Verilog RTL programming self-learning text material

Logic Design Workshop

Verilog RTL programming practice

Mar. 2012

v2r04

Dr. Keijiro Hayashi

This text material covers the very basics of Verilog RTL programming.

You will be able to write RTL code by yourself after you finish this text material.

However, while you study basics of what RTL is and how to write RTL code through this text material, you must read “Logic Design Workshop part I” to review your understandings are correct and to know details about Verilog RTL design.

Do not think that you have got enough knowledge to design logics in RTL before you completed “Logic Design Workshop part I” .

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.

RTL programming practice

Index

1. Verilog RTL

1.1. What is RTL
1.2. Simulation and synthesis
1.3. RTL source code structure

2. Generate arbitrary signal and observe it

Exercise 2.1 1-bit waveform and initial
Exercise 2.2 Procedure and racing
Exercise 2.3 Vector waveform and initial
Exercise 2.4 Repeated waveform and always
Exercise 2.5 Repeated waveform and task
Exercise 2.6 How to use implicit event
Exercise 2.7 How to use named event
Exercise 2.8 Observe signals periodically
Exercise 2.9 Signals synchronized to clock
Exercise 2.10 Clock and signal timing

3. Combinational logic and test bench

Exercise 3.1 All-in-one module for testing 4-bit AND
Exercise 3.2 Test bench and target module
Exercise 3.3 Using two instances of one module
Exercise 3.4 How to use function
Exercise 3.5 Parameterize bit width
Exercise 3.6 Assign for combinational logic
Exercise 3.7 Always for combinational logic
Exercise 3.8 Sign bit and Concatenating operator
Exercise 3.9 Missing case item and latching
Exercise 3.10 Compare signed data in Verilog1995
Exercise 3.11 Arithmetic operation in Verilog1995
Exercise 3.12 Test bench and coverage

RTL programming practice

Index

4. Flip-flops and sequential logic

Exercise 4.1 Sequential lamps

Exercise 4.2 Counter with limiter

Exercise 4.3 Up down counter and STT

Exercise 4.4 Sequential accumulator with overflow checker

Exercise 4.5 Automatic test bench

Exercise 4.6 Serial bit pattern detector

Exercise 4.7 Data change range checker

Exercise 4.8 One byte data buffer

Exercise 4.9 Average of sequential four 4-bit data

Exercise 4.10 6-stage bound flasher

5. RTL coding style suitable for optimization

Exercise 5.1 For-loop and if statement

Exercise 5.2 On-bit-block counter

6. Advanced exercises

Exercise 6.1 Snake game

Exercise 6.2 LRU algorithm

Exercise 6.3 Two ring flasher

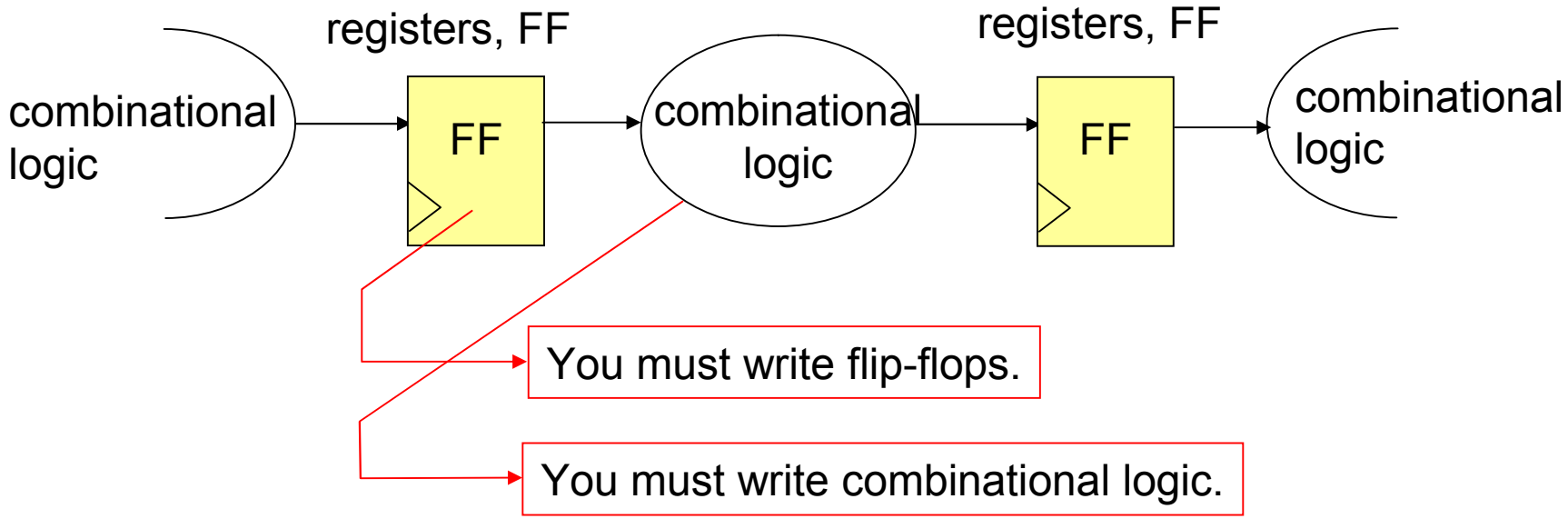
Exercise 6.4 Elevator

Exercise 6.5 Asynchronous data transfer

Exercise 6.6 Cyclic FIFO buffer

Chapter 1. Verilog RTL

1.1. What is RTL

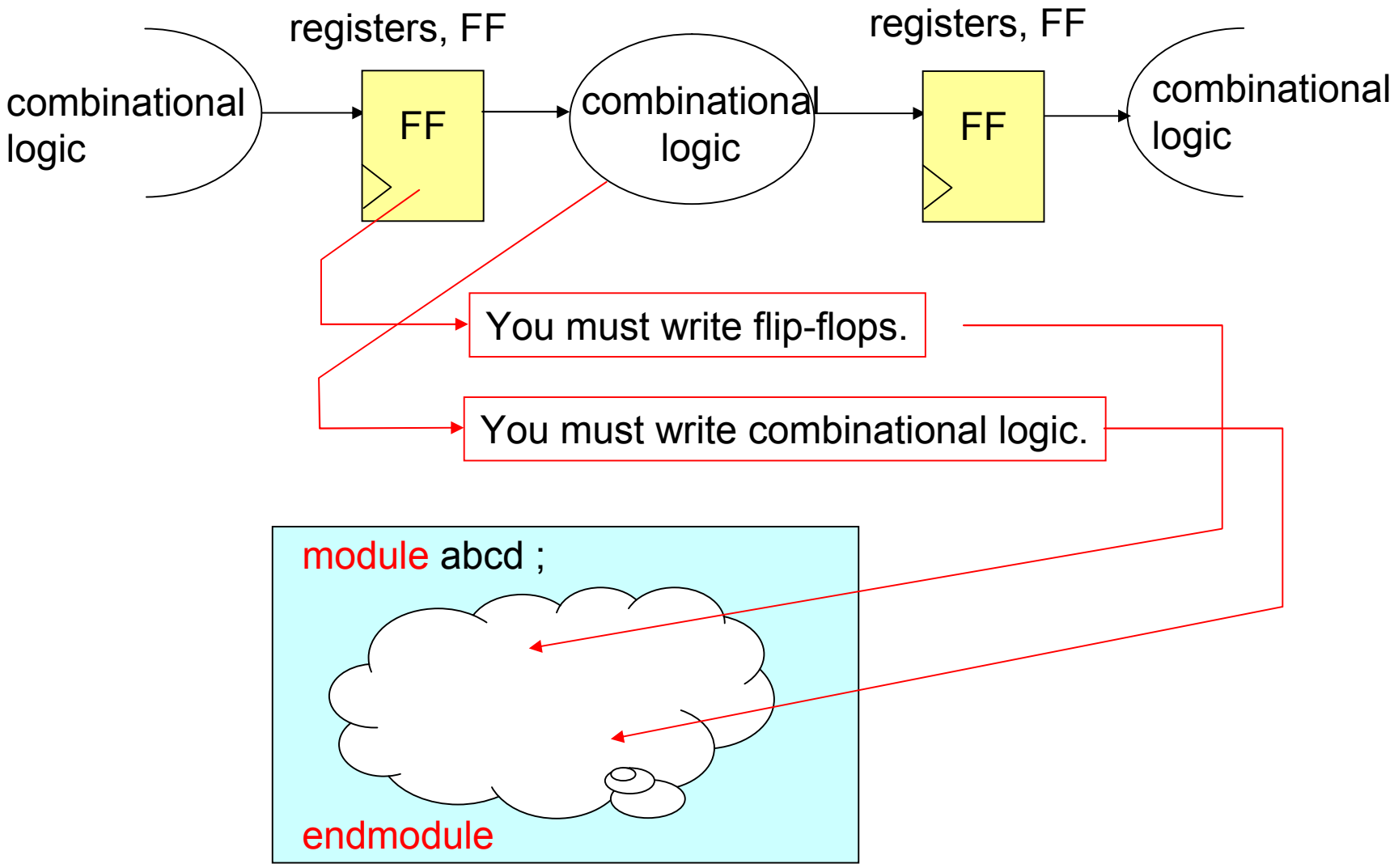


In RTL design, registers (usually they are flip-flops) and combinational logic are two major logic elements.

You have to describe them explicitly by using Verilog programming language.

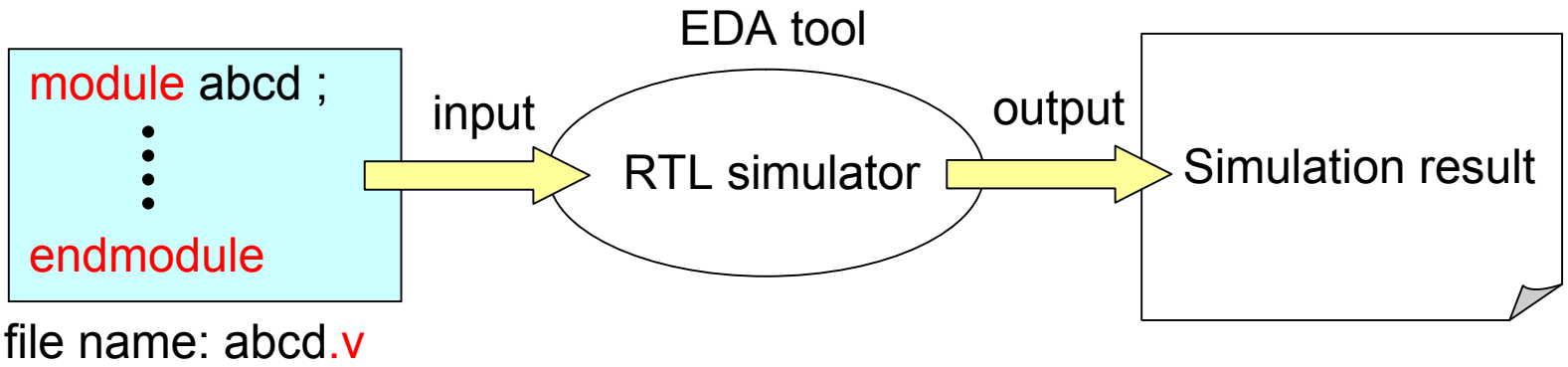
The description must be written in a module as shown on the next page.

RTL: Register Transfer Level
FF: Flip-Flop



file name: abcd.v

1.2. Simulation and synthesis



A file containing modules can be an input to an RTL simulator.

In general it is recommended to use a file name “abcd.v” for a file whose contents is a module named “abcd”. If there are several modules in one file, use the module name appearing at the top of the file.

There is no concept such as "main program in C language". A simulator can automatically find out which module to execute first.

EDA: Electronic Design Automation

To run a Verilog simulation, input the file abcd.v to an RTL simulator.

Verilog RTL source file

```
module abcd ;
    :
    :
endmodule
```

file name: abcd.v

Input the file to an RTL simulator

In Renesas developing environment;

> bs –source,,,, vcs -R abcd.v

VCS activating command

When using a free RTL simulator, cver ;

> cver abcd.v

cver activating command

VCS: Verilog simulator made by Synopsys

RTL programming rules

(1) Give everything a meaningful name. Do not give name such as temp1, temp2, ,,, ST1, ST2,,,, etc. Use names such as mem_rd_strb.

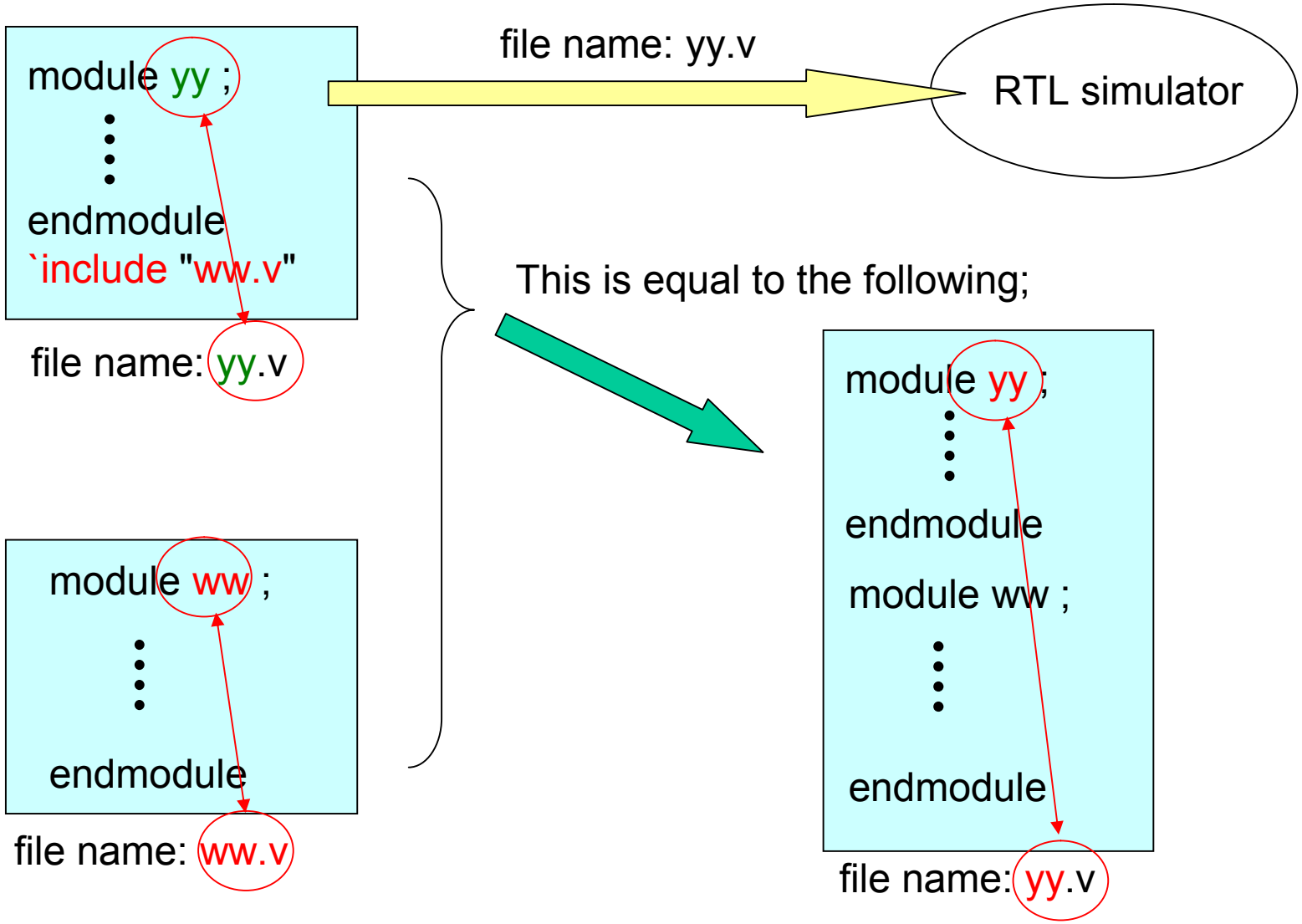
(2) If a module in a file has a name abcd, the file name must be abcd.v.

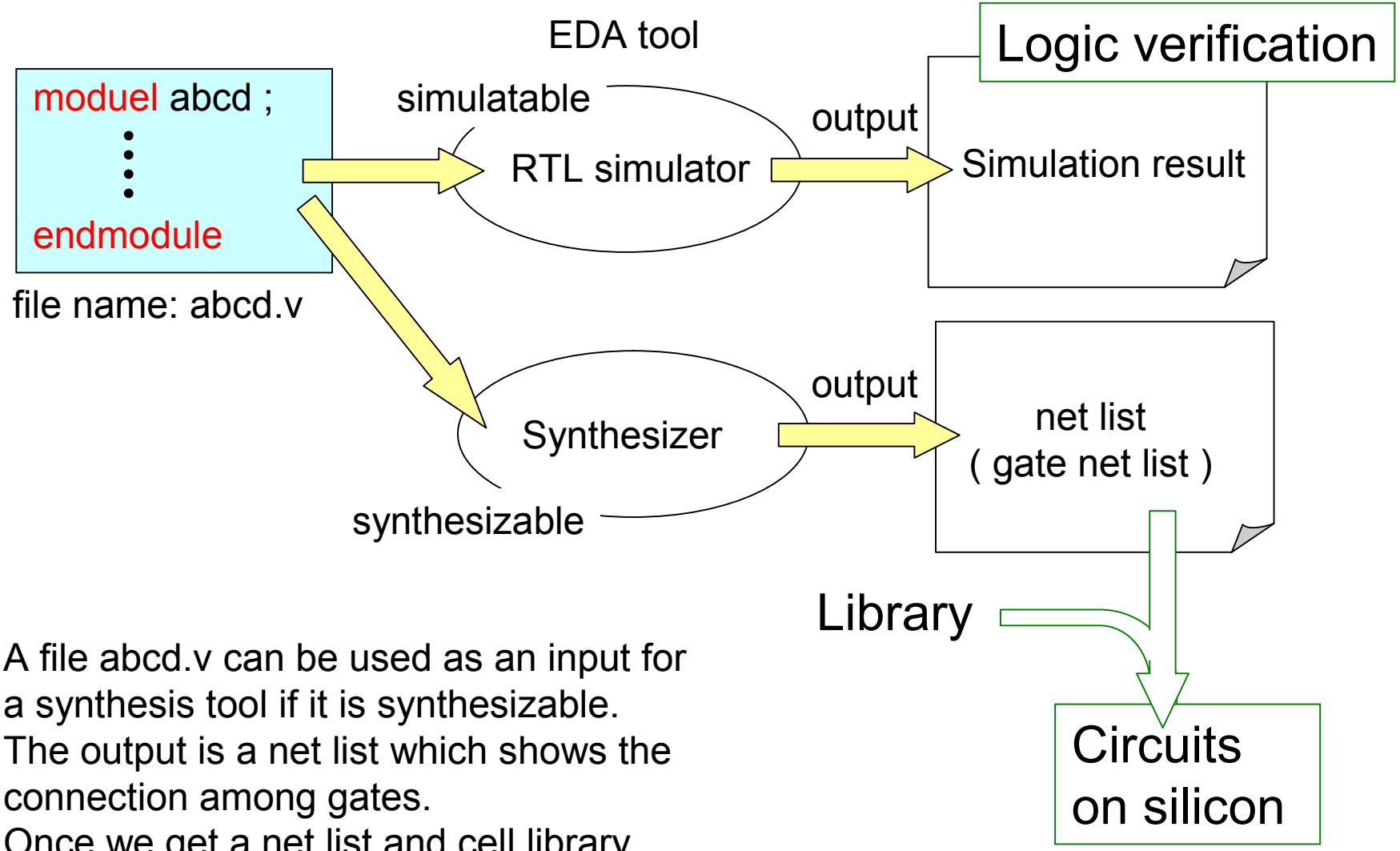
If there are several modules in a file, name the file using the module's name which appears at the top of the file.

(3) Do not append version/revision number to a module name. Version/revision number can be given in a file name.

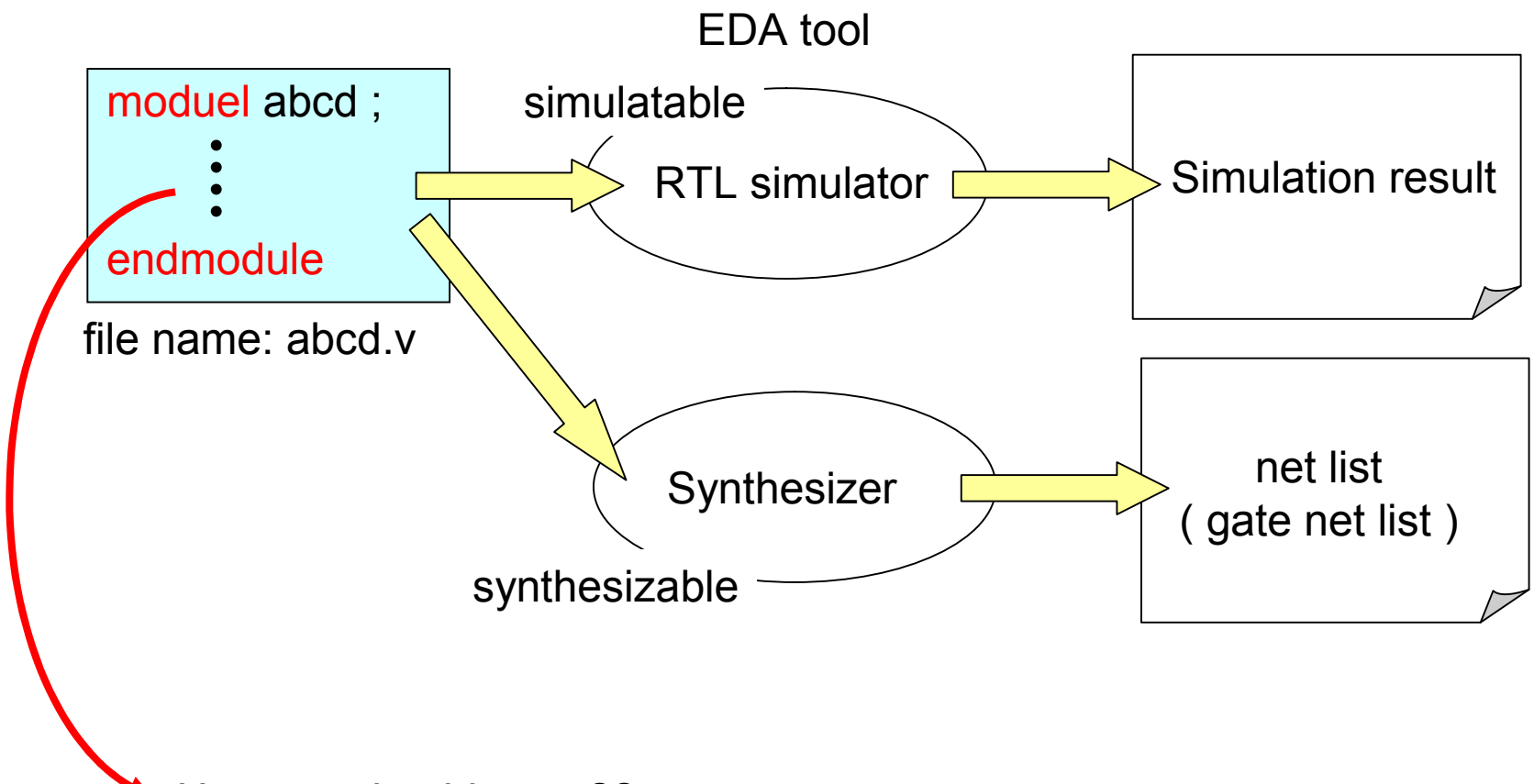
Example; module name abcd, file name abcd_v1r3.v

We can use compiler directive, `include, where ` is a grave accent, to incorporate other files as shown below.





A file abcd.v can be used as an input for a synthesis tool if it is synthesizable.
The output is a net list which shows the connection among gates.
Once we get a net list and cell library data, we can create a mask pattern for the module abcd.



How to write this part??

RTL format and types of lexical tokens in Verilog RTL

1.3. RTL source code structure

An RTL source code consists of sequence of tokens.

Format: free format

Position has no meaning.

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

same

Sequence matters.

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

different

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

equivalent

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

The types of lexical tokens:

Identifier, comment, space, numbers, etc.

The types of lexical tokens:

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.

abcd = efg + hij ;

ignored ignored ignored ignored

This sentence can be disintegrated into abcd, =, efg, +, hij, and ; using lexical tokens, =, +, and ;.

For this line, spaces are not needed to separate abcd, efg, and hij.

abc d=ef g+hij ;

These spaces are not ignored.
Therefore, this statement is illegal.

different

abcd=efg+hij ;

The types of lexical tokens:

Type	description	comment
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.

```
/* do not use
 this block comment because
it is not clear that this line is a comment or not

 */
```

```
// this is comment line
// use one-line comment is recommended
// because it is very clear that this line is a comment
```

The types of lexical tokens:

Type	description	comment
Operator	Perform specific manipulation over the operand(s).	Operators are single-, double-, or triple-character sequences.

sig_a = yy + in_a ; add operator

if (a == b) begin equality operator

>>> arithmetic shift right operator

The types of lexical tokens:

Type	description	comment
Number	Constant numbers can be specified as integer constants or real constants.	

integer, signed

64

real

2.3

16-bit hexa, unsigned

16’hFF05

8-bit binary, unsigned

8’b0101_1100 ;

4-bit binary, signed
This is equal to decimal -3.

4’sb1101

The types of lexical tokens:

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.

“abcd efg” ← OK

“abcd
efg” ← NG

sig_y = sig_w + “ab” ;

⇕

sig_y = sig_w + 16’h6162 ;

Use ¥n as shown below if you want to make it appear as two lines.

“abcd ¥n efg”

Note: ¥ is the same to back slash(\ ).

ASCII: American Standard Code for Information Interchange

The types of lexical tokens:

Type	description	comment
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.

sig_abc ——— simple identifier

\sig_abc
\ab+cd
} ——— escaped identifier

~~$sig_abc~~

~~8to4converter~~

converter8to4

Do not use escaped identifier.

About escape

Do not use escaped identifier.

abcd+efgh ← This can not be an identifier because "+" is a Verilog keyword.

\abcd+efgh ← This can be an identifier because escape character is attached at the beginning.

space

\abcd+efgh = abcd+efgh;

The signal named abcd+efgh is equal to abcd + efgh.

No space

\abcd+efgh=abcd+efgh;

This is a signal named abcd+efgh=abcd+efgh; .

The types of lexical tokens:

Type	description	comment
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.

Keywords are shown on the next page.

They are reserved to define the language constructs therefore we can not use them in the other context than they are supposed to be used.

HDL: Hardware Description Language

Verilog2001（keywords)

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

RTL programming rules

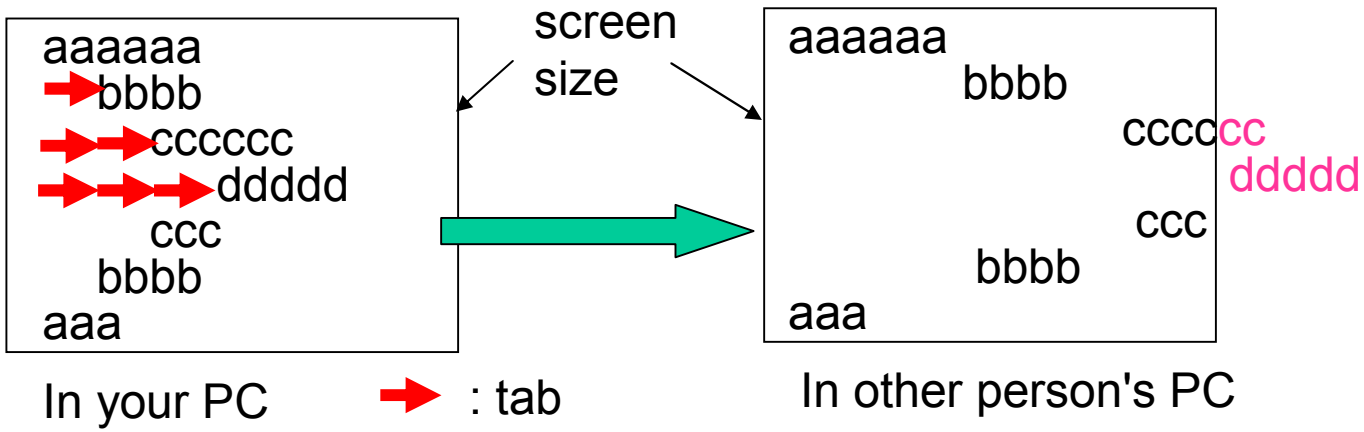
(1) Do not use tab for indentation. Use 2 spaces for indentation.

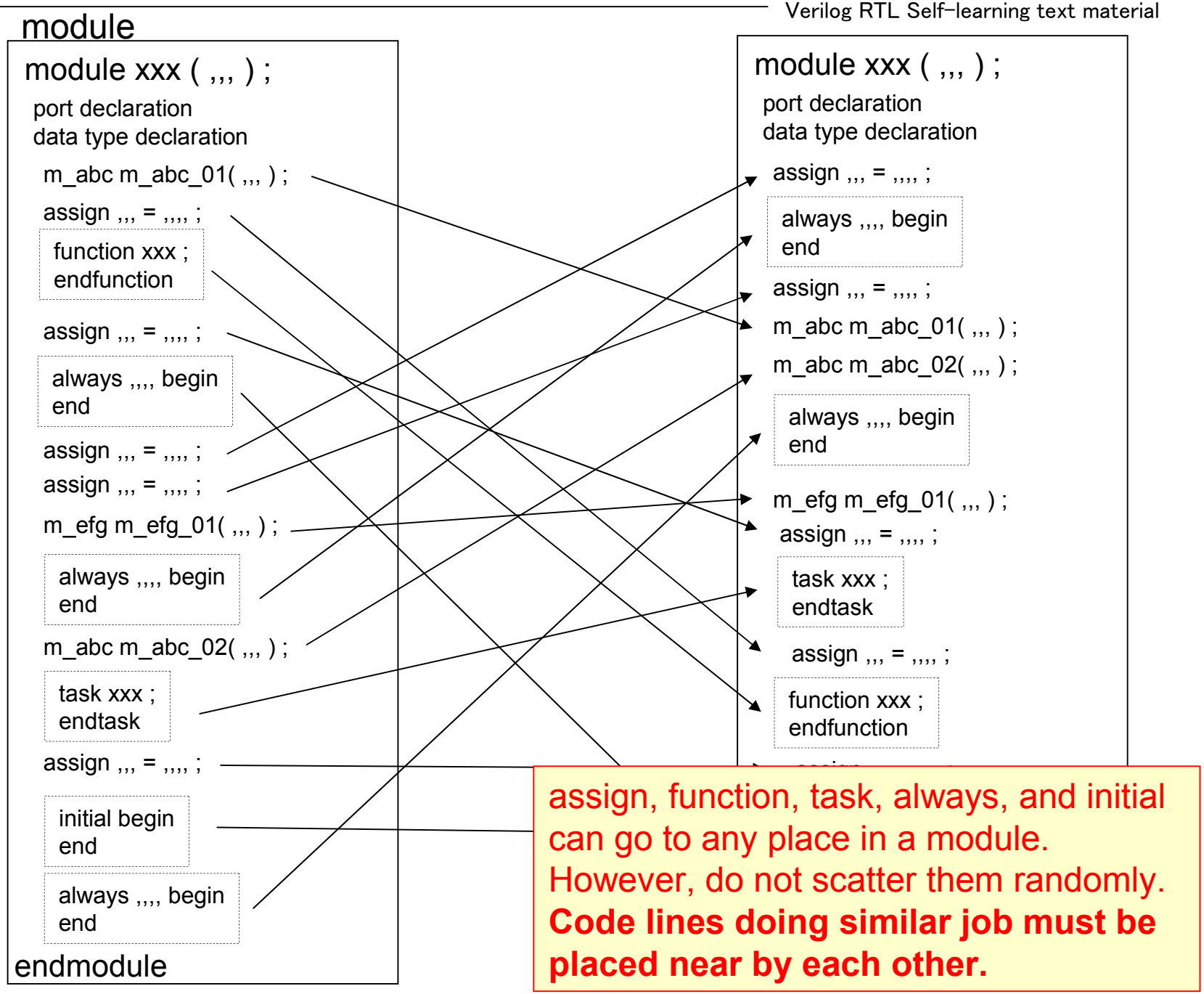
(2) Write header information to show general description of the module, author name, creation date, updated date and reasons, etc.

(3) Give comments using one line comment (//) so that other engineers can easily understand the code.

(4) Write a related code lines in one place. Do not scatter them in different places.

Tab settings are different from PC to PC. Even if your code using tabs looks beautiful in you PC, it may be very hard to see in other's PC.





assign, function, task, always, and initial can go to any place in a module. However, do not scatter them randomly. **Code lines doing similar job must be placed near by each other.**

RTL programming rules

assign, function, task, always, and initial can go to any place in a module but the following places are not allowed.
(1) procedures can not be declared in a procedure,
(2) modules can not be instanciated in a procedure,
(3) continuous assign, except procedural continuous assign, must not appear in procedures.

```
always ,,,, begin
    always ,,,, begin
    end
end
```

```
function xxx ;
    m_abc m_abc_01( ,,, ) ;
endfunction
```

```
initial ,,,, begin
    task xxx ;
    endtask
end
```

```
always ,,,, begin
    function xxx ;
    endfunction
end
```

```
function xxx ;
    assign ,,, = ,,,, ;
endfunction
```

```
initial ,,,, begin
    m_abc m_abc_02( ,,, ) ;
end
```

```
always ,,,, begin
    m_abc m_abc_01( ,,, ) ;
end
```

```
task xxx ;
    always ,,,, begin
    end
endtask
```

```
module ,,,, ;
    module xxx ;
    endmodule
endmodule
```

Chapter 2. Generate arbitrary signal and observe it

To verify Verilog RTL code using an RTL simulator, we need to input signals to the module under test and observe the output of the module to check if the code works correctly.

In this chapter, we will learn how to write RTL code to generate arbitrary signals and techniques to observe those signals. We do not care if the code is synthesizable or not because it does not go into silicon.

Verilog world has only 1 and 0 (and Hi-z) therefore we can not generate a waveform shown in Fig.1. We can only generate a wave form shown in Fig. 2.

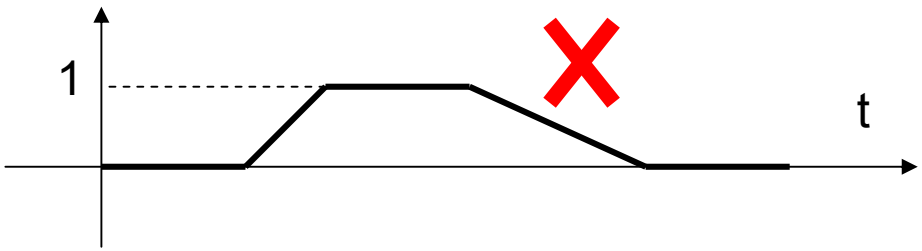


Fig.1 Waveform unable to generate

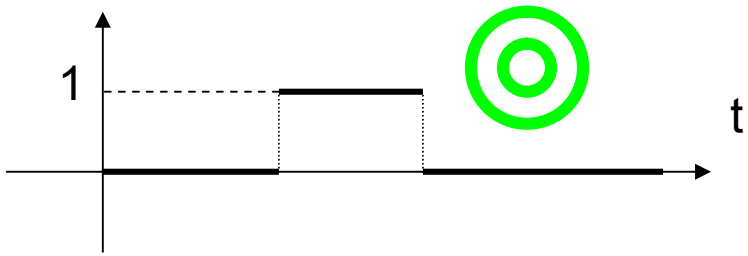
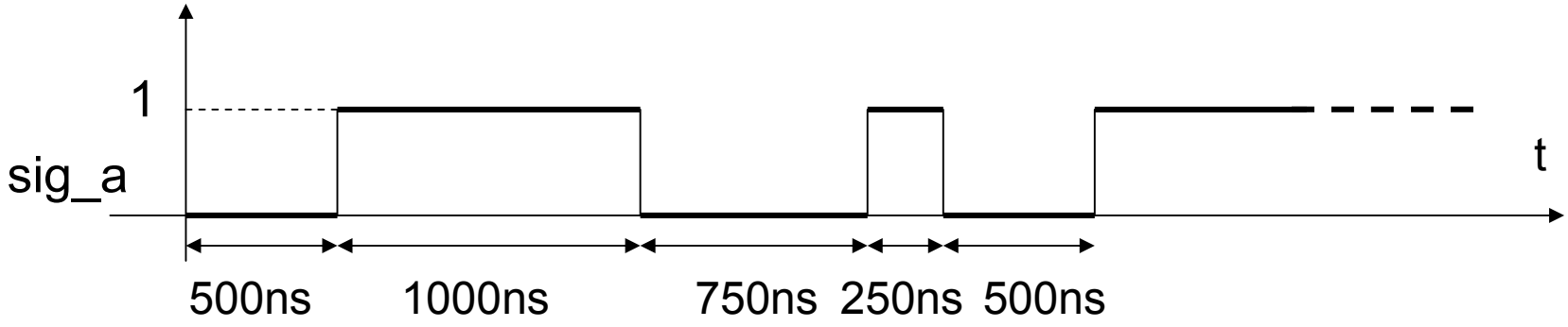


Fig.2 Waveform possible to generate

Ex 2-1. 1-bit waveform and initial: Write a logic block to generate and observe a signal (sig_a) of which waveform is shown below.



To create a signal which does not repeat the same pattern, we can use initial construct which is executed only once by a simulator.
To observe signals and display the values of them on a terminal, we can use a system task $monitor. It watches signals once it is activated, and if any change of watching signals are detected, it output message onto a terminal screen.

To set initial values, we can use "initial construct" as below.
In RTL simulation, an initial construct is executed only once at time 0.
And if there are several initial constructs, they are all executed at time 0.

```
initial sig_a = 0 ;   // in_a becomes 0 at time 0
initial sig_b = 1 ;   // in_b becomes 1 at time 0
```

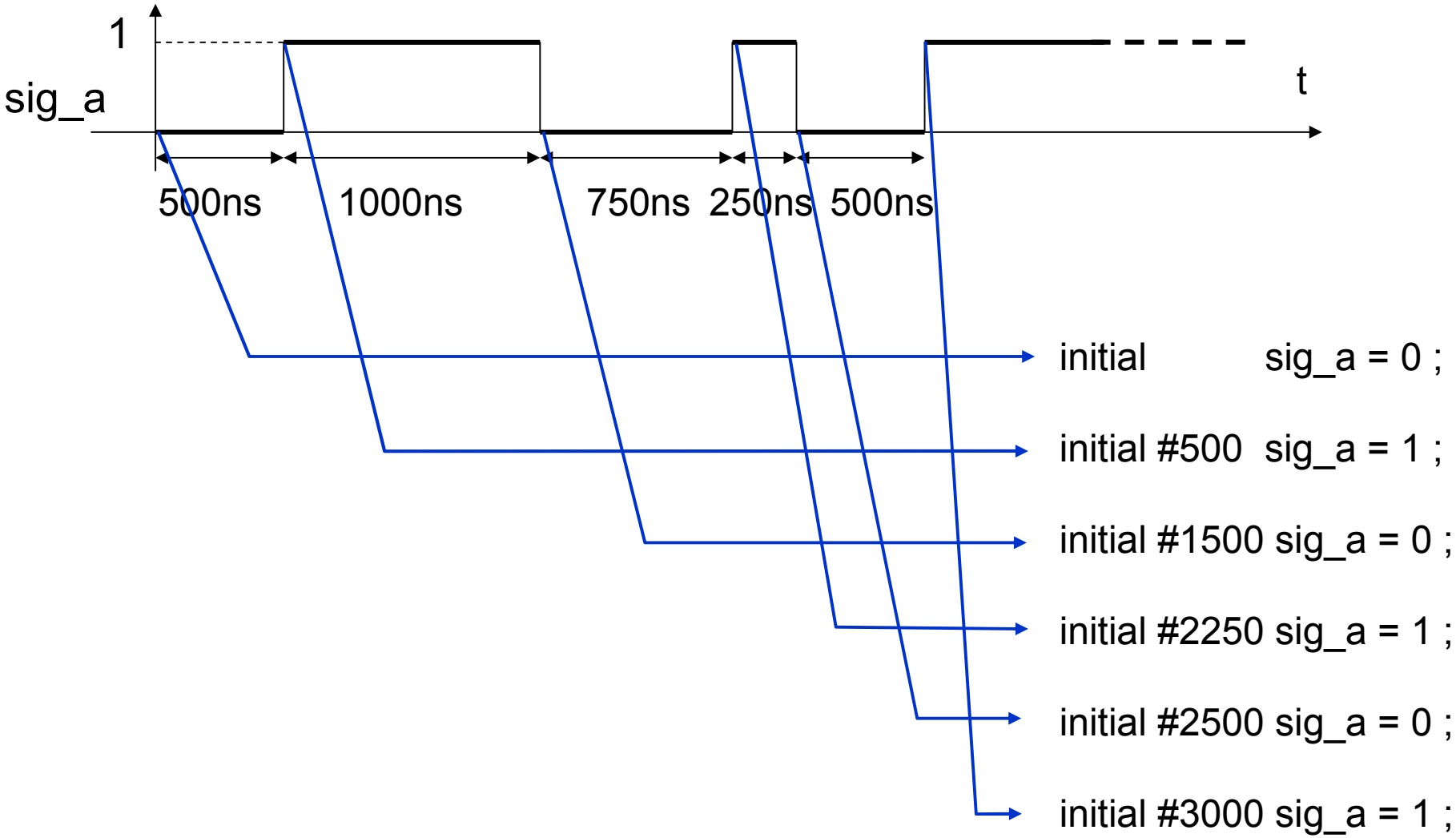
Executed only once at time 0.

The value of sig_a, before it is given the value 0, is x (unknown).

By using delay, specified by #, we can delay the execution of the assignment as below;

```
initial #10 sig_a = 0 ;   // in_a becomes 0 at time 10
initial #20 sig_b = 1 ;   // in_b becomes 1 at time 20
```

By 6 initial constructs shown below, a simulator will generate a wave form of sig_a.



Because initial constructs are all executed at time 0, two code blocks below will create the same wave form of sig_a.

```
initial          sig_a = 0 ;
initial #500     sig_a = 1 ;
initial #1500    sig_a = 0 ;
initial #2250    sig_a = 1 ;
initial #2500    sig_a = 0 ;
initial #3000    sig_a = 1 ;
```

```
initial #2500    sig_a = 0 ;
initial          sig_a = 0 ;
initial #3000    sig_a = 1 ;
initial #500     sig_a = 1 ;
initial #2250    sig_a = 1 ;
initial #1500    sig_a = 0 ;
```

However, the above sequences of initial constructs are not easy to understand what is done as a whole.
Using begin end block, sequential block, is suitable to show which is executed after which because in a sequential block all sentences are executed in serial.

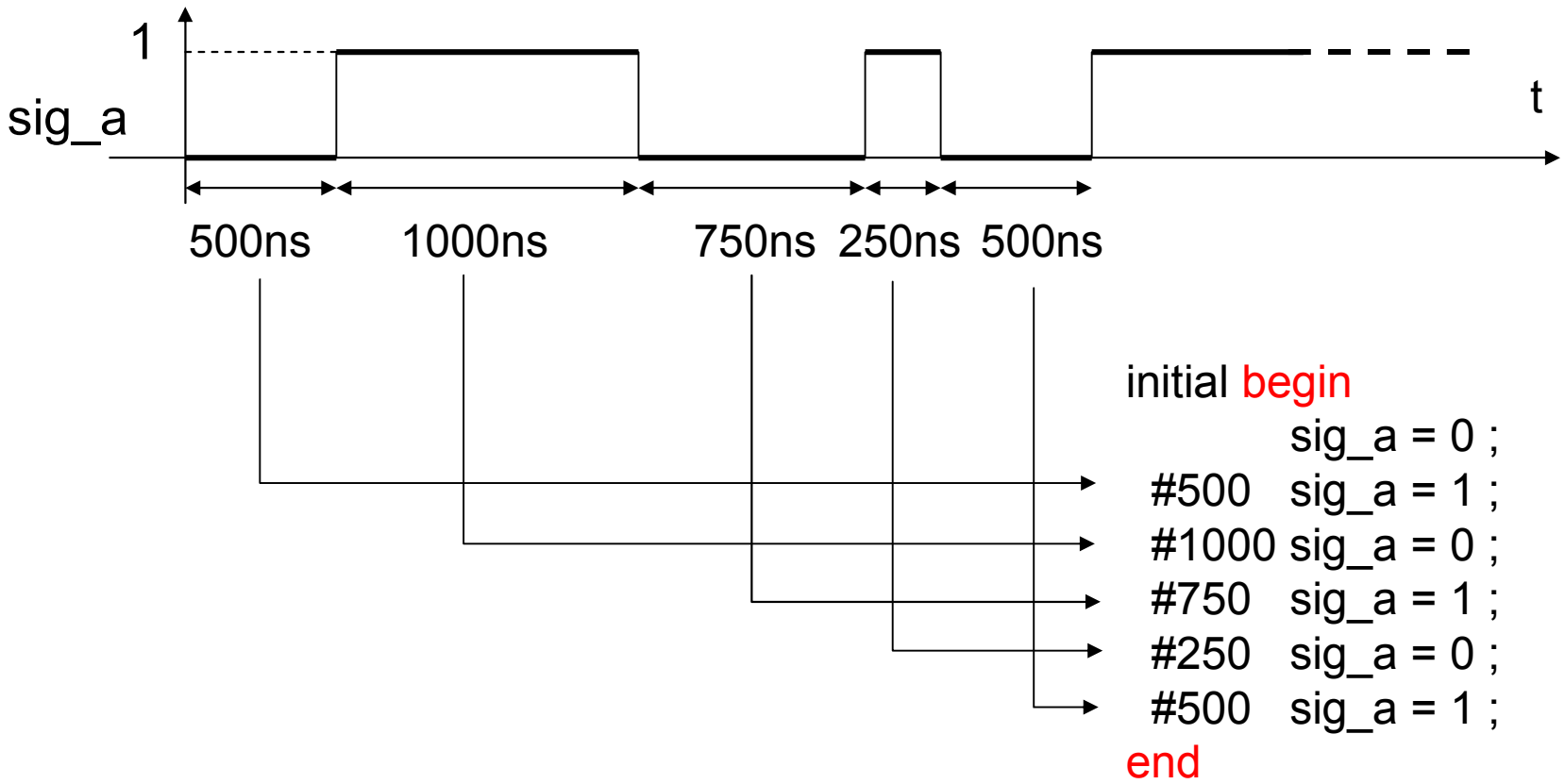
The code block on the left can be written as the code block on the right by using begin end block, sequential block. Note the difference of delay times.

```
initial          sig_a = 0 ;
initial #500     sig_a = 1 ;
initial #1500    sig_a = 0 ;
initial #2250    sig_a = 1 ;
initial #2500    sig_a = 0 ;
initial #3000    sig_a = 1 ;
```

```
initial  begin
         sig_a = 0 ;
  #500   sig_a = 1 ;
  #1000  sig_a = 0 ;
  #750   sig_a = 1 ;
  #250   sig_a = 0 ;
  #500   sig_a = 1 ;
end
```

Do not use a programming style on the left.
The right one is a common style.

Sentences in a sequential block are executed one by one.
A sentence is executed only after the preceding sentence completed.



To observe values, we can use “$monior system task” as below.

After invoked, $monitor keep watching these signals. Whenever there is a change, it outputs a message on a terminal window.

$monitor (“ *format* “, var1, var2, var3, ,,,,) ;

text %b, *text* %h, *text* %d, *text* %o

binary hexa decimal octal

An example:

```
$monitor ( “time=%d, in_a = %b, in_b = %b, out_y = %b”,
           $stime, in_a,      in_b,        out_y  ) ;
```

A system function to return simulation time.

RTL programming rules

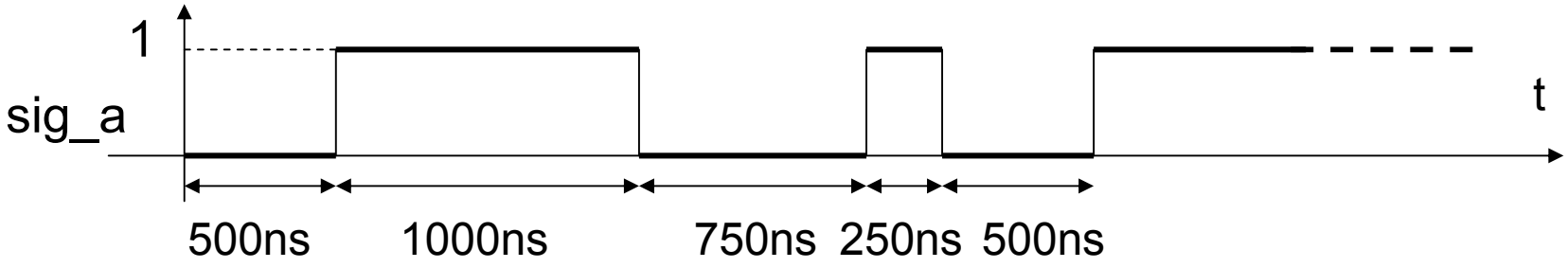
(1) Only variable data type is allowed as LHS, Left Hans Side, in a procedure. Variable data type can be declared by reg keyword.
(2) Initial construct is a procedure (*1). Therefore the rule says "sig_a must be declared by reg keyword.

```
initial begin
          sig_a = 0 ;
  #500    sig_a = 1 ;
  #1000   sig_a = 0 ;
  #750    sig_a = 1 ;
  #250    sig_a = 0 ;
  #500    sig_a = 1 ;
end
```

*1: In Verilog, there are 4 procedures. They are initial construct, always construct, function, and task.

Items written on the Left Hand Side, LHS, of procedural assignment must be declared as reg or integer as below;

```
reg sig_a ; // this data type declaration
            //  makes sig_a's data type "variable".
```




```
`timescale 1ns/100ps
module arbit_binary_sig ;
reg sig_a ; // non-FF, 1-bit signal
initial begin
          sig_a = 0 ;
  #500    sig_a = 1 ;
  #1000   sig_a = 0 ;
  #750    sig_a = 1 ;
  #250    sig_a = 0 ;
  #500    sig_a = 1 ;
end
initial begin
  $monitor("time=%d, sig_a=%b",
              $stime, sig_a ) ;
  #4000   $finish ;
end
endmodule
```

file name: arbit_binary_sig.v

Coding example.

The result of a simulation

```
time=          0, sig_a=0
time=        500, sig_a=1
time=       1500, sig_a=0
time=       2250, sig_a=1
time=       2500, sig_a=0
time=       3000, sig_a=1
Halted at location
**arbit_binary_sig.v(15) time
4000000 ps from call to $finish
```

$finish is a system task to terminate the simulation. This sentence is needed to terminate the simulation.

Ex 2-2. Procedure and racing: Run the following two sequence of initial constructs and see the difference of the results. (Racing and "must not do" in RTL programming)

```
initial begin
          sig_a = 0 ;
  #500    sig_a = 1 ;
end

initial begin
          sig_a = 1 ;
  #500    sig_a = 0 ;
end
```

```
initial begin
          sig_a = 1 ;
  #500    sig_a = 0 ;
end

initial begin
          sig_a = 0 ;
  #500    sig_a = 1 ;
end
```

Initial construct and always construct are called "procedure" in Verilog. When there are several procedures in a simulation target, depending on tools or tool vendors, which procedure shall be executed first is different. Verilog standard does not define any rule for the execution order. Therefore, the code block above has "racing" problem.
Use `define compiler directive to write two alternative source code blocks in one file.

```
`define CASE1
module test_racing ;
reg sig_a ;
`ifdef CASE1
initial begin
        sig_a = 0 ;
 #500   sig_a = 1 ;
end
initial begin
        sig_a = 1 ;
 #500   sig_a = 0 ;
end
`else
initial begin
        sig_a = 1 ;
 #500   sig_a = 0 ;
end
initial begin
        sig_a = 0 ;
 #500   sig_a = 1 ;
end
`endif
```

(A)

(B)

```
initial begin
  $monitor(
      "t=%d, sig_a=%b",
       $stime, sig_a
     );
  #600 $finish ;
end

endmodule
```

Depending on CASE1 is defined or not, a compiler select either one of (A) or (B).

file name: test_racing.v

```
`define CASE1
module test_racing ;
reg sig_a ;
initial begin
        sig_a = 0 ;
 #500   sig_a = 1 ;
end
initial begin
        sig_a = 1 ;
 #500   sig_a = 0 ;
end
```

(A)

```
module test_racing ;
reg sig_a ;
initial begin
        sig_a = 1 ;
 #500   sig_a = 0 ;
end
initial begin
        sig_a = 0 ;
 #500   sig_a = 1 ;
end
```

(B)

Run this file two times, one with "`define CASE1" and another without it.

The result shall be different depending on `define CASE1 is written in the source code or not;

```
t=          0, sig_a=1
t=        500, sig_a=0
Halted at location ,,,,
```

and

```
t=          0, sig_a=0
t=        500, sig_a=1
Halted at location ,,,,
```

Note the difference.
This is called racing.

Depending on which initial is written first, the results are different.

This means that the same code may have different result if use a tool from a different vendors.

Racing problem

```
module test_racing ;
   reg sig_a;

initial begin
          sig_a = 1 ;
  #500    sig_a = 0 ;
end

initial begin
          sig_a = 0 ;
  #500    sig_a = 1 ;
end

initial begin // observe sig_a
   $monitor  ("t= %d, sig_a=%b",
                 $stime, sig_a ) ;
   #1000 ;
   $finish ;
end
endmodule
```

RTL programming rules

(1) Do not write the same variable on LHS, Left Hand Side, of the different initial constructs to avoid possible racing problem. The same variable must not appear on LHS of other structured procedures.

The same variable, sig_a, appearing in LHS of different initial constructs.

```
initial begin
        sig_a = 0 ;
  #500  sig_a = 1 ;
end

initial begin
        sig_a = 1 ;
  #500  sig_a = 0 ;
end
```

Write one variable in LHS of one initial construct.

```
initial begin
        sig_a = 0 ;
        sig_a = 1 ;
  #500  sig_a = 1 ;
        sig_a = 0 ;
end
```

This will not cause a racing problem.

```
initial begin
        sig_a = 0 ;
        sig_a = 1 ;
  #500  sig_a = 1 ;
        sig_a = 0 ;
end
```

sig_a will be given value 1 by the next sentence, therefore this may better be deleted.

sig_a will be given value 0 by the next sentence, therefore this may better be deleted.

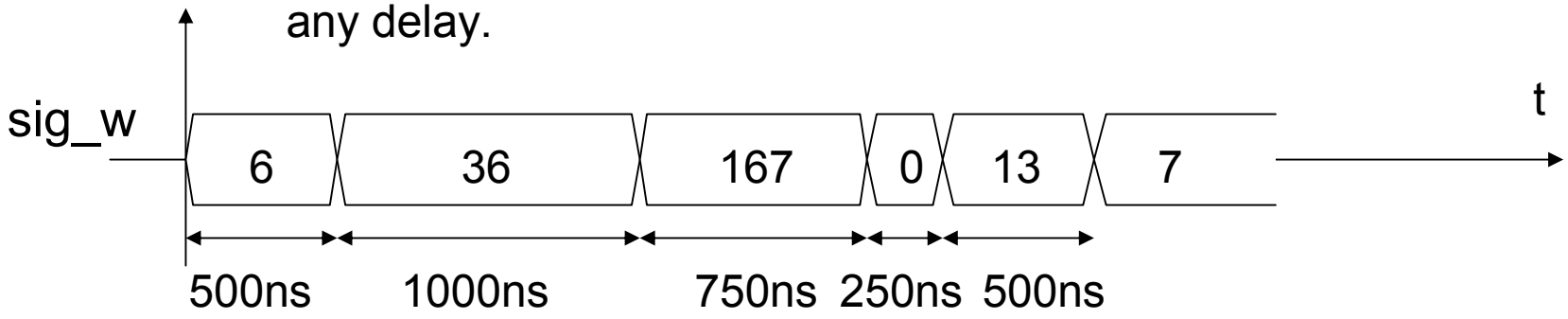
This is not a recommended coding style because it assign different values to the same variable at the same time.

Same

```
initial begin
        sig_a = 1 ;
  #500  sig_a = 0 ;
end
```

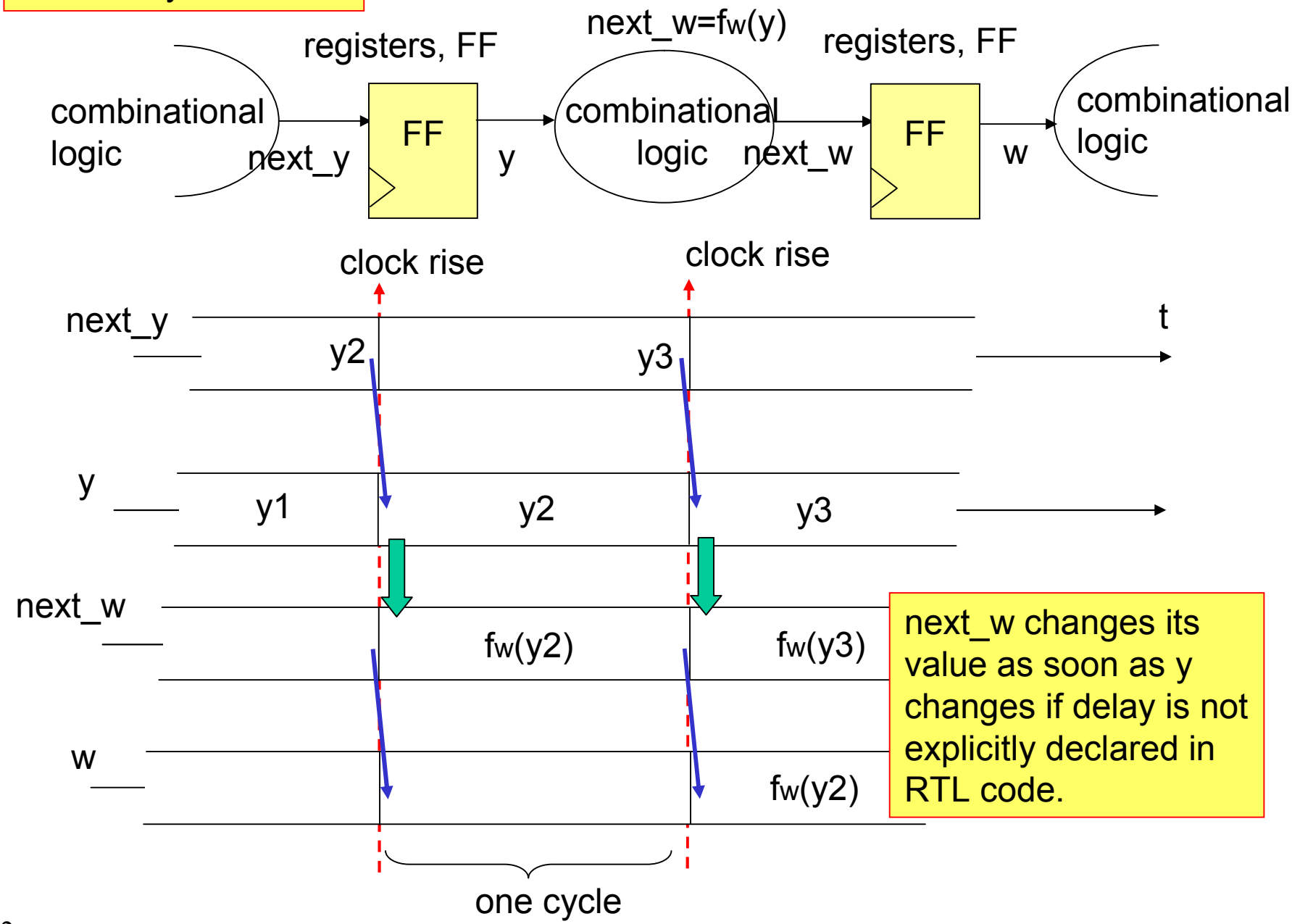
Ex 2-3. Vector waveform and initial: Write a logic block to generate and observe an 8-bit signal (sig_w) of which waveform is shown below.

Although the waveform shows transient time of change from 6 to 36, etc. RTL code does not have to take care of such transient (zero delay simulation). The signal may change from 6 to 36 at once, without any delay.



We can apply almost the same logic to the previous exercise. The only difference is the bit width of the signal. This time, the bit width is not 1 but 8.

Zero delay simulation



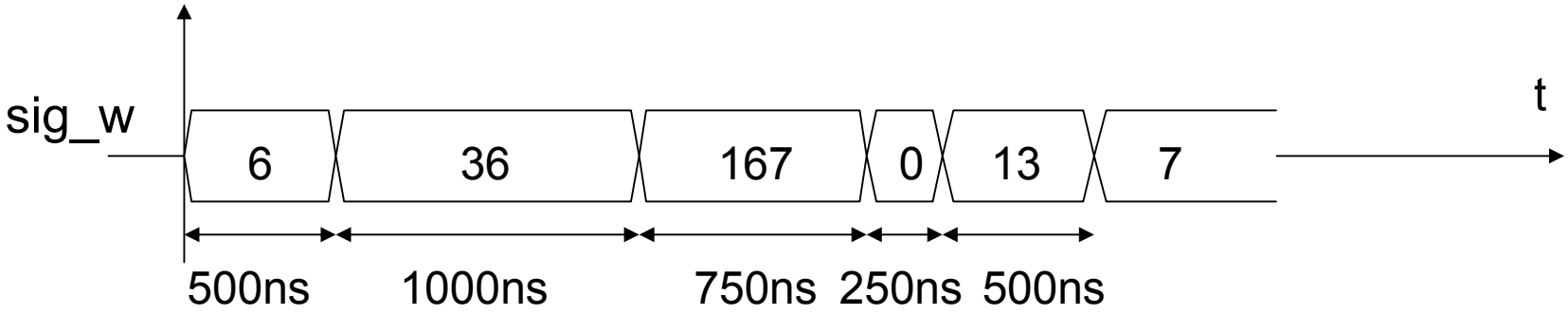
next_w changes its value as soon as y changes if delay is not explicitly declared in RTL code.

RTL programming rules

(1) Multi-bit signal, vector, must declare its bit size by range specification. The range specification is given by [MSB : LSB] where MSB is a bit number of Most Significant Bit and LSB is that of Least Significant Bit.

If the bit width is 8, then declare the range by [7:0].

MSB: Most Significant Bit
LSB: Least Significant Bit



Coding example.

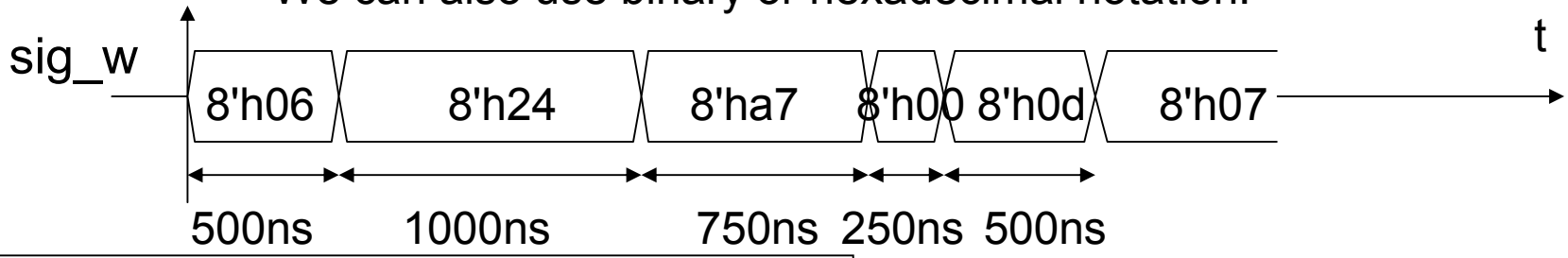
```
`timescale 1ns/100ps
module arbit_8bit_sig ;
reg [7:0] sig_w ; // non-FF, 8-bit signal
initial begin
          sig_w = 6 ;
  #500    sig_w = 36 ;
  #1000   sig_w = 167 ;
  #750    sig_w = 0 ;
  #250    sig_w = 13 ;
  #500    sig_w = 7 ;
end
initial begin
  $monitor("time=%d, sig_w=%d",
              $stime, sig_w ) ;
  #4000   $finish ;
end
endmodule
```

file name: arbit_8bit_sig.v

The result of a simulation

```
time=        0, sig_w=   6
time=      500, sig_w=  36
time=     1500, sig_w=167
time=     2250, sig_w=   0
time=     2500, sig_w=  13
time=     3000, sig_w=   7
Halted at location
**arbit_binary_sig.v(15) time
4000000 ps from call to $finish
```

We can also use binary or hexadecimal notation.




```
`timescale 1ns/100ps
module arbit_8bit_hex ;
reg [7:0] sig_w ; // non-FF, 8-bit signal
initial begin
          sig_w = 8'h06 ;
  #500    sig_w = 8'h24 ;
  #1000   sig_w = 8'ha7 ;
  #750    sig_w = 8'h00 ;
  #250    sig_w = 8'h0d ;
  #500    sig_w = 8'h07 ;
end
initial begin
  $monitor("time=%d, sig_w=%h",
              $stime, sig_w ) ;
  #4000   $finish ;
end
endmodule
```

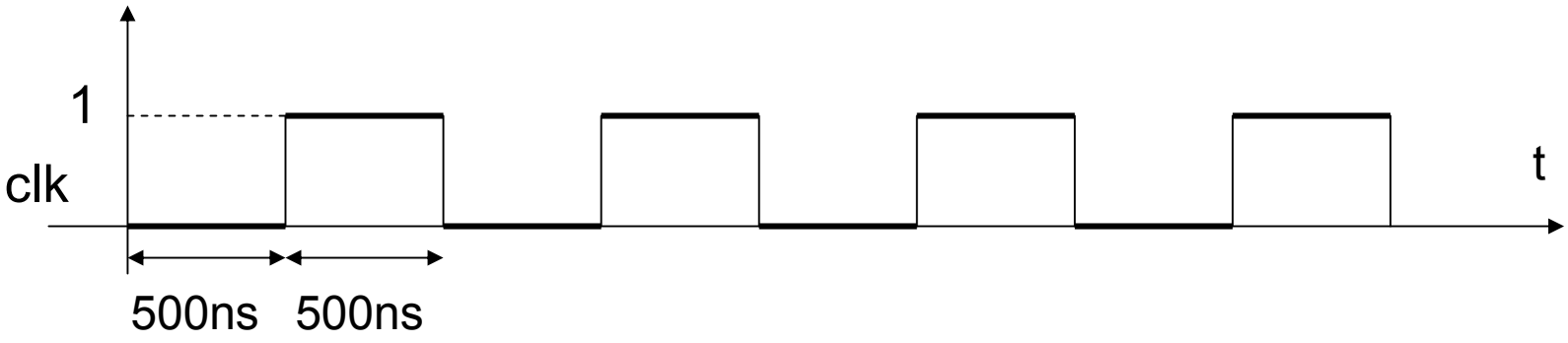
file name: arbit_8bit_hex.v

Coding example.

The result of a simulation

```
time=          0, sig_w=06
time=        500, sig_w=24
time=       1500, sig_w=a7
time=       2250, sig_w=00
time=       2500, sig_w=0d
time=       3000, sig_w=07
Halted at location
**arbit_8bit_hex.v(15) time
4000000 ps from call to $finish.
```

Ex 2-4. Repeated waveform and always: Write a logic block to generate and observe a clock signal (clk) of which waveform is shown below.



To generate repeating forever signal, we can use always construct which is activated at time 0 and run repeatedly by a simulator.

To do something repeatedly forever, we can use "always construct" as below.
In RTL simulation, an always construct is executed at time 0 and repeated forever.
And if there are several always constructs, they are all executed at time 0.

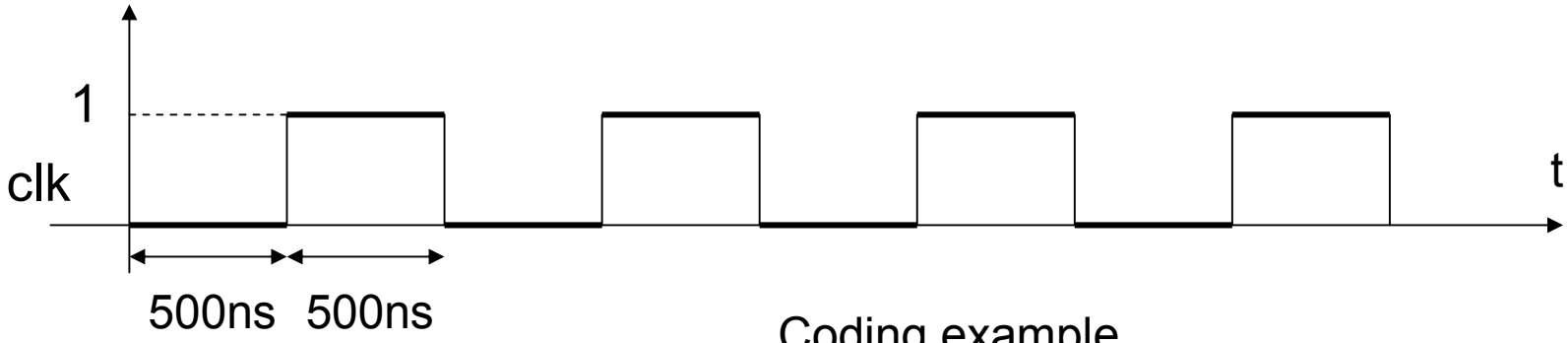
```
always  sig_a = 0 ;   // sig_a is given value 0 at time 0
                      // and repeatedly given 0 forever.
```

This code will hang up a simulator.

By using delay, or wait, specified by @, we can use always construct without making a simulator hung-up.

RTL programming rules

(1) # and @ are mandatory in always construct.
Because always construct is executed forever, there must be some delay or wait written in the always construct. If there is no delay or wait, a simulator program will hang up in infinite loop to execute the always construct and can do nothing else.



Coding example.

```
`timescale 1ns/100ps
module clk_sig ;
reg clk ; // non-FF, clock signal
always begin
  clk = 0 ;  #500 ;
  clk = 1 ;  #500 ;
end
initial begin
  $monitor("time=%d, clk=%b", $stime, clk ) ;
  #5000 $finish ;
end
endmodule
```

file name: clk_sig.v

The result of a simulation

```
time=        0, clk0
time=      500, clk1
time=     1000, clk0
time=     1500, clk1
time=     2000, clk0
time=     2500, clk1
time=     3000, clk0
time=     3500, clk1
time=     4000, clk0
time=     4500, clk1
Halted at location
**clk_sig.v(10) time 5000000 ps
from call to $finish.
```

The code on the previous page is not convenient for changing the cycle time. It is recommended to use parameter for such may-be-changed-often value as shown below.

```
`timescale 1ns/100ps
module clk_sig_para ;
parameter HF_CYCL=500 ;
reg clk ; // non-FF, clock signal
always begin
  clk = 0 ;  #HF_CYCL ;
  clk = 1 ;  #HF_CYCL ;
end
initial begin
  $monitor("time=%d, clk=%b", $stime, clk ) ;
  #(HF_CYCL*10) $finish ;
end
endmodule
```

file name: clk_sig_para.v

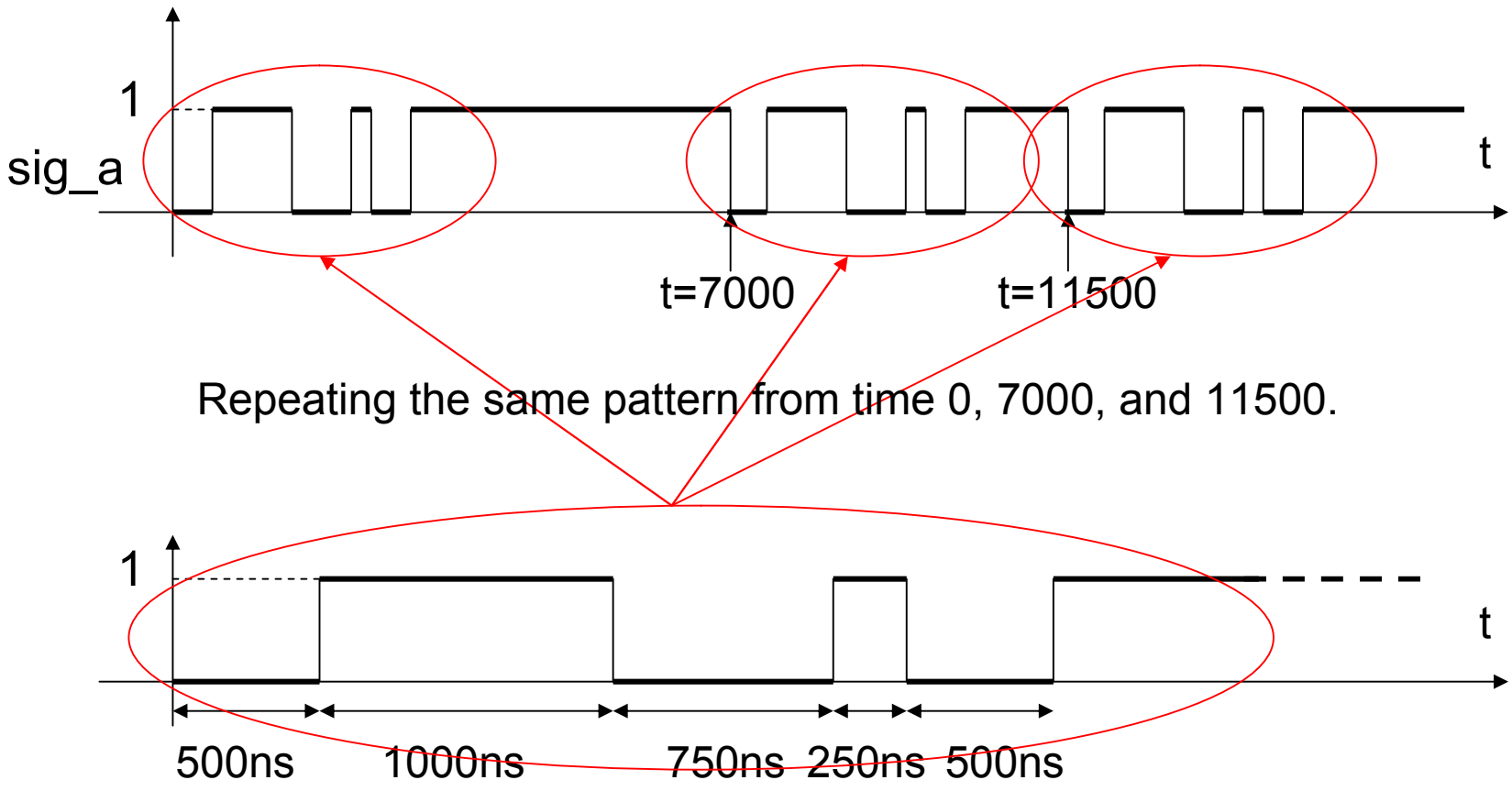
Coding example.

The result of a simulation

```
time=        0, clk0
time=      500, clk1
time=     1000, clk0
time=     1500, clk1
time=     2000, clk0
time=     2500, clk1
time=     3000, clk0
time=     3500, clk1
time=     4000, clk0
time=     4500, clk1
Halted at location
**clk_sig_para.v(11) time
5000000 ps from call to $finish.
```

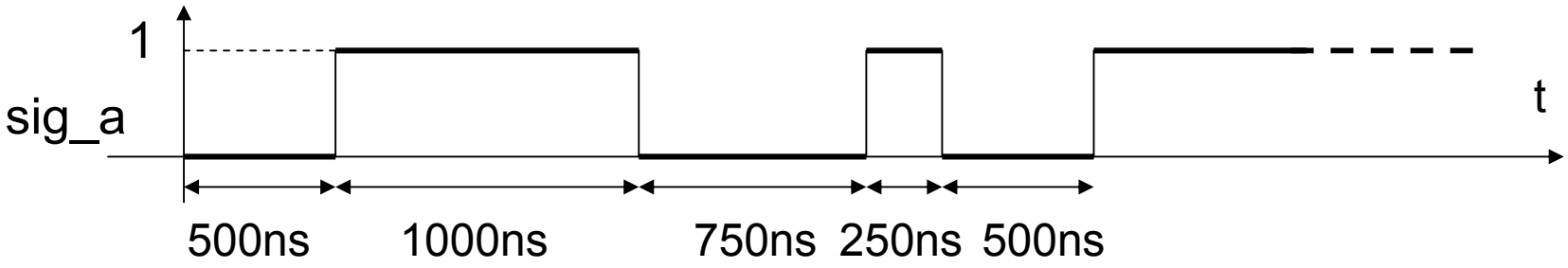
Run with different parameter values.

Ex 2-5. Repeated waveform and task: Write a logic block to generate and observe a signal (sig_a) of which waveform is shown below.



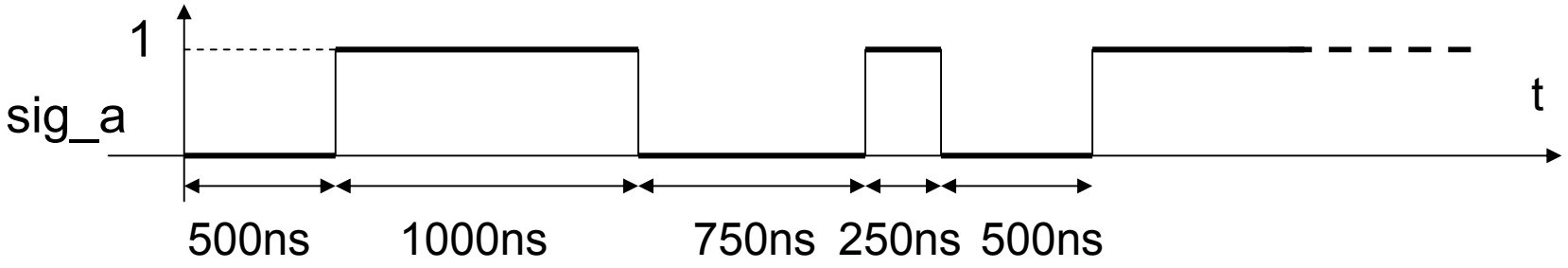
To generate the same pattern repeatedly, it is recommended to define and use a task which generates a basic pattern to be repeated.

First, let's think how to generate the wave form which is repeated 3 times.



The above can be generated by the initial construct shown on the right as we already studied in Ex. 2-1.

```
initial begin
          sig_a = 0 ;
  #500    sig_a = 1 ;
  #1000   sig_a = 0 ;
  #750    sig_a = 1 ;
  #250    sig_a = 0 ;
  #500    sig_a = 1 ;
end
```




```
`timescale 1ns/100ps
module arbit_binary_sig ;
reg sig_a ; // non-FF, 1-bit signal
initial begin
          sig_a = 0 ;
  #500    sig_a = 1 ;
  #1000   sig_a = 0 ;
  #750    sig_a = 1 ;
  #250    sig_a = 0 ;
  #500    sig_a = 1 ;
end
initial begin
  $monitor("time=%d, sig_a=%b",
              $stime, sig_a ) ;
  #4000   $finish ;
end
endmodule
```

Take out the basic wave form creating part and put it into a task.

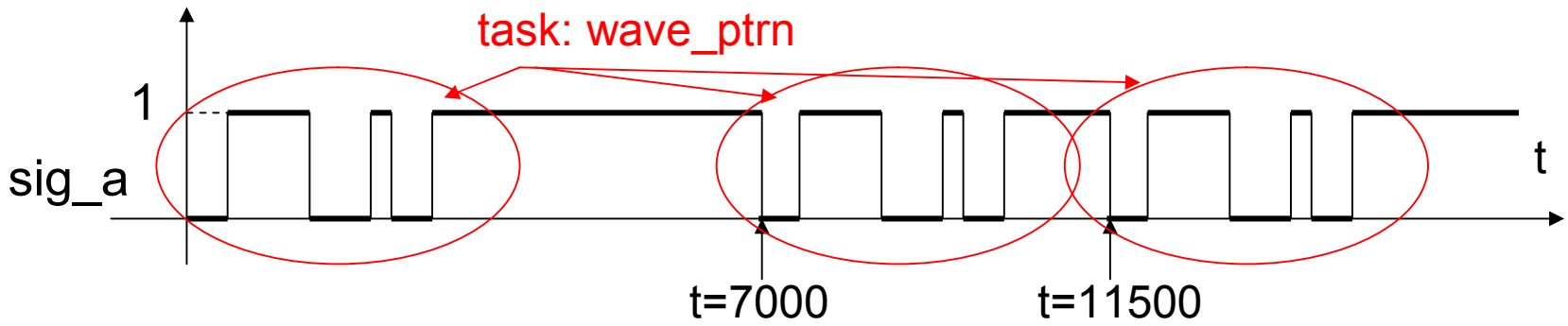
```
task wave_ptrn ;
begin
          sig_a = 0 ;
  #500    sig_a = 1 ;
  #1000   sig_a = 0 ;
  #750    sig_a = 1 ;
  #250    sig_a = 0 ;
  #500    sig_a = 1 ;
end
endtask
```

file name: arbit_binary_sig.v

RTL programming rules

(1) A task is executed only when it is invoked.
(2) To invoke a task, write the task name in a procedure as shown below.
(3) If LHS in a task is not declared as output argument, then assigning any value to LHS is effective at the time of the assignment.
(4) If LHS in a task is declared as output argument, then only the last assignment is effective.

```
a procedure
        ⋮
    task_name ;   ←  A task, task_name, is executed
        ⋮            when this sentence is executed.
end procedure

task task_name;
        ⋮
endtask
```



Coding example.

```
`timescale 1ns/100ps
module rept_sig ;
reg sig_a ; // non-FF, 1-bit variable
initial begin
        wave_ptrn ;
 #4000 wave_ptrn ;
 #1500 wave_ptrn ;
end
initial begin
 $monitor("time=%d, sig_a=%b",
            $stime, sig_a ) ;
 #20000 $finish ;
end
```

file name: rept_sig.v

To start the second basic wave pattern at time 7000, the delay here must be 4000 (= 7000 – 3000).

```
task wave_ptrn ;
begin
        sig_a = 0 ;
 #500   sig_a = 1 ;
 #1000  sig_a = 0 ;
 #750   sig_a = 1 ;
 #250   sig_a = 0 ;
 #500   sig_a = 1 ;
end
endtask

endmodule
```

The task has 3000 unit time delay in it.

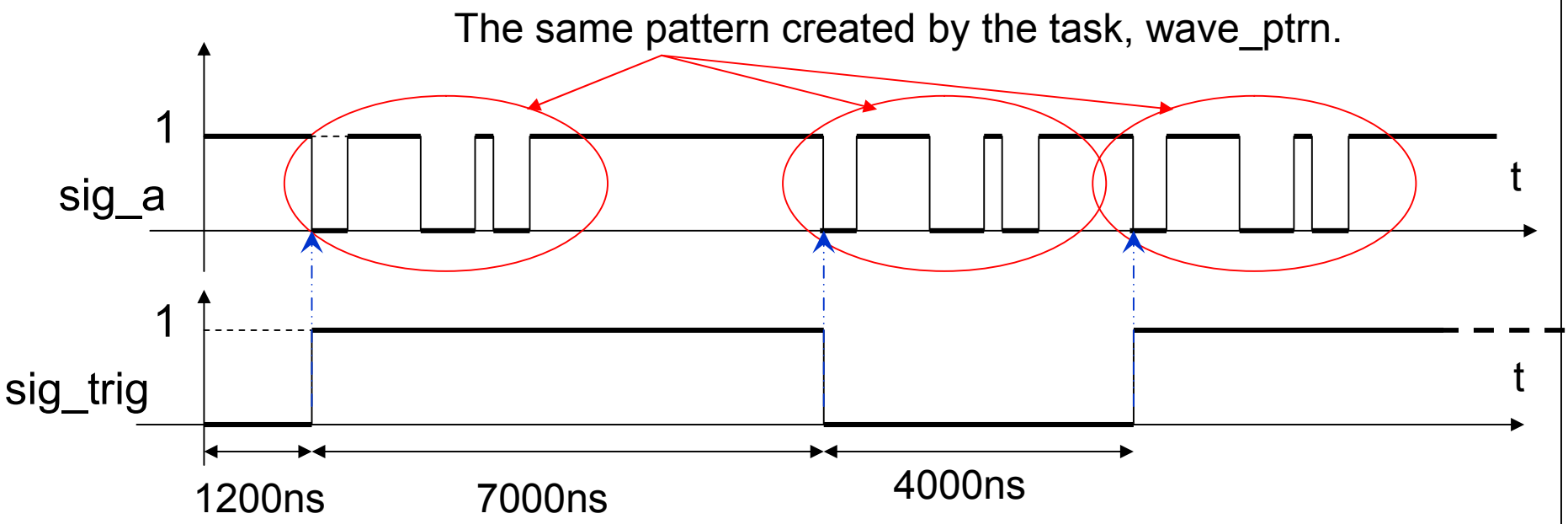
sig_a becomes 1 as soon as this assignment is executed.

The result of a simulation

```
time=          0, sig_a=0
time=        500, sig_a=1
time=       1500, sig_a=0
time=       2250, sig_a=1
time=       2500, sig_a=0
time=       3000, sig_a=1
time=       7000, sig_a=0
time=       7500, sig_a=1
time=       8500, sig_a=0
time=       9250, sig_a=1
time=       9500, sig_a=0
time=      10000, sig_a=1
time=      11500, sig_a=0
time=      12000, sig_a=1
time=      13000, sig_a=0
time=      13750, sig_a=1
time=      14000, sig_a=0
time=      14500, sig_a=1
Halted at location **rept_sig.v(12) time 20000000 ps from call to $finish.
```

Ex 2-6. How to use implicit event: Write a logic block to generate and observe signals (sig_a and sig_trig) of which waveforms are shown below. sig_a's change must be triggered by the change of sig_trig.

sig_a changes every time sig_trig changes. Whenever sit_trig changes, except at time 0, sig_a starts to change as defined by the task, wave_ptrn, in the previous exercise.



Signal's value change causes change value event in Verilog. It is called an implicit event, and can be used to synchronize procedures. To catch a timing such as sig_trig's change, we can use @(sig_trig).

A process to generate sig_trig

```
initial begin
        sig_trig = 0 ;
  #1200 sig_trig = 1 ;
  #7000 sig_trig = 0 ;
  #4000 sig_trig = 1 ;
  #5000 $finish ;
end
```

A process to generate sig_a

```
initial begin
  sig_a = 1 ; // 1 at start time
  #1 @ (sig_trig) wave_ptrn ;
   @ (sig_trig) wave_ptrn ;
   @ (sig_trig) wave_ptrn ;
end
```

This delay is needed to avoid catching sig_trig's change from unknown value x to 0 at time 0.

Each time sig_trig changes, the task, wave_ptrn, is invoked.

The total module can be written as shown on the next page.

Coding example.

```
`timescale 1ns/100ps
module rept_sig_evnt ;
reg sig_a, sig_trig ; // non-FF
initial begin
  $monitor("time=%d, sig_a=%b",
                $stime, sig_a ) ;
end

initial begin
   sig_a = 1 ; // 1 at start time
  #1  @ (sig_trig) wave_ptrn ;
      @ (sig_trig) wave_ptrn ;
      @ (sig_trig) wave_ptrn ;
end

initial begin
           sig_trig = 0 ;
  #1200 sig_trig = 1 ;
  #7000 sig_trig = 0 ;
  #4000 sig_trig = 1 ;
  #5000 $finish ;
end
```

```
task wave_ptrn ;
begin
           sig_a = 0 ;
  #500   sig_a = 1 ;
  #1000 sig_a = 0 ;
  #750   sig_a = 1 ;
  #250   sig_a = 0 ;
  #500   sig_a = 1 ;
end
endtask

endmodule
```

file name: rept_sig_evnt.v

Different coding (1)

We can use @, meaning wait, in a task. Therefore, we can modify the code as below.

Take @(sig_trin) out of the initial construct and put it into the task.

```
initial begin
  sig_a = 1 ; // 1 at start time
 #1 @ (sig_trig) wave_ptrn ;
    @ (sig_trig) wave_ptrn ;
    @ (sig_trig) wave_ptrn ;
end
```

```
task wave_ptrn ;
begin
 @ (sig_trig) sig_a = 0 ;
  #500   sig_a = 1 ;
  #1000 sig_a = 0 ;
  #750   sig_a = 1 ;
  #250   sig_a = 0 ;
  #500   sig_a = 1 ;
end
endtask
```

Different coding (2)

For this exercise, the wave pattern starting trigger is always the same, change of sig_trig, therefore we do not have to use a task, but can use initial construct and forever as shown on the right below.

```
initial begin
   sig_a = 1 ; // on at start time
  #1  @ (sig_trig) wave_ptrn ;
      @ (sig_trig) wave_ptrn ;
      @ (sig_trig) wave_ptrn ;
end

task wave_ptrn ;
begin
            sig_a = 0 ;
  #500      sig_a = 1 ;
  #1000     sig_a = 0 ;
  #750      sig_a = 1 ;
  #250      sig_a = 0 ;
  #500      sig_a = 1 ;
end
endtask
```

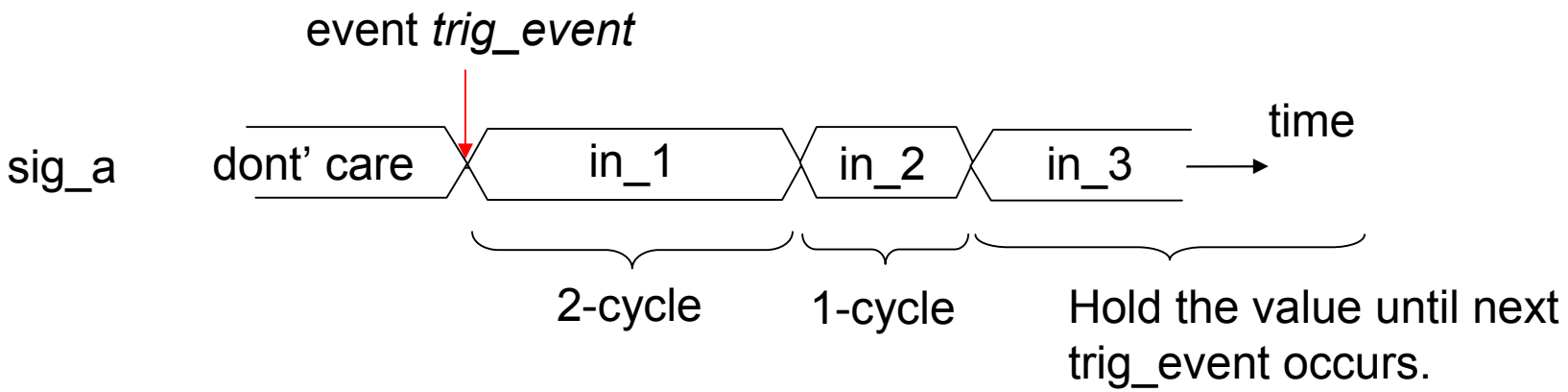
```
initial begin
  sig_a = 1 ;
  forever begin
  #1 @(sig_trig) begin
                 sig_a = 0 ;
     #500        sig_a = 1 ;
     #1000       sig_a = 0 ;
     #750        sig_a = 1 ;
     #250        sig_a = 0 ;
     #500        sig_a = 1 ;
    end
  end
end
```

This part is repeated forever.

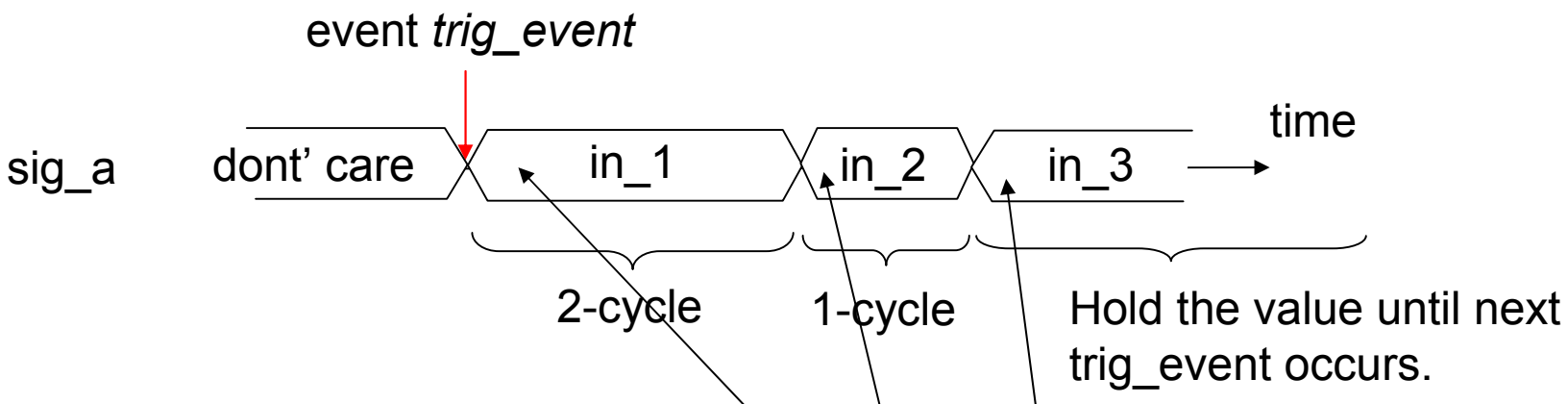
This is not a recommended style.

Task is very suitable to describe signal change which follows the same pattern or a protocol, therefore, it is very often used to model bus, peripheral, CPU and others.

Ex 2-7. How to use named event: Design a task named prot_1 which can create the output data pattern shown below. The task must have 3 input arguments and they are assigned to sig_a. For the first 2 cycles the first argument, in_1, must be assigned to sig_a, for the next 1 cycle the second argument, in_2, must be assigned to sig_a, and for the cycles after 3 cycles the third argument, in_3, must be assigned to sig_a.

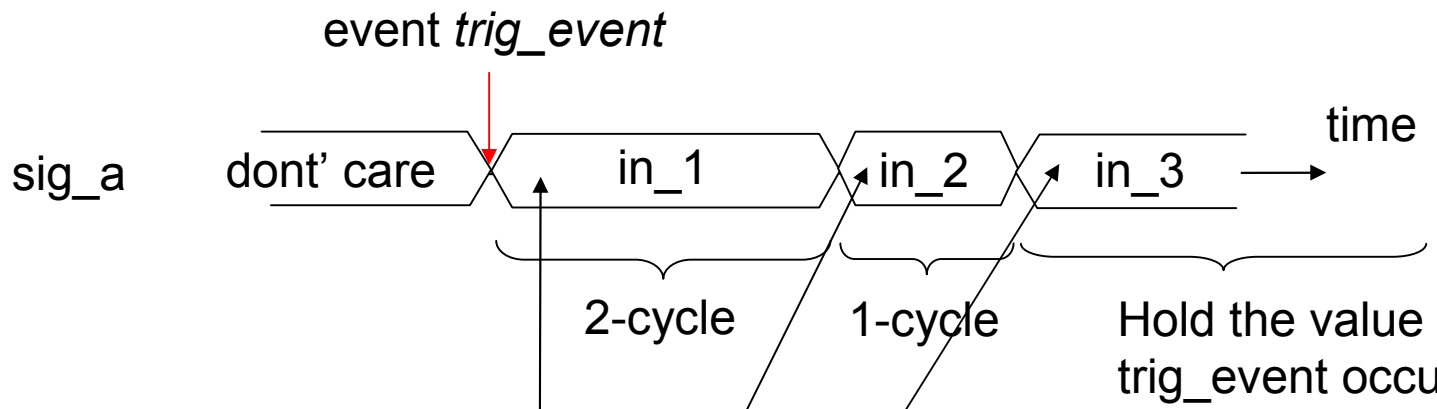


We can generate named event by using Verilog opreator "->". To generate a named event we can use -> *event_name* and to catch it we can use @(*event_name*).




```
begin
    wait for trig_event
        Set in_1 to output
        Delay 2 cycles.
        Set in_2 to output
        Delay 1 cycles.
        Set in_3 to output
end
```

This logic can be programmed as shown in the middle of the next page.



We can get a task as shown on the right.

```
begin
    wait for trig_event
        Set in_1 to output
        Delay 2 cycles.
        Set in_2 to output
        Delay 1 cycles.
        Set in_3 to output
end
```

```
@ ( trig_event ) ;
sig_a = in_1 ;
#(CYCL * 2 ) ;
sig_a = in_2 ;
#(CYCL * 1 ) ;
sig_a = in_3 ;
```

```
task prot_1 ;
input [3:0] in_1, in_2, in_3 ;
begin
@ ( trig_event ) ;
sig_a = in_1 ;
#(CYCL * 2 ) ;
sig_a = in_2 ;
#(CYCL * 1 ) ;
sig_a = in_3 ;
end
endtask
```

Now, write a test bench and check how task works.

```
module test_prot_1 ;
parameter CYCL = 10 ;
reg [3:0] sig_a ;
event trig_event ;
//
initial begin
prot_1( 1, 3, 2 ) ;
prot_1( 7, 0, 4 ) ;
prot_1( 5, 6, 3 ) ;
end
//
initial begin
# CYCL -> trig_event ;
#(CYCL*15) -> trig_event ;
#(CYCL*20) -> trig_event ;
#(CYCL*10) $finish ;
end
//
```

input arguments

```
initial begin
 $monitor("t=%d, sig_a=%d",
    $stime, sig_a ) ;
end
//
task prot_1 ;
input [3:0] in_1, in_2, in_3 ;
begin
@ ( trig_event ) ;
sig_a = in_1 ;
#(CYCL * 2 ) ;
sig_a = in_2 ;
#(CYCL * 1 ) ;
sig_a = in_3 ;
end
endtask
//
endmodule
```

"->" is a Verilog operator to generate an event.

file name: test_prot_1.v

Now, write a test bench and check how task works.

