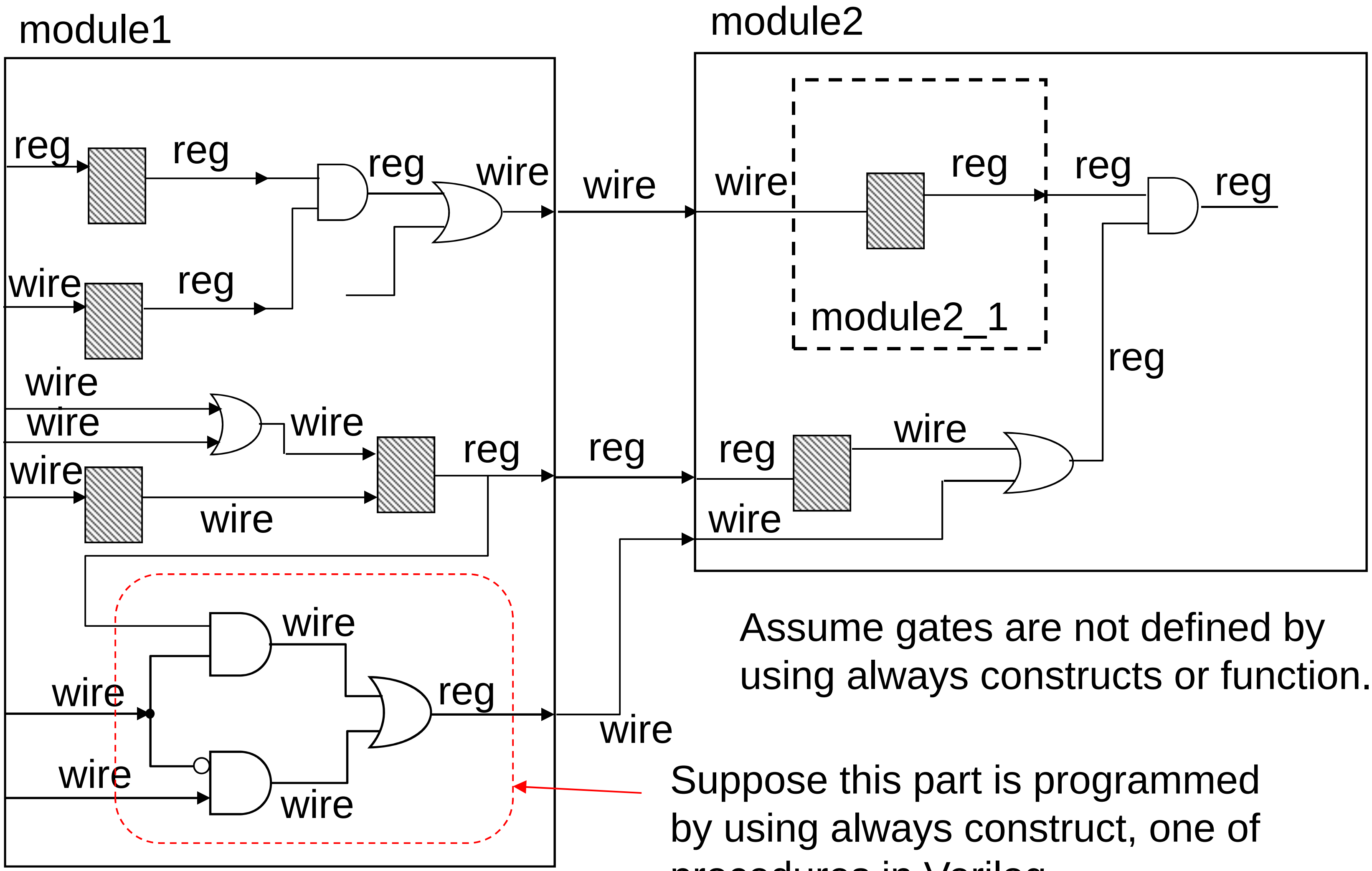


The output of memory element must be defined as *reg*.

They must be defined as *wire* at these points.

▨ : Memory element

Q3.6-3 : Correct the type of the variables shown below.



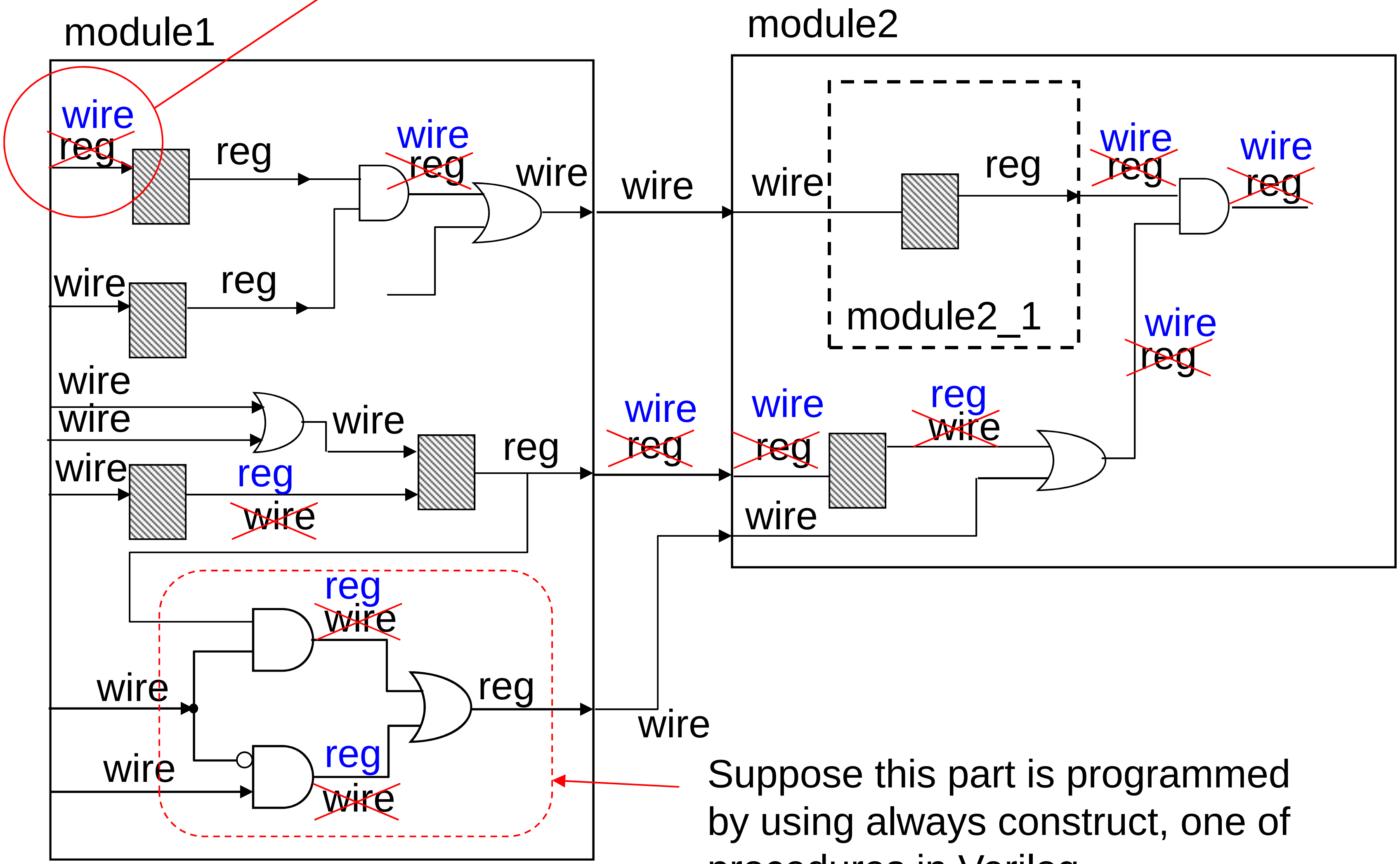
Assume gates are not defined by using always constructs or function.

Suppose this part is programmed by using always construct, one of procedures in Verilog.

▨ : Memory element

Q3.6-3 : A sample answer.

Note: reg data type cannot be declared as an input!!!



Suppose this part is programmed by using always construct, one of procedures in Verilog.

▨ : Memory element

3.6.3 Logic description

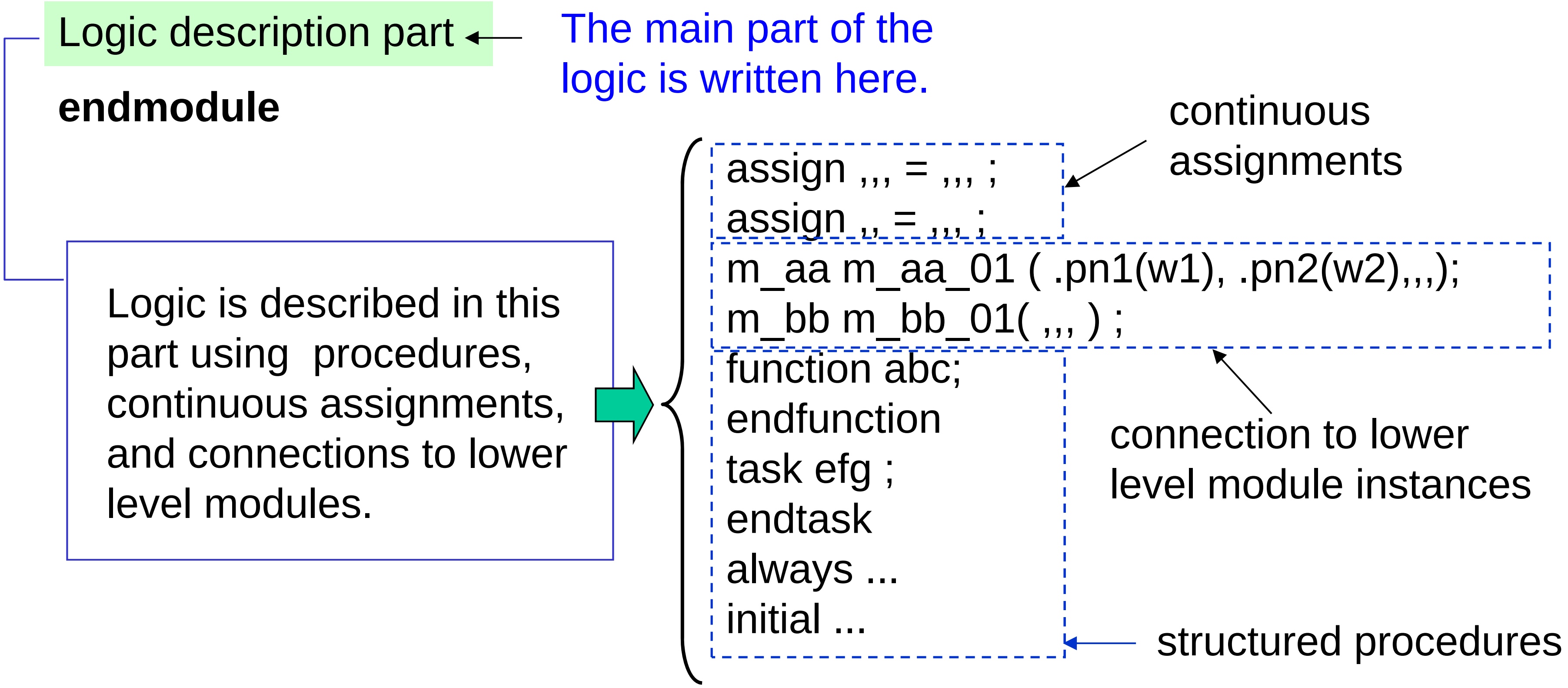
module *module name* **(***port name***,** *port name***, ..) ;**

module_port declaration

data type declaration

Logic description part

endmodule



The main part of the logic is written here.

Logic is described in this part using procedures, continuous assignments, and connections to lower level modules.

assign ,,, = ,,, ;
assign ,, = ,,, ;

continuous assignments

m_aa m_aa_01 (.pn1(w1), .pn2(w2),,,);
m_bb m_bb_01(,,,) ;

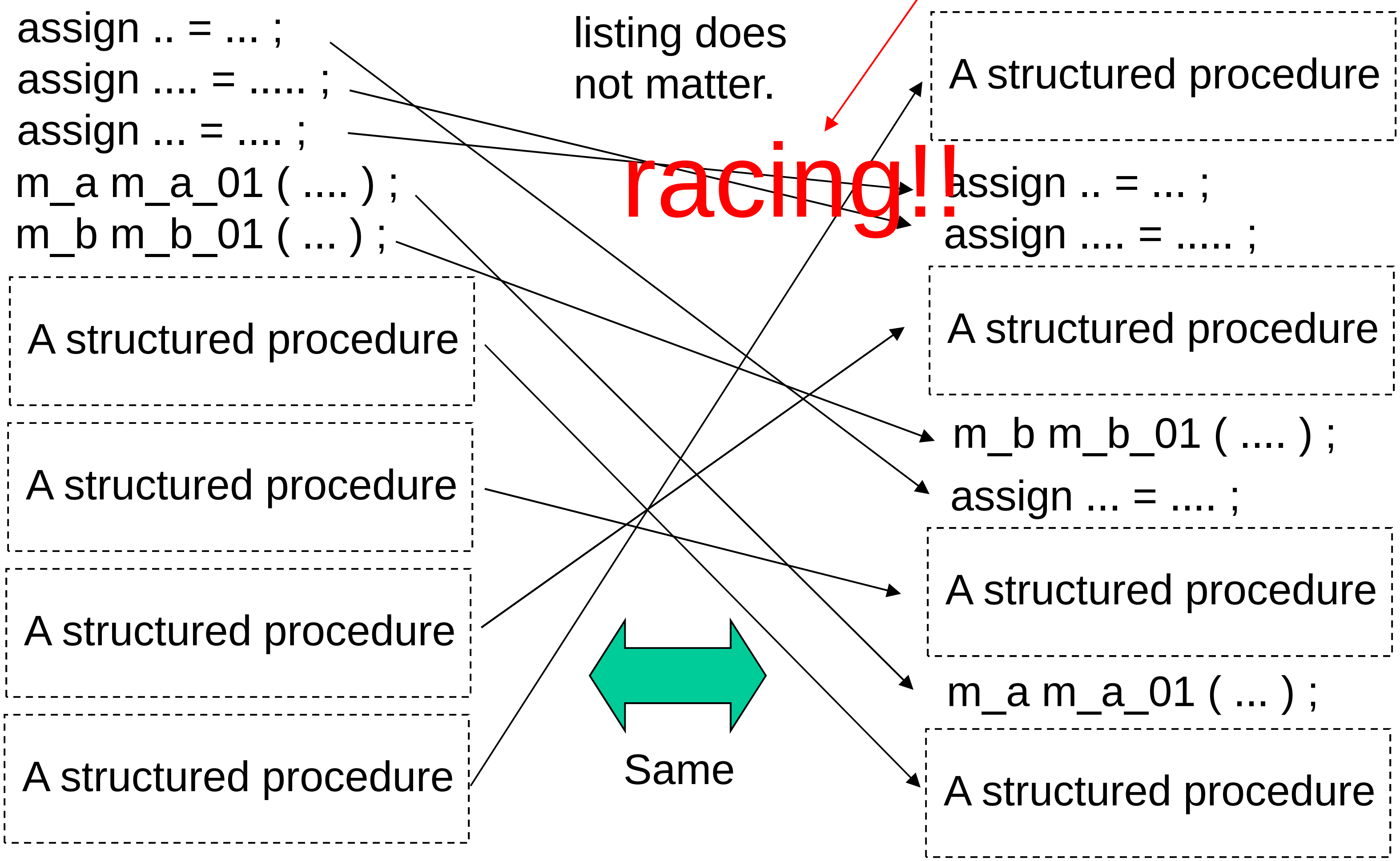
connection to lower level module instances

function abc;
endfunction
task efg ;
endtask
always ...
initial ...

structured procedures

module *module name* **(***port name*, *port name*, **..) ;**
module_port declaration
data type declaration

Two structures, right and left, are the same except for the racing problem.



```
assign .. = ... ;
assign .... = ..... ;
assign ... = .... ;
m_a m_a_01 ( .... ) ;
m_b m_b_01 ( ... ) ;
A structured procedure
A structured procedure
A structured procedure
A structured procedure
```

endmodule

The order of listing does not matter.

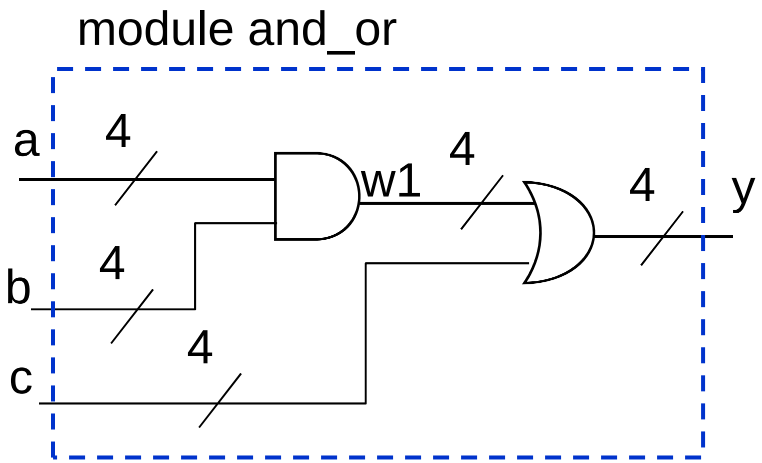
racing!!

Same

```
A structured procedure
assign .. = ... ;
assign .... = ..... ;
A structured procedure
m_b m_b_01 ( .... ) ;
assign ... = .... ;
A structured procedure
m_a m_a_01 ( ... ) ;
A structured procedure
```

Improve readability of the code by putting relevant code blocks near to each other.

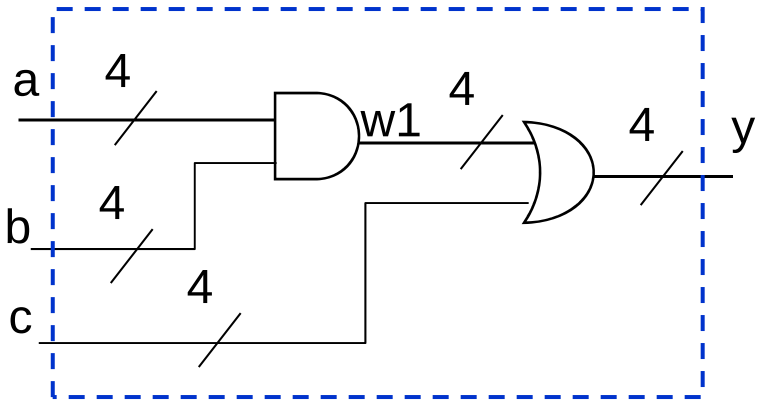
Q3.6-4 : Write a Verilog RTL module for the logic gate shown below by using continuous assignment and nets.



Use & for AND operator and | for OR operator.

Q3.6-4 : A sample answer

module and_or



```
module and_or ( a, b, c, y ) ;
input [3:0] a, b, c ;
output [3:0] y ;
wire [3:0] a, b, c ; // inputs
wire [3:0] y ;  // output
wire [3:0] w1 ; // internal net

  assign w1 = a & b ;
  assign y = w1 | c ;

endmodule
```

Use & for AND operator and | for OR operator.

The part in the dashed box can be written as below:

```
assign y = ( a & b ) | c ;
```

3.6.4 Module summary

Follow the following steps to write Verilog RTL source program of a module.

Step 1. Give a unique name to a module.

```
module module_name ;

endmodule
```

Give a unique meaningful name,
Use 5 to 16 lowercase characters.

Step 2. Find how many inputs and outputs are needed for the module.
And give them unique meaningful names, and then list them up
in a port list in the order of clock, reset, input and output signals.

```
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, ) ;

endmodule
```

List up all the interface signals including
clock and reset in a module port list.

Give a unique meaningful name for each signal,
Use 5 to 16 lowercase characters.

Step 3. Declare port using input/output keywords for all the signals listed up in a module port list. Use range specification for multi-bit signals.

```
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, ) ;
  input clk, rst ;
  input in_a;
  input [7:0] in_b,,,,, ;
  output out_c ;
  output [3:0] out_d,,, ;

endmodule
```

Declare input and output for all the signals listed in a module port list. Use range specification for multi-bit signals.

Step 4. Declare data type for input ports using wire keyword.

```
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, ) ;
 input clk, rst ;
 ,,,,,,,,,
 output [3:0] out_d,,, ;
 wire ckl, rst ;
 wire in_a ;
 wire [7:0] in_b,,,,, ;

endmodule
```

Declare data type of all inputs by using wire keyword. Use the same range specification for multi-bit signals.

Step 5-1. Find if output signals can be described by using continuous assign statement or not. For those output signals which can be described as “assign out_c = some_expression ;”, declare their data type by using wire keyword.

```
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, ) ;
 input clk, rst ;
 ,,,,,,,,,
 output [3:0] out_d,,, ;
 wire ckl, rst ;
 wire in_a ;
 wire [7:0] in_b,,,,, ;
 wire out_c ;
  wire [3:0] out_d,,, ;


endmodule
```

} Declare data type of outputs by using wire keyword if they appear on LHS of continuous assign statements.
Use the same range specification for multi-bit signals.

Step 5-2. For those output signals which can not be defined by continuous assign statements and have to be defined by procedural assign statements, declare their data type by using reg keyword.

→ Use reg keyword if a signal appears on LHS of procedural assign statements.

```
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, ) ;
 input clk, rst ;
 ,,,,,,,,,
 output [3:0] out_d,,, ;
 wire ckl, rst ;
 wire in_a ;
 wire [7:0] in_b,,,,, ;
 wire out_c ;
 wire [3:0] out_d,,, ;
 reg  out_f ;  // non-FF
 reg [15:0] out_g ; // FF
 reg [7:0] out_p ; // non-FF

endmodule
```

Declare data type of outputs by using reg keyword if they appear on LHS of procedural assign statements.
Use the same range specification for multi-bit signals.
Give comments to make it clear that if they are to be mapped into FF or not.

Step 6. Declare internal signals by using wire or reg keyword depending on if they appear on LHS of continuous assign or procedural assign statements.

```
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, ) ;
  input clk, rst ;
  ,,,,,,,,,
  output [3:0] out_d,,, ;
  wire ckl, rst ;
  wire in_a ;
  wire [7:0] in_b,,,,, ;
  wire out_c ;
  wire [3:0] out_d,,, ;
  reg  out_f ;  // non-FF
  reg [15:0] out_g ; // FF
  reg [7:0] out_p ; // non-FF
// internal signals
  wire intnl_sig_a;
  wire [3:0] intnl_sig_b ;
  reg intnl_sig_c ;  // non-FF
  reg [7:0] intnl_sig_d ; // FF

endmodule
```

Declare data type of internal signals. Use wire keyword if they appear on LHS of continuous assign statements, use reg keyword if they appear on LHS of procedural assign statements. Use range specification for multi-bit signals. Give comments for reg data type to make it clear that they are to be mapped into FF or not.

Step 7. Describe logic using continuous assign statements and procedures.

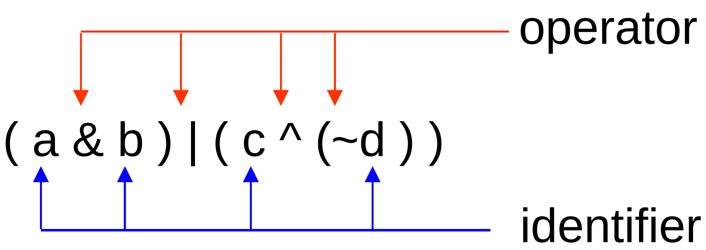
```
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, ) ;
 input clk, rst ;
 ,,,,,,,,,
 output [3:0] out_d,,, ;
 wire ckl, rst ;
 wire in_a ;
 wire [7:0] in_b,,,,, ;
 wire out_c ;
 wire [3:0] out_d,,, ;
 reg  out_f ;  // non-FF
 reg [15:0] out_g ; // FF
 reg [7:0] out_p ; // non-FF
// internal signals
 wire intnl_sig_a;
 wire [3:0] intnl_sig_b ;
 reg intnl_sig_c ;  // non-FF
 reg [7:0] intnl_sig_d ; // FF
// start logic description



endmodule
```

Write logic blocks of the module here using continuous assign statements and procedures and instantiating lower level modules.

3.7. Expressions

In Verilog RTL programming, we can write expressions by using Verilog operators and identifiers as shown below.

Verilog expression example:



(a & b) | (c ^ (~d))

Use ( ) to avoid possible misunderstanding of the precedence of the operations.

3.7.1 Operator

(1) Bitwise operator

syntax

op1 *operator* op2 → for binary operator

operator op1 → for bitwise negation

Bitwise operator performs bitwise manipulation on op1 and op2.
The operator combines a bit in op1, with its corresponding bit in op2, to calculate 1 bit for the result.

Bitwise operator

operator	operation	Example
&	Bitwise and	4'b1100 & 4'b1010 → 4'b1000
|	Bitwise inclusive or	4'b1100 | 4'b1010 → 4'b1110
^	Bitwise exclusive or	4'b1100 ^ 4'b1010 → 4'b0110
^~ or ~^	Bitwise equivalence	4'b1100 ~^ 4'b1010 → 4'b1001
~	Bitwise negation	~4'b1100 → 4'b0011

sizing and type rule of bitwise binary operator

	sizing	type
prior to the operation	max (L(op1), L(op2))	If both op1 and op2 are signed, sign bit extended, otherwise 0 filled.
result	⇧	signed if both op1 and op2 are signed, otherwise unsigned.

In the context of y = ~a ; a is sized to L(LHS) and bitwise negate operation applied after sizing.

Restrictions

[1] real expression can not be an operand of bitwise operator.

[2] Bitwise operator must not appear in LHS.

The calculation of each bit follows the rules shown in the table below.

Bit-wise binary and

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

Bit-wise binary or

|	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

Bit-wise binary xor

^	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

Bit-wise binary nxor

~^	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

Bit-wise unary negation

~	
0	1
1	0
x	x
z	x

Q3.7-1: Evaluate the following expressions.

(1) 4'b1101 & 8'sb1000_1001

(2) 4'sb1101 & 8'sb1110_1001

(3) 4'sb1101 ~^ 8'b1111_1001

(3) 4'sb1101 & 4'sb1x0z

(4) 8‘sb0101 | 8‘sb1xxz

In actual design work, make sure both operands have the same size.

Q3.7-1: A sample answer;

(1) 4'b1101 & 8'sb1000_1001

→ 8'b0000_1101 & 8'b1000_1001 → 8'b0000_1001

Sign bit not extended, result unsigned.

(2) 4'sb1101 & 8'sb1110_1001

→ 8'sb1111_1101 | 8'sb1110_1001 → 8'sb1110_1001

Sign bit (1) extended, result signed.

(3) 4'sb1101 ~^ 8'b1111_1001

→ 8'b0000_1101 ~^ 8'b1111_1001 → 8'b0000_1011

Sign bit not extended, result unsigned.

(3) 4'sb1101 & 4'sb1x0z

→ 4'sb1101 & 4'sb1x0z → 4'sb1x0x

No sizing, result signed.

(4) 8‘sb0100 | 8‘sb1xxz

→ 8‘sb0000_0100 | 8‘sb0000_1xxz → 8‘sb0000_11xx

No sizing, result signed.

(2) Binary Arithmetic operator

syntax

op1 *operator* op2

Binary arithmetic operator performs arithmetic operation of op1 and op2.

Binary arithmetic operator

operator	operation	Example
+	addition	4'b0100 + 4'b0010 → 4'b0110
-	subtraction	4'b0100 - 4'b0010 → 4'b0010
*****	multiplication	4'b0100 * 4'b0010 → 4'b1000
/	division	4'b0100 / 4'b0010 → 4'b0010
%	modulus	4'b0101 % 4'b0010 → 4'b0001
******	power	8'h05 ** 4'h2 → 8'h19

sizing and type rule of arithmetic operator except power operator

	sizing	type
prior to the operation	max (L(op1), L(op2))	If both op1 and op2 are signed, sign bit extended, otherwise 0 filled.
result	⇧	signed if both op1 and op2 are signed, otherwise unsigned.

For power operator, result size is L(op1) and type is signed if op1 is signed. op2 in power operator is self determined.

Restrictions

[1] % operator is not allowed in real expression.

[2] % and ** operators are not synthesizable.

[3] Arithmetic operator will not report any overflow error.

[4] In simulation, if any operand bit value is x or z, then the entire result value shall be x.

[5] Arithmetic operator must not appear in LHS.

Note:

Use arithmetic operator to get better logic circuit block.
Synthesis tool can create a better circuit than a human.

Make your arithmetic operation code in one sentence to get better synthesis result.

Example :

~~assign a = b*2 + c ;~~
~~assign d = e*3 - f ;~~
~~assign g = a - d ;~~

→ assign g = (b*2 + c) – (e*3 – f) ;

This code may get better synthesis result compared to that of shown on the left.

Note:

To catch the carry bit, make LHS 1 bit longer than RHS.

```
wire [7:0] a, b ;
wire [8:0] c ;
assign c = a + b ;
```

When using arithmetic operator, sign bit is extended for signed variables or nets. However, if part-select is used, sign bit will not be extended.

```
wire signed [7:0] a = 5 ;
wire signed [3:0] b = -7 ;
wire signed [8:0] c ;

assign c = a[7:0] + b[3:0] ;
```

c will be assigned the value 14. not the correct result –2.

```
   0_0000_0101
+  0_0000_1001
--------------
   0_0000_1110
```

```
wire signed [7:0] a = 5 ;
wire signed [3:0] b = -7 ;
wire signed [8:0] c ;

assign c = a + b ;
```

c will be assigned the value –2 which is correct.

```
   0_0000_0101
+  1_1111_1001
--------------
   1_1111_1110
```

Q3.7-2: Answer if the following sentences are correct or not.

(1) The bit pattern of y will be 0010 after the assignment is done.

```
reg signed [3:0] y ;
y = 4'b0110 + 3'sb100;
```

(2) The bit pattern of y will be 111011 after the assignment is done.

```
reg [5:0] y ;
y = 4'sb1110 - 3 ;
```

(3) Overflow can be known by checking if positive + positive becomes negative, or negative + negative becomes positive. We can use the following logic to check if overflow occurs in a+b.

```
reg signed [3:0] a, b ;
reg signed [3:0] c ;
reg overflow_flg ;
overflow_flg = ( ( (a >0) & (b>0) & ( (a+b)  <  0 ) ) |
                       ( (a <0) & (b<0) & ( (a+b) >= 0) ) ) ? 1: 0 ;
```

Q3.7-2: A sample answer,

(1) The bit pattern of y will be 0010 after the assignment is done.

```
reg signed [3:0] y ;
y = 4'b0110 + 3'sb100;
```

This is wrong: the first operand is unsigned, therefore, both operands are treated as unsigned. No sign-bit extension is done.

```
y = 4'b0110 + 3'sb100
  = 4'b0110 + 4'b0100
  = 4'b1010
```

No sign-bit extension

(2) The bit pattern of y will be 111011 after the assignment is done.

```
reg [5:0] y ;
y = 4'sb1110 - 3 ;
```

This is correct: the first operand is signed and the second operand is integer 3, meaning 32-bit signed 00000,,,,0011. Both operands are signed. They are sized to 6-bit.

y is only 6-bit, therefore sizing to 32-bit is not needed unless such truncation does not affect the result..

Sign bit extended.

```
4'sb1110  ──────→      111110
            sizing
    - 3   ──────→  + ) 111101
                      1111011
```

Q3.7-2: sample answer

(3) Overflow can be known by checking if positive + positive becomes negative, or negative + negative becomes positive. We can use the following logic to check if overflow occurs in a+b.

```
reg signed [3:0] a, b ;
reg signed [3:0] c ;
reg overflow_flg ;
overflow_flg = ( ( (a >0) & (b>0) & ( (a+b)  <  0) ) |
                 ( (a <0) & (b<0) & ( (a+b) >= 0) ) ) ? 1: 0 ;
```

This is wrong: Because 0 means integer, 32-bit signed, a+b is executed after a and b are sized to 32-bit. Adding 4-bit data will never cause overflow of 32-bit data. Therefore, overflow can not be detected by the above code. We have to use the following code.

```
reg signed [3:0] a, b ;
reg signed [3:0] c ;
reg signed [3:0] tmp;
reg overflow_flg ;
tmp = a + b ;  // add is done in 4-bit
overflow_flg = ( ( (a >0) & (b>0) & ( tmp  <  0) ) |
                 ( (a <0) & (b<0) & ( tmp >= 0) ) ) ? 1: 0 ;
```

(3) Unary Arithmetic operator

syntax

operator op1

Unary arithmetic operator, unary minus, changes the sign of the operand.

Unary arithmetic operator

operator	operation	Example
+	unary plus	+ 4'b0100 → 4'b0100
-	unary minus	- 4'b0100 → 4'b1100

Unary arithmetic operator has precedence over binary arithmetic operators.

sizing and type rule

	sizing	type
prior to the operation	L(op1)	—
result	L(op1)	signed if op1 is signed, otherwise unsigned.

Note that unary operator does not change the type of the operand.
If op1 is unsigned, "- op1" is still unsigned.

Q3.7-3: Answer if the following explanation is correct or not.

(1) The value of -a >= 5 is 1'b0 when a is declared as 4 bit unsigned variable and its bit pattern is 0101.

To answer this question, you need the knowledge of relational operator, please visit this question after you studied relational operators.

Q3.7-3: A sample answer.

(1) The value of -a > 5 is 1'b0 when a is declared as 4 bit unsigned variable and its bit pattern is 0101.

This is wrong; The value is 1'b1. a is declared as unsigned, so -a is still treated as unsigned. Therefore, the comparison is done as unsigned values.
First, a’s bit width is extended to 32 because “5” means 32-bit integer. And the unary minus operator is applied, the result is 32-bit unsigned 11111---111011, a very big positive number larger than 5.

(4) Relational operator

syntax

op1 *operator* op2

Relational operator performs comparison between op1 and op2.

Relational operator

operator	operation	example
>	greater	4'b1010 > 2'b11 → 1'b1
>=	greater or equal	4'sb1010 >= 4'sb0011 → 1'b0
<	smaller	4'sb1010 < 4'sb0000 → 1'b1
<=	smaller or equal	4'b1010 <= 4'b0100 → 1'b0

sizing and type rule

	sizing	type
prior to the operation	max (L(op1), L(op2))	If both op1 and op2 are signed, sign bit extended, otherwise 0 filled.
result	1 bit	unsigned

Restrictions

[1] If, due to x or z bits in the operands, the relation is ambiguous, then the result shall be a 1-bit unknown value (x).

[2] Only when both operands are signed, the expression shall be interpreted as a comparison between signed values.

[3] If either operand is a real operand, then the other operand shall be converted to an equivalent real value and the expression shall be interpreted as a comparison between real values.

[4] Relational operator must not appear in LHS.

Note:

4’b0101 > 4’b001x ⇦ This expression is evaluated to be x even if 4’b0101 is larger than 4’b0010 and 4’b0011.

However, in the actual circuit, the result shall be 1, comparing 0101 and (0010 or 0011) .

Also note that >,>=,<,<= are different from ==, !=.

Relational operator does not compare operands bit for bit. It compares them as numerical values as a whole, but equality (inequality) operator compares bit for bit.

The value of an expression 4’b0011 > 4’bx110 is 1’bx, but
the value of an expression 4’b0011 == 4’bx110 is 1’b0 because
bit for bit comparison results are x,0,1, and 0 for bit 3,2,1,and 0 respectively.
Therefore the total result is 1’b0. op1 == op2 can be 1’b1 only when all bits are exactly the same and no bit is x nor z.

Q3.7-4: Evaluate the following expressions.

(1) 8'hfc >= 16'h000f

(2) 8'shfc >= 16'sh000f

(3) 8'hff < (8'h00 -1)

Q3.7-4: A sample answer ;

(1) 8'hfc >= 16'h000f

→ 16'h00fc >= 16'h000f → 252 >= 15 → 1'b1

Sign bit not extended, compared as unsigned values.

(2) 8'shfc >= 16'sh000f

→ 16'shfffc >= 16'sh000f → -4 >= 15 → 1'b0

Sign bit extended, compared as signed values.

(3) 8'hff < (8'h00 -1)

→ 8'hff < (32'h0000_0000 - 32'sh0000_0001)

→ 8'hff < 32'hffff_ffff

→ 32'h0000_00ff < 32'hffff_ffff

→ 255 < very large positive value → 1'b1

Sign bit not extended, compared as unsigned values.
Note that "1" is integer, means 32-bit data.

(5) Equality operator

syntax

op1 *operator* op2

Equality operator compares op1 and op2 bit for bit.

Equality operator

operator	operation	example
==	logical equality	4'b1010 == 4'b1010 → 1'b1
!=	logical inequality	4'sb1010 != 4'sb101x → 1'bx
===	case equality	4'b100z === 4'b1x0z → 1'b0
!==	case inequality	4'b101x !== 4'b101z → 1'b1

Note that case equality/inequality operator can compare values having x and z bit.

sizing and type rule

	sizing	type
prior to the operation	max (L(op1), L(op2))	If both op1 and op2 are signed, sign bit extended, otherwise 0 filled.
result	1 bit	unsigned

Restrictions

[1] If, due to x or z bits in the operands, the relation is ambiguous, then the result shall be a 1-bit unknown value (x) for logical equality/inequality operator.

[2] If either operand is a real operand, then the other operand shall be converted to an equivalent real value and the expression shall be interpreted as a comparison between real values.

[3] Case equality/inequality operator is not synthesizable.

[4] Case equality/inequality operator is not applicable for real data type.

[5] Equality operator must not appear in LHS.

Q3.7-5: Evaluate the following expressions.

(1) 8'hfc == 16'h00fx

(2) 4'sb1010 != 6'sb11_1010

(3) 4'b10xz == 4'b10xz

(4) 4'b10xz === 4'b10xz

(5) 4'b10x0 == 4'b00x0

Q3.7-5: A sample answer,

(1) 8'hfc == 16'h00fx

→ 16'h00fc == 16'h00fx

→ 1'bx

Sign bit not extended, result is unknown because of x.

(2) 4'sb1010 != 6'sb11_1010

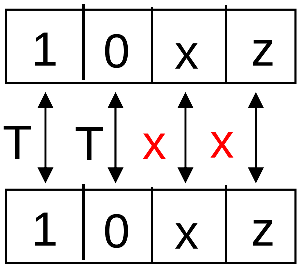
→ 6'sb11_1010 != 6'sb11_1010

→ 1'b0

Sign bit extended, result is false because they are equal.

(3) 4'b10xz == 4'b10xz

→ 1'bx



Result is unknown because of x and z. (x win)

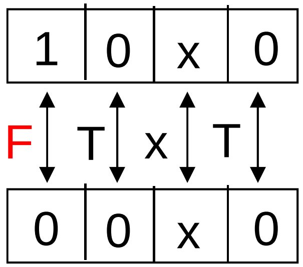
(4) 4'b10xz === 4'b10xz

→ 1'b1

Result is true, two operands are identical including x and z bit.

(5) 4'b10x0 == 4'b00x0

→ 1'b0



Result is false, because bit 3 is different (false win.).

(6) Logical operator

syntax

op1 *operator* op2

Logical operators are logical connective of op1 and op2.

Logical operator

operator	operation	example
&&	Logical and	4'b001x && 4'bx100 → 1'b1
||	Logical or	4'sb0000 || 4'sb1010 → 1'b1
!	Logical negation	! 4'b100z → 1'b0

An operand is thought to have the value shown below.

operand’s bit pattern	value	example
There is a bit whose value is 1.	1	01x0
All bits are 0.	0	0000
Neither of the above two cases.	x	0x0z

sizing and type rule

	sizing	type	note
prior to the operation	none *1	—	An operand is evaluated to be true if any one or more bits are 1, false if all bits are 0, otherwise x.
result	1 bit	unsigned	

*1: operands are self-determined.

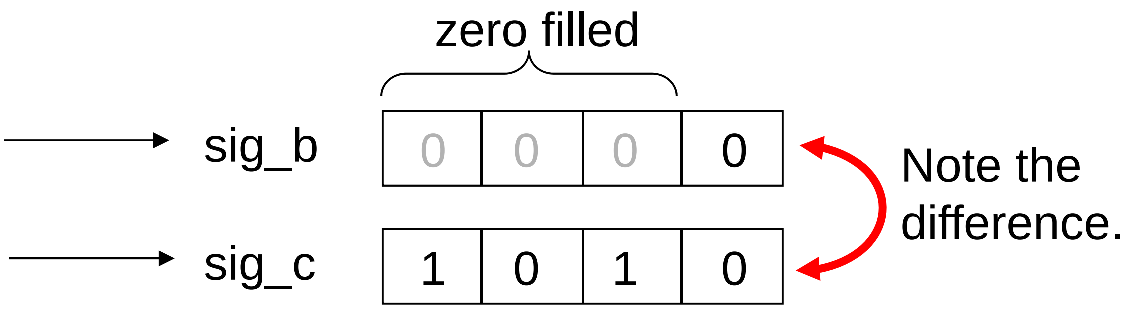
Restrictions

[1] Logical negation of x is still x.

[2] Logical operator must not appear in LHS.

Note on logical negation

```
wire [3:0] sig_a, sig_b, sig_c ;
assign sig_a = 4'b0101 ;

assign sig_b = ! sig_a ;
assign sig_c = ~sig_a ;
```



zero filled

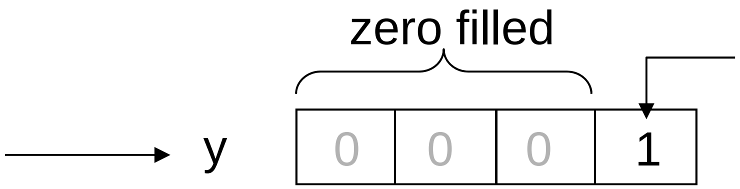
sig_b | 0 | 0 | 0 | 0

sig_c | 1 | 0 | 1 | 0

Note the difference.

Note that;

```
wire signed [3:0] y ;
assign  y = !(4'b0000) ;
```

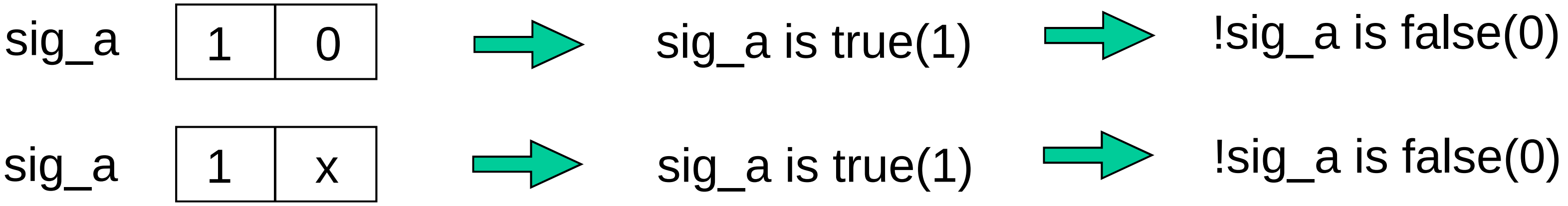


zero filled

y | 0 | 0 | 0 | 1

!(4'b0000) is an unsigned scalar (always positive).

If sig_a is 2’b1x, sig_a is evaluated to be true and !sig_a is evaluated to be false.



sig_a | 1 | 0 → sig_a is true(1) → !sig_a is false(0)

sig_a | 1 | x → sig_a is true(1) → !sig_a is false(0)

Q3.7-6: Evaluate the following expressions.

(1) 8'b0010_0xx0 && 4'b00x0

(2) 8'b0000_0000 && 4'b10zz

(3) 8'b0010_0xx0 || 4'b00x0

Q3.7-6: A sample answer;

(1) $$\underbrace{\text{8'b0010\_0xx0}}_{1}\ \&\&\ \underbrace{\text{4'b00x0}}_{x} \longrightarrow \text{1'bx}$$

(2) $$\underbrace{\text{8'b0000\_0000}}_{0}\ \&\&\ \underbrace{\text{4'b10zz}}_{1} \longrightarrow \text{1'b0}$$

(3) $$\underbrace{\text{8'b0010\_0xx0}}_{1}\ \ ||\ \ \underbrace{\text{4'b00x0}}_{x} \longrightarrow \text{1'b1}$$

(7) Shift operator

syntax

op1 *operator* op2

Shift operator execute shift operation on op1, the shift count is specified by op2.

shift operator

operator	operation	example
>>	Logical shift right	4'sb1010 >> 2 → 4'sb0010
>>>	Arithmetic shift right	4'sb1010 >>> 2 → 4'sb1110 4'b1010 >>> 2 → 4'b0010
<<	Logical shift left	4'sb1010 << 2 → 4'sb1000
<<<	Arithmetic shift left	4'sb1010 <<< 2 → 4'sb1000

Arithmetic shift is introduced in Verilog2001.

sizing and type rule

	sizing	type
prior to the operation	none	op2 is always treated as unsigned.
result	L(op1)	Signed if op1 is signed, otherwise unsigned.

Restrictions

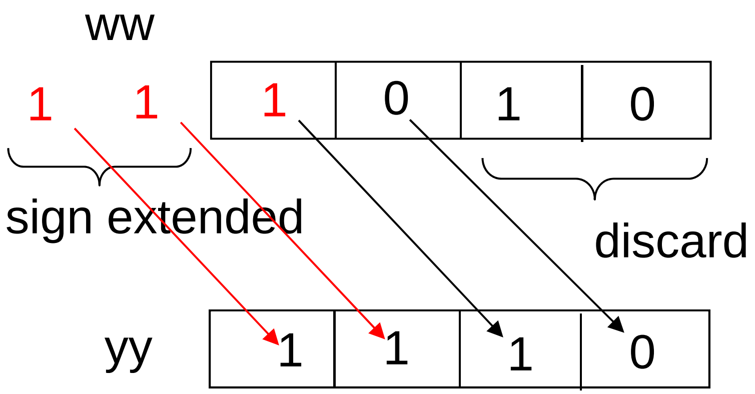
[1] Shift operator is not allowed for real expression.

[2] If op2 has x or z, the result is unknown.

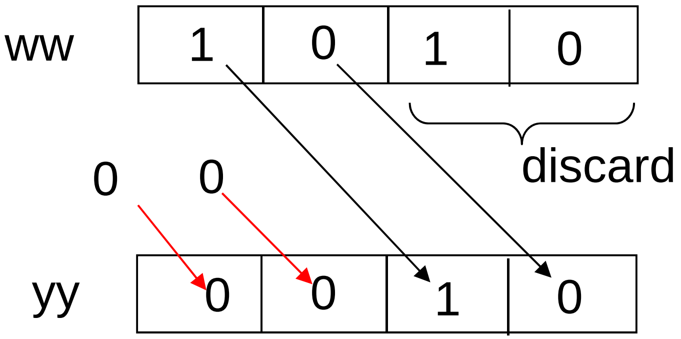
[3] Shift operator must not appear in LHS.

arithmetic shift

```
reg signed [3:0] ww ;
wire [3:0] yy ;
assign yy = ww >>> 2 ;
```

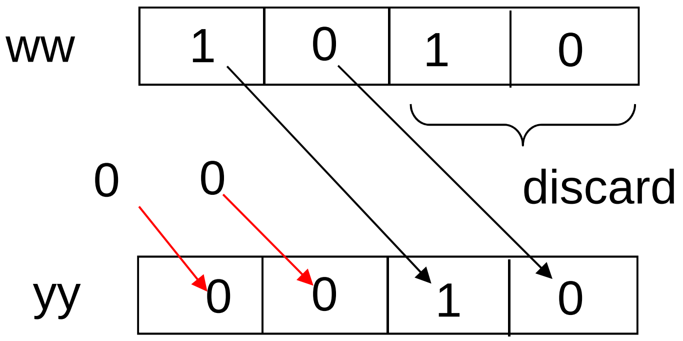


```
reg [3:0] ww ;
wire [3:0] yy ;
assign yy = ww >>> 2 ;
```



logical shift

```
reg signed [3:0] ww ;
wire signed [3:0] yy ;

assign yy = ww >> 2 ;
```



Q3.7-7: Evaluate the following expressions.

(1) 64'shff00_ffff_1100_0ff0 >>> ix ; where ix declared as "wire signed [3:0] ix " and is supposed to be 4'sb1001.

(2) 8'sbx001_1010 >>> ix ; where ix declared as "wire signed [3:0] ix " and is supposed to be 4'sb0011.

Q3.7-7: A sample answer;

(1) 64'shff00_ffff_1100_0ff0 >>> ix ; where ix declared as "wire signed [3:0] ix " and is supposed to be 4'sb1001.

ix is always treated as unsigned, therefore shift count is 9, not -7.
The left operand is signed, the sign bit is filled in for arithmetic shift right.

→ result: 64'shffff_807f_ff88_8007

(2) 8'sbx001_1010 >>> ix ; where ix declared as "wire signed [3:0] ix " and is supposed to be 4'sb0011.

The shift count is positive 3.
The left operand is signed, the sign bit (x) is filled in for arithmetic shift right.

→ result: 8'sbxxxx_0011

(8) Unary reduction operator

syntax

operator op1

Unary reduction operator performs a bitwise operation on op1 to produce a single-bit result.

Unary reduction operator

operator	operation	example
&	Reduction and	& a[3:0] → a[3] & a[2] & a[1] & a[0]
~&	Reduction nand	~& a[3:0] → ~(& a[3:0])
|	Reduction or	| a[3:0] → a[3] | a[2] | a[1] | a[0]
~|	Reduction nor	~| a[3:0] → ~(| a[3:0])
^	Reduction xor	^ a[3:0] → a[3] ^ a[2] ^ a[1] ^ a[0]
~^ or ^~	Reduction xnor	~^ a[3:0] → ~(^ a[3:0])

It is not recommended to use these operators because only Verilog supports this.

sizing and type rule

	sizing	type
prior to the operation	none *1	—
result	1 bit	unsigned

*1: the operand is self-determined.

Restrictions

[1] Unary reduction operator is not allowed for real expression.

[2] Unary reduction operator must not appear in LHS.

Note:

This operator may be used as below;

```
wire    cache_hit = |{tag_cmp[31:0]};
```

Q3.7-8: Evaluate the following expressions.

(1) & (64'sb1111)

(2) | (4'b10xz)

(3) ^ (4'b1011)

(4) ~^ (4'b0110)

(5) & (4'sb11xx)

Q3.7-8: A sample answer.

(1) & (64'sb1111)

$$\underbrace{0 \ \& \ 0 \ \& \ 0 \ \& \ ,,,,,,,,,0}_{\text{60 zeros}} \ \& \ 1 \ \& \ 1 \ \& \ 1 \ \& \ 1 = 0$$

(2) | (4'b10xz)

1 | 0 | x | z = 1

(3) ^ (4'b1011)

1 ^ 0 ^ 1 ^ 1 = 1

(4) ~^ (4'b0110)

~(0 ^ 1 ^ 1 ^ 0) = 1

(5) & (4'sb11xx)

1 & 1 & x & x = x

(9) Conditional operator (Ternary operator)

syntax

```
op1 ? op2 : op3
```

Conditional operator evaluate op1 and if it is true, then op2 is evaluated and used as the result, if it is false then op3 is evaluated and used as the result.

Conditional operator

operator	operation	example
? :	Conditional	flg ? 4'b1010 : 4'b1100 ; → 4'b1010 if flg is true 4'b1100 if flg is false 4'b1xx0 if flg is unknown

sizing and type rule

	sizing		type
prior to the operation	op1	no sizing *1	——
	op2, op3	max (L(op2), L(op3))	only if both op2 and op3 are signed, sign bit extended, otherwise 0 filled.
result	max (L(op2), L(op3))		signed if both op2 and op3 are signed, otherwise unsigned.

*1: the operand is self-determined.

Restrictions

[1] If op2 or op3 is real, and op1 is ambiguous, then the result shall be evaluated to be 0.

[2] Conditional operator must not appear in LHS.

If op1 evaluates to an ambiguous value (x or z), then both op2 and op3 shall be evaluated and their results shall be combined, bit by bit, using the following table to calculate final result.

?:	0	1	x	z
0	0	x	x	x
1	x	1	x	x
x	x	x	x	x
z	x	x	x	x

If both bits match, then use the matched value. Otherwise, use x as a result.

It is recommended to use conditional operator instead of if else statement.
However, do not nest conditional operator because it makes the code difficult to read.

Q3.7-9: Evaluate the following expressions.

(1) (a > b) ? 4'b1010 : 6'b111000, where a is 4-bit unsigned 1010 and b is 4-bit unsigned 1100.

(2) (a > b) ? 4'sb1010 : 6'b111000, where a is 4-bit unsigned 101x and b is 4-bit unsigned 1100.

(3) (a < b) ? 4'sb1010 : 6'sb111000, where a is 4-bit signed 1010 and b is 4-bit signed 1100.

(4) (^a) ? 4‘b1010 : 6‘sb111000, where a is 3-bit signed 111.

Q3.7-9: A sample answer.

(1) (a > b) ? 4'b1010 : 6'b111000, where a is 4-bit unsigned 1010 and b is 4-bit unsigned 1100.

The value is 6'b111000 because1010, decimal 10, is not larger than 1100, decimal12.

(2) (a > b) ? 4'sb1010 : 6'b111000, where a is 4-bit unsigned 101x and b is 4-bit unsigned 1100.

The value is 6'bxx10x0 because the result of the comparison is 1'bx. Sign bit of 4'sb1010 is not extended, because op3 is unsigned.

(3) (a < b) ? 4'sb1010 : 6'sb111000, where a is 4-bit signed 1010 and b is 4-bit signed 1100.

The value is 6'sb111010 because 1010, decimal -6, is smaller than 1100, decimal -4. Sign bit of 4'sb1010 is extended, because both op2 and op3 are signed.

(4) (^a) ? 4‘b1010 : 6‘sb111000, where a is 3-bit signed 111.

The value is 6'b001010 because ^a is 1'b1 and op2 is unsigned. Sign bit of op2 (4'b1010) is not extended

(10) Concatenation operator

syntax

```
{ op1, p2, op3,  ..... }
```

Concatenation operator joins bits together from one or more operands.

Concatenate operator

operator	operation	example
{ }	Concatenation	{ a[3:0], b[1:0] } → a[3] a[2] a[1] a[0] b[1] b[0] { 4{a[1:0]} } → a[1] a[0] a[1] a[0] a[1] a[0] a[1] a[0]

Replication number

sizing and type rule

	sizing	type
prior to the operation	none *1	—
result	L(op1) + L(op2) + L(op3) + ,,	unsigned

*1: Operands are self-determined.

Restrictions

[1] Unsized constant numbers shall not be allowed in concatenations.

[2] Replication number must be non-negative constant. It must not have x or z.

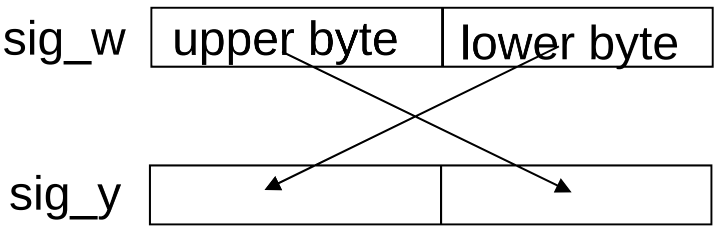
[3] Zero replication is not allowed if other non-zero replication appears in that concatenation.

[4] Real data type is not allowed in concatenation.

{ {0{a[1:0] }}, b[3:0] } is OK, but { 0 {a[1:0] } } used alone is illegal because in { } only 0 replication appears.

Byte swap;

```
wire [15:0] sig_y, sig_w ;

assign sig_y = { sig_w[7:0] , sig_w[15:8] } ;
```



⇕

```
assign { sig_y[7:0] , sig_y[15:8] } = sig_w[15:0] ;
```

Concatenate operator can appear in LHS. ← *Useful !!*

Bit rotation :

```
assign next_sig_w[15:0] = { sig_w[0] , sig_w[15:1] } ;   ←  shift rotate right
assign next_sig_w[15:0] = { sig_w[14:0] , sig_w[15] } ;  ←  shift rotate left
```

Bit size flexibility

```
parameter B_WD = 16 ;
reg [B_WD-1:0] sig_w ;

     sig_w = { B_WD {1'b0} } ;               ←    set 16'b0 to sig_w

     sig_w = { {(B_WD-1) {1'b0} } , 1'b1 } ;       ←    set 16'b1 to sig_w
```

Q3.7-10: Find the bit pattern of the following expressions.

(1) { {1'b1}, {8{1'b0}}, {1'b1}}

(2) {{3{2'b10}}, {4'h7}}

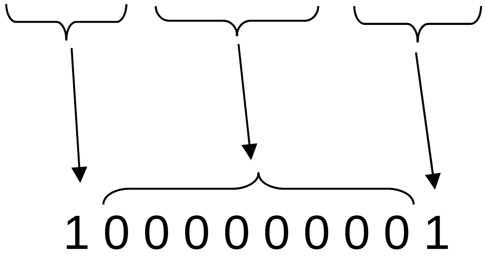
(3) {2{{1'b1},{3'o6}}}

Note: { {1'b1}, 8{1'b0}, {1'b1}}

This will cause error. Use { } as shown in Q3.7-10 (1).

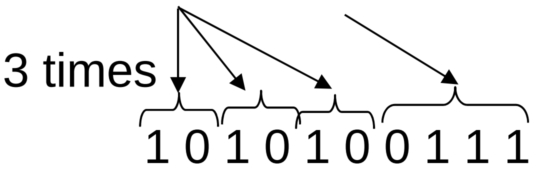
Q3.7-10: A sample answer.

(1) { {1'b1}, {8{1'b0}}, {1'b1}}



1 0 0 0 0 0 0 0 0 1

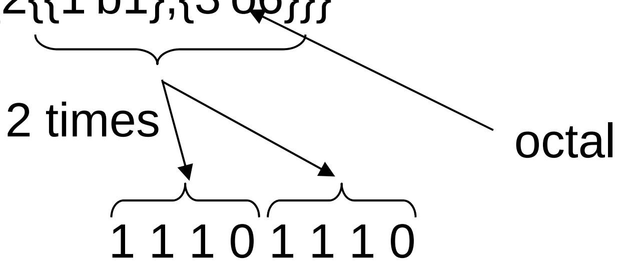
(2) {{3{2'b10}}, {4'h7}}



3 times

1 0 1 0 1 0 0 1 1 1

(3) {2{{1'b1},{3'o6}}}



2 times

octal

1 1 1 0 1 1 1 0

(11) Event trigger operator

syntax

operator event_name

Event trigger operator makes named event occur.

Event trigger operator

operator	operation	example
->	Event trigger	-> abc ; ⟶ A named event abc occurs.

Events (named events) and event trigger operator are not synthesizable. They can be used only in simulation.

(12) Event or operator

syntax

name or name or name or ,,,,,

Event or operator performs logical or operation of any numbers of events.

Event or operator

operator	operation	example
or	Event or	a or b or c → Logical or of a, b, and c events.

"," is allowed to be used instead of "or".

a or b or c ↔ a, b, c

Using "or" is recommended.

3.7.2 Operator precedence

+ - ~ ! & ~& | ~| ^ ~^ ^~	Unary operator	Highest
**	Arithmetic operator	
* / %	Arithmetic operator	
+ -	Arithmetic operator (binary)	
<< <<< >> >>>	Shift operator	
< <= > = >	Relational operator	
== != === !==	Case equality operator	
&	Bitwise and operator	
^ ~^ ^~	Bit-wise exclusive or/equivalence	
|	Bit-wise inclusive or operator	
&&	Logical and operator	
||	Logical or operator	
? :	Conditional operator	
{ }	Replication operator	Lowest

3.7.3 Sizing and expression type rule

In general, Verilog expression is mostly used in the following syntax.

LHS = *expression* ; → example; assign y = a + b ;

or

expression → example; if (a & b) y = 0 ;

There are two types of expression, one is self-determined and the other is context-determined. Sizing occurs depending on whether operands are self-determined or context-determined as shown in the table below.

expression	sizing
Context-determined	Expression is evaluated after operands are sized to max(L(LHS), L(op1), L(op2)) if the expression itself must be sized to max (L(op1), L(op2))
Self-determined	operands are not sized to evaluate their values.

Examples;

Suppose y is 8-bit, a1 is 4-bit, and a2 is 2-bit;

	y = a1 & a2 ;	y = a1 && a2 ;
operands	Operands are context-determined because the operator is bitwise and.	Operands are self-determined because the operator is logical and.
sizing before operation	Operands are sized to 8, max(8, 4, 2).	No sizing.
operation	Bitwise and applied to 8-bit operands.	Operands are evaluated to be 1-bit without sizing.
result	8-bit result is assigned to y	1-bit result is sized to L(y) and assigned to y. See 3.5.

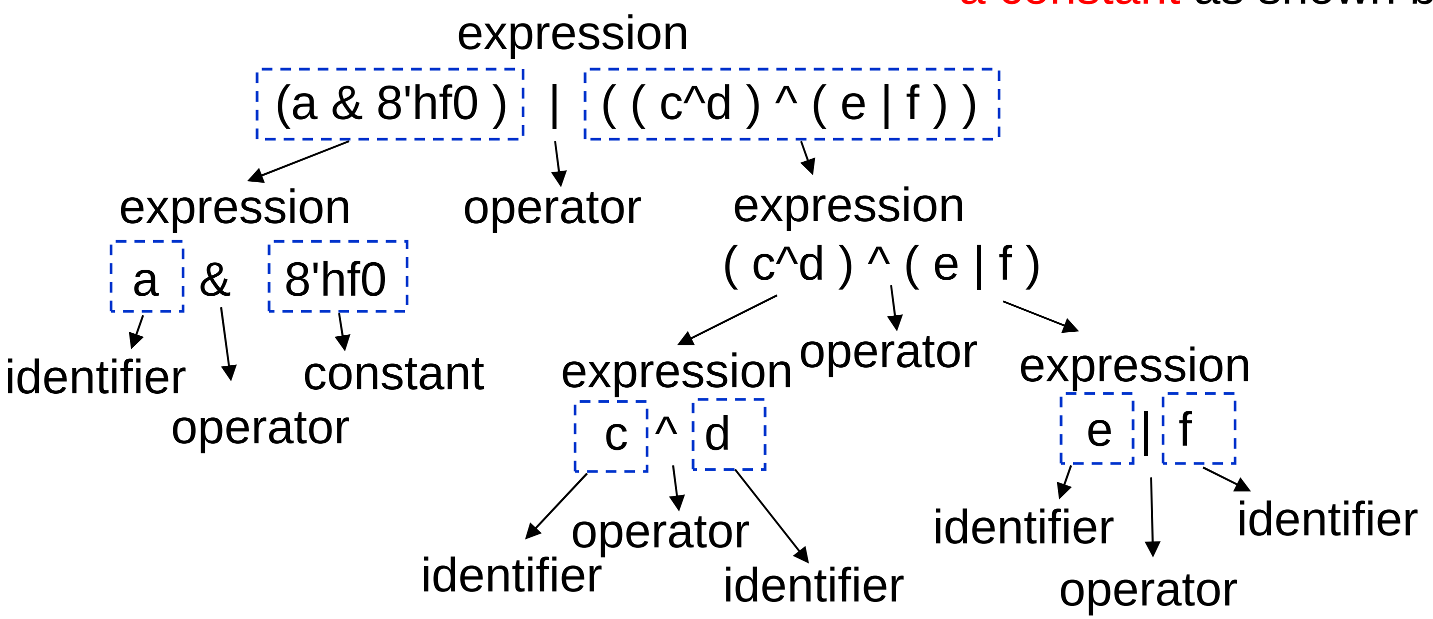
sizing and type rule of expressions

operator		expression	result bit size	type
unary arithmetic *op* a	+, -	*context*	L(a)	Signed if a is signed.
bitwise negate *op* a	~	*context*		
binary arithmetic a *op* b	+, -, /, %	*context*	max(L(a),L(b))	Signed if a and b are signed.
binary bitwise a *op* b	&,|,^,~^,^~	⇧	⇧	
logical negation	!	*self*	1-bit	unsigned
logical and / or	&&, ||	⇧	⇧	⇧
unary reduction	&,~&,|,~|, ^,~^,^~,	⇧	⇧	⇧
relational	<=,>=,<,>	*context*	1-bit	unsigned
equality/inequality	==,!=, ===,!==	⇧	⇧	⇧
shift a *op* b	>>,<<, >>>,<<<	a :*context*, b :*self*	L(a)	Signed if a is signed.
conditional a? b: c	? :	a:*self*, b, c :*context*	max(L(b),L(c))	Signed if b and c are signed.
concatination	{a,b,c,,}	*self*	L(a)+L(b)+L(c),,	unsigned

context means context-determined and sizing occurs before the operation,
self means self-determined and sizing does not occur.

In general, an operand in an expression can be an expression.

In such cases, expressions must be disintegrated down to a simple net or variable specified by identifier or a constant as shown below.



expression

(a & 8'hf0) | ((c^d) ^ (e | f))

expression operator expression

a & 8'hf0

identifier operator constant

(c^d) ^ (e | f)

expression operator expression

c ^ d

identifier operator identifier

e | f

identifier operator identifier

(1) Determine the size and type of each expression applying the rule in the table "sizing and type rule of expressions" from bottom to top.
(2) Propagate the type and size of the expression back down to a simple operand

Q3.7-11: Show the sizing and the result type of the following expressions.

(1) y = ~4'b1010 ; where y is declared 6-bit signed, then what the value of y shall be after this assignment is executed.

(2) a & (b <= (c+1)) where a, b, and c are 4-bit signed, 5-bit signed, and 6-bit unsigned respectively.

(3) a || (b & (^(c|d))) where a, b, c, and d are 4-bit signed, 5-bit unsigned, 6-bit signed, and 4-bit signed respectively.

(4) a + { b , (c - 4'sb0011)} where a, b, and c are 16-bit signed, 6-bit signed, and 8-bit signed respectively.

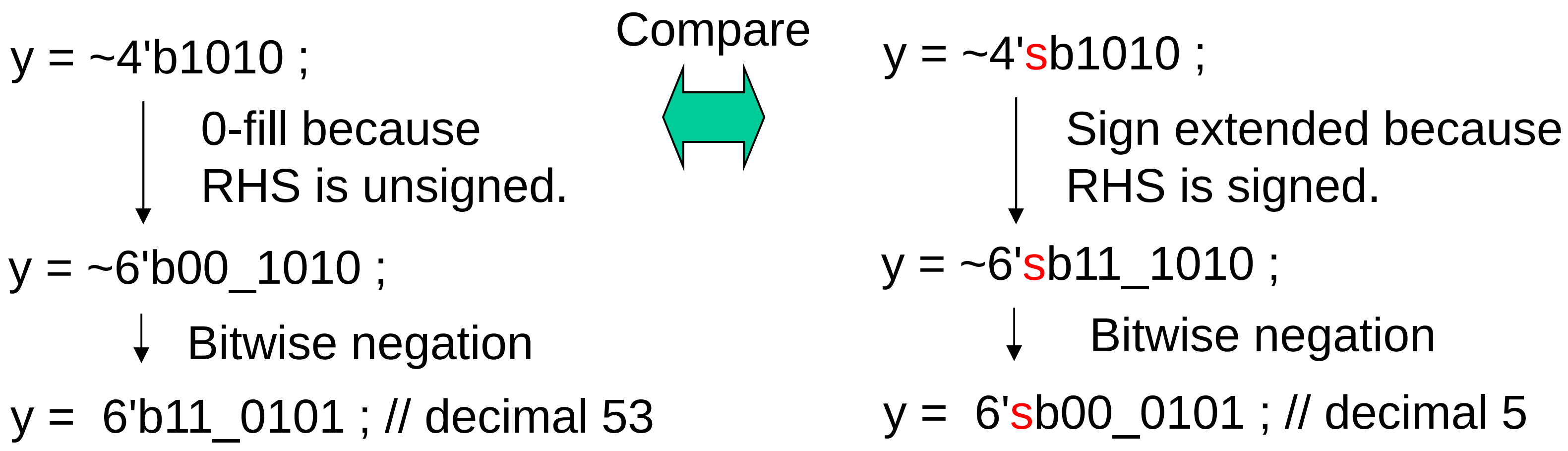
(5) Try much more complex cases such as

```
(~a) & ( (b^c) | ( ( d==2)? e : f  ) + {g, (h & ( k && (m&55)))} )
```

Q3.7-11: A sample answer

(1) y = ~4'b1010 ; where y is declared 6-bit signed, then what the value of y shall be after this assignment is executed.

The assignment is 4-bit to 6bit, and the operand is not self-determined, therefore, the operand is sized to 6-bit before the operation applied.



Compare

y = ~4'b1010 ;

0-fill because RHS is unsigned.

y = ~6'b00_1010 ;

Bitwise negation

y = 6'b11_0101 ; // decimal 53

y = ~4'sb1010 ;

Sign extended because RHS is signed.

y = ~6'sb11_1010 ;

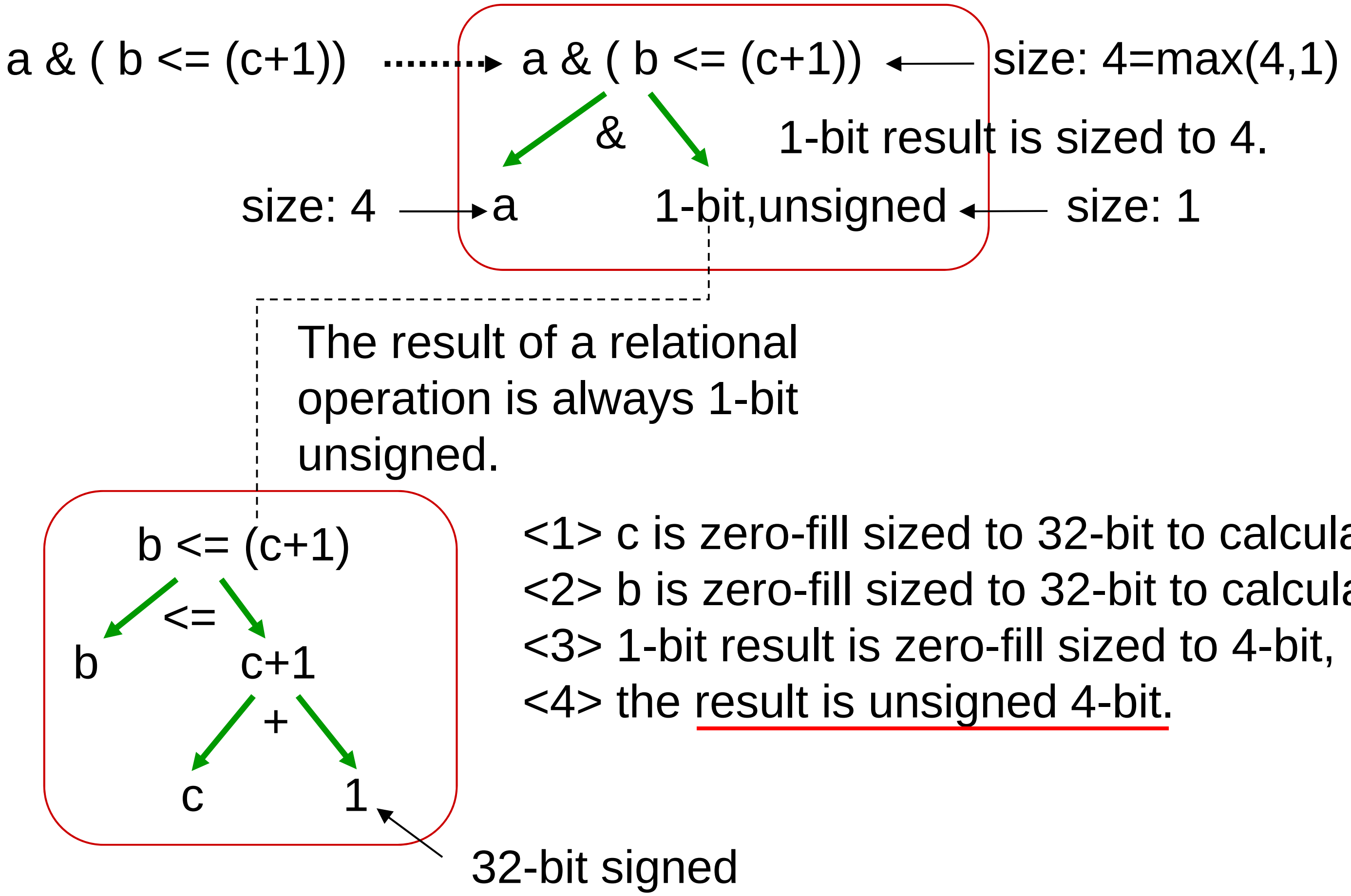
Bitwise negation

y = 6'sb00_0101 ; // decimal 5

The RHS is 6-bit unsigned. But y is declared as signed, therefore, the value of y will be 6-bit decimal -11.

Q3.7-11: A sample answer (continued)

(2) a & (b <= (c+1)) where a, b, and c are 4-bit signed, 5-bit signed, and 6-bit unsigned respectively.

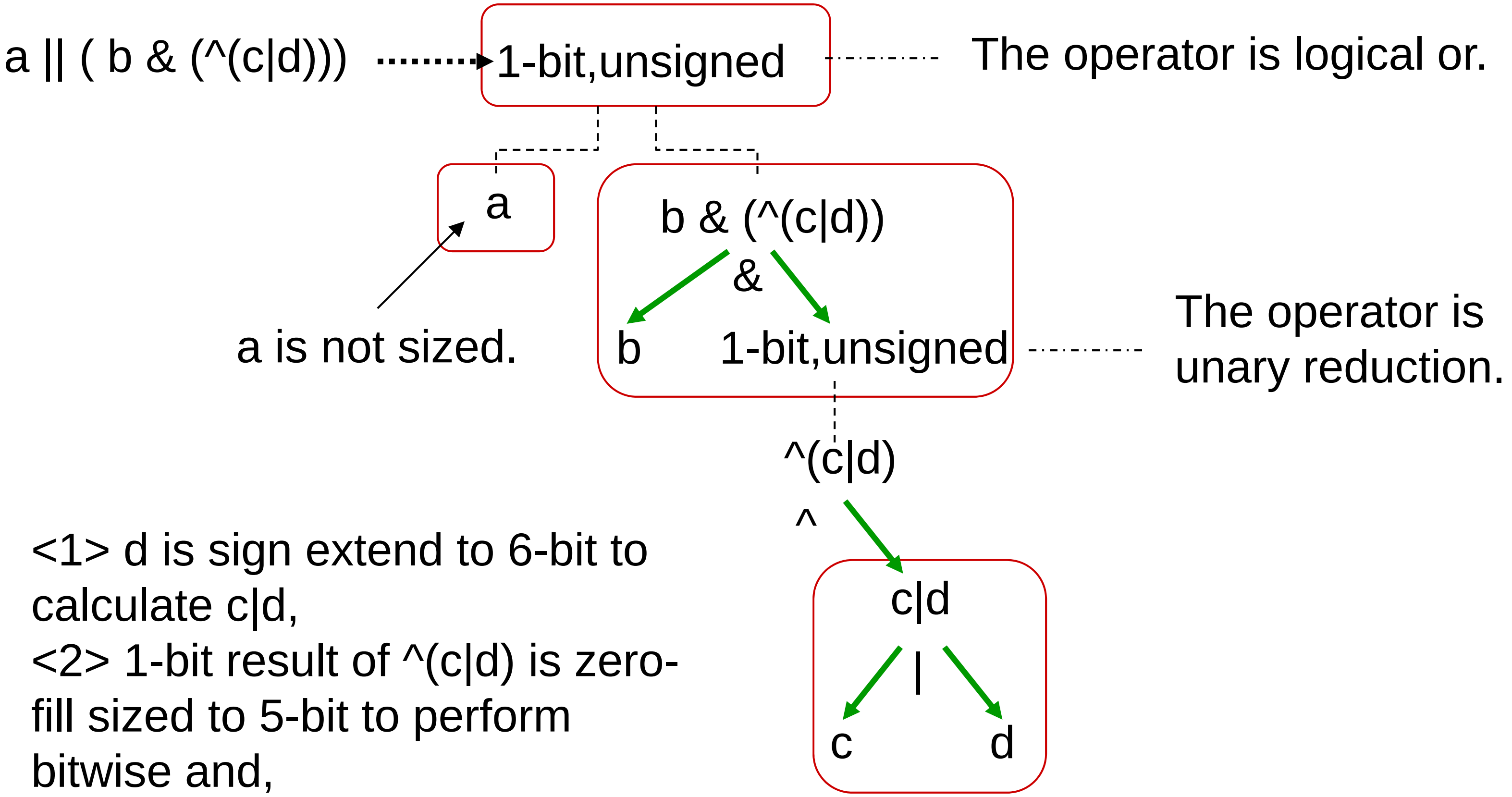


The result of a relational operation is always 1-bit unsigned.

<1> c is zero-fill sized to 32-bit to calculate c+1,
<2> b is zero-fill sized to 32-bit to calculate b<= (c+1),
<3> 1-bit result is zero-fill sized to 4-bit,
<4> the result is unsigned 4-bit.

Q3.7-11: A sample answer (continued)

(3) a || (b & (^(c|d))) where a, b, c, and d are 4-bit signed, 5-bit unsigned, 6-bit signed, and 4-bit signed respectively.



a || (b & (^(c|d)))

1-bit,unsigned

The operator is logical or.

a

a is not sized.

b & (^(c|d))

&

b

1-bit,unsigned

The operator is unary reduction.

^(c|d)

^

c|d

|

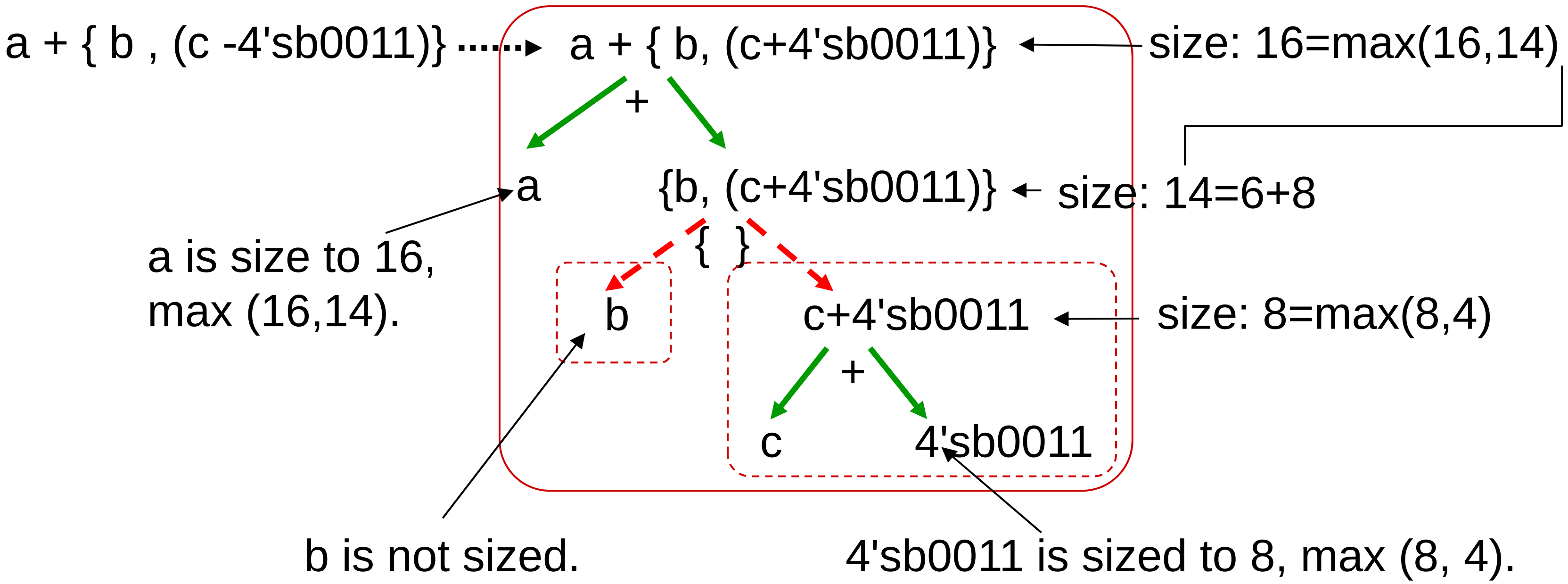
c

d

<1> d is sign extend to 6-bit to calculate c|d,
<2> 1-bit result of ^(c|d) is zero-fill sized to 5-bit to perform bitwise and,
<3> the result is unsigned 1-bit .

Q3.7-11: A sample answer (continued)

(4) a + { b , (c - 4'sb0011)} where a, b, and c are 16-bit signed, 6-bit signed, and 8-bit signed respectively.

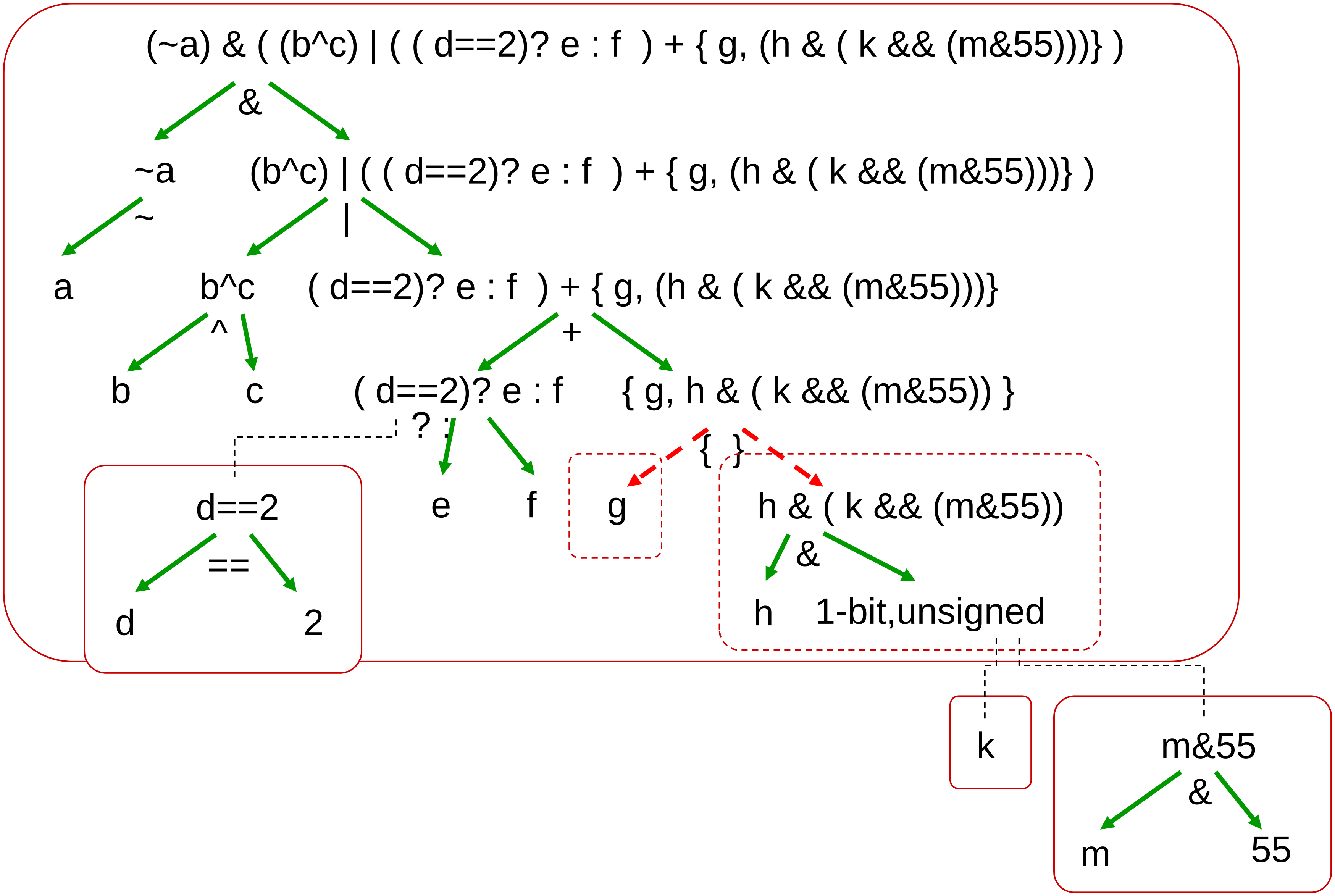


<1> 4'sb0011 is sign extend to 8-bit to calculate c+4'sb0011,,
<2> b (not sized) and 8-bit result of add is concatenated to make 14-bit result,
<3> a is zero-fill sized to 16-bit and concatenated result is zero-fill sized to 16-bit, and added to make unsigned 16-bit result.

Q3.7-11: A sample answer (continued)

(5) Try much more complex cases such as

(~a) & ((b^c) | ((d==2)? e : f) + {g, (h & (k && (m&55)))})


(~a) & ((b^c) | ((d==2)? e : f) + { g, (h & (k && (m&55)))})
&
~a
(b^c) | ((d==2)? e : f) + { g, (h & (k && (m&55)))})
~
|
a
b^c
(d==2)? e : f) + { g, (h & (k && (m&55)))}
^
+
b
c
(d==2)? e : f
{ g, h & (k && (m&55)) }
? :
{ }
d==2
e
f
g
h & (k && (m&55))
==
&
d
2
h
1-bit,unsigned
k
m&55
&
m
55

When assigning a constant value to variables, it is recommended to make the bit width of LHS and LHS equal.

```
reg [15:0] sig_a ;

sig_a = 1 ;
```

△

Verilog rule says that an integer (1) is at lease 32-bit. Therefore, the above expression must be OK. However, it is recommended to use an expression on the right to make it sure that the bit width of LHS and RHS are always equal for any possible bit length.

```
reg [15:0] sig_a ;

sig_a = 16’h0001 ;
```

○

```
parameter SIG_A_BW = 16 ;
reg [SIG_A_BW -1:0] sig_a ;

sig_a = { { (SIG_A_BW-1){1’b0} }, {1’b1} } ;
```

◎

3.8 Parameter and define

3.8.1 Parameter

Use parameter for constants to improve readability and reusability. By using parameters, constants can be changed easily for applying the logic to different environment or to other projects.

△

```
if ( cntr > 50 ) begin
  ,,,,,,,,
```

→

◎

```
parameter CNTR_MAX = 50 ;

if ( cntr > CNTR_MAX ) begin
  ,,,,,,,,
```

It is clear that 50 is the maximum value of the counter.

△

```
wire [15:0] sig_w ;
```

→

◎

```
parameter SIG_W_BW = 16 ;

wire [SIG_W_BW-1:0] sig_w ;
```

Bit width can be modified easily by changing a parameter.

```
module abc ;

wire [ 31:0] sig_y ;
            ⋮

// check MSB of sig_y
 if ( sig_y[ 31 ] == 1'b1 )
   begin
```

The above code
has less flexibility.

By changing this parameter from 32 to 64, same logic code can be applied for 64-bit width design.

```
module abc ;
parameter SIG_Y_BW = 32 ;

wire [SIG_Y_BW –1 : 0] sig_y ;
            ⋮

// check MSB of sig_y
 if ( sig_y[SIG_Y_BW –1 ] == 1'b1 )
   begin
```

Be sensitive about bit width when using parameter.

Parameter defined without bit width specification has 32-bit length.

```
parameter CNST_ONE = 1 ;

wire [7:0] a, b ;
assign w = ( a[7:0 ] < ( b[7:0] – CNST_ONE ) )? 1'b1 : 1'b0 ;
```

w = 1 if a is 8'hFF and b is 8'h00.

```
parameter CNST_ONE = 8'h01 ;

wire [7:0] a, b ;
assign w = ( a[7:0 ] < ( b[7:0] – CNST_ONE ) )? 1"b1 : 1'b0 ;
```

w = 0 if a is 8'hFF and b is 8'h00.

When defining a constant by using parameter keyword, define bit width.

The followings are both legal;

```
parameter [7:0] CNST_ONE = 1 ;
parameter signed [7:0] C_ONE = 1 ;
```

Q3.8-1 Write an expression by using parameter BIT_W which can represent the following expressions separated by commas. BIT_W must be equal to the bit width of each expressions.

(1) 1’b1, 2’b10, 3’b100, 4’b1000, 5’b1_0000, 8’b1000_0000

BIT_W is not less than 1.

(2) 1’b1, 2’b01, 3’b001, 4’b0001, 5’b0_0001, 8’b0000_0001

BIT_W is not less than 1.

(3) 2’b10, 3’b010, 4’b0010, 5’b0_0010, 8’b0000_0010

BIT_W is not less than 2.

(4) 2’b10, 4’b1010, 6’b10_1010, 8’b1010_1010, 16’b1010_1010_1010_1010

BIT_W is even and not less than 2.

Q3.8-1 Sample answers.

(1) 1’b1, 2’b10, 3’b100, 4’b1000, 5’b1_0000, 8’b1000_0000

```
{ {1’b1}, {(BIT_W-1){1’b0} } }
```

This is simulatable and synthesizable, but will get warning with BIT_W=1

(2) 1’b1, 2’b01, 3’b001, 4’b0001, 5’b0_0001, 8’b0000_0001

```
{ {(BIT_W-1){1’b0} } , {1’b1} }
```

(3) 2’b10, 3’b010, 4’b0010, 5’b0_0010, 8’b0000_0010

```
{ {(BIT_W-2){1’b0} } , {2’b10} }
```

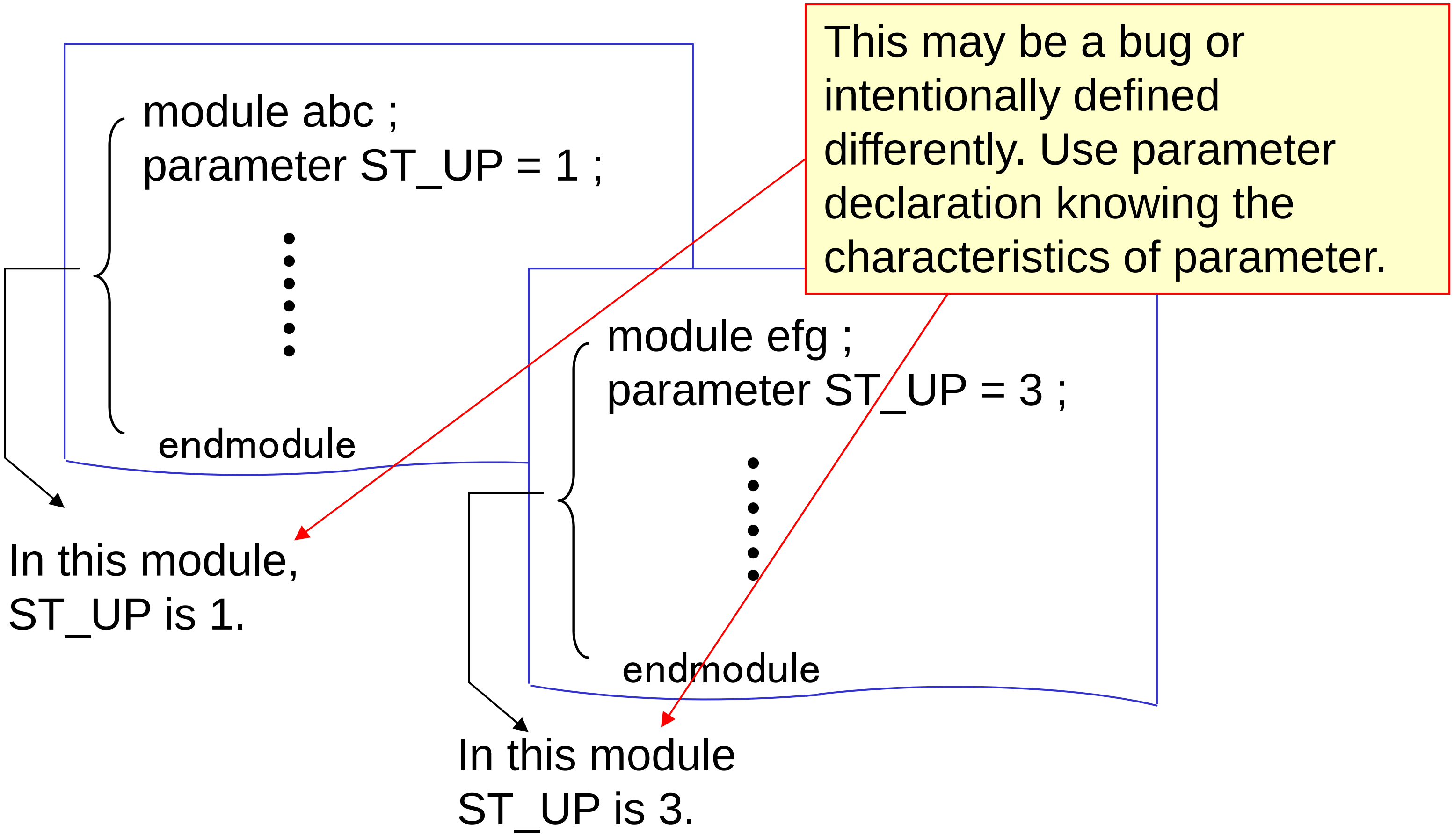
This is simulatable and synthesizable, but will get warning with BIT_W=2

(4) 2’b10, 4’b1010, 6’b10_1010, 8’b1010_1010, 16’b1010_1010_1010_1010

```
{ {(BIT_W/2){2’b10} } }
```

Parameter is local

Parameter declaration is valid only within the module. Parameters defined in upper layer modules are not passed to lower layer modules.



```
module abc ;
parameter ST_UP = 1 ;
      ⋮
endmodule
```

In this module, ST_UP is 1.

```
module efg ;
parameter ST_UP = 3 ;
      ⋮
endmodule
```

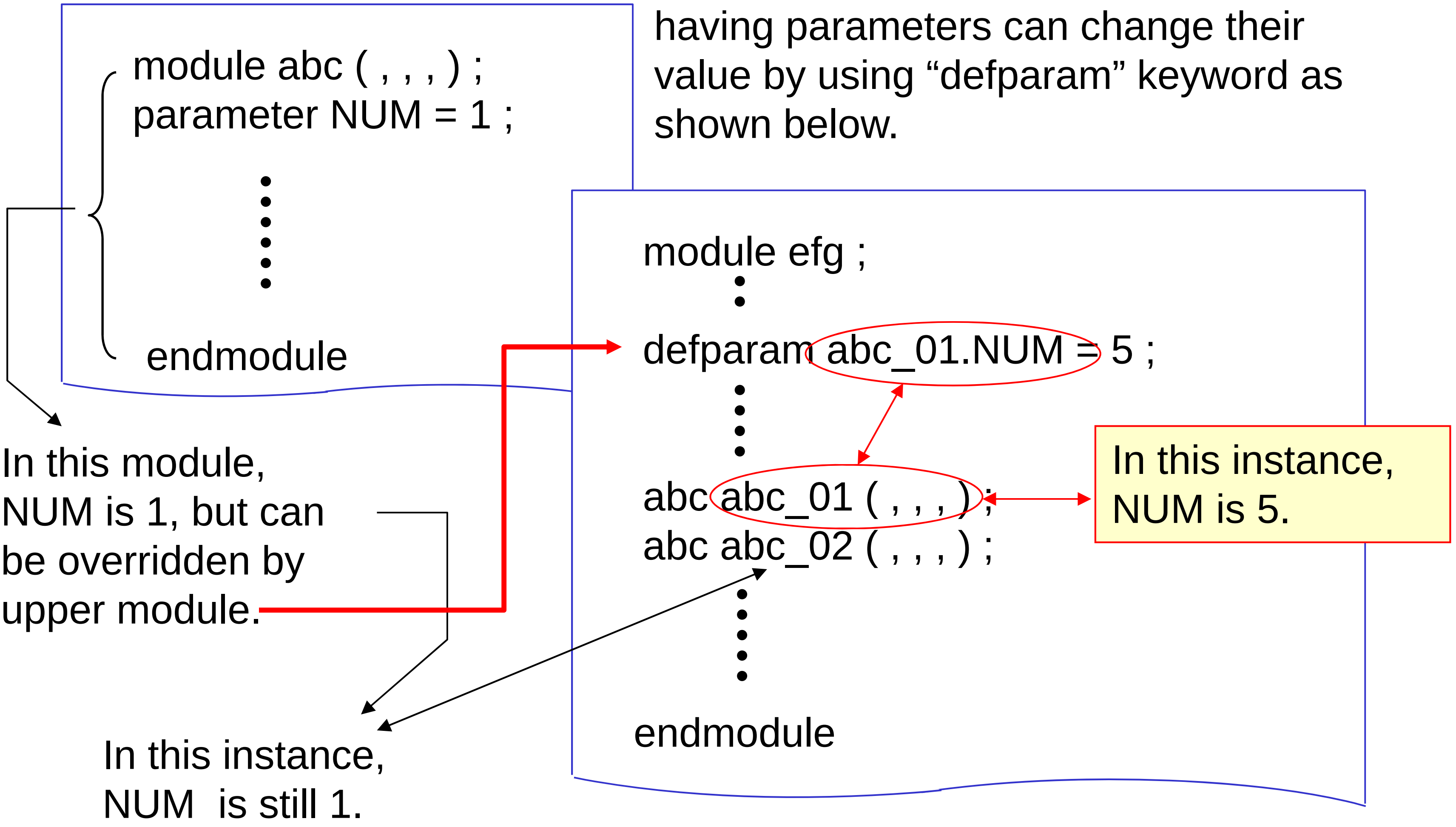
In this module ST_UP is 3.

This may be a bug or intentionally defined differently. Use parameter declaration knowing the characteristics of parameter.

parameter has strong locality.

Parameter can be over-written as shown below.

Therefore, a module using a module having parameters can change their value by using “defparam” keyword as shown below.



```
module abc ( , , , ) ;
parameter NUM = 1 ;
        :
 endmodule
```

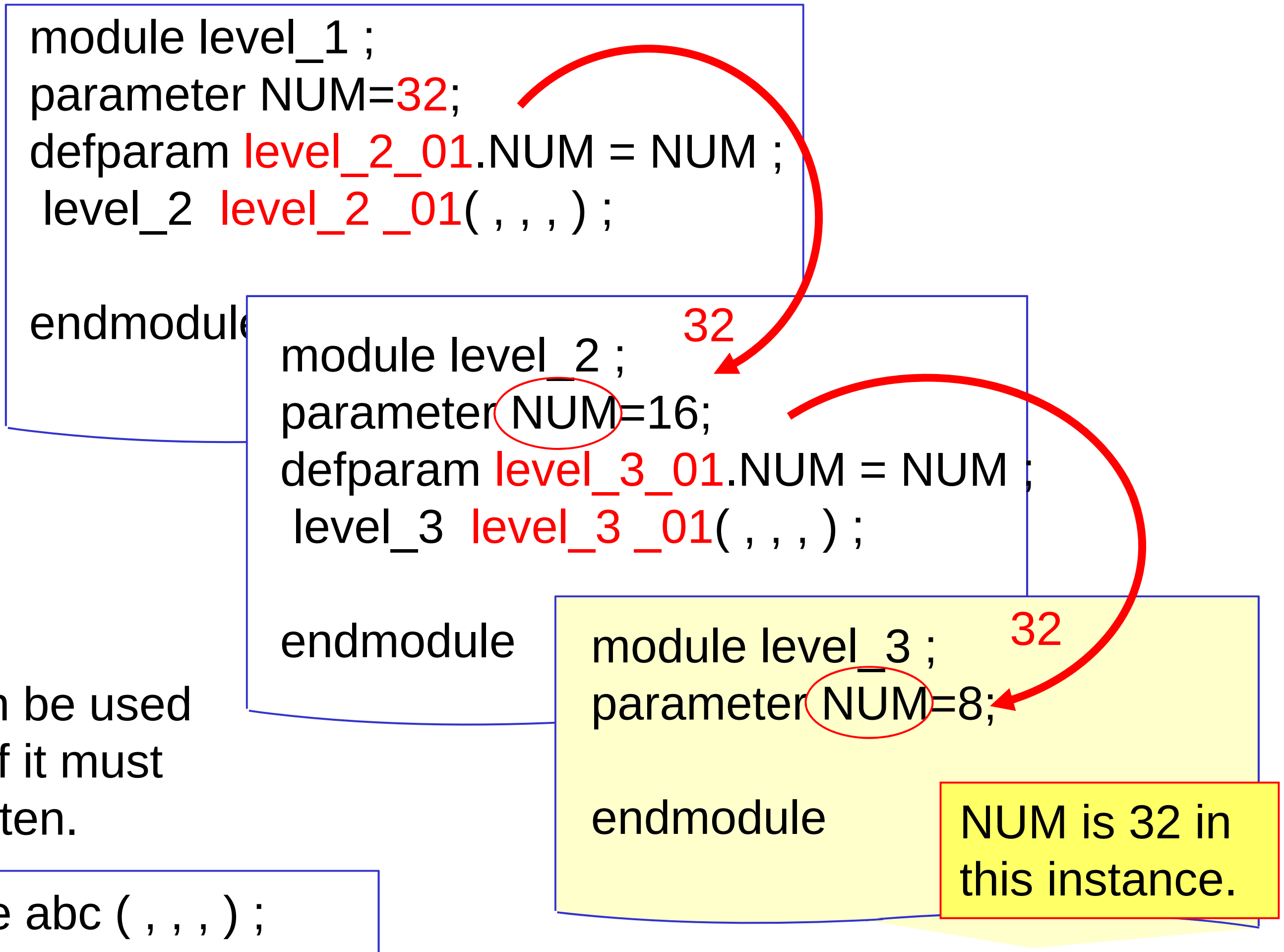
In this module, NUM is 1, but can be overridden by upper module.

```
module efg ;
    :
defparam abc_01.NUM = 5 ;
    :
abc abc_01 ( , , , ) ;
abc abc_02 ( , , , ) ;
    :
endmodule
```

In this instance, NUM is 5.

In this instance, NUM is still 1.

Nesting defparam



```
module level_1 ;
parameter NUM=32;
defparam level_2_01.NUM = NUM ;
 level_2  level_2 _01( , , , ) ;

endmodule
```

```
module level_2 ;
parameter NUM=16;
defparam level_3_01.NUM = NUM ;
 level_3  level_3 _01( , , , ) ;

endmodule
```

```
module level_3 ;
parameter NUM=8;

endmodule
```

NUM is 32 in this instance.

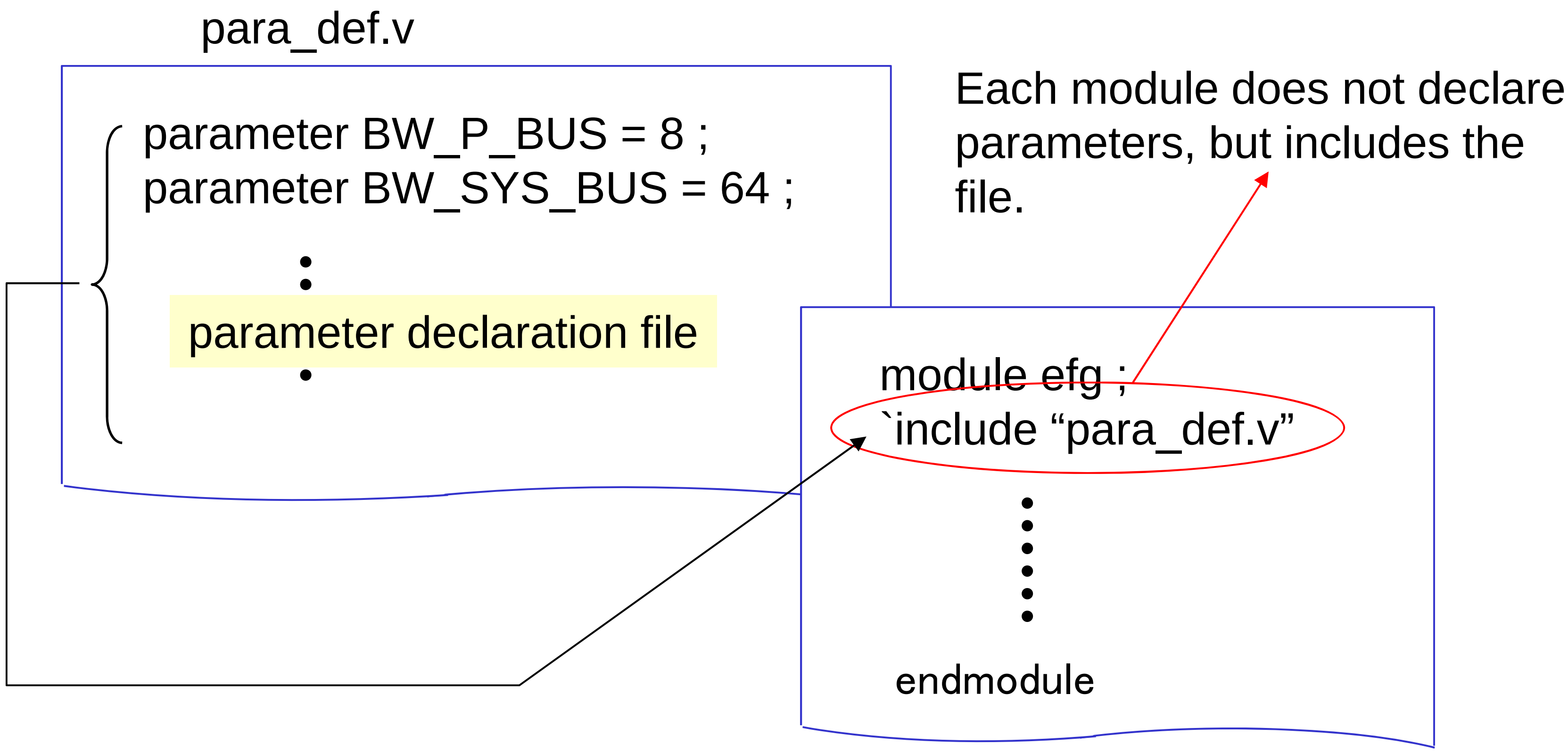
localparam

localparam can be used for parameter if it must not be overwritten.

```
module abc ( , , , ) ;
localparam NUM = 10 ;
        :
 endmodule
```

overwriting MUM, defined by “localparam”, is illegal.

To use the same parameter definition throughout a project, it is better to create one file of parameter declarations as below and include it into each module using the parameters.



para_def.v

```
parameter BW_P_BUS = 8 ;
parameter BW_SYS_BUS = 64 ;
        :
        •
```

parameter declaration file

Each module does not declare parameters, but includes the file.

```
module efg ;
`include “para_def.v”
      :
      :
      :
endmodule
```

In this module, BW_P_BUS is 8 and BW_SYS_BUS is 64.

Consideration on what shall be parameterized.

Parameters can be used to make the code flexible, adjustable for various environments.
Therefore, we have to make it clear what shall be declared as parameters and what are not. Chose parameters so that changing one parameter will result in correct modification of the code.

Example:

good

```
parameter HF_CYCLE = 50 ;
parameter CYCLE = HF_CYCLE*2 ;
```

Only HF_CYCLE must be rewritten to change the cycle.

wrong

```
parameter HF_CYCLE = 50 ;
parameter CYCLE = 100;
```

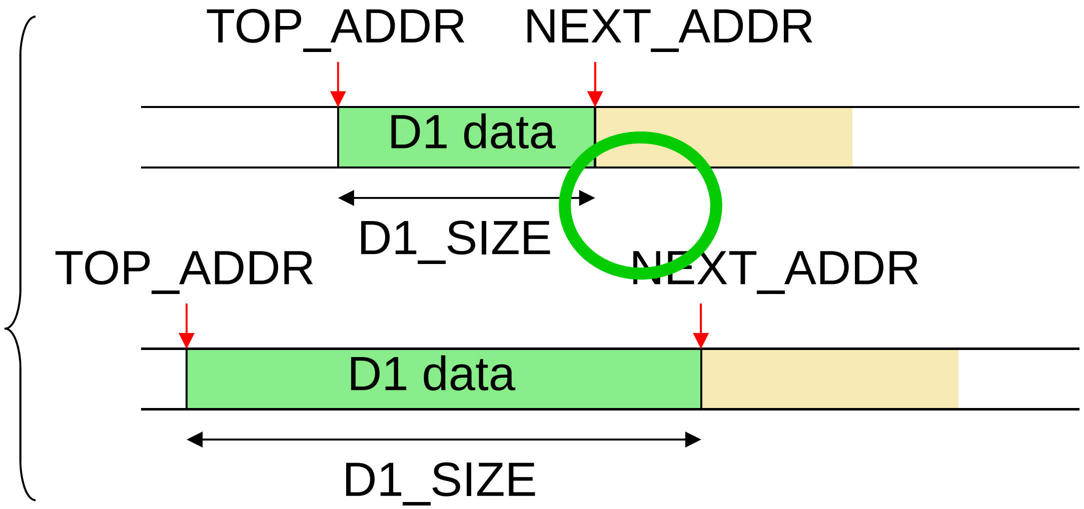
Both HF_CYCLE and CYCLE must be rewritten to change the cycle.

How to define parameters depends on the characteristics of the system. Depending on the nature of the system some way of defining parameters is good, but the same definition may be wrong if it is applied to different system having different characteristics as shown on the next slides.

```
parameter TOP_ADDR = 1000 ;
parameter D1_SIZE = 200 ;
parameter NEXT_ADDR = TOP_ADDR + D1_SIZE ;
```

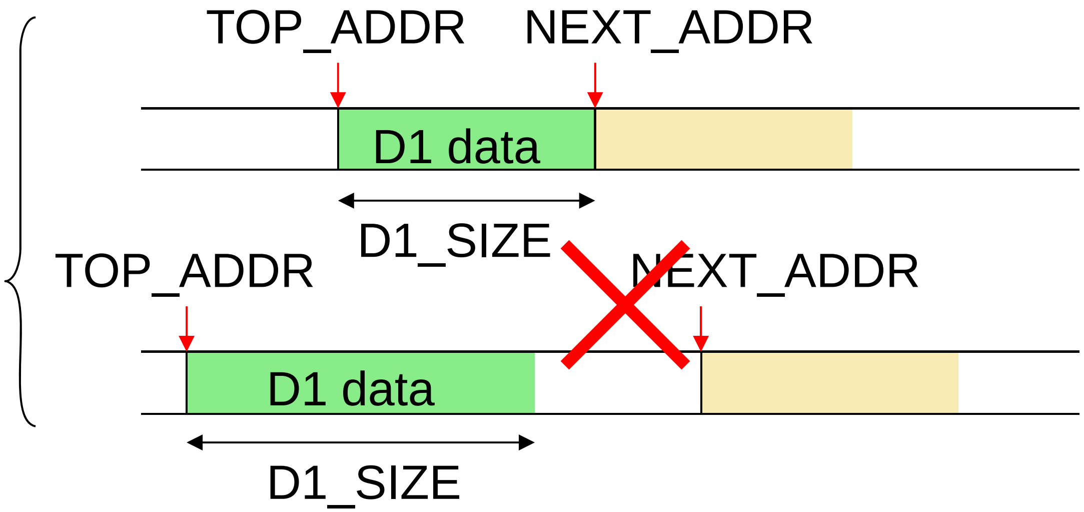
Good if the relation shown on the right always holds true.

NEXT_ADDR always starts right after D1 data.



Wrong if the relation does not always hold true.

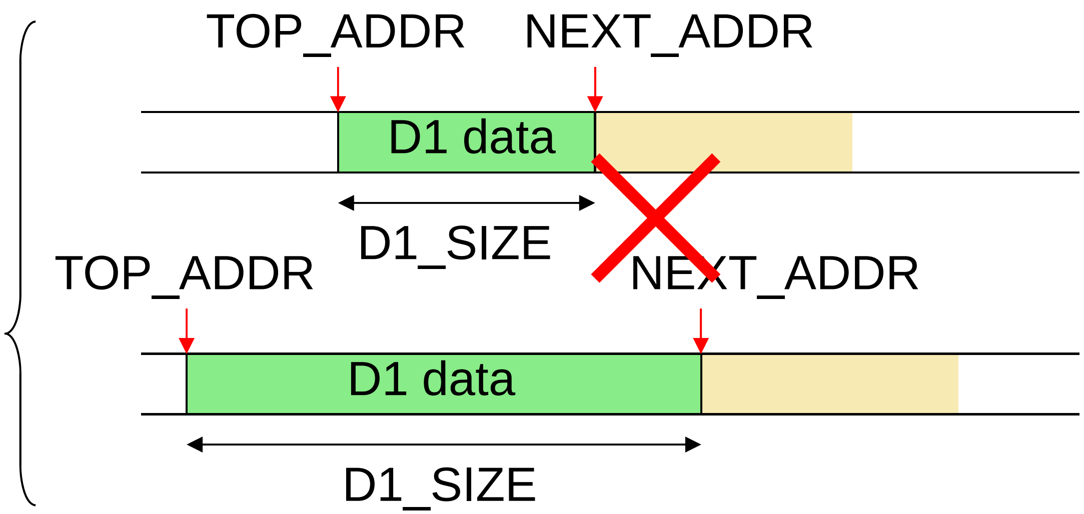
NEXT_ADDR does not always start right after D1 data.



```
parameter TOP_ADDR = 1000 ;
parameter NEXT_ADDR = 1200 ;
```

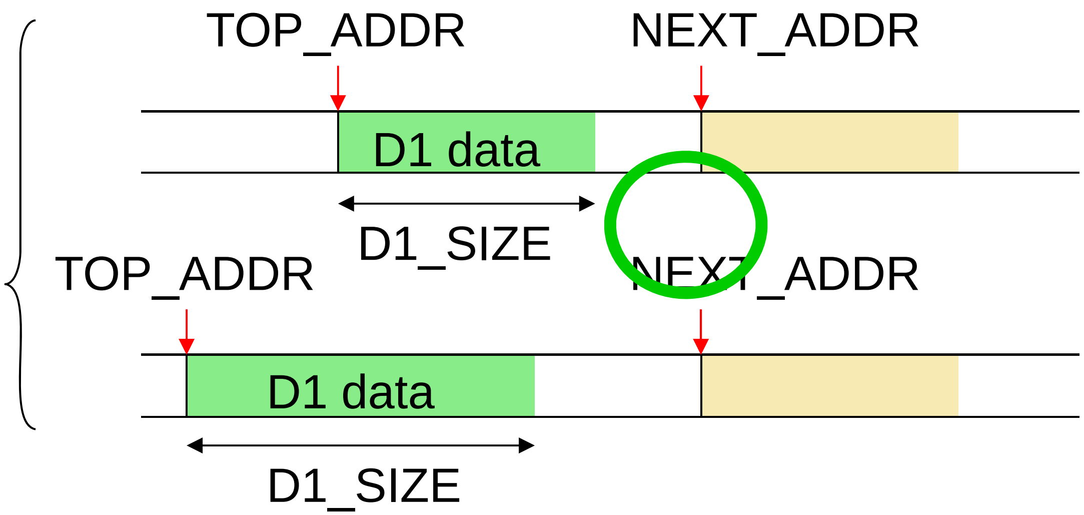
Wrong if the relation shown on the right always holds true.

NEXT_ADDR always starts right after D1 data.



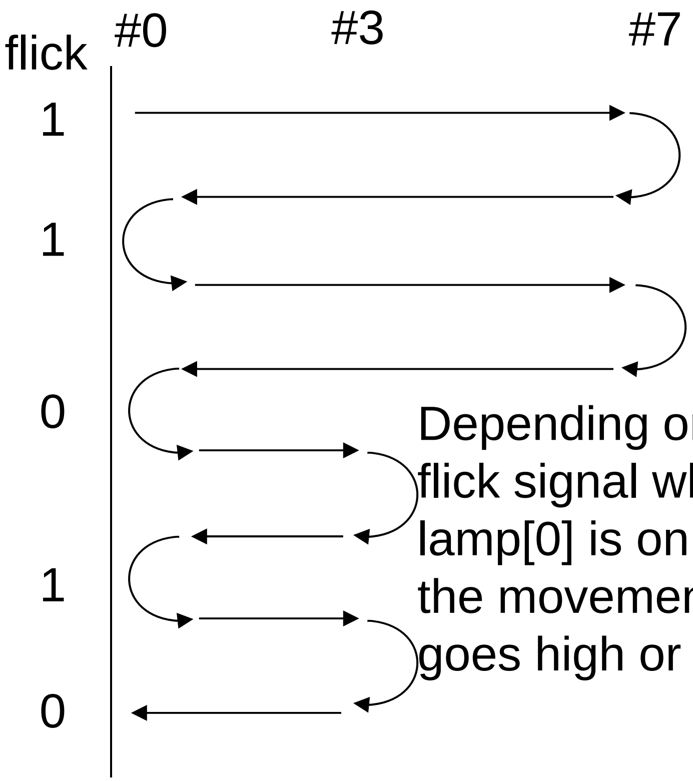
Good if the D1 data size has no relation with NEXT_ADDR.

NEXT_ADDR is defined by other factors than TOP_ADDR and D1 data size.



Take example of 10.4 bound flasher.

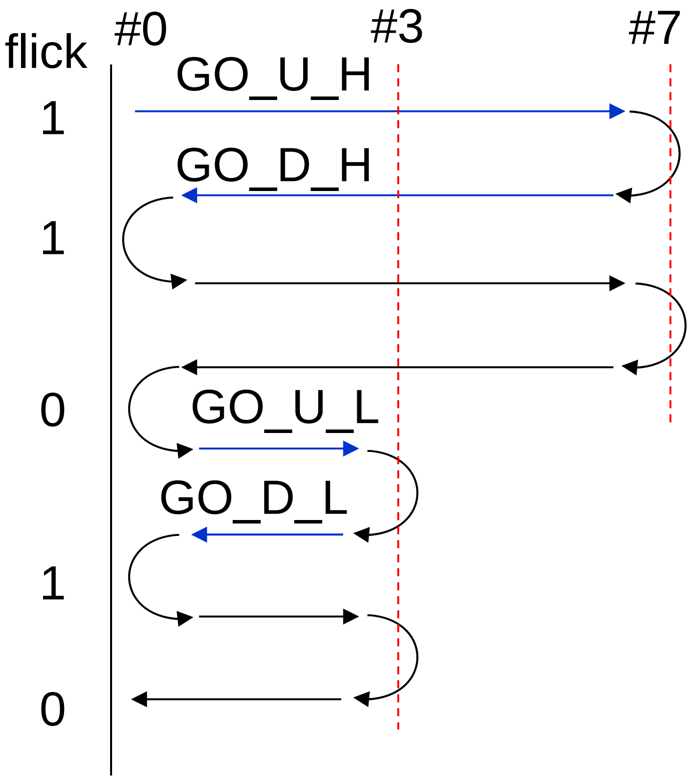
This is a 8-bit lamp system. One bit corresponds to one lamp and #k bit on means #k lamp is on. At most only one bit is on at a time and the on bit position moves right and left, up and down, every clock cycle.
The system is expected to behave as shown on the right. On bit position moves from #0 to #7 and comes back to #0 and go up to #3, and goes down to #0.



Depending on flick signal when lamp[0] is on, the movement goes high or low.

Now design the system using RTL.
Most simple program must look like the one on the next slide.

```
parameter NUM_LP= 8 ;
reg [ NUM_LP - 1 : 0] lamp ;

case(state)
,,,
,,
GO_U_H : begin
   next_state =( lamp[ 6 ] ==1’b1 )?
      GO_D_H : state ;
end
,,,
,,
GO_U_L : begin
 next_state =( lamp[ 2 ] ==1’b1 )?
      GO_D_L : state ;
end
,,
endcase
```



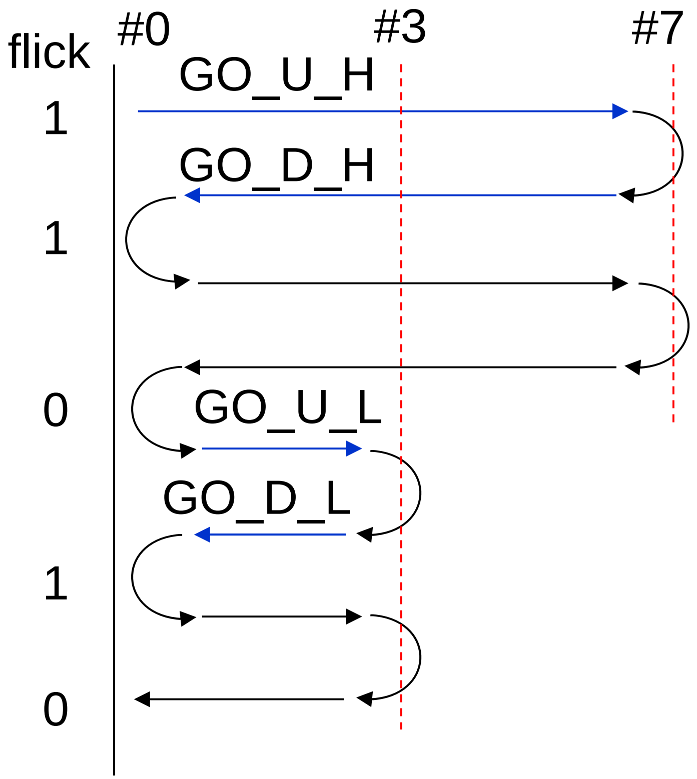
But this code will work properly only when
(1) the first round always goes up to #7 lamp,
(2) another round always goes up to #3 lamp,
(3) bouncing always at #0 lamp, and
(4) NUM_LP is 8.

