



# **Modelsim**을 이용한 **Verilog** 실습

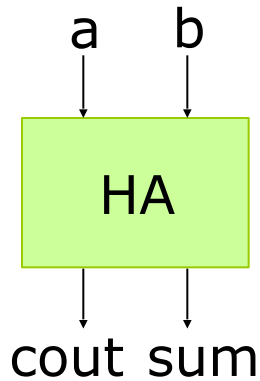
SoC Design Automation Lab.



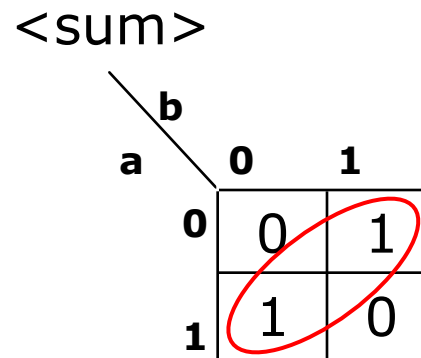
# Half Adder(1)

## ❖ Half adder cell (HA) revisited

- **Block Diagram**



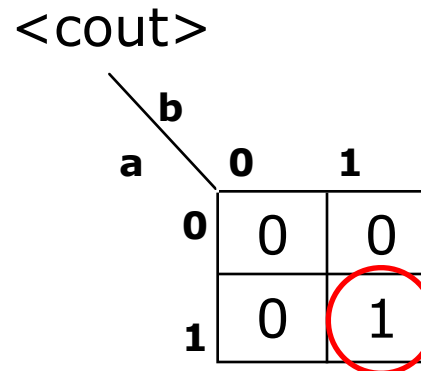
- **K-Map**