A sample result

```
t=        0, sig_a= x
t=       10, sig_a= 1
t=       30, sig_a= 3
t=       40, sig_a= 2
t=      160, sig_a= 7
t=      180, sig_a= 0
t=      190, sig_a= 4
t=      360, sig_a= 5
t=      380, sig_a= 6
t=      390, sig_a= 3
Halted at location **test_pr
```

Next, rewrite the file so that the task has output arguments.

```
module test_prot_1 ;
parameter CYCL = 10 ;
reg [3:0] sig_a ;
event trig_event ;
//
initial begin
prot_1( 1, 3, 2, sig_a ) ;
prot_1( 7, 0, 4 , sig_a ) ;
prot_1( 5, 6, 3 , sig_a ) ;
end
//
initial begin
# CYCL -> trig_event ;
#(CYCL*15) -> trig_event ;
#(CYCL*20) -> trig_event ;
#(CYCL*10) $finish ;
end
//
initial begin
 $monitor("t=%d, sig_a=%d",
    $stime, sig_a ) ;
end
//
```

```
task prot_1 ;
input [3:0] in_1, in_2, in_3 ;
output [3:0] sig_out ;
reg [3:0] sig_out ;
begin
@ ( trig_event ) ;
sig_out = in_1 ;
#(CYCL * 2 ) ;
sig_out = in_2 ;
#(CYCL * 1 ) ;
sig_out = in_3 ;
end
endtask
//
endmodule
```

file name: test_prot_1_ng.v

Now, run this file and check the result.

former result

```
t=        0, sig_a= x
t=       10, sig_a= 1
t=       30, sig_a= 3
t=       40, sig_a= 2
t=      160, sig_a= 7
t=      180, sig_a= 0
t=      190, sig_a= 4
t=      360, sig_a= 5
t=      380, sig_a= 6
t=      390, sig_a= 3
Halted at location **test_pr
```

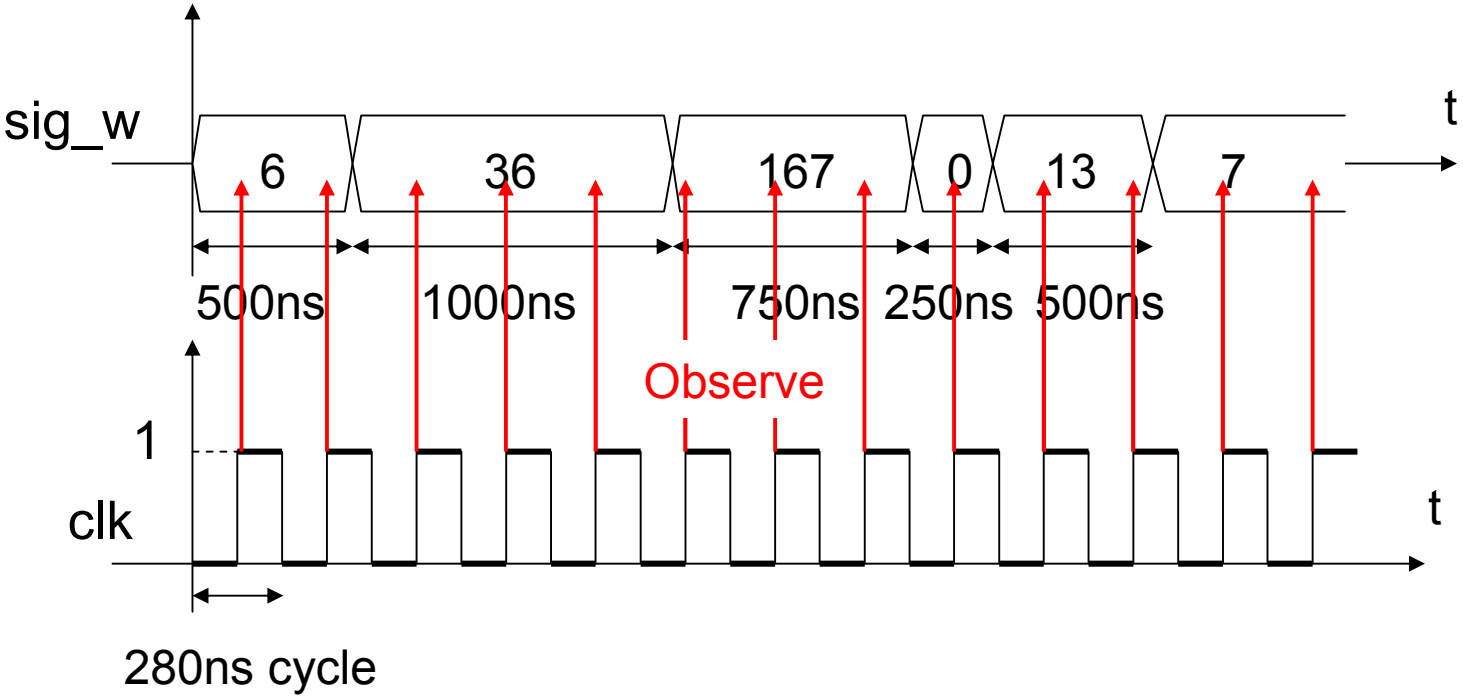
A sample new result

```
t=        0, sig_a= x
t=       40, sig_a= 2
t=      190, sig_a= 4
t=      390, sig_a= 3
Halted at location **test_pr
```

If a task has output argument, assigning any values to the output arguments becomes effective when endask is executed. Therefore, only the last assignment in the task is effective. In this example, only "sig_out = in_3 ;" is effective because it is the last assignment in the task.

Ex 2-8. Observe signals periodically: Write a logic block to observe a 8-bit signal (sig_w) which can be generated by the logic block of Ex2-3 at every clock rise time.

Use the same logic in Ex2-3 to create sig_w.



We can use @(posedge clk) to catch a clock rise event. To observe signals at specific timing, we have to invoke a system task $strobe at such specific timing.

We can use various style as shown below to observe signals at clock rise time.

```
initial begin
  @(posedge clk )$strobe( ,,,,, ) ;
  @(posedge clk )$strobe( ,,,,, ) ;
  @(posedge clk )$strobe( ,,,,, ) ;
  @(posedge clk )$strobe( ,,,,, ) ;
  @(posedge clk )$strobe( ,,,,, ) ;
,,,,
,,,
end
```

This style is applicable only for limited times of observation.

```
initial begin
  forever begin
    @(posedge clk )$strobe(  ,,, ) ;
  end
end
```

```
always begin
  @(posedge clk) begin
    $strobe( ,,, ) ;
  end
end
```

Recommended style

```
always @ ( posedge clk ) begin
  $strobe( ,,, ) ;
end
```

The syntax of $strobe is the same to that of $monitor.

$strobe (" *format* ", var1, var2, var3, ,,,,) ;

text %b, *text* %h, *text* %d, *text* %o

example:

```
$strobe ( "time=%d, in_a = %b, in_b = %b, out_y = %h",
                $stime, in_a,      in_b,        out_y  ) ;
```

This sentence may output the following message on the terminal when it is invoked, not when in_a, in_b, or out_y change.

t= 50, in_a =01100110, in_b =1100, out_y = 3f

```
`timescale 1ns/100ps
module obsrv_at_clk_rise ;
parameter HF_CYCL=140 ;
reg clk ; // non-FF, clock
reg [7:0] sig_w ; // non-FF
always @ ( posedge clk ) begin
  $strobe("t=%d, sig_w=%d",
              $stime, sig_w ) ;
end

initial begin
          sig_w = 6 ;
  #500    sig_w = 36 ;
  #1000   sig_w = 167 ;
  #750    sig_w = 0 ;
  #250    sig_w = 13 ;
  #500    sig_w = 7 ;
  #700    $finish ;
end
```

The result of a simulation

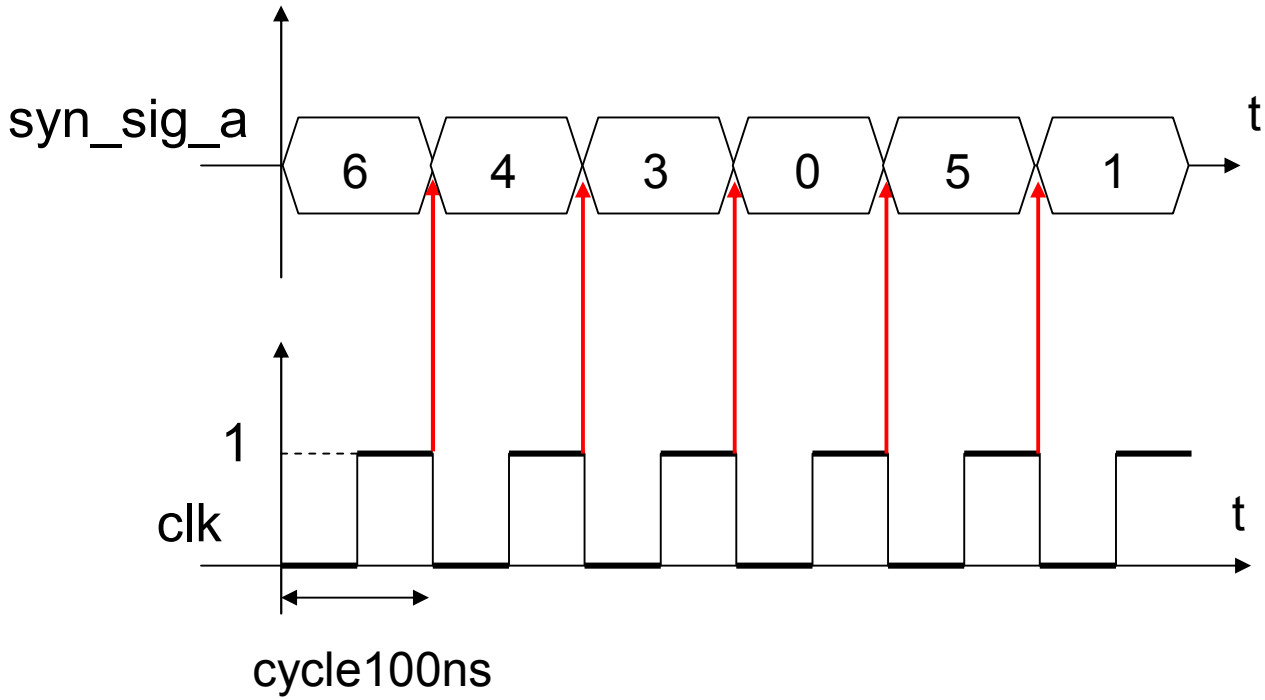
```
t=        140, sig_w=  6
t=        420, sig_w=  6
t=        700, sig_w= 36
t=        980, sig_w= 36
t=       1260, sig_w= 36
t=       1540, sig_w=167
t=       1820, sig_w=167
t=       2100, sig_w=167
t=       2380, sig_w=  0
t=       2660, sig_w= 13
t=       2940, sig_w= 13
t=       3220, sig_w=  7
t=       3500, sig_w=  7
Halted at location
**obsrv_at_clk_rise.v(18) time
```

```
always begin
  clk = 0 ;  #HF_CYCL ;
  clk = 1 ;  #HF_CYCL ;
end

endmodule
```

file name: obsrv_at_clk_rise.v

Ex 2-9. Signals synchronized to clock: Write a logic block to generate a 4-bit signal (syn_sig_a) of which waveform is shown below. The change of the signal is triggered by fall edges of clk.

In synchronous design, input signals must be stable at clock rise time. A logic block designed in this exercise can be used to generate input signals for debugging a synchronous system.



We can catch a fall edge event by using @(negedge clk) and use it to trigger syn_sig_a's change.

```
`timescale 1ns/100ps
module gen_syn_sig ;
parameter HF_CYCL=50 ;
reg clk ; // non-FF, clock
reg [7:0] syn_sig_a ; // non-FF
initial begin
  $monitor("time=%d, clk=%b, sig_a=%d",
                $stime, clk, syn_sig_a ) ;
end
initial begin
                syn_sig_a = 6 ; #1;
 @(negedge clk) syn_sig_a = 4 ;
 @(negedge clk) syn_sig_a = 3 ;
 @(negedge clk) syn_sig_a = 0 ;
 @(negedge clk) syn_sig_a = 5 ;
 @(negedge clk) syn_sig_a = 1 ;
 #(HF_CYCL*2)  $finish ;
end
```

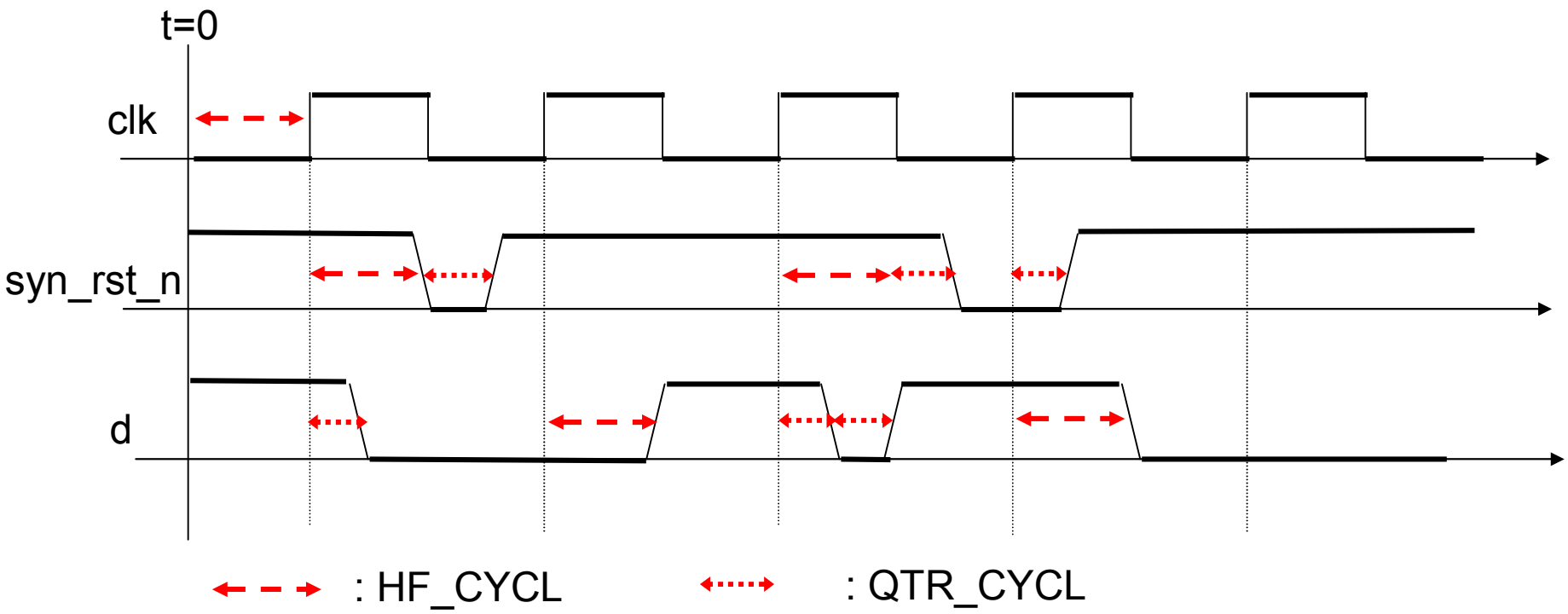
The result of a simulation

```
time=        0, clk=0, sig_a=  6
time=       50, clk=1, sig_a=  6
time=      100, clk=0, sig_a=  4
time=      150, clk=1, sig_a=  4
time=      200, clk=0, sig_a=  3
time=      250, clk=1, sig_a=  3
time=      300, clk=0, sig_a=  0
time=      350, clk=1, sig_a=  0
time=      400, clk=0, sig_a=  5
time=      450, clk=1, sig_a=  5
time=      500, clk=0, sig_a=  1
time=      550, clk=1, sig_a=  1
Halted at location
**gen_syn_sig.v(17) time
600000 ps from call to $finish.
```

```
always begin
  clk = 0 ;  #HF_CYCL ;
  clk = 1 ;  #HF_CYCL ;
end

endmodule
```

file name: gen_syn_sig.v

Ex 2-10. Clock and signal timing: Write a logic block to generate test input signals shown below for DFF, D-type flip-flop, with synchronous active low reset.



Use the following parameters.
parameter QTR_CYCL = 20 ;
parameter HF_CYCL = QTR_CYCL * 2 ;

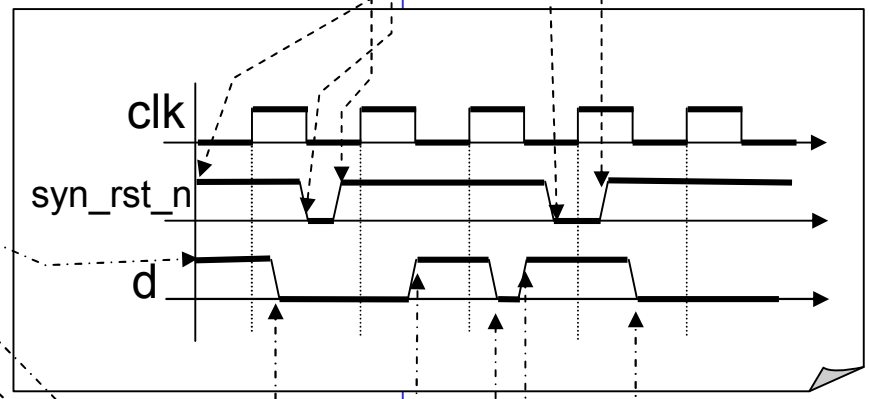
```
module test_dff  ;
parameter QTR_CYCL = 5 ;
parameter HF_CYCL = QTR_CYCL * 2 ;

reg clk, syn_rst_n ; // non-FF
reg d ;  // non-FF
wire q ;

// connect signals to shifter
dff_syn_rst dff_syn_rst_01( .clk(clk), .syn_rst_n(syn_rst_n),
                    .d(d), .q(q) ) ;
// connection end

always begin      // clock generator
   clk = 1'b0 ;  # HF_CYCL ;
   clk = 1'b1 ;  # HF_CYCL  ;
end

initial  begin
  $monitor  ("t= %d, clk=%b, syn_rst_n=%b, d=%b, q=%b",
                    $stime, clk, syn_rst_n, d, q ) ;
end
```

```
module dff_syn_rst ( clk, syn_rst_n,
                     d, q ) ;
parameter FF_DLY = 1 ;
input clk, syn_rst_n ;
input d ;
output q ;
wire clk, syn_rst_n, d ;
reg     q ; // FF
always @ ( posedge clk )
begin
   if ( syn_rst_n  ==  1'b0 ) begin
        q <=#FF_DLY 1'b0 ;
   end
   else  begin
        q <=#FF_DLY   d ;
    end
end
endmodule
```

Write this code block right after endmodule source line of module test_dff.

Checked on cver

```
initial begin   // for reset signal syn_rst_n
   syn_rst_n = 1'b1 ;
   @ ( posedge clk )
         #(HF_CYCL) syn_rst_n = 1'b0 ;
         #(QTR_CYCL)   syn_rst_n = 1'b1 ;
   @ ( posedge clk ) ;
   @ ( posedge clk )
         #(HF_CYCL + QTR_CYCL) syn_rst_n = 1'b0 ;
   @ ( posedge clk )  #(QTR_CYCL) syn_rst_n = 1'b1 ;
   @ ( posedge clk )  ;
   @ ( posedge clk )  $finish ;
end

initial begin     // for input signal d
   d = 1'b1 ;
   @ ( posedge clk )
         #(QTR_CYCL) d = 1'b0 ;
   @ ( posedge clk )
         #(HF_CYCL)  d = 1'b1 ;
   @ ( posedge clk )
         #(QTR_CYCL) d = 1'b0 ;
         #(QTR_CYCL) d = 1'b1 ;
   @ ( posedge clk )
         #(HF_CYCL) d = 1'b0 ;
end

endmodule
```



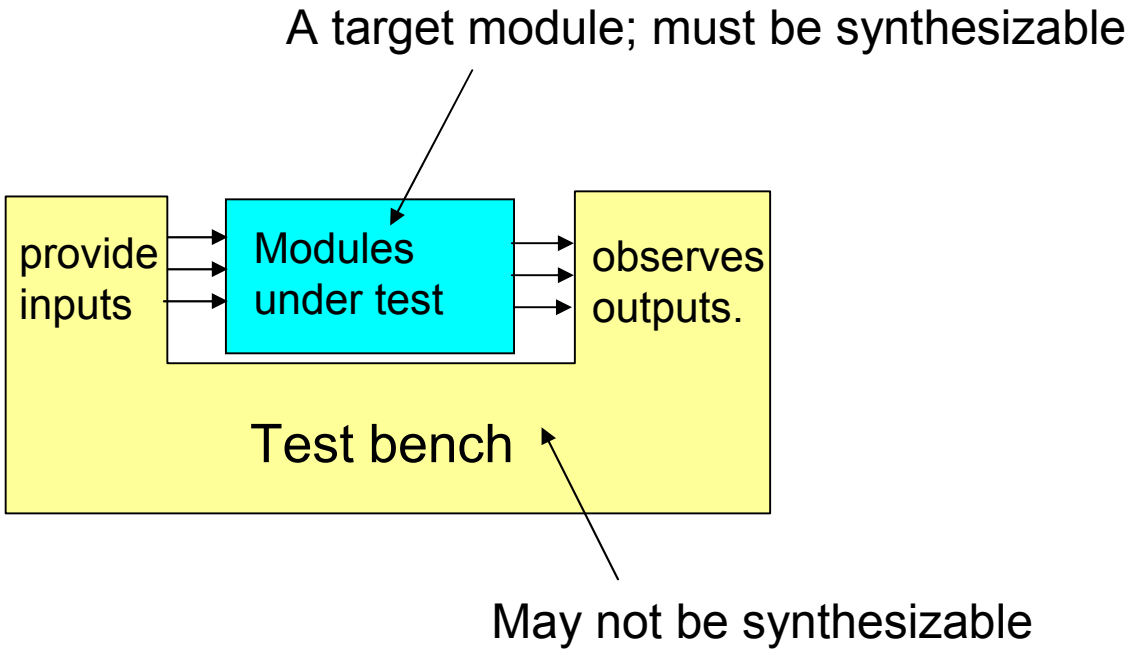
file name: test_dff.v

Insert source code lines for module dff_syn_rst here.

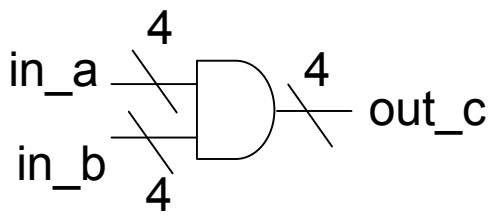
Chapter 3. Combinational logic and test bench

A combinational logic block does not have any memory elements in it.

It can be created by connecting simple AND, OR, NOT, EOR, etc.



Ex 3-1. All-in-one module for testing 4-bit AND: Write a module which has a 4-bit AND gate and logic blocks to generate in_a and in_b and to observe out_c to test the AND gate.



We can define out_c as an output of AND gate by using a continuous assignment such as "assign out_c = in_a & in_b ;".
in_a and in_b can be generated by using initial construct. To observe out_c, use $strobe system task.

RTL programming rules

(1) An output of combinational logic gate can be given by using continuous assign statement.
(2) A continuous assign must start with assign key word.
(3) The data type of an LHS of a continuous assign must be net data type. Net data type is declared by using wire keyword.

out_c appears on LHS of continuous assign, therefore it must be a net.

```
wire out_c ;

assign out_c = in_a & in_b ;
```

Unlike a procedural assignment used in a structured procedures, continuous assign is executed whenever its RHS, Right Hand Side, change the value.
If in_a or in_b is changed, out_c is given the new value evaluated by the new value of in_a and in_b.

Creating the input signals and observing the output can be done by the following initial construct.

```
initial begin
      in_b[3:0] = 4'b1100 ;
  $strobe("time1=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
#10 in_a[3:0] = 4'b0101 ;
  $strobe("time2=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
#10 in_b[3:0] = 4'b1111 ;
  $strobe("time3=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
#10 $finish ;
end
```

Set 4'b1100 to in_b at time 0.

Show the values on a terminal right after in_b is given the value 1100.

This is not a good programming style. Use one initial construct for one LHS variable.

```
module and_gate_wo_input ;
reg [3:0] in_a, in_b ; // non-FF
wire [3:0] out_c ;

// logic start
assign out_c[3:0] = in_a[3:0] & in_b[3:0] ;

initial begin
      in_b[3:0] = 4'b1100 ;
  $strobe("time1=%d, in_a=%b, in_b=%b,
                $stime, in_a, in_b, out_c ) ;
#10 in_a[3:0] = 4'b0101 ;
  $strobe("time2=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
#10 in_b[3:0] = 4'b1111 ;
  $strobe("time3=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
#10 $finish ;
end

endmodule
```

Note that this module has no input port nor output port. in_a and in_b are inputs to the AND gate defined in this module, but they are not the inputs to this module. out_c is the output of the AND gate, but it is not an output of this module which is going out of this module. Therefore, no input port nor output port for this module is needed.

Now create this file in your PC and run it to see the result.

file name: and_gate_wo_input_v0.v

A sample result

```
time1=      0, in_a=xxxx,  in_b=1100, out_c=xx00
time2=     10, in_a=0101, in_b=1100, out_c=0100
time3=     20, in_a=0101, in_b=1111, out_c=0101
```

```
initial begin
      in_b[3:0] = 4'b1100 ;
  $strobe("time1=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
#10 in_a[3:0] = 4'b0101 ;
  $strobe("time2=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
#10 in_b[3:0] = 4'b1111 ;
  $strobe("time3=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
#10 $finish ;
end
```

using $strobe

```
initial begin
      in_b[3:0] = 4'b1100 ;
  $strobe("time1=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
#10 in_a[3:0] = 4'b0101 ;
  $strobe("time2=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
#10 in_b[3:0] = 4'b1111 ;
  $strobe("time3=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
#10 $finish ;
end
```

We can use $monitor system task to observe signals.

```
initial begin
      in_b[3:0] = 4'b1100 ;
#10 in_a[3:0] = 4'b0101 ;
#10 in_b[3:0] = 4'b1111 ;
#10 $finish ;
end
```

This is not a good programming style. Use one initial construct for one LHS variable.

using $monitor

```
initial begin
 $monitor("time=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
end
```

RTL programming rules

(1) Use one initial construct for one LHS, Left Hand Side, variable. Do not write several variables on LHS of one initial construct.

```
initial begin
  a = 0 ;
  b = 0 ;
#10 a = 1 ;
#10 b = 1 ;
end
```

```
initial begin
  a = 0 ;
#10 a = 1 ;
end

initial begin
  b = 0 ;
#20 b = 1 ;
end
```

Updating the file and_gate_wo_input.v by using $monitor system task and using one initial construct for one LHS variable will result in the following code block.

```
module and_gate_wo_input ;
reg [3:0] in_a, in_b ;
wire [3:0] out_c ;

// logic start
assign out_c[3:0] = in_a[3:0] & in_b[3:0] ; // (1)

initial begin
#10 in_a[3:0] = 4'b0101 ;
end
initial begin
      in_b[3:0] = 4'b1100 ;
#20 in_b[3:0] = 4'b1111 ;
end

initial begin
 $monitor("time=%d, in_a=%b, in_b=%b, out_c=%b",
                $stime, in_a, in_b, out_c ) ;
#30 $finish ;
end
endmodule
```

Use one initial construct for one LHS

file name: and_gate_wo_input_v1.v

Run this file on your PC and see yourself how it works.

Ex 3-2. Test bench and target module: Disintegrate the logic block written in Ex 3-1 into a target module which must be implemented into silicon and test bench module which does not to into silicon.

```
module and_gate_wo_input ;

    AND gate part

    in_a and in_b generating part
    out_c observing part

endmodule
```

file name: and_gate_wo_input.v

```
module

    AND gate part

endmodule
```

```
module

    in_a and in_b generating part
    out_c observing part

endmodule
```

Disintegrated modules are connected together by instanciating them and connecting them together by wires. Therefore, disintegrated modules must have input port and/or output ports.

RTL programming rules

(1) The data type of an input port of a module must be net. Therefore signals appearing in the input port list must be declared by wire key word in data type declaration.

```
module and_gate( in_a, in_b, ,,, ) ;

input   [3:0] in_a, in_b ;
,,,
wire [3:0] in_a, in_b ;


endmodule
```

signals in input port list must be declared as wire.

Output port can be either reg or wire.

RTL programming rules

(1) A module port list must be written in a sequence of clock, reset, inputs and outputs.

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

input  clk ; // clock signal
input  rst ; // reset signal
input [3:0] in_a, in_b ; // input signals
output [3:0] out_c ; // output signal



endmodule
```

List up the ports in a sequence of;
first clock signals, then reset signals,
and then input signals, and lastly output signals.

First, pull out the target module from and_gate_wo_input module.

```
module and_gate_wo_input ;
reg [3:0] in_a, in_b ;
wire [3:0] out_c ;

// logic start
assign out_c[3:0] = in_a[3:0] & in_b

initial begin
      in_b[3:0] = 4'b1100 ;
#10 in_a[3:0] = 4'b0101 ;
#10 in_b[3:0] = 4'b1111 ;
#10 $finish ;
end
initial begin
 $monitor("time=%d, in_a=%b, in_b
                $stime, in_a, in_b, out_
end
endmodule
```

This part must be the target block.

```
module and_gate( in_a, in_b, out_c ) ;

input  [3:0] in_a, in_b ;
output [3:0] out_c ;

wire [3:0] in_a, in_b ;
wire [3:0] out_c ;

// logic start
assign out_c[3:0] =
      in_a[3:0] & in_b[3:0] ; // (1)

endmodule
```

in_a and in_b must come from outside of this module and out_c must go out of this module. Therefore, they must be declared as input and output ports

Next, extract test input and observe block which must not go into silicon.

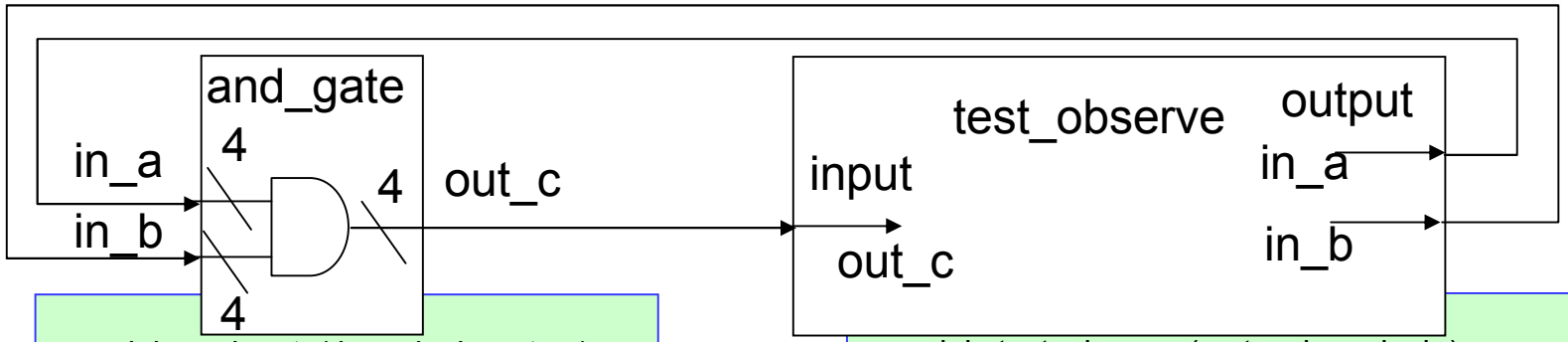
```
module and_gate_wo
reg [3:0] in_a, in_b ;
wire [3:0] out_c ;

// logic start
assign out_c[3:0] = in

initial begin
      in_b[3:0] = 4'b110
#10 in_a[3:0] = 4'b010
#10 in_b[3:0] = 4'b111
#10 $finish ;
end
initial begin
 $monitor("time=%d, i
              $stime, in
end
endmodule
```

```
module test_observe ( out_c, in_a, in_b ) ;

input   [3:0] out_c;
output [3:0] in_a, in_b ;

reg [3:0] in_a, in_b ;
wire [3:0] out_c ;

initial begin
      in_b[3:0] = 4'b1100 ;
#10 in_a[3:0] = 4'b0101 ;
#10 in_b[3:0] = 4'b1111 ;
#10 $finish ;
end
initial begin
  $monitor("time=%d, in_a=%b, in_b=%b,   out_c=%b",
              $stime, in_a, in_b, out_c ) ;
end
endmodule
```

Note that out_c is an input, and in_a and in_b are outputs in this module, whereas in_a and in_b are inputs and out_c is an output in the target module.

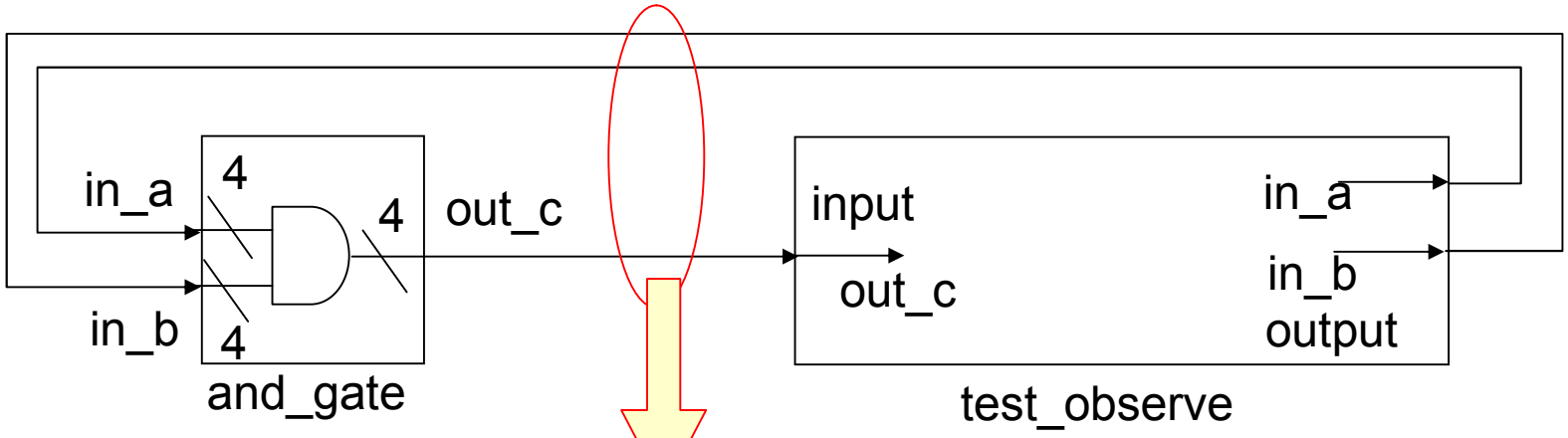
And also note that in_a and in_b are declared as register because procedural assignment is used to assign values to these signals.

Now the original module is disintegrated into and_gate module and test_observe module.
We need to connect these two together as shown below by introducing a new module, and_gate_top_lvl.




```
module and_gate( in_a. in_b, out_c ) ;

input   [3:0] in_a, in_b ;
output [3:0] out_c ;

wire [3:0] in_a, in_b ;
wire [3:0] out_c ;

// logic start
assign out_c[3:0] =
     in_a[3:0] & in_b[3:0] ; // (1)

endmodule
```

```
module test_observe ( out_c, in_a, in_b ) ;

input   [3:0] out_c;
output [3:0] in_a, in_b ;

reg [3:0] in_a, in_b ;
wire [3:0] out_c ;

initial begin
      in_b[3:0] = 4'b1100 ;
#10 in_a[3:0] = 4'b0101 ;
#10 in_b[3:0] = 4'b1111 ;
#10 $finish ;
end
initial begin
   $monitor("time=%d, in_a=%b, in_b=%b,   out_c=%b",
              $stime, in_a, in_b, out_c ) ;
end
endmodule
```




```
module and_gate_top_lvl ;

wire [3:0] in_a, in_b ;
wire [3:0] out_c ;

// module connection
and_gate    and_gate_01 ( .in_a( in_a ), .in_b ( in_b ),
                          .out_c(out_c) ) ;
test_observe   test_observe _01 (.out_c(out_c),
                          .in_a( in_a ), .in_b ( in_b ) ) ;

endmodule
```

and_gate_top_lvl has no input nor output port because no signal is going out of this module and coming in from outside.

In the top level module, all the connecting signal lines must be defined as nets.

```
// module connection
and_gate    and_gate_01 ( .in_a( in_a ). .in_b ( in_b ),
                                .out_c(out_c) ) ;
test_observe   test_observe _01 (.out_c(out_c),
                                .in_a( in_a ). .in_b ( in_b ) ) ;
```

"and_gate and_gate_01" means that the module and_gate is incorporated in and_gate_top_lvl module, and the instance is given the name "and_gate_01".

It is recommended to give the name to an instance such as "modulename_01".

Instance is an object which goes into silicon. The same module can be incorporated into a module again and again. If it is incorporated 3 times, there must be three instances, one instance for one incorporation.

The connection among the two modules are done using "port connection by name" in the example above. ".in_a" means the port name "in_a". ".in_a(in_a)" means signal "in_a" is connected to the port "in_a".

Disintegrated modules can be in one file or in several files as shown in this slide or in the next slide.

```
module and_gate_top_lvl ;
    •
    •
    •
    •
endmodule

module and_gate( ,,,, ) ;
    •
    •
    •
    •
endmodule

module test_observe( ,,, ) ;
    •
    •
    •
    •
endmodule
```

file name: and_gate_top_lvl.v

```
module and_gate_top_lvl ;
    •
    module connection
    •
endmodule
`include "and_gate.v"
`include "test_observe.v"
```

file name: and_gate_top_lvl.v

```
module and_gate( ,,,, ) ;
    Logic to by synthesized
endmodule
```

file name: and_gate.v

```
module test_observe( ,,, ) ;
    Logic for simulation
endmodule
```

file name: test_observe.v

```
module and_gate_top_lvl ;
      •
      module connection
      •
 endmodule
 `include "and_gate.v"
 `include "test_observe.v"
```

file name: and_gate_top_lvl.v

```
module and_gate( ,,,, ) ;
      Logic to by synthesized
 endmodule
```

file name: and_gate.v

```
module test_observe( ,,, ) ;
      Logic for simulation
endmodule
```

file name: test_observe.v

```
module test_and_gate ;
      •
      •
      module connection
      •
      •
      •
      Logic for simulation
      •
 endmodule
 `include "and_gate.v"
```

file name: test_and_gate.v

```
module and_gate( ,,,, ) ;
      Logic to by synthesized
 endmodule
```

file name: and_gate.v

Now, create files for Exercise 3-2 by using the style shown on the right.

```
module and_gate_top_lvl ;
wire [3:0] in_a, in_b ;
wire [3:0] out_c ;
// module connection
and_gate    and_gate_01 ( .in_a( in_a ). .in_b ( in_b ),
                          .out_c(out_c) ) ;
test_observe   test_observe _01 (.out_c(out_c),
                          .in_a( in_a ). .in_b ( in_b ) ) ;

endmodule
```

```
module and_gate( in_a. in_b, out_c ) ;
input   [3:0] in_a, in_b ;
output [3:0] out_c ;
wire [3:0] in_a, in_b ;
wire [3:0] out_c ;
// logic start
assign out_c[3:0] =  in_a[3:0] & in_b[3:0] ; // (1)
endmodule
```

```
module test_observe ( out_c, in_a, in_b ) ;
input   [3:0] out_c;
output [3:0] in_a, in_b ;
reg [3:0] in_a, in_b ;
wire [3:0] out_c ;
initial begin
      in_b[3:0] = 4'b1100 ;
#10 in_a[3:0] = 4'b0101 ;
#10 in_b[3:0] = 4'b1111 ;
#10 $finish ;
end
initial begin
   $monitor("time=%d, in_a=%b, in_b=%b,   out_c=%b",
              $stime, in_a, in_b, out_c ) ;
end
endmodule
```

```
module test_and_gate ;
      •
      •
      •
      module connection
      •
      •
      •
      Logic for simulation
      •
endmodule
`include "and_gate.v"
```

file name: test_and_gate.v

```
module and_gate( ,,,, ) ;
      Logic to by synthesized
endmodule
```

file name: and_gate.v

```
module and_gate( in_a. in_b, out_c ) ;
input   [3:0] in_a, in_b ;
output [3:0] out_c ;
wire [3:0] in_a, in_b ;
wire [3:0] out_c ;
// logic start
assign out_c[3:0] =  in_a[3:0] & in_b[3:0] ; // (1)
endmodule
```

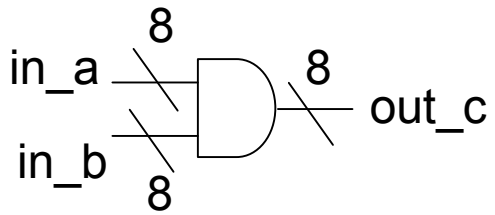
file name: and_gate.v

```
module test_and_gate ;
 reg [3:0] in_a, in_b ; // inputs for AND gate
 wire [3:0] out_c ;   // output of AND gate

 and_gate and_gate_01 ( .in_a(in_a), .in_b(in_b),
                                .out_c(out_c) )
 initial begin
       in_b[3:0] = 4'b1100 ;
 #10 in_a[3:0] = 4'b0101 ;
 #10 in_b[3:0] = 4'b1111 ;
 #10 $finish ;
 end
 initial begin
   $monitor("time=%d, in_a=%b, in_b=%b,   out_c=%b",
              $stime, in_a, in_b, out_c ) ;
 end
 endmodule

`include "and_gate.v"
```

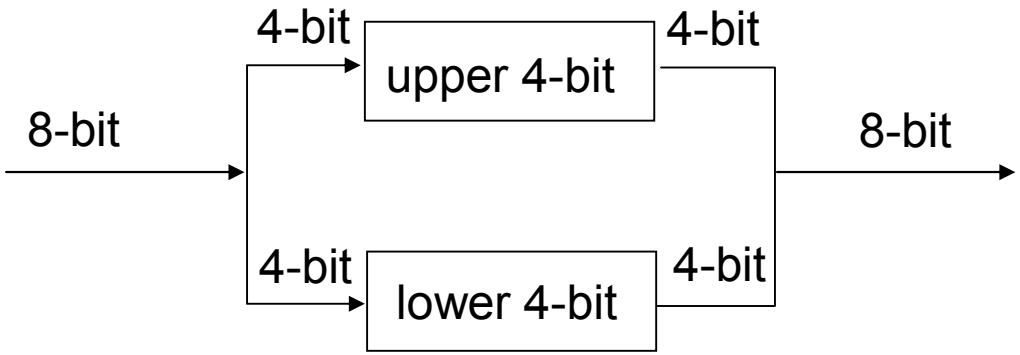
file name: test_and_gate.v

Run this file and see the result yourself.

Ex 3-3. Using two instances of one module: Write a module equivalent to the gate shown below by using and_gate.v file without changing any single character in the file.

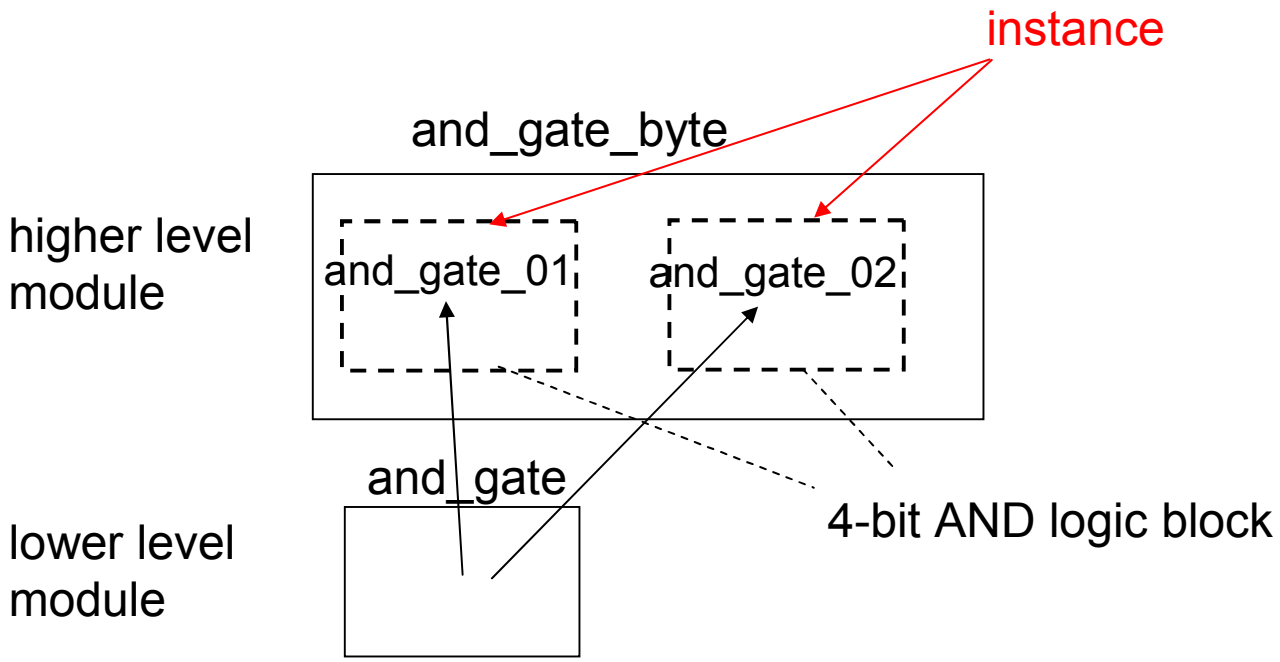


In this exercise, we will learn how to use existing modules. Let's design 8-bit AND logic block by using 4-bit AND logic block, and_gate.v which we designed in Ex 3-2.
Disintegrate 8-bit signals into two 4-bit signals and apply and_gate module to each 4-bit signals.



and_gate module can be used twice in the and_gate_byte module. Each of the and_gate module used in and_gate_byte is called “instance”.

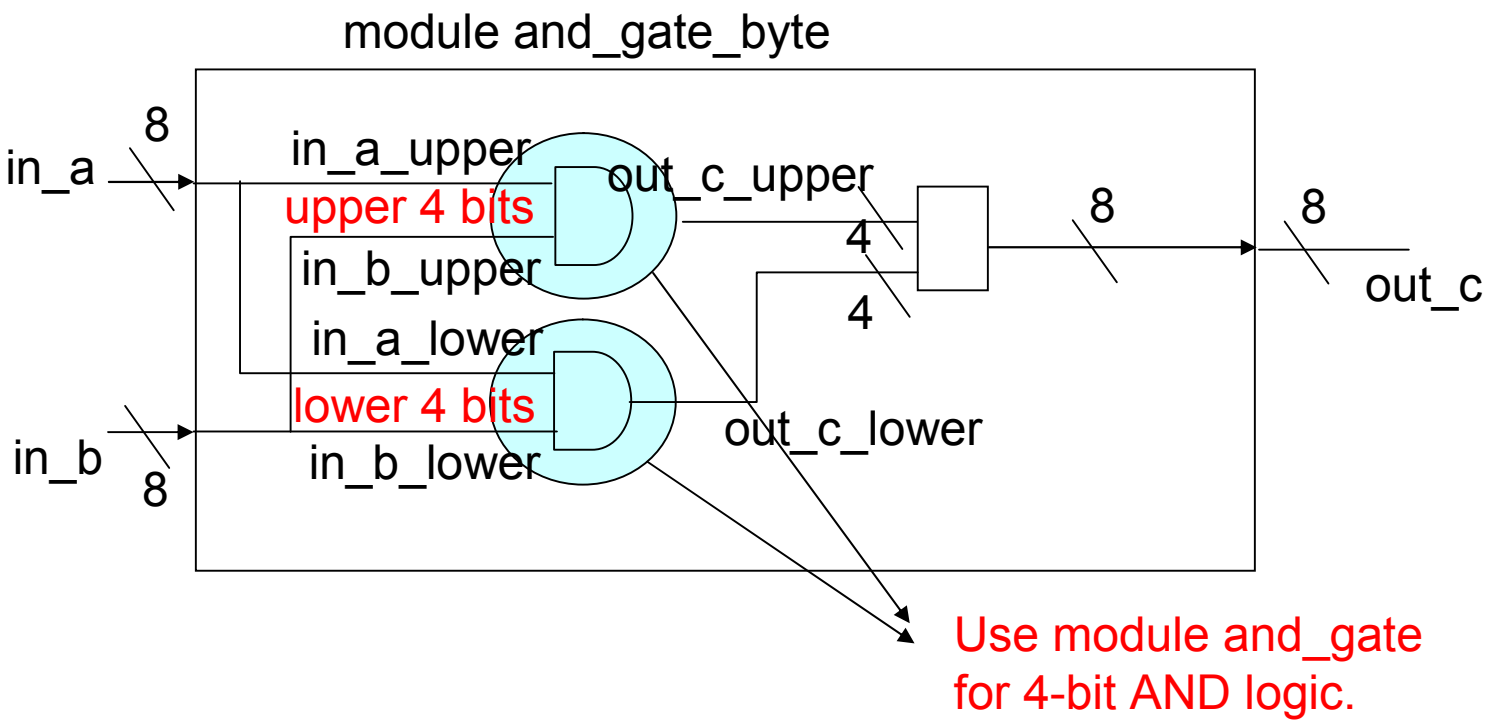
There must be two instances, and_gate_01 and and_gate_02, because two and_gate is needed to build 8-bit AND gate.



The wire connection among the two instances and the ports of and_gate_byte module are shown in this slide.

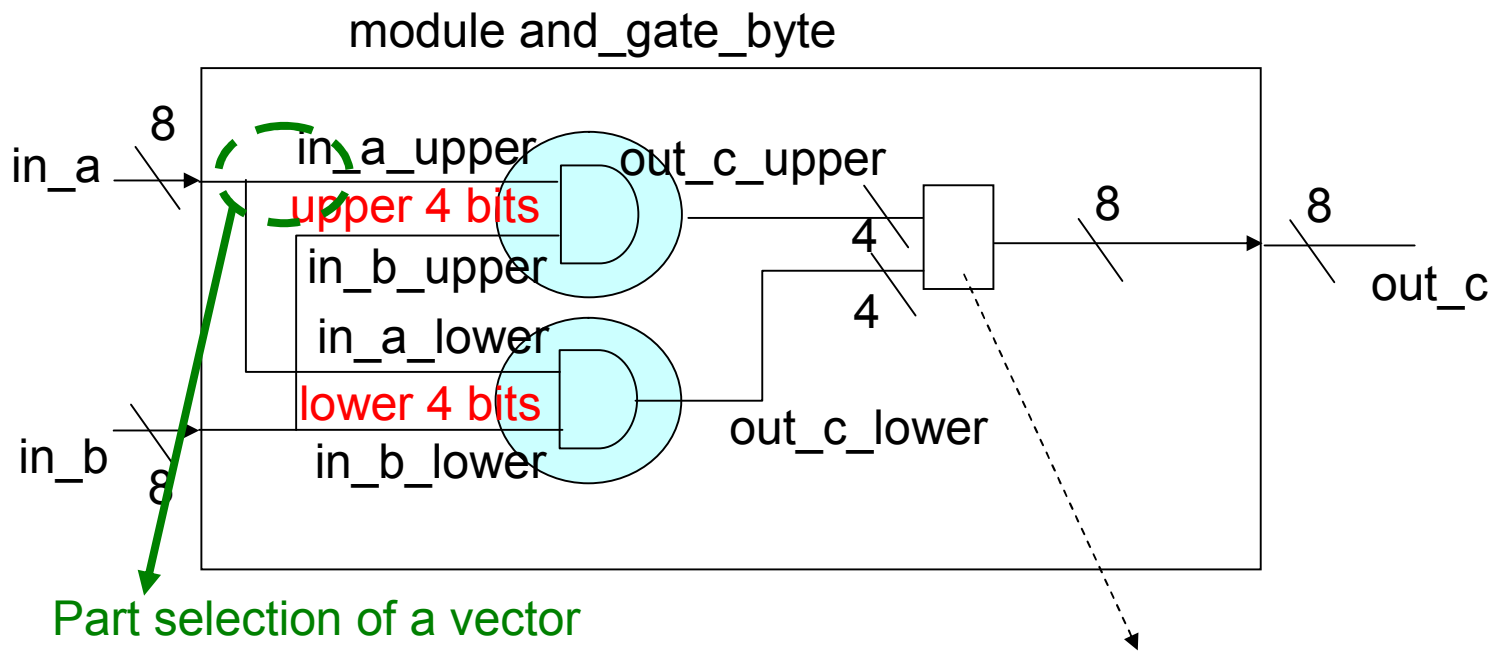
Upper 4-bit of the 8-bit input in_a must go into and_gate_01 and the output of and_gate_01 must go into upper 4-bit of the 8-bit out_c.

Lower 4-bit of the 8-bit input in_a must go into and_gate_02 and the output of and_gate_02 must go into lower 4-bit of the 8-bit out_c.



To identify a part of n-bit vector signal, such as upper 4-bit of 8-bit signal, we can use part selection of a vector as shown in this slide.

in_b[7:4] means 4-bit signal consists of in_b[7], in_b[6], in_b[5], and in_b[4].



assign in_b_upper[3:0] = in_b[7:4] ;

assign out_c[7:4] = out_c_upper [3:0] ;

Now, write the module and_gate_byte and name the file and_gate_byte_v0.v

```
module and_gate_byte ( in_a, in_b, out_c ) ;
  input [7:0] in_a, in_b ;
  output[7:0] out_c ;

  wire [7:0] in_a, in_b ;
  wire [7:0] out_c ;
  // internal connecting wire
  wire [3:0] in_a_upper, in_a_lower ;
  wire [3:0] in_b_upper, in_b_lower ;
  wire [3:0] out_c_upper, out_c_lower ;

  // separate byte to upper and lower
  assign in_a_upper[3:0] = in_a[7:4] ;
  assign in_a_lower[3:0] = in_a[3:0] ;
  assign in_b_upper[3:0] = in_b[7:4] ;
  assign in_b_lower[3:0] = in_b[3:0] ;
  // merge upper and lower
  assign out_c[7:4] = out_c_upper [3:0] ;
  assign out_c[3:0] = out_c_lower [3:0] ;
  // module conndection
  and_gate and_gate_01 ( .in_a( in_a_upper ),.in_b ( in_b_upper ),
                         .out_c(out_c_upper) ) ;
  and_gate and_gate_02 ( .in_a( in_a_lower ), .in_b ( in_b_lower ),
                         .out_c(out_c_lower) ) ;
endmodule
```

The module and_gate_byte has 8-bit ports because the target logic is 8-bit AND logic.

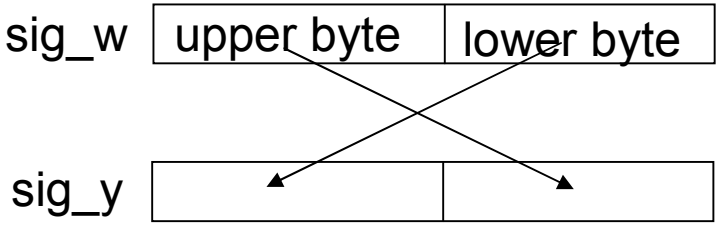
"assign in_a_upper[3:0] = in_a[7:4];" means that
in_a[7] is assigned to in_a_upper[3],
in_a[6] to in_a_upper[2],
in_a[5] to in_a_upper[1], and
in_a[4] to in_a_upper[0].

Next, think of improving this code by using { } Verilog operator. { } can be used as shown in the next slide.

file name: and_gate_byte_v0.v

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] ;
```

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

```
module and_gate_byte ( in_a, in_b, out_c ) ;
  input [7:0] in_a, in_b ;
  output[7:0] out_c ;

  wire [7:0] in_a, in_b ;
  wire [7:0] out_c ;
  // internal connecting wire
  wire [3:0] in_a_upper, in_a_lower ;
  wire [3:0] in_b_upper, in_b_lower ;
  wire [3:0] out_c_upper, out_c_lower ;

  // separate byte to upper and lower
  assign { in_a_upper, in_a_lower }  = in_a ;
  assign { in_b_upper, in_b_lower }  = in_b ;
  // merge upper and lower
  assign out_c = { out_c_upper, out_c_lower } ;
  // module conndection
  and_gate   and_gate_01 ( .in_a( in_a_upper ),.in_b ( in_b_upper ),
                           .out_c(out_c_upper) ) ;
  and_gate   and_gate_02 ( .in_a( in_a_lower ), .in_b ( in_b_lower ),
                           .out_c(out_c_lower) ) ;
endmodule
```

An example of improved code block using { }.

file name: and_gate_byte_v1.v

Next, write a test bench to test this module.
Name the file: test_and_gate_byte.v.

```
module test_and_gate_byte ;
reg [7:0] in_a, in_b ;
wire [7:0] out_c ;

and_gate_byte   and_gate_byte_01   (
                .in_a( in_a), .in_b(in_b),
                .out_c(out_c)
                ) ;
initial
 begin
      in_b[7:0] = 8'hC0 ;
#10 in_a[7:0] = 8'h55 ;
#10 in_b[7:0] = 8'hF3 ;
#10 $finish ;
end
initial   begin
   $monitor("time=%d, in_a=%h,  in_b=%h, out_c=%h",
                $stime, in_a, in_b, out_c ) ;
end
endmodule
`include "and_gate.v"
`include "and_gate_byte_v1.v"
```

Now, run this file on your PC to see the result.

file name: test_and_gate_byte.v

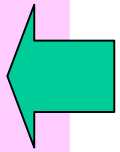
A sample result

```
time=        0, in_a=xx,  in_b=c0, out_c=X0
time=       10, in_a=55,  in_b=c0, out_c=40
time=       20, in_a=55,  in_b=f3,  out_c=51
```

Next think further simplification.

You may use part selection of a vector.

```
module and_gate_byte ( in_a, in_b, out_c ) ;
  input [7:0] in_a, in_b ;
  output[7:0] out_c ;

  wire [7:0] in_a, in_b ;
  wire [7:0] out_c ;
  // internal connecting wire
  wire [3:0] in_a_upper, in_a_lower ;
  wire [3:0] in_b_upper, in_b_lower ;
  wire [3:0] out_c_upper, out_c_lower ;

  // separate byte to upper and lower
  assign { in_a_upper, in_a_lower }  = in_a ;
  assign { in_b_upper, in_b_lower }  = in_b ;
  // merge upper and lower
  assign out_c = { out_c_upper, out_c_lower } ;
  // module conndection
  and_gate   and_gate_01 ( .in_a( in_a_upper ),.in_b ( in_b_upper ),
                           .out_c(out_c_upper) ) ;
  and_gate   and_gate_02 ( .in_a( in_a_lower ), .in_b ( in_b_lower ),
                           .out_c(out_c_lower) ) ;
endmodule
```



Can't we eliminate this part?

file name: and_gate_byte_v1.v

```
module and_gate_byte ( in_a, in_b, out_c ) ;
  input [7:0] in_a, in_b ;
  output[7:0] out_c ;

  wire [7:0] in_a, in_b ;

  wire [3:0] out_c_upper, out_c_lower ; // internal connecting wire

  wire [7:0] out_c = { out_c_upper, out_c_lower } ;

  // separate byte to upper and lower
  and_gate and_gate_01 ( .in_a( in_a[7:4] ), .in_b ( in_b[7:4] ),
                         .out_c(out_c_upper) ) ;
  and_gate and_gate_02 ( .in_a( in_a[3:0] ), .in_b ( in_b[3:0] ),
                         .out_c(out_c_lower) ) ;

endmodule
```

This is called net declaration assignment.
It is same to;

```
wire [7:0] out_c ;
assign out_c = { out_c_upper, out_c_lower }
;
```

Part selection is used. 4-bit out of 8-bit vector is used to connect ports.

file name: and_gate_byte_v2.v

It is not recommended to use net declaration assignment in general. However, we may use it when the assignment is very simple and obvious.

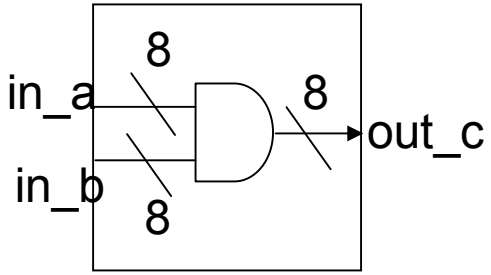
```
module and_gate_byte ( in_a, in_b, out_c ) ;
  input [7:0] in_a, in_b ;
  output[7:0] out_c ;

  wire [7:0] in_a, in_b ;

  wire [3:0] out_c_upper, out_c_lower ; // internal connecting wire

  wire [7:0] out_c = { out_c_upper, out_c_lower } ;

  // separate byte to upper and lower
  and_gate and_gate_01 ( .in_a( in_a[7:4] ),.in_b ( in_b[7:4] ),
                         .out_c(out_c_upper) ) ;
                                  out_c[7:4]
  and_gate and_gate_02 ( .in_a( in_a[3:0] ), .in_b ( in_b[3:0] ),
                         .out_c(out_c_lower) ) ;
                                  out_c[3:0]
endmodule
```

We can eliminate these part by using part select of out_c.

file name: and_gate_byte_~~v2~~.v
v3

Run those files, and_gate_byte_v0.v, _v1.v, _v2.v, and_v3.v on your PC and check yourself if the results are exactly the same.

Ex 3-4. How to use function: Write a function func_and_gate having the same functionality to module and_gate, and rewrite the code in and_gate_byte_v3.v using it.
Name the updated module and_gate_byte_f.



and_gate_byte_f

Logic in and_gate.v is so simple and there is no memory element in it. Therefore, it can be described by using function instead of module.

Function can be invoked as many time as you like. Each line invoking a function needs silicon area for the function. That is, if a function is called in many lines, silicon size will increase.

```
module

    assign y = func_name( argy,,,) ;
    assign w = func_name( argw,,,) ;

    function [n:0] func_name ;
    input ,,,, ;
    reg ,,, ;

    endfunction

endmodule
```

A function can be invoked by continuous assign and by procedural assign.

A function definition starts with function key word and end with endfunction key word. It must have at least one input argument.

RTL programming rules

(1) function must be declared inside the module.
(2) function can not go beyond module boundary.
(3) In function, LHS must be register data type.
(4) function can have only on output. (No output argument allowed, output is given through its name.)
(5) function must have at least one input argument. It may have several inputs.
(6) No # nor @ allowed in function.

```
module and_gate( in_a. in_b, out_c ) ;

  input   [3:0] in_a, in_b ;
  output [3:0] out_c ;

  wire [3:0] in_a, in_b ;
  wire [3:0] out_c ;

  // logic start
  assign out_c[3:0] =
       in_a[3:0] & in_b[3:0] ; // (1)

endmodule
```

Because the moduel has two inputs in_a and in_b, the function must have two inputs arguments.

```
function [3:0]  f_and_gate ;

input   [3:0] in_a, in_b ;

// logic start
  f_and_gate [3:0] = in_a[3:0] & in_b[3:0] ;
//
endfunction
```

We do not have to declare data type for the input arguments of a function. We do not have to declare data type of in_a and in_b.

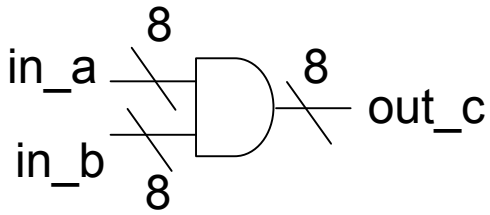
The output of a function is defined by the function name. We have to assign values to f_and_gate [3:0]. If the output consists of mutli bits, we need bit width declaration for the function.

```
module and_gate_byte ( in_a, in_b, out_c ) ;
  input [7:0] in_a, in_b ;
  output[7:0] out_c ;

  wire [7:0] in_a, in_b ;
  wire [7:0] out_c ;
  //
  assign out_c[7:4] = f_and_gate( in_a[7:4] , in_b[7:4] ) ;
  assign out_c[3:0] = f_and_gate( in_a[3:0] , in_b[3:0] ) ;
  //
  // function
  function [3:0]  f_and_gate ;
  input   [3:0] in_a, in_b ;
  // logic start
   f_and_gate [3:0] = in_a[3:0] & in_b[3:0] ;
  //
  endfunction
  //
endmodule
```

and_gate_byte_f.v

Run this file on your simulator and see yourself if it works correctly.

Ex 3-5. Parameterize bit width: Write a module equivalent to the gate shown below by modifying and_gate.v file. Use parameter to extend the bit width of in_a, in_b, and out_c.



In this exercise , we will learn how to use parameter.

Usually we do not use 4-bit AND logic block to create 8-bit AND logic block. We apply AND operator directly on 8-bit signals to get 8-bit AND. In this exercise we will learn a method to make bit size flexible.

First, think of changing bit width declaration of the signals.

We can modify and_gate module by changing 3 to 7 to get 8-bit AND as below.

```
module and_gate( in_a. in_b, out_c ) ;

input   [3:0] in_a, in_b ;
output [3:0] out_c ;

wire [3:0] in_a, in_b ;
wire [3:0] out_c ;

// logic start
assign out_c[3:0] =
      in_a[3:0] & in_b[3:0] ; // (1)

endmodule
```

Rewriting the bit width

```
module and_gate( in_a. in_b, out_c ) ;

input   [7:0] in_a, in_b ;
output [7:0] out_c ;

wire [7:0] in_a, in_b ;
wire [7:0] out_c ;

// logic start
assign out_c[7:0] =
      in_a[7:0] & in_b[7:0] ; // (1)

endmodule
```

Isn't there more flexible way??

Parameterize

```
module and_gate( in_a, in_b, out_c ) ;

 // parameter
 parameter SIG_BW = 8 ;

 // port
 input   [SIG_BW-1:0] in_a, in_b ;
 output [SIG_BW-1:0] out_c ;

 wire [SIG_BW-1:0] in_a, in_b ;
 wire [SIG_BW-1:0] out_c ;

 // logic start
 assign out_c = in_a & in_b ; // (2)

 endmodule
```

file name: and_gate_p.v

An example shown in this slide defines a parameter named SIG_BW whose value is 8.

"input [SIG_BW-1:0] in_a, in_b ;" means that input port in_a and in_b have the bit-width SIG_BW, 8 in this example.

By using this parameter, when we are asked to create AND logic for 32 bits signal, only one line, parameter SIG_BW = 8 ;, has to be changed to "parameter SIG_BW = 32 ;". No other lines have to be modified.

This is an answer for Ex 3-5.

Parameterize

```
module and_gate( in_a, in_b, out_c ) ;

 // parameter
 parameter SIG_BW = 8 ;

 // port
 input   [SIG_BW-1:0] in_a, in_b ;
 output [SIG_BW-1:0] out_c ;

 wire [SIG_BW-1:0] in_a, in_b ;
 wire [SIG_BW-1:0] out_c ;

 // logic start
 assign out_c = in_a & in_b ; // (2)

 endmodule
```

file name: and_gate_p.v

When the bit width of the parameter itself has to be defined, we can use declarations shown below.

Unless the bit width is not declared, parameter has 32-bit length.

```
parameter SIG_BW = 16'h08 ;

parameter [15:0] SIG_BW = 8 ;
```

Now, type the file on the left and create a test bench file “test_and_gate_p.v” to test the module on the left.

```
module test_and_gate_p ;
// parameter
parameter SIG_BW = 16 ;
defparam and_gate_01.SIG_BW = SIG_BW ;
// signals
reg [SIG_BW-1:0] in_a, in_b ;
wire [SIG_BW-1:0] out_c ;

and_gate and_gate_01 (
                .in_a(in_a), .in_b(in_b),
                .out_c(out_c)
               ) ;
initial begin
      in_b = 16'hC0C0 ;
#10 in_a = 16'h55AA ;
#10 in_b = 16'hF310 ;
#10 $finish ;
end
initial begin
   $monitor("time=%d, in_a=%h,  in_b=%h, out_c=%h",
                $stime, in_a, in_b, out_c ) ;
end
endmodule
`include "and_gate_p.v"
```

file name: test_and_gate_p.v

Parameter can be overwritten by using defparam key word.

"defparam and_gate_01.SIG_BW = 32;" means the parameter SIG_BW defined in an instance and_gate_01 shall be overwritten by 32.

"defparam and_gate_01.SIG_BW = SIG_BW ;" means that the parameter defined in the instance and_gate_01 shall be overwritten by SIG_BW defined in the module test_and_gate_p, that is 16.

Now, run this file on your PC and see yourself how defparam works.

Ex 3-6. Assign for combinational logic: Write a module equivalent to the truth table shown below. An input port is 4-bit in_a and an output port is 3-bit out_y. Use continuous assign only to write the module. Do not use always constructs for the module.

in_a[3]	in_a[2]	in_a[1]	in_a[0]	out_y
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

In this exercise , we will learn how to use assign and conditional operator. A sentence may

assign yyy = (*expression*)? *expression1* : *expression2* ;

First, expression is evaluated and if it is true then the value of expression1 is assigned to yyy. If the expression is not true, then the value of expression2 is assigned to yyy.

Do not try to find a solution for all bits at one time. Think the logic for each bit. First find the logic for out_y[2] and then out_y[1] and then finally out_y[0].

Investigating the truth table, we can find out that out_y[2] is 1 only when in_a is 4’b0000.

in_a[3]	in_a[2]	in_a[1]	in_a[0]	out_y
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

The sentence below will assign 0 to out_y[2] if in_a is not 4’b0000, and assign 1 if in_a is 4’b0000.

```
assign out_y[2] = ( in_a == 4'b0000 )? 1 : 0 ;
```

Now, did you get the idea??

Think how can we write a code line for out_y[1] before going to the next slide.

in_a[3]	in_a[2]	in_a[1]	in_a[0]	out_y
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

```
assign out_y[2] = ( in_a == 4'b0000 )? 1 : 0 ;

assign out_y[1] = ( (in_a[3] == 1'b1 ) | ( in_a[3:2]==2'b01) )? 1 : 0 ;
```

Before going to the next slide,
think about out_y[0].

```
module prity_encdr ;

endmodule
```

in_a[3]	in_a[2]	in_a[1]	in_a[0]	out_y
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

```
assign out_y[2] = ( in_a == 4'b0000 )? 1 : 0 ;

assign out_y[1] = ( (in_a[3] == 1'b1 ) | ( in_a[3:2]==2'b01) )? 1 : 0 ;

assign out_y[0] = ( (in_a[3] == 1'b1 ) | ( in_a[3:1]==3'b001) )? 1 : 0 ;
```

Now create a file named prity_encdr.v by using the idea on this page.

Parameter is not used, because this logic can not work correctly for bit sizes other than 4.

```
module prity_encdr ( in_a, out_y ) ;
input [3:0]   in_a ;
output [2:0] out_y ;
//
wire [3:0]   in_a ;
wire [2:0] out_y ;
//
assign out_y[2] = ( in_a == 4'b0000 )? 1'b1 : 1'b0 ;
assign out_y[1] = ( (in_a[3] == 1'b1 ) | ( in_a[3:2]==2'b01)   )? 1'b1 : 1'b0 ;
assign out_y[0] = ( (in_a[3] == 1'b1 ) | ( in_a[3:1]==3'b001) )? 1'b1 : 1'b0 ;

endmodule
```

file name : prity_encdr.v

```
integer i ;
for ( i=0 ; i<=15 ; i=i+1 ) begin


 end
```

Next, create a test bench to test the file above. Use “for statement” to generate all the possible patterns of in_a[3:0].
Name the file test_prity_encdr.v.

```
module test_prity_encdr ;
reg [3:0] in_a;
wire [2:0] out_y ;

prity_encdr prity_encdr_01 (
                    .in_a(in_a), .out_y(out_y)
                   ) ;
integer i ;
initial begin
 for ( i=0 ; i<=15 ; i=i+1 ) begin
   #10 in_a = i ;
 end
#10 $finish ;
end

initial begin
  $monitor( "in_a=%b. out_y=%d", in_a, out_y ) ;
end
endmodule
`include "prity_encdr.v"
```

Without this delay, the simulation will not end successfully.
Think why??

file name : test_prity_encdr.v

The for statement is equal to the following code lines.

```
module test_prity_encdr ;
reg [3:0] in_a;
wire [2:0] out_y ;

prity_encdr prity_encdr_01 (
                .in_a(in_a), .out_y(out_y)
               ) ;
integer i ;
initial begin
 for ( i=0 ; i<=15 ; i=i+1 ) begin
   #10 in_a = i ;
 end
#10 $finish ;
end

initial begin
  $monitor( "in_a=%b. out_y=%d", in_a, out_y ) ;
end
endmodule
`include “prity_encdr.v”
```

file name : test_prity_encdr.v

```
initial begin
  #10 in_a = 0 ;
  #10 in_a = 1 ;
  #10 in_a = 2 ;
  #10 in_a = 3 ;
  #10 in_a = 4 ;
  #10 in_a = 5 ;
        ⋮
  #10 in_a = 12 ;
  #10 in_a = 13 ;
  #10 in_a = 14 ;
  #10 in_a = 15 ;
```

Run this file on your PC to see the result.

```
module test_prity_encdr ;
reg [3:0] in_a;
wire [2:0] out_y ;

prity_encdr prity_encdr_01 (
                .in_a(in_a), .out_y(out_y)
               ) ;
integer i ;
initial begin
 for ( i=0 ; i<=15 ; i=i+1 ) begin
   #10 in_a = i ;
 end
#10 $finis
end

initial beg
  $monito
end
endmodu
`include "prity_encdr.v"
```

file name : test_prity_encdr.v

```
assign out_y[2] =
        ( in_a == 4'b0000 )?
                    1'b1 : 1'b0 ;
```

if in_a is unkown

→ (in_a == 4'b0000) is 1'bx

→ out_y[2] is 1'bx ;

in_a is not given any value before t=10.

```
in_a=xxxx. out_y=x
in_a=0000. out_y=4
in_a=0001. out_y=0
in_a=0010. out_y=1
in_a=0011. out_y=1
in_a=0100. out_y=2
in_a=0101. out_y=2
in_a=0110. out_y=2
in_a=0111. out_y=2
in_a=1000. out_y=3
in_a=1001. out_y=3
in_a=1010. out_y=3
in_a=1011. out_y=3
in_a=1100. out_y=3
in_a=1101. out_y=3
in_a=1110. out_y=3
in_a=1111. out_y=3
```

An example of the simulation result

```
module test_prity_encdr ;
reg [3:0] in_a;
wire [2:0] out_y ;

prity_encdr prity_encdr_01 (
                    .in_a(in_a), .out_y(out_y)
                   ) ;
initial begin
// for ( i=0 ; i<=15 ; i=i+1 ) begin
  for ( in_a = 0 ; in_a<=15; in_a = in_a +1 ) begin
      #10 ;
 end
#10 $finish ;
end

initial begin
  $monitor( "in_a=%b. out_y=%d", in_a, out_y ) ;
end
endmodule
`include “prity_encdr.v”
```

file name : test_prity_encdr_ng.v

Modify the file as show on the left and name it test_prity_encdr_ng.v.

Run it to see yourself what will happen.

```
module test_prity_encdr ;
reg [3:0] in_a;
wire [2:0] out_y ;

prity_encdr prity_encdr_01 (
                    .in_a(in_a), .out_y(out_y)
                   ) ;
initial begin
// for ( i=0 ; i<=15 ; i=i+1 ) begin
  for ( in_a = 0 ; in_a<=15; in_a = in_a +1 ) begin
      #10 ;
 end
#10 $finish ;
end

initial begin
  $monitor( "in_a=%b. out_y=%d", in_a, out_y ) ;
end
endmodule
`include “prity_encdr.v”
```

file name : test_prity_encdr_ng.v

Modify the file as show on the left and name it test_prity_encdr_ng.v.

Run this to see yourself what will happen.

Your simulator will fall into infinite loop because in_a cannot become larger than 15. Note that in_a is a 4-bit signal. 15 added by one is 0 for 4-bit data.

Ex 3-7. Always for combinational logic: Write a module equivalent to the truth table, the same to the previous Ex, shown below using an always construct. An input port is 4-bit in_a and an output port is 3-bit out_y.

in_a[3]	in_a[2]	in_a[1]	in_a[0]	out_y
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

Combinational logic used many times in a module must be written by using function. But if it is used only once, always construct can be used.

```
always @ ( a or b or c or ,,,,  ) begin
        •
        •
        •
        •
        •
  out_y =   ,,,,,, ;
end
```

sensitivity list

RTL programming rules

always construct can generate combinational logic block if
(1) all signals appearing in RHS, Right Hand Side, are written in the sensitivity list as change value events,
(2) all variable appearing in LHS are given values in every path, and
(3) blocking procedural assign is used for assignment.

```
a = b & c ;
```

This is blocking procedural assignment.

RHS: Right Hand Side
LHS: Left Hand Side

In this exercise, we must use always construct, which is a procedure. We can use case statement instead of simple conditional operator.

First, think of case statement which is equivalent to the truth table.

in_a[3]	in_a[2]	in_a[1]	in_a[0]	out_y
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

case expression

case item

```
casez ( in_a )
  4'b1??? : begin
              out_y = 3'd3 ;
      end
  4'b01?? : begin
              out_y = 3'd2 ;
      end
  4'b001? : begin
              out_y = 3'd1 ;
      end
  4'b0001 : begin
              out_y = 3'd0 ;
      end
  4'b0000 : begin
              out_y = 3'd4 ;
      end
endcase
```

For casez, ? means don't care. For example, case item 4'b01?? means that case expression and the case item will match if case expression is 4'b0100, 4'b0101, 4'b0110, or 4'b0111.

The casez statement must go into the always construct because case statement is allowed only in procedures.

always @ (in_a) begin

•
•
•
•

end

in_a[3]	in_a[2]	in_a[1]	in_a[0]	out_y
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

case expression

case item

```
casez ( in_a )
  4'b1??? : begin
              out_y = 3'd3 ;
      end
  4'b01?? : begin
              out_y = 3'd2 ;
      end
  4'b001? : begin
              out_y = 3'd1 ;
      end
  4'b0001 : begin
              out_y = 3'd0 ;
      end
  4'b0000 : begin
endc
```

Modify the file prity_encdr.v by using always construct instead of using assign and name it prity_encdr_alwy.v.

The module prity_encdr must looks like an example shown in this slide.

out_y can not be net data type in this module. It must be register data type, because it appears on LHS of a procedural assignment.

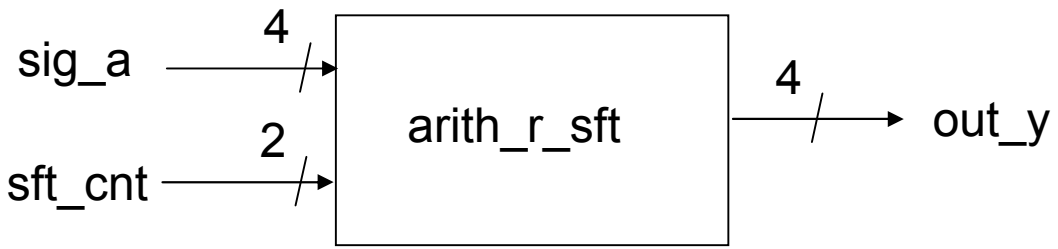
```
module prity_encdr ( in_a, out_y ) ;
input [3:0]   in_a ;
output [2:0] out_y ;
//
wire [3:0]   in_a ;
reg [2:0] out_y ;
//
always @ ( in_a ) begin
  casez ( in_a )
    4'b1??? : begin
              out_y = 3'd3 ;
            end
    4'b01?? : begin
              out_y  = 3'd2 ;
            end
    4'b001? : begin
              out_y = 3'd1 ;
            end
    4'b0001 : begin
              out_y = 3'd0 ;
            end
    4'b0000 : begin
              out_y = 3'd4 ;
            end
  endcase
end
endmodule
```

Parameter is not used, because this logic can not work correctly for bit sizes other than 4.

Run this file on your PC to see yourself if the module works correctly.

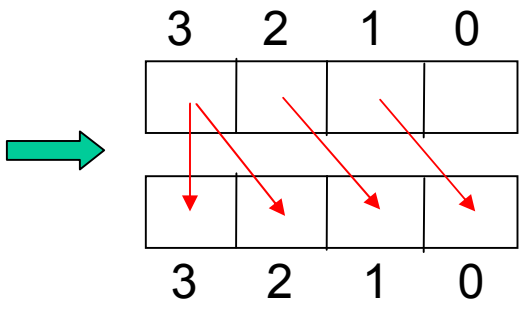
file name : prity_encdr_alwy.v

Ex 3-8. Sign bit and oncatenating operator: Write a module which shift a 4-bit input signal sig_a to the right and output the 4-bit result out_y. The sift count is given by a 2-bit input signal sft_cnt.
Use Verilog1995, do not use Verilog2001.
Although Verilog1995 can not handle signed signal, the module must extend the MSB bit in shifting to the right, for example, if sig_a's MSB is on, 1 must be filled to the bit positions where they are shifted out.
Do not use Verilog shift operators, >> and >>>.



A combinational logic can be described by using always construct .
The sift operation can be written by using a Verilog operator { } .

{ sig_a[3], sig_a[3:1] }



The operation may be different depending on the shift count sft_cnt, therefore case statement can be applied for different shift count. Such logic can be described by using case statement as shown below.

Case statement can be written only in always construct, initial construct, function, and task.
We must write the case statement in always construct so that whenever sft_cnt or sig_a changes, out_y is to be updated.

```
case ( sft_cnt )
  2'd0 : begin  out_y = sig_a ;  end
  2'd1 : begin  out_y = { sig_a[3], sig_a[3:1] } ; end
  2'd2 : begin  out_y = { {2{sig_a[3]} }, sig_a[3:2] } ; end
  2'd3 : begin  out_y = {4{sig_a[3]} } ; end
 default : begin  out_y = 4'bxxxx ; end
endcase
```

However, code lines in case item 2'd0, 2'd1, 2'd2, and 2'd3 are different in many parts.

Consistent style, beauty of the code

```
case ( sft_cnt )
  2'd0 : begin  out_y = sig_a ;                                end
  2'd1 : begin  out_y = { sig_a[3], sig_a[3:1] } ;             end
  2'd2 : begin  out_y = { {2{sig_a[3]} }, sig_a[3:2] } ; end
  2'd3 : begin  out_y = {4{sig_a[3]} } ;                       end
 default : begin  out_y = 4'bxxxx ; end
endcase
```

We can not apply systematic checking if every code line has different structures.

The following code lines have the same structure and are almost identical, except the numbers in Italic.
It is very easy to check them. We can say that if one line is correct, then other lines are correct just by checking Italic part only.

```
2'd0 : begin  out_y = { {0{sig_a[3]} }, sig_a[3:0] } ; end
2'd1 : begin  out_y = { {1{sig_a[3]} }, sig_a[3:1] } ; end
2'd2 : begin  out_y = { {2{sig_a[3]} }, sig_a[3:2] } ; end
2'd3 : begin  out_y = { {3{sig_a[3]} }, sig_a[3:3] } ; end
```

Every code line has the same structures.

Consistent code and checking

```
case ( sft_cnt )
  2'd0 : begin  out_y = sig_a ;                          end
  2'd1 : begin  out_y = { sig_a[3], sig_a[3:1] } ;       end
  2'd2 : begin  out_y = { {2{sig_a[3]} }, sig_a[3:2] } ; end
  2'd3 : begin  out_y = {4{sig_a[3]} } ;                 end
endcase
```

Is this OK?

Is this OK?

Is this OK?

Is this OK?

Every code line has different checking viewpoints.

```
2'd0 : begin  out_y = { {0{sig_a[3]} }, sig_a[3:0] } ; end
2'd1 : begin  out_y = { {1{sig_a[3]} }, sig_a[3:1] } ; end
2'd2 : begin  out_y = { {2{sig_a[3]} }, sig_a[3:2] } ; end
2'd3 : begin  out_y = { {3{sig_a[3]} }, sig_a[3:3] } ; end
```

Is this OK?

Coding example.

```
module arith_r_sft ( sig_a, sft_cnt, out_y ) ;
input [3:0] sig_a ;
input [1:0] sft_cnt ;
output [3:0] out_y ;
wire [3:0] sig_a ;
wire [1:0] sft_cnt ;
reg [3:0] out_y ; // non-FF

always @ ( sig_a or sft_cnt ) begin
case ( sft_cnt )
 2'd0 : begin  out_y = { {0{sig_a[3]} }, sig_a[3:0] } ; end
 2'd1 : begin  out_y = { {1{sig_a[3]} }, sig_a[3:1] } ; end
 2'd2 : begin  out_y = { {2{sig_a[3]} }, sig_a[3:2] } ; end
 2'd3 : begin  out_y = { {3{sig_a[3]} }, sig_a[3:3] } ; end
 default : begin  out_y = 4'bxxxx ; end
endcase
end
endmodule
```

file name: arith_r_sft.v

Parameter is not used, because this logic can not work correctly for bit sizes other than 4.

This style is much easier to check the code from the view point of consistency.

This is for debug.

Write a test bench yourself and check the result.

Another coding style

A combinational logic can be written by using function.

```
reg [3:0] out_y ; // non-FF

always @ ( sig_a or sft_cnt ) begin
case ( sft_cnt )
  2'd0 : begin
     out_y = { {0{sig_a[3]} }, sig_a[3:0] }  ;
   end
  2'd1 : begin
     out_y = { {1{sig_a[3]} }, sig_a[3:1] } ;
    end
  2'd2 : begin
    out_y = { {2{sig_a[3]} }, sig_a[3:2] } ;
   end
  2'd3 : begin
    out_y = { {3{sig_a[3]} }, sig_a[3:3] } ;
   end
 default : begin
     out_y = 4'bxxxx ;
 end
endcase
end
```

```
wire [3:0] out_y ; // non-FF

assign out_y = sft_r_f(sig_a, sft_cnt) ;

function [3:0] sft_r_f ;
input [3:0] aa ;
input [1:0] cnt ;
begin
case (cnt )
  2'd0 : begin
            sft_r_f = { {0{aa[3]} }, aa[3:0] }  ;
     end
  2'd1 : begin
            sft_r_f = { {1{aa[3]} }, aa[3:1] }  ;
     end
  2'd2 : begin
            sft_r_f = { {2{aa[3]} }, aa[3:2] }  ;
    end
  2'd3 : begin
            sft_r_f = { {3{aa[3]} }, aa[3:3] }  ;
     end
 default : begin
            sft_r_f = 4'bxxxx ;
    end
endcase
end
endfunction
```

if else, case, and conditional statement

```
case ( a )
  b : begin
        yy = 4'b0101 :
    end
  default : begin
        yy = 4'b1100 ;
    end
endcase
```

```
if ( a == b ) begin
   yy = 4'b0101 :
end
else begin
   yy = 4'b1100 ;
end
```

```
yy = ( a == b ) ?  4'b0101
                 : 4'b1100 ;
```

case statement, if statement, and conditional statement can work in similar way.

Those shown on this page are almost the same but in detail they are different.

If a is 2'b1x and b is 2'b1x, the result of the case statement is "yy=4'b0101", the result of if statement is "yy=4'b1100", and the result of conditional statement is "yy=4'bx10x". See online text material for details.

if else, case, and conditional statement

```
case ( a )
  b : begin
        assign yy = 4'b0101 :
    end
  default : begin
        assign yy = 4'b1100 ;
    end
endcase
```

NG

```
if ( a == b ) begin
  assign yy = 4'b0101 :
end
else begin
  assign yy = 4'b1100 ;
end
```

NG

```
assign yy = ( a == b ) ?  4'b0101
                        : 4'b1100 ;
```

OK

case statement and if statement can be used only in procedure, therefore we can not use continuous assign in case statement nor in if statement.

Only procedural assignment is allowed in structured procedures.

However we can use continuous assign for conditional statement. It can also be used in structured procedures.

Ex 3-9. Missing case item and latching: Write a module equivalent to the truth table shown below using an always construct. An input port is 2-bit in_a and an output port is 3-bit out_y.

in_a[1]	in_a[0]	out_y
0	0	3'b001
0	1	3'b110
1	0	3'b101

Let's see what happens with 2-bit case expression which can take 4 possible values if only 3 case items are given.
In this exercise we will study about case statement and default.

Create a file case_dflt.v by using case statement on the right. Type a test bench into the same file.

```
case ( in_a )
  2'b00 : begin
      out_y = 3'b001 ;
    end
  2'b01 : begin
      out_y = 3'b110 ;
    end
  2'b10 : begin
      out_y = 3'b101 ;
    end
endcase
```

```
module case_deflt ( in_a, out_y ) ;
input [1:0]   in_a ;
output [2:0] out_y ;
//
wire [1:0]   in_a ;
reg [2:0] out_y ; // non-FF
//
always @ ( in_a ) begin
case ( in_a )
  2'b00 : begin
      out_y = 3'b001 ;
    end
  2'b01 : begin
      out_y = 3'b110 ;
    end
  2'b10 : begin
      out_y = 3'b101 ;
    end
endcase
end
endmodule
```

```
module test_case_deflt;
reg [1:0] in_a;
wire [2:0] out_y ;

case_deflt case_deflt_01 ( .in_a(in_a), .out_y(out_y) ) ;

initial begin
      in_a = 2'b00 ;
#10 in_a = 2'b01 ;
#10 in_a = 2'b10 ;
#10 in_a = 2'b11 ;
#10 $finish ;
end

initial begin
  $monitor( "in_a=%b. out_y=%b", in_a, out_y ) ;
end
endmodule
```

Run this on your PC.

file name : case_deflt.v

```
initial begin
      in_a = 2'b00 ;
#10 in_a = 2'b01 ;
#10 in_a = 2'b10 ;
#10 in_a = 2'b11 ;
#10 $finish ;
end
```

A sample result

```
in_a=00. out_y=001
in_a=01. out_y=110
in_a=10. out_y=101
in_a=11. out_y=101
```

The result must looks like the one shown above.

Note that for unexpected input in_a=11, we got the result out_y=101 which is the same to the previous line, in_a=10 and out_y=101.

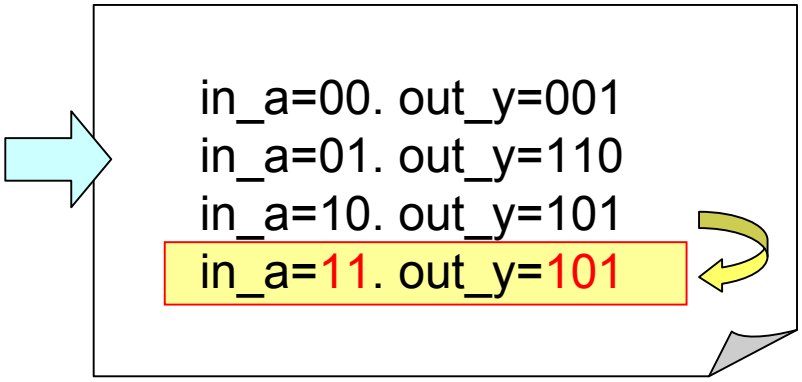
```
initial begin
      in_a = 2'b00 ;
#10 in_a = 2'b01 ;
#10 in_a = 2'b10 ;
#10 in_a = 2'b11 ;
#10 $finish ;
end
```

```
in_a=00. out_y=001
in_a=01. out_y=110
in_a=10. out_y=101
in_a=11. out_y=101
```

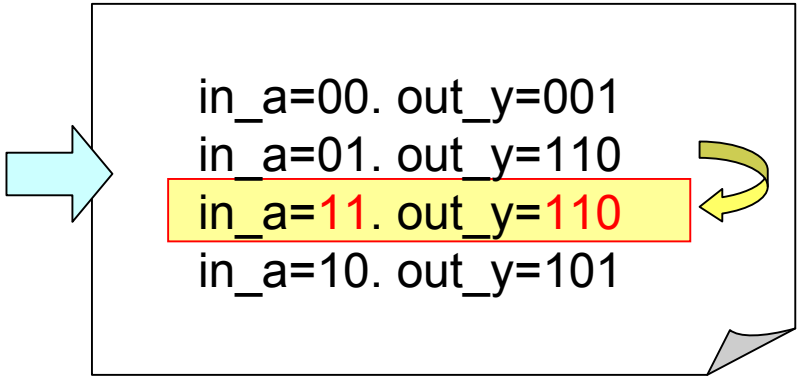
```
initial begin
      in_a = 2'b00 ;
#10 in_a = 2'b01 ;
#10 in_a = 2'b11 ;
#10 in_a = 2'b10 ;
#10 $finish ;
end
```

Now, change the initial construct as shown on the left and run the modified file.

```
initial begin
      in_a = 2'b00 ;
#10 in_a = 2'b01 ;
#10 in_a = 2'b10 ;
#10 in_a = 2'b11 ;
#10 $finish ;
end
```




```
initial begin
      in_a = 2'b00 ;
#10 in_a = 2'b01 ;
#10 in_a = 2'b11 ;
#10 in_a = 2'b10 ;
#10 $finish ;
end
```



Note that, this time, for unexpected inout in_a=11, we got out_y=110 which is the same to the previous line, in_a=01 and out_y=110.

in_a=00. out_y=001
in_a=01. out_y=110
in_a=10. out_y=101
in_a=11. out_y=101

```
case ( in_a )
  2'b00 : begin
      out_y = 3'b001 ;
    end
  2'b01 : begin
      out_y = 3'b110 ;
    end
  2'b10 : begin
      out_y = 3'b101 ;
    end
endcase
```

This phenomena is called latching.

The case statement dose not say anything about the case where in_a is 2'b11. A simulator tries to output some value even for such case, and output previous value of out_y. Because register type signal is expected to hold the value assigned until an new assignment is done, it is natural to output the previous value.

simulation : latching

synthesis : creating a latch

in_a=00. out_y=001
in_a=01. out_y=110
in_a=10. out_y=101
in_a=11. out_y=101

```
case ( in_a )
  2'b00 : begin
      out_y = 3'b001 ;
    end
  2'b01 : begin
      out_y = 3'b110 ;
    end
  2'b10 : begin
      out_y = 3'b101 ;
    end
endcase
```

A synthesis tool will create a memory element called latch for such cases.

However, a latch is difficult to use in large scale design. It is better to avoid using a latch.

Moreover, if a latch is created when you wrote an always construct to define combinational logic, it will cause some malfunction of your logic.

We need some countermeasure to avoid latches.

simulation : latching

synthesis : creating a latch

To avoid latches, default is used.

A sample code on the left assigns 0 to out_y when in_a is unexpected 2’b11.

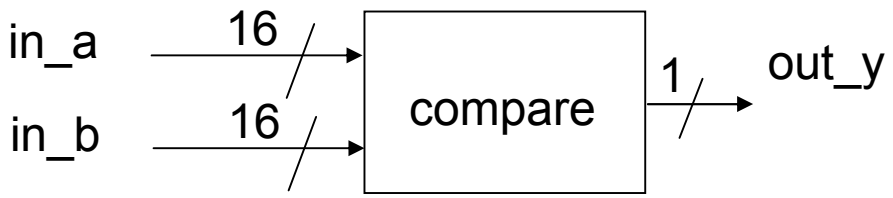
Another sample code on the right assigns unknown value to out_y when in_a is unexpected 2’b11.

Either code will do. The default on the right will be neglected in synthesis because it assigns unknown value. The default on the left will not be neglected.

```
case ( in_a )
  2'b00 : begin
      out_y = 3'b001 ;
    end
  2'b01 : begin
      out_y = 3'b110 ;
    end
  2'b10 : begin
      out_y = 3'b101 ;
    end
  default : begin
      out_y = 3’b000 ;
    end
endcase
```

```
case ( in_a )
  2'b00 : begin
      out_y = 3'b001 ;
    end
  2'b01 : begin
      out_y = 3'b110 ;
    end
  2'b10 : begin
      out_y = 3'b101 ;
    end
  default : begin
      out_y = 3’bxxx ;
    end
endcase
```

Ex 3-10. Compare signed data in Verilog1995: Write a module using combinational logic to compare two 16-bit signals, in_a and in_b, and outputs 1 if in_a is smaller than in_b, otherwise outputs 0.
Both in_a and in_b can be negative, and use Verilog1995.



In this exercise we will study how to handle negative values in Verilog1995. In Verilog1995, all the signals are treated as unsigned, that is, positive. Therefore, when handling negative value, a programmer has to take care of negative values.

Because Verilog1995 can not handle negative values, take special care for the cases where sign-bit is 1.

RTL programming rules

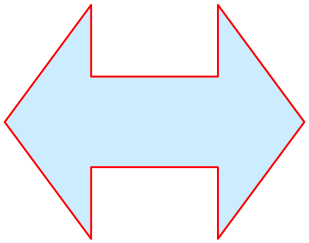
(1) Based numbers are unsigned, unless they are declared with s character in base numbers.
(2) Use signed key word for signed variables and nets.

8'hF9

8-bit + 249

8'shF9

8-bit - 7



```
wire [7:0] sig_y ;
assign sig_y = -7 ;
```

8-bit + 249

Verilog1995

```
wire signed [7:0] sig_y ;
assign sig_y = -7 ;
```

8-bit - 7

Verilog2001

8'b1000_0000 → 128

8'sb1000_0000 → -128

The MSB is set 1 and "s" declared in base number, therefore it means minus value, -128.

```
wire signed [7:0] sig_y ;
assign sig_y = 8'b1000_0000 ;
assign flg = ( sig_y < -10 )?  1 : 0 ;
```

→ sig_y = -128
flg = 1

sig_y is declared as signed, and the MSB bit is given the value 1 by the assign statement.Therefore, sig_y is -128.

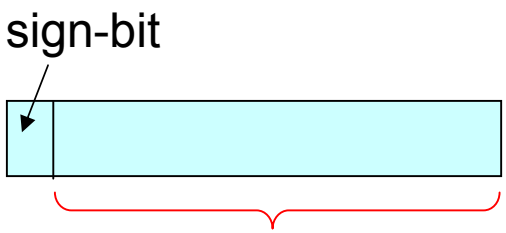
```
wire [7:0] sig_w ;
assign sig_w = 8'sb0000_1010 ;
assign flg = ( sig_w < -10 )?  1 : 0 ;
```

→ sig_w = 10
flg = 1

sig_w is not declared as a signed variable, therefore the comparison is done assuming both operands are positive.

-10 is a large positive value if it is treated as a positive value. We have to declare sig_w signed so that the result of the comparison, 10 <= -10, results in false.

Let's think how to handle sign bit.



Larger the value of this part, the larger positive value when sign-bit is 0 or the larger negative value when sign-bit is 1.

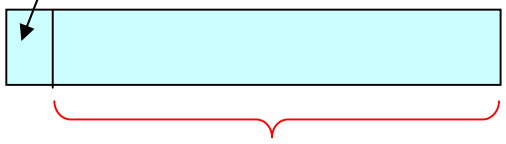
1101	>	1001	0011	>	1110
-3		-7	3		-2
0101	>	0001	0111	>	1001
5		1	7		-7

Depending on the value of the MSB, we have to apply different way of comparing.

Especially when the two operand have different MSB value such as comparing 0011(3) and 1110(-2).

```
case ( { in_a[SIG_BW-1], in_b[SIG_BW-1] } )
  2'b11, 2'b00 : begin
      out_y = ( in_a[SIG_BW-2:0] < in_b[SIG_BW-2:0] )?
                                  1'b1 : 1'b0 ;
    end
  2'b10 : begin  // in_a negative, in_b positive
      out_y = 1'b1 ;
    end
  2'b01 : begin // in_a positive, in_b negative
       out_y = 1'b0 ;
    end
```

sign-bit



Larger the value of this part, the larger positive value when sign-bit is 0 or the larger negative value when sign-bit is 1.

1101	>	1001	0011	>	1110
-3		-7	3		-2
0101	>	0001	0111	>	1001
5		1	7		-7

Now, create compare logic and a test bench into one file named smaller_chk.v.

```
module smaller_chk ( in_a, in_b, out_y ) ;
parameter SIG_BW = 16 ; // must be larger than 1
input [SIG_BW-1:0] in_a, in_b ;
output out_y ;
//
wire [SIG_BW-1:0] in_a, in_b ;
reg out_y ;
//
always @ ( in_a or in_b ) begin
 case ( { in_a[SIG_BW-1], in_b[SIG_BW-1] } )
      2'b11, 2'b00 : begin  // both negative, or both positive
           out_y = ( in_a[SIG_BW-2:0] < in_b[SIG_BW-2:0] )? 1'b1 : 1'b0 ;
        end
      2‘b10 : begin  // in_a negative, in_b positive
           out_y = 1'b1 ;
        end
      2'b01 : begin // in_a positive, in_b negative
           out_y = 1'b0 ;
        end
      default : begin // for debug. This will not be reached in normal case
           out_y = 1'bx ;
        end
   endcase
end
endmodule
```

```
module test_smaller_chk ;
parameter SIG_BW = 16 ;
reg [SIG_BW-1:0] in_a, in_b ;
wire out_y ;
//
smaller_chk    smaller_chk_01 ( .in_a(in_a), .in_b(in_b), .out_y(out_y) ) ;
//
initial begin
      in_a = 16'h5101 ;
#10 in_a = 16'h8F45 ;
#10 in_a = 16'h6310 ;
#10 in_a = 16'h7682 ;
#10 in_a = 16'hE503 ;
#10 $finish ;
end
initial begin
      in_b = 16'h5131 ;
#10 in_b = 16'h7023 ;
#10 in_b = 16'hE237 ;
#10 in_b = 16'h7335 ;
#10 in_b = 16'hE99F ;
end
initial begin
  $monitor( " in_a=%b. ¥n in_b=%b, out_y=%b", in_a, in_b, out_y ) ;
end
endmodule
```

file name: smaller_chk.v

Run this file on your PC to see the result.

The result shall look like the one shown below.

```
in_a=0101000100000001.
in_b=0101000100110001, out_y=1
in_a=1000111101000101.
in_b=0111000000100011, out_y=1
in_a=0110001100010000.
in_b=1110001000110111, out_y=0
in_a=0111011010000010.
in_b=0111001100110101, out_y=0
in_a=1110010100000011.
in_b=1110100110011111, out_y=1
```

Now, let's update the module by using Verilog2001.

In Verilog2001, we can handle negative values by using signed key word.

relational operators:	>, >=, <, <=	These are applicable for negative values.
Logical equality operator:	==	
Arithmetic shift operator:	>>>, <<<	

The code becomes as simple as shown in the next page, because we do not have to take care of the sign bits. Verilog2001 can take care of them.