We must write more robust code.

To make the code robust, properly work even if the requirements changed, we have to make our code prepared for the change of the specification.

But it is impossible to make our code ready for any kind of changes.

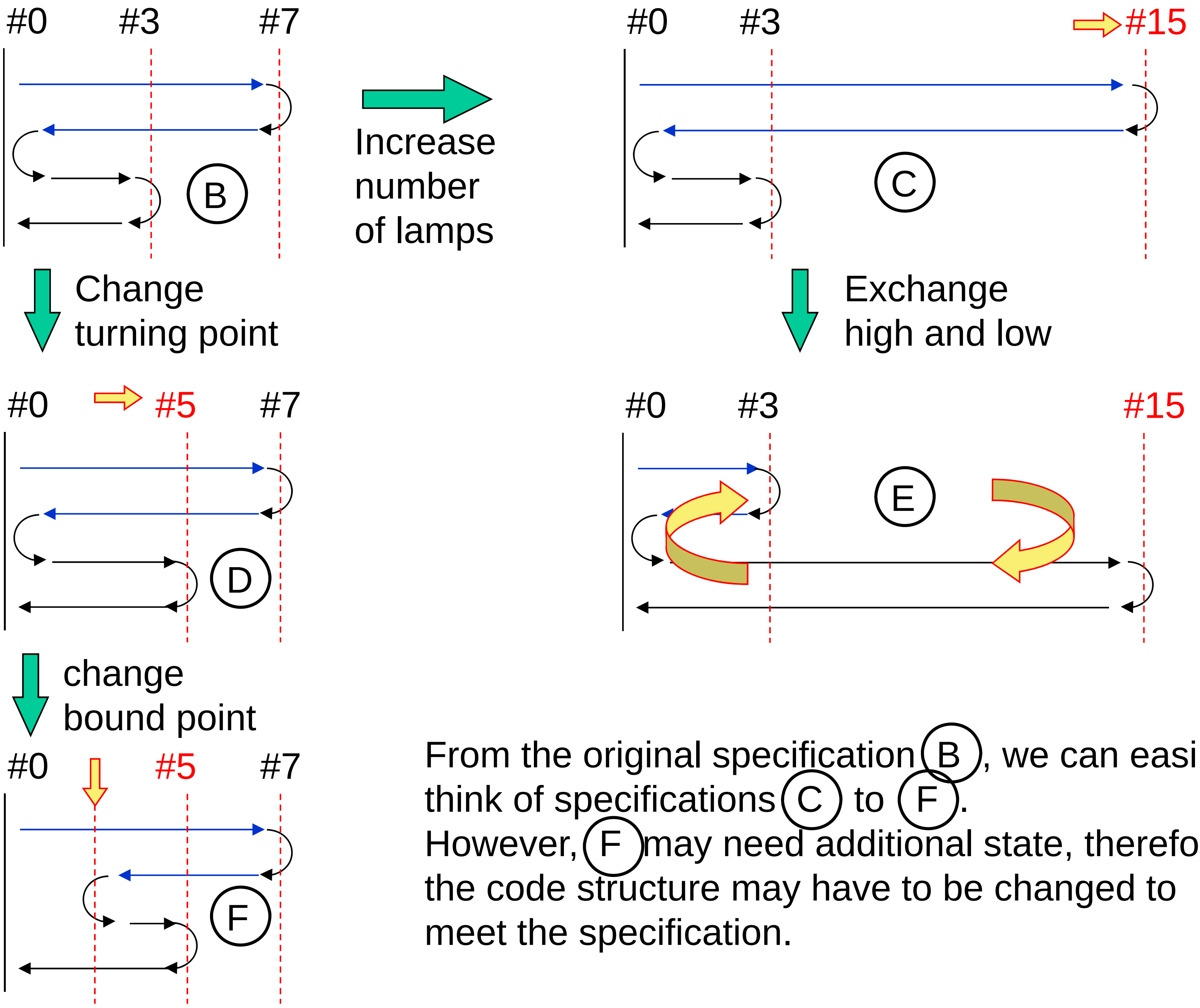
Make our code ready for most probable changes, such as number of lamps or turning points.



What are the possible changes of the specification?

This is the most important thing to think when defining parameters.

Use your imaginative power.



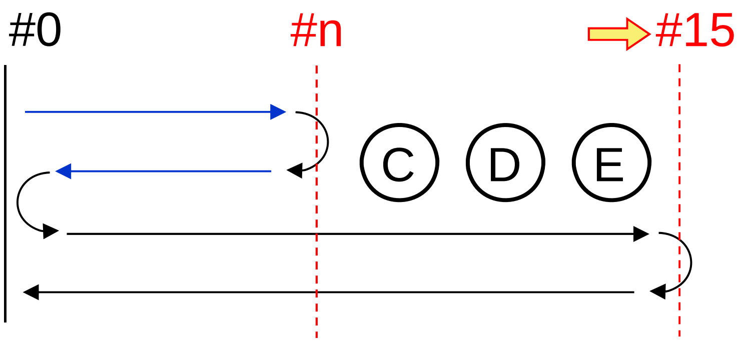
Increase number of lamps

Change turning point

Exchange high and low

change bound point

From the original specification Ⓑ, we can easily think of specifications Ⓒ to Ⓕ. However, Ⓕ may need additional state, therefore the code structure may have to be changed to meet the specification.



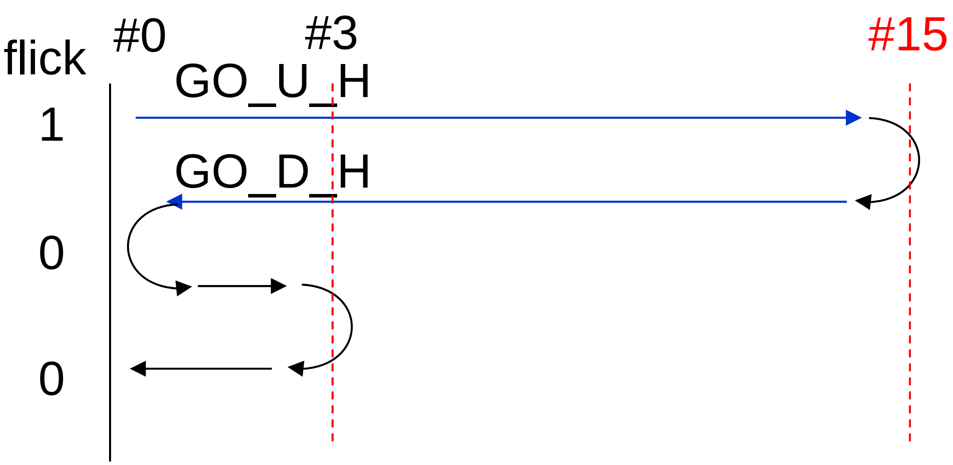
To be ready for the system variety, Ⓒ, Ⓓ, and Ⓔ, the following parameters must be introduced beside the number of lamps.

The first round may not turn back at max. → A new parameter FST_LMT for the first turn back point must be introduced.

The second round may not turn back at #3. → A new parameter SCND_LMT for the second turn back point must be introduced.

Always bounce at #0 → Do not have to parametrize #0.

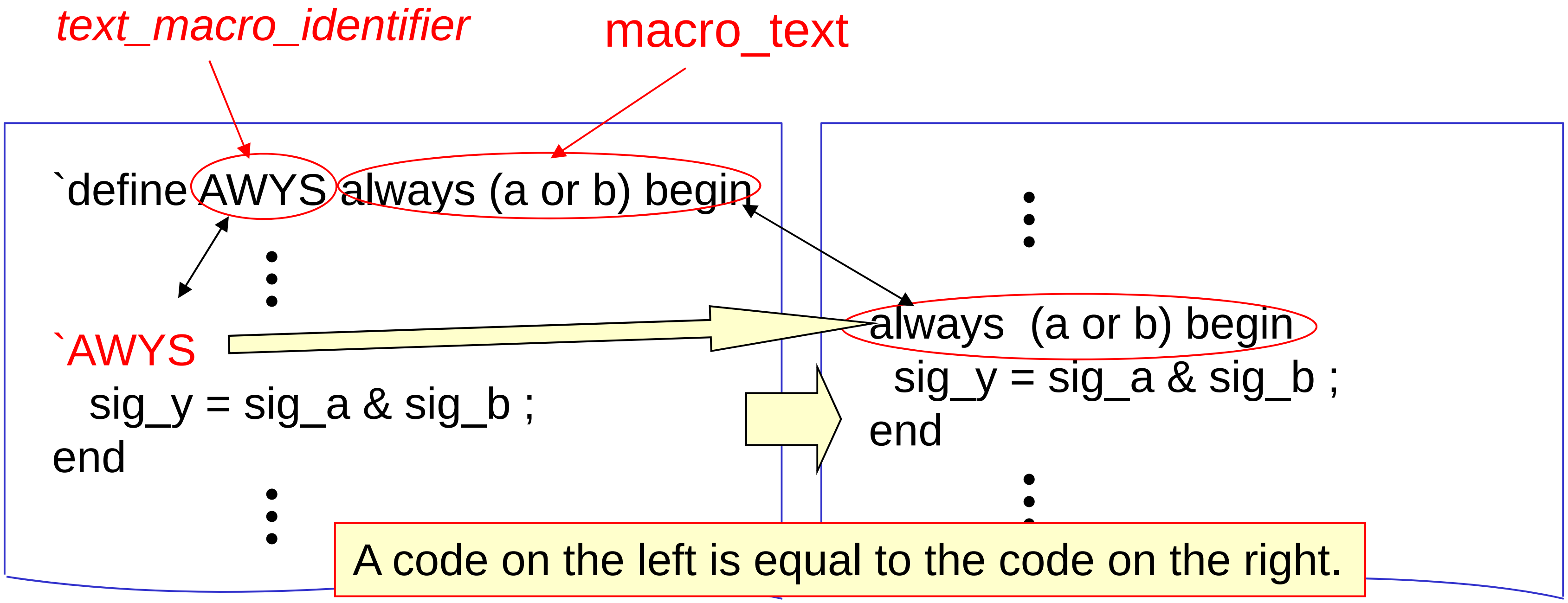
```
parameter NUM_LP= ~~8~~ 16;
parameter FST_LMT=NUM_LP-1 ;
parameter SCND_LMT= 3 ;
reg [ NUM_LP - 1 : 0] lamp ;

case(state)
,,,
,,
GO_U_H : begin
   next_state =( lamp[ FST_LMT-1 ] ==1'b1 )?
      GO_D_H : state ;
end
,,,
,,
GO_U_L : begin
 next_state =( lamp[ SCND_LMT-1 ] ==1'b1 )?
      GO_D_L : state ;
end
,,
endcase
```



The code on the left is ready for the system variety of Ⓒ, Ⓓ , and Ⓔ .

3.8.2 Define

“define” is a compiler directive and replaces A text macro identifier with a macro text.
The example below shows what “define” does.



text_macro_identifier

macro_text

```
`define AWYS always (a or b) begin
        ⋮
`AWYS
   sig_y = sig_a & sig_b ;
end
        ⋮
```

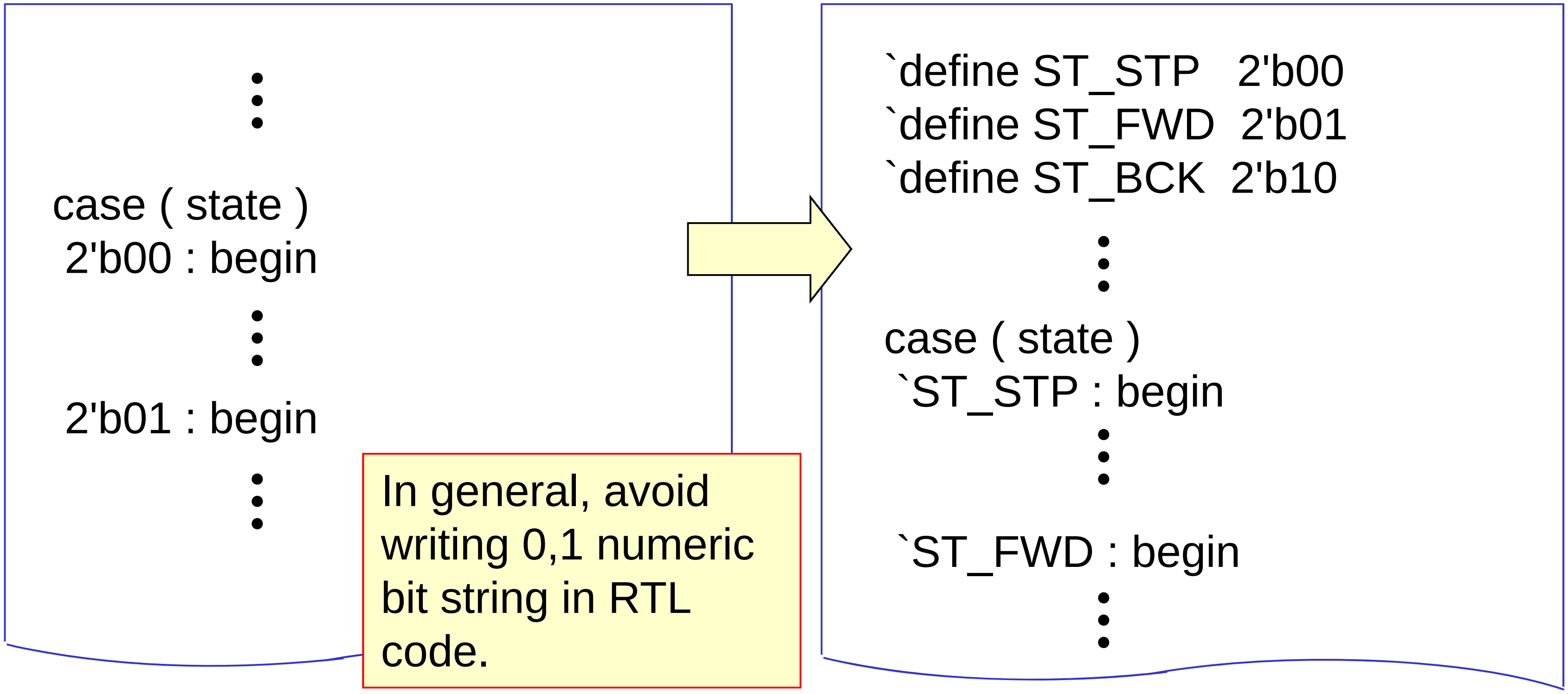
```
        ⋮
always  (a or b) begin
  sig_y = sig_a & sig_b ;
end
        ⋮
```

A code on the left is equal to the code on the right.

Do not use tricky technique as above.
The above code is only for explanation.

Keywords are not allowed to be replaced by define.

“define” can be used, in a similar way with parameters, to improve readability of code as shown below.



```
        ⋮
case ( state )
 2'b00 : begin
        ⋮
 2'b01 : begin
        ⋮
```

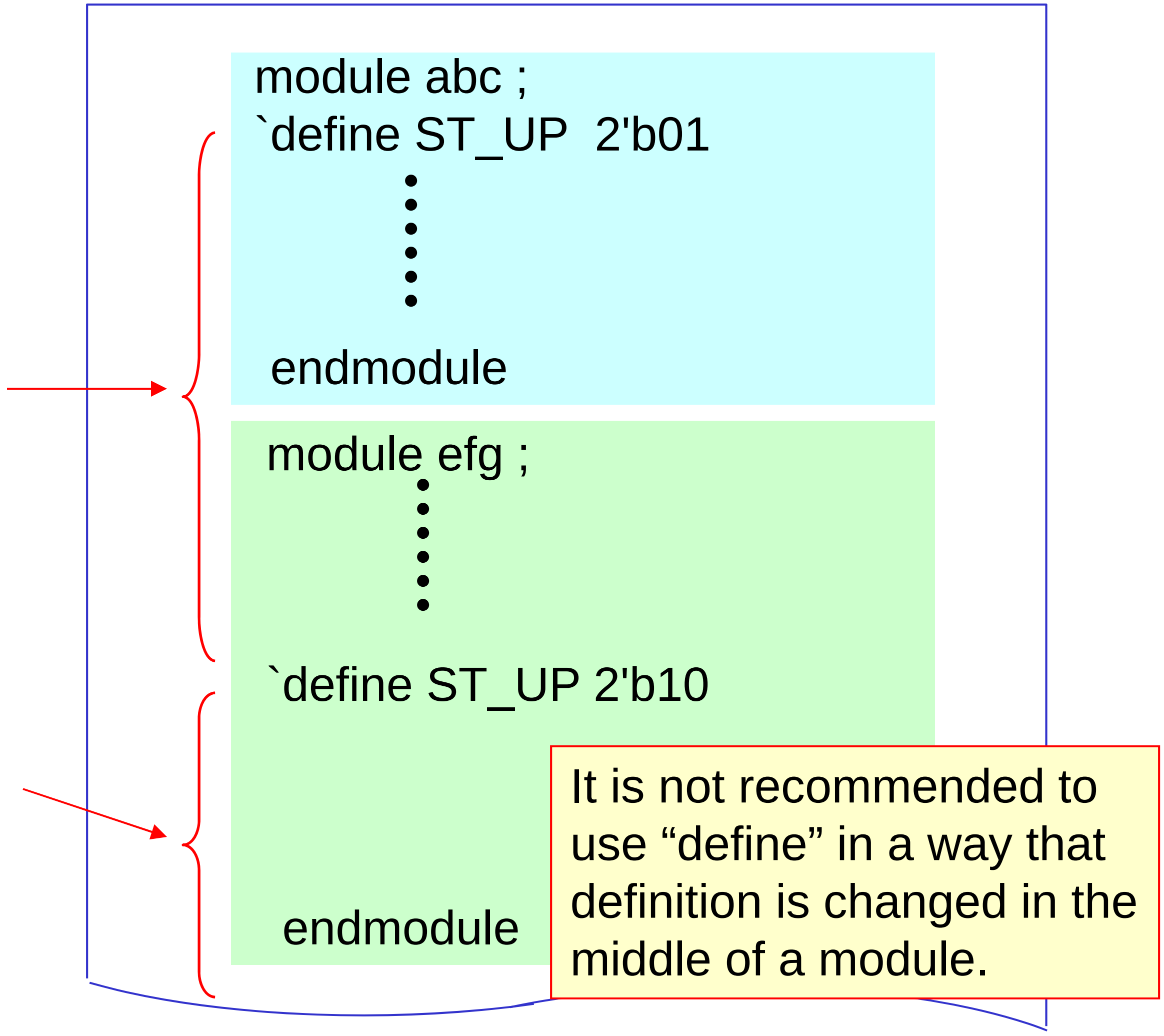
```
`define ST_STP   2'b00
`define ST_FWD  2'b01
`define ST_BCK  2'b10
          ⋮
case ( state )
 `ST_STP : begin
          ⋮
 `ST_FWD : begin
          ⋮
```

In general, avoid writing 0,1 numeric bit string in RTL code.

“define” can be used to define state variable’s value, but in general, using parameter is preferable.

Define is global

Define is valid after it appears first, until it is overwritten by other define. In the example below, ST_UP is 2'b01 until the code line where it is defined as 2'b10.



Define is valid over module boundary, therefore ST_up is 2'b01 in this range.

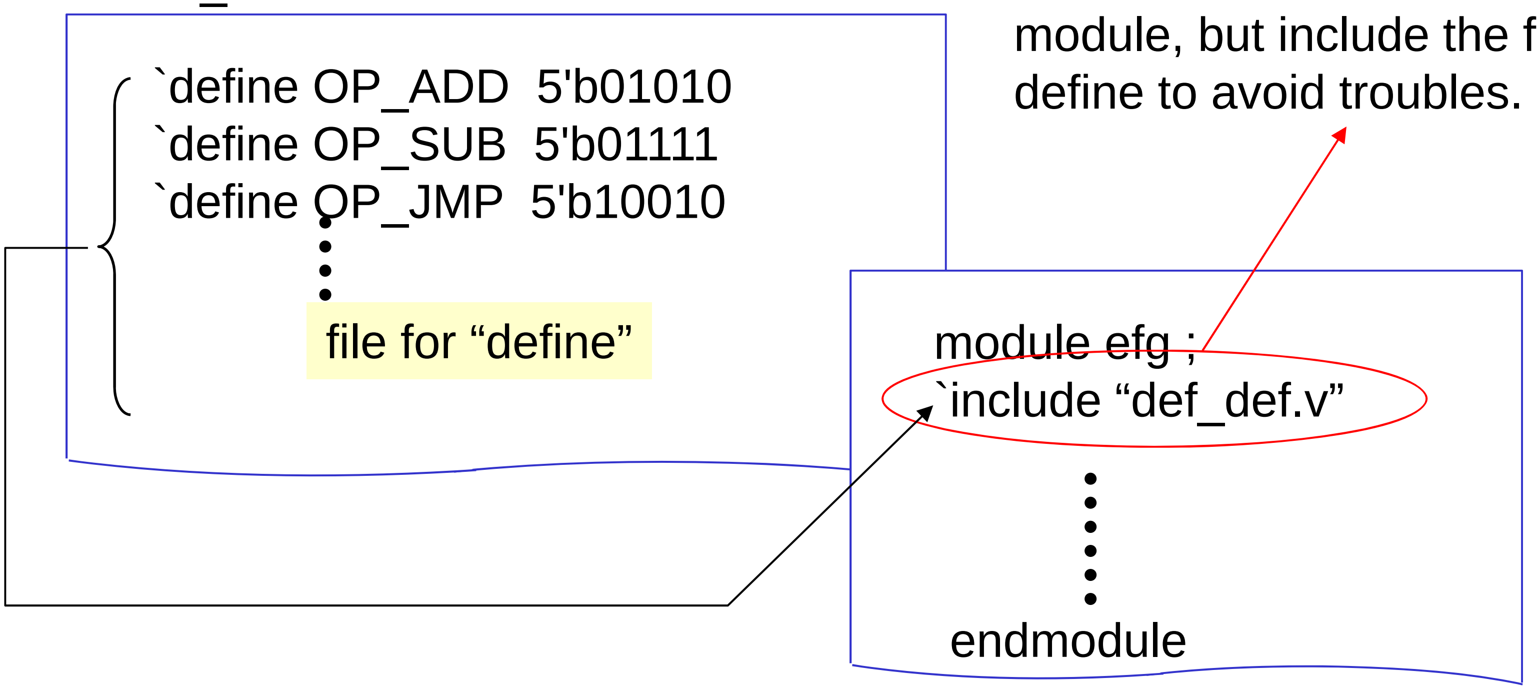
ST_UP is 2'b10 in this range.

```
module abc ;
`define ST_UP  2'b01
        •
        •
        •
        •
        •
        •
 endmodule
```

```
module efg ;
        •
        •
        •
        •
        •
        •
 `define ST_UP 2'b10

  endmodule
```

It is not recommended to use “define” in a way that definition is changed in the middle of a module.

When using “define”, it is recommended to create a file for define as shown below.



def_def.v

```
`define OP_ADD  5'b01010
`define OP_SUB  5'b01111
`define OP_JMP  5'b10010
      :
```

```
module efg ;
`include “def_def.v”
      :
endmodule
```

Do not declare define in each module, but include the file for define to avoid troubles.

In this module, OP_ADD is replaced by 5'b01010, and OP_SUB is replaced by 5'b01111.

3.8.3 Summary of parameter and define

"parameter and define" summary

	parameter	define
function	Hold constant value as 32-bit data unless bit size specified.	Replace character strings.
valid range	Valid only in the module declared.	From the code line declared to the line declared differently. It can go beyond module boundary.
usage	Bit size, Bus width, delay time, cycle time, memory address, bus address, etc.	Specific bit pattern, character strings, non-numerical value, etc.

There is no definite restriction about the usage of parameter and define.
Follow the rule which may be introduced for each project.

In any case, avoid writing 0 and 1 bit pattern in RTL code, use parameter or define and improve readability and maintainability of the code.

3.9. Structured procedure

4 structured procedures, function, task, initial construct, and always construct, are available in Verilog.
In structured procedures, we can use various control statements including timing control such as wait and delay.

3.9.1 Block statement

In structured procedure, if there is more than one statement to execute, they must be grouped in one block.
"begin end" and "fork join" are available for this purpose. Use "begin end" for logic which must be synthesized.

```
function abc ;
input a, b, c ;
  // reg w ;
    abc = a & ( b | c ) :
endfunction
```

⇔

```
function abc ;
input a, b, c ;
   reg w ;
   begin
     w = b | c ;
     abc = a & w ;
   end
endfunction
```

Do not use fork-join block, unless it makes the code extremely simple.

Block statement

block	Sequential block	Parallel block

syntax	begin : block_name
	statements;
	end
		fork : block_name
		statements;
		join

execution	Statements between begin and end are executed in sequence, one after another.	Statements between fork and join are executed concurrently. *1
delay	Delay values are treated relative to the previous statement.	Delay values are treated relative to the simulation time entering the block.
control	Exit the block after the last sentence executed.	Exit the block when the last time-ordered sentence executed.

*1: Delay control can be used to provide time-ordering for assignments.

Block_name is optional. When no name given, the syntax on the right shall be used.

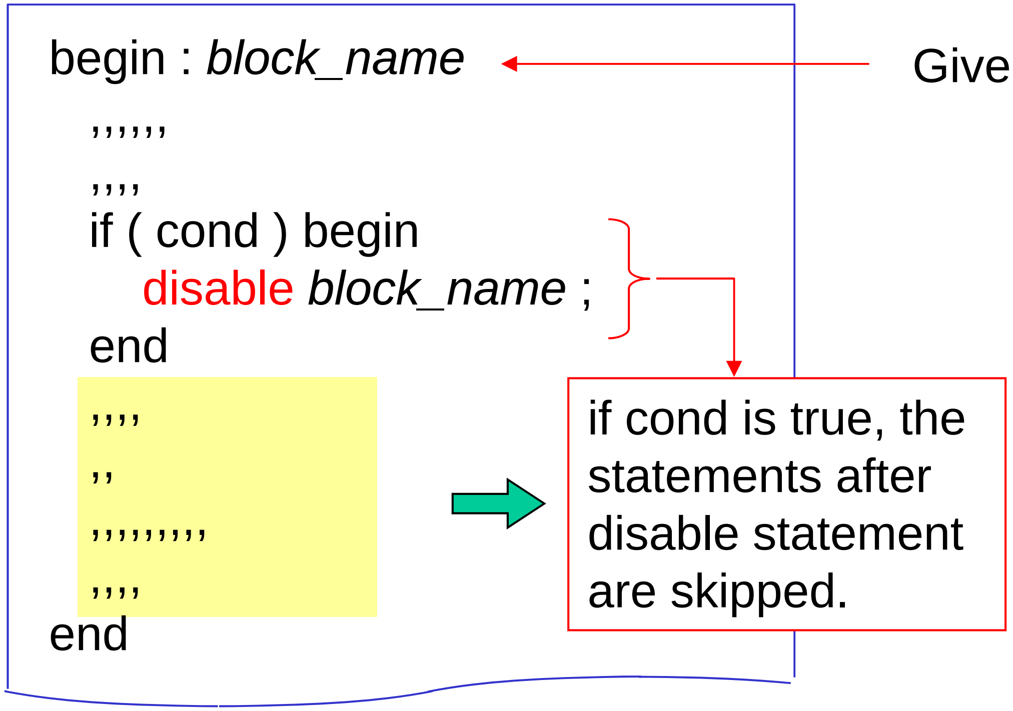
```
begin
   statements;
end
```

```
fork
   statements;
join
```

(1) Named block

A block with name is called named block. And a disable statement gives a mechanism for breaking from a loop statement, or skipping statements in order to continue with another iteration of a looping statement.
Disable key word is not synthesizable.

Disabling named block (1)

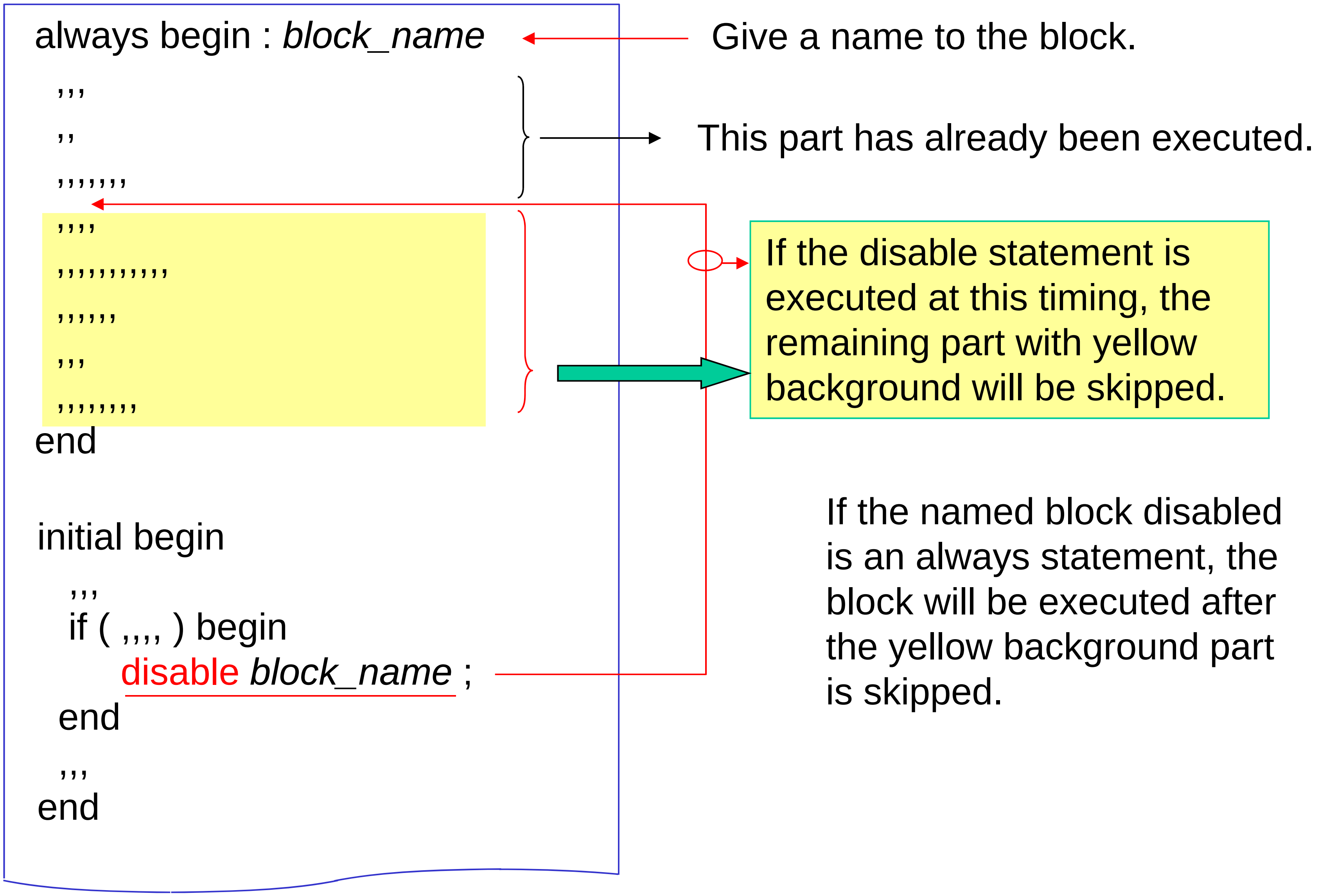


```
begin : block_name
   ,,,,,,
   ,,,,
   if ( cond ) begin
       disable block_name ;
   end
   ,,,,
   ,,
   ,,,,,,,,,
   ,,,,
end
```

Give a name to the block.

if cond is true, the statements after disable statement are skipped.

Disabling named block (2)



```
always begin : block_name
  ,,,
  ,,
  ,,,,,,,
  ,,,,
  ,,,,,,,,,,
  ,,,,,,
  ,,,
  ,,,,,,,,
end

initial begin
   ,,,
   if ( ,,,, ) begin
        disable block_name ;
   end
   ,,,
end
```

Give a name to the block.

This part has already been executed.

If the disable statement is executed at this timing, the remaining part with yellow background will be skipped.

If the named block disabled is an always statement, the block will be executed after the yellow background part is skipped.

(2) Using begin end for a single statement

Throughout this text material, to show as many logic as possible in one page, “begin end” is sometimes omitted when it is allowed to omit it.
However, in RTL coding in actual daily job, do not omit “begin end”.

This description is allowed,
but not recommended.

```
if ( a ) y = b ;
else   y = c ;
```

△ ✗

```
case (a)
 AA : y = b ;
 BB : y = c ;
       :
```

Not recommended

⇒

Using “begin end” is recommended,
even if it is not needed.

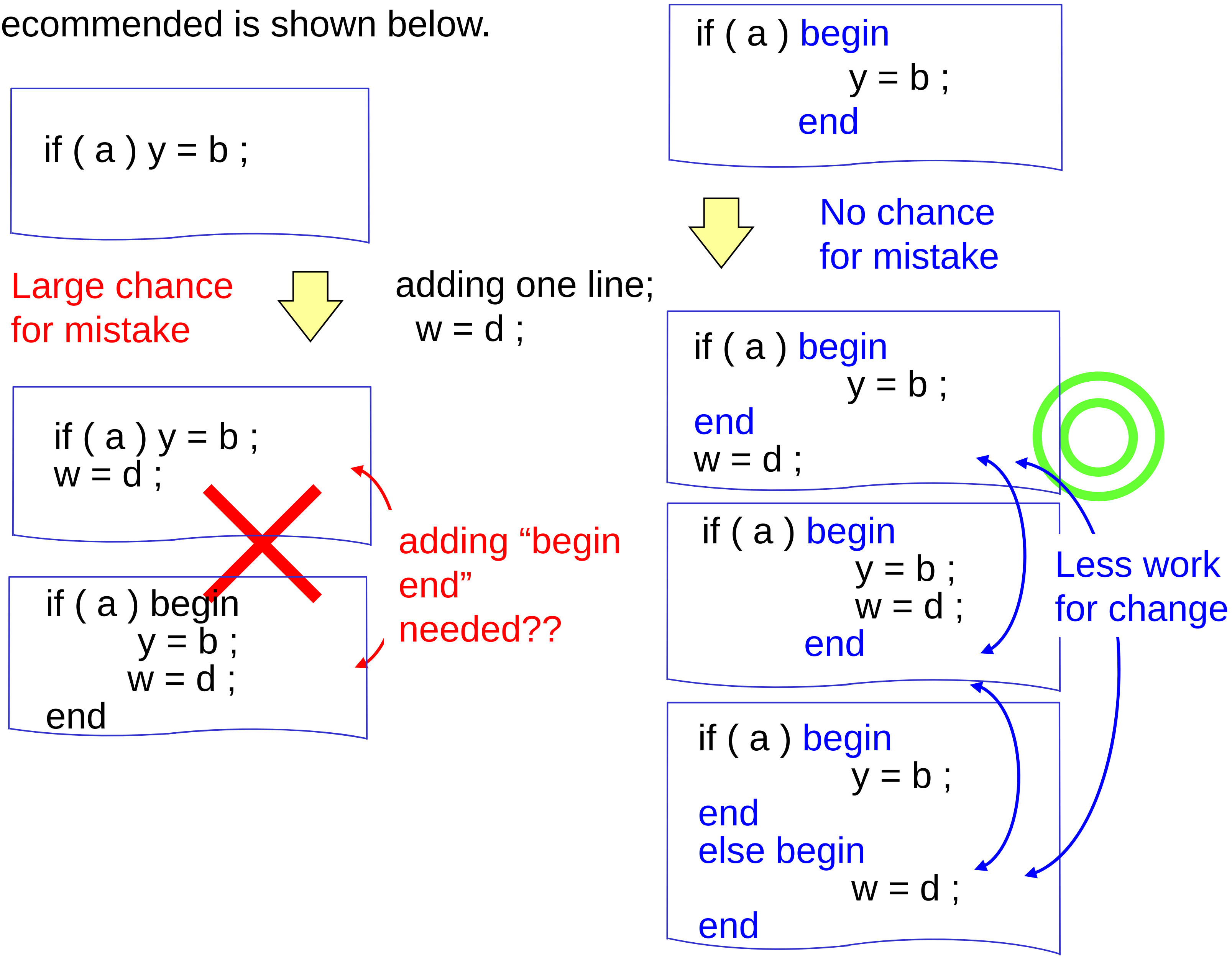
```
if ( a ) begin
           y = b ;
         end
else   begin
           y = c ;
         end
```

◎

```
case (a)
 AA : begin
           y = b ;
         end
 BB :   begin
           y = c ;
         end
```

Recommended style

The reason why using “begin end” is recommended is shown below.



```
if ( a ) y = b ;
```

Large chance for mistake

adding one line;
w = d ;

```
if ( a ) y = b ;
w = d ;
```

adding “begin end” needed??

```
if ( a ) begin
        y = b ;
        w = d ;
end
```

```
if ( a ) begin
        y = b ;
    end
```

No chance for mistake

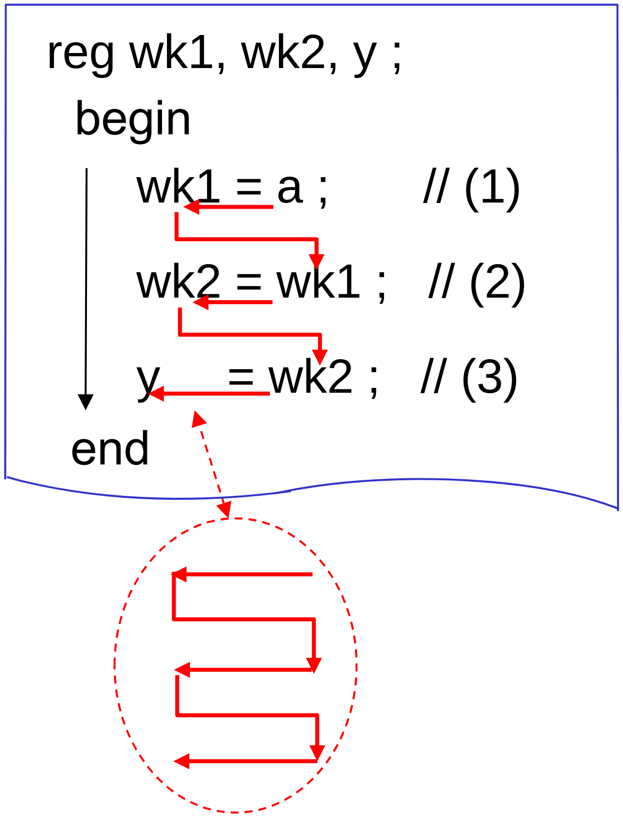
```
if ( a ) begin
        y = b ;
end
w = d ;
```

```
if ( a ) begin
        y = b ;
        w = d ;
    end
```

```
if ( a ) begin
        y = b ;
end
else begin
        w = d ;
end
```

Less work for change

(3) Blocking procedural assignment in a sequential block

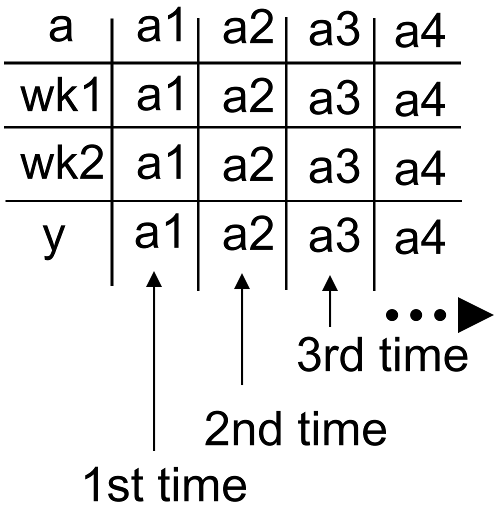


```
reg wk1, wk2, y ;
 begin
    wk1 = a ;       // (1)
    wk2 = wk1 ;   // (2)
    y     = wk2 ;   // (3)
 end
```

Each operation is done serially because it blocks the execution of the following statement.

If blocking procedural assignments are used in a sequential block as shown on the left, execution of statement (2) is blocked by statement (1). Therefore, in simulation, wk1 is given the value of a, and then wk1's value which is equal to the value of a, is assigned to wk2 by (2).

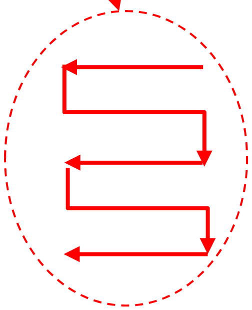
If the sequential block is executed for the 1st time with a's value a1, the result must be as shown below.



a	a1	a2	a3	a4
wk1	a1	a2	a3	a4
wk2	a1	a2	a3	a4
y	a1	a2	a3	a4

If the order of statements is changed, the simulation result will be very different.

```
reg wk1, wk2, y ;
 begin
    y    = wk2 ;   // (3)

    wk2 = wk1 ;   // (2)

    wk1 = a ;       // (1)
 end
```



Each operation is done in serial because it blocks the execution of the following statement.

When executing (3), wk2 is not given any value yet, therefore its value is initial (x). So y is given the value x by (3) if the block is executed for the 1st time.

If the sequential block is executed for the 1st time with a's value a1, the result must be as shown below.

a	a1	a2	a3	a4
wk1	a1	a2	a3	a4
wk2	x	a1	a2	a3
y	x	x	a1	a2

1st time

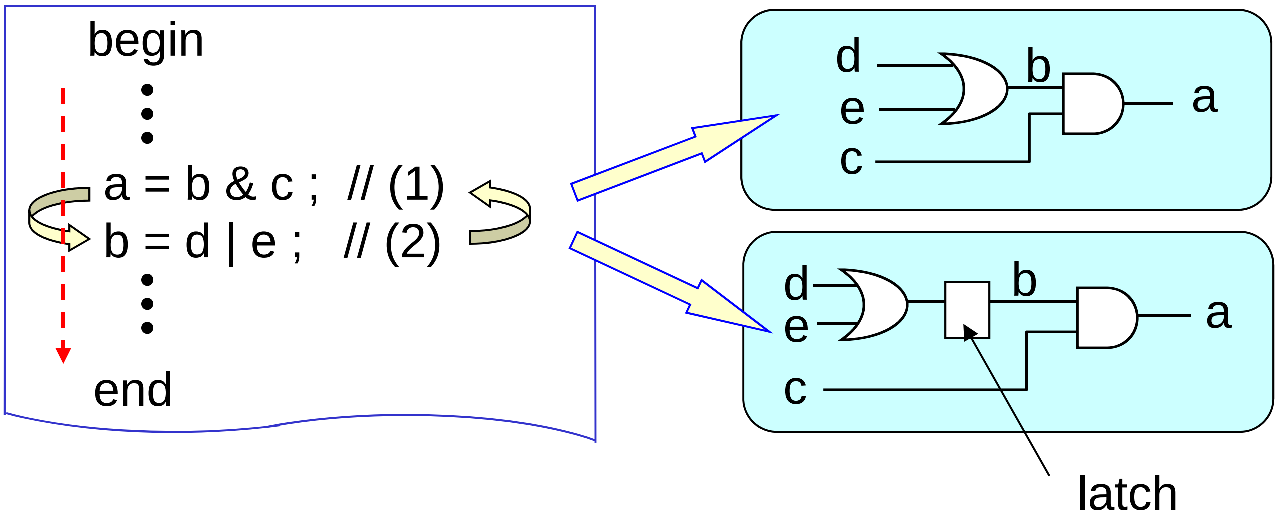
2nd time

3rd time • • • • ▶

wk1 keeps the value a1 until the next assignment occurs. Therefore, when the block is executed for the 2nd time, wk2 will be given the value a1 by (2).

If the sequential block shown below is executed the results will be different depending on the order of the two statements (1) and (2), because of the change of c, d or e.

If (1) is written before (2), the simulator will place a latch on signal line b because b must keep the previous value until the new assignment by (2) occurs.
If (1) is written after (2), the simulator will not place a latch because (2) can use the newly updated value of b always.

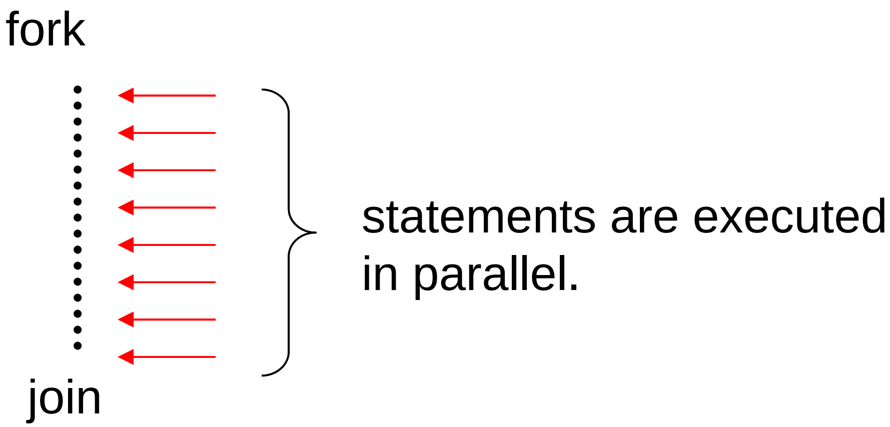


```
begin
   :
 a = b & c ;  // (1)
 b = d | e ;  // (2)
   :
end
```

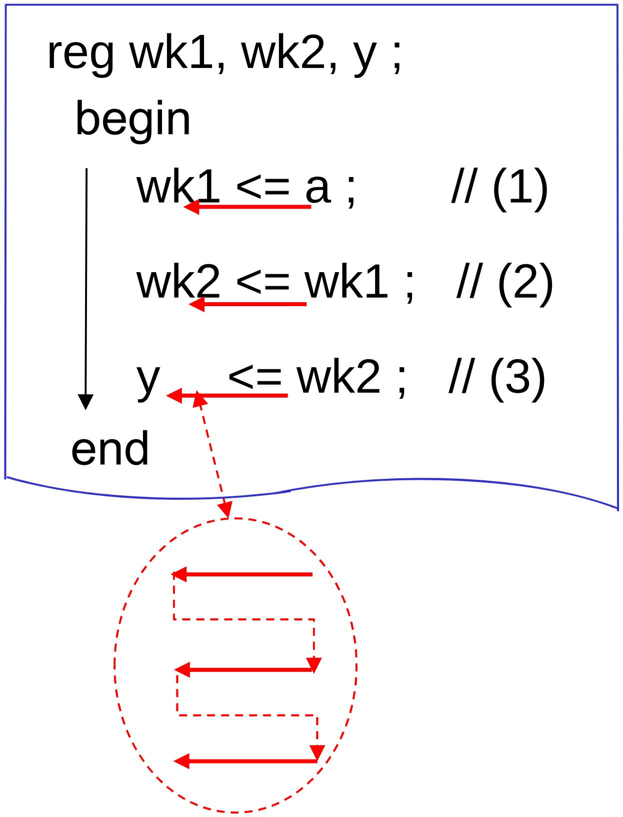
As shown on the previous page, you must be careful about the order of sentences. However, sequential execution caused by blocking procedural assignment is very similar to software program execution. Therefore you can apply your common sense of programming C code.

note on fork join

In case of fork join construct, blocking procedural assignment statement shall not prevent the execution of statements that follow it.



(4) Nonblocking procedural assignment in sequential block



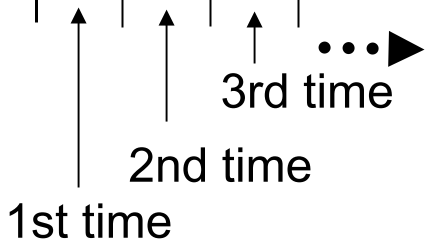
```
reg wk1, wk2, y ;
  begin
      wk1 <= a ;        // (1)
      wk2 <= wk1 ;   // (2)
      y    <= wk2 ;   // (3)
  end
```

Each operation is done in parallel because it does not block the execution of the following statement.

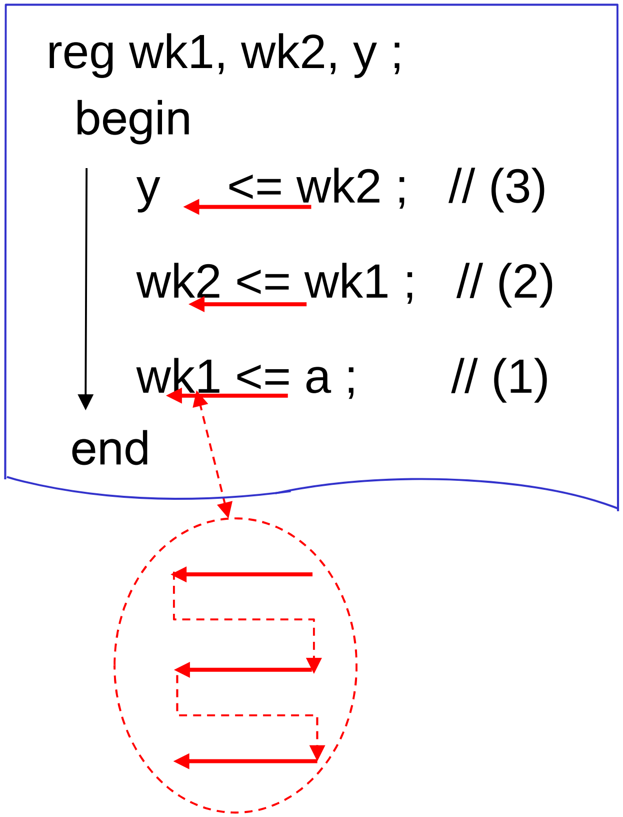
If nonblocking procedural assignments are used in a sequential block as shown on the left, when wk1 is evaluated in (2), its value is unknown (x) because (1) does not block the execution of (2). Therefore wk2 is given the value of wk1 which is initial (unknown). Thus y is given the value x.

If the sequential block is executed for the first time with a's value a1, the result must be as shown below.

a	a1	a2	a3	a4
wk1	a1	a2	a3	a4
wk2	x	a1	a2	a3
y	x	x	a1	a2



If the order of statements is changed, the simulation result is the same.

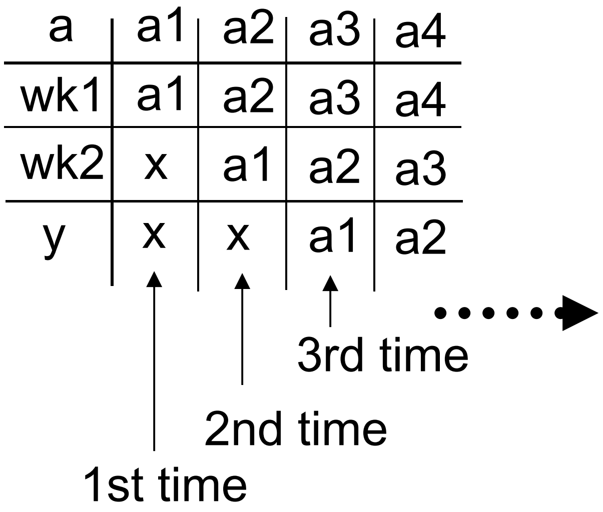


```
reg wk1, wk2, y ;
  begin
     y   <= wk2 ;   // (3)

     wk2 <= wk1 ;   // (2)

     wk1 <= a ;     // (1)
  end
```

Each operation is done in parallel because it does not block the execution of the following statement.

Because nonblocking procedural assignments are executed in parallel, changing the order has no effect over the simulation

If the sequential block is executed for the first time with a's value a1, the result must be as shown below.



a	a1	a2	a3	a4
wk1	a1	a2	a3	a4
wk2	x	a1	a2	a3
y	x	x	a1	a2

Where to use blocking and nonblocking procedural assignments??

In general, Use blocking procedural assignment only. (Do not use nonblocking procedural assignment.)

Use nonblocking procedural assignment only in always construct for flip-flops and latches.

3.9.2 Control statement

In structured procedures, control statements can be used.

control statement

	statement	note
conditional statement	if-else statement	if else, if else if structure.
case statement	case statement	—
	casex statement	Treat x and z as do-not-care
	casez statement	Treat z as do-not-care
loop statement	for statement	Repeat while condition holds true.
	while statement	Execute a statement while an expression is false.
	repeat statement	Execute a statement a fixed number of times.
	forever statement	Continuously execute a statement.

(1) If-else, if-else-if construct

syntax

```
if (Boolean expression)
        sentence1 ;
else sentence2 ;
```

```
if (Boolean expression1)
            sentence1 ;
else if (Boolean expression2)
            sentence2 ;
else        sentence3 ;
```

sentence1 is executed if the expression is true, if not, *sentence2* is executed.

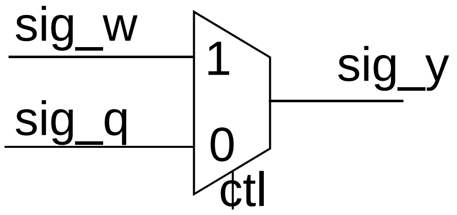
sentence1 is executed if the expression1 is true, if not, expression2 is evaluated and if it is true *sentence2* is executed, if not sentence3 is executed.

If the expression is evaluated to be ambiguous because of x or z, then the control flow takes the false path. ("Pessimistic if")

Difference between if-else and conditional operator

```
assign sig_y = ( ctl == 1'b1 )? sig_w : sig_q ;
```

```
if ( ctl == 1'b1 ) begin
   sig_y = sig_w ;
end
else begin
   sig_y = sig_q ;
end
```



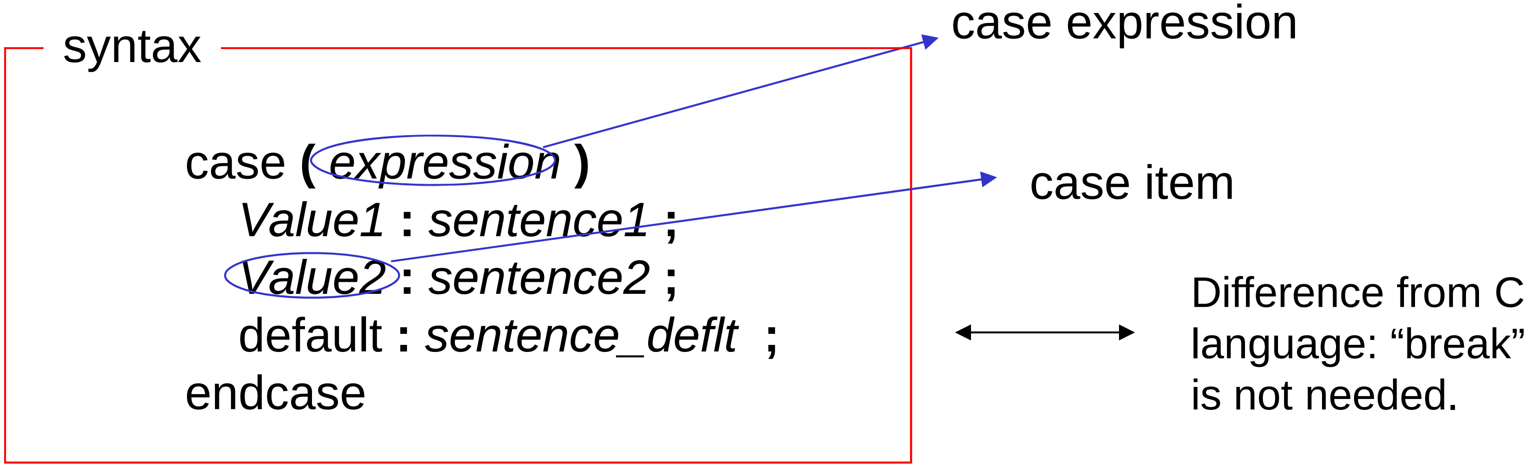
expression \ value of ctl	1'b1	1\`b0	1\`bx

if (ctl == 1'b1) sig_y = 3'b110 ;
else sig_y = 3'b100 ;
	3'b110	3'b100	3'b100

sig_y = (ctl == 1'b1)? 3'b110 : 3'b100 ;	3'b110	3'b100	3'b1x0

It is recommended to use conditional operator rather than using if-else statement wherever possible. However, do not nest conditional operator because it makes the code hard to read. In such cases, use if-else-if construct.

(2) Case statement



syntax

```
case ( expression )
    Value1 : sentence1 ;
    Value2 : sentence2 ;
    default : sentence_deflt  ;
endcase
```

case expression

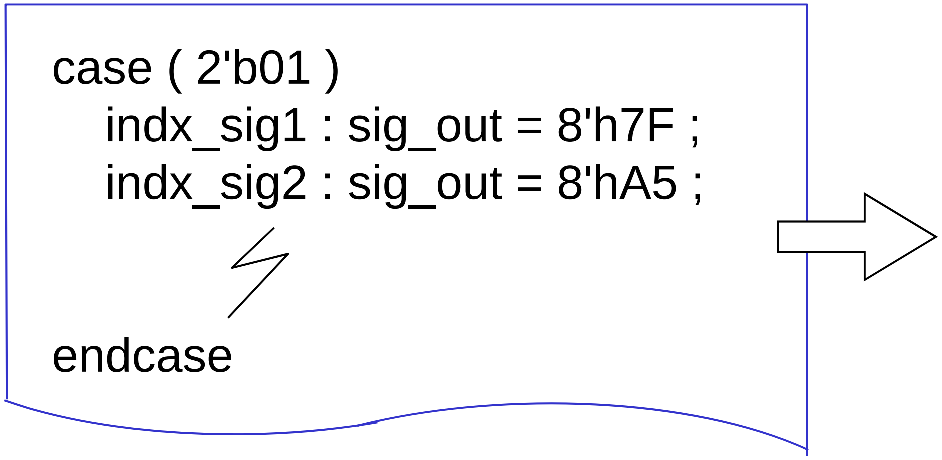
case item

Difference from C language: “break” is not needed.

Case expression is compared with first case item, value1, and sentence1 is executed if they are equal. If not it is compared with second case item, and sentence2 is executed if they are equal. If not,,,,, and so on. If all values do no match with case expression, then sentence_deflt is executed.

Always write “default” part, to detect possible bugs and to avoid creating latches.

Case expression and case items are equivalent. Case statement just compares case expression and case items using case equality operator. Case expression can be a constant as shown below.



```
case ( 2'b01 )
    indx_sig1 : sig_out = 8'h7F ;
    indx_sig2 : sig_out = 8'hA5 ;

endcase
```

This case statement assigns 8'h7F to sig_out if sig_sig1 = 2'b01, or 8'hA5 if indx_sig2 = 2'b01.

Check what happens if both indx_sig1 and indx_sig2 are equal to 2'b01 at the same time.

coverage of case items

When case items do not cover all the possible cases of case expression, latching occurs as shown below.
This will result in mismatch between RTL simulation and gate level simulation or unexpected latches are created by the synthesis tool.

```
begin
   case (d_code_in )
       2'b00 : dec2to4 =  4'h1  ;
       2'b01 : dec2to4 =  4'h2  ;
       2'b10 : dec2to4 =  4'h4  ;
   endcase
end
```

In the example on the left, case items do not cover all the possible cases.
2'b11 case is missing.

Therefore, if d_code_in is 2'b11, the simulator can not find what value to generate, therefore, it outputs the previous value, this is called latching.
If synthesized, this code may create an unexpected latch.

coverage of case items (continued)

Even if you know that theoretically 2'b11 case will never happen, an EDA tool can not know it.

```
begin
   case (d_code_in )
       2'b00 : dec2to4 =  4'h1  ;
       2'b01 : dec2to4 =  4'h2  ;
       2'b10 : dec2to4 =  4'h4  ;
      default : dec2to4 = 4'hx ;
   endcase
end
```

You must insert a default case to notify the tool that 2'b11 case never happens so that you don't care about 2'b11 case.

default : dec2to4 = 4'hx ;

By inserting the default case as above;

(1) the synthesis tool will not create a latch because it treats default assigning x as don't care,

(2) if d_code_in becomes 2'b11 in simulation, then dec2to4 becomes 4'hx. Therefore we can easily detect something unexpected happened. This is a conventional debugging method.

coverage of case items (continued)

The debugging technique assigning x value in default case is applicable even if case items cover all the possible cases.

```
begin
   case (d_code_in )
       2'b00 : dec2to4 =  4'h1  ;
       2'b01 : dec2to4 =  4'h2  ;
       2'b10 : dec2to4 =  4'h4  ;
       2'b11 : dec2to4 =  4'h8  ;
      default : dec2to4 = 4'hx ;
   endcase
end
```

This code can output 4'hx if d_code_in becomes other than 2'b00, 2'b01, 2'b10, and 2'b11 by some logic error.

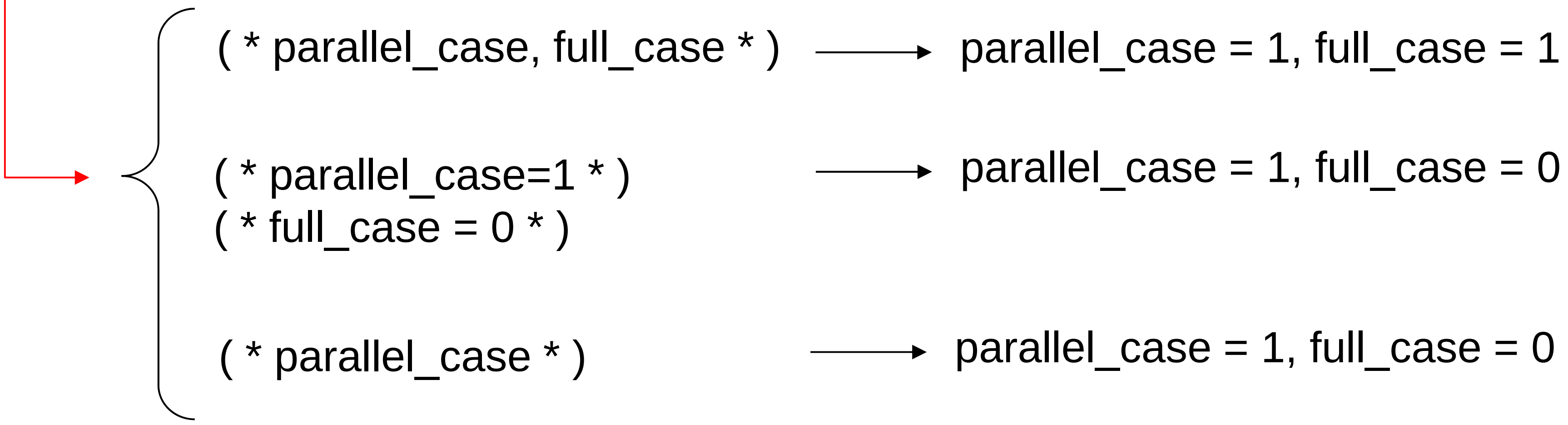
The code on the left can detect a logic error if such an error places 2'b0x or 2'bxx, etc. on d_code_in.

In Verilog2001, case attribute is available as below;

```
( * parallel_case = 1, full_case = 1 * )
case ( case_expression )
  case_item1 : ,,,, ;
  case_item2 : ,,,,,,,,,,,, ;
  case_item3 : ,,, ;
  ,,,,,
  ,,
endcase
```

When full_case is declared, the synthesis tool will treat missing case items as don't care and will not generate a latch.

When parallel_case is declared, the synthesis tool will treat case_items as never taking the same value at the same time.



(* parallel_case, full_case *) → parallel_case = 1, full_case = 1

(* parallel_case=1 *)
(* full_case = 0 *) → parallel_case = 1, full_case = 0

(* parallel_case *) → parallel_case = 1, full_case = 0

```
wire [3:0] a, b, c ;
reg q ;

case ( a )
  b :  q = y ;
  c :  q = w ;
endcase
```

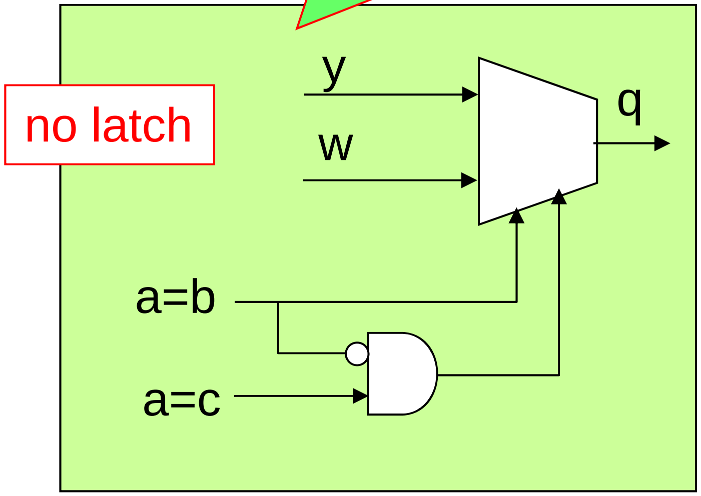
This will create a latch.

```
wire [3:0] a, b, c ;
reg q ;

( * full_case * )
case ( a )
  b :  q = y ;
  c :  q = w ;
endcase
```

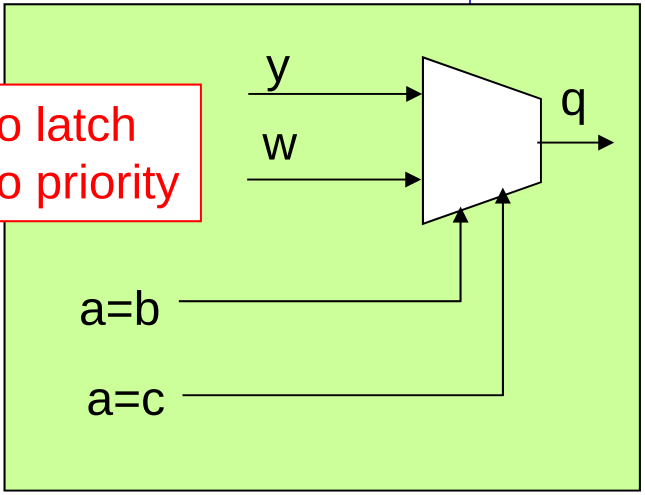
This will not create a latch.

```
wire [3:0] a, b, c ;
reg q ;

( * full_case, parallel_case * )
case ( a )
  b :  q = y ;
  c :  q = w ;
endcase
```

Synthesis



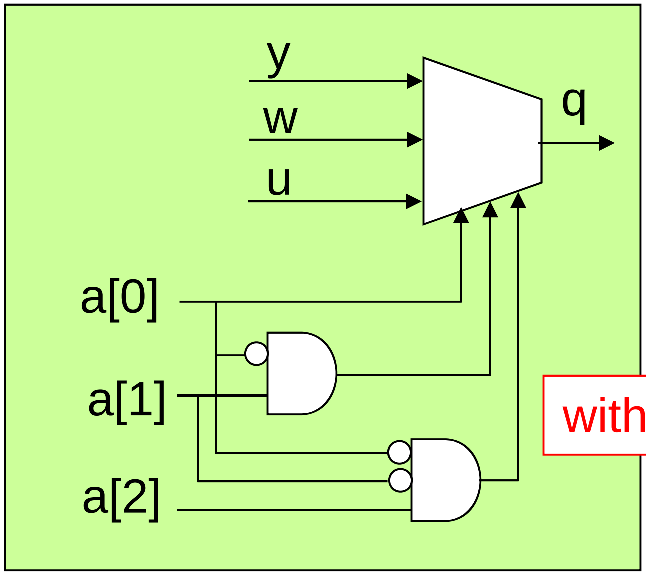
no latch



no latch
no priority

```
wire [3:0] a, b, c ;
reg q ;

( * full_case * )
case ( a )
  4'b0001 :  q = y ;
  4'b0010 :  q = w ;
  4'b0100 :  q = u ;
endcase
```

Synthesis



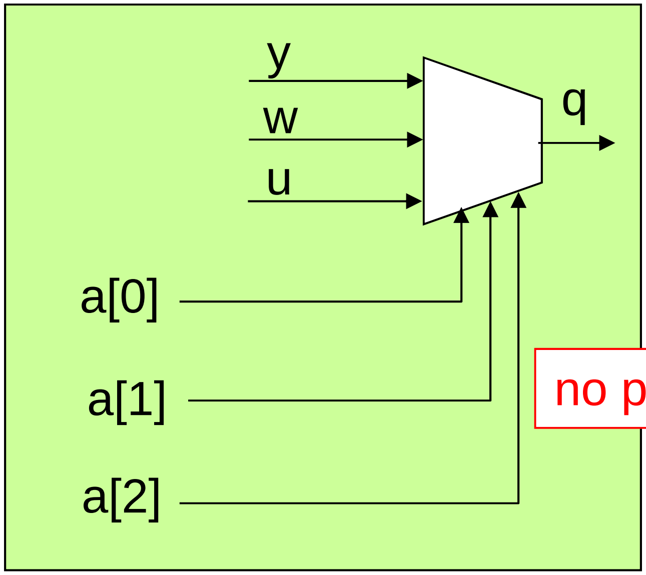
with priority

don't care about a[3]

By full_case attribute, it is declared that a can be 0001, 0010, or 0100, and no other cases will never occur. Therefore, a synthesis tool can know that judging a[0]=1, a[1]=1, and a[2]=1 is enough to know which case occurred.
It can also know that a[3] has no meaning because a[3] is always 0 and it never becomes 1.
There will be a priority logic block because a[0] and a[1] becomes 1 at the same time, q=y must be executed, not q=w.

```
wire [3:0] a, b, c ;
reg q ;

( * full_case, parallel_case * )
case ( a )
  4'b0001 :  q = y ;
  4'b0010 :  q = w ;
  4'b0100 :  q = u ;
endcase
```

Synthesis



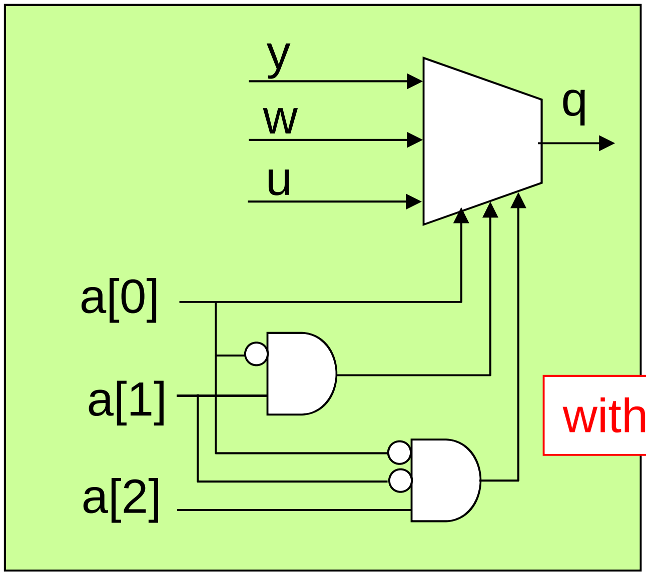
no priority

don't care about a[3]

By parallel_case attribute, it is declared that a will never become 0001, 0010, and 0100 at the same time.
And with full_case additionally, this means that a[0], a[1], and a[2] will never become 1 at the same time.
Therefore no priority logic block is needed. If a[1]=1, it is assured that a[0] is not 1.

```
wire [3:0] a, b, c ;
reg q ;

( * full_case * )
case ( 1'b1 )
  a[0] :  q = y ;
  a[1] :  q = w ;
  a[2] :  q = u ;
endcase
```

Synthesis

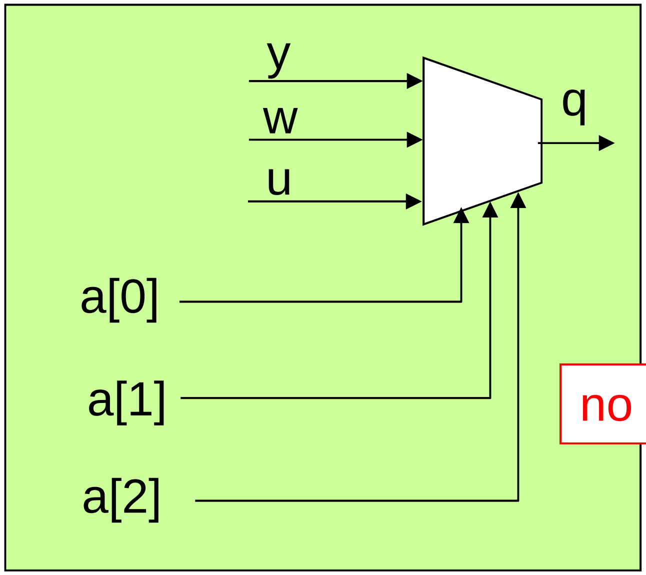


with priority

don't care about a[3]

```
wire [3:0] a, b, c ;
reg q ;

( * full_case, parallel_case * )
case ( 1'b1 )
  a[0] :  q = y ;
  a[1] :  q = w ;
  a[2] :  q = u ;
endcase
```

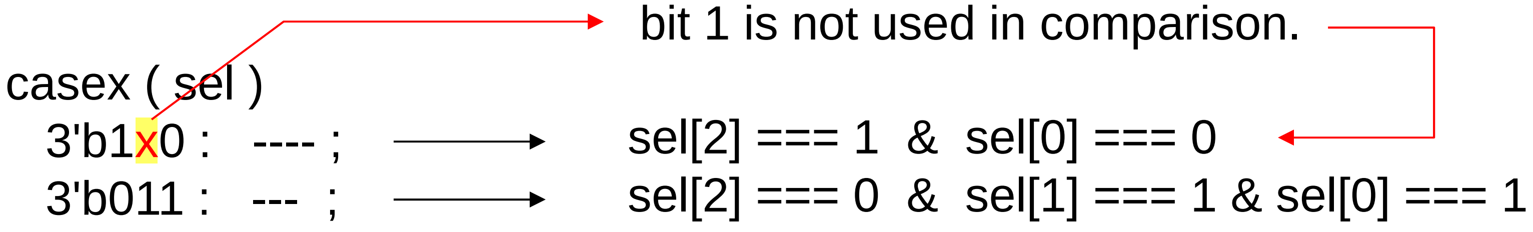
Synthesis



no priority

casex statement

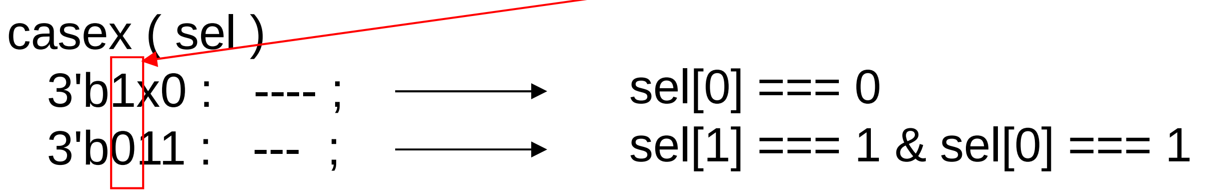
Using casex instead of case, x and z are treated as do-not-care. In comparing the case expression and a case item, if there is x or z bit in either/both of them, then that bit is treated as do-not-care.



bit 1 is not used in comparison.

```
casex ( sel )
  3'b1x0 :   ---- ;   →   sel[2] === 1  &  sel[0] === 0
  3'b011 :   ---  ;   →   sel[2] === 0  &  sel[1] === 1 & sel[0] === 1
```

If bit 2 of sel becomes x, then bit 2 is not compared even if there is no x in bit 2 of case items.



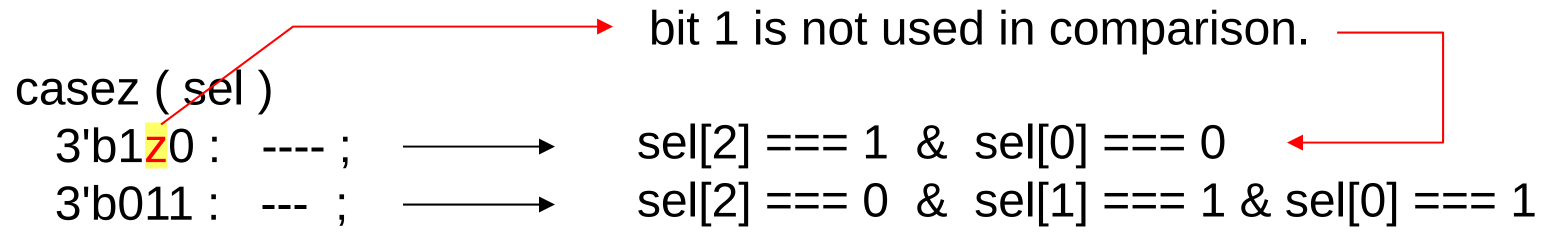
```
casex ( sel )
  3'b1x0 :   ---- ;   →   sel[0] === 0
  3'b011 :   ---  ;   →   sel[1] === 1 & sel[0] === 1
```

This is very dangerous, because variables can very easily become x.

Never use casex .

casez statement

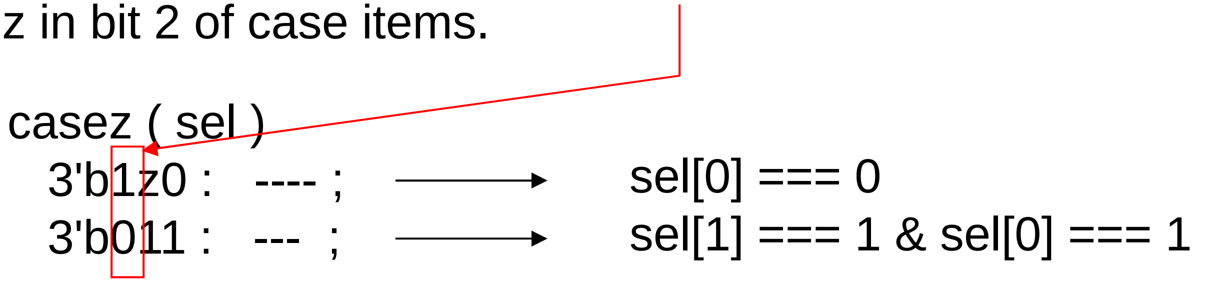
Using casez instead of case, z is treated as do-not-care. In comparing the case expression and a case item, if there is z bit in either/both of them, then that bit is treated as do-not-care.



bit 1 is not used in comparison.

```
casez ( sel )
  3'b1z0 :   ---- ;   →   sel[2] === 1  &  sel[0] === 0
  3'b011 :   ---  ;   →   sel[2] === 0  &  sel[1] === 1 & sel[0] === 1
```

If bit 2 of sel becomes z, then bit 2 is not compared even if there is no z in bit 2 of case items.



```
casez ( sel )
  3'b1z0 :   ---- ;   →   sel[0] === 0
  3'b011 :   ---  ;   →   sel[1] === 1 & sel[0] === 1
```

Casez is also dangerous, however sel has less chance to have z bit compared to having x bit.

Using casez is not recommended. However, if it is necessary to use a wild card, use casez instead of casex.

By using casez, we can implement a truth table with many x (don't care) as below.

a[3]	a[2]	a[1]	a[0]	y
1	x	x	x	q1
0	1	x	x	q2
0	0	x	0	q3
0	0	0	1	q4
0	0	1	1	q5

```
casez ( a[3:0] )
   4'b1??? : y = q1 ;
   4'b01?? : y = q2 ;
   4'b00?0 : y = q3 ;
   4'b0001:  y = q4 ;
   4'b0011:  y = q5 ;
    default :  y = 4'bxxxx ;
endcase
```

? is equal to z in Verilog.

However, for the reason explained in the previous page, avoid using casez.
Use casez only in a situation where, without using casez, code may become very large and difficult to understand.

First choice

Best style !!

◎

```
case ( sel )
   3'b000,
   3'b001,
   3'b100,
   3'b101 :  y = --- ;
   3'b010 :  y = --- ;
   3'b011 :  y = --- ;
   3'b110,
   3'b111 :  y = --- ;
   default :  y = xxx ;
endcase
```

```
case ( sel )
   3'b000, 3'b001, 3'b100, 3'b101 :  y = --- ;
   3'b010 :                         y = --- ;
   3'b011 :                         y = --- ;
   3'b110, 3'b111 :                 y = --- ;
   default :    y = xxx ;
endcase
```

Second choice

○ △

```
casez ( sel )
   3'b?0? :  y = --- ;
   3'b010 :  y = --- ;
   3'b011 :  y = --- ;
   3'b11? :  y = --- ;
   default :  y = xxx ;
endcase
```

Make sure sel never becomes z (?).

Third choice

```
if ( sel[1] == 1'b0 )  y = --- ;
else if ( sel[2] == 1'b0 )  y = --- ;
       else if ( sel[0] == 1'b1) y = --- ;
              else               y = --- ;
```

△

Worst choice

```
casex ( sel )
   3'b?0? :  y = --- ;
   3'b010 :  y = --- ;
   3'b011 :  y = --- ;
   3'b11? :  y = --- ;
   default :  y = xxx ;
endcase
```

Never do this !!

(3) Loop statement

We can use loop statement in verilog RTL programming. There are four loop instructions available. They are forever, repeat, while, and for.

Syntax of loop instructions:

forever *statement* ; *1

⇨ Continuously repeat the *statement* forever.

repeat (expression) *statement* ;*1

⇨ Execute the *statement* a fixed number of times. The number of executions is set by the expression. If the expression evaluates to unknown, high-z, or a zero value, then no statement will be executed

while (expression) *statement* ; *1

⇨ Execute the *statement* while the expression is true. If a while instruction starts with a false value, then no statement will be executed

*1: In many cases these are not synthesizable or there may be heavy description dependency. Therefore avoid using these statements in synthesizable code.

for (assignment; expression; assignment) *statement* ;

Execute the *statement* until the expression becomes false.

At the initial step, the first assignment will be executed.

At the second step, the expression will be evaluated. If the expression is false (an unknown, high-z, or zero), then the for statement will be terminated. Otherwise, the *statement* and second assignment will be executed.

After that, the second step is repeated.

If loop instructions are used in a module which has to be synthesized, care must be taken. You have to be aware what structure will be generated with loop instructions.

You have to be aware that they are different from those in C programming language.

For loop index of for loop, we can use an integer or register. When using a register variable check bit size so that the expression is evaluated correctly.

Example:

```
reg [1:0] sel ;
    :
for(  sel = 0 ; sel <= 3 ; sel = sel + 1 )  begin
    :
    :
end
```

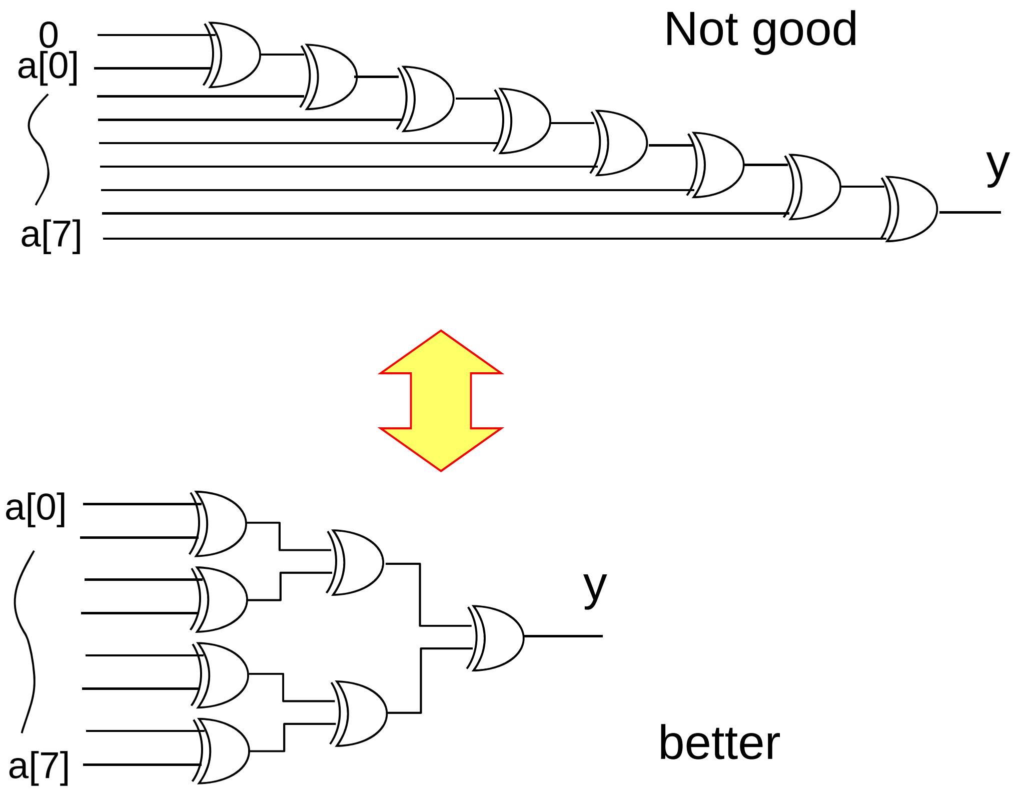
This code will not work because sel has only two bits. It can not be larger than 3 and creates an infinite loop.

```
integer k ;
reg [1:0] sel ;
    :
for(  k = 0 ; k <= 3 ; k = k + 1 )  begin
 sel [1:0] = k ;
    :
end
```

This code will run as intended

In general avoid using “for loop” because for loop sometimes creates an undesirable logic as shown below.

```
integer k;
begin
 y =1’b0;
 for (k=0; k<8; k=k+1)
   y = y^a[k];
end
```



In the future, tools may become more clever to generate the net list shown on the right.

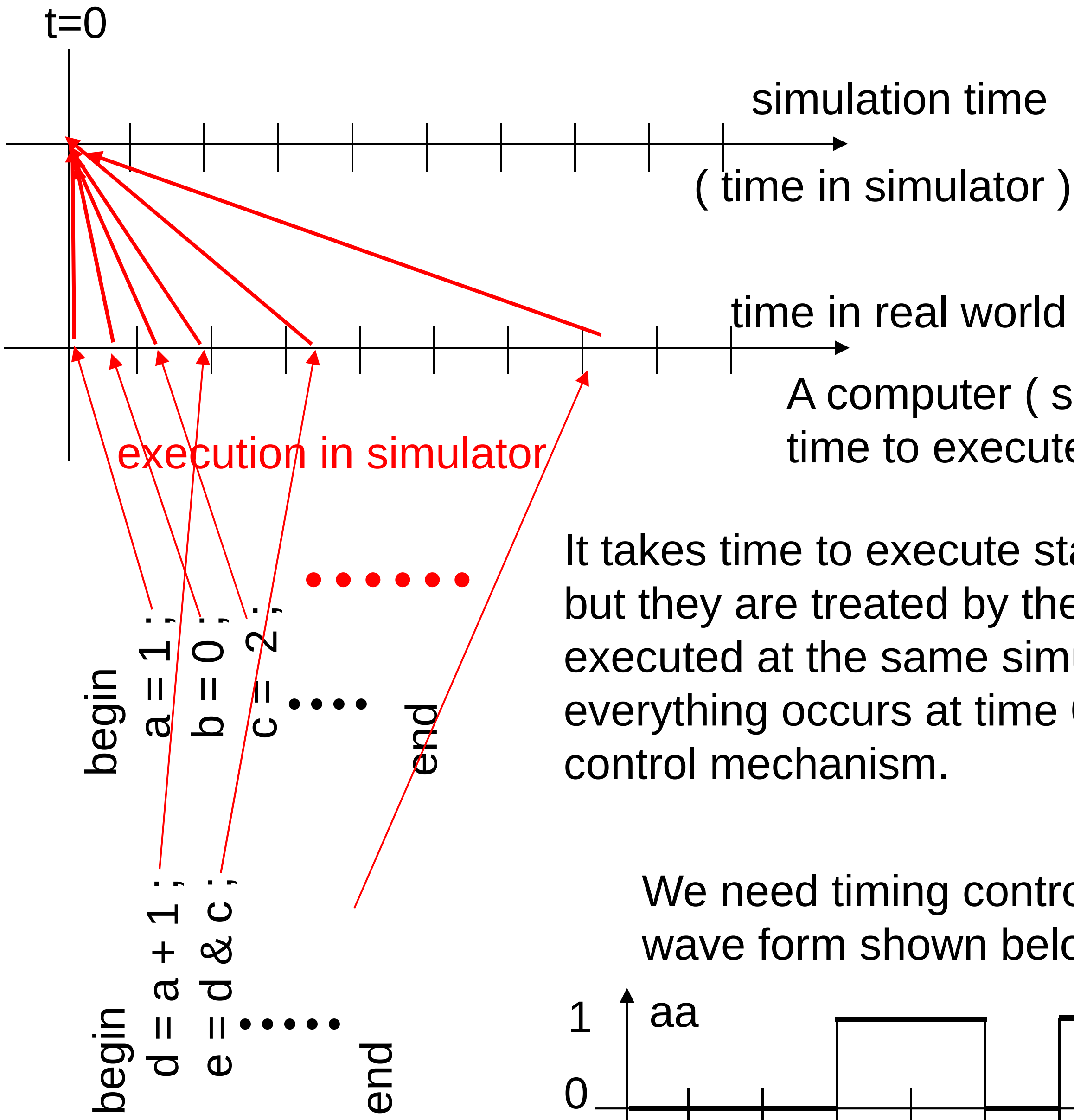
3.9.3 Procedural timing control

Structured procedures are executed at the following timings in simulation. Once it starts, statements in procedures are executed in sequence, in case of begin-end-block, or in parallel, in case of fork-join-block. In microscopic view, statements in parallel block are also executed serially.

execution timing of procedures

procedures	execute timing
function	When the arguments change their values if the function is used in a continuous assignment. When a sentence calling the function is executed if it is used in a procedural assignment.
task	When a task is invoked, i.e. at the time when task invoking statement is executed.
initial construct	When simulation started, at time 0.
always construct	When simulation started, at time 0.

Although, it takes time for the simulator to execute one statement after another, they are treated by the simulator as though executed at the same simulation time.

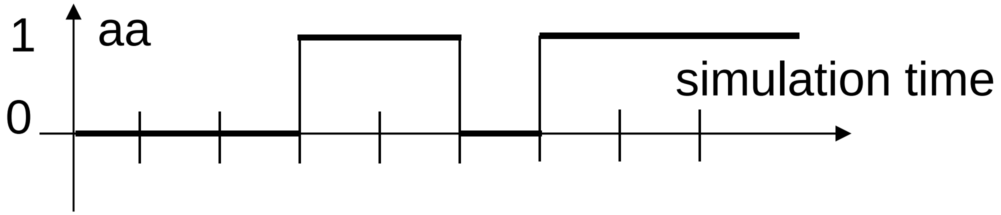


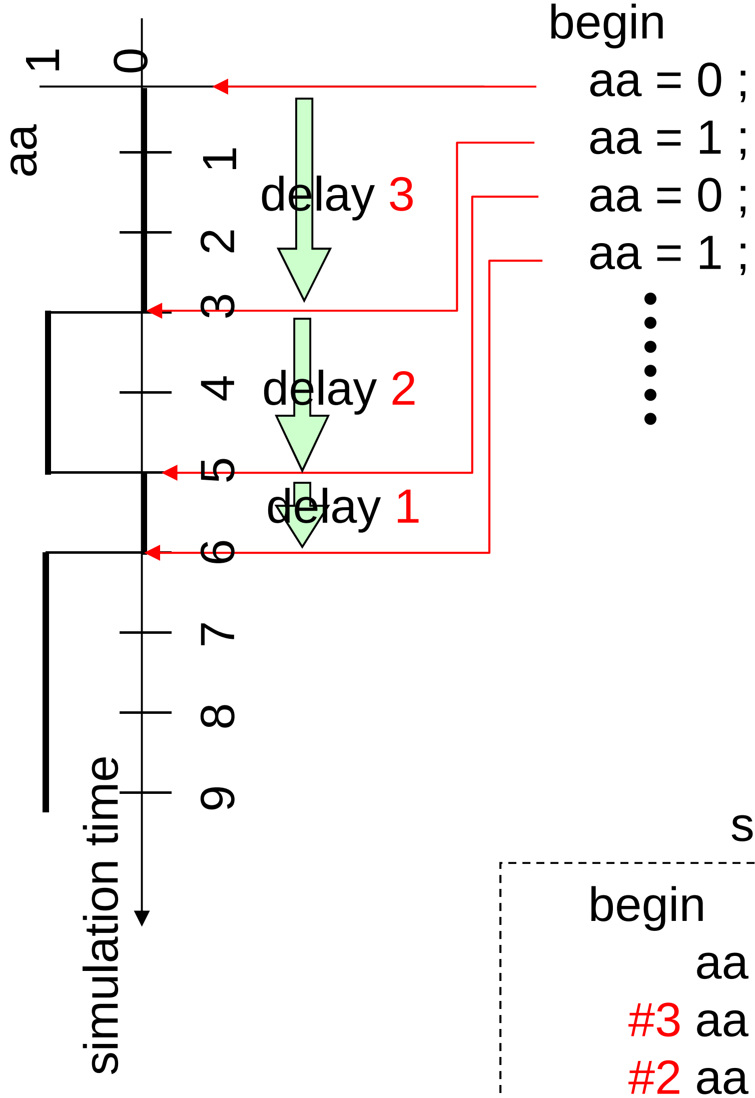
```
begin
  a = 1 ;
  b = 0 ;
  c = 2 ;
  ••••
end

begin
  d = a + 1 ;
  e = d & c ;
  •••••
end
```

A computer (simulator) needs time to execute statements.

It takes time to execute statements in a simulator, but they are treated by the simulator as though executed at the same simulation time, that is, everything occurs at time 0, if there is no timing control mechanism.

We need timing control to enable the wave form shown below.





```
begin
   aa = 0 ;
   aa = 1 ;
   aa = 0 ;
   aa = 1 ;
   ⋮
```

To be able to generate the waveform, the second statement shall be executed at simulation time 3, and the third statement at time 5, and the fourth statement at time 6, and so on.

One way to realize the above is to delay the execution of the second statement 3 time units after the execution of the first statement as shown on the left.

A code to generate the wave form is shown below.

statements with delay specification

```
begin
      aa = 0 ;
   #3 aa = 1 ; // delay 3 after the above line
   #2 aa = 0 ; // delay 2 after the above line
   #1 aa = 1 ; // delay 1 after the above line
```

(1) Delay control

syntax

```
# dly_t  statement ;          // <1>

LHS = # dly_t  expression ; // <2>

wire [15:0] # dly_t  d_bus ; // <3>
```

Intra-assignment timing control

<1> In case of sequential block, *dly_t* time unit after the preceding statement is executed, and in case of parallel block, *dly_t* time unit after the block is entered, the expression is executed.

<2> When control is passed to this statement, the expression is evaluated and the value shall be assigned to LHS after *dly_t* time unit. Equivalent to the following code;

```
    temp = expression ;
#dly_t LHS = temp ;
```

<3> All the signals propagate on d_bus are delayed by *dly_t* time unit.

The unit time can be defined by timescale compiler directive.

Restrictions

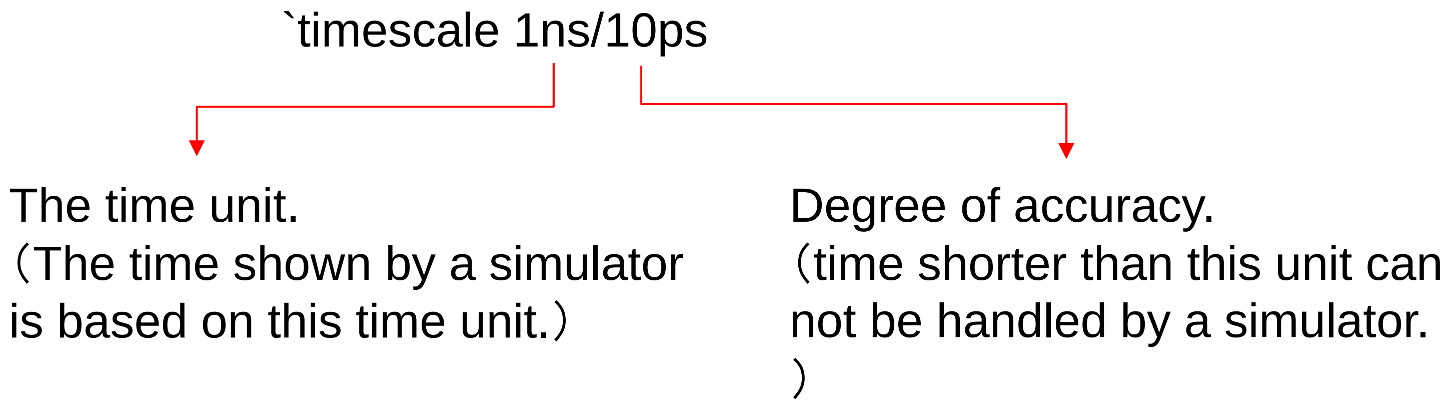
[1] # is not synthesizable.

[2] # will be neglected by a synthesis tool.

[3] # is not allowed in function.

[4] Do not use # in modules that have to be synthesized except for avoiding racing problems.

The `timescale compiler directive specifies the unit of measurement for time and delay values and the degree of accuracy for delays as below.



`timescale 1ns/10ps

The time unit.
（The time shown by a simulator
is based on this time unit.）

Degree of accuracy.
（time shorter than this unit can
not be handled by a simulator.
）

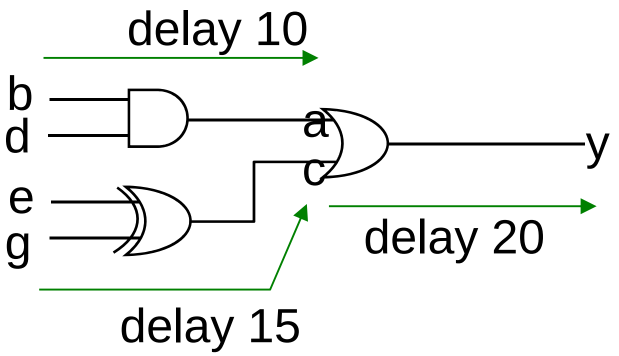
With the specification above, #1 means 1 ns delay and #0.4 means 0.4 ns = 400 ps delay.

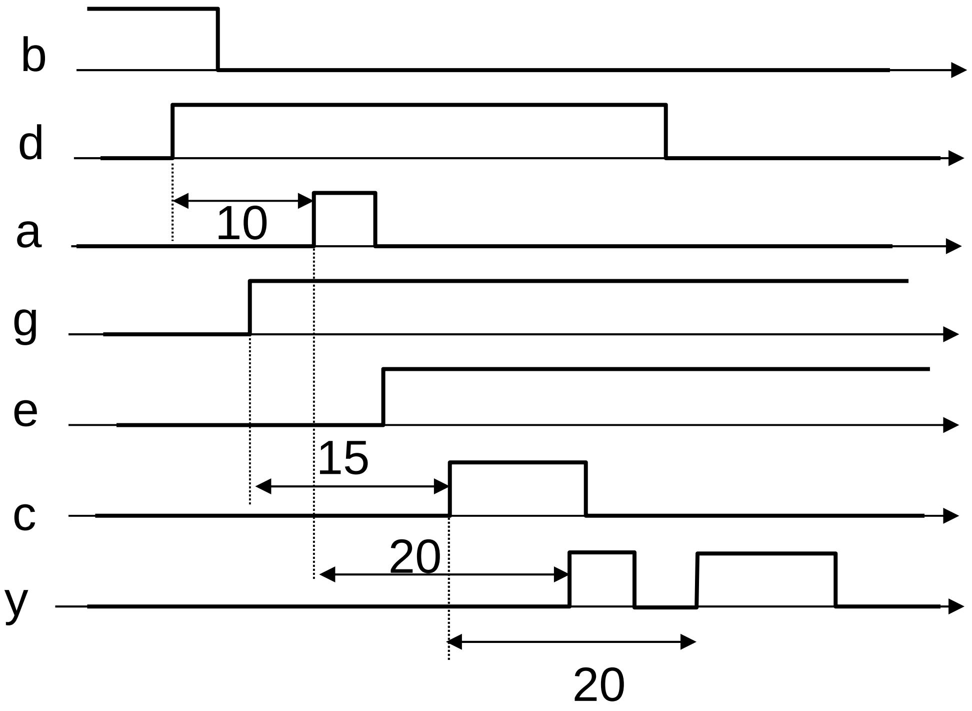
Given the above compiler directive, 10 ps is the smallest time unit a simulator uses for time calculation.

Time specification shorter than the accuracy is treated as zero. For example, with a compiler directive such as `timescale 1ns/100ps, #0.04 is treated as zero delay because 0.04x1ns = 40ps is smaller than the degree of accuracy (100 ps).

Timing analysis by dynamic simulation

```
sig_a = #10  sig_b & sig_d ;
sig_c = #15  sig_e ^  sig_g ;
sig_y = #20  sig_a | sig_c ;
```





Do not use # in synthesizable RTL code.

We do not do dynamic simulation for timing analysis any more.

⇨ Writing delay in RTL code is an old technique to do timing analysis by dynamic RTL simulation.
Only in FF definition, # can be used to avoid racing.

(2) Wait control

By using @ or “wait”, the execution of a procedural statement can be synchronized to an event or each other.

Syntax

```
@ event_name  statement ;           // <1>

LHS = @ event_name  expression ; // <2>

wait ( expression ) statement ;       // <3>
```

Intra-assignment timing control

<1> If the procedural control flow has reached this line, the statement will be executed when the event occurs.

<2> When the procedural control flow reached this line, the expression is evaluated and after the event occurs, it is assigned to LHS.

```
                temp = expression ;
@(event_name) LHS = temp ;
```

<3> When the procedural control flow reached this line, the expression is evaluated and if it is true the statement is executed, if not true, the statement is not executed until the expression becomes true.

Restrictions

[1] event is not memorized, therefore, if event occurs before code line with @ is executed, then @ will miss the event already occurred.

[2] @ is not synthesizable except the one used in the form of "always @".

[3] Do not mix edge trigger and level trigger by using event or operator, it will cause error in synthesis.

[4] @ and wait are not allowed in function.

```
begin
        a= 0 ;
#10 ;
        a = 1 ; // <1>
end

 begin
    @ ( posedge a ) y = 1 ; //<2>
 end
```

If <1> is executed prior to <2>, y=1 will not be executed.
<2> must be executed before <1>.

<2> must be in waiting state for posedge event of a, before <1> is executed.

Typical usage of @ and wait

@ *sig_y*

Wait for the change of the value of sig_y.

@ (posedge *sig_y*)

Wait for the change of the value of sig_y from 0 to other than 0, or from other than 1 to 1.

@ (negedge *sig_y*)

Wait for the change of the value of sig_y from 1 to other than 1, or from other than 0 to 0.

@ *event_identifier*

Wait for the event identified by *event_identifier* triggered.

wait (*expression*)

Wait for the expression to become true. The wait statement shall evaluate a condition, and, if it is false, the procedural statements following the wait statement shall remain blocked until that condition becomes true before continuing.

```
@ ( sig_y )  sig_w = 4'b0101 ;
```

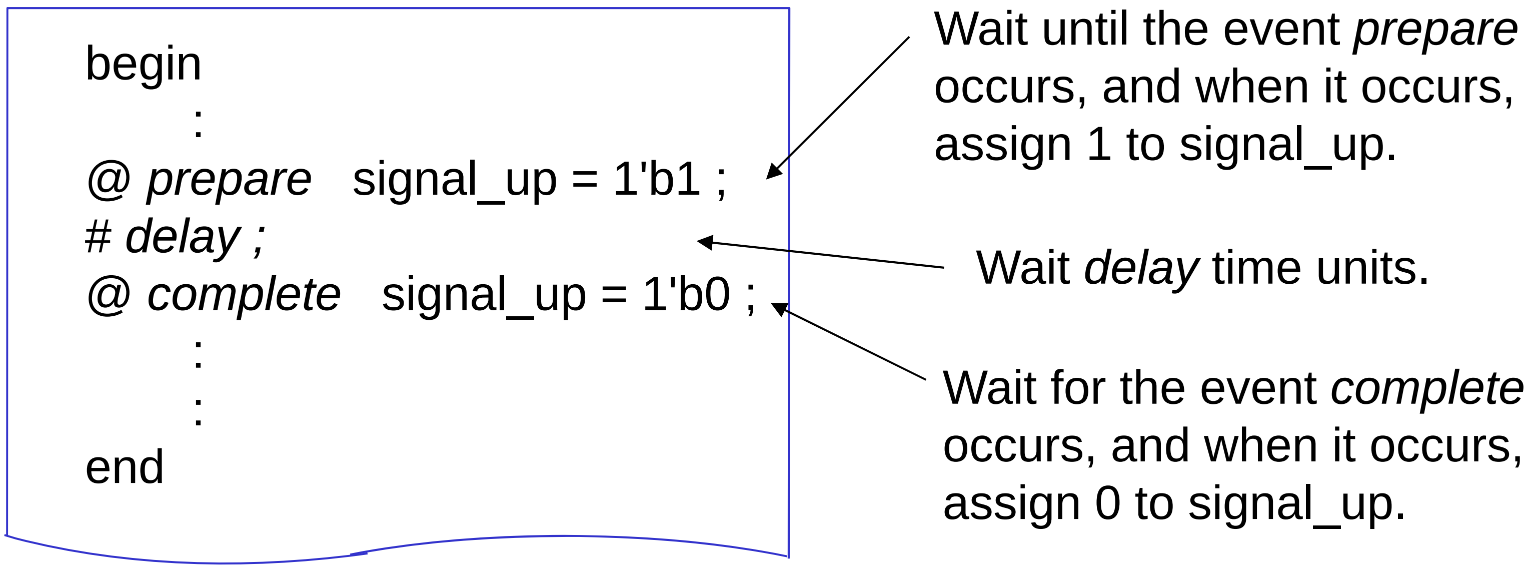
This waits for sig_y's change. Therefore, if sig_y has already changed before this statement is executed, then the following statements will not be executed until sig_y changes its value again.

↕

```
wait ( sig_y )  sig_w = 4'b0101 ;
```

This waits for sig_y becomes 1 (true). Therefore, if sig_y has already changed from 0 to 1 before this statement is executed, then the following statements will be executed.
If sig_y is 1 when this statement is executed, sig_w = 4'b0101 will be executed without wait.

Example_1:



```
begin
      :
@ prepare   signal_up = 1'b1 ;
# delay ;
@ complete   signal_up = 1'b0 ;
      :
      :
end
```

Wait until the event *prepare* occurs, and when it occurs, assign 1 to signal_up.

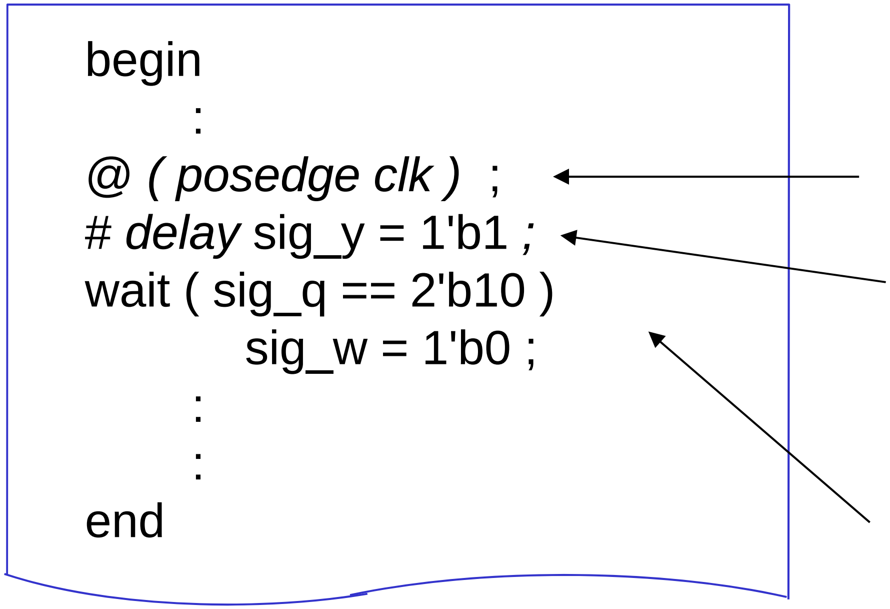
Wait *delay* time units.

Wait for the event *complete* occurs, and when it occurs, assign 0 to signal_up.

```
event prepare, complete ;
begin
      ⋮
  -> prepare ;
      ⋮
  -> complete ;
      ⋮
end
```

The event “*prepare*” and “*complete*” occur when corresponding “->” operator is executed.

Example_2:



```
begin
        :
@ ( posedge clk )  ;
# delay sig_y = 1'b1 ;
wait ( sig_q == 2'b10 )
            sig_w = 1'b0 ;
        :
        :
end
```

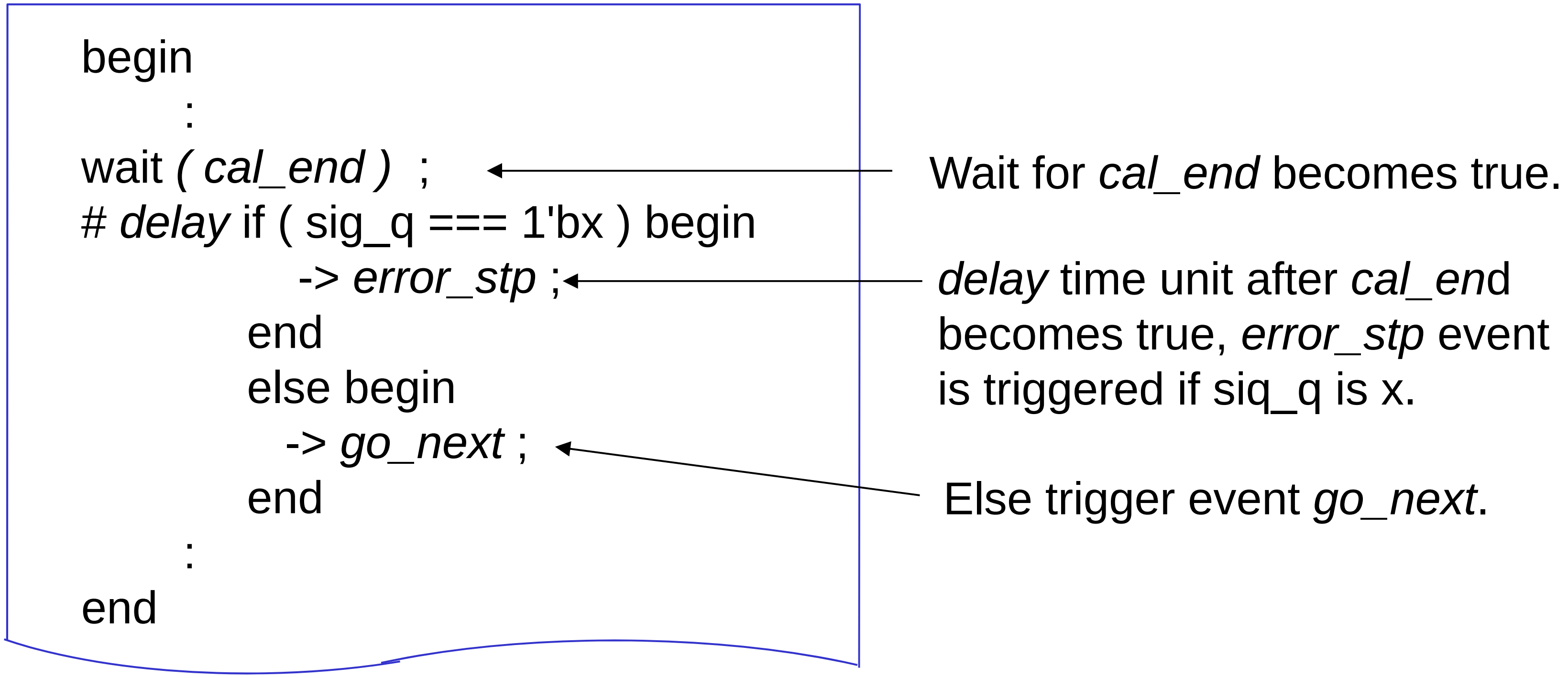
Wait for clock rise.

This sentence is executed delay time unit later, right after clock rise.

Wait for sig_q to become 2'b10. When it becomes 2'b10, sig_w is given the value 1'b0.

Do not write a sophisticated or tricky program using wait and other control statements. Always “simple is best”.

Example_3:



```
begin
        :
wait ( cal_end )  ;
# delay if ( sig_q === 1'bx ) begin
                -> error_stp ;
            end
            else begin
               -> go_next ;
            end
        :
end
```

Wait for *cal_end* becomes true.

delay time unit after *cal_end* becomes true, *error_stp* event is triggered if siq_q is x.

Else trigger event *go_next*.

Do not write a sophisticated or tricky program using wait and other control statements. Always “simple is best”.

Example_4: Displays signals at clock rise time

```
always @ ( posedge clk ) begin

  $strobe(“t=%d, sig_a=%b, sig_b=%b, ,,, ”,
                $stime, sig_a, sig_b,,,  ) ;

end
```

Note; signals in lower hierarchical module can be specified by *instance name. instance name. signal name*.

Q3.9-1; Answer if the following statements are correct or not.

(1) Following two pieces of RTL code behave same.

```
begin
  @ ( a ) ;
  @ ( b ) ;
     y = 1 ;
end
```

⟺

```
begin
  @ ( a or b ) ;
     y = 1 ;
end
```

(2) Following two pieces of RTL code behave same.

```
begin
  wait ( aa == 1 )
     y = 1 ;
end
```

⟺

```
begin
  @ ( aa == 1 )
     y = 1 ;
end
```

Q3.9-1; A sample answer

(1) Following two pieces of RTL code behave same. ⇦ Wrong.

```
begin
  @ ( a ) ;
  @ ( b ) ;
    y = 1 ;
end
```

⇔

```
begin
  @ ( a or b ) ;
    y = 1 ;
end
```

In the one on the left, if event b occurs before event a, y=1 will not be executed.

(2) Following two pieces of RTL code behave same. ⇦ Wrong.

```
begin
  wait ( aa == 1 )
    y = 1 ;
end
```

⇔

```
begin
  @ ( aa == 1 )
    y = 1 ;
end
```

In the one on the right, if aa keep the value 1 and does not change its value, y=1 will not be executed.

Q3.9-2; Answer if the following statements are correct or not.

(1) If the time scale is given by 1ns/1ns, then the following two code behave same.

```
begin
 #1 a=0 ;
 #1.2 a = 1;
 #2.3 a = 0;
 #1.4 a = 1;
end
```

⟺

```
begin
 #1 a=0 ;
 #1 a = 1;
 #2 a = 0;
 #1 a = 1;
end
```

(2) Following two pieces of RTL code behave same for single bit a.

`@ ( posedge a or negedge a )` ⟺ `@ ( a )`

Q3.9-2; A sample answer

(1) If the time scale is given by 1ns/1ns, then the following two codes behave the same. ⇐ Correct.

```
begin
 #1.2 a=0 ;
 #1.3 a = 1;
 #2.3 a = 0;
 #1.4 a = 1;
end
```

⇔

```
begin
 #1 a=0 ;
 #1 a = 1;
 #2 a = 0;
 #1 a = 1;
end
```

Because the degree of accuracy is 1ns, .2 .3 and .4 are neglected. If we set the time scale 1ns/100ps, then they behave differently.

(2) Following two pieces of RTL code behave the same for single bit a. ⇐ Wrong.

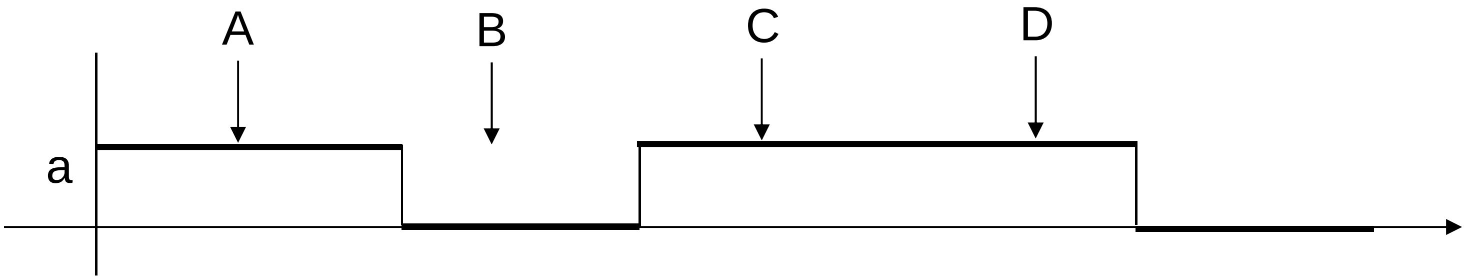
`@ ( posedge a or negedge a )` ⇔ `@ ( a )`

There will be no change direction event if a changes from x to z.
But it shall be a change value event.

Q3.9-3; Answer the following questions.

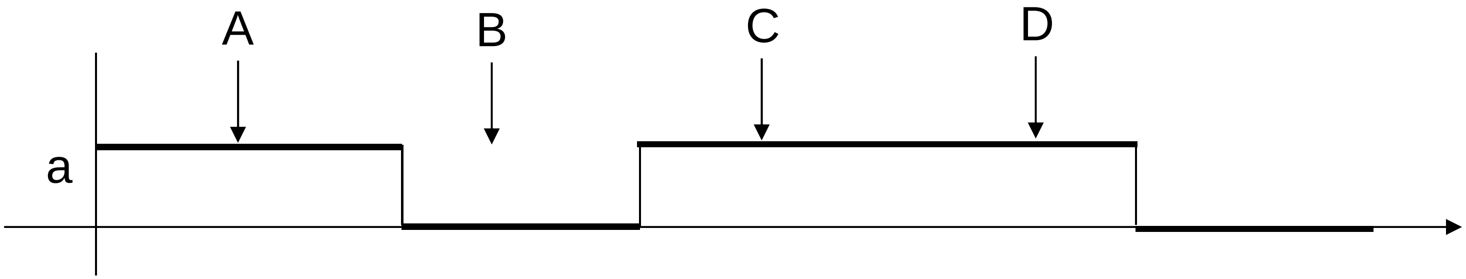
(1) While signal a is changing as shown below, if <1> is executed at the timing shown as A, B, C, or D, when shall f be given the value 1 respectively.

@ (a) begin f=1 ; end // <1>



(2) While signal a is changing as shown below, if <1> is executed at the timing shown as A, B, C, or D, when shall f be given the value 1 respectively.

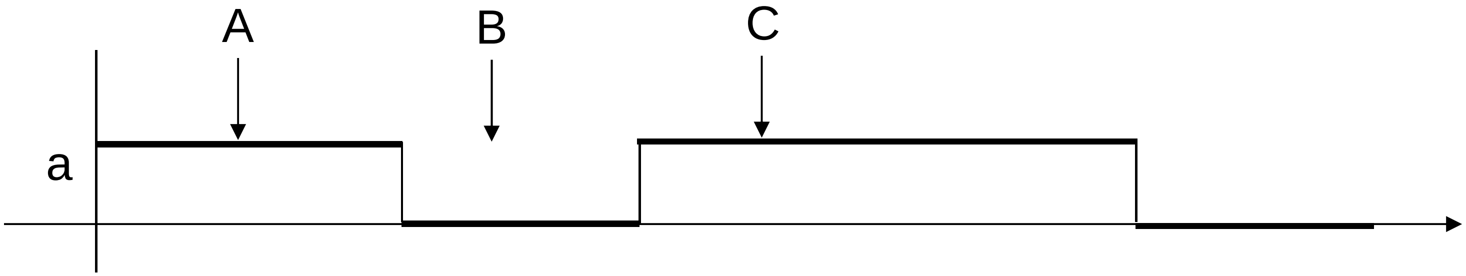
wait (a) begin f=1 ; end // <1>



Q3.9-3; Answer the following questions.