$$\text{sum} = a \wedge b;$$

- **Truth Table**

a	b	cout	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

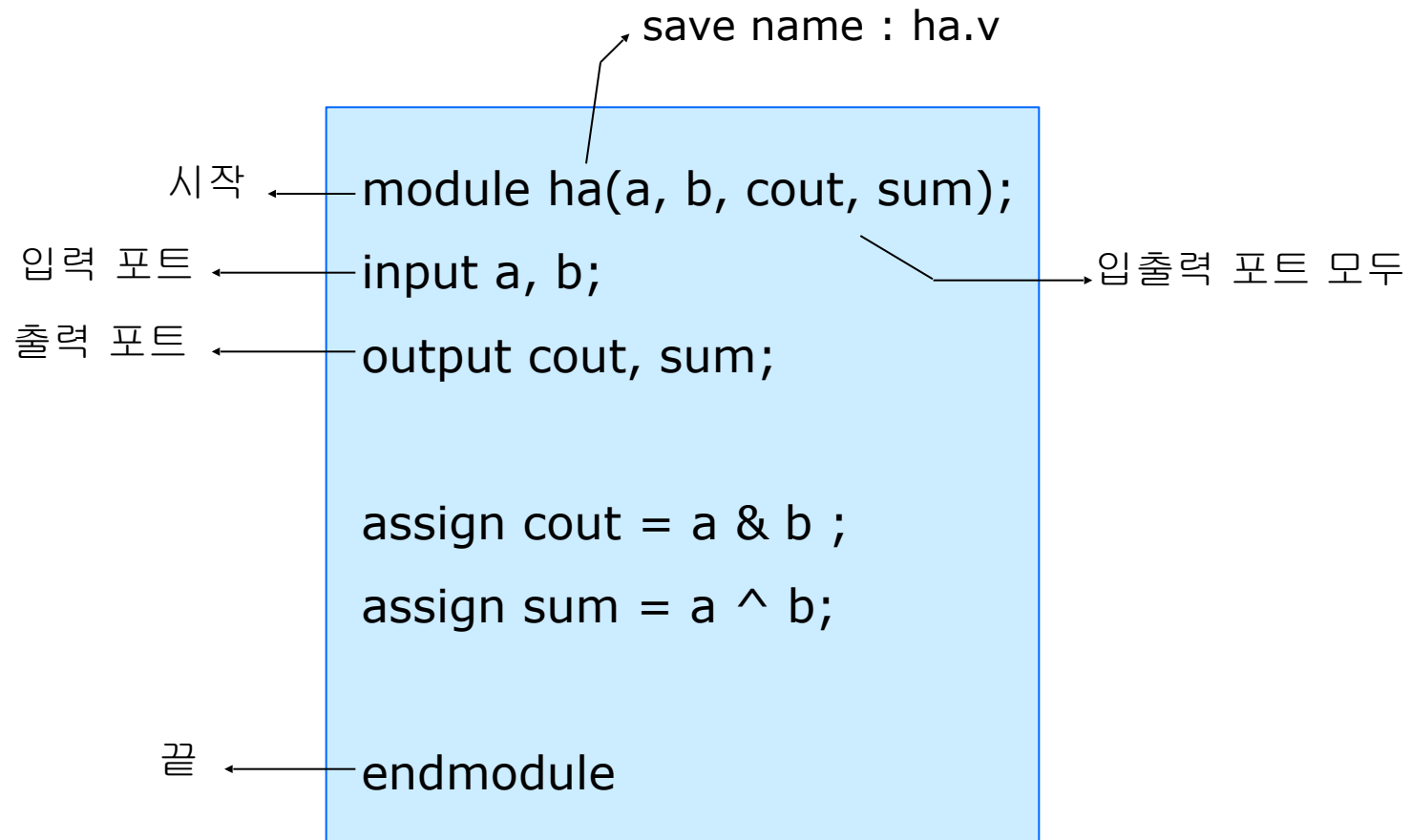


$$\text{cout} = a \& b;$$



# Half Adder(2)

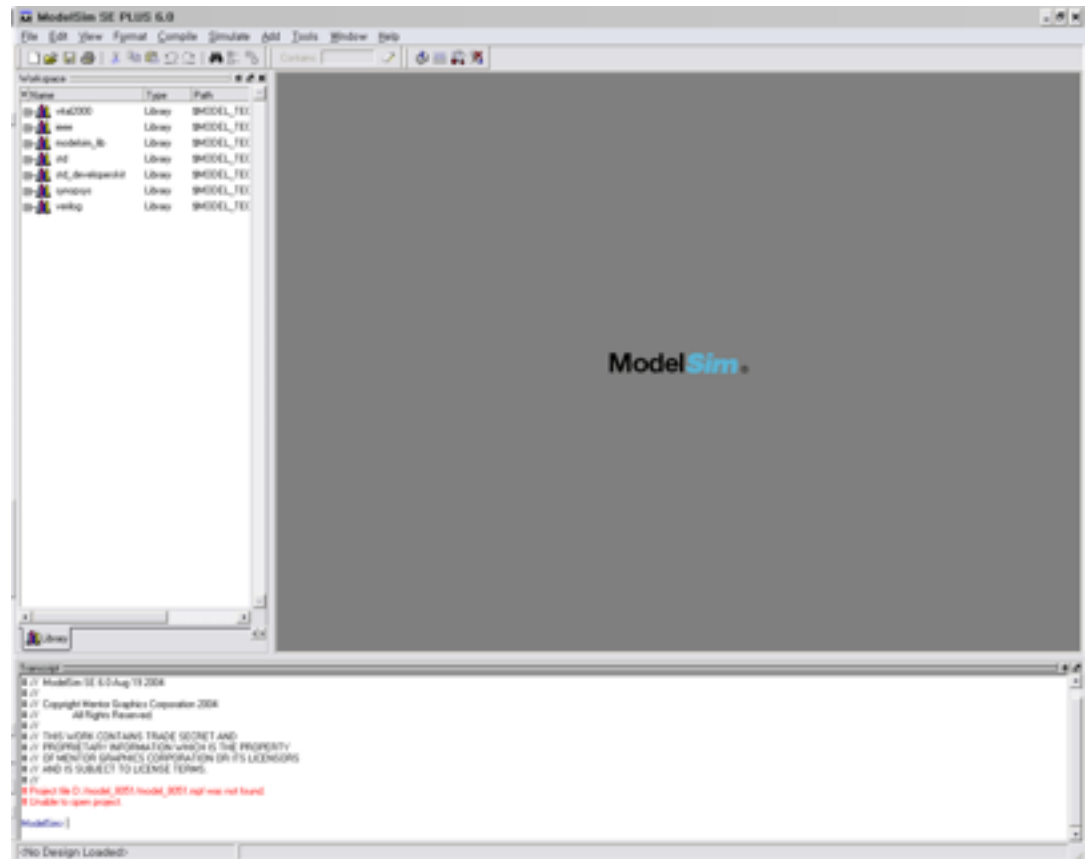
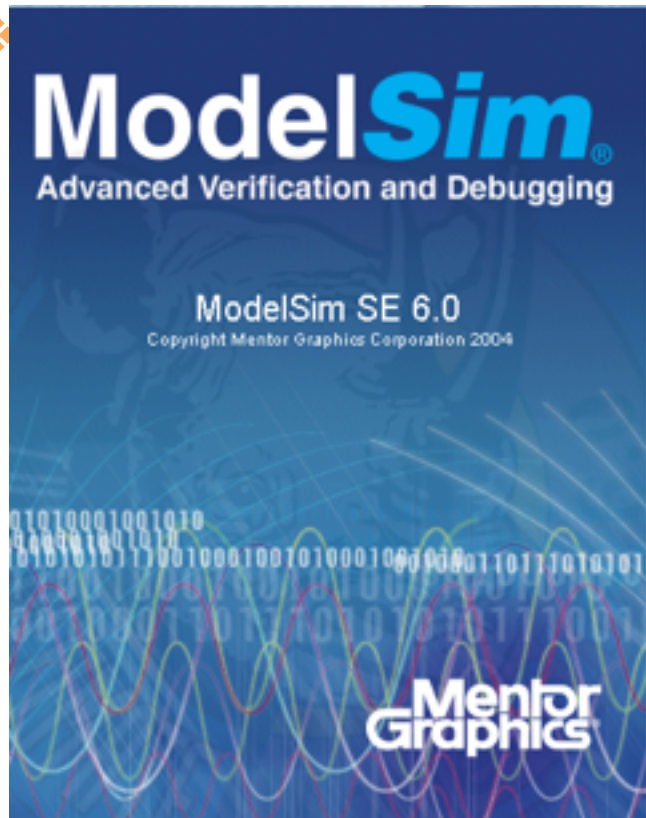
## ❖ Verilog coding