The code must be very simple as shown on this slide.

```
module smaller_chk ( in_a, in_b, out_y ) ;
parameter SIG_BW = 16 ;
input signed[SIG_BW-1:0] in_a, in_b ;
output out_y ;
//
wire signed[SIG_BW-1:0] in_a, in_b ;
reg out_y ; // non-FF
//
always @ ( in_a or in_b ) begin
  out_y = ( in_a < in_b )? 1'b1 : 1'b0 ;
end
endmodule
```

test bench part not shown here

file name : smaller_chk_sign.v

```
wire out_y ;
assign out_y = ( in_a < in_b )? 1'b1 : 1'b0 ;
```

It is so simple that we do not have to use always construct.
The module can be modified by using continuous assign.

```
module smaller_chk ( in_a, in_b, out_y ) ;
parameter SIG_BW = 16 ;
input signed[SIG_BW-1:0] in_a, in_b ;
output out_y ;
//
wire signed[SIG_BW-1:0] in_a, in_b ;
reg out_y ;
//
always @ ( in_a or in_b ) begin
 out_y = ( in_a < in_b )? 1'b1 : 1'b0 ;
end
endmodule
```

test bench part not shown here

file name : smaller_chk_sign_sign.v

Run this file on your PC to see the result.

RTL programming rules

(1) Part select of vector is always unsigned even if the vector is declared signed and part select specifies entire bits.

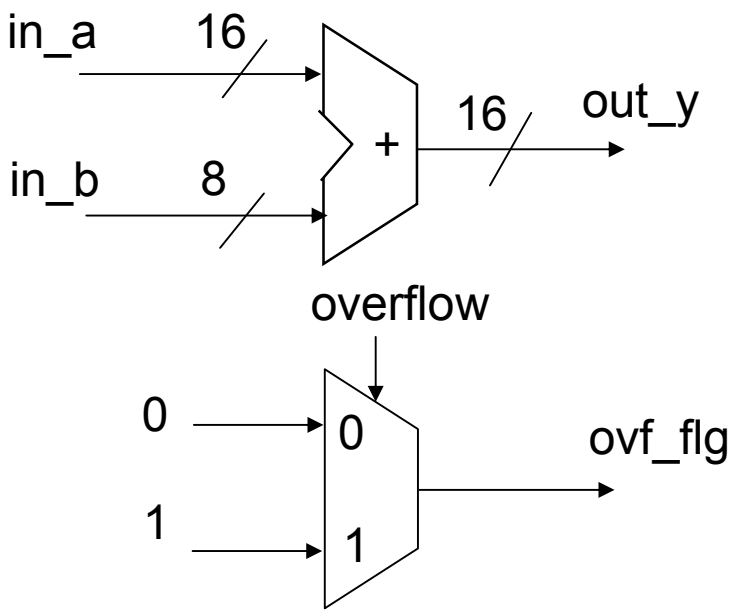
Now, modify the code as shown below to see how part selection of vector works. Run the code below on your PC and check the result yourself.

```
module smaller_chk ( in_a, in_b, out_y ) ;
parameter SIG_BW = 16 ;
input signed[SIG_BW-1:0] in_a, in_b ;
output out_y ;
//
wire signed[SIG_BW-1:0] in_a, in_b ;
reg out_y ;
//
always @ ( in_a or in_b ) begin
 out_y = ( in_a [SIG_BW-1:0] < in_b [SIG_BW-1:0] )? 1'b1 : 1'b0 ;
end
endmodule

      test bench part not shown here
```

This code will not get a correct result because part select is used.

file name : smaller_chk_sign_bgy.v

Ex 3-11. Arithmetic operation in Verilog1995: Write a module which adds 16-bit input in_a and 8-bit input in_b and places 16-bit result on out_y. in_a and in_b may be negative.
The logic must output ovf_flg to indicate overflow. Use Verilog1995.



Because today's synthesis tools are good in efficiency, use Verilog arithmetic operator instead of hand-craft arithmetic logic block.

If we can use Verilog2001, it shall be as simple as below left.

Verilog2001

```
wire signed [15:0] in_a ;
wire signed [7:0] in_b ;
wire signed [15:0] out_y ;
//
assign out_y = in_a + in_b ;
```

sign bit of in_b is extended automatically

Verilog1995

```
wire [15:0] in_a ;
wire [7:0] in_b ;
wire [15:0] out_y ;
//
assign out_y = in_a + in_b  ;
```

sign bit not extended.

How to extend the sign bit??

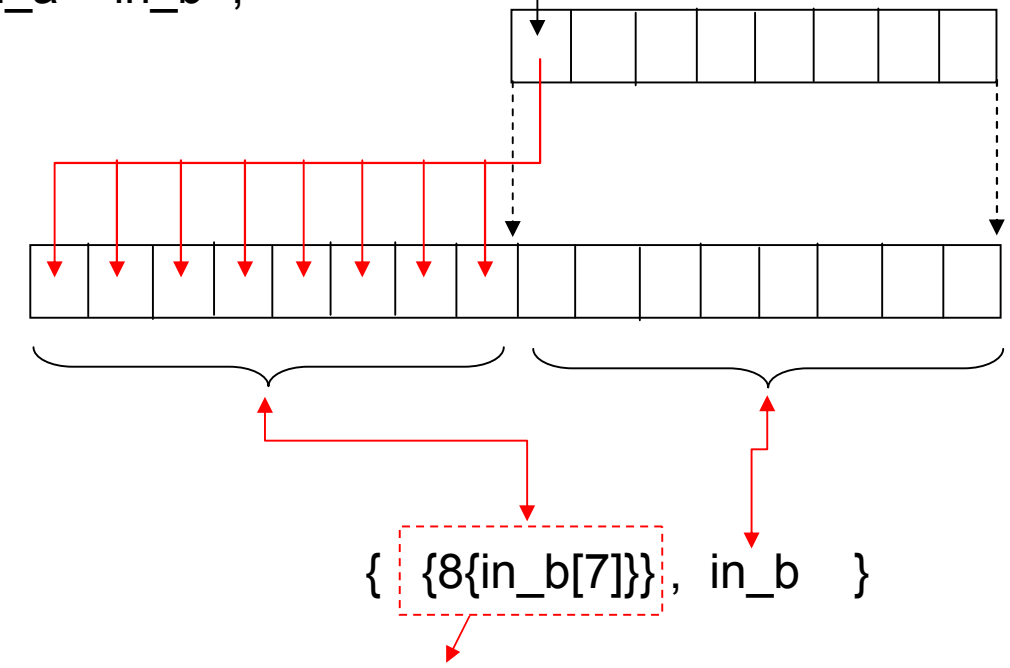
Verilog1995

```
wire [15:0] in_a ;
wire [7:0] in_b ;
wire [15:0] out_y ;
//
assign out_y = in_a + in_b  ;
```

sign bit not extended.

bit select of a vector

in_b[7]



{ {8{in_b[7]}}, in_b }

{ 8 { yyy } } means {yyy} repeated 8 times.

in_a	in_b	out_y	ovf_flg
positive	positive	positive	0
positive	positive	negative	1
positive	negative	positive	0
positive	negative	negative	0
negative	positive	positive	0
negative	positive	negative	0
negative	negative	positive	1
negative	negative	negative	0

ovr_flg must be set base on the truth table shown in this slide.

If you agree to this table, go to the next page.

in_a	in_b	out_y	ovf_flg
positive	positive	positive	0
positive	positive	negative	1
positive	negative	positive	0
positive	negative	negative	0
negative	positive	positive	0
negative	positive	negative	0
negative	negative	positive	1
negative	negative	negative	0

```
assign ovf_flg = ( (~in_a[15]) & (~in_b[7]) & out_y[15] ) |
                 ( in_a[15] & in_b[7] & (~out_y[15]) )?    1 : 0 ;
```

Now, create a file named arith_add_95.v which can add 16-bit input in_a and 8-bit input in_b and check overflow. Type a test bench in the same file. Introduce parameter to make bit width adjustable.

revised in V2r04

```
module arith_add ( in_a, in_b, out_y, ovf_flg ) ;
parameter LNG_BW = 16 ;
parameter SHRT_BW = 8 ; // must be smaller or equal to LNG_BW
input [LNG_BW-1:0] in_a ;
input [SHRT_BW-1:0]in_b ;
output[LNG_BW-1:0] out_y ;
output ovf_flg ;  // over flow indicator
//
wire [LNG_BW-1:0] in_a ;
wire [SHRT_BW-1:0]in_b ;
wire [LNG_BW-1:0] out_y ;
wire ovf_flg ;
//
assign out_y = in_a + { {(LNG_BW-SHRT_BW){in_b[SHRT_BW-1]}}, in_b } ;
assign ovf_flg = ( (~in_a[LNG_BW-1]) & (~in_b[SHRT_BW-1]) &
                   out_y[LNG_BW-1] ) | ( in_a[LNG_BW-1] &
                   in_b[SHRT_BW-1] & (~out_y[LNG_BW-1]) )? 1'b1 : 1'b0 ;
//
endmodule

module test_arith_add ;
parameter LNG_BW = 16 ;
parameter SHRT_BW = 8 ;
reg [LNG_BW-1:0] in_a ;
reg [SHRT_BW-1:0]in_b ;
wire [LNG_BW-1:0] out_y ;
```

```
wire ovf_flg ;
//
arith_add    arith_add_01 ( .in_a(in_a), .in_b(in_b),
                            .out_y(out_y), .ovf_flg(ovf_flg) ) ;
initial begin
      in_a = 1096 ;
#10 in_a = 32765 ;
#10 in_a = -32760 ;
#10 in_a = -32760 ;
#10 in_a = -2970 ;
#10 in_a = -32700 ;
#10 $finish ;
end
initial begin
      in_b = 20 ;
#10 in_b = 7 ;
#10 in_b = -2 ;
#10 in_b = -200 ;
#10 in_b = -180 ;
#10 in_b = -120 ;
end
initial begin
  $monitor( "in_a=   %b,¥n in_b=        %b,¥n out_y= %b,    ovf=%b¥n",
                in_a, in_b, out_y, ovf_flg ) ;
end
endmodule
```

file name : arith_add_95.v

Run this file on your PC and check the result.

A sample result

```
in_a=    0000010001001000,
 in_b=              00010100,
 out_y= 0000010001011100,    ovf=0

in_a=    0111111111111101,
 in_b=              00000111,
 out_y= 1000000000000100,    ovf=1

in_a=    1000000000001000,
 in_b=              11111110,
 out_y= 1000000000000110,    ovf=0

in_a=    1000000000001000,
 in_b=              00111000,
 out_y= 1000000001000000,    ovf=0

in_a=    1111010001100110,
 in_b=              01001100,
 out_y= 1111010010110010,    ovf=0

in_a=    1000000001000100,
 in_b=              10001000,
 out_y= 0111111111001100,    ovf=1

Halted at location **arith_add_95.v(39
```

Ex 3-12. Test bench and coverage: Copy and paste the following modules and run them to check the 8-bit unary EOR logic module can place its 1-bit output out_y correctly depending on the number of on-bits in its 8-bit input in_a.

```
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<=3; k=k+1) begin
  cnt=cnt+in_a[k]+in_a[k+3] ;
 end
 out_y=( cnt[0] )? 1 : 0 ;
end
endmodule
```

file name: u_eor.v

Run test_u_eor.v on you PC to see the result.

```
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

A simulation result	bit count
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

For all possible bit count, 0 to 8, the result is OK.
And the test data cover all the paths in RTL code lines.

The target module, u_eor, is correct, because the above simulation result is correct.

Do you agree with this conclusion??

If you agree, then try the test data aa=8'b0000_1001 to find a bug.

Do not think
"my code is correct, because the simulation result is OK."

In many cases, "simulation result OK" just means your test data or test bench is poorly designed.

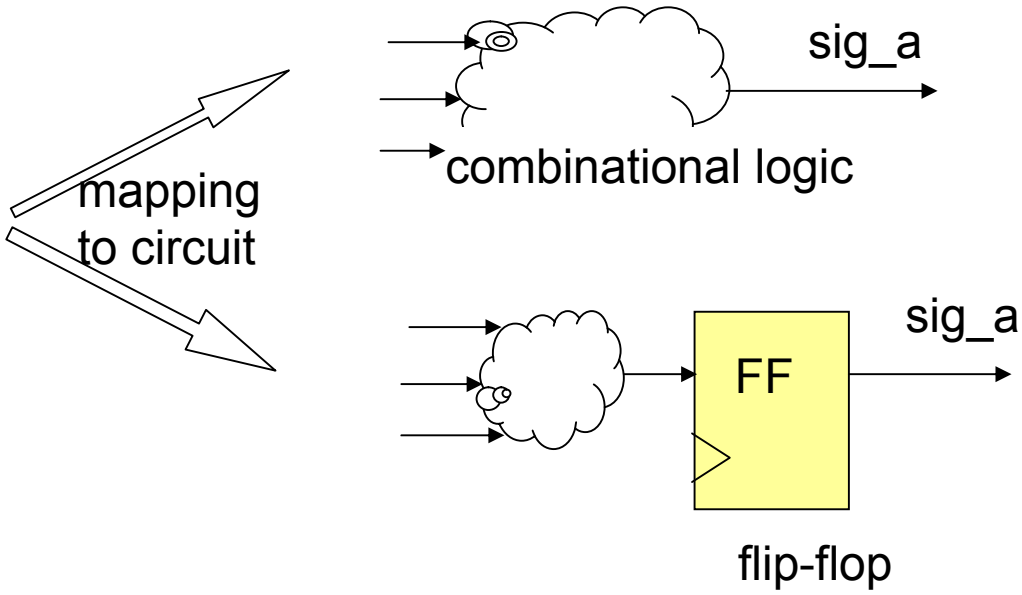
Chapter 4. Flip-flops and sequential logic

Flip-flop is a most important logic element in RTL design.

It can be defined by using an always construct.

Registers (variables) in Verilog RTL can hold values in simulator, but they can not hold any value in actual silicon unless they are mapped into flip-flops by a synthesis tool.

```
reg sig_a ;

begin
   :
   :
  sig_a = --------- ;
   :
   :
end
```



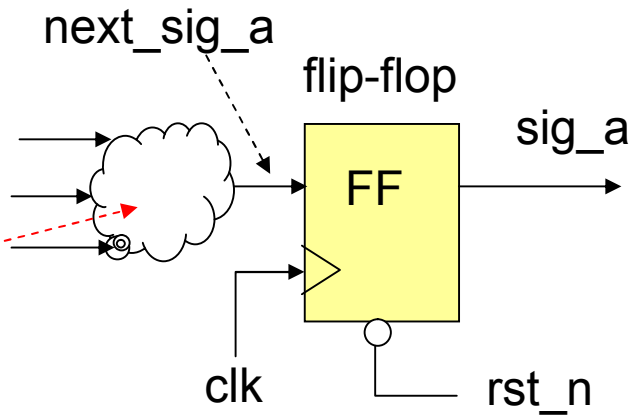
RTL programming rules

always construct can generate a flip-flop if
(1) “posedge clk" is written in a sensitivity list. No change value event must appear in the sensitivity list. (posdege rst or negedge rst_n is allowed.)

```
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
    sig_a <= 0 ;
  end
  else begin
    sig_a <= next_sig_a ;
  end
end

assign next_sig_a = ,,,,,,,,, ;
```

nonblocking procedural assignment

Do not use **other** styles until you fully understand what each line means.

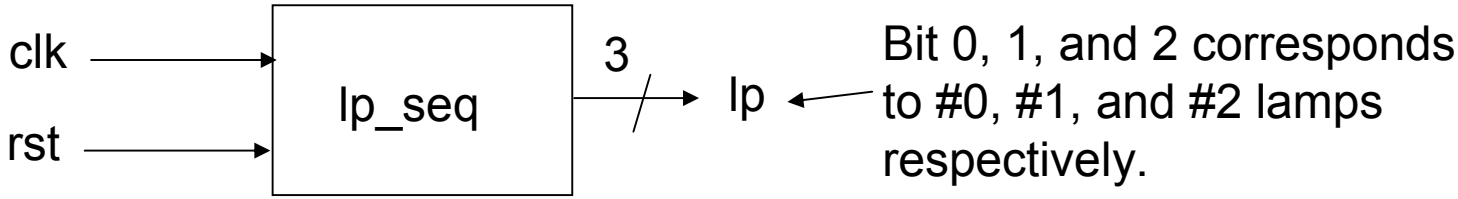


You must write “posedge clk or negedge rst_n” exactly the same order for “posedge clk” and “negedge rst_n”. If your write “negedge rst_n or posedge clk”, a synthesis tool can not create a correct flip-flop.

The tool thinks that the first argument is a clock signal.

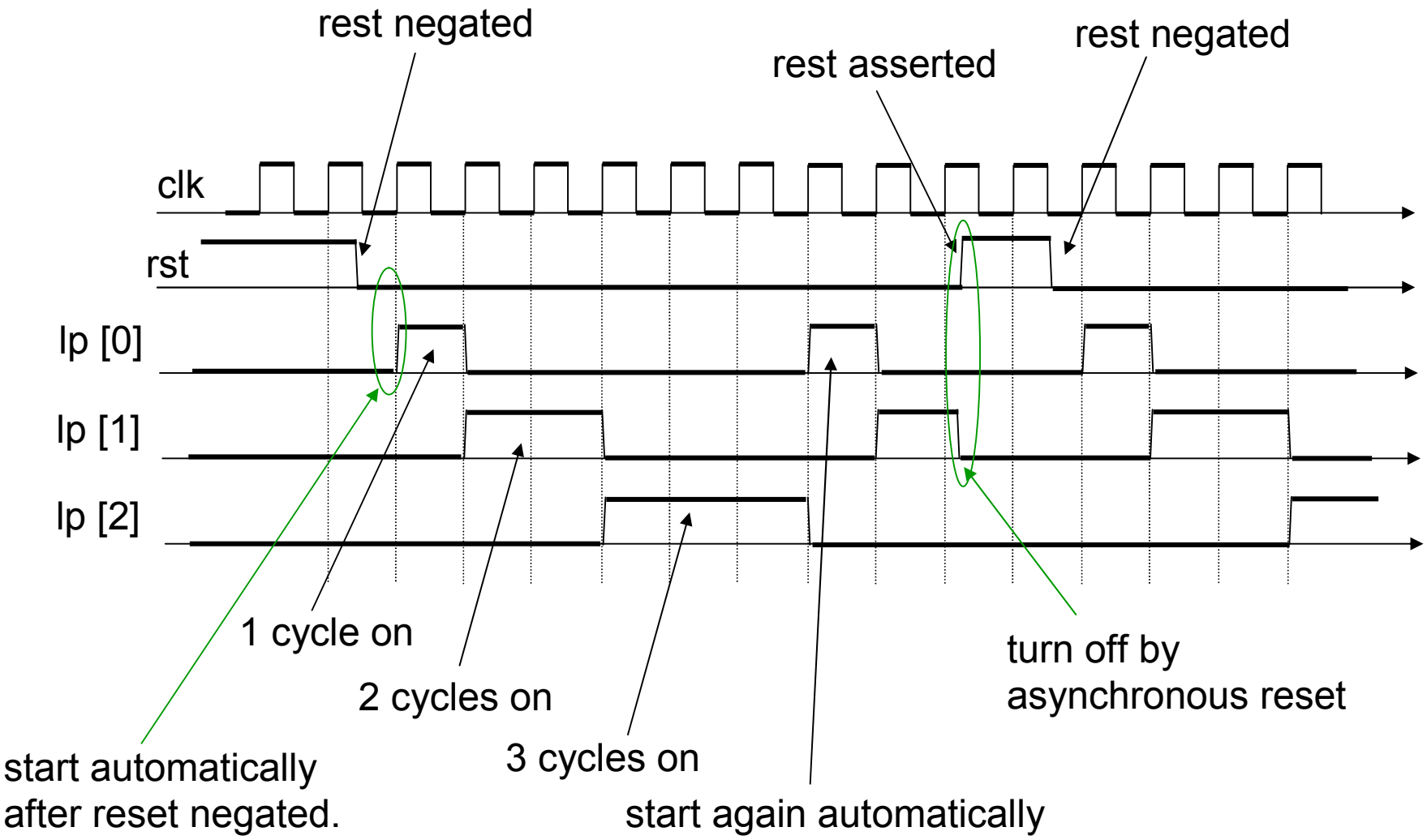
Ex 4-1. Sequential lamps: Write a module lp_seq of which specification is given below.

There are 3 lamps, #0, #1, and #2 lamps.
#0 lamp illuminates one clock cycle,
#1 lamp illuminates two clock cycles, and
#2 lamp illuminates three clock cycles.
They illuminate in turn, that is, first #0 lamp and next #1 lamp and lastly#2 lamp.
While one lamp is on, other lamps are off.
After #2 lamp illuminate 3 cycles, #0 lamp illuminates again and keeps the same on off operation forever unless reset asserted.
When reset (asynchronous active high) asserted, all lamps become off.
At the first rise edge of the clock after reset negated, #0 lamp must turn on.
Each lamp's turn on and off occur at clock rise time.



This is a typical sequential logic, we need memory elements to implement this features into silicon. The first thing we have to do is to determine how many and what kind of memory elements are needed.

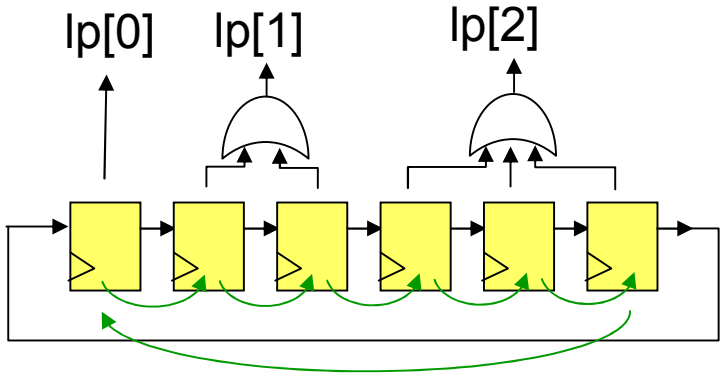
The system must behave as shown below.



lamp itself : Assign one flip-flop for one lamp.

→ This idea can be extended to implement the specified logic as below.

Assign one flip-flop to lp [0], two flip-flops to lp [1], and three flip-flops to lp[2] as shown below. Then shifting 1 bit through this shift register can realize the logic needed.

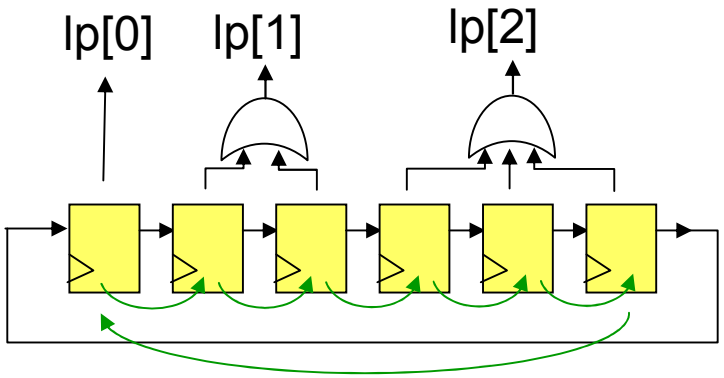


This idea is same to assigning memory element for state variable. “On” position of flip-flops represents the state.

This shift registers can be modeled by the code on the right.

```
always @ ( posedge clk ) begin
   lp_reg[5:0] <=
            { lp_reg[4:0] , lp_reg[5] } ;
end
```

1 bit shift rotate operation




```
always @ ( posedge clk ) begin
   lp_reg[5:0] <=
            { lp_reg[4:0] , lp_reg[5] } ;
end
```

Next think about reset ;

- On reset, these flip-flop must be turned off.
- When reset negated, we must turn on lp_reg[0].

This can be done as below.

```
always @ ( posedge clk or posedge rst ) begin
  if ( rst_n ) begin
     lp_reg[5:0] <= 6'b00_0000 ;
  end
   else begin
     lp_reg[5:0] <= 6'b00_0001 ;
  end
end
```

```
always @ ( posedge clk ) begin
   lp_reg[5:0] <=
          { lp_reg[4:0] , lp_reg[5] } ;
end
```

```
always @ ( posedge clk or posedge rst ) begin
  if ( rst_n ) begin
       lp_reg[5:0] <= 6'b00_0000 ;
  end
   else begin
       lp_reg[5:0] <= 6'b00_0001 ;
  end
end
```

```
always @ ( posedge clk or posedge rst ) begin
  if ( rst_n ) begin
       lp_reg[5:0] <= 6'b00_0000 ;
  end
   else begin
       lp_reg[5:0] <= next_lp_reg[5:0] ;
  end
end
assign next_lp_reg[5:0] =
       ( lp_reg ) ? { lp_reg[4:0] , lp_reg[5] } : 6'b00_0001 ;
```

```
module lp_seq ( clk, rst, lp ) ;
input clk, rst ;
output [2:0] lp ;
wire clk, rst ;
wire [2:0] lp ;

reg [5:0] lp_reg ;   // FF for state
wire [5:0] next_lp_reg ;  // combinational logic

assign lp[0] = lp_reg[0] ;
assign lp[1] = lp_reg[1] | lp_reg[2]  ;
assign lp[2] = lp_reg[3] | lp_reg[4] | lp_reg[5]  ;

always @ ( posedge clk  or posedge rst ) begin
  if ( rst ) begin
     lp_reg <= 6'b00_0000 ;
  end
  else begin
    lp_reg <= next_lp_reg ;
  end
end

assign next_lp_reg =
      ( lp_reg ) ? { lp_reg[4:0] , lp_reg[5] } : 6'b00_0001 ;

endmodule
```

file name: lp_seq.v

This is a module we wanted. It is synthesizable, timing accurate and hardware resource accurate.

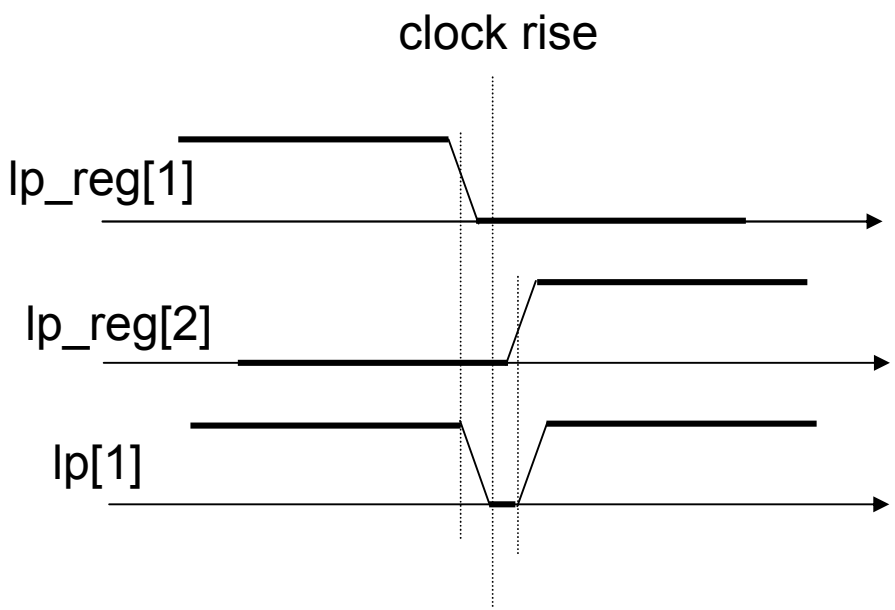
Parameter is not used because this logic works correctly only when number of lamps are 3.

Run this code by using the test bench on the next page.

```
module test_lp_seq ;
parameter HF_CYCL = 5 ;
parameter CYCL = HF_CYCL * 2 ;
//
reg clk, rst ;
wire [2:0] lp ;
//
lp_seq lp_seq_01( .clk(clk), .rst_n(rst), .lp(lp) ) ;
//
initial begin
 $monitor("t=%d, rst=%b, lp=%b",
                $stime, rst, lp ) ;
 rst = 1'b1 ;
# CYCL rst = 1'b0 ;
#(CYCL * 20 ) $finish ;
end
//
always begin
  clk = 1'b0 ;   #HF_CYCL ;
  clk = 1'b1 ;   #HF_CYCL ;
end
endmodule
`include "lp_seq.v"
```

file name: test_lp_seq.v

Our module lp_seq does not output lp[2:0] directly from FF. The output is given through combinational logic.



lp[1] = lp_reg[1] | lp_reg[2]

Therefore, if there is some skew on FF output signals then lp signal may experience some flicker as shown on the left.

```
module lp_seq_cnt ( clk, rst, lp ) ;
input clk, rst ;
output [2:0] lp ;
wire clk, rst ;
wire [2:0] lp ;

reg [2:0] st_reg ;   // FF for state
reg [2:0] next_st_reg ;  // combinational logic

assign lp[0] = ( st_reg == 1 ) ;
assign lp[1] = ( (st_reg == 2) | (st_reg == 3) )  ;
assign lp[2] = ( (st_reg == 4) | (st_reg == 5) | (st_reg == 6 ) )  ;

always @ ( posedge clk  or posedge rst ) begin
  if ( rst ) begin
     st_reg <= 3'b000 ;
  end
  else begin
    st_reg <= next_st_reg ;
  end
end

always @ ( st_reg ) begin
    if ( st_reg >= 3'h6 ) begin
       next_st_reg = 3'b001 ;
    end
    else begin
       next_st_reg = st_reg + 3'b001 ;
    end
end
endmodule
```

This code is almost same to the previous code. It uses counter instead of shifter and uses the counter as state variable.

Parameter is not used because this logic works correctly only when number of lamps are 3.

Number of FF is reduced from previous 6 to 3. However adder and 7 3-bit comparators are newly needed.

file name: lp_seq_cnt.v

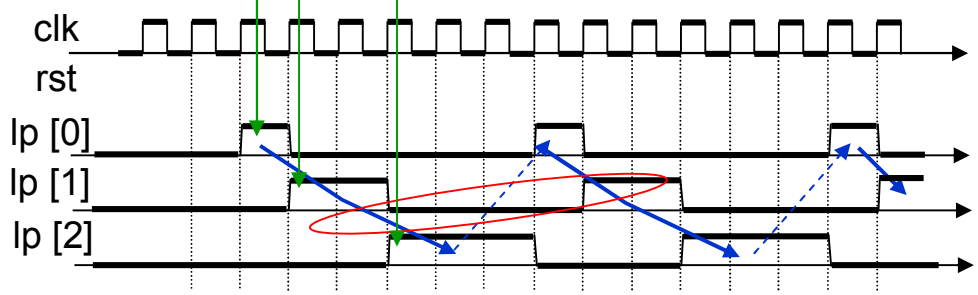
You may come to an idea shown below. It can work almost same to the module we wrote. But it is not synthesizable. Do not try to write a target module code in this way. The target module must go into silicon.

At each clock rise time do something, and it must be repeated forever.
The basic behavior of the lamps must looks like the code below.

```
always begin
  @ ( posedge clk )   lp[2:0] = 3'b001 ;
  @ ( posedge clk )   lp[2:0] = 3'b010 ;
  @ ( posedge clk ) ;
  @ ( posedge clk )   lp[2:0] = 3'b100 ;
  @ ( posedge clk ) ;
  @ ( posedge clk ) ;
end
```

This does not care about reset signal.

Next think how to handle reset.



A code block below written on the idea in the previous page is not synthesizable. It can be used only as a model of the target module, if you want.

Discontinue loop of updating lamp signal. → Use “disable” to discontinue the loop.

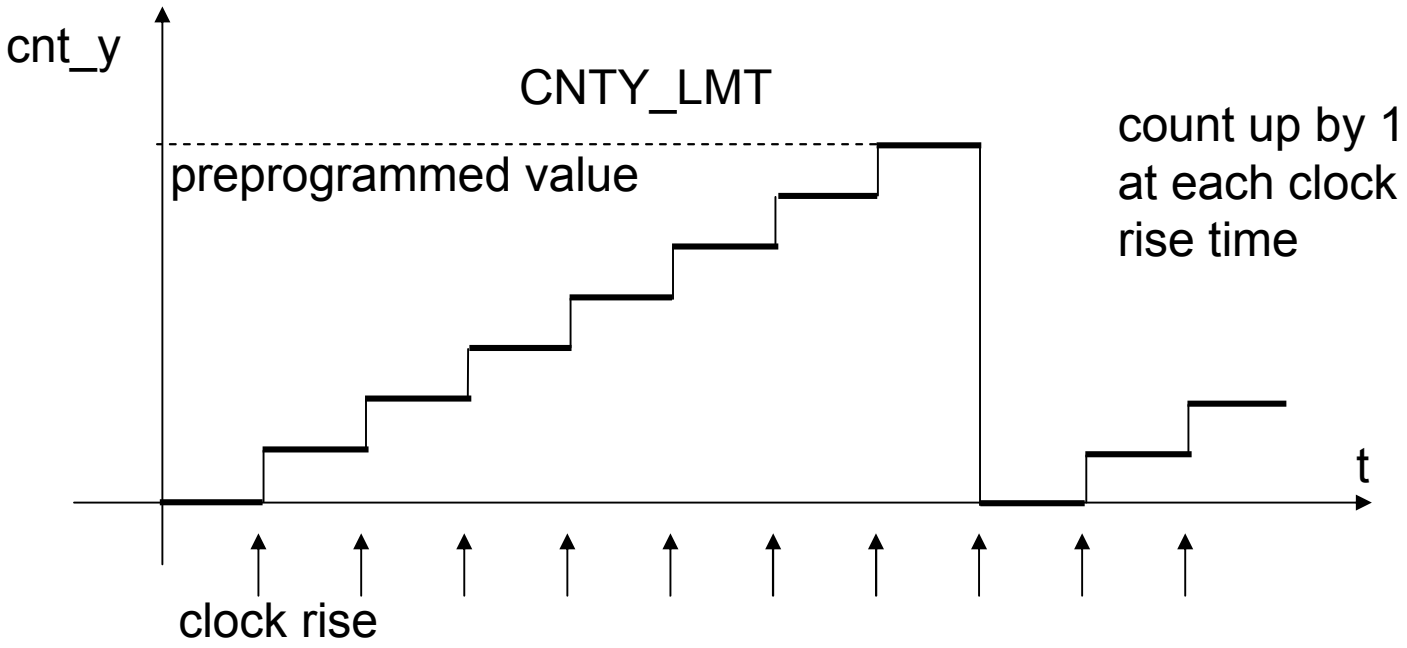
Turn off lamps triggered by reset signal. → Use another always construct and reset lp when rst asserted.

```
always begin : main_loop
   @ ( posedge clk )   lp[2:0] = 3'b001 ;
   @ ( posedge clk )   lp[2:0] = 3'b010 ;
   @ ( posedge clk ) ;
   @ ( posedge clk )   lp[2:0] = 3'b100 ;
   @ ( posedge clk ) ;
   @ ( posedge clk ) ;
end
```

Just wait for another two cycles.

```
always
  @ ( posedge clk or posedge rst  )
begin
    if ( rst == 1'b1 ) begin
      lp[2:0] = 3'b000 ;
      disable main_loop ;
    end
  end
```

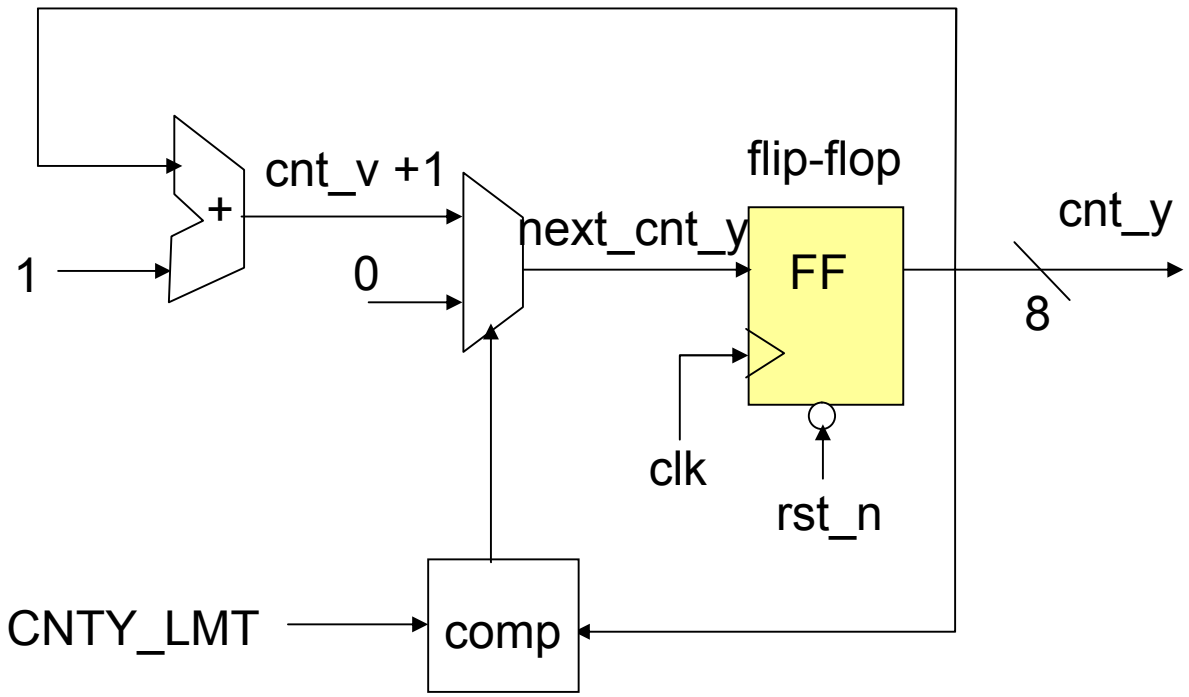
Ex 4-2. Counter with limiter: Write a 8-bit counter module which add 1 to its 8-bit output cnt_y at each clock rise time. The counter value must become 0 on reset and each time it is going to become larger than a constant value (CNTY_LMT).



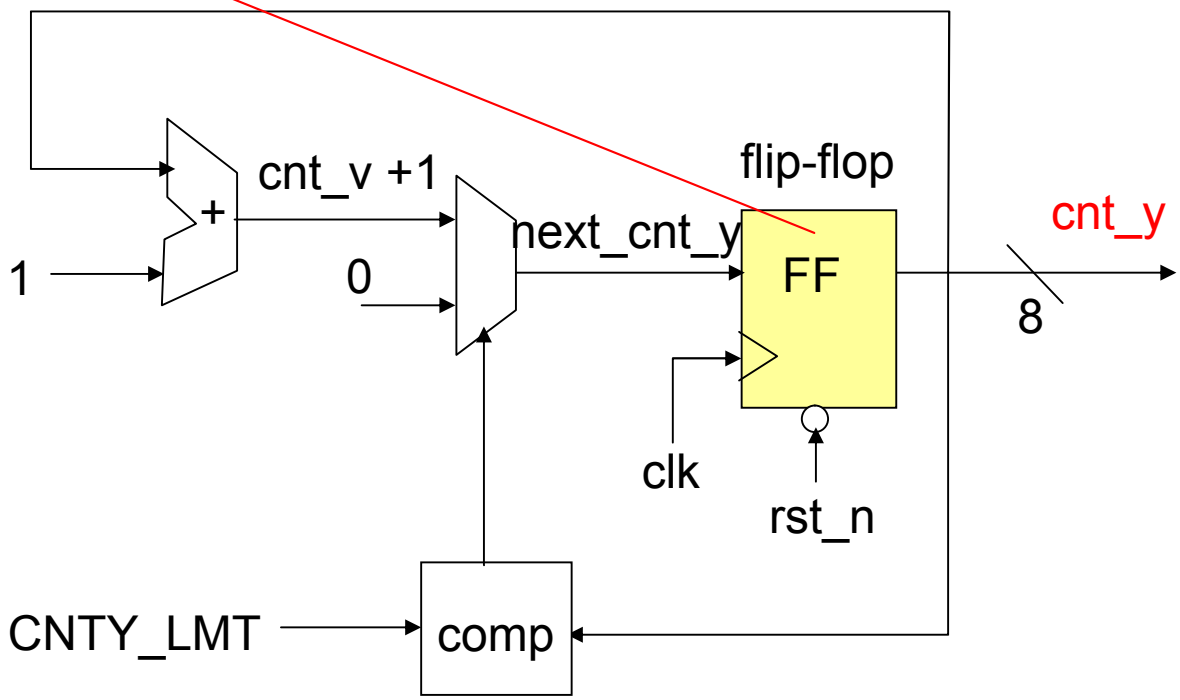
The value of the counter must be memorized so that it can be incremented by 1 at each clock rise time. Therefore, it must be implemented as a 8-bit flip-flop.

cnt_y must be defined as an output of a flip-flop whose bit width is 8.

And the input for this flip-flop, next_cnt_y, can be defined as an output of the combinational logic shown in this slide.

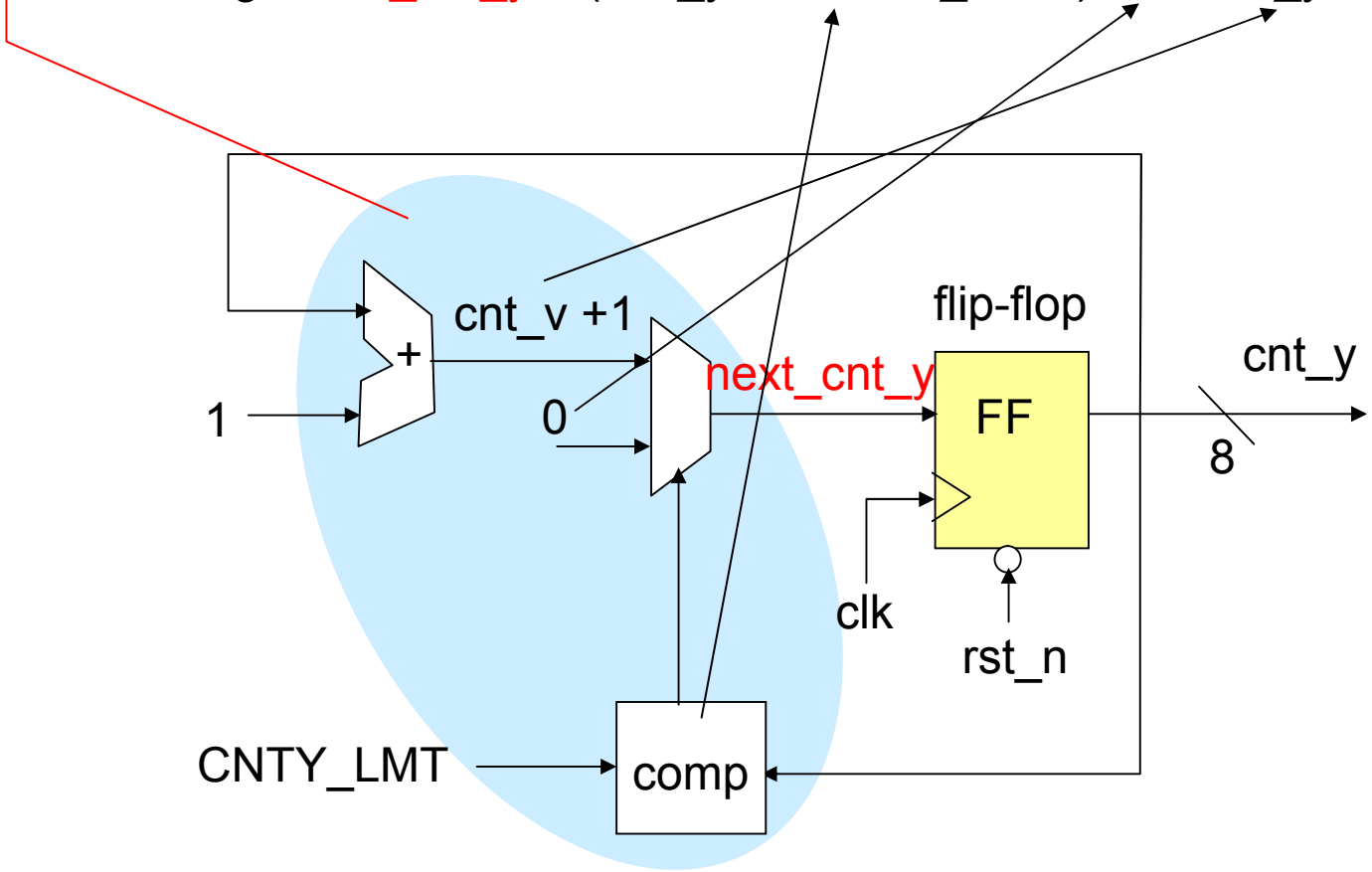


```
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
    cnt_y <= 0 ;
  end
  else begin
    cnt_y <= next_cnt_y ;
  end
end
```



next_cnt_y, the output of the combinational logic shaded by the blue oval, can be coded as shown in this slide.

```
assign next_cnt_y = ( cnt_y == CNTY_LMT )?  0 : cnt_y + 1 ;
```



Now, create a file named cntr_w_lmt.v for Ex 4-2. Write a test bench in the same file.

```
module cntr_w_lmt ( clk, rst_n, cnt_y ) ;
parameter CNTY_BW = 8 ;
parameter CNTY_LMT = 200 ; // limiter for counter max
//
input clk, rst_n ;
output [CNTY_BW-1:0] cnt_y ;
//
wire clk, rst_n ;
reg [CNTY_BW-1:0] cnt_y ; // FF for counter
//
wire  [CNTY_BW-1:0] next_cnt_y ; // input for counter FF
//
always @ (posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
    cnt_y <= { CNTY_BW {1'b0} } ;
  end
  else begin
    cnt_y <= next_cnt_y ;
  end
end
assign next_cnt_y = ( cnt_y == CNTY_LMT )?
                        { CNTY_BW {1'b0} } : cnt_y + 1 ;

endmodule
```

```
module test_cntr_w_lmt ;
parameter HF_CYCL = 50 ;
parameter CYCL = HF_CYCL * 2 ;
parameter CNTY_BW = 8 ;
defparam cntr_w_lmt_01.CNTY_LMT = 7 ;
//
reg clk, rst_n ;
wire [CNTY_BW-1:0] cnt_y ;
//
cntr_w_lmt    cntr_w_lmt_01 ( .clk(clk), .rst_n(rst_n), .cnt_y(cnt_y) ) ;
//
initial begin
rst_n = 1'b0 ;
#(CYCL*2) rst_n = 1'b1 ;
#(CYCL*20) $finish ;
end
always begin
clk = 0 ; #HF_CYCL ;
clk = 1 ; #HF_CYCL ;
end
always @ ( posedge clk ) begin
  $strobe( "time=%d, rst_n=%b, cnt_y=%d",
              $stime, rst_n, cnt_y ) ;
end
endmodule
```

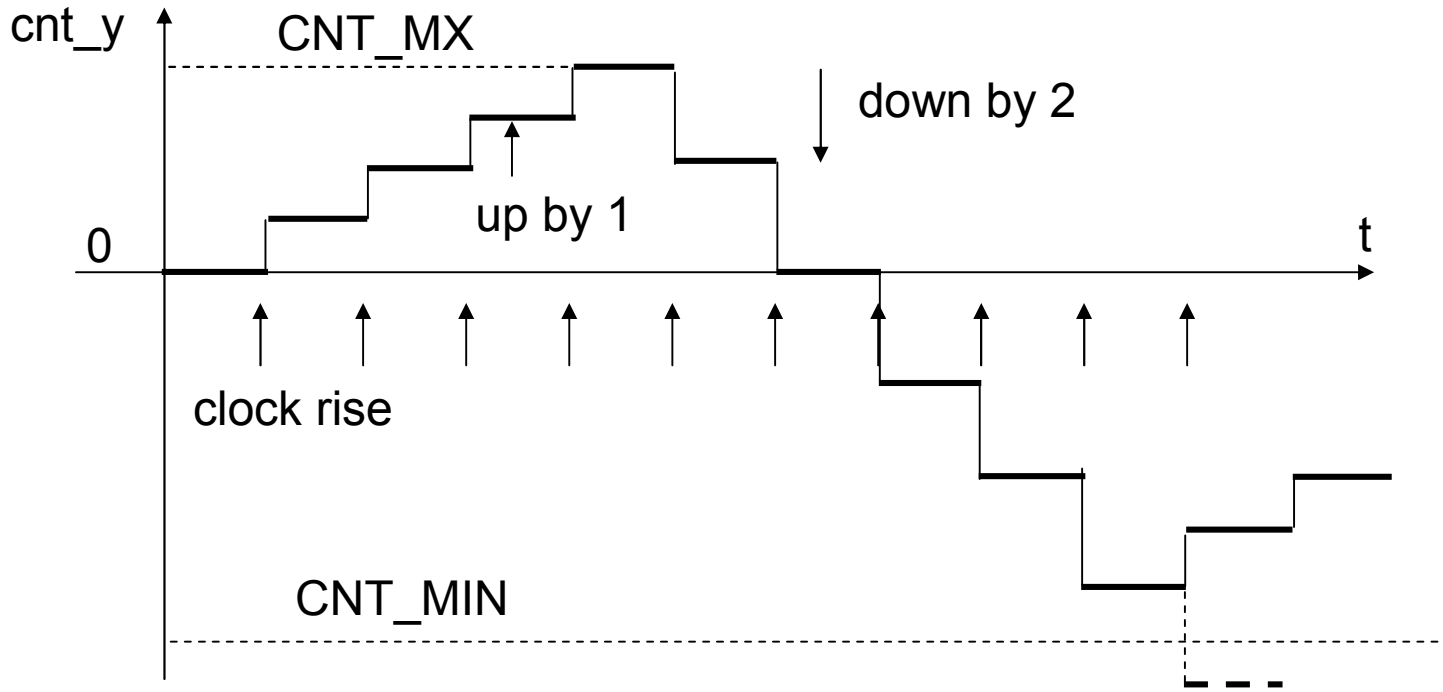
file name: cntr_w_lmt.v

Run this file and check the result.

A sample result

```
time=        50, rst_n=0, cnt_y=  0
time=       150, rst_n=0, cnt_y=  0
time=       250, rst_n=1, cnt_y=  1
time=       350, rst_n=1, cnt_y=  2
time=       450, rst_n=1, cnt_y=  3
time=       550, rst_n=1, cnt_y=  4
time=       650, rst_n=1, cnt_y=  5
time=       750, rst_n=1, cnt_y=  6
time=       850, rst_n=1, cnt_y=  7
time=       950, rst_n=1, cnt_y=  0
time=      1050, rst_n=1, cnt_y=  1
time=      1150, rst_n=1, cnt_y=  2
time=      1250, rst_n=1, cnt_y=  3
time=      1350, rst_n=1, cnt_y=  4
time=      1450, rst_n=1, cnt_y=  5
time=      1550, rst_n=1, cnt_y=  6
time=      1650, rst_n=1, cnt_y=  7
time=      1750, rst_n=1, cnt_y=  0
time=      1850, rst_n=1, cnt_y=  1
time=      1950, rst_n=1, cnt_y=  2
time=      2050, rst_n=1, cnt_y=  3
time=      2150, rst_n=1, cnt_y=  4
Halted at location **cntr_w_lmt.v(
```

Ex 4-3. Up down counter and STT: Write an up/down counter which count up to CNT_MX by 1 and then count down to CNT_MIN by 2. When it reaches CNT_MIN or going to become smaller than CNT_MIN, start counting up again to CNT_MX and after it reaches CNT_MX, count down to CNT_MIN. Repeat this up and down operation until asynchronous active low reset signal applied. Use Verilog2001. Write a state transition table, STT, before writing the code.



Because this system must behave differently while counting up and counting down. There must be some kind of state variable needed.
cnt_y and state must be defined as flip-flop to memorize their values.

How many signals do we need which must be memorized???

Do we need state?? → Yes, we need.

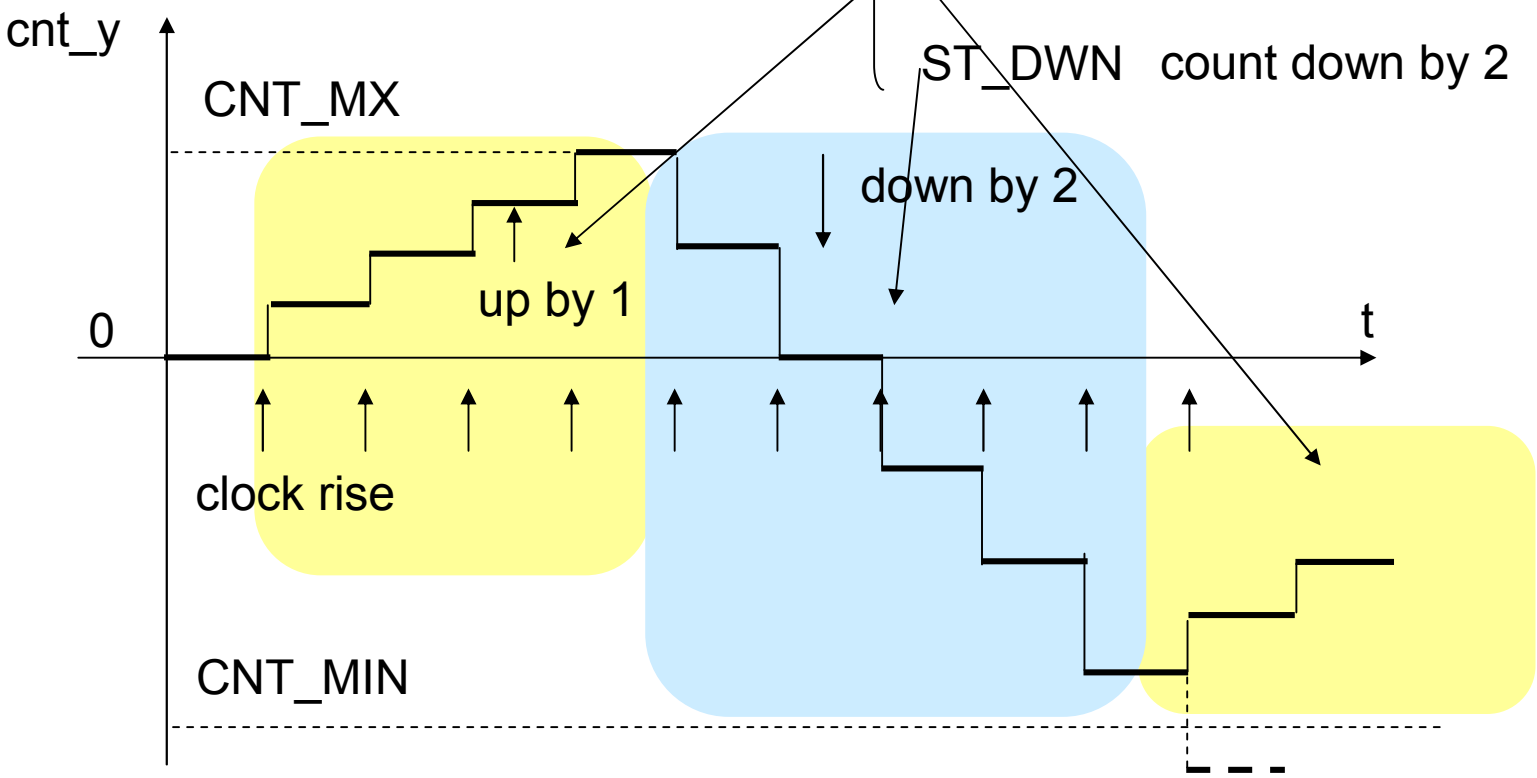
State must be memorized.

→ State variable must be a flip-flop.

ST_INTL while reset asserted

ST_UP count up by 1

ST_DWN count down by 2

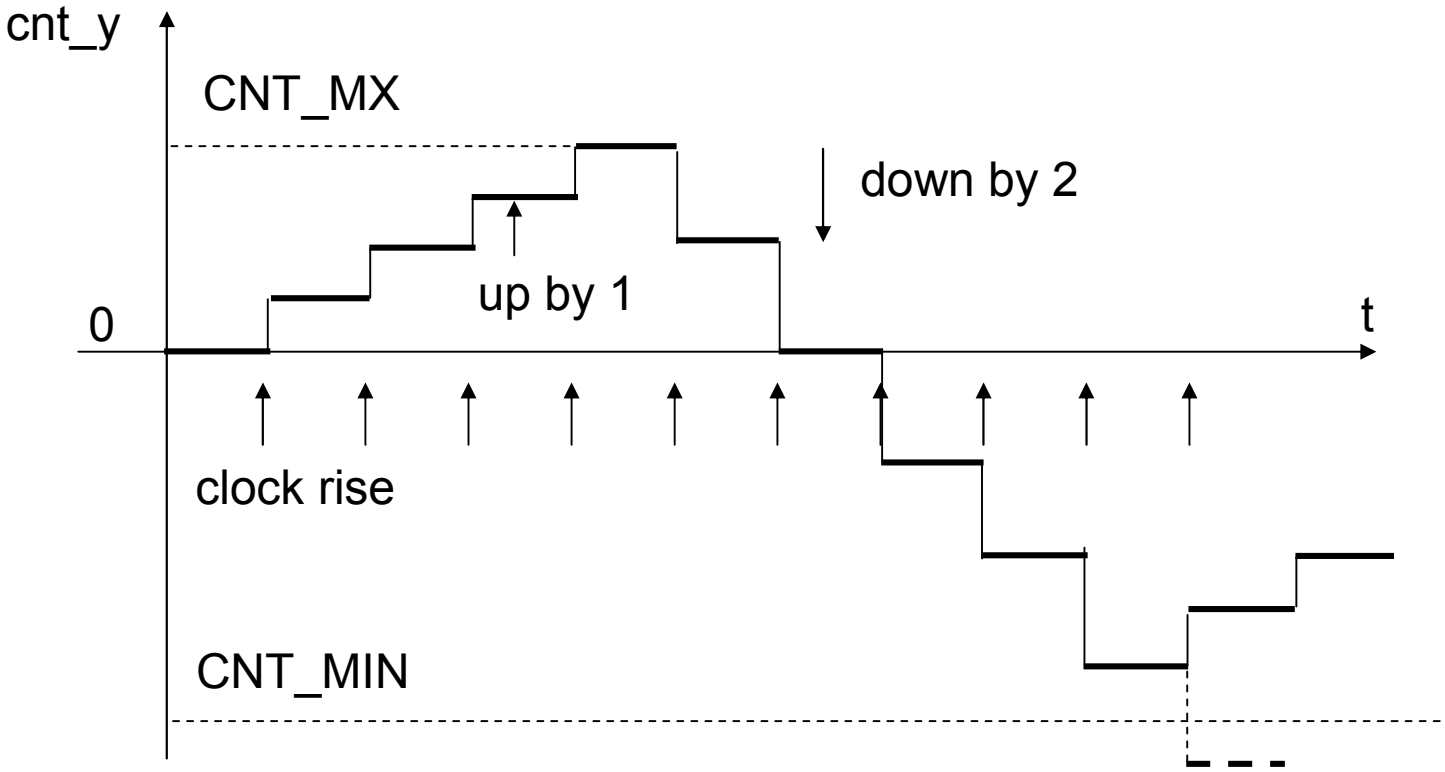


How many signals do we need which must be memorized???

Do we have to memorize counter value?? → Yes, we have to because it must keep its value for one cycle.

cnt_y must be memorized.

→ cnt_y must be a flip-flop.



Now, draw a state transition table for state variable and cnt_y.

event \ state		ST_INTL	ST_UP	ST_DWN
rst_n=0		⇨ ST_INTL		
rst_n=1	clk	cnt_y = 0 ⇨ ST_UP	if cnt_y>=CNT_MX cnt_y = cnt_y-2 ⇨ ST_DWN else cnt_y = cnt_y+1	if cnt_y<=CNT_MIN+1 cnt_y = cnt_y+1 ⇨ ST_UP else cnt_y = cnt_y-2
rst_n=1	other than above	no-operation		

Now, define flip-flops for state and cnt_y.

```
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
    state <= ST_INTL ;
  end
  else begin
    state <= next_state ;
  end
end

always @ ( posedge clk ) begin
   cnt_y <= next_ cnt_y ;
end
```

The code must looks like the one shown on this slide.

You must follow the template of a flip-flop.

You may use reset signal to initialize cnt_y, but we do not have to apply reset signal to cnt_y. The code shown in this slide dose not apply reset on cnt_y.

Next, create logic for next_state.

event \ state	ST_INTL	ST_UP	ST_DWN
rst_n=0	⇨ ST_INTL		
rst_n=1, clk	⇨ ST_UP	if cnt_y>=CNT_MX ⇨ ST_DWN else no change	if cnt_y<=CNT_MIN+1 ⇨ ST_UP else no change

next_state = ST_UP ;

For the state ST_INTL, state always changes to ST_UP.

Therefore the code corresponding this box, is “next_state = ST_UP ; “.

event \ state	ST_INTL	ST_UP	ST_DWN
rst_n=0	⇨ ST_INTL		
rst_n=1, clk	⇨ ST_UP	if cnt_y>=CNT_MX ⇨ ST_DWN else no change	if cnt_y<=CNT_MIN+1 ⇨ ST_UP else no change

next_state = ST_UP ;

next_state = (cnt_y >= CNT_MX)? ST_DWN : state ;

event \ state	ST_INTL	ST_UP	ST_DWN
rst_n=0	⇨ ST_INTL		
rst_n=1, clk	⇨ ST_UP	if cnt_y>=CNT_MX ⇨ ST_DWN else no change	if cnt_y<=CNT_MIN+1 ⇨ ST_UP else no change

```
next_state = ST_UP ;

next_state = (cnt_y >= CNT_MX)? ST_DWN : state ;

next_state = (cnt_y <= CNT_MIN+1)? ST_UP : state ;
```

Integrate these three sentences into one by using case statement.

event \ state	ST_INTL	ST_UP	ST_DWN
rst_n=0	⇨ ST_INTL		
rst_n=1 clk	⇨ ST_UP	if cnt_y>=CNT_MX ⇨ ST_DWN else no change	if cnt_y<=CNT_MIN+1 ⇨ ST_UP else no change

```
case ( state )
  ST_INTL :
    next_state = ST_UP ;
  ST_UP :
    next_state = (cnt_y >= CNT_MX)? ST_DWN : state ;
  ST_DWN :
    next_state = (cnt_y <= CNT_MIN+1)? ST_UP : state ;
endcase
```

Next, let's think next_cnt_y logic.

We can get this table for cnt_y.

event \ state	ST_INTL	ST_UP	ST_DWN
rst_n=0	no-operation		
rst_n=1, clk ↑	cnt_y = 0	if cnt_y>=CNT_MX cnt_y = cnt_y-2 else cnt_y = cnt_y+1	if cnt_y<=CNT_MIN+1 cnt_y = cnt_y+1 else cnt_y = cnt_y-2

Applying the same procedure, we can get a logic block for cnt_y.

event \ state	ST_INTL	ST_UP	ST_DWN
rst_n=0	no-operation		
rst_n=1, clk ↑	cnt_y = 0	if cnt_y>=CNT_MX cnt_y = cnt_y-2 else cnt_y = cnt_y+1	if cnt_y<=CNT_MIN+1 cnt_y = cnt_y+1 else cnt_y = cnt_y-2

```
case ( state )
  ST_INTL : begin
            next_ cnt_y = 0 ;
        end
  ST_UP   : begin
            next_ cnt_y = (cnt_y >= CNT_MX)? cnt_y-2 : cnt_y+1 ;
        end
  ST_DWN : begin
            next_ cnt_y = (cnt_y <= CNT_MIN+1)? cnt_y+1 : cnt_y-2 ;
        end
endcase
```

Now, combine these pieces of code into one file named updwn_cntr_w_lmt.v.

```
module updwn_cntr_w_lmt ( clk, rst_n, cnt_y ) ;
parameter CNTY_BW = 8 ;
parameter CNT_MX = 100 ; // must not be smaller than 2,
                         // nor larger than 2**CNTY_BW -1
parameter CNT_MIN = -30 ;  // must smaller than CNT_MX-1, but
                           // must not be smaller than -2**(CNTY_BW-1)
//
parameter ST_INTL = 2'b00 ;
parameter ST_UP = 2'b01 ;
parameter ST_DWN = 2'b10 ;
//
input clk, rst_n ;
output signed [CNTY_BW-1:0] cnt_y ;
//
wire clk, rst_n ;
reg signed [CNTY_BW-1:0] cnt_y ; // FF for counter
//
reg [1:0] state ; // FF for state variable
reg [1:0] next_state ; // non-FF, input for state FF
reg  signed [CNTY_BW-1:0] next_cnt_y ; // non-FF, input for counter FF
//
```

cnt_y must be declared as signed because it must be able to handle negative value.

```
// FF definitions
always @ (posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
    state <= ST_INTL ;
  end
  else begin
    state <= next_state ;
  end
end

// inputs for FF data creattion
always @ ( state or cnt_y ) begin
 case ( state )
   ST_INTL : begin
             next_state = ST_UP ;
         end
   ST_UP    : begin
             next_state = (cnt_y >= CNT_MX)? ST_DWN : state ;
         end
   ST_DWN : begin
             next_state = (cnt_y <= CNT_MIN+1)? ST_UP : state ;
         end
   default : begin
             next_state = 2'bxx ;
         end
 endcase
end
```

default is added to avoid creating a latch for next_state.

```
always @ ( posedge clk ) begin
   cnt_y <= next_cnt_y ;
end

always @ ( state or cnt_y ) begin
  case ( state )
   ST_INTL : begin  // initialize counter in INTL state
            next_cnt_y = { CNTY_BW {1'b0} }  ;
        end
   ST_UP    : begin // next_cnt_y must not eceed CNT_MX
            next_cnt_y = (cnt_y >= CNT_MX)? cnt_y-2 : cnt_y+1 ;
        end
   ST_DWN : begin // next_cnt_y must not be smaller than CNT_MIN
            next_cnt_y = (cnt_y <= CNT_MIN+1)? cnt_y+1 : cnt_y-2 ;
        end
   default : begin
            next_cnt_y = { CNTY_BW {1'bx} } ;
        end
  endcase
end
//
endmodule
```

default is added to avoid a latch for next_cnt_y.

file name: updwn_cntr_w_lmt.v

```
module test_updwn_cntr_w_lmt ;
parameter HF_CYCL = 50 ;
parameter CYCL = HF_CYCL * 2 ;
parameter CNTY_BW = 8 ;
defparam updwn_cntr_w_lmt_01.CNT_MX = 4 ;
defparam updwn_cntr_w_lmt_01.CNT_MIN = -5 ;
//
reg clk, rst_n ;
wire signed[CNTY_BW-1:0] cnt_y ;
//
updwn_cntr_w_lmt    updwn_cntr_w_lmt_01 (
                                .clk(clk),  .rst_n(rst_n),  .cnt_y(cnt_y)
                                ) ;
//
initial begin
rst_n = 1'b0 ;
#(CYCL*2) rst_n = 1'b1 ;
#(CYCL*20) $finish ;
end
```

```
always begin
clk = 0 ; #HF_CYCL ;
clk = 1 ; #HF_CYCL ;
end

always @ ( posedge clk ) begin
  $strobe( "time=%d, rst_n=%b, cnt_y=%d",
              $stime, rst_n, cnt_y ) ;
end
endmodule
//
`include "test_updwn_cntr_w_lmt.v"
```

file name: test_updwn_cntr_w_lmt.v

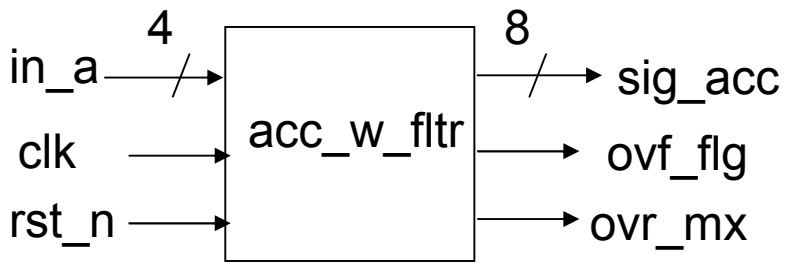
Run test_updwn_cntr_w_lmt.von your PC
and check the result.
Try changing parameters.

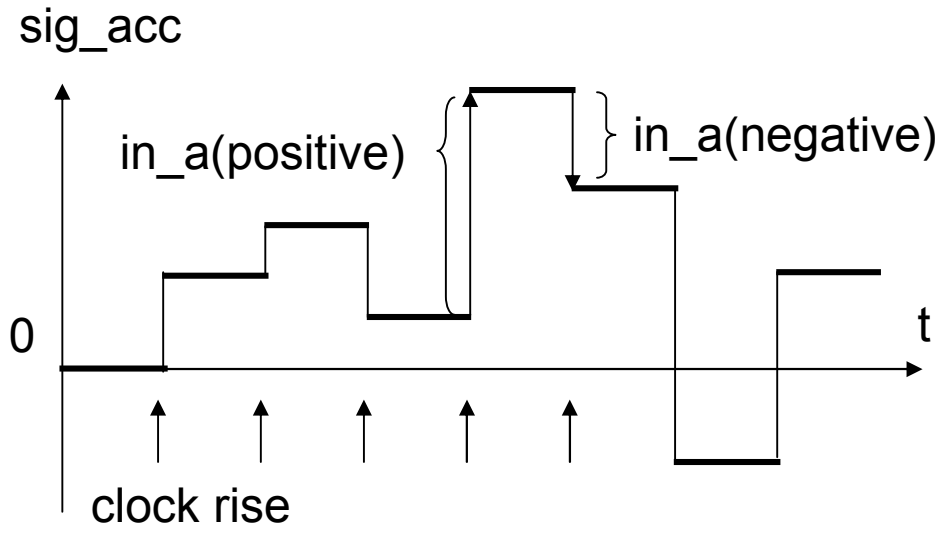
```
defparam updwn_cntr_w_lmt_01.CNT_MX = 4 ;
defparam updwn_cntr_w_lmt_01.CNT_MIN = -5 ;
```

A sample result

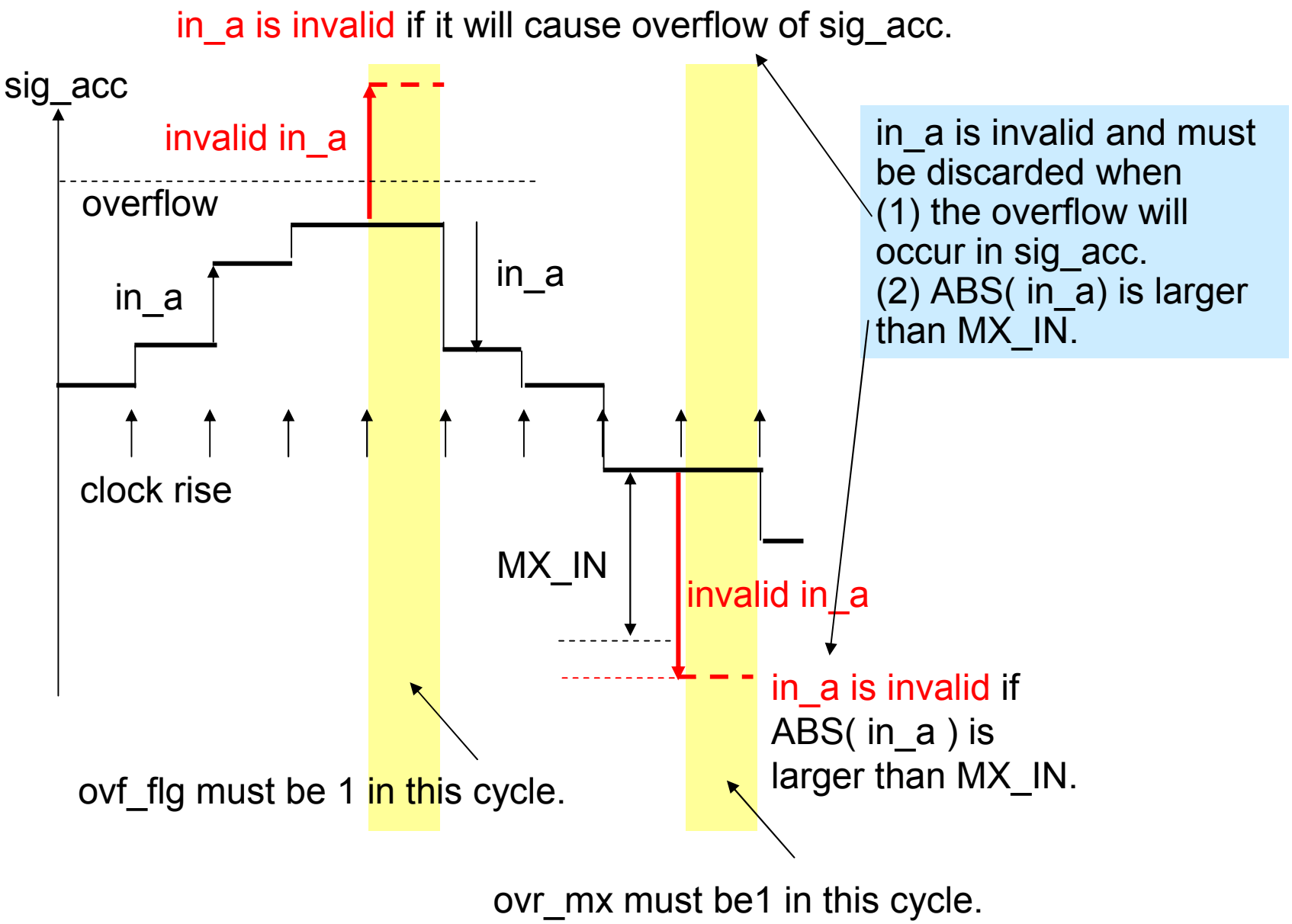
```
time=      50, rst_n=0, cnt_y=  0
time=     150, rst_n=0, cnt_y=  0
time=     250, rst_n=1, cnt_y=  0
time=     350, rst_n=1, cnt_y=  1
time=     450, rst_n=1, cnt_y=  2
time=     550, rst_n=1, cnt_y=  3
time=     650, rst_n=1, cnt_y=  4
time=     750, rst_n=1, cnt_y=  2
time=     850, rst_n=1, cnt_y=  0
time=     950, rst_n=1, cnt_y= -2
time=    1050, rst_n=1, cnt_y= -4
time=    1150, rst_n=1, cnt_y= -3
time=    1250, rst_n=1, cnt_y= -2
time=    1350, rst_n=1, cnt_y= -1
time=    1450, rst_n=1, cnt_y=  0
time=    1550, rst_n=1, cnt_y=  1
time=    1650, rst_n=1, cnt_y=  2
time=    1750, rst_n=1, cnt_y=  3
time=    1850, rst_n=1, cnt_y=  4
time=    1950, rst_n=1, cnt_y=  2
time=    2050, rst_n=1, cnt_y=  0
time=    2150, rst_n=1, cnt_y= -2
Halted at location **updwn_cntr_w_
```

Ex. 4-4. Sequential accumulator with overflow checker: Write a module which accumulates 4-bit input in_a at each clock rise time. The accumulation must be done only when in_a is valid. If it is not valid, discard it. in_a is invalid when ABS(in_a) is larger than a predefined value MX_IN or accumulation will cause overflow. The first in_a, right after reset negated, must be neglected.
The accumulated value must be placed on 8-bit output, sig_acc.
The module must place 1 to output ovf_flg when in_a will cause overflow.
Also it must place 1 to output ovr_mx when ABS(in_a) is larger than MX_IN.
Use Verilgo2001. The initial value of sig_acc must be 0.





Use FF for outputs, sig_acc, ovf_flg, and ovr_mx. Use $random to generate random number for test inputs.



Do we need state to realize the system???

Yes, we do.
Because, there are two cases,
(1) reset is asserted and no accumulation done, and
(2) reset is negated and accumulation must be done.

ST_INTL
no accumulation

ST_RUN
accumulate in_a

How many signals do we need which must be memorized???

(1) sig_acc must be memorized to hold the accumulated value.

(2) ovf_flg must be memorized so it keeps the same value for one cycle.

(3) ovr_mx must be memorized so it keeps the same value for one cycle.

About FF output of ovr_mx and ovf_flg

revised in v2r03

You may have a question like “ovr_mx and ovf_flg can be implemented as outputs of combinational logic, they may not be implemented as FFs.”

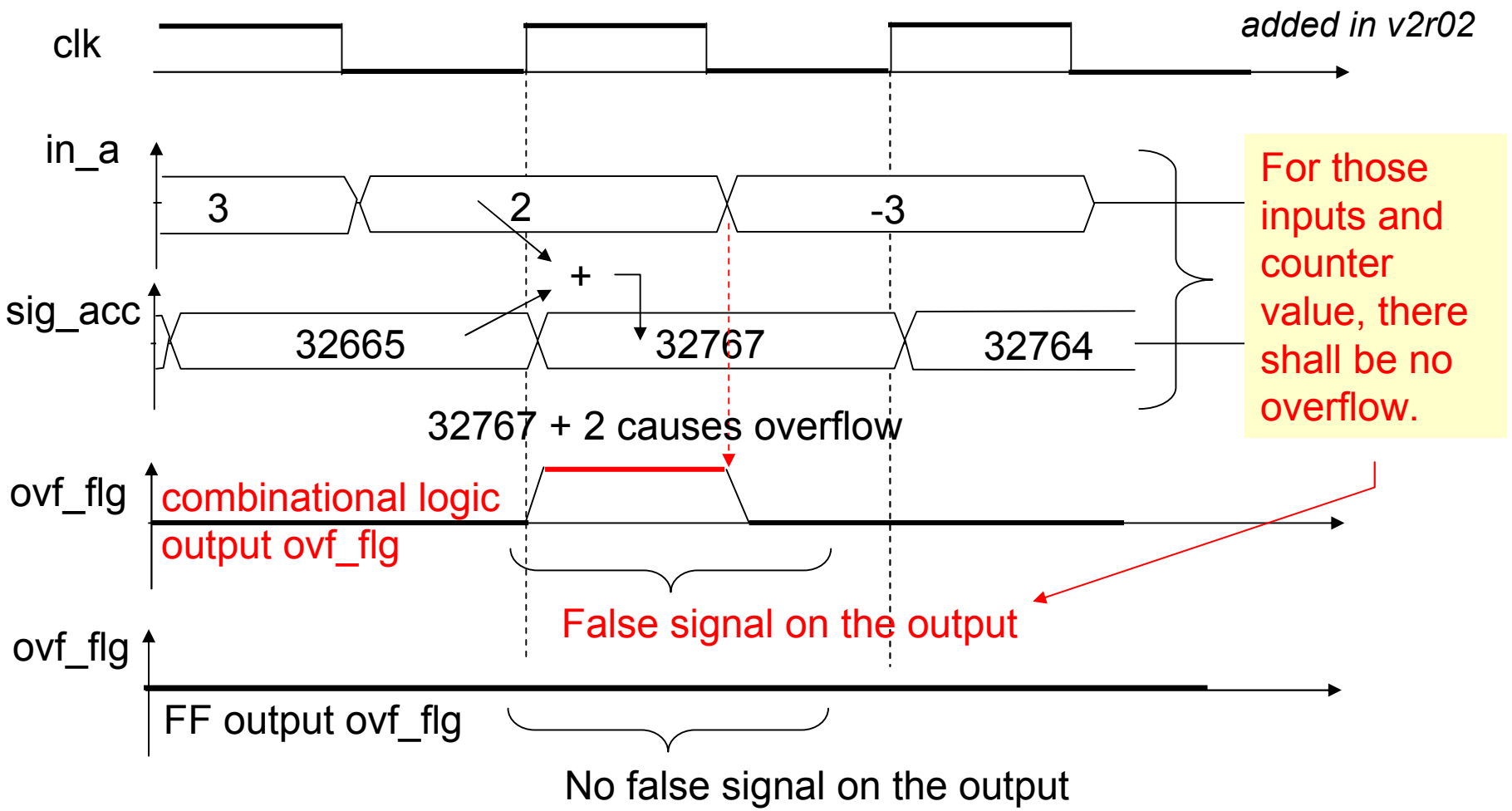
Combinational logic idea must looks like the following;

```
reg [7:0] sum ;

always @ ( in_a or sig_acc ) begin
sum = in_a + sig_acc ;
ovf_flg = ((in_a[3] & sig_acc[7] ) & (~sum[7]) ) |
          (((~in_a[3]) & (~sig_acc[7]) ) & sum[7] )
end
```

Flag is an output of combinational logic.

The logic means that whenever in_a + sig_acc causes overflow, ovf_flg becomes 1. Therefore ovf_flg can change its value whenever in_a or sig_acc changes.

→ This will cause the problem shown on the next page.



The logic on the previous slide shows that ovf_flg becomes 1, while in_a is 2 and sig_acc is 32767 because signed 16-bit data can not hold 32769.
Therefore, false signal will appear on ovf_flg.
If we use FF for ovf_flg, such false data will not come out. Please look carefully into the sample code and see yourself such false data will not come out.

revised in v2r02

Now we can get the state transition table as below.

event \ state		ST_INTL	ST_RUN
rst_n=0		sig_acc =0	ovf_flg = 0 ovr_mx = 0 ⇨ ST_INTL
rst_n=1	clk	no-operation ⇨ ST_RUN	(1) if ABS(in_a) > MX_IN , ovr_mx=1 else ovr_mx=0 (2) if *overflow* , ovf_flg =1 else ovf_flg = 0 (3) if ~ovr_mx & ~ovr_flg , sig_acc = sig_acc + in_a else sig_acc = sig_acc
rst_n=1	other than above	no-operation	

First, think state control logic.

This is a state transition table focusing on just a state variable only.

event		ST_INTL	ST_RUN
rst_n=0		⇨ ST_INTL	
rst_n=1	clk	⇨ ST_RUN	no-change
rst_n=1	other than above	no-change	

```
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
    state <= ST_INTL ;
  end
  else begin
    state <= next_state ;
  end
end
//
assign next_state = ST_RUN ;
```

We can get this code.

From ST_INTL, state changes to ST_RUN if reset negated.

So far as reset asserted, state will never change to ST_RUN even if next_state is assigned the value ST_RUN always.

Next, think ovr_mx.

revised in v2r02

This is a state transition table focusing on just ovr_mx only.

event		ST_INTL	ST_RUN
rst_n=0		ovr_mx=0	
rst_n=1	clk	no-operation	(1) if ABS(in_a) <= MX_IN, ovr_mx=0 else ovr_mx=1
rst_n=1	other than above	no-operation	

```
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
    ovr_mx <= 0 ;
  end
  else begin
    ovr_mx <= next_ovr_mx ;
  end
end
//

wire [3:0] abs_in_a;
assign abs_in_a =
        ( in_a[3] )? -in_a : in_a ;
```

```
always @ ( state or abs_in_a ) begin
  case ( state )
    ST_INTL : begin
        next_ovr_mx = ovr_mx ;
    end
    ST_RUN : begin
        next_ovr_mx_flg =
            ( abs_in_a <= MX_IN ) ?  0 : 1 ;
    end
  endcase
end
```

Next, think ovf_flg.

This is a state transition table focusing on just ovf_flg only.

event		ST_INTL	ST_RUN
rst_n=0		ovf_flg =0	
rst_n=1	clk	no-operation	(2) if *overflow* , ovf_flg =1 else ovf_flg = 0
rst_n=1	other than above	no-operation	

```
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
    ovf_flg <= 0 ;
  end
  else begin
    ovf_flg <= next_ovf_flg ;
  end
end
//
```

```
always @ ( state or in_a ) begin
  case ( state )
    ST_INTL : begin
        next_ovf_flg = ovf_flg ;
      end
    ST_RUN : begin
        next_ovf_flg =  See next page
      end
  endcase
end
```

Overflow can be detected by checking sign bits of the result and operands as in the expression below. It checks sign bits of next_sig_acc, that is a result of sig_acc+in_asig_acc, and next_sig_acc, and in_a.

positive + positive becomes negative

```
next_ovf_flg = ( (~sig_acc[7]) & (~in_a[3]) & next_sig_acc[7] ) |
               ( sig_acc[7] & in_a[3] & (~next_sig_acc[7]) )? 1 : 0 ;
```

negative + negative becomes positive

This is a state transition table focusing on sig_acc

event		ST_INTL	ST_RUN
rst_n=0		sig_acc = 0	
rst_n=1	clk	no-operation	(3) if ~ovr_mx & ~ovr_flg, sig_acc = sig_acc + in_a else sig_acc = sig_acc

In this procedure, sig_acc is given a new value in a path but not given any value in the other path. There are two ways to implement this as shown below. A style on the left is used for ovr_mx and ovf_flg.

```
always @ ( posedge ,,,) begin
   sig_acc = next_sig_acc ;
end
always @ ( ,,,, ) begin
   if ( condition ) begin
     next_sig_acc =
sig_acc+in_a;
   end
   else  begin
     next_sig_acc = sig_acc ;
   end
end
```

FF without enable input

different on condition

```
always @ ( posedge ,,,) begin
   if (condition) begin
        sig_acc = next_sig_acc ;
   end
end

always @ ( ,,,, ) begin
     next_sig_acc = sig_acc+in_a;
end
```

always the same

FF with enable input

FF without enable input

```
always @ ( posedge ,,,) begin
  sig_acc = next_sig_acc ;
end
always @ ( ,,,, ) begin
  if ( condition ) begin
    next_sig_acc = sig_acc+in_a;
  end
  else  begin
    next_sig_acc = sig_acc ;
  end
end
```

FF with enable input

```
always @ ( posedge ,,,) begin
  if (condition) begin
      sig_acc = next_sig_acc ;
  end
end

always @ ( ,,,, ) begin
    next_sig_acc = sig_acc+in_a;
end
```

Logically the same

different

next_sig_acc is not sig_acc+in_a depending on *condition*.

next_sig_acc is always aig_acc+in_a regardless of *condition*.

next_ovf_flg can not be calculated correctly by the expression below because depending on *condition*, next_sig_acc is not sig_acc+in_a.

```
next_ovf_flg = ( (~sig_acc[7]) & (~in_a[3]) & next_sig_acc[7] ) |
               ( sig_acc[7] & in_a[3] & (~next_sig_acc[7]) )? 1 : 0 ;
```

next_ovf_flg is correct only when next_sig_acc is sig_acc+in_a.

We have to use "FF with enable input" sytle for sig_acc so that next_ovf_flg has correct value always.

This is a state transition table focusing on sig_acc

event		ST_INTL	ST_RUN
rst_n=0		sig_acc = 0	
rst_n=1	clk	no-operation	(3) if ~ovr_mx & ~ovr_flg, sig_acc = sig_acc + in_a else sig_acc = sig_acc

```
always @ ( posedge clk or negedge rst_n) begin
  if ( rst_n == 0) begin
    sig_acc = 0 ;
  end
  else begin
   if ( in_a_valid ) begin
     sig_acc = next_sig_acc ;
   end
  end
end
//
assign next_sig_acc =
                sig_acc + in_a ;
```

```
reg in_a_valid ;
always @ ( state or ovr_mx
                    or ovf_flg ) begin
  case ( state )
    ST_INTL : begin
        in_a_valid = 0 ;
      end
    ST_RUN : begin
        in_a_valid = ((~ovr_mx)& (~ovf_flg))? 1 : 0 ;
      end
  endcase
end
```

This is to discard the first input. If you want to accept it, change this to 1.

```
module acc_w_fltr ( clk, rst_n, in_a, sig_acc, ovr_mx, ovf_flg ) ;
parameter IN_BW = 4 ;
parameter OUT_BW = 8 ;
parameter MX_IN = 5 ;
//
parameter ST_INTL = 1'b0 ;
parameter ST_RUN =1'b1 ;
//
input clk, rst_n ;
input signed [IN_BW-1:0] in_a ;
output signed [OUT_BW-1:0] sig_acc ;
output ovr_mx, ovf_flg ;  // overflow indicator and over MX_IN indicator
//
wire clk, rst_n ;
wire signed [IN_BW-1:0] in_a ;
reg signed [OUT_BW-1:0] sig_acc ; // FF for accumulation
reg ovr_mx,  ovf_flg ; // FF for overflow indicator and over MX_IN indicator
//
```

Parameters are introduced to make the code flexible.