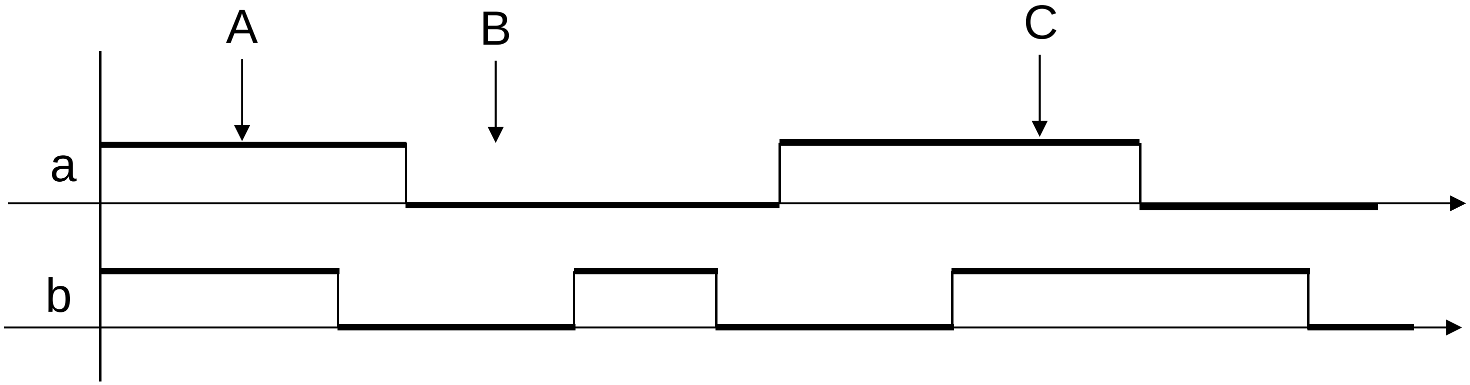
(3) While signal a is changing as shown below, if <1> is executed at the timing shown as A, B, or C, when shall f be given the value 1 respectively.

```
@ ( negedge a ) begin  f=1 ; end // <1>
```



(4) While signal a and b are changing as shown below, if <1> is executed at the timing shown as A, B, or C, when shall f be given the value 1 respectively.

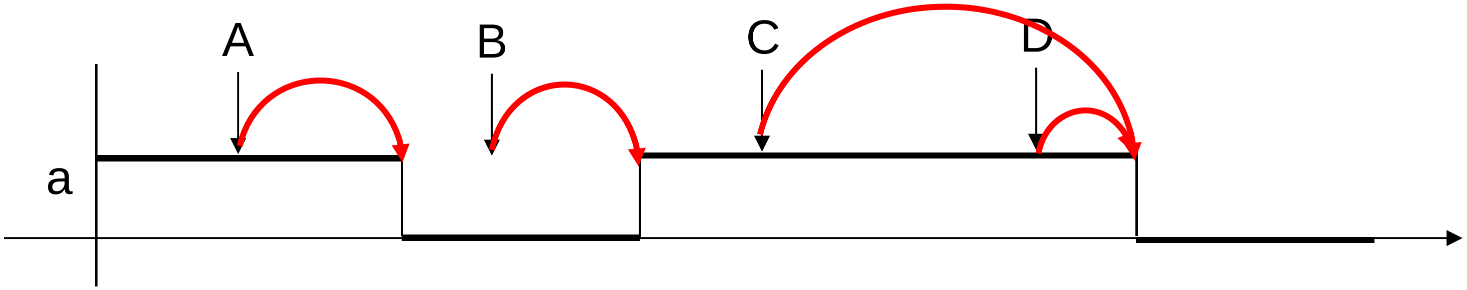
```
@ ( a & b ) begin  f=1 ; end // <1>
```



Q3.9-3; A sample answer

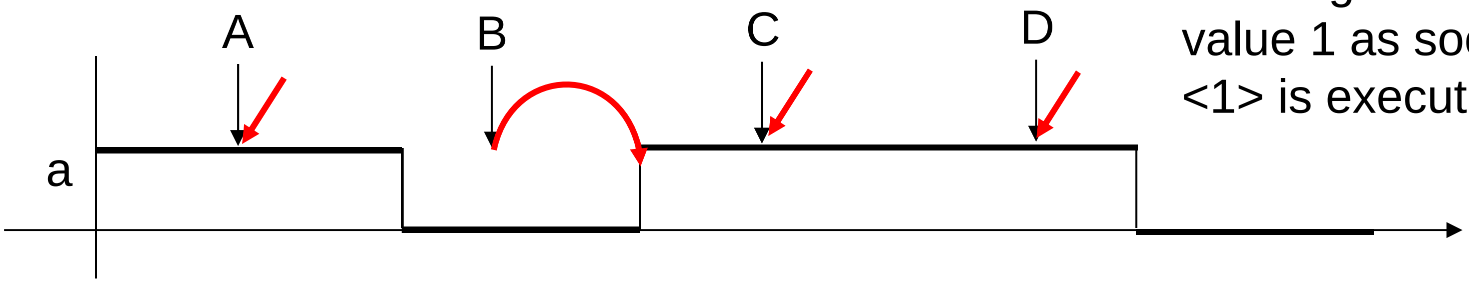
(1) While signal a is changing as shown below, if <1> is executed at the timing shown as A, B, C, or D, when shall f be given the value 1 respectively.

@ (a) begin f=1 ; end // <1>



(2) While signal a is changing as shown below, if <1> is executed at the timing shown as A, B, C, or D, when shall f be given the value 1 respectively.

wait (a) begin f=1 ; end // <1>

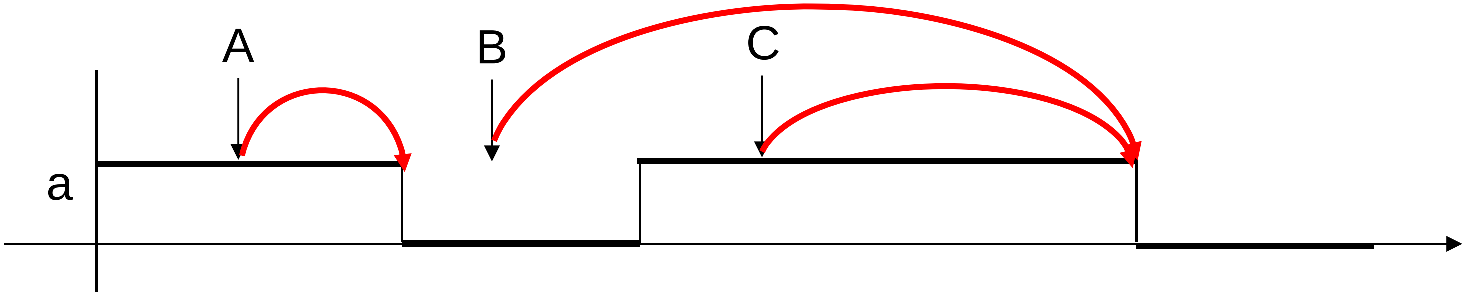


For A, C, and D, f will be given the value 1 as soon as <1> is executed.

Q3.9-3; A sample answer

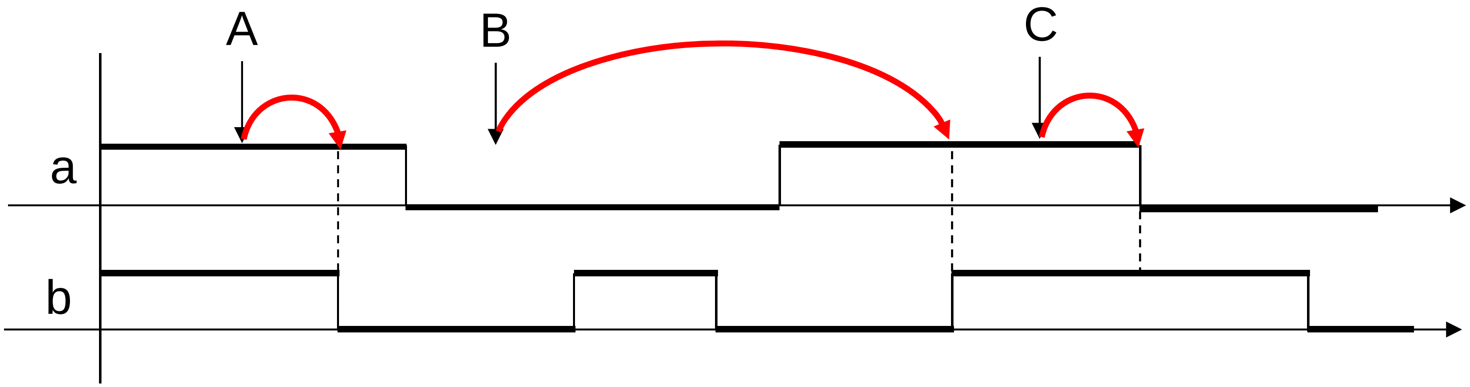
(3) While signal a is changing as shown below, if <1> is executed at the timing shown as A, B, or C, when shall f be given the value 1 respectively.

@ (negedge a) begin f=1 ; end // <1>



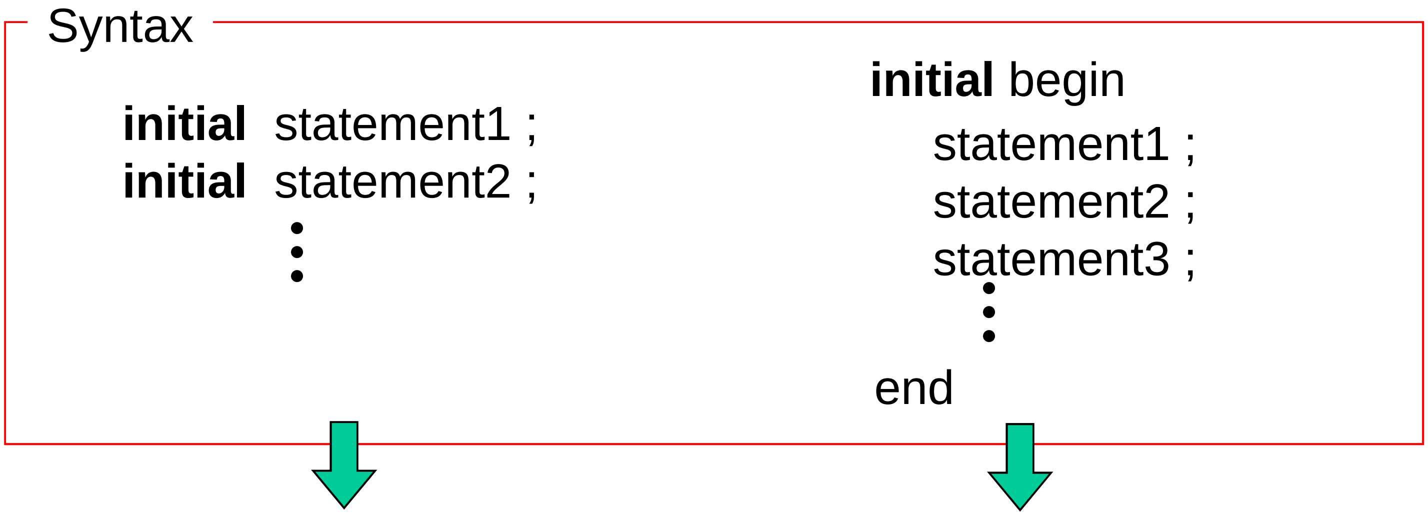
(4) While signal a and b are changing as shown below, if <1> is executed at the timing shown as A, B, or C, when shall f be given the value 1 respectively.

@ (a & b) begin f=1 ; end // <1>



3.9.4 Initial construct

The initial constructs are enabled at the beginning of a simulation, at time 0. The initial construct shall execute only once, and its activity shall cease when the statement has finished.



Syntax

```
initial  statement1 ;
initial  statement2 ;
   :
```

```
initial begin
    statement1 ;
    statement2 ;
    statement3 ;
      :
end
```

statement1, statement2, statement3, and so on are executed only once at time 0.
Which one is to be executed first is tool or vendor dependent.

statement1 is executed at time 0, statement2 is executed next to the statement1, and so on. They are executed once in serial, but logically at the same time if there is no timing control statement.

initial construct

	explanation	comment
input / output arguments	No arguments. References are done directly to the external variables.	
assignment	Blocking procedural assignment and nonblocking procedural assignment are allowed.	LHS must be variable data type.
timing control	An initial construct can contain time-controlling statements. No memory element can be created.	# and @ are allowed.
simulation	Executed only once at time 0.	
synthesis	not synthesizable	
others	tasks and functions can be enabled in initial construct.	

Restrictions

[1] Initial constructs must be written inside a module.

[2] Initial constructs can not be written (declared) in procedures.

[3] Do not write the same variable on LHS in different initial constructs or in different structured procedures.

Example1 ;

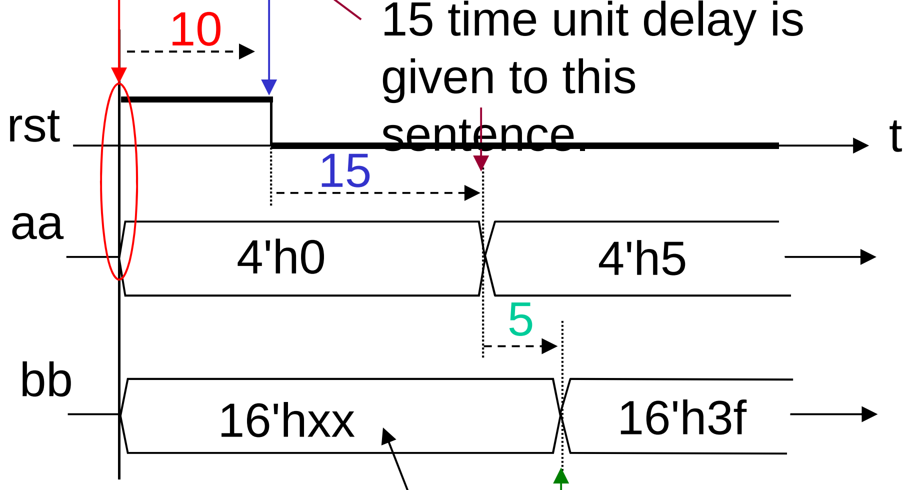
```
initial   begin
                rst = 1'b1 ;
                aa = 4'h0 ;
  # 10          rst = 1'b0 ;
  # 15          aa  = 4'h5 ;
  # 5           bb  = 16'h3f ;
          ⋮
end
```

This code is not synthesizable.

These two sentences have no timing control, therefore they are executed at time 0

10 time unit delay is given to this sentence.

15 time unit delay is given to this sentence.

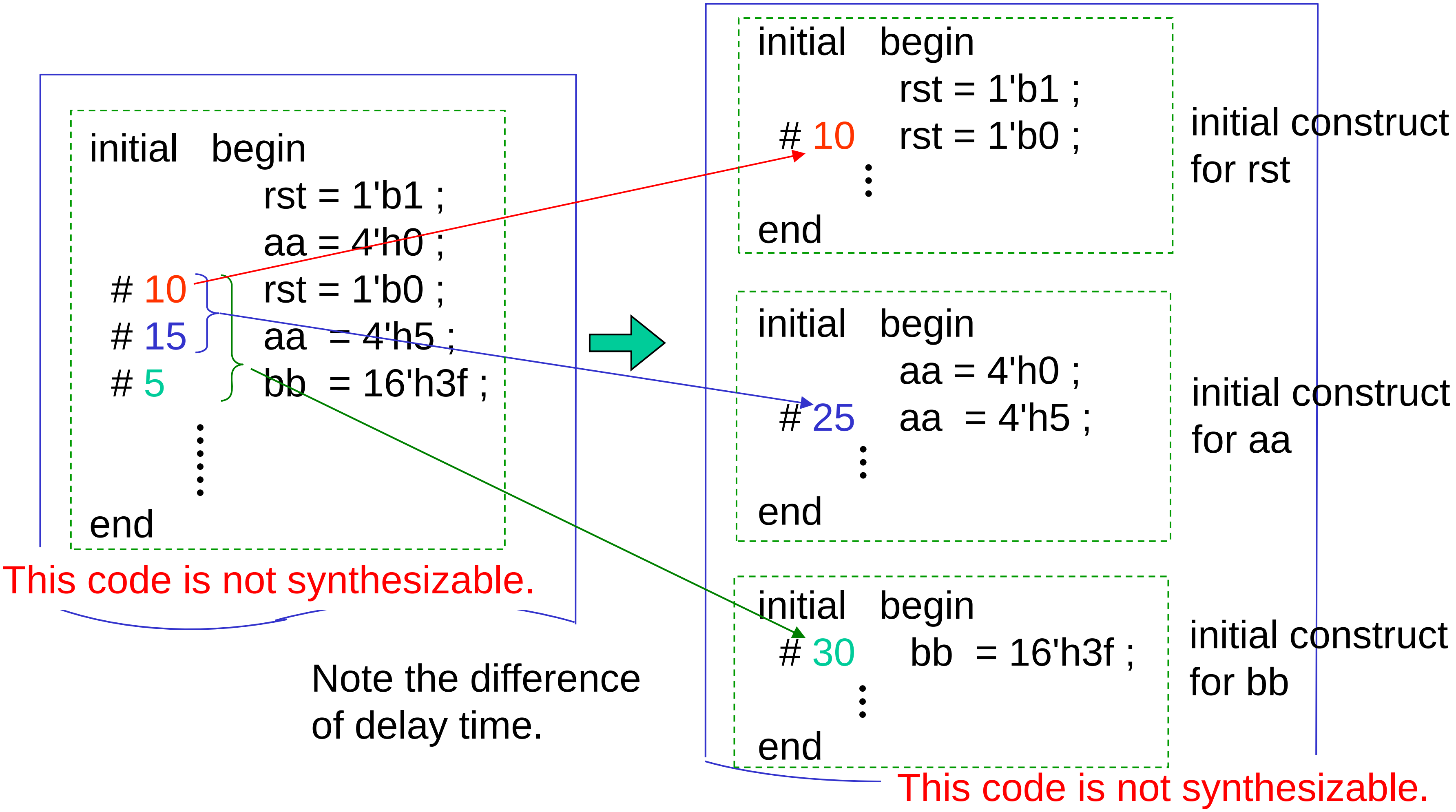


5 time unit delay is given to this sentence.

Variables not assigned any value have initial value x.

The example shown on the previous page is not a recommended style because in one initial construct many variables are given their values.

It is recommended to use one initial construct for one output variable.



```
initial   begin
                rst = 1'b1 ;
                aa = 4'h0 ;
  # 10          rst = 1'b0 ;
  # 15          aa  = 4'h5 ;
  # 5           bb  = 16'h3f ;
          ⋮
end
```

This code is not synthesizable.

Note the difference of delay time.

```
initial   begin
                rst = 1'b1 ;
  # 10          rst = 1'b0 ;
      ⋮
end
```

initial construct for rst

```
initial   begin
                aa = 4'h0 ;
  # 25          aa  = 4'h5 ;
      ⋮
end
```

initial construct for aa

```
initial   begin
  # 30           bb  = 16'h3f ;
      ⋮
end
```

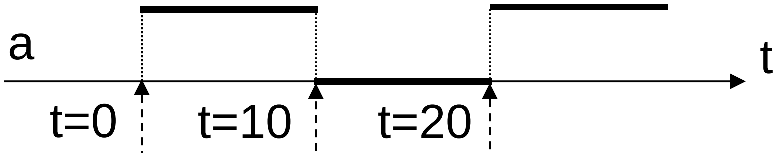
initial construct for bb

This code is not synthesizable.

Example2 ;

```
initial   begin
     a <= 1 ;
     a <= #20  1 ;
     a <= #10  0 ;
          :
          :
          :
          :
end
```

This code is not synthesizable.



Using nonblocking assignment will result in the above waveform because all right hand sides are evaluated before executing each assignment.

Do not use nonblocking procedural assignment. Example 2 is just to show how nonblocking procedural assignment works.

Q3.9-4 : Are the two program given below same or not? If not, what the difference?

```
begin
    a <= 1 ;
    a <= #20  1 ;
    a <= #10  0 ;
end
```

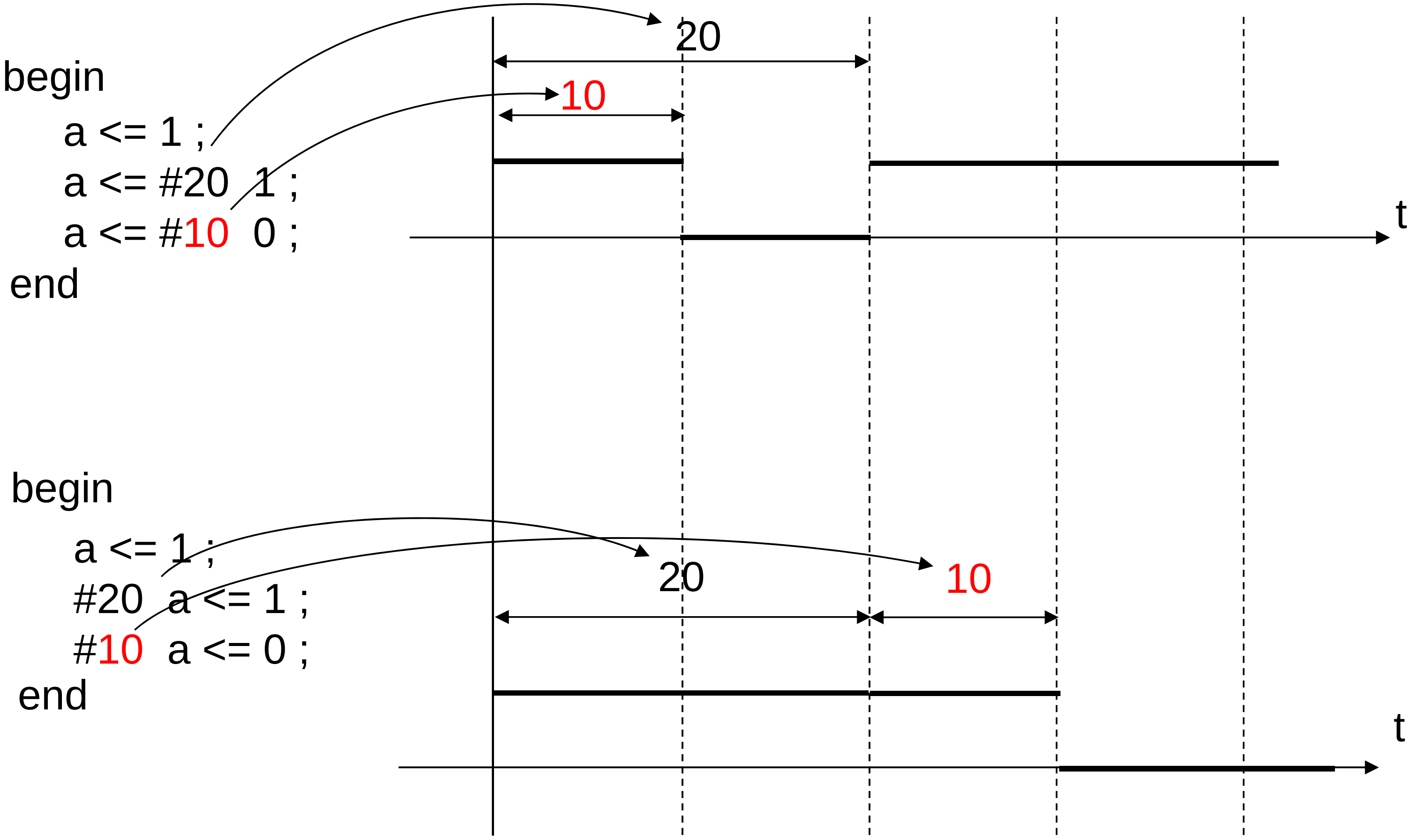
⇔

```
begin
    a <= 1 ;
    #20  a <= 1 ;
    #10  a <= 0 ;
end
```

Do not use nonblocking procedural assignment. Q3.9-2 is just to show how nonblocking procedural assignment works.

Q3.9-4 : A sample answer.

They are different as shown below.



```
begin
    a <= 1 ;
    a <= #20  1 ;
    a <= #10  0 ;
end
```

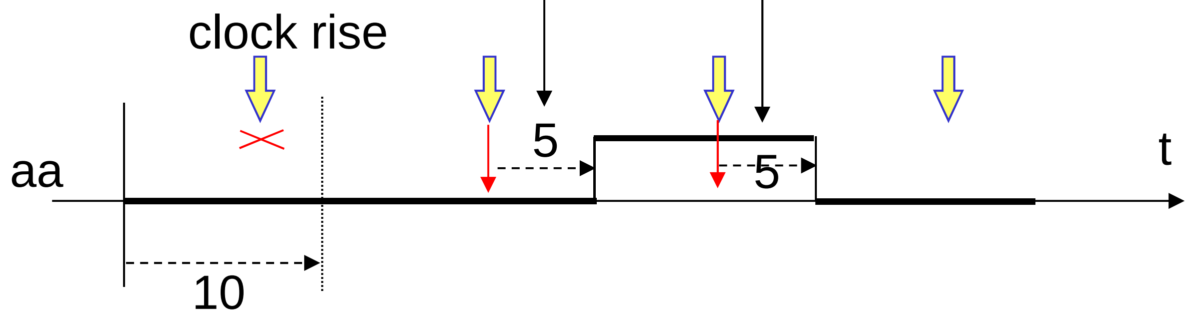
```
begin
    a <= 1 ;
    #20  a <= 1 ;
    #10  a <= 0 ;
end
```

Example3 ;

```
initial   begin
    aa <= 1'b0 ;
 #10 @ ( posedge clk )
    aa <=#5 1'b1 ;
@ ( posedge clk ) aa <=#5 1'b0 ;
        ⋮
end
```

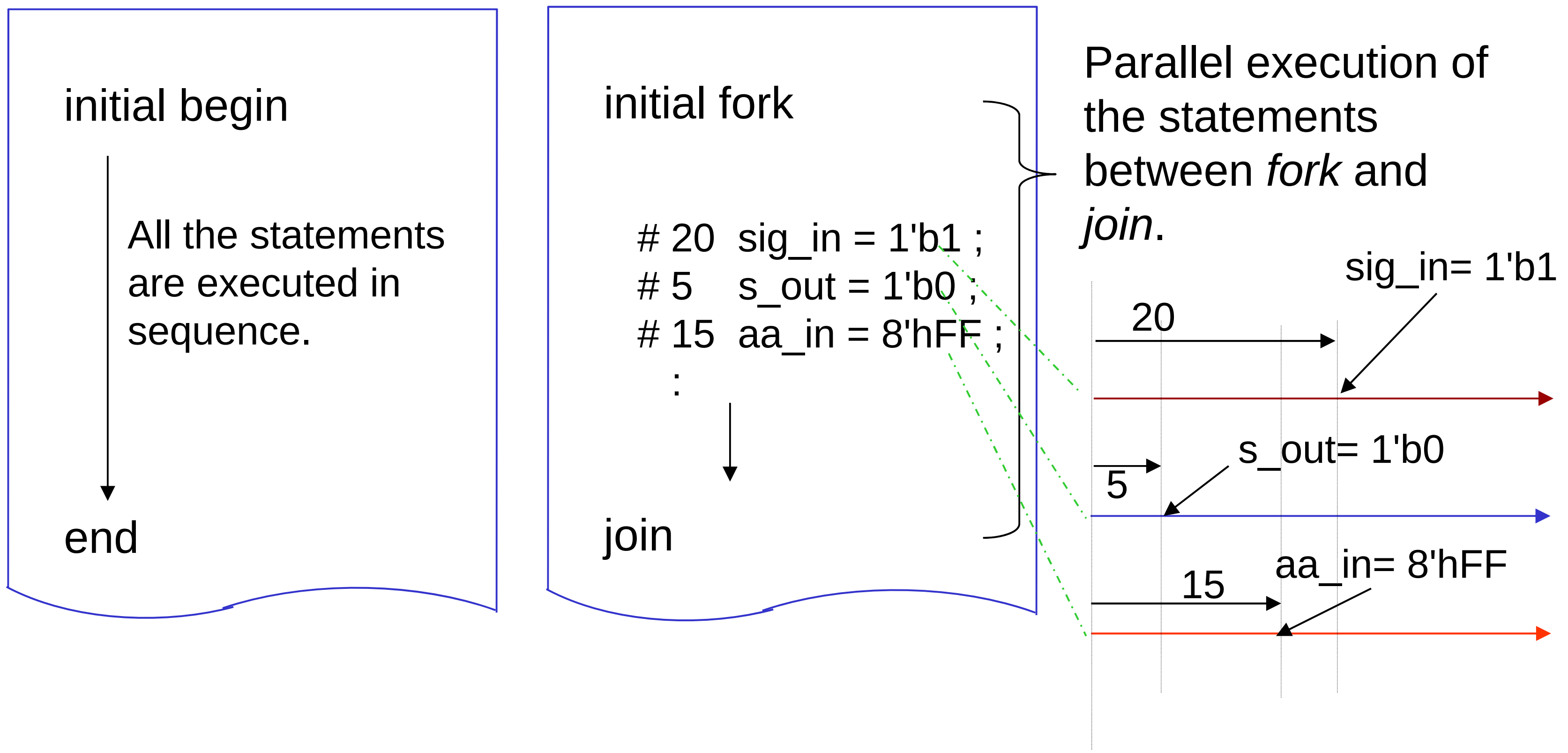
Do not use nonblocking procedural assignment. Example 3 is just to show how nonblocking procedural assignment works.

This code is not synthesizable.



The first clock rise is discarded because it occurs before 10 unit time delay.

If you want several statements executed in parallel in initial construct, use *fork join* block.



```
initial begin

    All the statements
    are executed in
    sequence.

end
```

```
initial fork

  # 20  sig_in = 1'b1 ;
  # 5   s_out = 1'b0 ;
  # 15  aa_in = 8'hFF ;
     :

join
```

Parallel execution of the statements between *fork* and *join*.

Do not use fork/join, if you do not have to. fork/join creates a heavy load on the simulator.

Q3.9-5 : Are the two program given below same or not?
If not, what the difference?

(1)

```
initial a = 0 ;
initial #30 a = 1 ;
initial #50 a = 0 ;
initial #10 a = 1 ;
initial #20 a = 0 ;
```

⇔

```
initial begin
        a = 0 ;
 #10 a = 1 ;
 #10 a = 0 ;
 #10 a = 1 ;
 #20 a = 0 ;
end
```

(2)

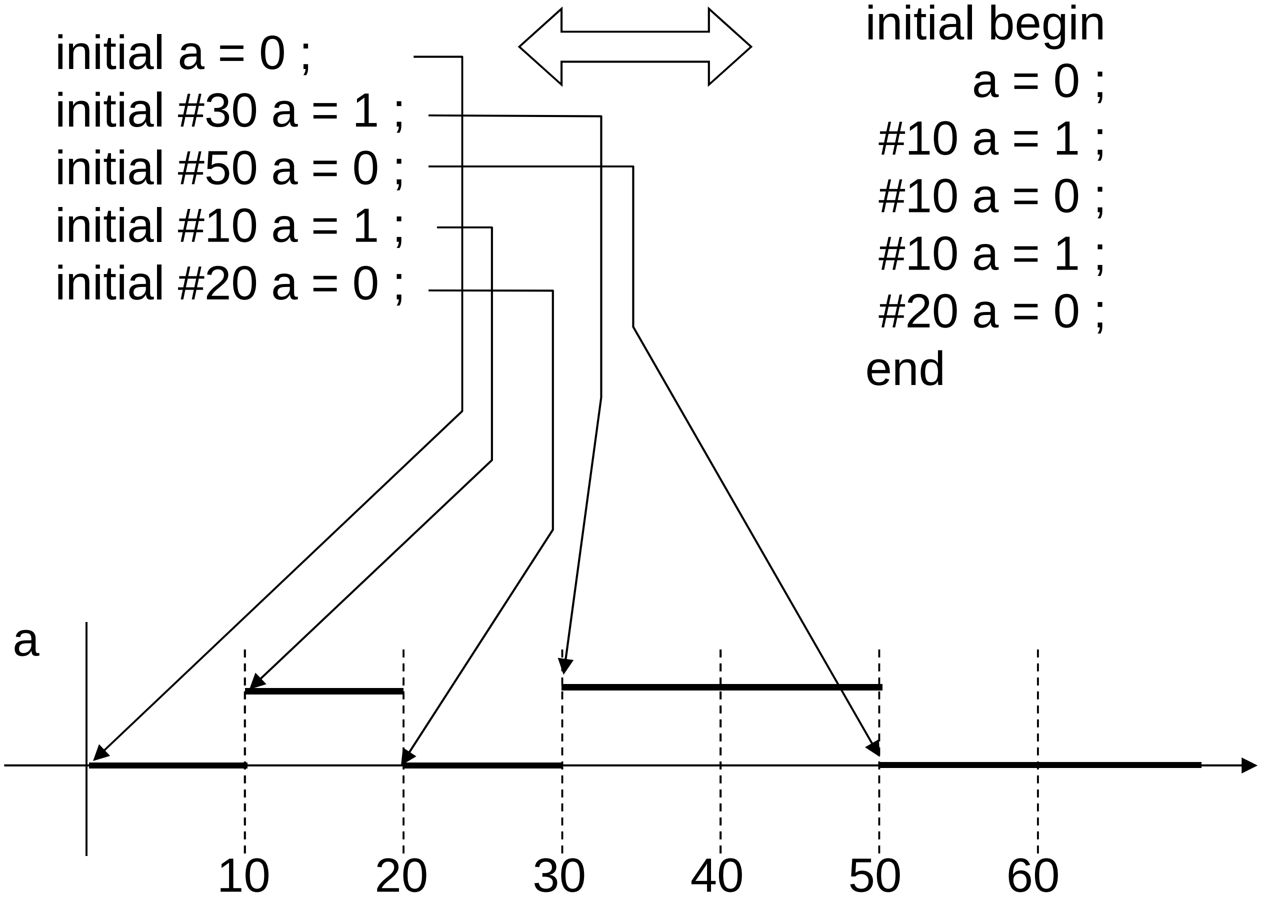
```
initial a = 0 ;
initial #10 a = 1 ;
initial #10 a = 0 ;
initial #30 a = 1 ;
initial #50 a = 0 ;
```

⇔

```
initial begin
        a = 0 ;
 #10 a = 1 ;
        a = 0 ;
 #20 a = 1 ;
 #20 a = 0 ;
end
```

Q3.9-5 : A sample answer.

(1) They are the same.



```
initial a = 0 ;
initial #30 a = 1 ;
initial #50 a = 0 ;
initial #10 a = 1 ;
initial #20 a = 0 ;
```

```
initial begin
      a = 0 ;
 #10 a = 1 ;
 #10 a = 0 ;
 #10 a = 1 ;
 #20 a = 0 ;
end
```

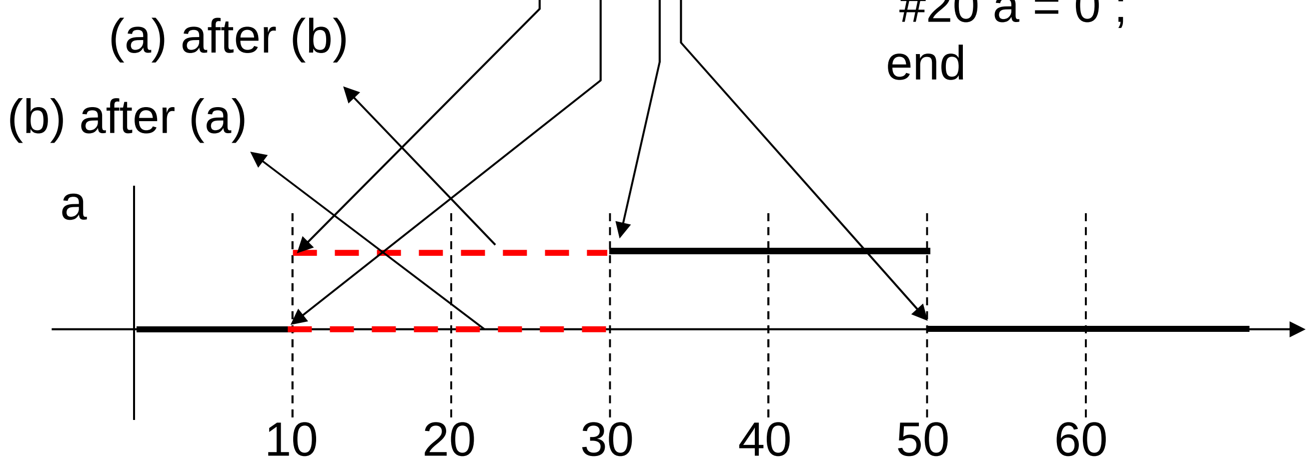
Q3.9-5 : A sample answer.

(2) They are different.

Depending on which one of (a) or (b) is executed earlier, a is 1 or 0 from time 10 to 30, and which one executes earlier is dependent on tools or vendors. But the one on the right means a is 0 from time 10 to 30 regardless of tools and vendors.

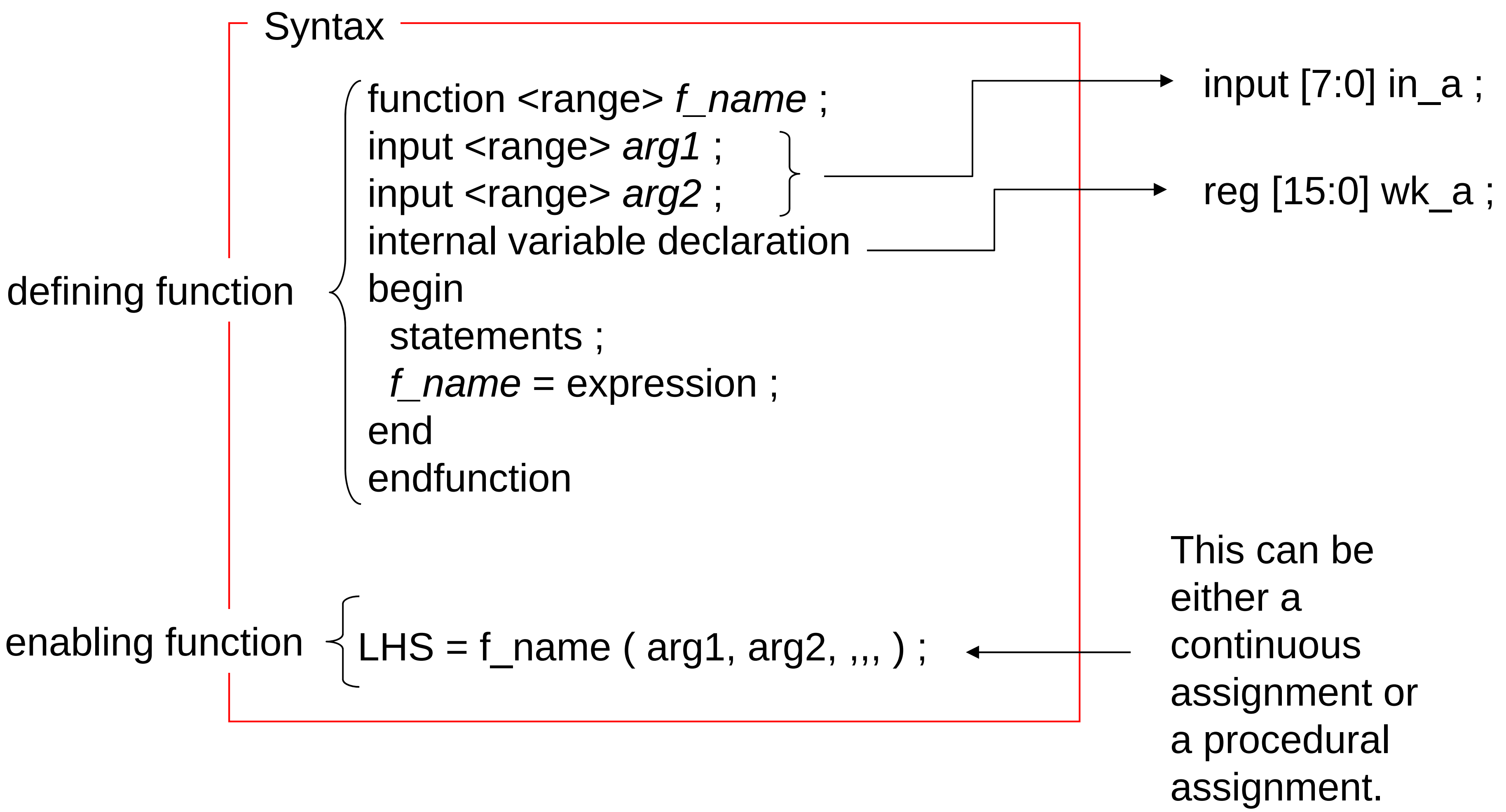
```
initial a = 0 ;
initial #10 a = 1 ; // (a)
initial #10 a = 0 ; // (b)
initial #30 a = 1 ;
initial #50 a = 0 ;
```

```
initial begin
        a = 0 ;
 #10 a = 1 ;
        a = 0 ;
 #20 a = 1 ;
 #20 a = 0 ;
end
```



3.9.5 function

Function provides a means of splitting code into small parts that are frequently used in a model.



Syntax

defining function

```
function <range> f_name ;
input <range> arg1 ;
input <range> arg2 ;
internal variable declaration
begin
  statements ;
  f_name = expression ;
end
endfunction
```

input [7:0] in_a ;

reg [15:0] wk_a ;

enabling function

```
LHS = f_name ( arg1, arg2, ,,, ) ;
```

This can be either a continuous assignment or a procedural assignment.

Instead of

```
function <range> f_name ;
input <range> arg1 ;
input <range> arg2 ;
```

Use this style.

, it is allowed to declare inputs in a function port list as below

```
function <range> f_name
 (
    input <range> arg1,
    input <range> arg2,
    ,,,,,
  ) ;
```

Function

	explanation	comment
input arguments	A function shall have at least one input type argument.	
output arguments	A function shall not have an output or inout type argument. (It shall return a single value, not via output argument but via its identifier.)	Same as function in C language.
assignment	Blocking procedural assignment must be used in a function. Nonblocking procedural assignment is not allowed.	All the internal variables must be variable data type.
timing control	A function shall create combinational logic, therefore no delay nor wait control can be used. No memory element can be created.	# and @ are not allowed.
simulation	A function shall execute in one simulation time unit.	
synthesis	A function shall not create any memory element. It shall create combinational logic.	
others	A function can not enable a task. It can enable other functions.	

Restrictions

[1] @ and # are not allowed in function.

[2] Nonblocking procedural assignment is not allowed.

[3] Function can not enable tasks. (Can not call tasks.)

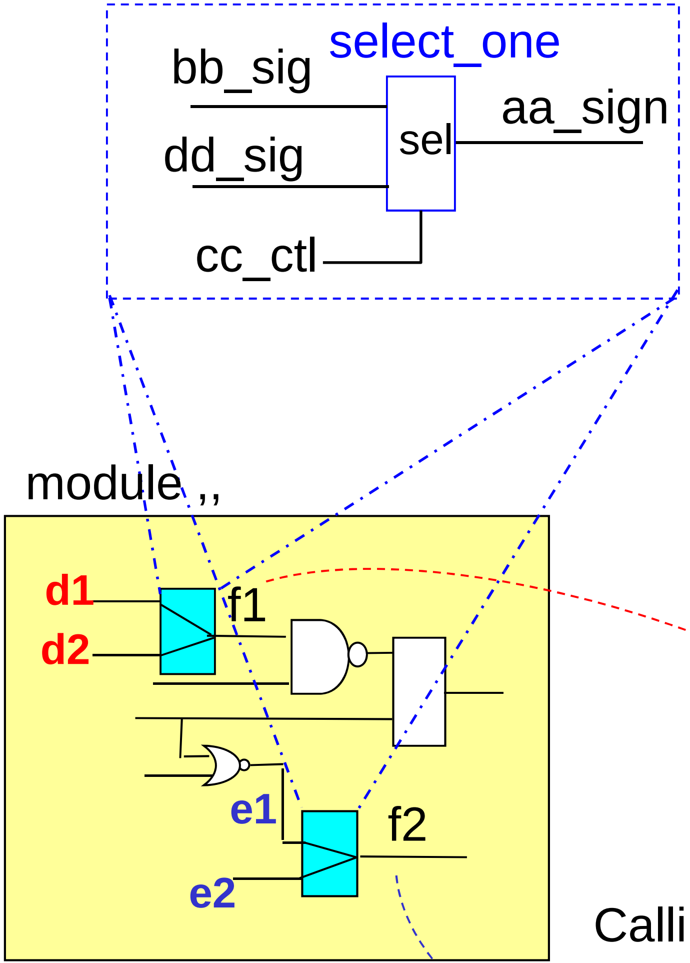
[4] Function can not have multiple outputs. The value of the function is returned via function name.

[5] Function must be written inside a module. It must not be written (declared) in structured procedures.

[6] Do not write external variables in LHS.

function statement

An example of a function:



```
module ,,,

  wire [7:0] d1, d2, e1, e2 ;
  wire c1, c2 ;
                  :

  function [7:0] select_one ;
  input [7:0] bb_sig, dd_sig
  input cc_ctl ;
  begin
    if ( cc_ctl == 1'b1 ) begin
      select_one = bb_sig ;
    end
    else  begin
      select_one  = dd_sig ;
    end
  end
  endfunction
                    :
  assign f1 = select_one ( d1, d2, c1 ) ;
                    :
  assign f2 = select_one ( e1, e2, c2 ) ;
                    :
 endmodule
```

Function description

Calling function

function can be used only inside the module where it is declared.

In verilog2001, recursive call of function becomes possible.
For such purpose, automatic key word is introduced.

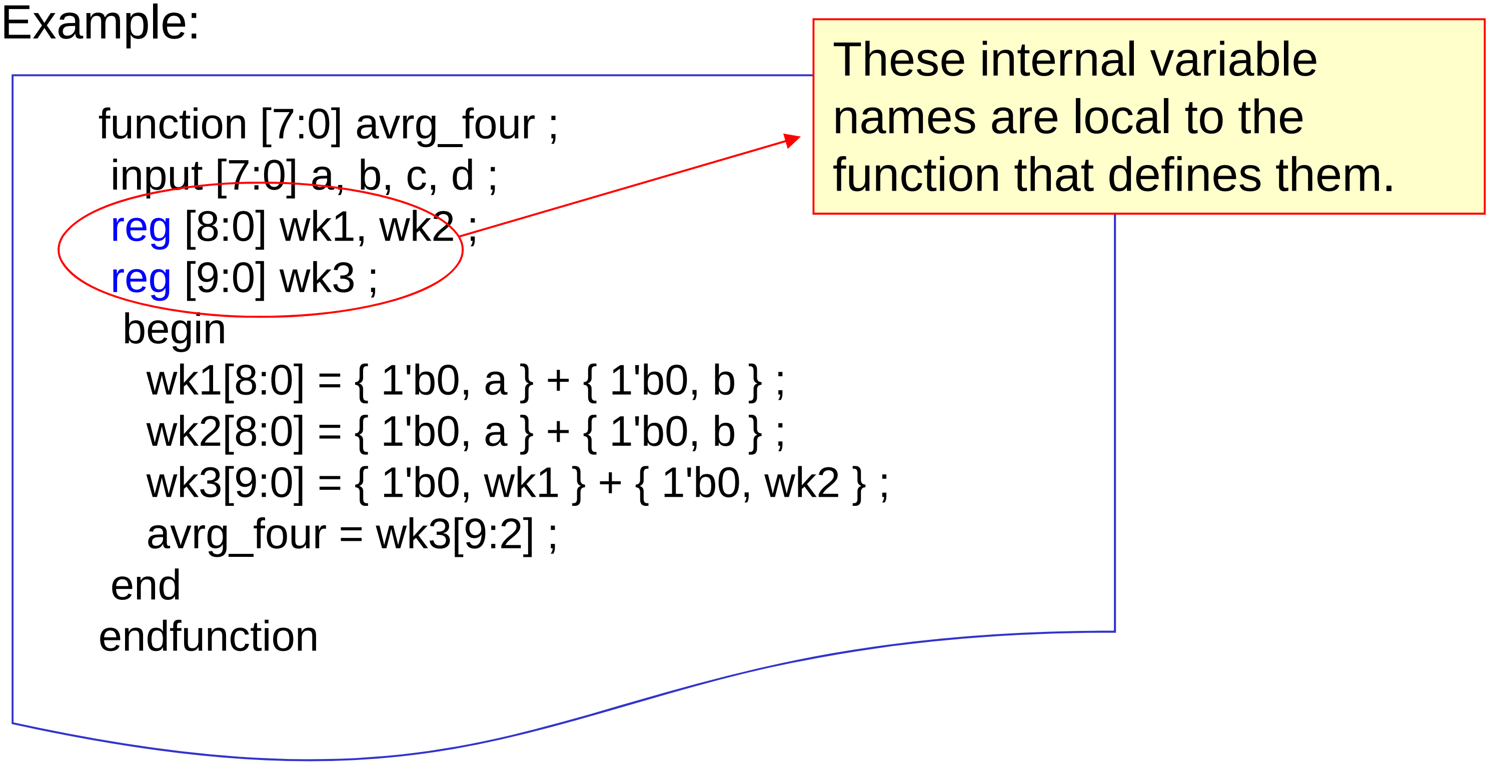
```
function automatic function_identifier ;
     ⋮
endfunction
```

The keyword automatic declares an automatic function that is reentrant, with all the function declarations allocated dynamically for each concurrent function call. Automatic function items cannot be accessed by hierarchical references.
Automatic functions can be invoked through the use of their hierarchical name.

In a function, all the RHS, if not local, must be declared as input arguments. If they are not declared as input arguments, it refers to a signal outside the function directly.

Internal variables used in function must be variable data type.

Example:

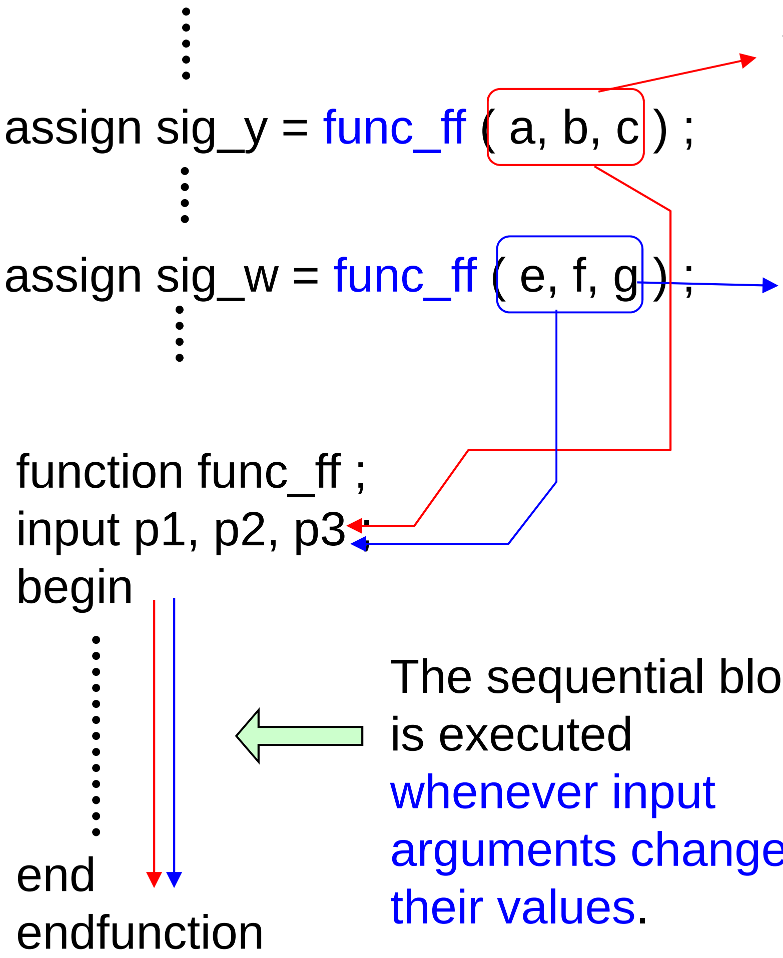


```
function [7:0] avrg_four ;
 input [7:0] a, b, c, d ;
 reg [8:0] wk1, wk2 ;
 reg [9:0] wk3 ;
  begin
    wk1[8:0] = { 1'b0, a } + { 1'b0, b } ;
    wk2[8:0] = { 1'b0, a } + { 1'b0, b } ;
    wk3[9:0] = { 1'b0, wk1 } + { 1'b0, wk2 } ;
    avrg_four = wk3[9:2] ;
 end
endfunction
```

These internal variable names are local to the function that defines them.

execute timing of a function

In RTL simulation, function is executed whenever its arguments are updated.



```
⋮
assign sig_y = func_ff ( a, b, c ) ;
⋮
assign sig_w = func_ff ( e, f, g ) ;
⋮

function func_ff ;
input p1, p2, p3 ;
begin
  ⋮
end
endfunction
```

When input arguments a, b, or c change their value, function func_ff will be executed.

When input arguments e, f, or g change their value, function func_ff will be executed.

The sequential block is executed whenever input arguments change their values.

sig_y will be given a new value if a, b, or c changes their vaule. But sig_w will not given the new value unless e, f, or g changes.

synthesis and simulation of a function

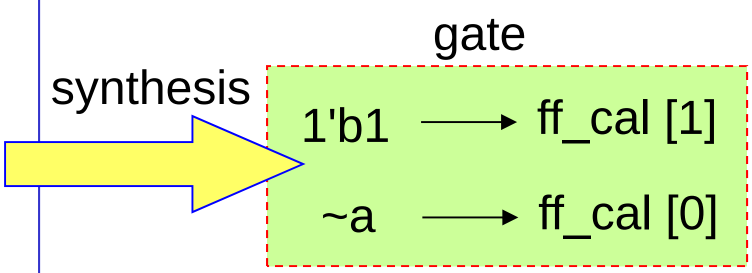
If there is a path in which a variable is not given its value, a synthesis tool will treat such a path as “don't care” and create a combinational circuit without any memory elements.

```
function [1:0] ff_cal ;
input a, b ;
begin
  if ( a ) begin
        ff_cal = 2'b10 ;
  end
  else begin
    if ( b ) begin
       ff_cal = 2'b 11 ;
    end
//    else begin
//       ff_cal = 2'b01 ;
//    end
  end
end
endfunction
```

RTL

missing else part

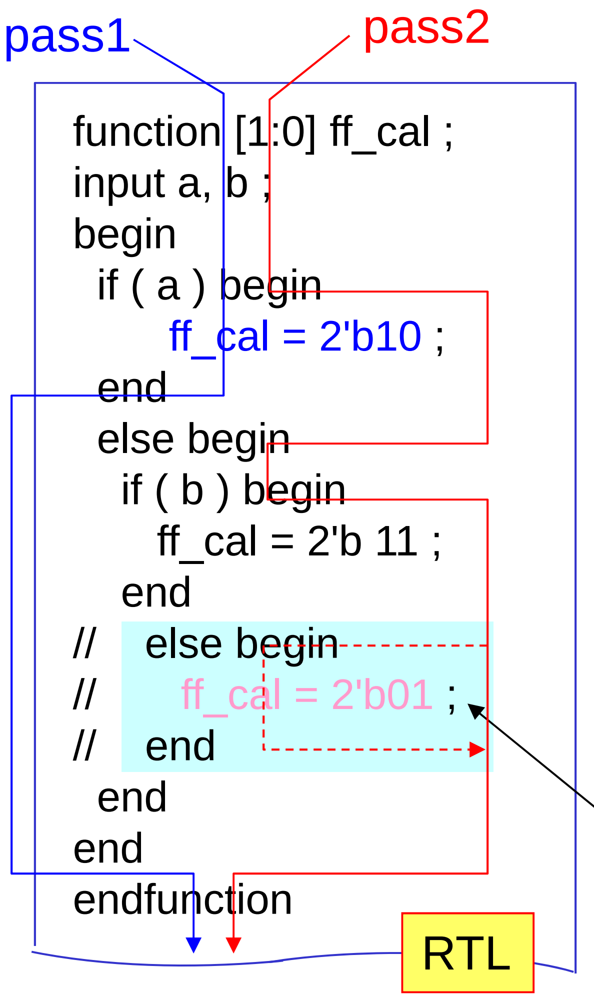
In an example on the left, else path is missing in case a=0 and b=0. A synthesis tool thinks that in this path any output may be accepted because no data is given in the path.



A possible synthesis result of the function

synthesis and simulation of a function (continued)

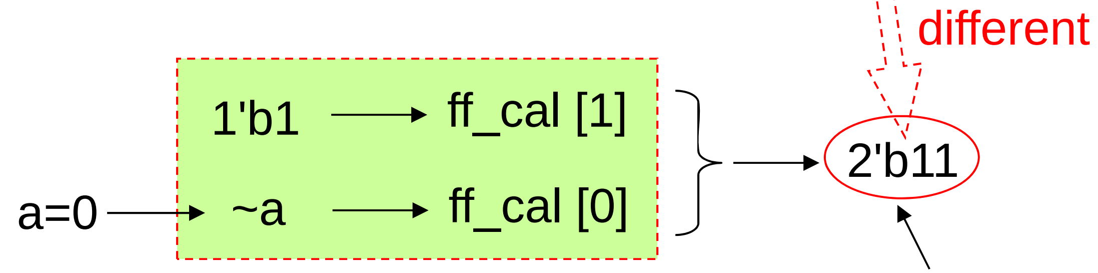
When simulated, the RTL simulator will assign previous value for the variable in such a path where no value is assigned to it. This causes latching.



```
function [1:0] ff_cal ;
input a, b ;
begin
  if ( a ) begin
      ff_cal = 2'b10 ;
  end
  else begin
    if ( b ) begin
       ff_cal = 2'b 11 ;
    end
//   else begin
//     ff_cal = 2'b01 ;
//   end
  end
end
endfunction
```

missing else part

When the function ff_cal is invoked with a=1, ff_cal is given the value 2'b10. (pass1)
And if it is invoked with a=0 and b=0 next time, ff_cal is not given any value because else part which shall be executed in such a case is missing. (pass2)
A simulator places the previous value, 2'b10, to ff_cal in pass2. This must be different from net level simulation result which is shown below.



gate

ff_cal shall be 2'b11 for a=0 and b=0 case.

synthesis and simulation of a function (continued)

As investigated in the previous pages, if a variable is not given a value in some paths, latching will occur in RTL simulation and the simulation result may be different between RTL simulation and gate level simulation.

Therefore, we must be careful to assign some value to all the variables which appear on LHS in every paths of the procedure.

The same latching will occur in simulating case statement if case items do not cover all the possible cases.

```
function   [3:0] dec2to4 ;
input [1:0]  d_code_in ;
begin
    case (d_code_in )
         2'b00 : dec2to4 =  4'h1  ;
         2'b01 : dec2to4 =  4'h2  ;
        default : dec2to4 = 4'hx ;
    endcase
end
endfunction
```

This default helps to suppress latching. However, if case item can be 2'b10 in normal cases, this default will not help removing mismatch between RTL and gate level simulation.
In such a case use a default shown below if you really don't care about 2'10 and 2'b11 cases.

```
default : dec2to4 = 4'h2 ;
```

Summary on if-else and case in function

In function,

(1) do not use if without else part,

(2) do not use case with missing case items,

they must cover all the possible cases

to avoid latching operation of a simulator and

to avoid mismatch of the results of RTL level

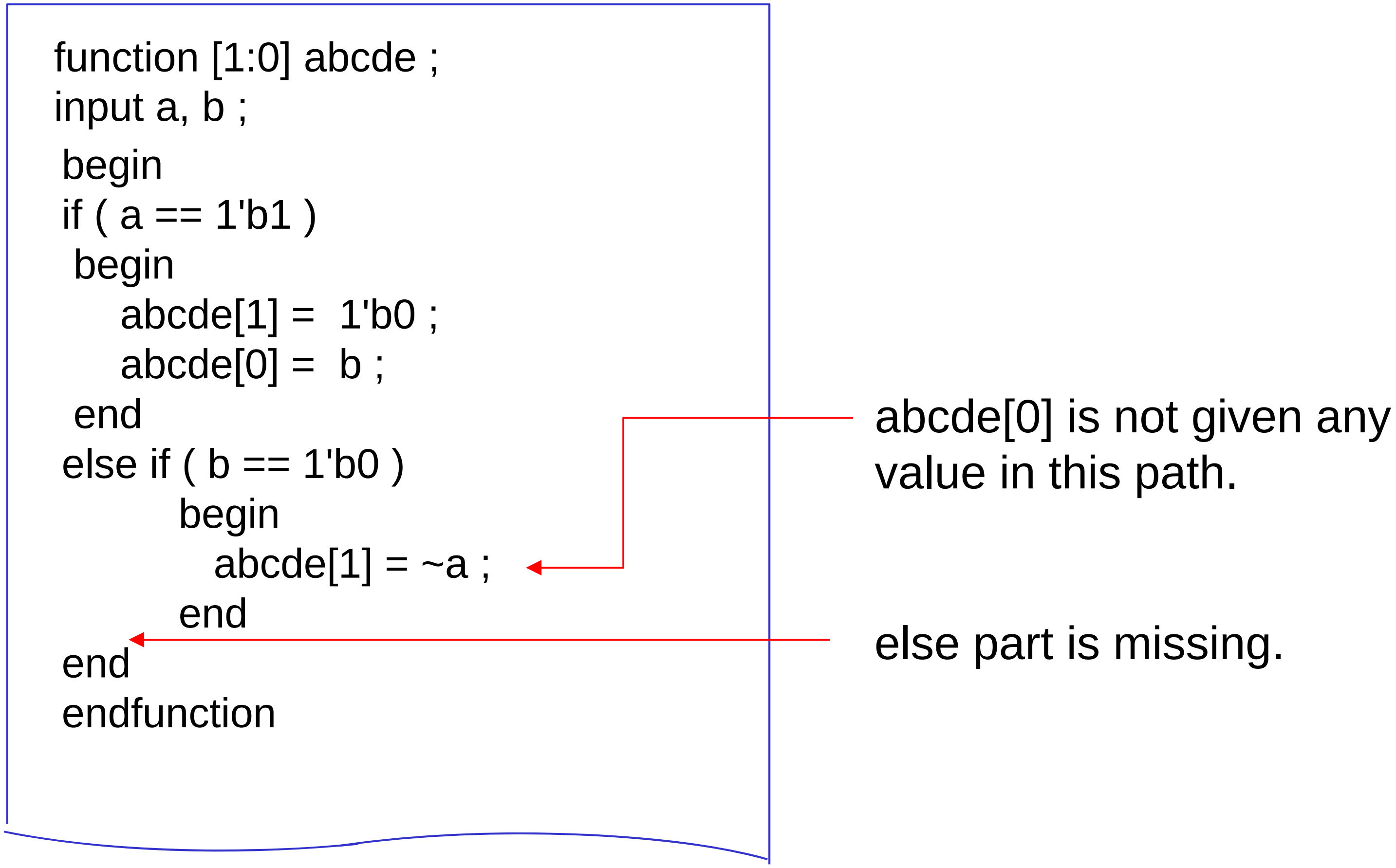
and gate level simulation.

The same rule must be applied to task.

Q3.9-6: Check if the following code has any problem or not for creating combinational logic by function statement.

```
function [1:0] abcde ;
input a, b ;
 begin
 if ( a == 1'b1 )
  begin
      abcde[1] =  1'b0 ;
      abcde[0] =  b ;
  end
 else if ( b == 1'b0 )
          begin
             abcde[1] = ~a ;
          end
 end
 endfunction
```

Q3.9-6: A sample answer.



```
function [1:0] abcde ;
input a, b ;
 begin
 if ( a == 1'b1 )
  begin
      abcde[1] =  1'b0 ;
      abcde[0] =  b ;
  end
 else if ( b == 1'b0 )
          begin
             abcde[1] = ~a ;
          end
 end
 endfunction
```

abcde[0] is not given any value in this path.

else part is missing.

system function

Several system functions are available.

$time, $stime, $realtime

$time returns a 64-bit time, scaled to the timescale unit of the module that invoked it, while $stime and $realtime return a 32-bit time and a real number time respectively.

$random

$random returns a new 32-bit random number each time it is called.

$signed, $unsigned

$signed and $unsigned can cast the type signed and unsigned. See the next page.

In verilog2001, system function $signed and $unsigned are introduced to cast types. These functions evaluate the value of input expression and return the value with the same size and value of the input expression and the type defined by the function.
By using these functions, for example, we can compare signed value and unsigned value as signed value as shown below.

```
wire signed [7:0] sig_a ; // signed
wire [3:0] sig_b ;            // unsigned
wire a_gt_b_flg ;

assign a_gt_b_flg = ( sig_a < $signed( { 1’b0, sig_b } ) ) ;
```

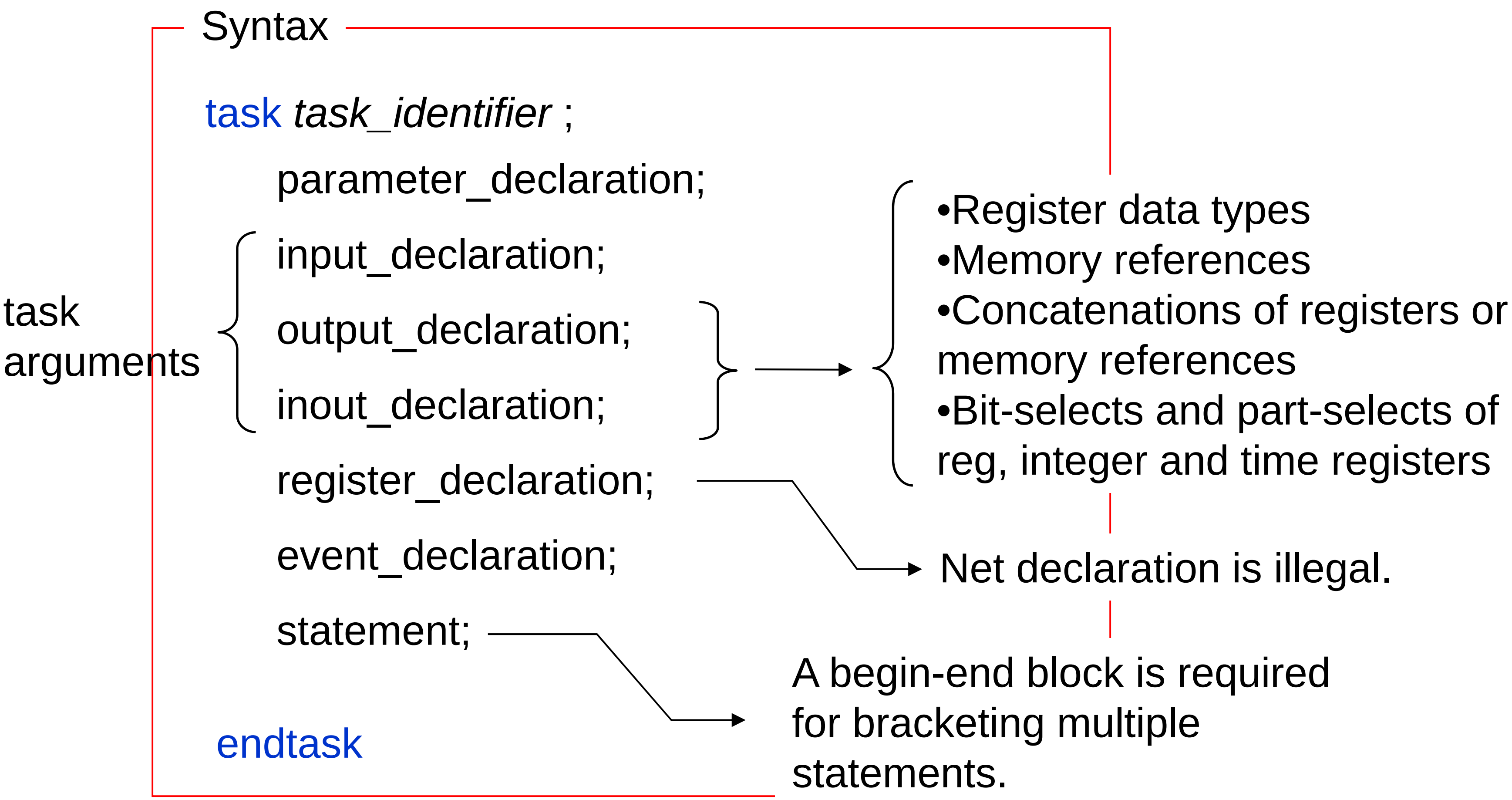
In the above example, if we do not use $signed, comparison is done between the two operands assuming they are both unsigned, positive. But, by using $signed, comparison is done between the two signed values.

When sig_a is -6 (1111_1010) and sig_b is 10 (1010),

sig_a < $signed({ 1’b0, sig_b }) is true, → -6 < 10
sig_a == $signed(sig_b) is true, and
sig_a > sig_b is true. → -6 > 10

3.9.6 Task

Tasks are almost the same as functions, but they can have several outputs. They are very useful for creating a structured test bench by providing functionalities such as bus operation or CPU operation.



Syntax

task *task_identifier* ;

parameter_declaration;

input_declaration;

output_declaration;

inout_declaration;

register_declaration;

event_declaration;

statement;

endtask

task arguments

- •Register data types
- •Memory references
- •Concatenations of registers or memory references
- •Bit-selects and part-selects of reg, integer and time registers

Net declaration is illegal.

A begin-end block is required for bracketing multiple statements.

Instead of

```
task task_identifier ;
      input_declaration;
      output_declaration;
      inout_declaration;
```

, it is allowed to declare inputs and outputs in a task port list as below

```
task task_identifier
 (
    input <range> arg1,
    output <range> arg2,
    input <range> arg3,
  ) ;
```

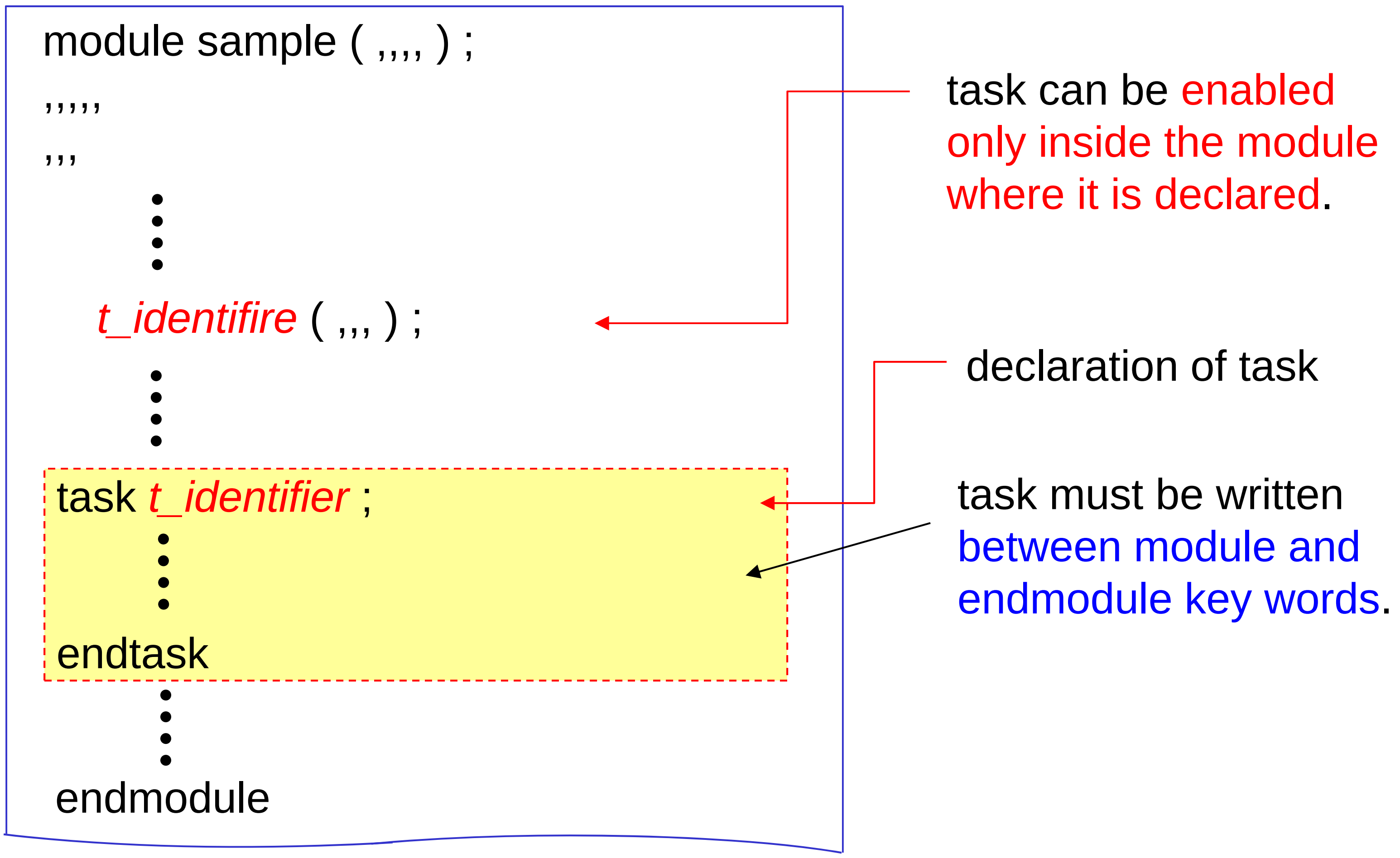
Task

	explanation	comment
input arguments	A task can have zero or more arguments of any type, input, output, or inout.	It shall not return a value via its identifier.
output arguments		
assignment	Both blocking and nonblocking procedural assignment are allowed.	All the internal variables must be variable data type.
timing control	A task can contain time-controlling statements. No memory element can be created.	# and @ are allowed.
simulation	A task may not be executed in one simulation time unit.	
synthesis	A task is synthesizable, but it may not be synthesizable depending on the features implemented.	
others	A task can enable other tasks and functions. Regardless of how many tasks have been enabled, control shall not return until all enabled tasks have completed.	

Restrictions

[1] Task must be written inside a module. It must not be written (declared) in structured procedures.

[2] Task can be invoked from inside procedures. It can not be invoked from outside the procedures.



```
module sample ( ,,,, ) ;
,,,,,
,,,
        •
        •
        •
        •
    t_identifire ( ,,, ) ;
        •
        •
        •
        •
 task t_identifier ;
        •
        •
        •
        •
 endtask
        •
        •
        •
        •
 endmodule
```

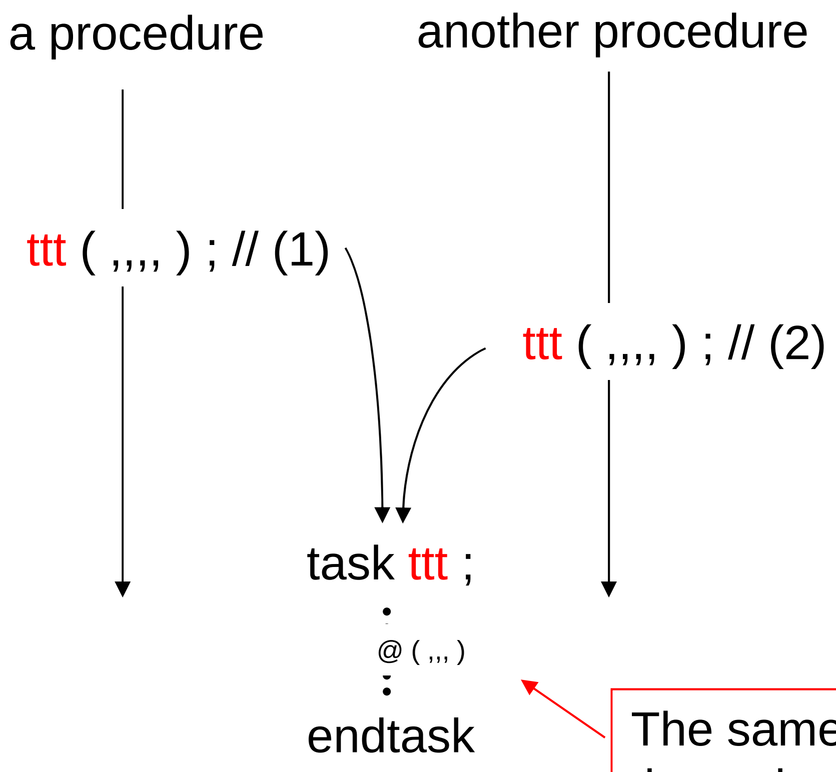
task can be enabled only inside the module where it is declared.

declaration of task

task must be written between module and endmodule key words.

A task may be enabled several times in a module. If two or more instances are enabled concurrently, then all registers and events declared in that task should be static, i.e., one variable for all instances.

Care must be taken in case that two or more instances are enabled concurrently.



a procedure

another procedure

ttt (,,,,) ; // (1)

ttt (,,,,) ; // (2)

task ttt ;

@ (,,,)

endtask

While a task ttt is running in a procedure, the same task ttt may run in another procedure.

This can happen because # and @ are allowed in a task.

If such concurrent processing may occur, local work space used by (1) may be destroyed by (2) because there is only one work space for ttt even if it is activated concurrently. If this happens, the simulation result may not be correct.

The same local work space to execute the task ttt is used for (1) and (2).

The local work shall be discarded on executing endtask.

Use automatic keyword if concurrent execution of a task is mandatory.

In verilog2001, recursive call of task becomes possible.
For such purpose, automatic key word is introduced.

```
task automatic task_identifier ;
     ⋮
endtask
```

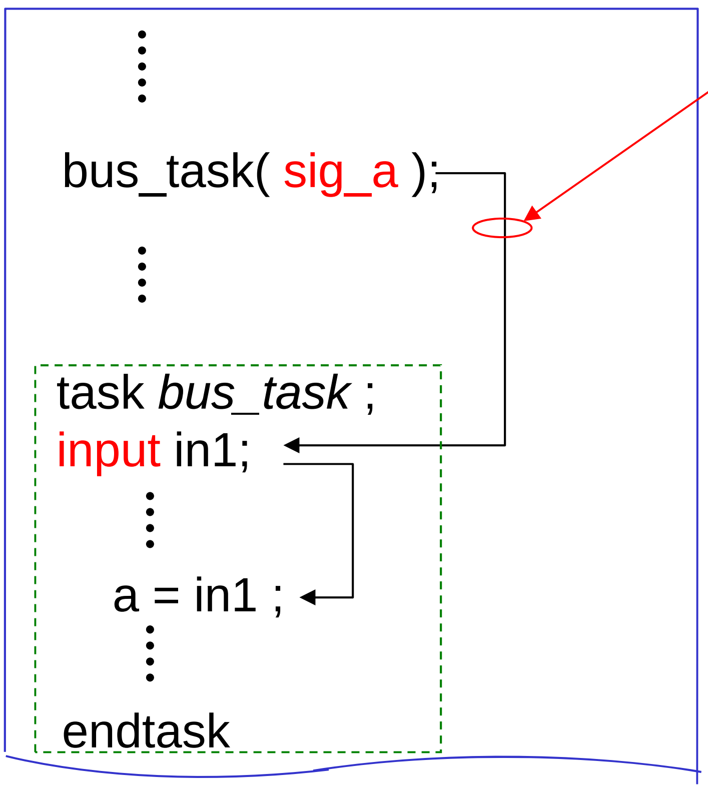
Tasks with the keyword automatic are automatic tasks.
All items declared inside automatic tasks are allocated dynamically for each invocation.
Automatic task items cannot be accessed by hierarchical references.
Automatic tasks can be invoked through use of their hierarchical name.

task and arguments

With input arguments, values are given to the inputs when the task is enabled.

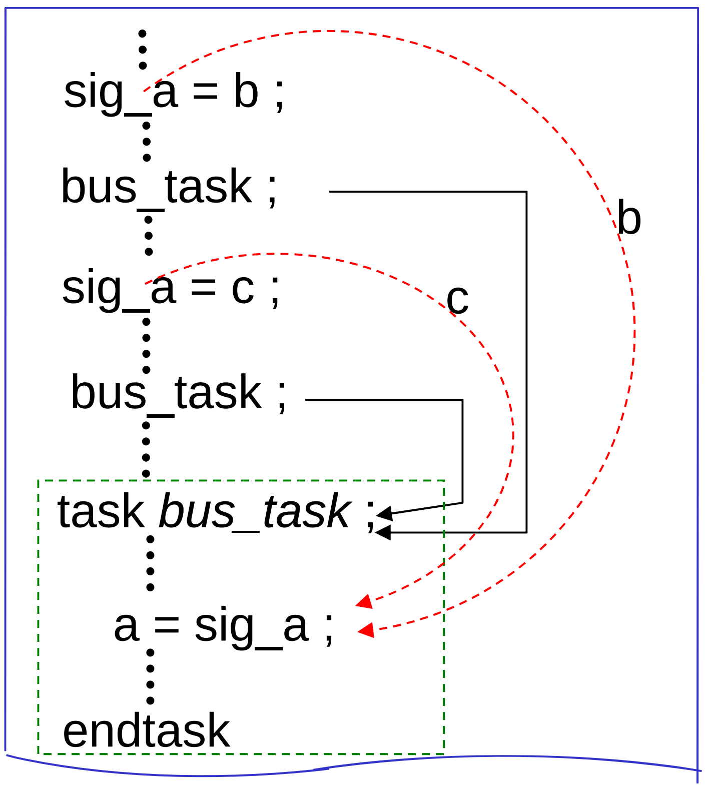
⇨ If no argument given, it refers to external signals directly. Therefore, values are passed to the task when the signals are given values.

With input argument

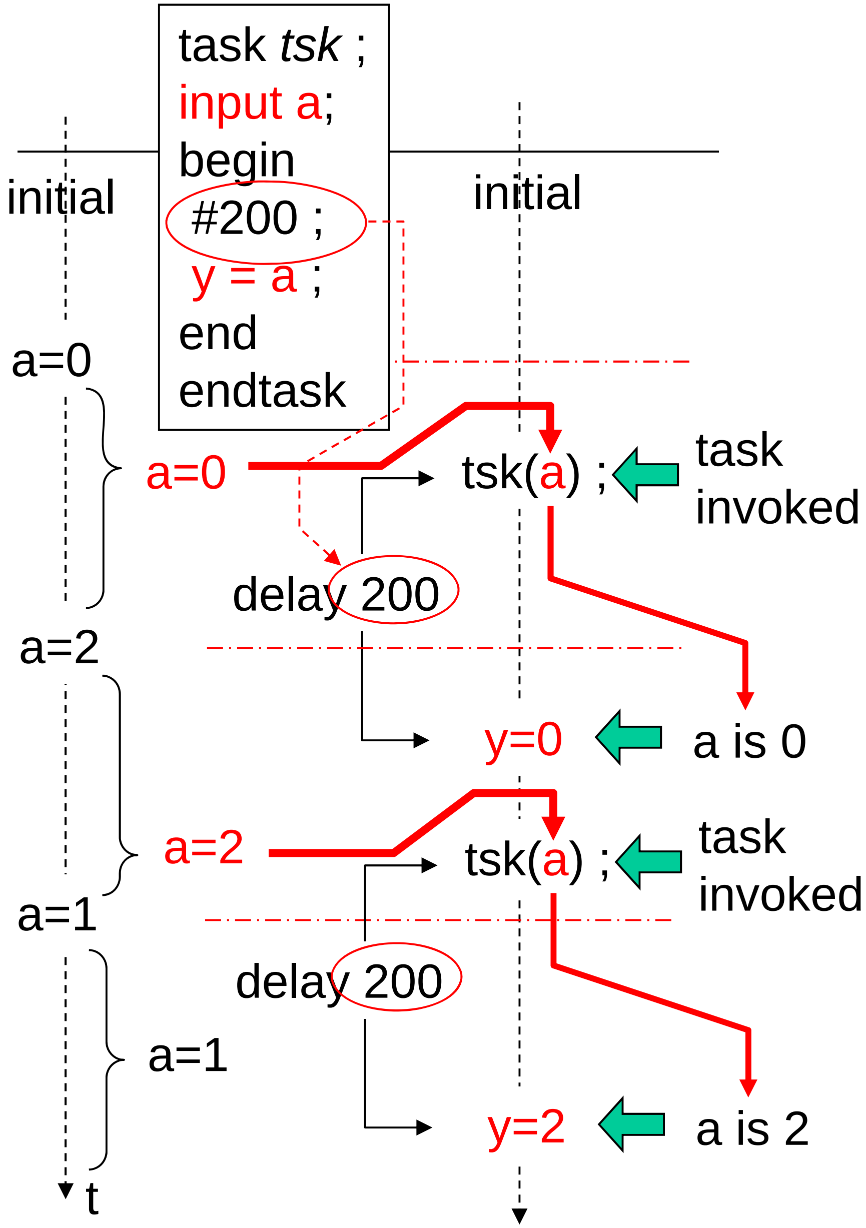


```
⋮
bus_task( sig_a );
⋮
task bus_task ;
input in1;
⋮
a = in1 ;
⋮
endtask
```

Without input argument

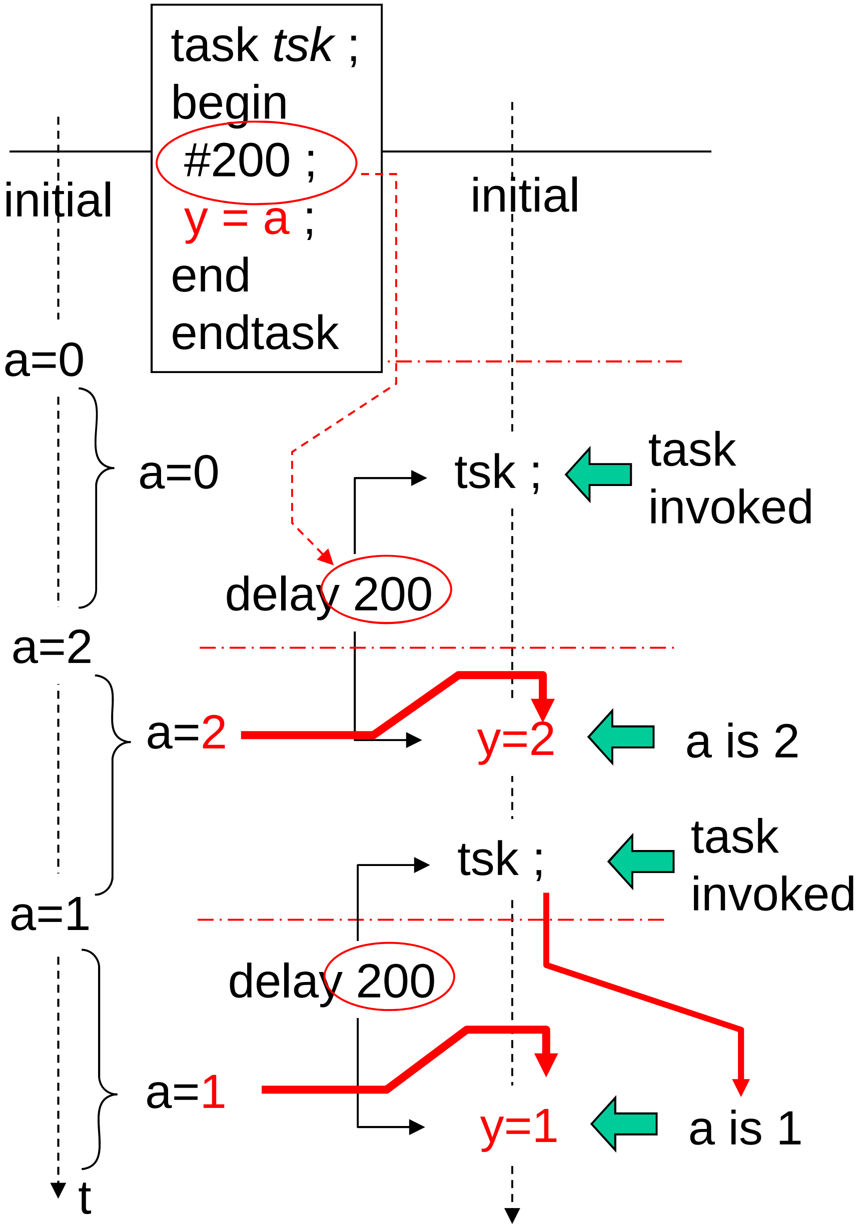


```
⋮
sig_a = b ;
⋮
bus_task ;
⋮
sig_a = c ;
⋮
bus_task ;
⋮
task bus_task ;
⋮
a = sig_a ;
⋮
endtask
```



```
task tsk ;
input a;
begin
 #200 ;
 y = a ;
end
endtask
```

Task with input argument

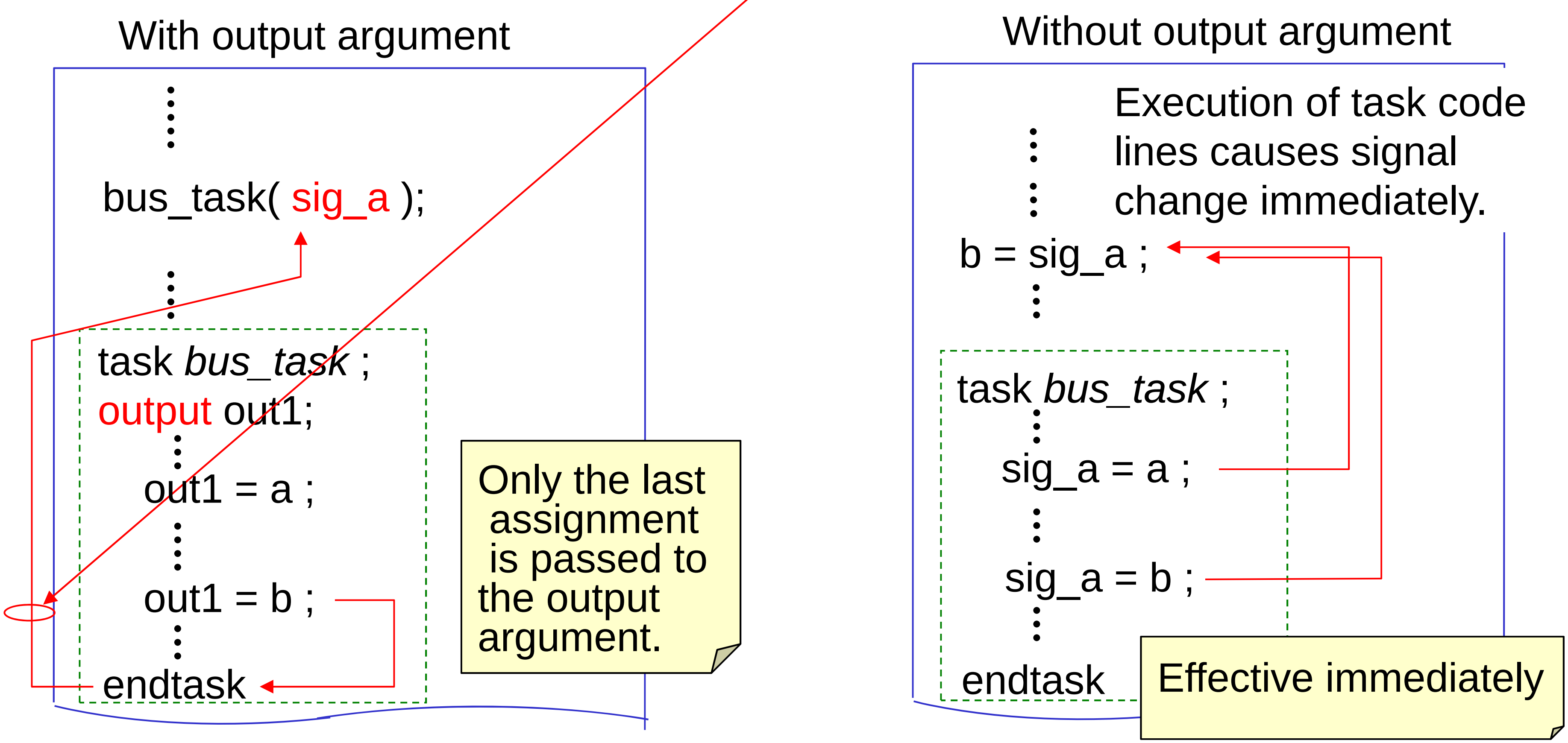


```
task tsk ;
begin
 #200 ;
 y = a ;
end
endtask
```

Task without input argument

With output arguments, outputs are given values when the task is completed.

⇨ Only the last assignment to an output or an inout argument is passed to the corresponding task enabling arguments. If no output argument is given, it refers to external signals (global variables) directly. And the variables are given values when they are assigned values. In such a case, all changes of these variables are effective immediately



With output argument

```
:
bus_task( sig_a );
:
task bus_task ;
output out1;
  :
   out1 = a ;
  :
   out1 = b ;
  :
endtask
```

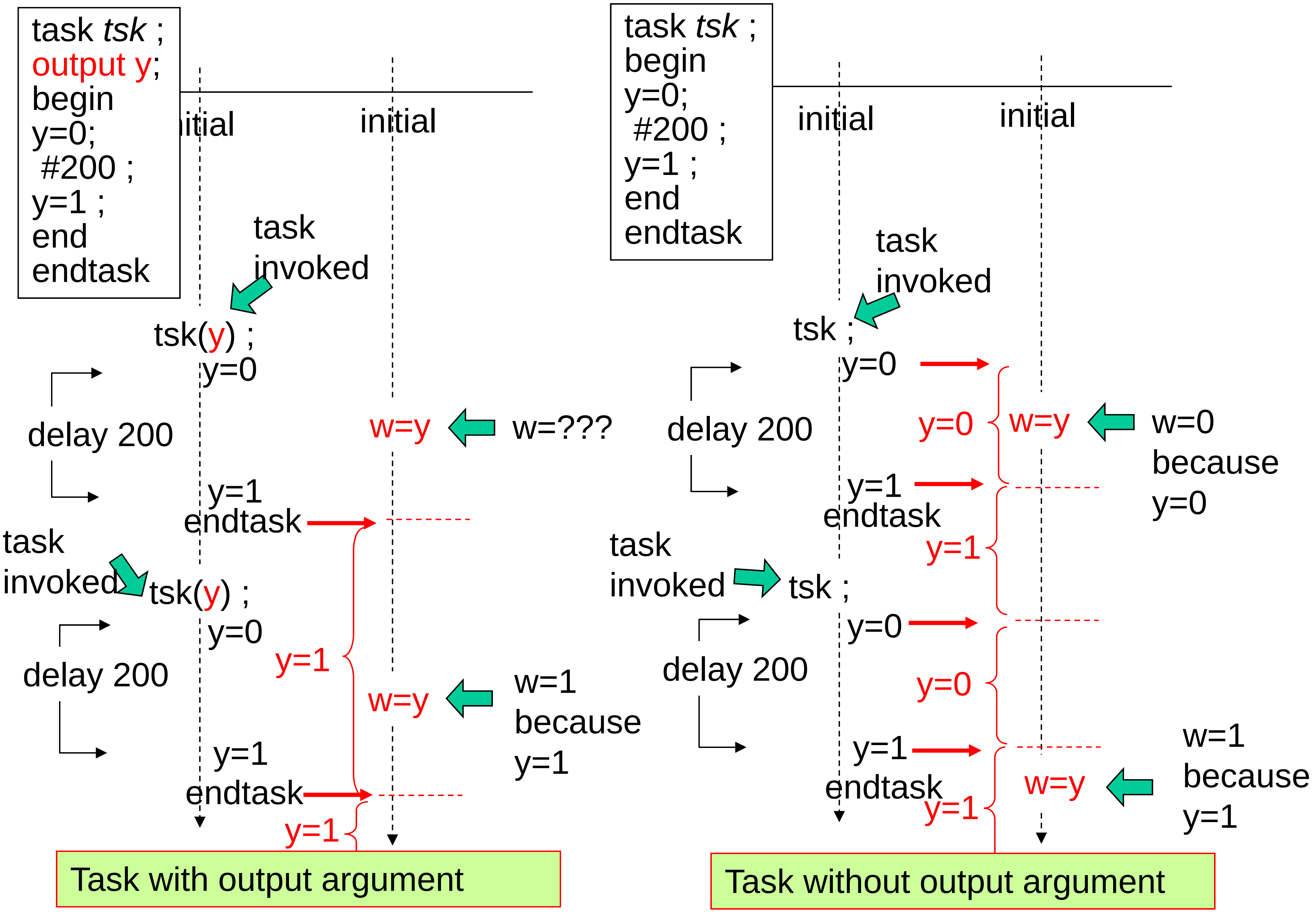
Only the last assignment is passed to the output argument.

Without output argument

Execution of task code lines causes signal change immediately.

```
:
:
b = sig_a ;
:
task bus_task ;
  :
   sig_a = a ;
  :
   sig_a = b ;
  :
endtask
```

Effective immediately



```
task tsk ;
output y;
begin
y=0;
 #200 ;
y=1 ;
end
endtask
```

initial

initial

task invoked

tsk(y) ;

y=0

delay 200

w=y

w=???

y=1

endtask

task invoked

tsk(y) ;

y=0

y=1

delay 200

w=y

w=1 because y=1

y=1

endtask

y=1

Task with output argument

```
task tsk ;
begin
y=0;
 #200 ;
y=1 ;
end
endtask
```

initial

initial

task invoked

tsk ;

y=0

delay 200

y=0

w=y

w=0 because y=0

y=1

endtask

y=1

task invoked

tsk ;

y=0

delay 200

y=0

y=1

endtask

y=1

w=y

w=1 because y=1

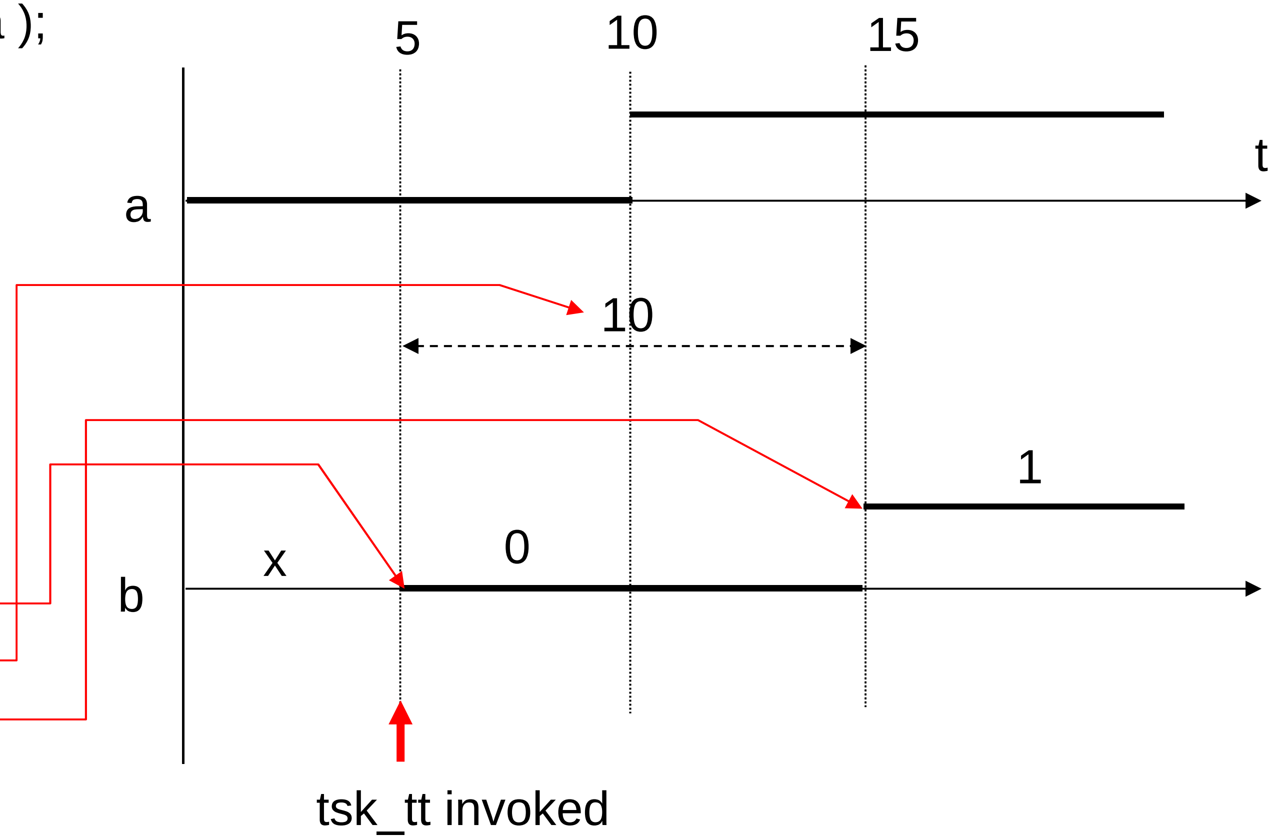
Task without output argument

Q3.9-7: Draw a time chart of b when the following piece of code is executed in RTL simulation.

```
initial begin
  #5 tsk_tt ( a );
end
initial begin
  a = 0 ;
  #10 a = 1;
end

task tsk_tt;
input in_a;
begin
 b = in_a ;
 #10;
 b = ~in_a;
end
endtask
```

Q3.9-7: A sample answer.

```
initial begin
  #5 tsk_tt ( a );
end
initial begin
  a = 0 ;
  #10 a = 1;
end

task tsk_tt;
input in_a;
begin
 b = in_a ;
 #10;
 b = ~in_a;
end
endtask
```

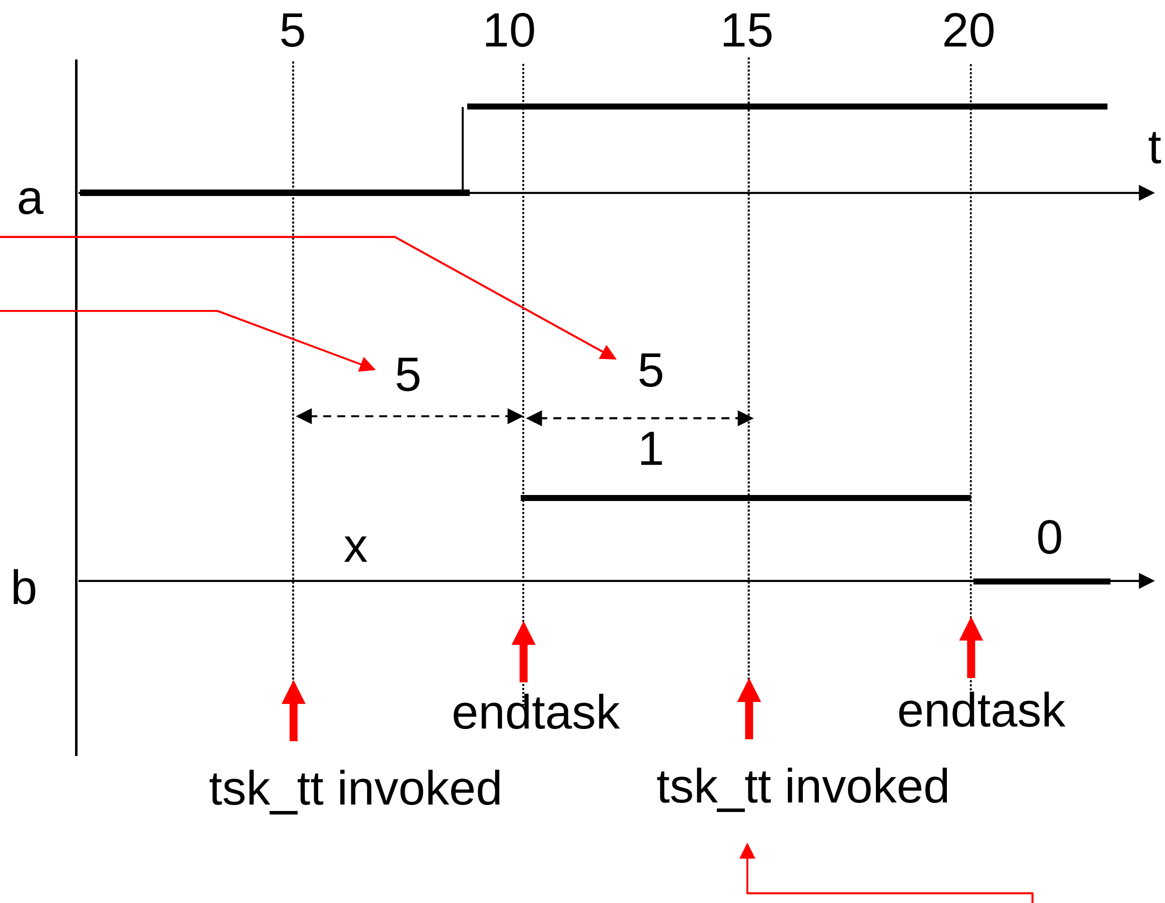


Q3.9-8: Draw a time chart of b when the following piece of code is executed in RTL simulation.

```
initial begin
  #5 tsk_tt ( b );
  #5 tsk_tt ( b );
end
initial begin
  a = 0 ;
  #9  a = 1;
end

task tsk_tt;
output out_b;
begin
 out_b = ~a ;
 #5;
end
endtask
```

Q3.9-8: A sample answer.

```
initial begin
  #5 tsk_tt ( b );
  #5 tsk_tt ( b );
end
initial begin
  a = 0 ;
  #9  a = 1;
end

task tsk_tt;
output out_b;
begin
 out_b = ~a ;
 #5;
end
endtask
```



There is 5 units time delay in tsk_tt, therefore tsk_tt will be invoked at time 15 for the second time.

Example of a task:

```
initial begin
in_a = 1;
w_clk_rise ;
in_a = 3 ;
w_clk_rise ;
in_a = 9 ;
w_clk_rise ;
in_a = 2 ;
w_clk_rise ;
in_a = 4 ;
end
task w_clk_rise ;
   begin
      @ ( posedge clk );
   end
endtask
```

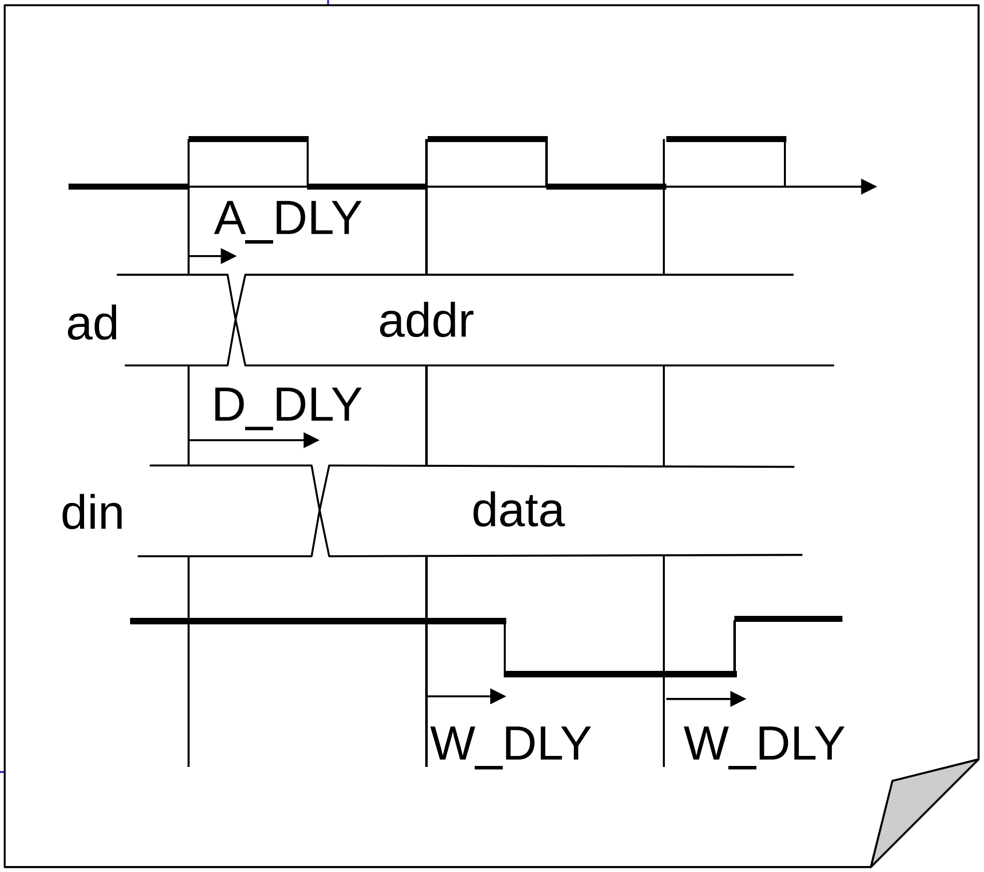
same

```
initial begin
in_a = 1;
@( posedge clk ) ;
in_a = 3 ;
@( posedge clk ) ;
in_a = 9 ;
@( posedge clk ) ;
in_a = 2 ;
@( posedge clk ) ;
in_a = 4 ;
end
```

Example of a task:

Nonblocking procedural assignment is used to execute these two sentences in parallel.

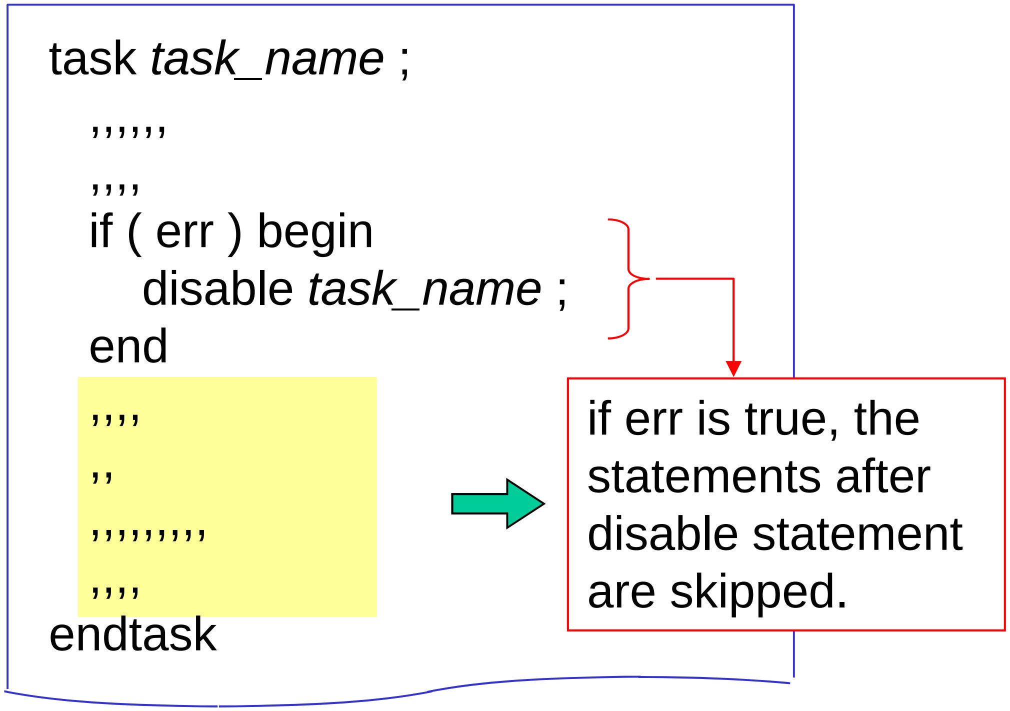
```
task cpu_bus_wrtreg ;
input [3:0] addr ;
input [15:0] data ;
begin
  @ ( posedge clk )
    ad <= #A_DLY addr ;
    din <= #D_DLY data ;
  @ ( posedge clk )
    wr_n <= #W_DLY 1'b0 ;
  @ ( posedge clk )
    wr_n <= #W_DLY 1'b1 ;
end
endtask
```



disabling task

A task can be disabled by disable statement. When a task is disabled, the results of the following activities are not specified.
(a) results of output and inout arguments,
(b) scheduled, but not executed, nonblocking assignments,
(c) procedural continuous assignments.

Disabling task



```
task task_name ;
   ,,,,,,
   ,,,,
   if ( err ) begin
       disable task_name ;
   end
   ,,,,
   ,,
   ,,,,,,,,,
   ,,,,
endtask
```

if err is true, the statements after disable statement are skipped.

Using nonblocking procedural assign is sometimes dangerous, because it behaves different from human instinct.

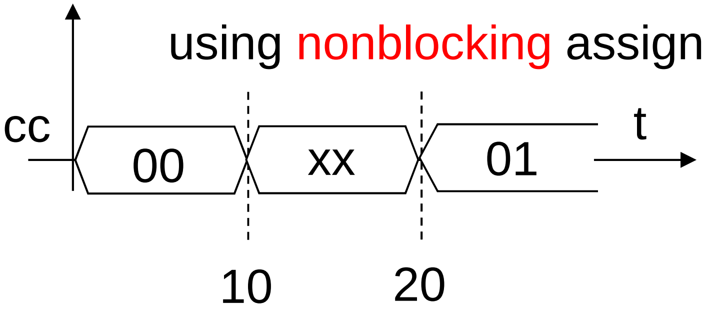
```
task tsk_1 ;
input [1:0] aa;
output [1:0] bb;
begin
  bb <= aa ;
end
endtask
```

```
module tst_nb_assign ;
reg [1:0] cc ;
initial begin
  cc = 2’b00 ;
#10 tsk_1( 2’b01 , cc ); // (1)
#10 tsk_1( 2’b10 , cc ); // (2)
#10 $finish ;
end
endmodule
```

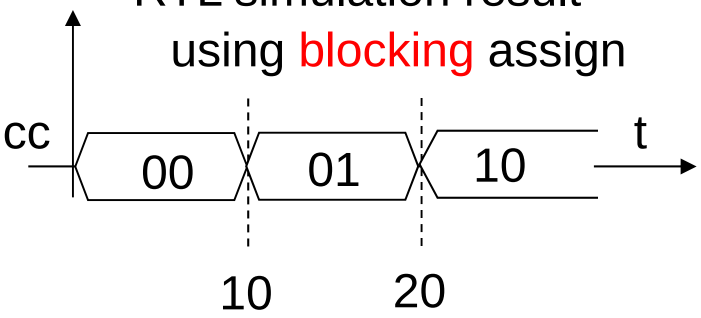
Because nonblocking procedural assign does not block the execution of endtask, when endtask is executed in (1) bb’s value still initial value x. Therefore cc becomes x at time 10. When (2) is executed, bb is already given the value 01 in the previous execution (1). Therefore cc becomes 01 at time 20.

If we change the nonblocking to blocking assign, then the result matches with human instinct.

RTL simulation result using nonblocking assign



RTL simulation result using blocking assign



As you can see in the previous example, using nonblocking procedural assign is dangerous especially combined with control statements as below.

```
 ●
 ●
a<= 0 ;
#10 ;
 ●
 ●
 ●
a <= 1 ;
if (a) begin  (1)  end
else  begin  (2)  end
 ●
 ●
 ●
```

When evaluating a, the value of a may not be 1 but 0 because a <= 1 does not block the execution of if statement. Therefore (2) must be executed even if there is a <= 1 ; before if statement.

system task

A simulator system has several functions called system tasks which can be used in simulation.

Typical system tasks are shown below.

$finish

This ends simulation.

$display (format, signal_name1, signal_name2,)

This displays the arguments' value on a terminal screen when it is called.

$strobe (format, signal_name1, signal_name2,)

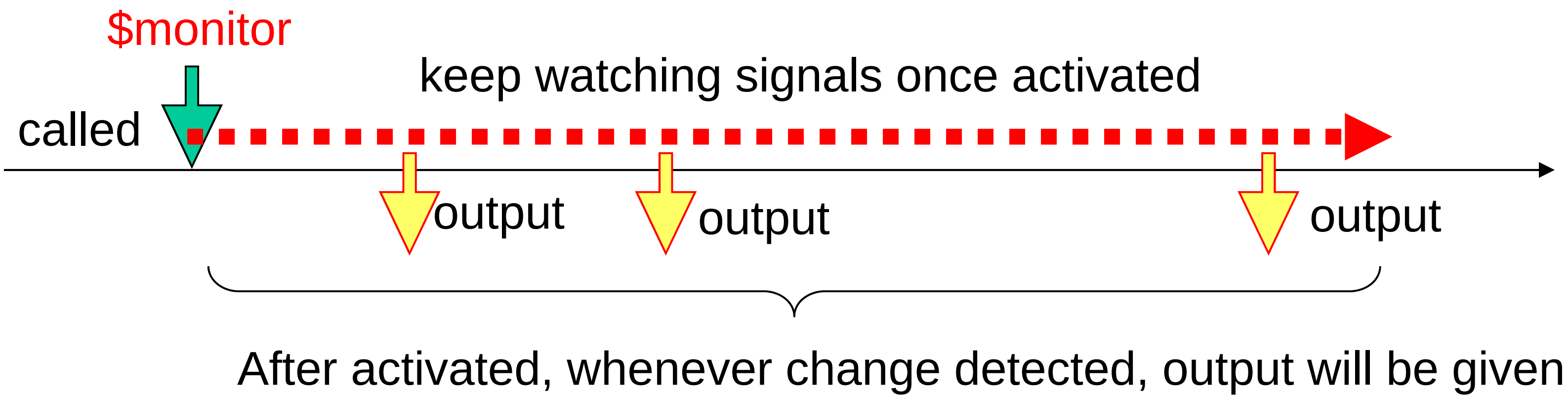
This displays the arguments' value on a terminal screen after their values are updated.

$monitor (format, signal_name1, signal_name2,)

This displays the arguments' value whenever they are updated.

$fmonitor (format, signal_name1, signal_name2,)

This outputs into a file.



$monitor

keep watching signals once activated

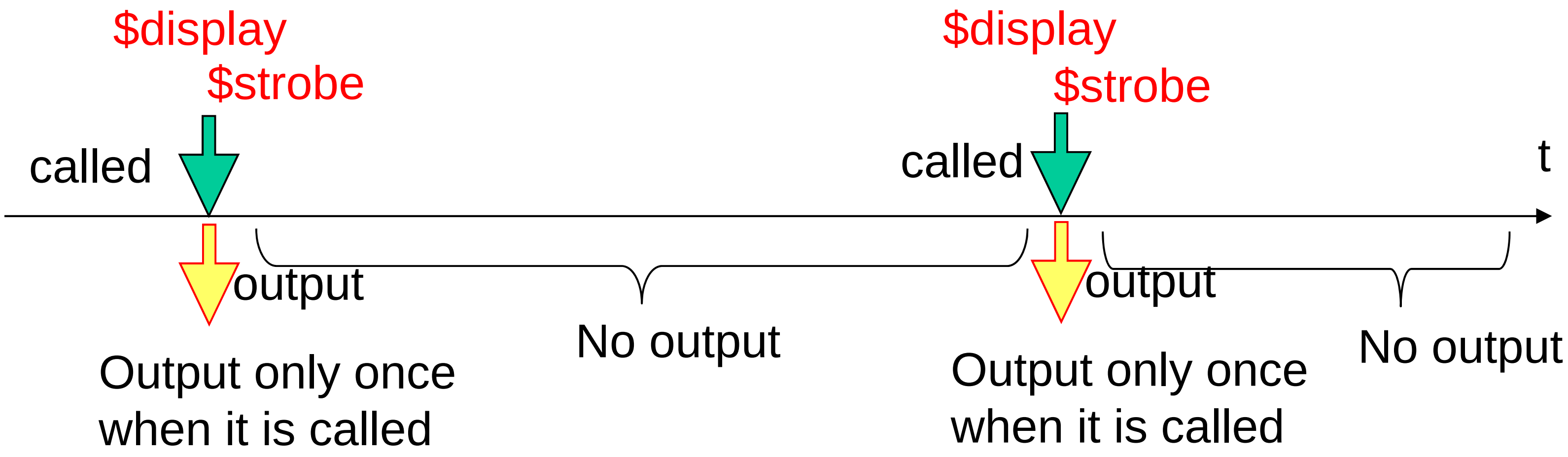
called

output

output

output

After activated, whenever change detected, output will be given



$display

$strobe

called

$display

$strobe

called

t

output

No output

output

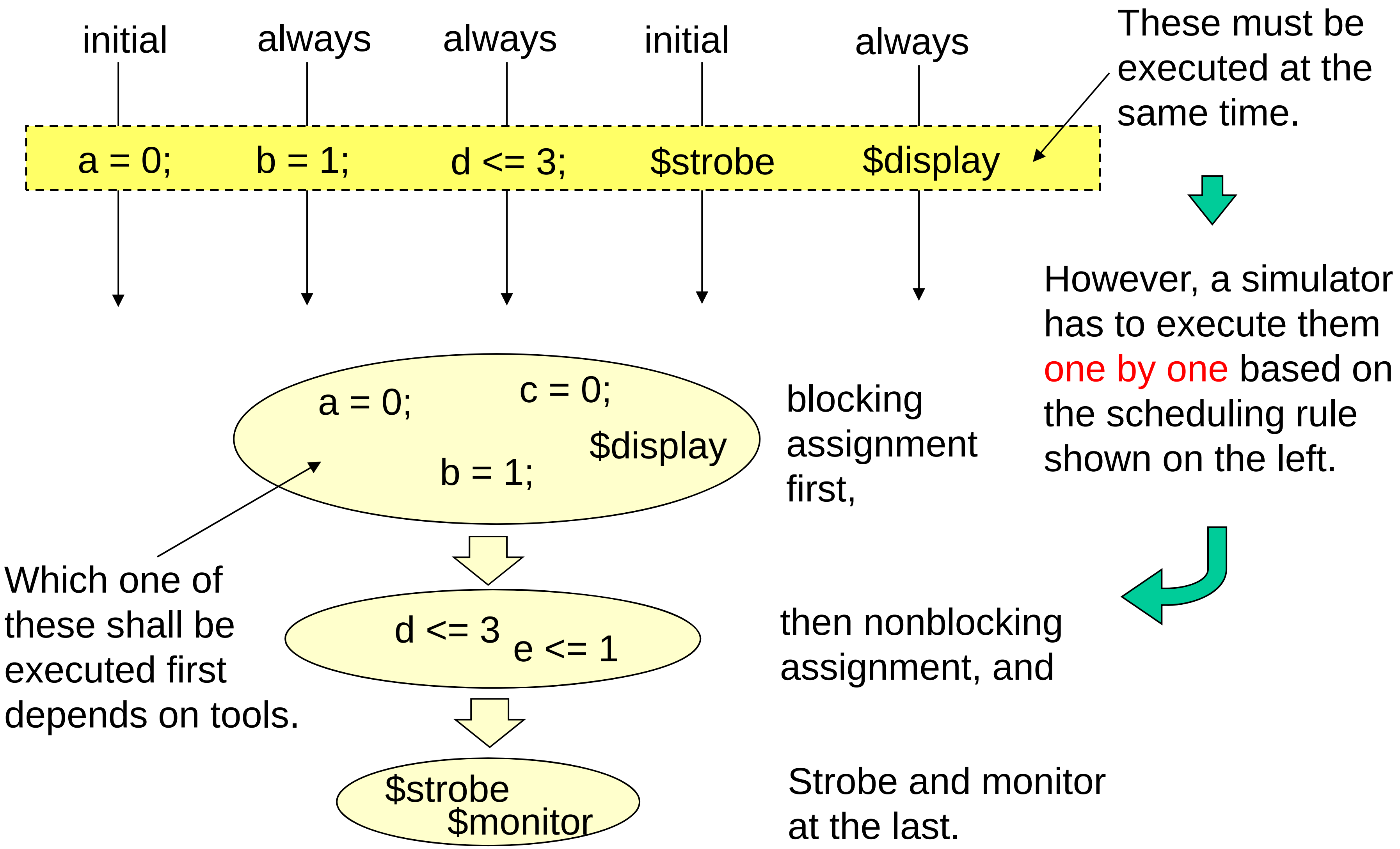
No output

Output only once
when it is called

Output only once
when it is called

$strobe and $display have almost the same functionality except that $strobe is scheduled to be executed at the last stage of the time.