# Modelsim 시작하기

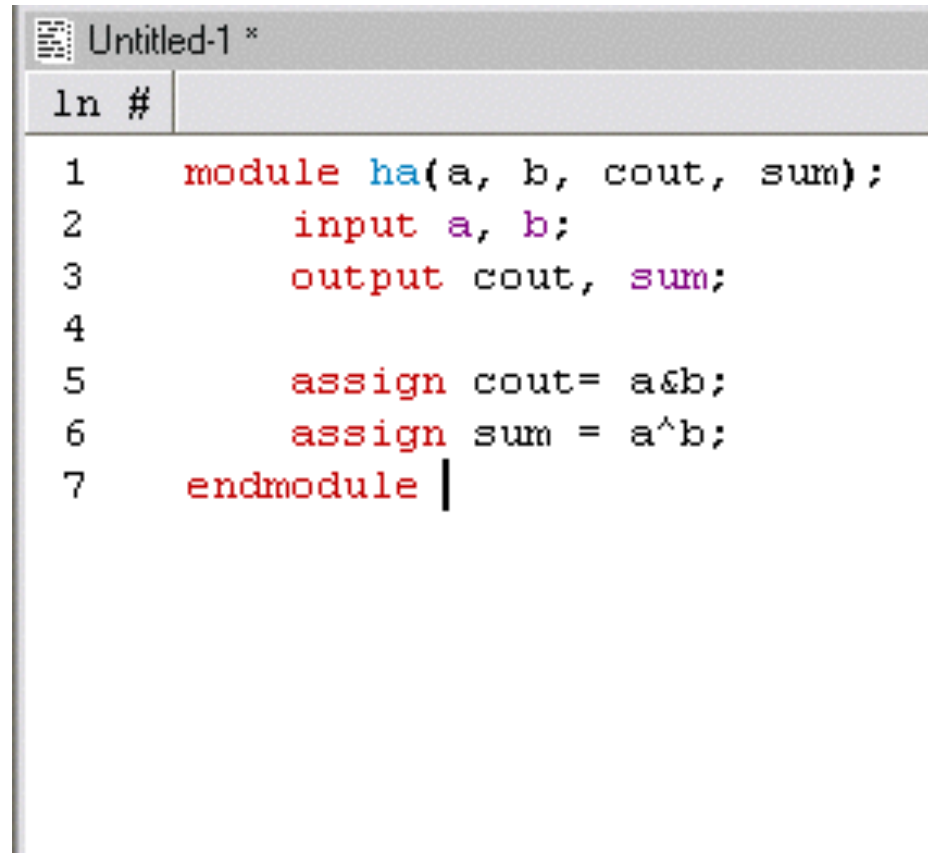
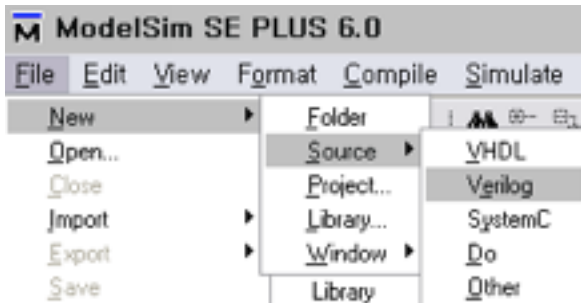
## ❖ 시작 – Modelsim SE – Modelsim