```
reg state ; // FF, state
wire next_state ; // input for state FF
wire signed [OUT_BW-1:0] next_sig_acc ; // input for FF sig_acc
reg next_ovf_flg, next_ovr_mx ; // non-FF, input for ovf_flg FF and ovr_mx FF
reg in_a_valid ; // non-FF, enable flag to set sig_acc FF
wire [IN_BW-1:0] abs_in_a ; // absolute value of in_a
//

// logic start

// state FF and state control logic
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
    state <= ST_INTL ;
  end
  else begin
    state <= next_state ;
  end
end
//
assign next_state = ST_RUN ;
```

```
// over max flag
assign abs_in_a =  ( in_a[IN_BW-1] )? -in_a : in_a ;  // absolute value of in_a
always @ ( state or abs_in_a ) begin
  case ( state )
    ST_INTL : begin
          next_ovr_mx = ovr_mx ;
        end
    ST_RUN : begin
          next_ovr_mx = ( abs_in_a  <= MX_IN ) ?  1'b0 : 1'b1 ;
        end
    default : begin next_ovr_mx = 1'bx ; end
  endcase
end

always @ ( posedge clk or negedge rst_n ) begin
   if ( rst_n == 1'b0 ) begin
      ovr_mx <= 0 ;
   end
   else begin
      ovr_mx <= next_ovr_mx ;
   end
end
//
```

default is added for debug

```
// overflow flag
wire ovf_tmp ; // temporally work for next_ovf_flg
assign ovf_tmp =
    ( (~sig_acc[OUT_BW-1]) & (~in_a[IN_BW-1]) & next_sig_acc[OUT_BW-1] ) |
    ( sig_acc[OUT_BW-1] & in_a[IN_BW-1] &  (~next_sig_acc[OUT_BW-1]) )?
     1'b1 : 1'b0 ;
always @ ( state or in_a ) begin
  case ( state )
    ST_INTL : begin
          next_ovf_flg = ovf_flg ;
        end
    ST_RUN : begin
          next_ovf_flg = ovf_tmp ;
        end
    default : begin  next_ovf_flg = 1'bx ; end
  endcase
end
always @ ( posedge clk or negedge rst_n ) begin
   if ( rst_n == 1'b0 ) begin
      ovf_flg <= 0 ;
   end
   else begin
      ovf_flg <= next_ovf_flg ;
   end
end
```

default is added for debug

```
//
// enable flag, in_a_valid, control logic
always @ ( state or next_ovr_mx or next_ovf_flg ) begin
  case ( state )
    ST_INTL : begin
          in_a_valid = 1'b1 ;
        end
    ST_RUN : begin
          in_a_valid = ((~next_ovr_mx) & (~next_ovf_flg))? 1'b1 : 1'b0 ;
        end
    default : begin
          in_a_valid = 1'bx ;
        end
  endcase
end
```

next_ovf_flg is used instead of ovf_flg.

next_ovr_mx is used instead of ovr_mx.

default is added for debug

```
//
// sig_acc FF and sig_acc control logic
always @ ( posedge clk or negedge rst_n ) begin
   if ( rst_n == 1'b0) begin
      sig_acc = {OUT_BW{1'b0}} ;
   end
   else begin
     if ( in_a_valid ) begin
       sig_acc = next_sig_acc ;
     end
   end
end
// next_sig_acc must always be set to caluculate ovf_ind
assign next_sig_acc = sig_acc + in_a ; // Verilog2001
//


endmodule
```

file name: acc_w_fltr.v

```
module test_acc_w_fltr ;
parameter IN_BW = 4 ;
defparam acc_w_fltr_01.IN_BW = IN_BW ;
parameter OUT_BW = 8 ;
defparam acc_w_fltr_01.OUT_BW = OUT_BW ;
defparam acc_w_fltr_01.MX_IN = 5 ;
//
parameter HF_CYCL = 5 ;
parameter CYCL = HF_CYCL * 2 ;
parameter TST_CYCL = 300 ;
//
reg clk, rst_n ;
reg signed [IN_BW-1:0] in_a ;
wire signed [OUT_BW-1:0] sig_acc ;
wire ovf_flg, ovr_mx ;
integer k ;
//
```

```
// module connection
acc_w_fltr   acc_w_fltr_01 ( .clk(clk), .rst_n(rst_n),
                             .in_a(in_a), .sig_acc(sig_acc),
                             .ovr_mx(ovr_mx), .ovf_flg(ovf_flg)  ) ;
//
always begin
  clk=0 ; #HF_CYCL ;
  clk=1 ; #HF_CYCL ;
end
//
initial begin
  rst_n = 1'b0 ;
  #CYCL rst_n = 1'b1 ;
  #(CYCL*29) rst_n = 1'b0 ;
  #CYCL rst_n = 1'b1 ;
  #(CYCL * TST_CYCL ) $finish ;
end
//
```

```
// give random numbers for in_a
initial begin
  #CYCL ;
  for ( k=0 ; k <= TST_CYCL+30 ; k = k+1 ) begin
     in_a = $random ;
     #CYCL ;
  end
end
//
// observe signals at clock rise time
always @ ( posedge clk ) begin
  $strobe
  ("t=%d, r=%b, in_a=%d, acc=%d, ovr_mx=%b, ovf=%b",
   $stime, rst_n, in_a, sig_acc, ovr_mx, ovf_flg ) ;
end

endmodule
`include "acc_w_fltr.v"
```

32-bit random number is created.

file name: test_acc_w_fltr.v

Now, run this file on your PC and see the result.

parameter OUT_BW = 8 ;

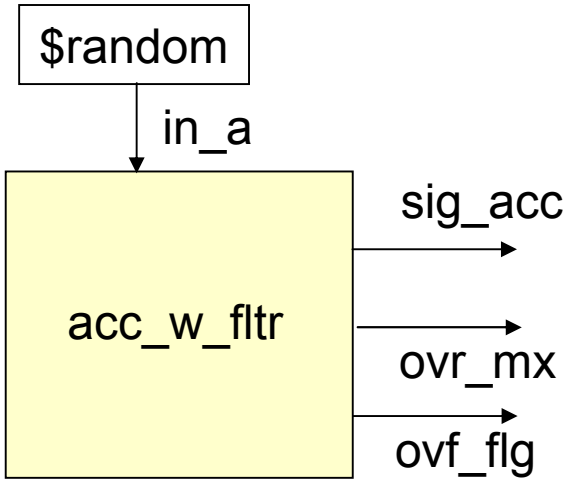
A part of a sample result

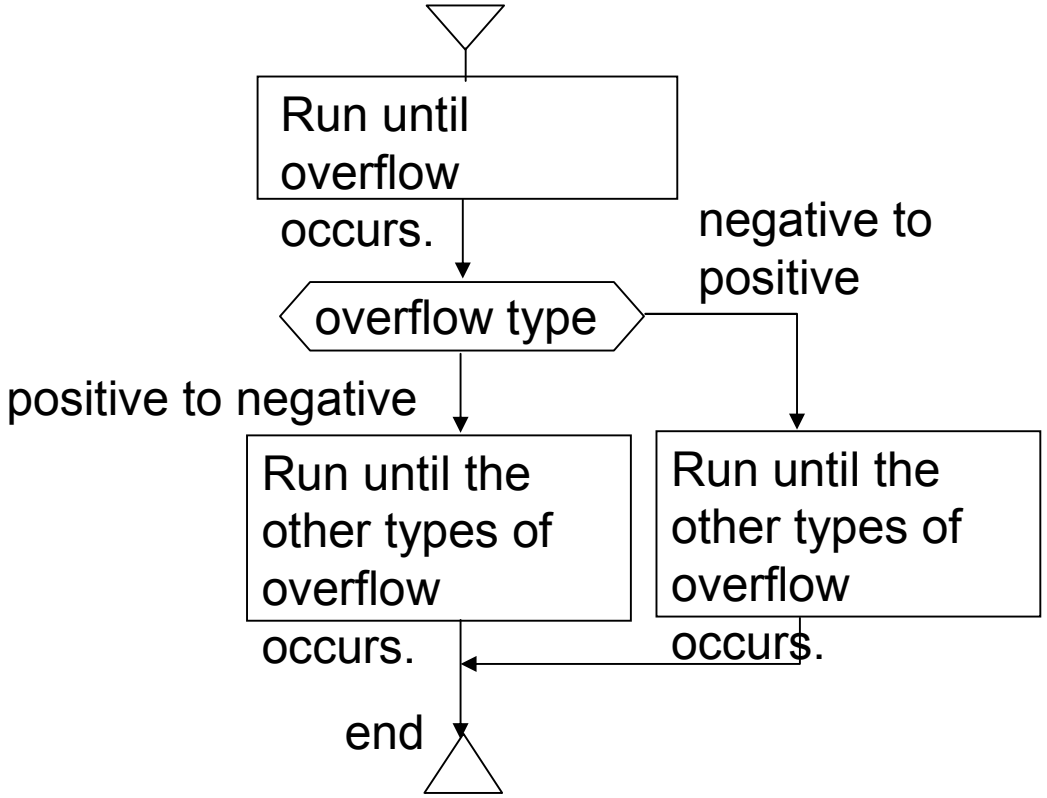
```
t=      515, r=1, in_a=  2, acc= -1, ovr_mx=0, ovf=0
t=      525, r=1, in_a= -6, acc= -1, ovr_mx=1, ovf=0
t=      535, r=1, in_a=  1, acc=  0, ovr_mx=0, ovf=0
t=      545, r=1, in_a= -8, acc=  0, ovr_mx=1, ovf=0
t=      555, r=1, in_a= -8, acc=  0, ovr_mx=1, ovf=0
t=      565, r=1, in_a= -7, acc=  0, ovr_mx=1, ovf=0
t=      575, r=1, in_a= -5, acc= -5, ovr_mx=0, ovf=0
t=      585, r=1, in_a=  6, acc= -5, ovr_mx=1, ovf=0
t=      595, r=1, in_a=  6, acc= -5, ovr_mx=1, ovf=0
t=      605, r=1, in_a= -2, acc= -7, ovr_mx=0, ovf=0
t=      615, r=1, in_a= -4, acc=-11, ovr_mx=0, ovf=0
t=      625, r=1, in_a= -6, acc=-11, ovr_mx=1, ovf=1
t=      635, r=1, in_a= -5, acc=-16, ovr_mx=0, ovf=0
```

-11 + (-6) = -17, it can not be held in 5-bit variable, therefore overflow detected.

To make overflow easily occur, OUT_BW is changed to 5 instead of 8.

Ex. 4-5. Automatic test bench: Write a test bench which can automatically test the module acc_w_fltr written in Ex. 4-4. The test bench must check, at each clock rise time, sig_acc, ovr_mx, and ovf_flg are calculated correctly.





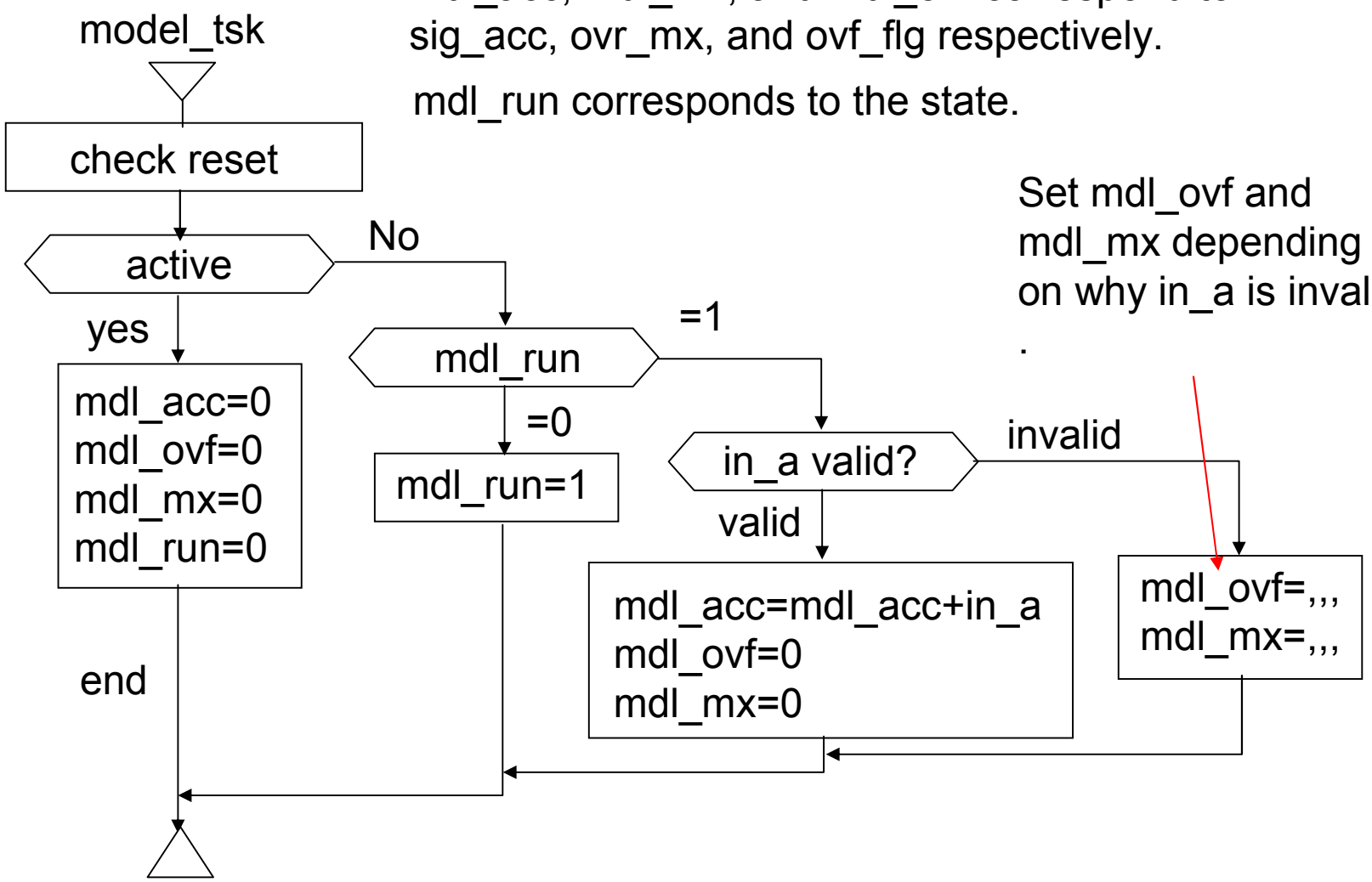
ovr_mx can easily become 1, therefore the test bench focuses on overflow. There are two types of overflow, large positive number becomes negative and small negative number becomes positive. The test bench runs until both types of overflow occur.

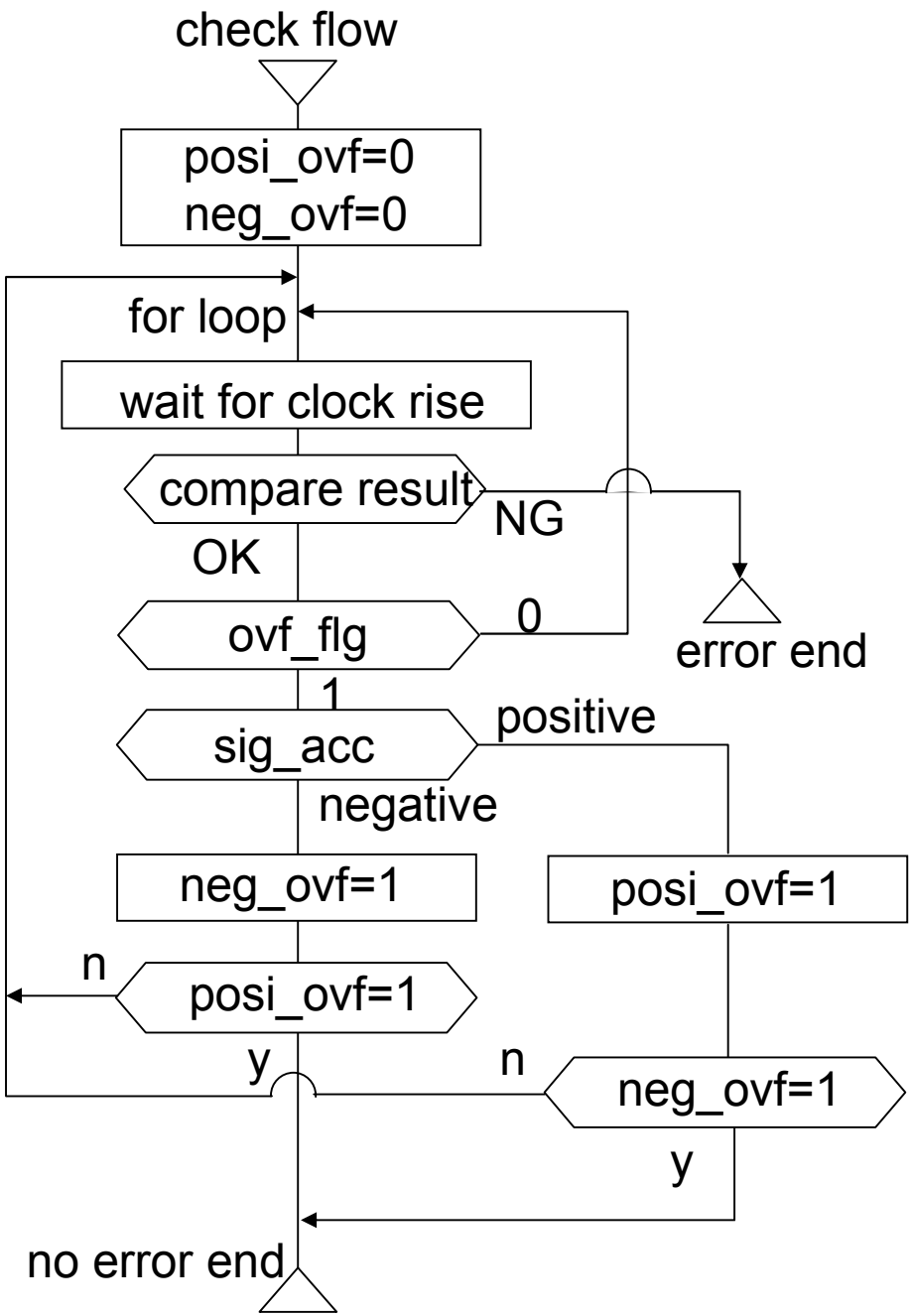
First, let's write a task as a golden model of acc_w_fltr which is invoked at each clock rise time and behaves same to the target module.

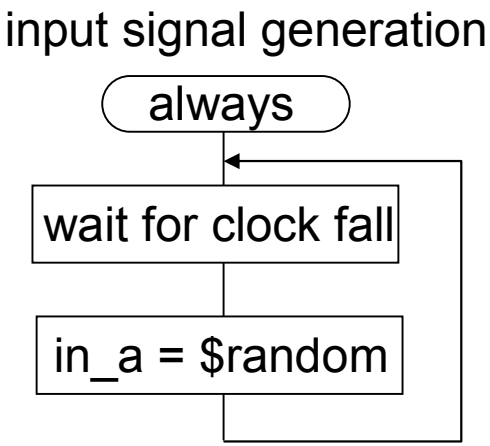
mdl_acc, mdl_mx, and mdl_ovf correspond to sig_acc, ovr_mx, and ovf_flg respectively.

mdl_run corresponds to the state.

Set mdl_ovf and mdl_mx depending on why in_a is invalid .



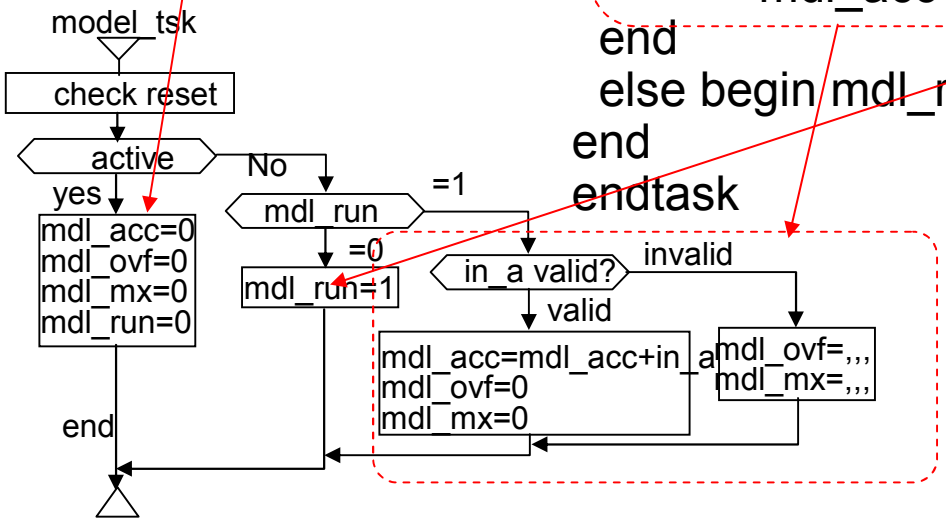




This is a procedure to check the module acc_w_fltr. First, input is accumulated until overflow occurs. and input is accumulated again until another type of overflow occurs.

golden model

```
task model_tsk;
reg signed [OUT_BW-1:0]  next_acc ;
begin
if( rst_n == 1'b0 ) begin
  mdl_acc = 0 ;
  mdl_ovf = 0 ;
  mdl_mx = 0 ;
  mdl_run = 0 ;
end
else begin
  if ( mdl_run ) begin
  next_acc = mdl_acc + in_a ;
  mdl_ovf =
       ( (in_a >= 0) & ( mdl_acc >= 0 ) & ( next_acc < 0) )  |
       ( ( in_a < 0 ) & ( mdl_acc < 0 ) & (next_acc >= 0 ) ) ;
  if ( mdl_run ) begin
    mdl_mx = ( ( in_a > MX_IN ) | ( in_a < -MX_IN) ) ?
             1 : 0;
    mdl_acc = (mdl_ovf | mdl_mx )?
             mdl_acc : mdl_acc + in_a ;
  end
  else begin mdl_run = 1 ; end
end
endtask
```



You can see golden model is much simpler than the target module. In the model, we don't have to care about hardware resources such as flip-flops.

RTL programming rules

(1) For y = a *op* b ; , where *op* is a verilog operator, bit size of a and b are extended to the bit size or y.
This is true for the case of a >= b. a and b are sized to Max(bit size of a, bit size of b). See an example below for detail.

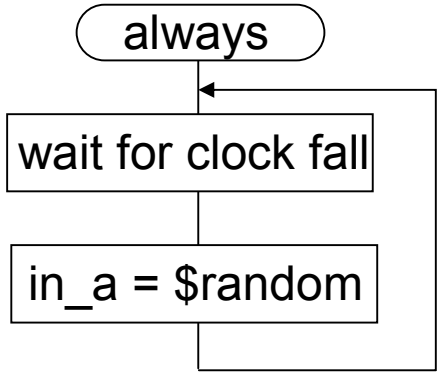
```
wire signed [7:0] a, b ;
wire signed [7:0] c ;
reg flg1, flg2 ;

assign c = a + b ;

flg1 = ( c > 0 )? 1: 0 ;

flg2 = ( a + b > 0 )? 1: 0 ;
```

By this assignment, c will be 36 of which bit pattern is 0010_0100, if a is -106 of wich bit pattern is 1001_0110, b is -114 of which bit pattern is 1000_1110. This means overflow occurred in calculating c.

Therefore, this may become true if a and b are negative and overflow occurs.

However, this will never becomes true, even if overflow occurs for 8-bit negative a and b. Because 0 means 32-bit value, a + b is calculated after a and b are size-extended to 32-bit.

test input generation

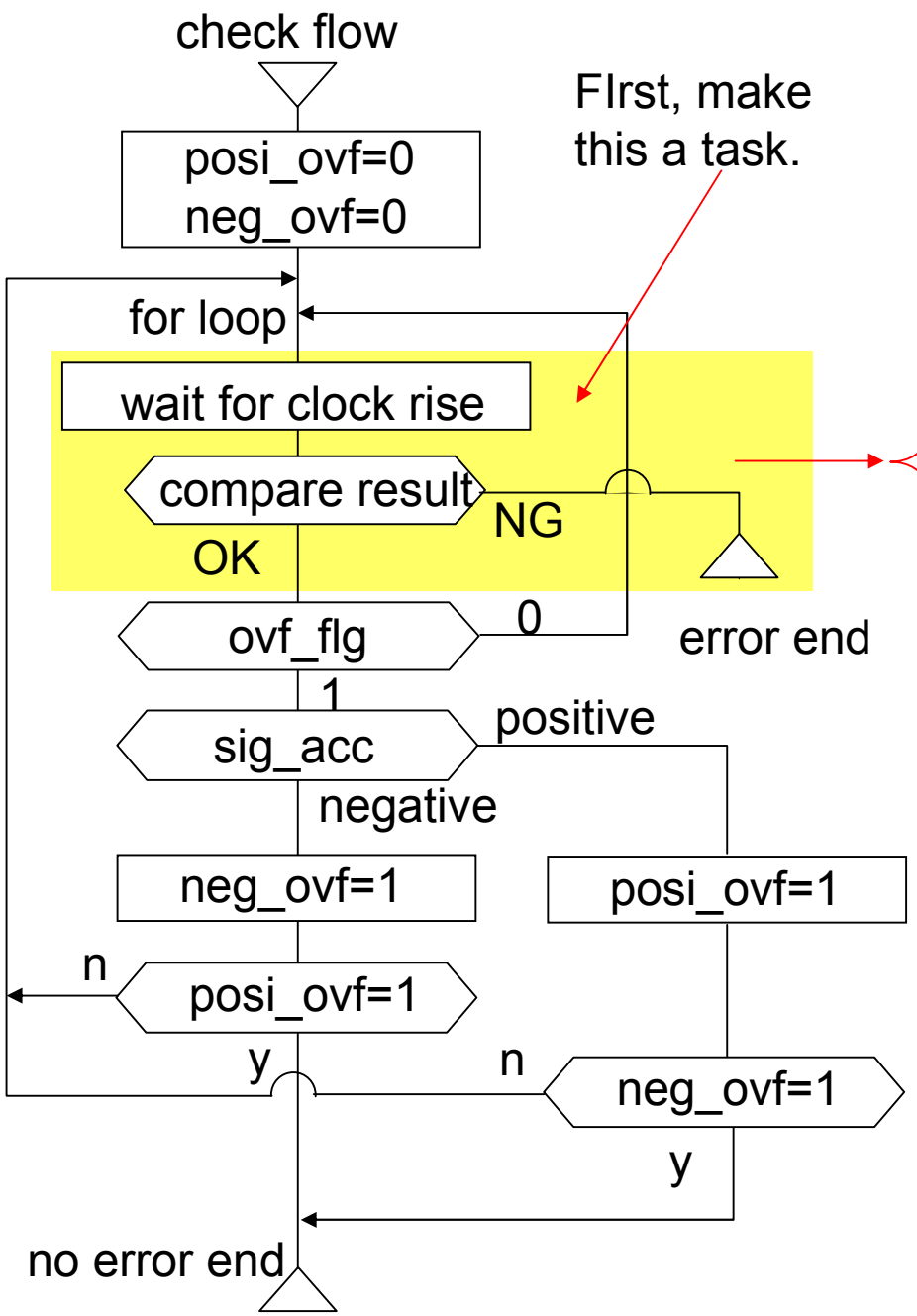


To make in_a stable at clock rise time, negedge of the clock is used.

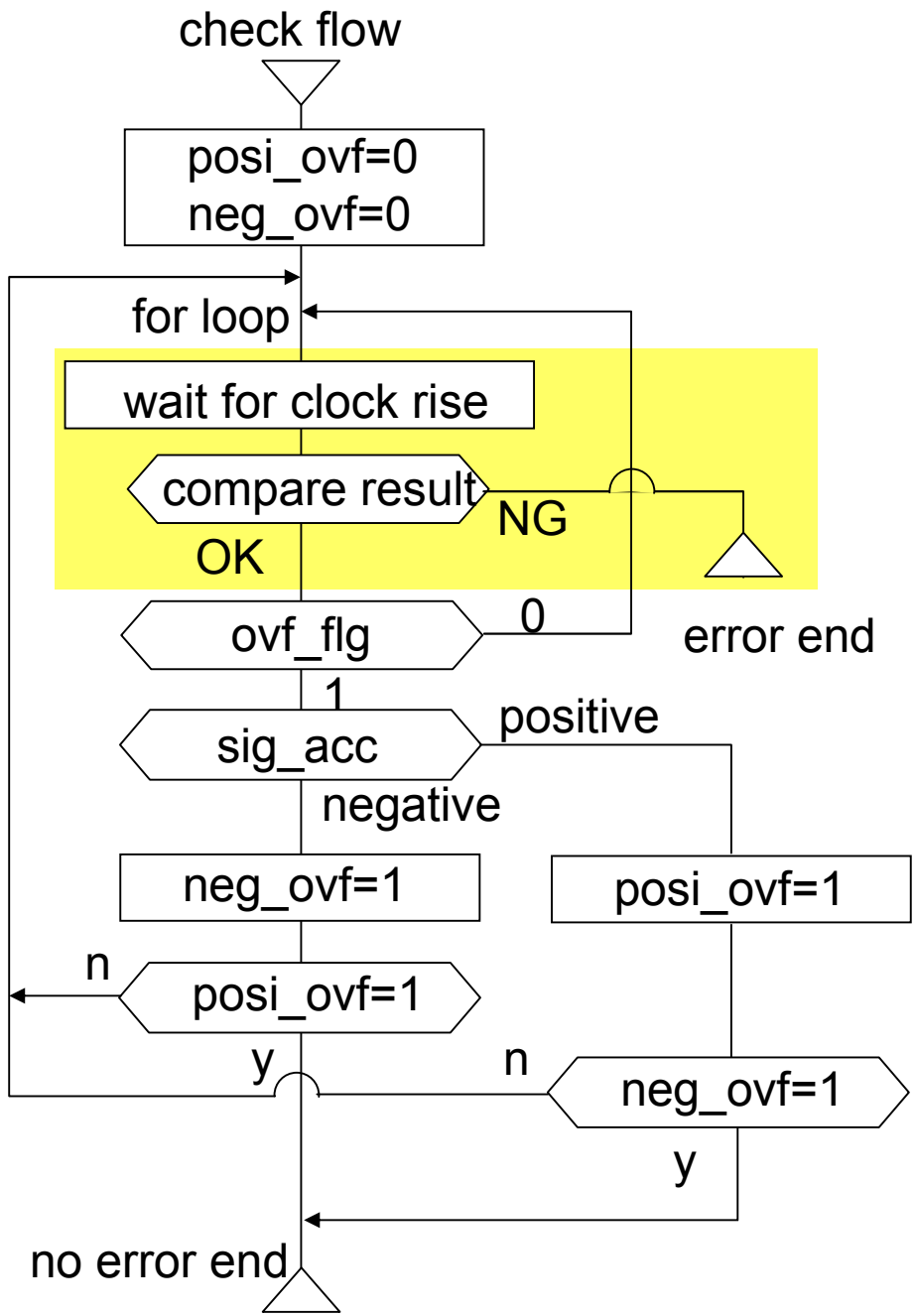
```
always begin   @ ( negedge clk ) ;
   in_a = ($random % 8)  ; //  in_a is from -7 to 7
end
```

modulus operator

For testing 8-bit sig_acc and 4-bit in_a, random number may not be a good idea to apply. Because it will need many simulation cycles to make overflow of sig_acc.
To make overflow of 8-bit sig_acc by adding 4-bit in_a, apply the same sign values until overflow occurs.




```
task checker ;
begin
@(posedge clk ) begin
if ( (sig_acc != mdl_acc ) begin
   $display
       ("error in acc, model=%d, tgt=%d",
                 mdl_acc, sig_acc ) ;
   #5 $finish ;
end
if ( (ovf_flg != mdl_ovf ) begin
   $display
      ("error in ovf_flg, model=%b, tgt=%b",
                 mdl_ovf, ovf_flg ) ;
   #5 $finish ;
end
if ( (ovr_mx != mdl_mx ) begin
   $display
      ("error ovr_mx, model=%b, tgt=%b",
                 mdl_mx, ovr_mx ) ;
   #5 $finish ;
end
end
endtask
```




```
initial begin
reg posi_ovf, neg_ovf ;
integer k;
posi_ovf = 0 ;
neg_ovf = 0 ;
for ( k = 0;  k <= MX_T; k=k+1 ) begin
   checker ;  //compare target and golden
   if ( ovf_flg ) begin
     if ( sig_acc > 0 )  begin
        posi_ovf = 1 ;
     end
     else begin
        neg_ovf = 1 ;
     end
   end
   if ( posi_ovf & neg_ovf ) begin
          $display ( "test OK" ) ;
          #5 $finish ;
   end
end
$display("test not complete in %d cycle",
              MX_T );
end
```

```
module test_auto_acc_w_fltr ;
parameter HF_CYCL = 5 ;
parameter CYCL = HF_CYCL * 2 ;
parameter MX_T = 500 ;  // max test cycle time
//
parameter IN_BW = 4 ;
defparam acc_w_fltr_01.IN_BW = IN_BW ;
parameter OUT_BW = 5 ;
defparam acc_w_fltr_01.OUT_BW = OUT_BW ;
parameter MX_IN = 5 ;
defparam acc_w_fltr_01.MX_IN = MX_IN ;
//
reg clk, rst_n ;
reg signed [IN_BW-1:0] in_a ;
wire signed [OUT_BW-1:0] sig_acc ;
wire ovr_mx, ovf_flg ;
//
reg signed [OUT_BW-1:0] mdl_acc ; // output of model
reg mdl_ovf, mdl_mx, mdl_run ;   // output of model
//
reg posi_ovf, neg_ovf ; // indicator to show what kind of overflow occurred
integer k;  // loop counter
//
```

Using smaller bit width 5 for sig_acc than the specification of Ex. 4-4.

```
initial begin // main root to check result
posi_ovf = 0 ;  // first no overflow
neg_ovf = 0 ;  // first no overflow
for ( k = 0;  k <= MX_T; k=k+1 ) begin  // loop to test
   @(posedge clk ) begin
   model_tsk ; // invoke the golden model
   #1 checker ;  //compare target and golden
   if ( ovf_flg ) begin
     if ( sig_acc > 0 )  begin
        posi_ovf = 1 ;  // set positive to negative overflow
     end
     else begin
        neg_ovf = 1 ;  // set negative to positive overflow
     end
   end
   if ( posi_ovf & neg_ovf ) begin   // two types of overflow occurred
          $display ( "test OK" ) ;
          #5 $finish ;
   end
end
end
$display("test not complete in %d cycle",   // only one or no overflow occurred
             MX_T );
end
```

```
task checker ;  // compare simulation result with golden model
begin
if ( sig_acc != mdl_acc ) begin
   $display
       ("error in acc, model=%d, tgt=%d",
                 mdl_acc, sig_acc ) ;
   #5 $finish ;
end
if ( ovf_flg != mdl_ovf ) begin
   $display
      ("error in ovf_flg, model=%b, tgt=%b",
                 mdl_ovf, ovf_flg ) ;
   #5 $finish ;
end
if ( ovr_mx != mdl_mx ) begin
   $display
      ("error ovr_mx, model=%b, tgt=%b",
                 mdl_mx, ovr_mx ) ;
   #5 $finish ;
end
end
endtask
```

```
task model_tsk; // golden model of the target module
reg signed [OUT_BW-1:0]  next_acc ;
begin
if( rst_n == 1'b0 ) begin
   mdl_acc = 0 ;
   mdl_ovf = 0 ;
   mdl_mx = 0 ;
   mdl_run = 0 ;
end
else begin
  next_acc = mdl_acc + in_a ;
  mdl_ovf = ( (in_a >= 0) & ( mdl_acc >= 0 ) & ( next_acc < 0) )  |
                ( ( in_a < 0 ) & ( mdl_acc < 0 ) & ( next_acc >= 0 ) ) ;
  if ( mdl_run ) begin
     mdl_mx = ( ( in_a > MX_IN ) | ( in_a < -MX_IN) ) ?
              1 : 0;
     mdl_acc = (mdl_ovf | mdl_mx )?  mdl_acc : mdl_acc + in_a ;
  end
  else begin mdl_run = 1 ; end
end
// $strobe("t=%d, run=%b, in_a=%d, acc=%d, mx=%b, ovf=%b",
//      $stime, mdl_run, in_a, mdl_acc, mdl_mx, mdl_ovf );  // for debug
end
endtask
```

Important

If we use (mdl_acc+in_a) < 0 instead of next_acc < 0, overflow is not detected correctly. Because 0 is 32-bit, therefore mdl_acc+ in_a is calculated after both operand are extended to 32-bit. Overflow will not occur for such large bit size. We are adding at most 8-bit values.

```
always begin   @ ( negedge clk ) ;
   in_a = ($random % 8)  ; //  in_a is from -7 to 7
end
always begin
  clk = 0; #HF_CYCL;
  clk = 1; #HF_CYCL;
end
initial begin
 rst_n = 0 ;
#(CYCL*5) rst_n = 1 ;
end
// connection to target module
acc_w_fltr   acc_w_fltr_01 ( .clk(clk), .rst_n(rst_n),
                             .in_a(in_a), .sig_acc(sig_acc),
                             .ovr_mx(ovr_mx), .ovf_flg(ovf_flg) ) ;
//always @( posedge clk ) begin  // for debug
//$strobe("t=%d, rst=%b, in_a=%d, acc=%d, mx=%b, ovf=%b",
//       $stime, rst_n, in_a, sig_acc, ovr_mx, ovf_flg );
//end
endmodule
`include "acc_w_fltr.v"
```

file name: test_auto_acc_w_fltr.v

A part of a sample result:

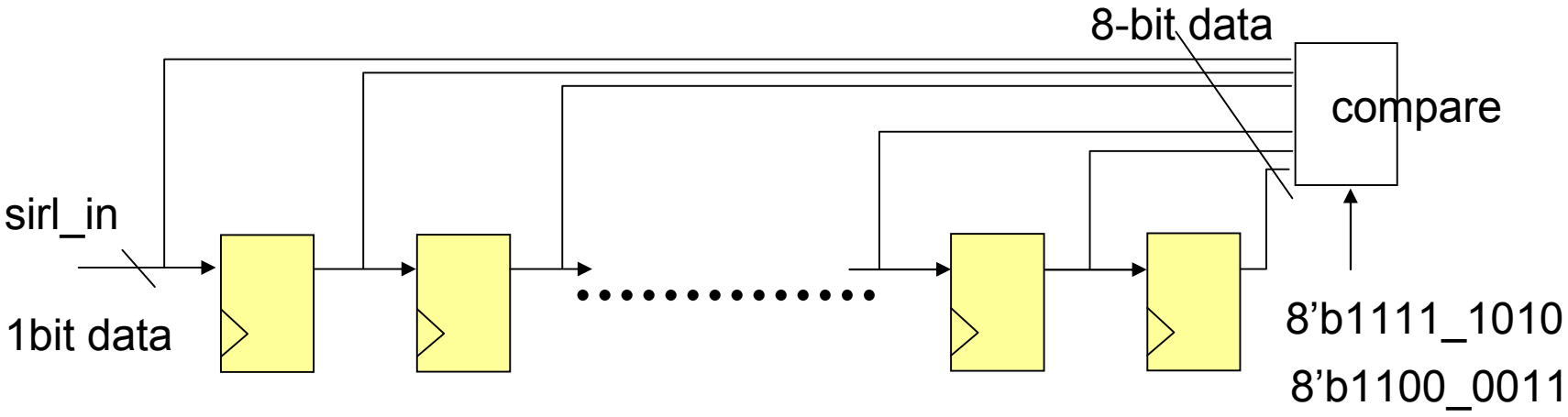
This simulation is done with $strobe system task active in model_tsk.

The golden model's output

The target model's output

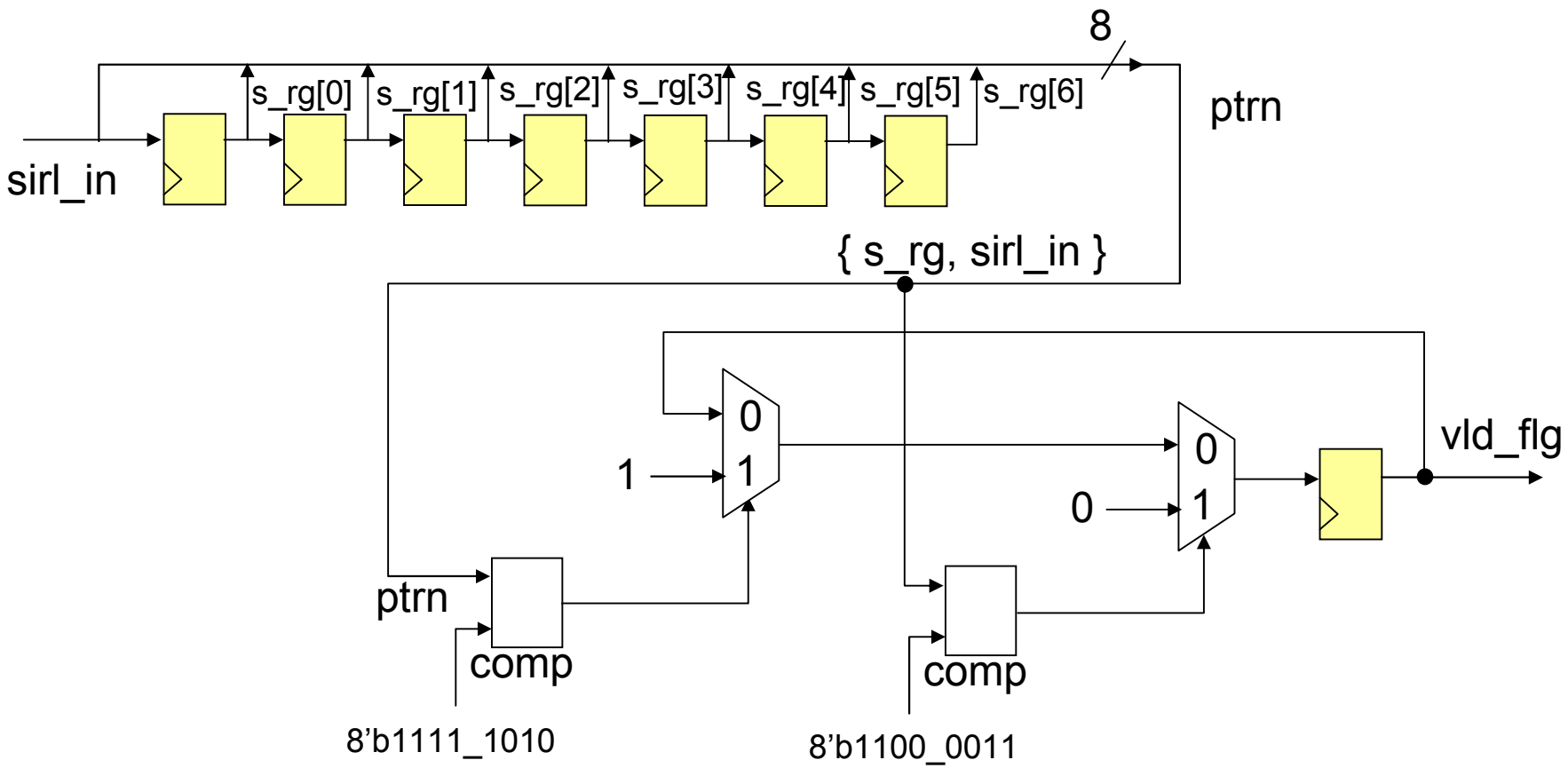
```
t=        185, run=1, in_a=  5, acc= 14, mx=0, ovf=0
t=        185, rst=1, in_a=  5, acc= 14, mx=0, ovf=0
t=        195, run=1, in_a= -1, acc= 13, mx=0, ovf=0
t=        195, rst=1, in_a= -1, acc= 13, mx=0, ovf=0
t=        205, run=1, in_a= -6, acc= 13, mx=1, ovf=0
t=        205, rst=1, in_a= -6, acc= 13, mx=1, ovf=0
t=        215, run=1, in_a=  7, acc= 13, mx=1, ovf=1
t=        215, rst=1, in_a=  7, acc= 13, mx=1, ovf=1
t=        225, run=1, in_a=  2, acc= 15, mx=0, ovf=0
t=        225, rst=1, in_a=  2, acc= 15, mx=0, ovf=0
t=        235, run=1, in_a= -2, acc= 13, mx=0, ovf=0
                 •
                 •
                 •
t=        395, rst=1, in_a= -5, acc=-14, mx=0, ovf=0
t=        405, run=1, in_a=  3, acc=-11, mx=0, ovf=0
t=        405, rst=1, in_a=  3, acc=-11, mx=0, ovf=0
t=        415, run=1, in_a= -3, acc=-14, mx=0, ovf=0
t=        415, rst=1, in_a= -3, acc=-14, mx=0, ovf=0
t=        425, run=1, in_a= -6, acc=-14, mx=1, ovf=1
t=        425, rst=1, in_a= -6, acc=-14, mx=1, ovf=1
test OK
Halted at location **test_auto_acc_w_fltr.v(48) time
```

Ex 4-6. Serial bit pattern detector: Write a module which receives 1-bit data sirl_in at every clock rise time and check if the received bit pattern becomes 8'b1111_1010 or 8'b1100_0011, MSB correspond to older bit and LSB corresponds to the newest bit received.
If 8'b1111_1010 received then place 1 to an 1-bit output signal vld_flg until 8'b1100_0011 received. If 8'b1100_0011 received then place 0 to vld_flg . Apply a synchronous reset signal rst to the module and reset vld_flg to 0 when rst asserted.

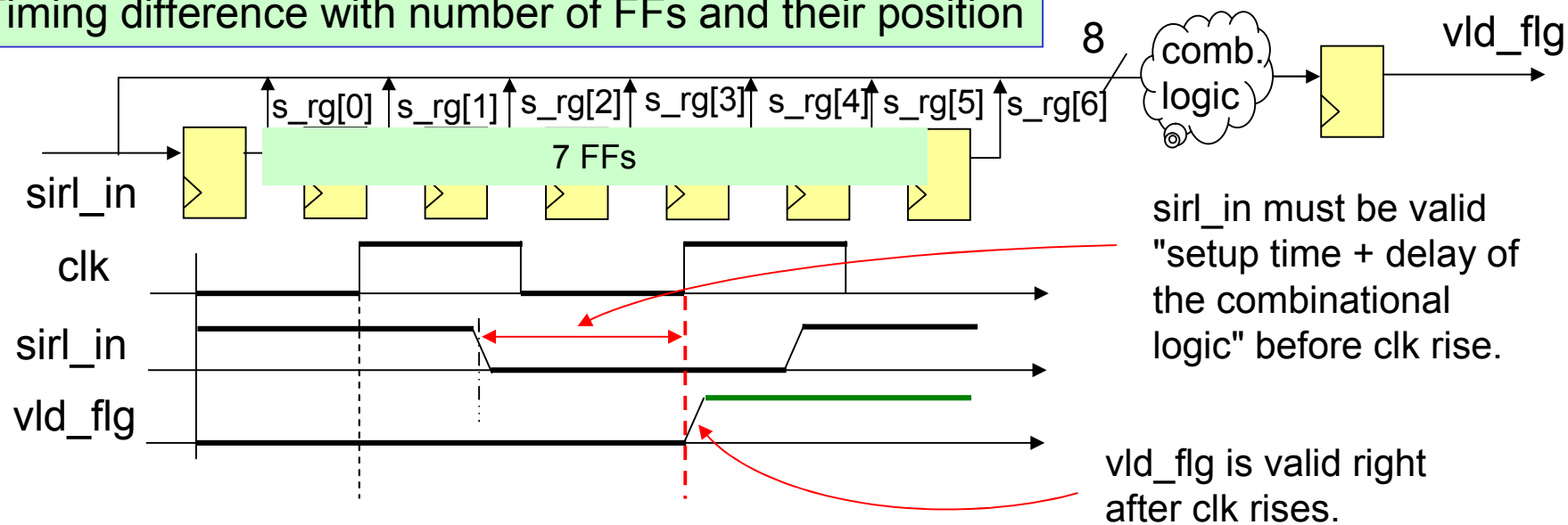


The module have to memorize 8 bits serial input bit string and compare it to 8'b1111_1010 and 8'b1100_0011. Use shift register to memorize serially given 8-bit data.

The whole module must looks like below.

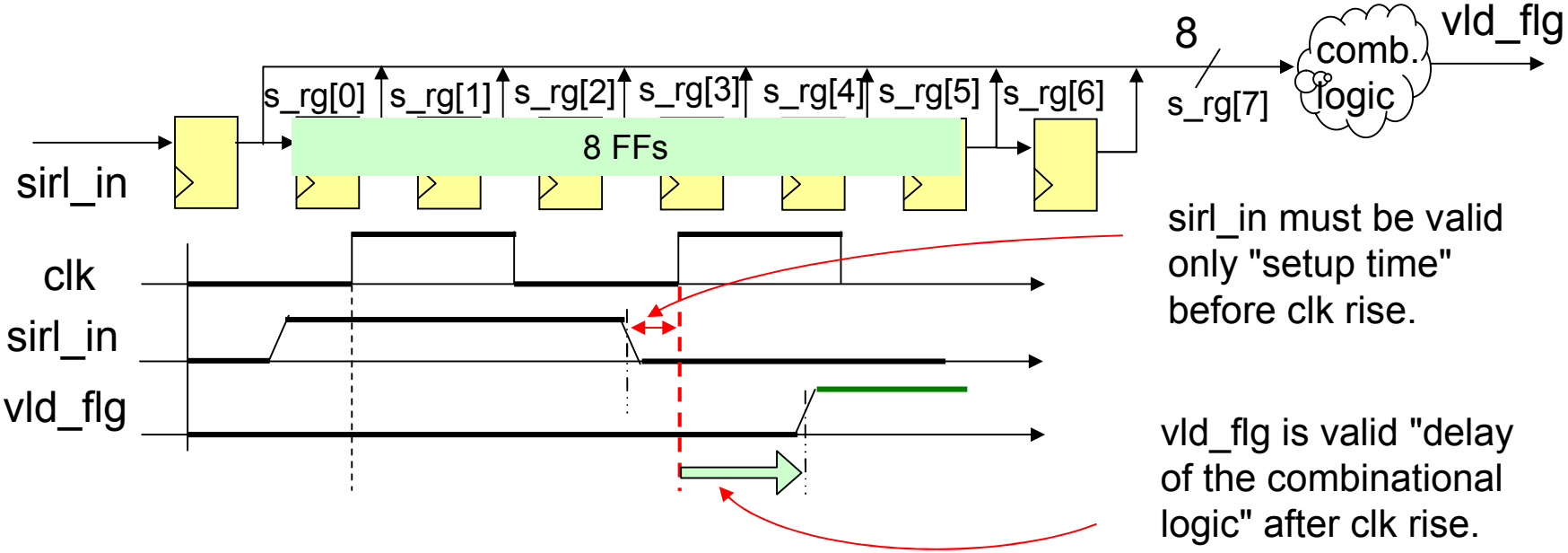


Timing difference with number of FFs and their position



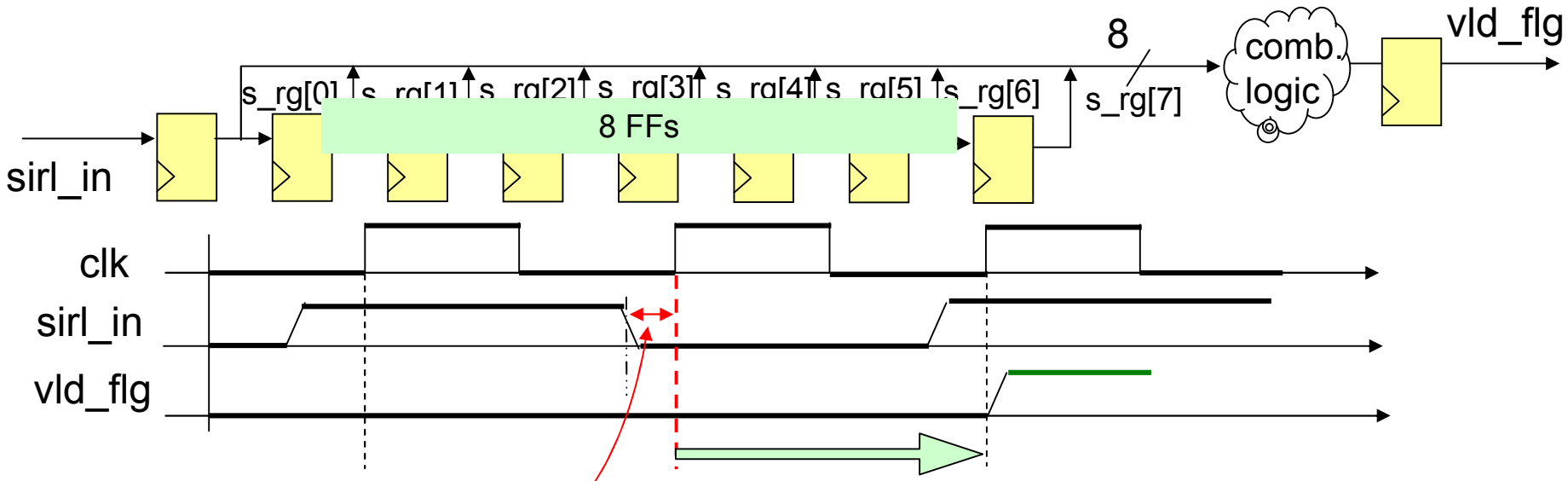
sirl_in must be valid "setup time + delay of the combinational logic" before clk rise.

vld_flg is valid right after clk rises.



sirl_in must be valid only "setup time" before clk rise.

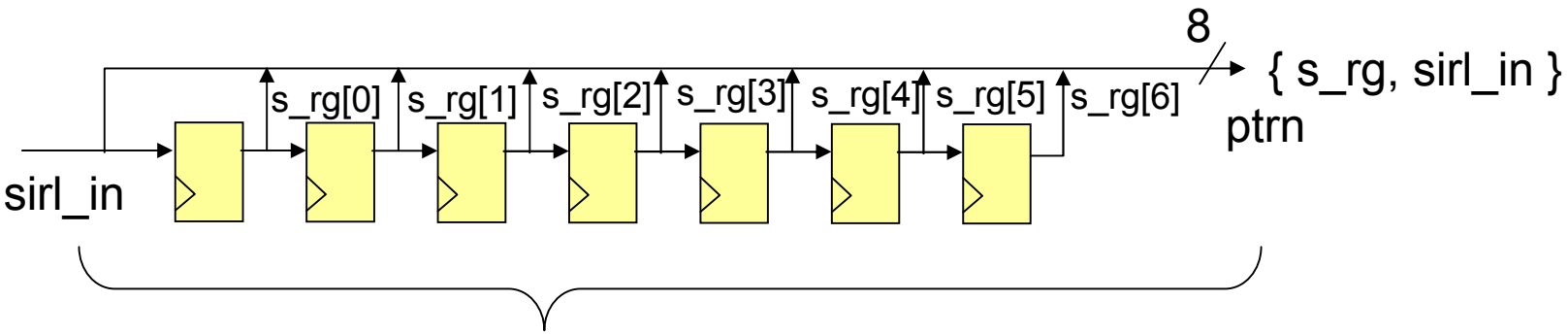
vld_flg is valid "delay of the combinational logic" after clk rise.



sirl_in must be valid only "setup time" before clk rise.

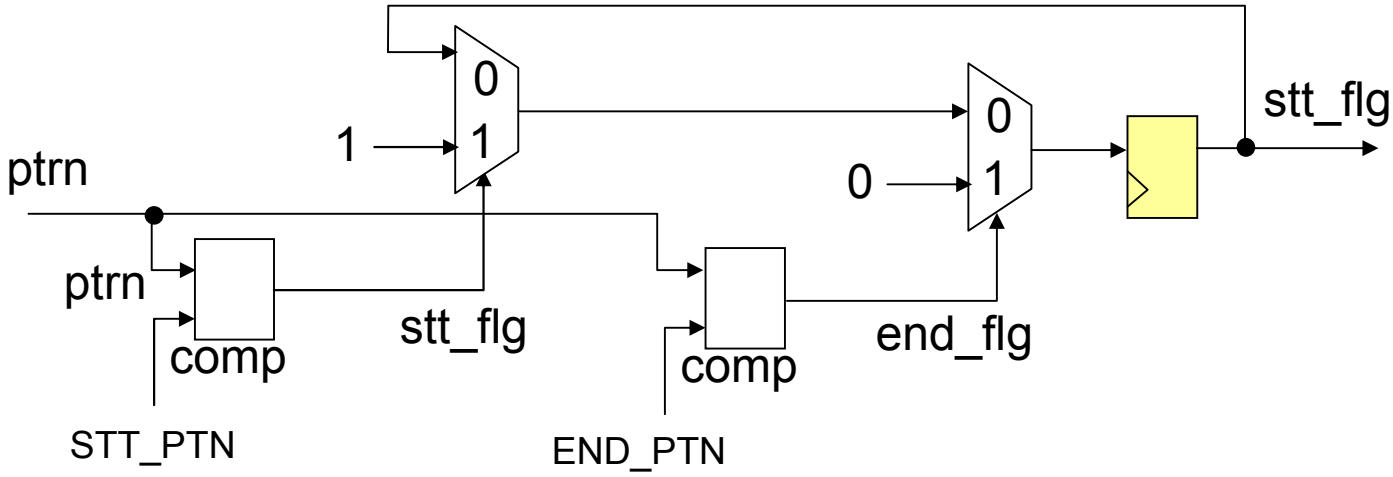
vld_flg is valid right after the next clk rise.

Shift register part can be written as below.




```
assign ptrn = { s_rg, sirl_in } ;

always @ ( posedge clk ) begin
  if ( rst == 1'b1 ) begin
    s_rg[6:0] <= 0 ;
  end
  else begin
    s_rg[0] <= sir_in ;
    s_rg[1] <= s_rg[0] ;
    s_rg[2] <= s_rg[1] ;
    s_rg[3] <= s_rg[2] ;
    s_rg[4] <= s_rg[3] ;
    s_rg[5] <= s_rg[4] ;
    s_rg[6] <= s_rg[5] ;
 end
end
```

Pattern checking and flag setting part can be written as below.




```
assign stt_flg = (ptrn == STT_PTN);
assign end_flg = (ptrn == END_PTN);

always @ ( ptrn or stt_flg ) begin
  if ( stt_flg ) begin
    next_vld_flg = 1 ;
  end
  else begin
     if ( end_flg ) begin
         next_vld_flg = 0 ;
     end
     else begin
         next_vld_flg = stt_flg ;
     end
 end
end
```

```
always @ ( posedge clk ) begin
  if ( rst == 1'b1 ) begin
    vld_flg <= 0 ;
  end
  else begin
    vld_flg <= next_vld_flg ;
 end
end
```

Coding example.

```
module bit_ptrn_chk ( clk, rst, sirl_in, vld_flg ) ;
parameter STT_PTN = 8'b1111_1010 ;  // hex fa
parameter END_PTN = 8'b1100_0011 ; // hex c3
input clk ;
input rst ;
input sirl_in ;
output vld_flg ;
wire clk ;
wire rst ;
wire sirl_in ;
reg vld_flg ; // FF
reg next_vld_flg ; // non-FF
wire [7:0] ptrn ;
reg [6:0] s_rg ; // FF, shift register
reg [6:0] next_s_rg ; // non-FF
wire stt_flg ;
wire end_flg ;

assign ptrn = { s_rg, sirl_in } ;
assign stt_flg = (ptrn == STT_PTN);
assign end_flg = (ptrn == END_PTN);
```

```
always @ ( ptrn or stt_flg ) begin
  if ( stt_flg ) begin
    next_vld_flg = 1'b1 ;
  end
  else begin
     if ( end_flg ) begin
        next_vld_flg = 1'b0 ;
     end
     else begin
        next_vld_flg = vld_flg ;
     end
 end
end
```

```
always @ ( posedge clk ) begin
  if ( rst == 1'b1 ) begin
    s_rg[6:0] <= 7'b0000_000 ;
  end
  else begin
    s_rg[0] <= sirl_in ;
    s_rg[1] <= s_rg[0] ;
    s_rg[2] <= s_rg[1] ;
    s_rg[3] <= s_rg[2] ;
    s_rg[4] <= s_rg[3] ;
    s_rg[5] <= s_rg[4] ;
    s_rg[6] <= s_rg[5] ;
 end
end
always @ ( posedge clk ) begin
  if ( rst == 1'b1 ) begin
    vld_flg <= 1'b0 ;
  end
  else begin
    vld_flg <= next_vld_flg ;
 end
end
endmodule
```

Coding example.

file name: bit_ptrn_chk.v

It is tedious to assign serial bit date by using initial construct shown below left. Therefore prepare parallel to serial conversion task and use it to generate 32-bit length serial data from 32-bit parallel data.

```
initial begin
            sirl_in = 1 ;
# CYCL sirl_in = 1 ;
# CYCL sirl_in = 0 ;
# CYCL sirl_in = 1 ;
# CYCL sirl_in = 0 ;
# CYCL sirl_in = 1 ;
# CYCL sirl_in = 1 ;
# CYCL sirl_in = 1 ;
# CYCL sirl_in = 0 ;
# CYCL sirl_in = 1 ;
# CYCL sirl_in = 1 ;
# CYCL sirl_in = 0 ;
# CYCL sirl_in = 1 ;
# CYCL sirl_in = 0 ;
,,,,,,
,,,
end
```

```
initial
    para_sirl( 32'b11010110110 10,,,, ) ;
    para_sirl( 32'b011100,,,, ) ;
    ,,,
end

task para_sirl  ;
input [31:0] para_in ;
integer k ;
begin
for ( k=0 ; k<32 ; k = k + 1 ) begin
    sirl_in = para_in[31-k] ;
    @ (negedge clk ) ;
  end
end
endtask
```

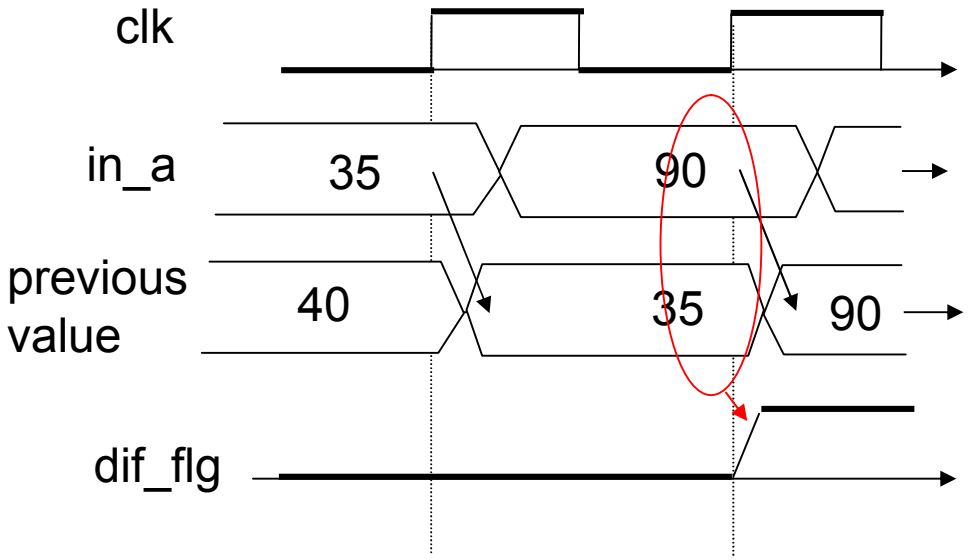
Coding example of a test bench.

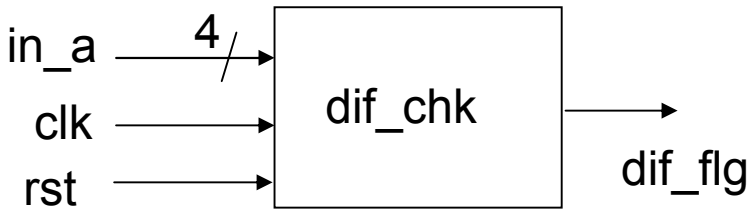
```
`timescale 1ns/100ps
module test_bit_ptrn_chk ;
parameter HF_CYCL = 50 ;
parameter CYCL= HF_CYCL * 2 ;
reg clk, rst ; // non-FF
reg sirl_in ; // non-FF
wire flg ;
bit_ptrn_chk  bit_ptrn_chk_01(
      .clk(clk), .rst(rst),
      .sirl_in(sirl_in), .vld_flg(flg) ) ;
initial begin
 rst = 1'b1 ;
#(CYCL*2) rst = 1'b0 ;
#(CYCL*12) rst = 1'b1 ;
#CYCL rst = 1'b0 ;
#(CYCL*60) rst = 1'b1 ;
#CYCL rst = 1'b0 ;
#(CYCL*10) $finish ;
end
always begin
  clk = 0 ; #HF_CYCL ;
  clk = 1 ; #HF_CYCL ;
end
```

```
initial begin
  para_sirl (32'hffaf_fa0c ) ;
  para_sirl (32'h3faf_ac3f ) ;
  para_sirl (32'hc3af_ac33 ) ;
end

task para_sirl  ;
input [31:0] para_in ;
integer k ;
begin
for ( k=0 ; k<32 ; k = k + 1 ) begin
    sirl_in = para_in[31-k] ;
    @ (negedge clk ) ;
  end
end
endtask
always @ ( posedge clk ) begin
   $strobe(
     "t=%d,rst=%b, sirl_in=%d, flg=%b",
          $stime, rst, sirl_in, flg ) ;
end
endmodule
```

file name: test_bit_ptrn_chk.v

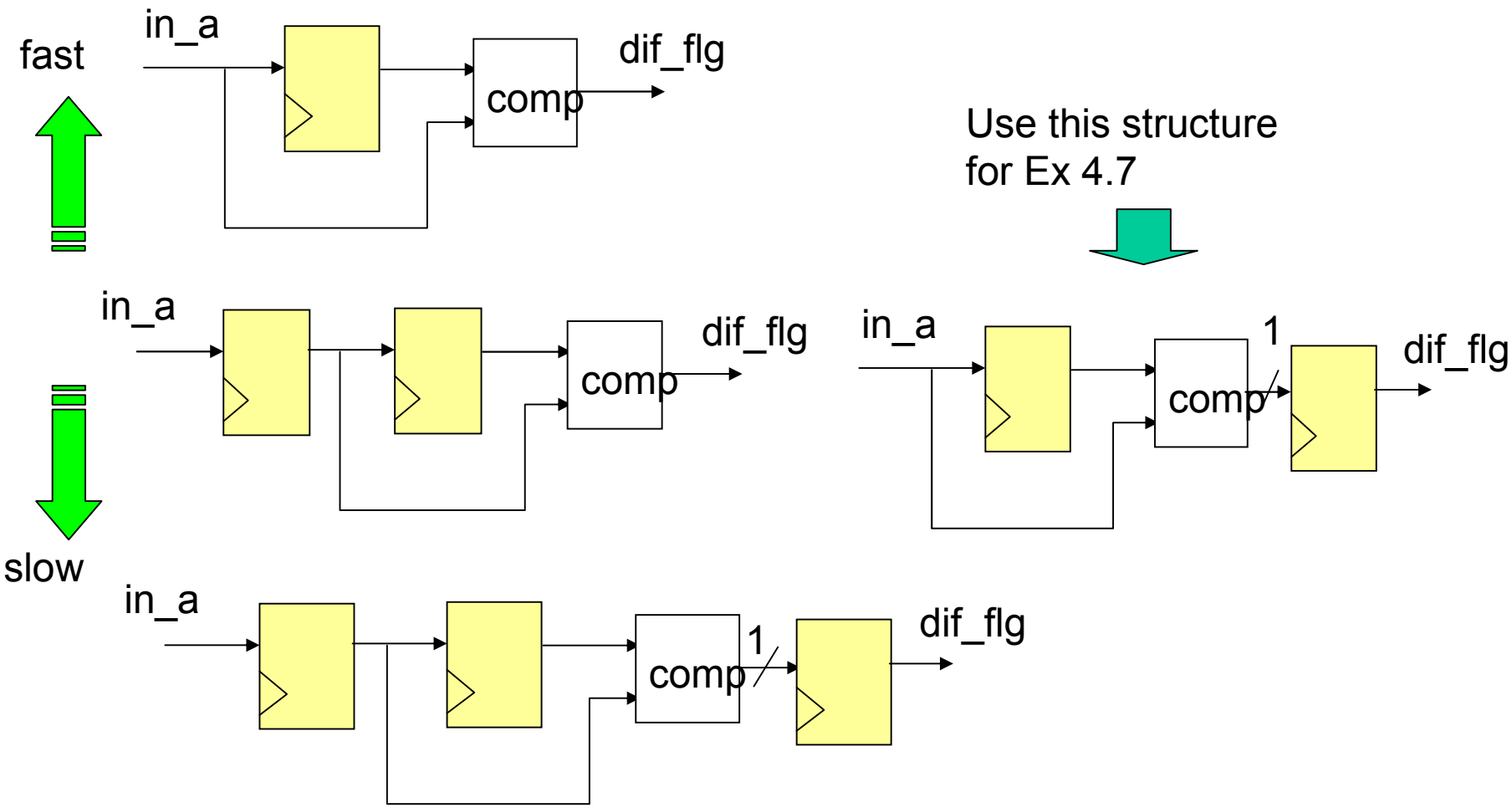
Ex 4-7. Data change range checker: Write a module which receives 8-bit input in_a at every clock rise time and if a received data is different from the previously received data more than 50, then output 1 on a 1-bit output signal dif_flg, else output 0. When only one data is received after reset asserted, then assume 0 was received previously.
Assume in_a is signed and use Verilog2001 specification.
Apply an asynchronous reset rst_n, active low, to initialize the module. dif_flg must be set 0 on reset.The output dif_flg must come out directly from a flip-flop.



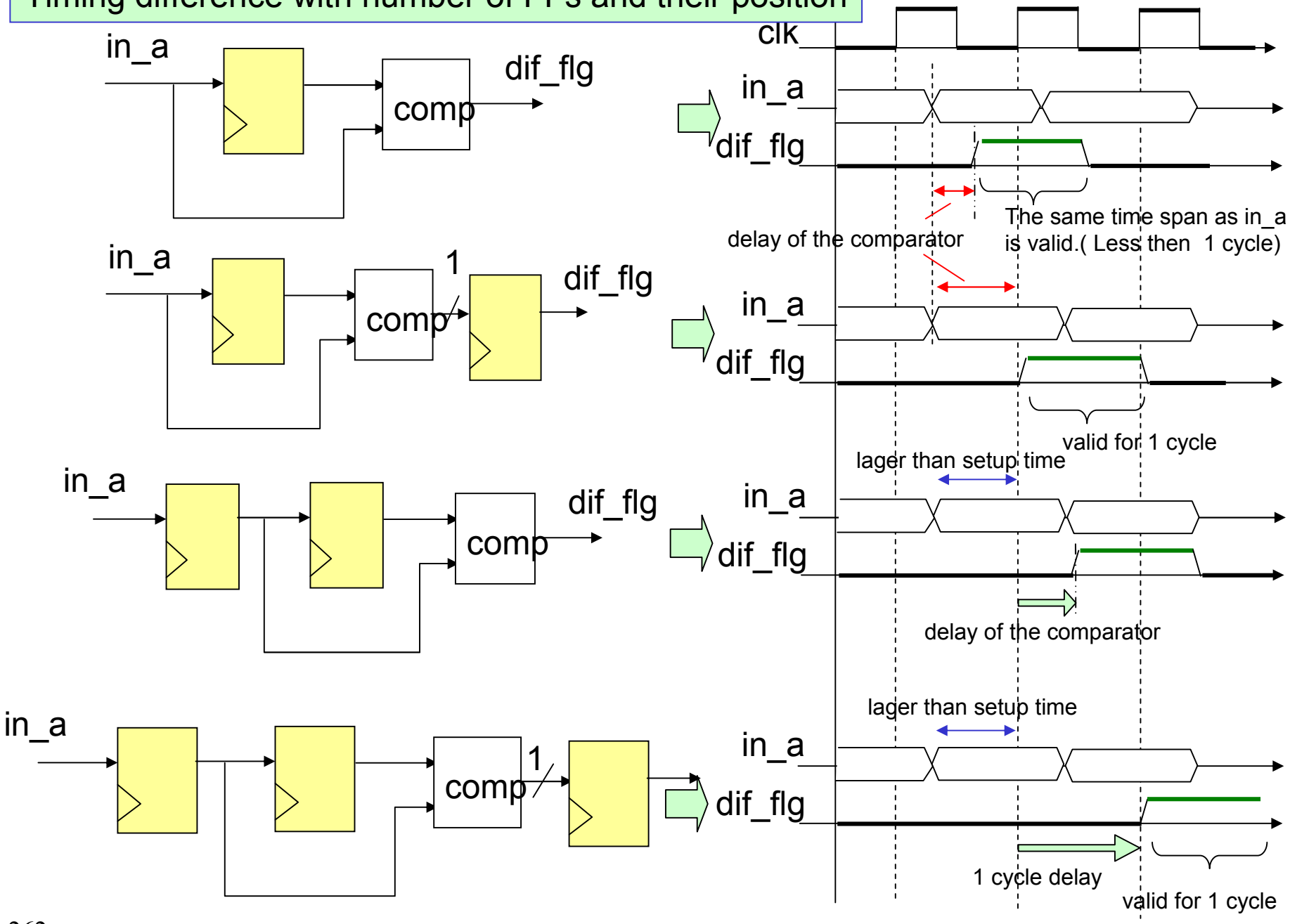


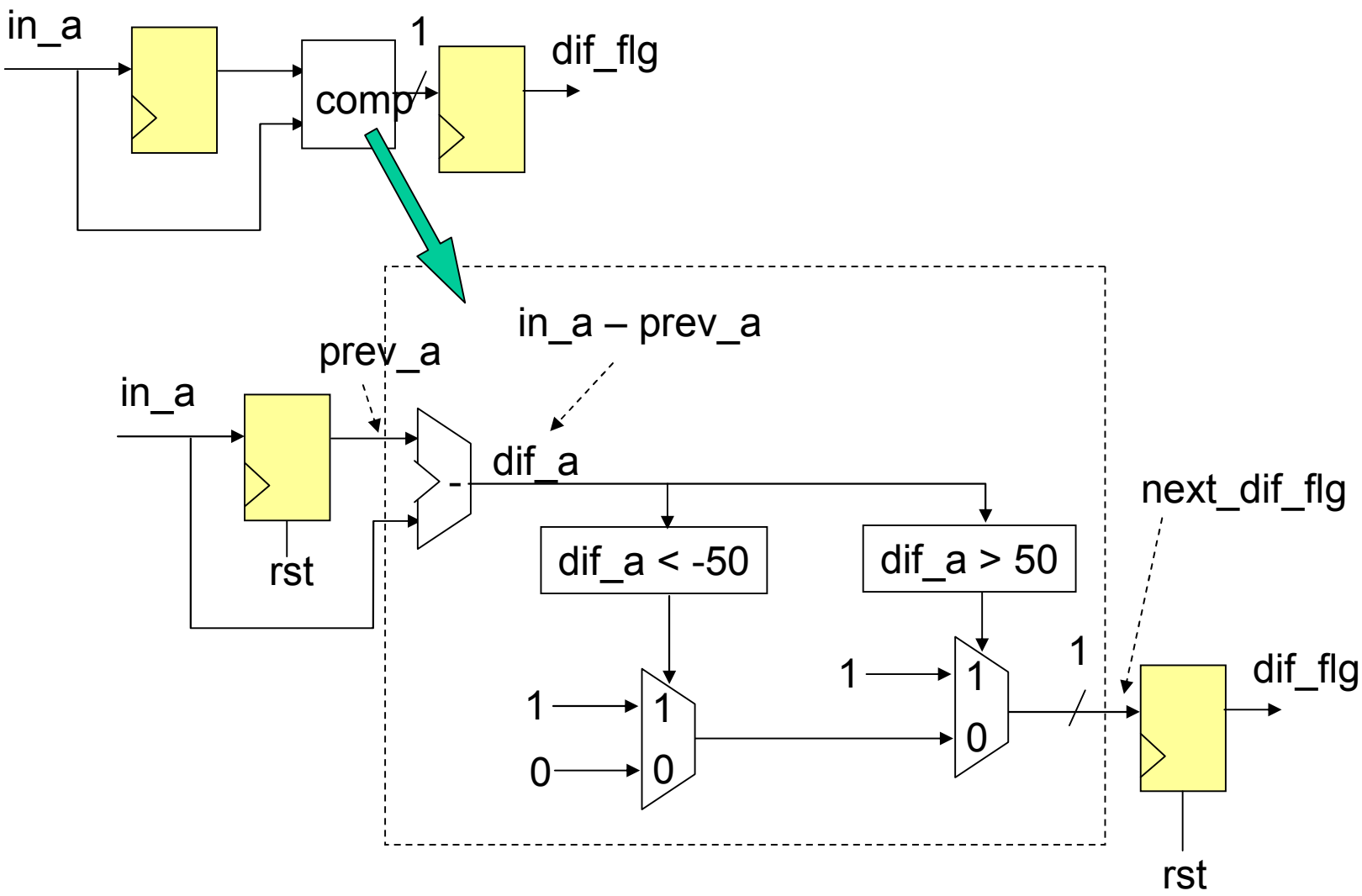
To compare current input with the previous input, we must memorize it in an 8-bit flip-flop. To handle signed signal, we have to use “signed ” keyword introduced in Verilog2001.

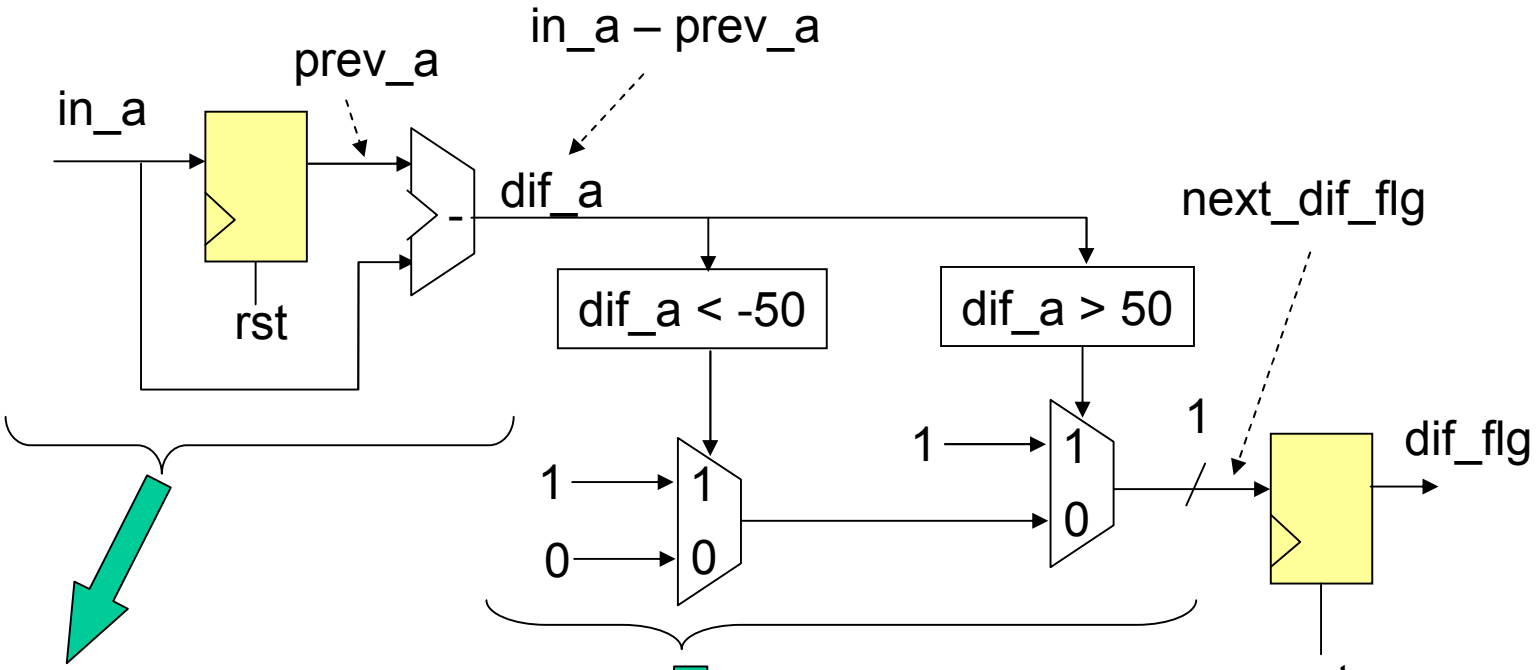
The target logic can be structured in a several ways as shown below if there is no specification about how to use FFs. This exercise says that we have to use the diagram shown on the middle right.
The configurations below have basically similar functionalities, but each of them has different timing issues. In general adding one FF result in one cycle delay.



Timing difference with number of FFs and their position








```
always @ ( posedge clk or
           posedge rst) begin
  if ( rst == 1'b1 ) begin
     prev_a <= 0 ;
  end
  else begin
     prev_a <= next_prev_a ;
  end
end
assign next_prev_a = in_a ;
assign dif_a = in_a – prev_a ;
```

```
always @ ( dif_a ) begin
if ( dif_a > LIMIT ) begin
    next_dif_flg = 1'b1 ;
end
else begin
   if ( dif_a < -LIMIT) begin
      next_dif_flg = 1'b1 ;
   end
   else begin
      next_dif_flg = 1'b0 ;
   end
end
end
```

```
always @ ( posedge clk
     or posedge rst) begin
  if ( rst == 1'b1 ) begin
     dif_flg <= 0 ;
  end
  else begin
     dif_flg <= next_dif_flg ;
  end
end
```

Coding example.

```
module dif_chk( clk, rst, in_a, dif_flg ) ;
parameter A_BW=8 ;
parameter DIF_LMT = 50 ;
input clk ;
input rst ;
input signed [A_BW-1:0] in_a ;
output dif_flg ;
wire clk ;
wire rst ;
wire signed [A_BW-1:0] in_a ;
reg dif_flg ;  //FF

reg next_dif_flg ; // non-FF
reg signed [A_BW-1:0] prev_a ; // FF
wire signed [A_BW-1:0] next_prev_a ;
wire signed [A_BW-1:0] dif_a ;

always @ ( posedge clk or
                posedge rst) begin
  if ( rst == 1'b1 ) begin
    prev_a <= {A_BW{1'b0}} ;
  end
  else begin
    prev_a <= next_prev_a ;
  end
end
```

```
assign next_prev_a = in_a ;
assign dif_a = in_a - prev_a ;

always @ ( posedge clk or
                posedge rst) begin
  if ( rst == 1'b1 ) begin
    dif_flg <= 1'b0 ;
  end
  else begin
    dif_flg <= next_dif_flg ;
  end
end
always @ ( dif_a ) begin
if ( dif_a > DIF_LMT ) begin
   next_dif_flg = 1'b1 ;
end
else begin
  next_dif_flg =( dif_a < -DIF_LMT)?
                    1'b1 : 1'b0 ;
end
end
endmodule
```

file name: dif_chk.v

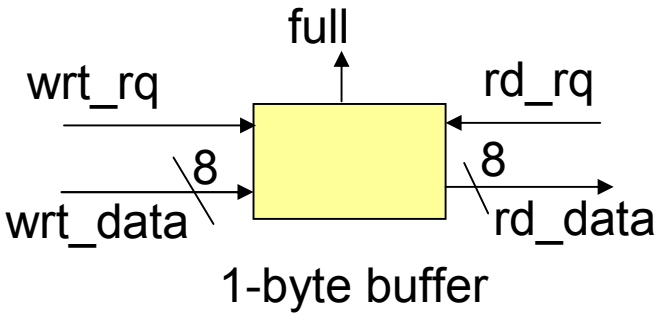
Coding example of a test bench.

```
module test_dif_chk ;
parameter A_BW=8 ;
parameter HF_CYCL = 50 ;
parameter CYCL= HF_CYCL * 2 ;
reg clk, rst ; // non-FF
reg signed [A_BW-1:0] in_a ; // non-FF
wire dif_flg ;

dif_chk  dif_chk_01( .clk(clk), .rst(rst),
            .in_a(in_a), .dif_flg(dif_flg) ) ;
initial begin
 rst = 1'b1 ;
#(CYCL*2) rst = 1'b0 ;
#(CYCL*2) rst = 1'b1 ;
#CYCL rst = 1'b0 ;
end
always begin
  clk = 0 ; #HF_CYCL ;
  clk = 1 ; #HF_CYCL ;
end
```

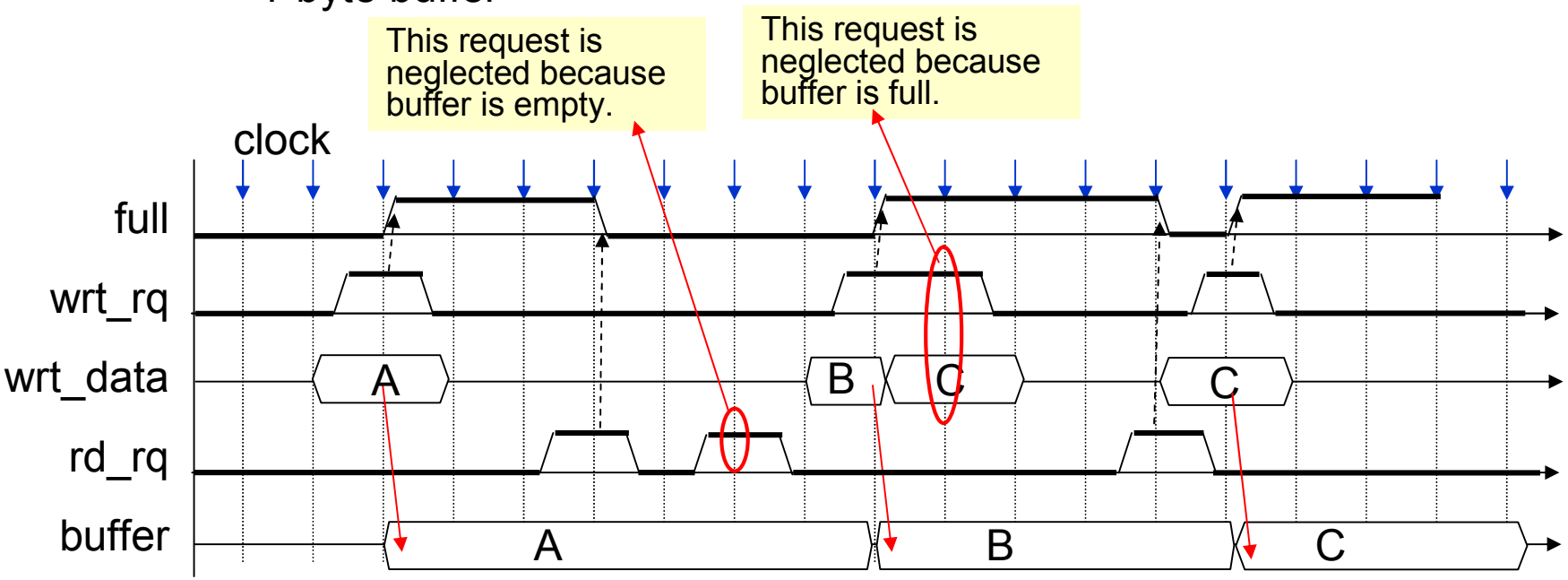
```
initial begin
in_a = 20 ;
#CYCL in_a = 100 ;
#CYCL in_a = 30 ;
#CYCL in_a = 80 ;
#CYCL in_a = 29 ;
#CYCL in_a = -21 ;
#CYCL in_a = -72 ;
#CYCL in_a = 0 ;
#CYCL in_a = 44 ;
#CYCL in_a = 80 ;
#CYCL in_a = 119 ;
#CYCL $finish ;
end
always @ ( posedge clk ) begin
  $strobe(
     "t=%d,rst=%b, in_a=%d, dif_flg=%b",
          $stime, rst, in_a, dif_flg ) ;
end
endmodule
```

file name: test_dif_chk.v

Ex 4-8. One byte data buffer: Write a module of 1-byte buffer of which behavior is shown in the wave form below. Read and write operations are synchronized with one clock signal named clk. Use active low asynchronous reset.



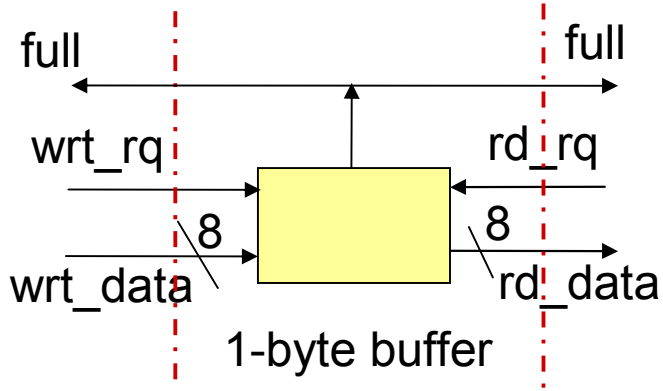
If both rd_rq and wrt_rq are asserted while buffer is empty, neglect rd_rq and accept wrt_rq. If both rd_rq and wrt_rq are asserted while buffer is full, neglect wrt_rq and accept rd_rq.



There is only one buffer for simplicity.

Explanation about the specification.

Request to send wrt_data by setting wrt_rq=1 when full=0.
If full=1, the request will be neglected and fail.



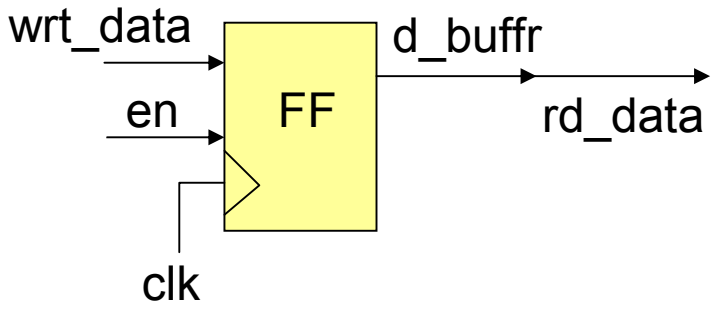
Request to receive data to rd_data when full=1.
If full=0, the request will be neglected and fail.

For simplicity, connect rd_data to buffer itself. This means that there is always some data on rd_data signal, but it is meaningful only when full=1.

A state transition table

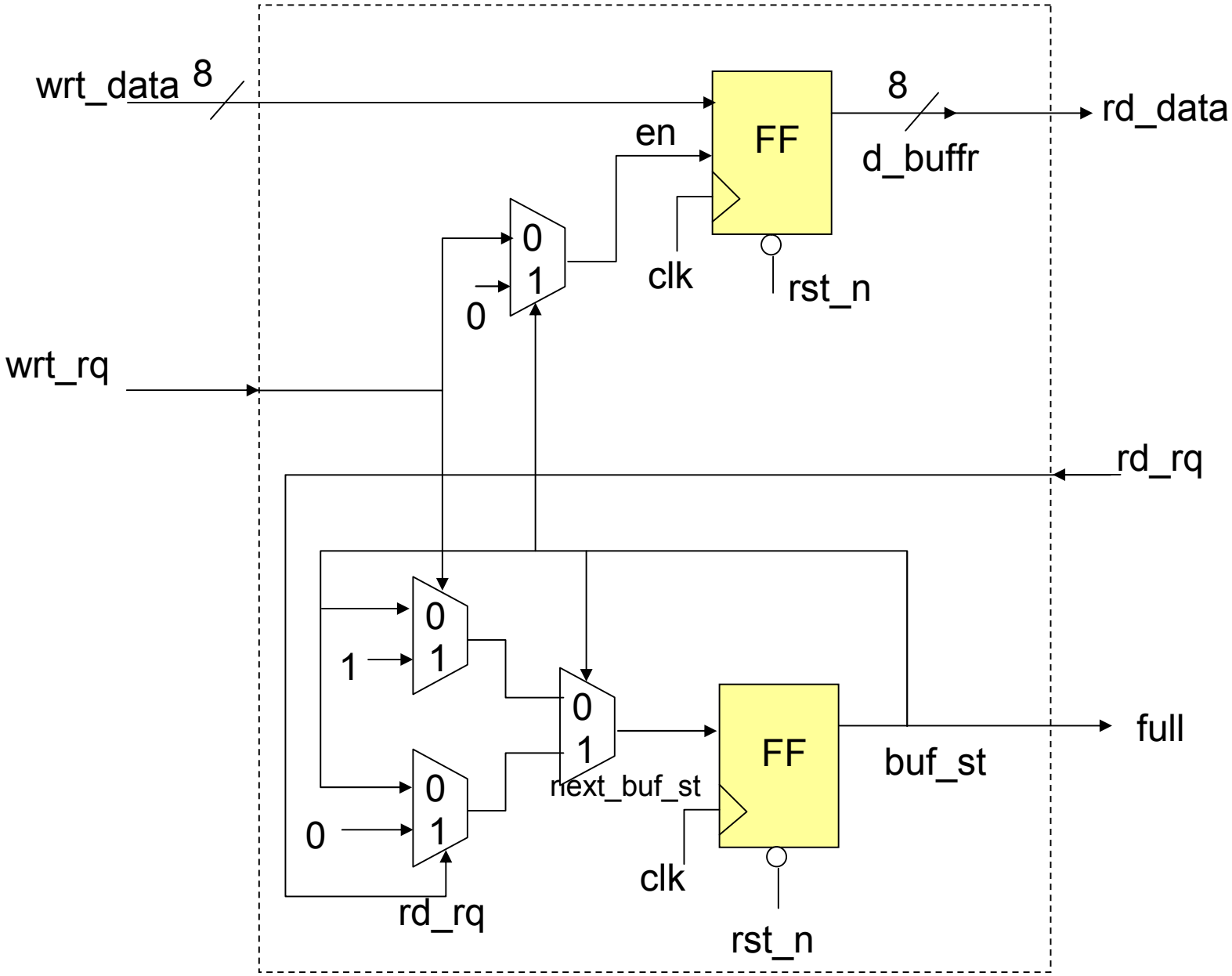
rst_n	wrt_rq	rd_rq	B_EMPT	B_FUL
0	—	—	⇨ B_EMPT	⇨ B_EMPT
1	0	0	no operation	no operation
	1	0	wrt_data → d_buffr ⇨ B_FUL	no operation
	0	1	no operation	⇨ B_EMPT
	1	1	wrt_data → d_buffr ⇨ B_FUL	⇨ B_EMPT

There is no operation, d_buffr → rd_data, because it is done always as on the right figure.



Write operation to d_buffr is activated by setting enable signal of FF to 1.

⇨ By introducing en, the table can be updated.



Final state transition table

rst_n	wrt_rq	rd_rq	B_EMPT	B_FUL
0	—	—	en=0 ⇨ B_EMPT	en=0 ⇨ B_EMPT
1	0	0	en=0	en=0
	1	0	en=1 ⇨ B_FUL	en=0
	0	1	en=0	en=0 ⇨ B_EMPT
	1	1	en=1 ⇨ B_FUL	en=0 ⇨ B_EMPT

This table can be mapped into RTL code as below.

```
case ( buf_st )
  B_EMPT : begin
       en = ( wrt_rq )? 1 : 0 ;
     end
  B_FUL : begin
        en = 0 ;
      end
endcase

case ( buf_st )
   B_EMPT : begin
       next_buf_st = ( wrt_rq )?
                          B_FUL : buf_st ;
     end
   B_FUL : begin
       next_buf_st =  ( rd_rq )?
                          B_EMPT : buf_st ;
     end
endcase
```

The total module is shown on the next page.

```
module one_byte_buffr ( clk, rst_n, wrt_rq, wrt_data,
                        rd_rq, full, rd_data ) ;
parameter N_BW = 8 ;
parameter B_EMPT = 1'b0 ;
parameter B_FUL = 1'b1 ;

input clk, rst_n ;
input wrt_rq ;
input [N_BW-1:0] wrt_data ;
input rd_rq ;

output full ;
output [N_BW-1:0] rd_data ;

wire clk, rst_n ;
wire wrt_rq ;
wire [N_BW-1:0] wrt_data ;
wire rd_rq ;

reg buf_st ;  // state FF
reg next_buf_st ; // non-FF
reg [N_BW-1:0] d_buffr ;  // buffer FF
reg en ;   // non-FF, enable for buffer FF

wire full = ( buf_st == B_FUL ) ;
wire [N_BW-1:0] rd_data = d_buffr ;
```

```
always @ ( posedge clk or
                    negedge rst_n ) begin
 if ( rst_n==1'b0 ) begin
    buf_st <= B_EMPT ;
 end

else begin
    buf_st <= next_buf_st ;
    if ( en ) begin
       d_buffr <=  wrt_data ;
    end
 end
end
```

```
always @ ( buf_st or wrt_rq or rd_rq ) begin
  case ( buf_st )
    B_EMPT : begin
          en = ( wrt_rq )? 1'b1 : 1'b0 ;
        end
    B_FUL : begin
          en = 1'b0 ;
        end
    default : begin
          en = 1'bx ;
        end
  endcase
end
always @ ( buf_st or wrt_rq or rd_rq ) begin
  case ( buf_st )
    B_EMPT : begin
          next_buf_st = ( wrt_rq )? B_FUL : buf_st ;
        end
    B_FUL : begin
          next_buf_st =  ( rd_rq )? B_EMPT : buf_st
;
        end
    default : begin
          next_buf_st = 1'bx ;
        end
 endcase
end
endmodule
```

default is added for debug

default is added for debug

file name: one_byte_buffr.v

Now run the code on the previous page with a test bench below.

```
module test_one_byte_buffr  ;
parameter H_CYC = 50 ;
parameter N_BW = 8 ;
reg rst_n, clk ; // non-FF
reg rd_rq, wrt_rq ; // non-FF
reg [N_BW-1:0] wrt_data ; // non-FF
wire [N_BW-1:0] rd_data ;
wire full ;
always begin
  clk = 0 ; #H_CYC ;
  clk = 1 ; #H_CYC ;
end
one_byte_buffr  one_byte_buffr_01 ( .clk(clk), .rst_n(rst_n),
    .rd_rq(rd_rq), .wrt_rq(wrt_rq), .wrt_data(wrt_data),
    .full(full), .rd_data(rd_data)   ) ;
initial begin
 $monitor(
   "t=%d, rst_n=%b, r_q=%b, w_q=%b, w_d=%h, r_d=%h, full=%b",
    $stime, rst_n, rd_rq, wrt_rq, wrt_data, rd_data, full ) ;
end
```

```
initial begin
rst_n = 0 ;
#(H_CYC*2) rst_n = 1 ;
end
initial begin
wrt_rq = 0 ;
#(H_CYC*4) wrt_rq = 1 ;
#(H_CYC*2) wrt_rq = 0 ;
#(H_CYC*8) wrt_rq = 1 ;
#(H_CYC*2) wrt_rq = 0 ;
end
```

```
initial begin
rd_rq = 0 ;
#(H_CYC*6) rd_rq = 1 ;
#(H_CYC*2) rd_rq = 0 ;
#(H_CYC*2) rd_rq = 1 ;
#(H_CYC*2) rd_rq = 0 ;
#(H_CYC*10) rd_rq = 1 ;
#(H_CYC*2) rd_rq = 0 ;
end
initial begin
wrt_data = 8'h00 ;
#(H_CYC*4) wrt_data = 8'hfa ;
#(H_CYC*10) wrt_data = 8'ha5 ;
#(H_CYC*50) $finish ;
end

endmodule
```

file name: test_one_byte_buffr.v

There is different idea for the buffer. The idea explained below is not a recommended way of design. It uses a technique called asynchronous design. The following pages are just for explanation. Never apply the method in actual jog.

A variable needed to manage empty/full of buffer:
→ full

A variable needed to hold written data:
→ d_bffr

Find program logic which can set values on these variables as defined in the specification.

The change of state is invoked by the change of rd_rq or wrt_rq.

```
always @ ( rd_rq or wrt_rq ) begin


end
```

With the above structure, the buffer can be implemented.

Try to write code to realize the given behavior by only using variables such as rd_rq, wrt_rq, full, d_bffr, wrt_data, and rd_data.

```
always @ ( rd_rq or wrt_rq ) begin
  case ( { rd_rq, wrt_rq } )
    2'b00 : begin end   // no-operation
    2'b01 : if ( ~full ) begin
                d_bffr = wrt_data ;
                full = 1 ;
            end
    2'b10 : if ( full ) begin
                rd_data = d_bffr ;
                full = 0 ;
            end
    2'b11 : if ( full ) begin
                rd_data = d_bffr ;
                full = 0 ;
            end
            else begin
                d_bffr = wrt_data ;
                full = 1 ;
            end
  endcase
end
```

rd_req or wrt_rq activate actions.

No change if no request.

If wrt_rq asserted while buffer is empty, update buffer and make state=full.

If rd_rq asserted while buffer is full, set rd_data and make state=empty.

If rd_rq and wrt_rq asserted while buffer is full, neglect wrt_rq and set rd_data and make state=empty.

If rd_rq and wrt_rq asserted while buffer is empty, neglect rd_rq and update buffer and make state=full.

The code on the previous page does not use reset signal.
Reset signal must be implemented as below.

```
always @ ( rd_rq or wrt_rq or rst_n ) begin
 if ( rst_n==1'b0 ) begin
                    full = 0 ;
 end
 else begin
   case ( { rd_rq, wrt_rq } )
     2'b00 : begin end   // no-operation
     2'b01 : if ( ~full ) begin
                    d_bffr = wrt_data ;
                    full = 1 ;
             end
     2'b10 : if ( full ) begin
                    rd_data = d_bffr ;
                    full = 0 ;
             ⋮
```

Add rst_n.

Add reset processing.

This will result in the following module.

```
module wrong_onebyte_buffr
     ( rst_n, rd_rq, wrt_rq, wrt_data, full, rd_data ) ;
input rst_n, rd_rq, wrt_rq ;
input [7:0] wrt_data ;
output full ;
output [7:0] rd_data ;
wire rst_n, rd_rq, wrt_rq ;
wire [7:0] wrt_data ;
reg full ;
reg [7:0] rd_data ;

reg [7:0] d_bffr ;

always @ ( rd_rq or wrt_rq
                    or rst_n ) begin
 if ( rst_n==1'b0 ) full = 0 ;
 else begin
```

```
   case ( { rd_rq, wrt_rq } )
      2'b00 : begin end   // no-operation
      2'b01 : if ( ~full ) begin
                        d_bffr = wrt_data ;
                        full = 1 ;
              end
      2'b10 : if ( full ) begin
                        rd_data = d_bffr ;
                        full = 0 ;
              end
      2'b11 : if ( full ) begin
                        rd_data = d_bffr ;
                        full = 0 ;
              end
              else begin
                        d_bffr = wrt_data ;
                        full = 1 ;
              end
   endcase
  end
 end

 endmodule
```

This is a target code which does not use synchronous design.

```
module test_wrong_onebyte_buffr ;
reg rst_n ;
reg rd_rq, wrt_rq ;
reg [7:0] wrt_data ;
wire full ;
wire [7:0] rd_data ;
wrong_onebyte_buffr  wrong_onebyte_buffr_01 ( .rst_n(rst_n),
    .rd_rq(rd_rq), .wrt_rq(wrt_rq), .wrt_data(wrt_data), .rd_data(rd_data)   ) ;
initial $monitor(
"rst_n=%b, rd_rq=%b, wrt_rq=%b, wrt_data=%h, rd_data=%h, ful=%b",
 rst_n, rd_rq, wrt_rq, wrt_data, rd_data, full ) ;
initial begin
rst_n = 0 ;
rd_rq = 0 ;  wrt_rq = 0 ;
#10 rst_n = 1 ;
#10 wrt_rq = 1 ;  wrt_data = 8'h5a ;
#10 wrt_rq = 0 ; rd_rq = 1 ;
#10 wrt_rq = 1 ; rd_rq = 0 ; wrt_data = 8'hf0 ;
#10 wrt_rq = 1 ; rd_rq = 0 ; wrt_data = 8'haa ;
#10 wrt_rq = 0 ; rd_rq = 1 ;
#10 rd_rq = 0 ;
#10 $finish ;
end
endmodule
```

Check the code on the previous page by using a test bench in this page.