While simulating RTL code, a simulator has to process many events simultaneous at certain time slot.



These must be executed at the same time.

However, a simulator has to execute them one by one based on the scheduling rule shown on the left.

blocking assignment first,

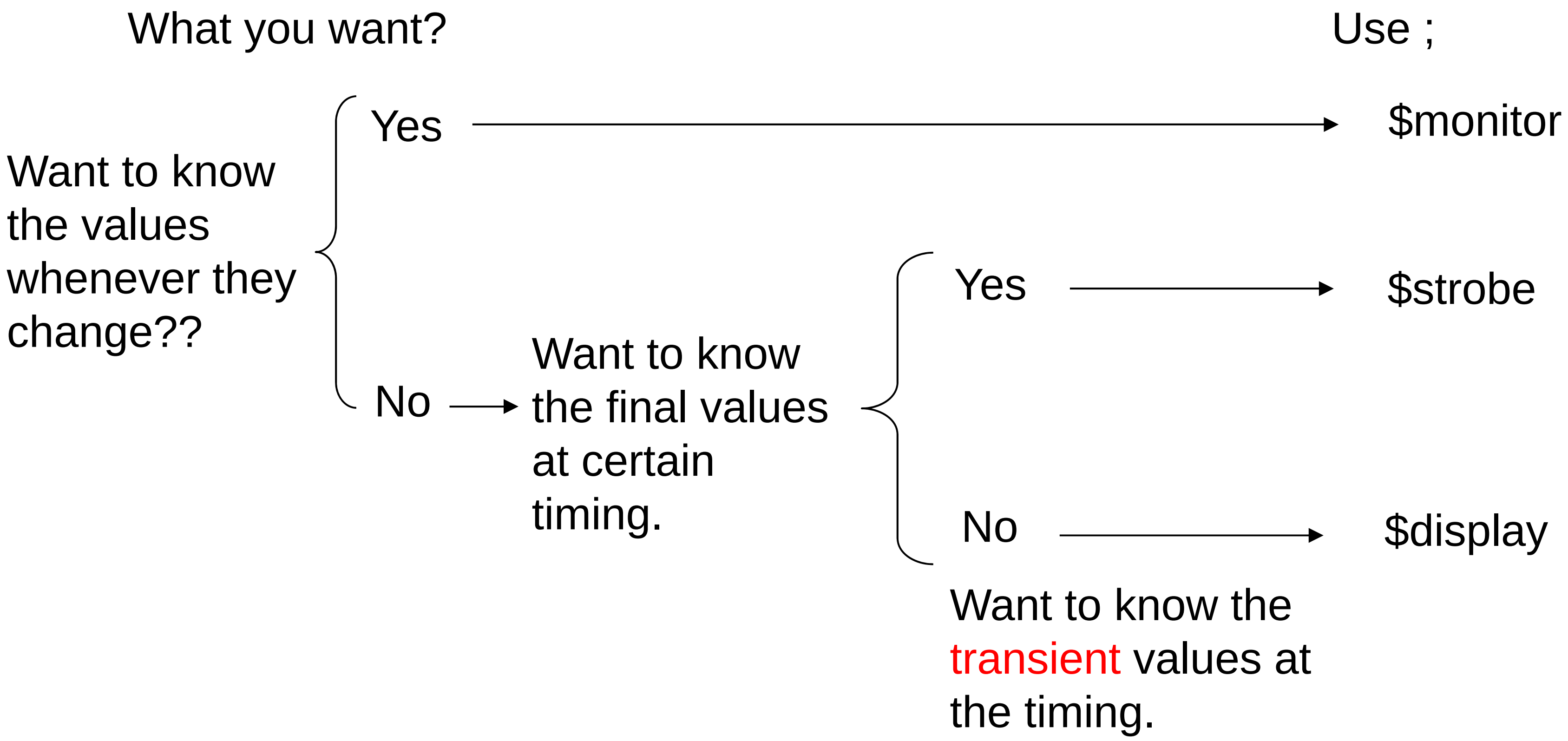
then nonblocking assignment, and

Strobe and monitor at the last.

Which one of these shall be executed first depends on tools.

Because of the scheduling rule on the previous page (see 5.1 for detail), $strobe or $monitor can show the final values of signals, the values after they become stable, at certain time spot.

On the contrary, $display can be scheduled at any timing depending when you invoke it and depending on tools.



What you want?

Use ;

Want to know the values whenever they change??

Yes → $monitor

No → Want to know the final values at certain timing.

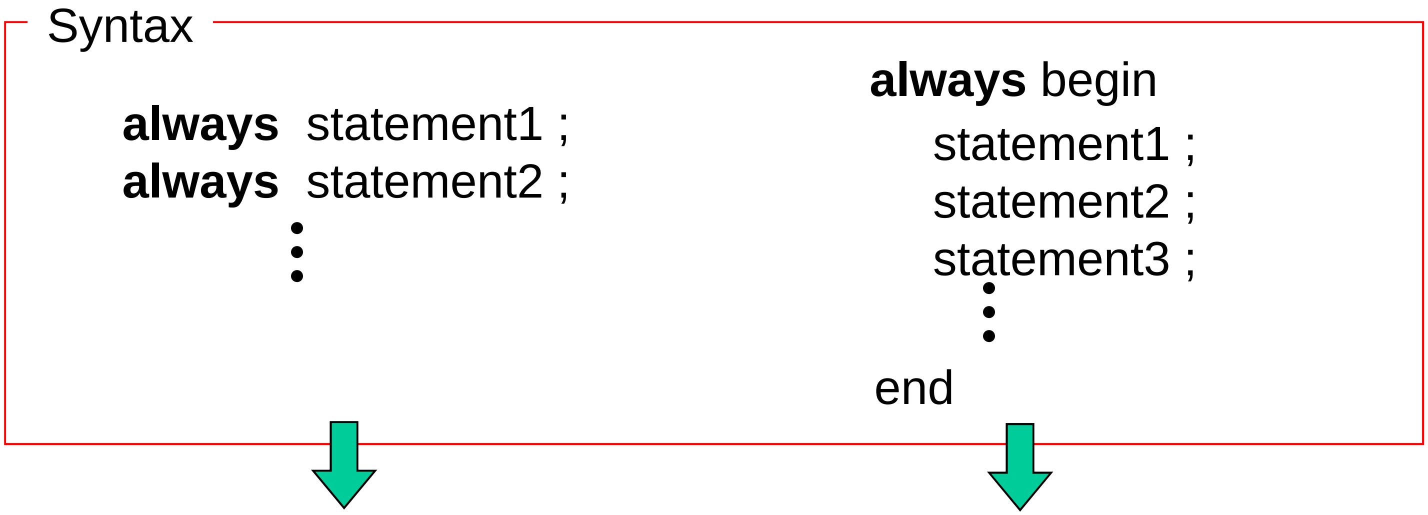
Yes → $strobe

No → $display

Want to know the transient values at the timing.

3.9.7 Always construct

The always constructs are enabled at the beginning of a simulation, at time 0. They shall execute repeatedly. Their activity shall cease only when the simulation is terminated.



Syntax

```
always  statement1 ;
always  statement2 ;
           :
```

```
always begin
   statement1 ;
   statement2 ;
   statement3 ;
      :
end
```

statement1, *statement2*, *statement3*, and so on are enabled at time 0 and they are repeatedly executed. Which one is to be executed first is tool or vendor dependent.

statement1 is executed at time 0, statement2 is executed next to the statement1, and so on. When end reached, statement1 ,2,3,,, are repeated again. This continues forever.

always construct

	explanation	comment
input / output arguments	No arguments. References are done directly to the external variables.	
assignment	Both blocking procedural assignment and nonblocking procedural assignment are allowed. Use blocking procedural assignment to describe combinational logic. Use nonblocking procedural assignment to describe flip-flop and latch.	LHS must be variable data type.
timing control	An always construct must contain time-controlling statements. Memory element can be created.	# and @ are mandatory.
simulation	Executed repeatedly.	
synthesis	synthesizable / not synthesizable	depend on the statements.
others	tasks and functions can be enabled in an always construct.	

Restrictions

[1] An always construct must be written inside a module.
It must not be written (declared) in structured procedures.

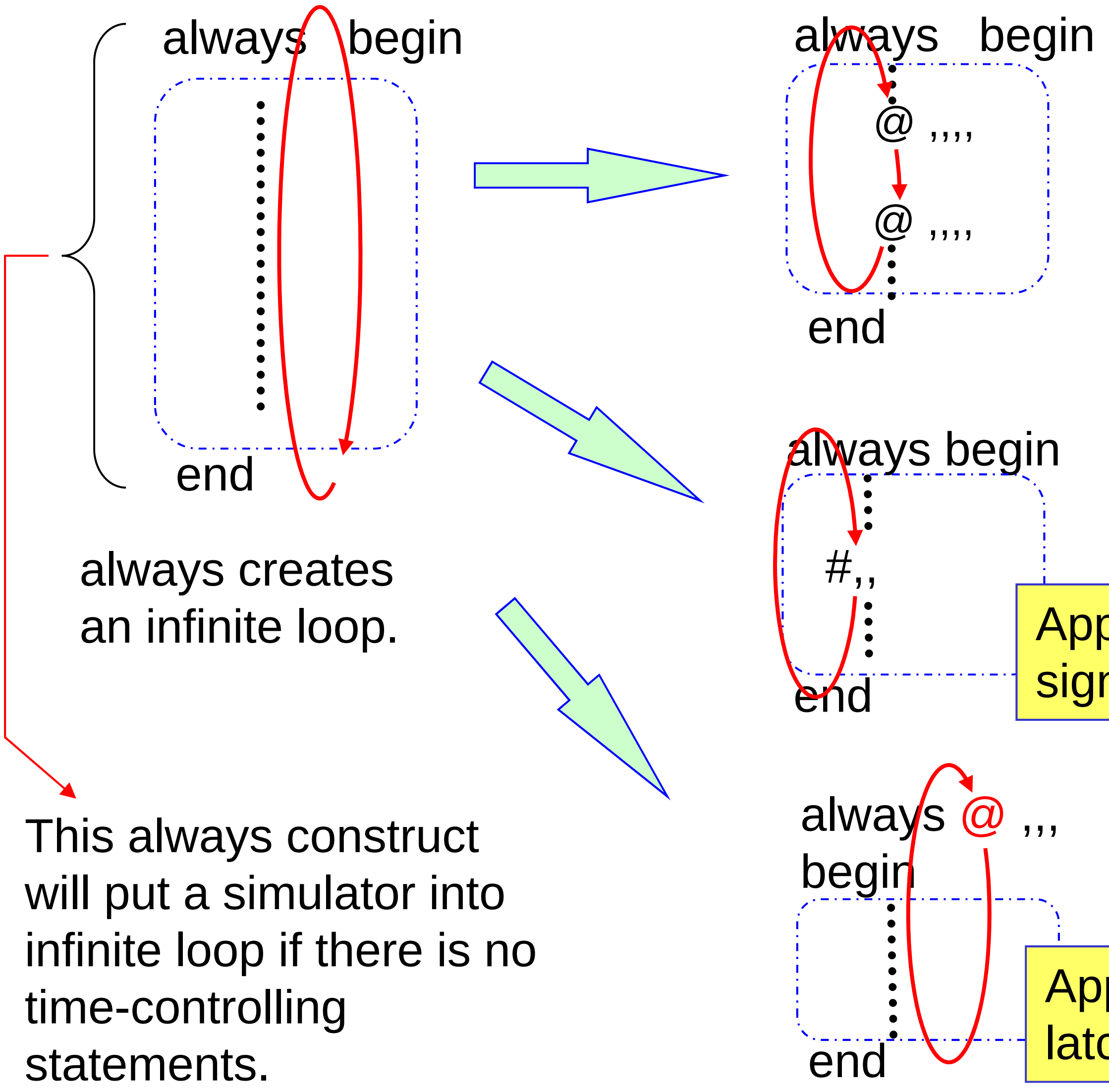
[2] Do not write the same LHS in different procedures.

The same variable sig_y is given a value in different always constructs.
A simulator may not reject this code, but a synthesis tool will reject this code.

```
always @ ( ,,,,, ) begin

  sig_y = ----- ;

end
        :
always @ (,,,, ) begin

  sig_y = ----- ;

end
```

Never define a variable in multiple always constructs.
Do not assign a value to the same variable from different structured procedures such as from initial construct and from always construct, or from initial construct and from another initial construct.

Typical structures of always construct



always creates an infinite loop.

This always construct will put a simulator into infinite loop if there is no time-controlling statements.

This style is not synthesizable.

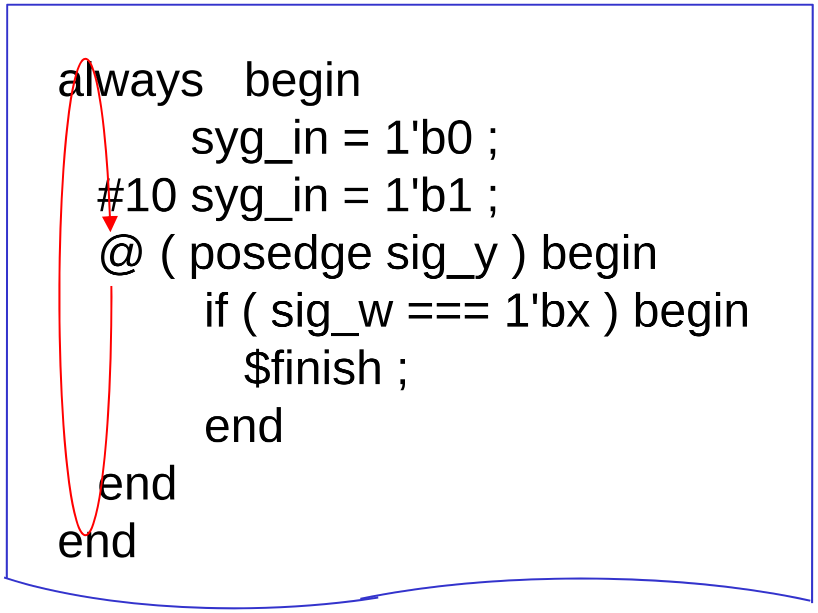
This style is not synthesizable.

Applicable to create a cyclic signal such as clock.

This style is synthesizable

Applicable to create a flip-flop, latch, or combinational logic.

Example1 ;



```
always   begin
          syg_in = 1'b0 ;
   #10 syg_in = 1'b1 ;
   @ ( posedge sig_y ) begin
           if ( sig_w === 1'bx ) begin
              $finish ;
           end
   end
end
```

An example on the left place 1'b0 on syg_in and 10 unit time later place 1'b1 on sig_in, and wait for sig_y changes from 0 to 1.

When sig_y changes from 0 to 1, it checks if sig_w is x or not. If it is x then stops simulation, else it repeats the same procedure again.

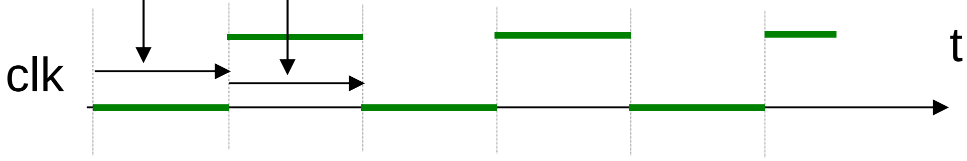
This code is not synthesizable.

Example2 ;   This is a recommended style of a simple clock creation logic.

```
parameter HALF_CYCLE = 500 ;

always   begin
   clk = 1'b0 ;  # HALF_CYCLE ;
   clk = 1'b1 ;  # HALF_CYCLE ;
end
```

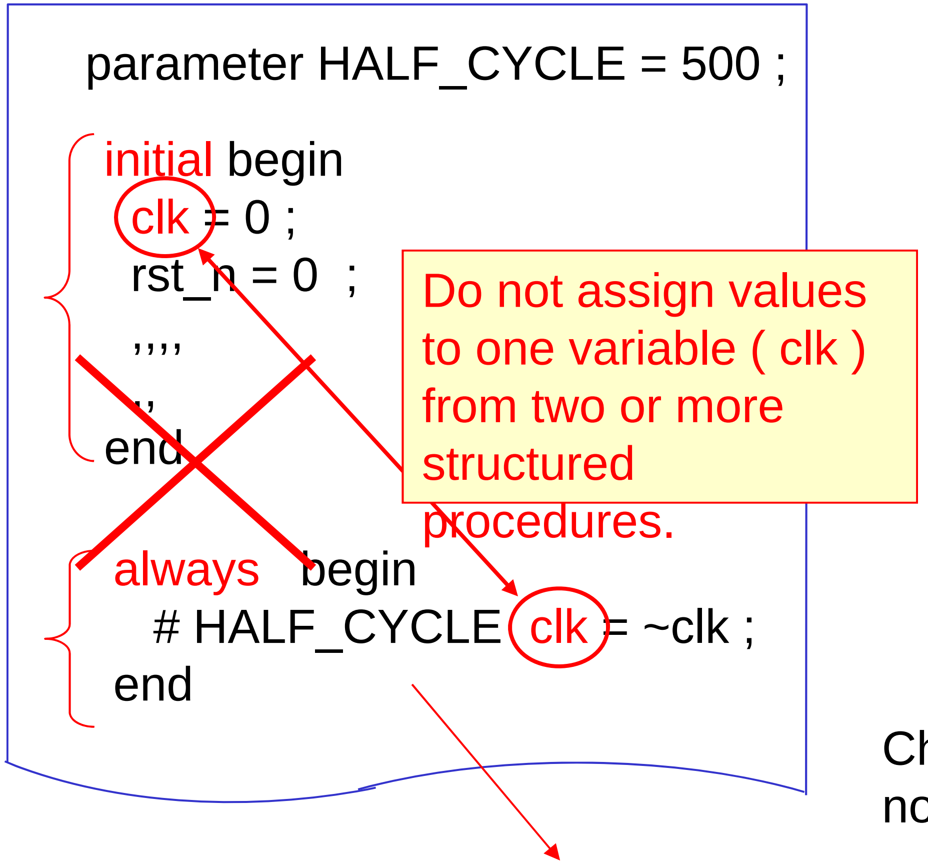
This code is not synthesizable.

The example on the left generates a clock signal named clk having 500*2 time units cycle time .



Do not make a clock signal rise at t=0, if you set clk=1 at t=0 then it means clock rises (from x to 1) at t=0.

The code below is an example of clock signal creation logic which is not recommended. Do not use the code below.



```
parameter HALF_CYCLE = 500 ;

initial begin
  clk = 0 ;
  rst_n = 0  ;
  ,,,,
  ,,
end

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

Do not assign values to one variable (clk) from two or more structured procedures.

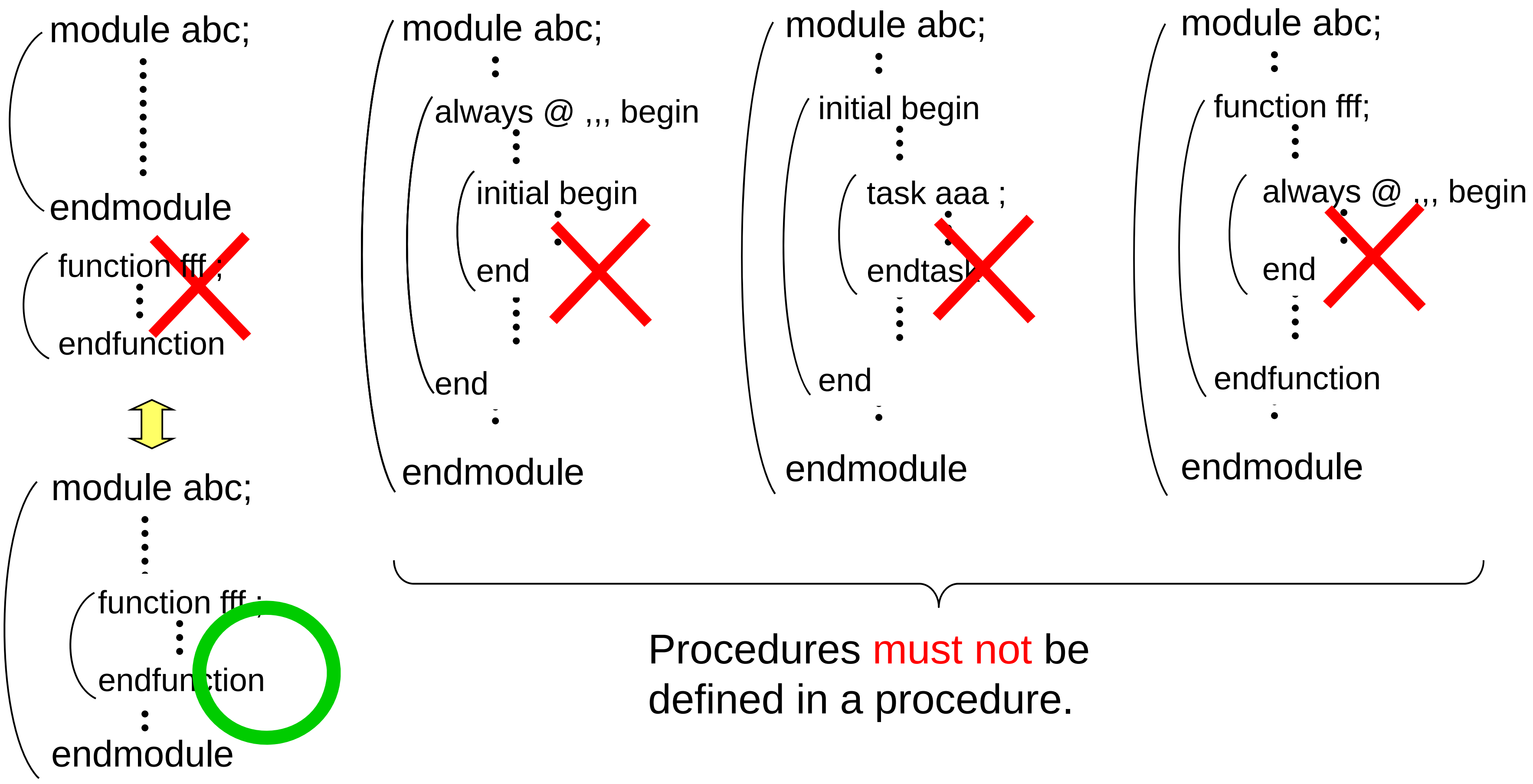
The code on the left does not cause any syntax error. However, you must be able to see potential problems of assigning values to one variable from more than one structured procedure.

Changing always to initial does not help improving the problem.

```
initial   begin
   forever # HALF_CYCLE  clk = ~clk ;
end
```

3.9.8 Structured procedure summary

(1) Where to write procedures



```
module abc;
   ⋮
endmodule
function fff ;
   ⋮
endfunction
```

```
module abc;
   ⋮
  function fff ;
     ⋮
  endfunction
   ⋮
endmodule
```

```
module abc;
   ⋮
  always @ ,,, begin
     ⋮
    initial begin
       ⋮
    end
     ⋮
  end
   ⋮
endmodule
```

```
module abc;
   ⋮
  initial begin
     ⋮
    task aaa ;
       ⋮
    endtask
     ⋮
  end
   ⋮
endmodule
```

```
module abc;
   ⋮
  function fff;
     ⋮
    always @ ,,, begin
       ⋮
    end
     ⋮
  endfunction
   ⋮
endmodule
```

Procedures must not be defined in a procedure.

Procedures must be defined in a module.

(2) How to write procedures

```
module abc;
      :
  always @ ,,, begin
      :
      :
    mod_efg mod_efg_01(,,, ) ;
      :
      :
  end
      :
endmodule
```

Module instantiation is not allowed in a procedure.

⇩

Instantiate modules outside of procedures.

```
module abc;
      :
  always @ ,,, begin
      :
      :
    assign y = expression ;
      :
      :
  end
      :
endmodule
```

Continuous assign is not allowed in a procedure.

⇩

Use procedural assignments.

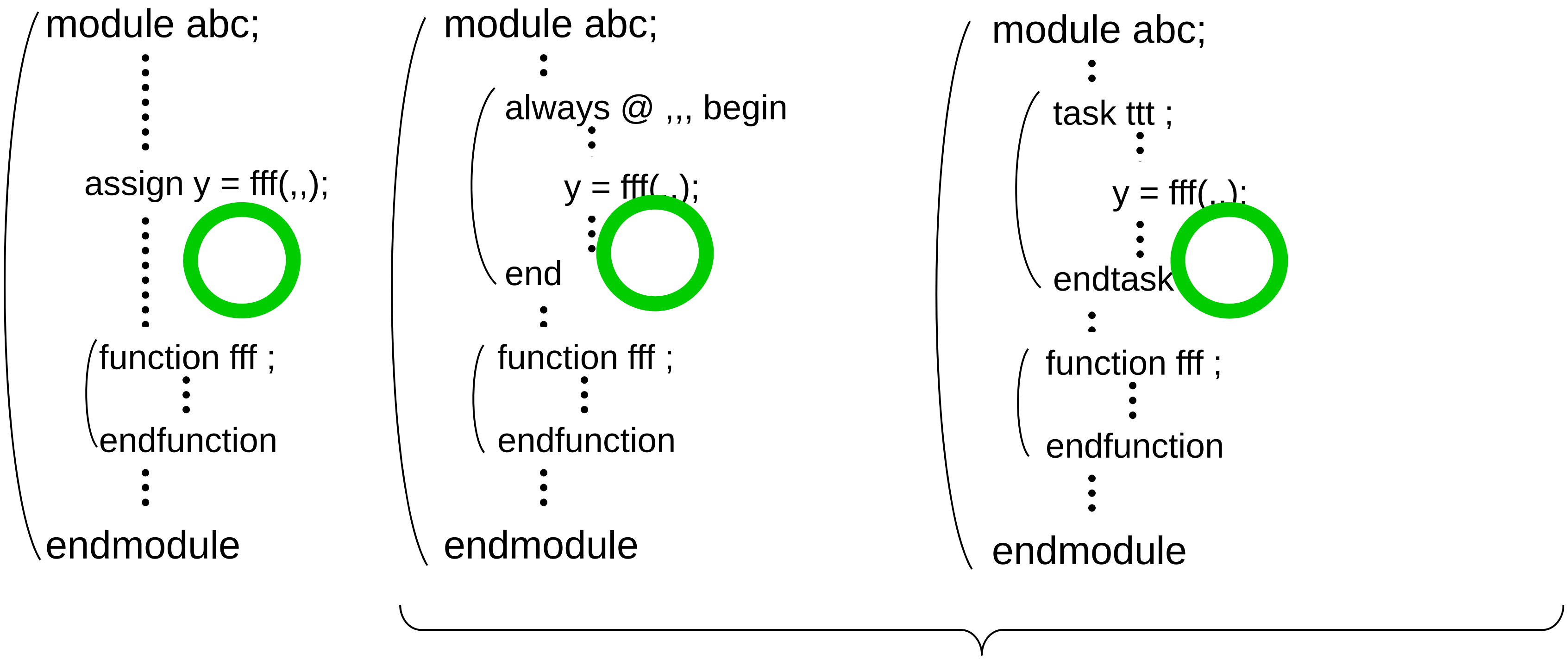
```
module abc;
      :
  wire abc ;
      :
    task ttt ;
      :
      abc = expression;
      :
    endtask
      :
endmodule
```

A net can not be on LHS in a procedure.

⇩

Declare all LHS as reg data type.

(3-1) Where to invoke a procedures: function.



```
module abc;
  ⋮
  assign y = fff(,,);
  ⋮
  function fff ;
    ⋮
  endfunction
  ⋮
endmodule
```

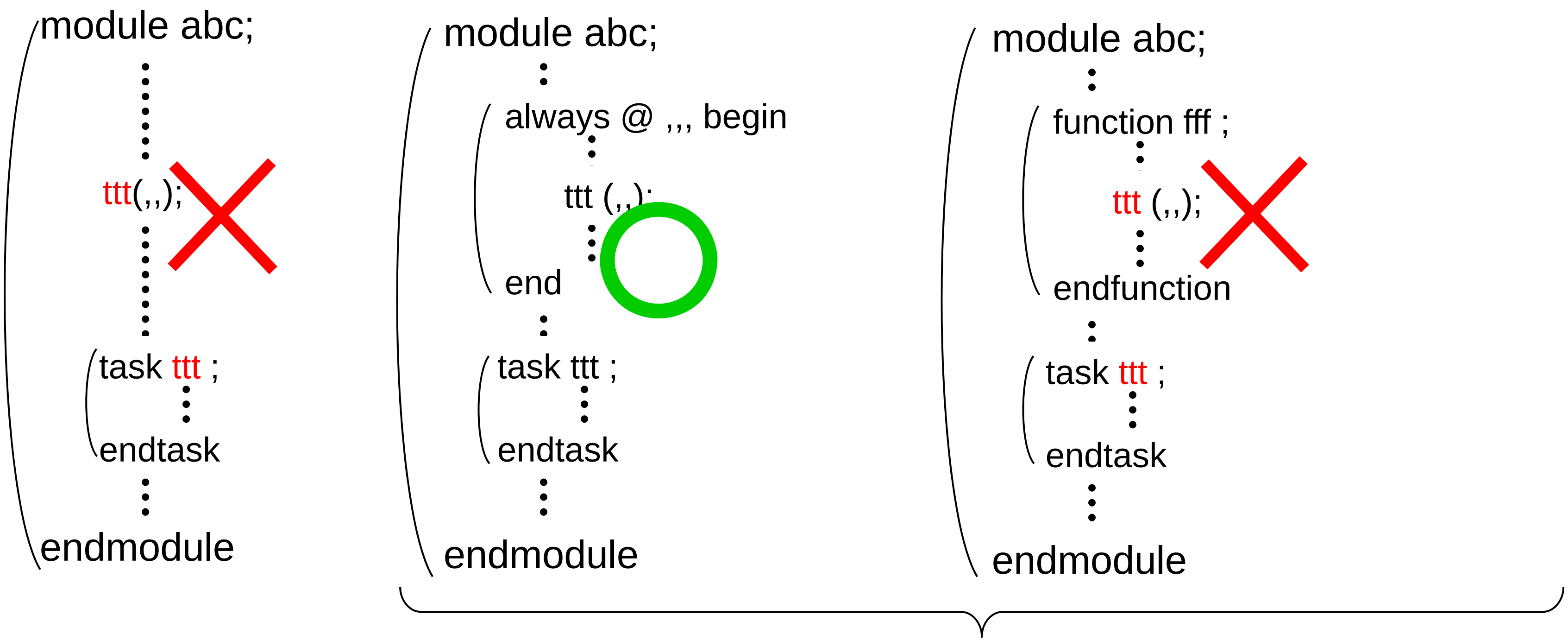
```
module abc;
  ⋮
  always @ ,,, begin
    ⋮
    y = fff(,,);
    ⋮
  end
  ⋮
  function fff ;
    ⋮
  endfunction
  ⋮
endmodule
```

```
module abc;
  ⋮
  task ttt ;
    ⋮
    y = fff(,,);
    ⋮
  endtask
  ⋮
  function fff ;
    ⋮
  endfunction
  ⋮
endmodule
```

A function can be invoked in RHS of a continuous assignment.

A function can be invoked in RHS of any kind of procedures: task, function, always and initial.

(3-2) Where to invoke a procedures: task.



```
module abc;
  ⋮
  ttt(,,);
  ⋮
  task ttt ;
    ⋮
  endtask
  ⋮
endmodule
```

```
module abc;
  ⋮
  always @ ,,, begin
    ⋮
    ttt (,,);
    ⋮
  end
  ⋮
  task ttt ;
    ⋮
  endtask
  ⋮
endmodule
```

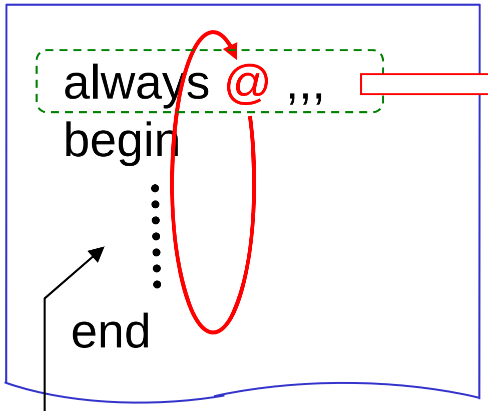
```
module abc;
  ⋮
  function fff ;
    ⋮
    ttt (,,);
    ⋮
  endfunction
  ⋮
  task ttt ;
    ⋮
  endtask
  ⋮
endmodule
```

A task can not be invoked outside of a procedure.

A task can be invoked in always, initial and task, but can not be invoked in functions.

3.10 Always construct in detail

To create logic which is synthesizable, we have to use the structure shown below.



No timing control between begin and end.

always @ (.........)

This is called sensitivity list.

event	timing
posedge clk	when clk rises
negedge rst_n	when rst_n falls
a or b	when a or b change

level signal
change value

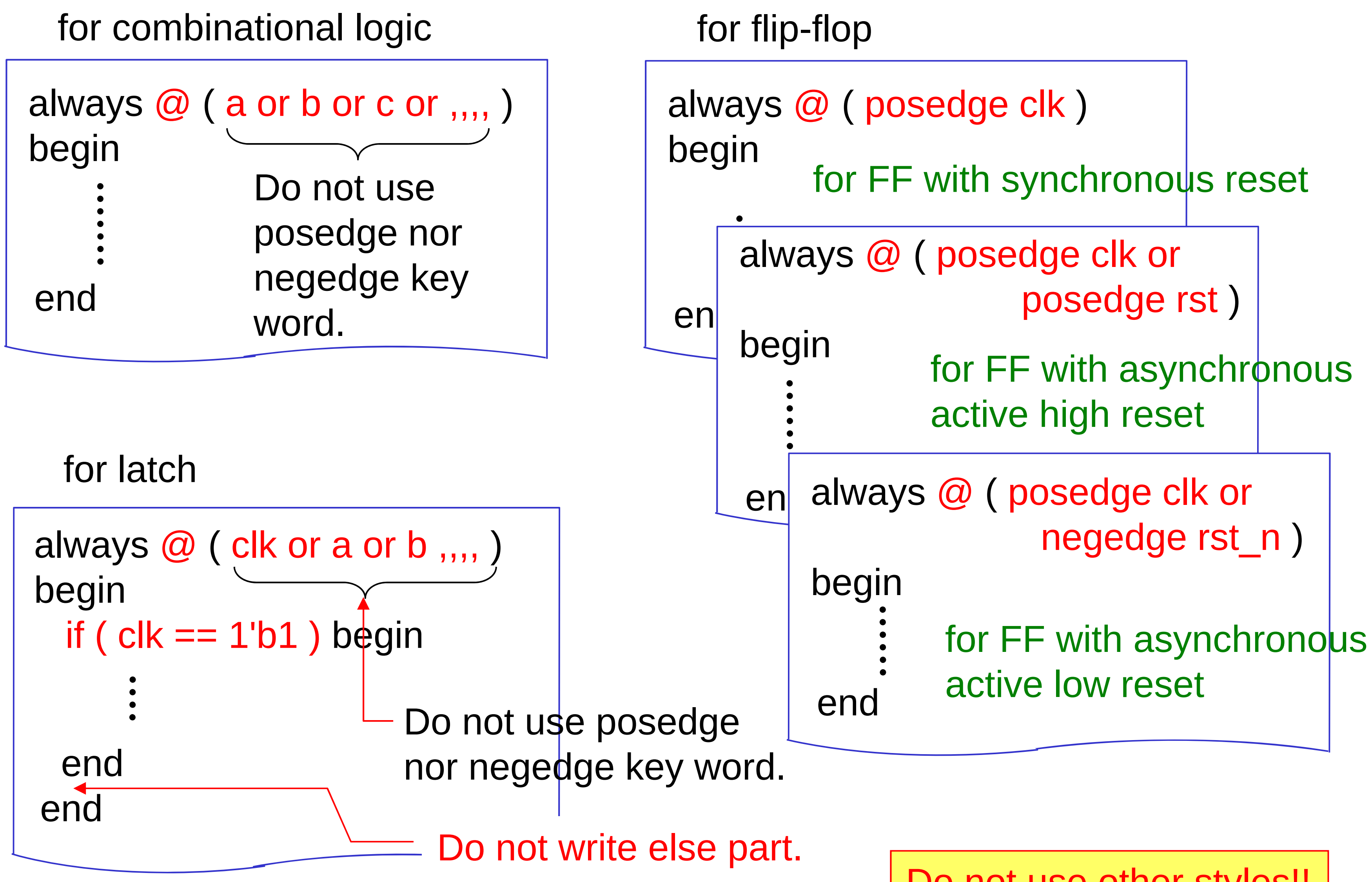
edge signal
change direction

We can write as many events as we want in a sensitivity list. But to be synthesizable, do not mix change value event and change direction event in it.

always @ (~~sig_a or posedge~~ clk)

This will be rejected by a synthesis tool.

The following styles are recommended for always construct to be synthesizable.



for combinational logic

```
always @ ( a or b or c or ,,,, )
begin
    ⋮
end
```

Do not use posedge nor negedge key word.

for flip-flop

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

for FF with synchronous reset

```
always @ ( posedge clk or
                   posedge rst )
begin
    ⋮
en
```

for FF with asynchronous active high reset

```
always @ ( posedge clk or
                   negedge rst_n )
begin
    ⋮
end
```

for FF with asynchronous active low reset

for latch

```
always @ ( clk or a or b ,,,, )
begin
   if ( clk == 1'b1 ) begin
        ⋮
   end
end
```

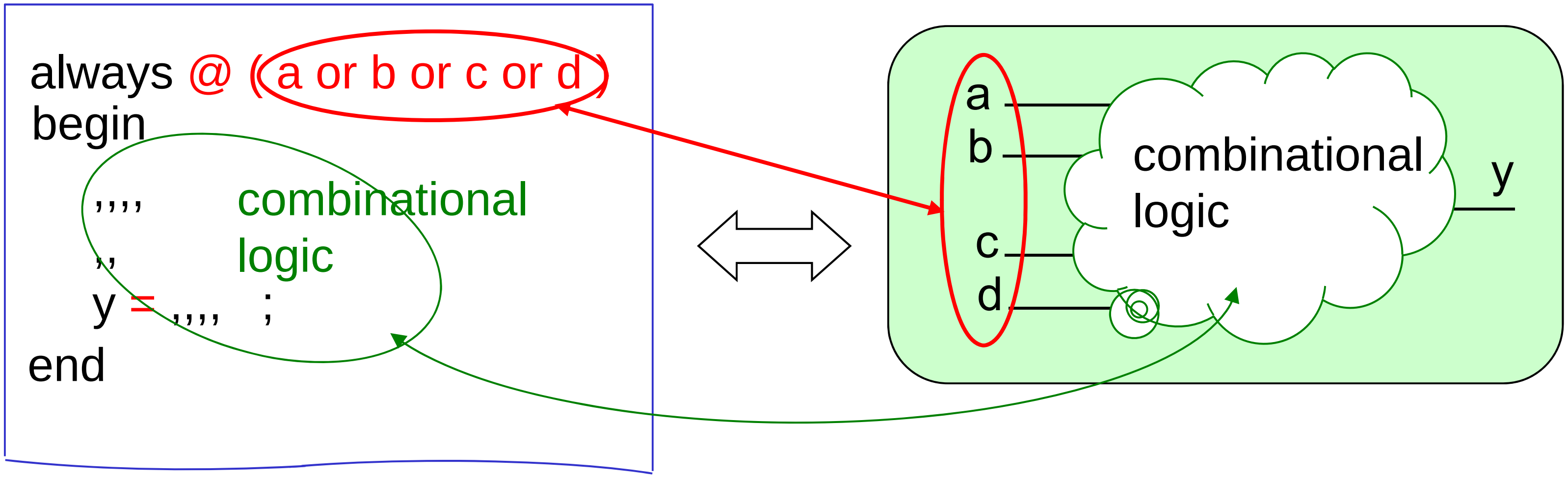
Do not use posedge nor negedge key word.

Do not write else part.

Do not use other styles!!

3.10.1 Always for combinational logic

We can define combinational logic by using the always construct shown below.



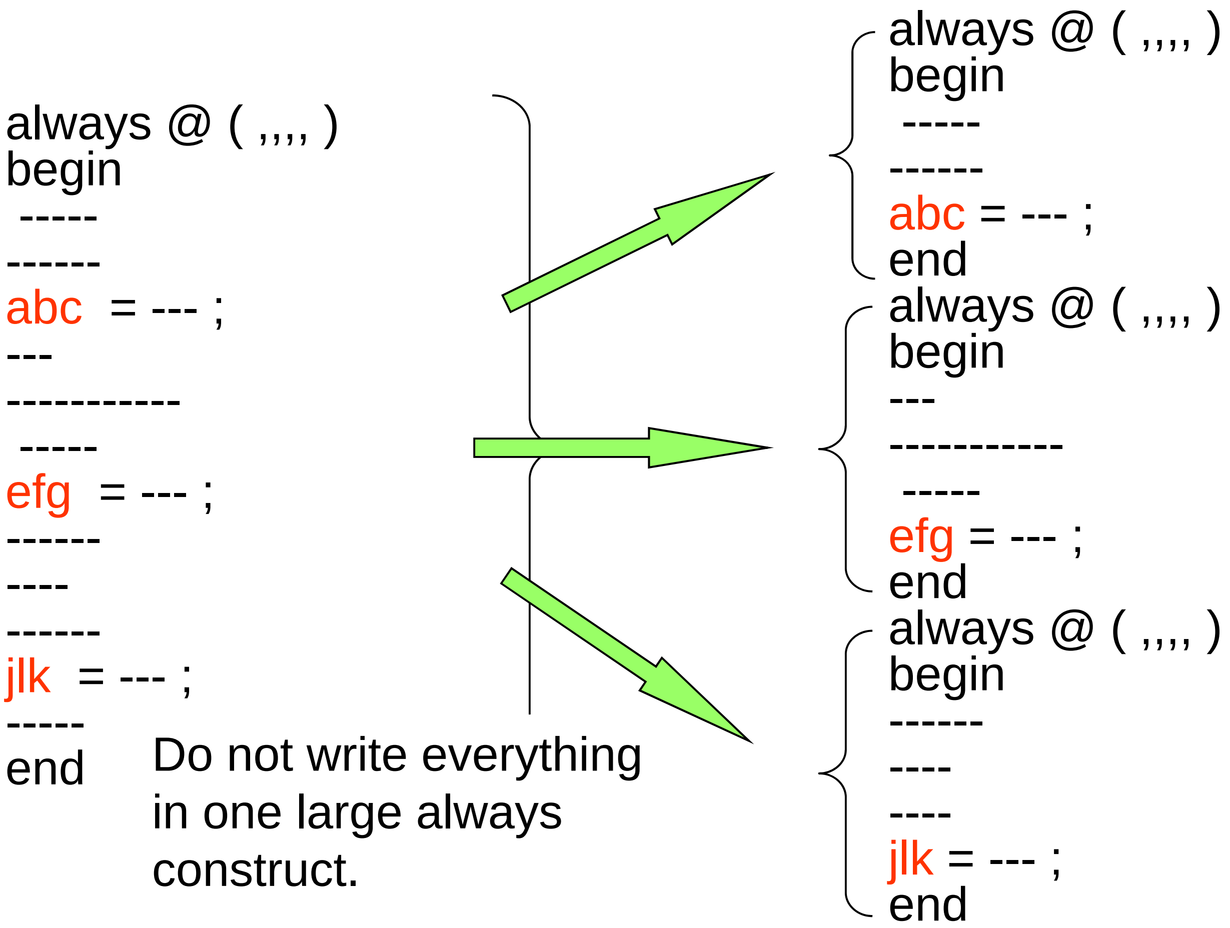
```
always @ (a or b or c or d)
begin
    ,,,,
    ,,
    y = ,,,,   ;
end
```

Whenever a, b, c, or d, which is the input to the logic, is updated, the logic in always construct is executed because these signals are written in the sensitivity list. Therefore, whenever input signals change the output signal y is updated.

This means that no memory element is needed to realize the logic.
And therefore, this always construct can create combinational logic.

One always construct can define many output variables as shown on the left below. However, defining many variables in one big always construct is not recommended. Because it makes logic complicated and debugging difficult.

Divide it into small always blocks which define only one output.



```
always @ ( ,,,, )
begin
 -----
------
abc  = --- ;
---
-----------
 -----
efg  = --- ;
------
----
------
jlk  = --- ;
-----
end
```

```
always @ ( ,,,, )
begin
 -----
------
abc = --- ;
end
always @ ( ,,,, )
begin
---
-----------
 -----
efg = --- ;
end
always @ ( ,,,, )
begin
------
----
----
jlk = --- ;
end
```

Do not write everything in one large always construct.

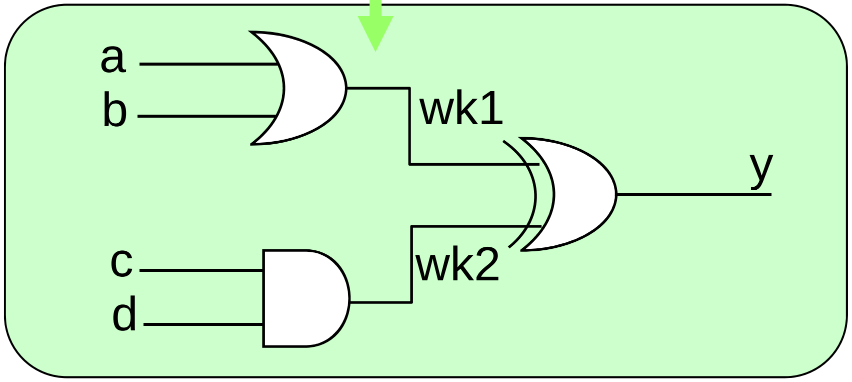
Use one always construct for one variable.

Do not use nonblocking procedural assignment for combinational logic.

Style1

```
always @ ( a or b or c or d )
begin
    wk1 <= a | b ;
    wk2 <= c & d ;
    y     <= wk1 ^ wk2 ;
end
```

When synthesized, this code will be mapped into the gate shown below.



However, as we have seen in 3.8.2, the above code will behave as if there are latches when simulated. Because, by the nature of nonblocking procedural assignment when y is assigned its value, wk1 and wk2 are not updated yet.

Therefore, simulation result of RTL and gate net list may be different if nonblocking procedural assignment is used.

Do not use nonblocking procedural assignment for combinational logic.(continued)

One way to avoid the problem is to list all the RHS into the sensitivity list as Style2.

Style2

```
always @ ( a or b or c or d or wk1 or wk2 )
begin
    wk1 <= a | b ;
    wk2 <= c & d ;
    y     <= wk1 ^ wk2 ;
end
```

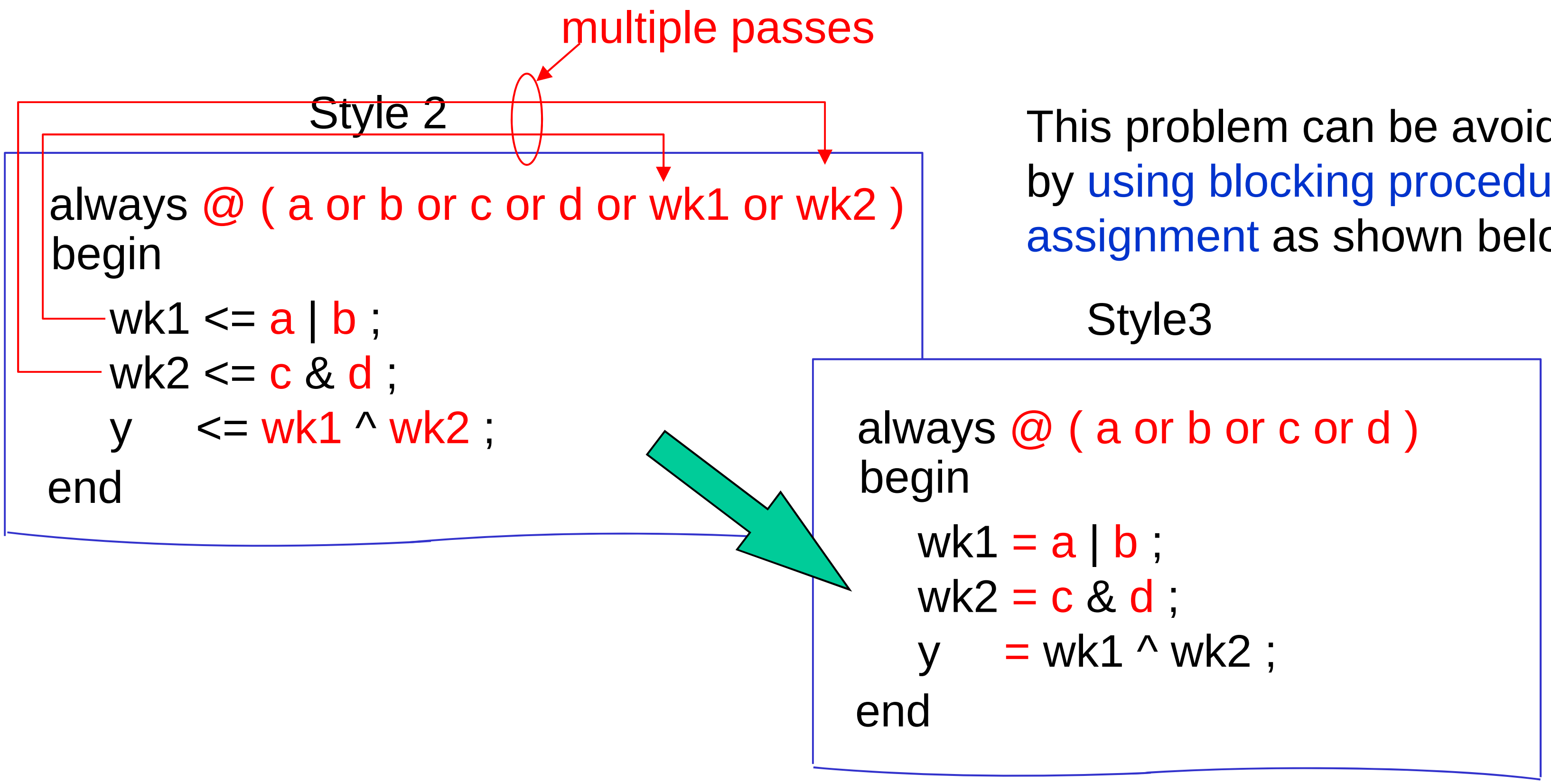
With wk1 and wk2 in the sensitivity list, the sequential block is executed whenever wk1 or wk2 change their value. Therefore, Style2 can be simulated same to the net list.

Run the code below and see how many times the sequential block is executed.

```
module tst_list ;
reg a, b, c, d, y ; // non-FF
reg wk1, wk2 ; // non-FF
always @ ( a or b or c or d or wk1 or wk2 )
begin
   $display ("I came here, t= %0d", $stime ) ;
   wk1 <= a | b ;
   wk2 <= c & d ;
   y      <= wk1 ^ wk2 ;
end
initial begin
  a = 0 ; b = 0; c = 1 ; d = 1 ;
  #10 a = 1 ;
  #10 c = 0 ;
  #10 $finish ;
end
endmodule
```

Do not use nonblocking procedural assignment for combinational logic.(continued)

However, Style2 causes a problem called multiple passes as seen by running the code on the previous page



Style 2

```
always @ ( a or b or c or d or wk1 or wk2 )
begin
    wk1 <= a | b ;
    wk2 <= c & d ;
    y    <= wk1 ^ wk2 ;
end
```

This problem can be avoided by using blocking procedural assignment as shown below.

Style3

```
always @ ( a or b or c or d )
begin
    wk1 = a | b ;
    wk2 = c & d ;
    y    = wk1 ^ wk2 ;
end
```

Style3 does not cause any mismatch between RTL and gate net list simulation results.

Recommended style

```
always @ ( a or b or c or d )
begin
     wk1 = a | b ;
     wk2 = c & d ;
     y     = wk1 ^ wk2 ;
end
```

◎

Use blocking procedural assignment.

Write all variables in a sensitivity list which appear in RHS except those variables appear in LHS.

not recommended style

```
always @ ( a or b or c or d or wk1 or wk2 )
begin
     wk1 <= a | b ;
     wk2 <= c & d ;
     y     <= wk1 ^ wk2 ;
end
```

△

Using nonblocking procedural assignment for combinational logic is not recommended.

Wrong / buggy style

```
always @ ( a or b or c or d )
begin
     wk1 <= a | b ;
     wk2 <= c & d ;
     y     <= wk1 ^ wk2 ;
end
```

✕

Unintentional latch

```
always @ ( a )
begin
     if ( a == 1'b1 ) begin
       y = 16'h95 ;
     end
end
```

“else” missing

If the code on the left is simulated, latching will occur for missing else path as we have seen in 3.8.2.

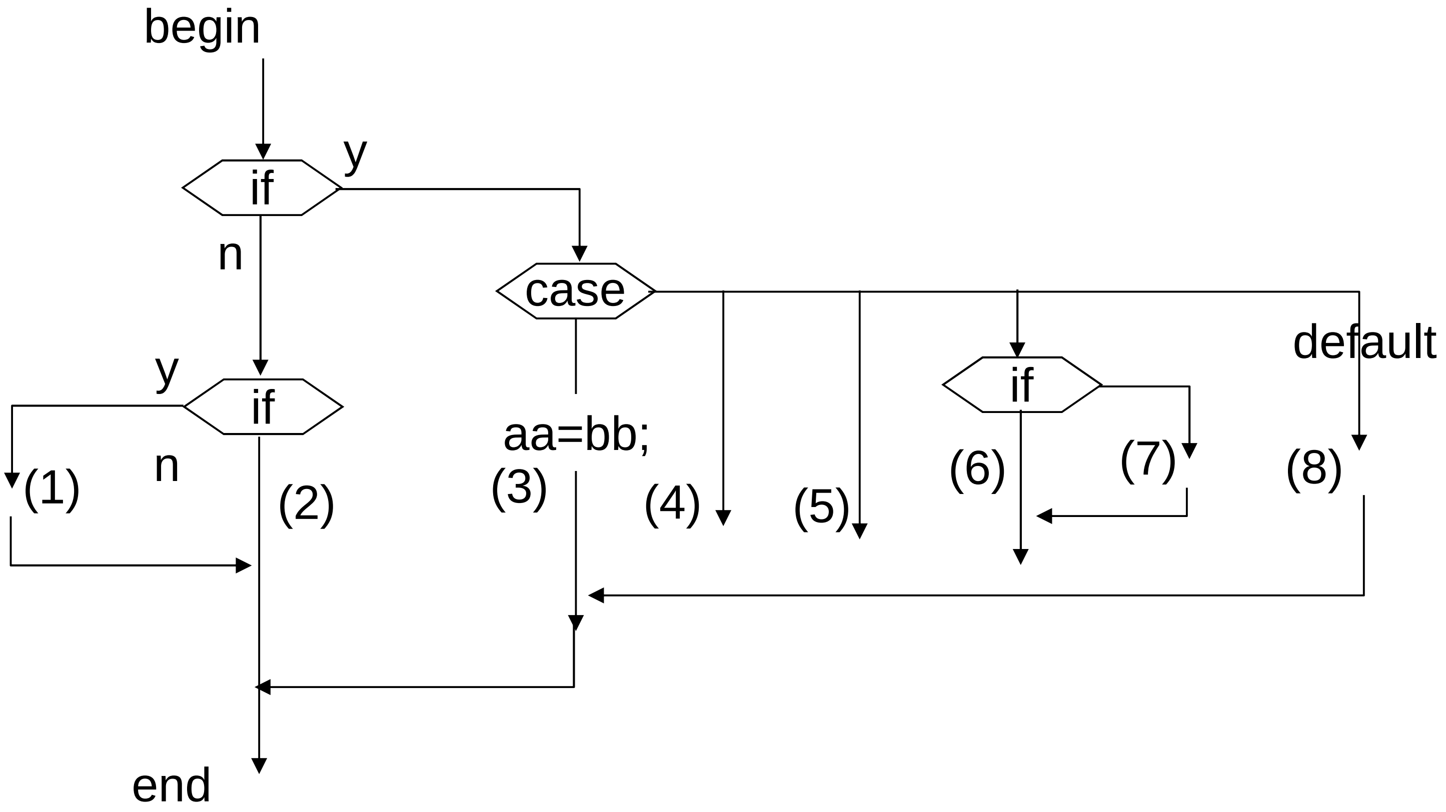
And a synthesis tool will create a memory element called latch so that the result of RTL simulation and gate net list simulation matches.

The same problem will occur if in a procedural path a variable is not given any value (missing else, missing case items, etc.). Such an always construct will create latches against the intention of a designer.

To avoid this problem, we must assign values to all the variables in any paths.

assign values to all the variables in any paths.

This means that if aa is assigned value bb in path (3), it must be given some value in path(1), (2), (4), (5), (6), (7), and (8). If no value is assigned in any one of those paths, for example, in path (7), the always construct may create a latch for aa when synthesized.



```
begin
  if ( ctl==1’b1) begin
    out_y = in_a ;
  end   ✗
end
```

```
begin
  if ( ctl==1’b1) begin
    out_y = in_a ;  ✗
  end
  else begin
    out_y = in_c ;
    out_w = in_b ;
   end
end
```

```
begin
  if ( ctl==1’b1) begin
    out_y[7:4] = in_a ;
  end
  else begin   ✗
    out_y[3:0] = in_c ;
  end
end
```

else missing; out_y is not given any value for missing else case.

out_w is not given any value for ctl=1’b1 case.

out_y[3:0] is not given any value for ctl=1’b1 case.
out_y[7:4] is not given any value for ctl is not 1’b1 case.

```
begin
  case(ctl[1:0])
   2’b00 : begin
       out_y = in_a ;
     end
   2’b01 : begin
       out_y = in_b ;
     end
   2’b10 : begin
      out_y = in_c ;
    end
  endcase
end
```

default missing;
out_y is not given
any value for
missing default
case(ctl=2’b11).

```
begin
  case(ctl[1:0])
   2’b00 : begin
       out_y = in_a ;
     end
   2’b01 : begin
       out_y = in_b ;
       out_w = in_e ;
     end
   2’b10 : begin
      out_y = in_c ;
    end
   2’b11 : begin
      out_y = in_d ;
    end
  endcase
end
```

out_w is not given
any value for
ctl=2’b00, 2’b10,
and 2’b11 cases.

```
begin
  case(ctl[1:0])
   2’b00 : begin
       out_y[7:0] = in_a ;
     end
   2’b01 : begin
       out_y[3:0] = in_b ;
     end
   2’b10 : begin
      out_y[7:0] = in_c ;
    end
   default : begin
      out_y[7:0] = 8’bx ;
  endcase
end
```

out_y[7:4] is not
given any value for
ctl is 2’b01 case.

```
always @ ( flg or in_a ) begin
  if ( flg==1’b0 ) begin
        out_y = in_a ;
   end
end
```

✗

```
always @ ( flg or in_b ) begin
  if ( ~flg==1’b0 ) begin
        out_y = in_b ;
   end
end
```

⇓

The same variable
out_y is given
values in different
always constructs.

⇔

```
always @ ( flg or in_a or in_b ) begin
  if ( flg==1’b0 ) begin
        out_y = in_a ;
   end
   else begin
        out_y = in_b ;
   end
end
```

○

```
reg [15:0] out_y ;

always @ ( flg or in_a or in_b ) begin
 if ( flg==1’b0 ) begin
     out_y[15:8] = in_a ;
  end
  else begin
     out_y[15:8] = in_b ;
  end
end
```

○

Although out_y is defined as a 16-bit variable and only bit 8 to 15 are given values, this code is OK because what are given values in flg=0 path are also given values in else path.

How to avoid unintentional latch

There are several ways to avoid the problem of unintentional latches.

Buggy!! ×

```
reg q1, q2 ;

always @ ( a or b or c or d )
begin
if ( a==b ) begin
  q1 = c ;
  q2 = d ;
end
else begin
  q1 = d ;
end
end
```

○

```
always @ ( a or b or c or d )
begin
if ( a==b ) begin
    q1 = c ;
    q2 = d ;
  end
  else begin
    q1 = d ;
    q2 = d ;
  end
end
```

Define q2 in else path same as true case.

◎

```
always @ ( a or b or c or d )
begin
  q2 = d ;
  if ( a==b ) begin
    q1 = c ;
  end
  else begin
    q1 = d ;
  end
end
```

Move q2 out of if block so that q2 is defined in any paths.

△ ×

```
always @ ( a or b or c or d )
begin
if ( a==b ) begin
    q1 = c ;
    q2 = d ;
  end
  else begin
     q1 = d ;
     q2 = 1'bx ;
  end
end
```

Give x (unknown) value to q2 to tell the synthesis tool that you do not care.

How to avoid unintentional latch (continued)

The best way to avoid a latch in general cases is to give value to variables before any conditional branch takes place.

```
always @ ( ,,,, )
begin
  sig_y = ----- ;
  case ( ---  )
    --- ; begin
             sig_y = ---- ;
          end
    --- ; begin
             sig_y = ---- ;
           end
  endcase
end
```

By giving value to sig_y at first, we can assign value to sig_y only in paths where it has to be assigned different values.

Note that using this technique to avoid unintentional latches may hide incompleteness of your code. Therefore, do not depend on this technique. Always check if there are missing paths where a variable is not given any value.

Summary of always construct for combinational logic

To create combinational logic by using always construct ;

(1) Use blocking procedural assignment,

(2) list all the signals in a sensitivity list which appear in RHS except for those signals appearing in LHS, and

(3) give values to all the signals appearing in LHS in every path of the procedure.

→ Do not use “if” without “else”.
Do not use case without case items covering all the possible cases.

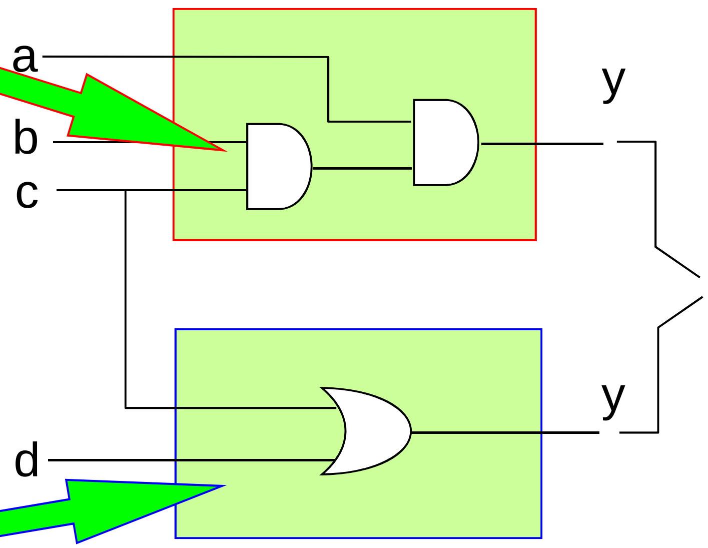
Even if (2) is not satisfied, a synthesis tool will create combinational logic correctly because it assumes all the RHS listed in a sensitivity list. But it will cause a mismatch between RTL and net list simulation results.

“Do not do always” for combinational logic

Do not use two or more always for one variable. (1)

```
always @ ( a or b or ,,, ) begin
      if (a) begin
        y = b & c ;
      end
      else begin
        y = 0 ;
      end
 end
 always @ ( a or b or ,,, ) begin

        y = c | d ;

  end
```

Synthesis tool creates one gate logic for one always.



Can't tell how to connect.

???

The same “y” on the LHS of the different always constructs.

Never do this.

“Do not do always” for combinational logic

Do not use two or more always for one variable even if they are complementary. (2)

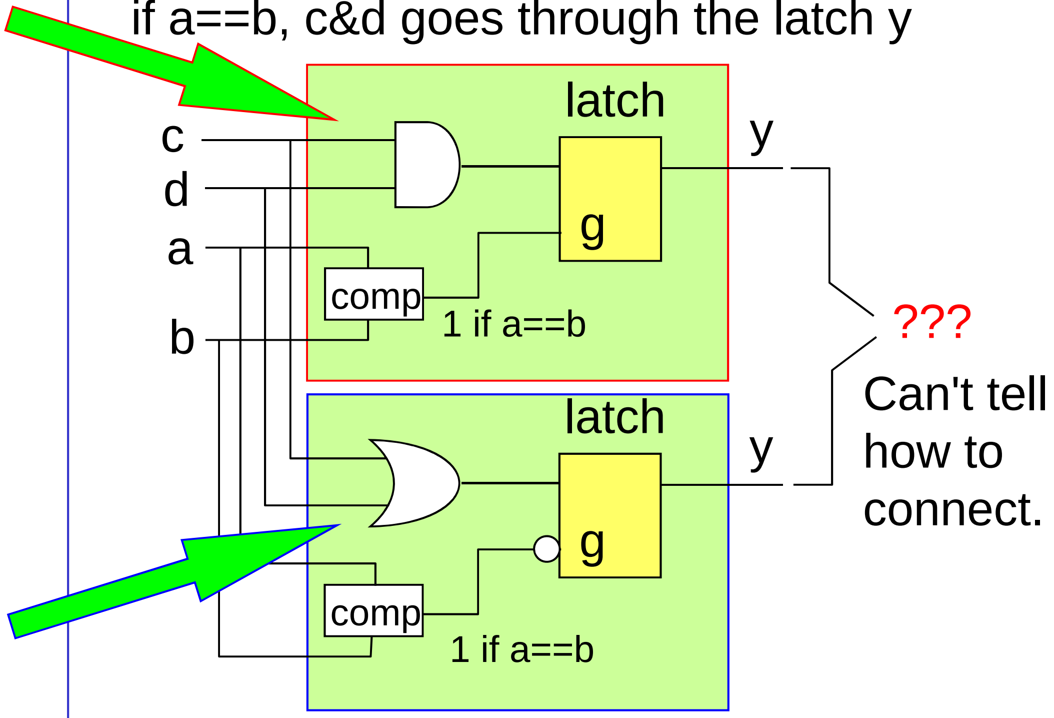
These two are complementary, when (a) is executed, (b) will not be executed, and vise versa.

```
always @ ( a or b or ,,, ) begin
      if ( a == b ) begin
        y = c & d ; // (a)
      end
      else begin
      end
 end
```

```
always @ ( a or b or ,,, ) begin
      if ( a == b ) begin
      end
      else begin
        y = c | d ; // (b)
      end
 end
```

Synthesis tool create one gate logic for one always.

if a==b, c&d goes through the latch y

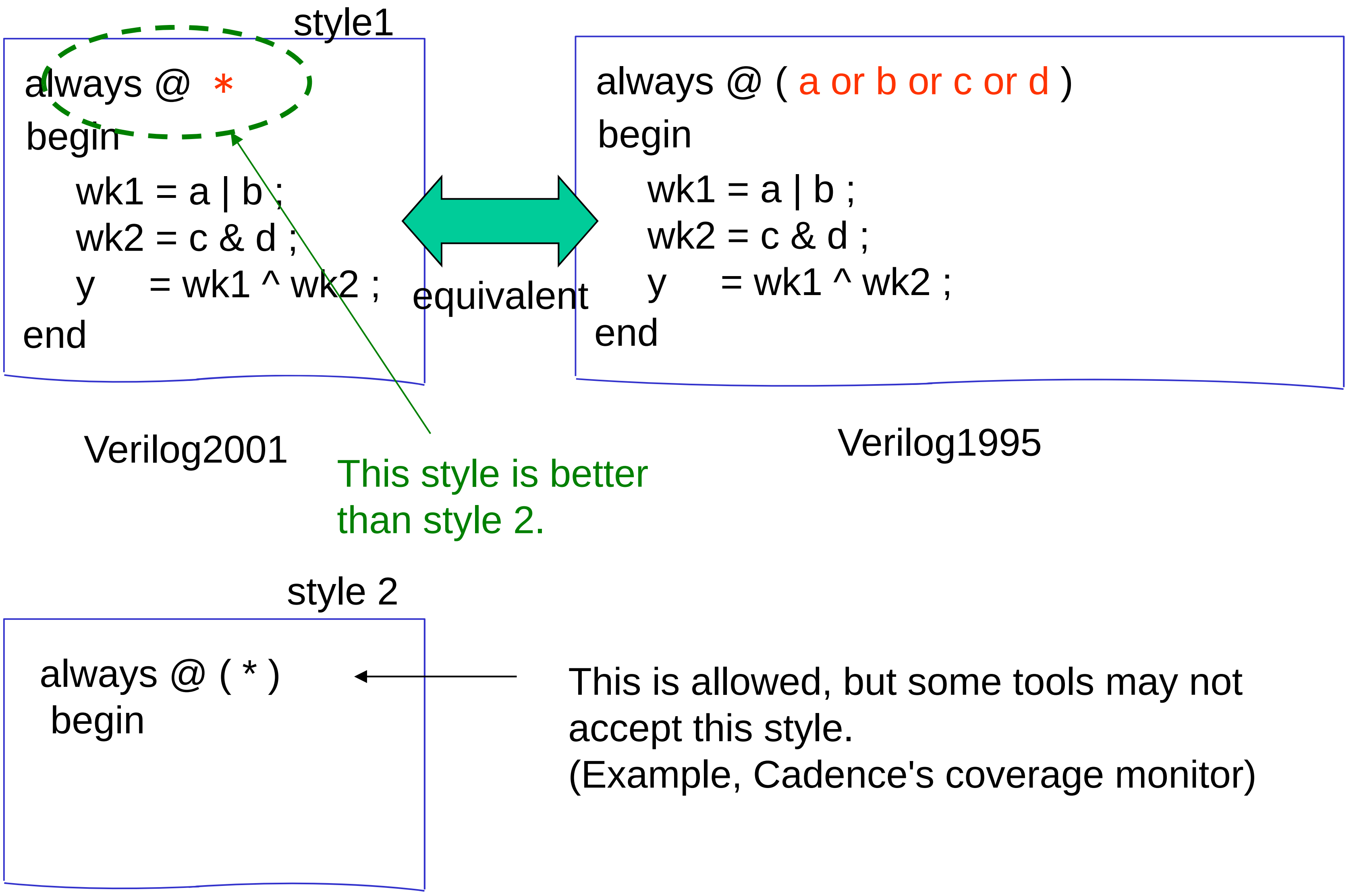


???

Can't tell how to connect.

if a!=b, c|d goes through the latch y

In Verilog2001, asterisk “ * ” can be used for implicit event expression.



style1

```
always @  *
begin
    wk1 = a | b ;
    wk2 = c & d ;
    y    = wk1 ^ wk2 ;
end
```

Verilog2001

equivalent

```
always @ ( a or b or c or d )
begin
    wk1 = a | b ;
    wk2 = c & d ;
    y    = wk1 ^ wk2 ;
end
```

Verilog1995

This style is better than style 2.

style 2

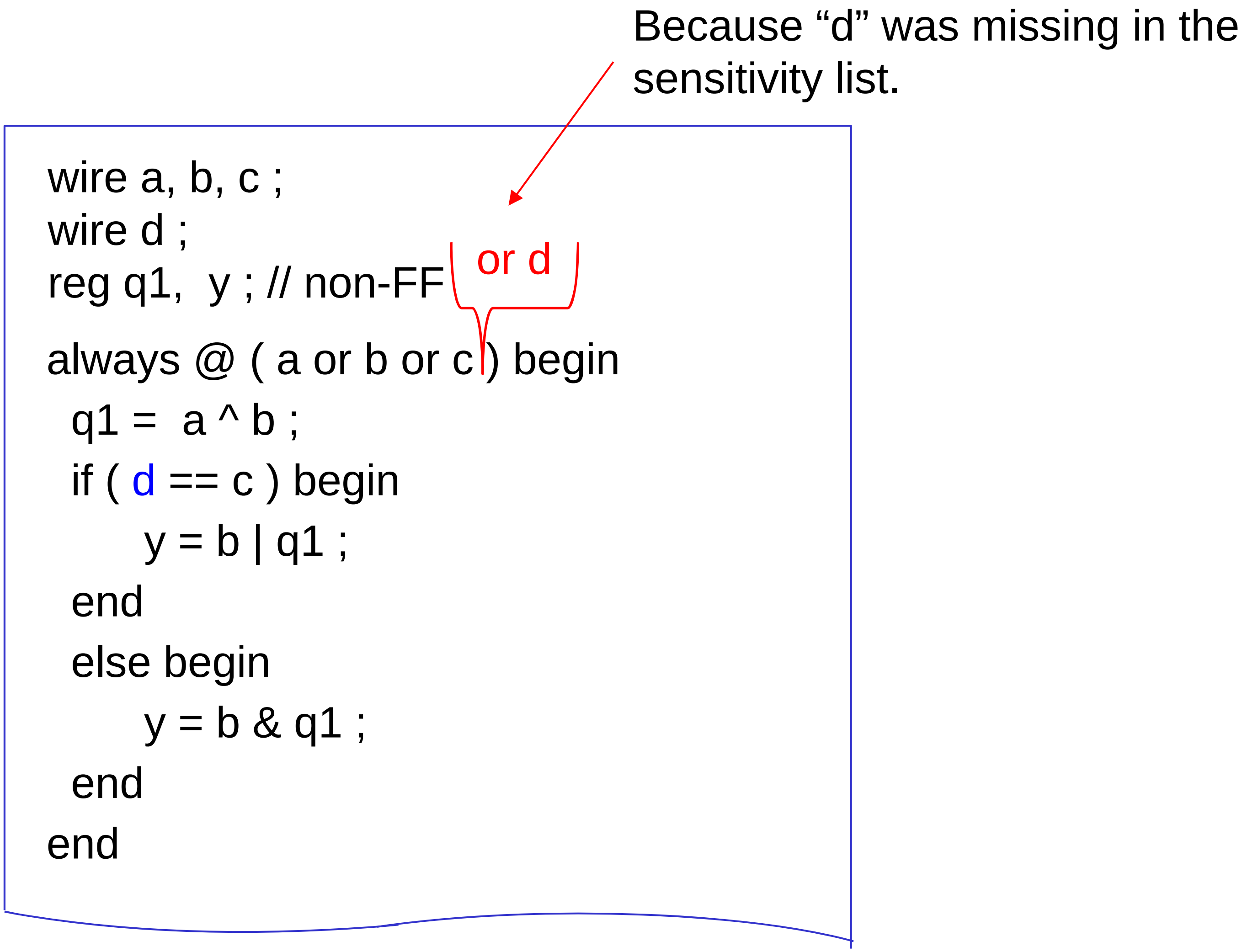
```
always @ ( * )
 begin
```

This is allowed, but some tools may not accept this style.
(Example, Cadence's coverage monitor)

Q3.10-1 : The following is a code intended to create combinational logic. Correct the code so that it will generate intended circuit.

```
wire a, b, c ;
wire d ;
reg q1,  y ; // non-FF

always @ ( a or b or c ) begin
  q1 =  a ^ b ;
  if ( d == c ) begin
        y = b | q1 ;
  end
  else begin
        y = b & q1 ;
  end
end
```

Q3.10-1 : sample answer



Because “d” was missing in the sensitivity list.

or d

```
wire a, b, c ;
wire d ;
reg q1,  y ; // non-FF
always @ ( a or b or c ) begin
  q1 =  a ^ b ;
  if ( d == c ) begin
        y = b | q1 ;
  end
  else begin
        y = b & q1 ;
  end
end
```

Q3.10-2 : The following is a part of a program intended to create combinational logic. Correct the code so that it will create combinational logic.

```
always @( posedge clk or negedge rst_n )
begin
      if ( rst_n == 1'b0 ) begin
           state <= INTL;
           a <= 12'h000;
      end
      else begin
            state <= next_state ;
            a <= next_a ;
      end
end
```

No error in this part.

```
always @ (state or flick ) begin
   case (state)
        INTL : begin
                if ( !flick ) next_state = INTL ;
                else begin
                     next_a[0] =1'b1;
                     next_state = UP_HIGH ;
                 end
               end
        UP_HIGH : begin
                 if ( !a[11] ) begin
                      next_a = a  <<  1'b1;
                      next_state = UP_HIGH ;
                  end
                  else begin
                      next_a[11] = 1'b0;
                      next_a[10] = 1'b1 ;
                      next_state = DWN_HIGH ;
                  end
                end
```

a and next_a are 12-bit signal.
state and next_state are 3-bit signal.

Q3.10-2 : Sample answer

```
always @ (state or flick ) begin
   case (state)
        INTL : begin
                if ( !flick ) next_state = INTL ;
                else begin
                     ~~next_a[0] =1'b1;~~
                     next_state = UP_HIGH ;
                 end
               end
        UP_HIGH : begin
                 if ( !a[11] ) begin
                      next_a = a  <<  1'b1;
                      next_state = UP_HIGH ;
                  end
                  else begin
                      ~~next_a[11] = 1'b0;~~
                      ~~next_a[10] = 1'b1 ;~~
                      next_state = DWN_HIGH ;
                  end
                end
                     :
```

or a

begin

next_a = a[11:0] ;
end

next_a = { a[11:1], 1'b1 };

next_a = { 1'b0, 1'b1, a[9:0] } ;

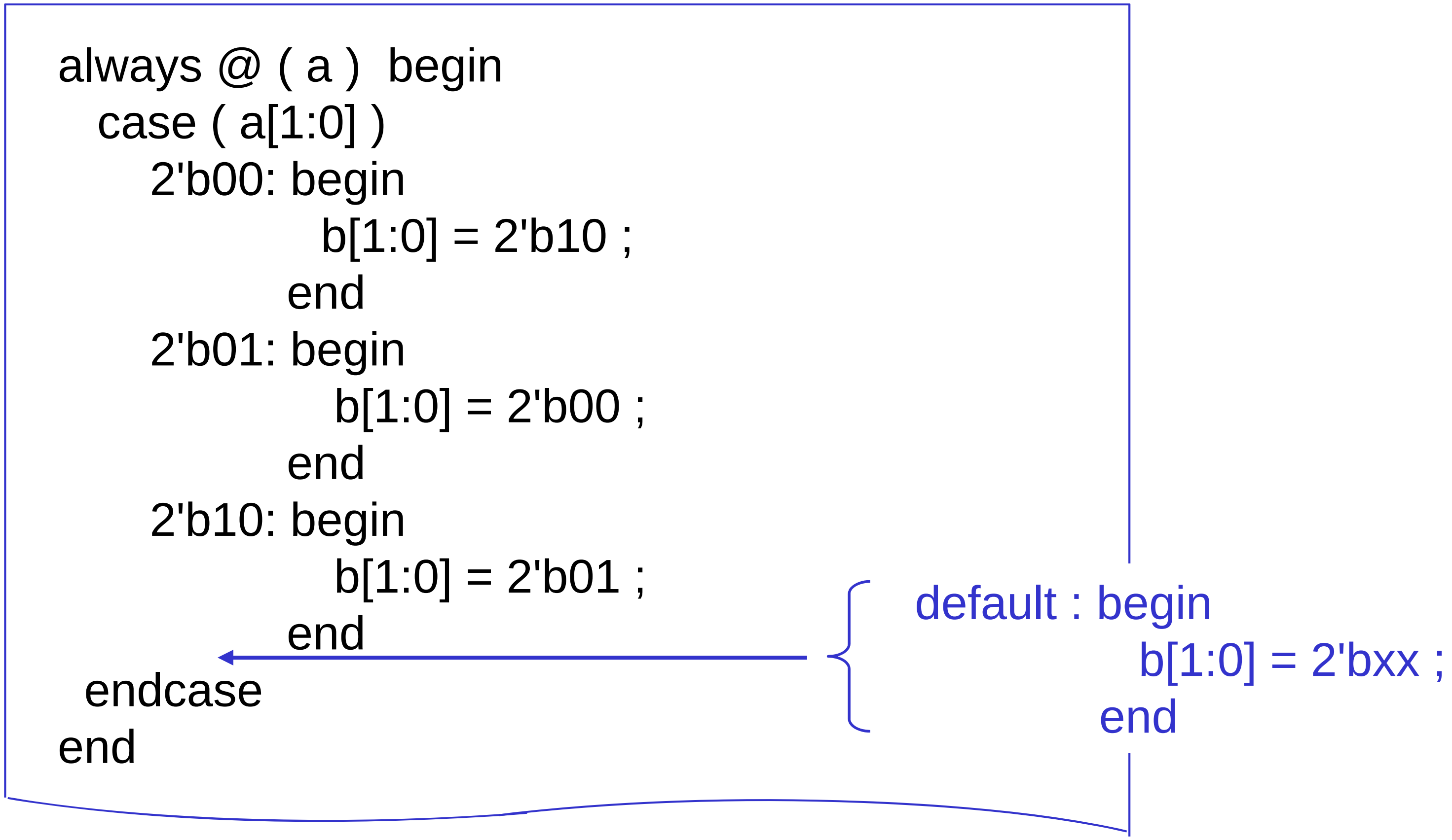
Q3.10-3 : In the following code, if the other part of the logic work correctly, it is guaranteed that case expression never becomes 2'b11. If so, the following code is OK or not?

```
always @ ( a )  begin
   case ( a[1:0] )
       2'b00: begin
                    b[1:0] = 2'b10 ;
                  end
       2'b01: begin
                     b[1:0] = 2'b00 ;
                  end
       2'b10: begin
                     b[1:0] = 2'b01 ;
                  end
  endcase
end
```

Case items cover only three cases, 00, 01, and 10, out of four possible cases. But it is guaranteed that 11 case will never happen.

Q3.10-3 : A sample answer

No, this code is not OK because the synthesis tool will create a latch to keep the value of b for such cases that case expression becomes 2'b11. We have to use default to prevent creating a latch.
The synthesis tool can not know that a[1:0] never becomes 2'b11.



```
always @ ( a )  begin
   case ( a[1:0] )
       2'b00: begin
                    b[1:0] = 2'b10 ;
                 end
       2'b01: begin
                     b[1:0] = 2'b00 ;
                 end
       2'b10: begin
                     b[1:0] = 2'b01 ;
                 end
  endcase
end
```

```
default : begin
                 b[1:0] = 2'bxx ;
              end
```

Q3.10-3 : A sample answer (continued)

We can add case item “2'b11” to prevent creating latch. Because 11 case never happens, we may assign arbitrary value to b for 11 case.

```
always @ ( a )  begin
   case ( a[1:0] )
       2'b00: begin
                    b[1:0] = 2'b10 ;
                end
       2'b01: begin
                     b[1:0] = 2'b00 ;
                end
       2'b10: begin
                     b[1:0] = 2'b01 ;
                end
```

We can use 2'bxx instead of 2'b01 to detect 11 case happened.

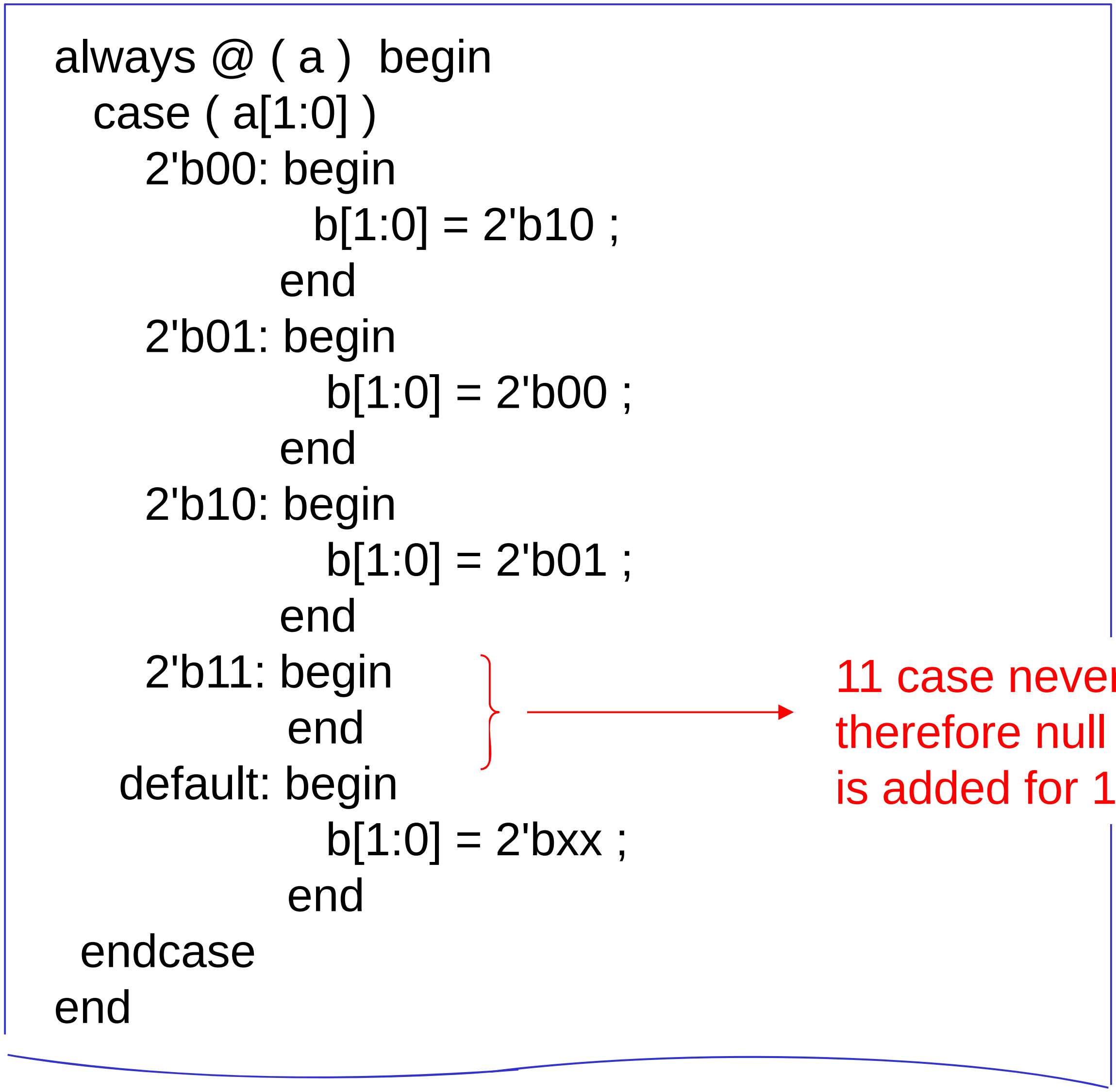
```
2'b11: begin
               b[1:0] = 2'b01 ;
           end
```

```
  endcase
end
```

```
default : begin
               b[1:0] = 2'bxx ;
          end
```

By adding 11 case, latch will not be created. But to checkout error case such as case expression becomes 2'b1x, write default part.

Q3.10-4 : Will the following code create a latch?



```
always @ ( a )  begin
   case ( a[1:0] )
       2'b00: begin
                    b[1:0] = 2'b10 ;
                 end
       2'b01: begin
                     b[1:0] = 2'b00 ;
                 end
       2'b10: begin
                     b[1:0] = 2'b01 ;
                 end
       2'b11: begin
                  end
     default: begin
                     b[1:0] = 2'bxx ;
                  end
  endcase
end
```

11 case never happens
therefore null statement
is added for 11 case.

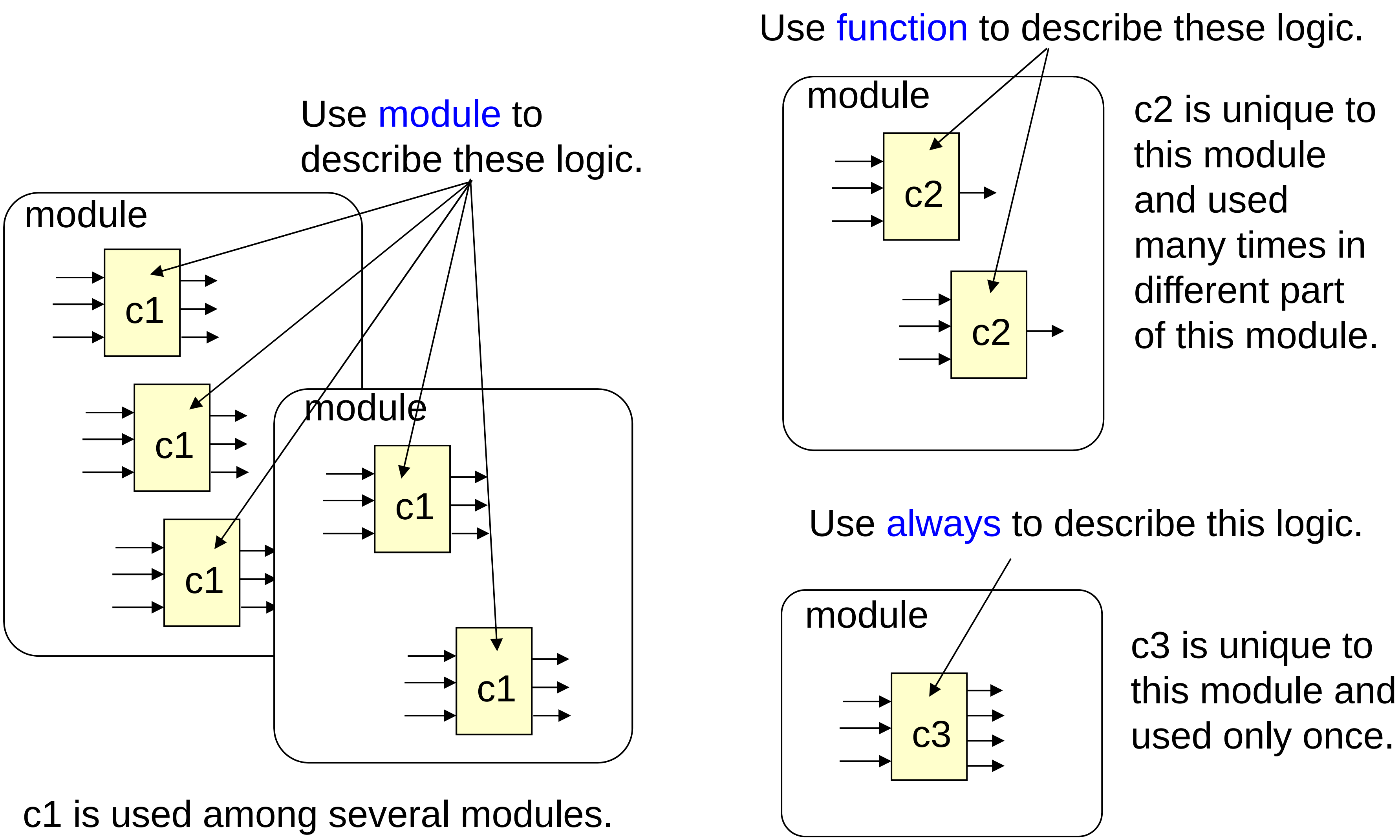
Q3.10-4 : A sample answer.

```
always @ ( a )  begin
   case ( a[1:0] )
       2'b00: begin
                   b[1:0] = 2'b10 ;
                 end
       2'b01: begin
                    b[1:0] = 2'b00 ;
                 end
       2'b10: begin
                    b[1:0] = 2'b01 ;
                 end
       2'b11: begin
                  end
     default: begin
                    b[1:0] = 2'bxx ;
                  end
  endcase
end
```

Yes, this code will create a latch because no value is given to b in 11 case .

This part will create a lath.

How to use module, function, and always for combinational logic



Use module to describe these logic.

c1 is used among several modules.

Use function to describe these logic.

c2 is unique to this module and used many times in different part of this module.

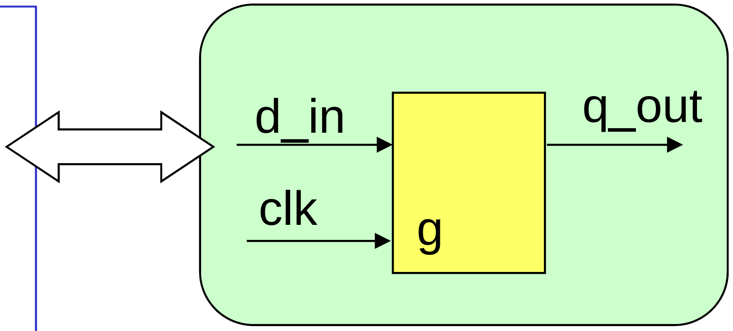
Use always to describe this logic.

c3 is unique to this module and used only once.

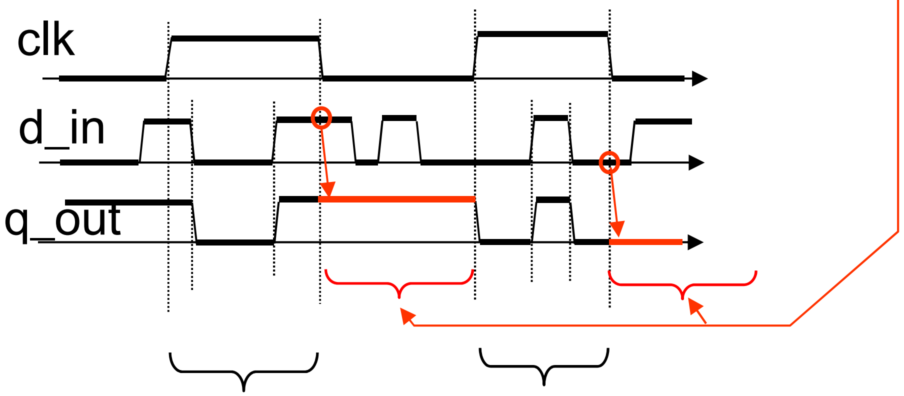
3.10.2 Always for latch

We can define a latch by using always construct shown below.

```
always @ ( clk or d_in )
 begin
     if ( clk == 1'b1 ) begin
          q_out <= d_in ;
     end
end
```



While clk is 0, latched value appears.



While clk is 1, d_in appears to q_out.
(the latch is transparent)

RTL simulation ;
While clk is 1, input d_in is placed on q_out.
When clk is 0, q_out is not assigned any value, therefore previous value, memorized value, is placed on q_out.

g	d	q
0	x	q_0
1	0	0
1	1	1

truth table for D-type transparent latch

Important!! : Right after the power on of latch, its value is unknown if clk = 0.

RTL code for D-type transparent latch

```
module d_latch( clk, d_in, q_out ) ;
input clk, d_in ;
output q_out ;
wire clk, d_in ;
reg     q_out ; // latch
//
always @ ( clk or d_in )
begin
    if ( clk  ==   1'b1 )  begin
          q_out <= d_din ;
    end
end
endmodule
```

If we drop d_in from the sensitivity list, a synthesis tool will still create a transparent latch, however it will not behave transparently in simulation.

Avoid using latch in synchronous design if possible.
Do not use a latch gated by non-clock signal.

Do not try to write RTL code to create a latch in the following manner.

```
always @ ( g or d_in )
begin
    case ( g )
      2'b01 : q_out <= d_in ;
      2'b10 : q_out <= ~d_in ;
      2'b11 : q_out <= 1'b1 ;
    endcase
end
```

~~This code expects that a latch will be created if there is a missing path.~~

Write gating logic explicitly.

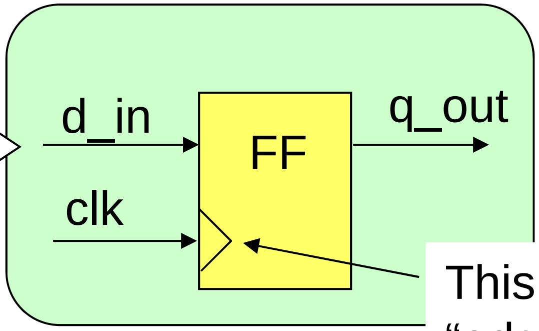
~~And it uses a signal other than a clock as a gate signal.~~

Use clock for gate signal.

3.10.3 Always for flip-flop

We can define a flip-flop by using always construct shown below.

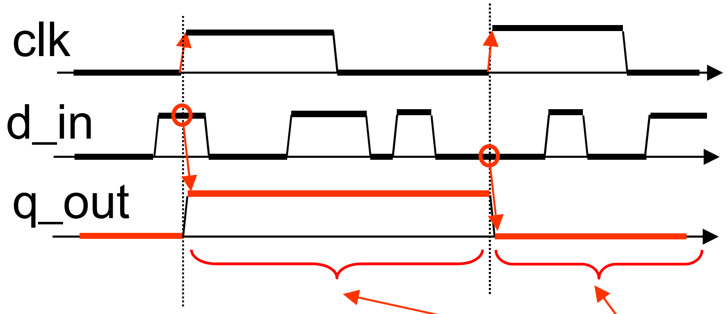
```
always @ ( posedge clk )
  begin

     q_out <= d_in

  end
```



This triangle means “edge trigger”.

q_out is memorized when the clock rises.

RTL simulation ;
Only when clk signal changes to 1, d_in is placed to q_out.
Otherwise, memorized value is placed to q_out.



When clock does not rise, memorized value appears on q_out.

How to write RTL code for FF correctly

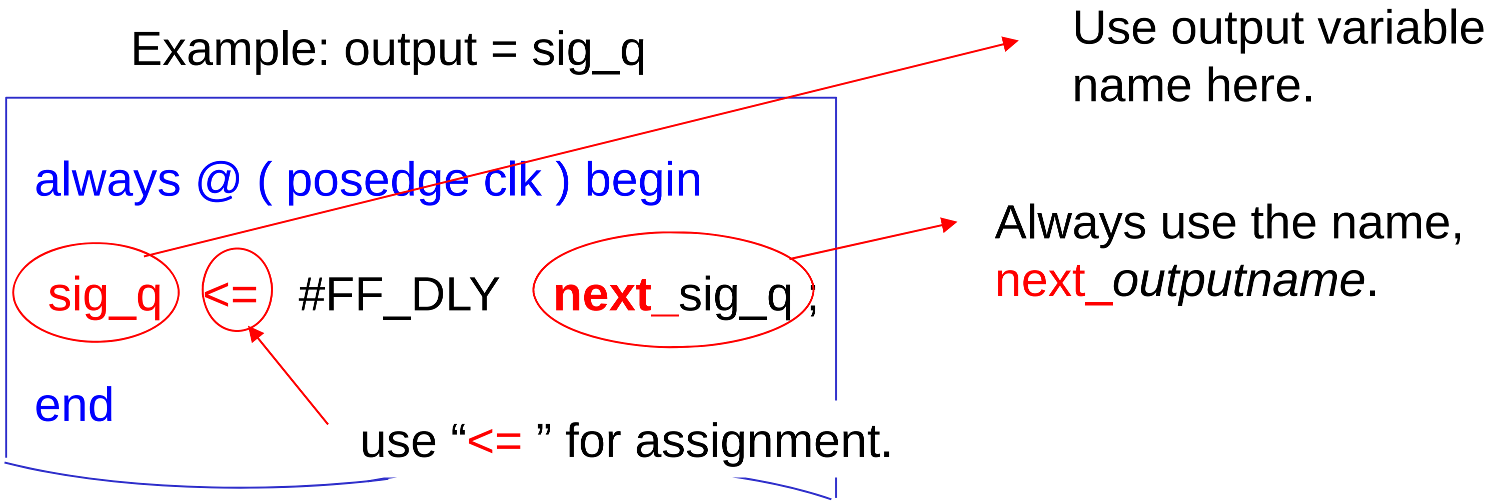
Follow the steps described here to write RTL code for a flip-flop.

step 1 : prepare template

```
always @ ( posedge clk ) begin


end
```

step 2 : Define name of the output variable

Example: output = sig_q



```
always @ ( posedge clk ) begin
sig_q <= #FF_DLY next_sig_q ;
end
```

Use output variable name here.

Always use the name, next_*outputname*.

use "<= " for assignment.

This is the simplest basic code for FF.

```
always @ ( posedge clk ) begin
     sig_q  <=  #FF_DLY   next_sig_q ;
end
```

Delay is introduced to avoid racing problem.

Give minimum value to FF_DLY such as 1.

step 3 : Think if reset is needed or not.

If reset is needed add reset.

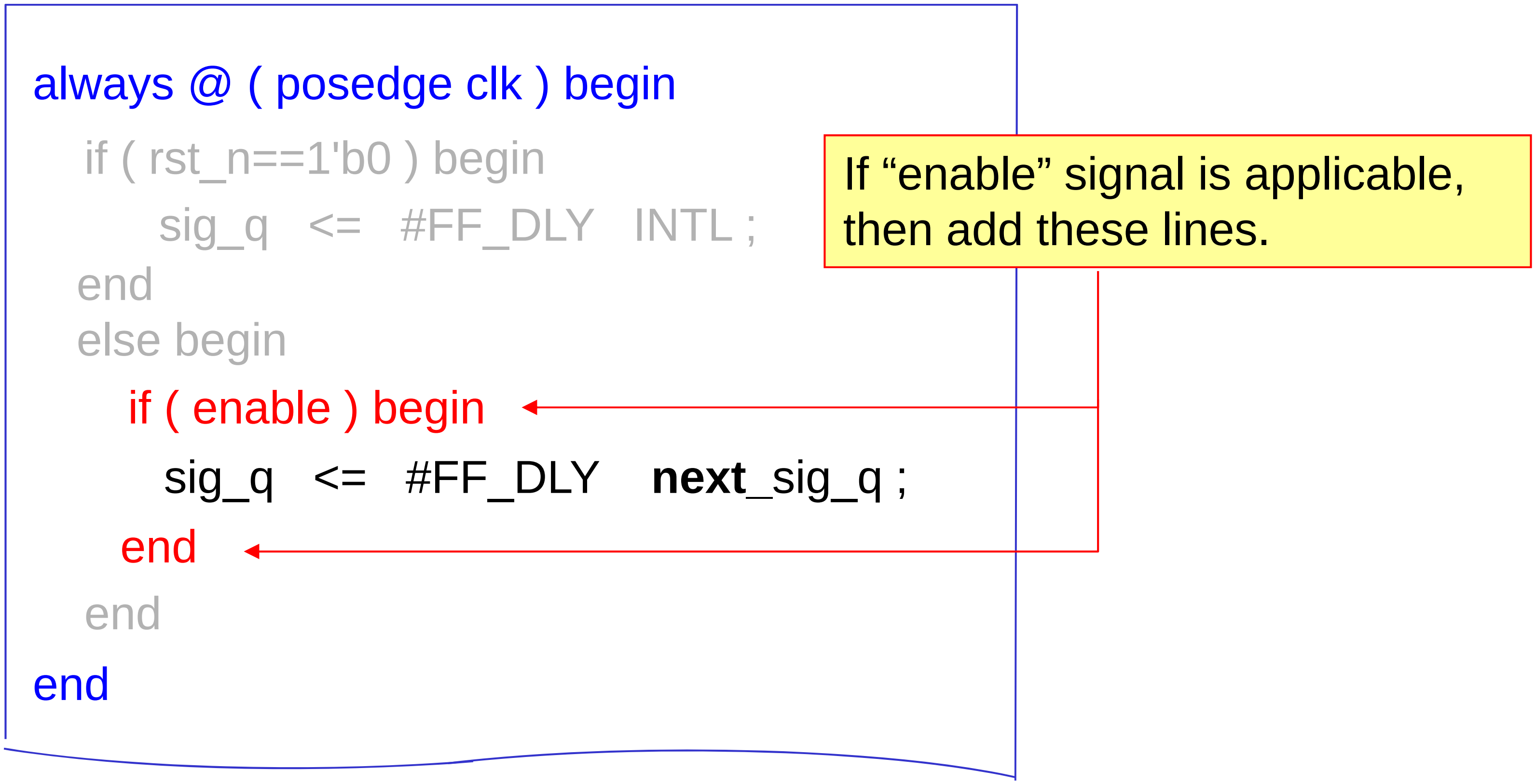
```
always @ ( posedge clk ) begin
   if ( rst_n==1'b0 ) begin
        sig_q  <=  #FF_DLY  INTL ;
   end
   else begin
        sig_q  <=  #FF_DLY   next_sig_q ;
   end
end
```

If the reset must be asynchronous, then add “negedge rst_n” here.

If reset needed add this part.

step 4 : Think if enable signal applicable or not.

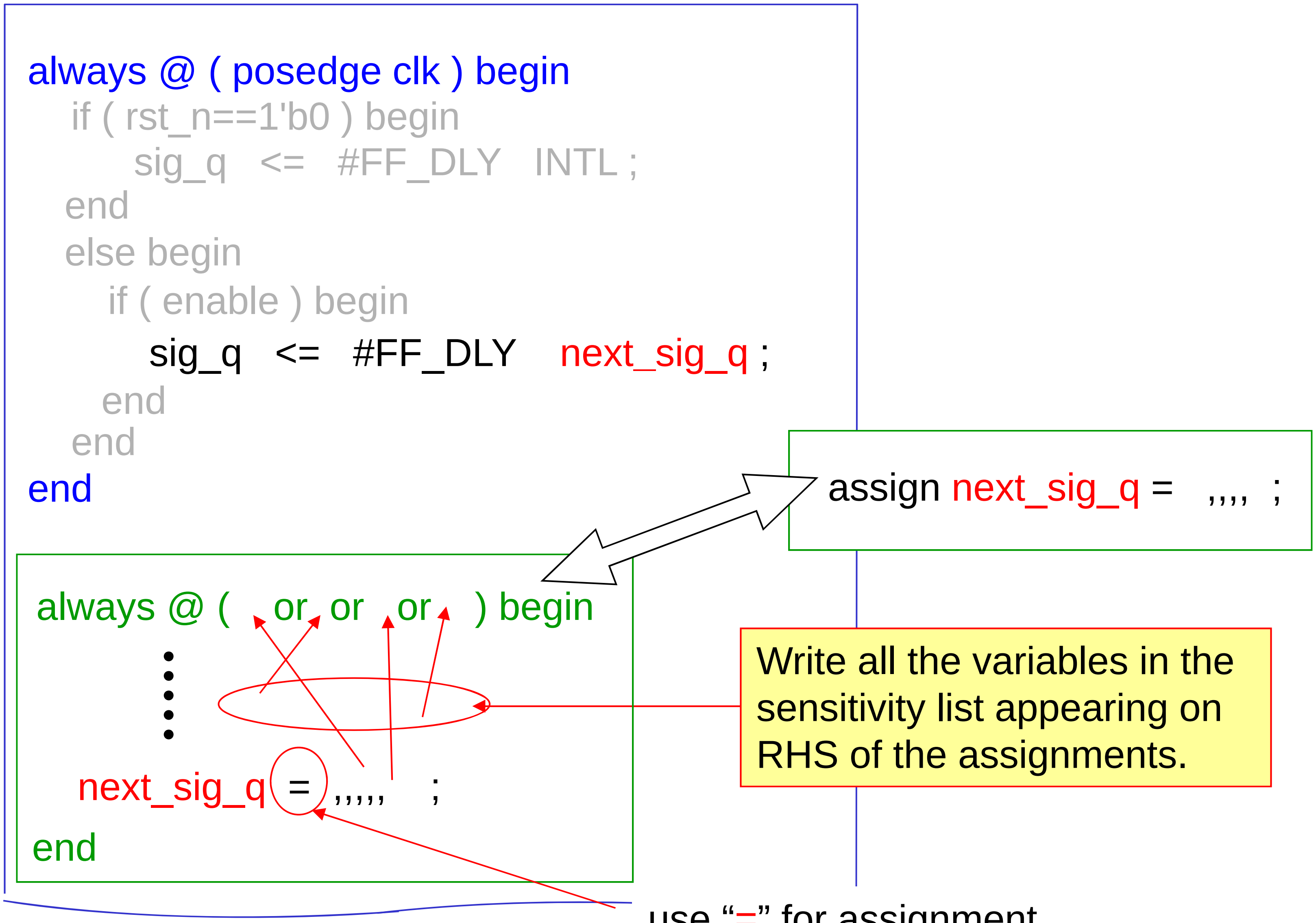
If “enable” applicable, then add enable.



```
always @ ( posedge clk ) begin
   if ( rst_n==1'b0 ) begin
       sig_q   <=   #FF_DLY   INTL ;
   end
   else begin
      if ( enable ) begin
        sig_q   <=   #FF_DLY    next_sig_q ;
      end
   end
end
```

If “enable” signal is applicable, then add these lines.

step 5 : Prepare logic for creating next_sig_q.



```
always @ ( posedge clk ) begin
   if ( rst_n==1'b0 ) begin
        sig_q   <=   #FF_DLY   INTL ;
   end
   else begin
      if ( enable ) begin
        sig_q   <=   #FF_DLY    next_sig_q ;
      end
   end
end
```

```
always @ (    or   or    or    ) begin
      ⋮
   next_sig_q = ,,,,,    ;
end
```

```
assign next_sig_q =   ,,,,  ;
```

Write all the variables in the sensitivity list appearing on RHS of the assignments.

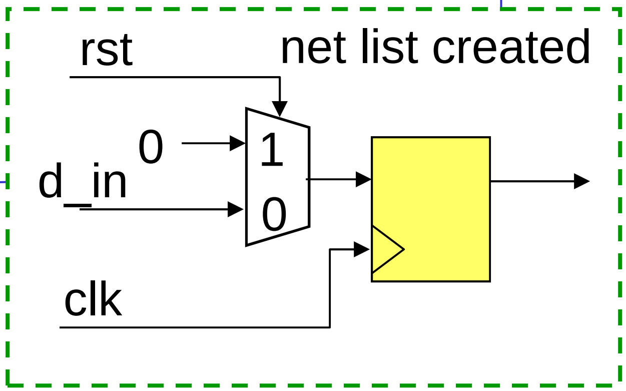
use “=” for assignment.

Synchronous reset vs. asynchronous reset

Synchronous reset is effective only when a clock signal is properly provided.
If the clock signal is not working, synchronous reset does not work.
Asynchronous reset is effective without clock.

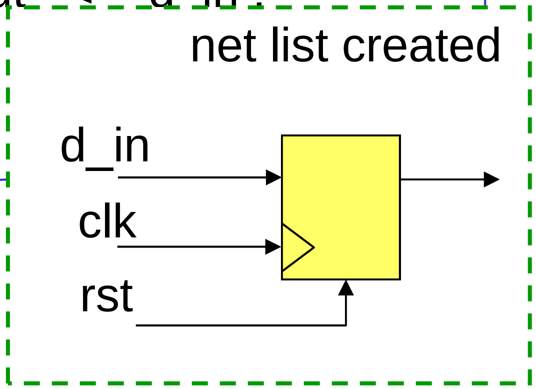
RTL code for synchronous reset

```
always @ ( posedge clk ) begin
    if ( rst ) begin
          q_out    <=   0 ;
   end
   else begin
          q_out   <=  d_in ;
    end
end
```



RTL code for asynchronous reset

```
always @ ( posedge clk or
                    poesdge rst ) begin
    if ( rst ) begin
          q_out    <=   0 ;
   end
   else begin
          q_out   <=  d_in ;
    end
end
```



Problem of synchronous reset

RTL coding for synchronous reset may create a net list in which reset does not work properly.

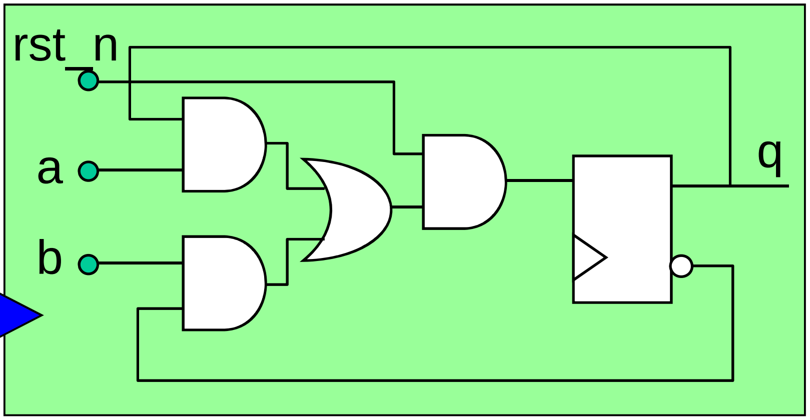
This is not a critical issue so far as you know the issue.

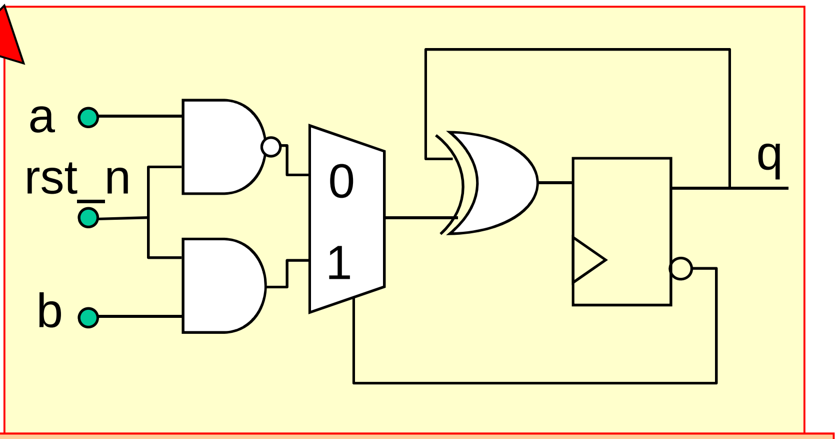
sample code

```
always @ ( posedge clk ) begin
  if ( rst_n==1'b0 ) q <=  #FF_DLY   1'b0 ;
  else begin
     if ( q ) q <= #FF_DLY   a ;
     else    q <= #FF_DLY   b ;
   end
end
```

Synthesis

net list





$$Q= R\ (Q A + \overline{Q} B)$$
$$= Q (RA) (\overline{(RB)} + 1) + \overline{Q} (RB)$$
$$= Q (RA) (\overline{RB} + Q) + \overline{Q} (RB)$$
$$= Q (RA + \overline{Q}) (\overline{RB} + Q) + \overline{Q} (RB)$$
$$= Q\ \overline{\overline{RA}\ Q}\ \overline{(RB) \overline{Q}} + \overline{Q} (RB)$$
$$= Q \overline{(\overline{RA}\ Q + RB \overline{Q})} + \overline{Q} (\overline{RA}\ Q + RB \overline{Q})$$
$$= Q \oplus (\overline{RA}\ Q + RB \overline{Q})$$

If initial value of q is x, q keeps value x in simulation even if reset asserted.

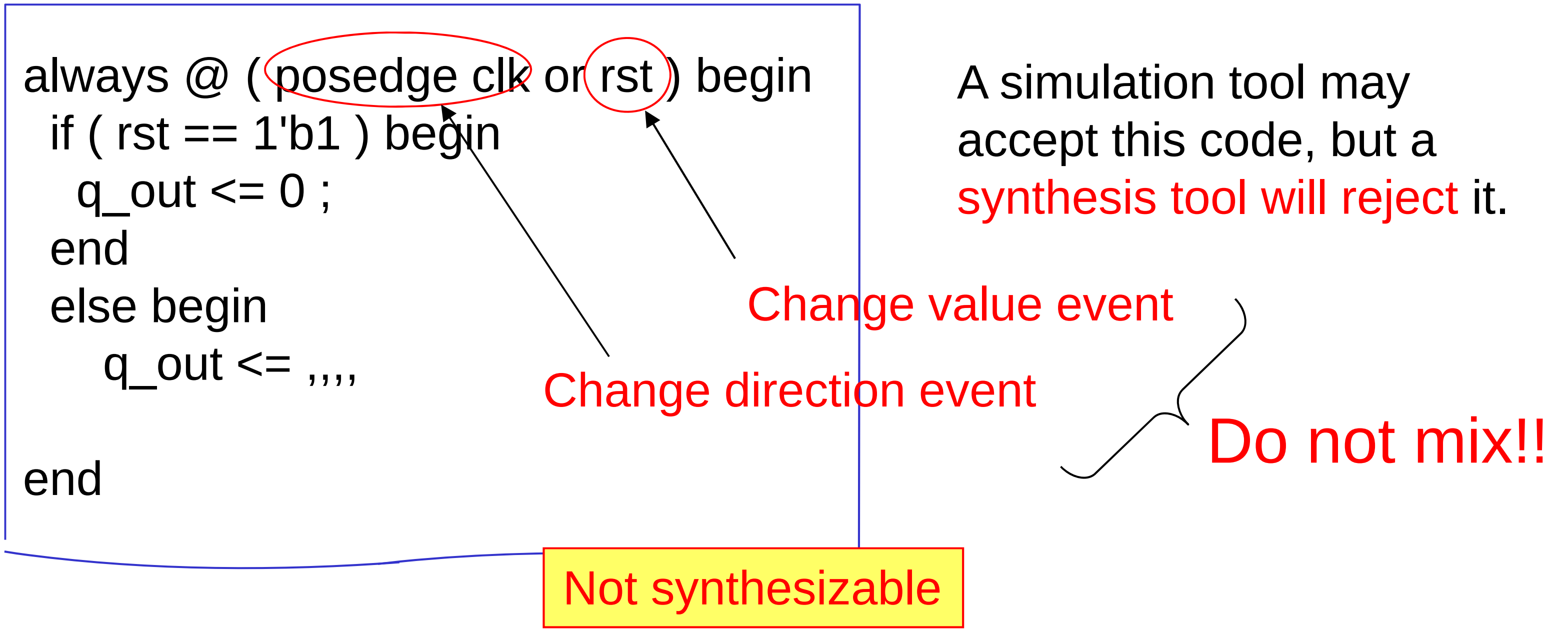
In general use asynchronous reset for system reset, or power on reset. For other cases use synchronous reset. Because synchronous reset signal can be handled just as same as data signal.

Although asynchronous reset does not have to be synchronized to a clock, if it is negated at clock rise time, it will cause metastability of the FF.
To avoid the metastability, frontend designers must take care not to negate an asynchronous reset at clock rise time. We may stop the clock while negating the reset or insert some gate to suppress reset signal negation.

Asynchronous reset lines are created in CTS (clock tree synthesis), that is, they are handled differently from the data lines. Therefore we must not refer to asynchronous reset in (synchronous) combinational logic.

“Must not” in writing RTL code for flip-flop (1)

Do not write change value event in a sensitivity list.



```
always @ ( posedge clk or rst ) begin
  if ( rst == 1'b1 ) begin
    q_out <= 0 ;
  end
  else begin
     q_out <= ,,,,

end
```

Change value event

Change direction event

Do not mix!!

Not synthesizable

A simulation tool may accept this code, but a synthesis tool will reject it.

“Must not” in writing RTL code for flip-flop (2)

Do not write reset signal as the first argument.

```
always @ ( posedge rst or posedge clk)
begin
  if ( rst == 1'b1 ) begin
    q_out <= 0 ;
  end
  else begin
      q_out <= ,,,,

end
```

A simulation tool may accept this code, but a synthesis tool treats rst as a clock signal.

Do not place reset signal in the first argument.

rst is treated as a clock signal

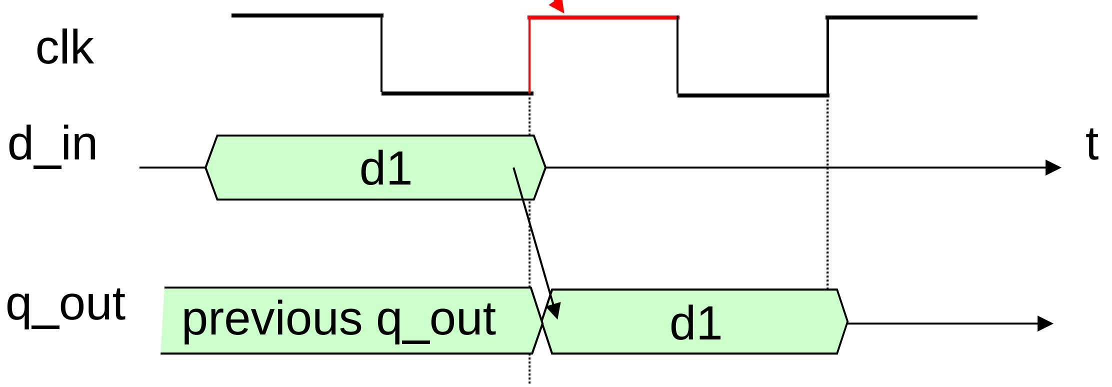
“Must not” in writing RTL code for flip-flop (3)

Do not write your code to check if the clock has really risen or not.
The following code may cause some problems to a synthesis tool.