# Source Code 작성하기

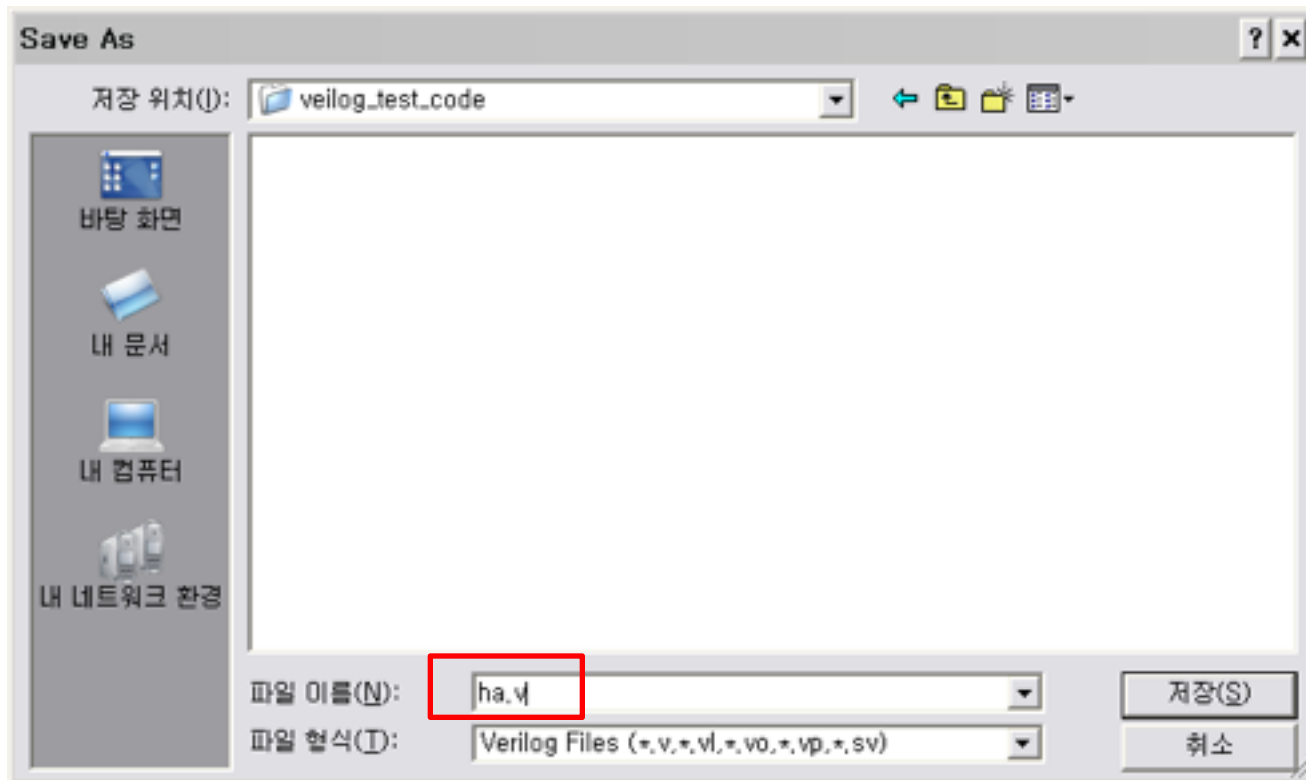
- ❖ 화면 상단의 주 메뉴에서 New-Source-Verilog를 선택하거나  아이콘을 선택하여 빈 창이 뜨면 code를 작성한다.