```
always @ ( rd_rq or wrt_rq  or rst_n ) begin
 if ( rst_n==1'b0 ) full = 0 ;
 else begin
   case ( { rd_rq, wrt_rq } )
    2'b00 : begin end  // no-operation
    2'b01 : if ( ~full ) begin
                    d_bffr = wrt_data ;
                    full = 1 ;
            end
    2'b10 : if ( full ) begin
                    rd_data = d_bffr ;
                    full = 0 ;
            end
    2'b11 : if ( full ) begin
                    rd_data = d_bffr ;
                    full = 0 ;
            end
            else begin
                    d_bffr = wrt_data ;
                    full = 1 ;
            end
  endcase
 end
end
```

A simulation result may looks correct, but the code has serious problem.

No edge signal written, therefore combinational logic or latch may be created by this always construct.

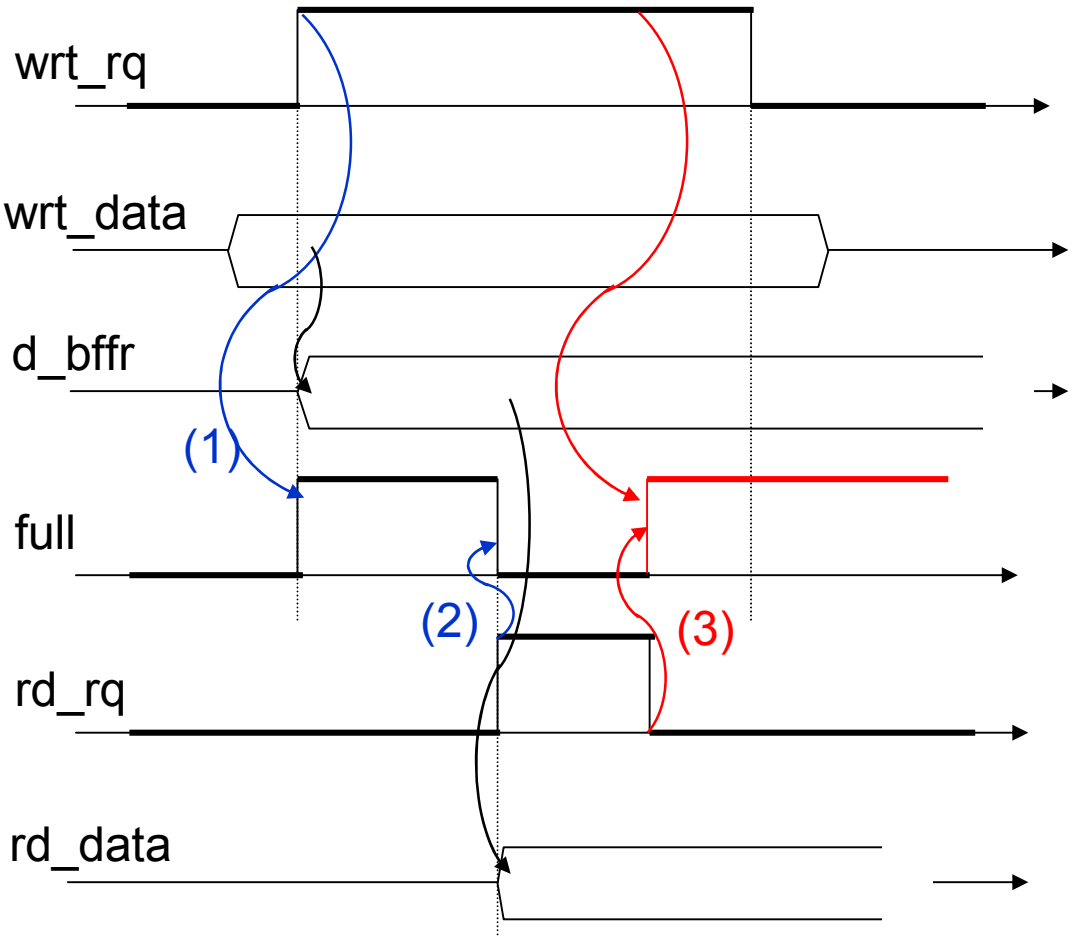
No assignment to b_full, d_bffr, rd_data in this path.

No assignment to rd_data in this path.

No assignment to d_buffr in this path.

Latches will be created for b_full, d_bffr, and rd_data. The gate signals are not explicitly written in the RTL code.

Problem1: In the following situation, buffer may become full even if it is read by rd_rq signal.



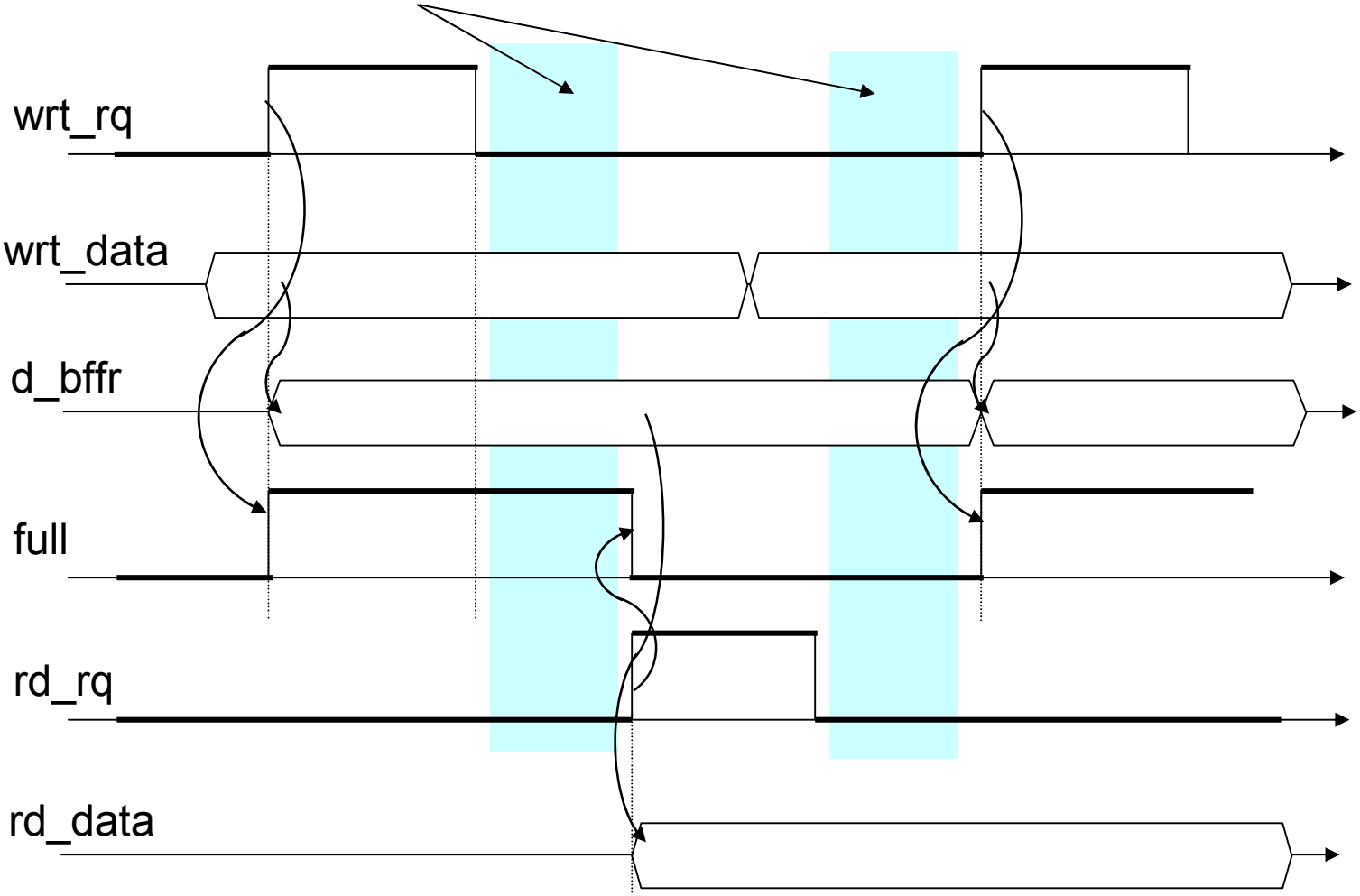
(1) b_ful becomes 1 by wrt_rq = 1.

(2) If rd_rq becomes 1 while wrt_rq=1, wrt_rq is neglected and data in buffer is copied to rd_data and full becomes 0.

(3) When rd_rq changes its value from 1 to 0, always construct is activated with full = 0, rd_rq = 0, wrt_rq = 1 therefore full is set to 1.

Once buffer becomes empty, but it is filled with previous write request and becomes full again.

The problem in the previous page may solved by making wrt_rq and rd_rq never become on at the same time.

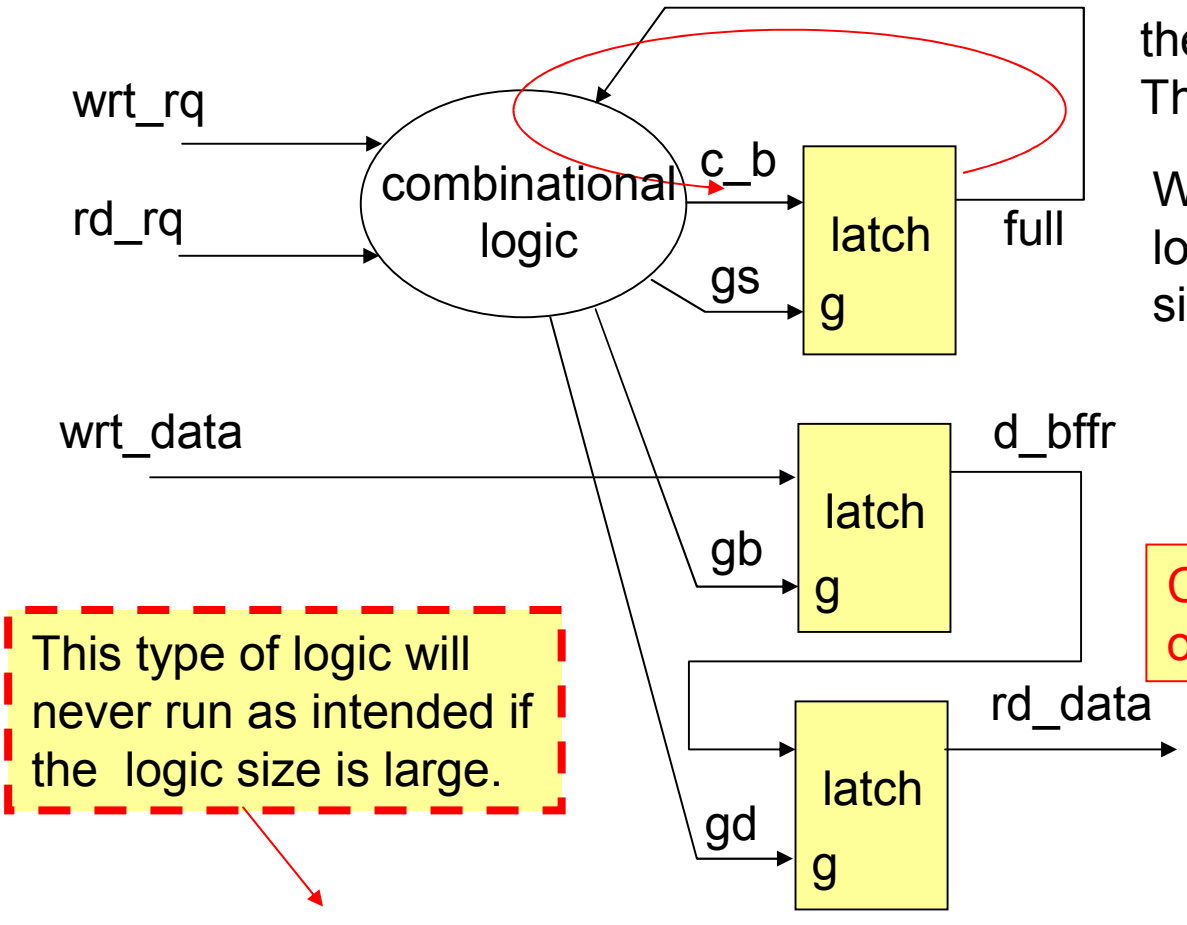


The RTL simulation may run successfully if above constraint are followed. However, created circuit may still have problems.

Problem2: A gate diagram given by a synthesis tool may looks like below.This circuit will never work correctly.

There is a closed loop among D-type transparent latch and the combinational logic. Therefore, c_b may oscillate.

We can not estimate what logic is used to create gate signals from the RTL code,



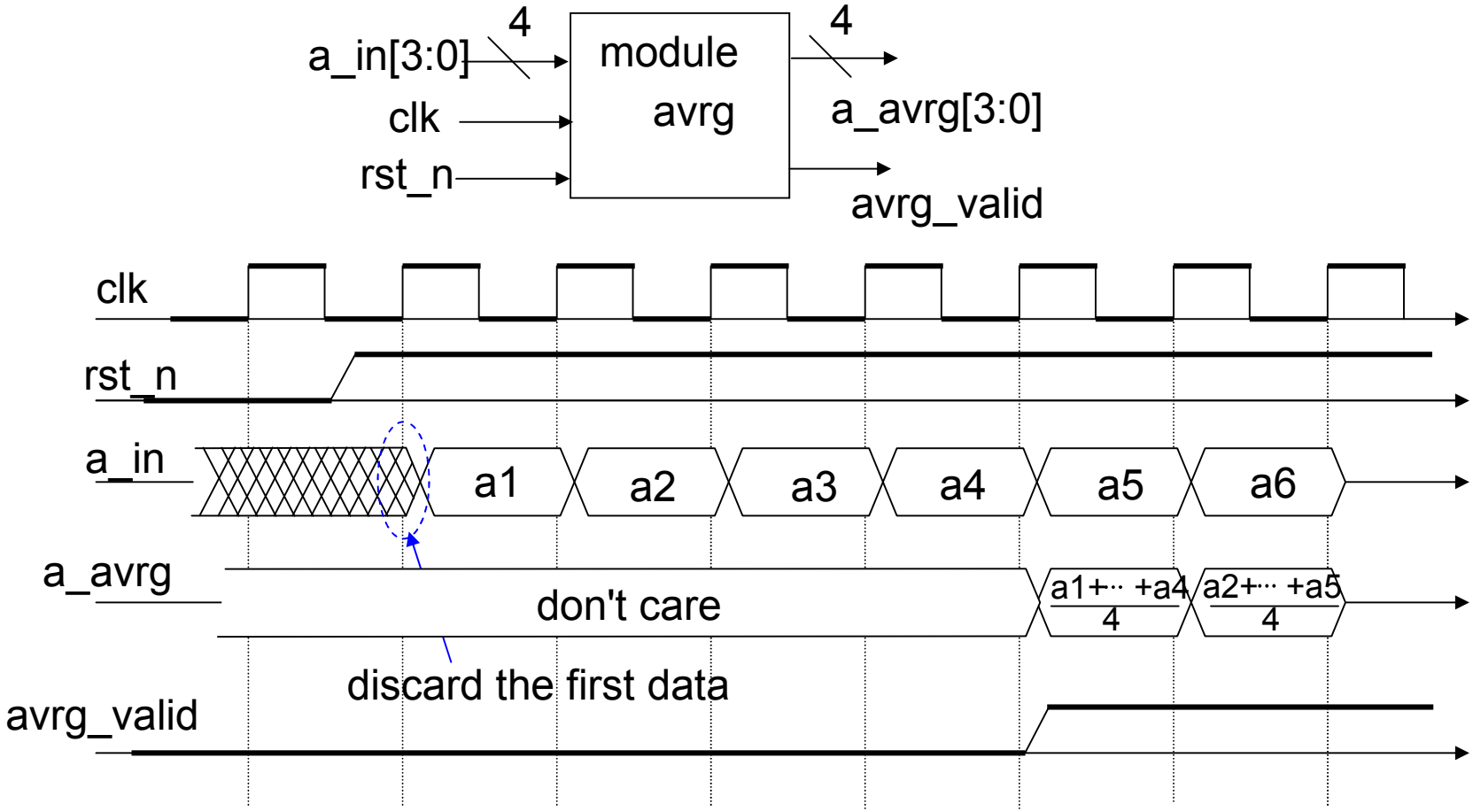
This type of logic will never run as intended if the logic size is large.

Can not predict on/off timing of the gate of latches.

Critical

Because STA is not applicable to this kind of logic. Note that there is no clock signal. Therefore, we do not have any tools that assure the timing issues.

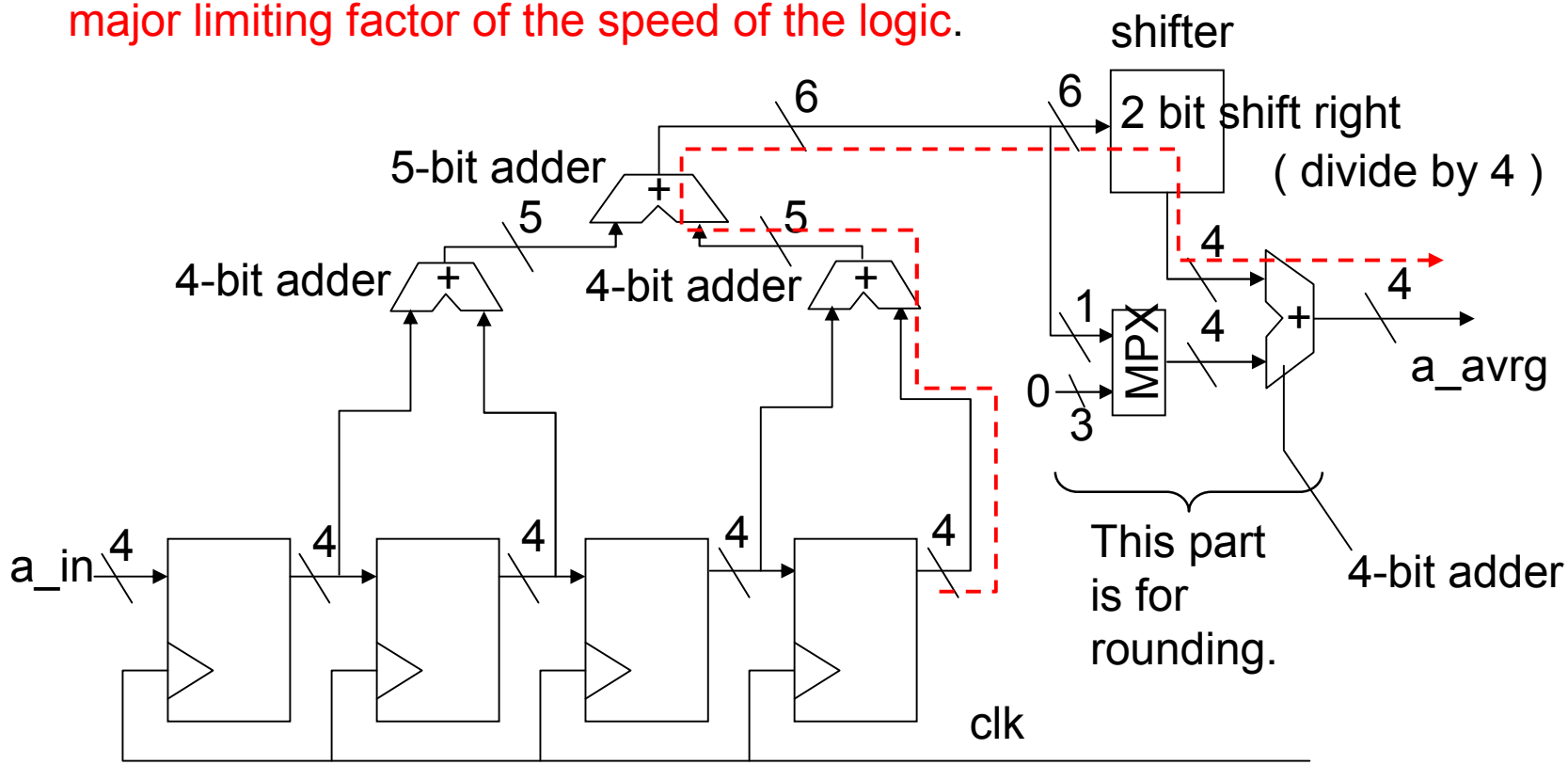
Ex 4-9. Average of sequential four 4-bit data: Write a module which output the average of latest four data coming from input a_in each time the clock clk rises. The result must be rounded, that is, 7.5 must be 8, 3.25 must be 3.
avrg_valid signal must be off while output a_avrg is not an average of the latest four data.



(1) How to calculate the average?

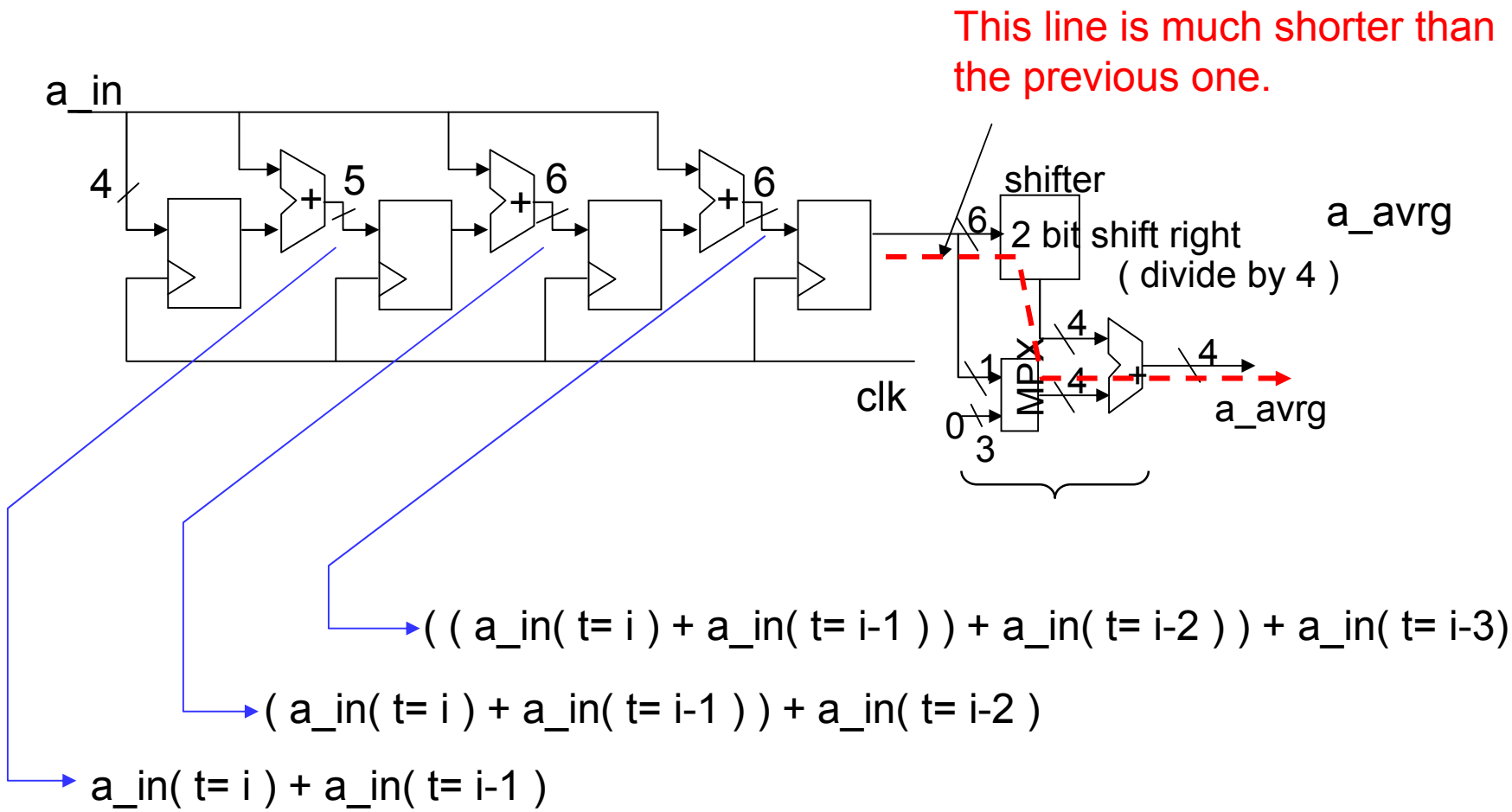
The following may be the most straight forward simple structure. However, it has 3 adders serially in the path indicated by the red dashed line.

In synchronous design, time needed between FFs, or FF to output, is major limiting factor of the speed of the logic.



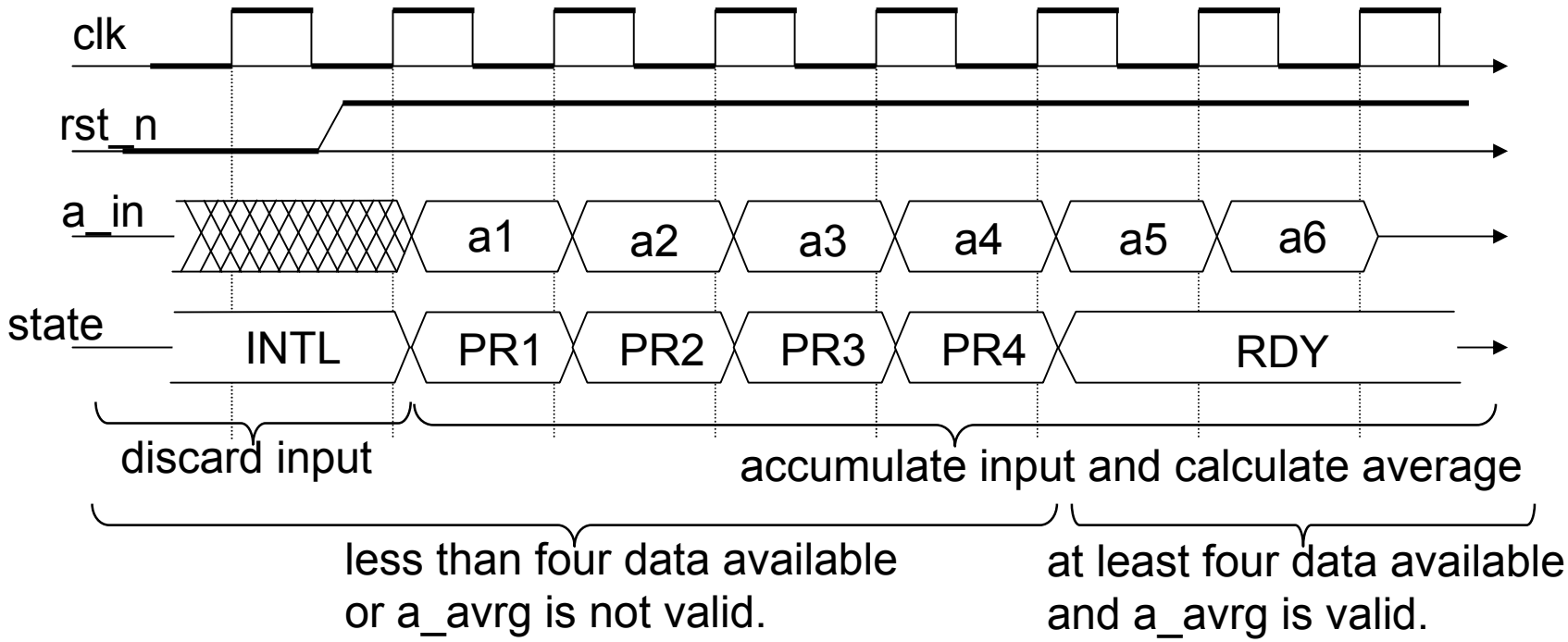
Therefore, the structure show on the next page may be better because the length from FF to FF, the red dashed line, is short.

In the design below, we only have one adder in the red dashed path. Therefore, this design may be better from the view point of speed.



(2) How to control state?

This logic needs a state to discard the first data. To simplify the logic, four states, PR1 to PR4, are introduced beside INTL(initial) and RDY(ready).

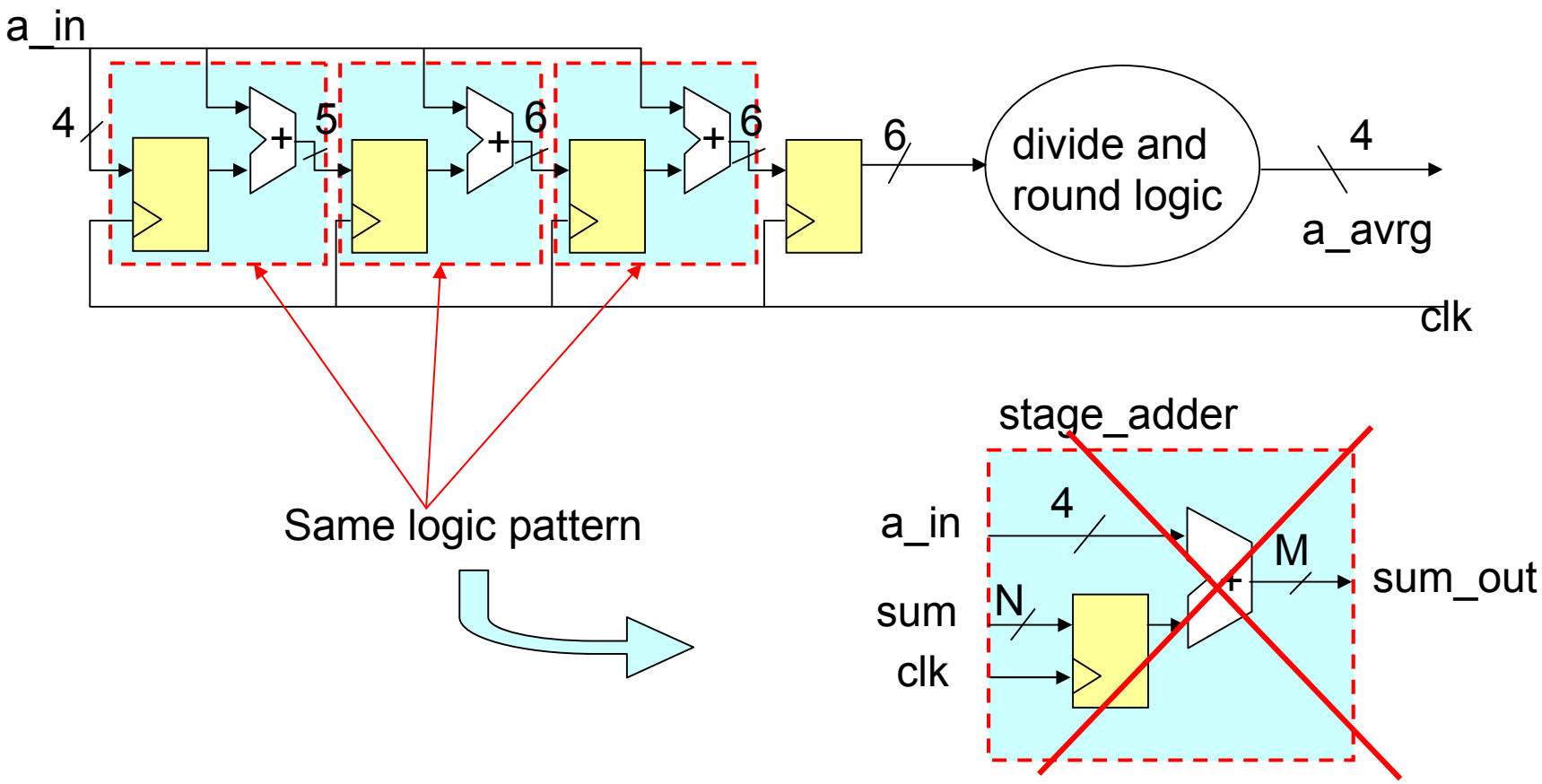


State control logic:

rst_n	clk \ state	INTL	PR1	PR2	PR3	PR4	RDY
0	x	⇨ INTL	⇨ INTL	⇨ INTL	⇨ INTL	⇨ INTL	⇨ INTL
1	↑	⇨ PR1	⇨ PR2	⇨ PR3	⇨ PR4	⇨ RDY	no operation
	else	no operation					

(3) Extract common logic block

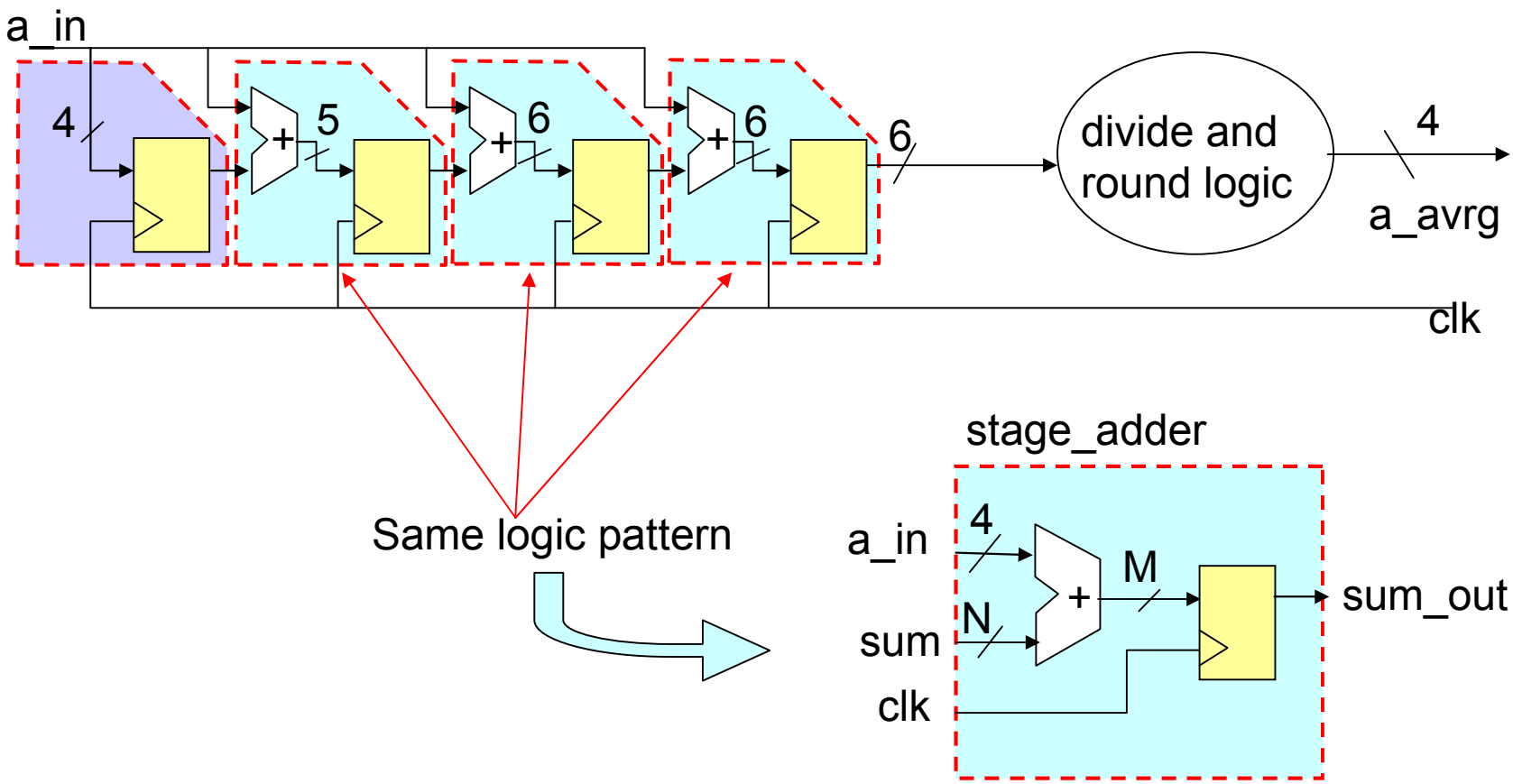
Extract common process pattern and make it a basic module.



However, this module structure is not a recommended style:

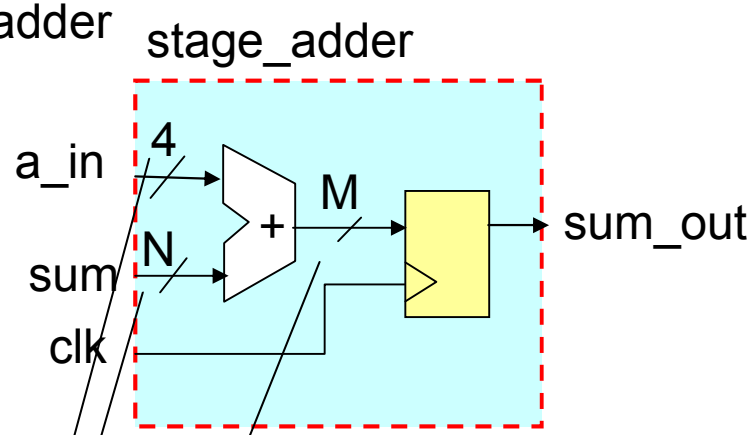
Direct path from input to output and FF path co-exist.

Output does not come directly from FF.



This sub-module structure is better because output comes from FF directly and all the signals go through almost equal logic stages.

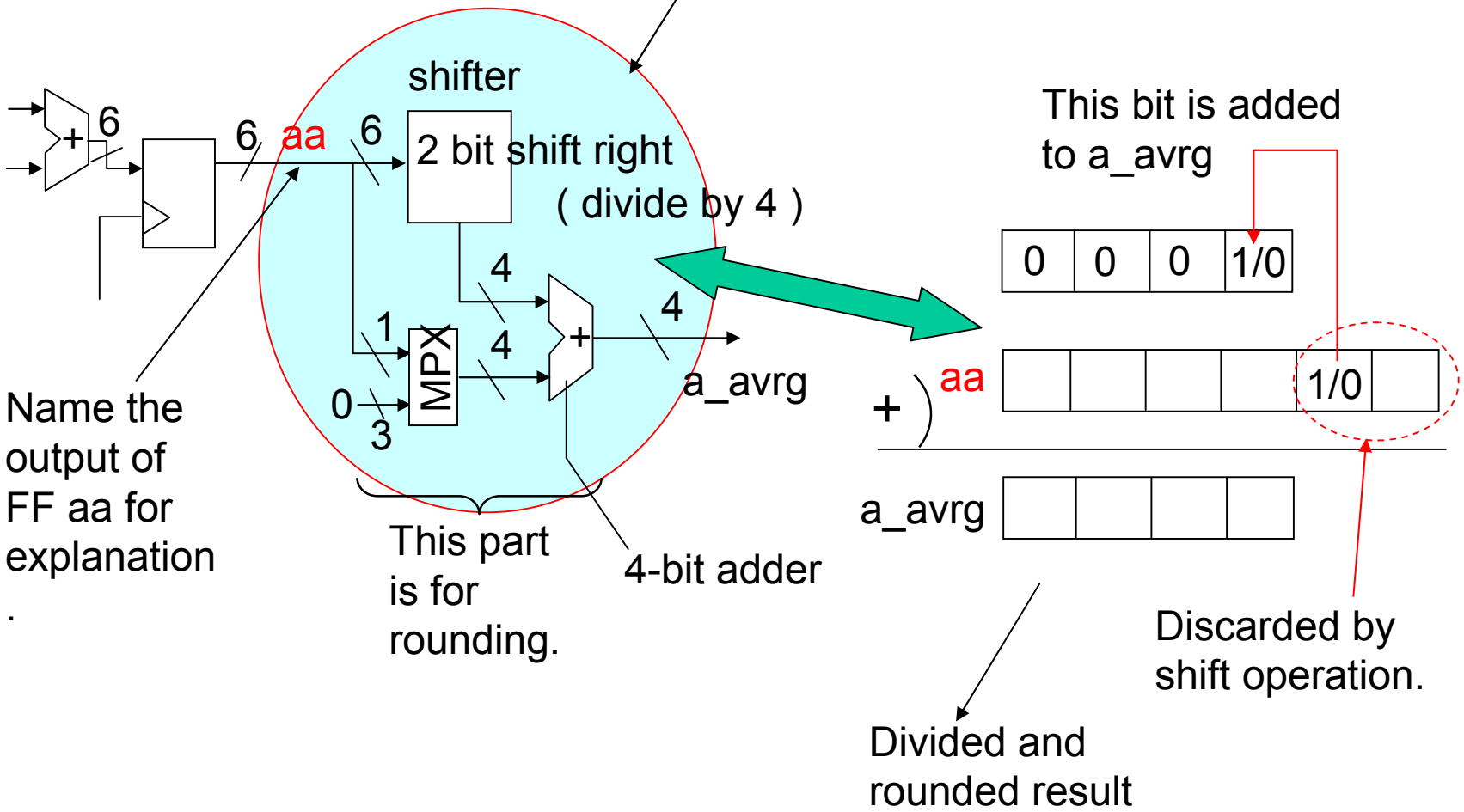
RTL code of stage_adder




```
//
module stage_adder ( clk, rst_n, a_in, sum, sum_out ) ;
//

parameter A_IN_BW = 4 ;    // a_in bit size
parameter SUM_IN_BW = 4 ;    // sum bit size
parameter SUM_OUT_BW = 5 ;    // sum_out bit size,
//              SUM_OUT_BW can be as large as SUM_IN_BW + 1
//
input clk, rst_n ;
input [A_IN_BW-1:0] a_in ;
input [SUM_IN_BW-1:0] sum ;
output [SUM_OUT_BW-1:0] sum_out ;
//
```

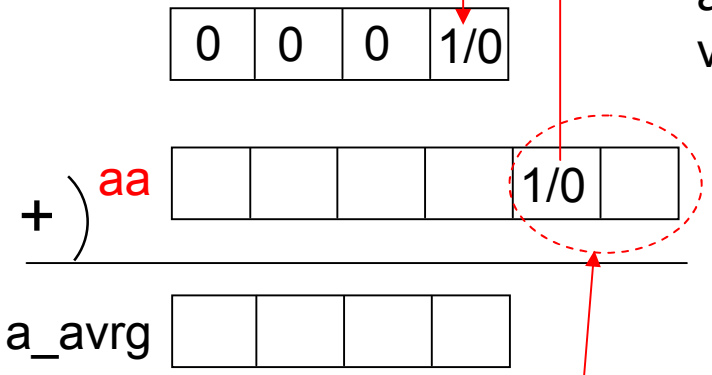
```
wire clk, rst_n ;
wire [A_IN_BW-1:0] a_in ;
wire [SUM_BW-1:0] sum ;
reg [SUM_OUT_BW-1:0] sum_out ; // FF
// internal variables
wire [SUM_OUT_BW-1:0] sum_out ;
//
//  max bit size can be as large as SUM_IN_BW +1
// LHS bit size must be larger than that of RHS to hold carry bit
assign next_sum_out = sum + a_in ;
//
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n==1'b0 ) begin
     sum_out <= 0 ;
  end
  else begin
     sum_out <= next_sum_out ;
  end
end
//
endmodule
```

(4) rounding operation



Can't this logic be much simpler?

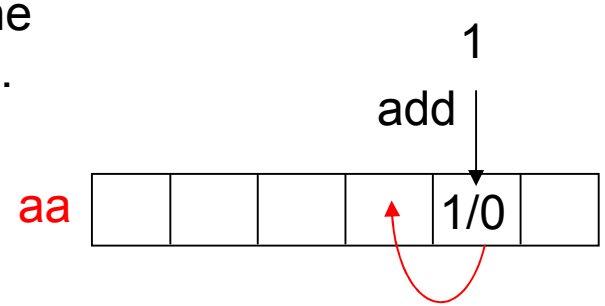
This bit is added to a_avrg



Discarded by shift operation.

This is the same to adding 1 to bit 1 of aa regardless of the value of bit 1 of aa.

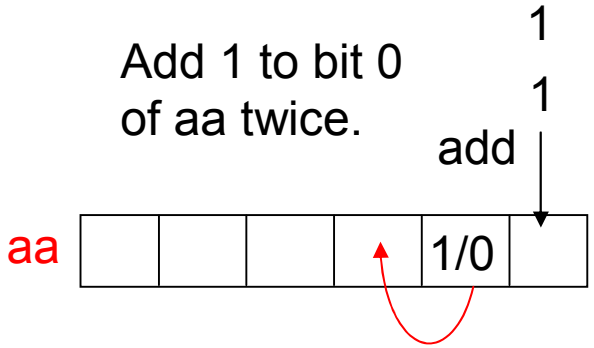
Add 1 to bit 1 of aa.

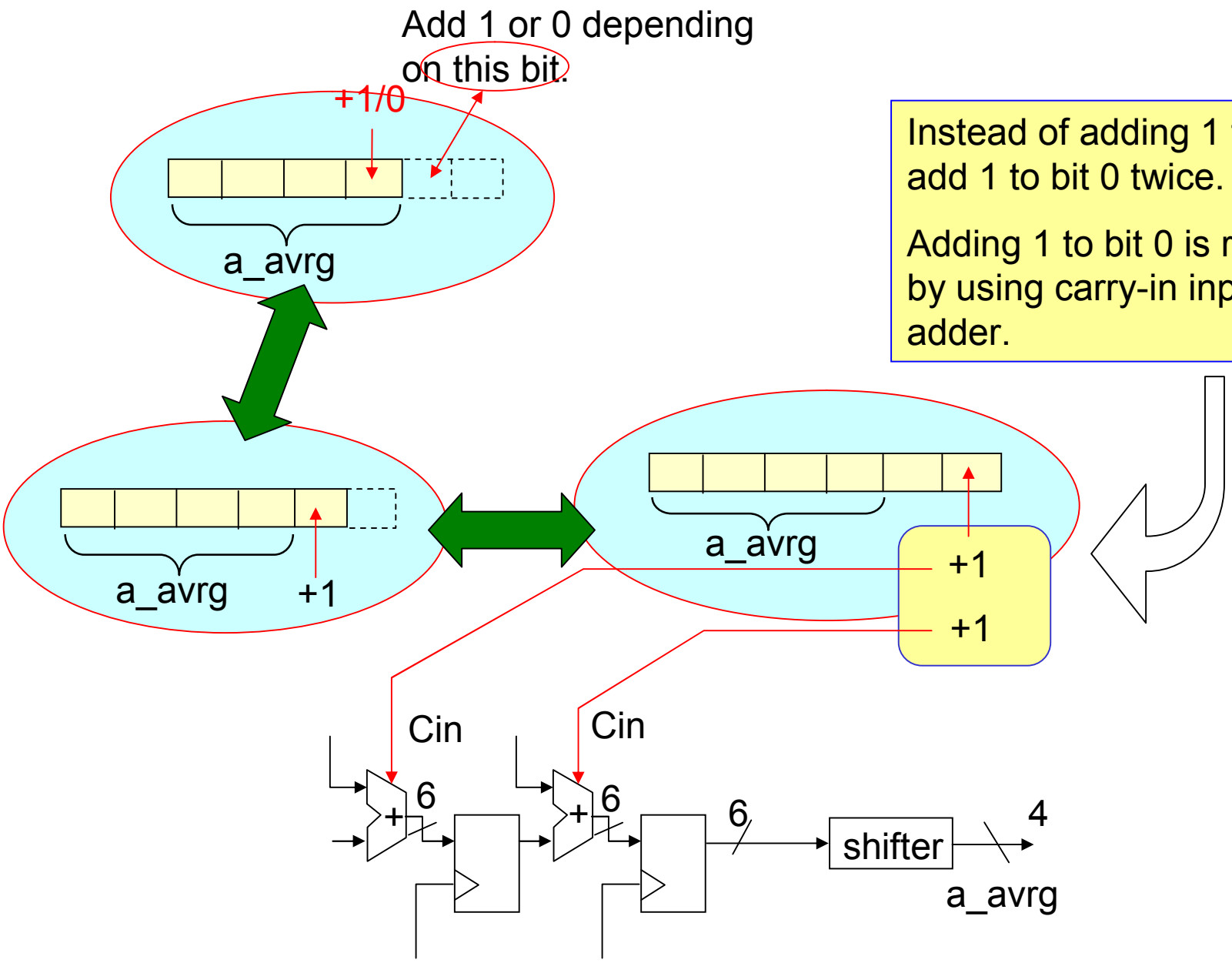


If bit 1 is 1, then carry to bit 2, if 0, no carry to bit 2.

Adding 1 to bit 1 is equal to add 1 to bit 0 twice.

Add 1 to bit 0 of aa twice.

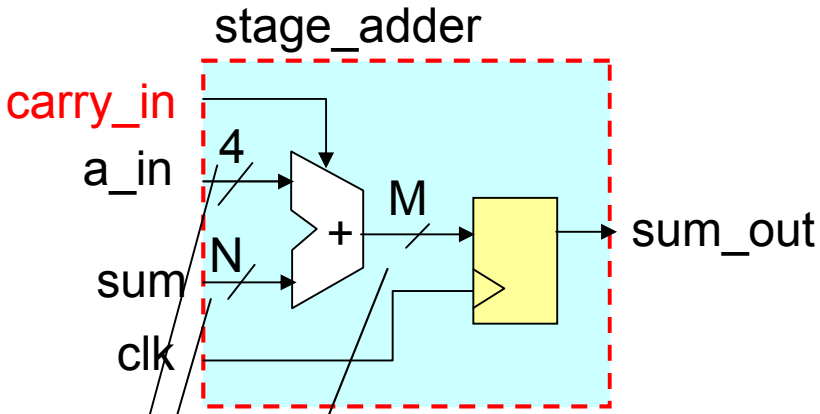




Instead of adding 1 to bit 1, add 1 to bit 0 twice.

Adding 1 to bit 0 is realized by using carry-in input of an adder.

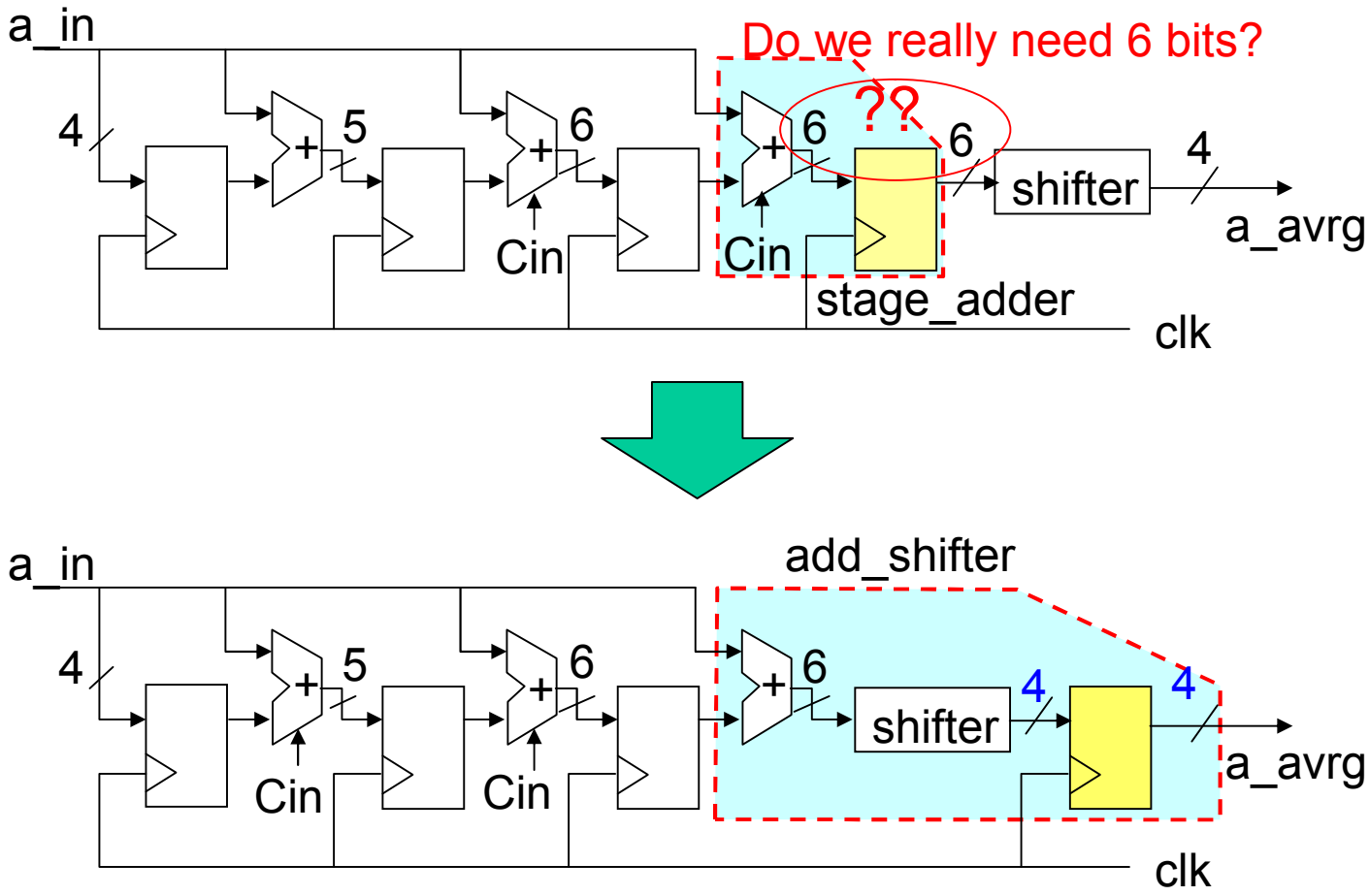
RTL code of stage_adder updated




```
//
module stage_adder ( clk, rst_n, a_in, sum, carry_in, sum_out ) ;
//

parameter A_IN_BW = 4 ;    // a_in bit size
parameter SUM_IN_BW = 4 ;    // sum input bit size
parameter SUM_OUT_BW = 5 ;    // sum_out bit size,
//              SUM_OUT_BW can be as large as SUM_IN_BW + 1
//
input clk, rst_n ;
input [A_IN_BW-1:0] a_in ;
input [SUM_IN_BW-1:0] sum ;
input carry_in ;
output [SUM_OUT_BW-1:0] sum_out ;
//
```

```
wire clk, rst_n ;
wire [A_IN_BW-1:0] a_in ;
wire [SUM_BW-1:0] sum ;
wire carry_in ;
reg [SUM_OUT_BW-1:0] sum_out ; // FF
//
// internal variables
wire [SUM_OUT_BW-1:0] sum_out ;
//
//  max bit size can be as large as SUM_IN_BW+1
// LHS bit size must be larger than that of RHS to hold carry bit
assign next_sum_out = sum + a_in + carry_in ;
//
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n==1'b0 ) begin
     sum_out <=  0 ;
  end
  else begin
     sum_out <=  next_sum_out ;
  end
end
//
endmodule
```

(5) Decrease bit size of the last FF



(6) state control implementation

State transition table

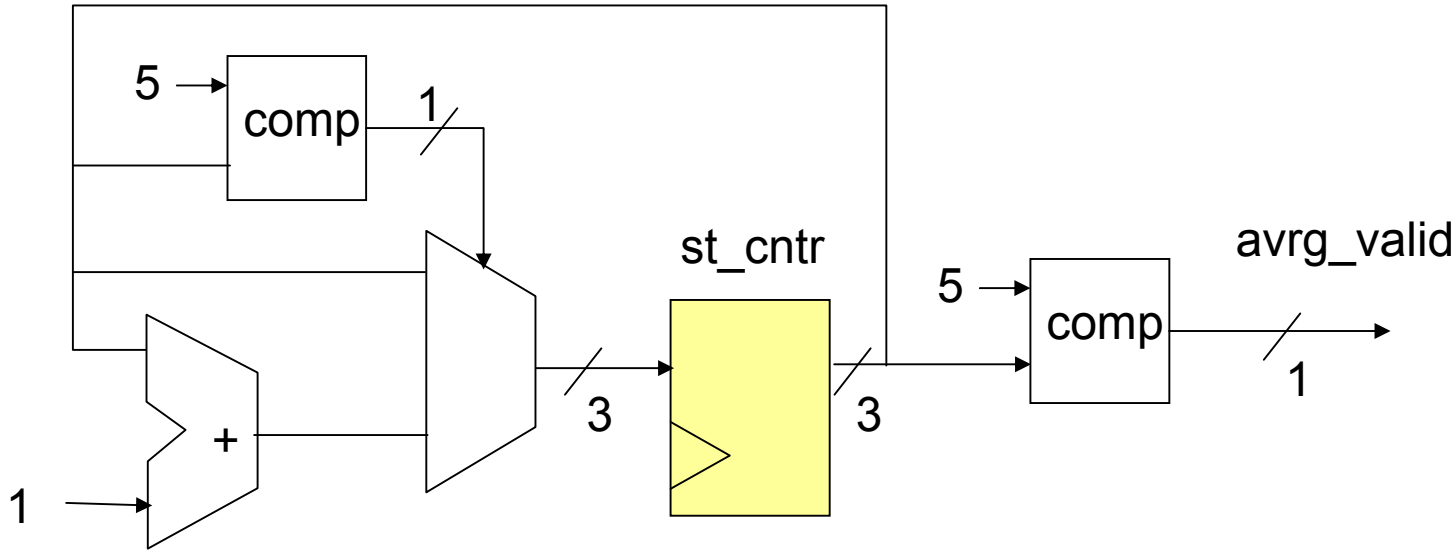
rst_n	clk \ state	INTL	PR1	PR2	PR3	PR4	RDY
0	x	FF1,2,3,4 = 0 avrg_valid=0 ⇨ INTL					
1	↑	no operation ⇨ PR1	*1 ⇨ PR2	*1 ⇨ PR3	*1 ⇨ PR4	*1 avrg_valid=1 ⇨ RDY	*1 ⇨ RDY
	else	no operation					

Instead of using state PR1,PR2,, etc., introduce a counter (st_cntr) and count up it by 1 at every clock rise time.

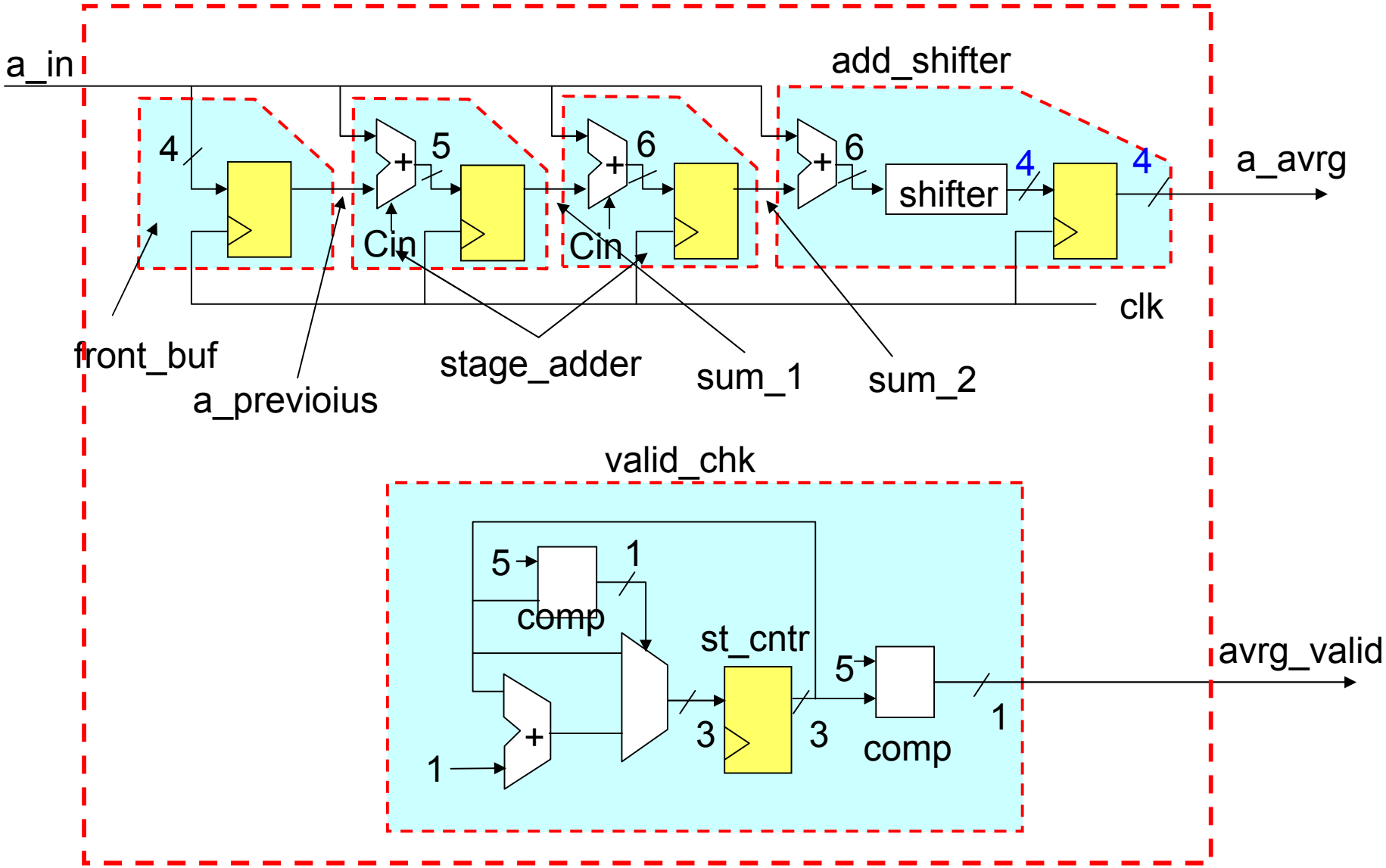
st_cntr = 0 ⇨ 1 ⇨ 2 ⇨ 3 ⇨ 4 ⇨ 5

INTL RDY

valid signal creation logic



(7) Data path structure and RTL code



```
//-----------------------------------------------------------------
// Project Name : RVC design training
//              : average of four four-bit numbers
// Module Name  : avrg
// Function     : this logic out put average of input a_in,
//                it assume that input is valid at ever clock rise time,
//                but discard the first data after reset negated.
// note : divide by 4 is implemented by shift operetaion in add_shifter
//        module, and the shift count is controlled by bit size of a_in and
//        bit size of sum input for add_shifter module
//        that is, shift count = SUM_IN_BW - A_IN_BW,
//        therefore if this relation is not applicable, the divide logic
//        in add_shifter module must be modified.
// Author       : K. Hayashi (idxxxx)
//--------------- modification history ------------------------------------------
// Version  Date        Author       Description
// ver.1.0  2010.xx.xx  K. Hayashi  first release
//-----------------------------------------------------------------
//
module avrg ( clk, rst_n, a_in, a_avrg, avrg_valid  ) ;
//
parameter A_IN_BW = 4 ;    // a_in bit size
```

```
//
parameter SUM_1_BW = A_IN_BW + 1 ;  // 1 bit extended for overflow
parameter SUM_2_BW = SUM_1_BW + 1 ; // another 1 bit extended
//
defparam front_buf_01.A_IN_BW = A_IN_BW ;
//
defparam stage_adder_01.A_IN_BW = A_IN_BW ;
defparam stage_adder_01.SUM_IN_BW = A_IN_BW ;
defparam stage_adder_01.SUM_OUT_BW = SUM_1_BW ;
//
defparam stage_adder_02.A_IN_BW = A_IN_BW ;
defparam stage_adder_02.SUM_IN_BW = SUM_1_BW ;
defparam stage_adder_02.SUM_OUT_BW = SUM_2_BW ;
//
defparam add_shifter_01.A_IN_BW = A_IN_BW ;
defparam add_shifter_01.SUM_IN_BW = SUM_2_BW ;
//
input clk, rst_n ;
input [A_IN_BW-1:0] a_in ;
output [A_IN_BW-1:0] a_avrg ;
output avrg_valid ;
//
```

```
wire clk, rst_n ;
wire [A_IN_BW-1:0] a_in ;
wire [A_IN_BW-1:0] a_avrg ; // average of four a_in
wire avrg_valid ;
//
wire [A_IN_BW-1:0] a_previous ;
wire [SUM_1_BW-1:0] sum_1 ;
wire [SUM_2_BW-1:0] sum_2 ;
//
// connection of stage adders
valid_chk valid_chk_01 ( .clk(clk), .rst_n(rst_n),
                .valid_out(avrg_valid) ) ;
front_buf front_buf_01 ( .clk(clk), .rst_n(rst_n),
                .a_in(a_in), .a_previous(a_previous) ) ;
stage_adder stage_adder_01 ( .clk(clk), .rst_n(rst_n),
                .a_in(a_in), .sum(a_previous), .carry_in ( 1'b1 ), .sum_out(sum_1) ) ;
stage_adder stage_adder_02 ( .clk(clk), .rst_n(rst_n),
                .a_in(a_in), .sum(sum_1), .carry_in( 1'b1 ), .sum_out(sum_2) ) ;
add_shifter add_shifter_01 ( .clk(clk), .rst_n(rst_n),
                .a_in(a_in), .sum(sum_2), .avrg_out(a_avrg) ) ;
//
endmodule
```

```
module valid_chk ( clk, rst_n, valid_out ) ;
parameter VALID_CNT_MAX = 5 ; // state counter limiter
//
input clk, rst_n ;
output valid_out ;
//
wire clk, rst_n ;
wire valid_out ;
//
// internal variables
reg [2:0] st_cntr ;  // FF, counter to count up to 5
reg [2:0] next_st_cntr ;  // non-FF
// logic for valid signal
// define FF for state counter
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
        st_cntr <= 0 ;
  end
  else begin
        st_cntr <=  next_st_cntr ;
  end
end
```

```
// define next_st_cntr
always @ ( st_cntr ) begin
   if ( st_cntr == VALID_CNT_MAX ) begin
      next_st_cntr = st_cntr ; // if counted up to 5, stop update
   end
   else begin
      next_st_cntr = st_cntr + 1 ; // update counter by 1
   end
end
//  valid=1 if cntr = VALID_CNT_MAX
assign valid_out = ( st_cntr == VALID_CNT_MAX ) ;
//
endmodule
```

```
module front_buf ( clk, rst_n, a_in, a_previous ) ;
//
parameter A_IN_BW = 4 ;    // a_in bit size
//
input clk, rst_n ;
input [A_IN_BW-1:0] a_in ;
output [A_IN_BW-1:0] a_previous ;
//
wire clk, rst_n ;
wire [A_IN_BW-1:0] a_in ;
reg [A_IN_BW-1:0] a_previous ; // FF
//
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n==1'b0 ) begin
     a_previous <= 0 ;
  end
  else begin
     a_previous <=  a_in ;
  end
end
//
endmodule
```

```
module stage_adder ( clk, rst_n, a_in, sum, carry_in, sum_out ) ;
//
parameter A_IN_BW = 4 ;    // a_in bit size
parameter SUM_IN_BW = 4 ;    // sum input bit size
parameter SUM_OUT_BW = 5 ;    // sum_out bit size,
//              SUM_OUT_BW can be as large as SUM_IN_BW + 1
//
input clk, rst_n ;
input [A_IN_BW-1:0] a_in ;
input [SUM_IN_BW-1:0] sum ;
input carry_in ;
output [SUM_OUT_BW-1:0] sum_out ;
//
wire clk, rst_n ;
wire [A_IN_BW-1:0] a_in ;
wire [SUM_IN_BW-1:0] sum ;
wire carry_in ;
reg [SUM_OUT_BW-1:0] sum_out ; // FF
//
wire [SUM_OUT_BW-1:0] next_sum_out ;
//
```

```
//  max bit size can be as large as SUM_IN_BW +1
// LHS bit size must be larger than that of RHS to hold carry bit
assign next_sum_out = sum + a_in + carry_in ;
//
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n==1'b0 ) begin
     sum_out <= 0 ;
  end
  else begin
     sum_out <=  next_sum_out ;
  end
end
//
endmodule
```

```
module add_shifter ( clk, rst_n, a_in, sum, avrg_out ) ;
//
parameter A_IN_BW = 4 ;    // a_in bit size
parameter SUM_IN_BW = 6 ;    // sum bit size
parameter SUM_OUT_BW = A_IN_BW ;    // sum_out bit size,
//              SUM_OUT_BW is equal to A_IN_BW because
//              lower (SUM_IN_BW - A_IN_BW) is shifted out
parameter SHIFT_CNT = SUM_IN_BW - A_IN_BW ;
//
input clk, rst_n ;
input [A_IN_BW-1:0] a_in ;
input [SUM_IN_BW-1:0] sum ;
output [A_IN_BW-1:0] avrg_out ;
//
wire clk, rst_n ;
wire [A_IN_BW-1:0] a_in ;
wire [SUM_IN_BW-1:0] sum ;
reg  [A_IN_BW-1:0] avrg_out ; // FF
// internal variables
wire [SUM_IN_BW:0] temp_add ; // 1 bit extended for future safty.
wire [A_IN_BW-1:0] next_avrg_out ;
```

```
//
//  add operation will not result in more than SUM_IN_BW
// LHS bit size must be larger than that of RHS to hold carry bit
assign temp_add = sum + a_in ;
assign next_avrg_out = temp_add >> SHIFT_CNT ;  // divide by 4
//
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n==1'b0 ) begin
     avrg_out <= 0 ;
  end
  else begin
     avrg_out <= next_avrg_out ;
  end
end
//
endmodule
```

file name: avrg.v

(8) test bench

```
module test_avrg  ;
parameter HF_CYCL=10 ;
parameter CYCL = HF_CYCL *2 ;
parameter A_IN_BW = 8 ;  // note bit size is not 4, modify avrg module to work correctly
event sig_chng_timing ;   // this is a event synchtonized to clock, half cycle after clock rise.
reg rst_n, clk ;
reg [A_IN_BW-1:0] a ;
wire[A_IN_BW-1:0] a_avrg ;
wire valid ;
// connect signals to game
avrg avrg_01 ( .rst_n( rst_n ), .clk( clk ),
              .a_in( a ),
              .a_avrg( a_avrg  ),
              .avrg_valid( valid )
             ) ;
// connection end
always begin
   clk = 0 ;  #HF_CYCL ;
   clk = 1 ;  #HF_CYCL ;
end

always @ ( negedge clk )  begin
    ->  sig_chng_timing ;
end

always @ ( posedge clk ) begin      // clock generator
      #2 $strobe("t=%d, rst=%b, clk=%b, a= %d, ff=%d,  %d,  %d, temp_add=%d, a_avrg=%d, v=%b",
              $stime, rst_n, clk, a,
              avrg_01.a_previous,
              avrg_01.sum_1,
              avrg_01.sum_2,
              avrg_01.add_shifter_01.temp_add,
              a_avrg, valid           ) ;
end
```

```
initial begin
   a= 0 ;
  #1 ;
  while ( rst_n == 1'b0 ) begin
      #1 ;
 end;
  @ ( sig_chng_timing ) a= 8 ;
  @ ( sig_chng_timing ) a= 16 ;
  @ ( sig_chng_timing ) a= 32 ;
  @ ( sig_chng_timing ) a= 64 ;
  @ ( sig_chng_timing ) a= 0 ;
  @ ( sig_chng_timing ) a= 0 ;
  @ ( sig_chng_timing ) a= 0 ;
  @ ( sig_chng_timing ) a= 0 ;
  @ ( sig_chng_timing ) a= 128 ;
  @ ( sig_chng_timing ) a= 128 ;
  @ ( sig_chng_timing ) a= 128 ;
  @ ( sig_chng_timing ) a= 128 ;
  @ ( sig_chng_timing ) a= 0 ;
  @ ( sig_chng_timing ) a= 0 ;
  @ ( sig_chng_timing ) a= 0 ;
  @ ( sig_chng_timing ) a= 0 ;
  @ ( sig_chng_timing ) a= 0 ;
  @ ( posedge clk ) #HF_CYCL  $finish ;
end
initial begin // give value to control variable
    rst_n = 1'b0 ;
    #(CYCL*2 ) rst_n = 1'b1 ;
    #(CYCL * 50) $finish ;
end
endmodule
```

file name: test_avrg.v

(9) Bit size extension.

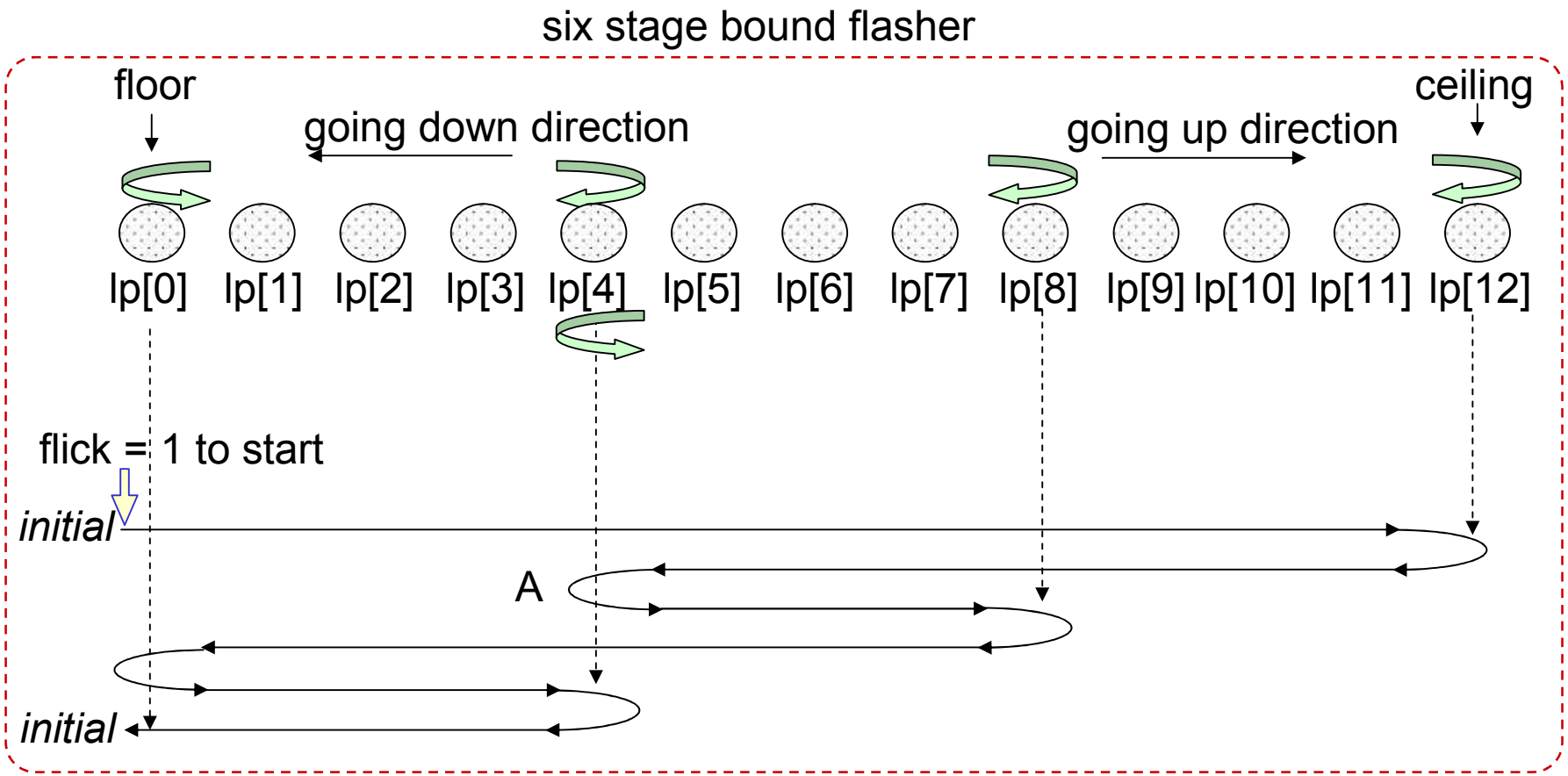
Extend the bit size to 8, not 4, by modifying the code as shown below and test the logic if it works correctly for 8-bit data.

```
module avrg ( clk, rst_n, a_in, a_avrg, avrg_valid  ) ;
//
parameter A_IN_BW = 8 ;    // a_in bit size
```

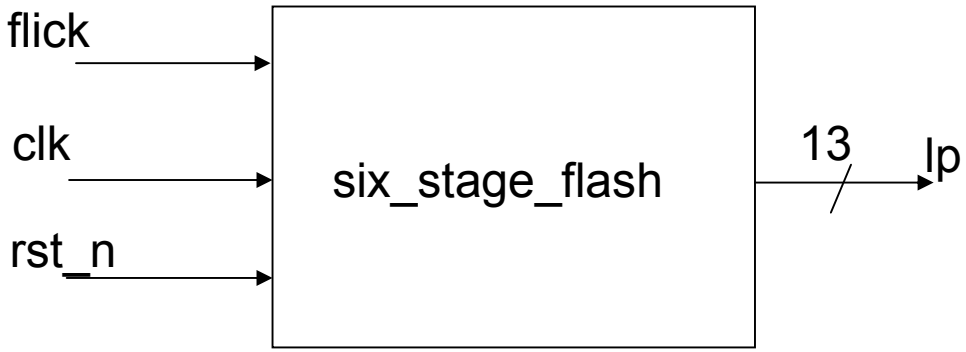
⋮

```
module test_avrg  ;
parameter HF_CYCL=10 ;
parameter CYCL = HF_CYCL *2 ;
parameter A_IN_BW = 8 ;
defparam avrg_01.A_IN_BW=A_IN_BW ;
,,,
,,
wire[A_IN_BW-1:0] a_avrg ;
wire valid ;
// connect signals to game
avrg avrg_01 ( .rst_n( rst_n ), .clk( clk ),
```

Ex 4-10. 6-stage bund flasher: Write a module named six_stage_flash, which outputs 13-bit signal lp as shown below. Right after asynchronous active low reset signal, rst_n, is negated, lp must be 0. When input signal flick becomes 1, lp[0] must be turned on. After that at each clock rise time, on-bit position must shift toward MSB. The input signal flick is valid only when lp is 0. On-bit position must go up and down as shown below.



In your code, you must use parameters for bit size of lp, turning points, initial values, etc. Your logic must work correctly by changing these parameters when the specification changes such as the second turning point, A, is not bit 4, but bit 2.
Use meaningful and easy to understand names for those parameters.



No sample code is given for this exercise.

Chapter 5. RTL coding style suitable for optimization

In general synthesis tools are getting more intelligent and becomes capable of generating optimized net list from RTL code which may not be written in recommended styles.
However, there still are weak points in synthesis tools therefore knowing what style result in less optimized gate net list and what style result in more optimized one will help writing good RTL code.

In this chapter, we will see several examples which result in easy to optimize gate diagram and hard to optimize gate diagram.

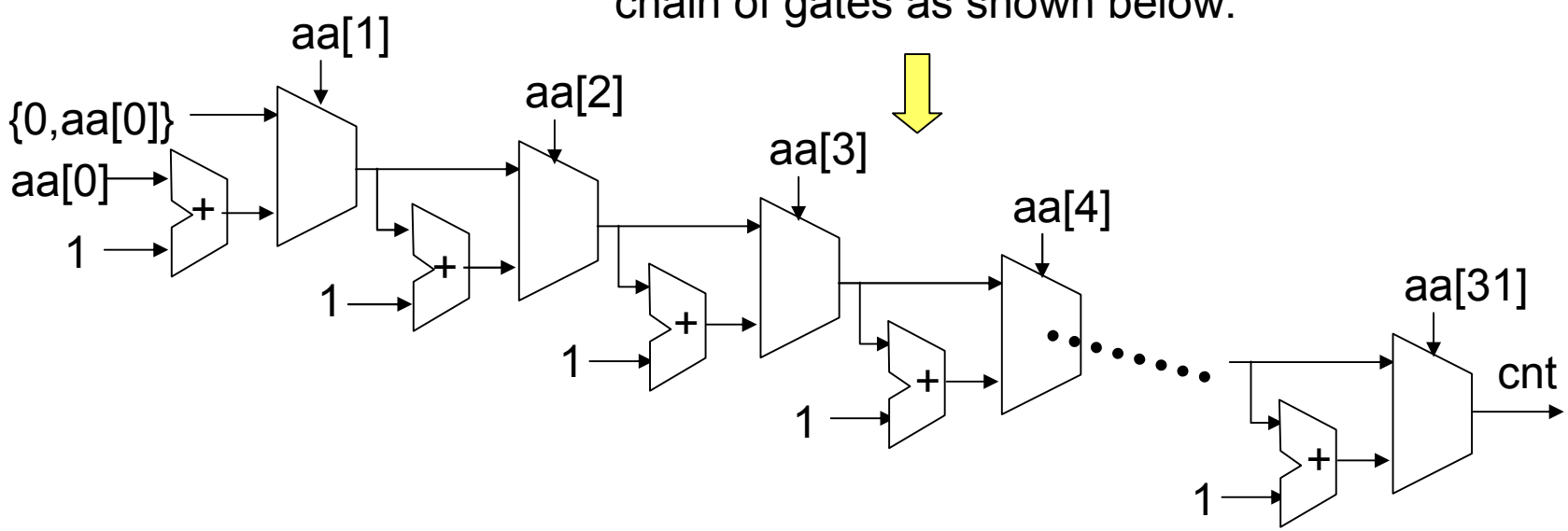
However, do not think today's weak point of synthesis tools is still their tomorrow's weak point.

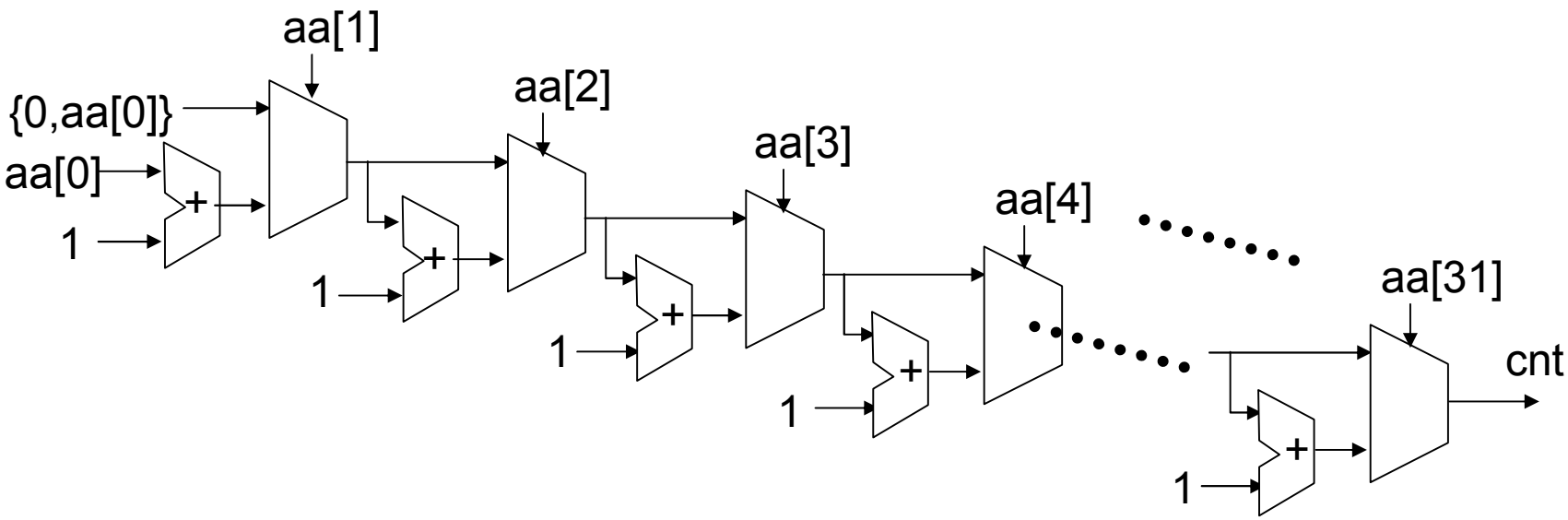
Ex 5-1. For-loop and if statement: Write a module which counts the number of on bit in a given 32-bit input signal sig_in and output the count onto an 6-bit output signal on_cnt.

To count on-bit in a signal aa, we can apply the logic below. The code below is synthesizable.

```
integer k ;
cnt = 0 ;
for ( k = 0 ; k < 32 ; k = k +1 ) begin
  if ( aa[k] == 1'b1 ) begin
      cnt = cnt +1 ;
  end
end
```

However, when synthesized, the code result in a very long serial chain of gates as shown below.





This gate diagram can not be minimized very much even if we apply optimization.

	Before optimization	After optimization		
		min area	min delay	min both
Area	1	0.77	0.66	0.63
Delay	1	0.77	0.32	0.32

The values are normalized to 1.

This shows that the coding style on the previous page is not suitable for synthesis and optimization.

Next, let's think logic to count on-bit without using for loop.

Divide 32-bit data into 8 4-bit data and apply a counting logic to each of 8 4-bit data and sum up them to calculate on-bit count for a given 32-bit data as shown below.

```
function [2:0] on_in_4 ;
input [3:0] aa ;
begin
  on_in_4 = aa[3] + aa[2] + aa[1] + aa[0] ;
end
endfunction

assign temp_31_24 = on_in_4(sig_in[31:28]) + on_in_4(sig_in[27:24])  ;
assign temp_23_16 = on_in_4(sig_in[23:20]) + on_in_4(sig_in[19:16])  ;
assign temp_15_8 = on_in_4(sig_in[15:12]) + on_in_4(sig_in[11:8])
assign temp_7_0 = on_in_4(sig_in[7:4]) + on_in_4(sig_in[3:0]) ;

assign temp_31_16 = temp_31_24 + temp_23_16 ;
assign temp_15_0 = temp_15_8 + temp_7_0 ;
assign on_cnt = temp_31_16 + temp_15_0 ;
```

The coding style in the previous page is suitable for optimization, and we can get good result as shown below.

		Before	After optimization		
			min area	min delay	min both
for (k=0; k<32; K=k+1) if (a[k]) cnt=cnt+1	Area	1	0.77	0.66	0.63
	Delay	1	0.77	0.32	0.32
function cnt = a[3]+a[2]+a[1]+a[0]; endfunction	Area	0.22	0.20	0.46	0.43
	Delay	0.17	0.16	0.11	0.11

The values are normalized to 1.

Next, let's take a different method to count on-bit. Again 32-bit data is divided into 8 4-bit data, but counting uses a different method as shown below.

```
function [2:0] on_in_4 ;
input [3:0] aa ;
begin
case ( aa[3:0] )
  4'b0000 : begin on_in_4 = 0 ; end
  4'b1000, 4'b0100, 4'b0010,
  4'b0001 : begin on_in_4 = 1 ; end
  4'b1100, 4'b1010, 4'b1001, 4'b0110, 4'b0101,
  4'b0011 : begin on_in_4 = 2 ; end
  4'b1110, 4'b1101, 4'b1011,
  4'b0111 : begin on_in_4 = 3 ; end
  4'b1111 : begin on_in_4 = 4 ; end
endcase
end
endfunction
```

This looks like huge logic and a little bit complicated, but it is not a big problem for synthesis tools. They can handle this style very well and generate compact gate diagram as small and as fast as on_in_4 = aa[3] + aa[2] + aa[1] + aa[0] .

The code in the previous page result in the bottom lines of the table below. It is known that the style can get good gate diagram after optimization. We can see that even if a source code looks huge, if the style is suitable for optimization, we can get compact gate diagram from such large code.

		Before	After optimization		
			min area	min delay	min both
for (k=0; k<32; K=k+1) if (a[k]) cnt=cnt+1	Area	1	0.77	0.64	0.62
	Delay	1	0.77	0.32	0.32
function cnt = a[3]+a[2]+a[1]+a[0]; endfunction	Area	0.22	0.20	0.46	0.43
	Delay	0.17	0.16	0.11	0.11
function case (a) 4‘b0000 : cnt=o ; 4'b1000, ,,	Area	1.34	0.24	0.50	0.49
	Delay	0.17	0.16	0.10	0.16

The values are normalized to 1.

By the way,

```
begin
 cnt = a[3]+a[2]+a[1]+a[0] ;
end
```

can be extended to the following expression.

```
begin
 cnt = a[31]+ a[30]+ a[29]+ a[28]+ a[27]+ ,,,,,,,,,+ a[2]+a[1]+a[0] ;
end
```

The above code can be mapped into the following for loop.

```
integer k ;
cnt = 0 ;
for ( k = 0 ; k < 32 ; k = k +1 ) begin
  cnt = cnt + a[k] ;
end
```

Now, let's see what the synthesis result of this code shall be.

The code in the previous page result in the bottom lines of the table below.

```
cnt = 0 ;
for ( k = 0 ; k < 32 ; k = k +1 ) begin
  cnt = cnt + a[k] ;
end
```

		Before	After optimization		
			min area	min delay	min both
for (k=0; k<32; K=k+1) if (a[k]) cnt=cnt+1	Area	1	0.77	0.64	0.62
	Delay	1	0.77	0.32	0.32
function cnt = a[3]+a[2]+a[1]+a[0]; endfunction	Area	0.22	0.20	0.46	0.43
	Delay	0.17	0.16	0.11	0.11
function case (a) 4'b0000 : cnt=o ; 4'b1000, ,,	Area	1.34	0.24	0.50	0.49
	Delay	0.17	0.16	0.10	0.16
for (k=0; k<32; K=k+1) cnt=cnt+1	Area	0.18	0.19	0.50	0.48
	Delay	0.16	0.16	0.10	0.10

Not good!!

Better, same level

The values are normalized to 1.

Coding example.

```
module cnt_on_bit ( sig_in, on_cnt ) ;
input [31:0] sig_in ;
output [5:0] on_cnt ;
wire [31:0] sig_in ;
wire [5:0] on_cnt ;
wire [3:0] temp_31_24, temp_23_16, temp_15_8, temp_7_0 ;
wire [4:0] temp_31_16, temp_15_0;

assign temp_31_24 = cnt(sig_in[31:28]) + cnt(sig_in[27:24]) ;
assign temp_23_16 = cnt(sig_in[23:20]) + cnt(sig_in[19:16]) ;
assign temp_15_8 = cnt(sig_in[15:12]) + cnt(sig_in[11:8]) ;
assign temp_7_0 = cnt(sig_in[7:4]) + cnt(sig_in[3:0]) ;

assign temp_31_16 = temp_31_24 + temp_23_16 ;
assign temp_15_0 = temp_15_8 + temp_7_0 ;
assign on_cnt = temp_31_16 + temp_15_0 ;
```

Note that LHS has one bit larger bit width than that of RHS to catch carry bit.

Case style is taken for this example.

```
function [2:0] cnt ;
input [3:0] aa;
begin
 case ( aa )
  4'b0000 : begin cnt = 0 ; end
  4'b1000, 4'b0100, 4'b0010,
  4'b0001 : begin cnt = 1 ; end
  4'b1100, 4'b1010, 4'b1001,
  4'b0110, 4'b0101,
  4'b0011 : begin cnt = 2 ; end
  4'b1110, 4'b1101, 4'b1011,
  4'b0111 : begin cnt = 3 ; end
  4'b1111 : begin cnt = 4 ; end
  default : begin cnt = 3'bxxx ; end
endcase
end
endfunction

endmodule
```

file name: cnt_on_bit.v

Ex 5-2. On-bit-block counter: Write a module which counts a number of on-bit blocksin a given 32-bit input signal sig_in and output the count onto an 6-bit output signal on_cnt.
Continuously-on-bits are counted as one block. An isolated on-bit whose adjacent bit is 0 must not be counted as one block.
If 32-bit input is 32‘b1100_0101_0101_1101_1111_0000_1010_1110 then the output must be 4 (a block 11,another block 1_11,another block 1_1111, and the other block 111, total 4 blocks.).
If the input is 32‘hffff_ffff_0000_0000 then the output must be 1.

To build up your RTL programming power, "think and try oneself" is the best.
However "Simulation result OK does not mean the code is OK."
Therefore, you must show or submit your code to elder generation engineers or supporters to see if your code is really OK or not.

No sample answer is given for this exercise.

Chapter 6. Advanced exercises

There are many solutions for these exercises.
Depending on what criteria, power consumption, speed, or area size, we think important, the best implementation is different.

For the exercises in this chapter, first try to write a code without looking into sample answers.

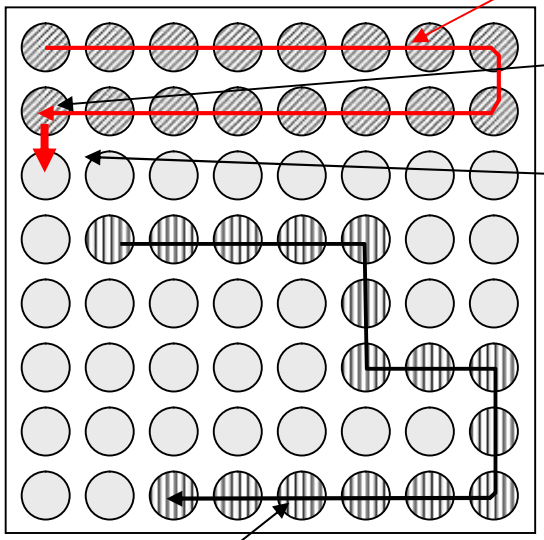
Work flow for solutions

(1) Investigate the function of the system.
(2) Find what is essential to the function,
(3) Write a design document including a state transition table,
(4) Write a RTL code for the target module,
(5) Write a test bench to test the target module, and
(6) Run the test bench with the target module to check the logic,
(7) Study a sample code and run it.

Some exercises has sample test benches. Your code is not OK if it is rejected by the sample test bench.

Ex 6-1. Snake game : Design a module to control a snake moving around in 8x8 lamp field. Moving direction is controlled by 4-bit ctl_dir input. Game shall stop when the snake hits a wall or its body.

8 x 8 lamp field



initial position of the snake body

initial position of the head

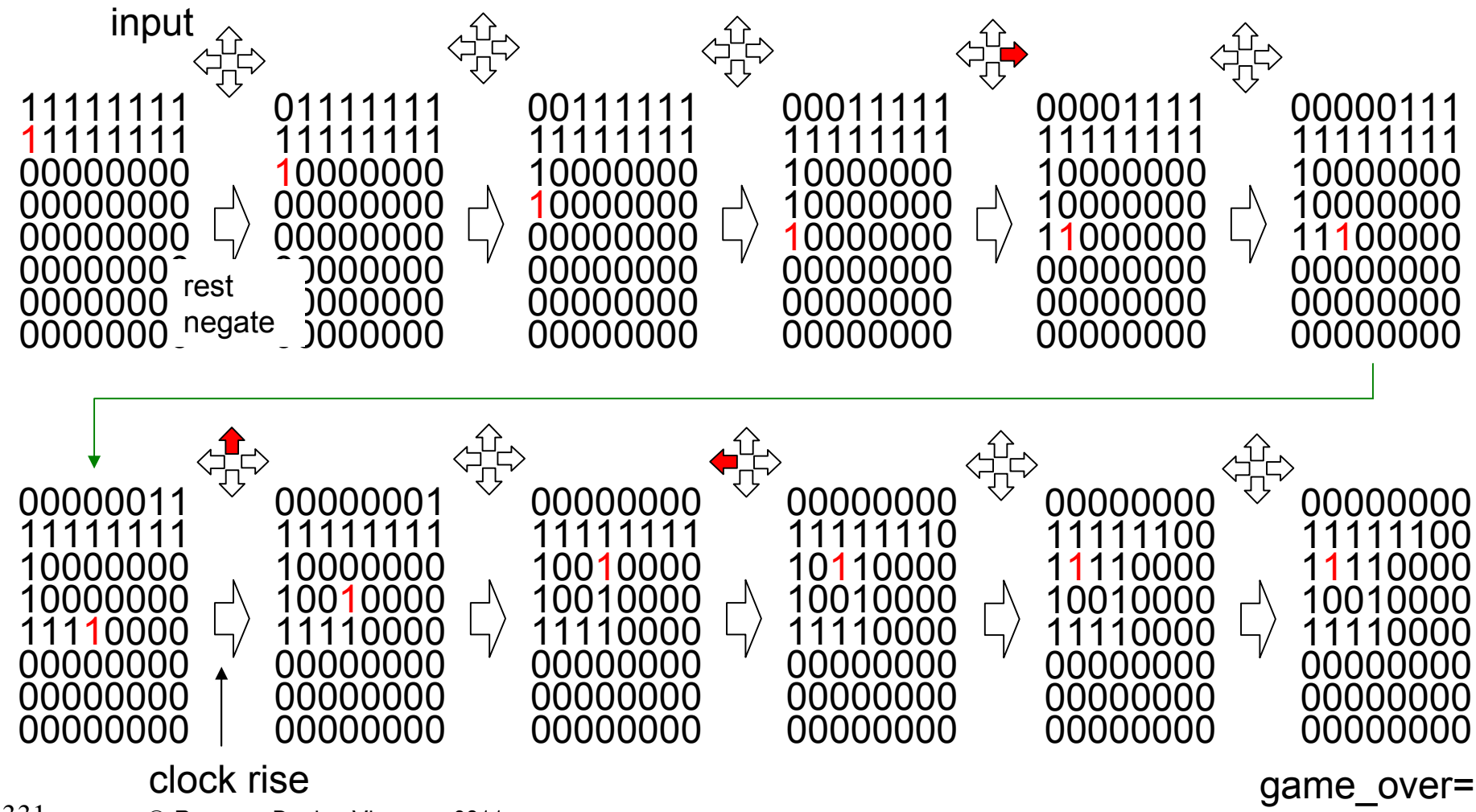
When reset signal is negated, the snake begins to move downward. At each rising edge of clock, it moves one lamp at a time. The snake length must always be 16 lamps.

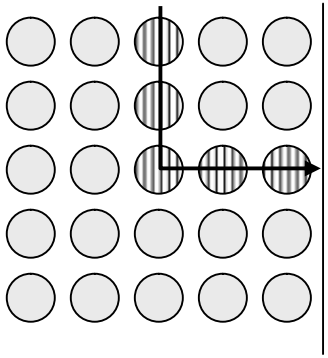
A player has to control the moving direction of the head by 4-bit control input ctl_dir.

The game ends if the head can not move any more. The head can neither go out of the lamp field nor cross its body.

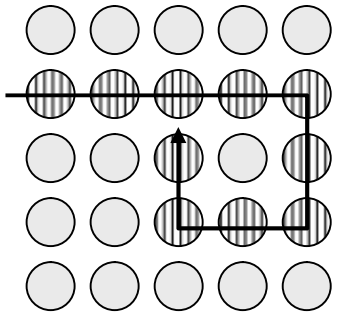
This snake is just for example. There shall be only one snake in the lamp field.

An example of the movement is shown below.
1 means lamp on, and 0 means off. The on lamps represent shake body.
It moves one lamp at clock rise time as shown below.



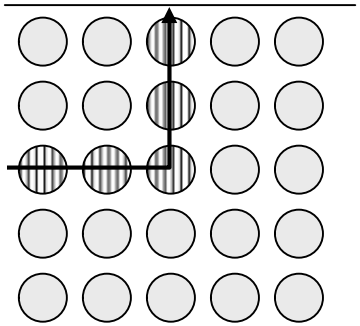


game ends unless the head is controlled to go up or down.



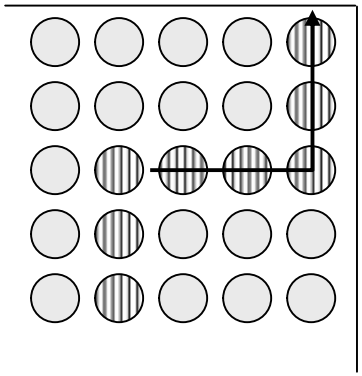
game end unless the head is controlled to go left or right.

(go_right will end up “game over” in the next clock cycle in this case.)

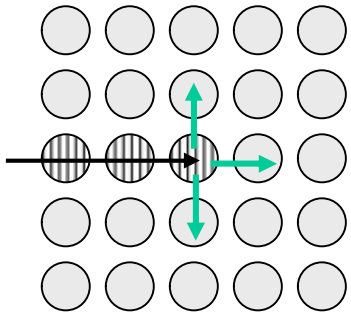


game ends unless the head is controlled to go right or left.

The head can move at most three directions as shown below. It can not go back. If go backward input given, the game is over.

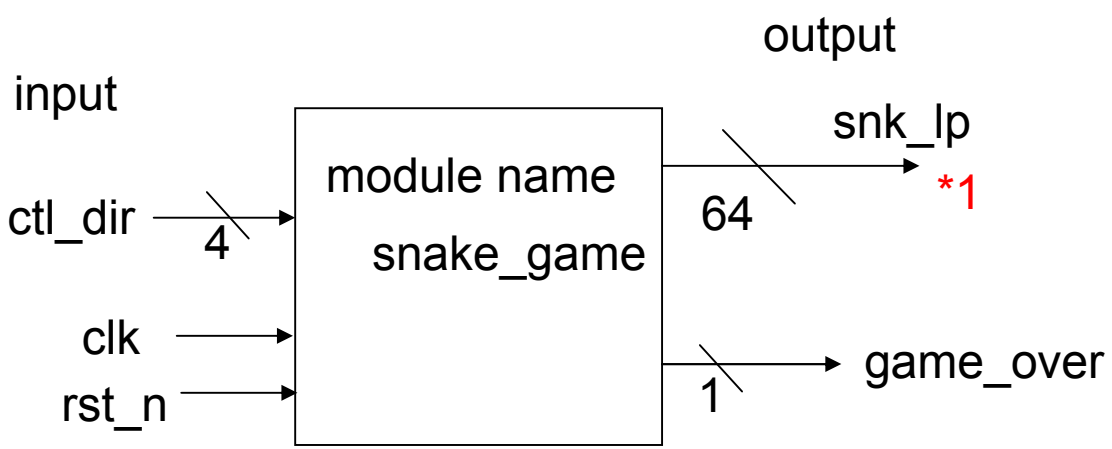


game end unless the head is controlled to go left.



→ : snake's current moving direction

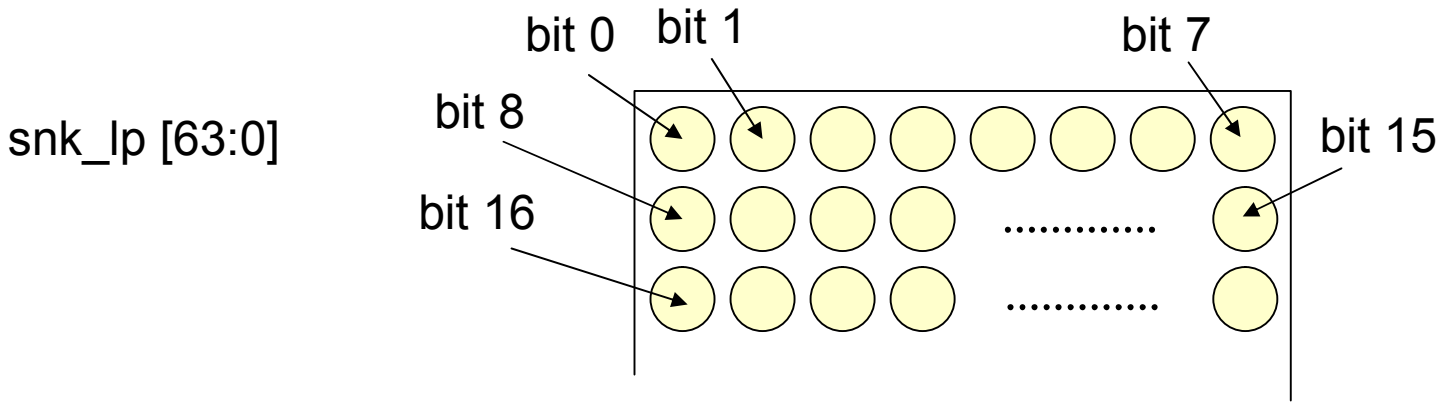
→ : movable direction



When game_over is asserted, the snake must stop moving and keep its position until reset asserted.

game_over does not becomes 0 once it is asserted unless reset asserted.

*1: In Verilog1995, array can not be used as a port. Therefore, output is defined as 8x8=64-bit length vector data. Each bit represents one lamp in the lamp field. Bit 0 corresponds to upper right corner lamp, bit 7 to upper left corner, bit 56 to bottom right corner, and bit 63 to bottom left corner lamp as shown below.



Detailed specification

(1) The initial position of the snake is: its head at bit-8 (snk_lp[8]), neck at bit-9 (snk_lp[9]), and the rest of the body at bit-10, bit-11, bit-12, bit-13, bit-14, bit-15, and bit-7, bit-6, bit-5, bit-4, bit-3, bit-2, bit-1, and end at the tail bit-0 (snk_lp[0]).

(2) When moving, the head can move only in one direction at a time, it is up, down, left or right. The rest of the body must go through the point where its head passed. It can not slide aside. All the body parts must follow the path which the preceding body part passed. All the body move one lamp at clock rise time.

(3) If there is a field edge or its body ahead and moving into the current direction will cause the head out of the field or collide into its body, game_over shall be asserted and the snake must stop at current position. Once asserted, game_over can be reset only by reset signal, rst_n.

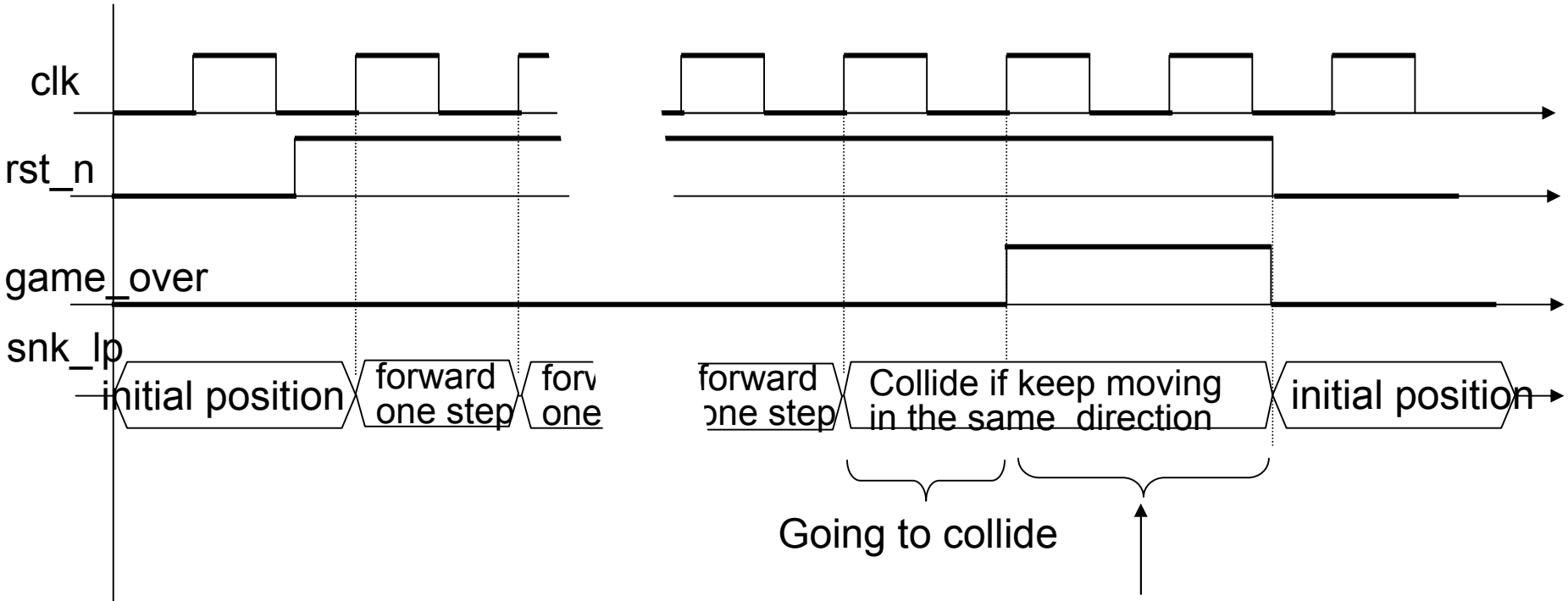
Detailed specification (continued)

(4) While reset asserted, the snake must be at the initial position and game_over shall be 0. At the first rise edge of the clock after reset negated, the snake must start moving automatically.

(5) The moving direction is controlled by the 4-bit input ctl_dir. Bit-3 of ctl_dir represents up direction, bit-2 down, bit-1 left, and bit-0 right, respectively. When all bits are 0, the moving direction in the previous clock cycle is used. If any one bit of ctl_dir is 1, the moving direction is a direction represented by the bit. No more than 1 bit of ctl_dir shall be on at the same time.

(6) Right after reset negated, the moving direction must be down, unless directed by ctl_dir. (Note, that if ctl_dir is not 4'b0000 nor 4'b0100, game_over will be asserted, because the snake can not move in the direction other than down.)

clk : 1-bit input signal, clock
rst_n : 1-bit input signal, asynchronous reset, active low
ctl_dir : 4-bit input signal, active high, synchronous to the clock
game_over : 1-bit output signal, active high, synchronous to the clock
snk_lp : 64-bit output signal, active high, synchronous to the clock



If keep moving into the current direction, the snake will collide its body or go out of the field. However, it is stopped with game_over asserted.

Note on the implementation

There are many ways on implementation. You can choose whichever you like.
However, game_over and snk_lp must be FF direct output.
You must not make game_over nor snk_lp as output of combinational logic.

Work flow for solutions

(1) Investigate the function of the system.
(2) Find what is essential to the function,
(3) Write a design document including a state transition table,
(4) Write a RTL code for the target module,
(5) Run the automatic test bench on the next pages with the target module you created.
If your code is rejected, correct your code.

Copy and paste the test bench module on the next pages into you PC and run it with your snake_game module to see if your code is OK or Not.

If your code is OK, it will output the following message on to your terminal screen.