```
always @ ( posedge clk ) begin
 ~~if ( clk == 1'b1 ) begin~~
   q_out <= d_in ;
 end
```

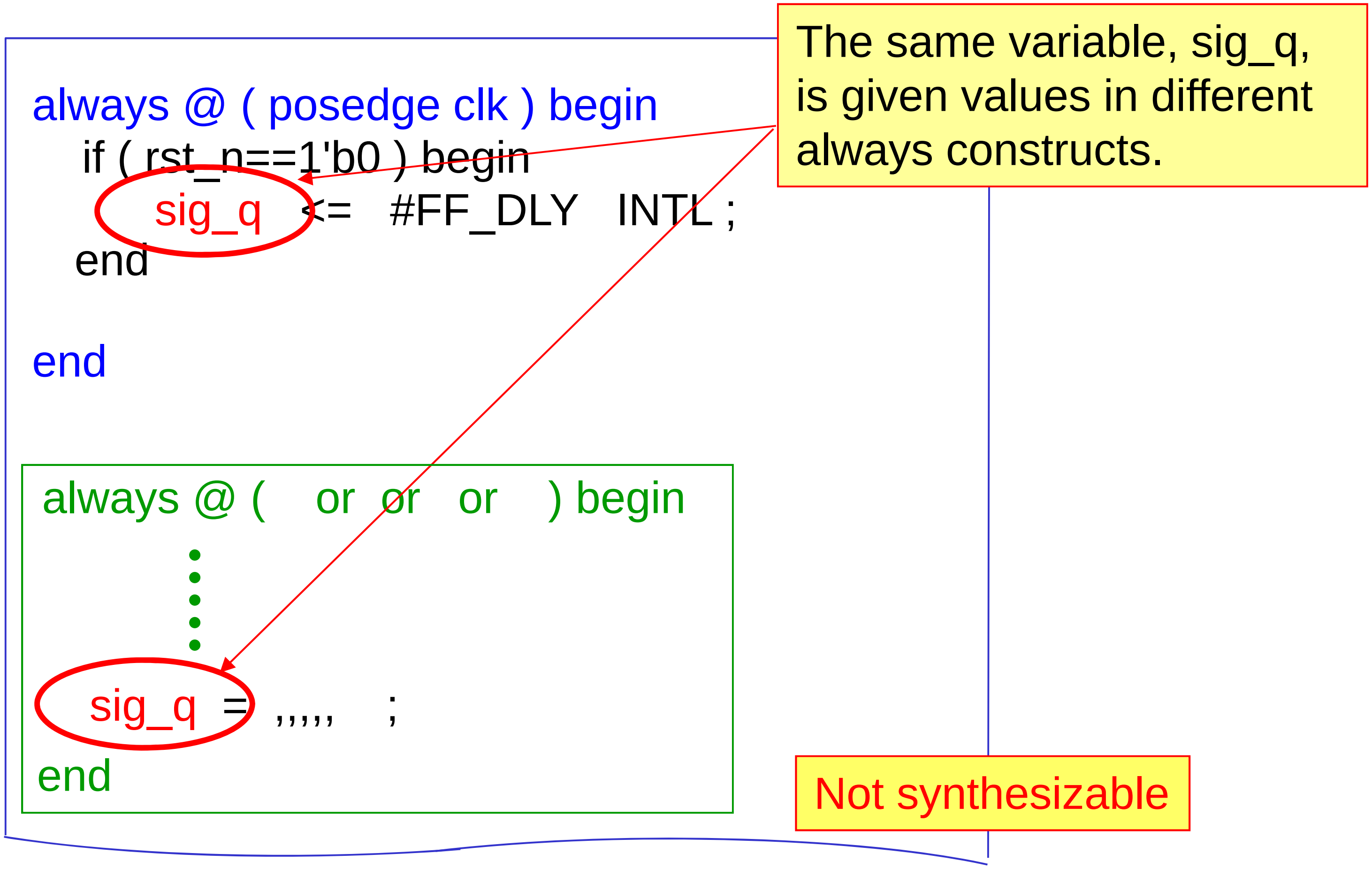
Do not check clock signal.

Not synthesizable



“Must not” in writing RTL code for flip-flop (4)

Never give value to the same variable from multiple always constructs.



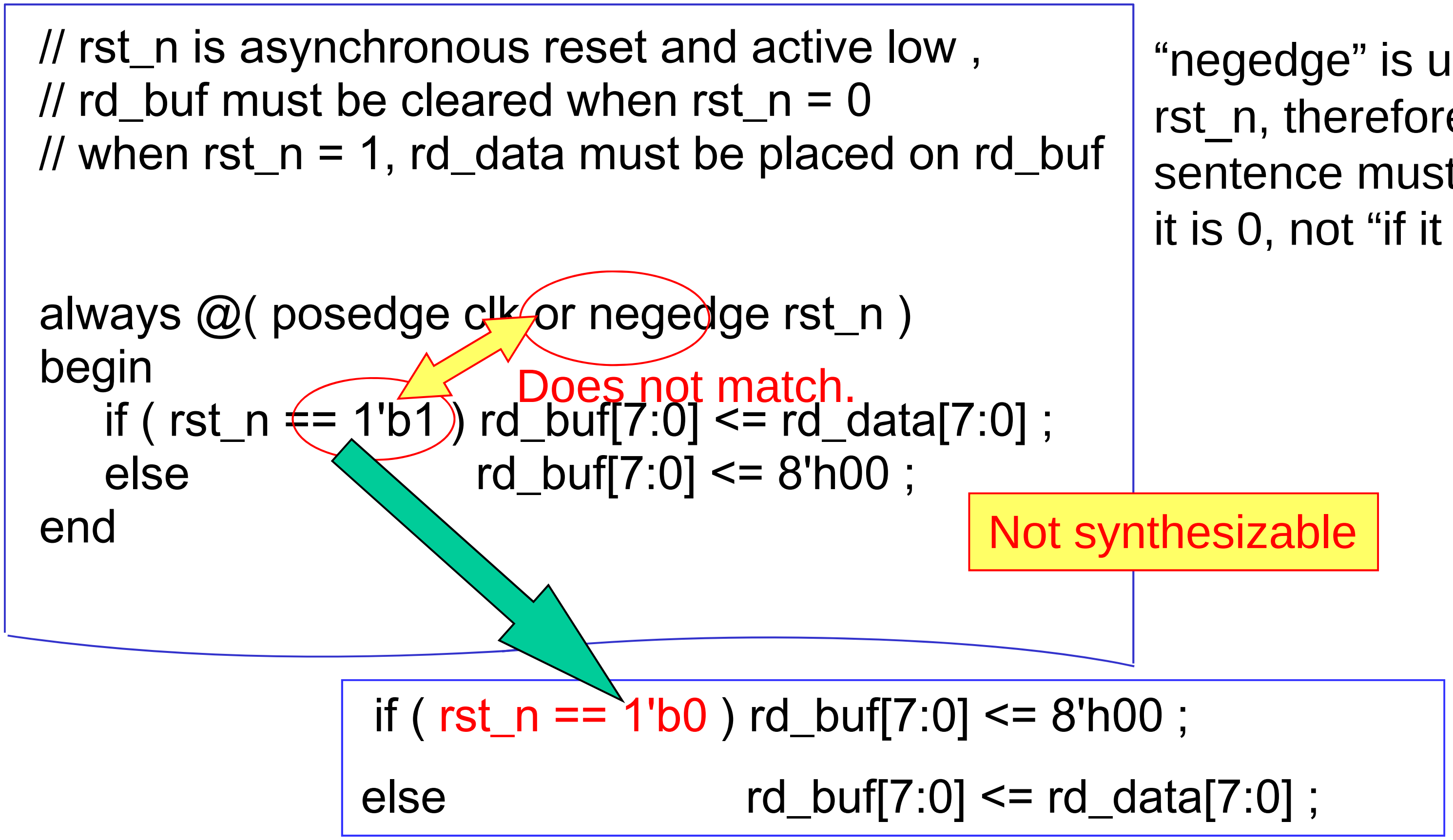
```
always @ ( posedge clk ) begin
   if ( rst_n==1'b0 ) begin
      sig_q  <=  #FF_DLY  INTL ;
   end

end

always @ (    or  or   or    ) begin
         ⋮
 sig_q  =  ,,,,,    ;
end
```

The same variable, sig_q, is given values in different always constructs.

Not synthesizable

“Must not” in writing RTL code for flip-flop (5)

Do not neglect the polarity of edge.



```
// rst_n is asynchronous reset and active low ,
// rd_buf must be cleared when rst_n = 0
// when rst_n = 1, rd_data must be placed on rd_buf


always @( posedge clk or negedge rst_n )
begin
     if ( rst_n == 1'b1 ) rd_buf[7:0] <= rd_data[7:0] ;
     else                 rd_buf[7:0] <= 8'h00 ;
end
```

Does not match.

Not synthesizable

```
if ( rst_n == 1'b0 ) rd_buf[7:0] <= 8'h00 ;

else                 rd_buf[7:0] <= rd_data[7:0] ;
```

“negedge” is used for rst_n, therefore if sentence must check if it is 0, not “if it is 1”.

“Must not” in writing RTL code for flip-flop (6)

Do not write unnecessary signal in a sensitivity list.

```
// when rd_req = 1,
// rd_data must be placed on rd_buf

always @( posedge clk or negedge rst_n )
begin
    if ( rd_req == 1'b1 )  begin
            rd_buf[7:0] <= rd_data[7:0] ;
    end
end
```

Does not match.

Not synthesizable

rst_n is not used in the sequential block, therefore, remove it from the sensitivity list.

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

“Must not” in writing RTL code for flip-flop (7-1)

Do not use complex if statement for asynchronous reset signal.

```
// rst is asynchronous reset and active high ,
// rd_buf must be cleared when rst = 1
// when rst = 0, rd_data must be placed on rd_
// clr is active high buffer clear signal

always @( posedge clk or posedge rst )
begin
     if (  ( rst == 1'b1 ) ~~| ( clr == 1'b1 )~~ )  begin
               rd_buf[7:0] <= 8'h00 ;
     end
     else    rd_buf[7:0] <= rd_data[7:0] ;
end
```

Not synthesizable

Use only simple if.

```
if (  rst == 1'b1 )  begin
            rd_buf[7:0] <= 8'h00 ;
end
else begin
   if  ( clr == 1'b1 ) begin
            rd_buf[7:0] <= 8'h00 ;
   end
   else begin
            rd_buf[7:0] <= rd_data[7:0]
;
   end
end
```

“Must not” in writing RTL code for flip-flop (7-2)

Do not use complex if statement.

```
always @ ( posedge clk
               or negedge rst_n or negedge st_n)
begin
  case ~~( { rst_n, st_n }~~ )
    2'b00, 2'b01 : q <= 1'b0 ;
    2'b10              : q <= 1'b1 ;
    2'b11 : begin
                case ( { j , k } )
                   2'b00:     q <= q ;
                   2'b01:     q <= 1'b0 ;
                   2'b10:     q <= 1'b1 ;
                   2'b11:     q <= ~q ;
                  default:  q <= 1'bx ;
                endcase
            end
  endcase
end
```

Only simple “if” statement is allowed for reset and set signal.

```
if ( rst_n  ==   1'b0 ) begin
      q <= #FF_DLY         1'b0 ;
end
else begin
   if ( st_n == 1'b0 ) begin
      q <= #FF_DLY         1'b1 ;
   end
   else begin
     case ( { j, k } )
```

Not synthesizable

“Must not” in writing RTL code for flip-flop (8)

Do not refer to an asynchronous signal in the combinational logic.

```
always @ ( posedge clk or negedge rst_n )
  if ( rst_n==1'b0 ) begin
    out_q <= INTL ;
  end
  else begin
    out_q <= next_out_q ;
  end
end

always @ * begin
  ~~if ( rst_n ) begin~~
    next_out_q = INTL ;
  end
  else begin
     case ( state )
       INTL : ,,,,,
        ,,,,
     endcase
   end
end
```

rst_n is an asynchronous signal.

Asynchronous signal must not be referenced in the combinational logic.

Cause problems in backend design

Synchronous signal and asynchronous signal must be handled differently in backend design.

We do not have to take care about reset in combinational logic for next_out_q.

```
always @ ( posedge clk or negedge rst_n )
  if ( rst_n==1'b0 ) begin
    out_q <= INTL ;
  end
  else begin
    out_q <= next_out_q ;
  end
end

always @ * begin
 case ( state )
       INTL : ,,,,,
        ,,,,
       ,,,,
        ,,,,
        ,,,,
     endcase
   end
end
```

We do not have to do any reset operation in this always construct for next_out_q because whatever the value of next_out_q might be, out_q will be given the initial value by the always construct for the out_q FF.

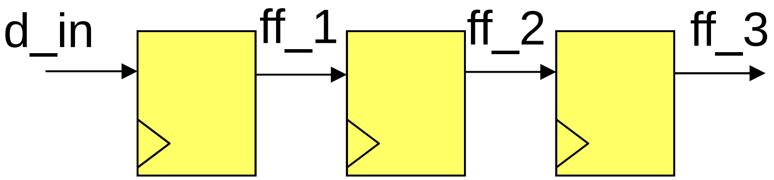
“Must not” in writing RTL code for flip-flop (9)

Do not use blocking procedural assignment.

Use nonblocking procedural assignment for FF.

```
always @ ( posedge clk )
begin
  ff_1 = d_in ;
  ff_2 = ff_1 ;
  ff_3 = ff_2 ;
end
```

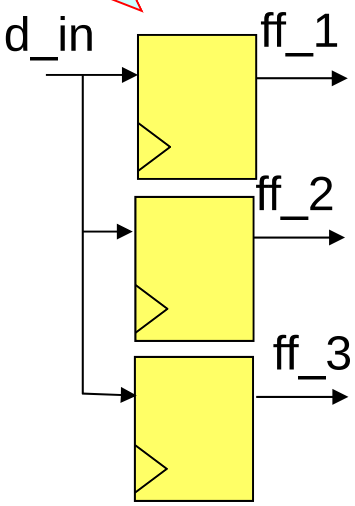
synthesizable, but may not work as intended.



RTL programmer's intention

different

Simulators behave differently from the above gate logic, therefore a synthesis tool may create the logic shown on the right.



net list created

Changing blocking procedural assignment to nonblocking procedural assignment will solve this problem.

```
ff_1 <= d_in ;
ff_2 <= ff_1 ;
ff_3 <= ff_2 ;
```

This code will create net list as intended.

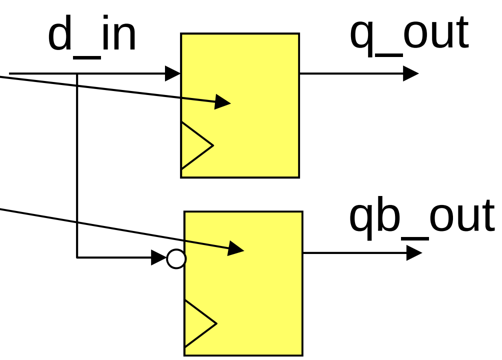
“Must not” in writing RTL code for flip-flop (10)

Do not define unnecessary signal in always construct for FF.

```
always @ ( posedge clk )
begin
 q_out <= d_in ;
 qb_out <= ~d_in ;
end
```

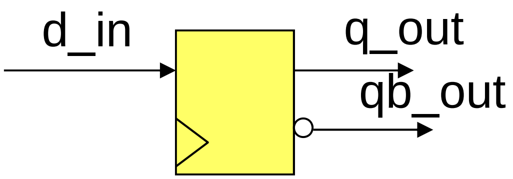
synthesizable, but create unnecessary FF.

All LHS in always construct are defined as FF. Therefore, RTL code on the left will create two FFs as below.



Moving qb_out out of always construct will solve the problem.

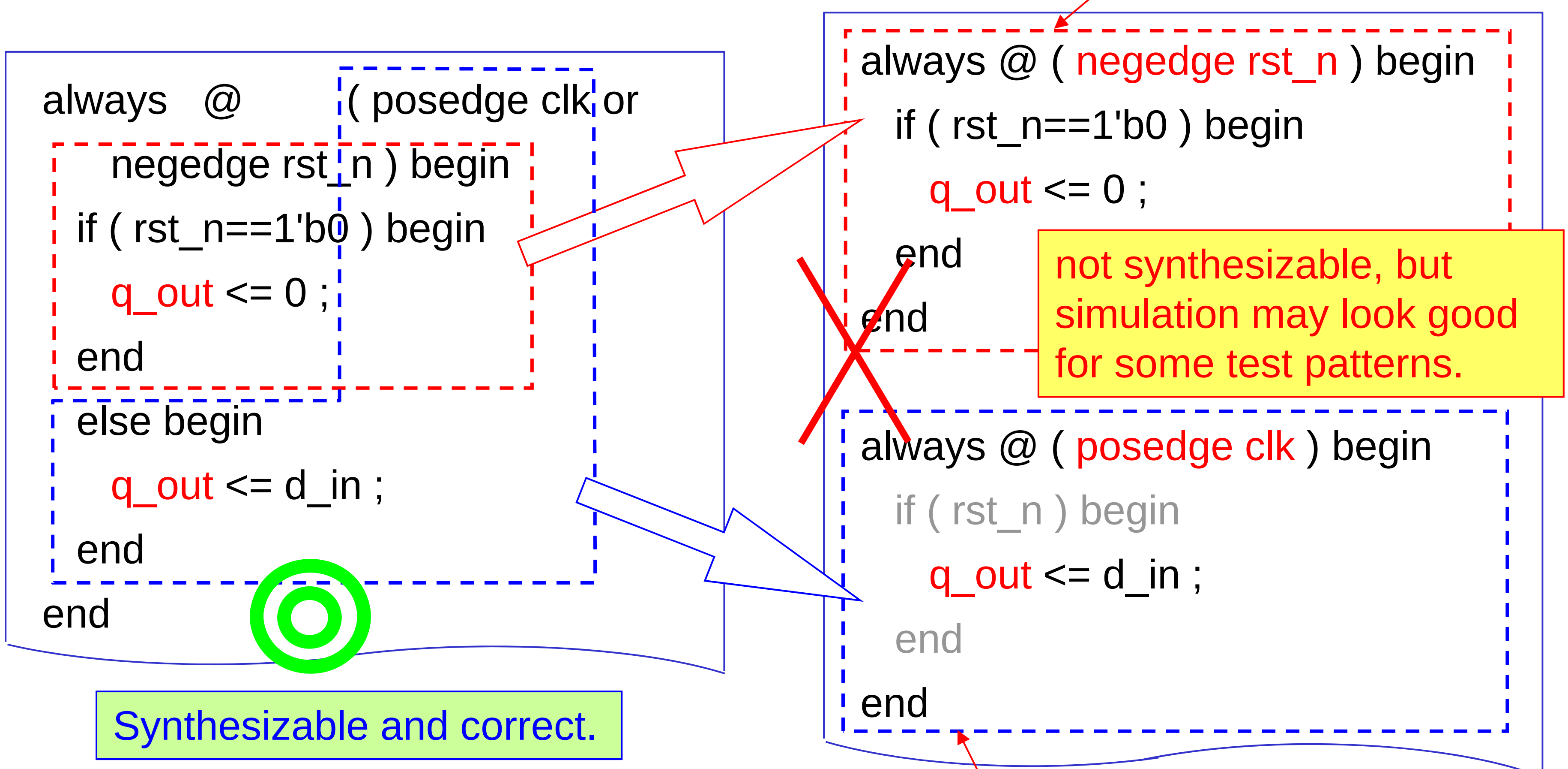
```
always @ ( posedge clk )
begin
  q_out <= d_in ;
end
assign   qb_out = ~q_out ;
```



This code will create only one FF.

“Must not” in writing RTL code for flip-flop (11)

Do not divide always construct for FF into two complementary codes such as reset path and non-reset path.



```
always   @   ( posedge clk or
    negedge rst_n ) begin
  if ( rst_n==1'b0 ) begin
    q_out <= 0 ;
  end
  else begin
    q_out <= d_in ;
  end
end
```

Synthesizable and correct.

always for reset case

```
always @ ( negedge rst_n ) begin
  if ( rst_n==1'b0 ) begin
    q_out <= 0 ;
  end
end
```

not synthesizable, but simulation may look good for some test patterns.

```
always @ ( posedge clk ) begin
  if ( rst_n ) begin
    q_out <= d_in ;
  end
end
```

always for non-reset case

“Must not” in writing RTL code for flip-flop (12)

Do not write one large always construct.

```
reg     q_state ; // FF
always @ ( posedge clk or negedge rst_n )
begin
    if ( rst_n  ==   1'b0 ) begin
          q_state <= initial value ;
      end
    else begin
        case (q_state )
            state1:     begin
                            logic1
                            q_sta
                        end
            state2:     begin
                            logic2;
                            q_state <= value2 :
                        end
                         :
                         :
                         :
            default:   q_state <= xxxx ;
        endcase
    end
end
```

Disintegrate one always construct into two, one for FF and one for combinational logic.

always for FF

```
reg     q_state ; // FF
reg     next_q_state ;  // non-FF
always @ ( posedge clk or negedge rst_n )
begin
    if ( rst_n  ==   1'b0 ) begin
          q_state <= initial value ;
      end
    else begin
           q_state <= next_q_state ;
      end
end
```

always for combinational logic

```
always @ ( ,,,, )
begin
    case ( q_state  )
        state1:  begin
                     logic1;
                     next_q_state = value1 :
                   end
         state2:   begin
                       logic2;
                        :
             default:   next_q_state = xxxx ;
      endcase
    end
end
```

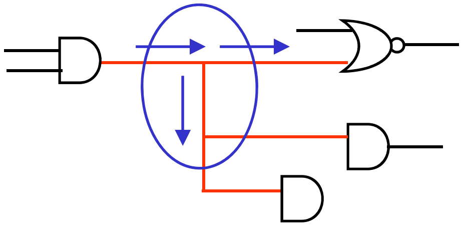
3.10.4 Always construct summary

When writing RTL code using always construct follow the rule below.

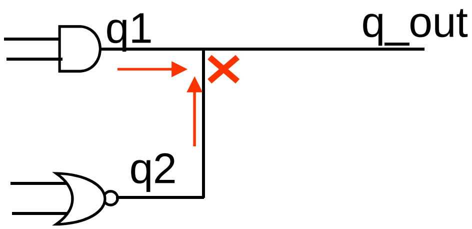
target logic	combinational logic	flip-flop	latch
event type in a sensitivity list	Change value events only	Change direction events only	Change value events only
signals in a sensitivity list	All signals appearing on RHS	Clock and if needed reset signal only	Clock and all signals appearing on RHS
sensitivity list example	(a or b or c or ,,,)	(posedge clk or negedge rst_n)	(clk or a or b ,,,)
applicable assignment	Blocking procedural assignment	Nonblocking procedural assignment	
programming guide	"If without else" is not allowed. Case items must cover all the cases.	Only simple if is allowed for reset signal.	Use simple if without else for clock signal.

3.11. Connection of signals

Because the input impedance of CMOS logic gate is very high, the output signal of a CMOS circuit can be connected to several inputs.



However, in general no more than one output signal is allowed to be connected to one input.



q1=High、q2=High → q_out=High

q1=Low、q2=Low → q_out=Low

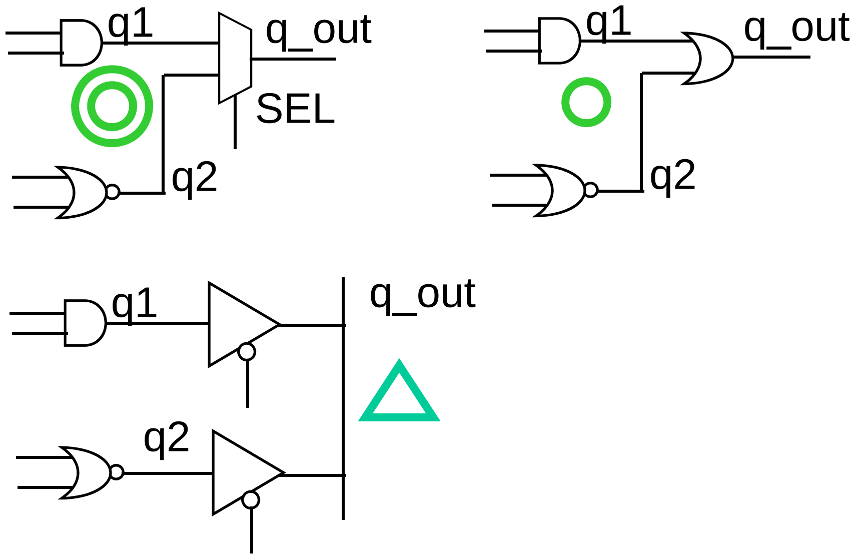
q1=High、q2=Low → q_out= ??

q1=Low、q2=High → q_out= ??

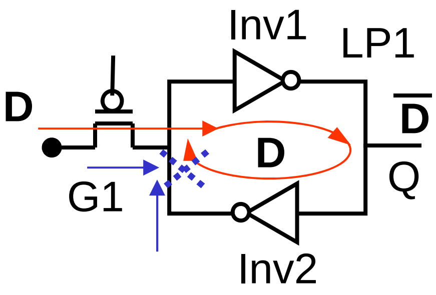
Because the value of q_out can not be decided in these cases, wiring two or more outputs to the same line is not recommended.

Output equivalent to OR gate may be the result, if High has stronger power than Low. However, never design based on this assumption.

Always use a gate or selector for connecting several outputs together.



In some special cases, outputs are wired OR.

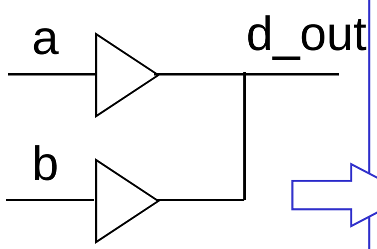


For example, in FF shown left, Inv1 and Inv2 are designed to have weak drivability, and output or G1 is designed to play a dominant role. Therefore, this loop will be in stable condition after G1 is set open to pass data D into the loop.

For the sake of simulation test, logic strength is defined in 8 levels.

supply0	supply1	Strong
strong0	strong1	
pull0	pull1	
large0	large1	
weak0	weak1	
medium0	medium1	
small0	small1	
highz0	highz1	Weak

Logic strength

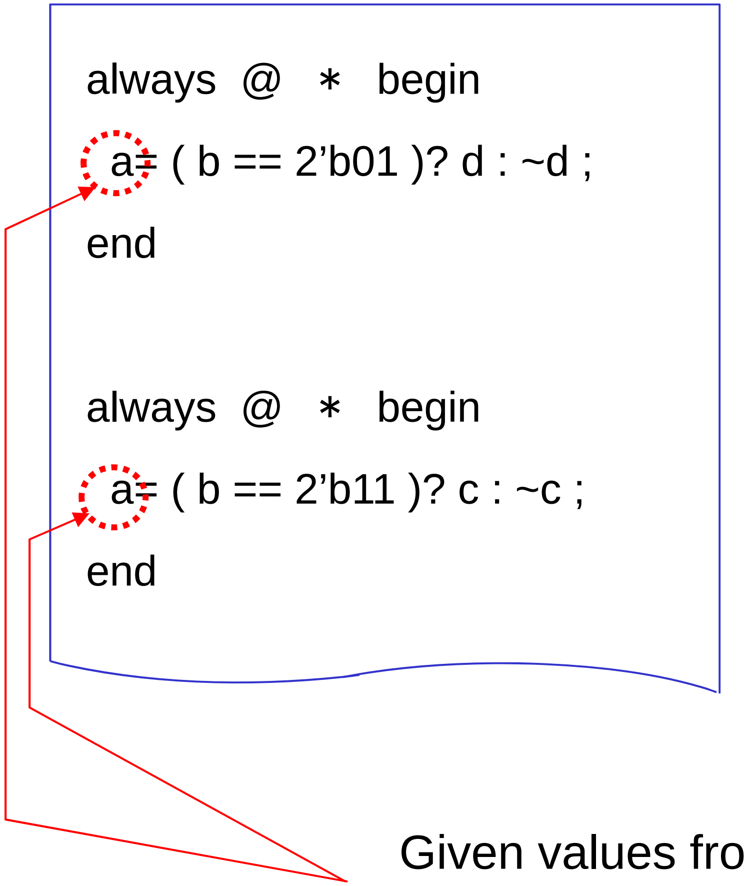


```
                :
assign (weak0, weak1) d_out = a ;
assign (strong0, strong1) d_out = b ;
                :
                :
```

Signal b is dominant

But, this feature is just for the test bench and only traditional primitive circuits use this. DA tools are not able to implement logic strength.

⇨ Do not use Logic Strength in your design.



```
always  @   ＊  begin
 a= ( b == 2’b01 )? d : ~d ;
end

always  @   ＊  begin
 a= ( b == 2’b11 )? c : ~c ;
end
```

The code on the left causes error in synthesis, because it can not be handled by the tool.
Never write code assigning values to the same variable from different structured procedures.

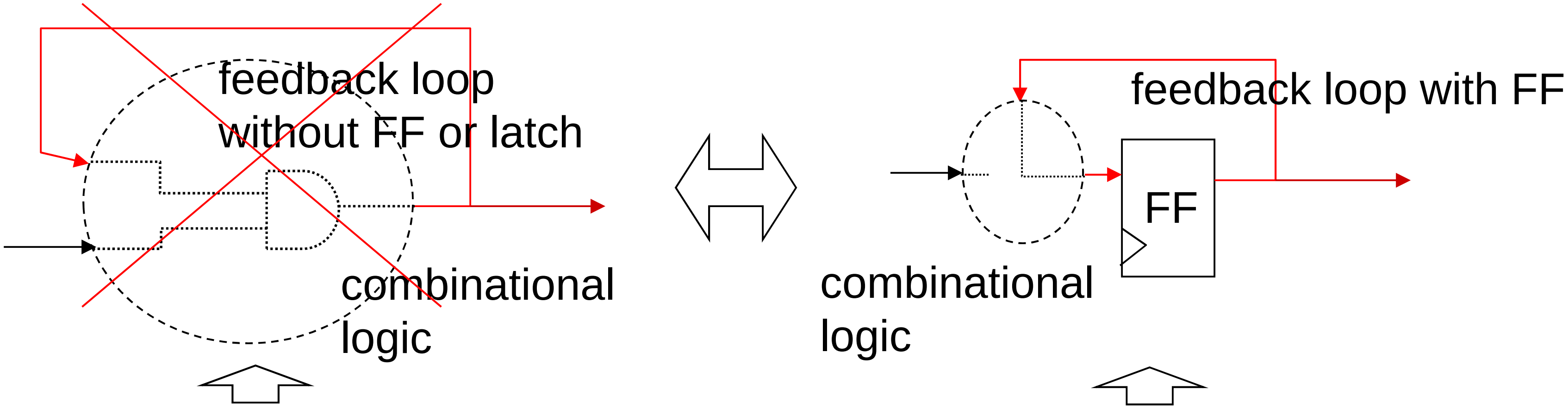
However, in RTL simulation, no err will be reported.
The result depend on which one of the two always constructs will be executed first.
(racing problem)

Given values from different structured procedures.

Never do this.

Feedback loop

→ Never code a feedback loop without FF or latch.



Never create logic with this kind of feedback loop.

This is OK.

~~assign a = b & a ;~~

This will create a feedback loop.

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

This will create a feedback loop with FF.

```
reg [3:0] wk ;
begin
   wk = a ;
   wk =  wk & b ;
   wk = wk | c ;
   wk = wk ^ d
   bad_sample = wk ;
end
```

Avoid programming confusingly similar to loop.

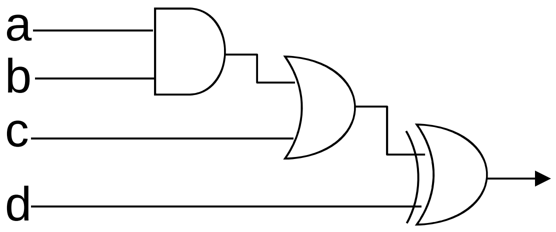
△

```
reg [3:0] wk1, wk2, wk3 ;

begin

wk1 = a & b ;
wk2 = wk1 | c ;
good_sample = wk2 ^ d ;

end
```

○

Because procedure part is executed serially (blocking procedural assignment), the above code will create logic shown below. However, it is not a recommended style.

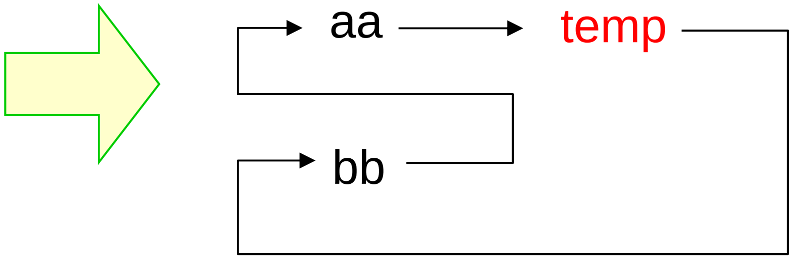
Avoid loop-like code.



```
reg [8:0] aa, bb ;
reg [8:0] temp ;

begin
   temp = aa ;
   aa = bb ;
   bb = temp ;
end
```

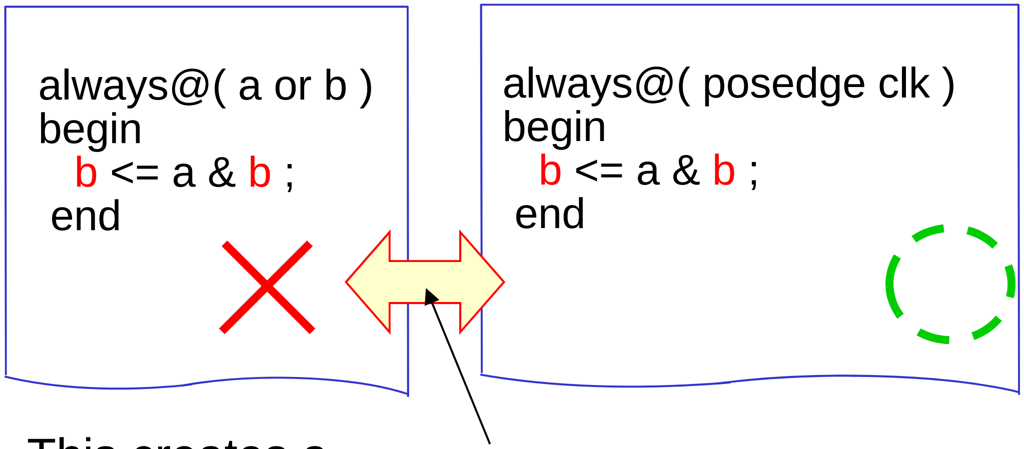
This is not allowed because of the loop.

This style of code is very common in software world, but it is not allowed in hardware logic.



Loop without any memory element is not allowed.

In simulation, each step is executed in serial, but in actual circuit, signals propagate in parallel.



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

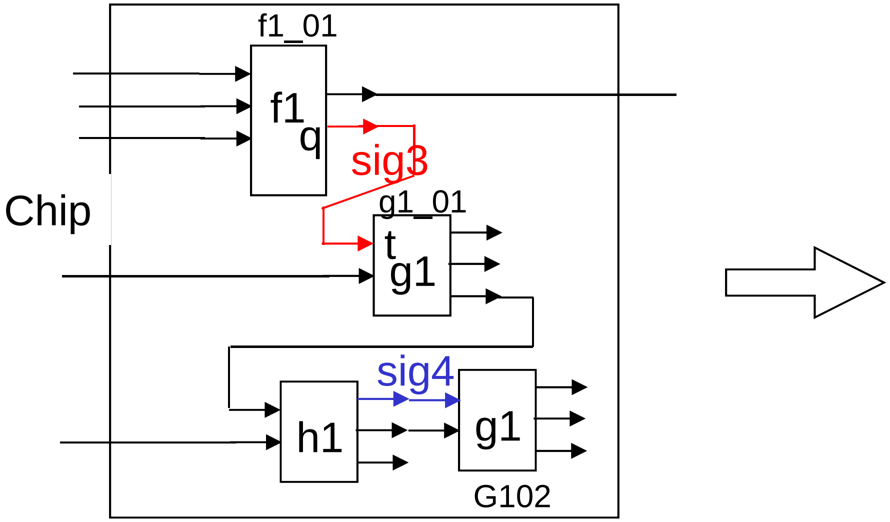
```
always@( posedge clk )
begin
   b <= a & b ;
 end
```

This creates a feedback loop.

These two are completely different.

Connection among modules

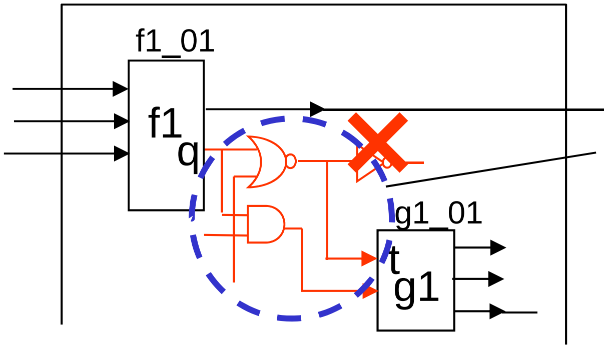
When logic is too big for one flat module, it can be divided into several modules. And the connection among modules can be done as below.



```
module Chip ( .. .. .. );
      :
      :
f1 f1_01(.. , .q(sig3),.. );
h1 h1_01(.. .. , .out1(sig4), .. ) ;
g1 g1_01( .t(sig3),.. );
g1 g1_02(.. .. , .t(sig4),.. );
      :
endmodule

module f1 (.. , q,.. );
       :
endmodule

module g1(t,.. .. );
       :
endmodule
```

instance name

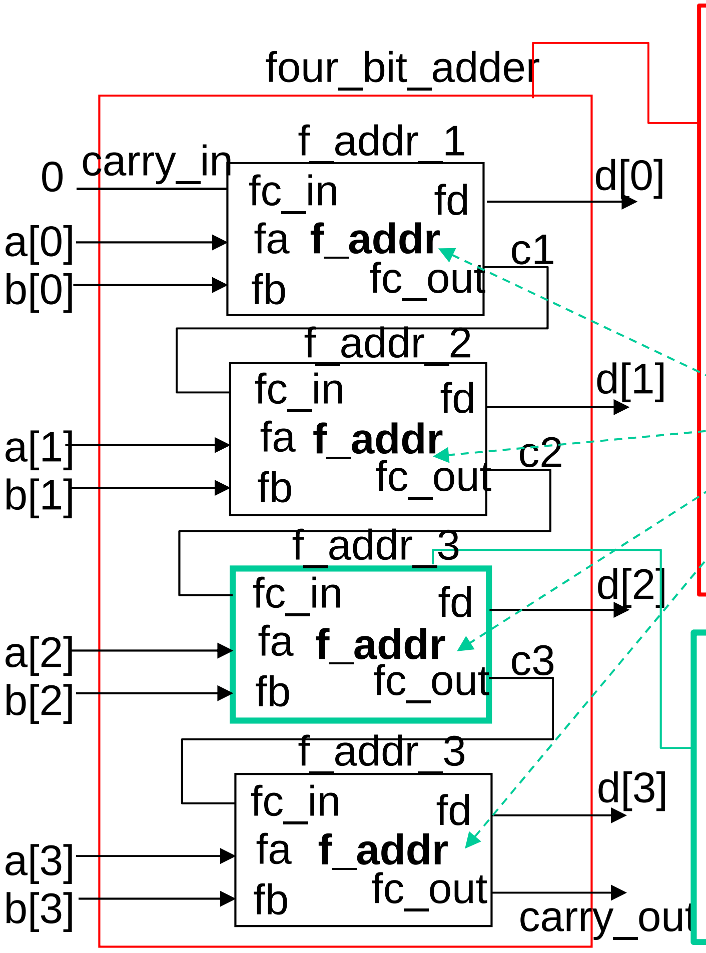
Do not insert any logic in between the modules. If inserted, it will make STA difficult.



Do not insert gates in between the modules.

A module may be very small. For example, when composing four bits adder, you may define 1 bit full adder as a module as below.

instance name



```
module four_bit_adder ( carry_in, a, b, d, carry_out);
input [3:0] a, b ;
input carry_in ;
output [3:0] d ;
output carry_out ;
wire [3:0] a, b ;
wire [3:0] d ;
wire carry_in, c1, c2, c3, carry_out ;
f_addr f_addr_1 ( .fa(a[0]), .fb(b[0]), .fd(d[0]), .fc_in(carry_in),.fc_out(c1) ) ;
f_addr f_addr_2 ( .fa(a[1]), .fb(b[1]), .fd(d[1]), .fc_in(c1),.fc_out(c2) ) ;
f_addr f_addr_3 ( .fa(a[2]), .fb(b[2]), .fd(d[2]), .fc_in(c2),.fc_out(c3) ) ;
f_addr f_addr_4 ( .fa(a[3]), .fb(b[3]), .fd(d[3]), .fc_in(c3),.fc_out(carry_out) ) ;
endmodule

module f_addr ( fc_in, fa, fb, fd, fc_out ) ;
input fa, fb, fc_in ;
output fd, fc_out ;
wire fa, fb, fd, fc_in, fc_out ;
assign fd =  fa ^ ( fb ^fc_in ) ;
assign fc_out = ( fa &fb ) | ( fb & fc_in ) | ( fc_in & fa ) ;
endmodule
```

Port connection by order. ←→ Port connection by name.

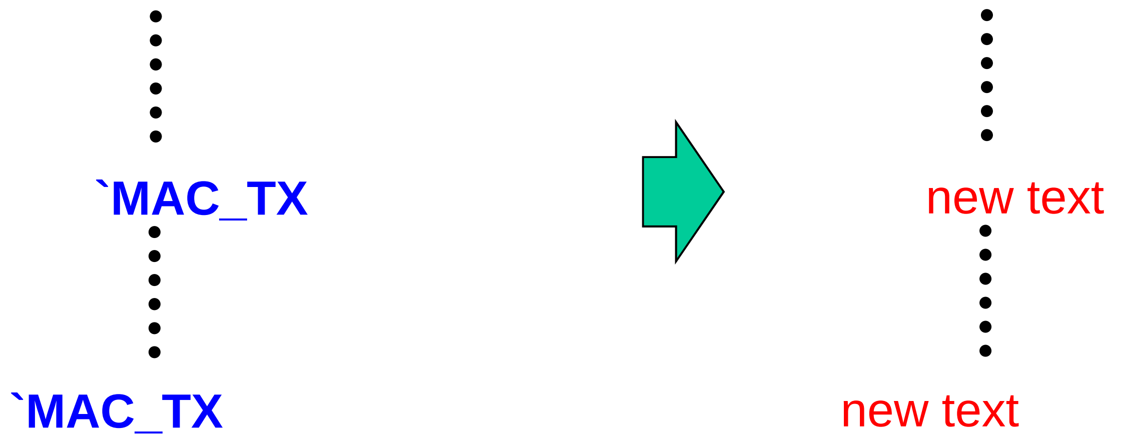
3.12. Compiler directives

The scope of compiler directives extends from the point where it is processed, across all files processed, to the point where another compiler directive supersedes it or the processing completes.

`define    `define *text_macro_identifier* macro_text

`define creates a macro for text substitution.

`define **MAC_TX** **new text**



`MAC_TX

`MAC_TX

new text

new text

`define *text_macro_identifier*(param1,param2,,,,) macro_text_with_param

```
`define max(a,b)   ((a) > (b) ? (a) : (b))

n = `max(p+q, r+s) ;
```

⬇

```
n = ((p+q) > (r+s)) ? (p+q) : (r+s) ;
```

Do not use define with arguments.

`undef

`undef *text_macro_identifier*

`undef shall undefine a previously defined text macro.

Escape character and define

The Verilog standard says “Escaped identifiers shall start with backslash and end with white space (space, tab, newline)”. But care must be taken when using define compiler directives with a backslash characters.

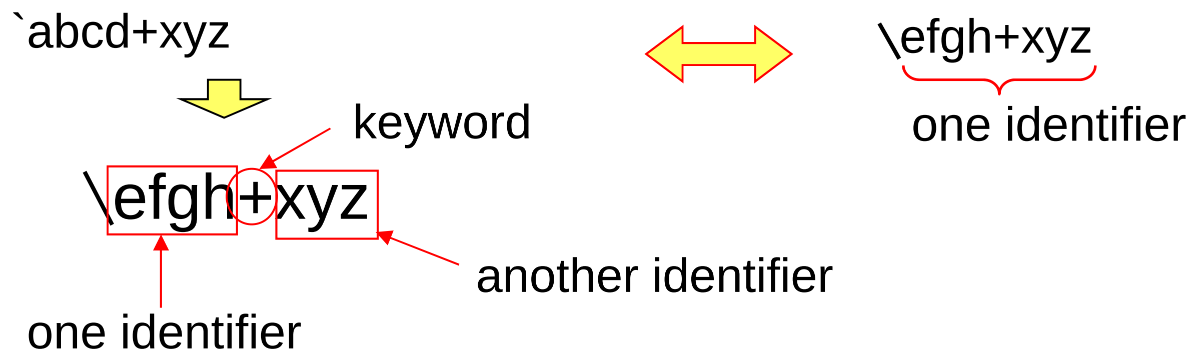
```
`define abcd   \efgh
```

The macro above will replace characters abcd with \efgh. Therefore an identifier shown left below must be equivalent to the identifier shown on the right.

`abcd+xyz ⟺ \efgh+xyz

But actually, they are not equivalent;

```
`define abcd   \efgh
```



`ifdef, `else, and `endif

```
`define SH4

`ifdef SH4
    This part is
    compiled.
`else
    This part is
    ignored.
`endif
```

```
`define M32

`ifdef SH4
    This part is
    ignored.
`else
  `ifdef M32
    This part is
    compiled.
  `else
    This part is
    ignored.
  `endif
`endif
```

```
`define SH4
`define MOBILE

`ifdef SH4
   `ifdef MOBILE
     This part is compiled.
   `else
     This part is ignored.
   `endif
`else
  `ifdef M32
    `ifdef MOBILE

    `else                     This part
                              is
    `endif                    ignored.
  `else

  `endif
`endif
```

In Verilog2001, **`elsif** compiler directive was introduced. It is equivalent to;
`else `ifdef ,,,

`ifndef, `else, `elsif, and `endif

Introduced in Verilog2001.

```
`define SH4

`ifndef SH4
    This part is
    ignored.
`else
    This part is
    compiled.
`endif
```

```
`define M32

`ifndef SH4
    This part is
    compiled.
`else
 `ifndef M32

               This part
 `else         is ignored.


 `endif
`endif
```

```
`define SH4
`define MOBILE

`ifndef SH4
  `ifndef MOBILE
                      This part
  `else               is
                      ignored.
  `endif
`else
 `ifndef M32
   `ifndef MOBILE
      This part is ignored.
   `else
      This part is compiled.
   `endif
 `else
      This part is ignored.
 `endif
`endif
```

Do not use define heavily for module variety

When making a derivative module by using `define A, a designer making the derivative is supposed to (1) verify his/her derivative with `define A is OK and (2) it does not affect the original.

However, (2) becomes very difficult in younger generations because number of possible combination of define increases very much.

Therefore, sometimes some of them are left unverified. This will cause a problem that such unverified combination will be used by others.

generation	a new derivative	possible combination of define	note
original	—	—	A designer is responsible to make sure that his/her modification does not affect the previous generations. But it becomes very hard to verify all of the possible combinations of define in older generations.
1st level	`define A	{ none } { A }	

3rd level	`define A
	`define B
		{ none } { A } { B }
		{ A B }	

4th level	`define B
	`define C
		{ none } { A } { B } { C }
		{ A B } { B C } { A C }
		{ A B C }	

Q3.12-1 Answer if the following explanation is correct or not.

(1) The following two pieces of code are the same for an RTL simulator.

```
`define OR_SEL

`ifdef AND_SEL
   y = a & b ;
`else
 `ifdef OR_SEL
    y = a | b ;
  `else
    y = a + b ;
 `endif
`endif
```

⇔

```
`define OR_SEL

     y = a | b ;
```

(2) The following two pieces of code are the same for an RTL simulator.

```
`define ST1 2'b01

`ST1 : begin
```

⇔

```
`define ST1 2'b01

2'b01 : begin
```

Q3.12-1 A sample answer.

(1) The following two pieces of code are the same for an RTL simulator.

```
`define OR_SEL

`ifdef AND_SEL
   y = a & b ;
`else
  `ifdef OR_SEL
     y = a | b ;
   `else
     y = a + b ;
  `endif
`endif
```

This is correct.

```
`define OR_SEL

     y = a | b ;
```

(2) The following two pieces of code are the same for an RTL simulator.

This is correct.

```
`define ST1 2'b01

`ST1 : begin
```

```
`define ST1 2'b01

2'b01 : begin
```

`begin_key_words and `end_key_words

A pair of directives, `begin_keywords and `end_keywords, can be used to specify what identifiers are reserved as keywords within a block of source code, based on a specific version of IEEE Std 1364. The `begin_keywords and `end_keywords directives only specify the set of identifiers that are reserved as keywords. The directives do not affect the semantics, tokens, and other aspects of the Verilog language.

`default_nettype

The directive `default_nettype controls the net type created for implicit net declarations. It can be used only outside of module definitions. Multiple `default_nettype directives are allowed. The latest occurrence of this directive in the source controls the type of nets that will be implicitly declared.

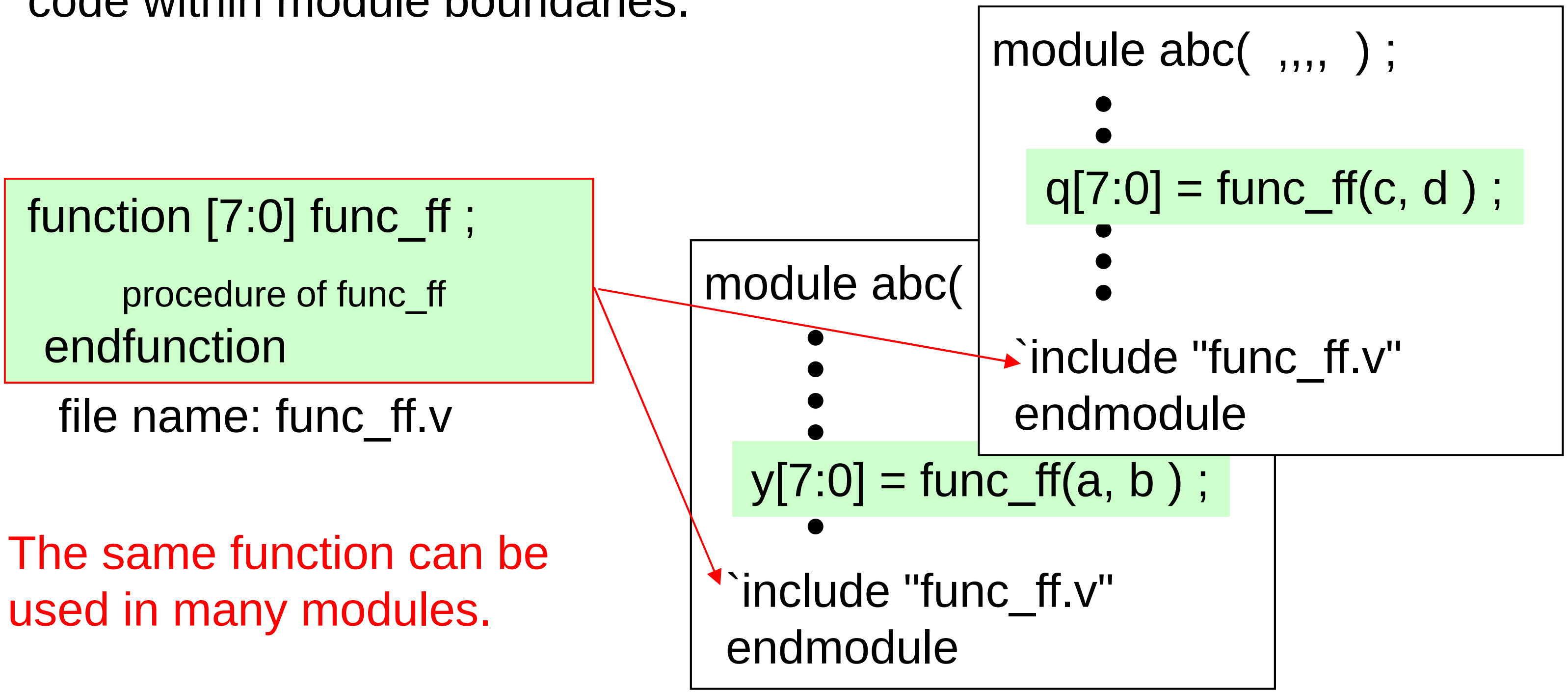
When the `default_nettype is set to none, all nets shall be explicitly declared. If a net is not explicitly declared, an error is generated.

`include

`include “*file_name*”

The file inclusion (`include) compiler directive is used to insert the entire contents of a source file in another file during compilation. The result is as though the contents of the included source file appear in place of the `include compiler directive.

The `include compiler directive can be used to include global or commonly used definitions and tasks without encapsulating repeated code within module boundaries.



```
function [7:0] func_ff ;
        procedure of func_ff
 endfunction
```

file name: func_ff.v

```
module abc(
   •
   •
   •
   •
 y[7:0] = func_ff(a, b ) ;
   •
`include "func_ff.v"
endmodule
```

```
module abc(  ,,,,  ) ;
   •
   •
 q[7:0] = func_ff(c, d ) ;
   •
   •
   •
`include "func_ff.v"
endmodule
```

The same function can be used in many modules.

`timescale

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

The `timescale compiler directive specifies the unit of measurement for time and delay values and the degree of accuracy for delays in all modules that follow this directive until another `timescale compiler directive is read.

If there is no `timescale specified or it has been reset by a `resetall directive, the time unit and precision are simulator-specific. It shall be an error if some modules have a `timescale specified and others do not.

3.13 Generate constructs

In Verilog2001, two kinds of generate constructs are introduced. They are loop generate constructs and conditional generate constructs.

3.13.1 Loop generate constructs

Loop generate constructs allow for modules with repetitive structure to be described more concisely. An example is shown below.

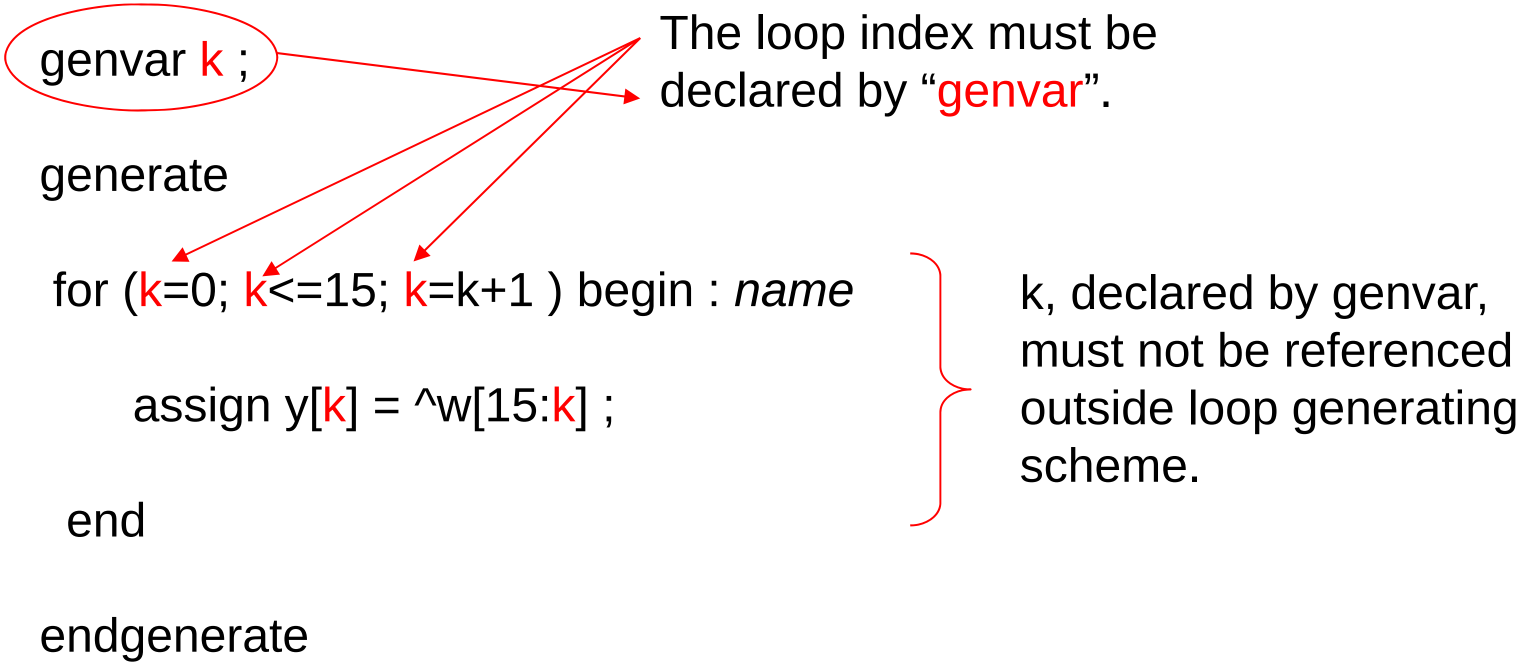
In Verilog1995	In Verilog2001

assign y[0] = ^w[15:0] ;
assign y[1] = ^w[15:1] ;
assign y[2] = ^w[15:2] ;
⋮
assign y[14] = ^w[15:14] ;
assign y[15] = ^w[15:15] ;
	genvar k ;
	
	generate
	for (k=0; k<=15; k=k+1) begin : *name*
	assign y[k] = ^w[15:k] ;
	end
	endgenerate

There is no way to write continuous assignment in for statement, because for statement is applicable only in structured procedures.	Continuous assignment can be used in for statements. Loop index “k” must be declared as “genvar”. for-loop block must have a name.

“generate” and “endgenerate” keywords may be used in a module to identify a *generate region*. A generate region is a textual span in the module description where generate constructs may appear. Using “generate” and “endgenerate” is optional but if the generate keyword is used, it shall be matched by an endgenerate keyword.

A loop index declared by genvar keyword is treated as 32-bit integer, but it does not exist at simulation time. It shall not be referenced anywhere other than in a loop generating scheme.



```
genvar k ;

generate

 for (k=0; k<=15; k=k+1 ) begin : name

      assign y[k] = ^w[15:k] ;

  end

endgenerate
```

The loop index must be declared by “genvar”.

k, declared by genvar, must not be referenced outside loop generating scheme.

3.13.2 Conditional generate constructs

Conditional generate constructs, if-generate and case-generate, select at most one generate block from a set of alternative generate blocks based on constant expressions evaluated during elaboration. The selected generate block, if any, is instantiated into the module.

if-generate	case-generate

parameter P1=8 ;
 	
generate
if (P1 <=8) begin
  bus_8bit   bus8_bit_01 (,,,,) ;
end
else begin
 bus_16bit   bus16_bit_01 (,,,,) ;
end
endgenerate
	parameter P1=8 ;
	
	generate
	case (P1) begin
	 8 : begin
	          bus_8bit bus8_bit_01 (,,,,) ;
	       end
	default : begin
	          bus_16bit bus16_bit_01 (,,,,) ;
	       end
	endcase
	endgenerate

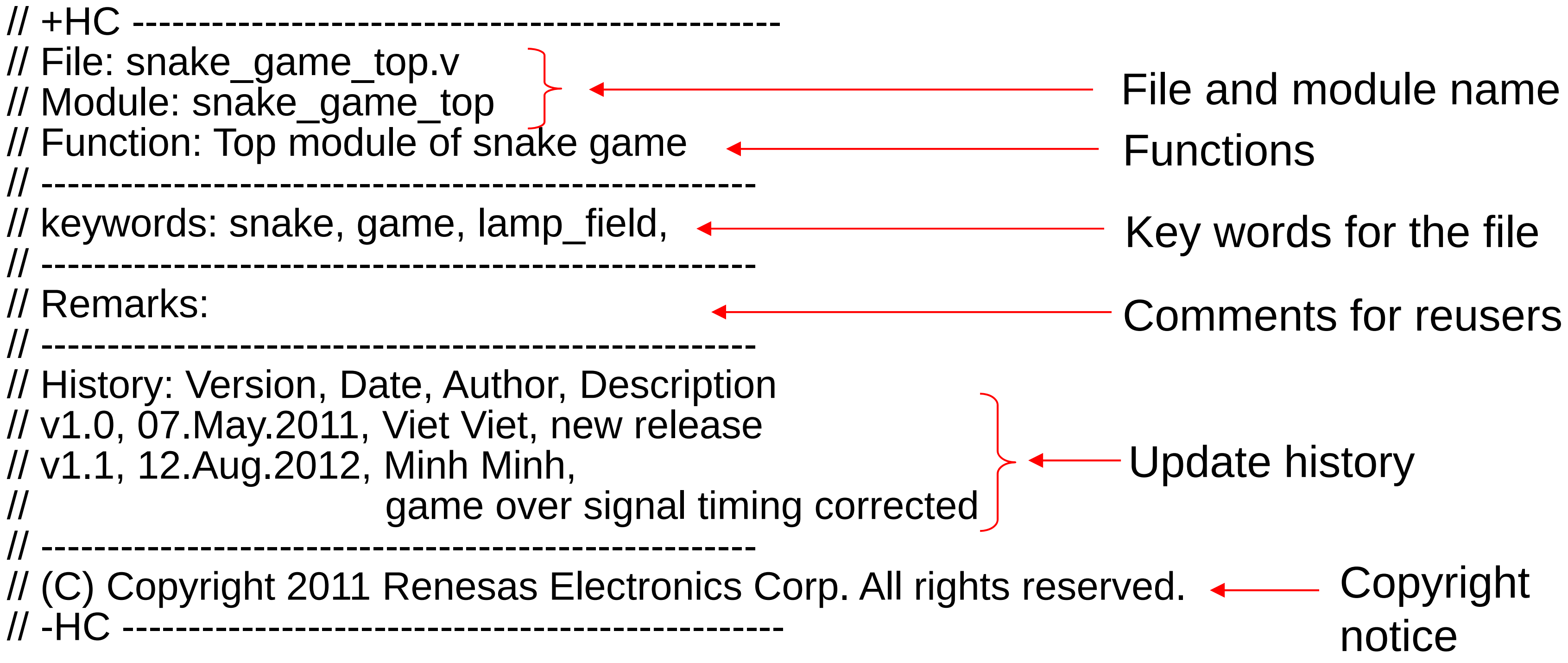
Note that only one of bus_8bit or bus_16bit is instantiated.

```
if_generate_construct ::=
        if ( constant_expression ) generate_block_or_null
        [ else generate_block_or_null ]

case_generate_construct ::=  case ( constant_expression )
        case_generate_item { case_generate_item } endcase

case_generate_item ::=
        constant_expression { , constant_expression } : generate_block_or_null
       | default [ : ] generate_block_or_null

generate_block ::=  module_or_generate_item
      | begin [ : generate_block_identifier ] { module_or_generate_item } end

generate_block_or_null ::= generate_block | ;

module_or_generate_item ::= local_parameter_declaration ;
      | continuous_assign | module_instantiation | initial_construnt
      | always_construct | loop_generage_construct
      | conditional _generate_construc | parameter_override
      | module_or_generate_item_declaration
```

3.14 RTL programming summary

Follow the following steps to write Verilog RTL source program of a module.

Step 1. Write a header for a file you are going to create to help others understand what are written, and help yourself to remind what are written, in the file.



```
// +HC ------------------------------------------------
// File: snake_game_top.v
// Module: snake_game_top
// Function: Top module of snake game
// ----------------------------------------------------
// keywords: snake, game, lamp_field,
// ----------------------------------------------------
// Remarks:
// ----------------------------------------------------
// History: Version, Date, Author, Description
// v1.0, 07.May.2011, Viet Viet, new release
// v1.1, 12.Aug.2012, Minh Minh,
//                         game over signal timing corrected
// ----------------------------------------------------
// (C) Copyright 2011 Renesas Electronics Corp. All rights reserved.
// -HC ------------------------------------------------
```

File and module name

Functions

Key words for the file

Comments for reusers

Update history

Copyright notice

When joining a project, discuss with a team leader what kind of information must be included in the header.

Step 2. Give a unique name to a module.

```
//////////////////////////////////
//  header
//////////////////////////////////
module module_name;

endmodule
```

Give a unique meaningful name,
Use 5 to 16 lowercase characters.

Step 3. Find how many inputs and outputs are needed for the module.
And give them unique meaningful names, and then list them up in a port list in the order of clock, reset, input and output signals.

```
//////////////////////////////////
//  header
//////////////////////////////////
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, ) ;

endmodule
```

List up all the interface signals including clock and reset in a module port list.

Give a unique meaningful name for each signal,
Use 5 to 16 lowercase characters.

Step 4. Declare parameters for constants, such as bit-width of signals used in the module, initial values of FFs, etc. to improve readability and reusability of the module.

```
//////////////////////////////////
//  header
//////////////////////////////////
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, ) ;
// parameters
  parameter BW_IN_B = 16 ; // Bit width of in_b,
                                 //  BW_IN_B can be either 4, 8, 16, 32, or 64
  parameter BW_OUT_D = 8 ; // Bit width of out_d,
                                // BW_OUT_D can be either 2, 4, 8, 16, or 32
  parameter ,,,,,,,

endmodule
```

Declare parameters and give comments for them.
Parameter ranges must also be declared.

Step 5. Declare port using input/output keywords for all the signals listed up in a module port list. Use range specification for multi-bit signals.

```
//////////////////////////////////
//  header
//////////////////////////////////
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, ) ;
  parameter BW_IN_B = 16 ; // Bit width of in_b,
                                        //  BW_IN_B can be either 4, 8, 16, 32, or 64
  parameter BW_OUT_D = 8 ; // Bit width of out_d,
                                         // BW_OUT_D can be either 2, 4, 8, 16, or 32
  parameter ,,,,,,,
// ports
  input clk, rst ;  // clock and reset signal
  input in_a;  //  comment, ,,,,,,,
  input [BW_IN_B -1 :0] in_b ; // ,,,,
  output out_c ; // ,,,,,,,,,
  output [BW_OUT_D -1 :0] out_d ; // ,,,,,,

endmodule
```

Declare input and output for all the signals listed in a module port list.
Use range specification for multi-bit signals.

Give comments to describe each port.

Step 6. Declare data type for input ports using wire keyword.

```
/////////////////////////////////
//  header
/////////////////////////////////
module module_name ( clk, rst, in_a, in_b,,,, out_c, out_d,,,, ) ;
// parameters
  parameter ,,,,
  parameter ,,,,,,,
// ports
  input clk, rst ;
  ,,,,,,
  ,,,
  output [BW_OUT_D -1 :0] out_d ;
// input data type
  wire ckl, rst ;
  wire in_a ;
  wire [BW_IN_B -1:0] in_b ;

endmodule
```

Declare data type of all inputs by using wire keyword. Use the same range specification for multi-bit signals.

Step 7-1. Find if output signals can be described by using continuous assign statement or not. For those output signals which can be described as “assign out_c = some_expression ;”, declare their data type by using wire keyword.

```
//////////////////////////////////
//  header
,,,,,,,,
,,,,,
// input data type
  wire ckl, rst ;
  wire in_a ;
  wire [BW_IN_B -1:0] in_b ;
  ,,,,
// output data type
  wire out_c ;
  wire [BW_OUT_D -1:0] out_d ;


endmodule
```

Declare data type of outputs by using wire keyword if they appear on LHS of continuous assign statements.
Use the same range specification for multi-bit signals.

Step 7-2. For those output signals which can not be defined by continuous assign statements and have to be defined by procedural assign statements, declare their data type by using reg keyword.

Use reg keyword if a signal appears on LHS of procedural assign statements.

```
//////////////////////////////////
//  header
,,,,,,,,
// output data type
  ,,,,,,
  ,,
  wire [BW_OUT_D -1 :0] out_d ;
 reg  out_f ;  // non-FF
 reg [BW_OUT_G -1 :0] out_g ; // FF
 reg [BW_OUT_P -1 :0] out_p ; // non-FF


endmodule
```

Give comments to describe output arguments. And also give note to make it clear that if they are to be mapped into FF or not.

Use the same range specification for multi-bit signals.

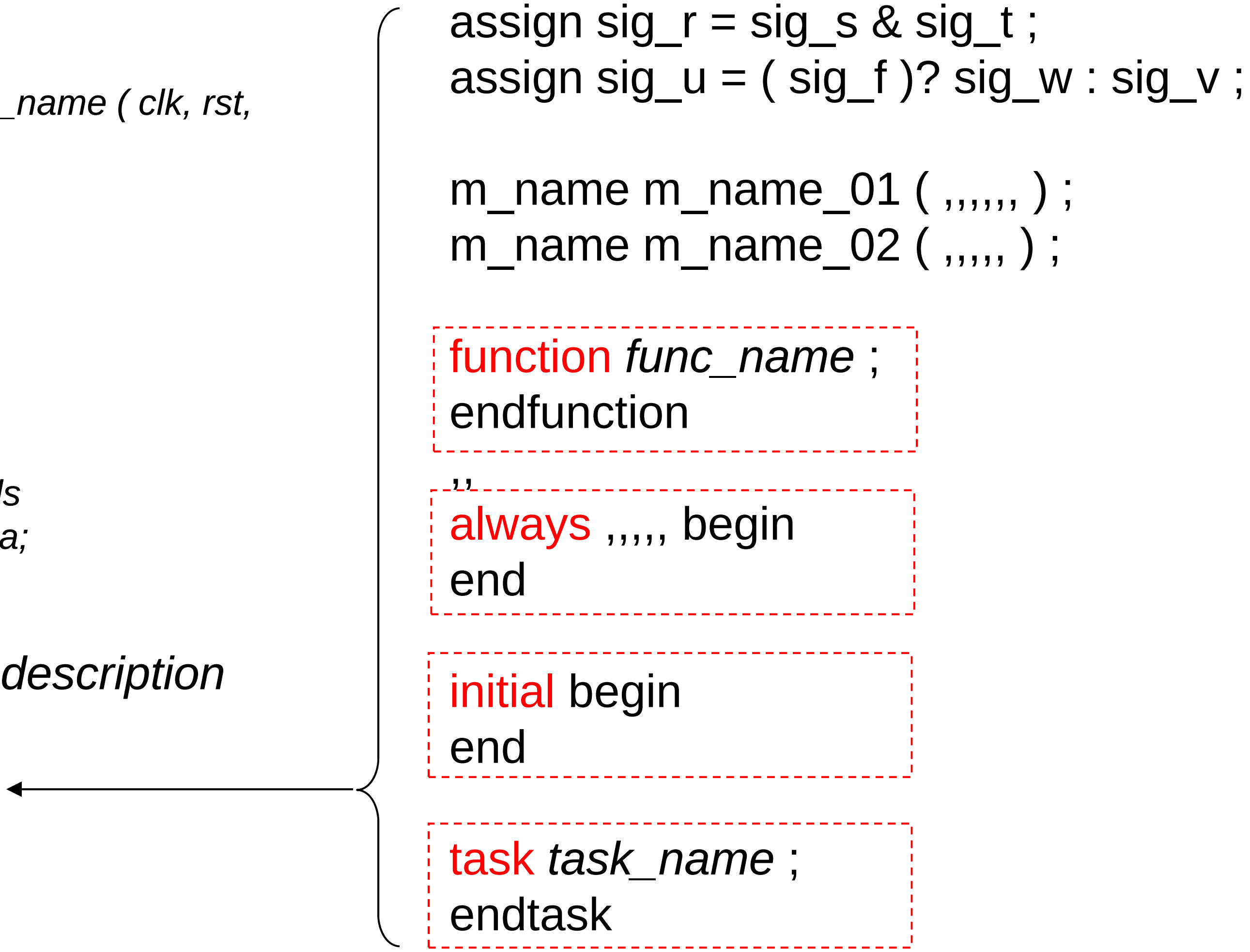
Step 8. Declare internal signals by using wire or reg keyword depending on if they appear on LHS of continuous assign or procedural assign statements.

```
////////////////
// header
,,,
,,
 reg [BW_OUT_P -1:0] out_p ; // non-FF
// internal signals
  wire intnl_sig_a; //,,,,
  wire [BW_SIG_B -1:0] intnl_sig_b ; // ,,,
  reg intnl_sig_c ;  // non-FF
  reg [BW_SIG_D -1:0] intnl_sig_d ; // FF

endmodule
```

Declare data type of internal signals.
Use wire keyword if they appear on LHS of continuous assign statements,
use reg keyword if they appear on LHS of procedural assign statements.
Use range specification for multi-bit signals.
Give comments for reg data type to make it clear that they are to be mapped into FF or not.
All internal signals must be commented to describe what they are used for.

Step 9. Describe logic using continuous assign statements, procedures and module instanciation.

```
//////////////////////
//  header
//////////////////////
module module_name ( clk, rst,
// parameters
  parameter ,,,
  ,,,,,,
// ports
  input clk, rst ;
  ,,,,,,,,,
// data type
  wire ,,,,
// internal signals
  wire intnl_sig_a;
 ,,,,
,,
// start logic description




endmodule
```



```
assign sig_r = sig_s & sig_t ;
assign sig_u = ( sig_f )? sig_w : sig_v ;

m_name m_name_01 ( ,,,,,, ) ;
m_name m_name_02 ( ,,,,, ) ;

function func_name ;
endfunction
,,
always ,,,,, begin
end

initial begin
end

task task_name ;
endtask
```

The following set of code lines must be written near to each other if they are related to the same logic block. Do not mix up unrelated code lines together.

```
assign sig_r = sig_s & sig_t ;
assign sig_u = ( sig_f )? sig_w : sig_v ;

m_name m_name_01 ( ,,,,,, ) ;
m_name m_name_02 ( ,,,,, ) ;

function func_name ;
endfunction
,,
always ,,,,, begin
end

task task_name ;
endtask
```

Step 9-1. Define FFs by using always constructs with posedge clk in a sensitivity list.

```
////////////////////////
//  header
,,,,
,,
// start logic description

// always for FF
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n == 1’b0  ) begin
   sig_ff <= INTL ;
  end
  else begin
    sig_ff <= next_sig_ff ;
  end
end


endmodule
```

(1) LHS must be declared by reg keyword.
(2) Use nonblocking procedural assignment.
(3) Write change direction event of clock and reset (for asynchronous reset) in the sensitivity list.

Write FFs using always constructs.

Apply delay to avoid racing,
if the project policy says so.

Step 9-2. Describe logic using continuous assign statements.

```
////////////////////////
//  header
,,,,
,,
// start logic description
// always for FF
always @ (posedge clk ,,,, )
,,,,
,,
// continuous assigns
assign sig_w = sig_u & sig_y ; // comment
assign sig_h = ( sig_en )? sig_s : sig t ; // comment
assign ,,,,,,


endmodule
```

Write logic blocks using continuous assign statements.

(1) LHS must be declared by wire keyword.
(2) Do not use @ nor #.

Step 9-3. Describe logic by instanciating lower level modules.

```
////////////////////////
//  header
,,,,
,,
// start logic description
//  always for FF
,,
// continuous assigns
assign sig_w = sig_u & sig_y ; // comment
assign sig_h = ( sig_en )? sig_s : sig t ; // comment
assign ,,,,,,
// module instanciation
m_name m_name_01 ( ,,,,,, ) ;
m_name m_name_02 ( ,,,,, ) ;


endmodule
```

(1) Output port must be connected to wire.
(2) Input port can be connected to wire or reg.

syntax

```
module_name module_name_01
            ( .port1_name(connecting_signal_name),
              .port2_name(connecting_signal_name),
               ,,,,,
              );
```

Step 9-4. Define combinational logic blocks by functions.

```
////////////////////////
//  header
,,,,
,,
assign ,,,,,,
// module instanciation
m_name m_name_01 ( ,,,,,, ) ;
,,,
// functions
function [BW_FUNC -1:0] func;
input [BW_F_IN -1:0] f_in_a ;

 ,,,,
 func = ,,,, ;
endfunction

endmodule
```

syntax

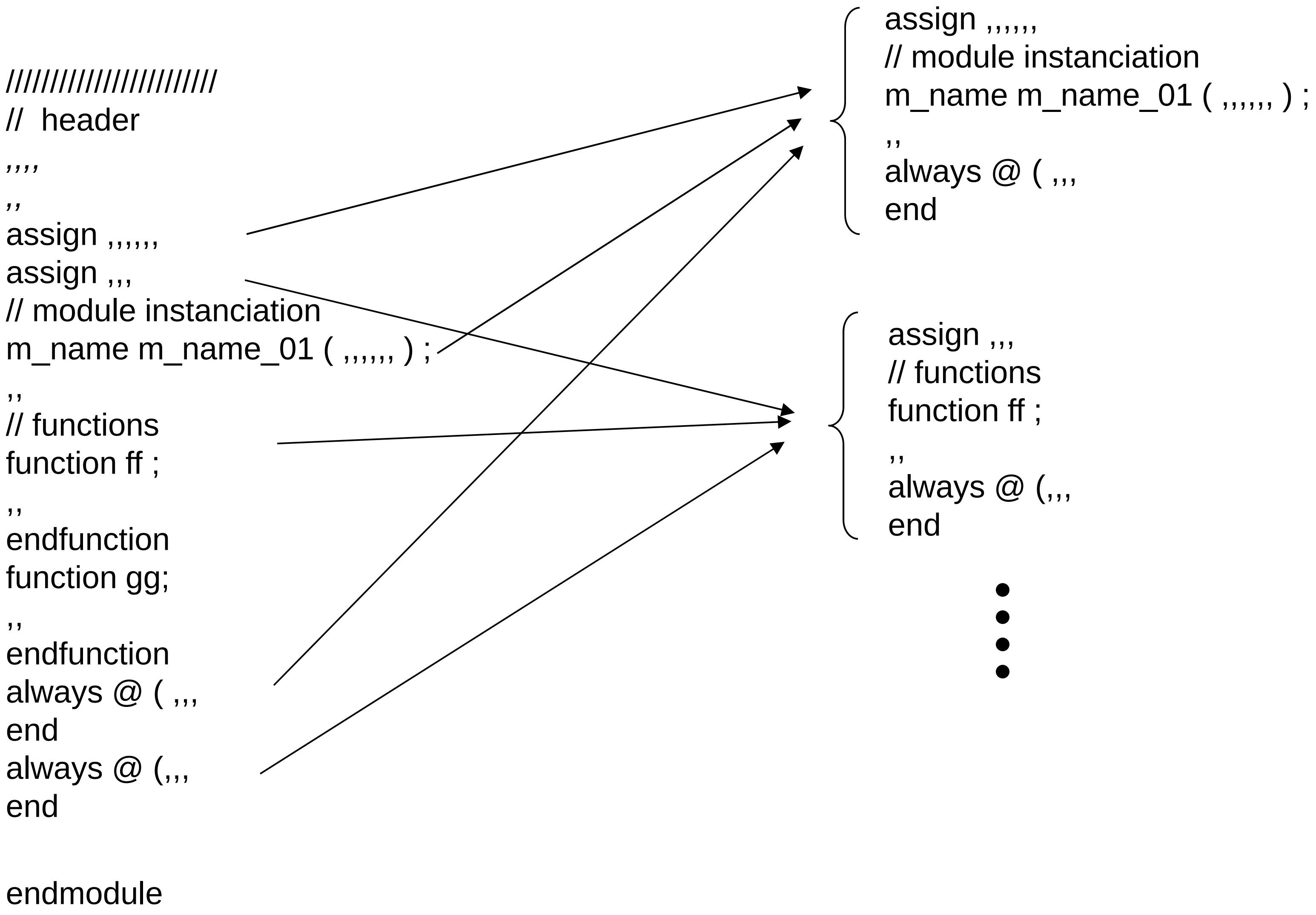
```
function range_declaration function_name ;
input range_declaration input_name ;
input range_declaration input_name ;
reg local_wk ;

function_name = ,,,, ;
endfunction
```

(1) Internal work must be declared by reg keyword.
(2) Do not use @ nor #.
(3) Use blocking procedural assignment only.
(4) Declare range if func is multi-bit.

Step 9-5. Define combinational logic blocks by always constructs.

```
////////////////////////
//  header
,,,,
,,
assign ,,,,,,
// module instanciation
m_name m_name_01 ( ,,,,,, ) ;
,,
// functions
,,,
// always for combinational logic
always @ ( sig_a or sig_b ,,, ) begin
 if ( ,,,,) ,,
 ,,,
 sig_w = ,,, ;
end

endmodule
```

Step 10. Review the code lines. If code lines related to the same logic block are distributed among other code lines, move them into one part so that the logic part can be seen at a glance.

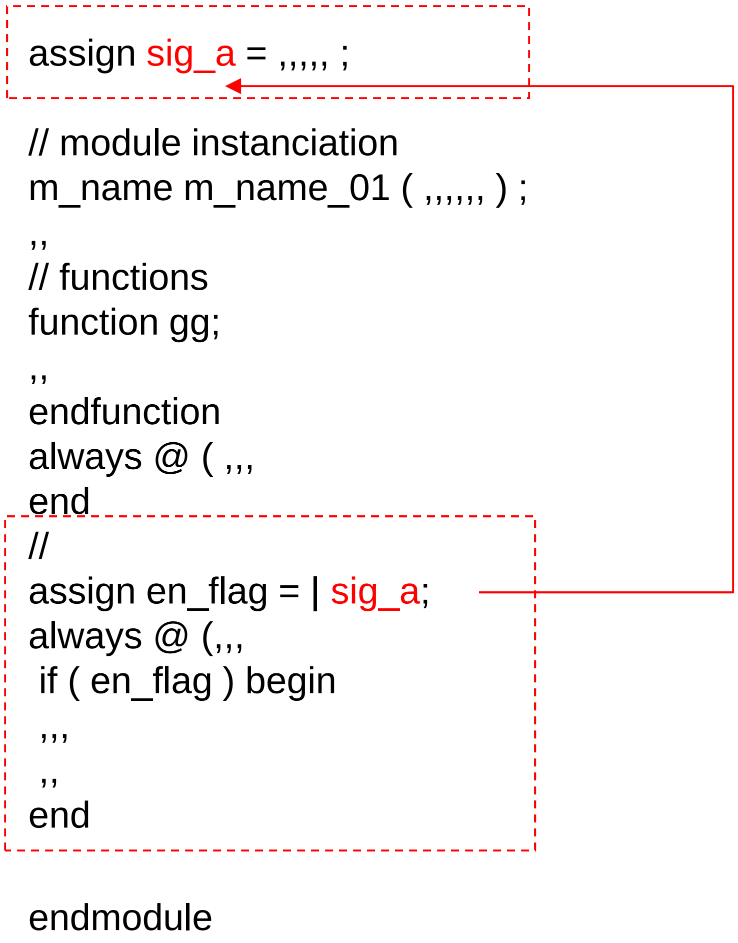


```
////////////////////////
//  header
,,,,
,,
assign ,,,,,,
assign ,,,
// module instanciation
m_name m_name_01 ( ,,,,,, ) ;
,,
// functions
function ff ;
,,
endfunction
function gg;
,,
endfunction
always @ ( ,,,
end
always @ (,,,
end

endmodule
```

```
assign ,,,,,,
// module instanciation
m_name m_name_01 ( ,,,,,, ) ;
,,
always @ ( ,,,
end
```

```
assign ,,,
// functions
function ff ;
,,
always @ (,,,
end
```

Step 11. Review the code lines. If code lines creating flags are placed far from the logic block where the original signal is generated, move them near to the original signal generating block.



```
assign sig_a = ,,,,, ;

// module instanciation
m_name m_name_01 ( ,,,,,, ) ;
,,
// functions
function gg;
,,
endfunction
always @ ( ,,,
end
//
assign en_flag = | sig_a;
always @ (,,,
 if ( en_flag ) begin
 ,,,
 ,,
end

endmodule
```

Move this flag signal generating code line into the code block generating sig_a.

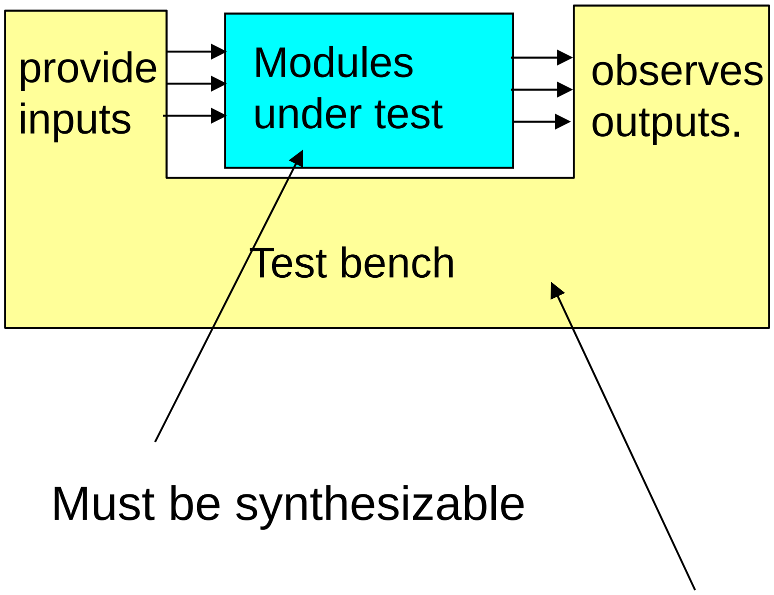
Step 12. Review the code lines to check if readability is good and if they are easy to understand and to reuse.
Improve the understandability, readability, and reusability.

3.15. Test bench

3.15.1 Structure of test bench

A test bench is a program which can give arbitrary inputs to modules under test and observe their outputs.

A test bench can be written in the form of a module.
We can use all features of Verilog programming technique, because test bench itself does not have to be synthesized.



provide inputs

Modules under test

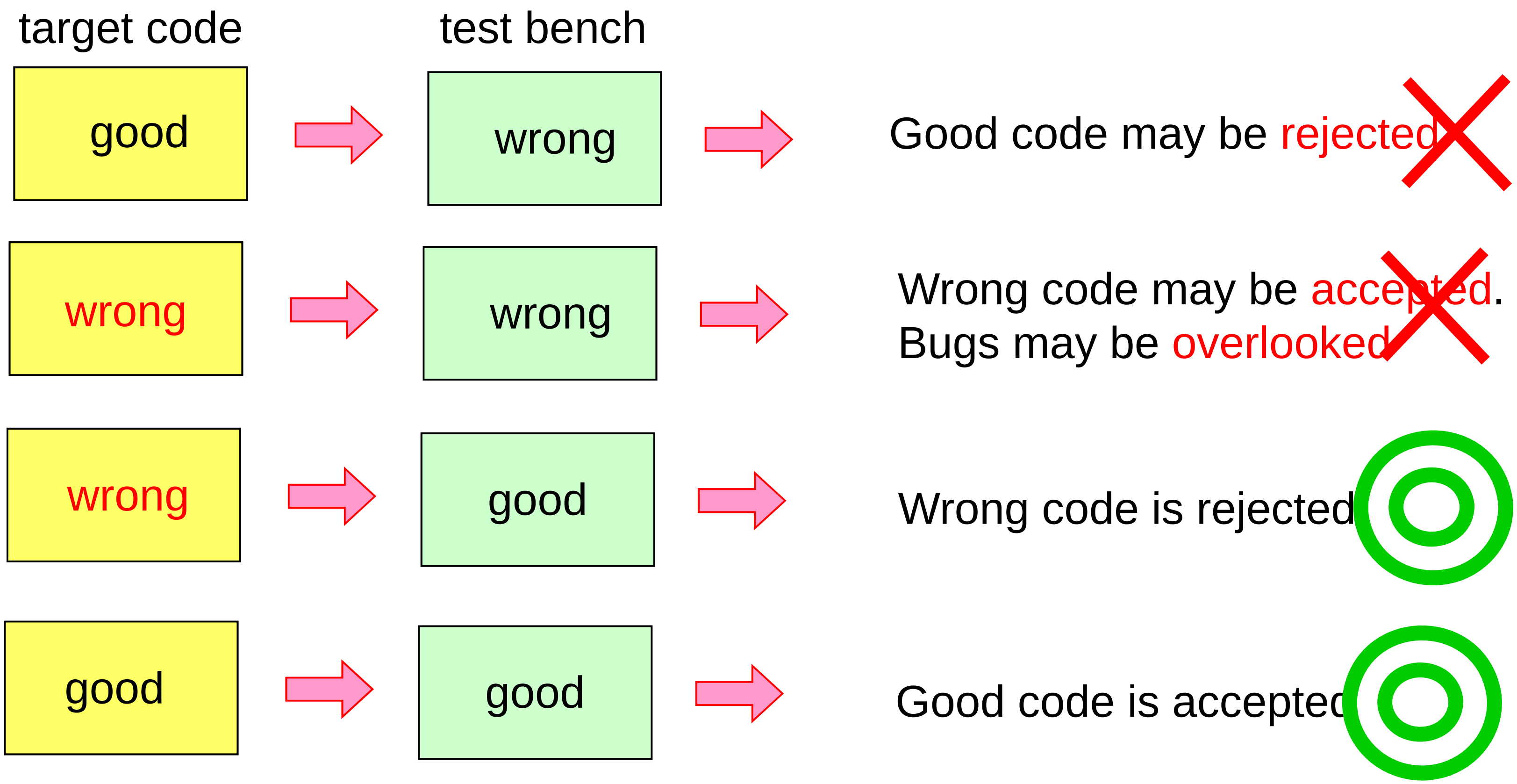
observes outputs.

Test bench

Must be synthesizable

May not be synthesizable

Test bench vs. target code which is more important??

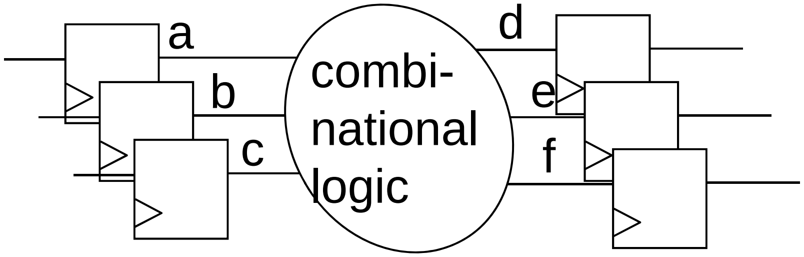


target code		test bench		
good	→	wrong	→	Good code may be rejected ✗
wrong	→	wrong	→	Wrong code may be accepted. Bugs may be overlooked ✗
wrong	→	good	→	Wrong code is rejected ◎
good	→	good	→	Good code is accepted ◎

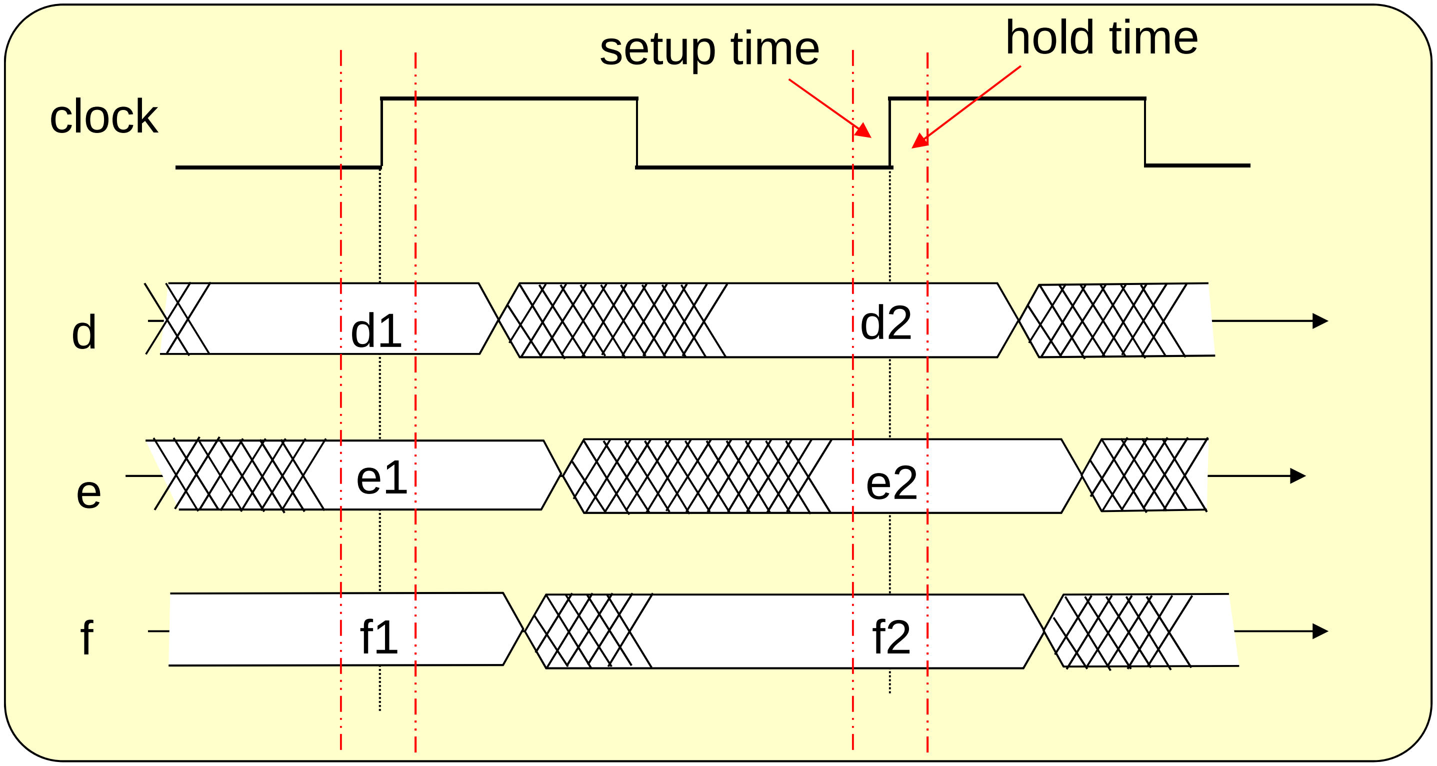
Damage of wrong test bench ≫ Damage of wrong target code

Test bench is much more vital for our products to assure that our products are bug free.

3.15.2 Zero delay simulation

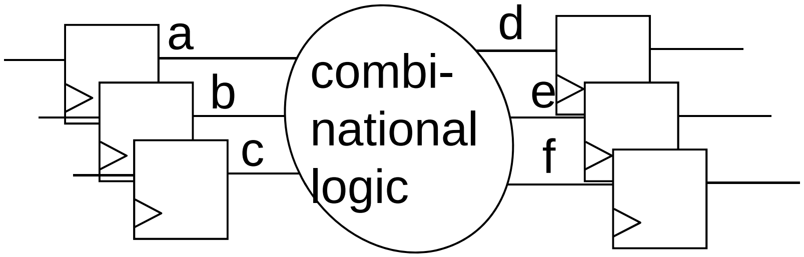


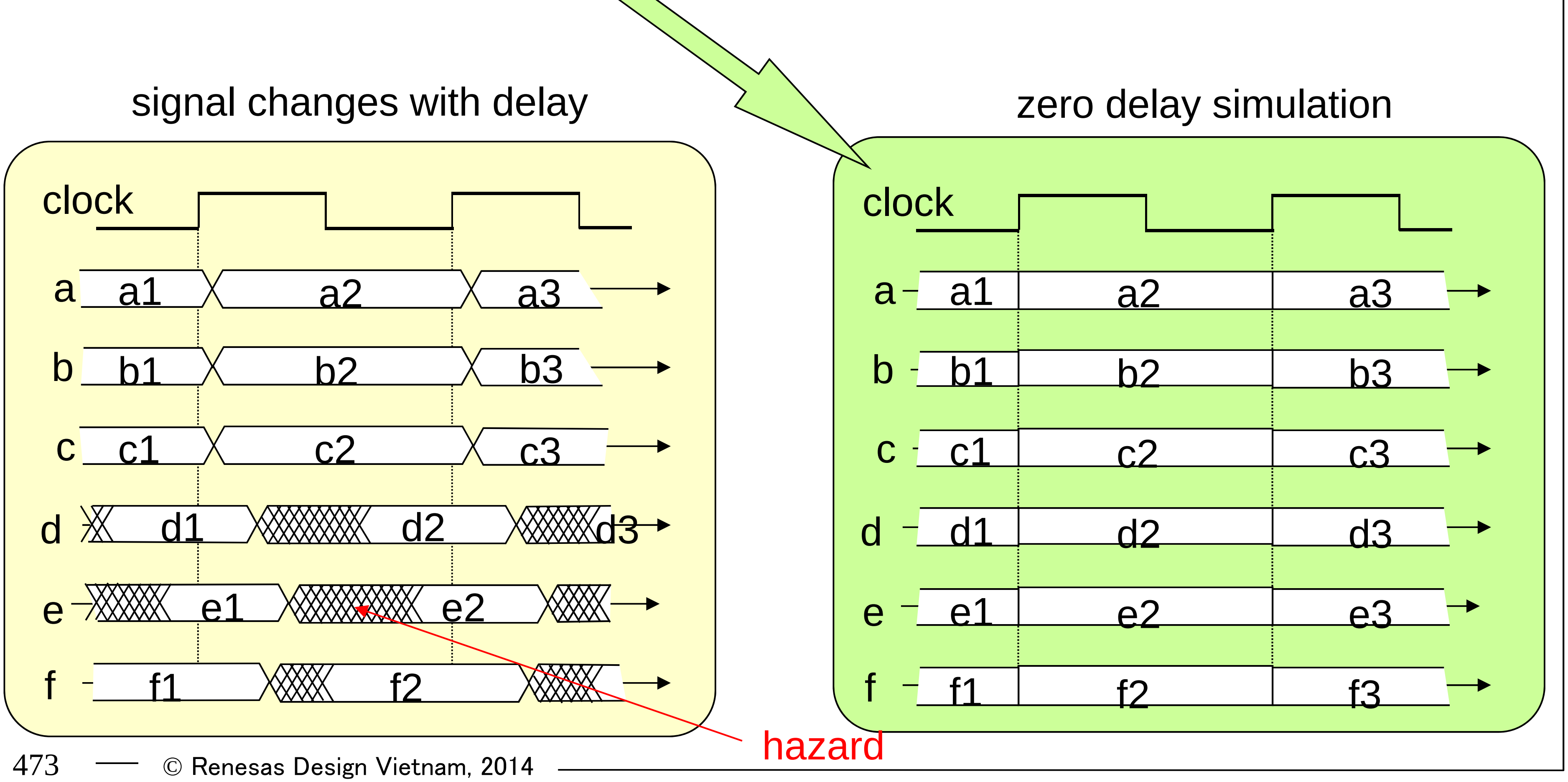
Whether signal d2, e2, and f2 become stable before setup time and keep their value during setup time + hold time are checked by STA, not by dynamic RTL simulation.



Zero delay simulation

Today, RTL simulation is done with no delay. It is called zero delay simulation.




signal changes with delay

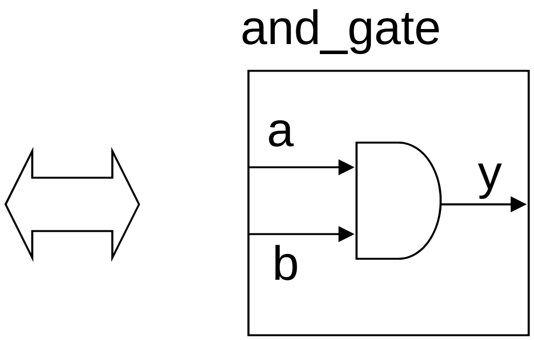
zero delay simulation

3.15.3 Test bench examples

(1) Test bench for simple AND logic.

Now try to create a test bench to test the module shown below.

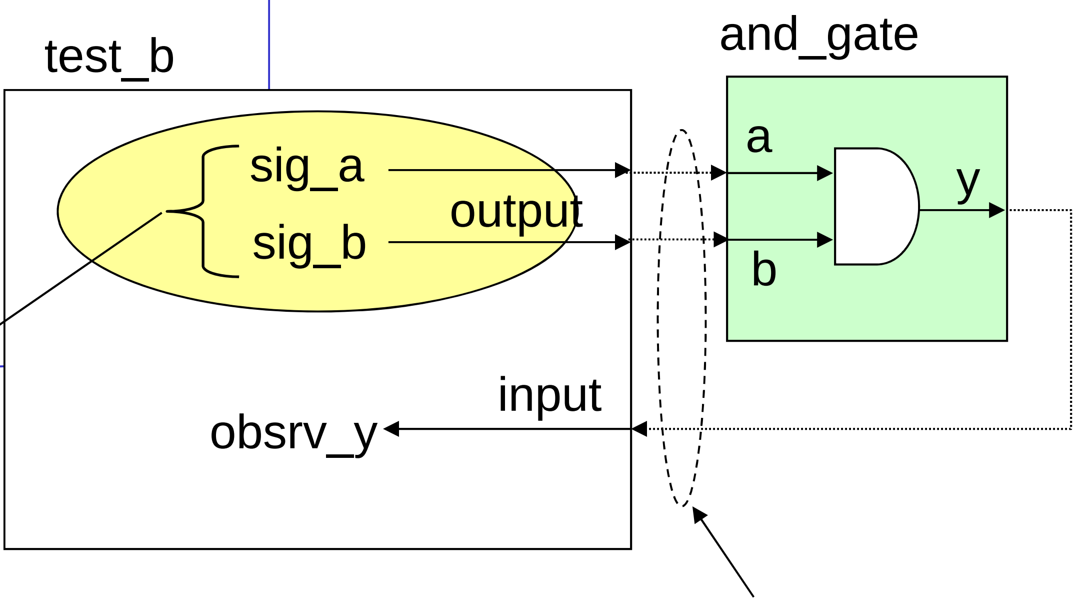
```
module and_gate ( a, b, y ) ;
input a, b ;
output y ;
wire a, b ;
reg y ; // non-FF
  always @ ( a or b ) begin
        y = a & b ;
  end
endmodule
```



To supply input signals to module and_gate, we need another module which can create two signals given to the module and_gate. Name this module “test_b”.

It must have two output ports to feed data to and_gate, and one input port to receive data from and_gate as shown on the next page.

```
module test_b ( obsrv_y, sig_a, sig_b ) ;
input obsrv_y ;
output sig_a, sig_b ;
        •
        •
        •
        •
        •
        •
        •
        •

endmodule
```



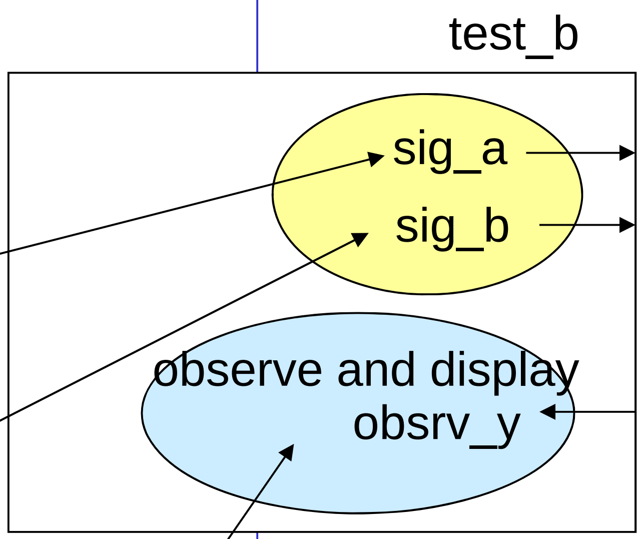
These signals can be created by using initial statement as below.

These connections are not defined yet.

```
initial begin
          sig_a = 0 ;
  # 10 sig_a = 1 ;
          •
          •
          •
          •

end
```

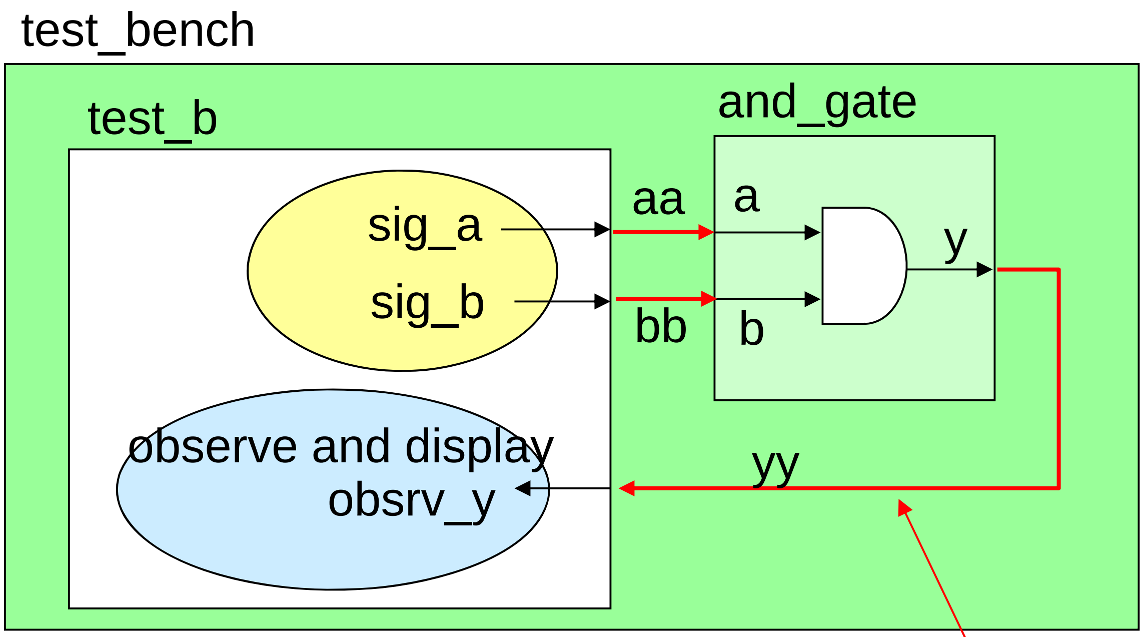
A variable sig_a appears on the left hand side of the procedural assignment. Therefore, it must be declared as register type variable.

```
module test_b ( obsrv_y, sig_a, sig_b ) ;
input obsrv_y ;
output sig_a, sig_b ;
wire obsrv_y ;
reg sig_a, sig_b ; // non-FF

initial begin
          sig_a = 0 ;
  # 10 sig_a = 1 ;
  # 10 $finish ;
end
initial begin
          sig_b = 1 ;
  # 5   sig_b = 0 ;
  # 10 sig_b = 1 ;
end

initial $monitor("a=%b, b=%b, y=%b",
                          sig_a, sig_b, obsrv_y ) ;
endmodule
```



test_b module is now completed, but the signal connection to and_gate is not defined yet.

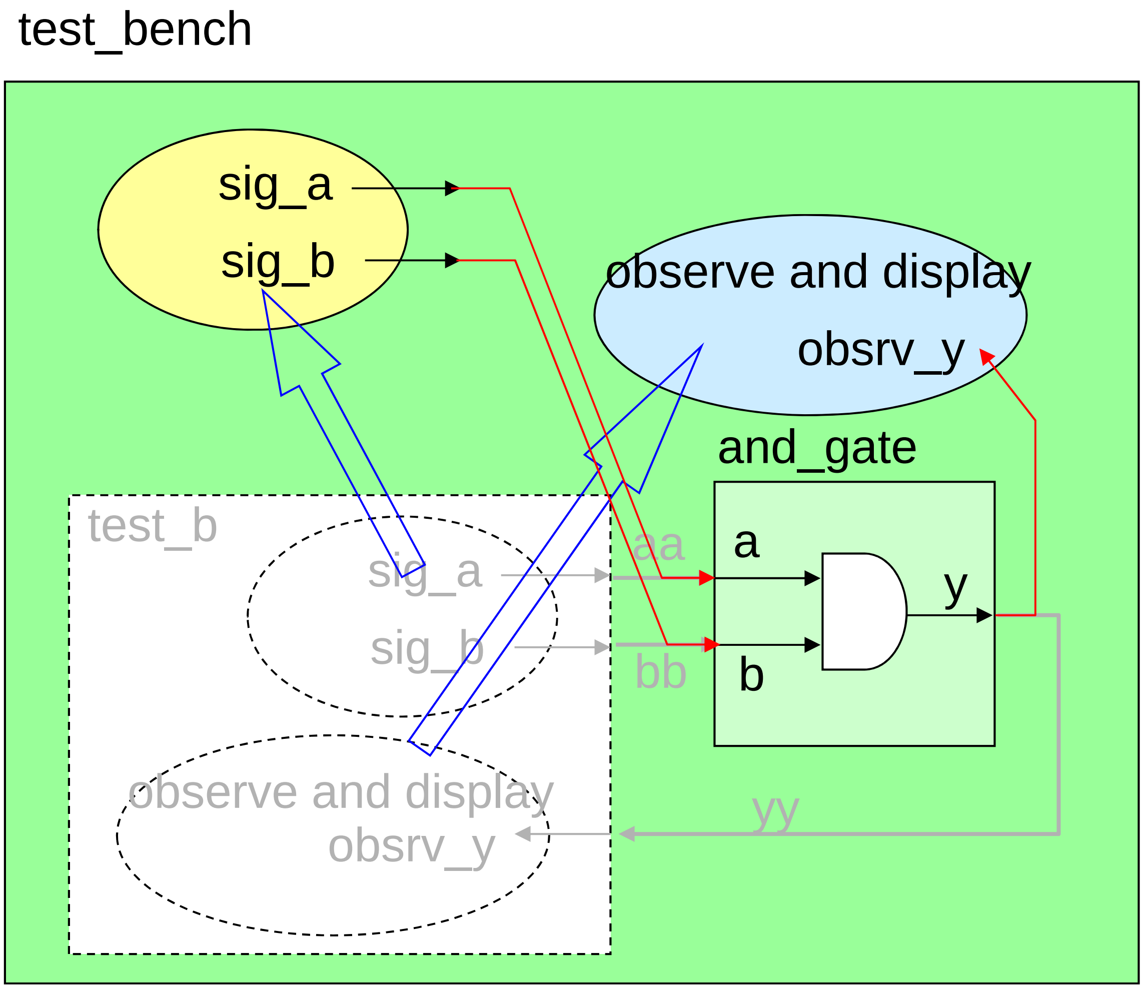
To define the connection between two modules, test_b and and_gate, we need an upper layer module such as test_bench as shown below.



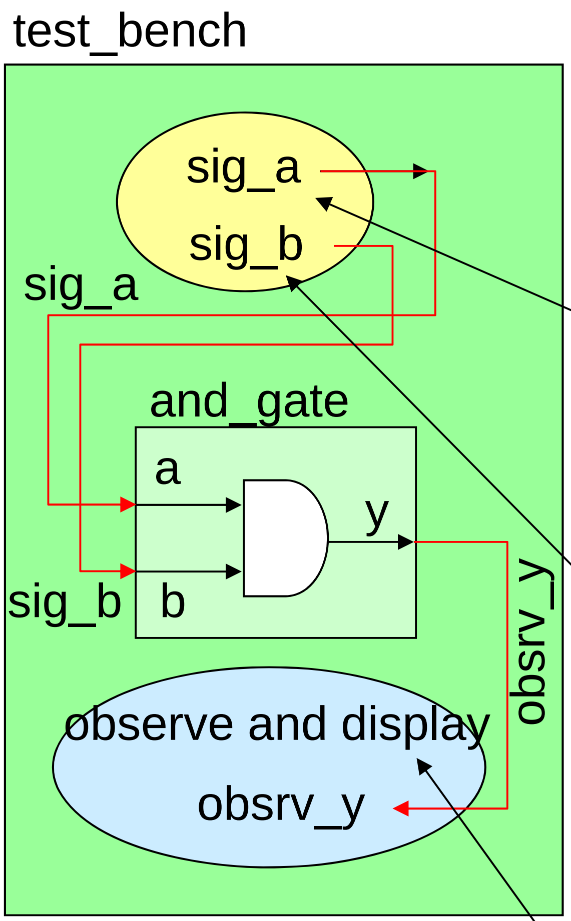
A module test_bench does not have to have an input port or output port because no input needed, and no output needed.

```
module test_bench ;    ←    No input, no output.
wire aa, bb, yy ;

test_b test_b_01 ( .sig_a(aa), .sig_b(bb), .obsrv_y(yy) ) ;
and_gate and_gate_01 ( .a(aa), .b(bb), .y(yy) ) ;

endmodule
```

In the previous example, we used three modules, including the target module, for testing.
However, by moving the input signal creating code out of test_b module into test_bench module, we do not need test_b module any more.



The test_bench on the the left has sig_a and sig_b creation logic, observe obsrv_y logic, and wire signal connection to and_gate in itself, not in different modules.

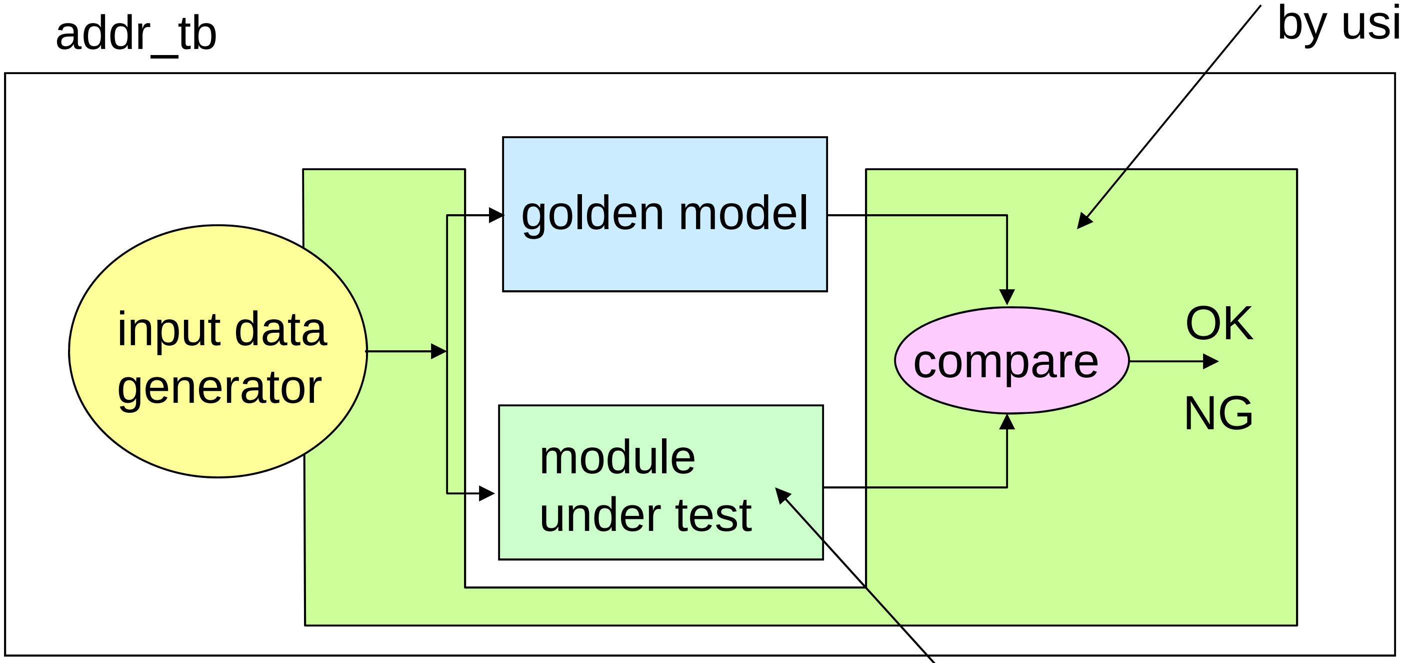


```
module test_bench ;
reg sig_a, sig_b ; // non-FF
wire obsrv_y ;
initial begin
          sig_a = 0 ;
  # 10 sig_a = 1 ;
  # 10 $finish ;
end
initial begin
          sig_b = 1 ;
  # 5   sig_b = 0 ;
  # 10 sig_b = 1 ;
end

and_gate and_gate_01 (
                      .a(sig_a), .b(sig_b), .y(obsrv_y)
                       ) ;

initial $monitor("a=%b, b=%b, y=%b",
                          sig_a, sig_b, obsrv_y ) ;
endmodule
```

This shows the connection of wires inside the test_bench module.

(3) Test bench using golden model

Let's create a test bench to test adder logic by using task. Logic structure of the test bench should look like below. We need an input data generator and a golden model to get the expected result for a certain test input, and compare logic of the outputs.

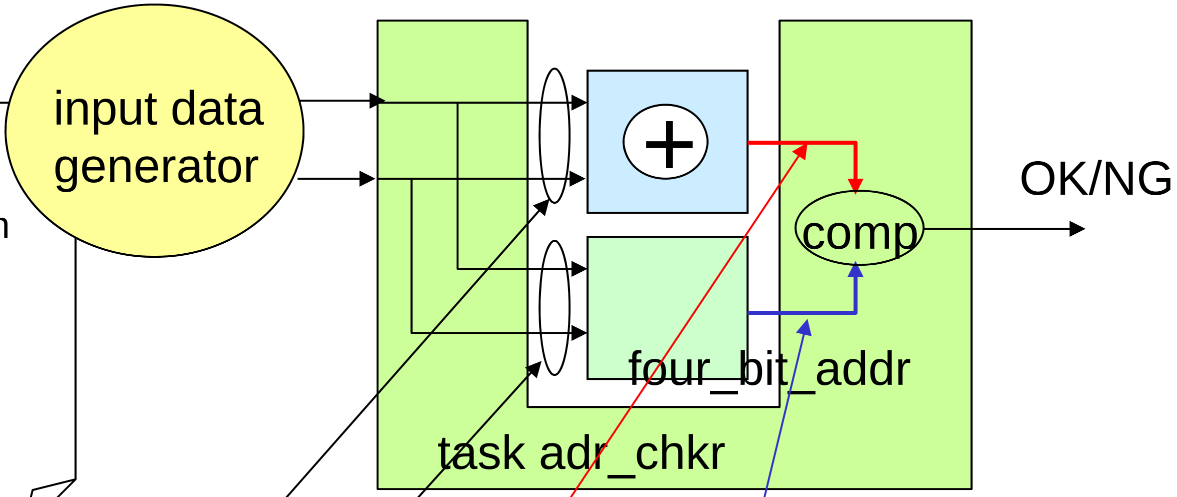


Create this part by using task.

The four_bit_addr module consists of four full adders.

If we can rely on Verilog add operator installed in a simulator, we can us Verilog add operator as a golden model.

```
for ( i=0 ; i=<15; i = i+1 ) begin
   for ( j=0; j=<15; j = j+1 ) begin
      enable task with
      input arguments i and j
   end
end
$finish
```



This logic can create all the possible combination of inputs.

```
task adr_chkr ;
parameter DLY = 1 ;
input [3:0] a1, a2 ;
begin
  g_mdl_a1 = a1 ;
  g_mdl_a2 = a2 ;
  dut_a1 = a1 ;
  dut_a2 = a2 ;
  #DLY ;
  if ( {g_mdl_co, g_mdl_b} != {dut_co, dut_b} ) begin
     $display ( “error a1=%h, a2=%h, golded=%b, %h, dut=%b, %h”,
                   a1,  a2, g_mdl_co, g_mdl_b, dut_co, dut_b  ) ;
     $finish ;
  end
end
```

Finish simulation when error detected.

We get the following code as a test bench.