# Save

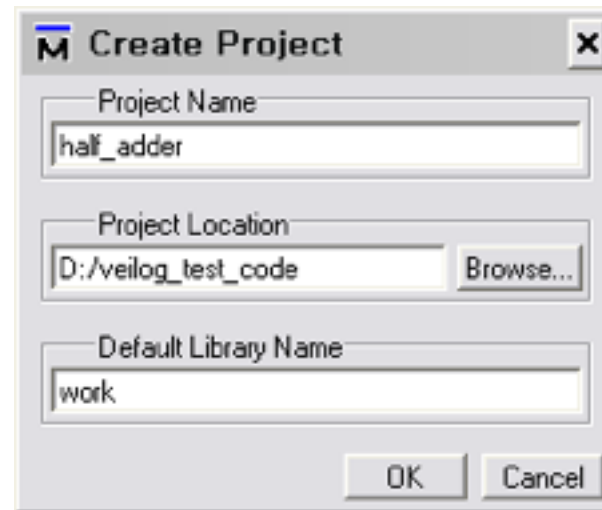
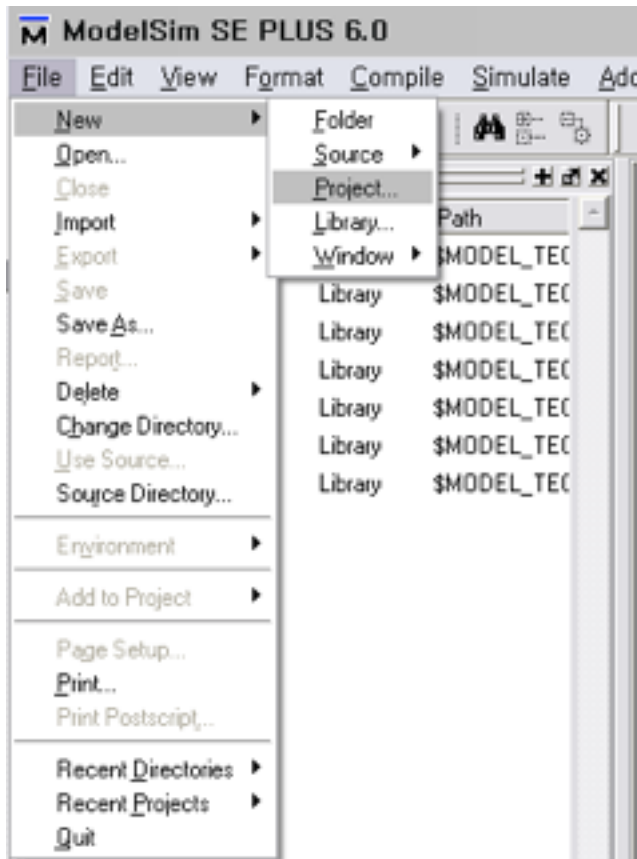
- ❖ 파일명과 module명은 같아야 한다.





# New Project File

❖ Project를 생성한다.

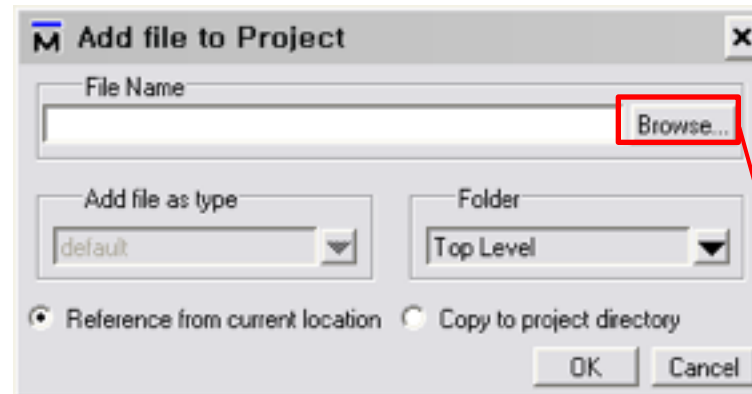
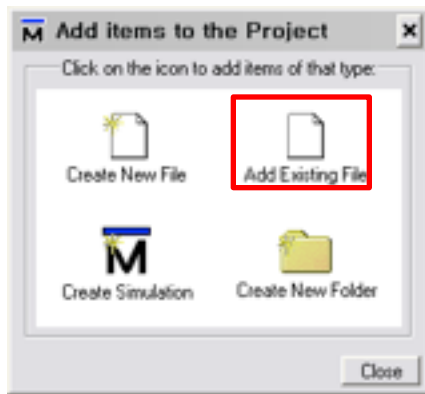


폴더 경로와 **Project Name** 을 지정한다.