```
pass test13,
 hit body from left
test end no error detected
```

A sample test bench, automatic tester

```
`timescale 1ns/100ps
module test_snake_game  ;
parameter HF_CYCL = 5 ;
parameter CYCL = HF_CYCL * 2 ;
parameter FLD_BW = 8 ;
parameter NUM_LP = FLD_BW * FLD_BW ;
parameter DIR_BW = 4 ;
reg rst_n, clk ;
reg [DIR_BW-1:0] ctl_dir ;
wire game_over ;
wire[NUM_LP-1:0] snk_lamp ;
wire [FLD_BW-1:0] tmp_lp_0, tmp_lp_1, tmp_lp_2, tmp_lp_3,
                  tmp_lp_4, tmp_lp_5, tmp_lp_6, tmp_lp_7 ;
wire [FLD_BW-1:0] lp_0, lp_1, lp_2, lp_3, lp_4, lp_5, lp_6, lp_7 ;
reg show_lamp ; // control flag to show lamp on screen, if 0 lamps are not
                // displayed on the screen at each clock rise time
integer kk ; // for loop counter
//
assign { tmp_lp_7, tmp_lp_6, tmp_lp_5, tmp_lp_4,
         tmp_lp_3, tmp_lp_2, tmp_lp_1, tmp_lp_0 } = snk_lamp ;
function [FLD_BW-1:0] bit_swap;
input [FLD_BW-1:0] in_a ;
reg [FLD_BW-1:0] temp ;
begin
  for ( kk = 0; kk <= FLD_BW-1; kk = kk+1 ) begin
    temp[kk] = in_a[FLD_BW-kk-1] ;
  end
  bit_swap = temp ;
end
endfunction
```

Do not change this part.

temporary work for bit swap

Disintegrate 64-bit signal into 8 8-bit signals

bit swap function

```
assign lp_7 = bit_swap(tmp_lp_7) ;
assign lp_6 = bit_swap(tmp_lp_6) ;
assign lp_5 = bit_swap(tmp_lp_5) ;
assign lp_4 = bit_swap(tmp_lp_4) ;
assign lp_3 = bit_swap(tmp_lp_3) ;
assign lp_2 = bit_swap(tmp_lp_2) ;
assign lp_1 = bit_swap(tmp_lp_1) ;
assign lp_0 = bit_swap(tmp_lp_0) ;
//
// connect signals to game
snake_game snake_game_01 ( .clk( clk ), .rst_n( rst_n ),
                .ctl_dir(ctl_dir),
                .game_over(game_over), .snk_lp(snk_lamp)
               ) ;
// connection end
always @ ( posedge clk ) begin
#1.1 ;
if ( show_lamp == 1 ) begin
    $strobe (
"t=%0d, r=%b, ctl_dir=%b, game_ovr=%b¥n 0=%b ¥n 1=%b ¥n 2=%b ¥n 3=%b ¥n 4=%b ¥n 5=%b
¥n 6=%b ¥n 7=%b ¥n ",
        $stime, rst_n, ctl_dir, game_over,
        lp_0, lp_1, lp_2, lp_3, lp_4, lp_5, lp_6, lp_7 ) ;
end
end

always begin     // clock generator
  clk = 1'b0 ; # HF_CYCL ;
  clk = 1'b1 ; # HF_CYCL ;
end
```

bit swap operation

connect target module

Control display or not, by show_lamp flag

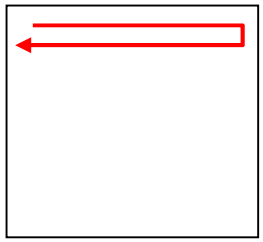
This must be written in one line.

clock generator

```
initial begin
show_lamp = 1 ;
// test1, run into wall at start
   ctl_dir = 4'b0010 ; // run to left wall to cause game over
   rst_n = 0 ;
$display("start test1,¥n run into wall at start") ;
 #(CYCL * 2 ) ;
 #CYCL  rst_n = 1 ;
 @( posedge clk )  #1.2 ;
 chcker ( 1'b1, 64'h00_00_00_00_00_00_ff_ff ) ;
$display("pass test1,¥n run into wall at start") ;
```

test1

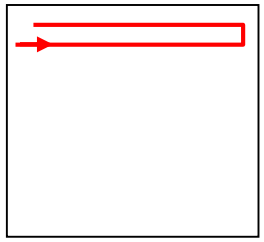
game over



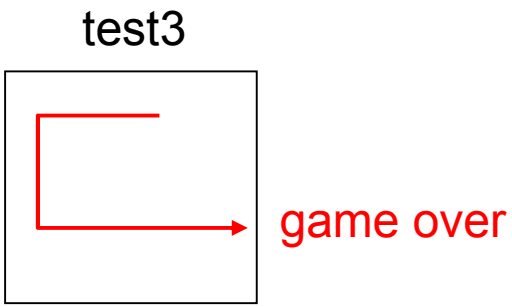
```
// test2, go back at start
   ctl_dir = 4'b0010 ; // go to left to cause game over
   rst_n = 0 ;
$display("start test2,¥n go back at start") ;
 #CYCL  rst_n = 1 ;
 @( posedge clk )  #1.2 ;
 chcker ( 1'b1, 64'h00_00_00_00_00_00_ff_ff ) ;
$display("pass test2,¥n go back at start") ;
```

test2

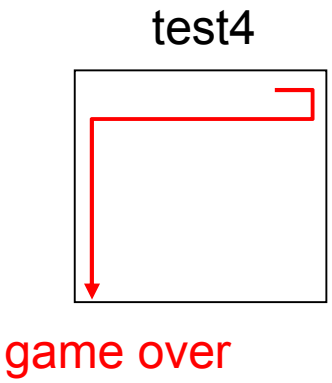
game over



```
// test3, game over running into right wall
   ctl_dir = 4'b0000 ;
   rst_n = 0 ;
$display("start test3,¥n game over running into right wall");
 #CYCL  rst_n = 1 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_00_00_01_ff_fe ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_00_01_01_ff_fc ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_01_01_01_ff_f8 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_01_01_01_01_ff_f0 ) ;
 ctl_dir = 4'b0001 ; // go right to hit wall
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_03_01_01_01_ff_e0 ) ;
 ctl_dir = 4'b0000 ; // keep go right to hit wall
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_07_01_01_01_ff_c0 ) ;
 ctl_dir = 4'b0000 ; // keep go right to hit wall
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_0f_01_01_01_ff_80 ) ;
 ctl_dir = 4'b0000 ; // keep go right to hit wall
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_1f_01_01_01_ff_00 ) ;
 ctl_dir = 4'b0000 ; // keep go right to hit wall
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_3f_01_01_01_7f_00 ) ;
 ctl_dir = 4'b0000 ; // keep go right to hit wall
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_7f_01_01_01_3f_00 ) ;
```

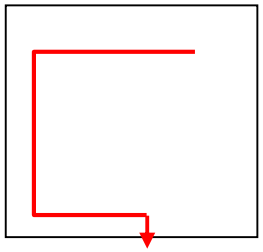


```
 ctl_dir = 4'b0000 ; // keep go right to hit wall
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_ff_01_01_01_1f_00 ) ;
 ctl_dir = 4'b0000 ; // keep go right to hit wall
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'h00_00_ff_01_01_01_1f_00 ) ;
 ctl_dir = 4'b0000 ; // keep go right to hit wall
$display("pass test3,¥n game over running into right wall");

// test4, game over running into bottom wall
 @( posedge clk ) #1.2 ;
   ctl_dir = 4'b0000 ;
   rst_n = 0 ;
$display("start test4,¥n game over running into bottom wall");
 #CYCL  rst_n = 1 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_00_00_01_ff_fe ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_00_01_01_ff_fc ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_01_01_01_ff_f8 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_01_01_01_01_ff_f0 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_01_01_01_01_01_ff_e0 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h01_01_01_01_01_01_ff_c0 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'h01_01_01_01_01_01_ff_c0 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'h01_01_01_01_01_01_ff_c0 ) ;
$display("pass test4,¥n game over running into bottom wall");
```



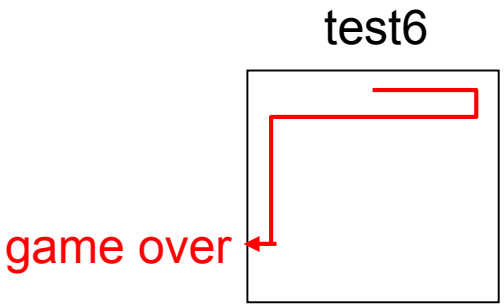
```
// test5, down and left and go into bottom wall
 @( posedge clk ) #1.2 ;
   ctl_dir = 4'b0000 ;
   rst_n = 0 ;
$display("start test5,¥n down and right and go into bottom wall") ;
show_lamp = 0 ;
 #CYCL  rst_n = 1 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_00_00_01_ff_fe ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_00_01_01_ff_fc ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_01_01_01_ff_f8 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_01_01_01_01_ff_f0 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_01_01_01_01_01_ff_e0 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h01_01_01_01_01_01_ff_c0 ) ;
show_lamp = 1 ;
 #3 ctl_dir = 4'b0001 ; // go right
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h03_01_01_01_01_01_ff_80 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h07_01_01_01_01_01_ff_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h0f_01_01_01_01_01_7f_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h1f_01_01_01_01_01_3f_00 ) ;
 #3 ctl_dir = 4'b0100 ; // go down to hit bottom wall
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'h1f_01_01_01_01_01_3f_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'h1f_01_01_01_01_01_3f_00 ) ;
$display("pass test5,¥n down and left and go into bottom wall") ;
```

test5



game over

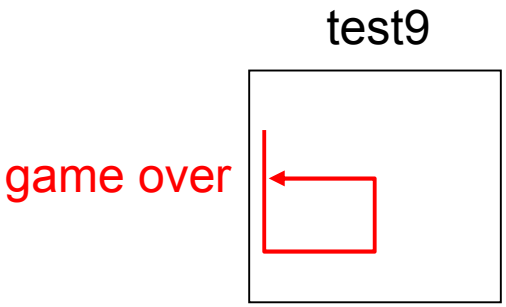
```
// test6, down and left and go into left wall
 @( posedge clk )  #1.2 ;
   ctl_dir = 4'b0000 ;
   rst_n = 0 ;
$display("start test6,¥n down and run into left wall") ;
show_lamp = 0 ;
 #CYCL  rst_n = 1 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_00_00_01_ff_fe ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_00_01_01_ff_fc ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_01_01_01_ff_f8 ) ;
show_lamp = 1 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_01_01_01_01_ff_f0 ) ;
 #3 ctl_dir = 4'b0010 ; // go left
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'h00_00_01_01_01_01_ff_f0 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'h00_00_01_01_01_01_ff_f0 ) ;
$display("pass test6,¥n down and run into left wall") ;
```



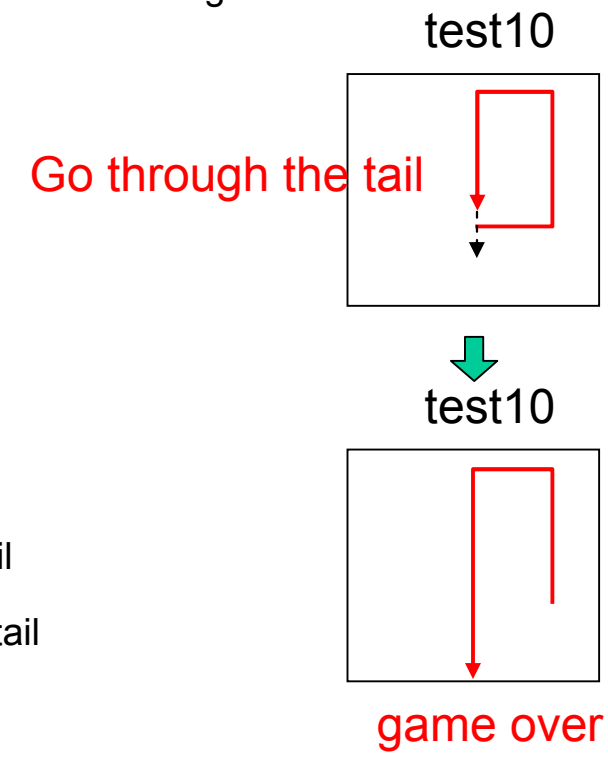
```
// test7, down and left and up and go into left wall
 @( posedge clk ) #1.2 ;
   ctl_dir = 4'b0000 ;
   rst_n = 0 ;
$display("start test7,¥n down and right and run into upt wall") ;
show_lamp = 1 ;
 #CYCL  rst_n = 1 ;
 #( CYCL * 6 ) ctl_dir = 4'b0001 ;
 #( CYCL * 7 ) ctl_dir = 4'b1000 ;
show_lamp = 1 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'hff_81_01_01_01_01_03_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'hff_81_81_01_01_01_01_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'hff_81_81_81_01_01_00_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'hff_81_81_81_81_00_00_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'hff_81_81_81_80_80_00_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'hff_81_81_80_80_80_80_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'hff_81_80_80_80_80_80_80 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'hff_81_80_80_80_80_80_80 ) ;
$display("pass test7,¥n down and right and run into up wall") ;
```

```
// test8, down and left and up and into top wall
 @( posedge clk ) #1.2 ;
   ctl_dir = 4'b0000 ;
   rst_n = 0 ;
$display("start test8,¥n down and rigth and up and run into right wall") ;
show_lamp = 0 ;
 #CYCL  rst_n = 1 ;
 #( CYCL * 6 ) #1.2 ctl_dir = 4'b0001 ;
 #( CYCL * 7 ) #1.2 ctl_dir = 4'b1000 ;
show_lamp = 1 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'hff_81_01_01_01_01_03_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'hff_81_81_01_01_01_01_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'hff_81_81_81_01_01_00_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'hff_81_81_81_81_00_00_00 ) ;
 #3 ctl_dir = 4'b0001 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'hff_81_81_81_81_00_00_00 ) ;
$display("pass test8,¥n down and left and up and into top wall") ;
```

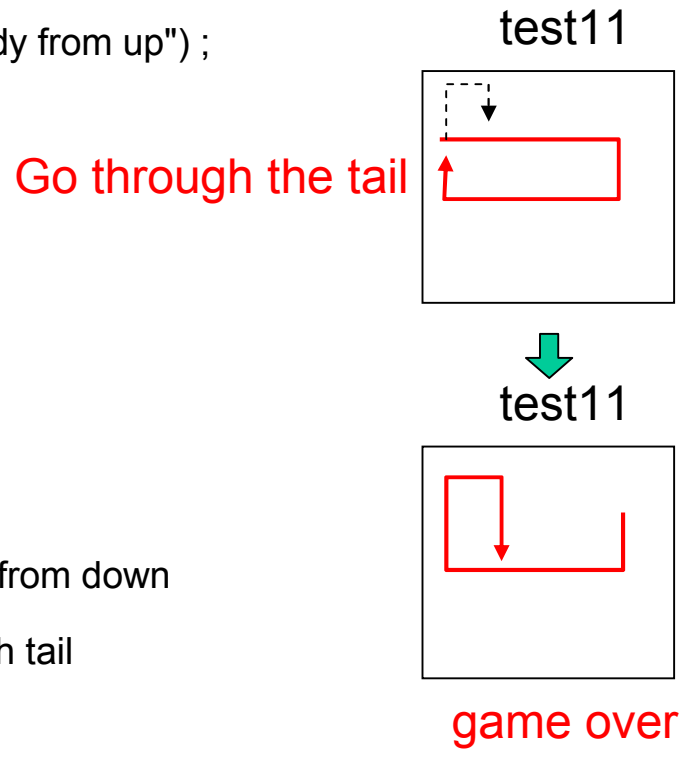
```
// test9, down and left and up and right and go into top wall
 @( posedge clk ) #1.2 ;
   ctl_dir = 4'b0000 ;
   rst_n = 0 ;
$display("start test9,¥n down and right and up and left and hit body") ;
show_lamp = 0 ;
 #CYCL  rst_n = 1 ;
 #( CYCL * 5 ) ctl_dir = 4'b0001 ;
 #( CYCL * 4 ) ctl_dir = 4'b1000 ;
 #( CYCL * 3 ) ctl_dir = 4'b0010 ;
show_lamp = 1 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_1f_11_11_19_01_07_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_1f_11_11_1d_01_03_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_1f_11_11_1f_01_01_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'h00_1f_11_11_1f_01_01_00 ) ;
$display("pass test9,¥n down and left and up and right and go into top wall") ;
```



```
// test10, down and left and up and right into right wall
 @( posedge clk ) #1.2 ;
   ctl_dir = 4'b0000 ;
   rst_n = 0 ;
$display("start test10,¥n down and right and up and left and down and through tail and runinto bottom wall") ;
show_lamp = 0 ;
 #CYCL  rst_n = 1 ;
 #( CYCL * 4 ) ctl_dir = 4'b0001 ;
 #( CYCL * 6 ) ctl_dir = 4'b1000 ;
 #( CYCL * 5 ) ctl_dir = 4'b0010 ;
 #( CYCL * 3 ) ctl_dir = 4'b0100 ;
show_lamp = 1 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_7f_40_40_40_48_78 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_7e_40_40_48_48_78 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_7c_40_48_48_48_78 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_78_48_48_48_48_78 ) ; // through tail
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_78_48_48_48_48_78 ) ; // head into tail
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_08_68_48_48_48_48_78 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h08_08_48_48_48_48_48_78 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'h08_08_48_48_48_48_48_78 ) ;
$display("pass test10,¥n down and right and up and left and down and through tail and runinto bottom wall") ;
```

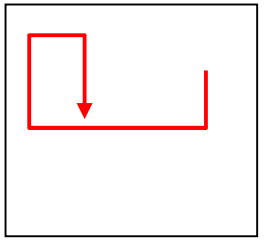


```
// test11,  through tail from down and hit body from up
 @( posedge clk ) #1.2 ;
   ctl_dir = 4'b0000 ;
   rst_n = 0 ;
$display("start test11,¥n through tail from down and hit body from up") ;
show_lamp = 0 ;
 #CYCL  rst_n = 1 ;
 #( CYCL * 1 ) ctl_dir = 4'b0001 ;
 #( CYCL * 6 ) ctl_dir = 4'b0100 ;
 #( CYCL * 2 ) ctl_dir = 4'b0010 ;
 #( CYCL * 4 ) ;
show_lamp = 1;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_7e_40_7f_03_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_7f_40_7f_01_00 ) ;
 #3 ctl_dir = 4'b1000 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_7f_41_7f_00_00 ) ; // hit tail from down
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_7f_41_7f_00_00 ) ; // through tail
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_00_00_7f_41_7d_01_00 ) ;
```



```
#3 ctl_dir = 4'b0001 ;
@( posedge clk ) #1.2 ;
chcker ( 1'b0, 64'h00_00_00_7f_41_79_03_00 ) ;
@( posedge clk ) #1.2 ;
chcker ( 1'b0, 64'h00_00_00_7f_41_71_07_00 ) ;
#3 ctl_dir = 4'b0100 ;
@( posedge clk ) #1.2 ;
chcker ( 1'b0, 64'h00_00_00_7f_41_65_07_00 ) ;
#3 ctl_dir = 4'b0000 ;
@( posedge clk ) #1.2 ;
chcker ( 1'b0, 64'h00_00_00_7f_45_45_07_00 ) ;
@( posedge clk ) #1.2 ;
chcker ( 1'b1, 64'h00_00_00_7f_45_45_07_00 ) ;  // hit body from up
@( posedge clk ) #1.2 ;
chcker ( 1'b1, 64'h00_00_00_7f_45_45_07_00 ) ;
$display("pass test11,¥n hrough tail from down and hit body from up") ;
```

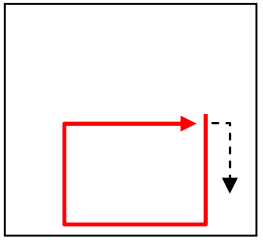
test11



game over

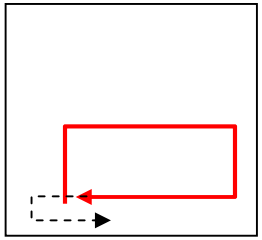
```
// test12, through tail from left, right and hit body from down
 @( posedge clk ) #1.2 ;
   ctl_dir = 4'b0000 ;
   rst_n = 0 ;
$display("start test12,¥n through tail from left, right and hit body from down") ;
 #CYCL  rst_n = 1 ;
show_lamp = 0 ;
 #( CYCL * 1 ) ctl_dir = 4'b0001 ;
 #( CYCL * 6 ) ctl_dir = 4'b0100 ;
 #( CYCL * 5 ) ctl_dir = 4'b0010 ;
 #( CYCL * 5 ) ctl_dir = 4'b1000 ;
 #( CYCL * 3 ) ctl_dir = 4'b0001 ;
 #( CYCL * 2 ) ;
show_lamp = 1;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h7e_42_42_5e_40_00_00_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h7e_42_42_7e_00_00_00_00 ) ; // hit tail from left
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h7e_42_42_7e_00_00_00_00 ) ; // through tail
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h7e_42_02_fe_00_00_00_00 ) ;
show_lamp = 0 ;
 #3 ctl_dir = 4'b0100 ;
 #( CYCL * 2 ) ctl_dir = 4'b0010 ;
 #( CYCL * 3 ) ;
show_lamp = 1 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h02_fa_82_fe_00_00_00_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_fe_82_fe_00_00_00_00 ) ; // hit tail from right
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_fe_82_fe_00_00_00_00 ) ; // through tail
```

test12



through tail

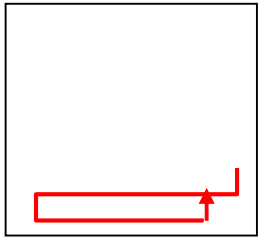
test12



through tail

```
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_ff_80_fe_00_00_00_00 ) ;
 #3 ctl_dir = 4'b0100 ;
show_lamp = 0 ;
 #( CYCL * 1 ) ctl_dir = 4'b0001 ;
 #( CYCL * 3 ) ctl_dir = 4'b0000 ;
show_lamp = 1 ;
 #( CYCL * 1 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h3f_ff_80_80_00_00_00_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h7f_ff_80_00_00_00_00_00 ) ;
 #3 ctl_dir = 4'b1000 ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'h7f_ff_80_00_00_00_00_00 ) ;
$display("pass test12,¥n through tail from left, right and hit body from down") ;
```

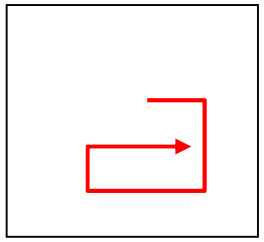
test12



game over

```
// test13, hit body from left
 @( posedge clk ) #1.2 ;
   ctl_dir = 4'b0000 ;
   rst_n = 0 ;
$display("start test13,¥n hit body from left") ;
 #CYCL  rst_n = 1 ;
show_lamp = 0 ;
 #( CYCL * 1 ) ctl_dir = 4'b0001 ;
 #( CYCL * 5 ) ctl_dir = 4'b0100 ;
 #( CYCL * 4 ) ctl_dir = 4'b0010 ;
 #( CYCL * 4 ) ctl_dir = 4'b1000 ;
 #( CYCL * 2 ) ctl_dir = 4'b0001 ;
 #( CYCL * 1 ) ;
show_lamp = 1;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_3e_22_2e_20_3c_00_00 ) ;
 @( posedge clk ) #1.2 ;
 chcker ( 1'b0, 64'h00_3e_22_3e_20_38_00_00 ) ; // hit body from left
 @( posedge clk ) #1.2 ;
 chcker ( 1'b1, 64'h00_3e_22_3e_20_38_00_00 ) ; // game over
$display("pass test13,¥n hit body from left") ;

 $display("test end no error detected") ;
 #1 $finish ;
end
```

test13



game over

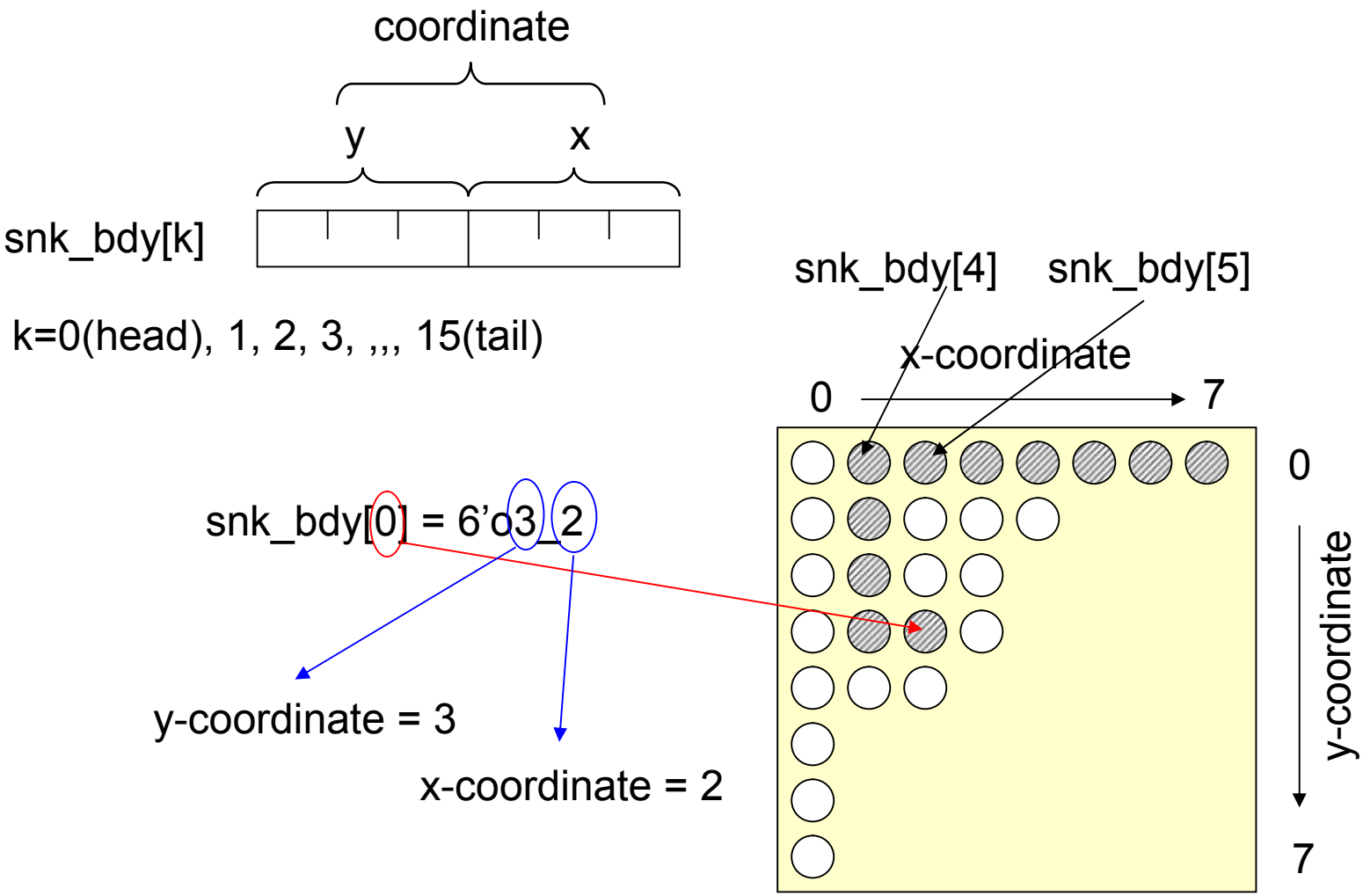
This task compares the output of the target module and the expected values.

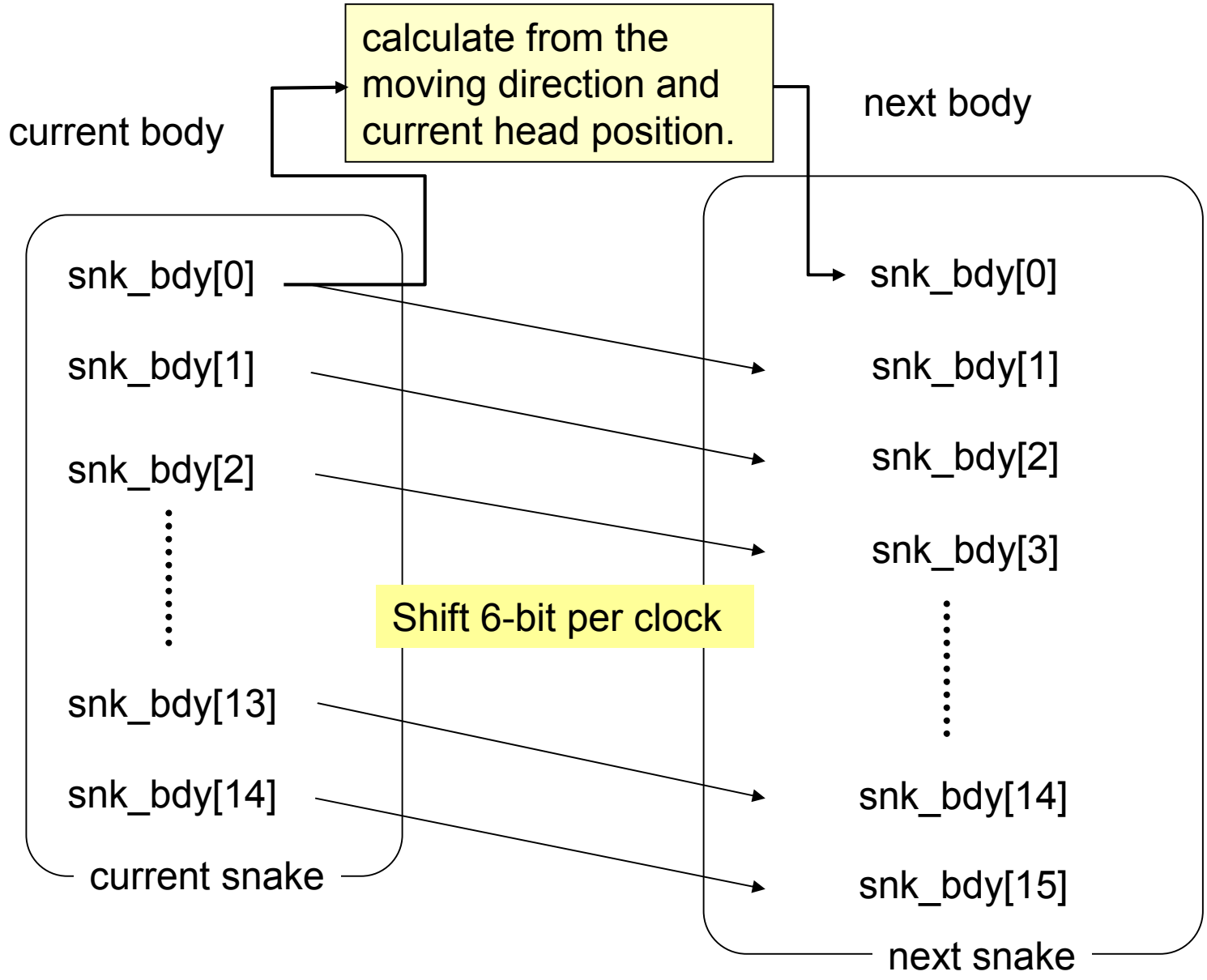
```
task chcker ;
input st ; // compare to game over
input [NUM_LP-1:0] pattn ; // compare to snk_lp

begin
 if( snk_lamp != pattn ) begin
    $display("bug stop, at t=%0d,¥n game_over=%b, snk_lamp=%h,¥n expected =%b, snk_lamp=%h¥n",
            $stime, game_over, snk_lamp, st, pattn ) ;
    #1 $finish ;
 end
 else begin
   if( st != game_over ) begin
      $display("bug stop, at t=%0d,¥n game_over=%b, snk_lamp=%h,¥n expected =%b, snk_lamp=%h¥n",
            $stime, game_over, snk_lamp, st, pattn ) ;
      #1 $finish ;
   end
 end
end
endtask

endmodule

`include "snake_game.v"
```

file name: test_snake_game.v

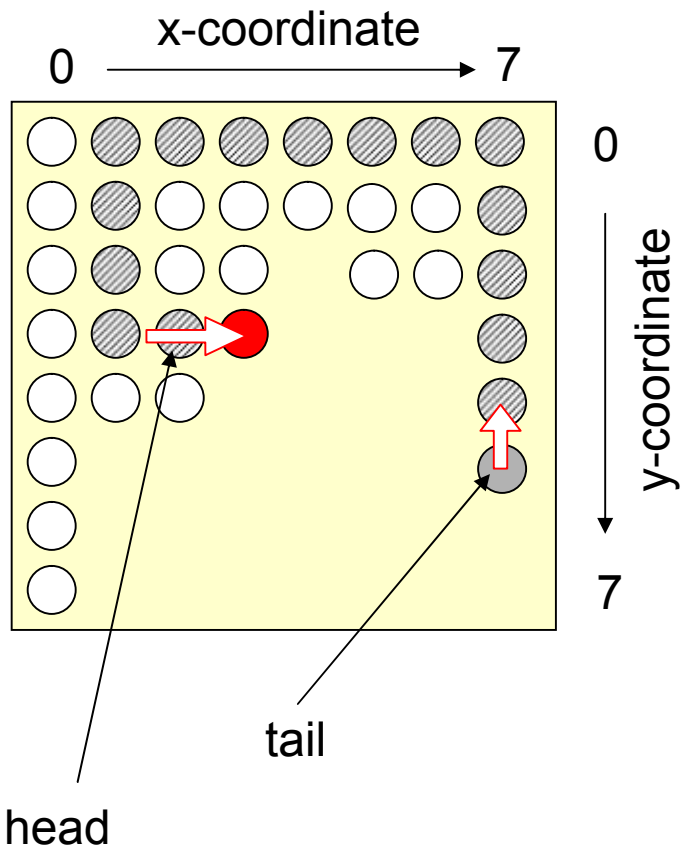
A sample target code

(1) How to describe the snake body





(2) How to control on/off of lamps



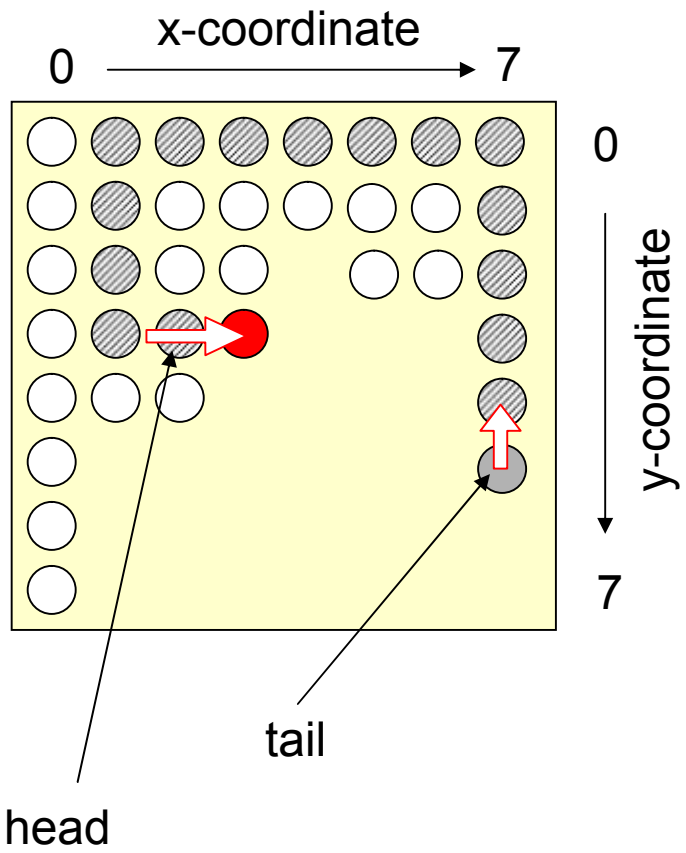
(a) Turn on one lamp in the direction of head movement.

(b) Turn off the lamp at the tail.

(c) If the tail is in the direction of head movement, then keep it on.

(d) Keep other lamps on and off as they are.

(3) How to check collision



(a) Head goes out of bound.

moving up ⇨ y-coordinate is 0

moving down ⇨ y-coordinate is 7

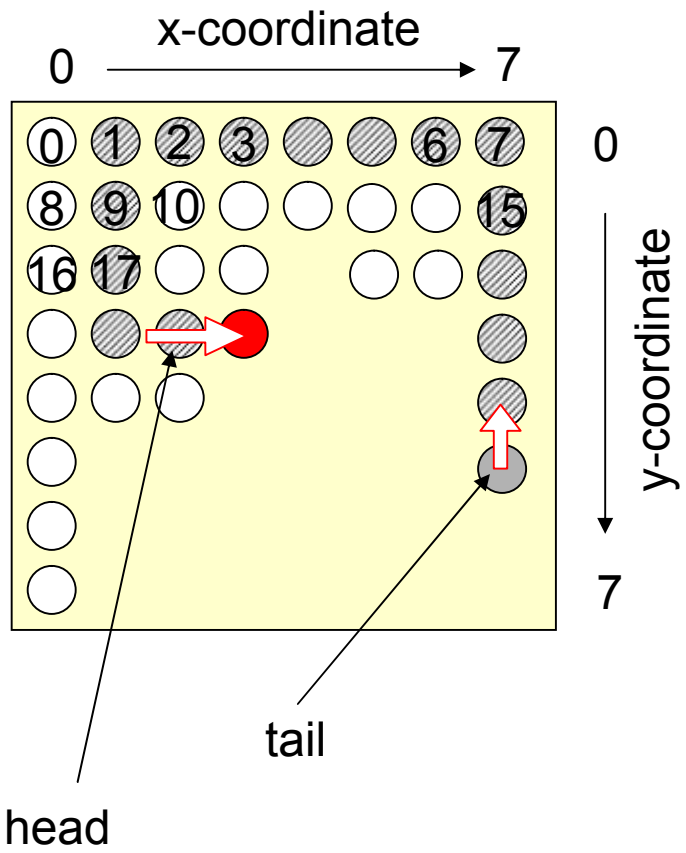
moving right ⇨ y-coordinate is 7

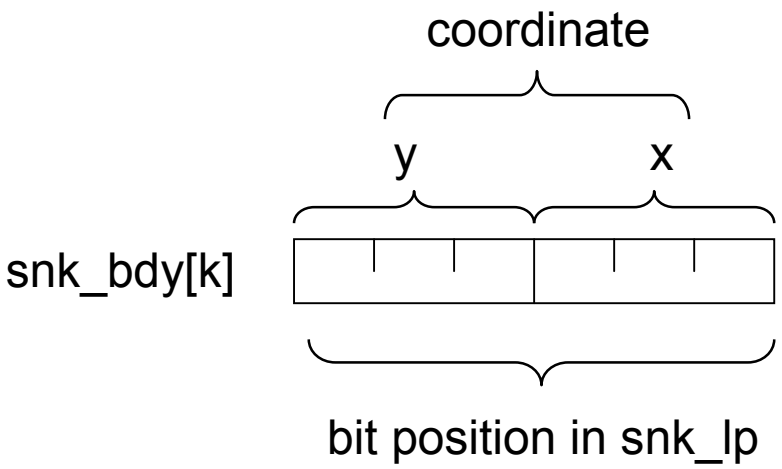
moving left ⇨ x-coordinate is 0

(b) Head hit the body except the tail.

A lamp in the head moving direction is already on. If it is a tail then it is not a collision because the tail will move to another position at the next clock cycle.

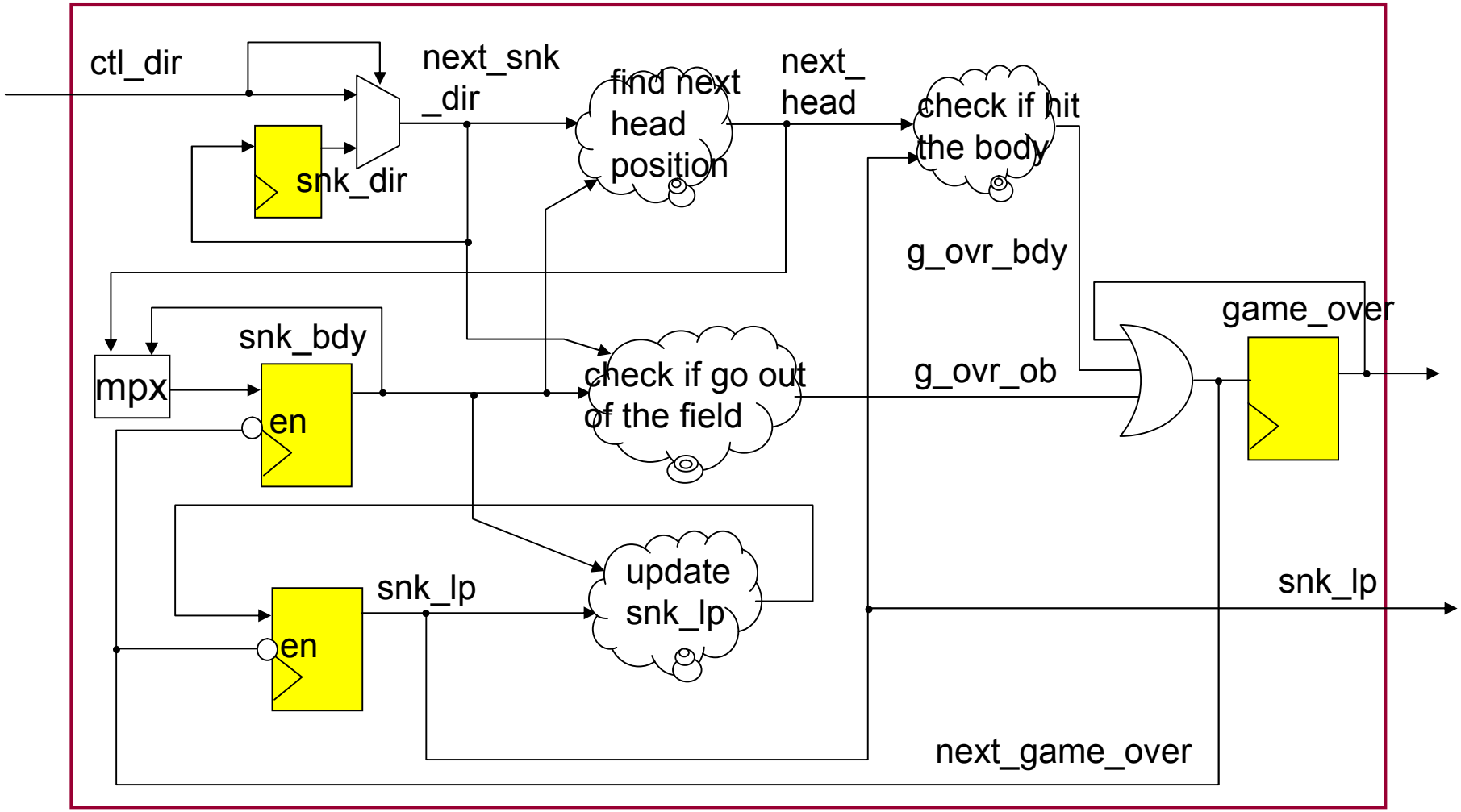
(4) lamp field FF representation





snk_lp[snk_bdy[0]] is a FF corresponding the snake head. And snk_lp[snk_bdy[15]] is a FF corresponding the snake tail.

The target module structure



A sample target code

```
//  snake game  module
//  (c) K. Hayashi, 2010
//
module snake_game ( clk, rst_n, ctl_dir, game_over, snk_lp ) ;
//
parameter FLD_BW = 8 ;
parameter LP_BW = FLD_BW * FLD_BW ;
parameter POSI_BW = 3 ;
parameter SNK_LNG = 16 ;
parameter SNK_BDY_BW = POSI_BW * 2 * SNK_LNG ;
parameter DIR_BW = 4 ;
parameter INTL_MV_DIR = 4'b0100 ; // first move down
//
input clk, rst_n ;
input [DIR_BW-1:0] ctl_dir ; // game input, head move direction
output game_over ;
output [LP_BW-1:0] snk_lp ;
//
wire clk, rst_n ;
wire [DIR_BW-1:0] ctl_dir ;
reg game_over ; // FF, game over flag output
reg [LP_BW-1:0] snk_lp ; // FF, on lamp means snake body
//
```

```
// internal variable
reg [DIR_BW-1:0] mv_dir ; // FF, memorized moving direction
wire[DIR_BW-1:0] next_mv_dir ; // next moving direction
wire mv_up, mv_dwn, mv_lft, mv_rgt ;
reg [POSI_BW * 2 -1:0] snk_bdy [0:SNK_LNG-1] ; // FF, array for snake body
reg [POSI_BW * 2 -1:0] next_head ; // non-FF for next head position
wire [POSI_BW  -1:0]  y_head, x_head ;
reg g_ovr_ob, g_ovr_bdy ;  // game over by hit body and hit wall
wire next_game_over ;
wire en ; // enable for lamp FF and snake FF
integer k ; // for-loop counter
```

```
// FF for body
always @ ( posedge clk ) begin
  if ( rst_n == 1'b0 ) begin
    snk_bdy[0] <= 6'o10 ;
    snk_bdy[1] <= 6'o11 ;
    snk_bdy[2] <= 6'o12 ;
    snk_bdy[3] <= 6'o13 ;
    snk_bdy[4] <= 6'o14 ;
    snk_bdy[5] <= 6'o15 ;
    snk_bdy[6] <= 6'o16 ;
    snk_bdy[7] <= 6'o17 ;
    snk_bdy[8] <= 6'o07 ;
    snk_bdy[9] <= 6'o06 ;
    snk_bdy[10] <= 6'o05 ;
    snk_bdy[11] <= 6'o04 ;
    snk_bdy[12] <= 6'o03;
    snk_bdy[13] <= 6'o02;
    snk_bdy[14] <= 6'o01 ;
    snk_bdy[15] <= 6'o00 ;
  end
 else begin
    if ( en == 1'b1 ) begin
      snk_bdy[0] <= next_head; // head
      for ( k = 0; k < 15; k = k+1 ) begin
        // body follow the head
        snk_bdy[k+1] <= snk_bdy[k] ;
      end
    end
  end
end
```

Set initial position of body

If not game over, move snk_body 6-bit. Next head position is evaluated in another always.

```
// next head position
assign { mv_up, mv_dwn, mv_lft, mv_rgt } = next_mv_dir ;
assign { y_head, x_head } = snk_bdy[0] ;
always @ ( y_head or x_head or  mv_up or mv_dwn or mv_lft or mv_rgt ) begin
  case ( 1'b1 )
    mv_up :    begin  next_head = { {y_head-1'b1} , {x_head} } ;  end
    mv_dwn : begin  next_head = { {y_head+1'b1} , {x_head} } ;  end
    mv_lft :     begin  next_head = { {y_head} , {x_head-1'b1} } ;  end
    mv_rgt :   begin  next_head = { {y_head} , {x_head+1'b1} } ;  end
    default:    begin  next_head = {(POSI_BW * 2 -1){1'bx}} ;       end
  endcase
end
//
// moving direction FF
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
    mv_dir <=  INTL_MV_DIR ; // initial direction
  end
  else begin
    mv_dir <= next_mv_dir ;
  end
end
//
assign next_mv_dir = ( ctl_dir )? ctl_dir : mv_dir ;
//
```

calculate next head position

FF for memorizing the moving direction. If ctl_dir=0, then keep the previous direction.

```
// out of bound check
always @ ( y_head or x_head or mv_up or mv_dwn or mv_lft or mv_rgt ) begin
  g_ovr_ob = 1'b0 ;
  case ( 1'b1 )
    mv_up  :   begin  if ( y_head == 3'o0 ) g_ovr_ob = 1'b1 ;  end
    mv_dwn : begin  if ( y_head == 3'o7 ) g_ovr_ob = 1'b1 ;  end
    mv_lft :     begin  if ( x_head == 3'o0 ) g_ovr_ob = 1'b1 ;  end
    mv_rgt :    begin  if ( x_head == 3'o7 ) g_ovr_ob = 1'b1 ;  end
    default :    begin                          g_ovr_ob = 1'bx ;  end
  endcase
end
// hit body check
always @ ( g_ovr_ob or next_head or
                 snk_bdy[SNK_LNG-1] or snk_lp[next_head] ) begin
  g_ovr_bdy = 1'b0 ;
  if ( g_ovr_ob == 1'b0 ) begin
    if ( next_head != snk_bdy[SNK_LNG-1] ) begin
       // next head is not current tail
      if ( snk_lp[next_head] == 1'b1 ) begin
        g_ovr_bdy = 1'b1 ;  // head hit body
      end
    end
  end
end
```

check if go out of field.
If go out, place 1 to g_ovr_ob.

check if run into the body.
If next_head hit the tail, then OK.
Otherwise, check if hit the body. If run into the body, place 1 to g_ovr_bdy.

```
// game over signal
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
    game_over <= 1'b0 ;
  end
  else begin
    game_over <= next_game_over ;
  end
end
//
assign next_game_over = g_ovr_bdy | g_ovr_ob | game_over ;
assign en = ~next_game_over ;
//
// field lamps
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n == 1'b0 ) begin
     snk_lp <= { {(LP_BW-SNK_LNG){1'b0}},{SNK_LNG{1'b1} }} ;
  end
  else begin
     if ( en == 1'b1 ) begin
       snk_lp[next_head] <= 1'b1 ;
       if ( next_head !=  snk_bdy[SNK_LNG-1] ) begin
         snk_lp[snk_bdy[SNK_LNG-1] ] <= 1'b0 ;
        end
     end
   end
end
//
endmodule
```

FF for game over flag

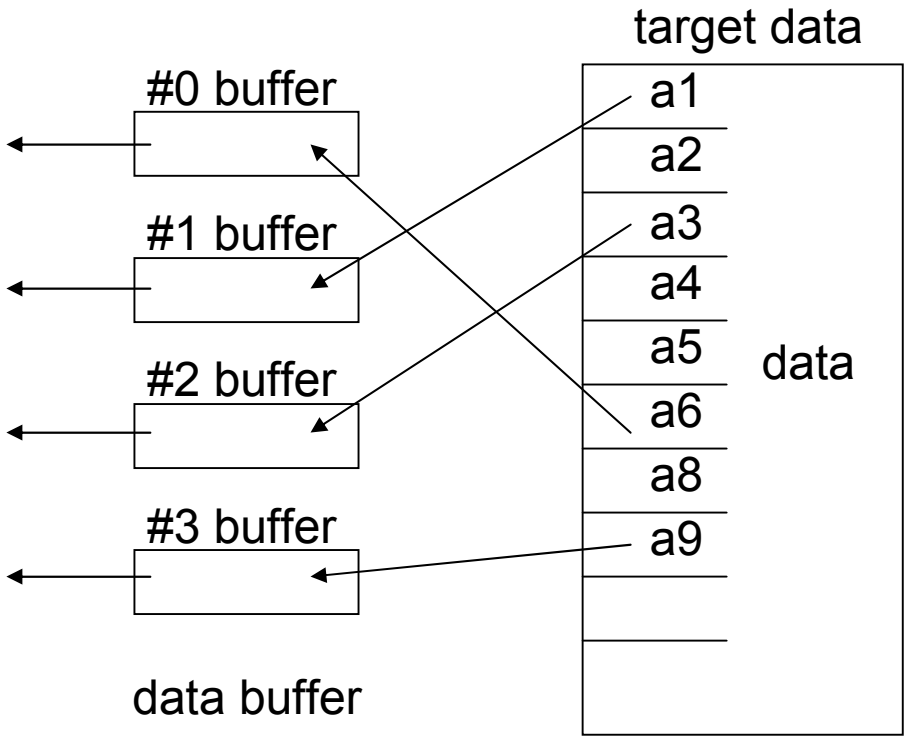
update game over.

FF for snk_lp. update it if not game over. Turn on next head lamp, and turn off the tail lamp.

file name: snake_game

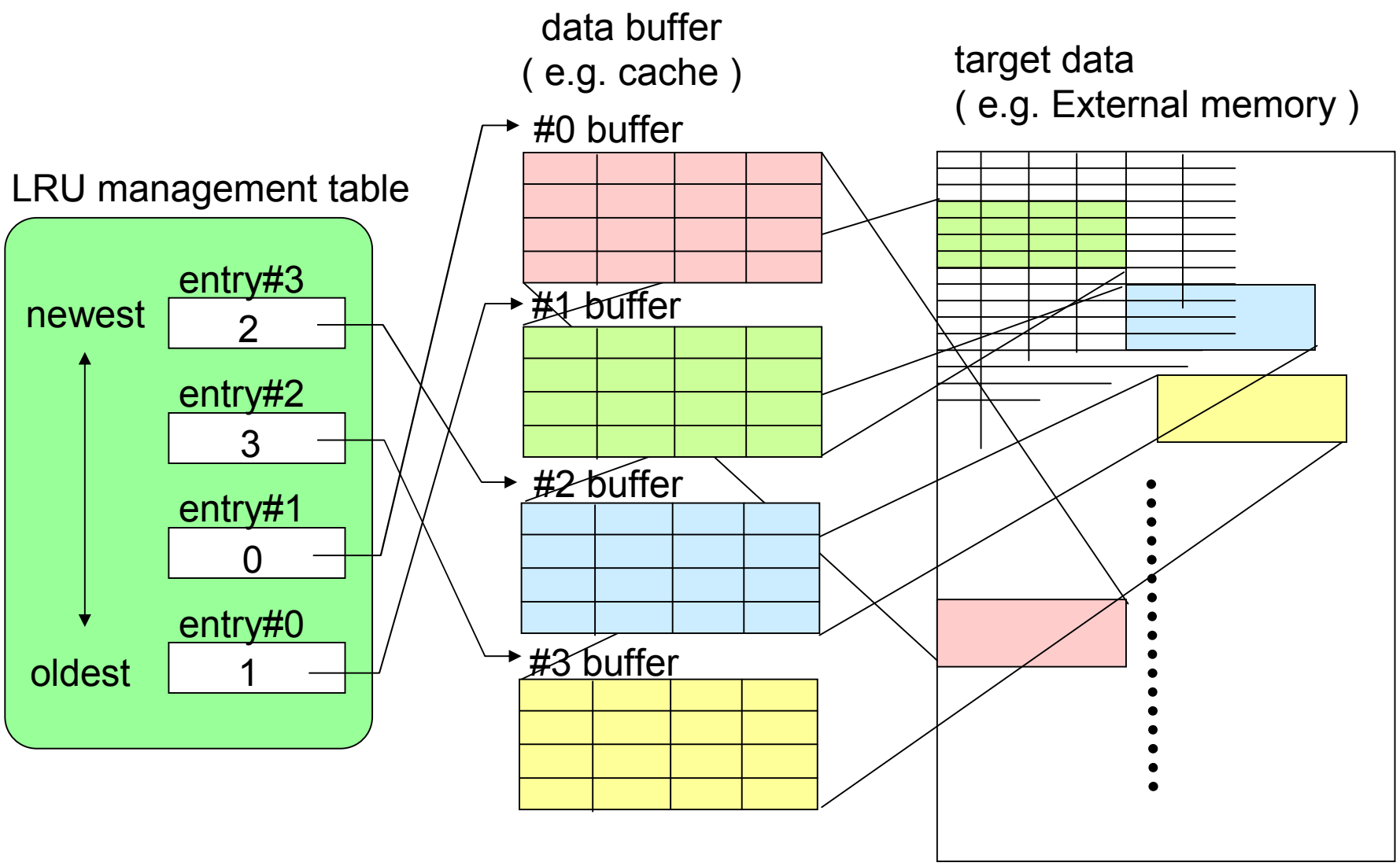
Ex 6-2. LRU algorithm: Create a module that finds out the Least Recently Used entry in the four entries specified below.

The target data are accessible only through data buffers named: #0 buffer, #1 buffer, #2 buffer, and #3 buffer. The following figure shows that data a1 using #1 buffer, a3 using #2 buffer, a6 using #0 buffer, and a9 using #3 buffer. Now, if data a2 is requested, one of the buffers must be replaced for a2. To select the buffer to be replaced for the new request, apply LRU strategy.



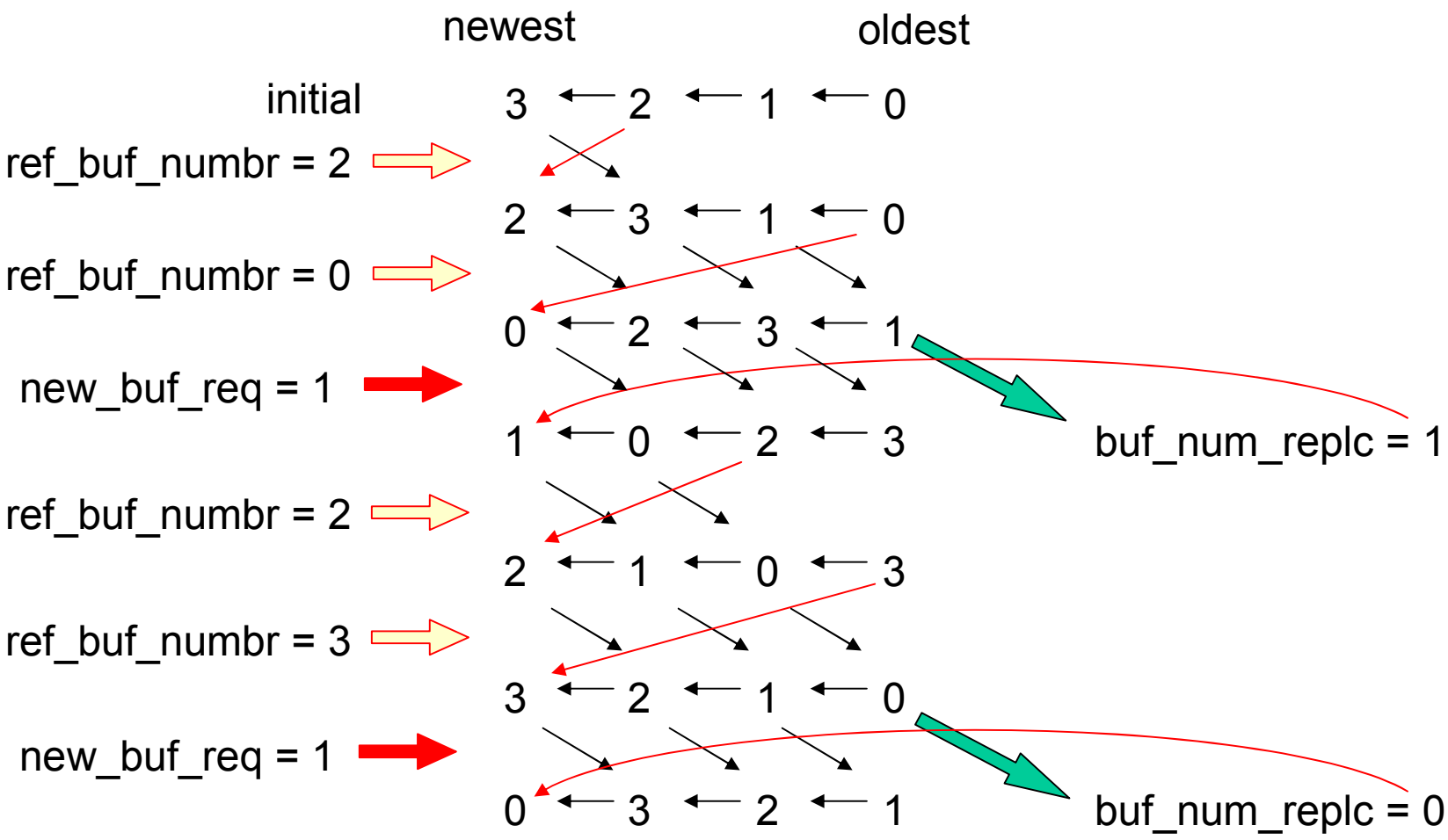
Objective:

Suppose the target data are randomly accessed, create a module which can output the buffer number to be replaced based on LRU algorithm.

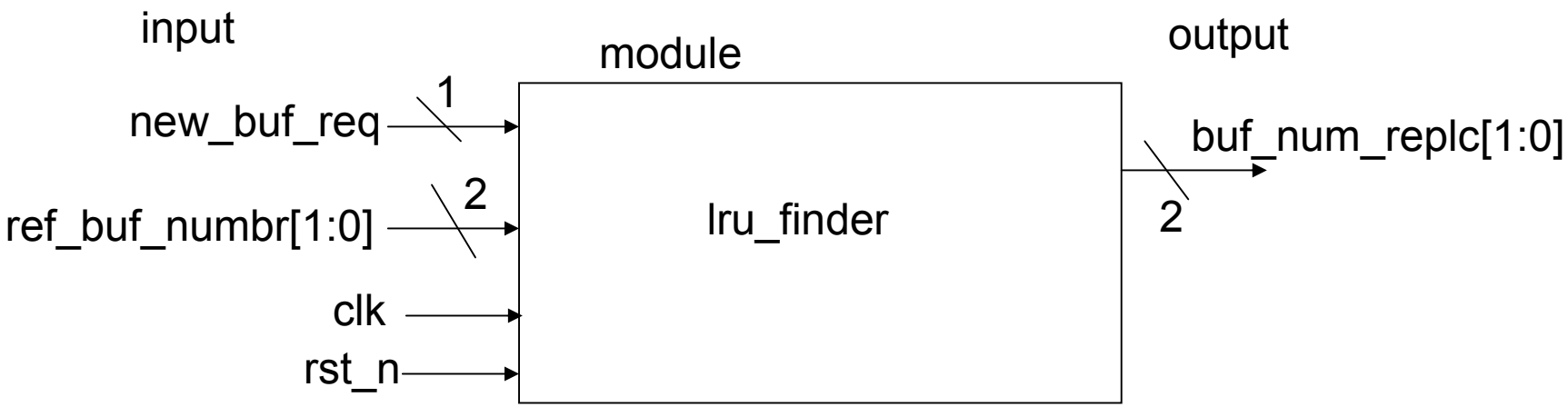


ref_buf_numbr = 2 means data in cache #2 buffer is referenced.
buf_num_replc = 1 means cache #1 buffer shall be replaced (saved back to External memory and shall be filled in with new data).

Operation example:



Create your own algorithm to find out buf_num_replc.

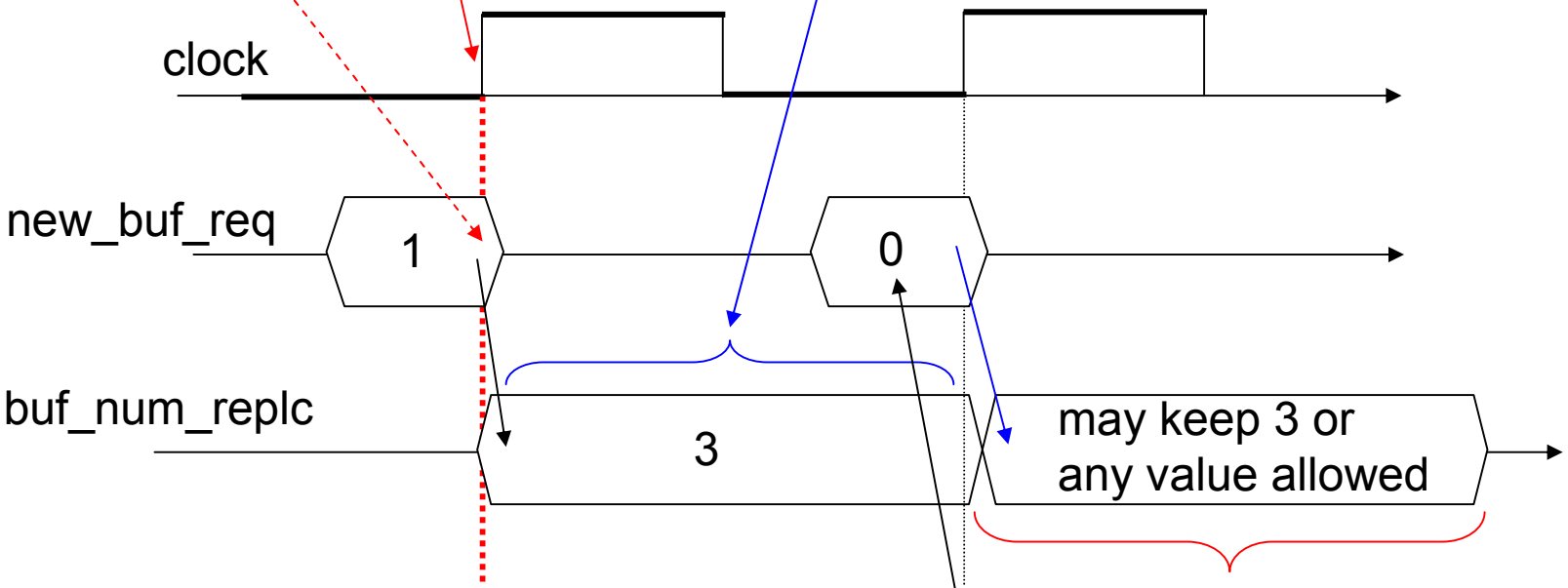


new_buf_req : 0 if no need to replace a buffer. 1 if a buffer must be replaced.
ref_buf_numbr : The number of the buffer accessed. This signal is valid only when new_buf_req = 0.
buf_num_replc : The number of the buffer to be replaced. This signal is valid only during a clock CYCL right after new_buf_req is set to 1. The buffer defined by this signal must be treated to be most recently referenced.

At the initial condition, the sequence of filling up the buffers is #0, #1, #2, and then #3, that is #3 is newest and #0 is oldest. The LRU logic in this exercise is only applicable after all entries are filled up.

Timing issue

The output buf_num_replc must become available right after the rise edge of the clock if new_buf_req is 1 at that rise edge of the clock.



buf_num_replc can be any number in this time period because no new buffer requested. Keeping the previous value (3) may be a good idea.

Note on code efficiency

Because the target of this exercise is to get accustomed to RTL programming, it is acceptable if your logic memorizes all the referenced sequence and use it to select the LRU entry.

Work flow for solutions

(1) Investigate the function of the system.
(2) Find what is essential to the function,
(3) Write a design document,
(4) Write a RTL code for the target module,
(5) Run the automatic test bench on the next pages with the target module you created.
If your code is rejected, correct your code.

Copy and paste the test bench module on the next pages into you PC and run it with your lru_finder module to see if it is OK or Not.

A sample test bench, automatic tester

Copy and paste this module into you PC and run it with your lru_finder module to see if it is OK or Not.

```
module test_lru_finder ;
parameter HF_CYCL = 5 ;
parameter CYCL = HF_CYCL*2 ;
//
reg clk, rst_n ;
reg b_rq ;
reg [1:0] ref ;
wire [1:0] rplc ;
//
// connect signals to game
lru_finder lru_finder_01( .clk( clk ), .rst_n( rst_n ),
                  .new_buf_req( b_rq ), .ref_buf_numbr( ref ),
                  .buf_num_replc( rplc )
                 ) ;
always begin      // clock generator
   clk = 1'b0 ; #HF_CYCL ;
   clk = 1'b1 ; #HF_CYCL ;
end
initial begin // give value to control variable
   rst_n = 1'b0 ;
   #1 rst_n = 1'b1 ;
   #(CYCL * 35 ) $display("test end with no error");
   #1 $finish ;
end
```

target module connection

clock generator

Termination timer

```
always @ ( posedge clk ) begin
    # 2  $strobe("t=%d, rst=%b, rq=%b, ref= %d, rplc=%b",
            $stime, rst_n, b_rq, ref, rplc
             ) ;
end
//
task chk_and_stop ; // check target result and stop if error
input [7:0] seq ; // buffer sequence
reg [1:0] w_seq [0:3] ;
integer k ;
begin
 { w_seq[0], w_seq[1], w_seq[2], w_seq[3] } = seq[7:0] ;
 for ( k = 0 ; k <=3 ; k = k+1 ) begin
 @(posedge clk) begin
  #2 if ( rplc != w_seq[k] ) begin
        $strobe("error at t=%d,  buf_num_rplc shall be %d, but was %d",
        $stime, w_seq[k], rplc ) ;
        #5 $finish;
    end
 end
end
end
endtask
```

show result at each clock rise time

Show lamps at each clock rise time

Divide input to be used in for loop

compare at clock rise time. If different, display error message and stop simulation.

```
initial begin
     b_rq = 1'b0 ;
      ref = 2'd0 ;
   #CYCL ref = 2'd1 ;
   #CYCL ref = 2'd0 ;
   #CYCL ref = 2'd3 ;
   #CYCL ref = 2'd2 ;
   #CYCL b_rq = 1'b1 ;
      ref = 2'dx ;
   chk_and_stop( 8'b01_00_11_10 ) ;
   @(negedge clk) b_rq = 1'b0 ;
      ref = 2'd0 ;
   #CYCL b_rq = 1'b1 ;
      ref = 2'dx ;
   chk_and_stop( 8'b01_11_10_00 ) ; // 1320
   @(negedge clk) b_rq = 1'b0 ;
      ref = 2'd2 ;
   #CYCL b_rq = 1'b1 ;
      ref = 2'dx ;
   chk_and_stop( 8'b01_11_00_10 ) ;//1302
```

buffer shall be
1->0->3->2

buffer shall be
1->3->2->0

buffer shall be
1->3->0->2

```
  @(negedge clk) b_rq = 1'b0 ;
     ref = 2'd1 ;
  #CYCL b_rq = 1'b1 ;
     ref = 2'dx ;
  chk_and_stop( 8'b11_00_10_01 ) ; //3021
  @(negedge clk) b_rq = 1'b0 ;
     ref = 2'd1 ;
  #CYCL b_rq = 1'b1 ;
     ref = 2'dx ;
  chk_and_stop( 8'b11_00_10_01 ) ; //3021
  @(negedge clk) b_rq = 1'b0 ;
     ref = 2'd3 ;
  #CYCL b_rq = 1'b1 ;
     ref = 2'dx ;
  chk_and_stop( 8'b00_10_01_11 ) ; // 0213
  @(negedge clk) b_rq = 1'b0 ;
     ref = 2'd0 ;
  #10 b_rq = 1'b1 ;
     ref = 2'dx ;
  chk_and_stop( 8'b10_01_11_00 ) ; // 2130
end
//
endmodule
```

buffer shall be 3->0->2->1

buffer shall be 3->0->2->1

buffer shall be 0->2->1->3

buffer shall be 2->1->3->0

file name: test_lru_finder

A sample target code

```
//*****************************************************
//  RTL programming exercise training sample answer
//  (c) RVC, 2010
//*****************************************************
module lru_finder ( clk, rst_n, new_buf_req,
                    ref_buf_numbr, buf_num_replc ) ;
input clk,  rst_n ;
input new_buf_req ;
input [1:0] ref_buf_numbr ;
output[1:0] buf_num_replc ;

wire clk, rst_n ;
wire new_buf_req ;
wire [1:0] ref_buf_numbr ;
wire [1:0] buf_num_replc ;


// internal signals
reg [7:0] ref_seq ;  // FF
reg [7:0] next_ref_seq ; // non-FF
wire [1:0] ref_numbr ;

//************* logics start **************
// This logic works for buffer full state
// assume buffers are used in sequence
//of #0, #1, #2, and #3 sequence
//

//********* select buffer ***************
assign buf_num_replc = ref_seq[1:0] ;
assign ref_numbr = ( new_buf_req == 1'b1 )?
                   ref_seq[7:6] : ref_buf_numbr ;
```

```
//********* reference sequence  *****************
always @ ( posedge clk or negedge rst_n ) begin
   if ( rst_n == 1'b0 ) begin
        // initialize reference sequence
        //  #0(old), #1, #2, and #3(new) order
      ref_seq <= 8'b00_01_10_11 ;
  end
  else begin
      ref_seq <= next_ref_seq ;
   end
end

always @ ( ref_seq[7:0] or ref_numbr ) begin
  case ( ref_numbr[1:0] )
      ref_seq[7:6] : begin
          new_ref_seq = { ref_seq[5:0], ref_seq[7:6] } ;
        end
      ref_seq[5:4] : begin
           new_ref_seq = { ref_seq[7:6], ref_seq[3:0], ref_seq[5:4] } ;
        end
      ref_seq[3:2] : begin
        new_ref_seq = { ref_seq[7:4], ref_seq[1:0], ref_seq[3:2] } ;
         end
      ref_ref_seq[1:0] : begin
          new_ref_seq = { ref_seq[7:2], ref_seq[1:0] } ;
       end
      default      :  begin
           new_ref_seq = 8'bxx_xx_xx_xx ;
       end
  endcase
end
endmodule
```

file name: lru_finder.v

Another solution

This solution is used in actual products.

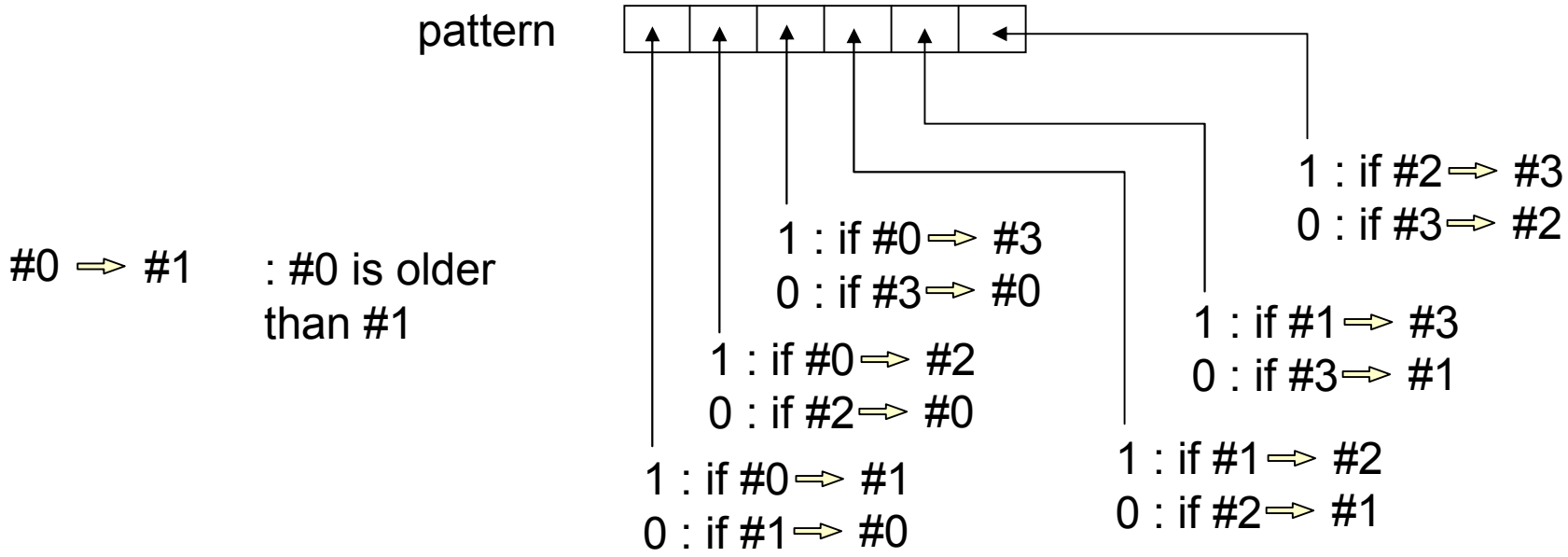
The code shown next is based on the following algorithm.

if the pattern is 111xxx then replace #0 and change the pattern by 000xxx

if the pattern is 0xx11x then replace #1 and change the pattern by 1xx00x

if the pattern is x0x0x1 then replace #2 and change the pattern by x1x1x0

if the pattern is xx0x00 then replace #3 and change the pattern by xx1x11



Another sample code

```
module lru_finder ( clk, rst_n,
                    new_buf_req, ref_buf_numbr,
                    buf_num_replc
                   ) ;
input clk, rst_n ;
input new_buf_req ;
input [1:0] ref_buf_numbr ;
output[1:0] buf_num_replc ;

wire clk, rst_n ;
wire new_buf_req ;
wire [1:0] ref_buf_numbr ;
reg [1:0] buf_num_replc ;

// internal signals
reg [5:0] ref_seq ; // FF
reg [5:0] next_seq ;  // non-FF
reg [1:0] oldest_buf ;  // non-FF
wire [1:0] ref_numbr ;
```

```
// =======   find the oldest entry  ==========
always @ ( ref_seq[5:0] ) begin
  casez ( ref_seq[5:0] )
    6'b111??? : begin
                    oldest_buf[1:0] = 2'b00 ;
        end
    6'b0??11? : begin
                    oldest_buf[1:0] = 2'b01 ;
        end
    6'b?0?0?1 : begin
                    oldest_buf[1:0] = 2'b10 ;
        end
    6'b??0?00 : begin
                    oldest_buf[1:0] = 2'b11 ;
        end
    default : begin
                    oldest_buf[1:0] = 2'bxx ;
        end
  endcase
end
```

```
//********* select buffer to be replaced ***************
always @ ( posedge clk ) begin
  if ( new_buf_req == 1'b1 ) begin
    buf_num_replc[1:0] <= oldest_buf[1:0] ;
  end
end

assign ref_numbr[1:0] = ( new_buf_req == 1'b1 )?
                                oldest_buf[1:0] : ref_buf_numbr[1:0] ;

//********* reference sequence  *****************
always @ ( posedge clk or negedge rst_n ) begin
   if ( rst_n == 1'b0 ) begin  // initialize reference sequence
      ref_seq[5:0] <= 6'b111_11_1 ;  //  #0(old), #1, #2, and #3(new) order
   end
   else begin
     ref_seq[5:0] <= next_seq[5:0] ;
   end
end
```

```
always @ ( ref_seq[5:0] or ref_numbr[1:0] ) begin
  case ( ref_numbr[1:0] )
   2'b00 : begin            // update pattern 000xxx
         next_seq[5:0] = { 3'b000, ref_seq[2:0] } ;
       end
   2'b01 : begin            // update pattern 1xx00x
         next_seq[5:0] = { 1'b1, ref_seq[4:3], 2'b0, ref_seq[0] } ;
       end
   2'b10 : begin            // update pattern x1x1x0
         next_seq[5:0] = { ref_seq[5], 1'b1,ref_seq[3],1'b1,ref_seq[1], 1'b0 } ;
       end
   2'b11 : begin           // update pattern xx1x11
         next_seq[5:0] = { ref_seq[5:4], 1'b1, ref_seq[2], 2'b11 } ;
       end
   default :   begin
         next_seq[5:0] = 6'bxxxxxx ;
       end
  endcase
end

endmodule
```

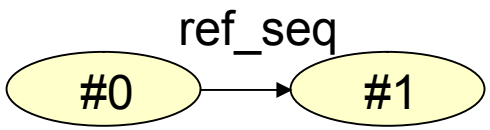
file name: lru_finder_v2.v

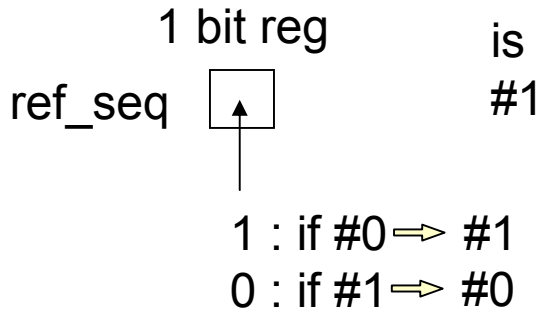
The background of the another solution

First , let's study several cases how LRU shall work.

(1) Two entries case

This case is obvious.





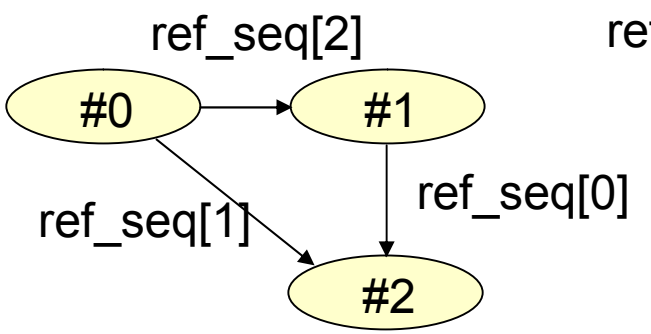
#0 ⇨ #1

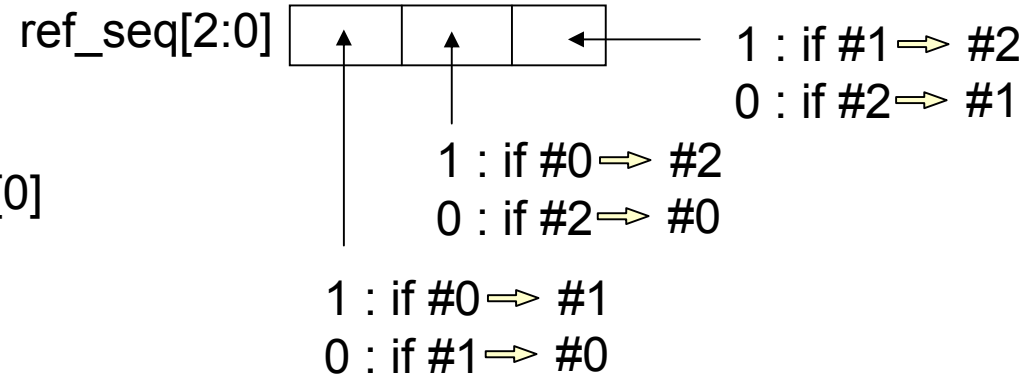
This means #0 is older than #1.

➡ There is only one path from #0 to #1, and this one bit information can show which one of two entries is newly referenced.

➡ By checking 1-bit ref_seq, we can determine which one of #0 or #1 shall be replaced.

(2) Three entries case





There are three paths: from #0 to #1, from #0 to #2, and from #1 to #2. Using old or new relation between the pairs we can define the referenced sequence for all three entries.

However there is some redundancy for this information. For example, if ref_seq[2] = 1 and ref_seq[0] = 1, then ref_seq[1] must be 1. This means that we can determine the referenced sequence is #0, #1 and then #3, by checking if ref_seq's bit pattern is 1x1 (x: don't care) or not. The don't care bit position is different depending on the referenced sequence.

These are the all possible patterns of reference

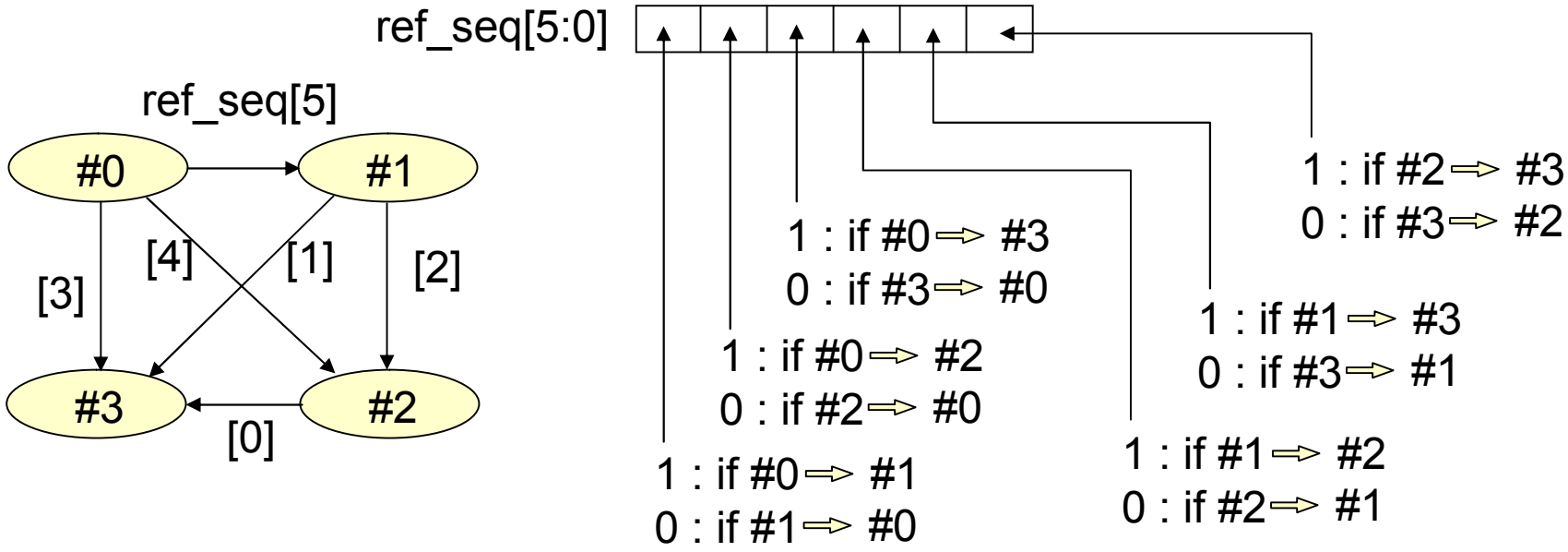
current reference sequence (old → new)	0 1 2	0 2 1	1 0 2	1 2 0	2 0 1	2 1 0
ref_seq	1 (1) 1	(1) 1 0	0 1 (1)	(0) 0 1	1 0 (0)	0 (0) 0
new ref_seq if reference to #0	(0) 0 1	0 (0) 0	(0) 0 1	(0) 0 1	0 (0) 0	0 (0) 0
update pattern	0 0 x	0 0 x	x 0 x	x x x	0 x x	x x x
			0 0 x	0 0 x	0 0 x	0 0 x

Equivalent

Thus we found a common update pattern for reference to #0 is 00x, where x means no change.

( ) denotes a redundant bit.

(3) for four entries



First look for the bit pattern of ref_seq so that we can tell what entry# to replace.

reference sequence	ref_seq pattern
0 1 2 3 →	1 (1)(1) 1 (1) 1
0 1 3 2 →	1 (1)(1)(1) 1 0
0 2 1 3 →	(1) 1 (1) 0 1 (1)
0 2 3 1 →	(1) 1 (1)(0) 0 1
0 3 1 2 →	(1)(1) 1 1 0 (0)
0 3 2 1 →	(1)(1) 1 0 (0) 0

111xxx then replace #0

reference sequence	ref_seq pattern
1 0 2 3 →	0 1 (1)(1)(1) 1
1 0 3 2 →	0 (1) 1 (1)(1) 0
1 2 0 3 →	(0) 0 1 1 (1)(1)
1 2 3 0 →	(0)(0) 0 1 (1) 1
1 3 0 2 →	(0) 1 0 (1) 1 (0)
1 3 2 0 →	(0) 0 (0)(1) 1 0

0xx11x then replace #1

reference sequence	ref_seq pattern
2 0 1 3 →	1 0 (1)(0) 1 (1)
2 0 3 1 →	(1) 0 1 (0) 0 (1)
2 1 0 3 →	0 (0) 1 0 (1)(1)
2 1 3 0 →	(0)(0) 0 0 1 (1)
2 3 0 1 →	1 (0) 0 (0)(0) 1
2 3 1 0 →	0 (0)(0)(0) 0 1

x0x0x1 then replace #2

reference sequence	ref_seq pattern
3 0 2 1 →	(1) 1 0 0 (0)(0)
3 0 1 2 →	1 (1) 0 1 (0)(0)
3 1 0 2 →	0 1 (0)(1) 0 (0)
3 1 2 0 →	(0) 0 (0) 1 0 (0)
3 2 0 1 →	1 0 (0)(0)(0) 0
3 2 1 0 →	0 (0)(0) 0 (0) 0

xx0x00 then replace #3

Next, look for the update pattern

reference sequence	ref_seq pattern	new pattern if reference to #0	update pattern
0 1 2 3	1 (1)(1) 1 (1) 1	1 2 3 0 : (0)(0) 0 1 (1) 1	000xxx
0 1 3 2	1 (1)(1)(1) 1 0	1 3 2 0 : (0) 0 (0)(1) 1 0	000xxx
0 2 1 3	(1) 1 (1) 0 1 (1)	2 1 3 0 : (0)(0) 0 0 1 (1)	000xxx
0 2 3 1	(1) 1 (1)(0) 0 1	2 3 1 0 : 0 (0)(0)(0) 0 1	000xxx
0 3 1 2	(1)(1) 1 1 0 (0)	3 1 2 0 : (0)(0) 0 1 0 (0)	000xxx
0 3 2 1	(1)(1) 1 0 (0) 0	3 2 1 0 : 0 (0)(0) 0 (0) 0	000xxx
1 0 2 3	0 1 (1)(1)(1) 1	1 2 3 0 : (0)(0) 0 1 (1) 1	x00xxx
1 0 3 2	0 (1) 1 (1)(1) 0	1 3 2 0 : (0) 0 (0)(1) 1 0	x00xxx
1 2 0 3	(0) 0 1 1 (1)(1)	1 2 3 0 : (0)(0) 0 1 (1) 1	xx0xxx
1 2 3 0	(0)(0) 0 1 (1) 1	1 2 3 0 : (0)(0) 0 1 (1) 1	xxxxxx
1 3 0 2	(0) 1 0 (1) 1 (0)	1 3 2 0 : (0) 0 (0)(1) 1 0	x0xxxx
1 3 2 0	(0) 0 (0)(1) 1 0	1 3 2 0 : (0) 0 (0)(1) 1 0	xxxxxx
2 0 1 3	1 0 (1)(0) 1 (1)	2 1 3 0 : (0)(0) 0 0 1 (1)	0x0xxx
2 0 3 1	(1) 0 1 (0) 0 (1)	2 3 1 0 : 0 (0)(0)(0) 0 1	0x0xxx
2 1 0 3	0 (0) 1 0 (1)(1)	2 1 3 0 : (0)(0) 0 0 1 (1)	xx0xxx
2 1 3 0	(0)(0) 0 0 1 (1)	2 1 3 0 : (0)(0) 0 0 1 (1)	xxxxxx
2 3 0 1	1 (0) 0 (0)(0) 1	2 3 1 0 : 0 (0)(0)(0) 0 1	0xxxxx
2 3 1 0	0 (0)(0)(0) 0 1	2 3 1 0 : 0 (0)(0)(0) 0 1	xxxxxx

000xxx

000xxx

reference sequence	ref_seq pattern	new pattern if reference to #0	update pattern
3 0 2 1	(1) 1 0 0 (0)(0)	3 2 1 0 : 0 (0)(0) 0 (0) 0	00xxxx
3 0 1 2	1 (1) 0 1 (0)(0)	3 1 2 0 : (0) 0 (0) 1 0 (0)	00xxxx
3 1 0 2	0 1 (0)(1) 0 (0)	3 1 2 0 : (0) 0 (0) 1 0 (0)	x0xxxx
3 1 2 0	(0) 0 (0) 1 0 (0)	3 1 2 0 : (0) 0 (0) 1 0 (0)	xxxxxx
3 2 0 1	1 0 (0)(0)(0) 0	3 2 1 0 : 0 (0)(0) 0 (0) 0	0xxxxx
3 2 1 0	0 (0)(0) 0 (0) 0	3 2 1 0 : 0 (0)(0) 0 (0) 0	xxxxxx

⇒ 000xxx

Thus we get 000xxx for #0 update pattern

reference sequence	ref_seq pattern	new pattern if reference to #1	update pattern
0 1 2 3	1 (1)(1) 1 (1) 1	0 2 3 1 : (1) 1 (1)(0) 0 1	xxx00x
0 1 3 2	1 (1)(1)(1) 1 0	0 3 2 1 : (1)(1) 1 0 (0) 0	xxx00x
0 2 1 3	(1) 1 (1) 0 1 (1)	0 2 3 1 : (1) 1 (1)(0) 0 1	xxxx0x
0 2 3 1	(1) 1 (1)(0) 0 1	0 2 3 1 : (1) 1 (1)(0) 0 1	xxxxxx
0 3 1 2	(1)(1) 1 1 0 (0)	0 3 2 1 : (1)(1) 1 0 (0) 0	xxx0xx
0 3 2 1	(1)(1) 1 0 (0) 0	0 3 2 1 : (1)(1) 1 0 (0) 0	xxxxxx

⇒ 1xx00x

reference sequence	ref_seq pattern	new pattern if reference to #1	update pattern
1 0 2 3	0 1 (1)(1)(1) 1	0 2 3 1 : (1) 1 (1)(0) 0 1	1xx00x
1 0 3 2	0 (1) 1 (1)(1) 0	0 3 2 1 : (1)(1) 1 0 (0) 0	1xx00x
1 2 0 3	(0) 0 1 1 (1)(1)	2 0 3 1 : (1) 0 1 (0) 0 (1)	1xx00x
1 2 3 0	(0)(0) 0 1 (1) 1	2 3 0 1 : 1 (0) 0 (0)(0) 1	1xx00x
1 3 0 2	(0) 1 0 (1) 1 (0)	3 0 2 1 : (1) 1 0 0 (0)(0)	1xx00x
1 3 2 0	(0) 0 (0)(1) 1 0	3 2 0 1 : 1 0 (0)(0)(0) 0	1xx00x
2 0 1 3	1 0 (1)(0) 1 (1)	2 0 3 1 : (1) 0 1 (0) 0 (1)	xxxx0x
2 0 3 1	(1) 0 1 (0) 0 (1)	2 0 3 1 : (1) 0 1 (0) 0 (1)	xxxxxx
2 1 0 3	0 (0) 1 0 (1)(1)	2 0 3 1 : (1) 0 1 (0) 0 (1)	1xxx0x
2 1 3 0	(0)(0) 0 0 1 (1)	2 3 0 1 : 1 (0) 0 (0)(0) 1	1xxx0x
2 3 0 1	1 (0) 0 (0)(0) 1	2 3 0 1 : 1 (0) 0 (0)(0) 1	xxxxxx
2 3 1 0	0 (0)(0)(0) 0 1	2 3 0 1 : 1 (0) 0 (0)(0) 1	1xxxxx
3 0 2 1	(1) 1 0 0 (0)(0)	3 0 2 1 : (1) 1 0 0 (0)(0)	xxxxxx
3 0 1 2	1 (1) 0 1 (0)(0)	3 0 2 1 : (1) 1 0 0 (0)(0)	xxx0xx
3 1 0 2	0 1 (0)(1) 0 (0)	3 0 2 1 : (1) 1 0 0 (0)(0)	1xx0xx
3 1 2 0	(0) 0 (0) 1 0 (0)	3 2 0 1 : 1 0 (0)(0)(0) 0	1xx0xx
3 2 0 1	1 0 (0)(0)(0) 0	3 2 0 1 : 1 0 (0)(0)(0) 0	xxxxxx
3 2 1 0	0 (0)(0) 0 (0) 0	3 2 0 1 : 1 0 (0)(0)(0) 0	1xxxxx

→ 1xx00x

→ 1xx00x

Thus we get 1xx00x for #1 update pattern

reference sequence	ref_seq pattern	new pattern if reference to #2	update pattern
2 0 1 3	1 0 (1)(0) 1 (1)	0 1 3 2 : 1 (1)(1)(1) 1 0	x1x1x0
2 0 3 1	(1) 0 1 (0) 0 (1)	0 3 1 2 : (1)(1) 1 1 0 (0)	x1x1x0
2 1 0 3	0 (0) 1 0 (1)(1)	1 0 3 2 : 0 (1) 1 (1)(1) 0	x1x1x0
2 1 3 0	(0)(0) 0 0 1 (1)	1 3 0 2 : (0) 1 0 (1) 1 (0)	x1x1x0
2 3 0 1	1 (0) 0 (0)(0) 1	3 0 1 2 : 1 (1) 0 1 (0)(0)	x1x1x0
2 3 1 0	0 (0)(0)(0) 0 1	3 1 0 2 : 0 1 (0)(1) 0 (0)	x1x1x0

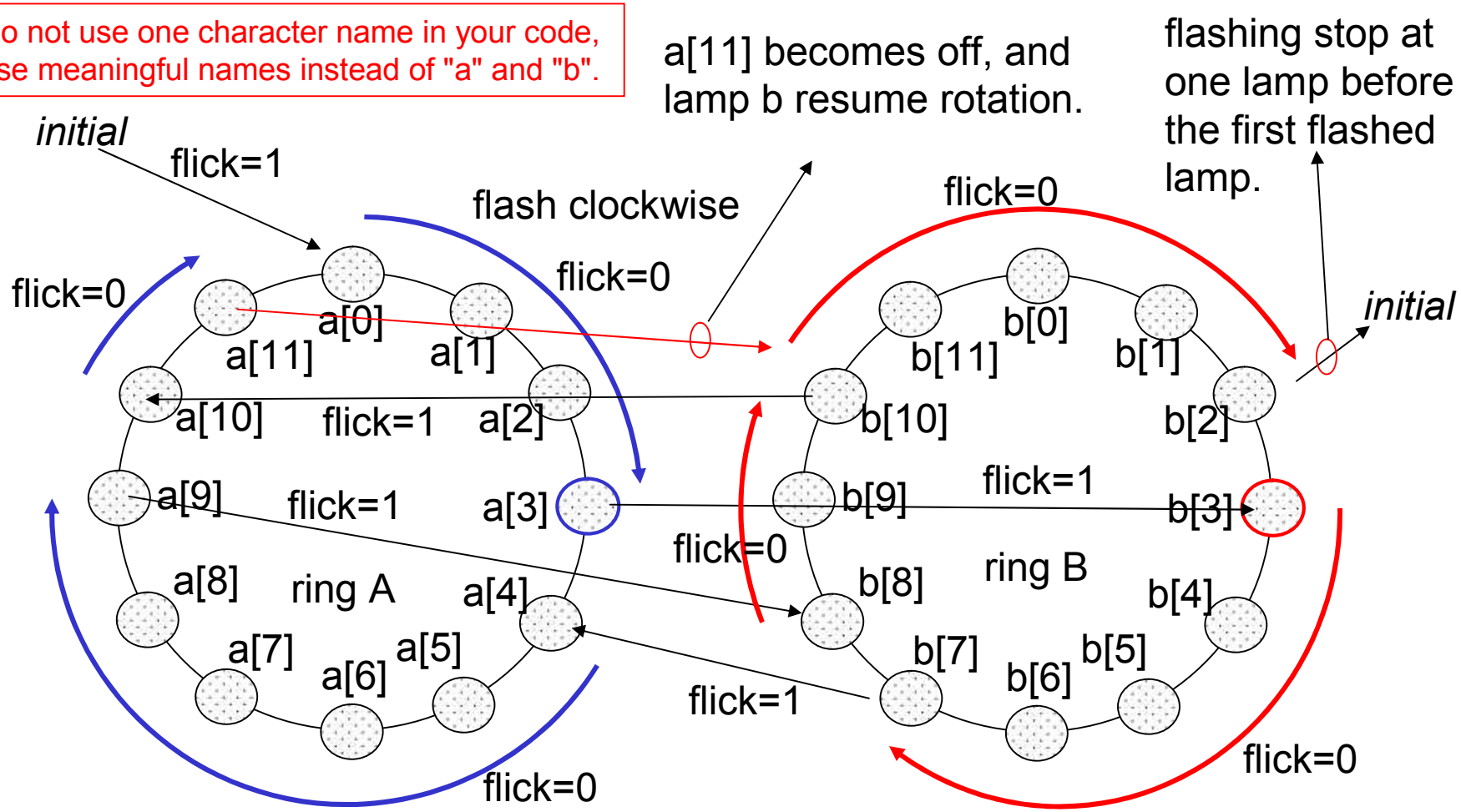
For other current patterns we can find that x1x1x0 is the update pattern for #2

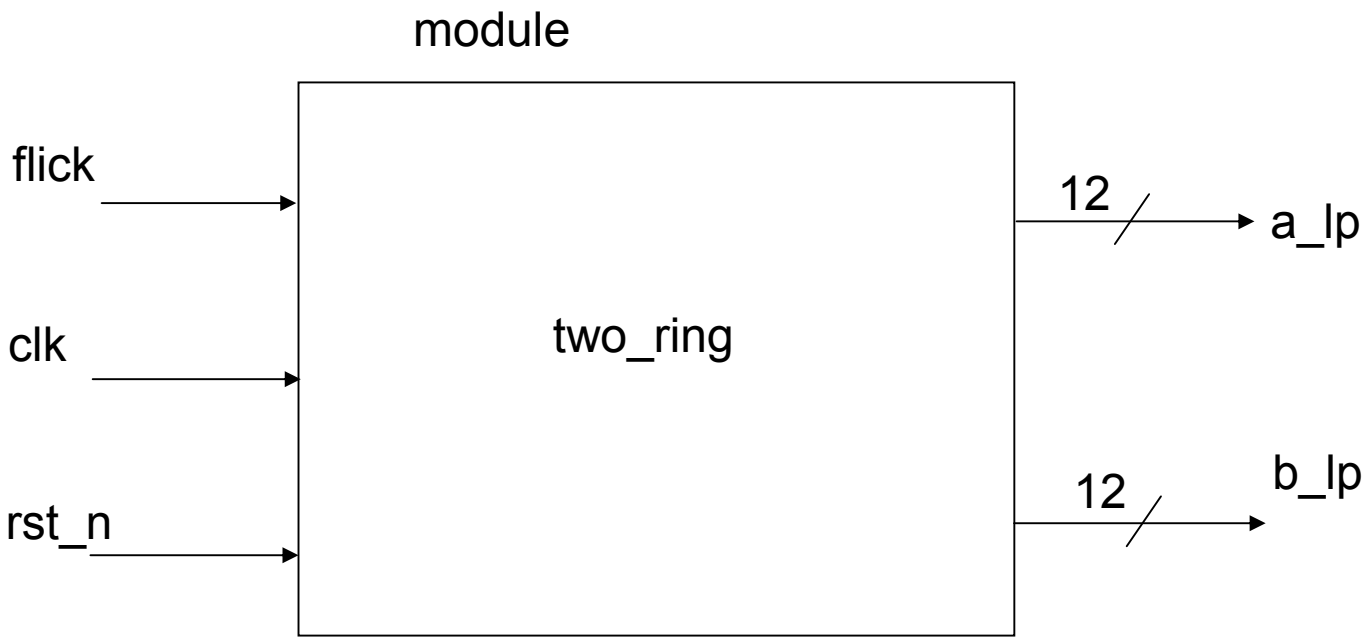
reference sequence	ref_seq pattern	new pattern if reference to #3	update pattern
3 0 2 1	(1) 1 0 0 (0)(0)	0 2 1 3 : (1) 1 (1) 0 1 (1)	xx1x11
3 0 1 2	1 (1) 0 1 (0)(0)	0 1 2 3 : 1 (1)(1) 1 (1) 1	xx1x11
3 1 0 2	0 1 (0)(1) 0 (0)	1 0 2 3 : 0 1 (1)(1)(1) 1	xx1x11
3 1 2 0	(0) 0 (0) 1 0 (0)	1 2 0 3 : (0) 0 1 1 (1)(1)	xx1x11
3 2 0 1	1 0 (0)(0)(0) 0	2 0 1 3 : 1 0 (1)(0) 1 (1)	xx1x11
3 2 1 0	0 (0)(0) 0 (0) 0	2 1 0 3 : 0 (0) 1 0 (1)(1)	xx1x11

For other current patterns we can find that xx1x11 is the update pattern for #3

Ex 6-3. Two ring flasher : Write a module for a two ring flasher of which movement is given in the figure below. a[n] represents a lamp in ring A, 0/1 of a[n] corresponds to off/on of #n lamp in ring A. b[m] represents a lamp in ring B, 0/1 of b[m] corresponds to off/on of #m lamp in ring B.

Do not use one character name in your code, use meaningful names instead of "a" and "b".





A sample test bench is prepared for this exercise.
Your code is not OK if it is rejected by the sample test bench.

Detailed specification

1. State definition

(1) *ring-on state* and *ring-off state*:
Ring A and ring B can be in either *ring-on state* or *ring-off state*.
Ring-on state means one lamp of the ring is on.
Ring-off state means all lamps of the ring is off.

(2) *Sys-initial state* and *sys-run state*:
The total system can be in either *sys-initial state* or *sys-run state*.
Sys-initial state means the both ring A and B are in *ring-off* state.
Sys-run state means either or both of ring A and B are in *ring-on state*.

2. *First-lamp* and *last-lamp* definition

(1) *First-lamp* is a lamp which turns on first when a ring becomes *ring_on* state from *ring-off* state.
(2) *Last-lamp* is a lamp next to the *first-lamp* in the counterclockwise direction.
(3) ring A's *first-lamp* is always #0.
(4) ring B's *first-lamp* can be either one of the flowing (A) or (B) depending on the flick signal when ring B becomes *ring-on* state from *ring-off* state.

2. *First-lamp* and *last-lamp* definition (continued)

(A) If flick is 0, then #0 is the *first lamp* of ring B.
(B) Otherwise, flick is 1, the lamp at the same position of the ring A's on lamp is the *first lamp* of ring B.

3. Operation of rings; *rotate operation* and *hold operation*.

Ring A and ring B must do *rotate operation* or *hold operation* at every clock rise time.
Rotate operation means the following 4 operations, (1) to (4).

(1) Turn on the first lamp of a ring which has not been in *ring-on* state after the most recent *sys_initial* state.
(2) Keep *ring-off* state of a ring which had been in *ring-on* state after the most recent *sys_initial* state and is currently in *ring-off* state.
(3) Turn off current on-lamp of the ring and at the same time turn on the next lamp in a clockwise direction if other than the *last_lamp* of the ring is on.
(4) Turn off current on-lamp of the ring if the *last_lamp* of the ring is on.

3. Operation of rings; *rotate operation* and *hold operation*. *(continued)*

Hold operation means the following operation (5).

(5) Neither on lamp position nor off lamp position of the ring change.

4. System's behavior definition (input flick is evaluated at clock rise time)

In *sys_initial* state, depending on flick input, either <1> or <2> must occur.

<1> If flick=0, the system must keep *sys_initial* state.
<2> If flick=1, ring A must do *rotate operation* and ring B must do *hold operation*.

In *sys_run* state, depending on flick input, either <3> or <4> must occur.

<3> If flick=0, both ring must do the same operation as the most recent operation.
<4> If flick=1, ring A must do the same operation of the most recent ring B's operation, and ring B must do the same operation of the most recent ring A's operation. (Both ring exchange their most recent operation.)

5. Exceptional cases

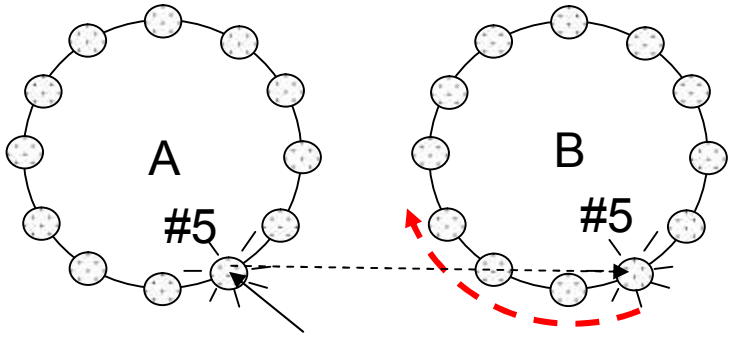
<5> If the *rotate operation* caused by <3> or <4> has to be applied to the *last-lamp* turned on or it has to be applied to the ring which became *ring-off* state after the most recent sys-initial state, then the other ring has to do *rotate operation* at the same time. That is, both rings must do *rotate operation* at the same time in such cases..

6. Reset

<6> When reset asserted, the system must become *sys_initial* state regardless of the current state. While reset is asserted, flick signal is invalid.

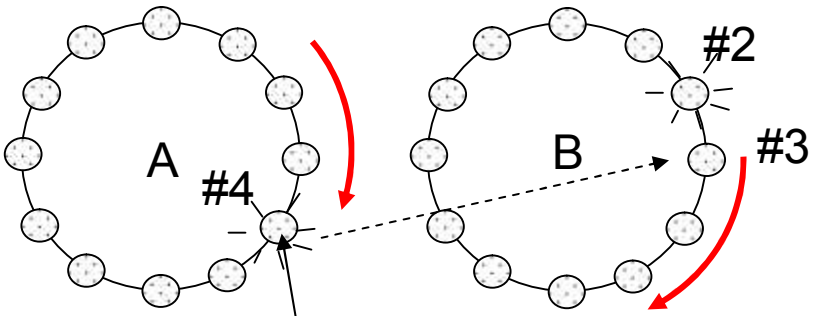
Additional explanation

When B off and a[5] on and flick is set to 1, then b[5] becomes the *first-lamp*.



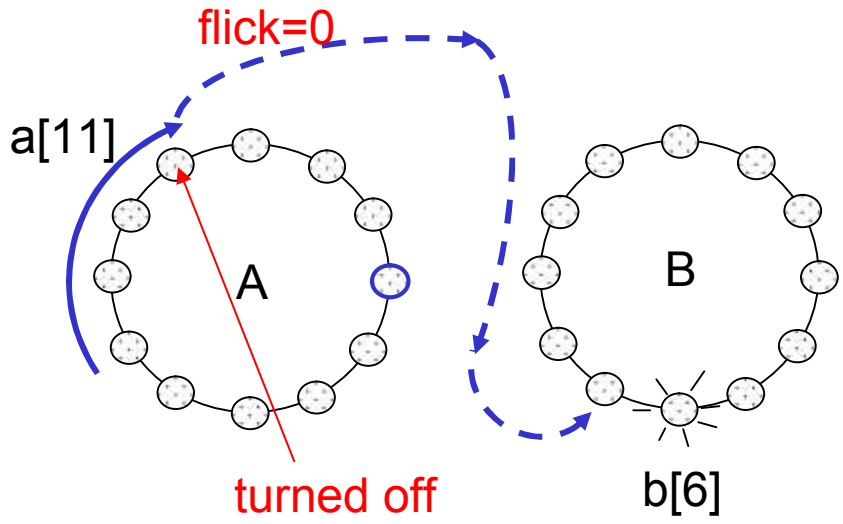
If flick becomes on at a[5] lamp, B starts from b[5] lamp.

When the rotating ring switches from ring A/B to B/A, ring A/B's lamp will stop rotation but keeps illuminating.



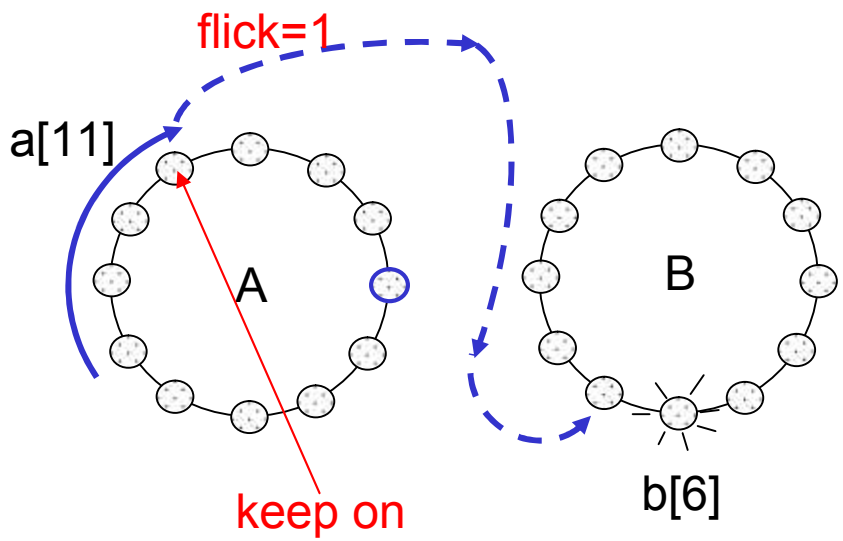
lamp stop moving but keep illuminating.

When a ring completed its one turn, all its lamps must become off as explained below.

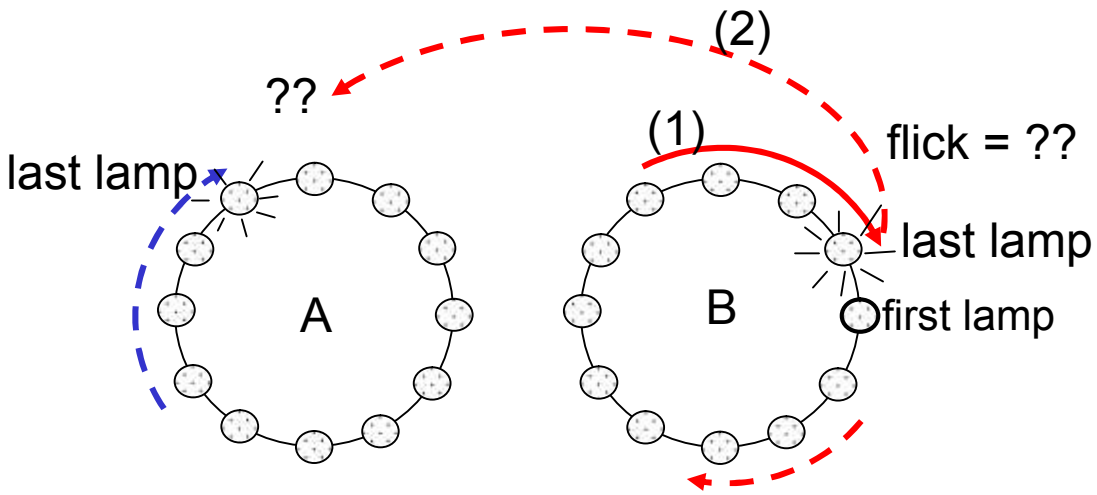


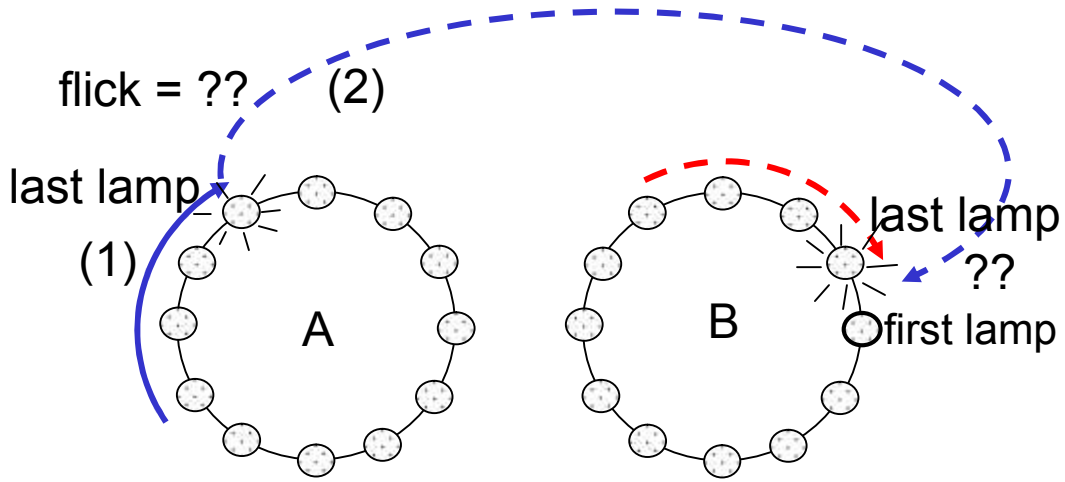
ring A is rotating. If flick is kept off, it will complete one turn. Therefore, a[11] and b[6] becomes off and b[7] becomes on at the same time.

However, if the switching is caused by flick signal at a[11], a[11] must not be turned off as shown on the right. b[6] becomes off and b[7] becomes on while a[11] is kept on.



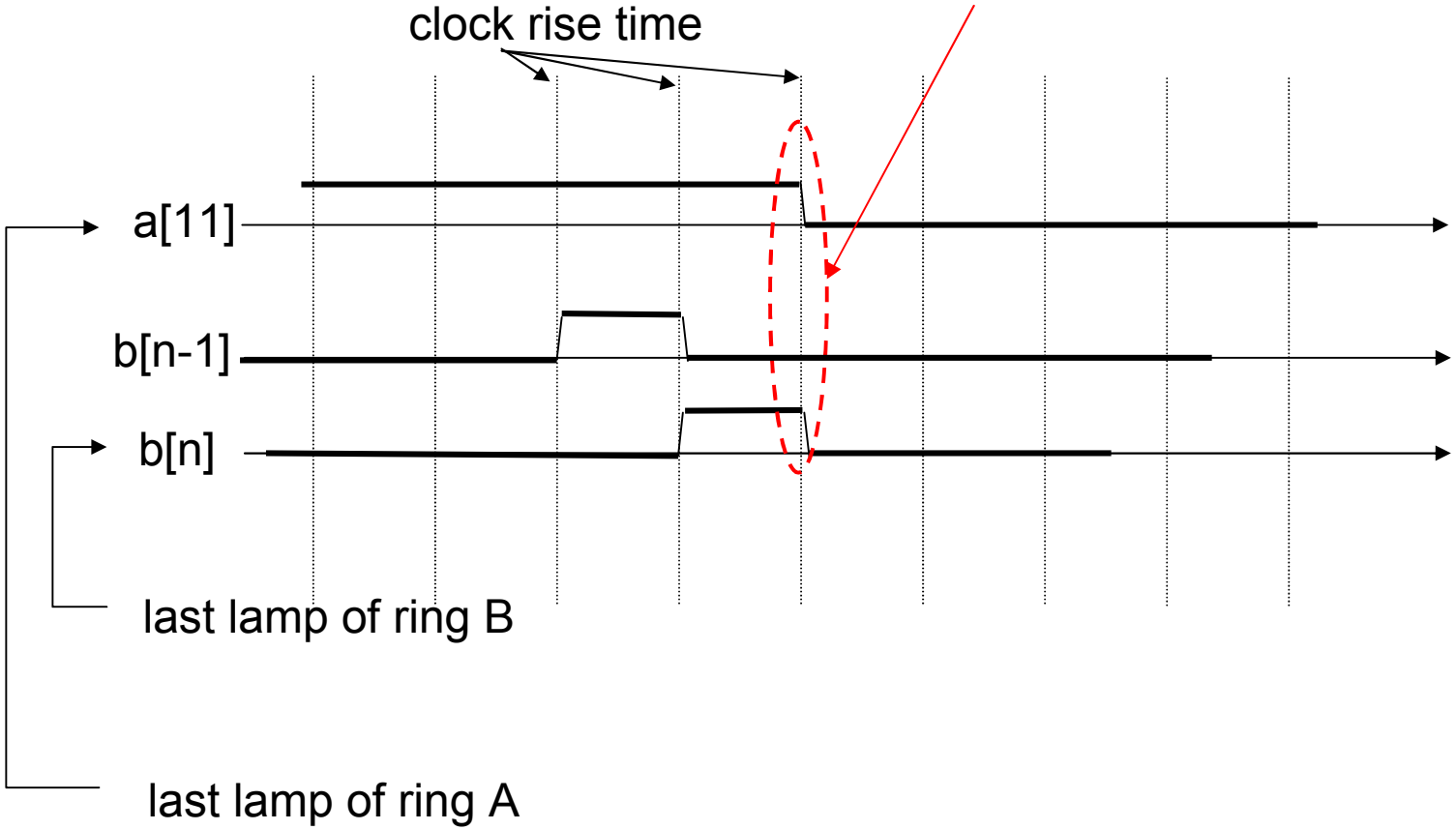
There are very special cases as below.





In both cases, on switching to another ring, the last lamps are already on, therefore switching means completion of the other ring's one turn. And because the completed ring can not rotate any more, original ring must resume rotation at once. This means the original ring must also complete the turn.
So, for these special cases, make your logic to turn off both lamps and go to initial state as shown on the time chart on the next page.

When both last lamps are on at the same time, they must be turned off at the clock rise time.



Work flow for solutions

(1) Investigate the function of the system.
(2) Find what is essential to the function,
(3) Write a design document including a state transition table,
(4) Write a RTL code for the target module,
(5) Run a sample test bench on the next pages with the target module you created.
If your code is rejected, correct your code.

Copy and paste the test bench module on the next pages into you PC and run it with your two_ting module to see if your code is OK or Not.

A sample test bench, automatic tester

Copy and paste this module into you PC and run it with your two_ting module to see if it is OK or Not.