```
module addr_tb ;
parameter DLY = 1 ;
integer i, j ;
wire [3:0] a1 = i ;
wire [3:0] a2 = j ;
reg [3:0] g_mdl_a1, g_mdl_a2 ; // non-FF
wire [3:0] g_mdl_b ;
reg [3:0] dut_a1, dut_a2 ; // non-FF
wire [3:0] dut_b ;
wire g_mdl_co ;
wire dut_co ;

//
initial begin                           // input data creation and check result
  for ( i=0 ; i <= 15; i = i+1 ) begin
     for ( j=0; j <= 15; j = j+1 ) begin
          adr_chkr( a1, a2 ) ;
     end
   end
$display("test OK") ;
$finish ;
end

// module connection
// golden model
g_mdl_adr g_mdl_adr_01( .a1(g_mdl_a1), .a2(g_mdl_a2), .b(g_mdl_b), .co(g_mdl_co) ) ;
//  module under test
four_bit_addr four_bit_addr_01( .a1(dut_a1), .a2(dut_a2), .b(dut_b), .co(dut_co) ) ;
```

This logic can create all the possible patterns for two inputs supplied to adder module.

checked on cver

```
// task          input to modules and check result . If bug, stop simulation
task adr_chkr ;
input [3:0] a1, a2 ;
begin
  g_mdl_a1 = a1 ;
  g_mdl_a2 = a2 ;
  dut_a1 = a1 ;
  dut_a2 = a2 ;
  #DLY ;
  if ( { g_mdl_co, g_mdl_b } != { dut_co, dut_b } ) begin
      $display ("error a1=%d, a2=%d, golden=%b, %d, dut=%b, %d",
                        a1,  a2, g_mdl_co, g_mdl_b, dut_co, dut_b  ) ;
      $finish ;
  end
end
endtask
```

task definition

```
endmodule
```

```
// golden model
module g_mdl_adr ( a1, a2, b, co ) ;
input [3:0] a1, a2 ;
output [3:0] b ;
output co ;
wire [3:0] a1, a2 ;
wire [3:0] b ;
wire co ;
   assign { co, b } = { 1'b0, a1 }  +  { 1'b0, a2 } ;
endmodule
```

This is a golden model
of 4-bit adder.

```
// module under test: four bit adder using four full adders
module four_bit_addr ( a1, a2, b, co ) ;
input [3:0] a1, a2 ;
output [3:0] b ;
output co ;
wire [3:0] a1, a2 ;
wire [3:0] b ;
wire co ;
wire c1, c2, c3  ;
f_addr f_addr_01( .aa(a1[0]), .bb(a2[0]), .dd(b[0]), .c_in(1'b0),.c_out(c1) ) ;
f_addr f_addr_02 ( .aa(a1[1]), .bb(a2[1]), .dd(b[1]), .c_in(c1),.c_out(c2) ) ;
f_addr f_addr_03 ( .aa(a1[2]), .bb(a2[2]), .dd(b[2]), .c_in(c2),.c_out(c3) ) ;
f_addr f_addr_04 ( .aa(a1[3]), .bb(a2[3]), .dd(b[3]), .c_in(c3),.c_out(co) ) ;
endmodule

//module under test: full addr
module f_addr ( c_in, aa, bb, dd, c_out ) ;
input  c_in, aa, bb ;
output dd, c_out ;
wire c_in, aa, bb ;
wire dd, c_out ;
assign dd = aa ^ ( bb ^ c_in )  ;
assign c_out =  (aa & bb)  | (bb & c_in) | ( c_in & aa ) ;
endmodule
```

The module under test

Q3.15-1 : Run the test bench in the previous page with the bug shown below. Check if the test bench can find the bug or not.

```
// module under test: four bit adder using full addr
module four_bit_addr ( a1, a2, b, co ) ;
input [3:0] a1, a2 ;
output [3:0] b ;
output co ;
wire [3:0] a1, a2 ;
wire [3:0] b ;
wire co ;
wire c1, c2, c3  ;
f_addr f_addr_01( .aa(a1[0]), .bb(a2[0]), .dd(b[0]), .c_in(1'b0),.c_out(c1) ) ;
f_addr f_addr_02 ( .aa(a1[1]), .bb(a2[1]), .dd(b[1]), .c_in(c1),.c_out(c2) ) ;
f_addr f_addr_03 ( .aa(a1[3]), .bb(a2[2]), .dd(b[2]), .c_in(c2),.c_out(c3) ) ;
f_addr f_addr_04 ( .aa(a1[3]), .bb(a2[3]), .dd(b[3]), .c_in(c3),.c_out(co) ) ;
endmodule

//module under test: full addr
module f_addr ( c_in, aa, bb, dd, c_out ) ;
input  c_in, aa, bb ;
output dd, c_out ;
wire c_in, aa, bb ;
wire dd, c_out ;
assign dd = aa ^ ( bb ^ c_in )  ;
assign c_out =  (aa & bb)  | (bb & c_in) | ( c_in & aa ) ;
endmodule
```

Bug

Q3.15-1 : A sample answer.

The test bench will show an error message as below.

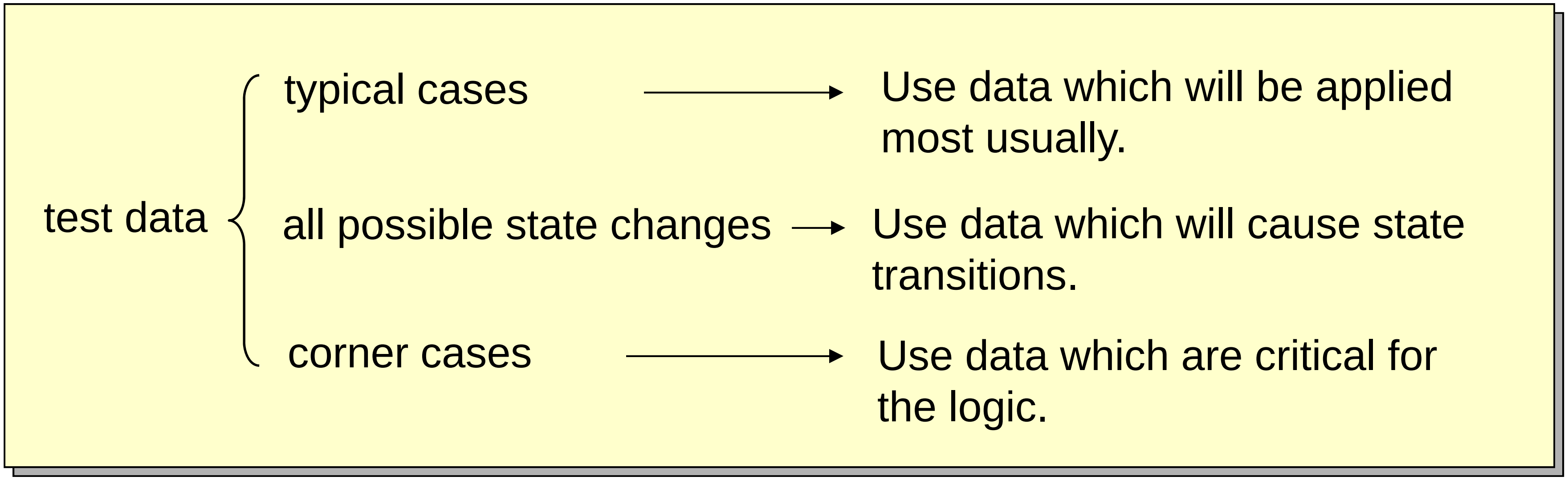
```
error a1= 4, a2= 0, golden=0,  4, dut=0,  0
```

3.15.4 Test data

While programming RTL code, it is important to assume various data as input and make your code prepared for such data. Your code will not work properly for data which you did not expect to come. This means your imaginative power decides the quality of your program.

It is very important for an engineer to be able to select or determine proper data to test a module he/she designed.

Test data shall be selected so that all the possible paths of your code are covered.



test data

typical cases → Use data which will be applied most usually.

all possible state changes → Use data which will cause state transitions.

corner cases → Use data which are critical for the logic.

Do not think
“my code is correct, because
the simulation result is OK.”

An example

u_eor is a module which executes unary eor operation on 8-bit input in_a and places the result ^in_a to its 1-bit output out_y.
If we get the following result by running the test bench on the right, can we say that the target code is OK??

```
                                         number of on-bit
in_a=00000000, out_y=0    ←――――   0 bit
in_a=00000001, out_y=1    ←――――   1 bit
in_a=10001000, out_y=0    ←――――   2 bits
in_a=01010100, out_y=1    ←――――   3 bits
in_a=11001001, out_y=0    ←――――   4 bits
in_a=01010111, out_y=1    ←――――   5 bits
in_a=11101011, out_y=0    ←――――   6 bits
in_a=11111101, out_y=1    ←――――   7 bits
in_a=11111111, out_y=0    ←――――   8 bits
Halted at location **test_u_eor
```

```
module test_u_eor ;
reg [7:0] aa ;
wire yy ;
u_eor u_eor_01(.in_a(aa), .out_y(yy));
initial begin
    aa=8'b0000_0000 ; // bit count=0
#5 aa=8'b0000_0001 ; // bit count=1
#5 aa=8'b1000_1000 ; // bit count=2
#5 aa=8'b0101_0100 ; // bit count=3
#5 aa=8'b1100_1001 ; // bit count=4
#5 aa=8'b0101_0111 ; // bit count=5
#5 aa=8'b1110_1011 ; // bit count=6
#5 aa=8'b1111_1101 ; // bit count=7
#5 aa=8'b1111_1111 ; // bit count=8
#5 $finish ;
end
initial begin
 $monitor("in_a=%b, out_y=%b",
 aa, yy );
end
endmodule
`include "u_eor.v"
```

file name: test_u_eor.v

Use your imaginative power what are the hidden part of the u_eor module.

```
module u_eor( in_a, out_y ) ;
input [7:0] in_a ;
output out_y ;
wire [7:0] in_a ;
reg out_y ;
//
integer k, cnt ;
always @ ( in_a ) beg
 cnt=0;
 for (k=0; k<=3; k=k+1)
  cnt=cnt+in_a[k]+in_a[k
 end
 out_y=( cnt[0] )? 1 : 0 ;
end
endmodule
```

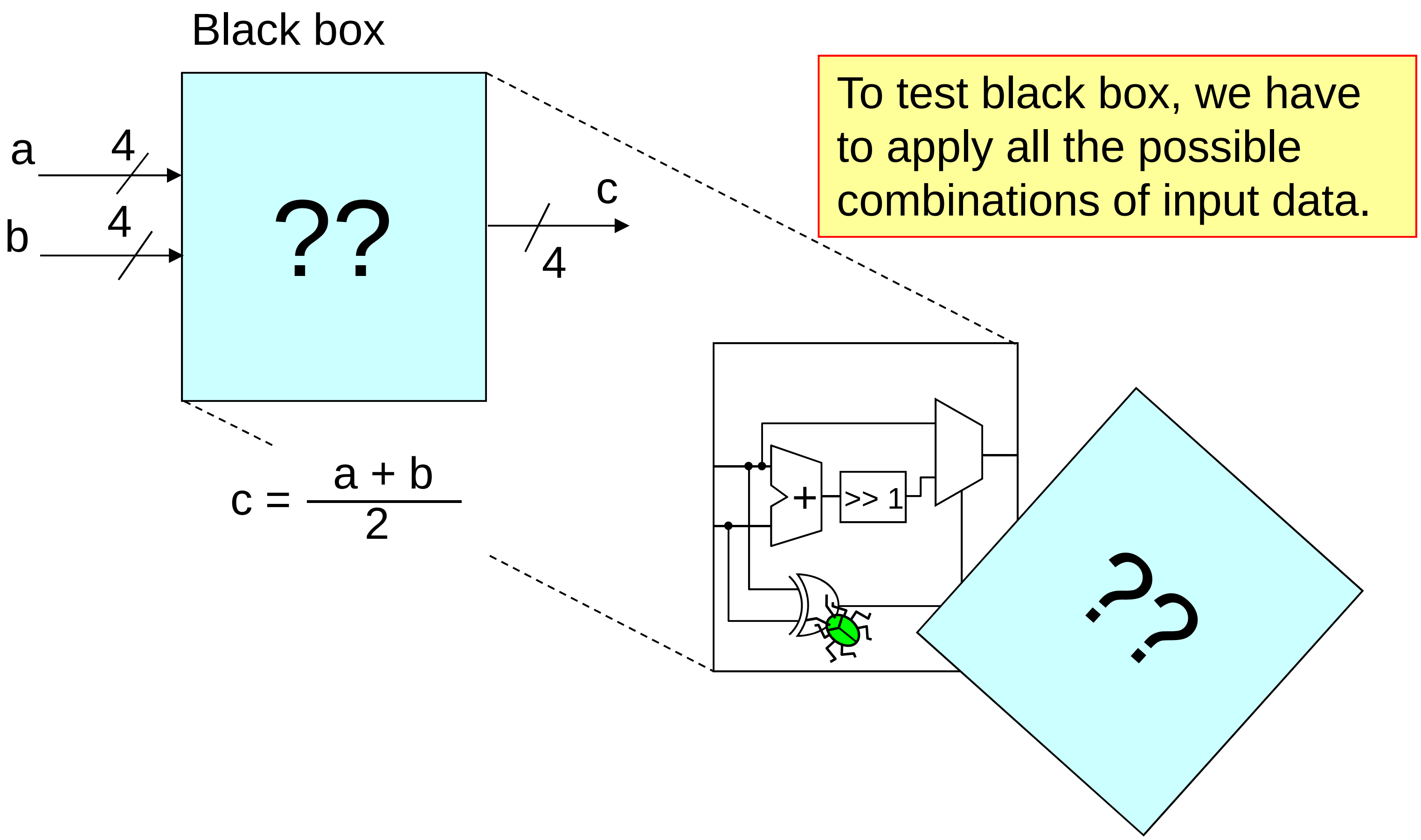
file name: u_eor.v

There are many many many chances for a wrong program passes poor test programs.

white box test vs. black box test

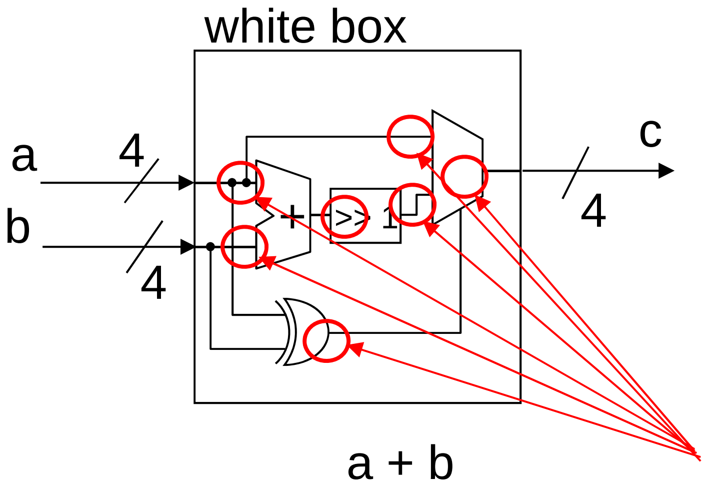
Black box test : Test **without** knowledge of internal structure and logic.

White box test : Test **with** knowledge of internal structure and logic.



$$c = \frac{a + b}{2}$$

To test black box, we have to apply all the possible combinations of input data.



$$c = \frac{a + b}{2}$$

To test white box, we do not have to apply all the possible combinations of input data. We can apply selective input data to check specific part of the design.

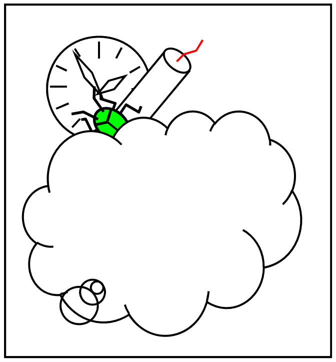
If the target is white box, we can create selective input data to activate specific part of the logic.

And once checked with some data, it may not have to be checked by using all the possible values. Typical and corner case data may be enough to be applied.

Q3.15-2 : In case of black box test, what kind of bugs can not be find out even if we apply all the possible combinations of input data?

Q3.15-2 : A sample answer.

Time bomb type bugs can not be found even if we apply all the possible patterns of input data.



Typical time bomb

No clean up after some operation.

⇨ Resource used up after long period of time, buffer over flow, value exceeding bit size, go beyond some boundary,,, etc.

If the target logic is sequential logic, then its behavior depends on its operating history. In such case, sometimes it is almost impossible to apply all the possible input patterns including input sequence.

How to select corner cases.

What data can check corner cases depend on the logic to handle them. However, we have to have a good sensitivity to tell what kind of data may be critical for various logic.

```
module u_eor( in_a, out_y ) ;
input [7:0] in_a ;
output out_y ;
wire [7:0] in_a ;
reg out_y ;
//
integer k, cnt ;
always @ ( in_a ) beg
 cnt=0;
 for (k=0; k<=3; k=k+1)
  cnt=cnt+in_a[k]+in_a[k
 end
 out_y=( cnt[0] )? 1 : 0 ;
end
endmodule
```

file name: u_eor.v

With the code on the left, bit 0 and bit3 can be a candidate for the corner cases. Actually all the following test data can not detect the bug, but a simple 8'b0000_1001 can detect the bug.

```
in_a=00000000, out_y=0
in_a=00000001, out_y=1
in_a=10001000, out_y=0
in_a=01010100, out_y=1
in_a=11001001, out_y=0
in_a=01010111, out_y=1
in_a=11101011, out_y=0
in_a=11111101, out_y=1
in_a=11111111, out_y=0
```

Use your imaginative power, there is no limitation of absurdity of bugs. Bugs always try to go beyond your imaginative power.

Q3.15-3 : What may be the corner cases for the following modules that execute unary exclusive OR operation.

```
module u_eor( in_a, out_y ) ;
input [7:0] in_a ;
output out_y ;
wire [7:0] in_a ;
reg out_y ;
//
integer k, cnt ;
always @ ( in_a ) begin
 cnt=0;
 for (k=0; k<=7; k=k+1) begin
  cnt=cnt+in_a[k] ;
 end
 out_y=( cnt[0] )? 1 : 0 ;
end
endmodule
```

```
module u_eor( in_a, out_y ) ;
input [3:0] in_a ;
output out_y ;
wire [3:0] in_a ;
reg out_y ;
//
integer k, cnt ;
always @ ( in_a ) begin
 case (in_a) begin
  4'b0000,4'b0011,4'b0101,4'b1001,
  4'b0110,4'b1010,4'b1100,4'b1111:
   begin  out_y = 0 ; end
  4'b0001,4'b0010,4'b0100,4'b1000,
  4'b0111,4'b1011,4'b1101,4'b1110 :
   begin  out_y = 1 ; end
 endcase
end
endmodule
```

Q3.15-3 : A sample answer

```
module u_eor( in_a, out_y ) ;
input [7:0] in_a ;
output out_y ;
wire [7:0] in_a ;
reg out_y ;
//
integer k, cnt ;
always @ ( in_a ) begin
 cnt=0;
 for (k=0; k<=7; k=k+1) begin
  cnt=cnt+in_a[k] ;
 end
 out_y=( cnt[0] )? 1 : 0 ;
end
endmodule
```

test data related to bit 0 and 7 can be a corner case.

```
module u_eor( in_a, out_y ) ;
input [3:0] in_a ;
output out_y ;
wire [3:0] in_a ;
reg out_y ;
//
integer k, cnt ;
always @ ( in_a ) begin
 case (in_a) begin
  4'b0000,4'b0011,4'b0101,4'b1001,
  4'b0110,4'b1010,4'b1100,4'b1111:
    begin  out_y = 0 ; end
  4'b0001,4'b0010,4'b0100,4'b1000,
  4'b0111,4'b1011,4'b1101,4'b1110 :
    begin out_y = 1 ; end
 endcase
end
endmodule
```

All possible patterns can be a corner case.

3.15.5 Procedural continuous assignment

The procedural continuous assignments are procedural statements that allow expressions to be driven continuously onto registers or nets.

procedural continuous assignment

form	description
assign deassign	*assign* shall override all procedural assignments to a register. *deassign* shall end a procedural continuous assignment. The value of the register shall remain the same until the register is assigned a new value.
force release	*force* shall override a procedural assignment that takes place on the variable until a *release* is executed on the variable. The value given to a register by *force* statement shall be maintained in the register until the next procedural assignment takes place, except in the case where a procedural continuous assignment is active on the register. A force procedural statement on a net overrides all drivers of the net until a release procedural assignment is executed on the net.

The LHS of assign/deassign must be a register reference or a concatenation of registers. A memory word (array reference), a bit-select, and a part-select of a register are not allowed.

The LHS of force/release is same as above. However a net data type is allowed.

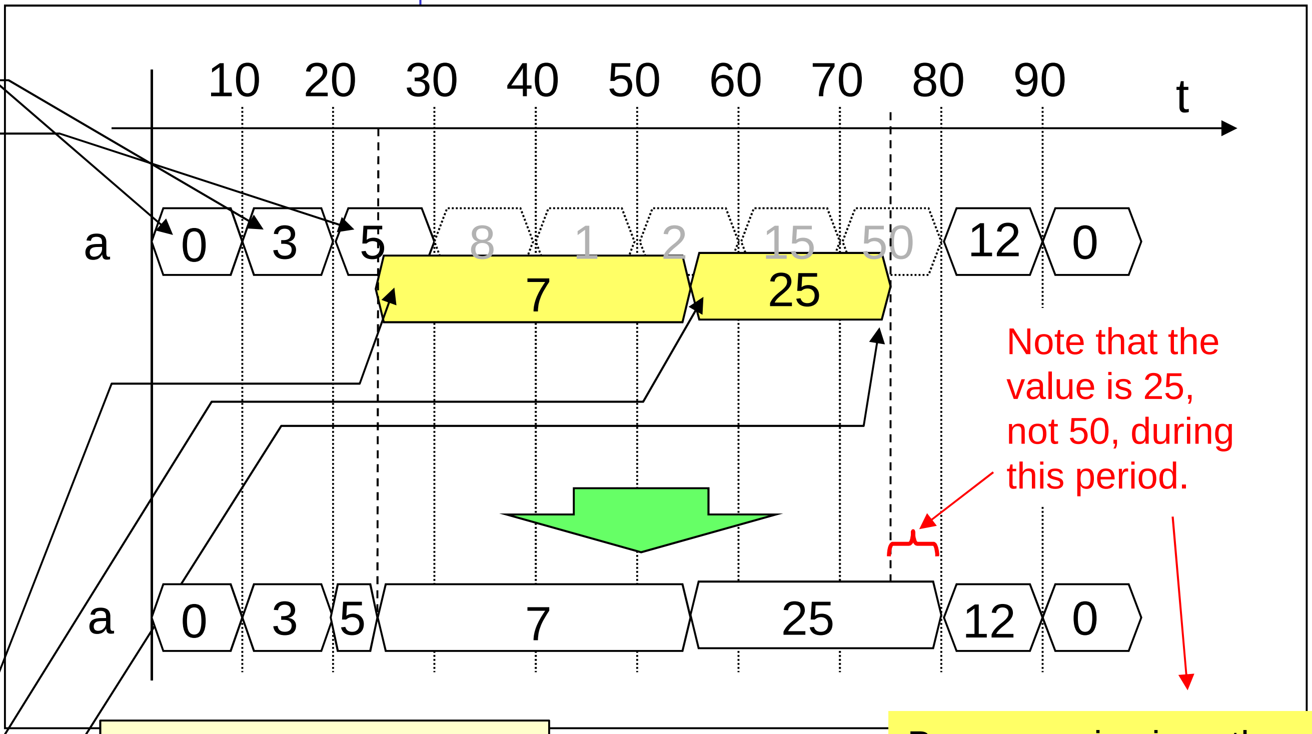
```
always @ ( clear or preset ) begin
  if ( clear ) begin
           assign q = 0 ;
  end
  else begin
     if ( preset ) begin
           assign q = 1 ;
     end
     else begin
          deassign q ;
     end
  end
end

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

q is kept 0 while clear=1,
and kept 1 while preset=1.

This code is just to show how assign/deassign work. Do not use this in actual job.

assign and deassign

```
reg [7:0] a ;

initial begin
a = 0 ;
#10 a = 3 ;
#10 a = 5 ;
#10 a = 8 ;
#10 a = 1 ;
#10 a = 2 ;
#10 a = 15 ;
#10 a = 50 ;
#10 a = 12 ;
#10 a = 0 ;
#10 $finish ;
end

initial begin
#25 assign a = 7 ;
#30 assign a = 25 ;
#20 deassign a ;
end
```



Note that the value is 25, not 50, during this period.

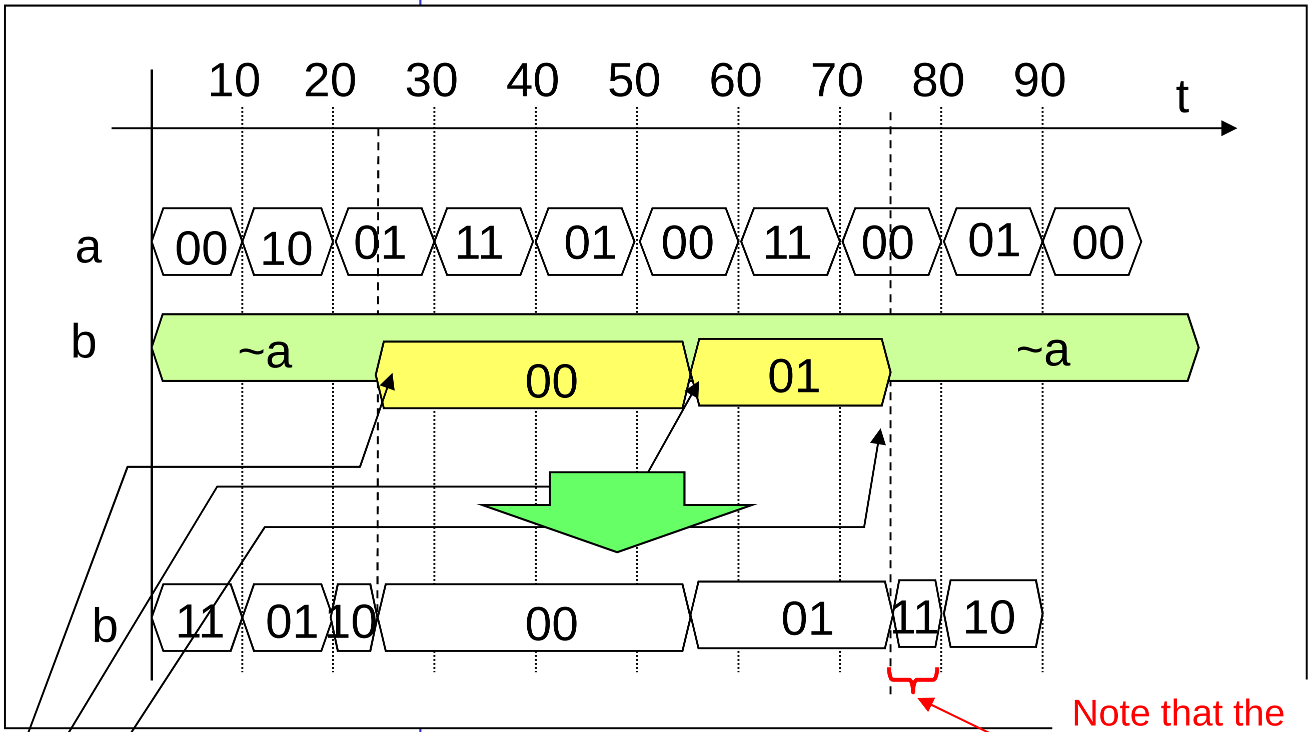
```
initial begin
#25 force a = 7 ;
#30 force a = 25 ;
#20 release a ;
end
```

force/release has the same effect.

Because a is given the values by procedural assign and no procedural assign is executed during this time period.

force and release on wire.

```
reg [1:0]  a ;
wire [1:0] b ;

assign b = ~a ;

initial begin
a = 2'b00 ;
#10 a = 2'b10 ;
#10 a = 2'b01 ;
#10 a = 2'b11 ;
#10 a = 2'b01 ;
#10 a = 2'b00 ;
#10 a = 2'b11 ;
#10 a = 2'b00 ;
#10 a = 2'b01 ;
#10 a = 2'b00 ;
#10 $finish ;
end

initial begin
#25 force b = 2'b00 ;
#30 force b = 2'b01 ;
#20 release b ;
end
```



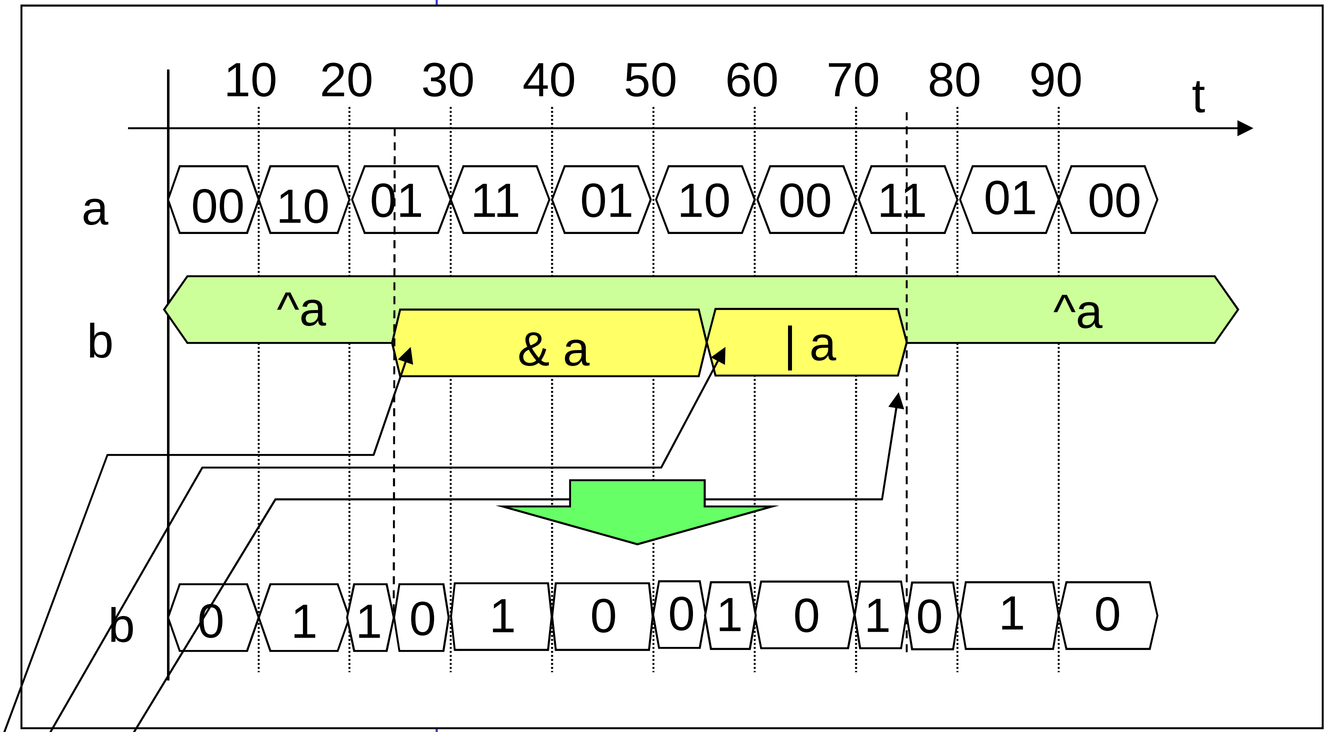
Note that the value is 11, not 01, during this period.

assign/deassign not allowed on net.

Because b is given the values by continuous assign, b changes its value whenever a changes.

force and release on wire.

```
reg [1:0] a ;
wire b ;

assign b = ^a ;

initial begin
a = 2'b00 ;
#10 a = 2'b10 ;
#10 a = 2'b01 ;
#10 a = 2'b11 ;
#10 a = 2'b01 ;
#10 a = 2'b10 ;
#10 a = 2'b00 ;
#10 a = 2'b11 ;
#10 a = 2'b01 ;
#10 a = 2'b00 ;
#10 $finish ;
end

initial begin
#25 force b = &a ;
#30 force b = |a ;
#20 release b ;
end
```



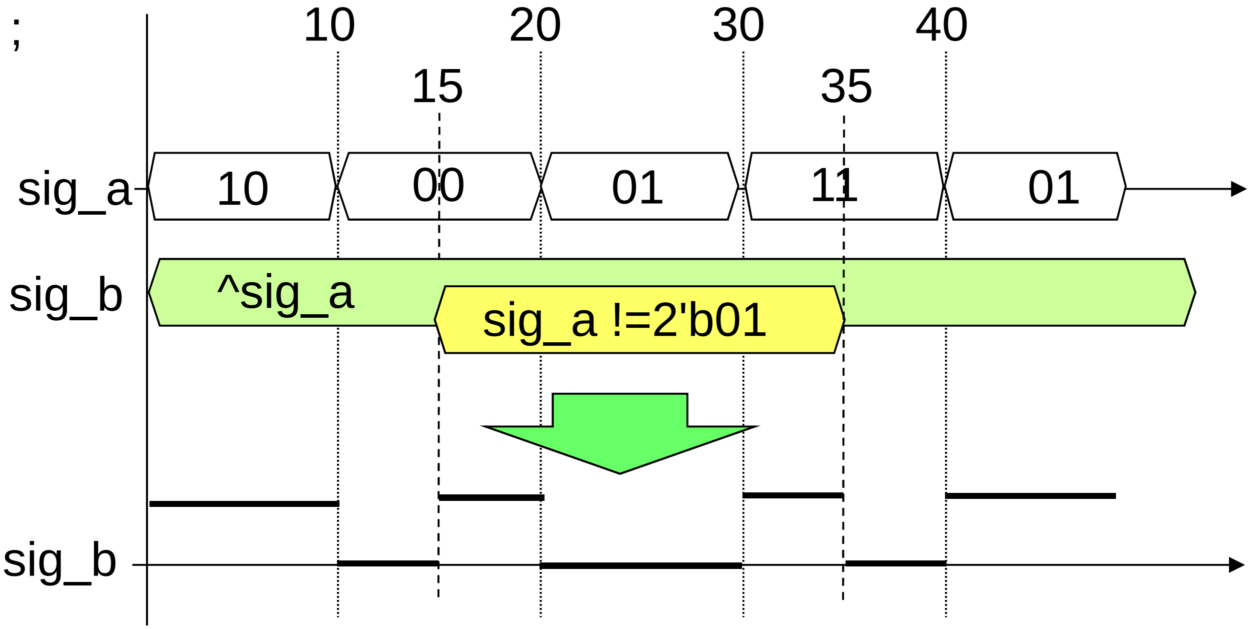
assign/deassign not allowed on net.

Q3.15-4 Draw a time chart of sig_b when the following piece of code is executed in RTL simulation.

```
reg [1:0] sig_a ;
wire sig_b ;

assign sig_b = ^sig_a ;

initial begin
sig_a = 2'b10 ;
#10 sig_a = 2'b00 ;
#10 sig_a = 2'b01 ;
#10 sig_a = 2'b11 ;
#10 sig_a = 2'b01 ;
end

initial begin
#15 force sig_b = (sig_a != 2'b01) ;
#20 release sig_b ;
end
```

Q3.15-4 A sample answer.

```
reg [1:0] sig_a ;
wire sig_b ;

assign sig_b = ^sig_a ;

initial begin
sig_a = 2'b10 ;
#10 sig_a = 2'b00 ;
#10 sig_a = 2'b01 ;
#10 sig_a = 2'b11 ;
#10 sig_a = 2'b01 ;
end

initial begin
#15 force sig_b =
(sig_a != 2'b01) ;
#20 release sig_b ;
end
```

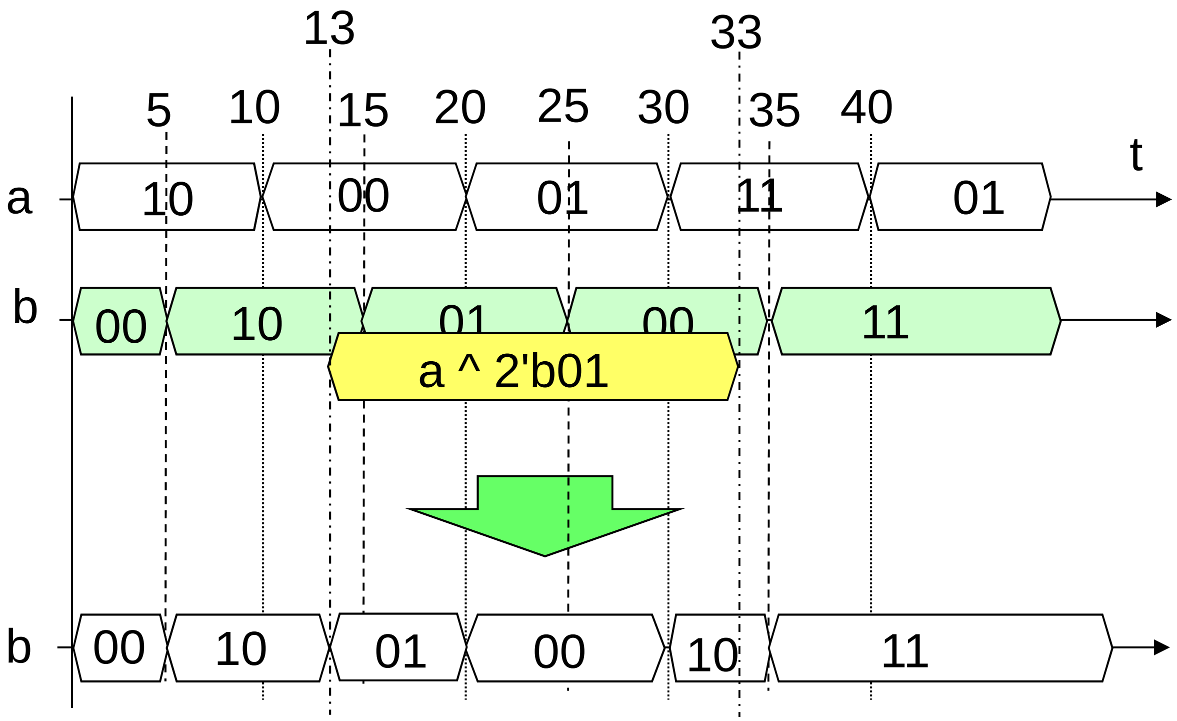


Q3.15-5 Draw a time chart of sig_b when the following piece of code is executed in RTL simulation.

```
reg [1:0] a ;
reg [1:0] b ;
initial begin
a = 2'b10 ;
#10 a = 2'b00 ;
#10 a = 2'b01 ;
#10 a = 2'b11 ;
#10 a = 2'b01 ;
end
initial begin
 b = 2'b00 ;
#5 b = a ;
#10 b = a ^ 2'b01 ;
#10 b = a & 2'b10 ;
#10 b = 2'b11 ;
end
initial begin
#13 assign b = a ^ 2'b01
;
#20 deassign b ;
end
```

Q3.15-5 A sample answer.

```
reg [1:0] a ;
reg [1:0] b ;
initial begin
a = 2'b10 ;
#10 a = 2'b00 ;
#10 a = 2'b01 ;
#10 a = 2'b11 ;
#10 a = 2'b01 ;
end
initial begin
 b = 2'b00 ;
#5 b = a ;
#10 b = a ^ 2'b01 ;
#10 b = a & 2'b10 ;
#10 b = 2'b11 ;
end
initial begin
#13 assign b = a ^ 2'b01 ;
#20 deassign b ;
end
```



How to use force release?

force/release is suitable to inject controlled values to specific variables.

```
initial begin
  #,,,,
  #,,,
  #,,,,
  #,,,  force my_mdl_01.panic_err = 1 ;
  #,,   release my_mdl_01.panic_err ;
  ,,
end
```

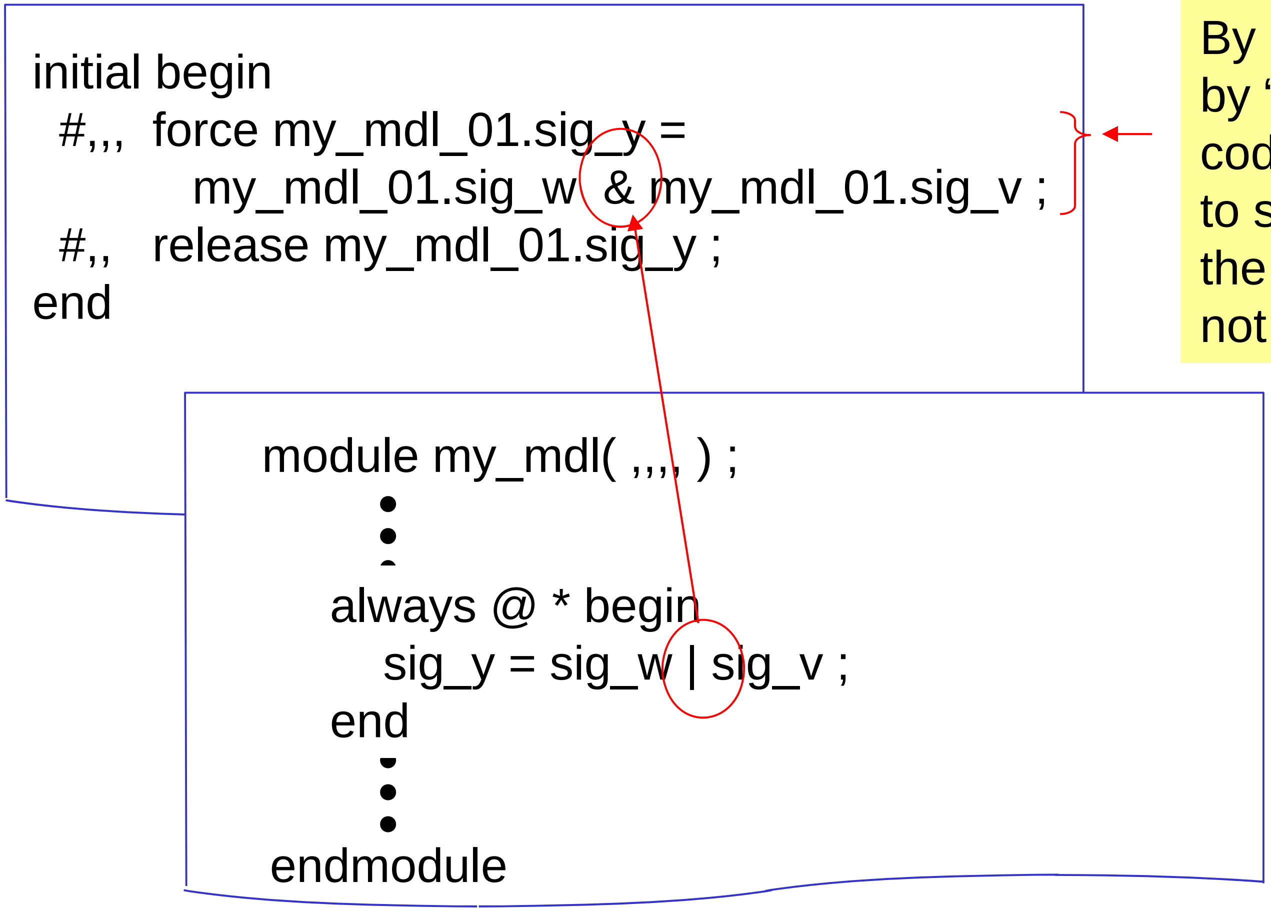
To inject error signal is a typical usage of force/release.

Overwrite panic_err by 1 to simulate error processing.

```
module my_mdl(,,,) ;
  ,,,,
  ,,,
   if ( strange_happen) panic_err = 1 ;
  ,,,,,
  ,,
  if (panic_error ) ,,,,,,,,
,,,,,,
endmodule
```

panic_err is designed to detect something unexpected happen. It is very difficult to simulate this part because pacnic_err will not become 1 in usual operation.

How to use force release?



```
initial begin
  #,,,  force my_mdl_01.sig_y =
            my_mdl_01.sig_w  & my_mdl_01.sig_v ;
  #,,   release my_mdl_01.sig_y ;
end
```

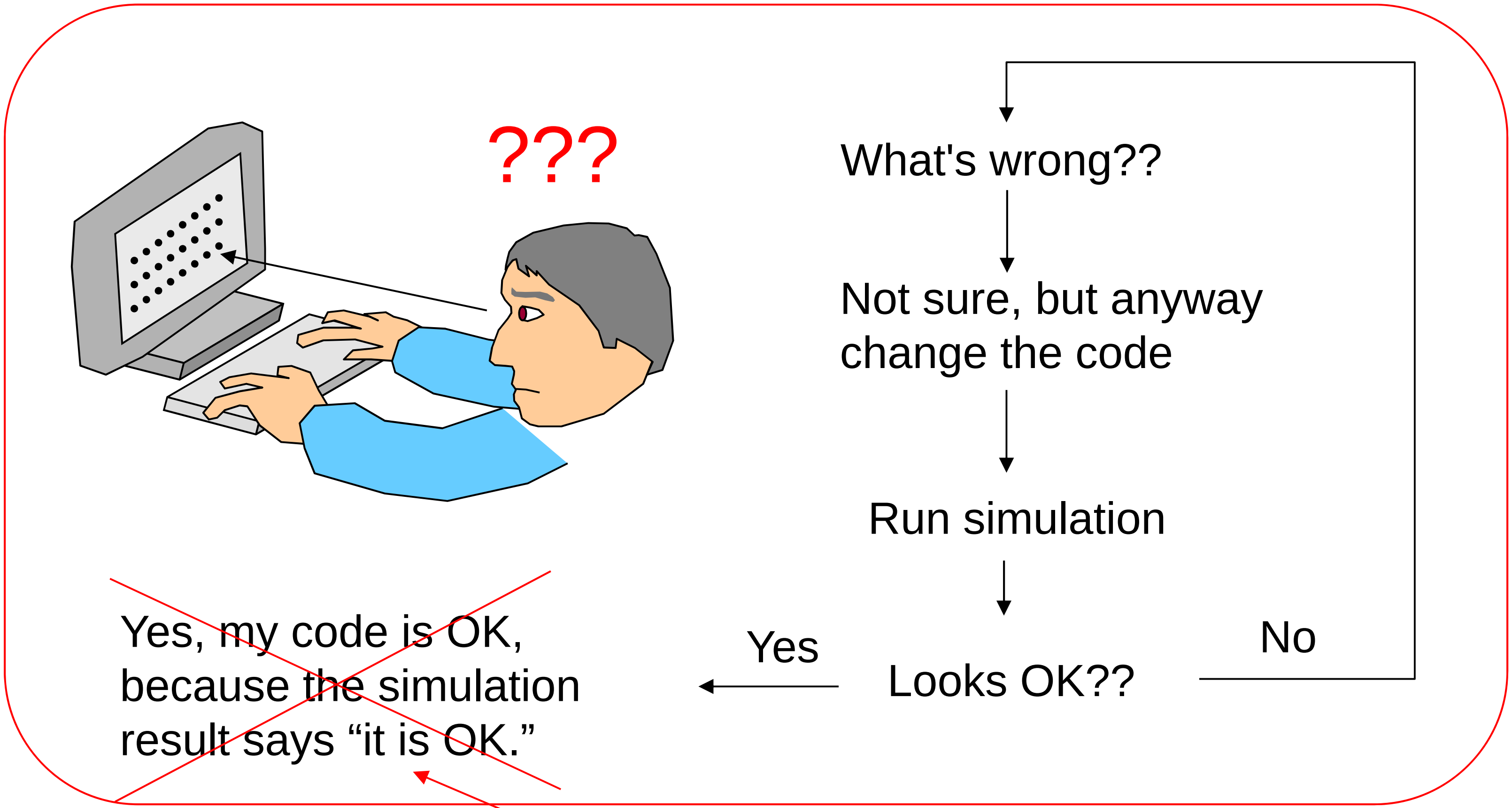
```
module my_mdl( ,,,, ) ;
      ●
      ●
      ●
    always @ * begin
        sig_y = sig_w | sig_v ;
    end
      ●
      ●
      ●
 endmodule
```

By overwriting the logic by “and” operation, this code can simulate logic to see what happens if the operation is “and”, not “or”.

3.15.6 How to execute debug

Many kind of bugs escape from RTL simulation test. This means that we can not be safe from bugs even if simulation test looks OK.

Therefore the following method is the most ineffective way to fix bugs. Never do this way.



What's wrong??

Not sure, but anyway change the code

Run simulation

Looks OK??

No

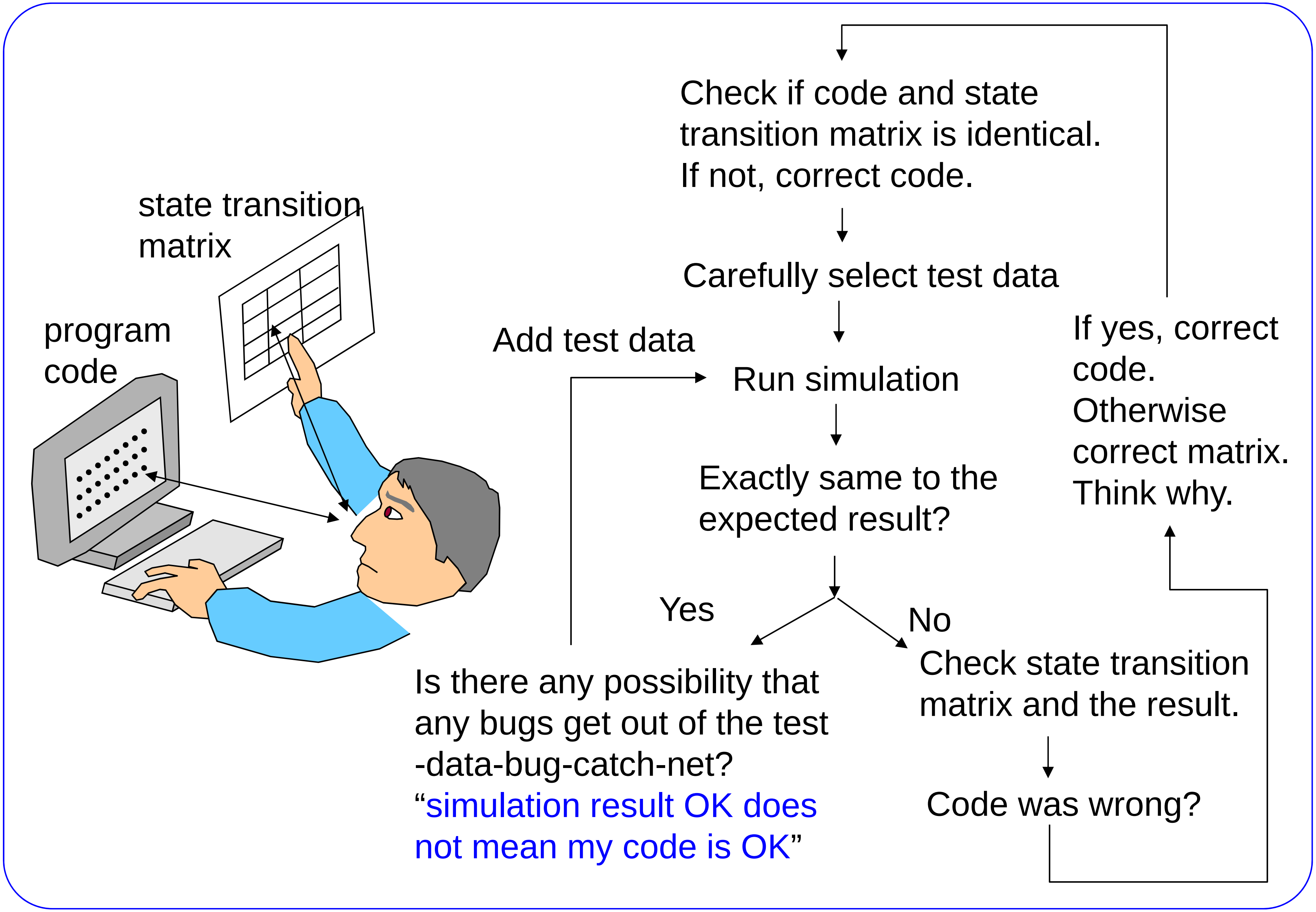
Yes

~~Yes, my code is OK, because the simulation result says “it is OK.”~~

This is not true.

Do not think
“my code is correct, because
the simulation result is OK.”

The recommended way is:



state transition matrix

program code

Check if code and state transition matrix is identical. If not, correct code.

Carefully select test data

Add test data

Run simulation

Exactly same to the expected result?

Yes

Is there any possibility that any bugs get out of the test -data-bug-catch-net? “simulation result OK does not mean my code is OK”

No

Check state transition matrix and the result.

Code was wrong?

If yes, correct code. Otherwise correct matrix. Think why.

4. Issues about unknown value x

4.1 Problems caused by unknown value x

unknown value x causes many problems. Typical problems are shown in the table below.

issues	description	comments
Handling of x is different in RTL and gate.	When x appears, RTL and gate level simulation results may be different.	This causes a big problem.
How to treat x is different among tools.	In simulation, x means unknown value. But for synthesis tool, x means “don’t care”.	RTL and gate level simulation results may be different.
There is no x in actual device. x has meaning only in simulation.	If there is x, simulation result and actual device behave differently.	If not initialized, initial value of FF or latch is x in simulation, 0 or 1 in actual devices.

(1) Handling of x is different in RTL and gate

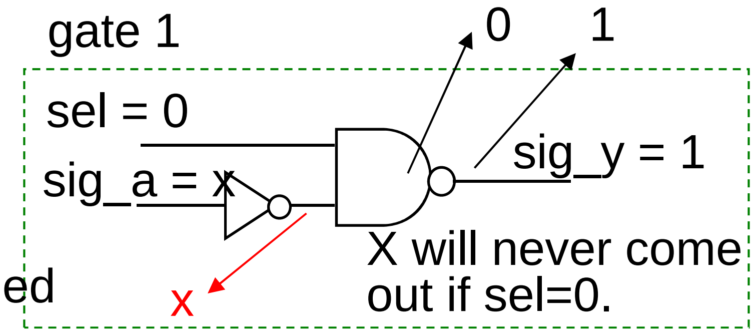
Example 1:

Depending on tools, netlists created from the same RTL code may output x or 1/0 value in gate level simulation.

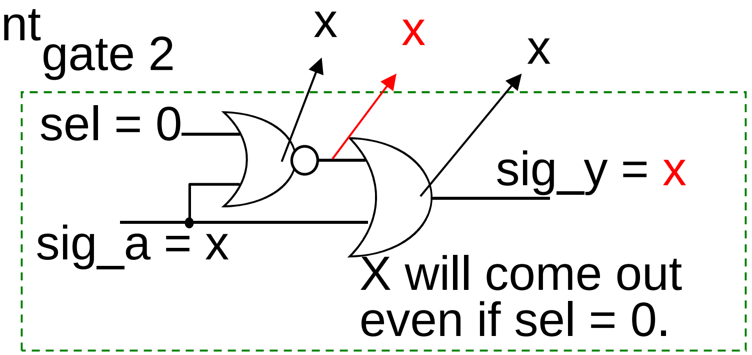
RTL (logic to suppress x propagating)

```
      :
assign sig_y = sel ? sig_a: 1'b1 ;
      :
```

synthesized result may be different

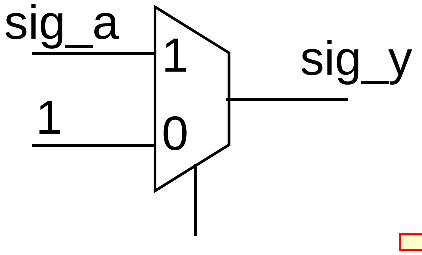


X will never come out if sel=0.



X will come out even if sel = 0.

If sel is 0, 1 always appear on sig_y even when sig_a is x.



RTL simulation and gate level simulation may have different results.

Example 2:

A netlist created from RTL code may output x in gate level simulation while the RTL code will not output x in RTL simulation.

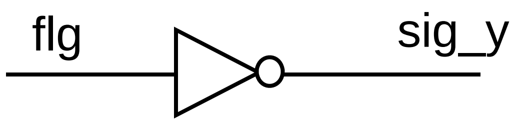
RTL

```
        :
if ( flg ) begin
    sig_y = 1'b0 ;
end
else begin
    sig_y = 1'b1 ;
end
        :
```

sig_y is 1
if flg is x.

synthesis

gate



sig_y is x
if flg is x.

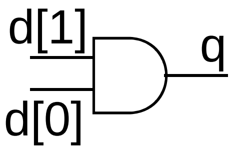
Results of RTL and gate level simulation are different.

Example 3:

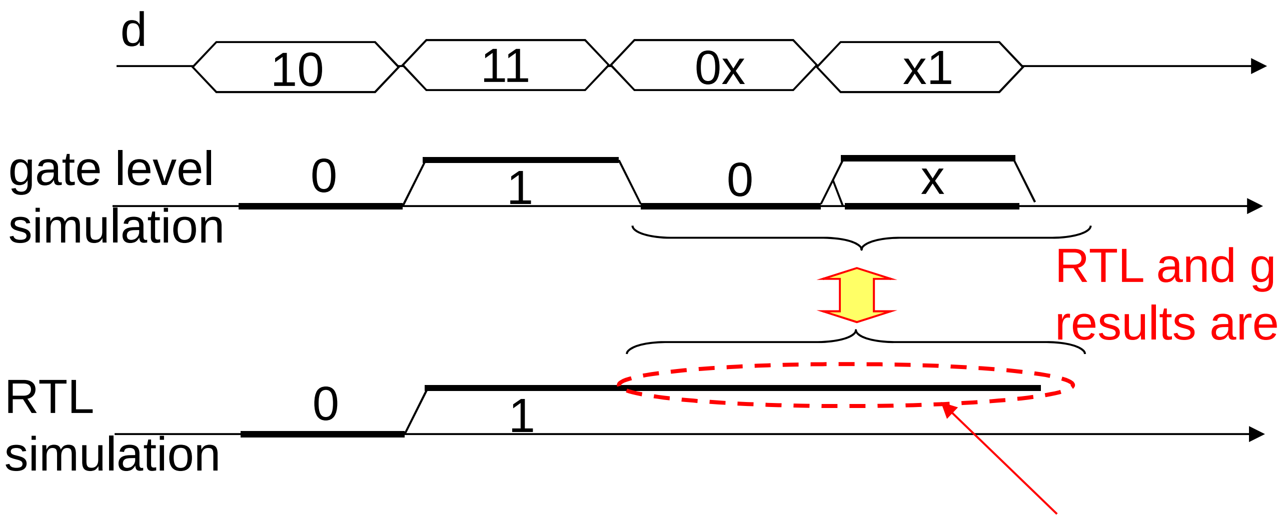
```
always @ ( d [1:0] ) begin
    case ( d[1:0] )
         2'b00, 2'b01, 2'b10 : q =  1'b0  ;
         2'b11                : q =  1'b1  ;
    endcase
end
```

No default

The code on the left is synthesized to an AND gate as below.



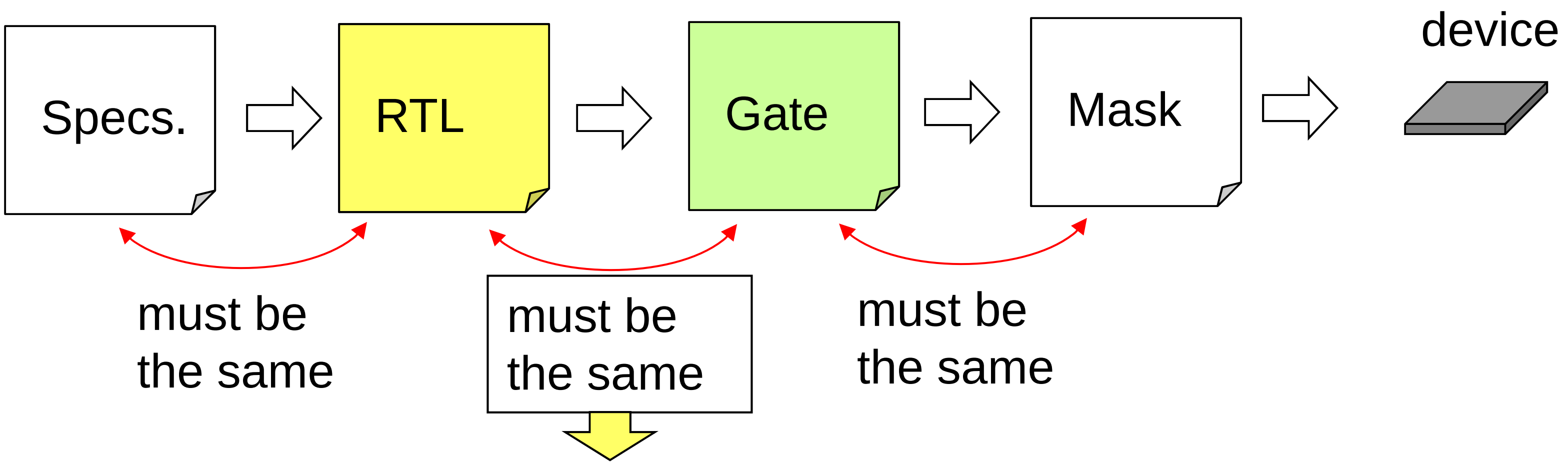
If unknown value x appears on d, gate level simulation and RTL simulation will give different results as below because RTL simulator tries to hold a previous value of q for x input. This is caused by missing default.



RTL and gate level simulation results are different.

Latching

Problem caused by the difference between RTL and gate level simulation.



Usually this is checked by comparing the results of simulation.

- RTL simulation result = gate level simulation result → netlist is equivalent to RTL
- RTL simulation result ≠ gate level simulation result → netlist and RTL are different

→ Therefore, if they are different, it means that something is wrong in RTL or gate. To find such “something” is very hard in large scale design.

→ Because of the huge cost to find such something, avoiding x is becoming a number one priority in logic design and testing.

(2) How to treat x is different among tools

In simulation, x means unknown value. But for synthesis tool, x means “don’t care”. In examples 1 and 2 shown below, RTL and gate level simulation may have different results.

Example 1:

```
begin
   case ( sel )
       2'b00 : y_out =  4'h1  ;
       2'b01 : y_out =  4'h2  ;
       2'b10 : y_out =  4'h4  ;
       default : y_out = 4'hx ;
   endcase
end
```

For the RTL code on the left, RTL simulator will place x on y_out if sel becomes 2’b11. But in gate level simulation, x will not come out for even if sel becomes 2’b11.

Example 2:

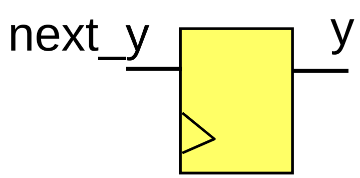
```
if ( flg ) begin
  y_out =  4’b10xx  ;
end
else begin
  y_out = 4’b0111 ;
end
```

For the RTL code on the left, RTL simulator will place x on y_out[1:0] if flg is true. But in gate level simulation, x will not come out on y_out when flg is true.

(3) There is no x in actual device. x has meaning only in simulation.

```
always @ ( posedge clk )
begin
  y <= next_y ;
end
```

⇔



Until clock rises, the value of y is unknown (x) in RTL.

Until clock rises, the value of y is unknown but it shall be either 0 or 1.

If y is used for a very important signal and it shall not be x at any time, we must apply asynchronous reset to this FF.

RTL simulation will not work correctly if such reset signal is not applied. However, in actual device an initial value of this FF shall be either 0 or 1, not x. If it happens to be 0, the logic may look working correctly although it has error of missing reset.

Therefore, if we overlooked missing reset in RTL simulation, we may not be able to find out the error by testing actual devices.

4.2 How to avoid problems caused by x

Once x appears in simulation, RTL or gate level, it is difficult to avoid problems caused by such x.
The best countermeasure for this issue is to prevent x coming into the simulation.

(1) Do not use x in RTL programming.

```
begin
   case ( sel )
       2'b00 : y_out =  4'h1  ;
       2'b01 : y_out =  4'h2  ;
       2'b10 : y_out =  4'h4  ;
       default : y_out = 4'hx ;
   endcase
end
```

```
begin
   case ( sel )
       2'b00 : y_out =  4'h1  ;
       2'b01 : y_out =  4'h2  ;
       2'b10 : y_out =  4'h4  ;
       default : y_out = 4'h4 ;
   endcase
end
```

Give some value to y_out, not x, even for don’t care cases.
If you do not care about sel=2’b11 case in the above code, assign such value as 4’h4, not x, to y_out.

This solution may increase silicon area size of this logic block.
Ask your team leader to use this countermeasure when you join actual jobs.

```
if ( flg ) begin
   y_out =  4’b10xx  ;
end
else begin
   y_out = 4’b0111 ;
end
```

⇒

```
if ( flg ) begin
   y_out =  4’b1011  ;
end
else begin
   y_out = 4’b0111 ;
end
```

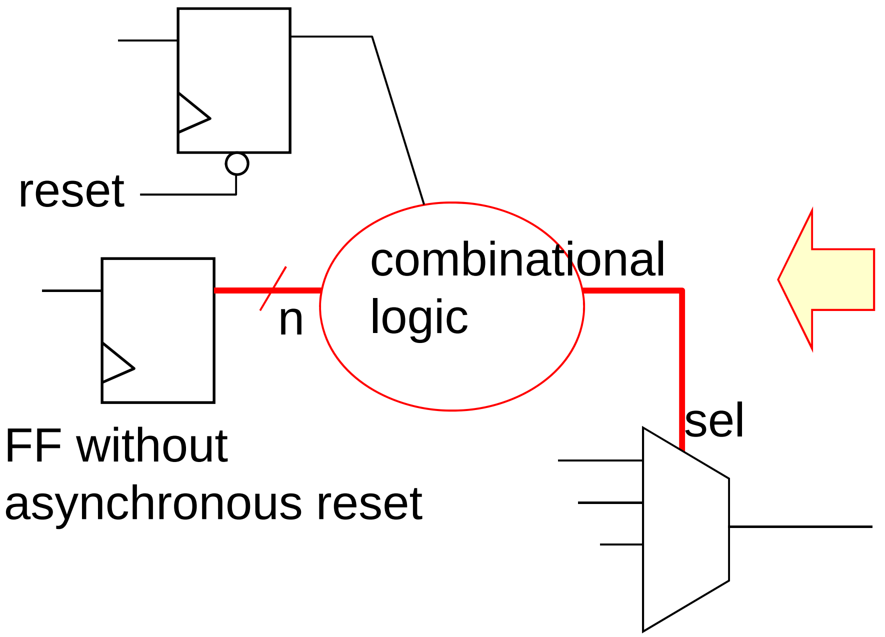
If you do not care about y_out[1:0] when flg is true, assign such value as 4’b1011 to y_out instead of 4’b10xx.

(2) Do not rise clock at time 0.

All initial values of variables in RTL are x. Therefore if we rise clock signal at time 0, x may comes into FFs.

After time 0, variables may have certain values other than x. It is safe not to rise clock signal at time 0.

(3) Do not connect an output of FF, which is not initialized by asynchronous reset, to case expressions nor to if statements.



Do not connect sel signal to FF which has no asynchronous reset because its initial value is unknown (x).

If the combinational logic is designed in a way that the unknown initial value of the FF does not propagate to sel signal, such connection is OK.

```
case ( y_sig )
   :
   :
endcase
```

```
if ( y_sig == --- ) begin
   :
end
```

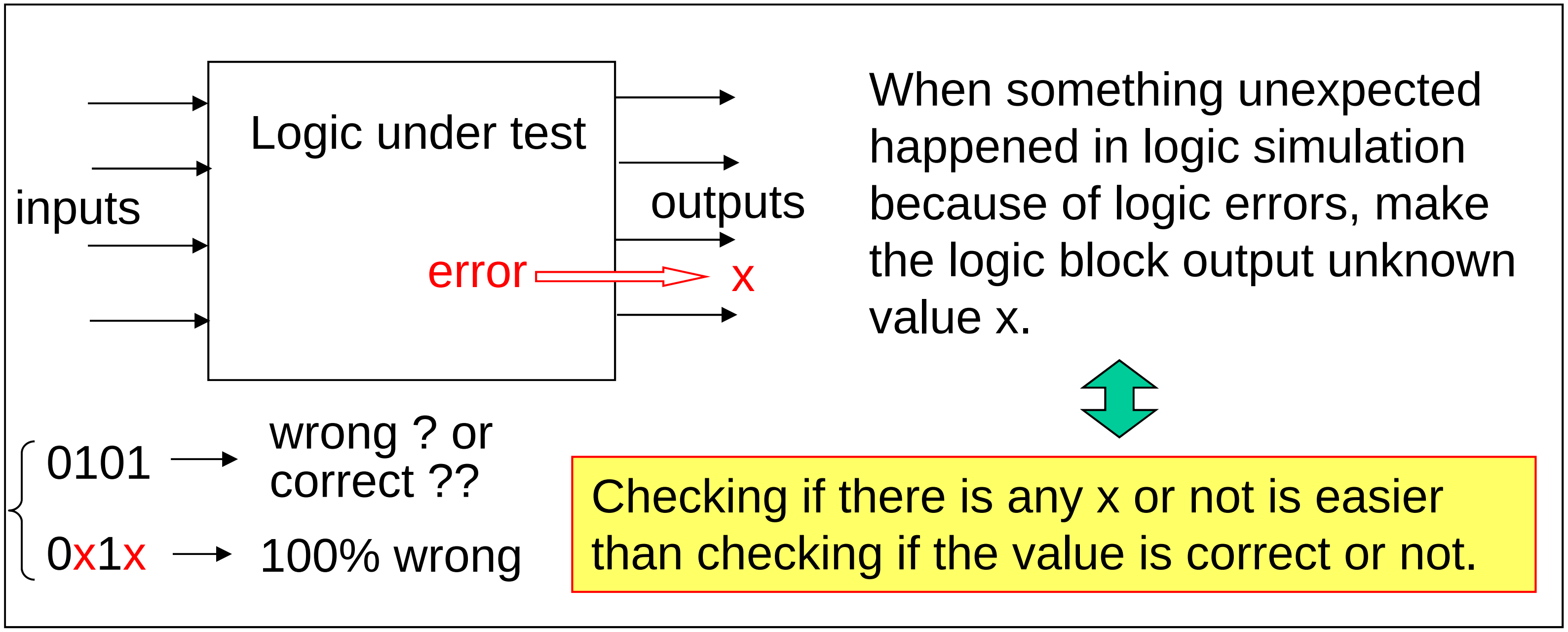
Do not connect y _sig to FF which has no asynchronous reset because its initial value is unknown (x).

If x appears on y_sig, it will cause trouble.

4.3 Debugging technique using x

Although it becomes very important to eliminate x out of logic simulation. x has been widely used for debugging in logic simulation.

When you join a job project, ask your supervisor if you can apply the debugging technique using x in the project you joined.



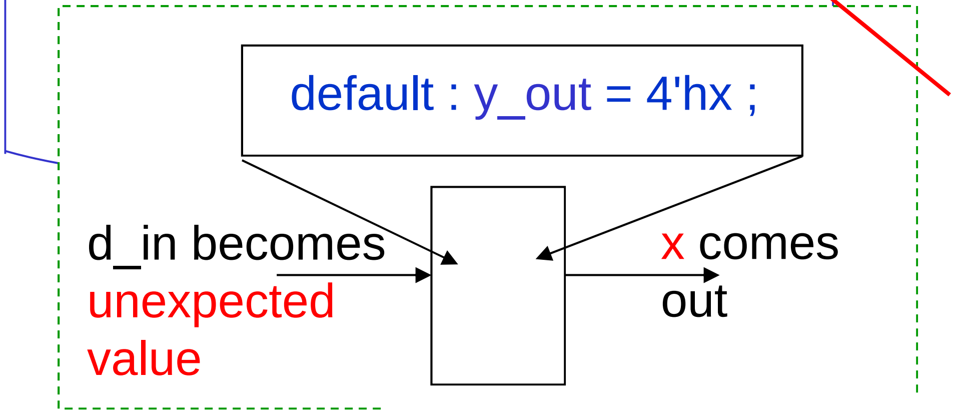
When something unexpected happened in logic simulation because of logic errors, make the logic block output unknown value x.

Checking if there is any x or not is easier than checking if the value is correct or not.

The idea of using x for debugging

A method commonly used among RTL designers for debug is to set unknown value if something unexpected occurs. However, this will cause a problem of mismatch between RTL and gate level simulation results as explained in 4.2.

```
// d_in = 2'b11 will never happen
begin
    case (d_in )
         2'b00 : y_out =  4'h1  ;
         2'b01 : y_out =  4'h2  ;
         2'b10 : y_out =  4'h4  ;
         default : y_out = 4'hx ;
    endcase
end
```



default : y_out = 4'hx ;

d_in becomes unexpected value

x comes out

This is useful for debugging.

In simulation:

This module can output x in case the input d_in becomes 2'b11 or x because of some logic bugs.

Without this default, the module will output one of 1,2, or 4 even if the input d_in is x or 2'b11.

In synthesis :

This part will not be implemented. x is ignored by synthesis tool. (same as “don't care”)

However generating x in simulation is not considered to be a good idea lately because x will cause mismatch between RTL and gate level simulation results and sometimes it will make advanced tool inapplicable.

```
always @ ( ,,,, )
begin
  case ( sel[1:0] ) begin
     2'b00 : y = a & b ;
     2'b01 : y = a | b ;
     2'b10 : y = a ^ b :
   default : y = 4'bxxxx ;
  endcase
end
```

RTL level simulation will give y value x if sel becomes 2'b11 by some logic error.

Mismatch problem may occur, but logic error can be detected.

```
always @ (  ,,, )
begin
  case ( sel[1:0] ) begin
     2'b00 : y = a & b ;
     2'b01 : y = a | b ;
     2'b10 : y = a ^ b :
   default : y = a & b ;
  endcase
end
```

RTL level simulation will never give y value x even if sel becomes 2'b11 by some logic error.

No mismatch problem, but some logic error may be overlooked.

This must be OK, because default case is not expected to happen.

Chose either style based on the project team's policy.

5. Racing

5.1 Verilog standard scheduling rule and racing

Verilog standard scheduling rule is shown on the following pages.
It does not specify any execution order among initial and always constructs.
Therefore, if there are two initial constructs as shown below, depending on which on executed first, the value of a can be either 0 or 1.

```
initial a = 0 ; // (1)
initial a = 1 ; // (2)
```

➡

If (1) is executed after (2), then a becomes 0 at time 0.
But if (2) is executed after (1), then a becomes 1 at time 0.

Scheduling among initial and always constructs are up to a tool and vendor. The same RTL code may have different simulation result if such execution order causes any difference of the behavior of the model.

⇨ This is a problem known as “racing”.

Verilog execution rule in the standard specification. (1)

```
while ( there are events ) {
  if( no active events ) {
    if( there are inactive events) {
                activate all inactive events ;
    } else if ( there are nonblocking assign update events ) {
                activate all nonblocking assign update events ;
        } else if ( there are monitor events ) {
                activate monitor events ;
            } else {
               advance T to the next event time ;
               activate all inactive events for time T ;
            }
   }
  E = any active event ;
  if ( E is an update event) {
        update the modified object ;
        add evaluation events for sensitive processes to event queue ;
  } else { /* shall be an evaluation event */
        evaluate the process ;
        add update events to the event queue ;
  }
}
```

Verilog execution rule in the standard specification. (2)

a) Active events occur at the current simulation time and can be processed in any order.

b) Inactive events occur at the current simulation time, but shall be processed after all the active events are processed.

c) Nonblocking assign update events have been evaluated during some previous simulation time, but shall be assigned at this simulation time after all the active and inactive events are processed.

d) Monitor events shall be processed after all the active, inactive, and nonblocking assign update events are processed.

e) Future events occur at some future simulation time. Future events are divided into future inactive events and future nonblocking assignment update events.

The processing of all the active events is called a simulation cycle.

The freedom to choose any active event for immediate processing is an essential source of nondeterminism in Verilog HDL.

Verilog execution rule in the standard specification. (3)

An explicit zero delay (#0) requires that the process be suspended and added as an inactive event for the current time so that the process is resumed in the next simulation cycle in the current time.

A nonblocking assignment creates a nonblocking assign update event, scheduled for a current or later simulation time.

The $monitor and $strobe system tasks create monitor events for their arguments. These events are continuously reenabled in every successive time step. The monitor events are unique in that they cannot create any other events.

Verilog execution rule in the standard specification. (4)

Continuous assignment

A continuous assignment statement corresponds to a process, sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event queue, using current values to determine the target.
A continuous assignment process is also evaluated at time 0 to ensure that constant values are propagated. This includes implicit continuous assignments

Procedural continuous assignment

A procedural continuous assignment (which is the assign or force statement) corresponds to a process that is sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event queue, using current values to determine the target.

A deassign or a release statement deactivates any corresponding assign or force statement(s).

Verilog execution rule in the standard specification. (5)

Blocking assignment

A blocking assignment statement with a delay computes the right-hand side value using the current values, then causes the executing process to be suspended and scheduled as a future event. If the delay is 0, the process is scheduled as an inactive event for the current time.
When the process is returned (or if it returns immediately if no delay is specified), the process performs the assignment to the left-hand side and enables any events based upon the update of the left-hand side. The values at the time the process resumes are used to determine the target(s). Execution may then continue with the next sequential statement or with other active events.

Nonblocking assignment

A nonblocking assignment statement always computes the updated value and schedules the update as a nonblocking assign update event, either in this time step if the delay is zero or as a future event if the delay is nonzero.

The values in effect when the update is placed on the event queue are used to compute both the right-hand value and the left-hand target.

Verilog execution rule in the standard specification. (6)

Switch (transistor) processing

The event-driven simulation algorithm depends on unidirectional signal flow and can process each event independently. The inputs are read, the result is computed, and the update is scheduled.

The Verilog HDL provides switch-level modeling in addition to behavioral and gate-level modeling. Switches provide bidirectional signal flow and require coordinated processing of nodes connected by switches.

The Verilog HDL source elements that model switches are various forms of transistors, called tran, tranif0, tranif1, rtran, rtranif0, and rtranif1.

Switch processing shall consider all the devices in a bidirectional switch-connected net before it can determine the appropriate value for any node on the net because the inputs and outputs interact. A simulator can do this using a relaxation technique. The simulator can process tran at any time. It can process a subset of tran-connected events at a particular time, intermingled with the execution of other active events.

5.2 Racing examples

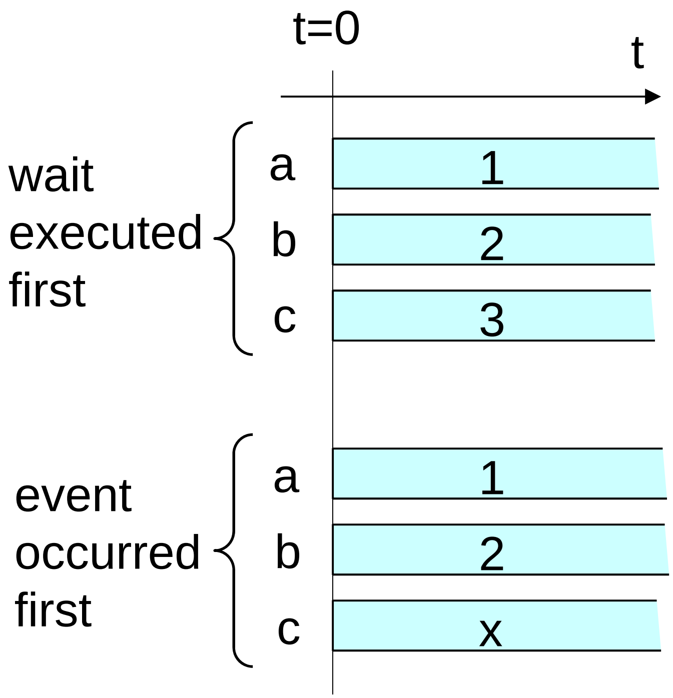
(1) Racing among always and initial

The RTL code on the left causes racing problem. Depending on which construct is executed first, the simulation result is different.

```
reg [7:0] a, b, c; //non-FF

always @ (a or b)
begin
   c = a + b;
end

initial begin
    a = 1; b = 2;
end
```

wait for event → change of a or b

event → a and b change from x to 1 and 2 respectively.

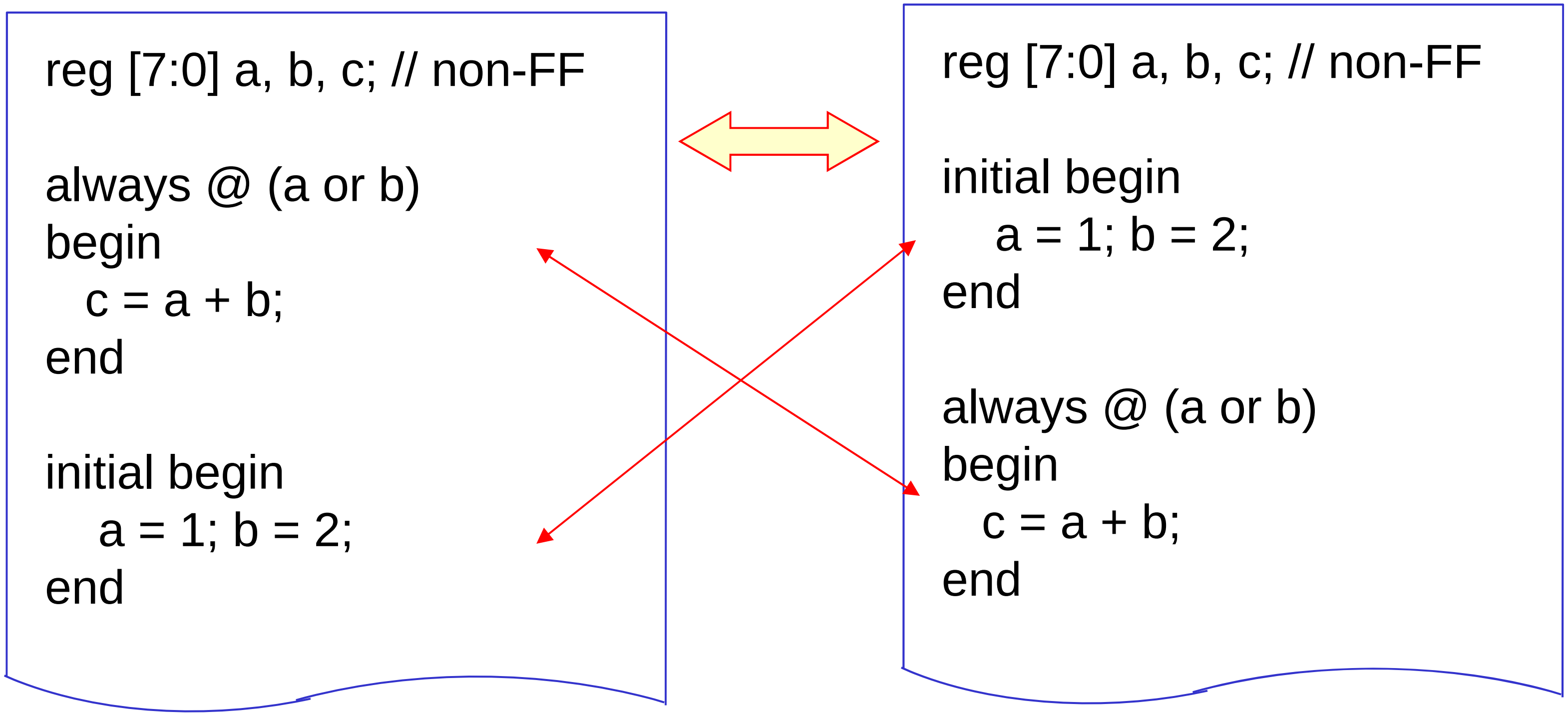


c keeps the value x until a or b is given a new value.

Because there is no standard rule for execution sequence of statements, a simulation result may have tool dependency.
Some tools execute from top to bottom, while others from bottom to top.

This means that there is order dependency in what sequence those statements are coded.

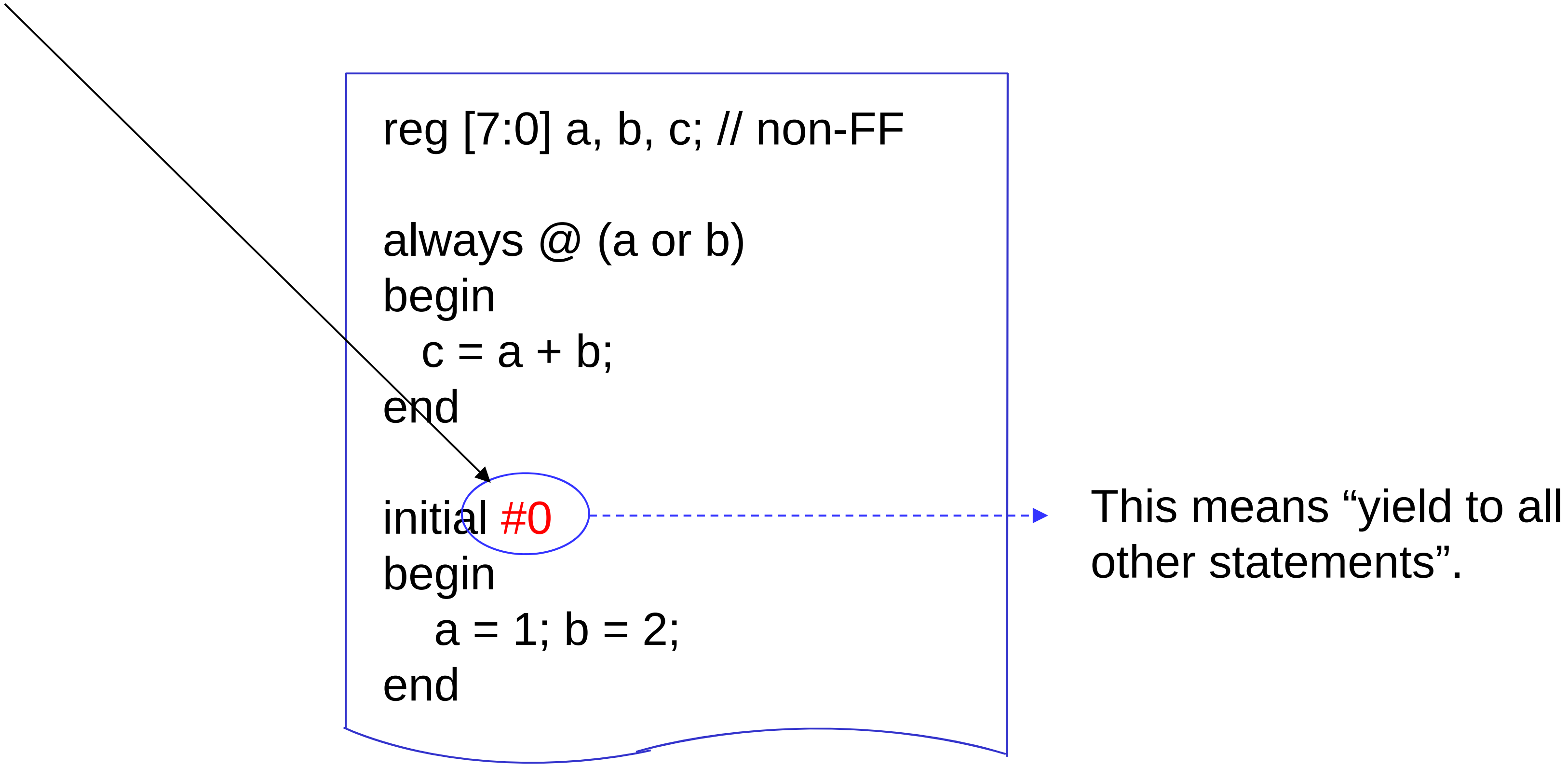
The following two programs may have different simulation results.



```
reg [7:0] a, b, c; // non-FF

always @ (a or b)
begin
  c = a + b;
end

initial begin
   a = 1; b = 2;
end
```

```
reg [7:0] a, b, c; // non-FF

initial begin
   a = 1; b = 2;
end

always @ (a or b)
begin
  c = a + b;
end
```

One way to avoid race condition between always and initial statements, using #0, delay zero, may be recommended.

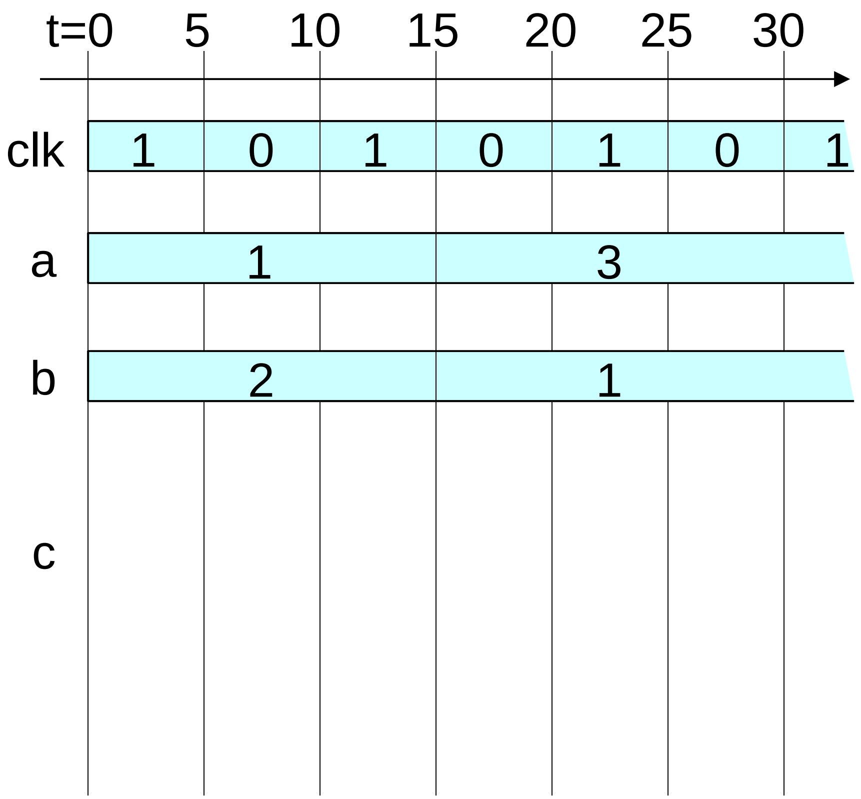


```
reg [7:0] a, b, c; // non-FF

always @ (a or b)
begin
   c = a + b;
end

initial #0
begin
    a = 1; b = 2;
end
```

This means “yield to all other statements”.

Q5.2-1 : For the verilog RTL code shown below, draw the expected result on the time chart below.

```
reg clk ; // non-FF
reg [7:0] a, b ; // non-FF
reg [7:0] c ; // FF

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

initial begin
    a = 1; b = 2;
    #15 a = 3; b=1 ;
end

always begin
    clk = 1;
    #5 clk = 0 ;
    #5 ;
end
```

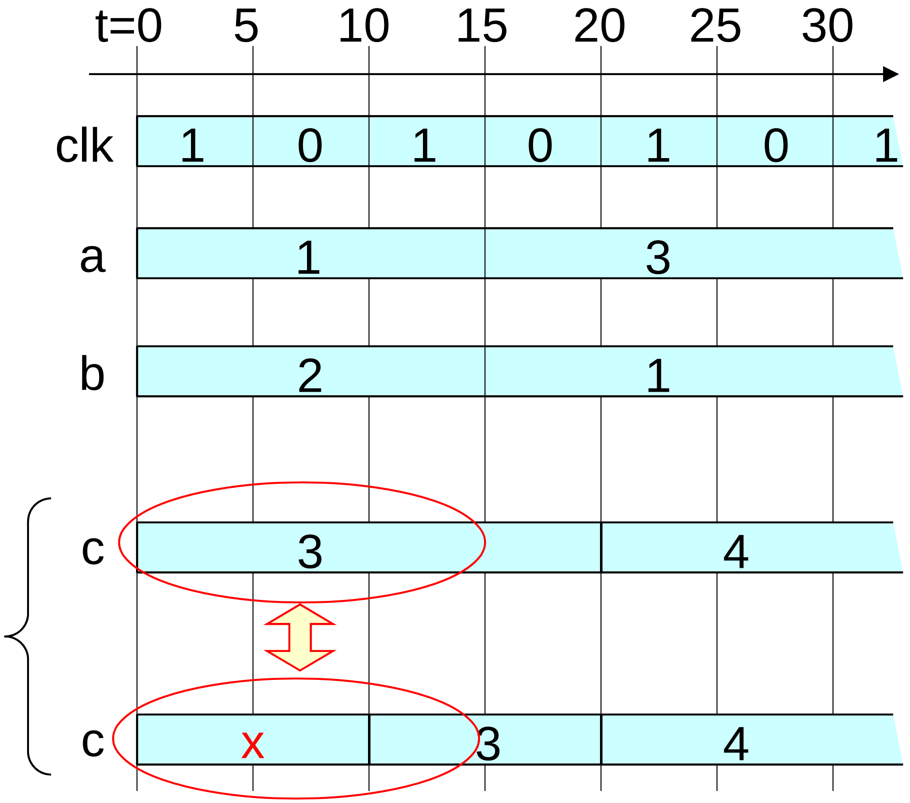


This is not a good example.
Do not make a rising clock at t=0.

Q5.2-1 : A sample answer.

There exists a race condition between the two always statements,
the result may be different depending on tools.
There are two possible results as shown below.

```
reg clk ; // non-FF
reg [7:0] a, b ; // non-FF
reg [7:0] c ; // FF

always @ (posedge clk)
begin
   c = a + b;
end
initial begin
    a = 1; b = 2;
    #15 a = 3; b=1 ;
end
always begin
    clk = 1;
    #5 clk = 0 ;
    #5 ;
end
```



Depending on the simulation
tool, the results are different.

Q5.2-2 : By using #0, avoid race condition as shown below.
Which one of the codes will work without race condition?

Code 1

```
reg clk ; // non-FF
reg [7:0] a, b ; // non-FF
reg [7:0] c ; // FF

always #0 @ (posedge clk)
begin
   c = a + b;
end

initial begin
    a = 1; b = 2;
    #15 a = 3; b=1 ;
end

always begin
    clk = 1;
    #5 clk = 0 ;
    #5 ;
end
```

Code 2

```
reg clk ; // non-FF
reg [7:0] a, b ; // non-FF
reg [7:0] c ; // FF

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

initial #0 begin
    a = 1; b = 2;
    #15 a = 3; b=1 ;
end

always begin
    clk = 1;
    #5 clk = 0 ;
    #5 ;
end
```

Code 3

```
reg clk ; // non-FF
reg [7:0] a, b ; // non-FF
reg [7:0] c ; // FF

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

initial begin
    a = 1; b = 2;
    #15 a = 3; b=1 ;
end

always #0 begin
    clk = 1;
    #5 clk = 0 ;
    #5 ;
end
```

This is not a good example. Do not make a rising clock at t=0

Q5-2 : A sample answer.

Code 1 and Code 3 may not cause racing. With code 2, “wait for clock rise” is executed after the first clock rise. With code 3, clock rises after “wait for clock rise” is executed. Code 2 can not solve the racing between two always statements.

From the viewpoint of eliminating x appearing in the result, code 3 may be better.

Code 1

```
always #0 @ (posedge clk)
begin
   c = a + b;
end

initial begin
    a = 1; b = 2;
    #15 a = 3; b=1 ;
end

always begin
    clk = 1;
    #5 clk = 0 ;
    #5 ;
end
```

Code 2

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

initial #0 begin
    a = 1; b = 2;
    #15 a = 3; b=1 ;
end

always begin
    clk = 1;
    #5 clk = 0 ;
    #5 ;
end
```

Code 3

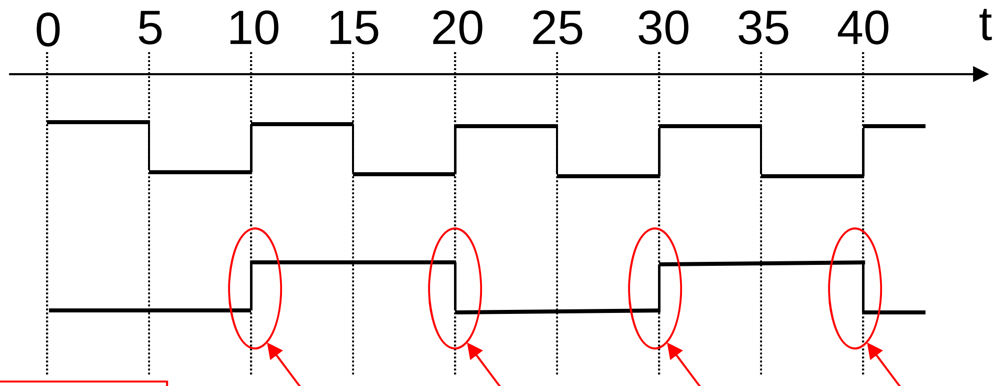
```
always @ (posedge clk)
begin
   c = a + b;
end

initial begin
    a = 1; b = 2;
    #15 a = 3; b=1 ;
end

always #0 begin
    clk = 1;
    #5 clk = 0 ;
    #5 ;
end
```

(2) Racing among clock and other signals

```
always begin
        clk = 1;
    #5 clk = 0 ;
    #5 ;
end
initial begin
          ctl_in = 0 ;
    #10 ctl_in = 1 ;
    #10 ctl_in = 0 ;
    #10 ctl_in = 1 ;
    #10 ctl_in = 0 ;
end
```

This is not a good example. Do not make a rising clock at t=0

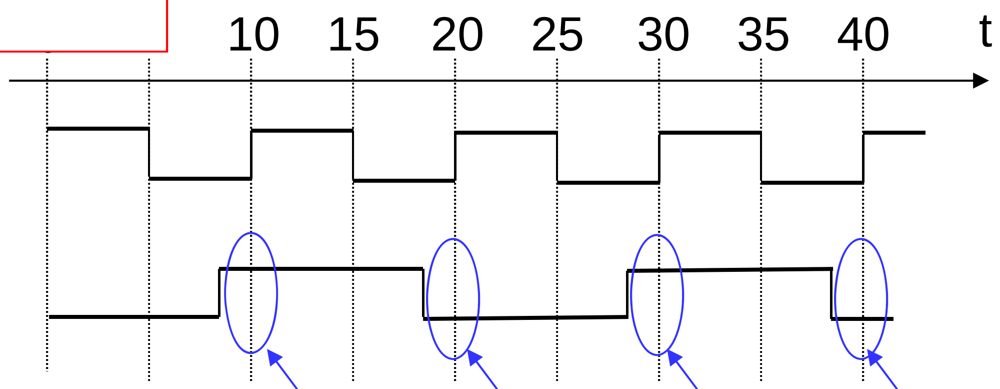
The result may be different because of race condition between clock and ctl_in.



ctl_in at clock rise time may be 0 or 1 depending on the tool.

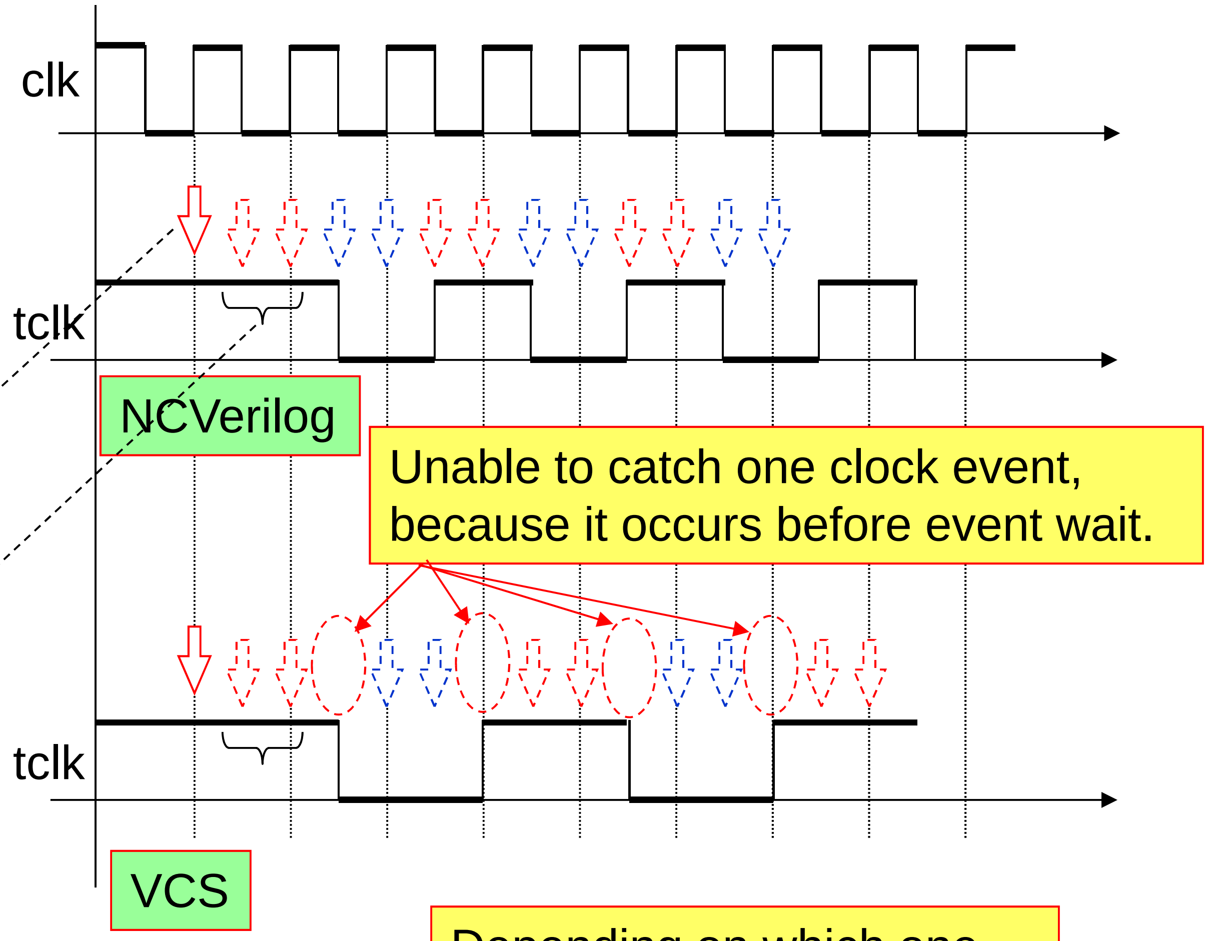
```
initial begin
          ctl_in = 0 ;
    #8   ctl_in = 1 ;
    #10 ctl_in = 0 ;
    #10 ctl_in = 1 ;
    #10 ctl_in = 0 ;
end
```

Avoid changing signal at clock rise time.



ctl_in at clock rise time is the same between the tools.

```
initial begin
   clk = 1'b1;
   forever begin
      #10 clk = ~clk;
   end
end

initial begin
    tclk = 1'b1;
    @(posedge clk);
    forever begin
        repeat(2) @(clk);
        #10 tclk = ~tclk;
    end
end
```

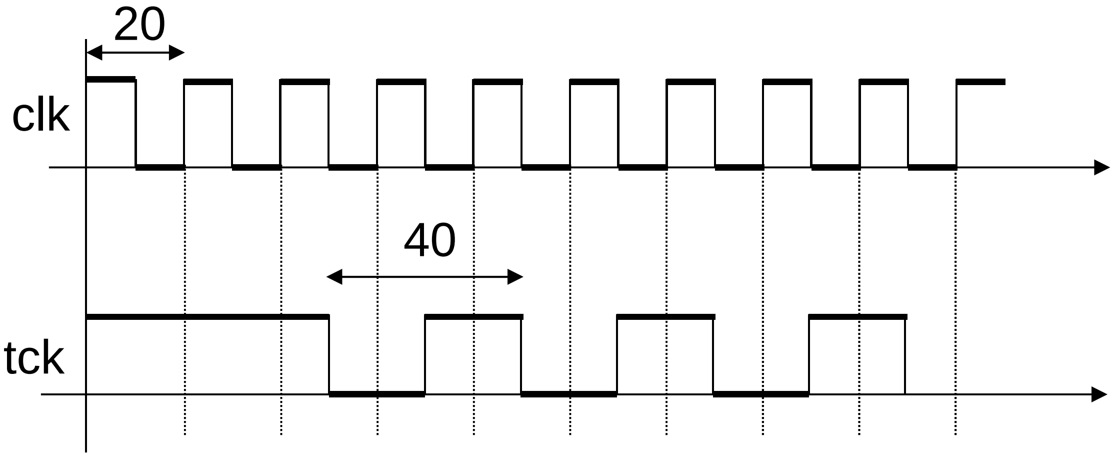


Unable to catch one clock event, because it occurs before event wait.

Depending on which one executed earlier, the results are different.

Q5.2-3 : Change the code below to remove racing problem.

```
initial begin
  clk = 1'b1;
  forever begin
    #10 clk = ~clk;
  end
end

initial begin
   tck = 1'b1;
   @(posedge clk);
   forever begin
      repeat(2) @(clk);
      #10 tclk = ~tclk;
   end
end
```

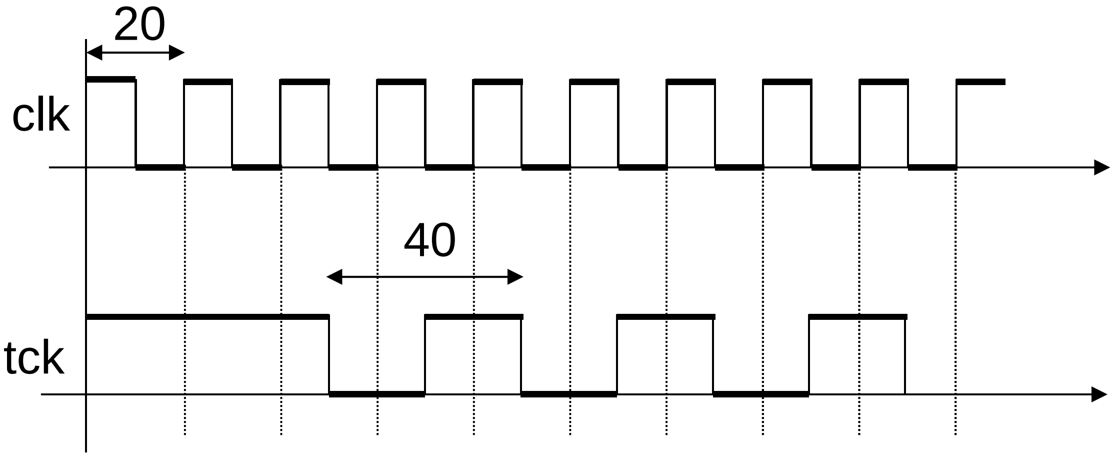


Raising clock signal at time 0 may cause problems.
Do not raise clock signal at time 0.
However, for Q6-3, do not change this part.

Correct this part only to remove racing problem shown on the previous page.

Q5.2-3 : A sample answer

```
initial begin
  clk = 1'b1;
  forever begin
     #10 clk = ~clk;
  end
end

initial begin
   tclk = 1'b1;
   @(posedge clk);
   forever begin
      repeat(2) @(clk);
      #10 tclk = ~tclk;
   end
end
```



The problem occurs from the fact that clock event (0 to 1 or 1 to 0) and clock event wait take place at the same time.
Therefore changing clock event to posedge can avoid the collision.

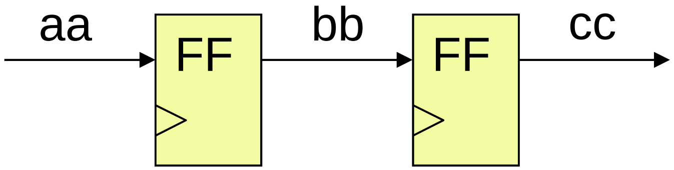
```
forever begin
     @ ( posedge clk ) ;
     #10 tclk = ~tclk;
  end
```

Rising clock at t=0 is not a good idea. Do not rise clock at t=0.

(3) Racing among FFs

code 1

```
always @ (posedge clk) begin  // FF3
      bb <= aa;        // (1)
      cc <= bb ;       // (2)
end
```



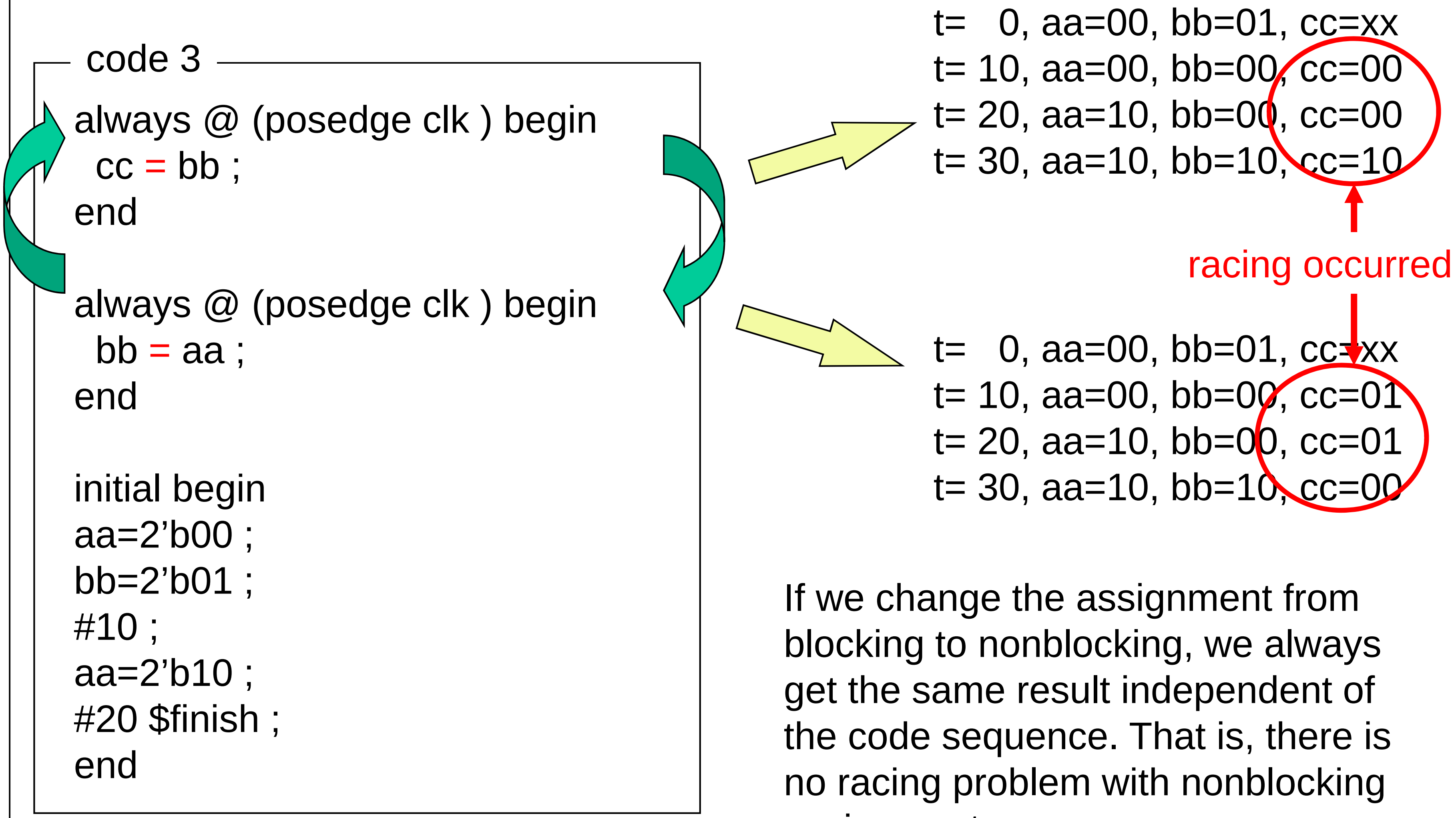
Code 1 is a typical coding style to define two sequential FFs and it will not cause a racing problem.

code 2

```
always @ (posedge clk) begin  // FF1
      bb =   aa;  // (3)
end

always @ (posedge clk) begin  //  FF2
      cc =  bb ;  // (4)
end
```

However, code 2 will cause a racing problem because depending on which always construct is executed earlier, the result is different as shown on the next page.

Code 3 gives different results as shown on the right depending on the sequence that always constructs are written.



code 3

```
always @ (posedge clk ) begin
  cc = bb ;
end

always @ (posedge clk ) begin
  bb = aa ;
end

initial begin
aa=2'b00 ;
bb=2'b01 ;
#10 ;
aa=2'b10 ;
#20 $finish ;
end
```

```
t=   0, aa=00, bb=01, cc=xx
t= 10, aa=00, bb=00, cc=00
t= 20, aa=10, bb=00, cc=00
t= 30, aa=10, bb=10, cc=10
```

racing occurred

```
t=   0, aa=00, bb=01, cc=xx
t= 10, aa=00, bb=00, cc=01
t= 20, aa=10, bb=00, cc=01
t= 30, aa=10, bb=10, cc=00
```

If we change the assignment from blocking to nonblocking, we always get the same result independent of the code sequence. That is, there is no racing problem with nonblocking assignment.

code 4

```
always @ (posedge clk) begin  // FF1
      bb <=   aa;  // (5)
end

always @ (posedge clk) begin  //  FF2
      cc <=  bb ;  // (6)
end
```

This code looks very like code 2. And depending on tools, (5) may be executed before (6) or vise versa.

However, the assignment is not done before all the blocking assignments are done. In Verilog standard rule, nonblocking assignments are executed after all the blocking updates are done.

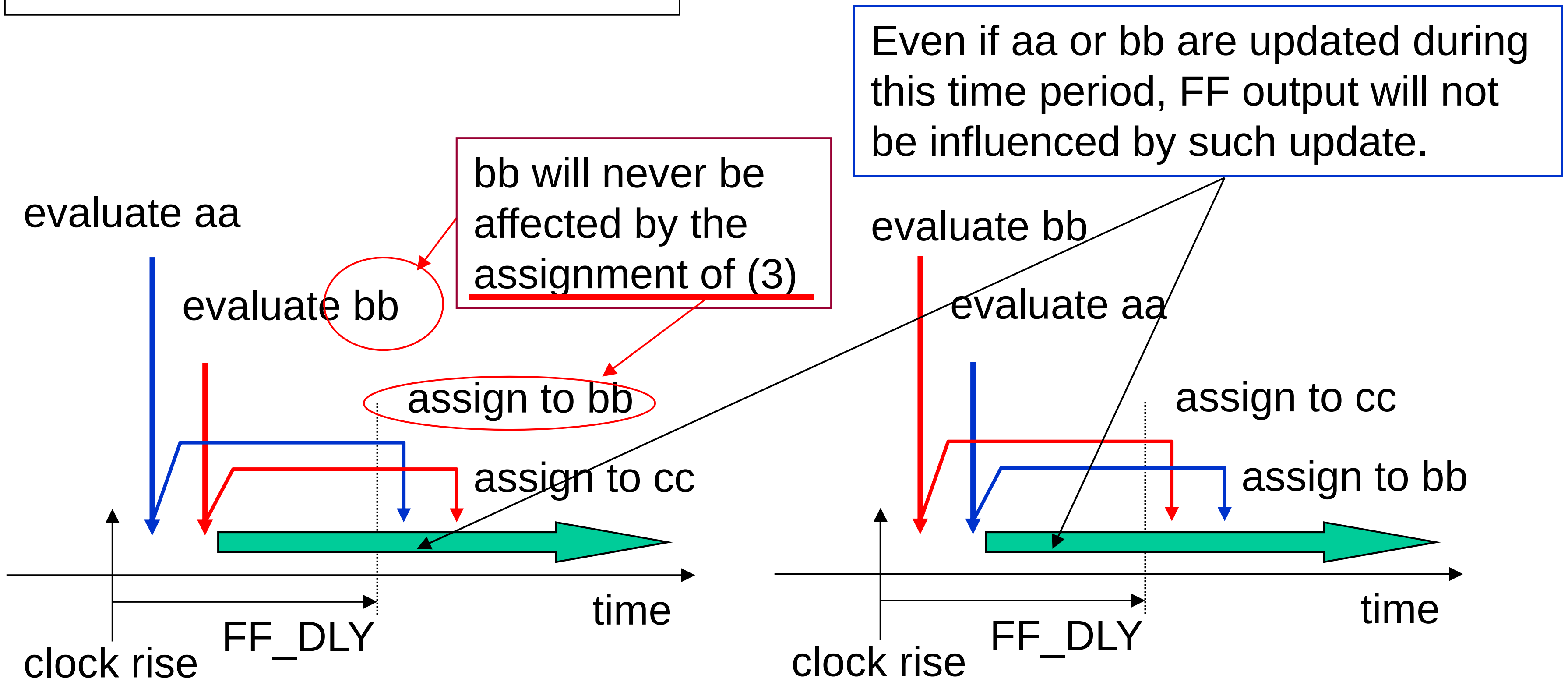
This means that In code 4 bb is not given the value of aa before bb in (6) is evaluated.
Therefore, code 4 has no racing problem.

Recently, adding delay to the assignment (5) and (6) is recommended as a countermeasure for the possible racing problems. This is to make it sure that racing problem will never occur to those FFs.

code 5

```
always @ (posedge clk) begin  // FF1
      bb <= #FF_DLY  aa;  // (3)
end

always @ (posedge clk) begin  // FF2
      cc <=#FF_DLY bb ;  // (4)
end
```

Code 4 without FF_DLY will not have racing problems.
However, adding delay as shown on the left is recommended recently to surely avoid racing problems in sophisticated logic.



Even if aa or bb are updated during this time period, FF output will not be influenced by such update.

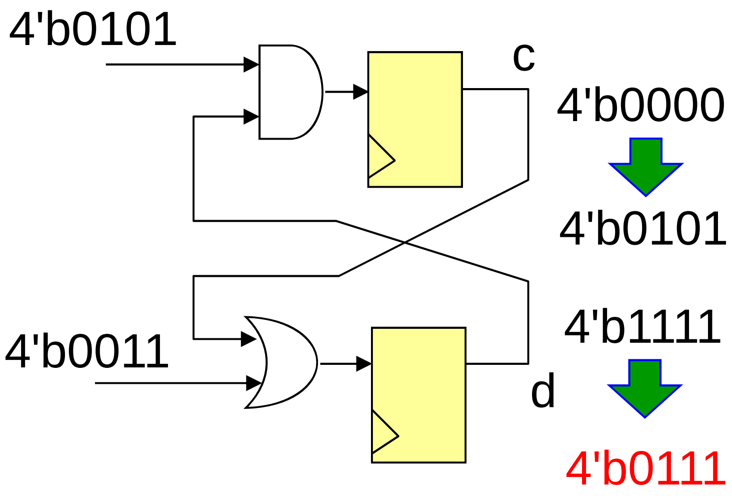
bb will never be affected by the assignment of (3)

Another example of racing problem of FF

```
always @ * begin                          (3)
    next_d = c | 4'h3 ;
end

always @ * begin                          (1)
    next_c = d & 4'h5 ;
end

always @ ( posedge clk ) begin            (2)
    c =  next_c ;
end

always @ ( posedge clk ) begin
     d = next_d ;                         (4)
end
```

Using blocking procedural assignment for FF is not recommended. The code above is just for explanation.

If c is 4'h0 and d is 4'hf before clock rises, and if c is assigned 4'b0101 by (1) and (2), and then (3) and (4) is executed, d becomes 4'b0111.



Unexpected result

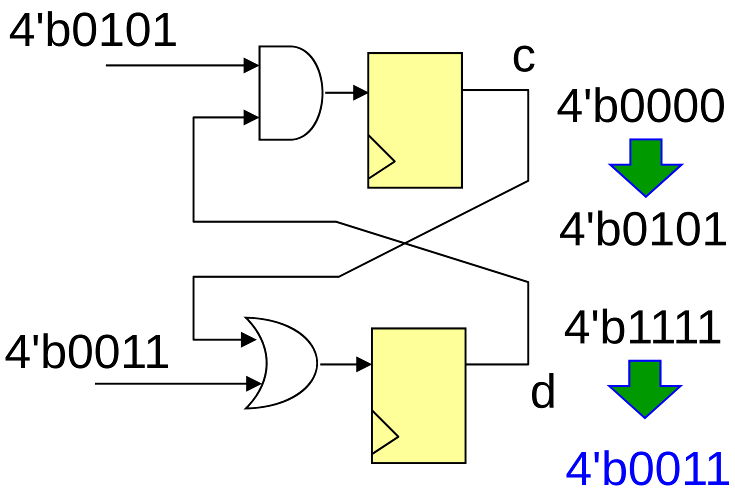
To avoid the problem, we can add delay in FF as below.

By the delay, c and d are assigned their new value 1 time unit after clock rises. Therefore, next_d is not affected by the change of c.

Actually, c does not change at clock rise time because of the delay, therefore, next_d does not change at clock rise time.

```
always @ * begin
    next_d = c | 4'h3 ;
end

always @ * begin
    next_c = d & 4'h5 ;
end

always @ ( posedge clk ) begin
    c = #1  next_c ;
end

always @ ( posedge clk ) begin
     d = #1  next_d ;
end
```

Using blocking procedural assignment for FF is not recommended. The code above is just for explanation.

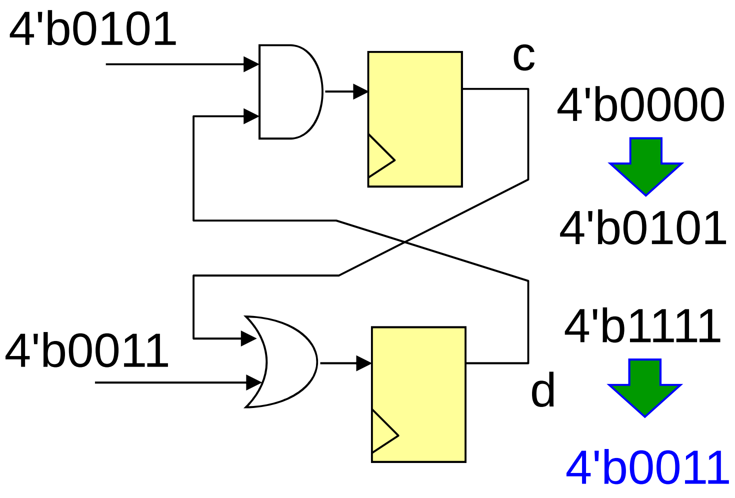


Expected result

The racing does not happen for such simple examples shown on the previous page, if we use nonblocking procedural assignment in always statements for FF.

However, it is recommended to apply delay for FF as shown on the next page.

```
always @ * begin
    next_d = c | 4'h3 ;
end

always @ * begin
    next_c = d & 4'h5 ;
end

always @ ( posedge clk ) begin
    c <= next_c ;
end

always @ ( posedge clk ) begin
     d <= next_d ;
end
```



Use delay to avoid race condition for FF

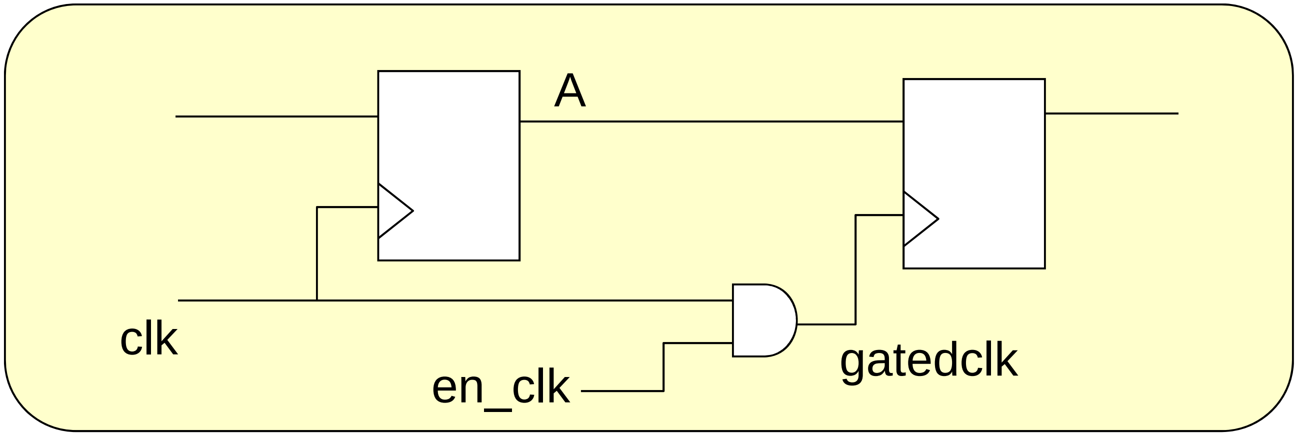
```
parameter FF_DELAY = 1 ;


always @ ( posedge clk or negedge rst_n )
begin
  if ( rst_n==1'b0 ) sig_q <= #FF_DELAY 1'b0 ;
  else            sig_q <= #FF_DELAY next_sig_q ;
end
```

◎

Using delay for FF is recommended to avoid racing problems.

Using delay is not widely accepted in RTC yet, follow the rule of each project you join.

Without delay, signal A and gatedclk cause racing.



In general, we must not use # in RTL code which will be synthesized, the example in the previous page is the only exception for this rule. We can use # for always statement for FF to avoid race condition.
Do not use # for other purpose.

RTL code to create gated clock can be written as below;

```
always @ ( clk or en_clk ) begin
   gatedclk  = clk & en_clk ;
end
```

en_clk must be controlled not to cause hazard.

However, it is better to write as below to avoid racing.

```
assign  gatedclk = clk & en_clk ;
```

Note: The above style does not guarantee that there shall be no racing.

(4) Racing caused by improper sensitivity list

```
reg [1:0] b ;

assign c = a ;
assign d = ~a ;
always  @ ( a ) begin
   b = { c, d } ;
end
```

Neither c nor d is listed in the sensitivity list on the left. Whenever a changes, c and d change their values, therefore the code on the left looks OK. But it will cause racing problem.

```
initial begin
a = 0 ;
#10 a = 1 ;
#10 a = 0 ;
#10 a = 1 ;
#10 $finish ;
end
```

```
a=0, b=xx, c=0, d=1
a=1, b=01, c=1, d=0
a=0, b=10, c=0, d=1
a=1, b=01, c=1, d=0
```

The result may different from what the designer intended.

```
reg [1:0] b ;

assign c = a ;
assign d = ~a ;
always  @ ( c or d ) begin
   b = { c, d } ;
end
```

improved code

When we improve the code as shown on the left, then the result looks like below.

```
a=0, b=01, c=0, d=1
a=1, b=10, c=1, d=0
a=0, b=01, c=0, d=1
a=1, b=10, c=1, d=0
```

```
initial begin
a = 0 ;
#10 a = 1 ;
#10 a = 0 ;
#10 a = 1 ;
#10 $finish ;
end
```

~~a=0, b=xx, c=0, d=1~~
~~a=1, b=01, c=1, d=0~~
~~a=0, b=10, c=0, d=1~~
~~a=1, b=01, c=1, d=0~~

```
reg [1:0] b ;

assign c = a ;
assign d = ~a ;
always  @ ( a ) begin
#0   b = { c, d } ;
end
```

Adding zero delay can avoid the racing problem in this case. However, this is not a good solution.

The best solution is to list up all the RHS in the sensitivity list.

(5) Racing around $display system task

```
assign p = q ;

initial begin
  q = 1 ;
  #1 q = 0 ;
 $display(p) ;
end
```

After assigning the value 0 to q, the simulator may either continue and execute the $display task or execute the update for p, followed by the $display task.

Therefore as a result of executing the code on the left, the simulator may display either 0 or 1.

Be careful about the timing when checking variables by using $display system task.

```
initial begin
a<= 0  ;
$display(“t=%d, a=%b”, $stime, a ) ;
#10 a<= 1 ;
$display(“t=%d, a=%b”, $stime, a ) ;
end
```

```
t=          0, a=x
t=         10, a=0
```

$display system task displays the value of variables at the time it is enabled.

```
initial begin
a<= 0  ;
$display(“t=%d, a=%b”, $stime, a ) ;
a<=#10  1 ;
$display(“t=%d, a=%b”, $stime, a ) ;
end
```

```
t=          0, a=x
t=          0, a=x
```

```
initial begin
a<= 0  ;
$display(“t=%d, a=%b”, $stime, a ) ;
a<=#10  1 ;
#1 $display(“t=%d, a=%b”, $stime, a ) ;
end
```

```
t=          0, a=x
t=          1, a=0
```

$strobe system task shows the value of variables after they settled.

```
initial begin
a<= 0  ;
$strobe(“t=%d, a=%b”, $stime, a ) ;
#10 a<= 1 ;
$strobe(“t=%d, a=%b”, $stime, a ) ;
end
```

```
t=          0, a=0
t=         10, a=1
```

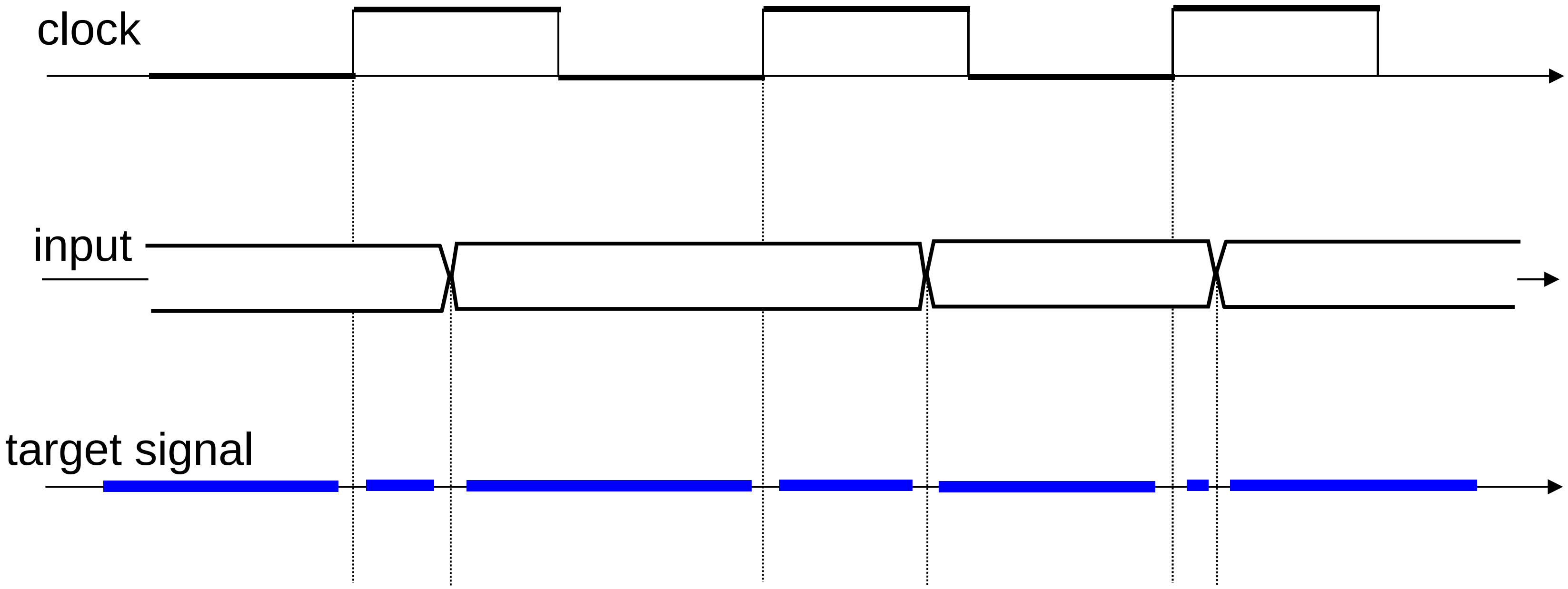
```
initial begin
a<= 0  ;
$strobe(“t=%d, a=%b”, $stime, a ) ;
a<=#10  1 ;
$strobe(“t=%d, a=%b”, $stime, a ) ;
end
```

```
t=          0, a=0
t=          0, a=0
```

```
initial begin
a<= 0  ;
$strobe(“t=%d, a=%b”, $stime, a ) ;
a<=#10  1 ;
#1 $strobe(“t=%d, a=%b”, $stime, a ) ;
end
```

```
t=          0, a=0
t=          1, a=0
```

To check the value of variables it is recommended to use $strobe system task and also avoid the timing when they change their value.



To avoid possible racing, observe target signal at blue thick lines.

5.3 How to avoid racing

(1) LHS in procedures

Never use the same variable on LHS of different procedures.

(2) Signal change timing

Never change signals at clock rise time except those signals defined in FFs.

(3) Sensitivity list of always construct for combinational logic

List all signals appearing on RHS except those appearing on LHS.

(4) Always construct of FF

(a) Using nonblocking procedural assignment.
(b) Applying delay in nonblocking procedural assignment.

Using delay to avoid racing is not widely accepted in RTE yet, follow the rule of each project you join.

Q5.3-1 Answer if the following statements are correct or wrong.

(1) The following piece of RTL code will cause a racing problem because the same variable clk is written on LHS of different procedures.

```
initial begin
      clk = 0 ;
end
always begin
  clk = ~clk ; #10 ;
end
```

Q5.3-1 Answer if the following statements are correct or wrong.

(2) The following pieces of RTL code will not cause a racing problem because input signal a and b are controlled not to change at clock rise time.

```
initial begin
       a = 0 ; b = 1 ;
#15 a = 1 ; b = 0 ;
#35 a = 0 ;
end
always begin
  clk = 1 ; #10 ;
  clk = 0 ; #10 ;
end
```

Q5.3-1 Answer if the following statements are correct or wrong.

(3) The following pieces of RTL code will not cause a racing problem because input signal a and b are controlled not to change at clock rise time.

```
initial begin
    a = 0 ; b = 1 ;
#15 a = 1 ; b = 0 ;
#35 a = 0 ;
end
always begin
  clk = 0 ; #10 ;
  clk = 1 ; #10 ;
end
always @ ( posedge clk ) begin
  y <= next_y ;
end
always @ ( a or b ) begin
  next_y = #15 a | b ;
end
```

Q5.3-1 A sample answer

(1) The following piece of RTL code will cause a racing problem because the same variable clk is written on LHS of different procedures.

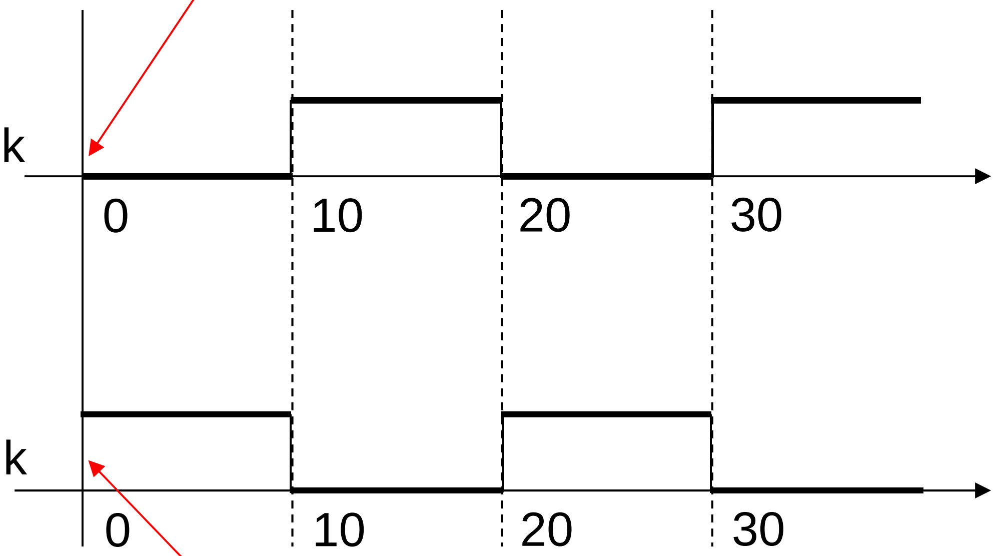
This statement is correct.

```
initial begin
       clk = 0 ;
end
always begin
  clk = ~clk ; #10 ;
end
```

At time 0, clk changes from x to x (~x) and then to 0.

If always is executed first and then initial.

If initial is executed first and then always.



At time 0, clk changes from x to 0 and then to 1 (~0).

Q5.3-1 A sample answer

(2) The following pieces of RTL code will not cause a racing problem because input signal a and b are controlled not to change at clock rise time.

This statement is wrong.

```
initial begin
       a = 0 ; b = 1 ;
#15 a = 1 ; b = 0 ;
#35 a = 0 ;
end
always begin
  clk = 1 ; #10 ;
  clk = 0 ; #10 ;
end
```

At time 0, clock signal clk rises from x to 1, and input signals a and b are changed at time 0 from x to 0 and x to 1 respectively.

Do not rise clock at time 0.

Q5.3-1 Answer if the following statements are correct or wrong.

(3) The following pieces of RTL code will not cause a racing problem because input signal a and b are controlled not to change at clock rise time.

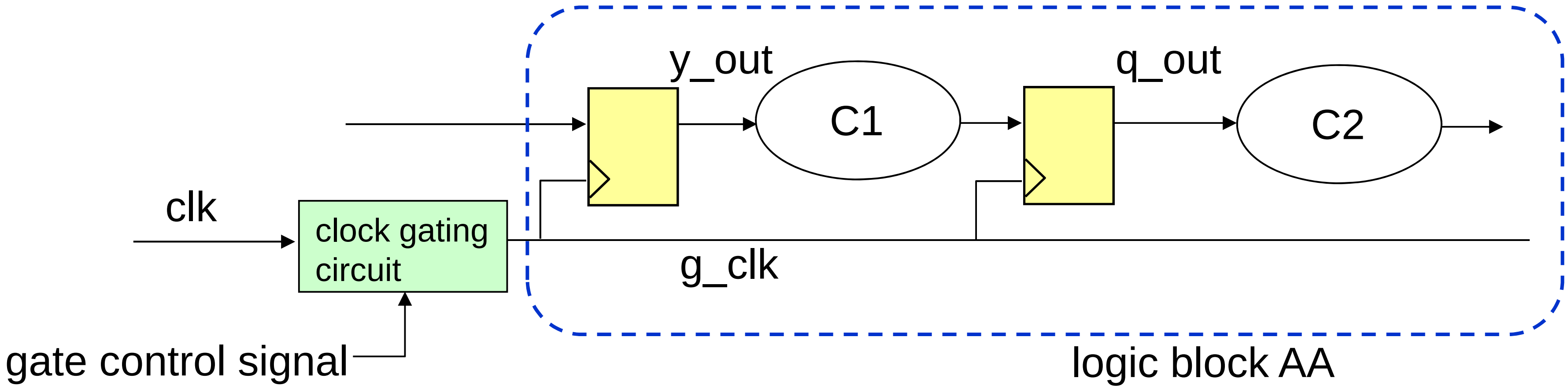
```
initial begin
       a = 0 ; b = 1 ;
#15 a = 1 ; b = 0 ;
#35 a = 0 ;
end
always begin
  clk = 0 ; #10 ;
  clk = 1 ; #10 ;
end
always @ ( posedge clk ) begin
  y <= next_y ;
end
always @ ( a or b ) begin
  next_y = #15 a | b ;
end
```

This statement is wrong.

a and b are controlled not to change at clock rise time, but next_y will change at clock rise time.

6. Clock gating

Clock gating is a technique to reduce power consumption by stopping clock signals.



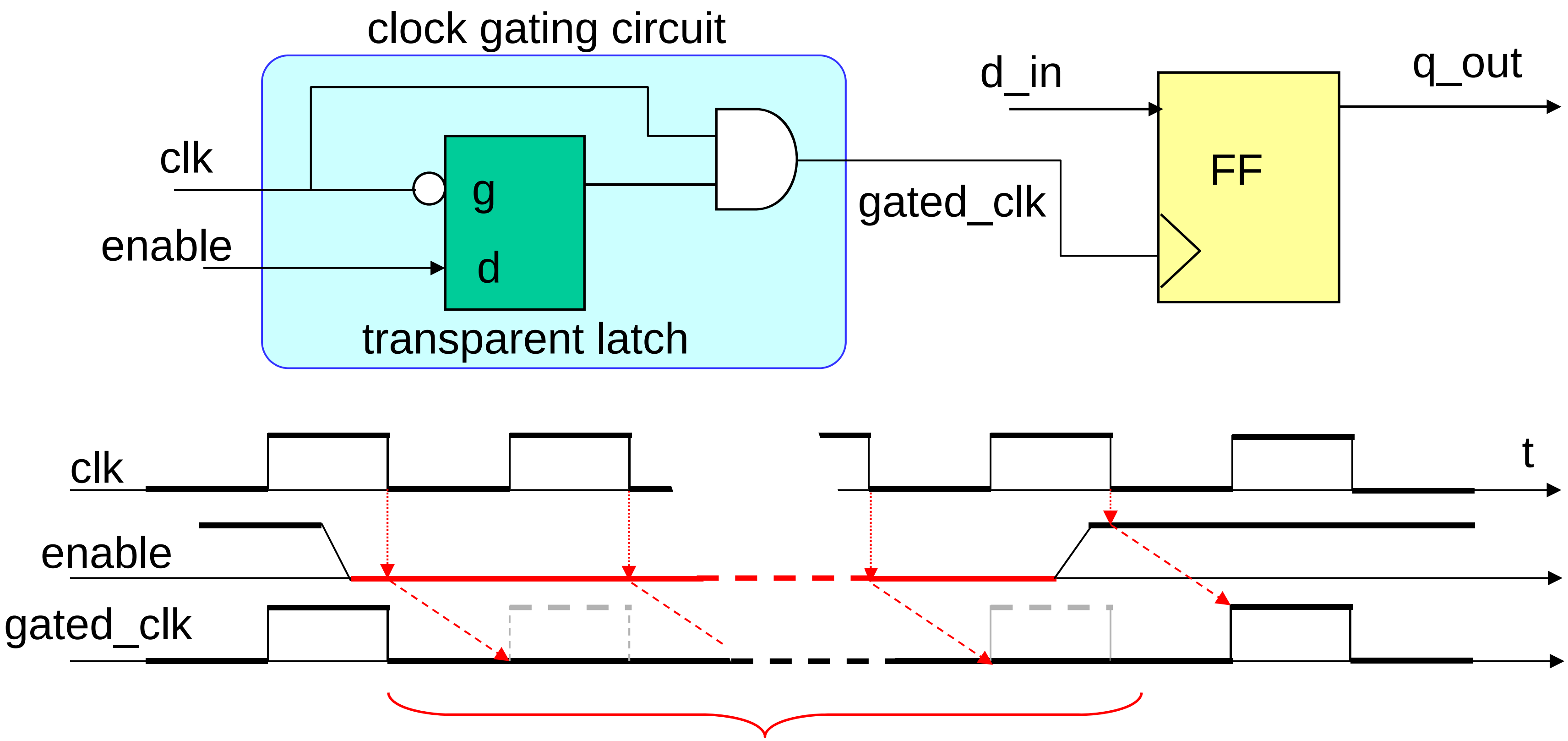
When g_clk is stopped, the value of y_out and q_out will not change. Therefore, there is no charging and discharging current in logic block AA, that is, there is no dynamic power consumption in AA.
Only leak current exists in it.

We have to control "gate control signal" to stop and activate g_clk.

6.1 gated clock and flip-flop

A commonly used clock gating circuit is shown in the following figure.

When enable=0, gated_clk is stopped and q_out will not change.

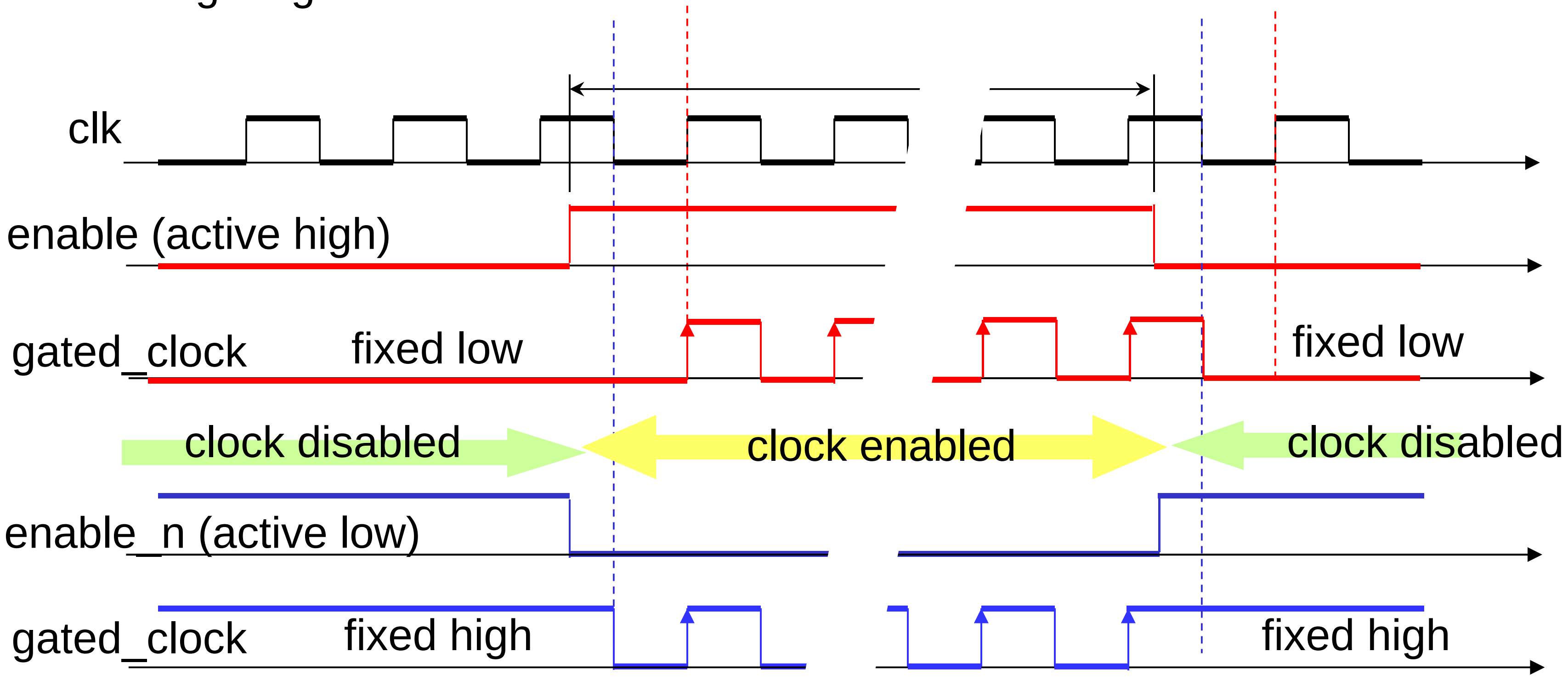


When enable signal = 0, gated_clk signal is kept zero to hold FF inactive.

In the previous page, clock signal is fixed to low while clock is disabled. It can be fixed high while clock is disabled.

In general, when enable signal is defined active high, then clock is fixed low while it is disabled. When enable signal is defined active low, then clock is fixed high while it is disabled.

Renesas's standard clock gating cell, CG cell, holds clock signal low while enable, active high, is negated. Some Renesas IPs are using the other type of clock gating.

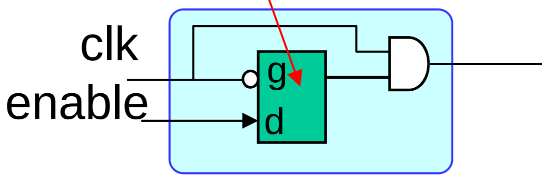


The clock gating circuit is designed by back-end designers. Front-end designers must not design its circuit.

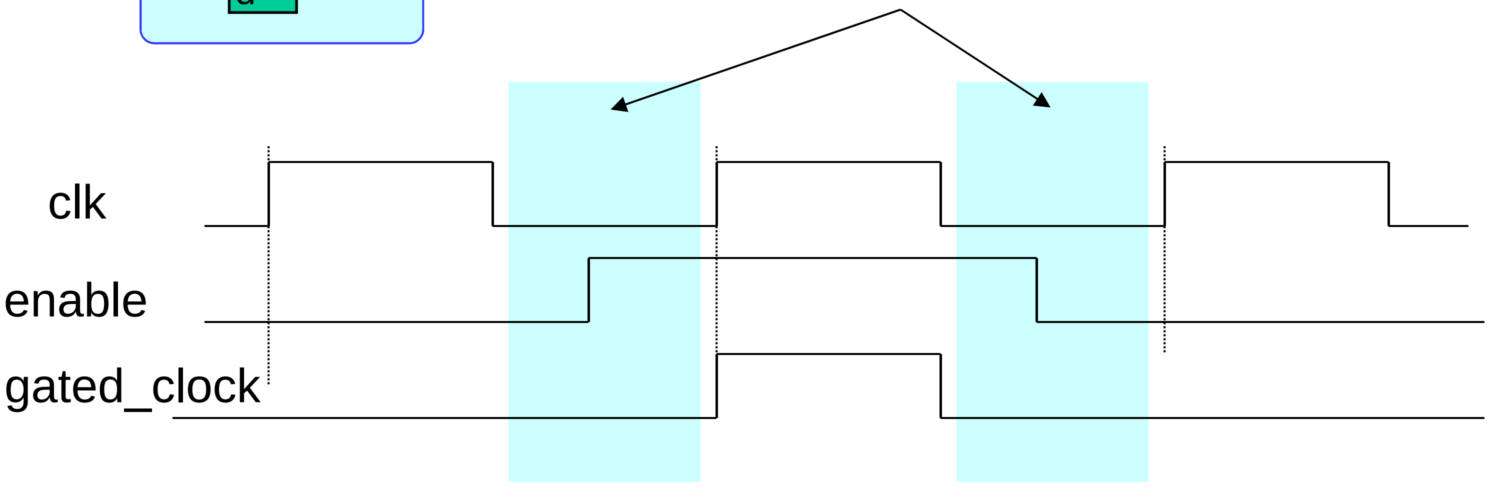
6.2 Clock gating cell

clock gating cell (1)

A transparent latch in a CG cell is to loosen the timing constraint for the enable signal and to ensure the minimum width for the gated clock pulse.



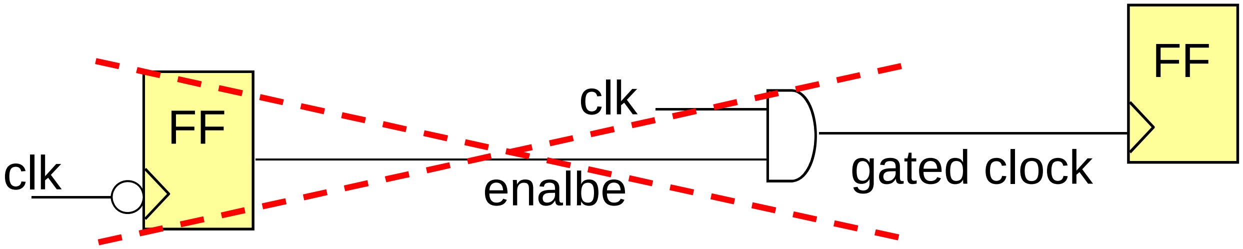
If an enable signal can be controlled to change only while the clock signal is low (blue background color part), then the latch is not needed in the CG cell.



To make enable signal change only when clock signal is low may cause difficulty in adjusting timing of enable signal. Therefore, the latch is used.

In case clock is slow and the enable signal is not in the critical path, FF using falling edge of the clock can be used for clock gating instead of latch as shown below.

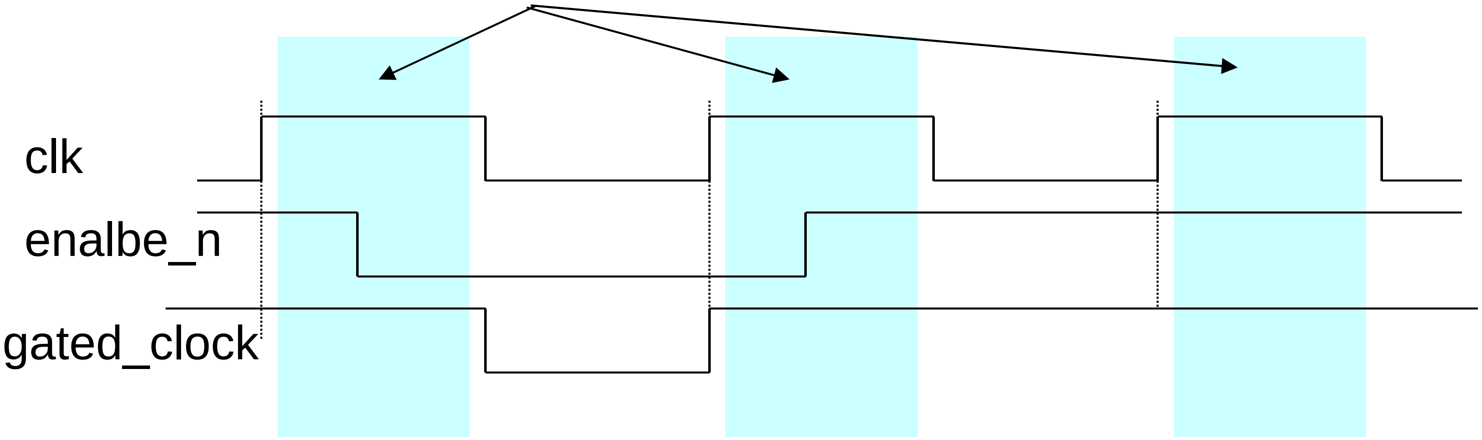
However, this is not recommended.



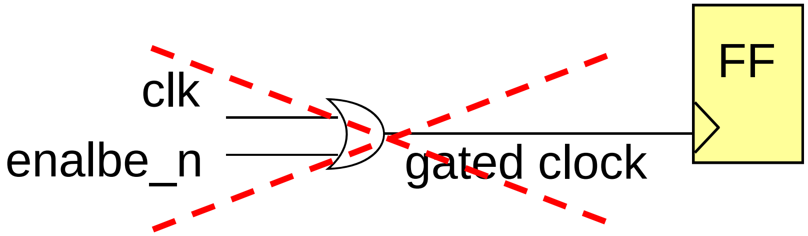
clock gating cell (2)

When the enable signal is defined as active low, the gated clock is fixed high while disabled.

No latch is needed if enable changes its value only while the clock signal is high (blue background color part).



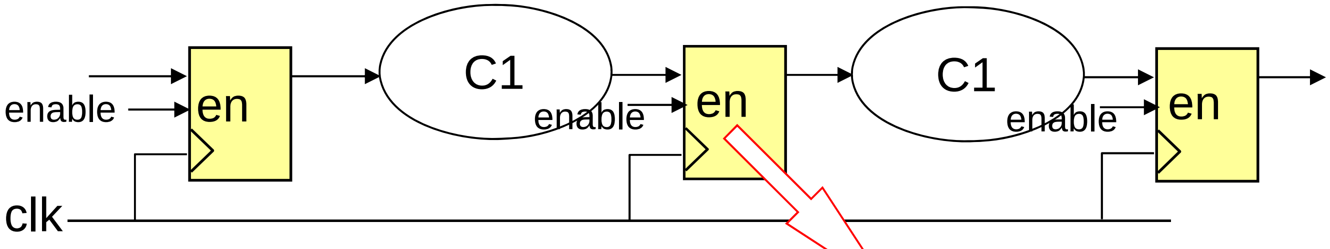
In case clock is slow and the enable signal is not in the critical path, a simple OR gate can be used for clock gating. Some conventional designs in Renesas use the circuit shown below, but it is not recommended any more.



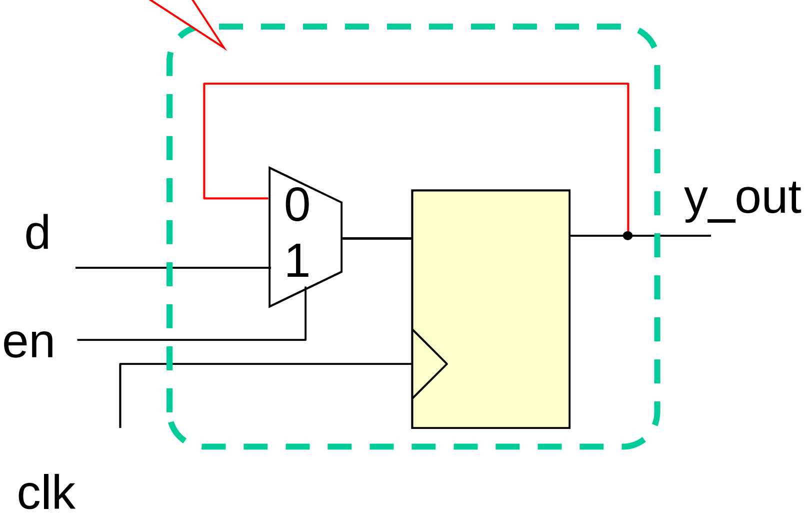
6.3 Power consumption of gated clock logic.

To evaluate power consumption in a gated clock system, compare the power consumption of gated clock system and self-loop system.

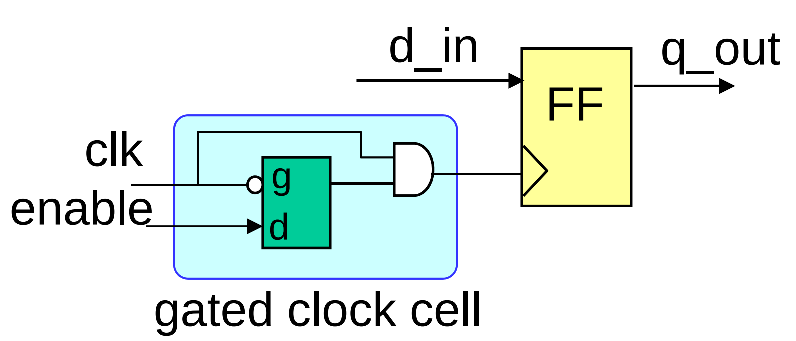
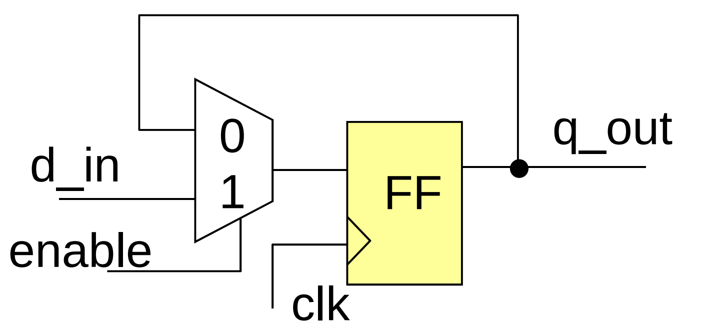
Self-loop system does not use a gated clock, but it uses the enable signal of FF. The output of FF with enable signal does not change if enable is 0 at clock rise time.



"enable" can be realized by a self-loop configuration shown on the right.
When en=0, y_out is given to the input of the FF.
Therefore, y_out will not change even if clk rises.



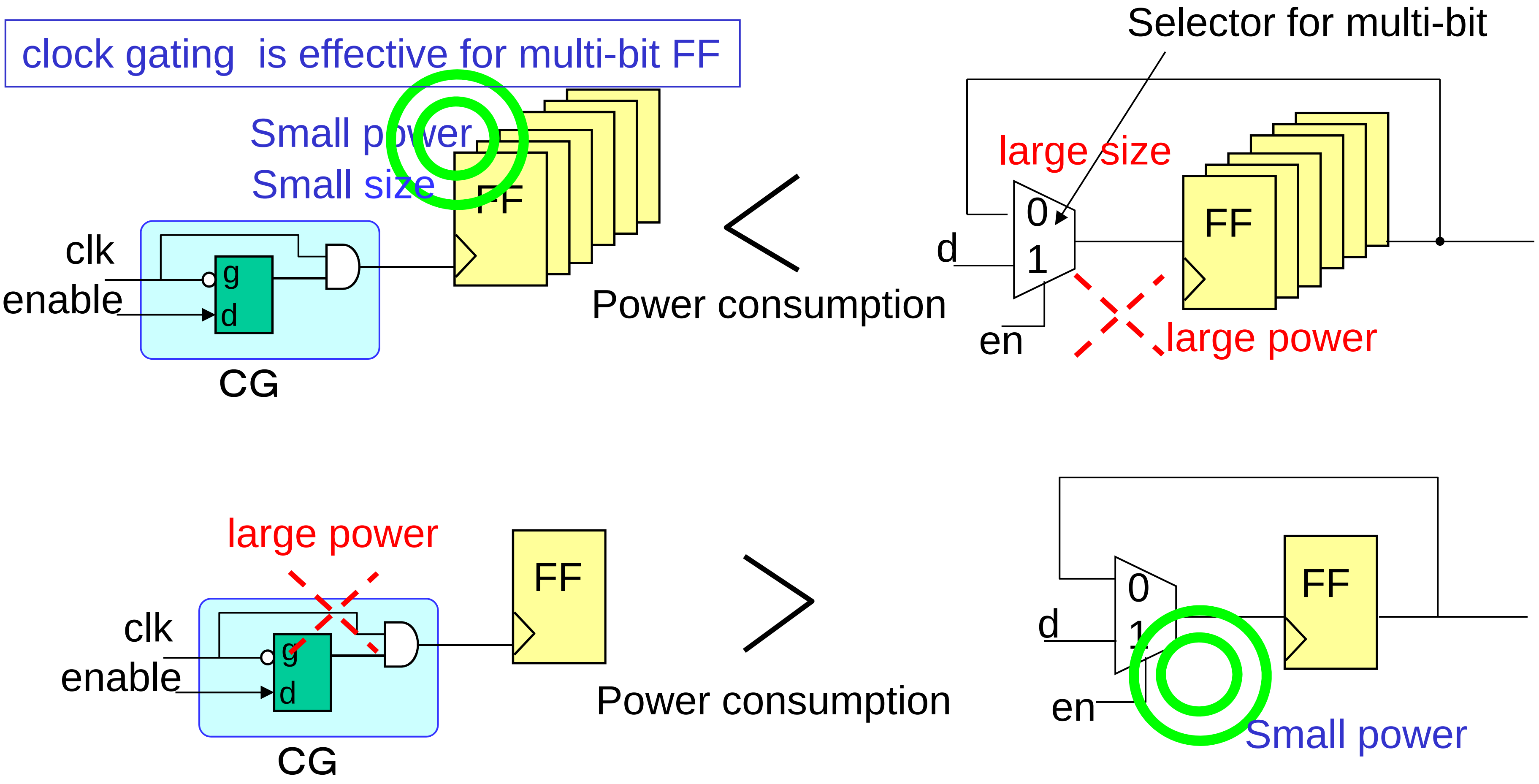
Comparing the two, gated clock and self-loop, it is known that gated clock can reduce power consumption much more than self-loop implementation.

	gated clock	self-loop
circuit	 d_in q_out FF clk enable g d gated clock cell	
power ＊1	0.8 (active ratio of enable:90%) < --- > 0.3 (10%)	1

＊1 : 90nm process generation, gated clock cell for 8-bit width FF, normalized to self-loop power consumption.

The above two are equivalent in function, however, their power consumption and area size are very different.
Power consumption aware synthesis tools can create gated clock logic from RTL code for FF with enable signal.

Which is better, gated clock or self-loop, is dependent on number of FFs which can be driven by the same gated clock as shown below.



clock gating is effective for multi-bit FF

Small power

Small size

Selector for multi-bit

large size

large power

Power consumption

large power

Power consumption

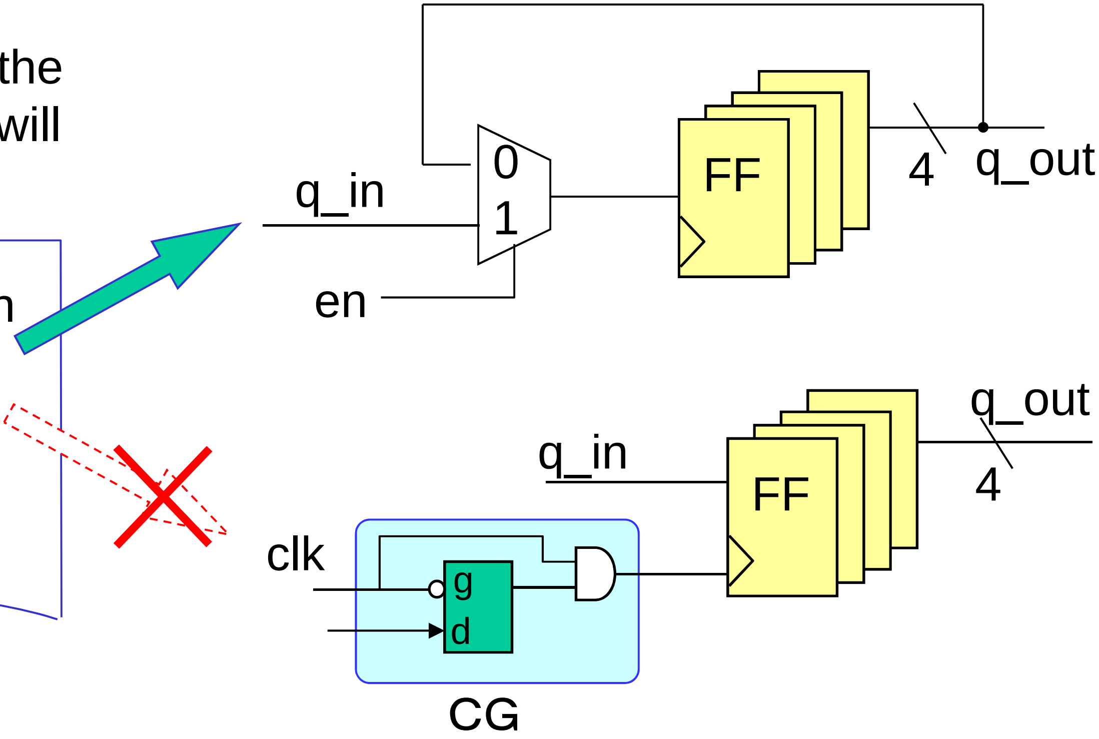
Small power

To get advantage of gated clock, Power Compiler has minimum limit for the number of FFs which are driven by the same gated clock. If it can not find enough FFs which can be driven by the same gated clock, it will not create a gated clock automatically for an enable signal. The number can be given to Power Compiler by “minimum_bitwidth” parameter.

If minimum_bitwidth is set to 8, the following RTL code for 4-bit FF will not create a gated clock.

```
always @ ( posedge clk ) begin
  if ( en ) begin
    q_out[3:0] <= q_in[3:0] ;
  end
end
```

RTL code for 4-bit FF with enable signal.

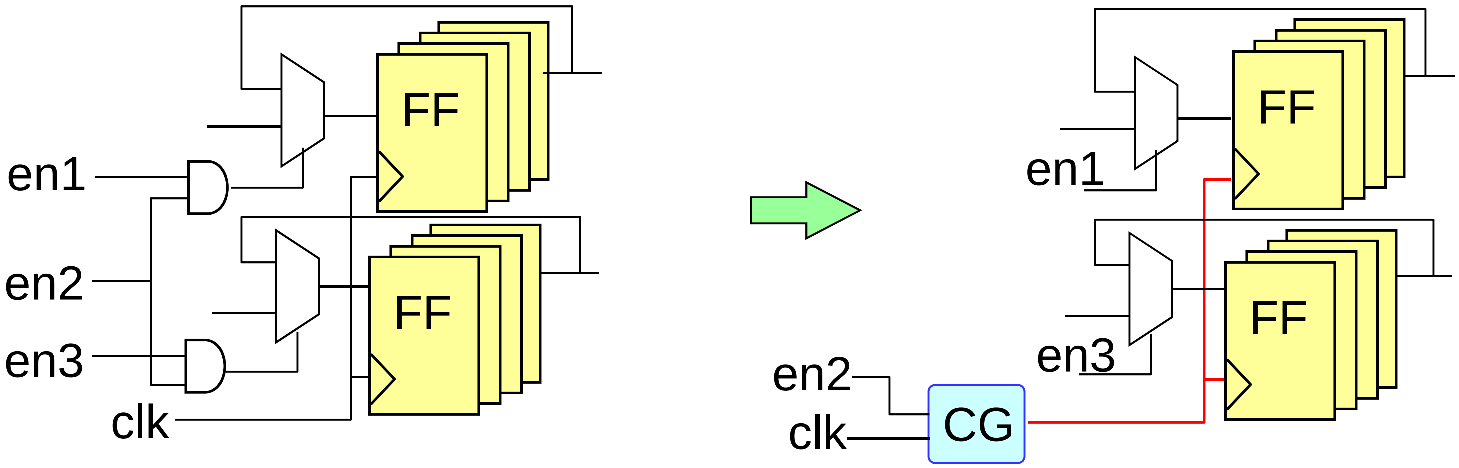


If minimum_bitwidth is set to 4, then the RTL code on the left will result in the above net list.

However, Power Compiler has an enhanced function to extract an enable signal which controls as many FFs as possible and it can create a gated clock for such enable signals as shown below.（Enhanced Clock Gating）

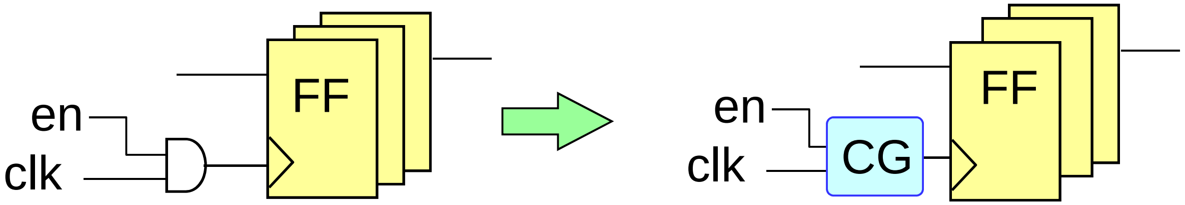
If minimum_bitwidth = 8, gated clock is not created for the logic shown on the left below because no enable signal is controlling 8 FFs, but only 4-bit FF is controlled by enable signals.

By using Enhanced Clock Gating, when “enhanced_minimum_bitwidth” is set to 8, then Power Compiler extracts en2 which can work as enable signal for 8-bit FF as shown on the right below and create a gated clock for en2 automatically.

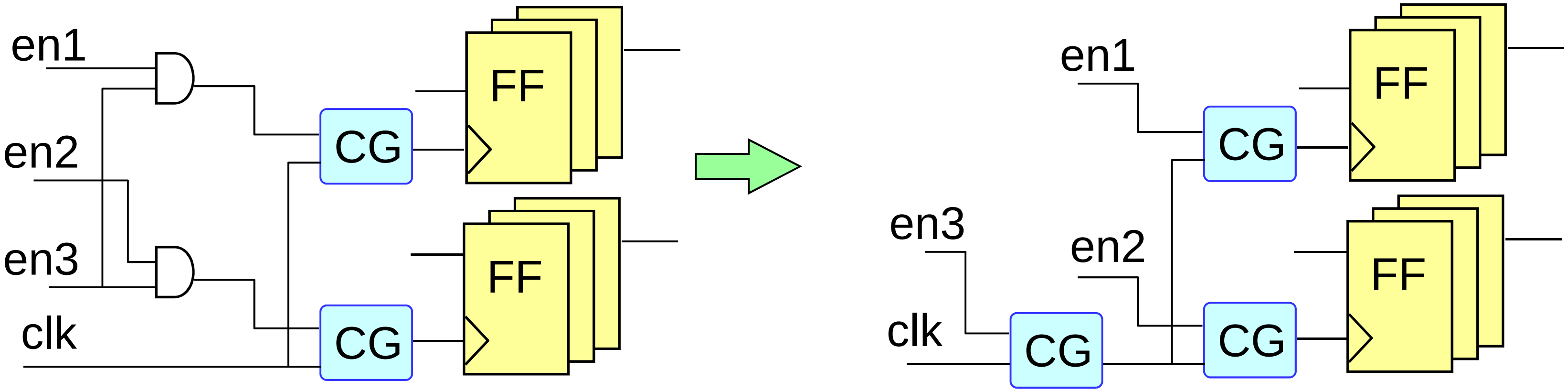


Enhanced Clock Gating

Power Compiler offers several functions such as replacing simple gate on clock line to CG cell: Module level Clock Gating and inserting CG cells in serial: Multi-Stage Clock Gating.



Module level Clock Gating



Multi-Stage Clock Gating

6.4 Clock gating and RTL code

Power Compiler, Synopsys tool, can create gated clock net list from RTL code for FF with enable signal. Several examples of such RTL code are shown below.

```
always @ ( posedge clk )
begin
  if ( enable )  q_out <= d_in ;
end
```

```
always @ ( posedge clk )
begin
  q_out <= ( enable )? d_in : q_out ;
end
```

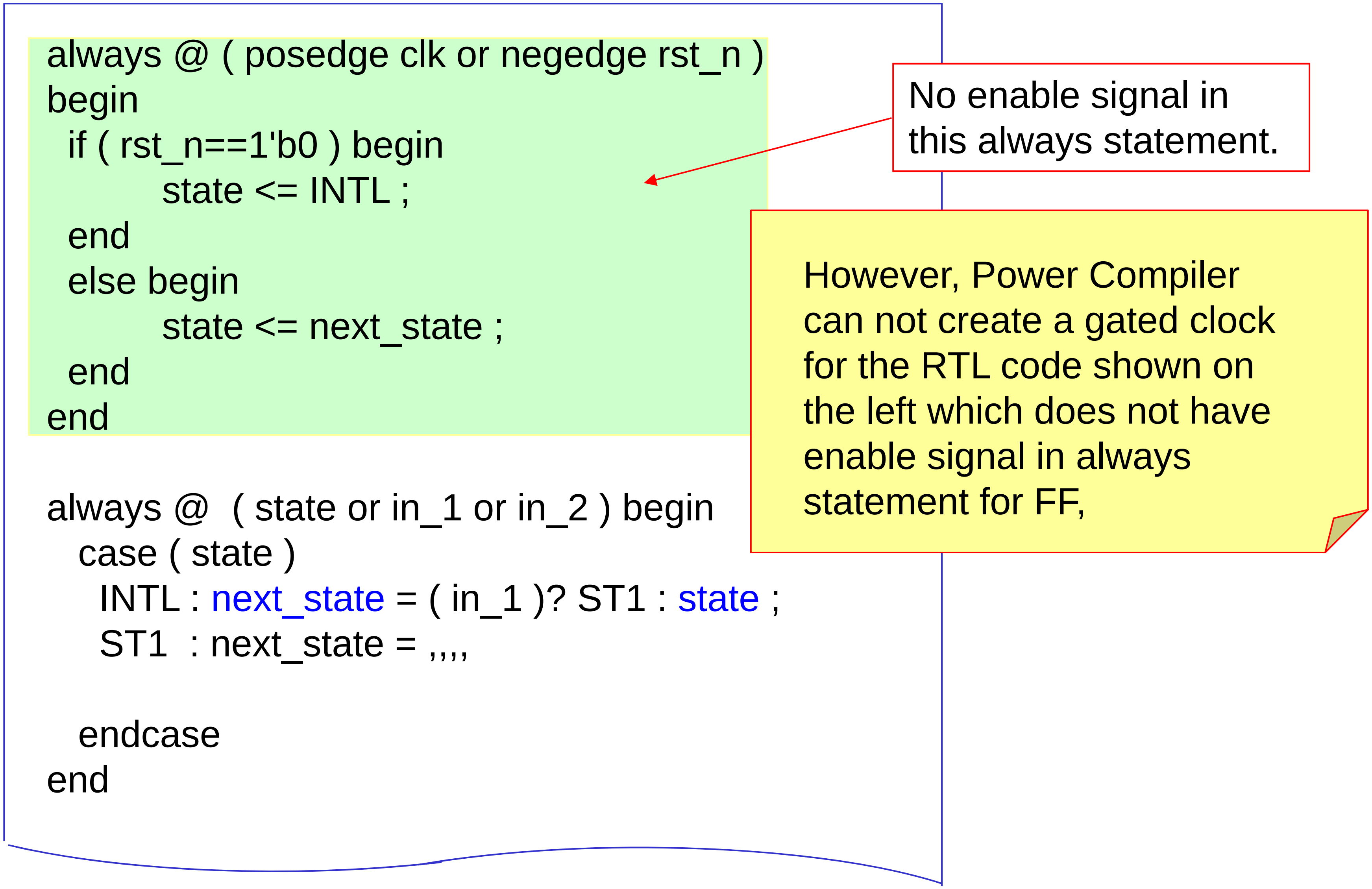
Power Compiler can create a gated clock from these RTL codes.

```
always @ ( posedge (
begin
   case ( enable )
     1'b1 :  q_out<= d_in ;
     1'b0 :  q_out <= q_out ;
   endcase
end
```

```
   ays @ ( posedge clk )
begin
  if ( enable )  q_out <= d_in ;
  else           q_out <= q_out ;
end
```

```
always @ ( posedge clk or negedge rst_n )
begin
  if ( rst_n==1'b0 ) begin
         q_out <= INTL ;
  end
  else begin
    if ( enable )  begin
          q_out <= d_in ;
    end
    else begin
          q_out <= q_out ;
    end
  end
end
```

Power Compiler can create a gated clock from this RTL code.



```
always @ ( posedge clk or negedge rst_n )
begin
  if ( rst_n==1'b0 ) begin
          state <= INTL ;
  end
  else begin
          state <= next_state ;
  end
end

always @  ( state or in_1 or in_2 ) begin
   case ( state )
     INTL : next_state = ( in_1 )? ST1 : state ;
     ST1  : next_state = ,,,,

   endcase
end
```

No enable signal in this always statement.

However, Power Compiler can not create a gated clock for the RTL code shown on the left which does not have enable signal in always statement for FF,

6.5 How to program clock gating.

When you want to control clock gating explicitly, use the common clock gating logic in a project. You must use RTL macro module designed for clock gating which shall be used throughout the project.

```
module my_logic ( clk,  ,,, ) ;
               ⋮
 clk_gate clk_gate_01 ( .clk(clk), .en(en)
               ⋮
always @ ( posedge gtd_clk or negedge r
begin
  if ( rst_n==1'b0 ) begin
           q_out <= INTL ;
  end
  else begin
           q_out <= d_in ;
  end
end            ⋮
```

Apply macro module commonly used in a project.

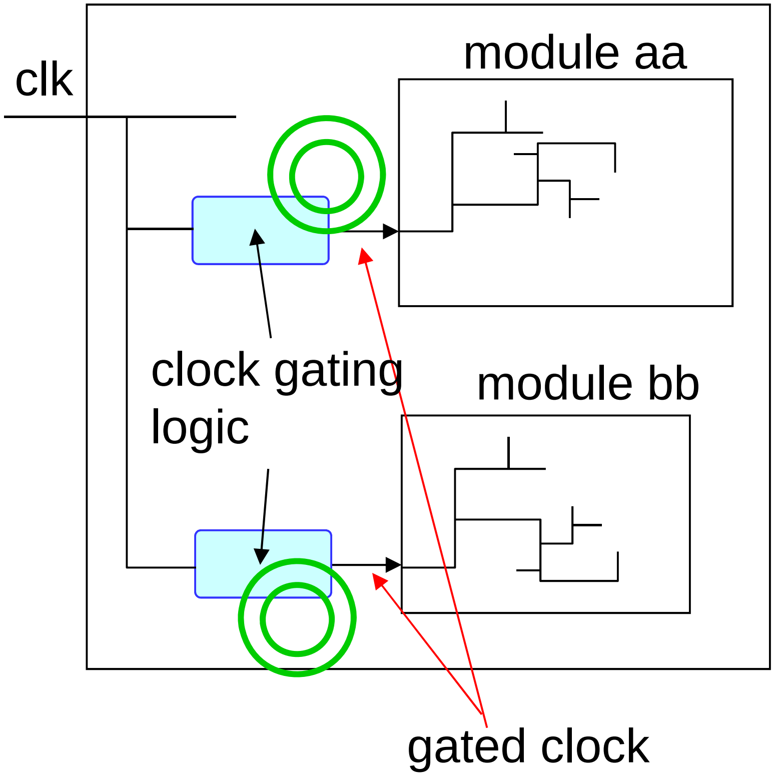
```
module clk_gate ( clk, en, gtd_clk ) ;
input clk, en ;
output gtd_clk ;
//
wire clk, en ;
wire gtd_clk ;
reg gate_sig ;  // latc
//
always @ ( clk or enable )  begin
   if ( clk == 1'b0 ) begin
         gate_sig <= enable ;
   end
end
assign gtd_clk = gate_sig & clk ;

endmodule
```

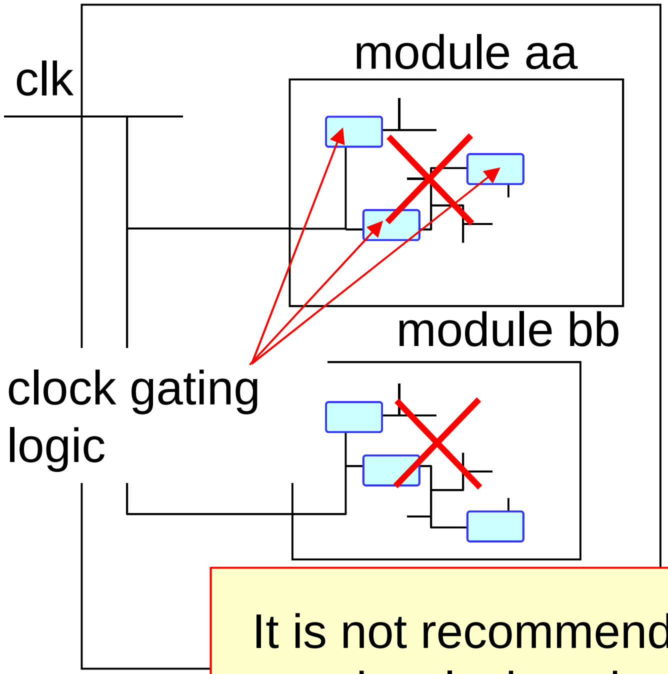
An example of macro module for clock gating.

However, it is not recommended to write clock gating instances scattered across the module.

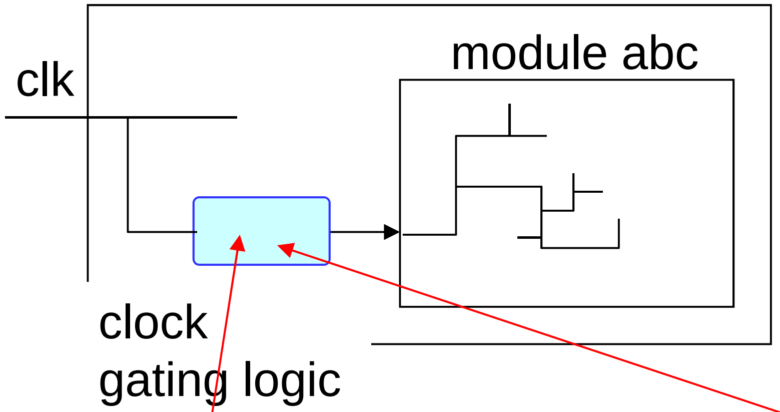
The recommended style is to apply the gated clock to a module as a whole, that is, to apply gated clock as a clock input to a module as shown on the left below.

Introducing RTL level clock gating





It is not recommended to write clock gating RTL code for each FF.



For the recommended style shown on the left, clock gating logic must be used as macro module which must be applied throughout the project.

This will make the mapping to the macro module to clock gating cell easy.

```
module clk_gate ( clk, en, gtd_clk ) ;
input clk, en ;
output gtd_clk ;

        •
        •
        •
        •
        •

assign gtd_clk = gate_

endmodule
```

Logic verification phase

Prepare this kind of macro module for clock gating and use it throughout the project.

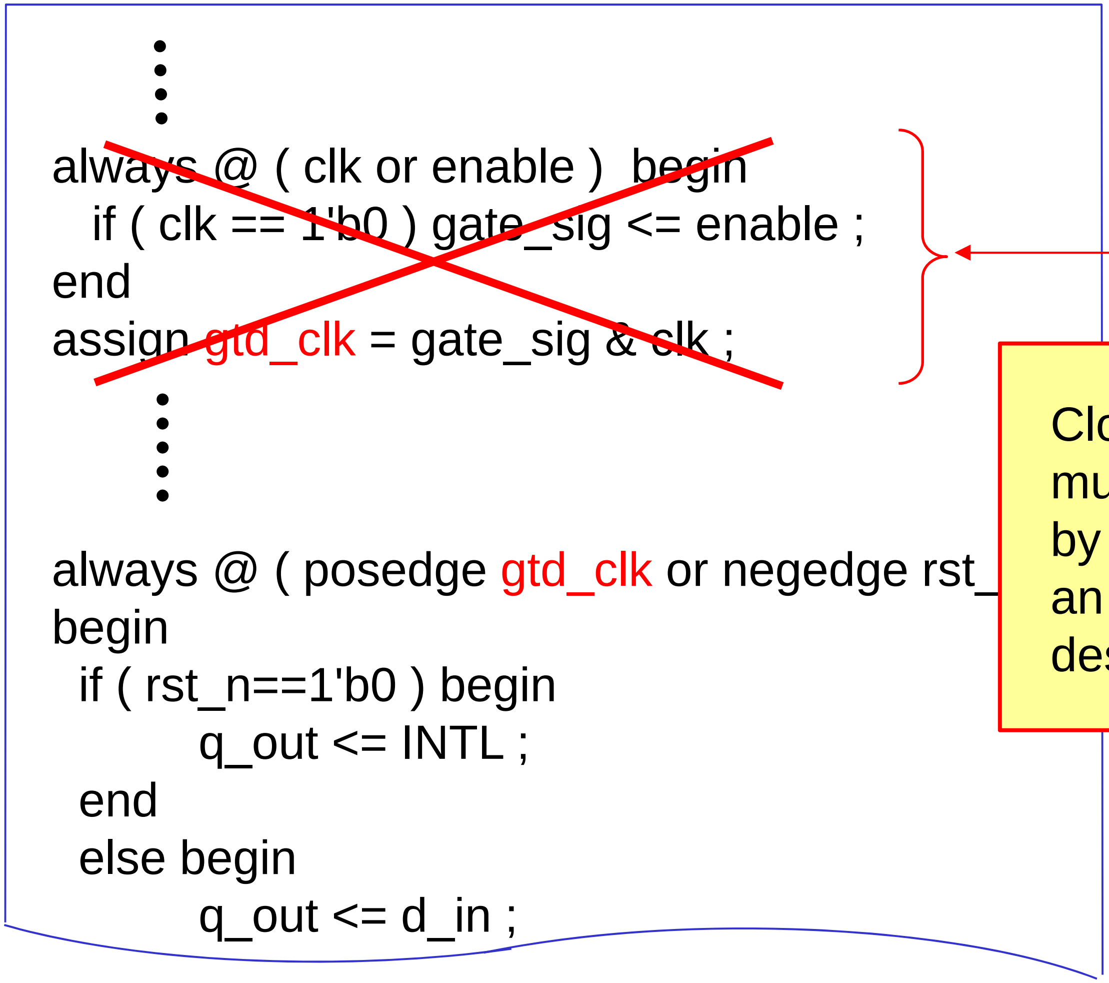
```
module clk_gate (clk, en, gtd_clk);
input clk, en ;
output gtd_clk ;

TM2GTDCELL CG
  ( .GCLK(gtd_clk),
    .CLK(clk),
    .CEN(en) );

endmodule
```

Synthesis phase

Use either one depending on development phase

This is the cell developed by back-end designers.

You must not write clock gating logic individually in modules as shown below.
It is important to use the same clock gating logic throughout the project.



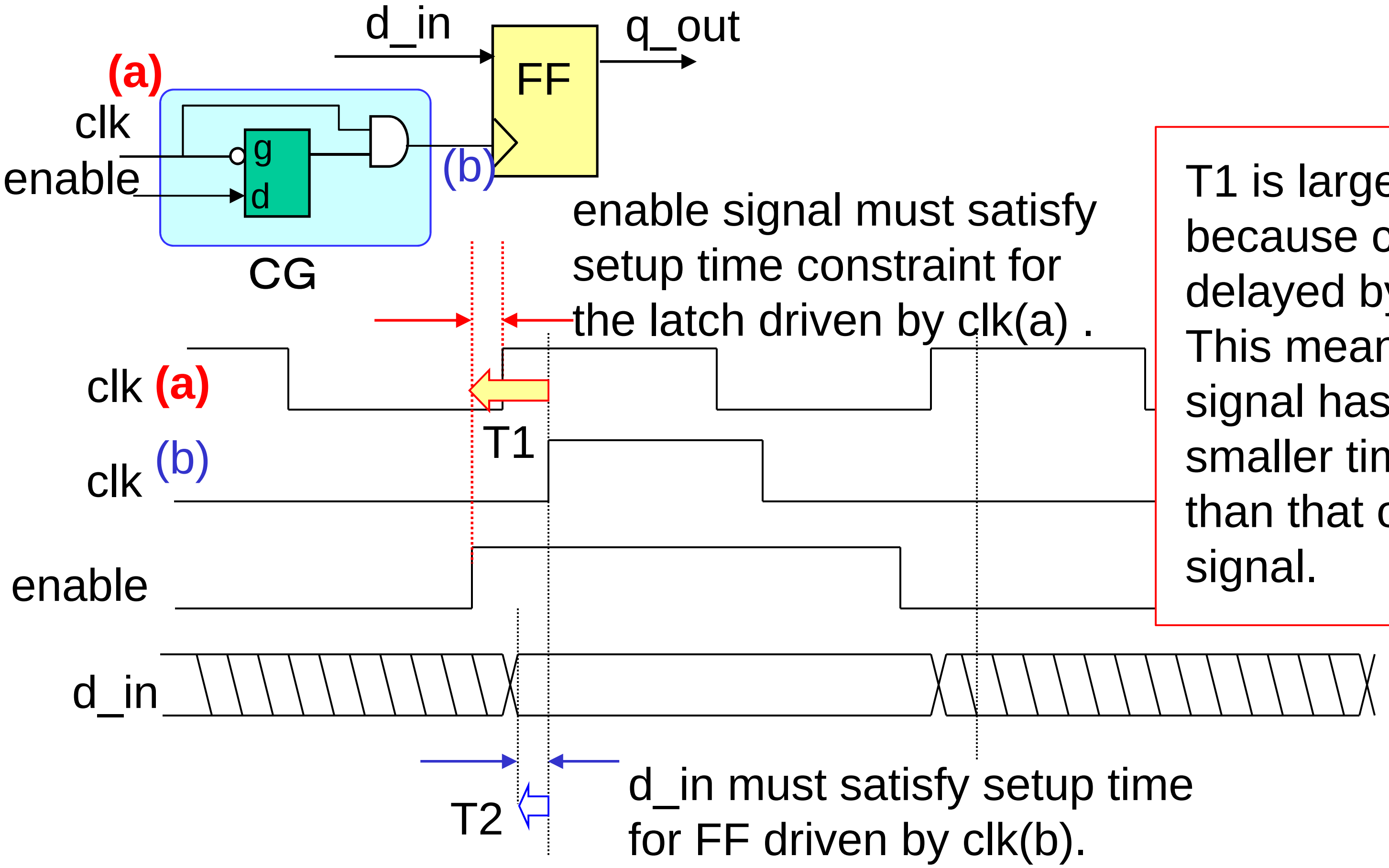
```
      ⋮
~~always @ ( clk or enable )  begin~~
~~   if ( clk == 1'b0 ) gate_sig <= enable ;~~
~~end~~
~~assign gtd_clk = gate_sig & clk ;~~
      ⋮

always @ ( posedge gtd_clk or negedge rst_
begin
  if ( rst_n==1'b0 ) begin
           q_out <= INTL ;
  end
  else begin
           q_out <= d_in ;
```

clock gating logic

Clock gating logic must be managed by a project, not by an individual designer.

6.6 Clock gating and timing issue

The enable signal for clock gating must arrive earlier than other data signals because it must satisfy setup time constraint of a latch in the CG cell. Therefore it must be noted that if the enable signal's timing is critical in non-gated clock logic, in a gated clock implementation the enable signal will become much more critical.

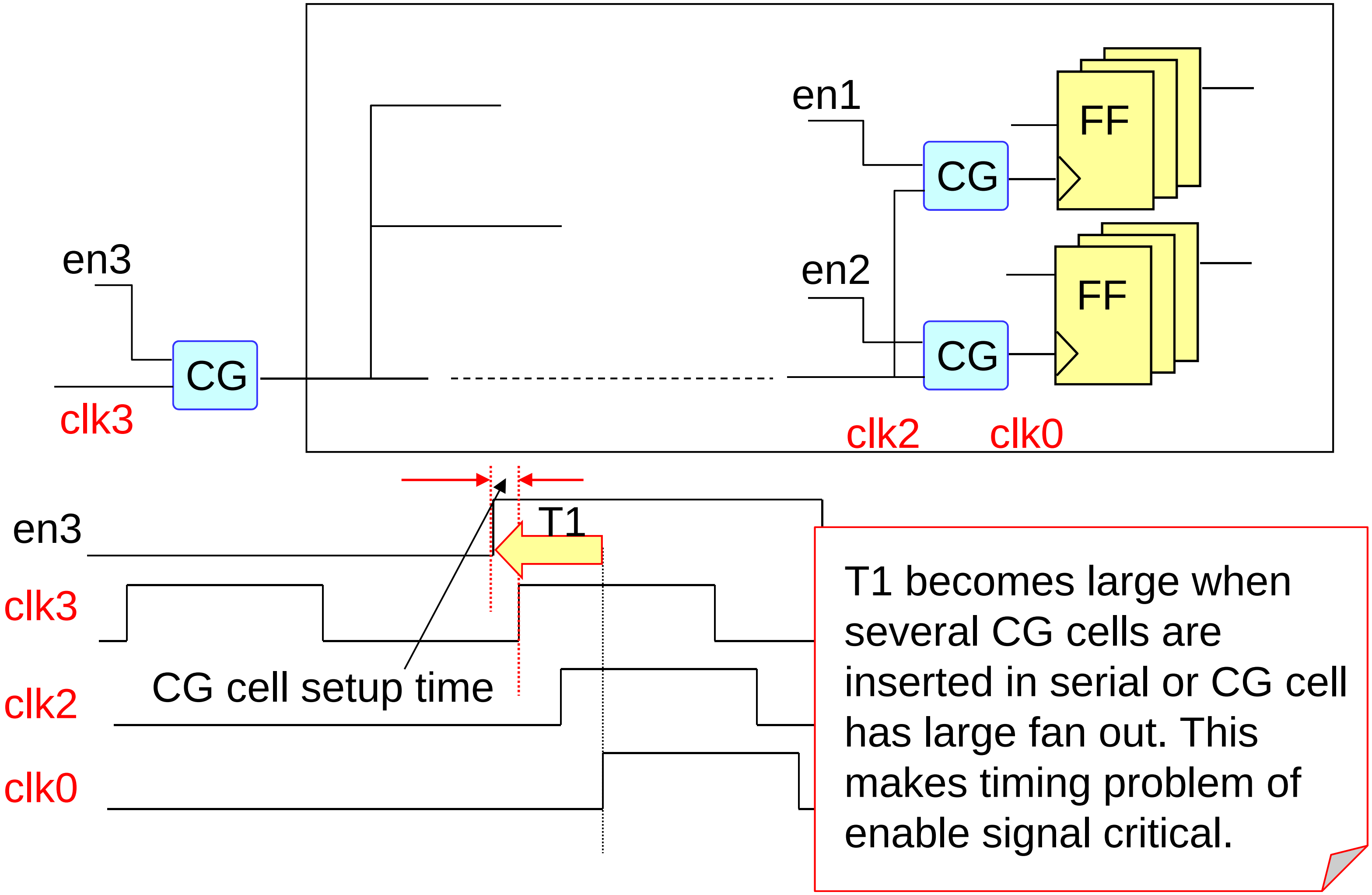


enable signal must satisfy setup time constraint for the latch driven by clk(a) .

T1 is larger than T2 because clk(b) is delayed by CG cell. This means enable signal has much smaller timing budget than that of d_in signal.

d_in must satisfy setup time for FF driven by clk(b).

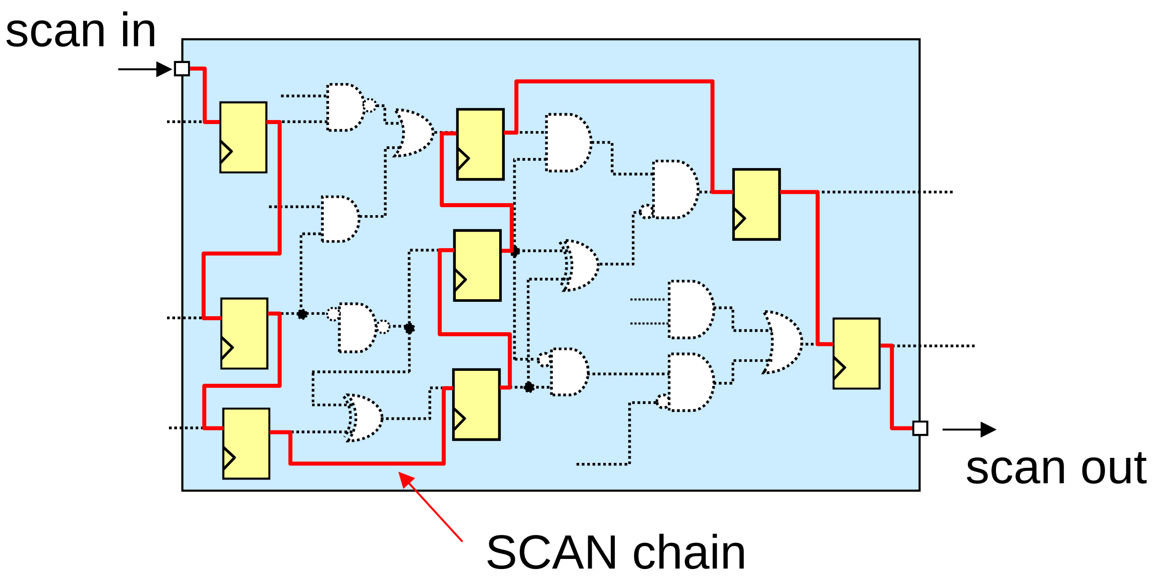
When applying module level clock gating or CG cell has large fan out, enable signal's setup time may become more critical as shown below.



T1 becomes large when several CG cells are inserted in serial or CG cell has large fan out. This makes timing problem of enable signal critical.

7. Design constraints with DFT

In deep-submicron process generation, DFT is mandatory to check out if there are any manufacturing faults.
To apply DFT, it is necessary that
(1) clock signal is always supplied during the test, and
(2) FF must not be initialized by reset signal during the test.



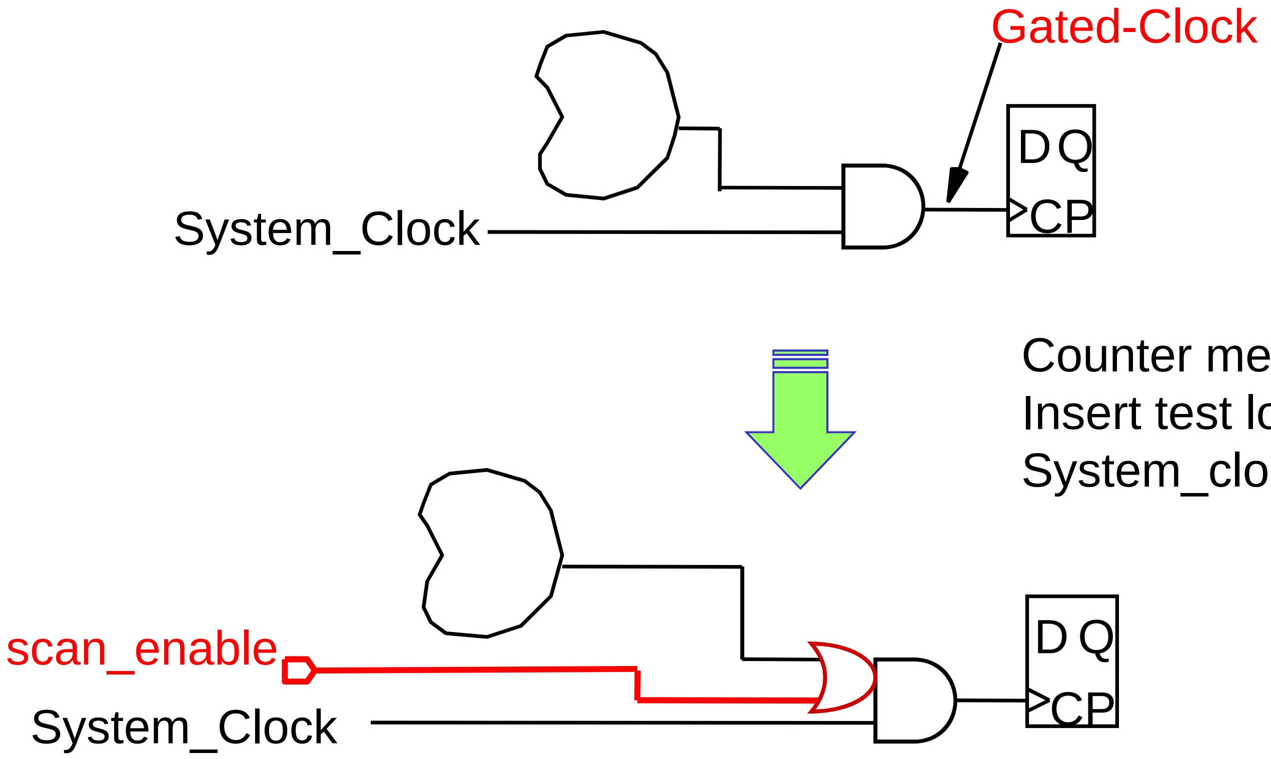
DFT can not work correctly if the scan chain breaks up.
SCAN data can not go beyond a FF to which no clock signal is coming.
SCAN data can be lost if a FF is initialized by reset signal.

RTL logic designers have to correct their logic to comply with these design constraints by hand.

7.1 Design constraints of gated clock

Clock signal to all the FFs must be directly controlled from outside the chip.

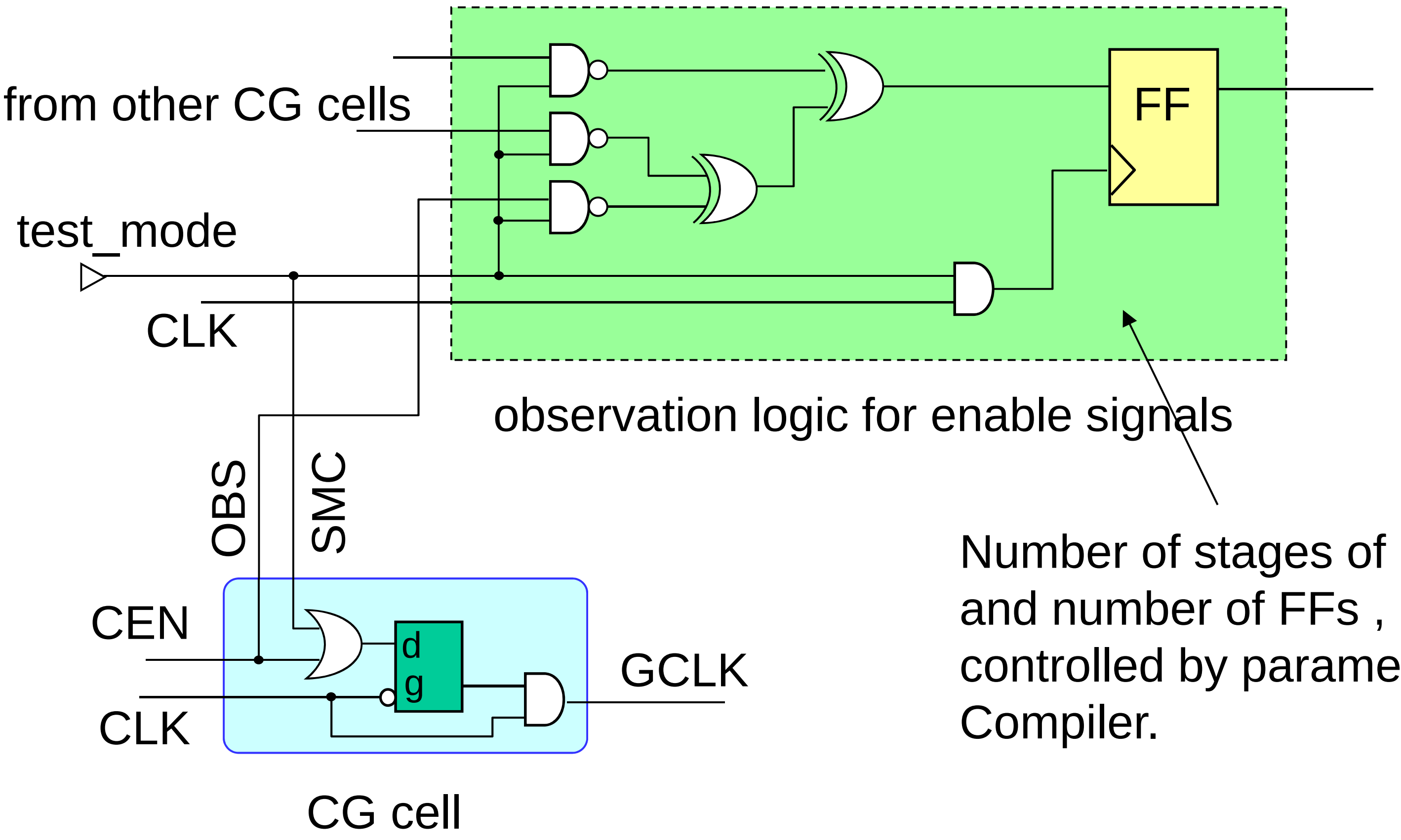
All clock inputs to FFs must be controlled directly to enable shift operation of the FFs. Those FFs using internal clock, which can not be controlled from an external pin of the chip can not be scanned.



Counter measure example;
Insert test logic to enable
System_clock input to the FF.

A cell developed for clock gating, CG cell, is ready for DFT and has an observation port to check enable signal and a test input port to let the clock signal go through the cell when test_mode is asserted.

Power Compiler can create a net list with CG cell from RTL code having FF with enable signal. It automatically converts enable signal into gated clock and also automatically creates observation logic for enable signals.



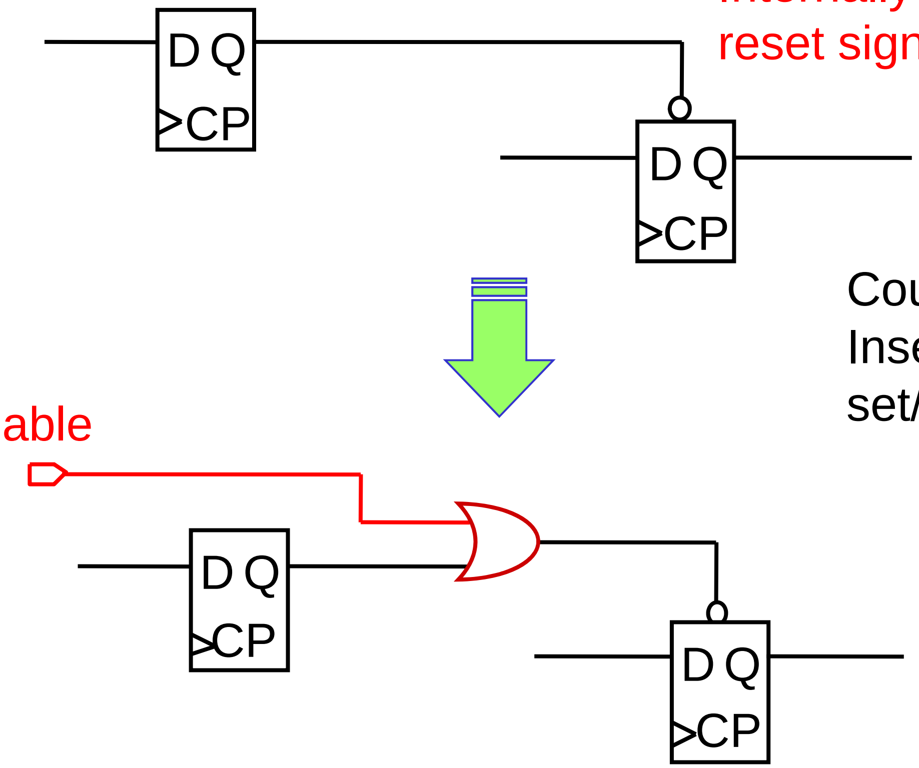
Number of stages of EOR gates and number of FFs , etc. can be controlled by parameter for Power Compiler.

When a designer uses gated clock explicitly, the observation logic for enable signals can not be automatically created for such clock gating logic.

Therefore, when using a clock gating macro, such a macro must be designed for testability and some observation logic must be coded by the designer.

7.2 Design constraints of reset

Set/reset signals of all FFs must be controlled from an external pin, or no set/reset signal applied while in scan operation.



Internally created reset signal

Counter measure example;
Insert test logic to disable set/reset signal for FFs.

scan_enable

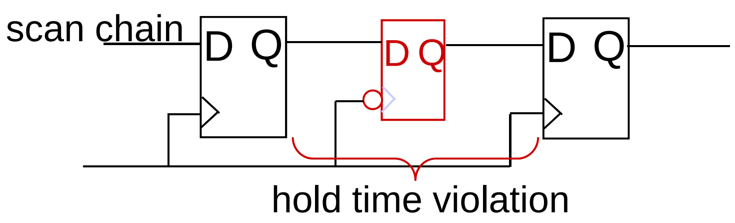
scan_enable : This signal is set on while doing scan operation.
test_mode : This signal is supposed to be 1 throughout test time.

7.3 Other design constraints

No.	Design constraints	Violation example	Level of Countermeasure	Affects
1	No feedback loop in combinational logic allowed.	Existence of the feedback loop in a combinational logic.	Avoid feedback loop or insert dummy latch in the loop.	None or degrade of coverage .
			None	Pattern generation impossible
2	Only edge trigger type FFs are usable.	FFs such as MUX cannot be added or level sensitive latches are used.	Use only allowed FFs.	None
			Handle such FFs as a black box.	Degrade of coverage
			None	Scan not applicable
3	All clocks to FFs must be directly controllable from input pin(s) of the chip.	Gated clock used, PLL is used for the clock, or asynchronous logic used.	Let the clock go into FF when testing.	Increase of test pins.
			Handle such FFs as a black box.	Degrade of coverage
			None	Scan not applicable
4	Set/reset signal of FFs can be controlled directly from input pin(s) of the chip or no set/reset operation while in test.	Reset/set signal is created by an internal logic.	Disable the logic while testing.	Increase of test pins.
			Handle FFs having such set/reset as a black box.	Degrade of coverage
			None	Scan not applicable

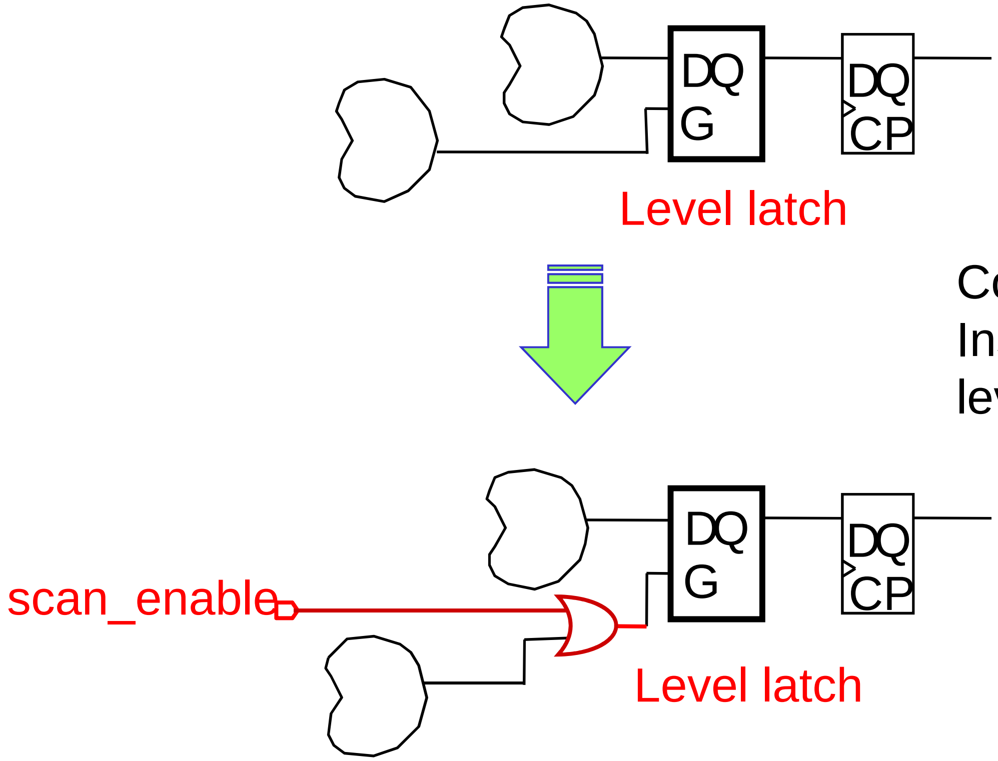
No.	Design constraints	Violation example	Level of Countermeasure	Affects
5	No hold time violation allowed while Scan_Shift operation.	Larger clock skew than FF's delay time.	Improve clock skew, or insert lock-up latch. *1	None
			None	Cannot be tested.
6	No hold time violation allowed for all paths while testing.	Hold time violation in false path.	Remove hold time violation in false path.	None
			None	Cannot be tested.
7	Internal bi-directional bus must be active only in one direction while testing.	Internal bus is controlled by FFs.	Activate only one direction while testing.	Degrade coverage
			None	Cannot be tested.
8	No black box allowed.	Existence of a black box module.	Insert FFs at input and output terminal.	None
			None	Degrade coverage

*1: Lock-up latch



Level latch and design constraints

If the target logic has FFs other than edge trigger type FF, it can not be modeled as a combinational logic. This results in poor testability.
Usable FFs are limited to those FFs which can operate properly with MUX inserted.



Counter measure example;
Insert test logic to make level latch through.

8. Sequential logic and its description

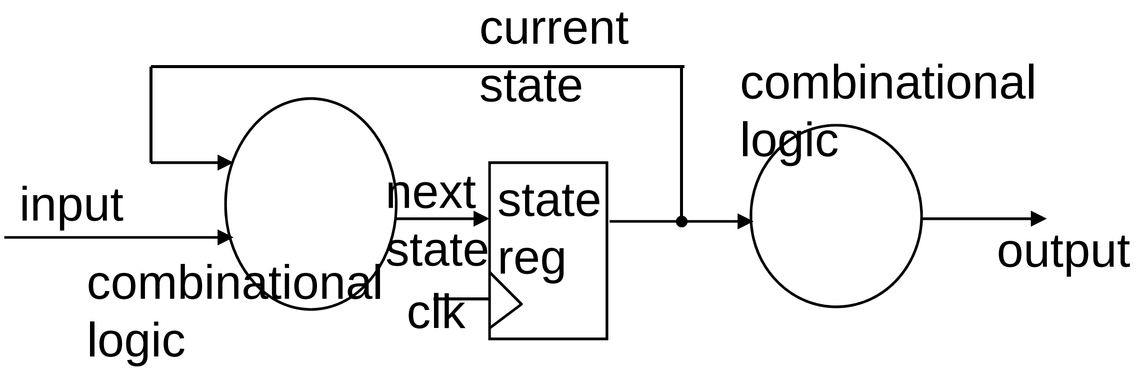
The output of sequential logic is not determined by the current inputs only. The output may vary depending on what condition, state, it is in.

And this is a significant characteristic of sequential logic. It is also called a “state machine”.

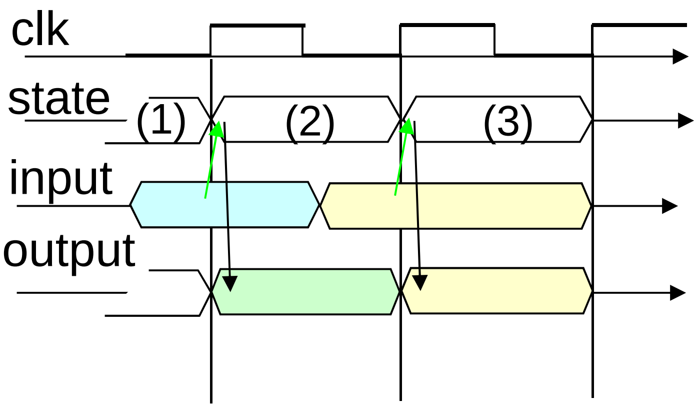
Almost all the logic we have to develop in actual work is sequential logic. To develop sequential logic, we need additional ability such as defining state so that total logic structure becomes beautiful.

8.1 Typical structure of sequential logic

<1> Moore type

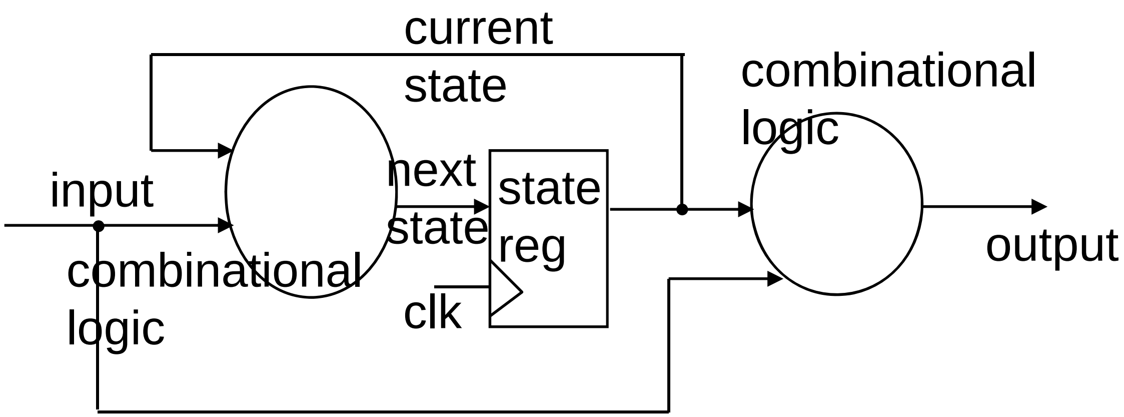


Generally Moore type needs more numbers of state than Mealy type, but it is recommended to use because it has no direct path from input to output. This makes STA simple.

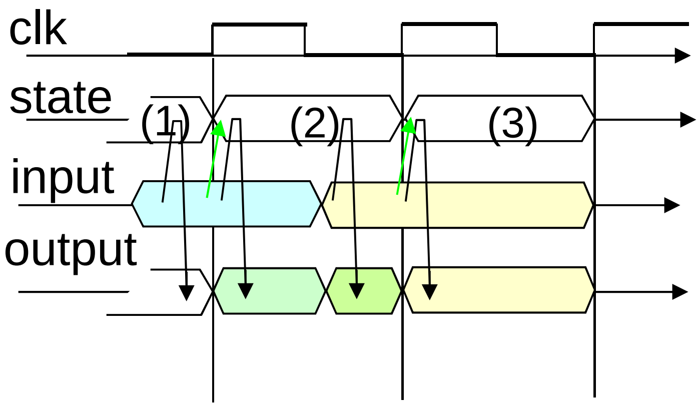


Output is determined only by the state.

<2> Mealy type



Generally Mealy type needs fewer states, but it is not recommended to use because it has a direct path from input to output. This makes STA difficult.



8.2 Sequential logic and its description

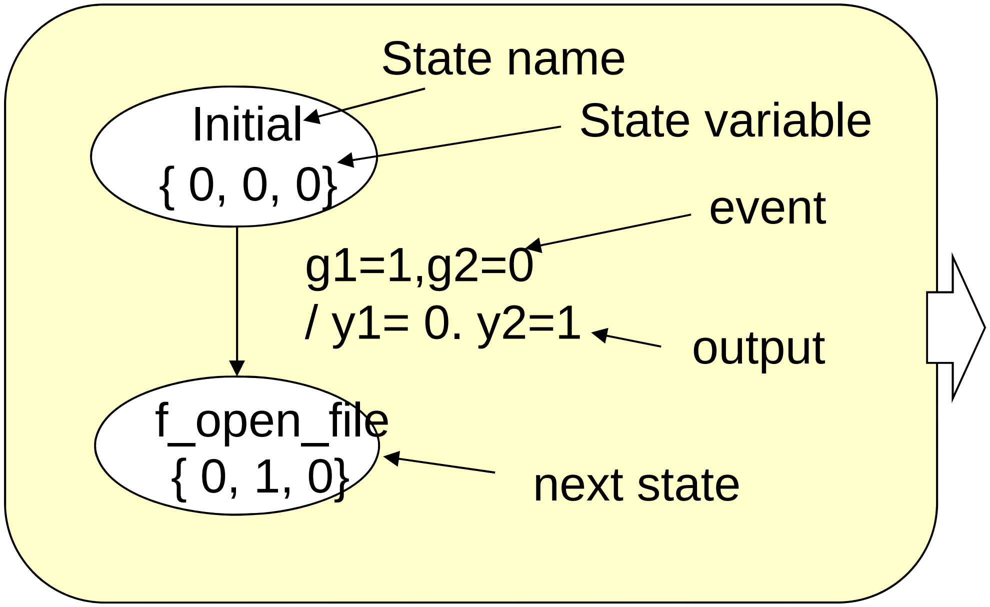
(1) Truth table for sequential logic → Also called "decision table"

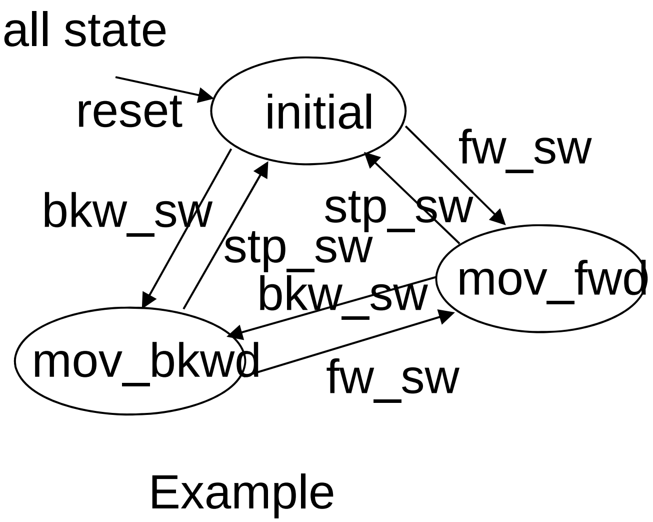
inputs variables			current state			next state			output variables	
g1	g2	g3	a	b	c	a	b	c	y1	y2
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	0	0	1
1	0	0	0	1	0	0	0	0	1	0
0	1	0	0	0	1	1	0	1	0	1
0	1	1	1	0	0	0	1	1	1	0

The same inputs get different result depending on the state.

This is the most significant characteristic of sequential logic.

(2) State transition diagram





State transition diagram is not suitable for checking the completeness of the logic. It is strongly advised to use a state transition table on the next page.

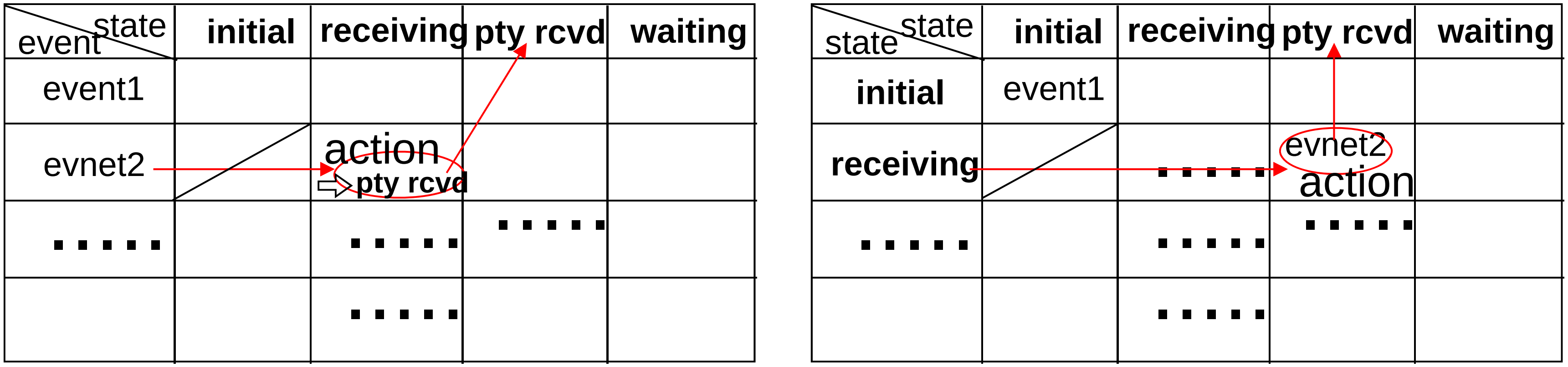
Use expression "x"as an event for no input such as automatic state transition with clock.

(3) State transition table (or matrix)

State transition matrix sometime tells much more than a state transition diagram.

Sometimes the matrix of the following form makes you notice a critical design error.

There are two types of state transition table as shown below. The one on the left is used in many cases.



event \ state	initial	receiving	pty rcvd	waiting
event1				
evnet2		action ⇨pty rcvd		
▪▪▪▪▪		▪▪▪▪▪	▪▪▪▪▪	
		▪▪▪▪▪		

state \ state	initial	receiving	pty rcvd	waiting
initial	event1			
receiving		▪▪▪▪▪	evnet2 action	
▪▪▪▪▪		▪▪▪▪▪	▪▪▪▪▪	
		▪▪▪▪▪		

receiving state moves to pty_rcvd state by event2

You must write the table if there are more than two states.

8.3 How to design sequential logic

Take the following steps to design sequential logic. You must understand the essentials of a system you are going to design.

Step 1: Analyze the total system and find what are the inputs and what are the outputs.

Step 2: Analyze the total system behavior and find its dependency on inputs and the operation history.

Step 3: Analyze the history dependency and find how many states are needed to specify the behavior of the system.

- Step 3-1: Define a state variable which has enough bit size to identify each state.
- Step 3-2: Assign a name and a value to the state variable for each state. (Each state must be identifiable by its name and value of the state variable.)

You must define an initial state. The system must be in the initial state right after power on.

Step 4: Draw a state transition table to make it clear how the system behave for certain inputs in each state.

⬆

Care must be taken how to bring the system into operational state from initial state. If several clock cycles are needed to bring up the system to operational state, several additional states can be introduced to identify such initializing steps.

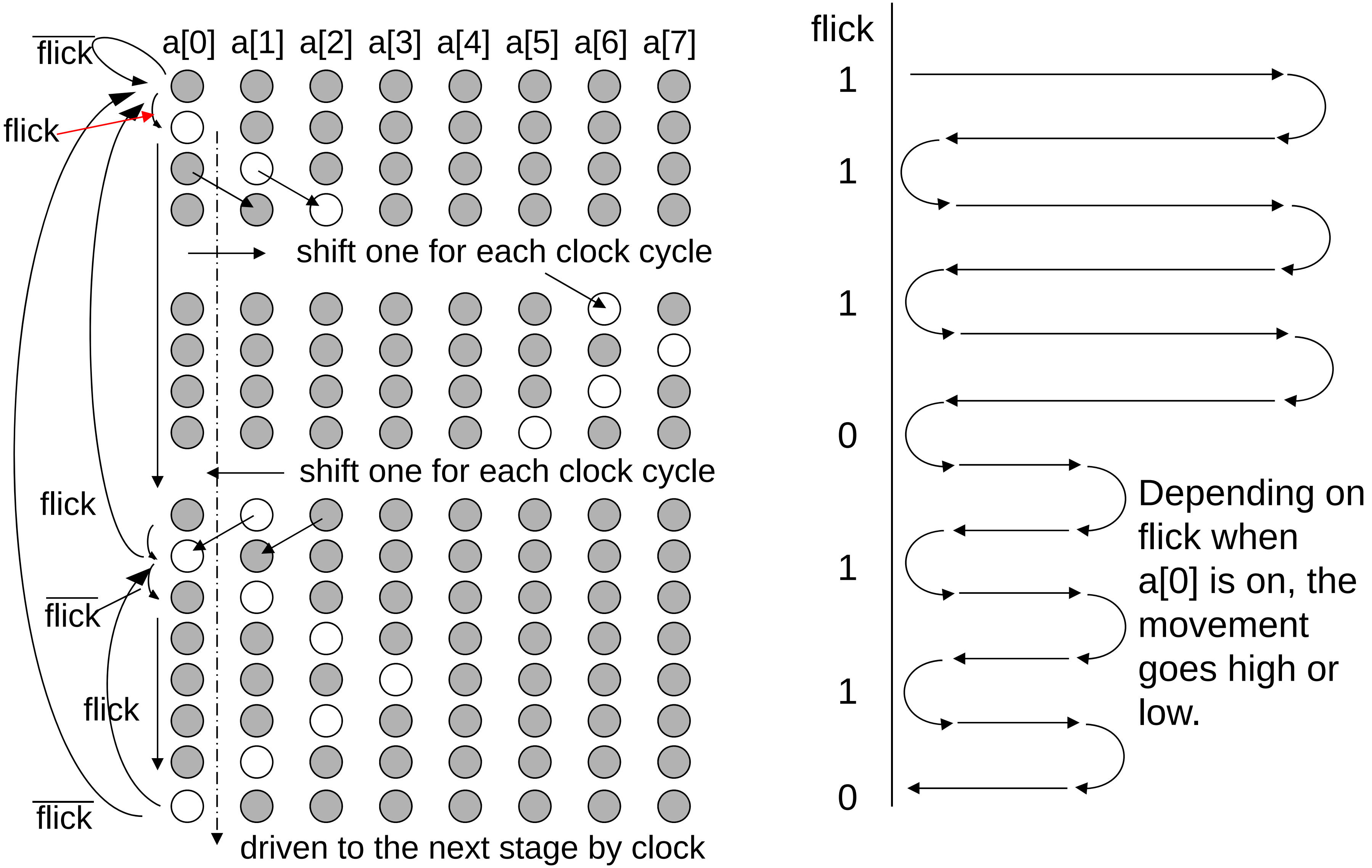
Step 5: Adjust the state boundary, precise condition where the state changes from one state to another, so that the logic in each state and the logic for state transition become simple. (Rewrite the state transition table if the state boundary is changed.)

Step 6: Write a Verilog RTL program by mapping the state transition table into RTL code.

9. Sequential logic example, Bound Flasher

9.1 Specification

There are 8 lamps, a[0] to a[7], and they flash in sequence as below.

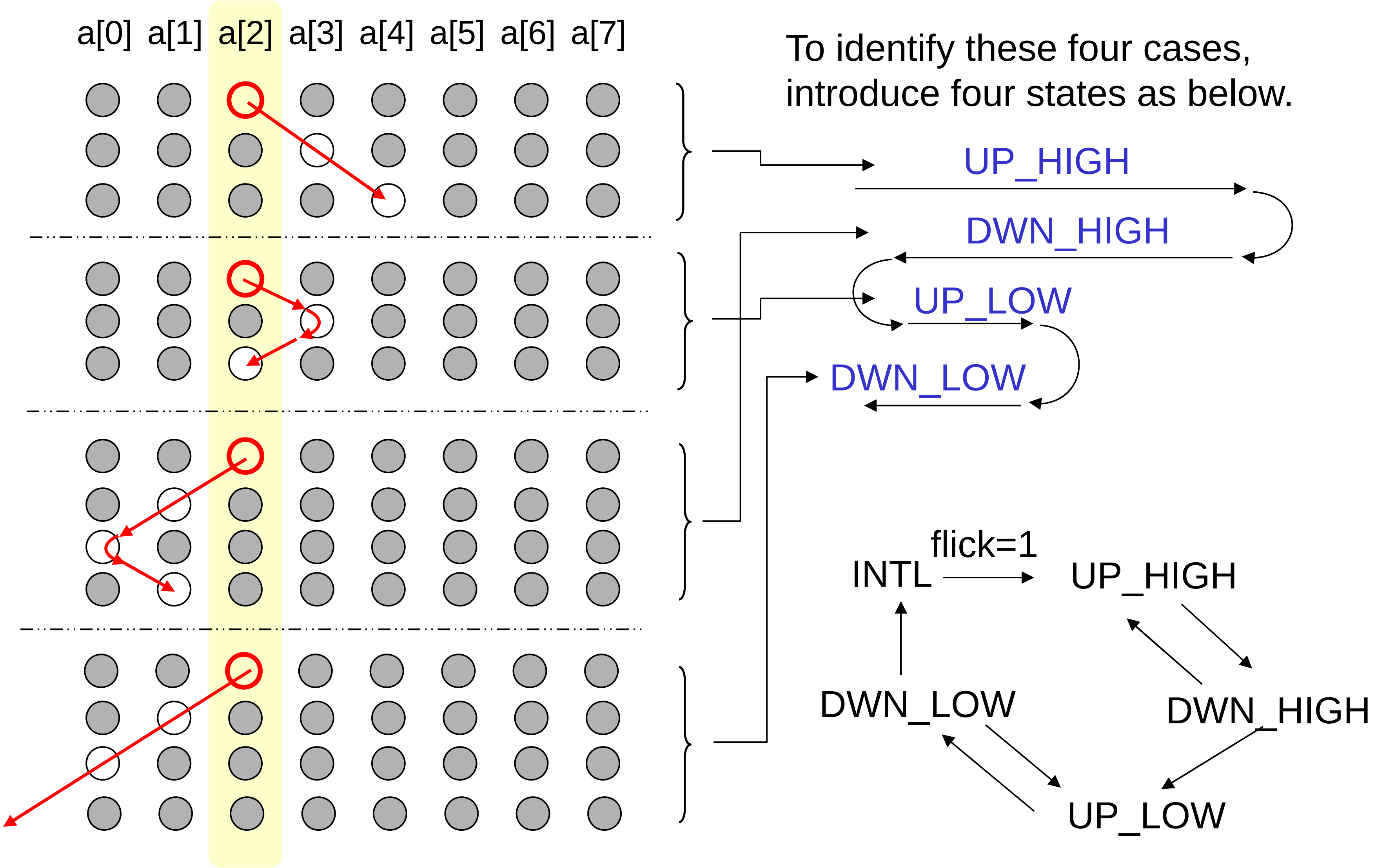

$\overline{\text{flick}}$
flick
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]
shift one for each clock cycle
shift one for each clock cycle
flick
$\overline{\text{flick}}$
flick
$\overline{\text{flick}}$
driven to the next stage by clock
flick
1
1
1
0
1
1
0

Depending on flick when a[0] is on, the movement goes high or low.

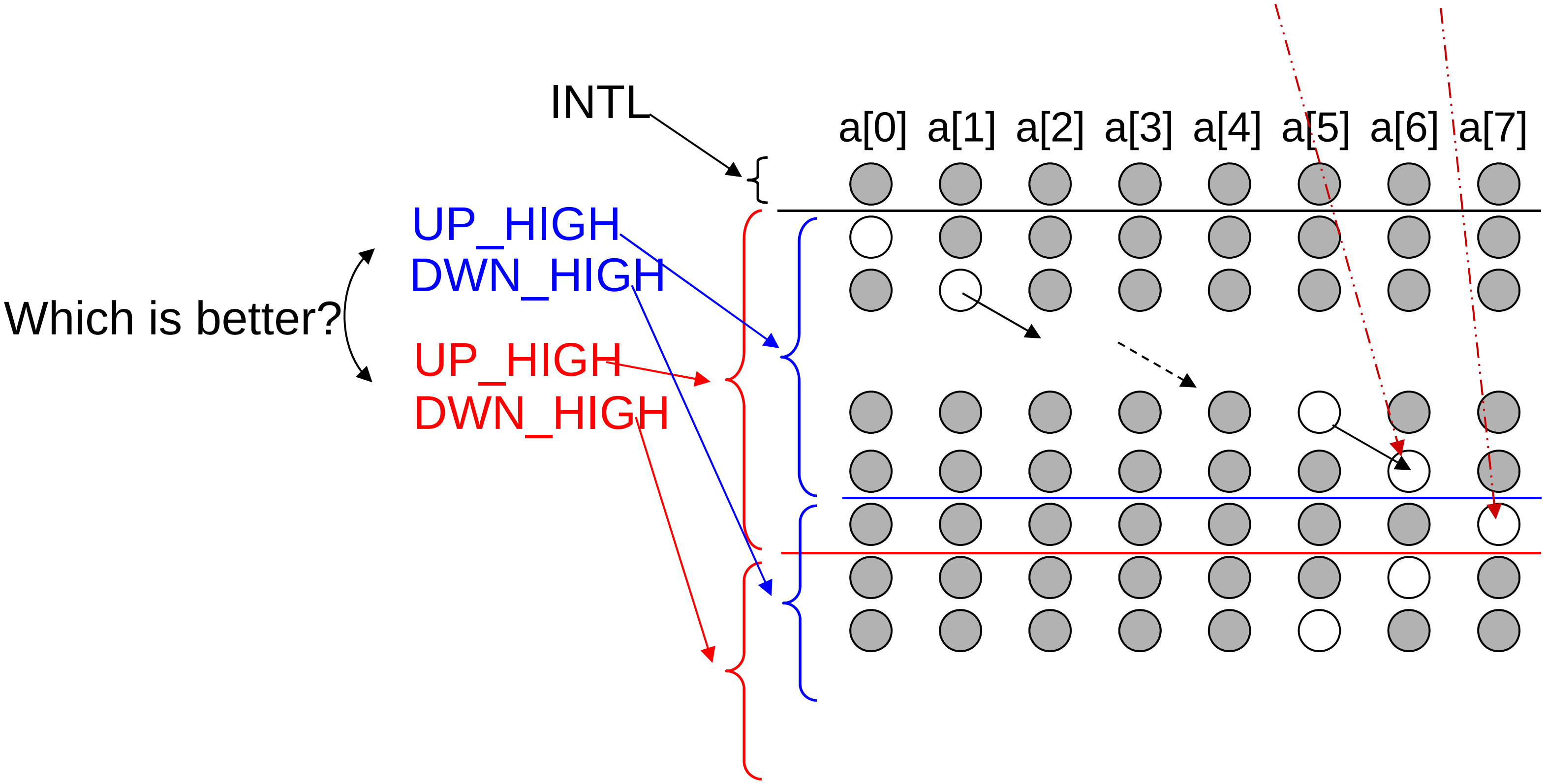
9.2 How to define state

Suppose lamp a[2] is on and flick is off, what will happen as the clock beats? There are at least four cases as shown below.

To identify these four cases, introduce four states as below.

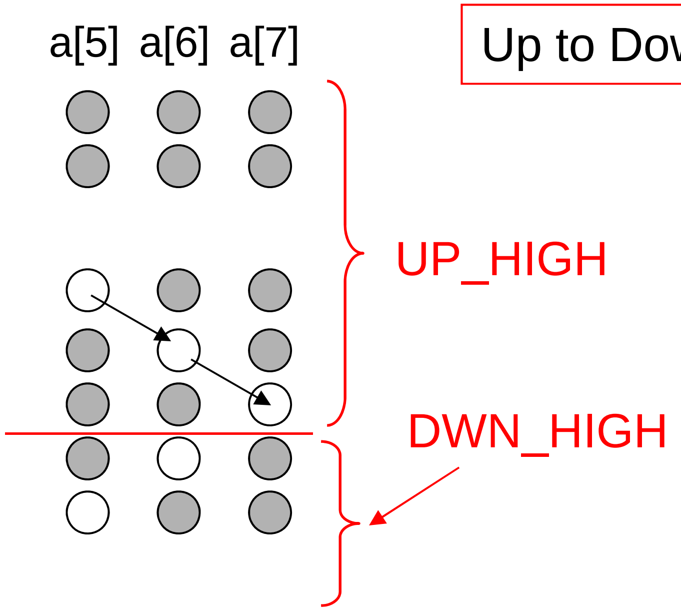


Now a question may arise; when shall the state change. For example, UP_HIGH to DWN_HIGH change may take place at either a[6] or a[7].



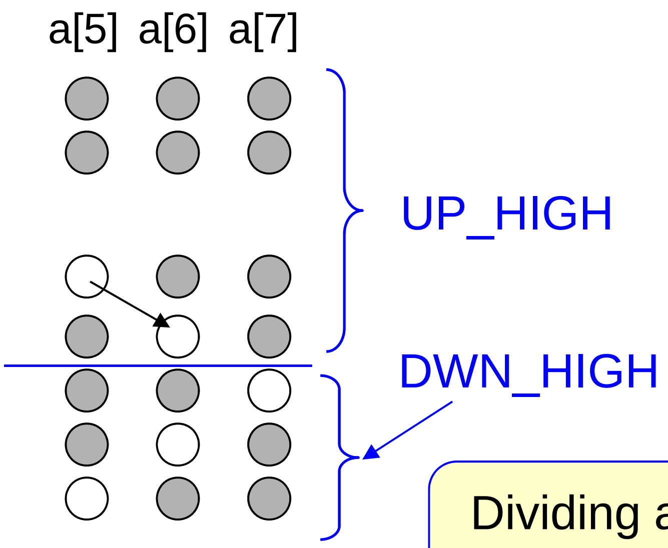
Judging point is "Operations done in the same state shall have uniformity or consistency."

Up to Down change



shift a one bit to up direction.
However, if a[7] is on, shift a
in reverse direction.

There is an exception.

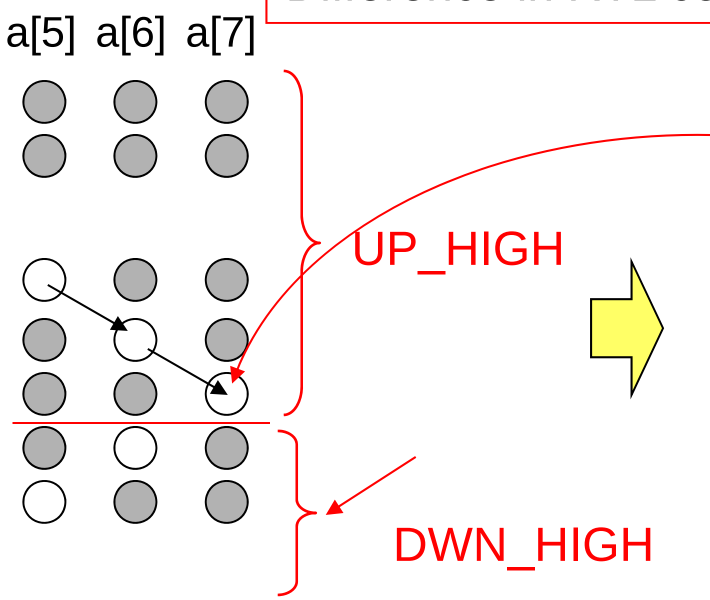


shift a one bit to up direction.

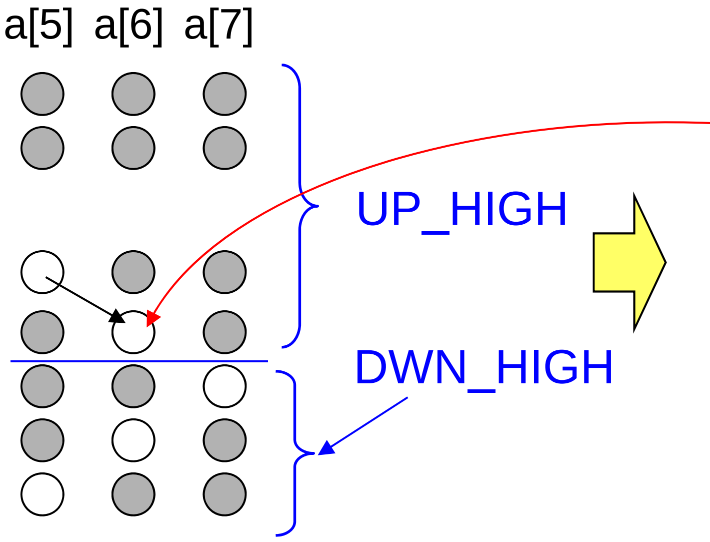
There is no exception.

Dividing at a[6] is better. With the same reason, UP_LOW to DWN_LOW shall be divided at a[2], not at a[3].

Difference in RTL code

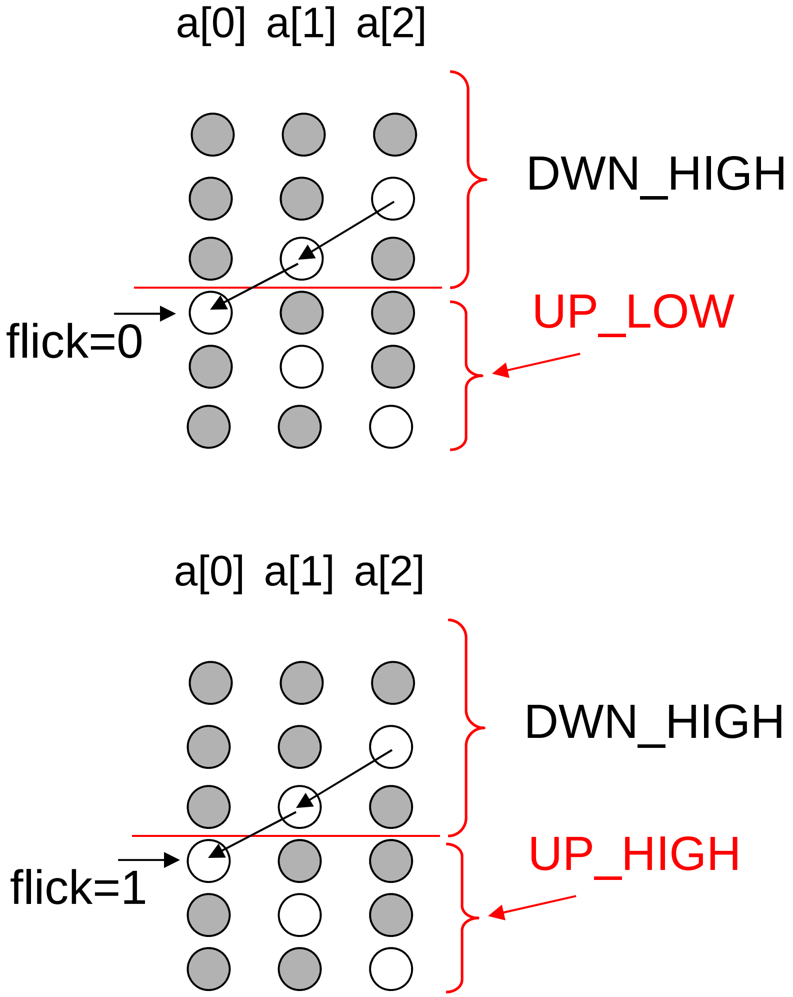


```
always @   *   begin
  case (state)

   UP_HIGH : next_state  =
              (a [7] == 1'b1 )?  DWN_HIGH :UP_HIGH ;

   endcase
end
always @   *   begin
  case (state )

    UP_HIGH : if (a[7]==1'b1) next_a = a >> 1 ;
              else              next_a = a << 1 ;
   endcase
end
```



```
always @   *   begin
  case (state)

   UP_HIGH : next_state  =
              (a [6] == 1'b1 )?  DWN_HIGH :UP_HIGH ;

   endcase
end
always @   *   begin
  case (state )

    UP_HIGH : next_a = a << 1 ;
   endcase
end
```

Down to Up change

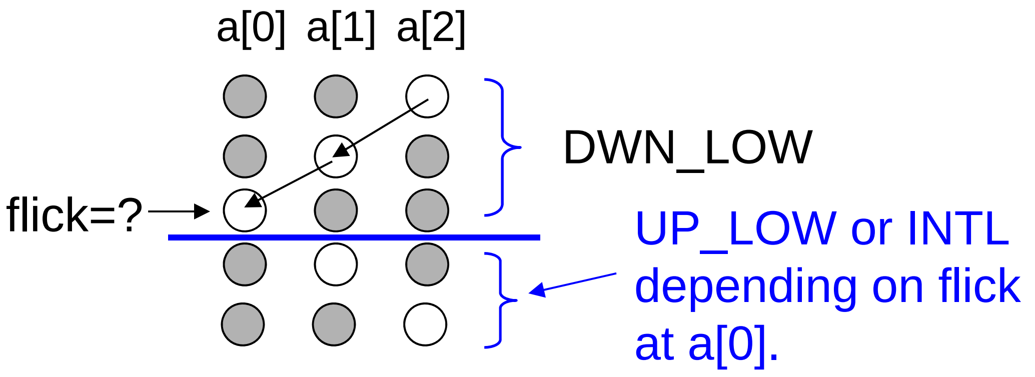


In case of down to up change, the next state depends on the current state and flick signal.

However in the current state we can not know valid value of flick because flick is valid at the beginning of the next state where a[0] is 1.

If we divide the state between a[1] and a[0], we can not tell which state to go to at the end of the current state.

Down to Up change

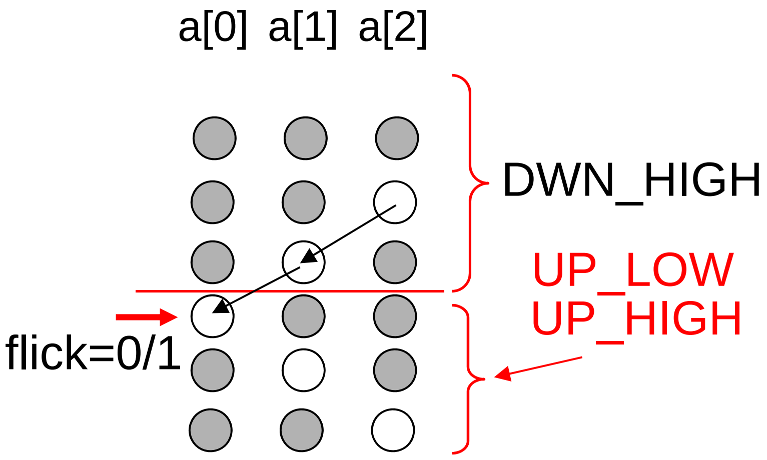


UP_LOW or INTL depending on flick at a[0].

If Down state includes a[0] on, we can tell what shall be the next state by checking flick when a[0] is on.

DWN_HIGH and DWN_LOW shall include a[0] on.

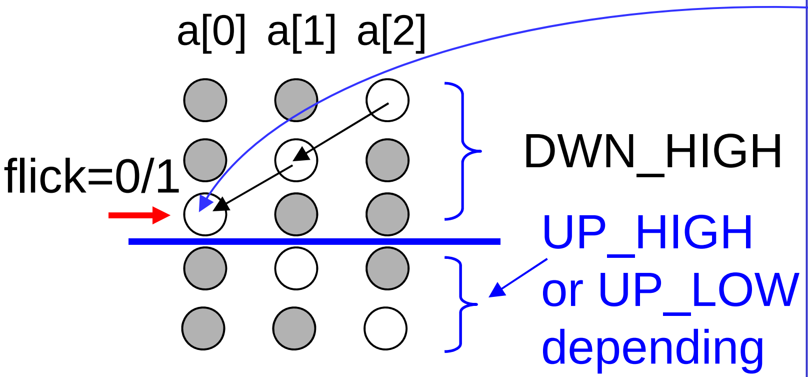
Difference in RTL code



UP_LOW
UP_HIGH

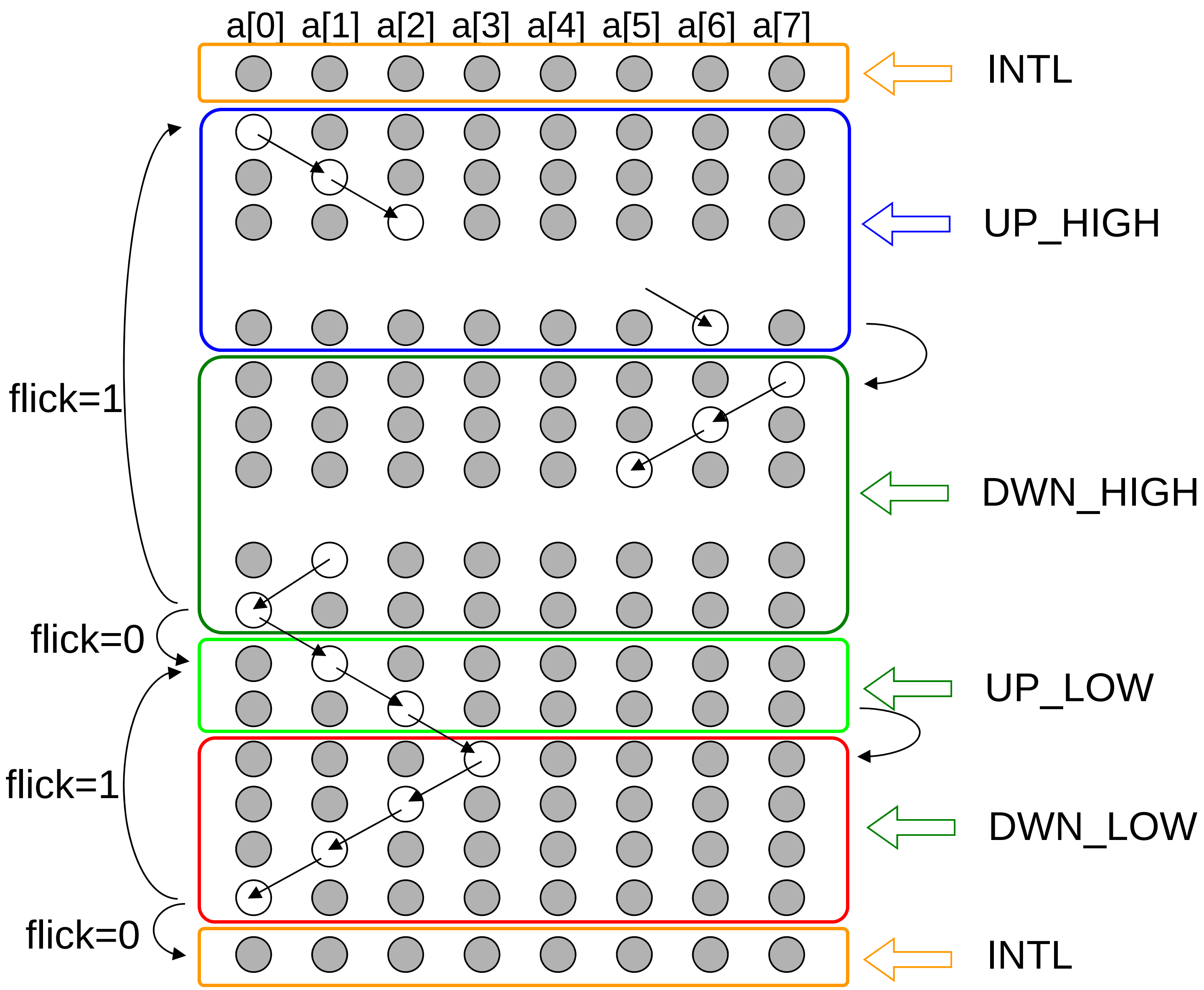
```
always @   *   begin
  case (state)

    DWN_HIGH : if (a ~~[0] == 1'b0 )~~next_state  = state;
                      ~~else  ( flick==1'b1 )?~~  UP_HIGH :UP_LOW;

    endcase
end
```

We can not create the code because a[0] never becomes 1 in DWN_HIGH state. Even if we change a[0] to a[1], flick is not valid when a[1] is 1.



UP_HIGH or UP_LOW depending on flick at a[0].