```
module test_two_ring ;
parameter HF_CYCL = 5 ;
parameter CYCL = HF_CYCL*2 ;

reg rst_n, clk, flick ;
wire [11:0] a_lp, b_lp ;
reg [11:0] g_a_lp, g_b_lp ;

// module connection
two_ring two_ring_01 ( .clk(clk), .rst_n(rst_n), .flick(flick),
                       .a_lp(a_lp), .b_lp(b_lp)
                       ) ;
// display the movements
always @ ( posedge clk ) begin
#2 $display("t=%d,r=%b,f=%b,a=%b,b=%b",
            $stime, rst_n, flick,
            a_lp, b_lp ) ;
end
initial begin
# (CYCL * 800 ) $finish;
end
```

When using a sample target code in the latter pages, change this part to multi_ring multi_ring_01.

Change this to .lamp_1(a_lp), .lamp_2(b_lp)

target module connection

Show lamps at each clock rise time

Termination timer

This must be larger than FF_DLY if delay is used in FF design.

```
/// test data and monitor
// clk and reset
always begin
 clk = 0 ; #HF_CYCL ;
 clk = 1 ; #HF_CYCL ;
end
```

clock generator

```
initial begin
rst_n=0 ; #(CYCL*3 + 3 ) ;
rst_n=1 ;
end
```

reset control

```
event timng ;  // timing event
always @ ( posedge clk ) begin
   #HF_CYCL -> timng ;
end
```

generate event "timng" at clock fall time

```
// invoke golden model and compare
always begin
@(posedge clk) golden_ring(g_a_lp, g_b_lp) ;
#2 cmp_chk ( g_a_lp, g_b_lp, a_lp, b_lp ) ;
end
```

Invoke golden model and compare at each clock rise time

This must be larger than FF_DLY if delay is used in FF design.

```
task golden_ring ;
output [11:0] a_lp, b_lp ;
integer a_cnt, b_cnt, b_pos ;
reg a_flg ; // non-FF
reg [11:0] temp ;  // non-FF. work for B bit position
begin
  if ( rst_n == 0 ) begin
    a_cnt = 0 ; // A lamp position counter
    b_cnt = 0 ; // B lamp position counter
    b_pos = 0 ;  // B start position
  end
  else begin
     if ( a_cnt == 0 ) begin // not started
       if ( flick ) begin
             a_cnt = 1 ;  // first lamp on
             a_flg = 1 ;   // ring A rotate
        end
     end
```

initialize if reset is active

a_cnt==0 means no lamp is on. Therefore, if flick==1, start rotating A

This logic does have to care if counter becomes larger than 12.

The golden model task will run well with the counter larger than 12.

```
else begin  // already started
  if ( flick ) begin
    if ( a_flg ) begin
      a_flg = 0 ;  // rign B rotate
      b_pos = ( b_cnt == 0 )? a_cnt-1 : b_pos ;
      b_cnt = b_cnt+1 ;
      if ( b_cnt > 12 ) begin
        a_flg = 1 ;
        a_cnt = a_cnt + 1 ;
      end
    end
    else begin
      a_flg = 1 ;  // rign A rotate
      a_cnt = a_cnt+1 ;
      if ( a_cnt > 12 ) begin
        a_flg = 0 ;
        b_cnt = b_cnt + 1 ;
      end
    end
  end
```

Rotate B, if B complete one turn, then rotate A. If B not started, set start position of B.

Rotate A, if A complete one turn, then rotate B.

```
    else begin // flick is 0
      if ( a_flg ) begin
        a_cnt = a_cnt+1 ;
        if ( a_cnt > 12 ) begin
          a_flg = 0 ;
          b_cnt = b_cnt + 1 ;
        end
      end
      else begin
        b_cnt = b_cnt+1 ;
        if ( b_cnt > 12 ) begin
          a_flg = 1 ;
          a_cnt = a_cnt + 1 ;
        end
      end
    end
  end
end
```

Rotate A, if A complete one turn, then rotate B.

Rotate B, if B complete one turn, then rotate A.

```
a_lp = 0 ;
if ( (a_cnt >=1) & (a_cnt <=12) ) begin
   a_lp[a_cnt-1] = 1'b1 ;
end
b_lp = 0 ;
if ( (b_cnt >=1) & (b_cnt <=12) ) begin
   temp = b_cnt+ b_pos -1 ;
   temp = ( temp >= 12 )? temp -12 : temp ;
   b_lp[temp] = 1'b1 ;
end
if ( ( a_cnt>12 ) & ( b_cnt > 12 )) begin
  a_cnt=0 ;
  b_cnt=0 ;
  b_pos = 0 ;
end
//$strobe ("t=%d, f=%b, a=%d, b=%d, a_flg=%b",
//$stime, flick, a_cnt, b_cnt, a_flg );
end
endtask
```

Map counter to bit position of a_lp

Map counter to bit position of b_lp

If both A and B completed one turn, initialize counters.

For debugging golden model

This task check if the output of the golden model and the target model are the same or not.

```
task cmp_chk ;
input [11:0] g_a_lp, g_b_lp ;
input [11:0] a_lp, b_lp ;
begin
 if ( g_a_lp != a_lp ) begin
   $display("at t=%d, error in a_lp  golden=%b, target=%b",
            $stime, g_a_lp, a_lp ) ;
   #10 $finish ;
 end
 else begin
   if ( g_b_lp != b_lp ) begin
     $display("at t=%d, error in b_lp  golden=%b, target=%b",
            $stime, g_b_lp, b_lp ) ;
     #10 $finish ;
   end
 end
end

endtask
```

Check ring A

Check ring B

```
initial begin
flick = 0 ;
while ( ~rst_n ) begin
   # 1 ;
end
```

Wait for reset negated.

```
 @ ( timng ) flick = 1 ;
 #CYCL flick = 0 ;
 wait ( g_b_lp[11] == 1 ) ;
 #CYCL $display("single turn check end");
```

simple test.
First A finish and then B

```
 @ ( timng ) flick = 1 ;
 #CYCL flick = 0 ;
wait ( g_a_lp[11] == 1 ) #CYCL ;
 @ ( timng ) flick = 1 ; // check if flick don't care
 wait ( g_b_lp[11] == 1 ) ;
 @ ( timng ) ;
 @ ( timng ) flick = 0 ;
 @ ( timng ) $display("single turn flick dont care check end");
```

First A finish and then B.
flick does not care after
A finished.

```
@ ( timng ) flick=1 ;
wait ( g_a_lp[11] == 1 ) ;
wait ( g_b_lp[11] == 1 ) ;
wait ( g_a_lp == 0 & g_b_lp ==0 ) ;
@ ( timng ) flick = 0 ;
#CYCL $display ("flick always 1 test end") ;
```

flick is always 1

```
@ ( timng ) flick = 1 ;
#(CYCL) flick = 0 ;
wait ( g_a_lp[11]==1 ) ;
@( timng ) flick=1 ;
#(CYCL) flick = 0 ;
wait ( g_b_lp[10] == 1 ) ;
#(CYCL*2) ;
#CYCL $display("a0->a11->b11->b0->Brot->b10->A/B off check end") ;
```

First A stop at a11
and then B.
A and B finish at
the same time.

```
@ ( timng ) flick = 1 ;
#(CYCL*2) flick = 0 ;
wait ( g_b_lp[11]==1 ) #CYCL ;
@ ( timng ) flick = 1 ;  // dont care check
wait ( g_a_lp[11] == 1 ) ;
@ ( timng ) #CYCL flick = 0 ;
#CYCL $display("a0->b0->Brot->b11->Boff/Arot check end") ;
```

B start at #0 and finish and then A

```
@ ( timng ) flick = 1 ;
#(CYCL*2) flick = 0 ;
wait ( g_b_lp[11]==1 ) ;
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_a_lp[11] == 1 ) ;
@ ( timng ) #CYCL ;
#CYCL $display("a0->b0->Brot->b11->a1->Arot->a11->A/Boff check end") ;
```

B start at #0 and at #11 B resume rotation and A/B finish at the same time.

```
@ ( timng ) flick = 1 ;
#(CYCL*2) flick = 0 ;
wait ( g_b_lp[11]==1 ) ;
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_a_lp[11] == 1 ) ;
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
#CYCL $display("flick = 1 at last A, last B check end") ;
```

Same to the previous case, but flick applied at the last lamp.

```
@ ( timng ) flick = 1 ;
#(CYCL*3) flick = 0 ;
wait ( g_b_lp[11] == 1 )  #CYCL ;
#CYCL $display("a0->b0->a1->Arot->a11->Aoff/b1->Brot->b11 check end") ;
```

See the message

```
@ ( timng ) flick = 1 ;
#(CYCL*3) flick = 0 ;
wait ( g_a_lp[11] == 1 ) ;
@ ( timng )  flick = 1 ;
#CYCL flick = 0 ;
wait ( g_b_lp[11] == 1 ) #CYCL ;
#CYCL $display("a0->b0->a1->Arot->a11->b1->Brot->b11->A/B off check end") ;
```

See the message

```
@ ( timng ) flick = 1 ;
#(CYCL*3) flick = 0 ;
wait ( g_a_lp[11] == 1 ) ;
@ ( timng )  flick = 1 ;
#CYCL flick = 0 ;
wait ( g_b_lp[6] == 1 ) ;
@ ( timng ) flick = 1 ;
wait ( g_b_lp[11] == 1 ) ;
@ ( timng ) #CYCL flick = 0 ;
#CYCL $display(
        "a0->b0->a1->Arot->a11->b1->Brot->b[6]/A off->Brot check end"
        ) ;
```

See the message

```
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_a_lp[4] == 1 ) ;
@ ( timng )  flick = 1 ;
#CYCL flick = 0 ;
wait ( g_b_lp [7] == 1 );
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_a_lp[6] == 1 );
@ ( timng ) flick = 1 ;
#(CYCL*3) flick = 0 ;
wait ( g_b_lp[11] == 1 ) ;
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_b_lp[3] == 1 ) #CYCL;
#CYCL $display("random turn A finich first check end") ;
```

```
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_a_lp[4] == 1 ) ;
@ ( timng )  flick = 1 ;
#CYCL flick = 0 ;
wait ( g_b_lp [7] == 1 );
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_a_lp[6] == 1 );
@ ( timng ) flick = 1 ;
#(CYCL*3) flick = 0 ;
wait ( g_b_lp[11] == 1 ) ;
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_a_lp[11]== 1 ) ;
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_b_lp[3] == 1 ) #CYCL;
#CYCL $display(
        "random turn A finish first and flick on last A lamp check end"
        ) ;
```

```
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_a_lp[2] == 1 ) ;
@ ( timng )  flick = 1 ;
#CYCL flick = 0 ;
wait ( g_b_lp [9] == 1 );
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_a_lp[6] == 1 );
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_a_lp[11] == 1 ) #CYCL;
#CYCL $display("random turn B finich first check end") ;
```

```
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_a_lp[2] == 1 ) ;
@ ( timng )  flick = 1 ;
#CYCL flick = 0 ;
wait ( g_b_lp [9] == 1 );
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_a_lp[6] == 1 );
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
wait ( g_b_lp[1] == 1 ) ;
@ ( timng ) flick = 1 ;
#CYCL flick = 0 ;
@ ( g_a_lp[11] == 1 ) #CYCL;
#CYCL $display(
   "random turn B finich first and flick at last B lamp check end"
      ) ;
#(CYCL*2) ;
$display("test OK, No error in the target module") ;
#5;
$finish;
end
//
endmodule
```

No bug if this sentence executed.

A sample answer

If the number of lamps in the ring is infinite, then rotating ring is toggled from A to B or from B to A every time flick becomes 1. Very simple logic is applicable to the system.
However, because of the limited number of lamps, the rings can not rotate forever. This makes the system complex.

Taking into the fact that if the last lamp is on, the ring can not keep rotating, we can chose "the last lamp on" as one of major events to determine the behavior of the system.
And we can draw a state transition table on the next page.
This table can be mapped into RTL code less than 300 lines.

However, the code is not applicable for systems having more than 2 rings.
Now, let's think of the architecture applicable to more than 2 rings system.

A state transition table for 2-ring system

last A is on	last B is on	flick \ state	INTL	A_ROT_B_NYS	A_PAU_B_ROT	A_ROT_B_PAU	A_FIN_B_ROT	A_ROT_B_FIN
0	0	0	nop	rot_a=1	rot_b=1	rot_a=1	rot_b=1	rot_a=1
		1	rot_a=1 → AR_BN	rot_b=1 → AP_BR	rot_a=1 → AR_BP	rot_b=1 → AP_BR		
	1	0			rot_a,b=1 → AR_BF	rot_a=1	rot_b=1 → INTL	
		1			rot_a=1 → AR_BP	rot_a,b=1 → AR_BF		
1	0	0		rot_a,b=1 → AF_BR	rot_b=1	rot_a,b=1 → AF_BR		rot_a=1 → INTL
		1		rot_b=1 → AP_BR	rot_a,b=1 → AF_BR	rot_b=1 → AP_BR		
	1				rot_a,b=1 → INTL			

NYS: not started yet

ROT: rotate

Pau: pause

FIN: completed

This can be mapped into RTL in about 200 to 250 code lines of RTL.

A sample answer extendable to many rings

Let's focus on what is essential to this system.

Essential operation

Make up for exceptional cases

Rotating ring changes with the flick signal.

⇨ Only a rotating ring receives flick, and it send "rotate request" to another ring. Then, when a ring finishes one turn, it also send "rotate request" to another ring, this time, not initiated by flick.

What makes essential operation inapplicable?

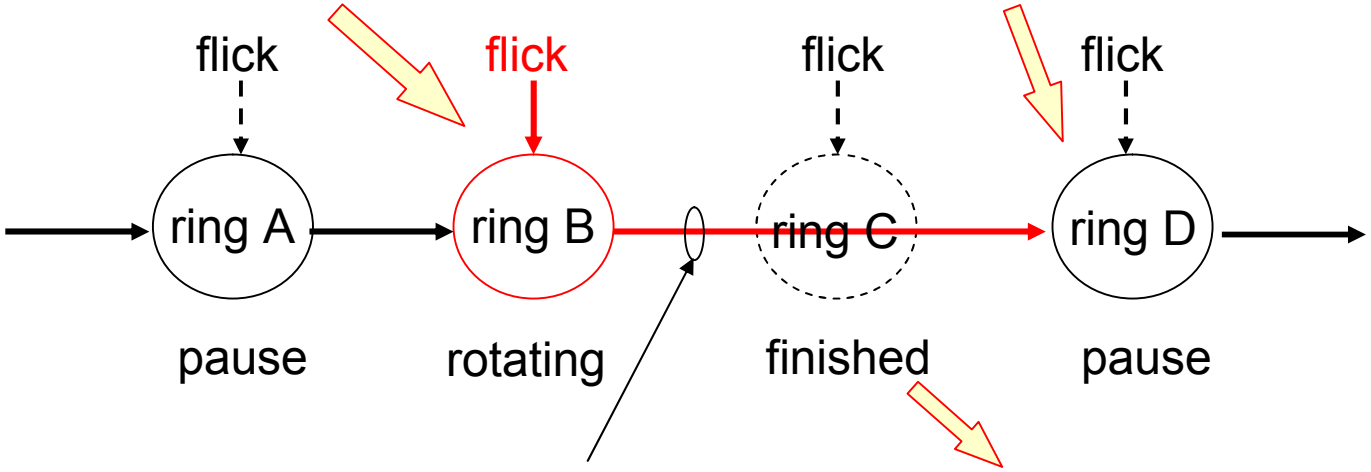
⇨ The next ring already finished or at the last lamp, so it can not start/resume rotation.

⇨ Such ring pass "rotate request" to another ring.

The idea on the previous page can be illustrated as below.

flick works only for a rotating ring, other rings neglect it.

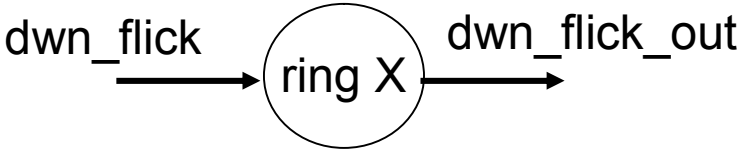
A ring starts/resumes rotation when it that receives "rotate request".



A rotating ring sends "rotate request" to the next ring when it receives flick or it finishes one turn.

A finished ring passes "rotate request" through.

Name "rotate request" dwn_flick.



goto_intl become on when all ring finished one turn.

State transition table of a ring

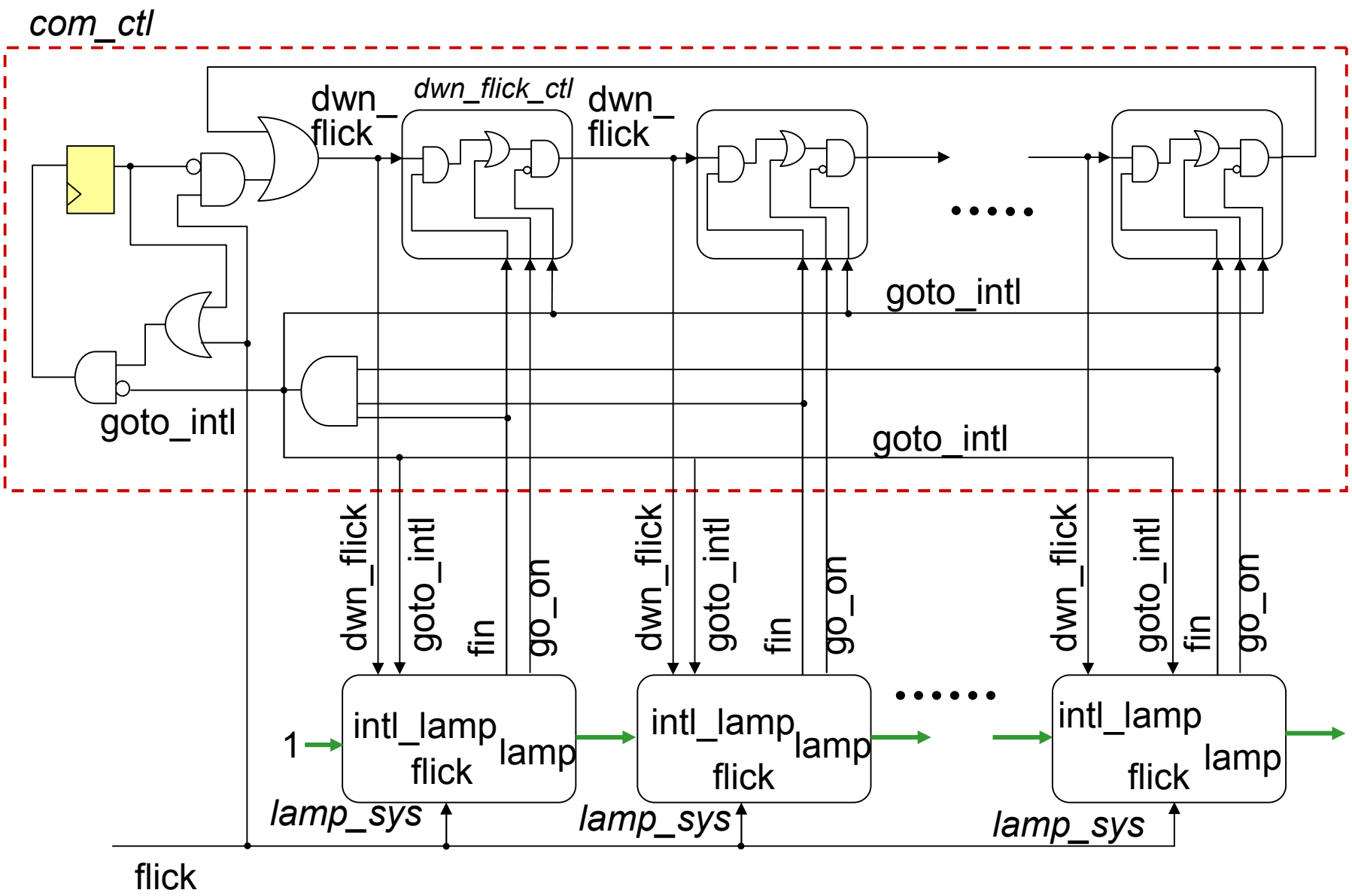
event: flick	event: dwn_flick	event: goto_intl	Not started yet INTL	Rotating ROT	Pause STP	One turn complete FIN
x	x	1		dwn_flick_out=0 ⇨ INTL	⇦	⇦
0	0	0	dwn_flick_out =0	if last lamp dwn_flick_out=1 ⇨ FIN else dwn_flick_out =0	dwn_flick_out =0	⇦
1	0	0	dwn_flick_out =0	dwn_flick_out =1 ⇨ STP		
0	1	0	turn on initial lamp dwn_flick_out =0 ⇨ ROT	⇧ (same to goto_intl=1)	if last lamp dwn_flick_out=1 ⇨ FIN else dwn_flick_out =0 ⇨ ROT	dwn_flick_out =1
1	1	0		dwn_flick_out =1 ⇨ ROT		

In the all boxes above, fin is calculated by (last lamp on or state is FIN)? 1:0 ;

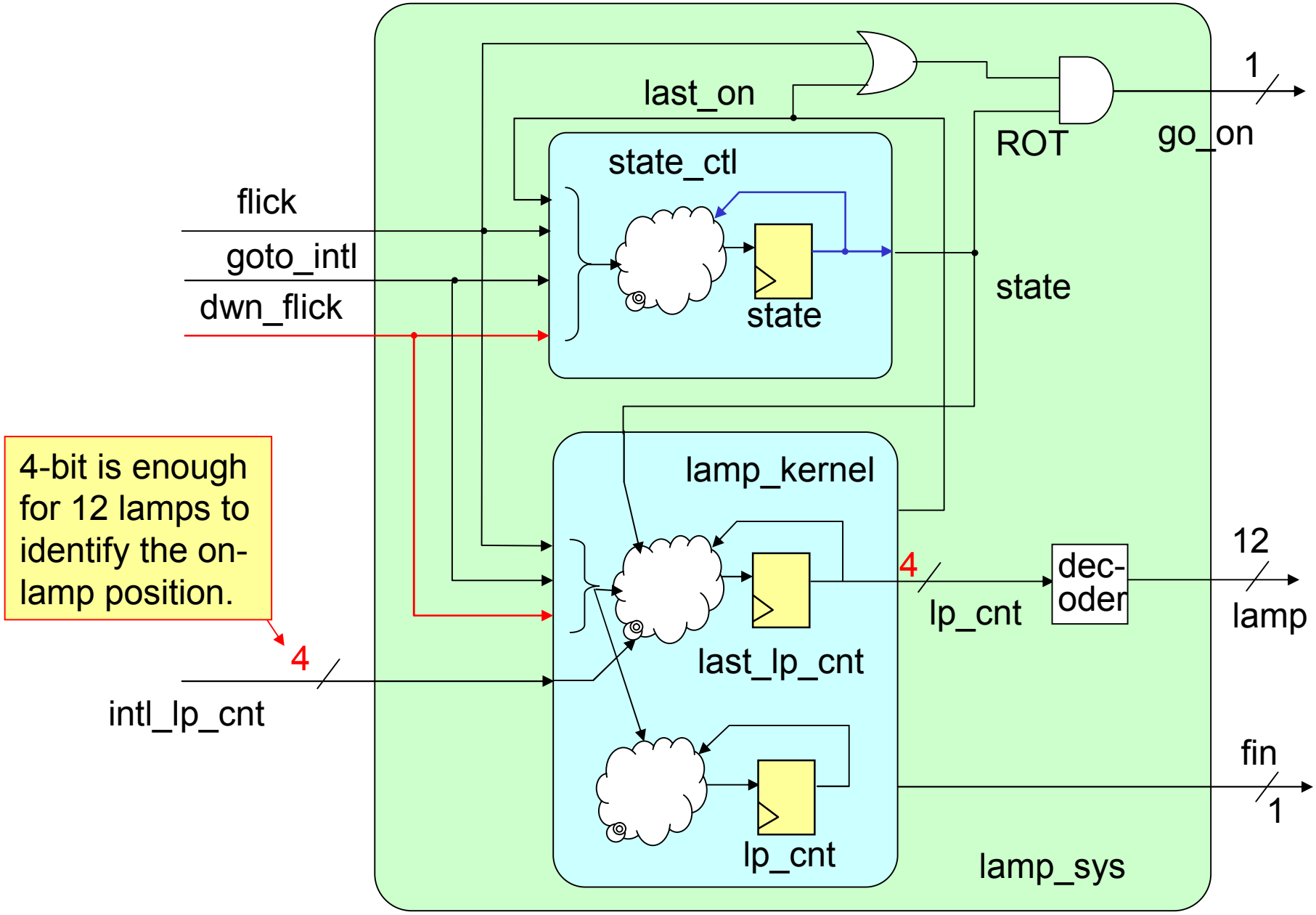
For all green boxes, lamp shall newly start, rotate, or all lamps become off.

Total system structure

The logic uses counters to keep on-lamp position of rings.



The structure of lamp_sys



A sample target code

For two rings, define is commented out.

This code is ready for 4 rings. By defining R4, it works for 4 rings, otherwise it works for 2 rings.

```
//`define R4
module multi_ring( clk, rst_n, flick,
    lamp_1, lamp_2
`ifdef R4
  , lamp_3, lamp_4
`endif
     ) ;
parameter NUM_LMP = 12 ;
parameter BW_LP_CNT = 4 ;
// first ring start count
parameter [BW_LP_CNT-1:0] INTL_CNT = 1
; //
input clk, rst_n ;
input flick ;
output [NUM_LMP-1:0]
    lamp_1, lamp_2
`ifdef R4
  , lamp_3, lamp_4
`endif
    ;

wire clk, rst_n ;
wire flick ;
wire [NUM_LMP-1:0]
    lamp_1, lamp_2
`ifdef R4
  , lamp_3, lamp_4
`endif
    ;
```

```
// common control signals
wire fin_1, fin_2
`ifdef R4
    , fin_3, fin_4
`endif
    ;
wire goto_intl ;

// internal variables
wire [BW_LP_CNT-1:0]
      lp_1_cnt, lp_2_cnt
`ifdef R4
     , lp_3_cnt, lp_4_cnt
`endif
    ;
wire   dwn_flick_1, dwn_flick_2
`ifdef R4
     ,dwn_flick_3, dwn_flick_4
`endif
    ;
// lamp position logic
assign lamp_1 = cnt_to_bit_posi(lp_1_cnt) ;
assign lamp_2 = cnt_to_bit_posi(lp_2_cnt) ;
`ifdef R4
assign lamp_3 = cnt_to_bit_posi(lp_3_cnt) ;
assign lamp_4 = cnt_to_bit_posi(lp_4_cnt) ;
`endif
```

convert from counter to bit position

```
// connection
com_ctl com_ctl_01( .clk(clk), .rst_n(rst_n), .flick(flick),
`ifdef R4
            .fin_v( {fin_4, fin_3, fin_2, fin_1 } ),
            .go_on_v( {go_on_4, go_on_3, go_on_2, go_on_1 } ),
            .dwn_flick_v( {dwn_flick_4, dwn_flick_3, dwn_flick_2, dwn_flick_1} ),
`else
            .fin_v( { fin_2, fin_1 } ),
            .go_on_v( { go_on_2, go_on_1 } ),
            .dwn_flick_v( { dwn_flick_2, dwn_flick_1} ),
`endif
            .goto_intl(goto_intl)    ) ;
```

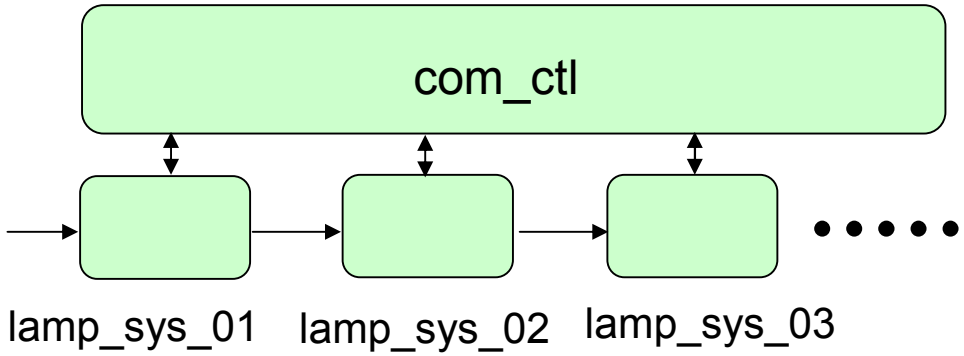
4 rings connection

2 rings connection

```
//
lamp_sys lamp_sys_01( .clk(clk), .rst_n(rst_n), .goto_intl(goto_intl),
                      .flick(flick), .dwn_flick(dwn_flick_1),
                      .intl_lp_cnt( INTL_CNT ),
                      .lp_cnt(lp_1_cnt),
                      .fin(fin_1), .go_on(go_on_1)
                  ) ;
lamp_sys lamp_sys_02( .clk(clk), .rst_n(rst_n), .goto_intl(goto_intl),
                      .flick(flick), .dwn_flick(dwn_flick_2),
                      .intl_lp_cnt( lp_1_cnt ),
                      .lp_cnt(lp_2_cnt),
                      .fin(fin_2), .go_on(go_on_2)
                  ) ;
`ifdef R4
lamp_sys lamp_sys_03( .clk(clk), .rst_n(rst_n), .goto_intl(goto_intl),
                      .flick(flick), .dwn_flick(dwn_flick_3),
                      .intl_lp_cnt( lp_2_cnt ),
                      .lp_cnt(lp_3_cnt),
                      .fin(fin_3), .go_on(go_on_3)
                  ) ;
lamp_sys lamp_sys_04( .clk(clk), .rst_n(rst_n), .goto_intl(goto_intl),
                      .flick(flick), .dwn_flick(dwn_flick_4),
                      .intl_lp_cnt( lp_3_cnt ),
                      .lp_cnt(lp_4_cnt),
                      .fin(fin_4), .go_on(go_on_4)
                  ) ;
`endif
// connection end
```

2 rings connection

4 rings connection



```
function [NUM_LMP-1:0] cnt_to_bit_posi ;
input [BW_LP_CNT-1:0] lp_cnt ;
reg   [BW_LP_CNT-1:0] bit_ix ;
//
begin
  cnt_to_bit_posi = 0 ;
  if ( lp_cnt != 0 ) begin
    bit_ix = lp_cnt -1'b1 ;
    cnt_to_bit_posi[bit_ix] = 1'b1 ;
  end
end
endfunction

endmodule
```

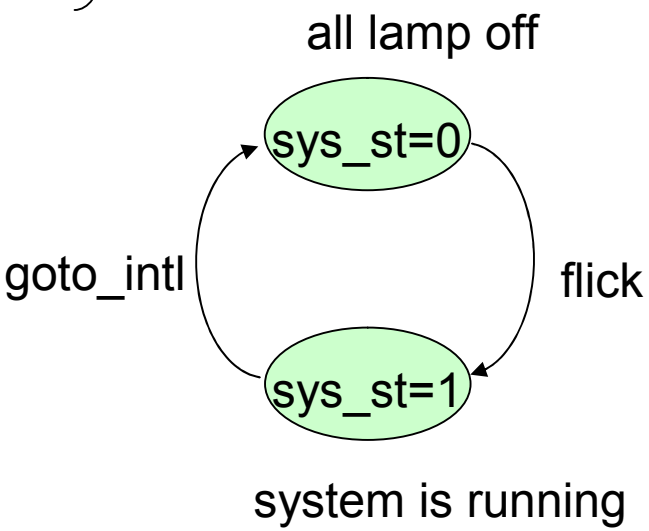
convert from counter to bit position

The followings are the lower level modules.

```
module com_ctl ( clk, rst_n, flick, fin_v, go_on_v, dwn_flick_v, goto_intl ) ;
`ifdef R4
parameter NUM_CRCL = 4 ;
`else
parameter NUM_CRCL = 2 ;
`endif
parameter FF_DLY =1 ;
input clk, rst_n ;
input flick ;
input [NUM_CRCL-1:0] fin_v ;
input [NUM_CRCL-1:0] go_on_v ;
output goto_intl ;
output [NUM_CRCL-1:0] dwn_flick_v ;

wire clk, rst_n ;
wire flick ;
wire [NUM_CRCL-1:0] fin_v ;
wire [NUM_CRCL-1:0] go_on_v ;
wire goto_intl ;
wire [NUM_CRCL-1:0] dwn_flick_v ;

// inernal variable
reg sys_st ; // 0 in initial state, 1 not in initial state
wire next_sys_st ;
wire dwn_flick_loop ; // dwn_flick_out of the Last circle

//
assign goto_intl = & fin_v[NUM_CRCL-1:0] ;
assign dwn_flick_v[0] = dwn_flick_loop | ( flick & (~sys_st) ) ;
```

```
// FF for sys_st
always @ ( posedge clk or negedge rst_n ) begin
  if ( ~rst_n ) begin
    sys_st <= 1'b0 ;
  end
  else begin
    sys_st <= #FF_DLY next_sys_st ;
  end
end

assign next_sys_st = (~goto_intl) & ( flick | sys_st )
;
```

FF to memorize "system started"



```
// connection
dwn_flick_ctl dwn_flick_ctl_01(
            .goto_intl(goto_intl),
            .dwn_flick(dwn_flick_v[0]),
            .flick(flick),
            .go_on(go_on_v[0]),
            .fin(fin_v[0]),
            .dwn_flick_out(dwn_flick_v[1])
        ) ;
dwn_flick_ctl dwn_flick_ctl_02(
            .goto_intl(goto_intl),
            .dwn_flick(dwn_flick_v[1]),
            .flick(flick),
            .go_on(go_on_v[1]),
            .fin(fin_v[1]),
`ifdef R4
            .dwn_flick_out(dwn_flick_v[2])
        ) ;
dwn_flick_ctl dwn_flick_ctl_03(
            .goto_intl(goto_intl),
            .dwn_flick(dwn_flick_v[2]),
            .flick(flick),
            .go_on(go_on_v[2]),
            .fin(fin_v[2]),
            .dwn_flick_out(dwn_flick_v[3])
        ) ;
dwn_flick_ctl dwn_flick_ctl_04(
            .goto_intl(goto_intl),
            .dwn_flick(dwn_flick_v[3]),
            .flick(flick),
            .go_on(go_on_v[3]),
            .fin(fin_v[3]),
`endif
            .dwn_flick_out(dwn_flick_loop)
        ) ;
//
endmodule
```

2 rings connection

4 rings connection

2 rings connection

```
module dwn_flick_ctl( goto_intl, dwn_flick, flick, go_on, fin, dwn_flick_out ) ;
input goto_intl, dwn_flick, flick, go_on, fin ;
output dwn_flick_out ;

wire goto_intl, dwn_flick, flick, go_on, fin ;
wire dwn_flick_out ;
//
assign dwn_flick_out = (~goto_intl) & ( go_on | ( fin & dwn_flick ) ) ;
//
endmodule

//   ************
module state_ctl ( clk, rst_n, goto_intl, flick, dwn_flick, last_on, state ) ;
parameter FF_DLY = 1 ;
parameter INTL = 2'b00 ; // initial, all lamp off
parameter ROT  = 2'b01 ; // rotating
parameter STP  = 2'b10 ; // stop
parameter FIN  = 2'b11 ; // finished one turn, all lamp off
//
input clk, rst_n ;
input goto_intl ;
input flick, dwn_flick ;
input last_on ;
output [1:0] state ;
//
wire clk, rst_n ;
wire goto_intl ;
wire flick, dwn_flick ;
wire last_on ;
reg [1:0] state ; // FF
//
reg [1:0] next_state ; // non-FF
// state transition logic
always @ ( posedge clk or negedge rst_n ) begin
  if ( ~rst_n ) state <= INTL ;
  else          state <=#FF_DLY next_state ;
end
```

FF for the state

```
always @ ( goto_intl or state or flick or dwn_flick or last_on ) begin
  if ( goto_intl ) begin  // when goto_intl==1 go to INTL state
    next_state = INTL ;
  end
  else begin
    case ( state )
      INTL : begin  // Other than first circle, start by dwn_flick
             next_state = ( dwn_flick )? ROT : INTL ;
           end
      ROT  : begin
             if ( dwn_flick ) begin // no other circle can rotate
               next_state = ROT ;
             end
             else begin // flick is effective
               if ( flick ) begin
                 next_state = STP ;
               end
               else begin
                 next_state = ( last_on )? FIN : ROT ;
               end
             end
           end
      STP  : begin  // in STP state, dwn_flick is effective
             if ( dwn_flick ) begin
               next_state = ( last_on )? FIN : ROT ;
             end
             else begin
               next_state = STP ;
             end
           end
      FIN  : begin // stay in FIN state unless goto_intl becomes 1
             next_state = FIN ;
           end
      default : begin
                next_state = 2'bxx ;
              end
    endcase
  end
end
endmodule
```

```
module lamp_kernel( clk, rst_n, goto_intl, state, flick, dwn_flick, intl_lp_cnt,
                    lp_cnt, last_on, fin ) ;
parameter FF_DLY = 1 ;
parameter NUM_LMP = 12 ;
parameter BW_LP_CNT = 4 ;
parameter INTL = 2'b00 ; // initial, all lamp off
parameter ROT  = 2'b01 ; // rotating
parameter STP  = 2'b10 ; // stop
parameter FIN  = 2'b11 ; // finished one turn, all lamp off
//
input clk, rst_n ;
input goto_intl ;
input [1:0] state ;
input flick, dwn_flick ;
input [BW_LP_CNT-1:0] intl_lp_cnt ; // initial start lamp position
output [BW_LP_CNT-1:0] lp_cnt ;
output last_on, fin ;
//
wire clk, rst_n ;
wire goto_intl ;
wire [1:0] state ;
wire flick, dwn_flick ;
wire [BW_LP_CNT-1:0] intl_lp_cnt ;
reg  [BW_LP_CNT-1:0] lp_cnt ;
wire last_on, fin ;
//
reg [BW_LP_CNT-1:0] next_lp_cnt ;  // non-FF
reg [BW_LP_CNT-1:0] last_lp_cnt ; // last lamp position FF
reg [BW_LP_CNT-1:0] next_last_lp_cnt ;  // non-FF
//
assign last_on = ( ( lp_cnt == last_lp_cnt ) &
                 ( ( state == ROT ) | ( state == STP ) ) ) ;
assign fin = ( last_on | ( state == FIN ) )? 1 : 0 ;
```

last_on and fin evaluation

```
// lamp FF logic
always @ ( posedge clk or negedge rst_n ) begin
  if( ~rst_n ) begin  // clear lamp on reset
    lp_cnt <= 0 ;
  end
  else begin
    lp_cnt <=#FF_DLY next_lp_cnt ;
  end
end
```

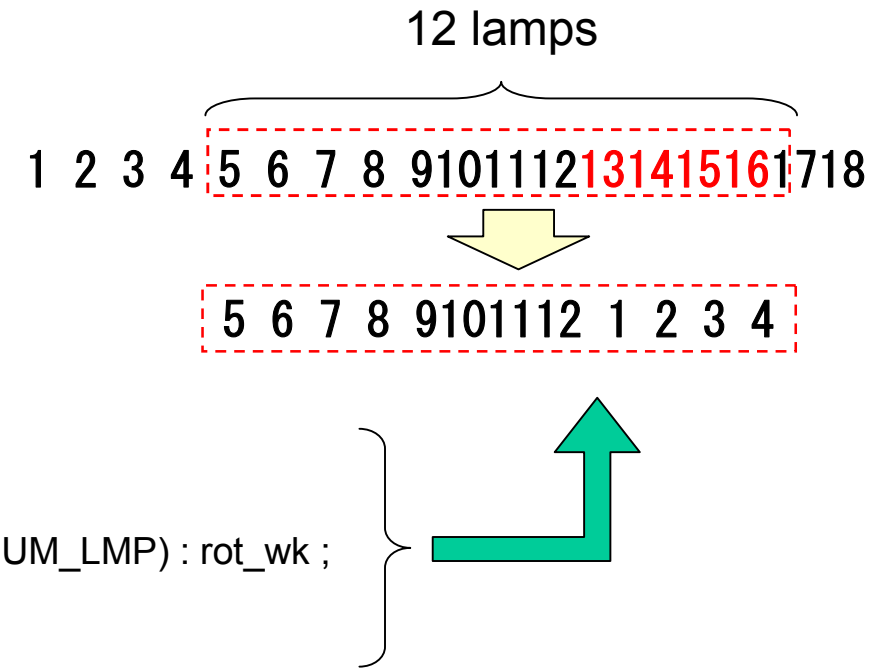
FF for lamp position counter

```
always @ ( goto_intl or state or flick or dwn_flick or last_on or intl_lp_cnt or
lp_cnt ) begin
  if ( goto_intl ) begin  // on go_to_intl clear lamp
    next_lp_cnt = 0 ;
  end
  else begin
    case ( state )
      INTL : begin
              case ( { flick, dwn_flick } )
                2'b01 : begin  // start from #0 lamp if flick=0
                        next_lp_cnt = 1 ;
                      end
                2'b11 : begin  // start from intl_lamp if flick=1
                        next_lp_cnt = intl_lp_cnt ;
                      end
                default : begin // if dwn_flick=0, do not start
                        next_lp_cnt = lp_cnt ;
                      end
              endcase
            end
```

```
      ROT  : begin
             if ( dwn_flick ) begin
               next_lp_cnt = rot_lamp(lp_cnt) ;
             end
             else begin
               if ( flick ) begin  // if flick=1 in ROT, stop
                 next_lp_cnt = lp_cnt ;
               end
               else begin
                 next_lp_cnt = ( last_on )? 0 : rot_lamp(lp_cnt) ;
               end
             end
           end
      STP  : begin
             if ( dwn_flick ) begin
               next_lp_cnt = ( last_on )? 0 : rot_lamp(lp_cnt) ;
             end
             else begin
               next_lp_cnt = lp_cnt ;
             end
           end
      default : begin
           next_lp_cnt = lp_cnt ;
         end
    endcase
  end
end
```

```
// last lamp position logic
always @ ( posedge clk or negedge rst_n ) begin
  if( ~rst_n ) begin  // set last lamp position
    last_lp_cnt <= NUM_LMP  ;
  end
  else begin
    last_lp_cnt <=#FF_DLY next_last_lp_cnt ;
  end
end
reg [BW_LP_CNT-1:0] temp_cnt ;
always @ ( state or flick or dwn_flick or intl_lp_cnt or last_lp_cnt ) begin
  case ( state )
    INTL : begin
              if ( flick & dwn_flick ) begin
                temp_cnt = intl_lp_cnt - 1'b1 ;
                 next_last_lp_cnt = (temp_cnt )?  temp_cnt : NUM_LMP ;
              end
              else begin
                 next_last_lp_cnt = last_lp_cnt ;
              end
           end
    default : begin
              next_last_lp_cnt = last_lp_cnt ;
           end
  endcase
end

function [BW_LP_CNT-1:0] rot_lamp ;
input  [BW_LP_CNT-1:0]  lp_cnt ;
reg  [BW_LP_CNT-1:0]  rot_wk ;
begin
  rot_wk = lp_cnt + 1'b1 ;
  rot_lamp = ( rot_wk > NUM_LMP )? (rot_wk - NUM_LMP) : rot_wk ;
end
endfunction

endmodule
```

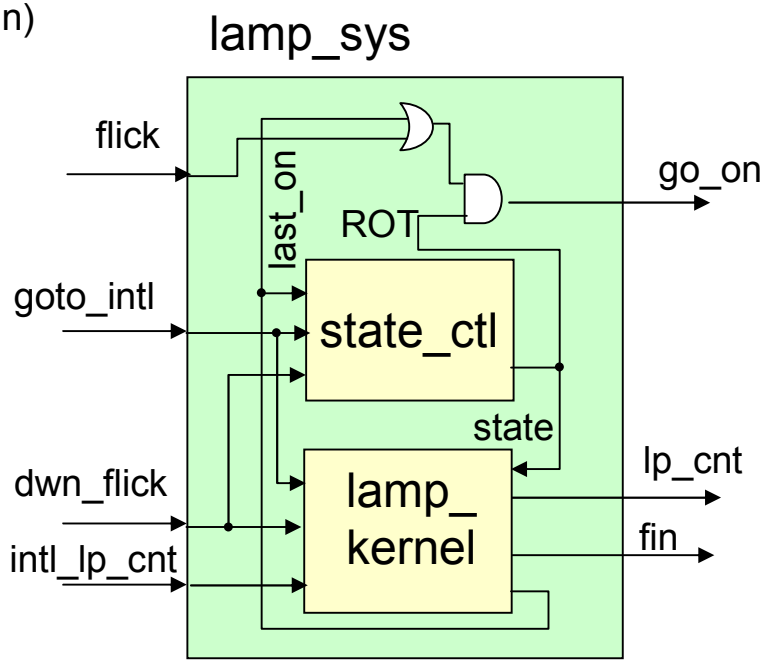
FF for last lamp position counter



```
module lamp_sys( clk, rst_n, goto_intl, flick, dwn_flick, intl_lp_cnt,
                 lp_cnt, fin, go_on ) ;
parameter NUM_LMP = 12 ;
parameter BW_LP_CNT = 4 ;
parameter FF_DLY = 1 ;

parameter INTL = 2'b00 ; // initial, all lamp off
parameter ROT  = 2'b01 ; // rotating
parameter STP  = 2'b10 ; // stop
parameter FIN  = 2'b11 ; // finished one turn, all lamp off

//
input clk, rst_n ;
input goto_intl ;  // 1 if go to INTL state
input flick, dwn_flick ;
input [BW_LP_CNT-1:0] intl_lp_cnt ;  // initial lamp position

output [BW_LP_CNT-1:0] lp_cnt ;  // On lamp position counter
output fin ;  // 1 if last lamp on or FIN state, else 0
output go_on ;  // 1 if dwn_flick must be given to the next circle
//
wire clk, rst_n ;
wire goto_intl ;
wire flick, dwn_flick ;
wire [BW_LP_CNT-1:0] intl_lp_cnt ;

wire [BW_LP_CNT-1:0] lp_cnt ;
wire fin ;
wire go_on ;

// internal variables
wire last_on ; // 1 if last lamp is on
```

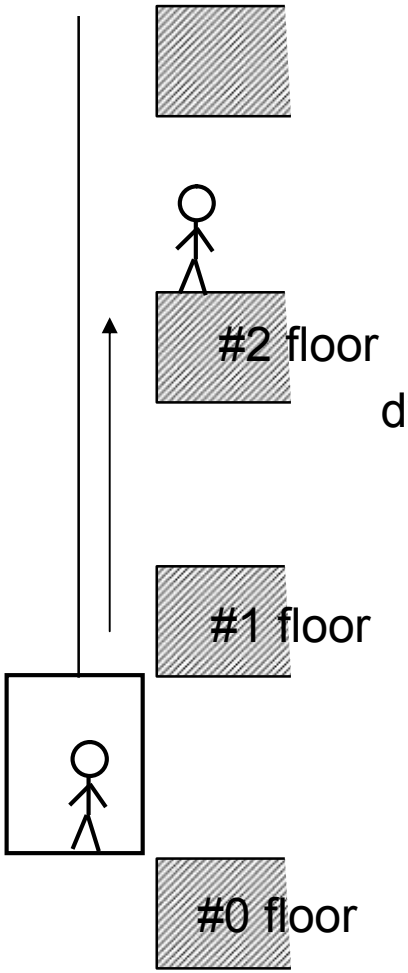
```
// control signals and connection
assign go_on = (state == ROT) & ( flick | last_on ) ;

state_ctl state_ctl_01( .clk(clk), .rst_n(rst_n),
                        .goto_intl(goto_intl),
                        .flick(flick), .dwn_flick(dwn_flick), .last_on(last_on),
                        .state(state)
                      ) ;
lamp_kernel lamp_kernel_01( .clk(clk), .rst_n(rst_n),
                        .goto_intl(goto_intl), .state(state),
                        .flick(flick), .dwn_flick(dwn_flick),
                        .intl_lp_cnt(intl_lp_cnt),
                        .lp_cnt(lp_cnt), .last_on(last_on), .fin(fin)
                      ) ;

endmodule
```

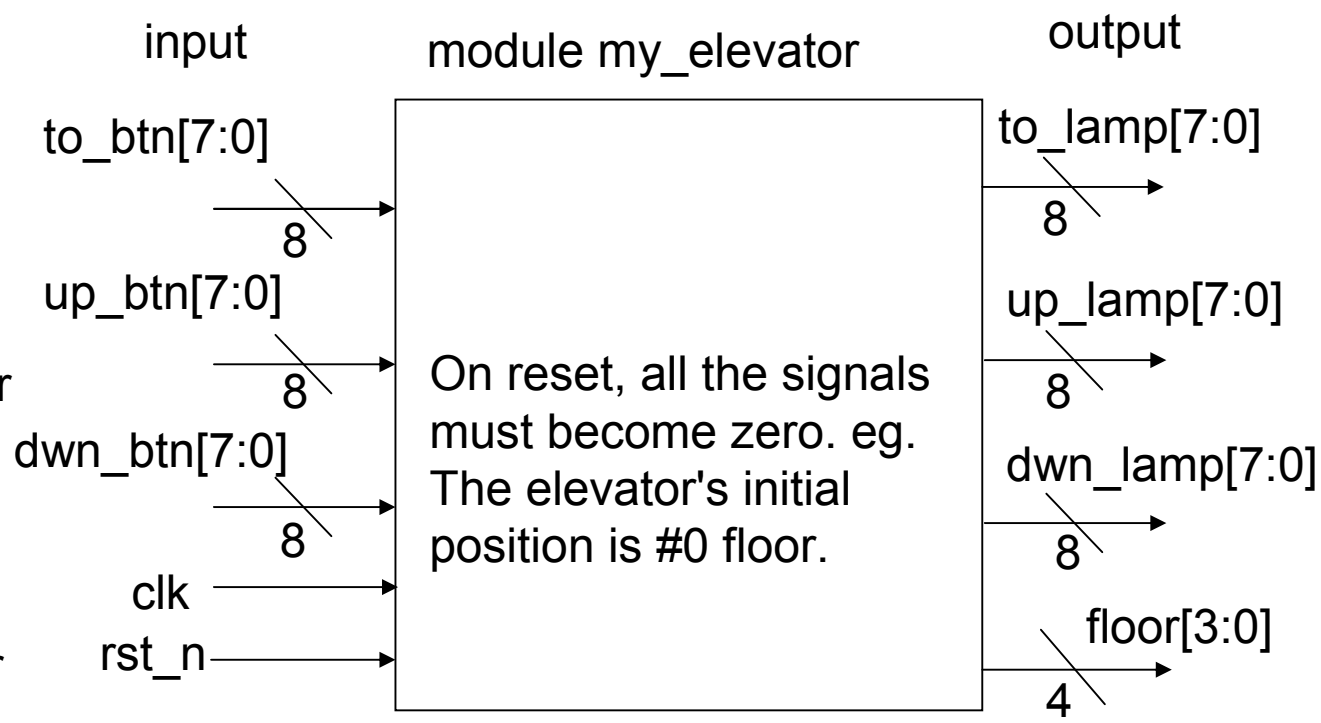
file name: multi_ring.v



Run this code for 4 rings system if you have time.

Ex 6-4. Elevator : Design a module to control an elevator cage moving up and down through 8 stories high building.





Each signal's bit position correspond to the floor. For example, up_btm[3] correspond to the up_call bottom installed at #3 floor.

to_btn[7:0] : installed in the elevator cage. If pushed, the corresponding to_lamp will become on at the next clock rise time.

to_lamp[7:0] : installed in the elevator cage. It shows the floor to stop and becomes on when to_btn pushed, and becomes off when the elevator stops at the designated floor.

up_btn[7:0] : installed on each floor. Bit i is installed on #i floor. Becomes on if up-going elevator is requested.

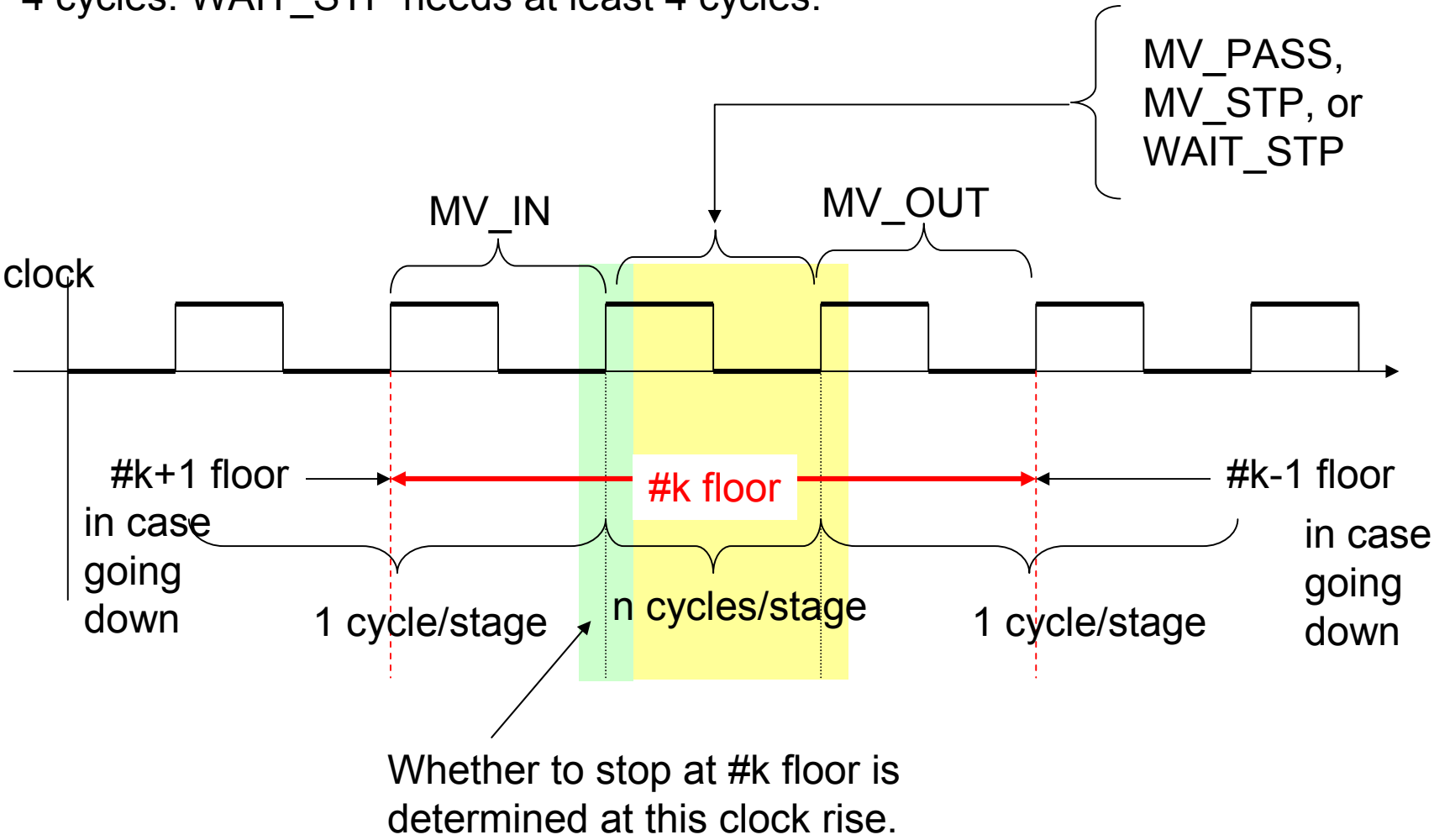
dwn_btn[7:0] : installed on each floor. Bit i is installed on #i floor. Becomes on if down-going elevator is requested.

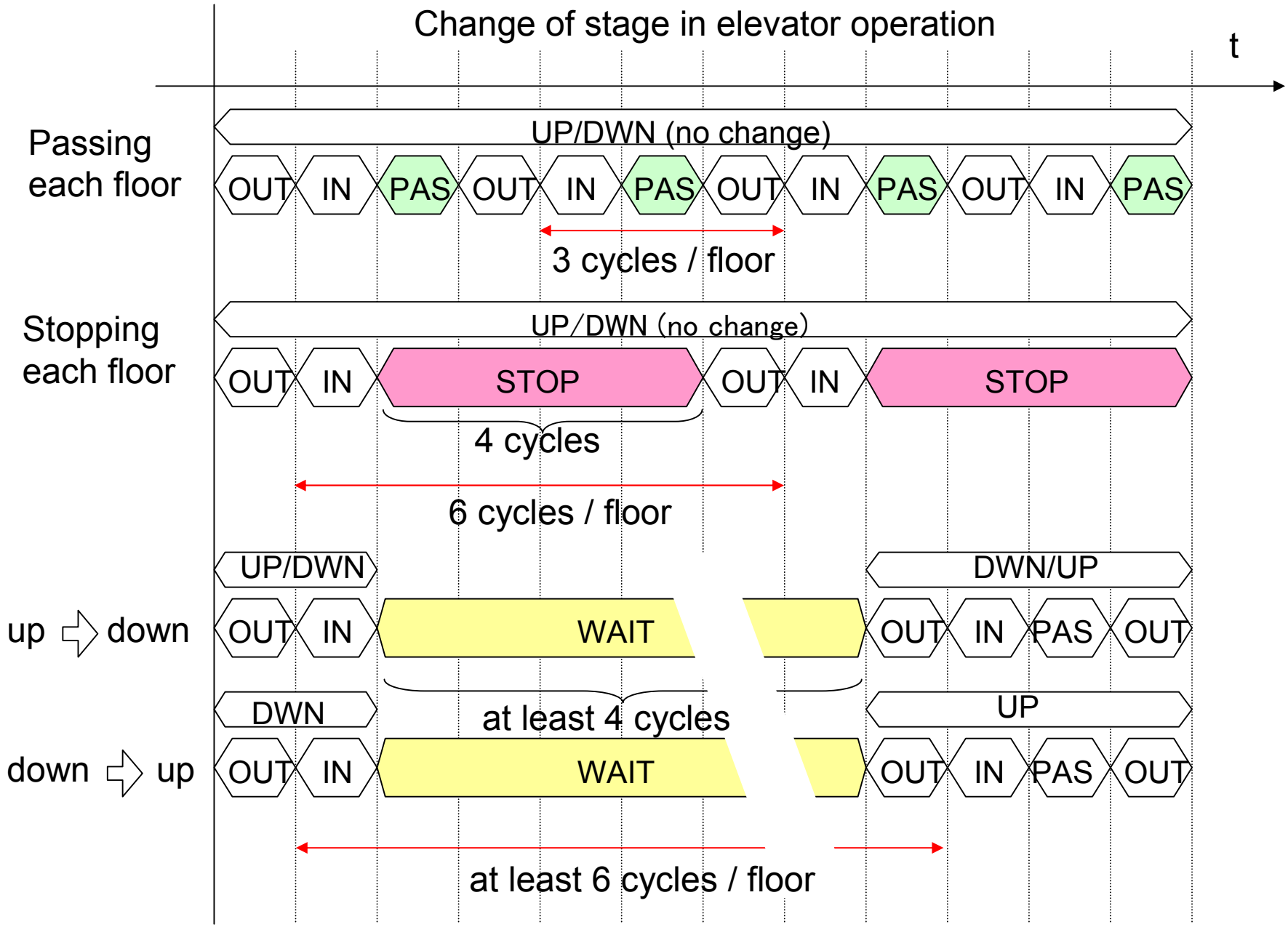
up_lamp[7:0] : installed on each floor. Bit i is installed on #i floor. Becomes on if up-going elevator request is accepted and becomes off when up_going elevator stops at the floor.

dwn_lamp[7:0] : installed on each floor. Bit i is installed on #i floor. Becomes on if down-going elevator request is accepted and becomes off when down_going elevator stops at the floor.

floor[3:0] : shows the position of the elevator in the building. floor = 3 means the elevator is at #3 floor.

The elevator cage needs 3 stages to pass one floor. The first stage is MV_IN, and the middle stage is MV_PASS, MV_STP, or WAIT_STP, and the last stage is MV_OUT. MV_IN, MV_PASS, and MV_OUT need 1 cycle. MV_STP needs 4 cycles. WAIT_STP needs at least 4 cycles.



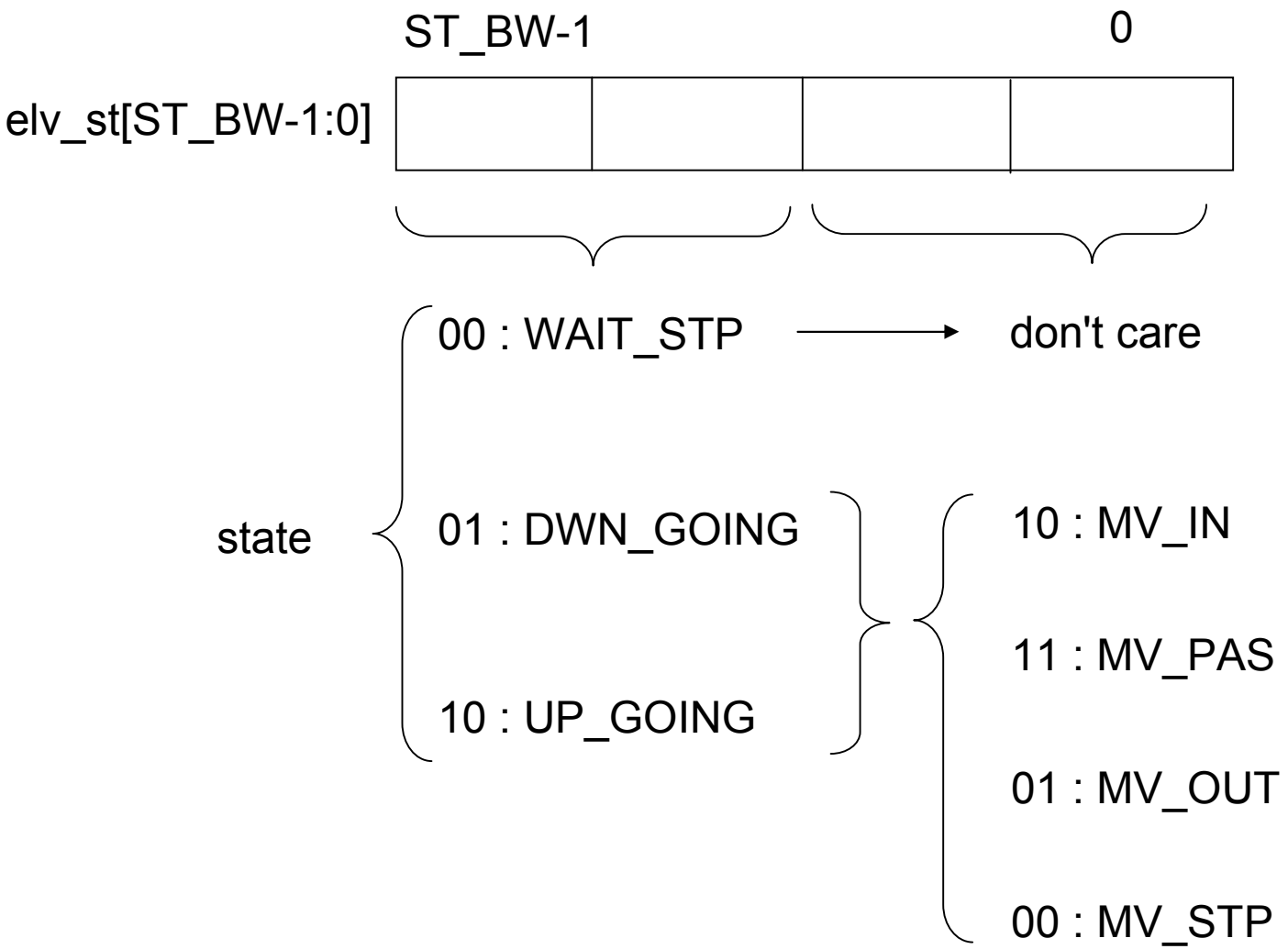


Elevator state

State (stage)			The next state
(1)	UP_ GOING or DWN_ GOING	MV_OUT	MV_IN of one floor up (incase UP_GOING) or MV_IN of one floor down (incase DWN_GOING). No change of UP/DWN direction.
(2)		MV_IN	(3)MV_IN, (4)MV_STP, or (5)WAIT_STP of the same floor depending on xx_lamp at clock rise time. UP/DWN direction may change.
(3)		MV_PAS	MV_OUT of the same floor. No change of UP/DWN direction.
(4)		MV_STP	Stay in this state 4 cycles, and then go to MV_OUT. No change of UP/DWN direction.
(5)	WAIT_STP		After staying in this state 4 cycles go to (1)MV_OUT of (5)WAIT_STP of the same floor depending on xx_lamp.

added in v25r03

Use the following state variable if needed.



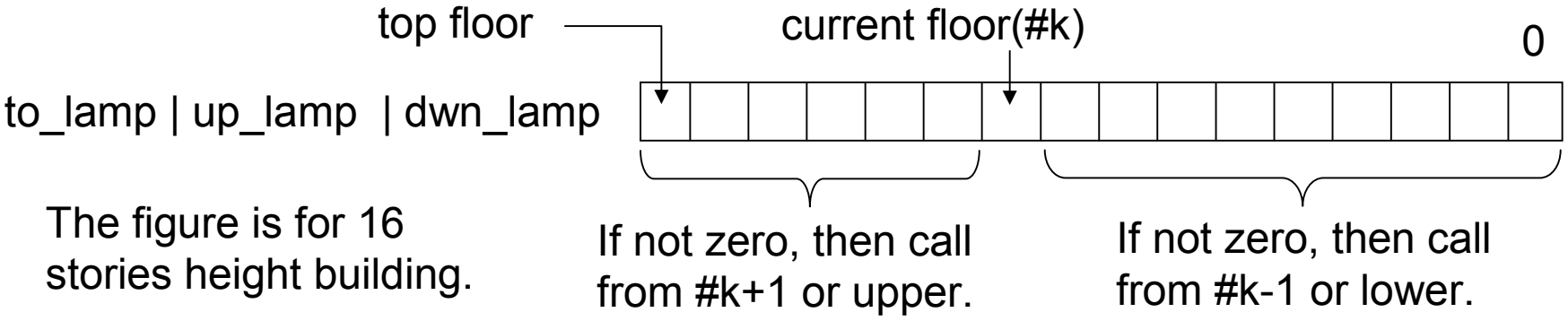
Action in each state

State	Operation at clock rise time
(1) MV_OUT	If UP_GOING add 1 to floor, if DWN_GOING subtract 1 from floor. ⇨ (2) MV_IN
(2) MV_IN	Go to { ⇨ (3) MV_PAS ⇨ (4) MV_STP ⇨ (5) WAIT_STP } by following the rule on the next page.
(3) MV_PAS	⇨ (2) MV_OUT
(4) MV_STP	After staying 4 cycles in this state ⇨ (2) MV_OUT
(5) WAIT_STP	After staying 4 cycles in this state, check the flowing at clock rise time. If any call from upper floors ⇨ (1) MV_OUT In case call from upper floors, set UP_GOING. If from lower floors, set DWN_GOING. If both at the same time set UP_GOING. else no call ⇨ (5) WAIT_STP

Call from upper/lower floors can be checked as shown on the next page.

Operation in case of MV_IN (UP_GOING)

lamp state		description
Call from #k+1 or upper	to_lamp[k]=1 or up_lamp[k]=1	⇨ (4) MV_STP
	other than above	⇨ (3) MV_PAS
No call from #k+1 or upper	Call from #k-1 or lower	UP_GOING → DWN_GOING ⇨ (4) MV_STP
	No call from #k-1 or lower	⇨ (5) WAIT_STP



The figure is for 16 stories height building.

Operation in case of MV_IN (DWN_GOING)

lamp state		description
Call from #k-1 or lower	to_lamp[k]=1 or dwn_lamp[k]=1	⇨ (4) MV_STP
	other than above	⇨ (3) MV_PAS
No call from #k-1 or lower	Call from #k+1 or upper	DWN_GOING → UP_GOING ⇨ (4) MV_STP
	No call from #k+1 or upper	⇨ (5) WAIT_STP

To which state to move from MV_IN and WAIT_STP state is determined by moving up/down direction and to_lamp, up_lamp, and dwn_lamp. These lamps are turned on by to_btn, up_btn, and dwn_btn respectively. However, depending on the state and direction, some of these bottoms are invalid as shown below. They can turn on lamps only when they are valid. Note that, bit n of xx_btn is always valid if n is not a current floor.

validity of bit k of xx_btn when the cage is at #k floor

bottom	state
to_btn	In the states, MV_IN, MV_PAS, MV_STP, or WAIT_STP, the bit k is invalid. In other cases, valid.
up_btn	In case of UP_GOING, invalid if in the state of MV_IN, MV_PAS, MV_STP, or WAIT_STP. In other cases valid.
dwn_btn	In case of DWN_GOING, invalid if in the state of MV_IN, MV_PAS, MV_STP, or WAIT_STP. In other cases valid.

Following the rule of validity of xx_btn, xx_lamp is controlled as below.

Operation applicable to to_lamp

floor	Operation
current floor (#k)	If in the state of MV_IN, MV_PAS, MV_STP, or WAIT_STP, then turn off to_lamp[k]. In other states apply the following operation. If to_btn[k] = 1, then to_lamp[k] = 1, else keep previous value.
other floor (#n) n is 0 to top floor except current floor.	If to_btn[n] = 1, then to_lamp[n] = 1, else keep previous value.

Actual turn on/off operation must be applied at clock rise time, therefore, xx_lamp shall become 1 in the next clock cycle.
Example; If to_btn[k] is on in MV_OUT state, to_lamp[k] will be on throughout MV_IN state.

Operation applicable to up_lamp

floor	Operation
current floor (#k)	When UP_GOING, and if in the state of MV_IN, MV_PAS, MV_STP, or WAIT_STP, then turn off up_lamp[k]. In other states apply the following operation. If up_btn[k] = 1, then up_lamp[k] = 1, else keep previous value.
other floor (#n) n is 0 to top floor except current floor.	If up_btn[k] = 1, then up_lamp[k] = 1, else keep previous value.

Operation applicable to dwn_lamp

floor	Operation
current floor (#k)	When DWN_GOING, and if in the state of MV_IN, MV_PAS, MV_STP, or WAIT_STP, then turn off dwn_lamp[k]. In other states apply the following operation. If dwn_btn[k] = 1, then dwn_lamp[k] = 1, else keep previous value.
other floor (#n) n is 0 to top floor except current floor.	If dwn_btn[k] = 1, then dwn_lamp[k] = 1, else keep previous value.

Now write Verilog RTL code for elevator.
You do not have to use state definition given in the previous pages. You can use your own state variables. However, the elevator cage must operate as described in the previous pages.

When writing the code, you must consider the followings;

The number of floors must be adjustable. Write the code so that you can easily change the number of floors.

Do not write flat code. Create a structured code so that the system can be tested from lower level primitive module to higher level top module.

You must also write a test bench to test your code.
You may use a test bench shown on the next pages besides your own test bench.

Work flow for solutions

(1) Investigate the function of the system.
(2) Find what is essential to the function,
(3) Write a design document including a state transition table,
(4) Write a RTL code for the target module,
(5) Run the automatic test bench on the next pages with the target module you created.
If your code is rejected, correct your code.

Copy and paste the test bench module on the next pages into you PC and run it with your my_elavator module to see if your code is OK or Not.

A sample test bench, automatic tester

```
// test bench for elevator exercise
//*********************************************
module test_my_elevator ;
parameter FL_NUM = 8 ;
parameter FL_BW = 4 ;
parameterHF_CYCL= 5 ;
parameter CYCL = HF_CYCL * 2 ;
reg rst_n, clk ;
reg[FL_NUM-1:0] to_btn,  up_btn,  dwn_btn ;
wire[FL_NUM-1:0] to_lamp, up_lamp, dwn_lamp ;
wire[FL_BW-1:0] floor ;
// work
reg [400:1] msg ;
reg [2:0] cnt ;

// connect signals to game
my_elevator my_elevator_01 ( .rst_n(rst_n), .clk(clk),
                .to_btn(to_btn), .up_btn(up_btn), .dwn_btn(dwn_btn),
                .to_lamp(to_lamp), .up_lamp(up_lamp), .dwn_lamp(dwn_lamp),
                .floor(floor)
               ) ;
// connection end

always begin      // clock generator
  clk = 1'b0 ; #HF_CYCL ;
  clk = 1'b1 ; #HF_CYCL ;
end
```

target module connection

clock generator

Display variables 2 units time after clock rise

```
always @ ( posedge clk ) begin
  #2   $strobe("t=%d, rst=%b, clk=%b,¥n tfb=%b, ucb=%b, dcb=%b,¥n tlp=%b, ulp=%b,
dlp=%b,¥n floor=%d, state=%b, cntr=%d, u_cl=%b, d_cl=%b",
            $stime, rst_n, clk, to_btn, up_btn, dwn_btn, to_lamp, up_lamp, dwn_lamp, floor,
            my_elevator_01.elv_st, my_elevator_01.st_ctl_01.sty_cntr,
            my_elevator_01.st_ctl_01.up_call, my_elevator_01.st_ctl_01.dwn_call
              ) ;
end

initial begin // give value to control variable
    rst_n = 1'b0 ;
    #CYCL rst_n = 1'b1 ;
end

initial begin
        to_btn = 8'b0000_0000 ;
        up_btn = 8'b0000_0000 ;
        dwn_btn = 8'b0000_0000 ;
$display("***** test 1 start, no movement if there is no request") ;
@( posedge clk ) #2 chk_stay( 1, "must stay at #0 with no request", 0, 8'h00, 8'h00, 8'h00 ) ;
// check no move for no request
#(CYCL*9) chk_stay( 0, "must stay at #0 with no request", 0, 8'h00, 8'h00, 8'h00 ) ;
// check no move for no request
$display("test 1 end¥n¥n") ;
```

Internal signals are observed, change this part to fit to your RTL code.

test1: Check keep no operation if no request.

test2: Check if the cage can go top floor without stopping any floor if up call from the top floor.

```
$display("test 2 start, up call from #7 only and go to #7 and wait") ;
#CYCL  up_btn = 8'b1000_0000 ;
#CYCL  chk_stay( 1, "up_lamp[7] must turn on", 0, 8'h00, 8'h80, 8'h00 ) ;
       up_btn = 8'b0000_0000 ;
@( posedge clk ) #2 chk_stay( 1, "start going up", 0, 8'h00, 8'h80, 8'h00 ) ;
#CYCL  chk_stay( 3, "keep going up", 1, 8'h00, 8'h80, 8'h00 ) ;
#CYCL  chk_stay( 3, "keep going up", 2, 8'h00, 8'h80, 8'h00 ) ;
#CYCL  chk_stay( 3, "keep going up", 3, 8'h00, 8'h80, 8'h00 ) ;
#CYCL  chk_stay( 3, "keep going up", 4, 8'h00, 8'h80, 8'h00 ) ;
#CYCL  chk_stay( 3, "keep going up", 5, 8'h00, 8'h80, 8'h00 ) ;
#CYCL  chk_stay( 3, "keep going up", 6, 8'h00, 8'h80, 8'h00 ) ;
#CYCL  chk_stay( 1, "arrive at #7", 7, 8'h00, 8'h80, 8'h00 ) ;
#CYCL  chk_stay( 0, "arrive at #7", 7, 8'h00, 8'h00, 8'h00 ) ;
#(CYCL*5) chk_stay( 1, "stay at #7", 7, 8'h00, 8'h00, 8'h00 ) ;
$display("test 2 end¥n¥n") ;
```

test3: While at top floor, down call from all the floors, and one cycle later up call from all the floors. Check go down stopping at all the floors and go up again stopping at all the floors up to #6 floor.

While at the top floor, xx_btn for the top floor must be neglected.

```
$display("test 3 start, dwn call from all the floor and next up call from all the floor") ;
      dwn_btn = 8'b1111_1111 ;
#CYCL  chk_stay( 1, "start #7 to down", 7, 8'h00, 8'h00, 8'h7f ) ;
      dwn_btn = 8'b0000_0000 ;
      up_btn = 8'b1111_1111 ;
#CYCL  chk_stay( 1, "start #7 to down", 7, 8'h00, 8'h7f, 8'h7f ) ;
      up_btn = 8'b0000_0000 ;
#CYCL  chk_stay( 1, "enter #6 to down", 6, 8'h00, 8'h7f, 8'h7f ) ;
#CYCL  chk_stay( 5, "stopping #6",      6, 8'h00, 8'h7f, 8'h3f ) ;
#CYCL  chk_stay( 1, "enter #5 to down", 5, 8'h00, 8'h7f, 8'h3f ) ;
#CYCL  chk_stay( 5, "stopping #5",      5, 8'h00, 8'h7f, 8'h1f ) ;
```

```
#CYCL  chk_stay( 1, "enter #4 to down", 4, 8'h00, 8'h7f, 8'h1f ) ;
#CYCL  chk_stay( 5, "stopping #4",      4, 8'h00, 8'h7f, 8'h0f ) ;
#CYCL  chk_stay( 1, "enter #3 to down", 3, 8'h00, 8'h7f, 8'h0f ) ;
#CYCL  chk_stay( 5, "stopping #3",      3, 8'h00, 8'h7f, 8'h07 ) ;
#CYCL  chk_stay( 1, "enter #2 to down", 2, 8'h00, 8'h7f, 8'h07 ) ;
#CYCL  chk_stay( 5, "stopping #2",      2, 8'h00, 8'h7f, 8'h03 ) ;
#CYCL  chk_stay( 1, "enter #1 to down", 1, 8'h00, 8'h7f, 8'h03 ) ;
#CYCL  chk_stay( 5, "stopping #1",      1, 8'h00, 8'h7f, 8'h01 ) ;
#CYCL  chk_stay( 1, "enter #0 to down", 0, 8'h00, 8'h7f, 8'h01 ) ;
#CYCL  chk_stay( 1, "down to up change",0, 8'h00, 8'h7f, 8'h00 ) ;
#CYCL  chk_stay( 4, "stopping #0",    0, 8'h00, 8'h7e, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #1 to up", 1, 8'h00, 8'h7e, 8'h00 ) ;
#CYCL  chk_stay( 5, "stopping #1",    1, 8'h00, 8'h7c, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #2 to up", 2, 8'h00, 8'h7c, 8'h00 ) ;
#CYCL  chk_stay( 5, "stopping #2",    2, 8'h00, 8'h78, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #3 to up", 3, 8'h00, 8'h78, 8'h00 ) ;
#CYCL  chk_stay( 5, "stopping #3",    3, 8'h00, 8'h70, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #4 to up", 4, 8'h00, 8'h70, 8'h00 ) ;
#CYCL  chk_stay( 5, "stopping #4",    4, 8'h00, 8'h60, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #5 to up", 5, 8'h00, 8'h60, 8'h00 ) ;
#CYCL  chk_stay( 5, "stopping #5",    5, 8'h00, 8'h40, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #6 to up", 6, 8'h00, 8'h40, 8'h00 ) ;
#CYCL  chk_stay( 5, "stopping #6",    6, 8'h00, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 0, "wait at  #6",    6, 8'h00, 8'h00, 8'h00 ) ;
$display("test 3 end¥n¥n") ;
```

test4: While at #6 floor, to_btn for #6, #4, #2, and #0 are on.
Check if pass #5, #3, and #1, and if stop at #4, #2, and #0.