```
always @   *   begin
  case (state)

    DWN_HIGH : if (a [0] == 1'b0 ) next_state  = state ;
                      else next_state=
                            (flick == 1'b1)?  UP_HIGH :UP_LOW ;

    endcase
end
```


a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
INTL
UP_HIGH
DWN_HIGH
UP_LOW
DWN_LOW
INTL
flick=1
flick=0
flick=1
flick=0

9.3 How to write a state transition table, STT

For these states we can draw a state transition table shown below.

The states transition table for the bound flasher

event \ state			INTL	UP_HIGH	DWN_HIGH	UP_LOW	DWN_LOW
rst_n=0			a=0 ⇨ INTL	a=0 ⇨ INTL	a=0 ⇨ INTL	a=0 ⇨ INTL	a=0 ⇨ INTL
rst_n=1	flick= 0	a[6] = 1	⇩	a << 1 ⇨ DWN_HIGH	a >> 1	⇩	⇩
		a[2] = 1	⇩	a << 1	a >> 1	a << 1 ⇨ DWN_LOW	a >> 1
		a[0] = 1	⇩	a << 1	a << 1 ⇨ UP_LOW	⇩	a >> 1 ⇨ INTL
		other than above	no operation	a << 1	a >> 1	a << 1	a >> 1
	flick= 1	a[6] = 1	⇩	a << 1 ⇨ DWN_HIGH	a >> 1	⇩	⇩
		a[2] = 1	⇩	a << 1	a >> 1	a << 1 ⇨DWN_LOW	a >> 1
		a[0] = 1	⇩	a << 1	a << 1 ⇨ UP_HIGH	⇩	a << 1 ⇨ UP_LOW
		other than above	a = 1 ⇨UP_HIGH	a << 1	a >> 1	a << 1	a >> 1

⇩ : shows that the same operation in “other than above” row is applied.

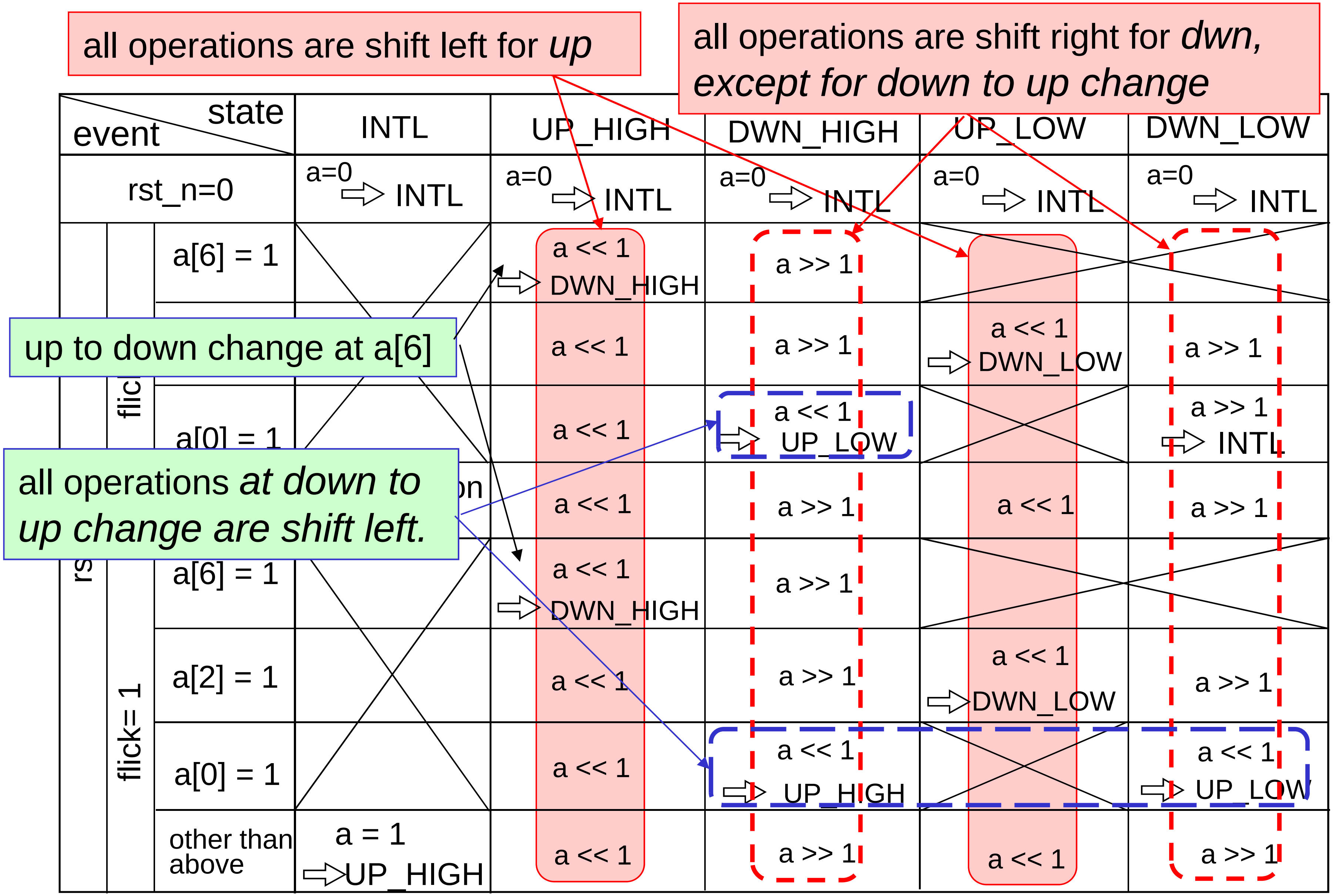
Q9.3-1: Apply consistency check to the table below.

The states transition table for the bound flasher

event \ state			INTL	UP_HIGH	DWN_HIGH	UP_LOW	DWN_LOW
rst_n=0			a=0 ⇨ INTL	a=0 ⇨ INTL	a=0 ⇨ INTL	a=0 ⇨ INTL	a=0 ⇨ INTL
rst_n=1	flick= 0	a[6] = 1		a << 1 ⇨ DWN_HIGH	a >> 1		
		a[2] = 1		a << 1	a >> 1	a << 1 ⇨ DWN_LOW	a >> 1
		a[0] = 1		a << 1	a << 1 ⇨ UP_LOW		a >> 1 ⇨ INTL
		other than above	no operation	a << 1	a >> 1	a << 1	a >> 1
	flick= 1	a[6] = 1		a << 1 ⇨ DWN_HIGH	a >> 1		
		a[2] = 1		a << 1	a >> 1	a << 1 ⇨DWN_LOW	a >> 1
		a[0] = 1		a << 1	a << 1 ⇨ UP_HIGH		a << 1 ⇨ UP_LOW
		other than above	a = 1 ⇨UP_HIGH	a << 1	a >> 1	a << 1	a >> 1

Every transition is synchronized to the clock.

Q9.3-1: A sample answer



all operations are shift left for *up*

all operations are shift right for *dwn*, *except for down to up change*

up to down change at a[6]

all operations *at down to up change are shift left.*

state event			INTL	UP_HIGH	DWN_HIGH	UP_LOW	DWN_LOW
rst_n=0			a=0 ⇨ INTL	a=0 ⇨ INTL	a=0 ⇨ INTL	a=0 ⇨ INTL	a=0 ⇨ INTL
rs	flic	a[6] = 1		a << 1 ⇨ DWN_HIGH	a >> 1		
				a << 1	a >> 1	a << 1 ⇨ DWN_LOW	a >> 1
		a[0] = 1		a << 1	a << 1 ⇨ UP_LOW		a >> 1 ⇨ INTL
			on	a << 1	a >> 1	a << 1	a >> 1
	flick= 1	a[6] = 1		a << 1 ⇨ DWN_HIGH	a >> 1		
		a[2] = 1		a << 1	a >> 1	a << 1 ⇨ DWN_LOW	a >> 1
		a[0] = 1		a << 1	a << 1 ⇨ UP_HIGH		a << 1 ⇨ UP_LOW
		other than above	a = 1 ⇨ UP_HIGH	a << 1	a >> 1	a << 1	a >> 1

Some discussion about state transition table

		event \ state	INTL	UP_HIGH	DWN_HIGH	UP_LOW	DWN_LOW
rst_n=0			a=0 ⇨ INTL	a=0 ⇨ INTL	a=0 ⇨ INTL	a=0 ⇨ INTL	a=0 ⇨ INTL
rst_n=1	flick= 0	a[6] = 1	⇩	a << 1 ⇨ DWN_HIGH	a >> 1	⇩	⇩
		a[2] = 1	⇩	a << 1	a >> 1	a << 1 ⇨ DWN_LOW	a >> 1
		a[0] = 1	⇩	a << 1	a << 1 ⇨ UP_LOW	⇩	a >> 1 ⇨ INTL
		other than above	no operation	a << 1	a >> 1	a << 1	a >> 1
	flick= 1	a[6] = 1	⇩	a << 1 ⇨ DWN_HIG			⇩
		a[2] = 1	⇩	a << 1			>> 1
		a[0] = 1	⇩	a << 1	UP_HIGH		<< 1 UP_LOW
		other than above	a = 1 ⇨UP_HIGH	a << 1	a >> 1	a << 1	a >> 1

Investigate the table shown on the previous page. It looks incomplete because “other than above” includes cases which will never happen.

⇩ : shows that the same operation in “other than above” row is applied.

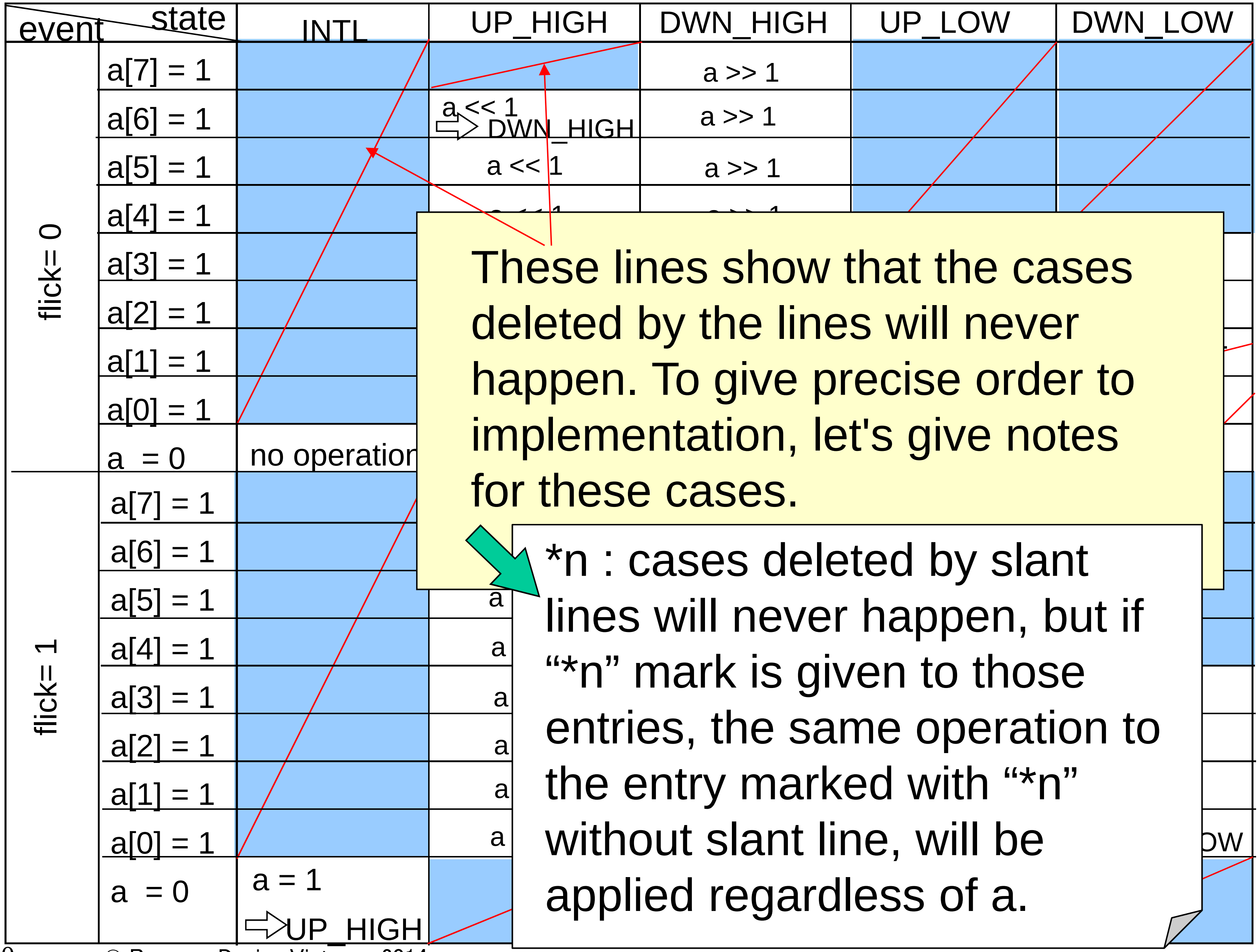
First, check how many events do we have if we choose flick and each one of the lamps as inputs instead of just choosing a[0], a[2], and a[6].

The lamp position has 9 possibilities, all off, a[0] on, a[1] on, a[2] on, a[3] on, a[4] on, , , , , , a[6] on, and a[7] on. Regarding flick, it can be either 0 or 1. Therefore event must be 9 x 2 (flick = 0/1) = 18 cases if we are going to be more precise than the table on the previous page.

If we draw the table with 18 entries above, the table shown on the next page can be obtained.
(To save space, omit rst_n=0 case, that is, let's make the table only for rst_n=1 only for a while.)

state transition table

state event		INTL	UP_HIGH	DWN_HIGH	UP_LOW	DWN_LOW
flick= 0	a[7] = 1			a >> 1		
	a[6] = 1		a << 1 ⇨ DWN_HIGH	a >> 1		
	a[5] = 1		a << 1	a >> 1		
	a[4] = 1		a << 1	a >> 1		
	a[3] = 1		a << 1	a >> 1		a >> 1
	a[2] = 1		a << 1	a >> 1	a << 1 ⇨ DWN_LOW	a >> 1
	a[1] = 1		a << 1	a >> 1	a << 1	a >> 1
	a[0] = 1		a << 1	a << 1 ⇨ UP_LOW		a >> 1 ⇨ INTL
	a = 0	no operation				
flick= 1	a[7] = 1			a >> 1		
	a[6] = 1		a << 1 ⇨ DWN_HIGH	a >> 1		
	a[5] = 1		a << 1	a >> 1		
	a[4] = 1		a << 1	a >> 1		
	a[3] = 1		a << 1	a >> 1		a >> 1
	a[2] = 1		a << 1	a >> 1	a << 1 ⇨ DWN_LOW	a >> 1
	a[1] = 1		a << 1	a >> 1	a << 1	a >> 1
	a[0] = 1		a << 1	a << 1 ⇨ UP_HIGH		a >> 1 ⇨ UP_LOW
	a = 0	a = 1 ⇨ UP_HIGH				



	event \ state	INTL	UP_HIGH	DWN_HIGH	UP_LOW	DWN_LOW
flick= 0	a[7] = 1			a >> 1		
	a[6] = 1		a << 1 ⇨ DWN_HIGH	a >> 1		
	a[5] = 1		a << 1	a >> 1		
	a[4] = 1		a << 1	a >> 1		
	a[3] = 1					
	a[2] = 1					
	a[1] = 1					
	a[0] = 1					
	a = 0	no operation				
flick= 1	a[7] = 1					
	a[6] = 1					
	a[5] = 1		a			
	a[4] = 1		a			
	a[3] = 1		a			
	a[2] = 1		a			
	a[1] = 1		a			
	a[0] = 1		a			ow
	a = 0	a = 1 ⇨UP_HIGH				

These lines show that the cases deleted by the lines will never happen. To give precise order to implementation, let's give notes for these cases.

*n : cases deleted by slant lines will never happen, but if "*n" mark is given to those entries, the same operation to the entry marked with "*n" without slant line, will be applied regardless of a.

State transition table (most precise)

	event \ state	INTL	UP_HIGH	DWN_HIGH	UP_LOW	DWN_LOW
flick= 0	a[7] = 1	*1	*3	a >> 1	*7	*9
	a[6] = 1	*1	a << 1 ⇨ DWN_HIGH	a >> 1	*7	*9
	a[5] = 1	*1	a << 1	a >> 1	*7	*9
	a[4] = 1	*1	a << 1	a >> 1	*7	*9
	a[3] = 1	*1	a << 1	a >> 1	*7	a >> 1 *9
	a[2] = 1	*1	a << 1	a >> 1	a << 1 ⇨ DWN_LOW	a >> 1
	a[1] = 1	*1	a << 1	a >> 1 *5	a << 1 *7	a >> 1
	a[0] = 1	*1	a << 1 *3	a << 1 ⇨ UP_LOW	*7	a >> 1 ⇨ INTL
	a = 0	*1 no operation	*3	*5	*7	*9
flick= 1	a[7] = 1	*2	*4	a >> 1	*8	*10
	a[6] = 1	*2	a << 1 ⇨ DWN_HIGH	a >> 1	*8	*10
	a[5] = 1	*2	a << 1	a >> 1	*8	*10
	a[4] = 1	*2	a << 1	a >> 1	*8	*10
	a[3] = 1	*2	a << 1	a >> 1	*8	a >> 1
	a[2] = 1	*2	a << 1	a >> 1	a << 1 ⇨ DWN_LOW	a >> 1
	a[1] = 1	*2	a << 1	a >> 1 *6	*8 a << 1	a >> 1 *10
	a[0] = 1	*2	a << 1 *4	a << 1 ⇨ UP_HIGH	*8	a >> 1 ⇨ UP_LOW
	a = 0	a = 1 *2 ⇨ UP_HIGH	*4	*6	*8	*10

*n : Cases deleted by slant lines will never happen. “*n” mark in these entries deleted by slant lines show that the same operation to the entry, marked with “*n” and not deleted with slant line, will be applied regardless of a.

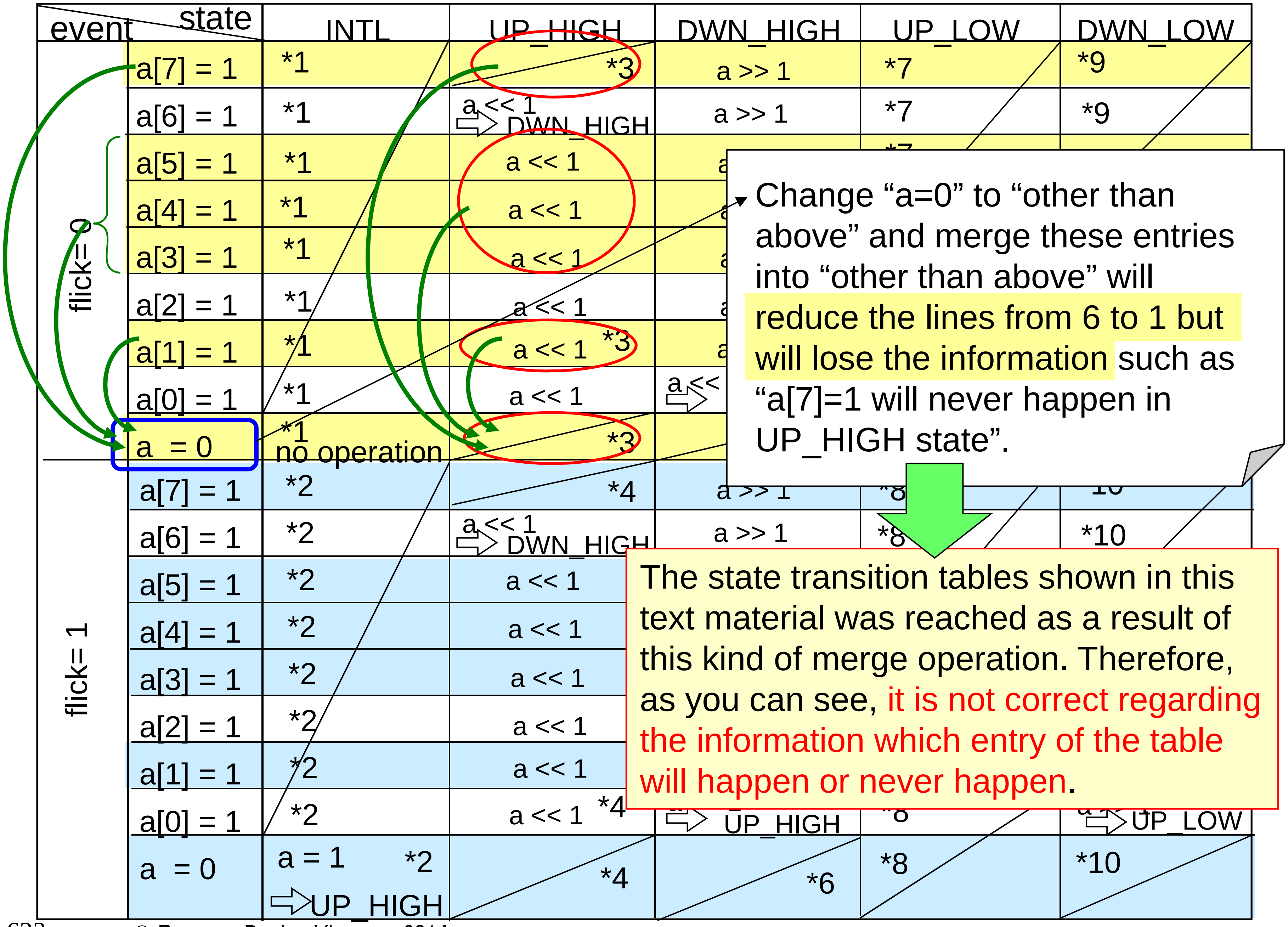
State transition table (most precise)

	state event	INTL	UP_HIGH	DWN_HIGH	UP_LOW	DWN_LOW
flick= 0	a[7] = 1	*1	*3	a >> 1	*7	*9
	a[6] = 1	*1	a << 1 ⇨ DWN_HIGH	a >> 1	*7	*9
	a[5] = 1	*1	a << 1	a >> 1	*7	*9
	a[4] = 1	*1	a << 1	a >> 1	*7	*9
	a[3] = 1	*1	[illegible]	[illegible]	*7	a >> 1 *9
	a[2] = 1	*	[illegible]	[illegible]	N_LOW	a >> 1
	a[1] = 1	*	[illegible]	[illegible]	< 1 *7	a >> 1
	a[0] = 1	*	[illegible]	[illegible]		a >> 1 ⇨ INTL
	a = 0	* n	[illegible]	[illegible]		*9
flick= 1	a[7] = 1	*	[illegible]	[illegible]		*10
	a[6] = 1	*	[illegible]	[illegible]		*10
	a[5] = 1	*	[illegible]	[illegible]		*10
	a[4] = 1	*	[illegible]	[illegible]		*10
	a[3] = 1	*	[illegible]	[illegible]		a >> 1
	a[2] = 1	*	[illegible]	[illegible]	N_LOW	a >> 1
	a[1] = 1	*	[illegible]	[illegible]	< 1	a >> 1 *10
	a[0] = 1	*	[illegible]	[illegible]		a >> 1 ⇨ UP_LOW
	a = 0	a ⇨ UP_HIGH	[illegible]	[illegible]	3	*10

This must be the most precise state transition table.

However, if cases for input events are so many, the table will become too huge to be handled by a designer. In such case, lines having same or similar operation and state transition are better to be merged into one line for better readability. The table in this page seems to have many lines which can be merged into one line.

*n : Cases deleted by slant lines will never happen. “*n” mark in these entries deleted by slant lines show that the same operation to the entry, marked with “*n” and not deleted with slant line, will be applied regardless of a.

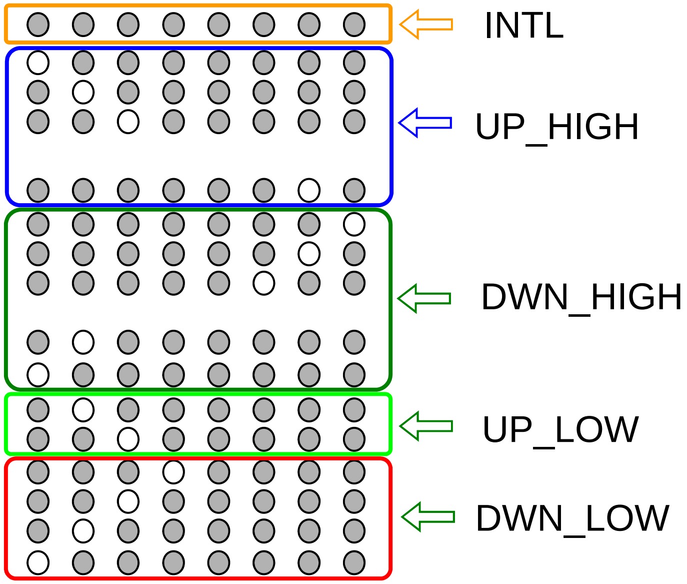


	event \ state	INTL	UP_HIGH	DWN_HIGH	UP_LOW	DWN_LOW
flick= 0	a[7] = 1	*1	*3	a >> 1	*7	*9
	a[6] = 1	*1	a << 1 ⇨ DWN_HIGH	a >> 1	*7	*9
	a[5] = 1	*1	a << 1	a	*7	
	a[4] = 1	*1	a << 1	a		
	a[3] = 1	*1	a << 1	a		
	a[2] = 1	*1	a << 1	a		
	a[1] = 1	*1	a << 1 *3	a		
	a[0] = 1	*1	a << 1	a << ⇨		
	a = 0	*1 no operation	*3			
flick= 1	a[7] = 1	*2	*4	a >> 1	*8	10
	a[6] = 1	*2	a << 1 ⇨ DWN_HIGH	a >> 1	*8	*10
	a[5] = 1	*2	a << 1			
	a[4] = 1	*2	a << 1			
	a[3] = 1	*2	a << 1			
	a[2] = 1	*2	a << 1			
	a[1] = 1	*2	a << 1			
	a[0] = 1	*2	a << 1 *4	⇨ UP_HIGH	8	⇨ UP_LOW
	a = 0	a = 1 *2 ⇨ UP_HIGH	*4	*6	*8	*10

Change "a=0" to "other than above" and merge these entries into "other than above" will reduce the lines from 6 to 1 but will lose the information such as "a[7]=1 will never happen in UP_HIGH state".

The state transition tables shown in this text material was reached as a result of this kind of merge operation. Therefore, as you can see, it is not correct regarding the information which entry of the table will happen or never happen.

Which lamp can be on in what state is defined by the following figure.



Therefore, state transition table may not necessarily define the relation between lamp position and state.

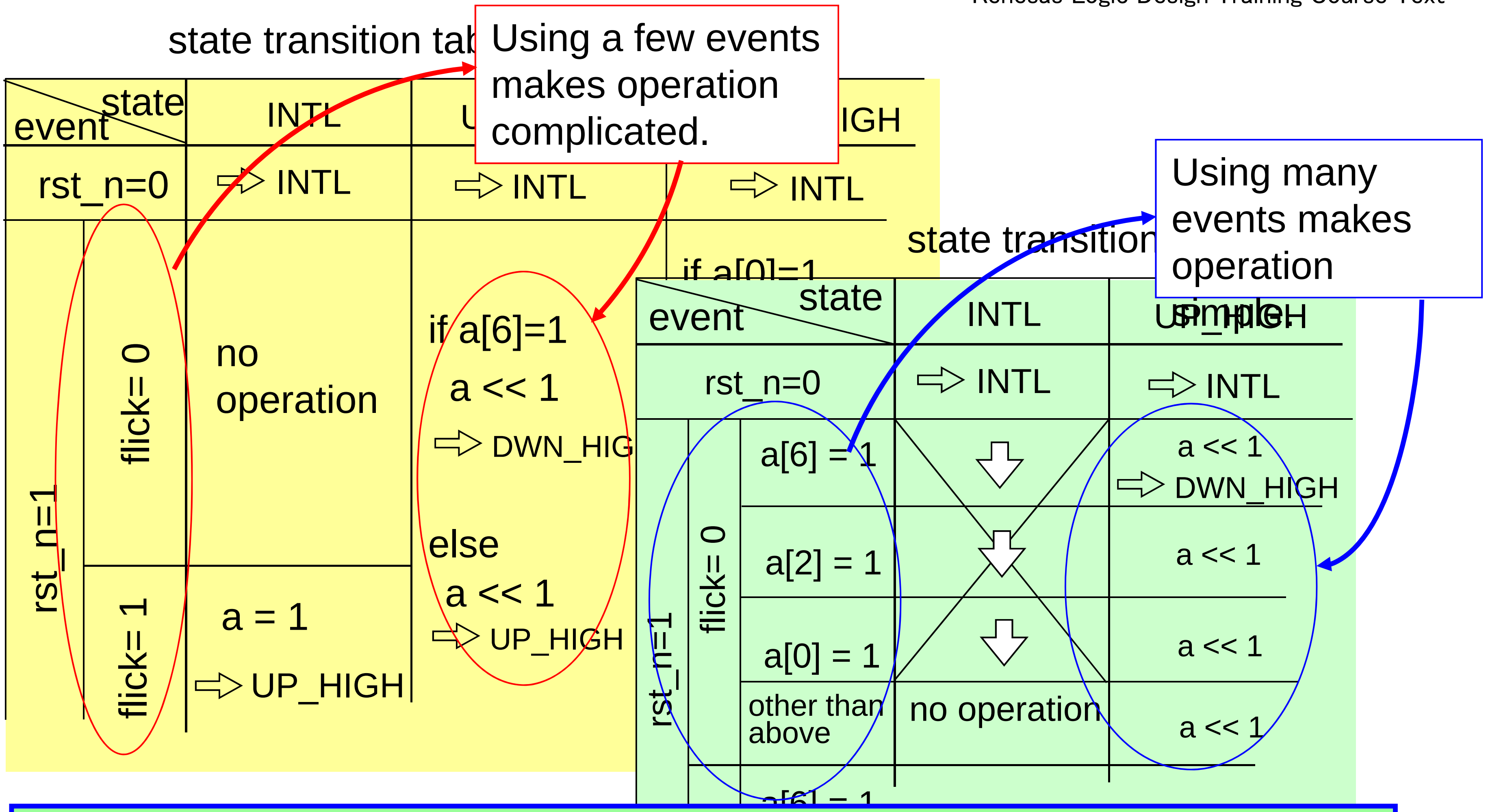
There may be better table structure for bound flasher.

Redraw the table with only one input flick.

state transition table

event \ state		INTL	UP_HIGH	DWN_HIGH	UP_LOW	DWN_LOW
rst_n=0		a = 0 ⇨ INTL	a = 0 ⇨ INTL	a = 0 ⇨ INTL	a = 0 ⇨ INTL	a = 0 ⇨ INTL
rst_n=1	flick= 0	no operation	if a[6]=1 a << 1 ⇨ DWN_HIGH else a << 1	if a[0]=1 a << 1 ⇨ UP_LOW else a >> 1	if a[2]=1 a << 1 ⇨DWN_LOW else a << 1	if a[0]=1 a = 0 ⇨ INTL else a >> 1
	flick= 1	a = 1 ⇨ UP_HIGH		if a[0]=1 a << 1 ⇨ UP_HIGH else a >> 1		if a[0]=1 a << 1 ⇨ UP_LOW else a >> 1

This may be better than the table having “other than above” as input events.



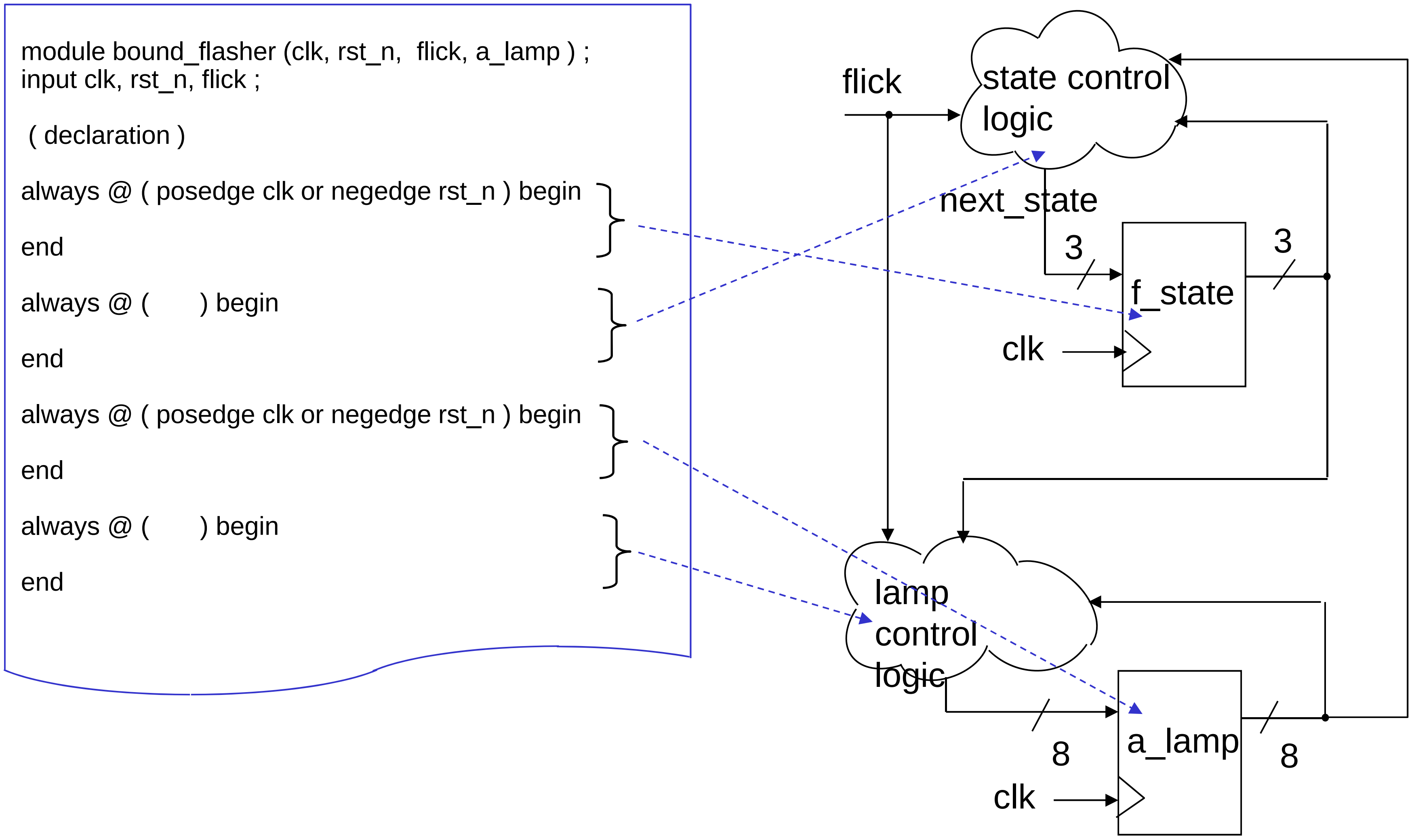
Using a few events makes operation complicated.

Using many events makes operation simple.

In general the style on the right is better because it can be very precise. However, if event listing becomes so large that some lines must be merged to make the table smaller and such merging may destroy the correctness of the table, then the style on the left must be better. We must carefully select what inputs to list up as events considering the correctness and simplicity.

9.4 RTL code for the bound flasher

Now, write an RTL code for the bound flasher using the following structure.



```
module bound_flasher (clk, rst_n,  flick, a_lamp ) ;
input clk, rst_n, flick ;

 ( declaration )

always @ ( posedge clk or negedge rst_n ) begin

end

always @ (       ) begin

end

always @ ( posedge clk or negedge rst_n ) begin

end

always @ (       ) begin

end
```

```
//-------------------------------------------------------
// Project Name   : Rensas RTL coding training, DesignWS_P1
//                : bound flasher
// File Name      : bound_flasher_v0.v
// Module Name    : bound_flasher
// Function       : this logic flash lamps in sequence at evry clock rise time
//                  lamp will go up and down.
// Note        : Following code does use * for sensitivity list.
//             In real project make rule to use * or not to use * in
//             sensitivity list. Using * can avoid possible error
//             such as missing variables from sensitivity list.
// Author         : K. Hayashi (idxxxx)
//-------------------------------------------------------
// History
// Version  Date       Author      Description
// ver.1.0  2007.xx.xx  K. Hayashi   New creation
// ver.1.1  2007.xx.xx  K. Hayashi   use parameter instead of define
// ver.1.2  2007.xx.xx  K. Hayashi   use begin end for all statement
//-------------------------------------------------------
//
module bound_flasher ( clk, rst_n, flick, a_lamp ) ;
// parameters
parameter INTL     = 3'b000 ; // initial state
parameter UP_HIGH  = 3'b001 ; // going up from bottom to top
parameter DWN_HIGH = 3'b010 ; // coming down from top to bottom
parameter UP_LOW   = 3'b100 ; // going up from bottom to middle
```

Checked on cver

```
parameter DWN_LOW  = 3'b101 ; // coming down from middle to bottom
//
parameter MX_LP = 8 ;  // number of lamps
parameter TP_TP = MX_LP -1 ; // top turning point
parameter MD_TP = 3 ;  // middle turning point
parameter FF_DLY = 1 ; // delay for FF to avoid racing
//  port definition
input clk, rst_n ;
input flick ;
output [MX_LP-1:0] a_lamp ;

wire clk, rst_n ;
wire flick ;
reg [MX_LP-1:0] a_lamp ;  // FF for lamps

// internal variables
reg [2:0] f_state ;       // FF for state
reg [2:0] next_f_state ;    // non-FF
reg [MX_LP-1:0] next_lamp ; // non-FF

// state FF definition
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n==1'b0 ) begin          // initialize state
      f_state <= #FF_DLY  INTL ;
  end
  else begin
      f_state <= #FF_DLY  next_f_state ;
  end
end
```

Checked on cver

```
always @ ( f_sate or flick or a_lamp ) begin
 case ( f_state )
  INTL    : begin
        next_f_state = ( flick )? UP_HIGH : f_state ;
      end
  UP_HIGH : begin
        next_f_state = ( a_lamp[TP_TP-1] )? DWN_HIGH : f_state ;
      end
 DWN_HIGH : begin
        if ( a_lamp[0] ) begin  // change direction at floor
          next_f_state = ( flick )? UP_HIGH : UP_LOW ;
        end
        else begin       // stay in    going down high
          next_f_state = f_state  ;
        end
      end
  UP_LOW : begin
        next_f_state = ( a_lamp[MD_TP-1] )? DWN_LOW : f_state ;
      end
 DWN_LOW : begin
       if ( a_lamp[0] ) begin   // bound at floor?
          next_f_state = ( flick )? UP_LOW : INTL ;
       end
       else begin
          next_f_state = f_state  ; // stay in    going down low
       end
      end
  default : begin   // set x for debug
         next_f_state = 3'bxxx ;
      end
 endcase
end
```

Checked on cver

```
always @ ( posedge clk or negedge rst_n )   begin      // lamp control logic
  if ( rst_n==1'b0 ) begin              // clear lamp on reset
         a_lamp <= #FF_DLY  { MX_LP{1'b0} } ;
  end
  else begin                        // set lamp
         a_lamp <= #FF_DLY  next_lamp ;
  end
end

always @ ( f_sate or flick or a_lamp )  begin
  case ( f_state  )
       INTL : begin
                next_lamp = ( flick )? { { (MX_LP-1){1'b0} } , 1'b1 } : a_lamp ;
              end
    UP_HIGH : begin        // up one step
                next_lamp = a_lamp <<1 ;
              end
   DWN_HIGH : begin        // down one step until floor
                next_lamp = ( a_lamp[0] )? a_lamp << 1 : a_lamp >> 1 ;
              end
     UP_LOW : begin         // up one step
                next_lamp = a_lamp <<1 ;
              end
    DWN_LOW : begin         // down one step until floor
                next_lamp = ( a_lamp[0] & flick )? a_lamp << 1 : a_lamp >> 1 ;
              end
    default : begin   // error, set x for debug
                next_lamp = { MX_LP{1'bx} } ;
              end
  endcase
end

endmodule
```

Checked on cver

A test bench can be coded as below.

```
module test_bndflsh  ;
parameter HALF_CYCLE = 5 ;
parameter CYCLE = HALF_CYCLE * 2 ;
parameter MX_LP = 8 ;
reg clk, rst_n, flick ;
wire [MX_LP-1:0] lamp ;
// connect signals to flasher
defparam bound_flasher_01.MX_LP = 8 ;
bound_flasher bound_flasher_01 (.clk(clk), .rst_n(rst_n), .flick(flick), .a_lamp(lamp) ) ;
// connection end
always begin      // clock generator
                        clk = 1'b0 ;
    # HALF_CYCLE    clk = 1'b1 ;
    # HALF_CYCLE  ;
end
always @ ( posedge clk ) $strobe ("t= %d, rst_n=%b, clk=%b, flick=%b, f_state=%b, lamp=%b",
                    $stime, rst_n, clk, flick, bound_flasher_01.f_state, lamp ) ;
initial begin
   rst_n = 0 ;
    # (CYCLE * 3 ) rst_n = 1'b1 ;
end
initial begin // give value to control variable
   flick = 1'b0 ;
   # (CYCLE * 5 ) flick = 1'b1 ;
   # (CYCLE * 20 ) flick = 1'b0 ;
   # (CYCLE * 24 ) flick = 1'b1 ;
   # (CYCLE ) flick = 1'b0 ;
   # (CYCLE * 20 ) flick = 1'b1 ;
   # (CYCLE * 20 ) flick = 1'b0 ;
   # (CYCLE * 10 ) $finish ;
end
endmodule
```

This line is not needed. It is just to show how it works. Change “8” to “12” and run simulator to see the result.

Checked on cver

Q9.4-1 : Run the following test bench and see how @ works.

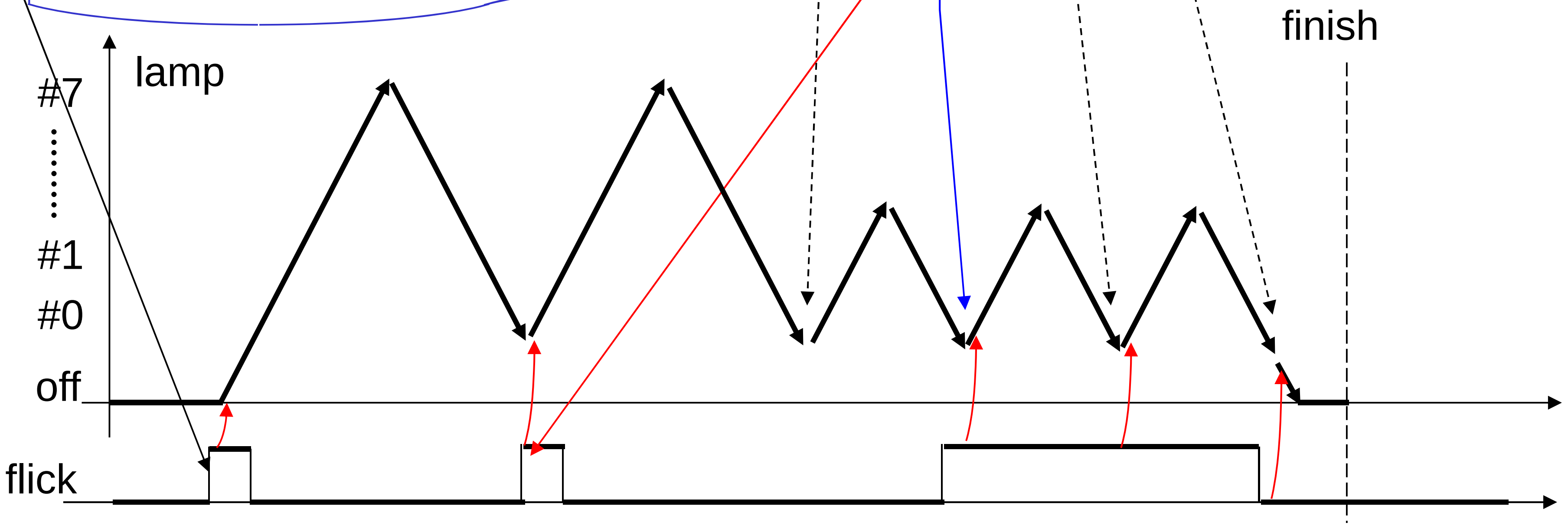
```
module test_bndflsh  ;
parameter HALF_CYCLE = 5 ;
parameter CYCLE = HALF_CYCLE * 2 ;
parameter MX_LP = 8 ;
  ------------------------------------------------------------
  |                                                          |
  |        Same as the previous page.                        |
  |                                                          |
  ------------------------------------------------------------
initial begin
    rst_n = 0 ;
     # (CYCLE * 3 ) rst_n = 1'b1 ;
end
initial begin // give value to control variable
    flick = 1'b0 ;
    # (CYCLE * 5 ) flick = 1'b1 ;
    # (CYCLE ) flick = 1'b0 ;
    # (CYCLE * 2 ) ;
    @ (lamp[0]==1'b1) # HALF_CYCLE flick = 1'b1 ;
    # (CYCLE) flick = 1'b0 ;
   @ (lamp[0]==1'b1) #(CYCLE + HALF_CYCLE);
   @ (lamp[0]==1'b1) #HALF_CYCLE flick =  1'b1 ;
    # CYCLE ;
   @ (lamp[0]==1'b1) #(CYCLE + HALF_CYCLE);
   @ (lamp[0]==1'b1)  flick = 1'b0 ;
   # (CYCLE * 2 )  $finish ;
end
endmodule
```

Checked on cver

Q9.4-1 : A sample answer.

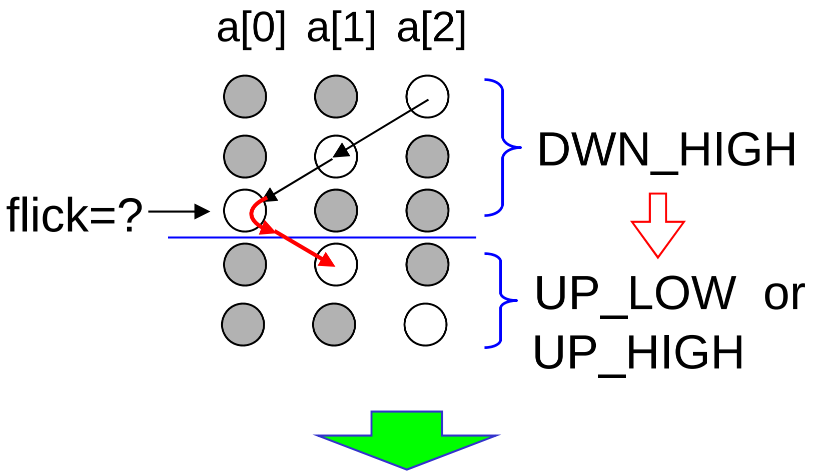
```
initial begin // give value to control variable
    flick = 1'b0 ;
    # (CYCLE * 5 ) flick = 1'b1 ;
    # (CYCLE ) flick = 1'b0 ;
    # (CYCLE * 2 ) ;
    @ (lamp[0]==1'b1) # HALF_CYCLE flick = 1'b1 ;
    # (CYCLE) flick = 1'b0 ;
   @ (lamp[0]==1'b1) #(CYCLE + HALF_CYCLE);
   @ (lamp[0]==1'b1) #HALF_CYCLE flick =  1'b1 ;
    # CYCLE ;
   @ (lamp[0]==1'b1) #(CYCLE + HALF_CYCLE);
   @ (lamp[0]==1'b1)  flick = 1'b0 ;
   # (CYCLE * 2 )  $finish ;
end
endmodule
```

By using @, we can control input signals synchronized to events.

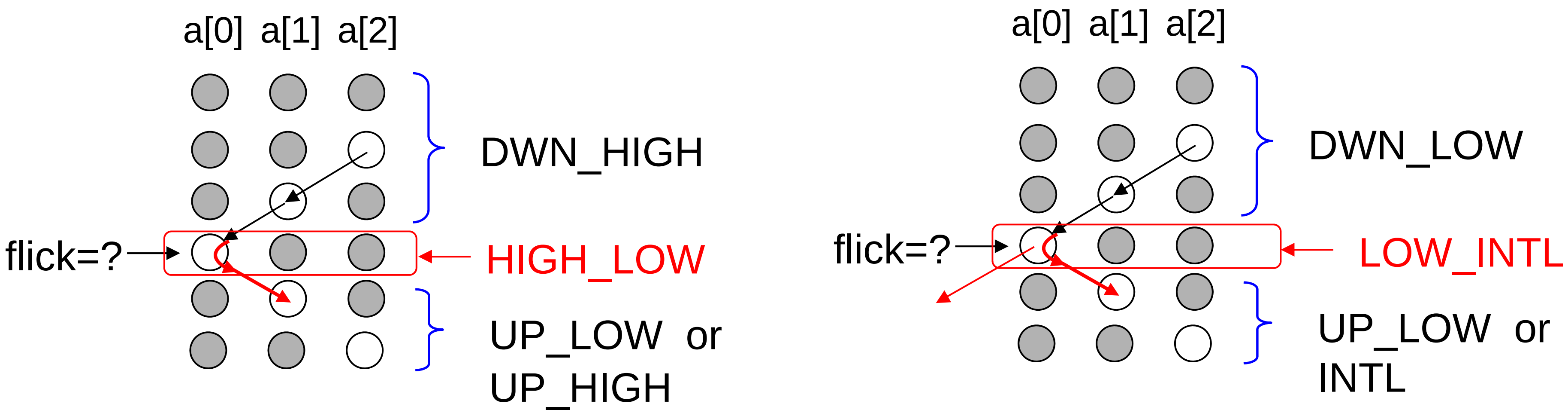


Checked on cver

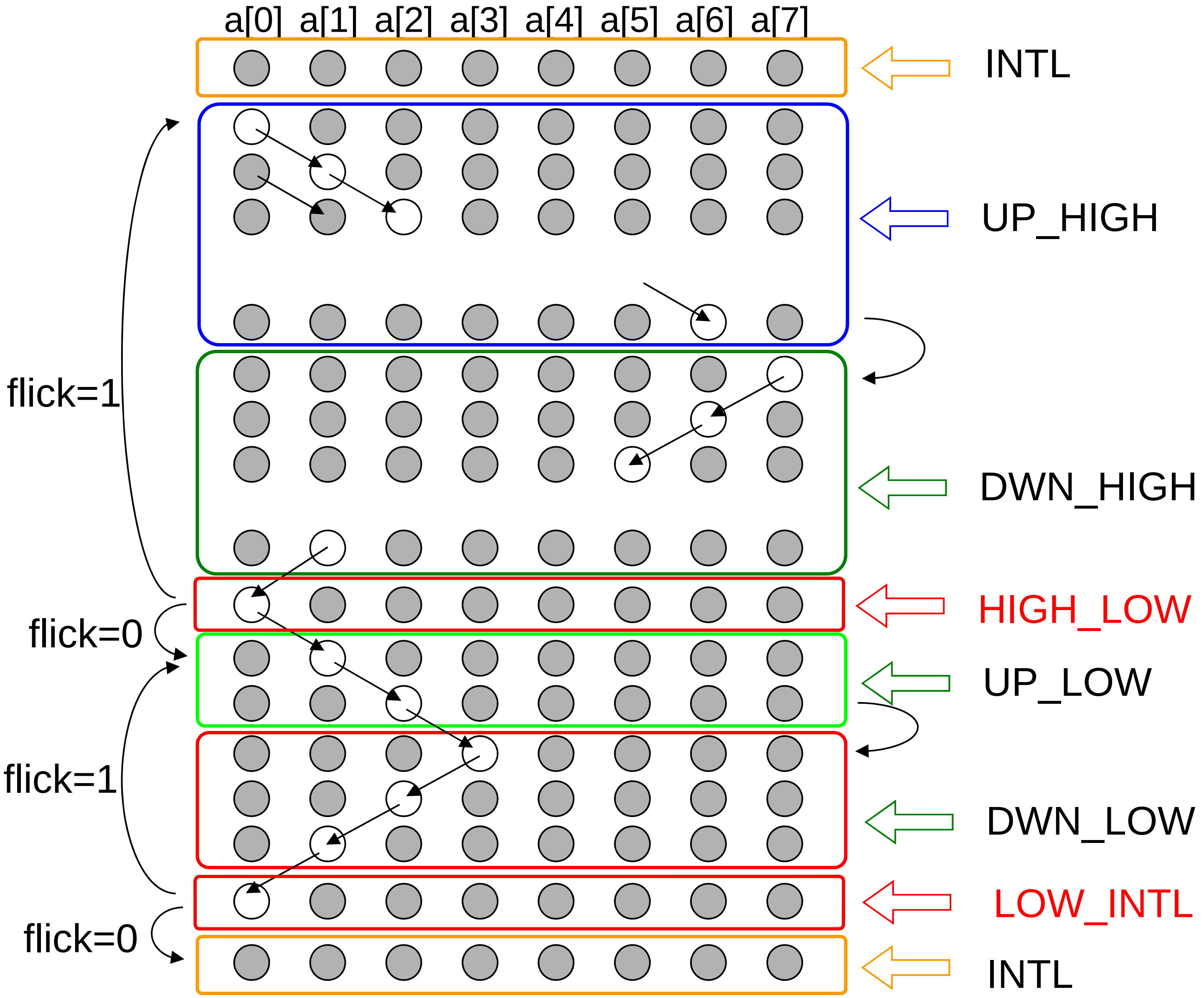
9.5 Introducing new states



We can introduce two more states, HIGH_LOW, meaning next is “go up high or go up low” depending on flick, and LOW_INTL, meaning next is “go up low or initial” depending on flick.
This will make the logic very simple because flick must be checked only in INTL and those newly introduced two states.



By introducing these two states, HIGH_LOW and LOW_INTL, total states are defined as shown on the next page.


a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7]
INTL
UP_HIGH
DWN_HIGH
HIGH_LOW
UP_LOW
DWN_LOW
LOW_INTL
INTL
flick=1
flick=0
flick=1
flick=0

The STT for the new logic is shown below.

*1: same to “other than above” case.

		state event	INTL	UP_ HIGH	DWN_ HIGH	HIGH_ LOW	UP_ LOW	DWN_ LOW	LOW_ INTL
rst_n=0			a = 0 ⇨ INTL	a = 0 ⇨ INTL	a = 0 ⇨ INTL	a = 0 ⇨ INTL	a = 0 ⇨ INTL	a = 0 ⇨INTL	a = 0 ⇨ INTL
rst_n=1	flick= 0	a[6] = 1	⇩	a << 1 ⇨DWN_HIGH	a >> 1	⇩	*1	⇩	⇩
		a[2] = 1	⇩	a << 1	a >> 1	⇩	a << 1 ⇨DWN_LOW	a >> 1	⇩
		a[1] = 1	⇩	a << 1	a >> 1 ⇨HIGH_LOW	⇩	a << 1	a >> 1 ⇨LOW_INTL	⇩
		a[0] = 1	⇩	a << 1	⇩	a << 1 ⇨UP_LOW	⇧	⇩	a = 0 ⇨INTL
		other than above	no operation	a << 1	a >> 1	⇧	⇧	a >> 1	⇧
	flick= 1	a[6] = 1	⇩	a << 1 ⇨DWN_HIGH	a >> 1	⇩	*1	⇩	⇩
		a[2] = 1	⇩	a << 1	a >> 1	⇩	a << 1 ⇨DWN_LOW	a >> 1	⇩
		a[1] = 1	⇩	a << 1	a >> 1 ⇨HIGH_LOW	⇩	a << 1	a >> 1 ⇨LOW_INTL	⇩
		a[0] = 1	⇩	a << 1	⇩	a << 1 ⇨UP_HIGH	⇧	⇩	a << 1 ⇨UP_LOW
		other than above	a = 1 ⇨UP_HIGH	a << 1	a >> 1	⇧	⇧	a >> 1	⇧

⇩ : the same operation in the row pointed by the allow is applied.

Q9.5-1 : Apply consistency check to the state transition table on the previous page.

Q9.5-1 : A sample answer.

Up operations are all << 1

Down operations are all << 1

		event \ state	INTL	UP_HIGH	DWN_HIGH	HIGH_LOW	UP_LOW	DWN_LOW	LOW_INTL
rst_n=0			a = 0 ⇨ INTL	a = 0 ⇨ INTL	a = 0 ⇨ INTL	a = 0 ⇨ INTL	a = 0 ⇨ INTL	a = 0 ⇨INTL	a = 0 ⇨ INTL
rst_n=1	flick	a[6] = 1		a << 1 ⇨ DWN_HIGH	a >> 1				
		a[2] = 1		a << 1	a >> 1		a << 1 ⇨ DWN_LOW	a >> 1	
		a[1] = 1		a << 1	a >> 1 ⇨ HIGH_LOW		a << 1	a >> 1 ⇨LOW_INTL	
		a[0] = 1		a << 1		a << 1 ⇨ UP_LOW			a = 0 ⇨INTL
		other than above	no operation	a << 1	a >> 1			a >> 1	
	flick= 1	a[6] = 1		a << 1 ⇨ DWN_HIGH	a >> 1				
		a[2] = 1		a << 1	a >> 1		a << 1 ⇨ DWN_LOW	a >> 1	
		a[1] = 1		a << 1	a >> 1 ⇨ HIGH_LOW		a << 1	a >> 1 ⇨LOW_INTL	
		a[0] = 1		a << 1		a << 1 ⇨ UP_HIGH			a << 1 ⇨UP_LOW
		other than above	a = 1 ⇨ UP_HIGH	a << 1	a >> 1			a >> 1	

An RTL code corresponding the new STT is shown below.

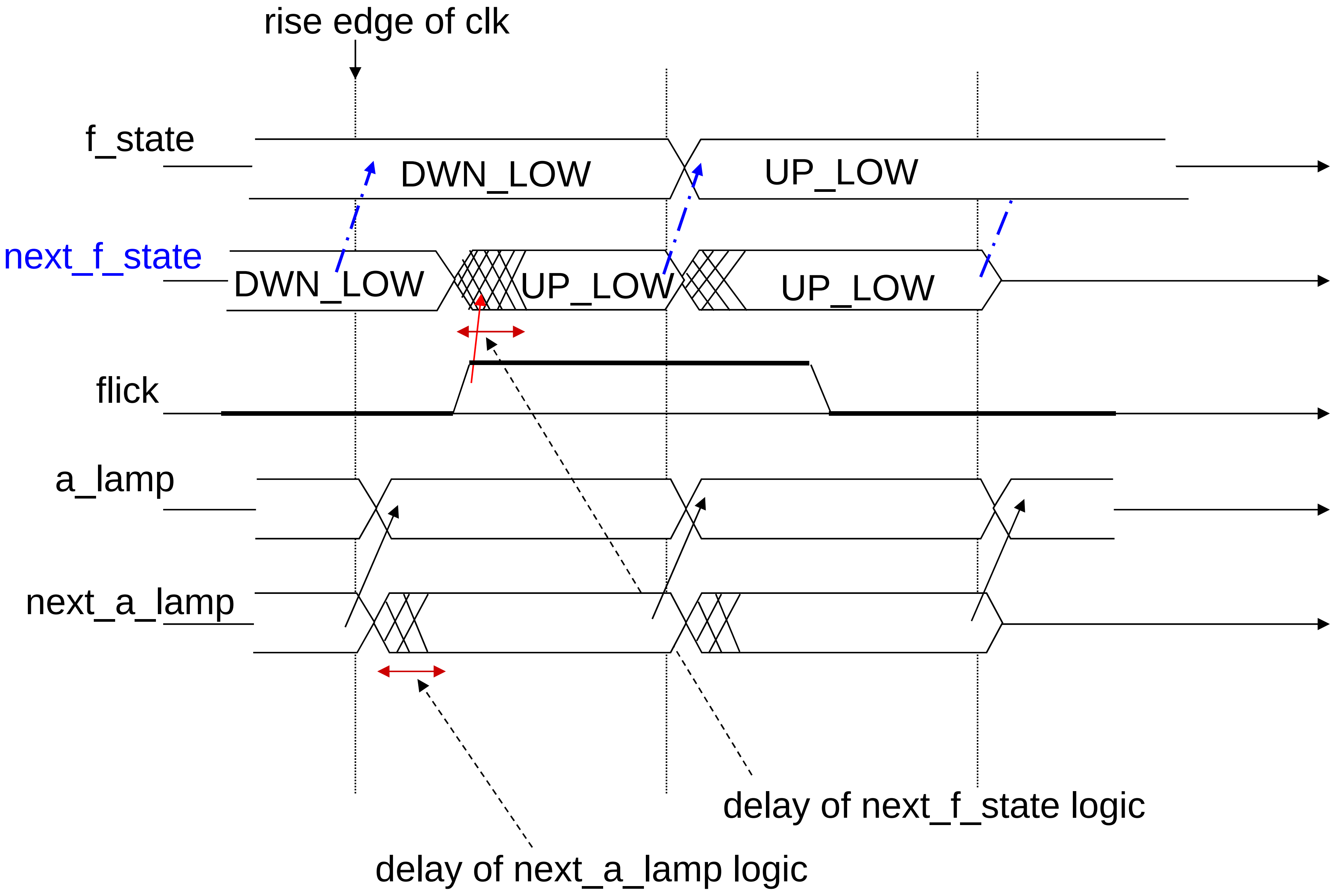
```
//-------------------------------------------------------------
// Project Name   : Rensas RTL coding training, DesignWS_P1
//                : bound flasher
// File Name      : bound_flasher_v1.v
// Module Name  : bound_flasher
// Function          : this logic flash lamps in sequence at every clock rise time
//                  lamp will go up and down.
// Note          : new two state HIGH_LOW and LOW_INTL introduced
// Author              : K. Hayashi (idxxxx)
//--------------------------------------------------------------
// History
// Version  Date        Author       Description
// ver.1.0  2007.xx.xx  K. Hayashi   New creation
// ver.1.1  2007.xx.xx  K. Hayashi   use parameter instead of define
// ver.1.2  2007.xx.xx  K. Hayashi   use begin end for all statement
//------------------------------------------------------------
//
module bound_flasher ( clk, rst_n, flick, a_lamp ) ;
// parameters
parameter INTL      = 3'b000 ; // initial state
parameter UP_HIGH  = 3'b001 ; // going up from bottom to top
parameter DWN_HIGH = 3'b010 ; // coming down from top to bottom
parameter HIGH_LOW = 3'b011 ; // go to UP_HIGH or UP_LOW depending on flick
parameter UP_LOW    = 3'b100 ; // going up from bottom to middle
parameter DWN_LOW  = 3'b101 ; // coming down from middle to bottom
parameter LOW_INTL = 3'b110 ; // go to UP_LOW or INTL depending on flick
//
parameter MX_LP = 8 ;  // number of lamps
parameter TP_TP = MX_LP-1 ; // top turning point
parameter MD_TP = 3 ;  // middle turning point
parameter FF_DLY = 1 ; // delay for FF to avoid racing
```

```
input clk, rst_n ;
input flick ;
output [MX_LP-1:0] a_lamp ;

wire clk, rst_n ;
wire flick ;
reg [MX_LP-1:0] a_lamp ;  // FF for lamps

// internal variables
reg [2:0] f_state ;       // FF for state
reg [2:0] next_f_state ;    // non-FF
reg [MX_LP-1:0] next_a_lamp ; // non-FF

// state FF definition
always @ ( posedge clk or negedge rst_n ) begin
   if ( rst_n==1'b0 ) begin           // initialize state
         f_state <= #FF_DLY  INTL ;
   end
   else begin
         f_state <= #FF_DLY  next_f_state ;
   end
end
```

```
always @  ( f_state or flick or a_lamp )  begin
  case ( f_state )
   INTL    : begin
               next_f_state = ( flick )? UP_HIGH : f_state ;
            end
   UP_HIGH : begin
               next_f_state = ( a_lamp[TP_TP-1] )? DWN_HIGH : f_state ;
            end
  DWN_HIGH : begin
               next_f_state = ( a_lamp[1] )? HIGH_LOW : f_state ;
            end
  HIGH_LOW : begin
               next_f_state = ( flick )? UP_HIGH : UP_LOW ;
            end
   UP_LOW : begin
               next_f_state = ( a_lamp[MD_TP-1] )? DWN_LOW : f_state ;
            end
  DWN_LOW : begin
               next_f_state = ( a_lamp[1] )? LOW_INTL : f_state ;
           end
 LOW_INTL : begin
               next_f_state = ( flick )? UP_LOW : INTL ;
           end
  default : begin   // set x for debug
               next_f_state = 3'bxxx ;
           end
  endcase
end
```

```
always @ ( posedge clk or negedge rst_n )   begin       // lamp control logic
  if ( rst_n==1'b0 ) begin              // clear lamp on reset
         a_lamp <= #FF_DLY  { MX_LP{1'b0} } ;
  end
  else begin                        // set lamp
         a_lamp <= #FF_DLY  next_a_lamp ;
  end
end

always @   ( f_state or flick or a_lamp )   begin
  case ( f_state  )
        INTL : begin
                 next_a_lamp = ( flick )? { { (MX_LP-1){1'b0} }, 1'b1 } : a_lamp ;
               end
      UP_HIGH,
       UP_LOW,
    HIGH_LOW : begin
                 next_a_lamp = a_lamp <<1 ; // up one step
               end
     DWN_HIGH,
     DWN_LOW : begin
                 next_a_lamp = a_lamp >> 1 ; // down one step
               end
    LOW_INTL : begin
                 next_a_lamp = ( flick )? a_lamp << 1 : { (MX_LP-1){1'b0} } ;
               end
    default : begin   // error, set x for debug
                next_a_lamp = { MX_LP{1'bx} } ;
              end
  endcase
end

endmodule
```

Timing check



Before going into a structured design, let’s look for other styles of implementation.

A good engineer must have several ways to solve the problem.

If he or she can have only one solution, we can not tell if it is good or bad because it is the only we have and we have to take it, even if it is bad. We have no other choice.

However, if we have several solutions, we can compare them and can find which is better over the others.

Having alternative solution is very important for designers.

Now, let’s see solutions;
One, using a counter instead of shifting the on-lamp-bit.
Another, using an extended counter, reducing the number of states.

9.6 Another solutions using counter

Introducing a counter to represent on lamp position

(1) Basic-counter solution

Instead of manipulating a_lamp bit, we can introduce a counter such as lp_cnt.

if a_lamp[k]=1,
#k lamp is on.

↔

if lp_cnt = 0, all lamps off
else
  a_lamp[lp_cnt-1] is on.

Then, by using lp_cnt instead of a_lamp, we can get a new module.

The new module has the same state definition. The only difference between the new one and the old one is using a on lamp bit position or position counter. Therefore, the change can be said to be “minor change”.

The new module using counter solution can be obtained by applying the following replacement rule to the module bound_flasher.

a_lamp bit manipulation solution	lp_cnt counter solution
a_lamp <= { MX_LP{1'b0} } ;	lp_cnt <= 0 ;
next_a_lamp = (flick)? { { (MX_LP-1){1'b0} }, 1'b1 } : a_lamp ;	next_lp_cnt = (flick)? 1 : lp_cnt ;
next_a_lamp = a_lamp <<1 ;	next_lp_cnt = lp_cnt + 1 ;
next_a_lamp = a_lamp >>1 ;	next_lp_cnt = lp_cnt - 1 ;
(a_lamp[1])?	(lp_cnt == 2)?

	always @ (lp_cnt) begin
	a_lamp = 0 ;
	if (lp_cnt) begin
	a_lamp[lp_cnt-1]= 1’b1;
	end
	end

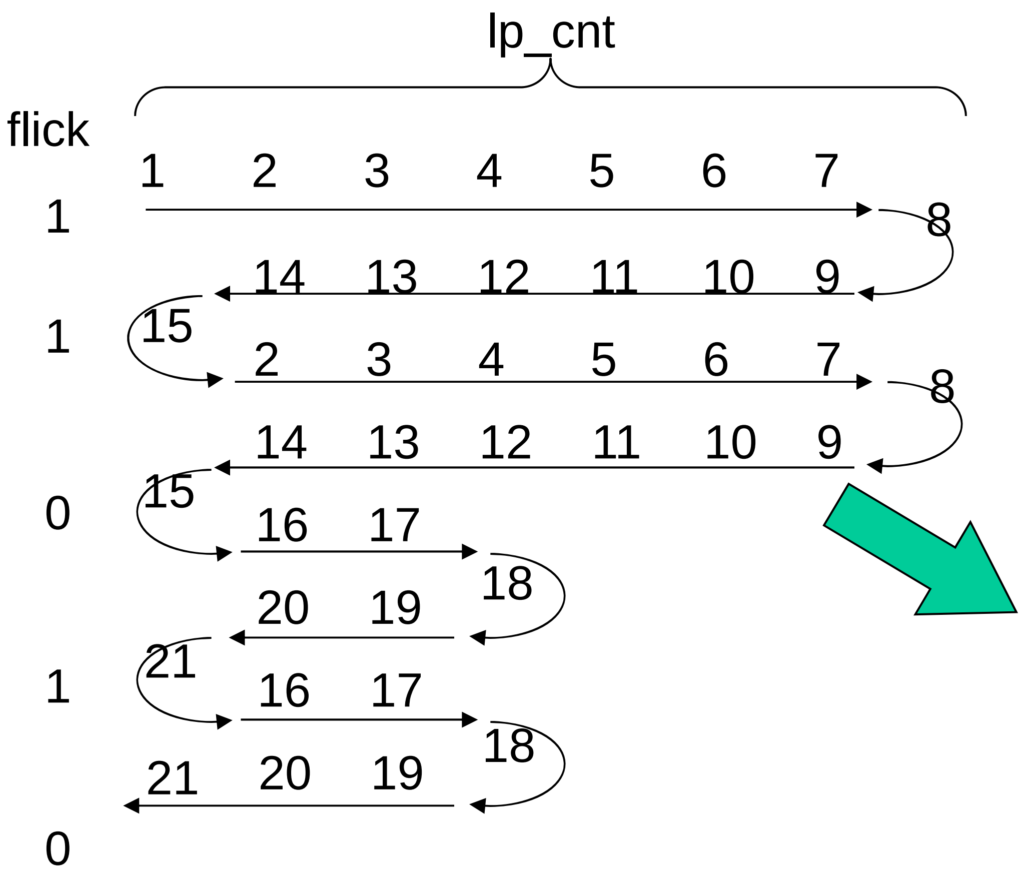
By replacing pieces of code on the left with those on the right, we can have a counter-solution. The one at the bottom must be added to map the counter to a_lamp signal.

Using a counter, sometimes results in smaller area if the number of lamps is large.
For example, if the number of lamps are 255, a_lamp needs 255-bits flip-flop. But lp_cnt needs only 8-bits flip-flop. It is obviously better to use counter in such cases.

Yet, another solution

(2) Extended-counter solution

Another solution using a counter is shown below. This idea can reduce the number of states very much. We can use as few as two states, INTL and RUN.



In INTL state, lp_cnt=0.
In RUN state, lp_cnt≠0.

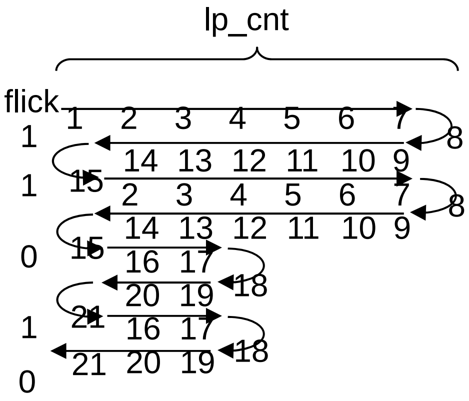
This idea is equivalent to introducing a new state variable which defines sub-state in RUN state,

STT for extended-counter solution

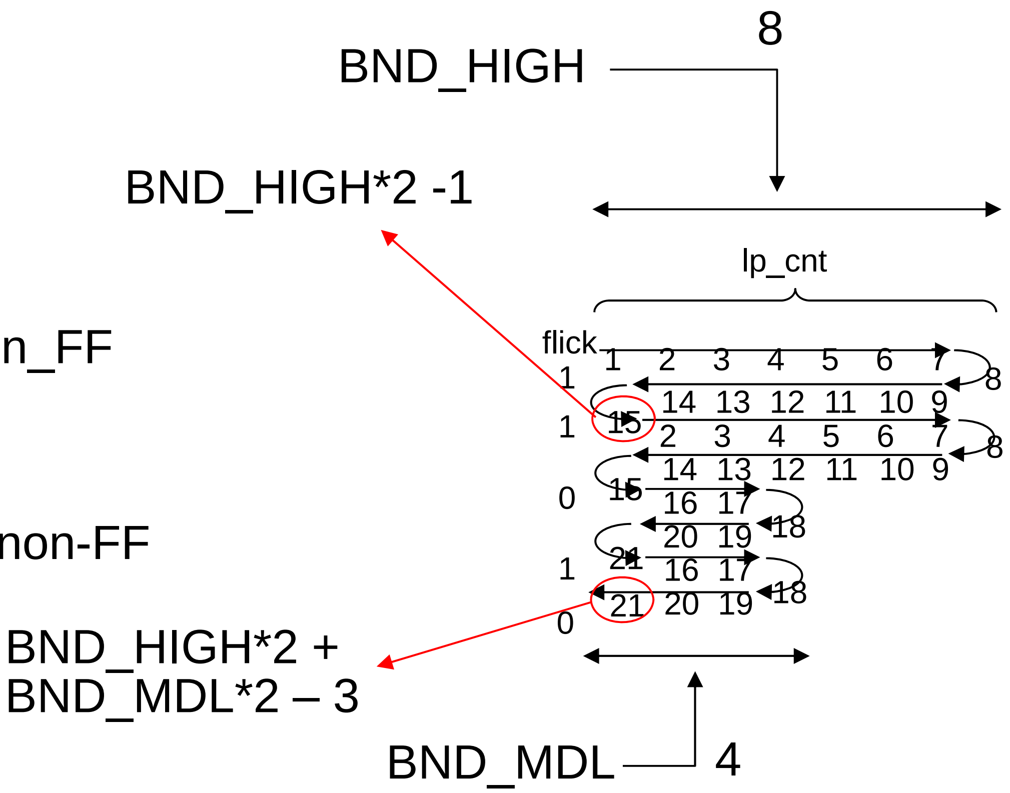
		ST_INTL	ST_RUN
rst_n=0		lp_cnt=0 ⇨ST_INTL	lp_cnt=0 ⇨ST_INTL

rst_n=1	flick=0	no operation	if lp_cnt=21 then
			lp_cnt = 0
			⇨ST_INTL
			else
			lp_cnt = lp_cnt +1

	flick=1	lp_cnt=1
		⇨ST_RUN
			if lp_cnt=15 then
			lp_cnt = 2
			else
			if lp_cnt=21 then
			lp_cnt = 16
			else
			lp_cnt = lp_cnt+1



```
module bnd_flsh_excntr ( clk, rst_n, flick, a_lamp ) ;
parameter NUM_LP = 8 ;
parameter ST_INTL = 1'b0 ;
parameter ST_RUN = 1'b1 ;
parameter BND_HIGH = NUM_LP ;
parameter BND_MDL = 4 ;
parameter BW_CNT = 4 ;
// 2**BW_CNT must be larger
// than BND_HIGH*2 + BND_MDL*2 – 3
parameter FF_DLY = 1 ;

input clk, rst_n ;
input flick ;
output [NUM_LP-1:0] a_lamp ;
wire clk, rst_n ;
wire flick ;
reg [NUM_LP-1:0] a_lamp ; // non_FF

// internal variable
reg [BW_CNT:0] lp_cnt ; // FF
reg [BW_CNT:0] next_lp_cnt ; // non-FF
reg state ;  // FF
wire next_state ;
```



```
// ***********  state control logic  ************
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n==1'b0 ) begin
     state <= <=#FF_DLY ST_INTL ;
  end
  else begin
     state <=#FF_DLY next_state ;
  end
end
assign next_state = (flick)? ST_RUN : state ;

// counter control logic
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n==1'b0 ) begin
     lp_cnt <= <=#FF_DLY 0 ;
  end
  else begin
     lp_cnt <=#FF_DLY next_lp_cnt ;
  end
end
```

```
// ***********  next_lp_cnt control logic  ************
always @ ( state or flick or lp_cnt ) begin
  case ( state )
     ST_INTL : begin
                   next_lp_cnt = (flick)? 1 : lp_cnt ;
               end
     ST_RUN : begin
           if ( flick ) begin
                if ( lp_cnt== BND_HIGH*2-1 )  begin  // at bottom?
                   next_lp_cnt = 2 ;
                end
                else begin   // if at second bottom, bound again, else forward 1 step
                   if (  lp_cnt== BND_HIGH*2 + BND_MDL*2 -3 ) begin
                        next_lp_cnt = BND_HIGH*2 ;
                   end
                   else  begin
                        next_lp_cnt = lp_cnt + 1 ;
                   end
                end
           end
           else begin // if second bottom, finish, else forward 1 step
              next_lp_cnt =(lp_cnt== BND_HIGH*2 + BND_MDL*2 -3 )?  0 : lp_cnt + 1 ;
           end
         end
       default : begin next_lp_cnt = {BW_CNT{1'bx}} ;  end
    endcase
end
```

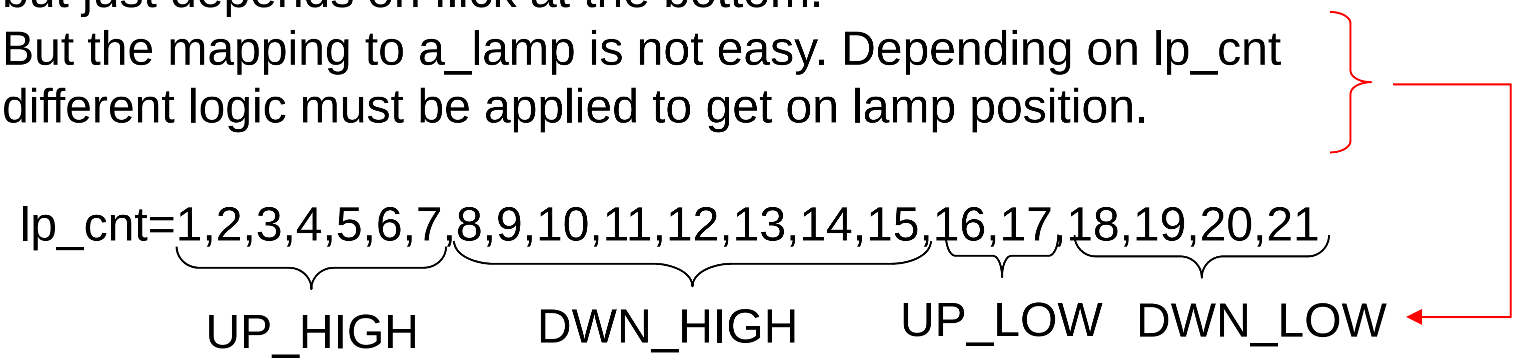
```
// mapping logic from lp_cnt to a_lamp
reg [BW_CNT:0] ix_cnt ; // bit position counter; k if #k-1 lamp on
always @ ( lp_cnt ) begin  // convert lp_cnt to bit position counter
  if ( lp_cnt <= BND_HIGH ) begin // going up high
    ix_cnt = lp_cnt ;
  end
  else begin
    if ( lp_cnt < BND_HIGH*2 ) begin // going down from high
      ix_cnt = BND_HIGH*2 - lp_cnt ;
    end
    else begin
       if ( lp_cnt < BND_HIGH*2+BND_MDL-2 ) begin // going up low
           ix_cnt = lp_cnt +2 - BND_HIGH*2 ;
       end
       else begin // going down from low
           ix_cnt = BND_HIGH*2 + BND_MDL*2 -2 - lp_cnt ;
       end
    end
  end
end
always @ ( ix_cnt ) begin  // bit position counter to bit position
  a_lamp = 0 ;
  if ( ix_cnt ) begin
    a_lamp[ix_cnt-1] = 1'b1 ;
  end
end
endmodule
```

The code on the previous pages shows us that there are only two states but lp_cnt works as a sub-level-state in ST_RUN state.

top level state	ST_INTL	ST_RUN
sub-level state	none	lp_cnt=1,2,3,4,5, ,,,, 19,20,21

Behavior of lp_cnt is simple. It does not depend on up nor down, but just depends on flick at the bottom.

But the mapping to a_lamp is not easy. Depending on lp_cnt different logic must be applied to get on lamp position.



lp_cnt=1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21

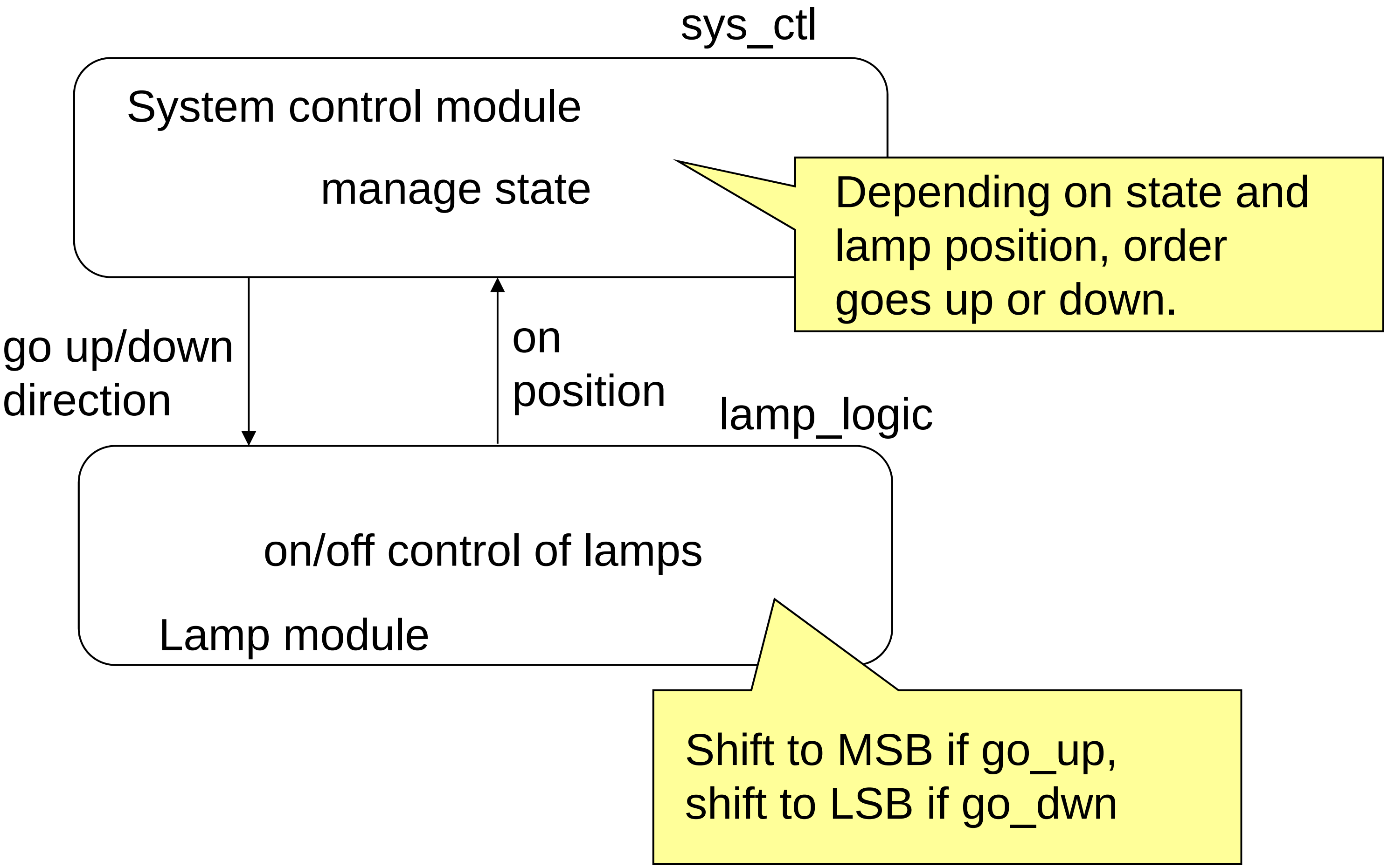
UP_HIGH DWN_HIGH UP_LOW DWN_LOW

```
assign st_up_high = ( (1 <= lp_cnt ) & (7 >= lp_cnt) ) ;
assign st_dwn_high = ( (8 <= lp_cnt ) & (15 >= lp_cnt) ) ;
assign st_up_low = ( (16 <= lp_cnt ) & (17 >= lp_cnt) ) ;
assign st_dwn_low = ( (18 <= lp_cnt ) & (21 >= lp_cnt) ) ;
assign sub_state={ st_up_high, st_dwn_high, st_up_low, st_dwn_low} ;
```

Whether this idea of an extended-counter solution means a good design or not depends on resulting silicon size, power consumption, etc.
But in general, we do not have to make a large effort to reduce the number of states unless it becomes too large to handle. Therefore, use your effort to make everything simple, not to reduce the number of states.

9.7 Structured design

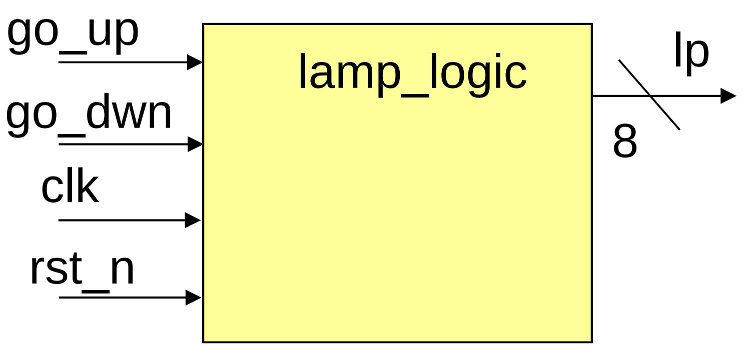
The design on the previous pages is called flat design because everything is written in one module and there is almost no hierarchy. Now let's design bound flasher in two blocks.



Depending on state and lamp position, order goes up or down.

Shift to MSB if go_up, shift to LSB if go_dwn

Lamp module must look like below.



```
always @ ( go_up or go_dwn or lp ) begin
if ( go_up ) begin
  if ( lp == 0  )      next_lp =  1  ;
  else                 next_lp = lp << 1 ;
end
else begin
   if ( go_dwn )   next_lp = lp >> 1 ;
   else               next_lp = lp ;
end
```

If ordered to go up while off, then turn on #0 lamp.

If ordered to go up while on, then shift to MSB.

If ordered to go down, shift to LSB.

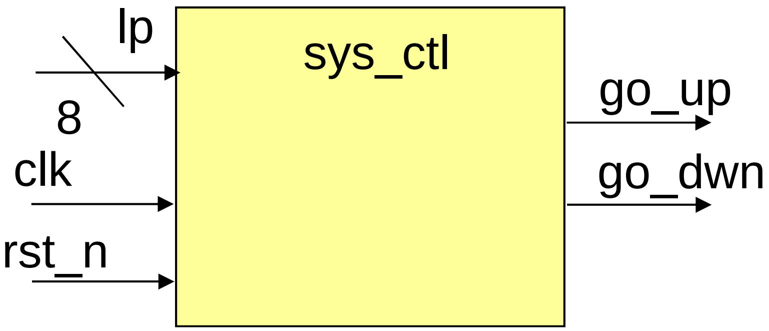
If no order, stay in current position.

The above logic can be extended to lamp_logic module on the next page.

```
module lamp_logic ( clk, rst_n, go_up, go_dwn, lp ) ;
parameter FF_DLY = 1 ;
paramater MX_LP = 8 ;
input clk, rst_n ;
input go_up, go_dwn ;
output [MX_LP -1:0]  lp ;

wire clk, rst_n ;
wire go_up, go_dwn ;
reg [MX_LP -1:0] lp ;    // FF

reg [MX_LP -1:0] next_lp ;  // non-FF

always @ ( posedge clk or
                         negedge rst_n ) begin
if ( rst_n==1'b0 ) begin
   lp <=#FF_DLY   { MX_LP { 1'b0 } } ;
end
else begin
  lp <=#FF_DLY    next_lp ;
end
end
```

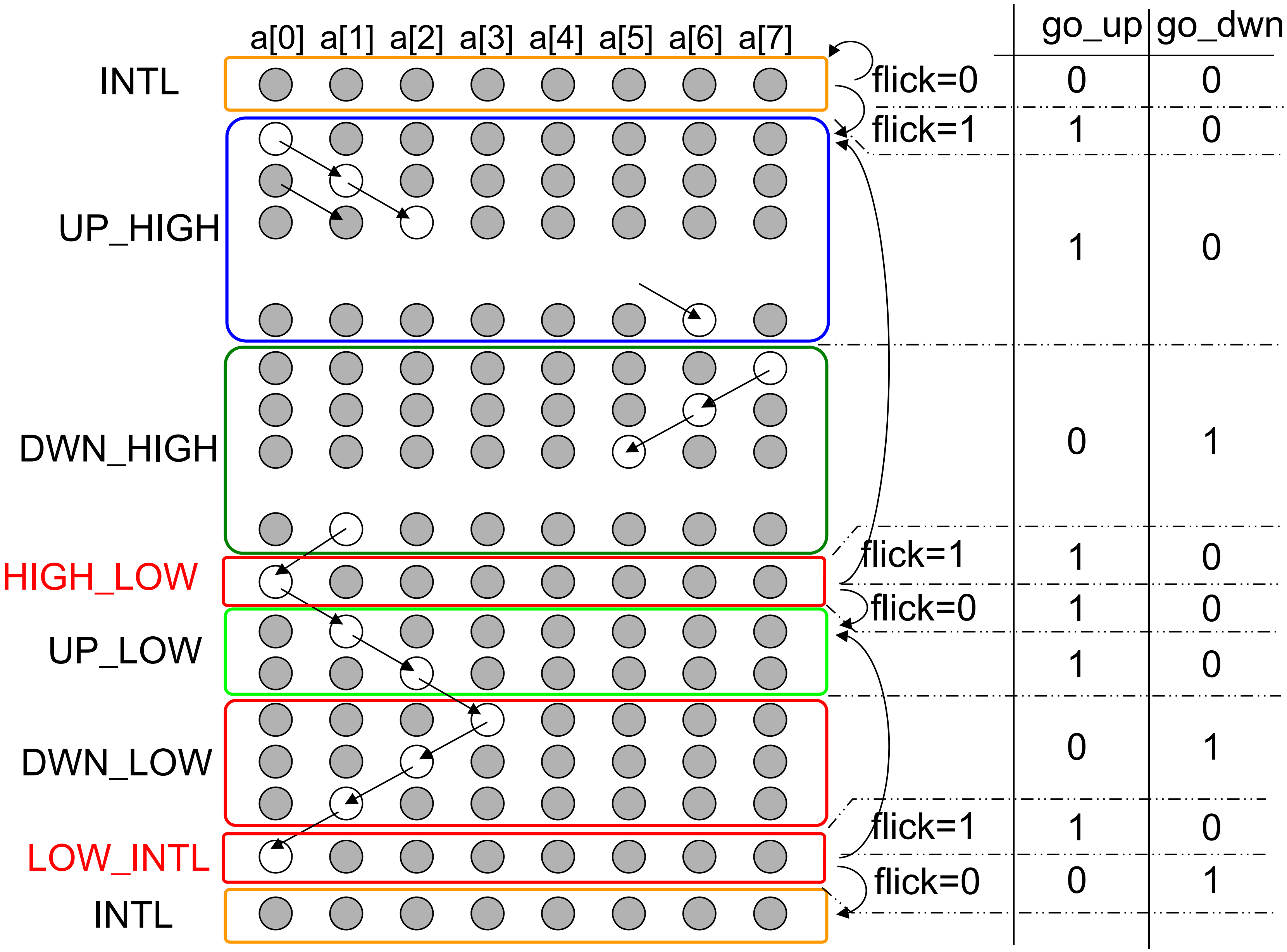
```
always @ ( go_up or go_dwn or lp ) begin
  if ( go_up ) begin
    if ( lp == { MX_LP { 1'b0 } }  ) begin
       next_lp = { { (MX_LP-1) { 1'b0 } }, 1'b1 }  ;
    end
    else begin
       next_lp = lp << 1 ;
    end
  end
  else begin
    if ( go_dwn ) begin
       next_lp = lp >> 1 ;
    end
    else begin
       next_lp = lp ;
    end
  end
end

endmodule
```

Next design sys_ctl module.



Principal part of sys_ctl must be written as below because go_up and go_dwn must be controlled as on the next page.

```
always @ ( state or flick ) begin
 case ( state )
   INTL :              go_up = ( flick )? 1 : 0 ;     go_dwn = 0 ;
   UP_HIGH, UP_LOW,
   HIGH_LOW :  go_up = 1 ;                            go_dwn = 0 ;
   DWN_HIGH, DWN_LOW : go_up = 0 ;     go_dwn = 1 ;
   LOW_INTL :   go_up = ( flick )? 1 : 0 ;      go_dwn = ( flick )? 0 : 1 ;
 endcase
end
```



	go_up	go_dwn
flick=0	0	0
flick=1	1	0
	1	0
	0	1
flick=1	1	0
flick=0	1	0
	1	0
	0	1
flick=1	1	0
flick=0	0	1

```
module sys_ctl ( clk, rst_n, flick, lp, go_up, go_dwn ) ;
// parameters
parameter INTL          = 3'b000 ; // initial state
parameter UP_HIGH    = 3'b001 ; // going up from bottom to top
parameter DWN_HIGH = 3'b010 ; // coming down from top to bottom
parameter HIGH_LOW = 3'b011 ; // go to UP_HIGH or UP_LOW depending on flick
parameter UP_LOW     = 3'b100 ; // going up from bottom to middle
parameter DWN_LOW  = 3'b101 ; // coming down from middle to bottom
parameter LOW_INTL   = 3'b110 ; // go to UP_LOW or INTL depending on flick

parameter MX_LP = 8 ;  // number of lamps
parameter TP_TP = MX_LP-1 ; // top turning point
parameter MD_TP = 3 ;  // middle turning point
parameter FF_DLY = 1 ; // delay for FF to avoid racing
// port definition
input clk, rst_n ;
input flick ;          // input signal to start system
input [MX_LP-1:0] lp ;
output go_up, go_dwn ;

wire clk, rst_n ;
wire flick ;
wire [MX_LP-1:0] lp ;  // lamps
reg go_up, go_dwn ;
//internal variables
reg [2:0] f_state ;   // FF for state
reg [2:0] next_f_state ; // non-FF
always @ ( posedge clk or negedge rst_n ) begin
   if ( rst_n==1'b0 ) begin            // initialize state
      f_state[2:0] <= #FF_DLY  INTL ;
   end
   else begin
      f_state[2:0] <= #FF_DLY  next_f_state[2:0] ;
   end
end
```

```
always @  ( f_state or flick or lp )  begin
  case ( f_state[2:0]  )
   INTL    : begin
              next_f_state[2:0] = ( flick )? UP_HIGH : f_state[2:0] ;
            end
   UP_HIGH : begin
              next_f_state[2:0] = ( lp [TP_TP-1] )? DWN_HIGH : f_state[2:0] ;
            end
  DWN_HIGH : begin
              next_f_state[2:0] = ( lp [1] )? HIGH_LOW : f_state[2:0] ;
            end
  HIGH_LOW : begin
              next_f_state[2:0] = ( flick )? UP_HIGH : UP_LOW ;
            end
   UP_LOW : begin
              next_f_state[2:0] = ( lp [MD_TP-1] )? DWN_LOW : f_state[2:0] ;
            end
  DWN_LOW : begin
              next_f_state[2:0] = ( lp [1] )? LOW_INTL : f_state[2:0] ;
            end
 LOW_INTL : begin
              next_f_state[2:0] = ( flick )? UP_LOW : INTL ;
            end
   default : begin
              next_f_state[2:0] = 3'bxxx ; // for debug
            end
 endcase
end
```

```
always @ ( f_state or flick ) begin
 case ( f_state )
   INTL : begin
               go_up = ( flick )? 1 : 0 ;
               go_dwn = 0 ;
            end
   UP_HIGH, UP_LOW,
   HIGH_LOW : begin
               go_up = 1 ;
               go_dwn = 0 ;
            end
   DWN_HIGH, DWN_LOW : begin
               go_up = 0 ;
               go_dwn = 1 ;
            end
   LOW_INTL : begin
               go_up = ( flick )? 1 : 0 ;
               go_dwn = ( flick )? 0 : 1 ;
            end
    default : begin
                go_up = 1'bx ;
                go_dwn = 1'bx ;
           end
 endcase
end

endmodule
```

Now write the top module “bound_flasher”.

```
// File Name      : bound_flasher_v2.v
// Module Name  : bound_flasher
// Function          : this logic flash lamps in sequence at every clock rise time
//                   lamp will go up and down.
// Note           :  structured design
// Author        : K. Hayashi (idxxxx)
//-------------------------------------------------------------
// History
// Version  Date        Author       Description
// ver.1.0  2008.xx.xx  K. Hayashi   structured design
//----------------------------------------------------------
//
module bound_flasher ( clk, rst_n, flick, a_lamp ) ;
parameter MX_LP = 8 ;  // number of lamps
input clk, rst_n ;
input flick ;
output [MX_LP-1:0] a_lamp ;
wire clk, rst_n ;
wire flick ;
wire [MX_LP-1:0] a_lamp ;

sys_ctl sys_ctl_01 ( .clk(clk), .rst_n(rst_n), .flick(flick), .lp(a_lamp),
                          .go_up(go_up), .go_dwn(go_dwn) ) ;
lamp_logic lamp_logic_01 ( .clk(clk), .rst_n(rst_n), .go_up(go_up),
.go_dwn(go_dwn),
                      .lp(a_lamp) ) ;
endmodule
```

10. Appendix

10.1 Design tips

(1) Bugs are where you do not think they are.
(2) Bugs appear when things happen in the way you never think of.
(3) Bad things happen when you wish them to never happen.
(4) Always confirm when you are not sure. Do not design on your own speculation.
(5) Make a document describing why you do so to let others understand your design.
(6) Simple is best. Sophisticated logic nobody can understand is nothing but garbage.
(7) Always think of others who work with you now and in the future. Do not leave any burden but ease of reuse to others.
(8) Look for simple and well structured design, and document less but sufficient for others.

Design review tips

(1) Check consistency

If somewhere in your code, sig_a is set. Then look for a similar part of the program and check if sig_a is set also. If not set, check the reason. If there is specific reason, it is OK. If not, it may be a bug.
Consistency check is a most powerful way to check your code. However your code must be well organized to apply this check.
If you feel there are so many exceptions that there is no consistency in your code, then reconstruct your logic structure and rewrite the code.

(2) Check cases which are different from your assumption.

Create complement pattern of your test data. Check whether such a combination or sequence of test data is possible or not. If possible, check your logic is immune to such test data.

(3) Check your code around which some processing mode or pattern changes.

When a variable becomes minimum, maximum, overflow, sign-bit on, go over some boundary, hit some condition, any change of mode / pattern, check your code is ready for such changes.

Design review tips

(4) Check your code handling singular case whether it is really singular or not.

Example; In cyclic buffer, suppose read pointer and write pointer both pointing to the top of a buffer. If your code handle such case as a singular case from the view point of buffer full or empty, your code may have a bug. Because read pointer = write pointer = 0 is same as the situation that read pointer = write pointer = 5. There is no reason to think the case, both are 0, as special.

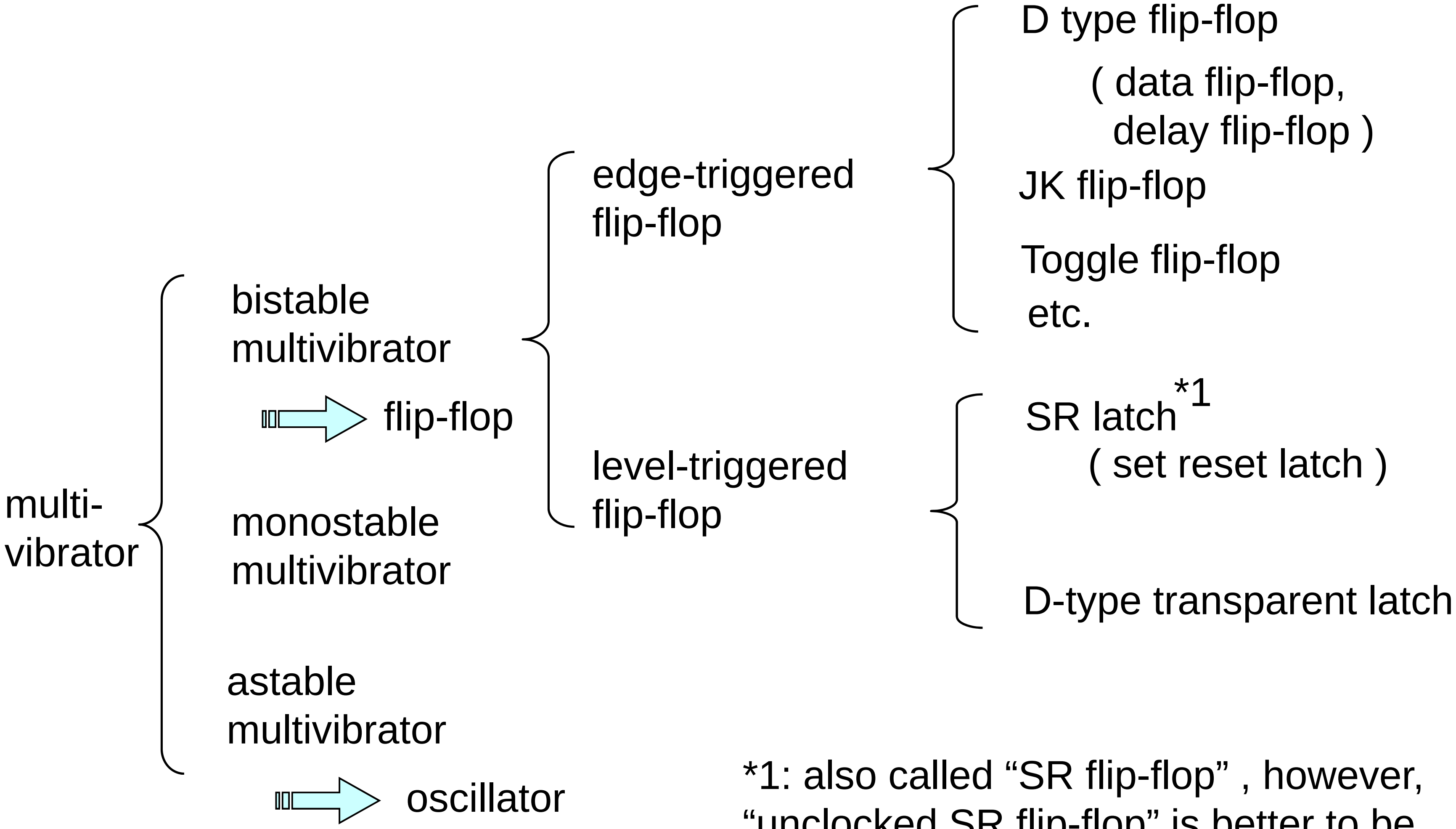
(5) Check timing.

Any data coming from a FF is one cycle delayed compared to its input signals. Verify the timing when data from FF is valid and available by drawing a time chart.

(6) Do not believe in the result of simulation.

Simulation covers just a part of your logic unless huge patterns of test data are applied. There are many cases where your test data unintentionally avoids a pitfall of your code unless you prepare the test data carefully.

10.2 Multi vibrator and FF



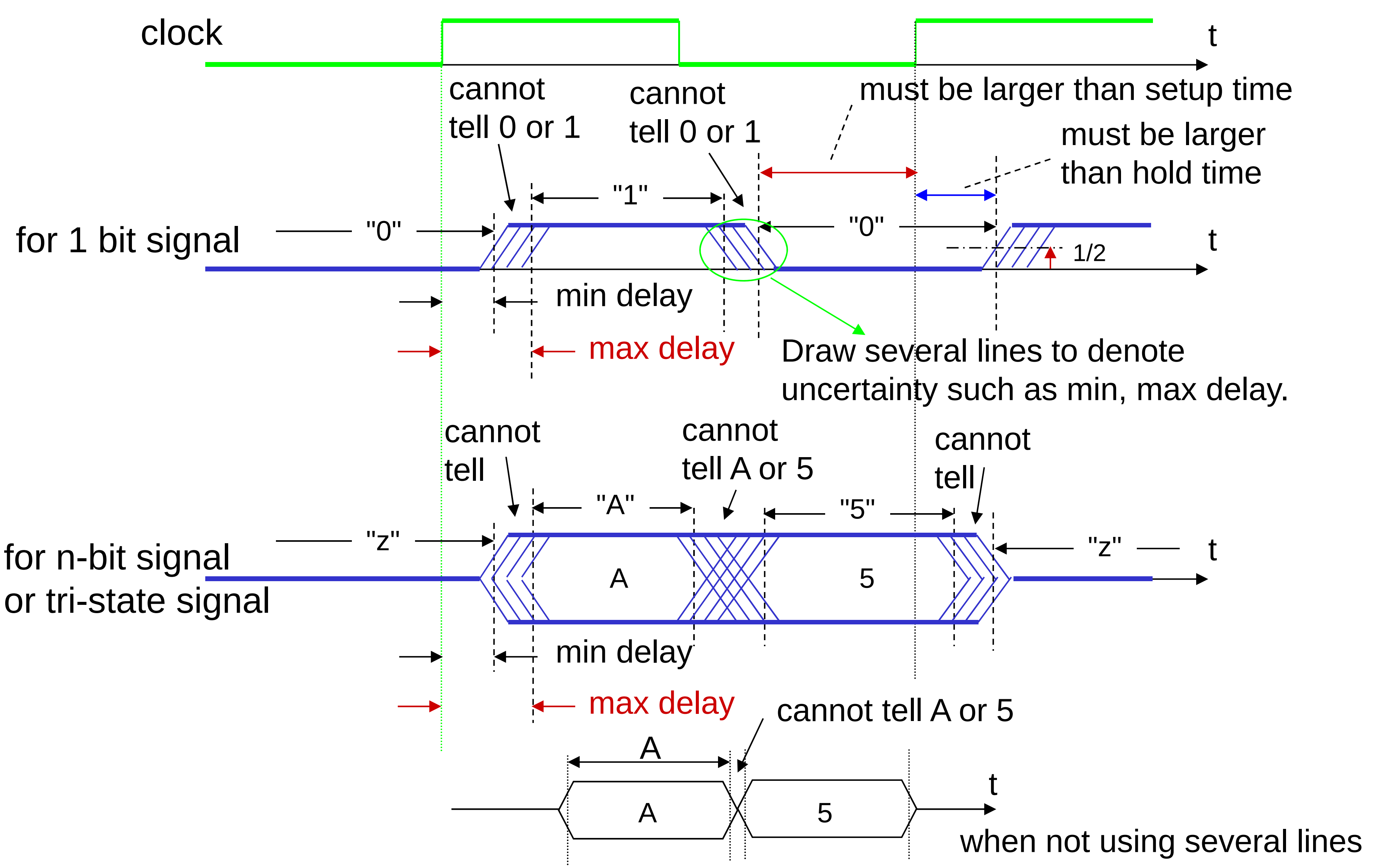
multi-vibrator

- bistable multivibrator → flip-flop
  - edge-triggered flip-flop
    - D type flip-flop (data flip-flop, delay flip-flop)
    - JK flip-flop
    - Toggle flip-flop
    - etc.
  - level-triggered flip-flop
    - SR latch[*1] (set reset latch)
    - D-type transparent latch
- monostable multivibrator
- astable multivibrator → oscillator

*1: also called “SR flip-flop” , however, “unclocked SR flip-flop” is better to be called “SR latch”. Because flip-flop is a circuit which has both data input and clock input.

10.3 Tips to draw time chart

When detailed and timing accurate chart is needed follow the drawing rules below.



Those are just an introduction.
Professional skills lie beyond this level,
but it is not far away.
Your effort will make them
come within your reach.

End of part I
to be continued to Part II