```
$display("test 4 start, to_call#6,#4,#2,#0 and then up call from #0");
@(posedge clk ) #2 to_btn = 8'b0101_0101 ;
#CYCL  chk_stay( 1, "start going dwn", 6, 8'h15, 8'h00, 8'h00 ) ;
       to_btn = 8'b0000_0000 ;
#CYCL  chk_stay( 1, "leave #6 to dwn", 6, 8'h15, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #5",        5, 8'h15, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 2, "pass  #5 to dwn", 5, 8'h15, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #4 to dwn", 4, 8'h15, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 5, "stay #4",         4, 8'h05, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #3 to dwn", 3, 8'h05, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 2, "pass #3",         3, 8'h05, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #2 to dwn", 2, 8'h05, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 5, "stay #2",         2, 8'h01, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #1 to dwn", 1, 8'h01, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 2, "pass #1",         1, 8'h01, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #0 to dwn", 0, 8'h01, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 5, "stay #0",         0, 8'h00, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 0, "wait st  #0",     0, 8'h00, 8'h00, 8'h00 ) ;
$display("test 4 end¥n¥n") ;
```

test5: up down mixed operation

```
$display("test 5 start, random test");
@(posedge clk ) #2 to_btn = 8'b0000_0011 ;
#CYCL  chk_stay( 1, "still at #0",     0, 8'h02, 8'h00, 8'h00 ) ;
       to_btn = 8'b0000_0000 ;
#CYCL  chk_stay( 1, "start to up",     0, 8'h02, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 1, "enter #1",        1, 8'h02, 8'h00, 8'h00 ) ;
#CYCL  chk_stay( 5, "stay at  #1",     1, 8'h00, 8'h00, 8'h00 ) ;
       up_btn =  8'b0010_1010 ;
       dwn_btn = 8'b1000_0011 ;
#CYCL  chk_stay( 1, "start to up",     1, 8'h00, 8'h28, 8'h81 ) ;
       up_btn =  8'b0000_0000 ;
       dwn_btn = 8'b0000_0000 ;
#CYCL  chk_stay( 1, "leave #1",        1, 8'h00, 8'h28, 8'h81 ) ;
#CYCL  chk_stay( 3, "passing  #2",     2, 8'h00, 8'h28, 8'h81 ) ;
#CYCL  chk_stay( 1, "enter #3",        3, 8'h00, 8'h28, 8'h81 ) ;
#CYCL  chk_stay( 5, "stay  #3",        3, 8'h00, 8'h20, 8'h81 ) ;
#CYCL  chk_stay( 3, "passing  #4",     4, 8'h00, 8'h20, 8'h81 ) ;
#CYCL  chk_stay( 1, "enter #5",        5, 8'h00, 8'h20, 8'h81 ) ;
#CYCL  chk_stay( 5, "stay  #5",        5, 8'h00, 8'h00, 8'h81 ) ;
#CYCL  chk_stay( 3, "passing  #6",     6, 8'h00, 8'h00, 8'h81 ) ;
```

```
#CYCL  chk_stay( 2, "enter #7",       7, 8'h00, 8'h00, 8'h81 ) ;
#CYCL  chk_stay( 4, "stay  #7",       7, 8'h00, 8'h00, 8'h01 ) ;
#CYCL  chk_stay( 3, "passing  #6",    6, 8'h00, 8'h00, 8'h01 ) ;
#CYCL  chk_stay( 3, "passing #5",     5, 8'h00, 8'h00, 8'h01 ) ;
#CYCL  chk_stay( 3, "passing #4",     4, 8'h00, 8'h00, 8'h01 ) ;
#CYCL  chk_stay( 3, "passing #3",     3, 8'h00, 8'h00, 8'h01 ) ;
#CYCL  chk_stay( 3, "passing #2",     2, 8'h00, 8'h00, 8'h01 ) ;
       to_btn = 8'b0000_0010 ;
#CYCL  chk_stay( 1, "enter   #1",     1, 8'h02, 8'h00, 8'h01 ) ;
       to_btn = 8'b0000_0000 ;
#CYCL  chk_stay( 5, "stay at #1",     1, 8'h00, 8'h00, 8'h01 ) ;
#CYCL  chk_stay( 1, "enter #0",       0, 8'h00, 8'h00, 8'h01 ) ;
#CYCL  chk_stay( 0, "stay at #0",     0, 8'h00, 8'h00, 8'h00 ) ;
$display("test 5 end¥n¥n") ;

$display("test complete with no error¥n") ;
$finish ;
end
```

```
// **********************
task chk_stay ;  // check if stay cnt cycles, if cnt=0 check 8 cycles
input [2:0] cnt ;
input [400:1] msg ;
input [FL_BW-1:0] flr ;
input [FL_NUM-1:0] to_lp, up_lp, dwn_lp ;
begin : THIS_LOOP
  chk_elv( msg, flr, to_lp, up_lp, dwn_lp ) ;
  if ( cnt == 1 ) begin
    disable THIS_LOOP ;
  end
  else begin
    #CYCL  chk_elv( msg, flr, to_lp, up_lp, dwn_lp ) ;
    if ( cnt == 2 ) begin
      disable THIS_LOOP ;
    end
    else begin
      #CYCL  chk_elv( msg, flr, to_lp, up_lp, dwn_lp ) ;
      if ( cnt == 3 ) begin
        disable THIS_LOOP ;
      end
      else begin
        #CYCL  chk_elv( msg, flr, to_lp, up_lp, dwn_lp ) ;
        if ( cnt == 4 ) begin
          disable THIS_LOOP ;
        end
```

Task chk_stay : check if variables are the same to input arguments during cnt cycles. If cnt=0, check for 8 cycles.

```
        else begin
          #CYCL  chk_elv( msg, flr, to_lp, up_lp, dwn_lp ) ;
          if ( cnt == 5 ) begin
            disable THIS_LOOP ;
          end
          else begin
            #CYCL  chk_elv( msg, flr, to_lp, up_lp, dwn_lp ) ;
            if ( cnt == 6 ) begin
              disable THIS_LOOP ;
            end
            else begin
              #CYCL  chk_elv( msg, flr, to_lp, up_lp, dwn_lp ) ;
              if ( cnt == 7 ) begin
                disable THIS_LOOP ;
              end
              else begin
                #CYCL  chk_elv( msg, flr, to_lp, up_lp, dwn_lp ) ;
              end
            end
          end
        end
      end
    end
  end
end
endtask
```

Task chk_elv : Check if variables are same to the input arguments. If not, then output message and terminate simulation.

```
// ********************
task chk_elv ;
input [400:1] msg ;
input [FL_BW-1:0] flr ;
input [FL_NUM-1:0] to_lp, up_lp, dwn_lp ;

begin
  if ( ( flr != floor ) | ( to_lp != to_lamp ) | ( up_lp != up_lamp ) | ( dwn_lp != dwn_lamp ) )
begin
        $strobe("t=%0d, %s, floor=%0d : expected=%0d", $stime, msg, floor, flr ) ;
        $strobe(" to_lamp=%b : expected=%b, btn=%b¥n up_lamp=%b : expected=%b,
btn=%b¥n dw_lamp=%b : expected=%b, btn=%b",
                to_lamp, to_lp, to_btn, up_lamp, up_lp, up_btn, dwn_lamp, dwn_lp, dwn_btn ) ;
        $strobe("state=%b", my_elevator_01.elv_st );
        #1 $finish ;
  end
end

endtask


endmodule
```

file name: test_my_elevator.v

A sample target code

```
// elevator control logic exercise
// top module
// (c) all right reserved, 2010, RVC, Corp.
module my_elevator ( clk, rst_n, to_btn, up_btn, dwn_btn,
                     to_lamp, up_lamp, dwn_lamp, floor      ) ;
parameter FL_NUM = 8 ;
parameter FL_BW = 4 ; // bit width needed for FL_NUM, if FL_NUM=8, FL_BW must be 4.
                      // if FL_NUM=16, FL_BW must be 5. if FL_NUM=32, FL_BW must be 6.
                      // if FL_NUM=20, FL_BW must be 6
//
input clk, rst_n ;
input [FL_NUM-1:0] to_btn, up_btn, dwn_btn ;
output [FL_NUM-1:0] to_lamp, up_lamp, dwn_lamp ;
output [FL_BW-1:0] floor ;
// module connection
st_ctl st_ctl_01( .clk(clk), .rst_n(rst_n), .floor(floor),
                  .to_lamp(to_lamp), .up_lamp(up_lamp), .dwn_lamp(dwn_lamp),
                  .elv_st(elv_st)
                 ) ;
elv_ctl elv_ctl_01( .clk(clk), .rst_n(rst_n), .elv_st(elv_st),
                    .floor(floor)
                 ) ;
lamp_ctl lamp_ctl_01( .clk(clk), .rst_n(rst_n), .elv_st(elv_st), .floor(floor),
                  .to_btn(to_btn), .up_btn(up_btn), .dwn_btn(dwn_btn),
                  .to_lamp(to_lamp), .up_lamp(up_lamp), .dwn_lamp(dwn_lamp) ) ;

endmodule
```

```
// elevator state control module
// (c) all right reserved, 2010, RVC, Corp.
module st_ctl ( clk, rst_n, floor,
                to_lamp, up_lamp, dwn_lamp, elv_st ) ;
parameter FF_DLY = 1 ;
parameter FL_NUM = 8 ;  // number of floors, one bit for one floor
parameter FL_BW = 4 ;   // bit width needed to hold floor number
parameter ST_BW = 4 ;   // bit width of state to hold UP_GOING and MV_IN
parameter CNT_BW = 4 ;  // bit width of counter to count up to 4,
                        // 4 is the minimum cycle to stay in WAIT_STP state
parameter WAIT_STP = 2'b00 ;
parameter UP_GOING = 2'b10 ;
parameter DWN_GOING = 2'b01 ;
parameter MV_IN  = 2'b10 ;
parameter MV_OUT = 2'b01 ;
parameter MV_STP = 2'b00 ;
parameter MV_PAS = 2'b11 ;
//
input clk, rst_n;
input [FL_BW-1:0] floor ;
input [FL_NUM-1:0] to_lamp, up_lamp, dwn_lamp ;

output [ST_BW-1:0] elv_st ;
//
wire clk, rst_n;
wire [FL_BW-1:0] floor ;
wire [FL_NUM-1:0] to_lamp, up_lamp, dwn_lamp ;
reg [ST_BW-1:0] elv_st ; // FF
```

```
// internal variables
reg [ST_BW:0] next_elv_st ; // non-FF
wire [1:0] mv_dir, mv_st ;
reg [CNT_BW-1:0] sty_cntr ; // FF, counter to stay at least 4 cycles
reg [CNT_BW-1:0] next_sty_cntr ; // non-FF
//
always @ ( posedge clk or negedge rst_n ) begin
  if ( rst_n==1'b0 ) begin
    elv_st <=#FF_DLY { WAIT_STP, 2'b00 } ;
  end
  else begin
    elv_st <=#FF_DLY next_elv_st ;
  end
end
// connect call check module
wire [FL_NUM-1:0] floor_call ;
assign floor_call = to_lamp | up_lamp | dwn_lamp ;
call_chk call_chk_01( .floor(floor), .floor_call(floor_call),
                      .up_call(up_call), .dwn_call(dwn_call) ) ;
//
always @ ( elv_st or floor or to_lamp or up_lamp or dwn_lamp
           or up_call or dwn_call or sty_cntr ) begin
  casez ( elv_st ) // synopsys parallel_case
   { UP_GOING, MV_OUT }, { DWN_GOING, MV_OUT } : begin // MV_OUT to MV_IN
       next_elv_st = { elv_st[ST_BW-1:ST_BW-2], MV_IN } ;
    end
   { UP_GOING, MV_PAS }, { DWN_GOING, MV_PAS } : begin // MV_PAS to MV_OUT
       next_elv_st = { elv_st[ST_BW-1:ST_BW-2], MV_OUT } ;
    end
```

```
{ UP_GOING, MV_STP },
{ DWN_GOING, MV_STP } : begin  // MV_STP to MV_OUT
      // stay in the same state 0,1,2,3 cycles??
    next_elv_st=( sty_cntr == 3'd3 )? { elv_st[ST_BW-1:ST_BW-2], MV_OUT } : elv_st;
  end
{ WAIT_STP, 2'b?? } : begin   // WAIT_STP to UP_GOING/DWN MV_OUT
    if ( sty_cntr == 3'd3 ) begin  // stay in the same state 0,1,2,3 cycles??
       if ( up_call == 1'b1 ) begin
             next_elv_st = { UP_GOING, MV_OUT } ;
       end
       else begin
         next_elv_st= ( dwn_call == 1'b1 )? { DWN_GOING, MV_OUT } : elv_st ;
       end
    end
    else begin
       next_elv_st = elv_st ;
    end
  end
{ UP_GOING, MV_IN }: begin   // UP_GOING MV_IN to MV_PAS, MV_STP or WAIT_STP
       if ( up_call == 1'b1 ) begin
         next_elv_st= ( to_lamp[floor] | up_lamp[floor] )? { UP_GOING, MV_STP } :
                          { UP_GOING, MV_PAS } ;
       end
       else begin
          // if no up_call, must stop at current floor change direction
         next_elv_st=( dwn_call == 1'b1 )? { DWN_GOING, MV_STP } : { WAIT_STP, 2'b00 }
;
       end
  end
```

```
  { DWN_GOING, MV_IN }: begin   // DWN_GOING MV_IN to MV_PAS, MV_STP or WAIT_STP
        if ( dwn_call == 1'b1 ) begin
          next_elv_st =( to_lamp[floor] | dwn_lamp[floor] )? { DWN_GOING, MV_STP } :
                        { DWN_GOING, MV_PAS } ;
        end
        else begin // Change direction or Wait if no call from up nor down
          next_elv_st=( up_call == 1'b1 )? { UP_GOING, MV_STP }: { WAIT_STP, 2'b00 } ;
        end
    end
  default : begin
        next_elv_st = 4'bxxxx ;
    end
 endcase
end
```

```
// stay counter
always @ ( posedge clk or negedge rst_n ) begin
  if ( ~rst_n ) begin
     sty_cntr <=#FF_DLY 3'b000 ;
  end
  else begin
     sty_cntr <=#FF_DLY next_sty_cntr ;
  end
end

always @ ( elv_st or sty_cntr ) begin
  next_sty_cntr = 3'b000 ;
  casez ( elv_st ) // synopsys parallel_case
    { UP_GOING, MV_STP }, { DWN_GOING, MV_STP },
    { WAIT_STP, 2'b?? } : begin  // count 4, from 0 to 3, to stay at least 4 cycles
          next_sty_cntr= ( sty_cntr < 3'b011 ) ? sty_cntr + 1'b1 : sty_cntr ;
      end
    default : begin
            next_sty_cntr = 0 ;
      end
  endcase
end

endmodule
```

```
// local module in st_ctl;
// check if there are call from upper floors or from lower floors
// (c) all right reserved, RVC, Corp.
//
module call_chk ( floor, floor_call, up_call, dwn_call ) ;
parameter FL_NUM = 8 ;
parameter FL_BW = 4 ;
//
input [FL_BW-1:0] floor ;
input [FL_NUM-1:0] floor_call ;
output up_call, dwn_call ;  // up_call is 1 if there is call from upper floor
//
wire [FL_BW-1:0] floor ;
wire [FL_NUM-1:0] floor_call ;
wire up_call, dwn_call ;

// internal varibale
wire [FL_NUM-1:0] up_wk, dwn_wk ;

// check floor_call[FL_NUM-1:floor+1] and floor_call[floor-1:0]
assign  { up_wk, dwn_wk } = { floor_call, {FL_NUM{1'b0}} } >> floor ;
assign  up_call = ( up_wk[FL_NUM-1:1] )? 1'b1 : 1'b0 ;
assign  dwn_call = ( dwn_wk[FL_NUM-1:0] )? 1'b1 : 1'b0 ;
//
endmodule
```

```
// lamp control moduel, turn on and off lamps
// (c) all right reserved, RVC, Corp.
//
module lamp_ctl ( clk, rst_n, elv_st, floor,
                  to_btn, up_btn, dwn_btn,
                  to_lamp, up_lamp, dwn_lamp ) ;
parameter FF_DLY = 1 ;
parameter FL_NUM = 8 ;
parameter FL_BW = 4 ;
parameter ST_BW = 4 ;
parameter WAIT_STP = 2'b00 ;
parameter UP_GOING = 2'b10 ;
parameter DWN_GOING = 2'b01 ;
parameter MV_IN  = 2'b10 ;
parameter MV_OUT = 2'b01 ;
parameter MV_STP = 2'b00 ;
parameter MV_PAS = 2'b11 ;
//
input clk, rst_n;
input [ST_BW-1:0] elv_st ;
input [FL_BW-1:0] floor ;
input [FL_NUM-1:0] to_btn, up_btn, dwn_btn ;
output [FL_NUM-1:0] to_lamp, up_lamp, dwn_lamp ;
//
wire clk, rst_n;
wire [ST_BW-1:0] elv_st ;
wire [FL_BW-1:0] floor ;
wire [FL_NUM-1:0] to_btn, up_btn, dwn_btn ;
```

```
reg [FL_NUM-1:0] to_lamp, up_lamp, dwn_lamp ; // FF,

// internal register
reg [FL_NUM-1:0] next_to_lamp, next_up_lamp, next_dwn_lamp ; // non-FF

// internal variables
wire [1:0] mv_dir, mv_st ;
//
assign { mv_dir, mv_st } = elv_st ;
//
//
always @ ( posedge clk or negedge rst_n ) begin
  if ( ~rst_n ) begin
      to_lamp[FL_NUM-1:0]  <=#FF_DLY {FL_NUM{1'b0}} ;
      up_lamp[FL_NUM-1:0]  <=#FF_DLY {FL_NUM{1'b0}} ;
      dwn_lamp[FL_NUM-1:0] <=#FF_DLY {FL_NUM{1'b0}} ;
  end
  else begin
      to_lamp[FL_NUM-1:0]  <=#FF_DLY next_to_lamp ;
      up_lamp[FL_NUM-1:0]  <=#FF_DLY next_up_lamp ;
      dwn_lamp[FL_NUM-1:0] <=#FF_DLY next_dwn_lamp ;
  end
end
```

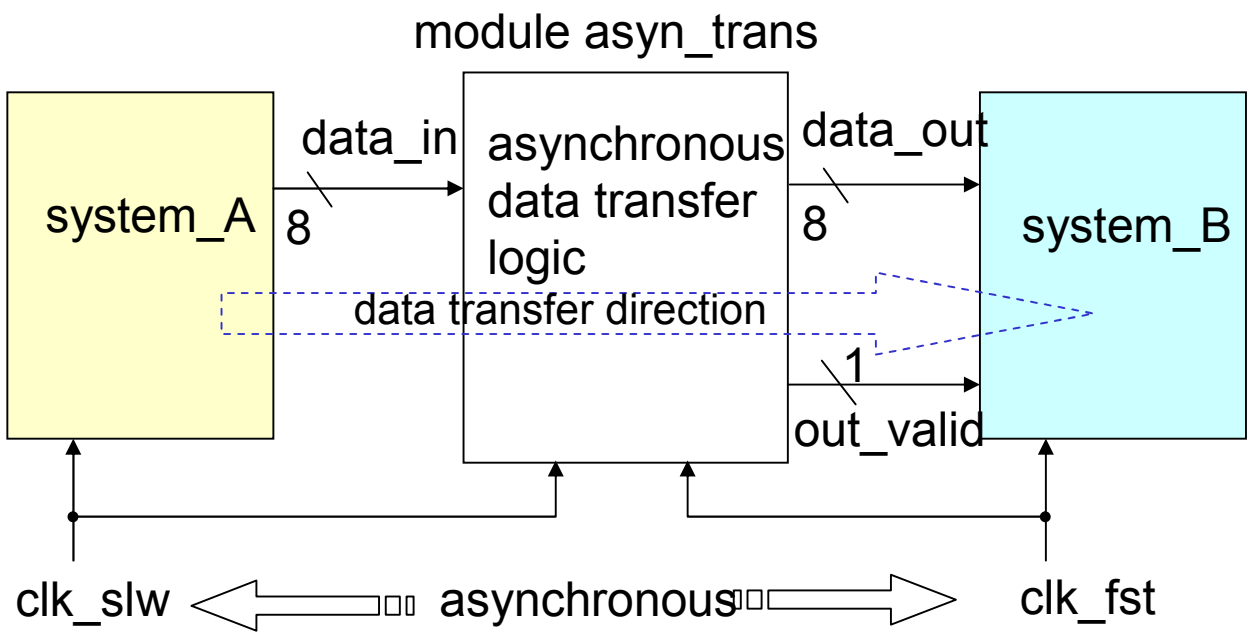
```
always @ ( elv_st or floor or to_btn or to_lamp ) begin
  next_to_lamp = to_lamp | to_btn ;
  casez ( elv_st ) // synopsys  parallel_case
    { UP_GOING, MV_IN }, { UP_GOING, MV_PAS }, { UP_GOING, MV_STP },
    { DWN_GOING, MV_IN }, { DWN_GOING, MV_PAS }, { DWN_GOING, MV_STP },
    { WAIT_STP, 2'b?? } : begin
            next_to_lamp[floor] = 1'b0 ;
        end
  endcase
end
always @ ( elv_st or floor or up_btn or up_lamp ) begin
  next_up_lamp = up_lamp | up_btn ;
  casez ( elv_st ) // synopsys parallel_case
    { UP_GOING, MV_IN }, { UP_GOING, MV_PAS }, { UP_GOING, MV_STP },
    { WAIT_STP, 2'b?? } : begin
            next_up_lamp[floor] = 1'b0 ;
        end
  endcase
end
always @ ( elv_st or floor or dwn_btn or dwn_lamp ) begin
  next_dwn_lamp = dwn_lamp | dwn_btn ;
  casez ( elv_st ) // synopsys  parallel_case
    { DWN_GOING, MV_IN }, { DWN_GOING, MV_PAS }, { DWN_GOING, MV_STP },
    { WAIT_STP, 2'b?? } : begin
            next_dwn_lamp[floor] = 1'b0 ;
        end
  endcase
end
//
endmodule
```

```
// elevator position control moduel, count up or down floor
// (c) all right reserved, RVC, Corp.
//
module elv_ctl ( clk, rst_n, elv_st, floor ) ;
//
parameter FL_BW = 4 ;
parameter ST_BW = 4 ;
parameter FF_DLY = 1 ;
parameter WAIT_STP = 2'b00 ;
parameter UP_GOING = 2'b10 ;
parameter DWN_GOING = 2'b01 ;
parameter MV_IN  = 2'b10 ;
parameter MV_OUT = 2'b01 ;
parameter MV_STP = 2'b00 ;
parameter MV_PAS = 2'b11 ;
//
input clk, rst_n ;
input [ST_BW-1:0] elv_st ;
output [FL_BW-1:0] floor ;
//
wire clk, rst_n ;
wire [ST_BW-1:0] elv_st ;
//
reg [FL_BW-1:0] floor ;
```

```
// internal variables
wire [1:0] mv_dir, mv_st ;
reg [FL_BW-1:0] next_floor ; // non-FF
//
assign { mv_dir, mv_st } = elv_st ;
always @ ( posedge clk or negedge rst_n ) begin
  if ( ~rst_n ) begin
    floor <=#FF_DLY {(FL_BW){1'b0}} ;  // initial floor = #0
  end
  else begin  // update floor only at MV_OUT stage
    if ( ( mv_dir !=2'b00 ) &  ( mv_st == MV_OUT ) ) begin
        floor <=#FF_DLY next_floor ;
    end
  end
end
always @ ( mv_dir or floor ) begin
  case ( mv_dir ) // sysnopsys parallel_case
    WAIT_STP : begin        next_floor = floor ;                end
    UP_GOING : begin        next_floor = floor + 1'b1 ;         end
    DWN_GOING : begin       next_floor = floor - 1'b1 ;         end
    default : begin         next_floor = {FL_BW{1'bx}} ;  end
  endcase
end
//
//
endmodule
```

file name: my_elevator

Ex 6-5. Asynchronous data transfer : Design data transfer logic which can transfer data from slow clock system to fast clock system, where two clock are running independently, that is, asynchronous.



(1) clk_fst is running at least 6 times faster than clk_slw,
(2) do not use cyclic buffer, use only 8-bit DFFs,
(3) data_in is directly connected to the output FF of system A.
(4) hold out_valid to 1 only for one clk_fst cycle while data_out is valid.

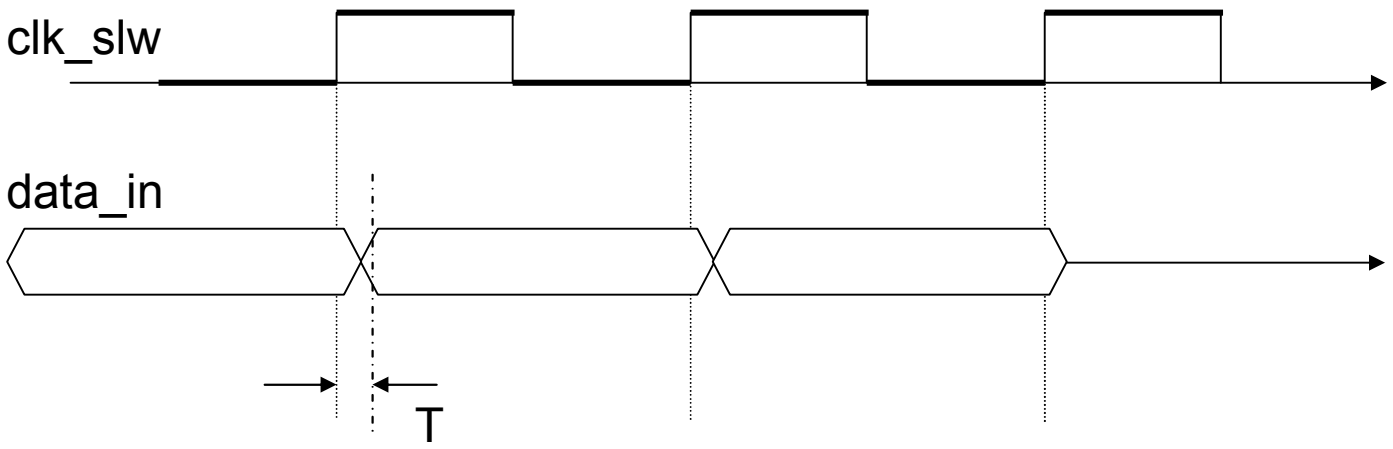
Detailed specification:

1) The time chart for data_in must be as shown below.
   T + FF's setup/hold time is much shorter than 1 cycle of clk_fst.

2) clk_slw's rise time is shorter than clk_fst's half cycle.

3) Duty of clk_slw is 50%. (While clk_slw is 1, clk_fst rises at least 4 times.

4) out_valid is on for only one cycle of clk_fst.



Work flow for solutions

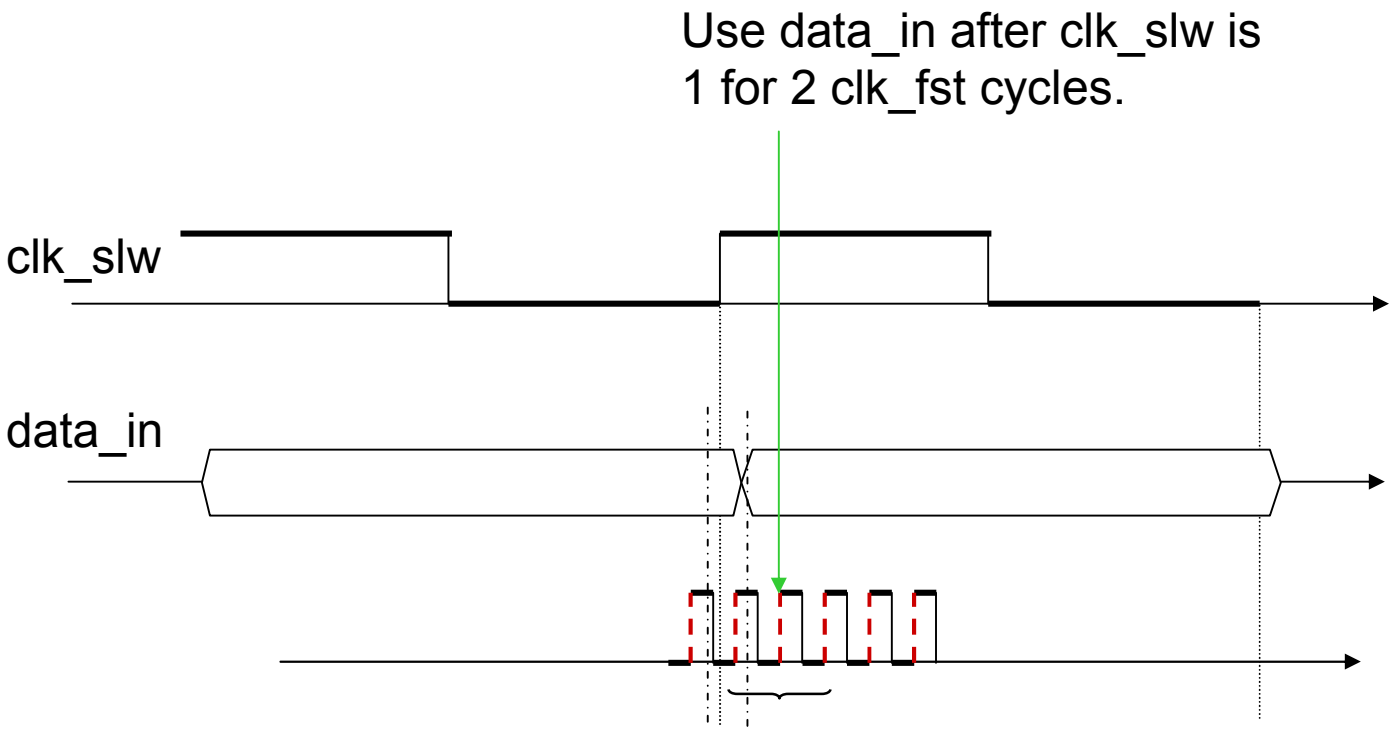
(1) Investigate the function of the system.
(2) Find what is essential to the function,
(3) Write a design document ,
(4) Write a RTL code for the target module,
(5) Run a test bench to see if your code is OK or not.

No sample test module is given for this exercise.
You have to create your own test bench.

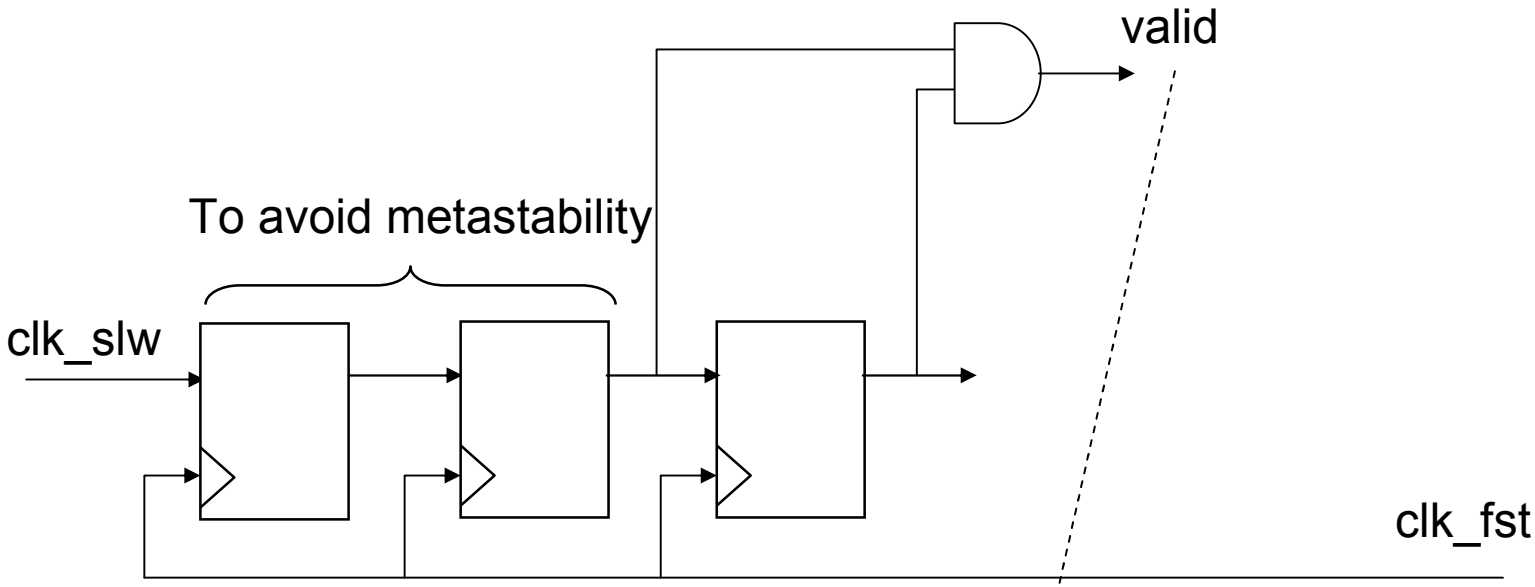
See the next pages for further understanding of the logic you have to implement.

How to know the timing of data_in is correctly received.

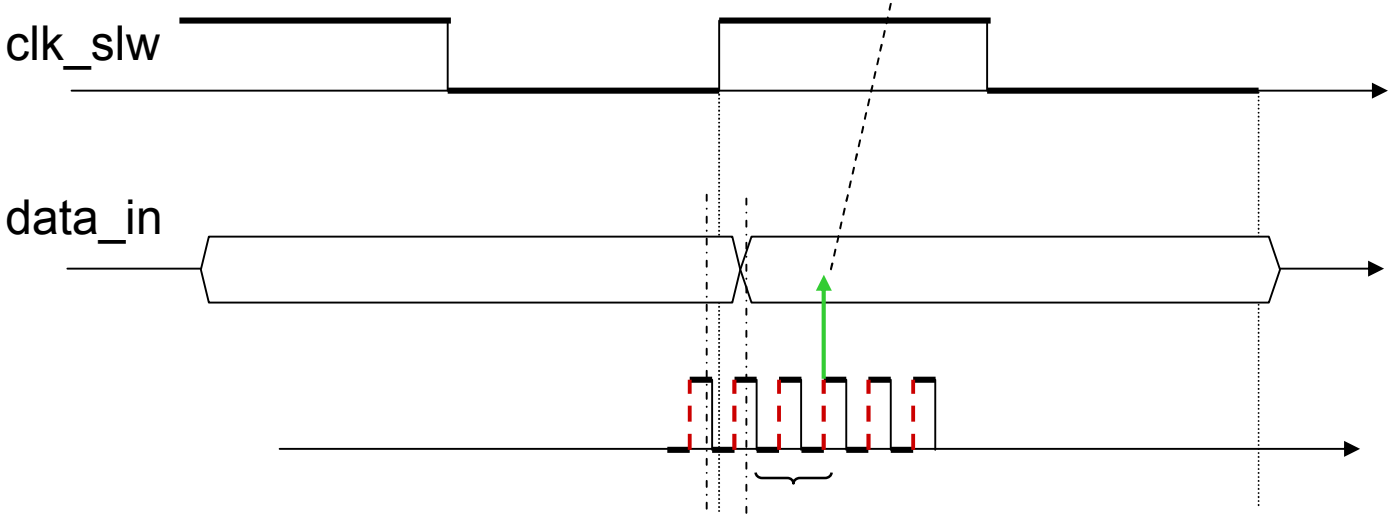
clk_fst is 6 times faster than clk_slw, therefore it is assured that at least 5 rise edge of clk_fst during clk_slw is 1.
To ensure using data_in after clk_slw rise and stable, use data_in at the timing when sampled clk_slw is continuously 1 for 2 clock cycles.

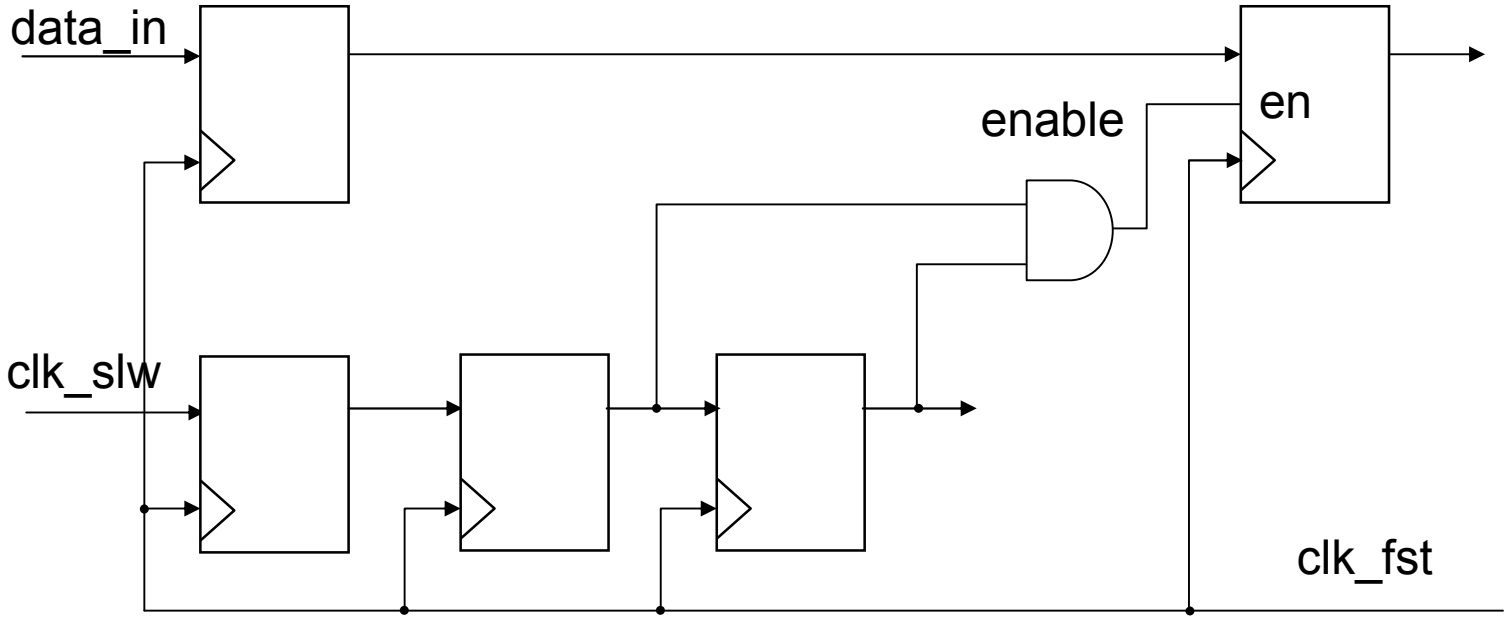


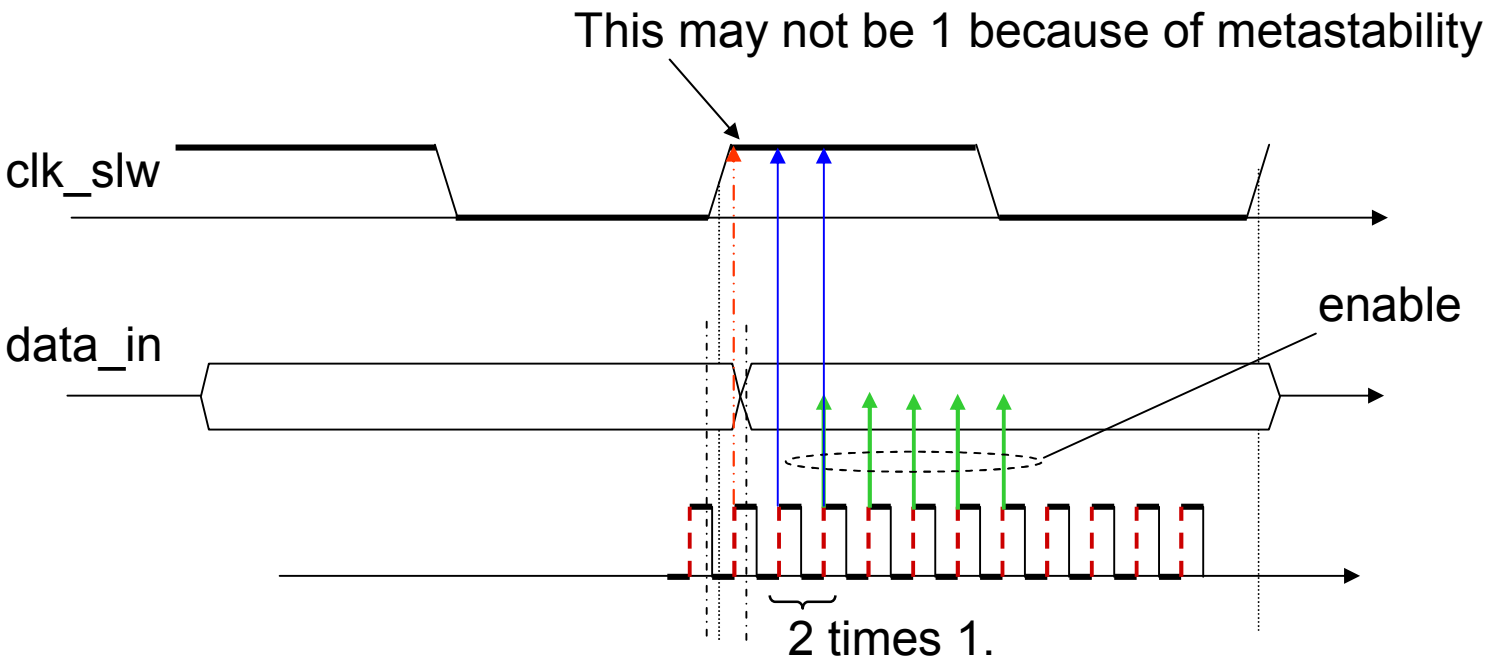
The first 1 may be caused by metastability, but using second 1 makes it sure that data_in is stable.



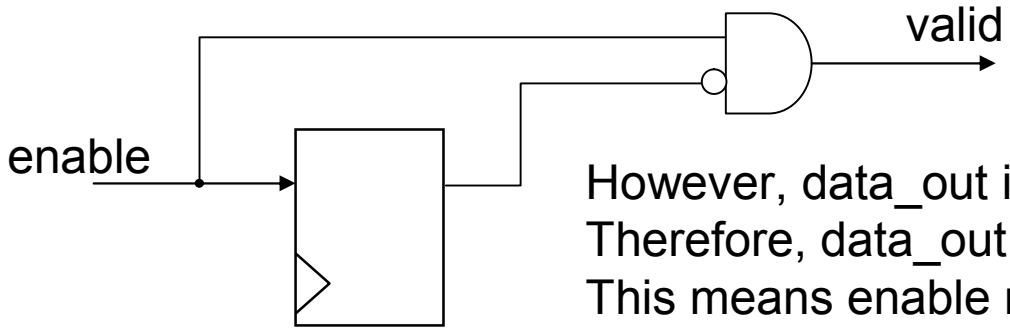
We can use this valid signal as enable signal to sample data_in.





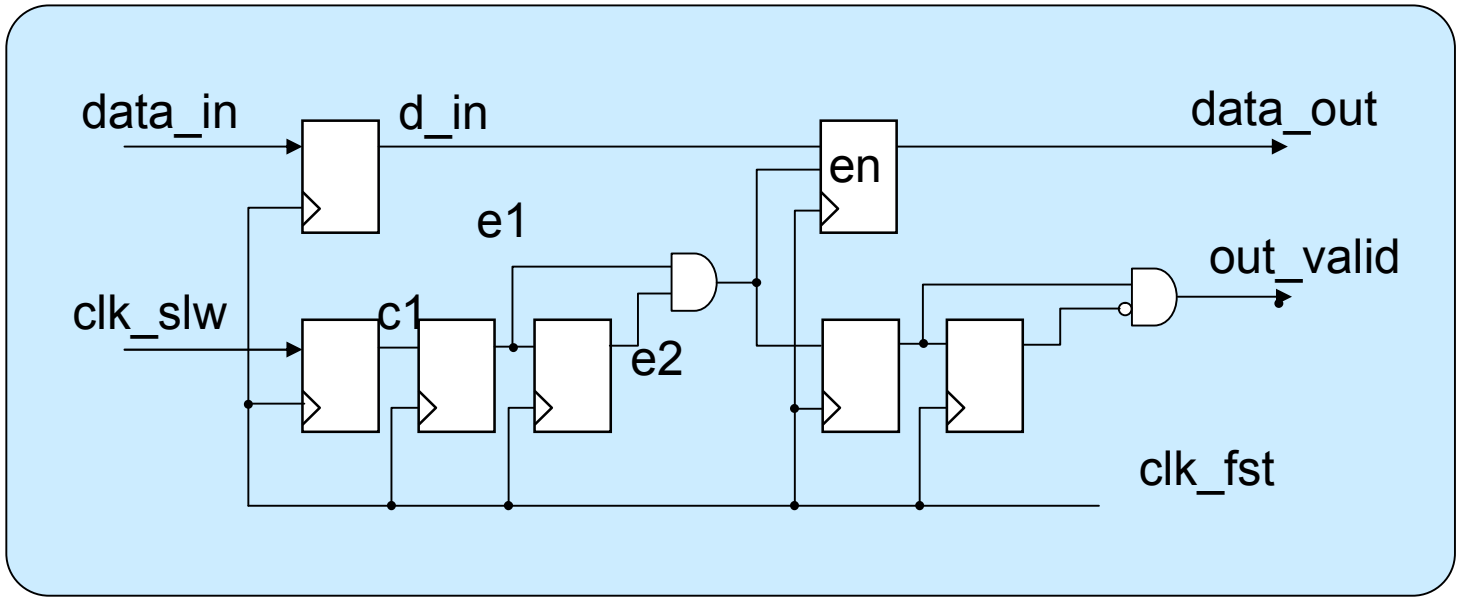


To make valid signal 1 only one clk_fst cycle, use the following logic.

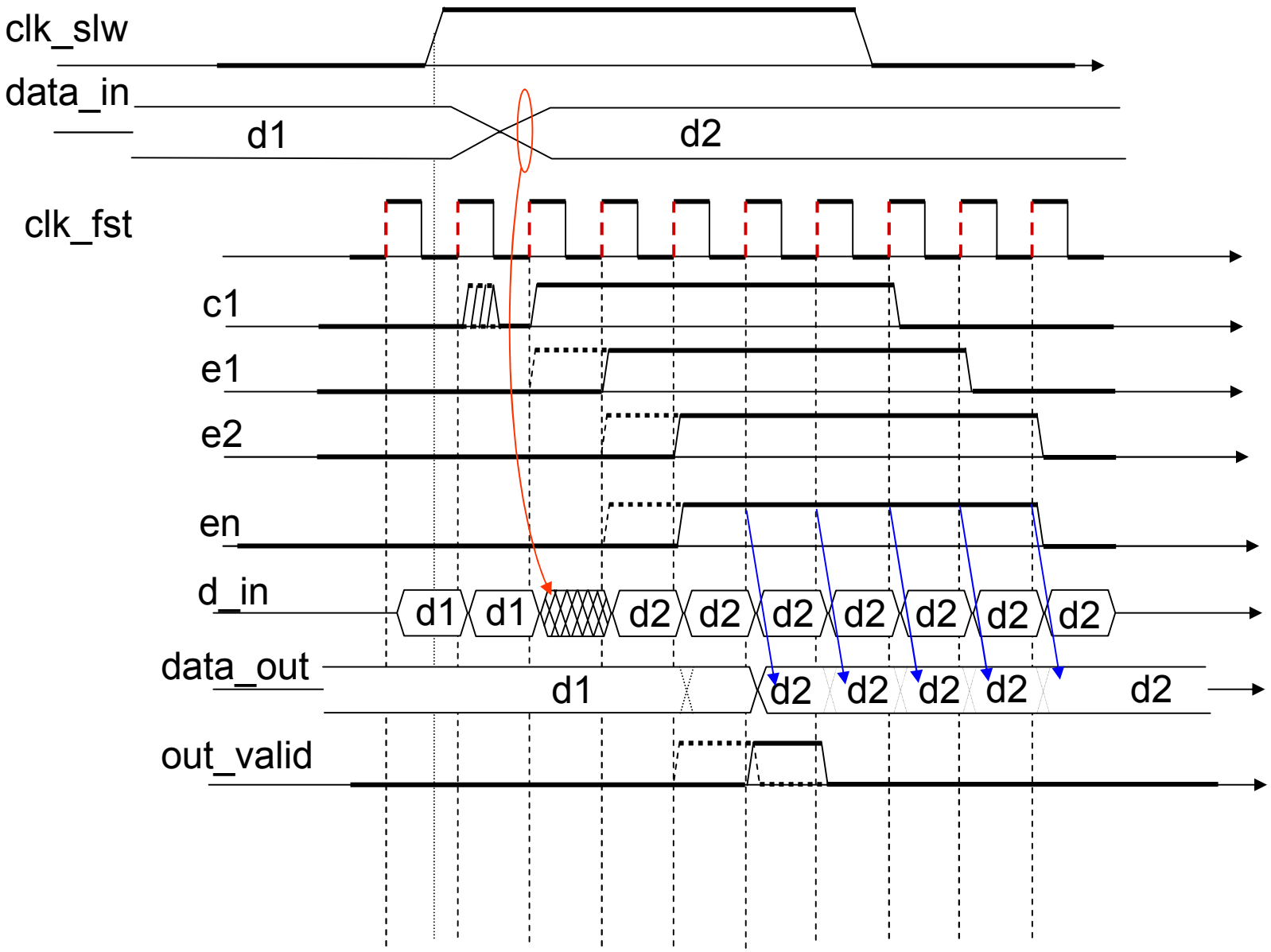


However, data_out is set by enable signal. Therefore, data_out is delayed one cycle. This means enable must be delayed one cycle.

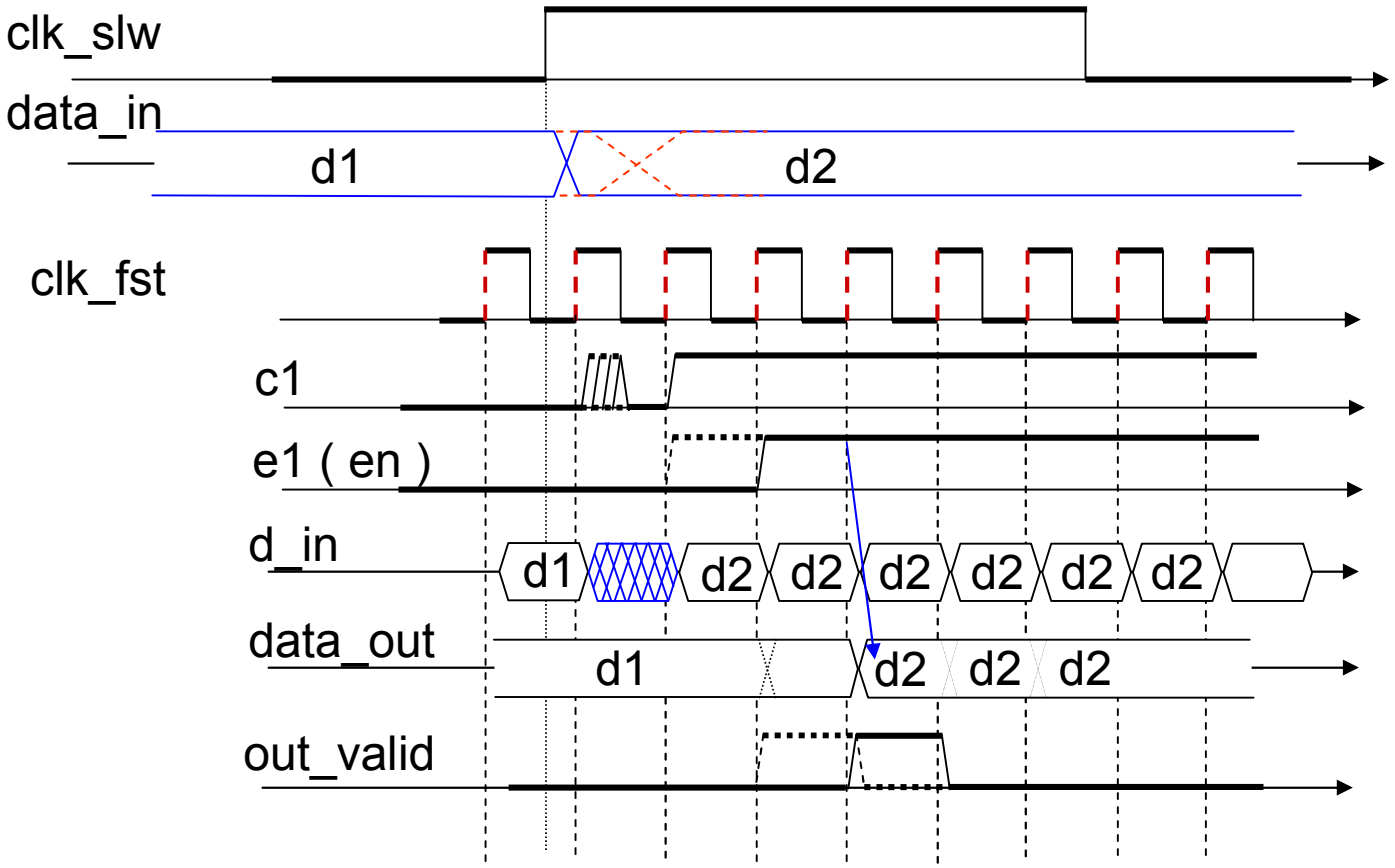
The total logic shall look like below.



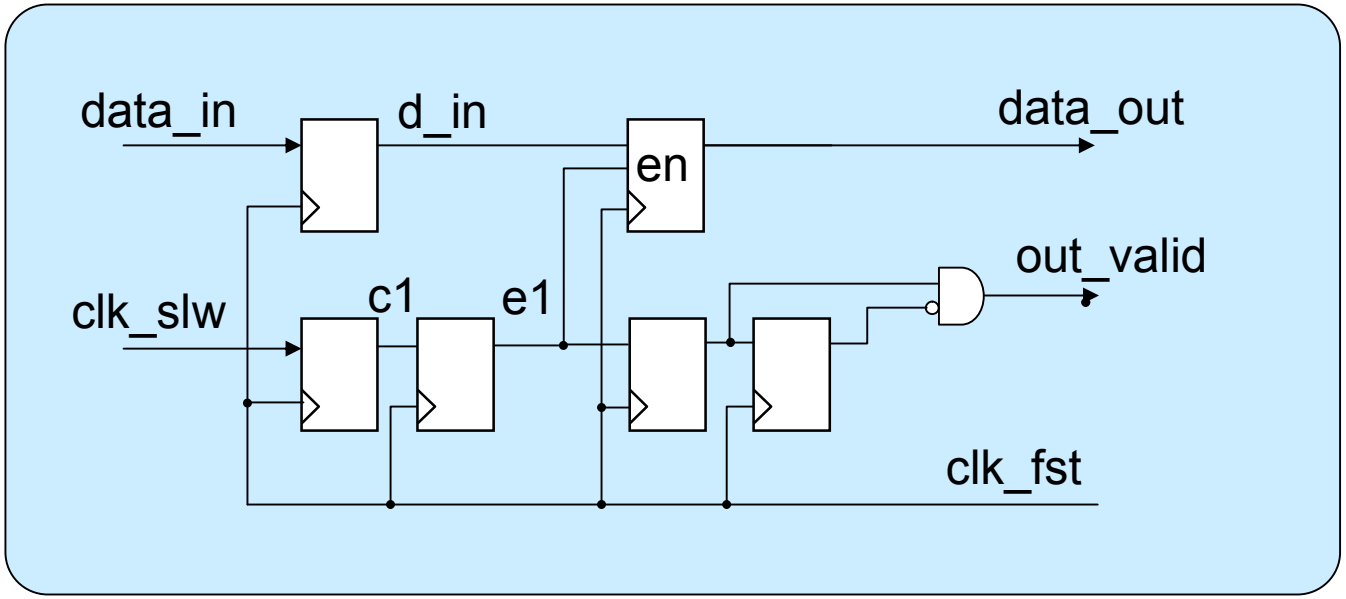
Total time chart



From the time chart on the previous page, using 2 continuous 1 of c1 is known to be redundant if the duration of data_in is not stable is short enough compared to clk_fst cycle. In such case we can use e1 as enable signal as shown below.



Such assumption gives us the following logic diagram.



A sample target code on the next pages uses 2 continuous 1 of e1.

No test bench is shown for this exercise.

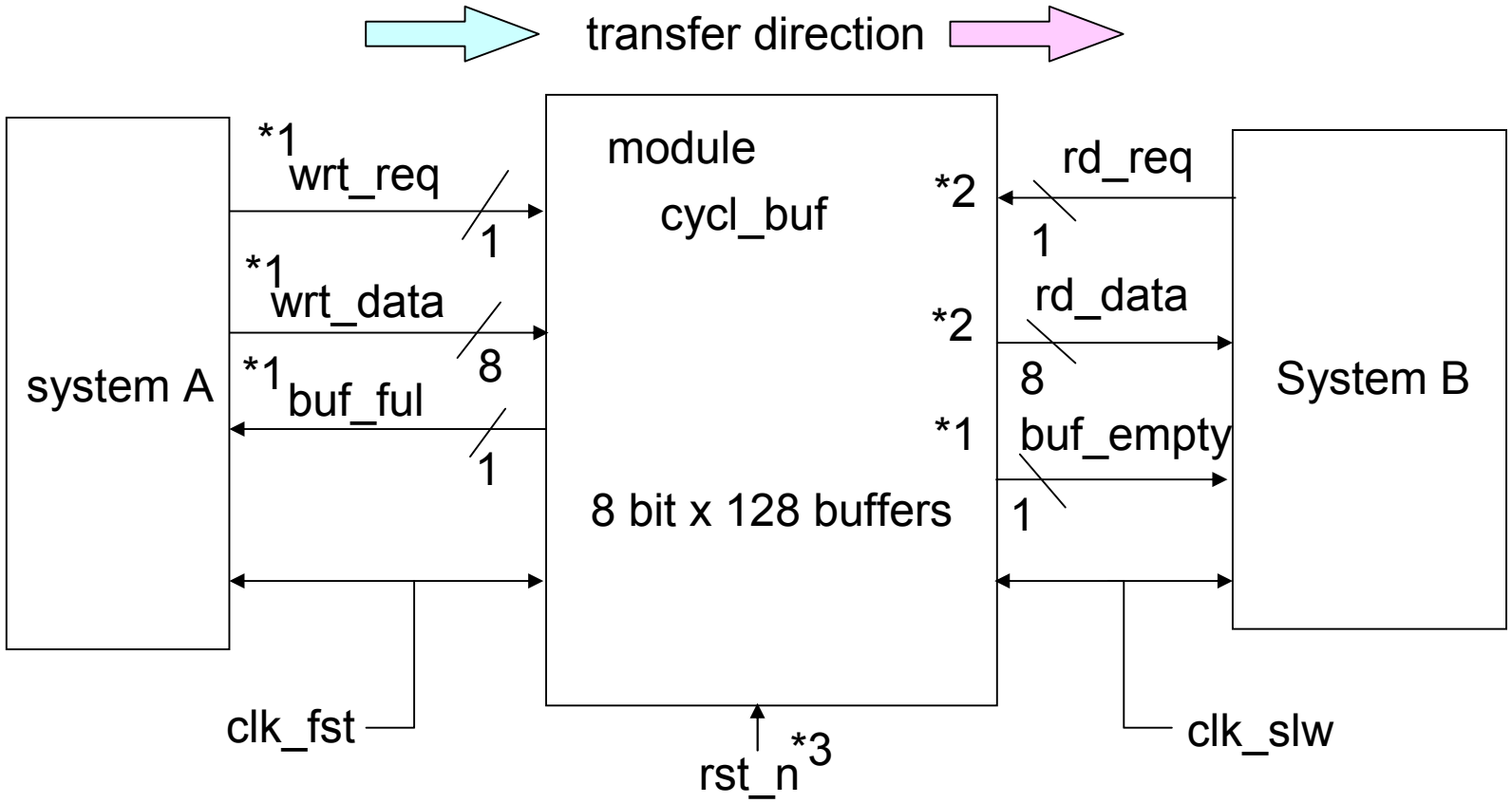
A sample target code

```
module async_trans ( clk_slw, clk_fst, rst_n,
                     data_in,
                     data_out, out_valid
                    ) ;
//*****************************************************
//  RTL programming exercise training sample answer
//  (c) RVC, 2010
//*****************************************************
// note:  this logic is applicable for clk_fst is 6 time faster than clk_slw.
//
input clk_slw, clk_fst ;
input rst_n ;
input [7:0] data_in ;
output [7:0] data_out ;
output out_valid ;
wire clk_slw, clk_fst ;
wire rst_n ;
wire [7:0] data_in ;
reg [7:0] data_out ;
wire out_valid ;
//internal valiables
// internal reg
reg      fst_clk_smpl, scnd_clk_smpl ; // FF
reg      smpl_d_1, smpl_d_2 ;  // FF
reg [7:0] data_reg_fst ; // FF
reg      reg_for_valid, secnd_reg_for_valid ; // FF
// wires
wire en_reg_set ;
```

```
//**************  logic start  *************
// create enable and out_valid signal
//
assign en_reg_set = smpl_d_1 & smpl_d_2 ;
assign out_valid = reg_for_valid & ( ~secnd_reg_for_valid ) ;

//**********  FF  *************
//
always @ ( posedge clk_fst or negedge rst_n ) begin
  if ( rst_n == 1'b0 )  begin
    fst_clk_smpl  <= 1'b0 ;
    scnd_clk_smpl <= 1'b0 ;
    smpl_d_1      <= 1'b0 ;
    smpl_d_2      <= 1'b0 ;
  end
  else begin
    fst_clk_smpl  <= clk_slw ;
    scnd_clk_smpl <= fst_clk_smpl ;
    smpl_d_1      <= scnd_clk_smpl ;
    smpl_d_2      <= smpl_d_1 ;
  end
end
```

```
always @ ( posedge clk_fst or negedge rst_n ) begin
  if ( rst_n == 1'b0 )  begin
    data_reg_fst[7:0] <= 8'b0000_0000 ;
    data_out[7:0]     <= 8'b0000_0000 ;
  end
  else begin
    data_reg_fst[7:0] <= data_in[7:0] ;
    if ( en_reg_set == 1'b1 ) data_out[7:0] <= data_reg_fst[7:0] ;
  end
end
always @ ( posedge clk_fst or negedge rst_n ) begin
  if ( rst_n == 1'b0 )  begin
    reg_for_valid <= 1'b0 ;
    secnd_reg_for_valid <= 1'b0 ;
  end
  else begin
    reg_for_valid <= en_reg_set ;
    secnd_reg_for_valid <= reg_for_valid ;
  end
end
endmodule
```

Ex 6-6. Cyclic FIFO buffer : Design a module which passes data from fast running system A to slow running system B.



*1 : synchronous to clk_fst

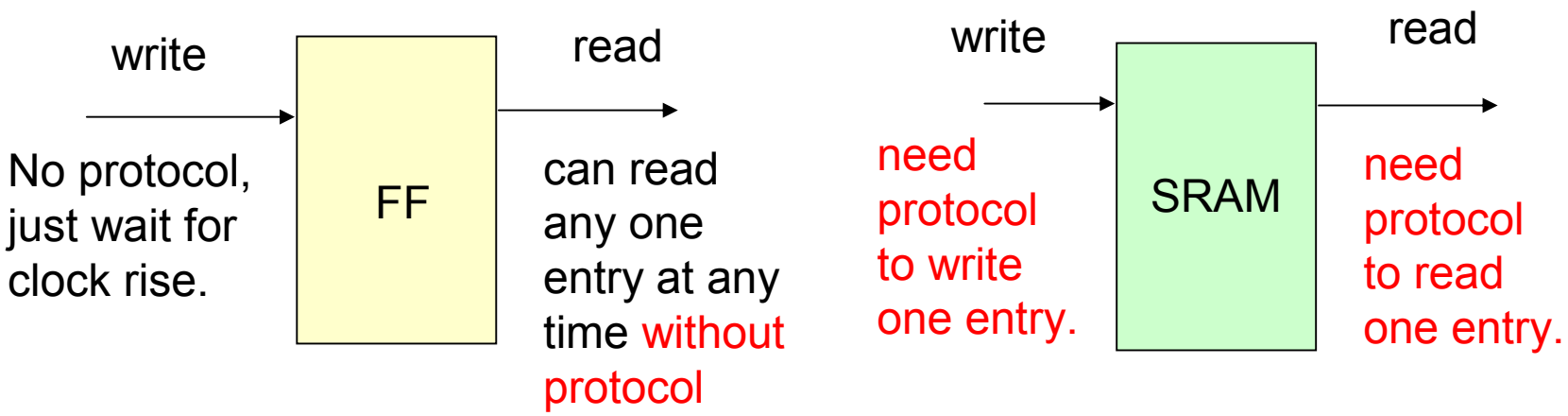
*2 : synchronous to clk_slw

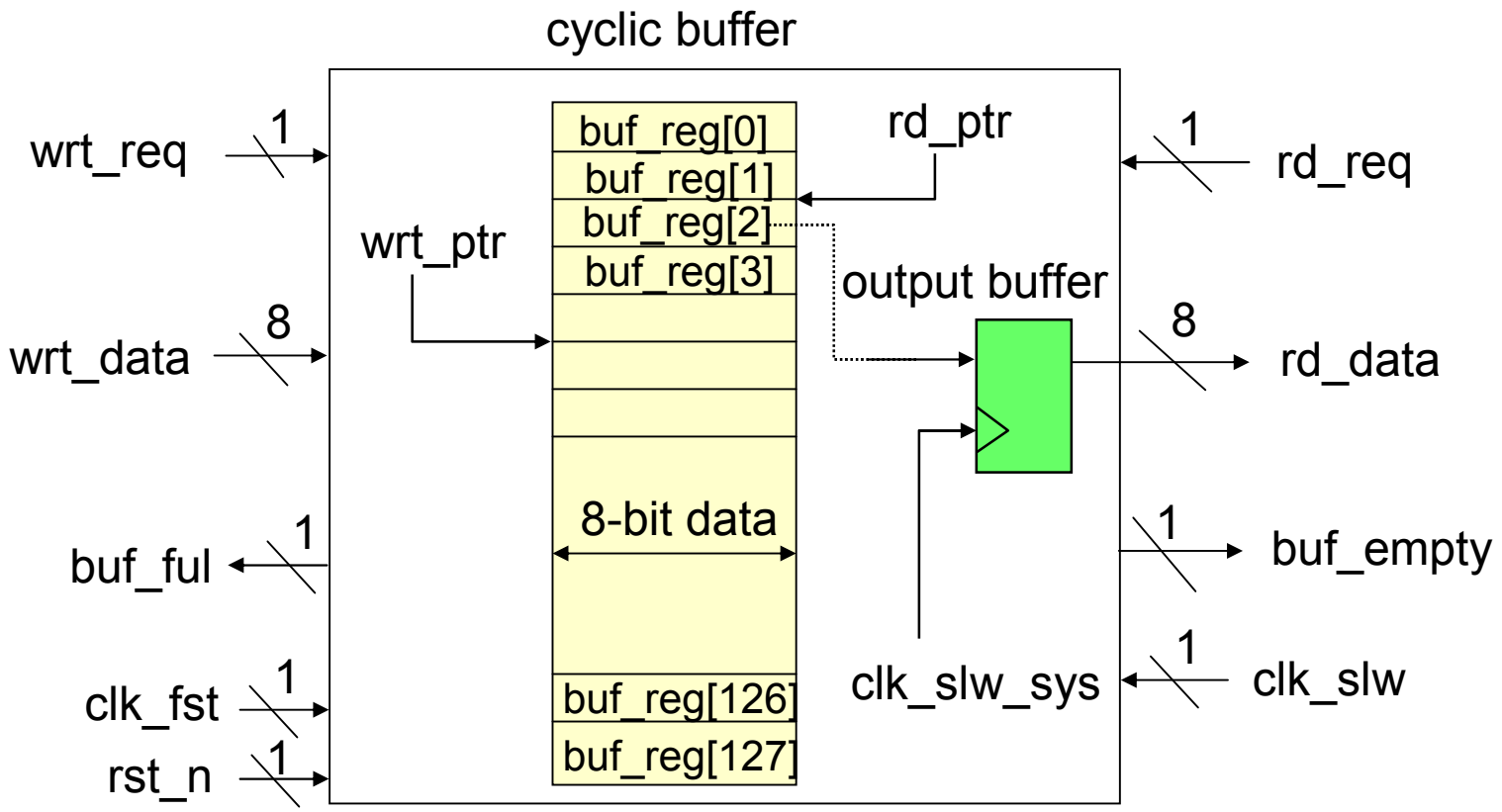
*3 : asynchronous reset

System A is running 4 times faster than System B and their clocks are synchronized to one another. That is, whenever clk_slw rises clk_fst rises at the same time.

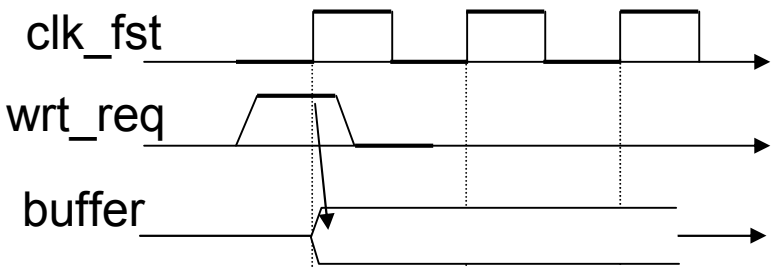
When buffer size is large, such as several k-byte or larger, we must use SRAM instead of FF. When using SRAM, major design issues are in how to write and read it from slow clock side and fast clock side.

In this RTL programming exercise, let's use FFs for buffer.

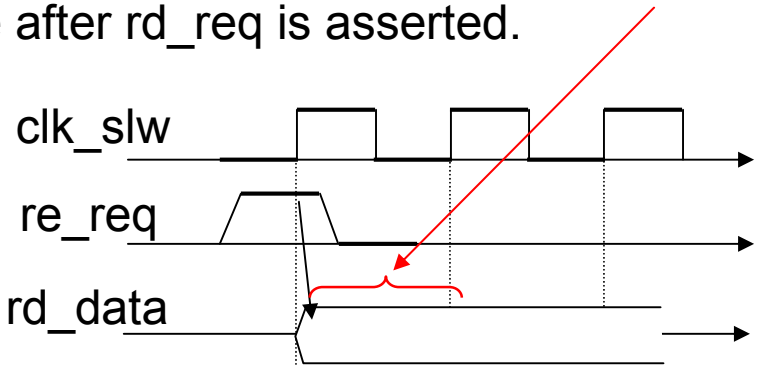




When the buffer is not full, wrt_data must go into the buffer at the rise edge of the clock if wrt_req is asserted.



When the buffer is not empty, buffer data must appear on rd_data in the next clock cycle after rd_req is asserted.



Work flow for solutions

(1) Investigate the function of the system.
(2) Find what is essential to the function,
(3) Write a design document including a state transition table,
(4) Write a RTL code for the target module,
(5) Run the automatic bench on the latter pages with your cycl_buf module to see if it is correct or not.

Use slow clock to update read pointer and output buffer.
Use fast clock to update write pointer and data buffer.
Because fast and slow clocks are synchronized, signals are stable at any rise edge of clocks except rd_req.
rd_req is stable only at rise edge of slow clock.

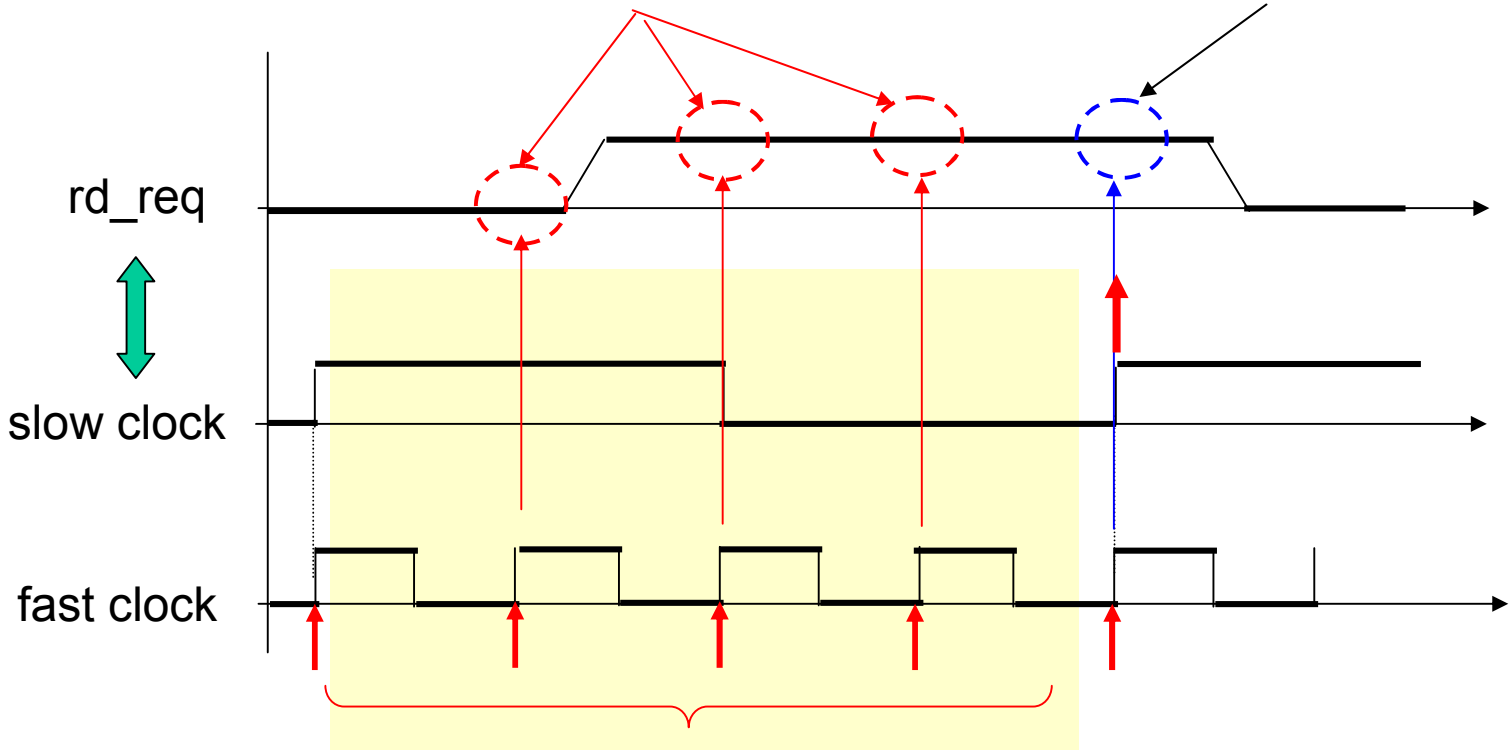
See the next pages for further understanding of the logic you have to implement.

You must use parameter BUF_LENG to make buffer size adjustable for 2**n (where n shall be 2, 3, 4, 5, ,,,, etc.).

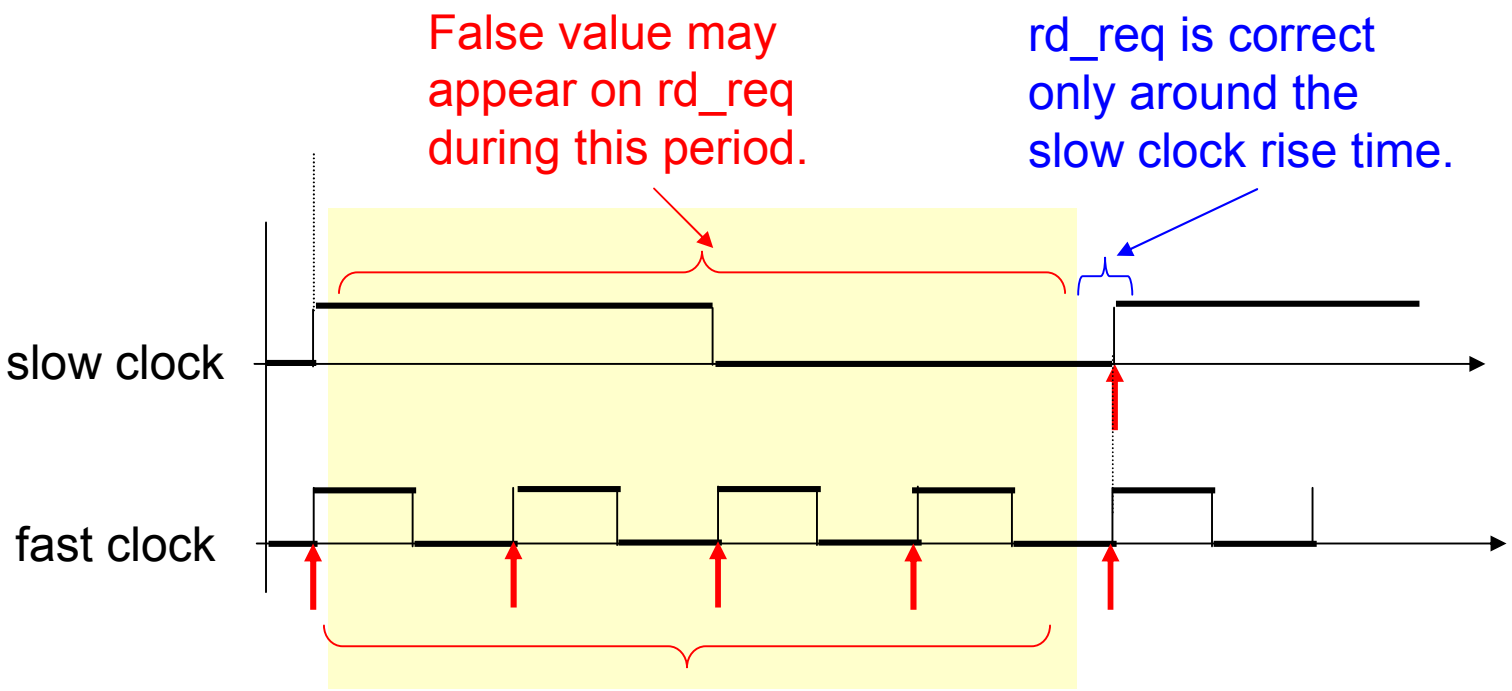
When looking for a solution, you must be careful that rd_req and next read pointer evaluated from current read pointer and rd_req may not be correct.

These rd_req do not update read pointer, because slow clock does not rise at these timing.

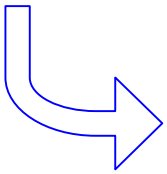
rd_req updates the read pointer correctly only at this timing.



During this time period, rd_req is invalid.



Therefore, for this time period, neither rd_req nor next read pointer calculated based on rd_req can be used in evaluating buffer state.

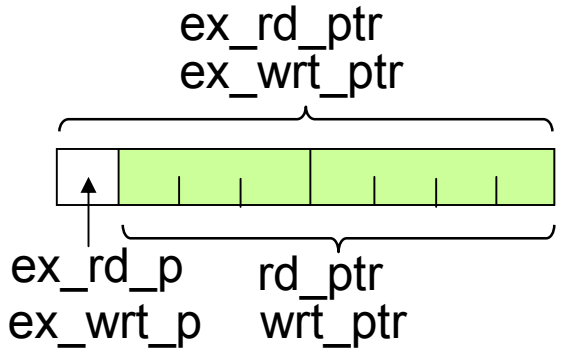


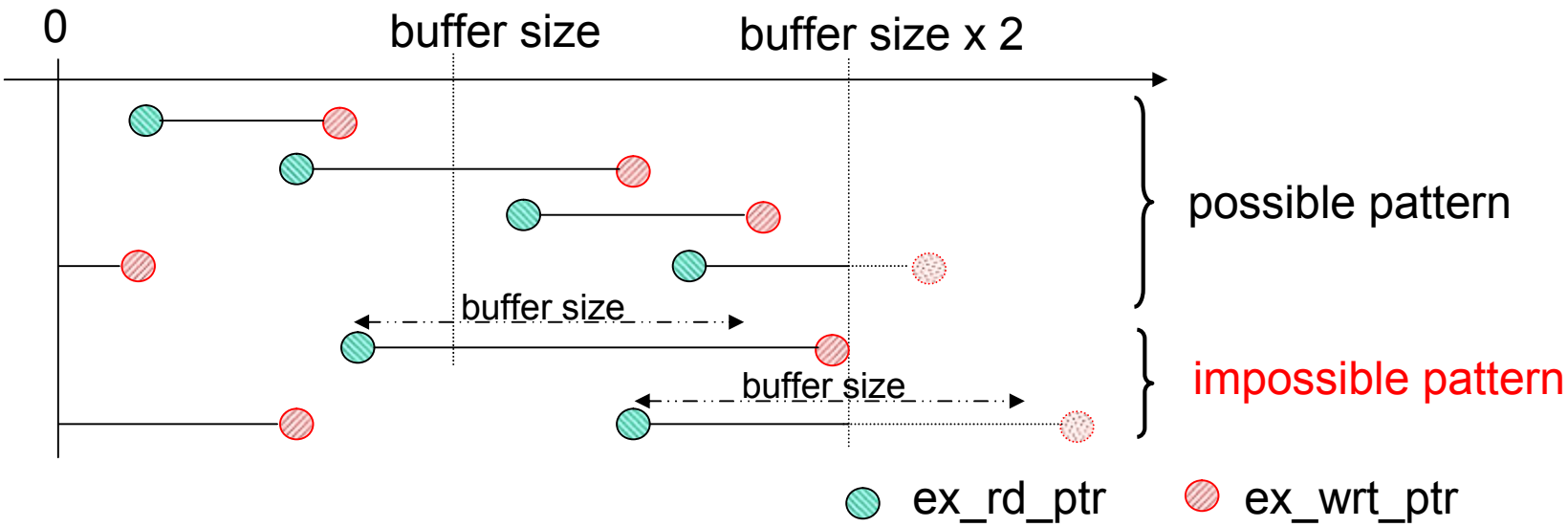
We can not use rd_req to evaluate buffer usage when fast clock rises but slow clock does not rise.
We have to use values updated by clock to evaluate buffer usage.

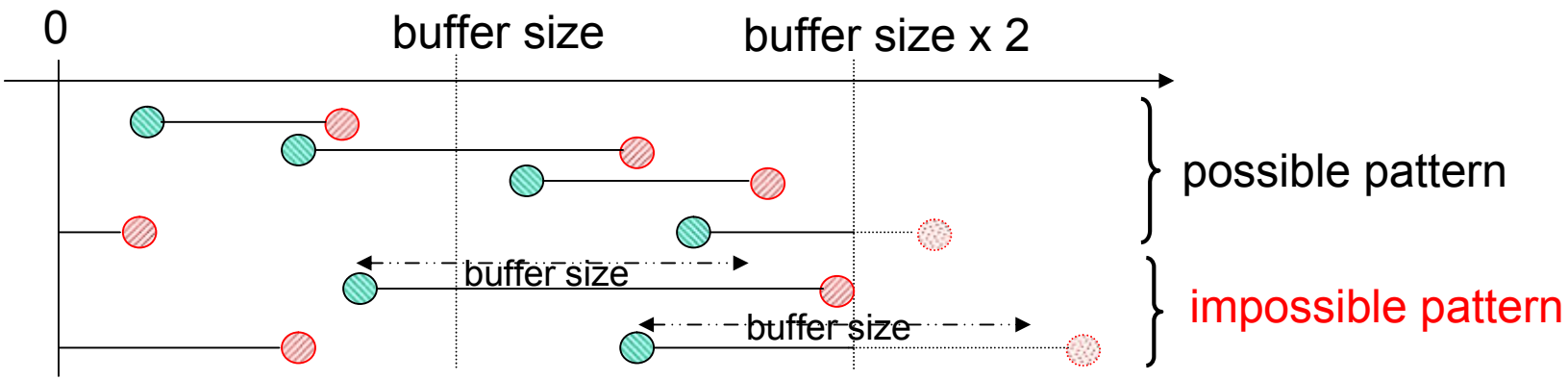
Introducing bit size extended pointer can be a solution for this problem.

If we use as many bits as necessary for the pointer to avoid its overflow ,
(1) rdd_ptr will never become larger than wrt_ptr, and
(2) wrt_ptr will never become larger than rd_ptr + buffer size.

Now append one bit to pointers as shown on the right and name the bit extended pointers ex_rd_ptr and ex_wrt_ptr.







ex_rd_p	ex_wrt_p	rd_ptr | wrt_ptr	buf_empty	buf_ful
0	0	rd_ptr < wrt_ptr	0	0
		rd_ptr = wrt_ptr	1	0
		rd_ptr > wrt_ptr		
	1	rd_ptr < wrt_ptr		
		rd_ptr = wrt_ptr	0	1
		rd_ptr > wrt_ptr	0	0
1	1	rd_ptr < wrt_ptr	0	0
		rd_ptr = wrt_ptr	1	0
		rd_ptr > wrt_ptr		
	0	rd_ptr < wrt_ptr		
		rd_ptr = wrt_ptr	0	1
		rd_ptr > wrt_ptr	0	0

buf_empty = 1 if rd_ptr=wrt_ptr and ex_rd_p=ex_wrt_p

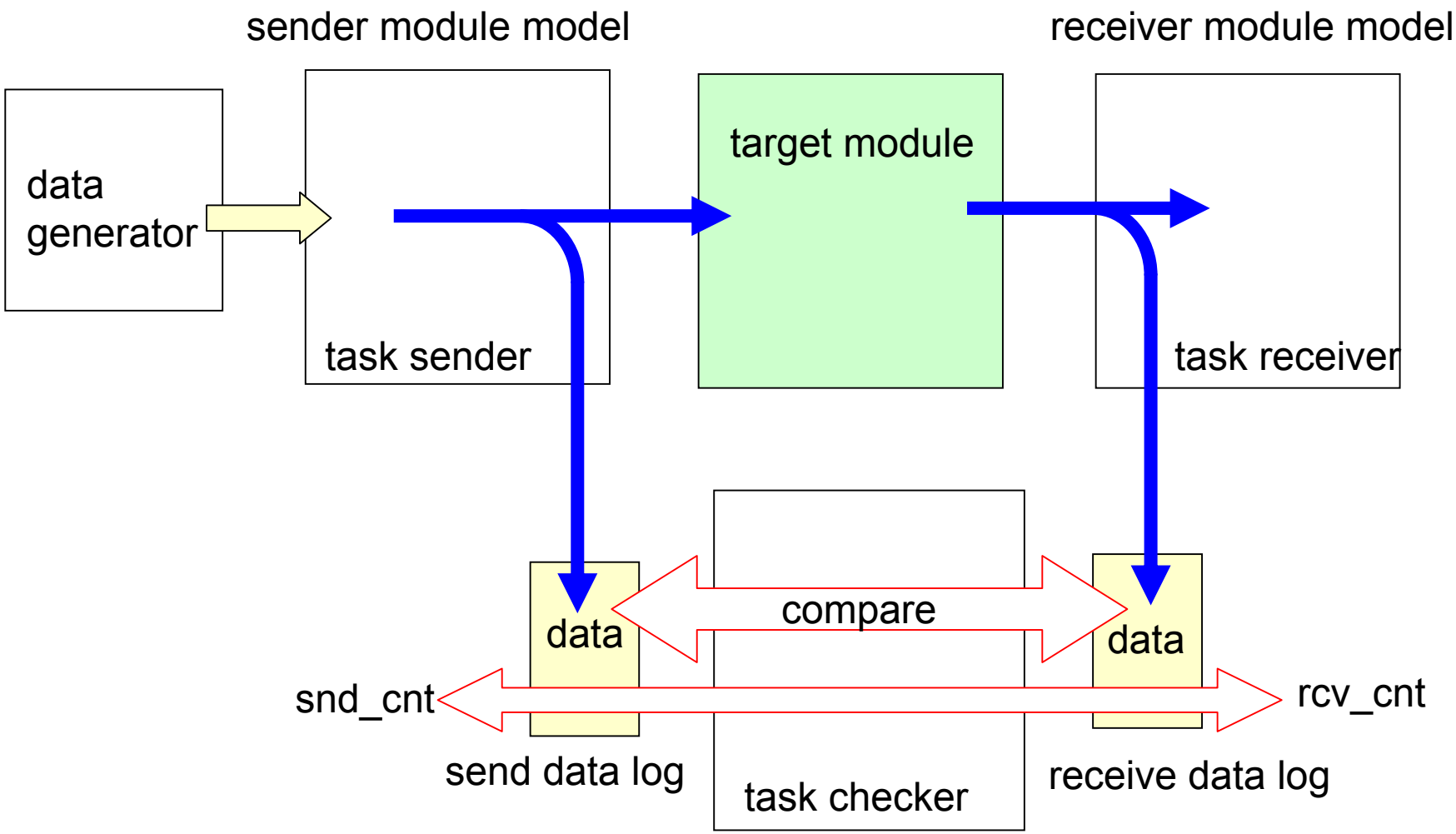
buf_ful = 1 if rd_ptr=wrt_ptr and ex_rd_p≠ex_wrt_p

We may get our solution by;

(1) Updating ex_rd_ptr at rise edge of slow clock,
(2) updating ex_wrt_ptr at rise edge of fast clock, and
(3) using them to evaluate buffer usage as shown on the previous page.

Now write some document yourself, and write a module cycl_buf based on your document.
And run it with the automatic test bench shown on the next pages.

A sample test bench, automatic tester



```
// automatic tester for cycl_buf
module test_cycl_buf ;
reg rst_n;
reg clk_fst, clk_slw;
//
parameter BUF_BW = 8 ;
parameter BUF_LENG = 8 ;
defparam cycl_buf_01.BUF_LENG = BUF_LENG ;
defparam cycl_buf_01.PTR_BW = 3 ;
//
reg [BUF_BW-1:0] data_source[0:255] ;
integer snd_cnt, rcv_cnt ;
reg [BUF_BW-1:0] snd_log[0:1023] ; // max data count is 1024
reg [BUF_BW-1:0] rcv_log[0:1023] ;
reg [BUF_BW-1:0] wrt_data ;
reg rd_req, wrt_req;
wire buf_ful, buf_empty ;
//
integer n, i ;
event snd_tming, rcv_tming ;
//
parameter SIM_MX=1000; // test cycles in slow clock
parameter FST_HF_CYCL =5;
parameter FST_CYCL = FST_HF_CYCL*2;
parameter SLW_HF_CYCL =FST_HF_CYCL*4;
parameter SLW_CYCL = SLW_HF_CYCL*2;
//
// module connection
cycl_buf cycl_buf_01(.rst_n(rst_n), .clk_fst(clk_fst), .clk_slw(clk_slw),
                     .wrt_req(wrt_req),  .wrt_data(wrt_data),
                     .rd_req(rd_req),    .rd_data(rd_data),
                     .buf_ful(buf_ful), .buf_empty(buf_empty)     ) ;
```

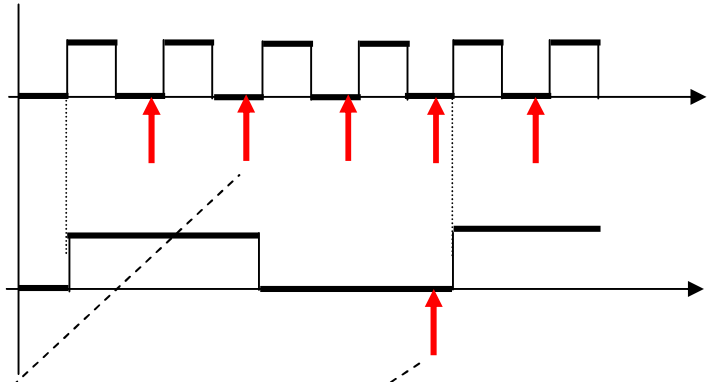
To make test time short, buffer length is changed to 8.

If your code does not parameterize pointer bit width, remove this line.

module connection

```
// clock generator
always begin // fast clock
 clk_fst=0; #FST_HF_CYCL;
 clk_fst=1; #FST_HF_CYCL;
end

initial begin // slow clock
 clk_slw=0; #FST_HF_CYCL;
 forever begin // synchronized to rise edge of fast clock
  clk_slw=1; #SLW_HF_CYCL;
  clk_slw=0; #SLW_HF_CYCL;
 end
end
//
// send and receive timing
always @ ( negedge clk_fst ) begin
  #(FST_HF_CYCL-1) -> snd_tming ;
end
always @ ( negedge clk_slw ) begin
  #(SLW_HF_CYCL-1) -> rcv_tming ;
end
//
initial begin
 rst_n = 0;
 #(SLW_CYCL*2) rst_n = 1;
 #(SLW_CYCL*SIM_MX+20) ;
 $display("Could not complete send or receive, error") ;
#1 $finish;
end
//
```

clock generator



buf_ful/sender and buf_empty/receiver evaluate timing.

reset and simulation stopper

```
initial begin // sender
 wait (rst_n ==1'b1 ) ;
 #(FST_CYCL*1);
 #1;
 sender(15);
 #(FST_CYCL*70);
 sender(4);
 #(FST_CYCL*60);
 sender(8);
 #(FST_CYCL*100);
end
//
```

data send sequence

```
initial begin // receiver
 wait (rst_n ==1'b1 ) ;
 #(SLW_CYCL*3);
 #1;
 receiver(7);
 #(SLW_CYCL*3);
 receiver(7);
 #(SLW_CYCL*5);
 receiver(13) ;
 #(SLW_CYCL*3);
```

data receive sequence

```
 $display ("ful=%b,emp=%b,r_p=%h, w_p=%h, buf = ¥n  %d %d %d %d %d %d %d %d ",
        buf_ful, buf_empty,
        cycl_buf_01.rd_ptr, cycl_buf_01.wrt_ptr,
        cycl_buf_01.buf_reg[0], cycl_buf_01.buf_reg[1], cycl_buf_01.buf_reg[2], cycl_buf_01.buf_reg[3],
        cycl_buf_01.buf_reg[4], cycl_buf_01.buf_reg[5], cycl_buf_01.buf_reg[6], cycl_buf_01.buf_reg[7]
        ) ;
 #1 ;
 checker ;
end
```

check result

```
initial begin
  snd_cnt =0;
  rcv_cnt=0;
  wrt_req=0;
  rd_req=0;
end
//
task sender;
input [8:0] cnt ; // send data count
integer k ;
begin
  k=0;
  //wrt_req=0;
  while ( k < cnt ) begin
    @( snd_tming ) begin
      if ( buf_ful == 1'b0 ) begin
        k=k+1;
        wrt_req = 1'b1;
        wrt_data = data_source[snd_cnt] ;
        snd_log[snd_cnt] = wrt_data;
        snd_cnt=snd_cnt+1;
        #1 $strobe("t=%d, wrt_req=%b, snd_cnt=%d, data=%d",
             $stime, wrt_req, snd_cnt, wrt_data ) ;
      end
    end
  end
  #2 wrt_req =0;  // rest wrt_req after clock rise
end
endtask
```

set initial values for counter and request

sender task :
send requested number of data into buffer,
loop until all the data sent into buffer.

```
task receiver;
input [8:0] cnt ;  // receive data count
integer j ;
begin
  j=0;
  rd_req=0;
  while ( j < cnt ) begin
    @( rcv_tming ) begin
      if ( buf_empty == 1'b0 ) begin
        j=j+1;
        rd_req =1 ;
        @( posedge clk_slw ) begin
          #1 rcv_log[rcv_cnt] = rd_data;
          rcv_cnt=rcv_cnt+1;
          $strobe("t=%d, rd_req=%b, rcv_cnt=%d, data=%d",
             $stime, rd_req, rcv_cnt, rd_data ) ;
        end
      end
   end
  end
  #2 rd_req =0;  // rest rd_req after clock rise
end
endtask
//
```

receiver task:
receive requested number of data from buffer,
loop until all the data come out of buffer.

```
//
task checker; // compare send and receive data
integer m ;
begin
  if ( snd_cnt !== rcv_cnt ) begin
   $display("send and receive data count miss-match, snd=%d, rcv=%d",
            snd_cnt, rcv_cnt );
   #1 $finish;
  end
  else begin
    for ( m=0; m <= snd_cnt; m=m+1 ) begin
      if ( rcv_log[m] !== snd_log[m] ) begin
        $display("data miss-match at %d -th data, snd=%d, rcv=%d",
             m+1, snd_log[m], rcv_log[m] );
        #1 $finish ;
      end
    end
    #1;
    $display("%0d data sent ¥n simulation complete with no error",
             snd_cnt);
    #1 $finish ;  end
end
endtask
```

checker task: compare sent and received data count, and sent and received data

```
// sending data creation
integer kk ;
initial begin
 for( kk = 0; kk <=255 ; kk = kk+1 ) begin
   data_source[kk] = kk ;
 end
end
```

Create data to send and receive.
You may replace
data_source[kk] = kk ;
by
data_source[kk] = $random;.

```
always @ ( posedge clk_fst ) begin
 $strobe("t=%d, r=%b, fst=%b, slw=%b, f=%b, e=%b, w_p=%d, r_p=%d, w_req=%b, r_req=%b",
      $stime, rst_n, clk_fst, clk_slw, buf_ful, buf_empty,
      cycl_buf_01.wrt_ptr, cycl_buf_01.rd_ptr,
      wrt_req, rd_req );
end
//
//

//
endmodule
//
`include "cycl_buf.v"
```

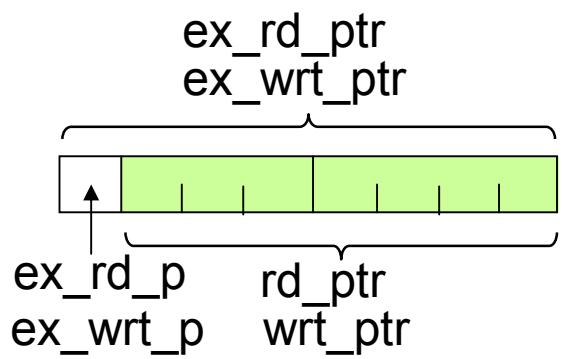
print state, request, and pointers

file name: test_cycl_buf.v

This tester does not check the timing automatically, you have to check with your eyes about the timing.
if you are interested, improve the module so that it can check the timing automatically.

A sample target code

```
module cycl_buf ( rst_n, clk_fst, clk_slw,  wrt_req,  wrt_data, rd_req,
                  rd_data,  buf_ful, buf_empty ) ;
parameter BUF_BW = 8 ;
parameter BUF_LENG = 128 ;
parameter PTR_BW = 7 ;
//
input rst_n, clk_fst, clk_slw ;
input wrt_req;
input [BUF_BW-1:0] wrt_data;
input rd_req;
output [BUF_BW-1:0] rd_data;
output buf_ful, buf_empty;
//
wire rst_n, clk_fst, clk_slw ;
wire wrt_req;
wire [BUF_BW-1:0] wrt_data;
wire rd_req;
reg [BUF_BW-1:0] rd_data; // FF
wire buf_ful;
wire buf_empty;
//
wire [PTR_BW-1:0] wrt_ptr;
wire [PTR_BW-1:0] rd_ptr;
wire ex_wrt_p, ex_rd_p ;
reg [PTR_BW:0] ex_wrt_ptr, ex_rd_ptr ;
reg [BUF_BW - 1:0] buf_reg [0:BUF_LENG-1 ] ; // buffer array//
//
assign { ex_wrt_p, wrt_ptr } = ex_wrt_ptr ;
assign { ex_rd_p,    rd_ptr } = ex_rd_ptr ;
//
```



```
//buffer write operation
always @ (posedge clk_fst or negedge rst_n) begin
        if ( rst_n == 1'b0 )  ex_wrt_ptr <=  0 ;
        else begin
            if (  ( wrt_req == 1'b1 ) & ( buf_ful == 1'b0 )  ) begin
                buf_reg[ wrt_ptr ] <= wrt_data ;
                ex_wrt_ptr <= ex_wrt_ptr + 1'b1;
            end
        end
end

//buffer read operation
always @ (posedge clk_slw or negedge rst_n) begin
        if ( rst_n == 1'b0 )  ex_rd_ptr <=  0 ;
        else begin
            if (  ( rd_req == 1'b1 ) & ( buf_empty == 1'b0 )  ) begin
          //      rd_data = buf_reg[ rd_ptr ] ;
                ex_rd_ptr <= ex_rd_ptr + 1'b1;
                rd_data <= buf_reg[ rd_ptr ] ;
            end
         end
end
//
assign buf_ful       = ( wrt_ptr == rd_ptr ) & ~( ex_wrt_p == ex_rd_p ) ;
assign buf_empty = ( wrt_ptr == rd_ptr ) &   ( ex_wrt_p == ex_rd_p ) ;
//
endmodule
```

file name: cycl_buf.v

Concluding remarks

Engineering has always several solutions.
A good engineer can find multiple solutions and choose the best one for the purpose of the design.

Do not be satisfied when you got one solution for an exercise. Always try to find better solutions.

End of the text material