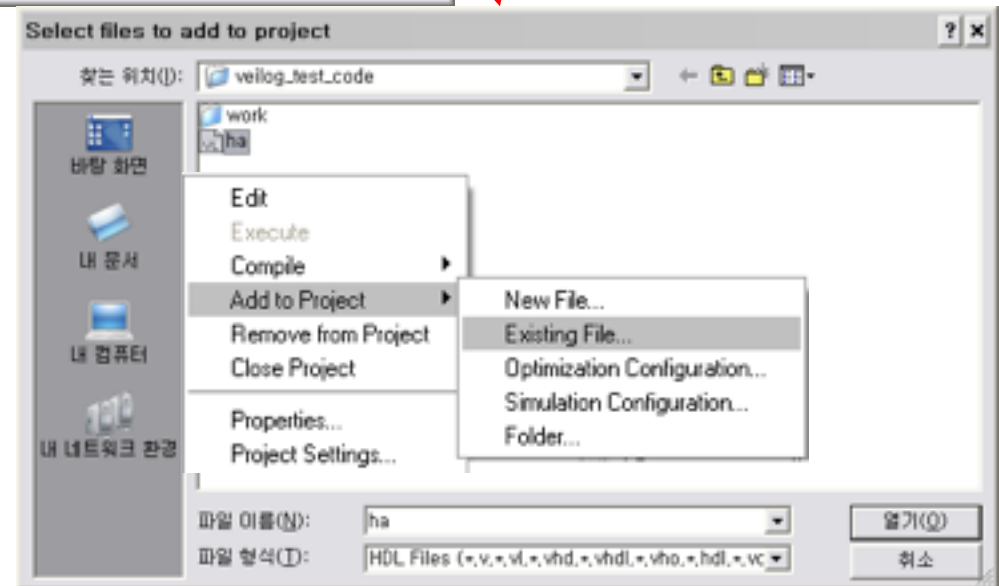


# Project에 파일 추가하기

## ❖ Source code 추가하기



**Verilog Code** 파일  
을 선택한다.



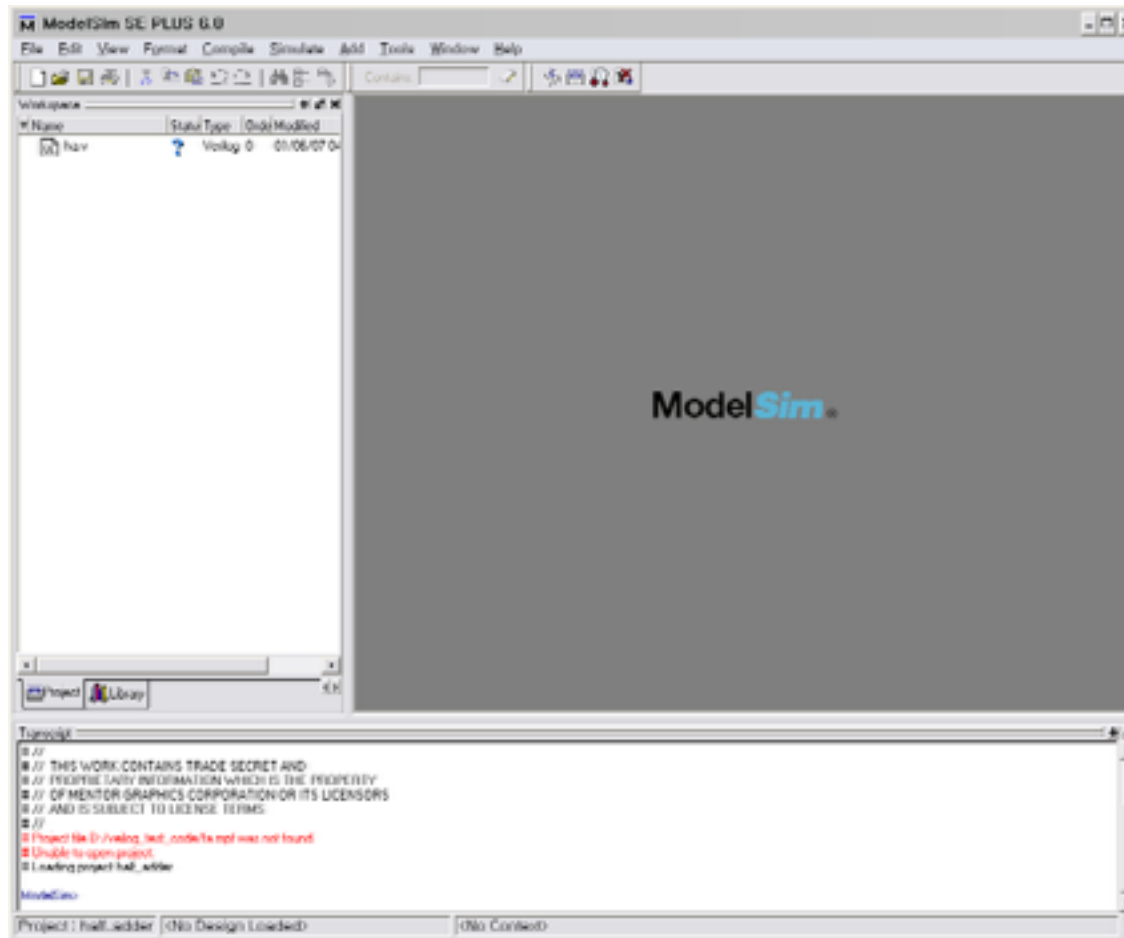
**Workspace**창에서 팝업창을 띄워서  
**Add to Project-Existing File...**  
을 선택하여 추가한다.






# Compile (1)

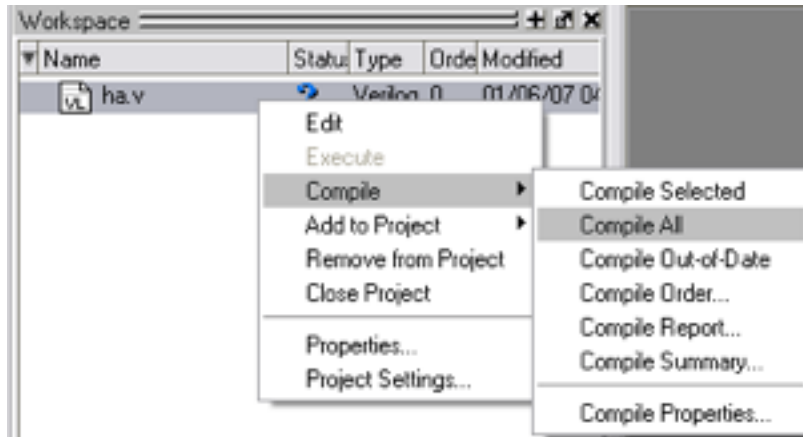
- ❖ Workspace에 파일이 추가되고 Compile이 되지 않았으므로 ? 표시가 뜬다.




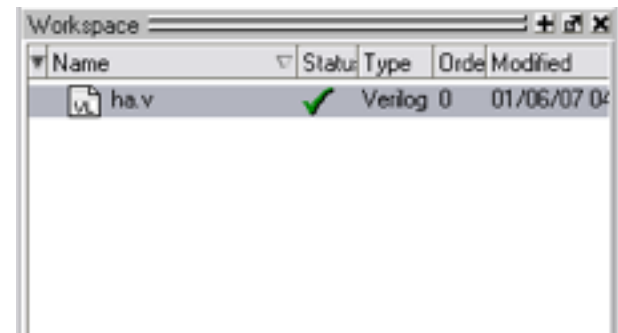



## Compile (2)

- ❖ 팝업창의 **Compile-Selected(or All)**을 누르거나  아이콘을 눌러 컴파일 한다.



컴파일이 완료되면 가 표시된다.



☞ 에러가 있으면 가 표시된다.

팝업창의 **Compile-Compile Report**를 눌러 에러내용을 확인할 수 있다.



# Simulation (1)

## ❖ Test bench 작성하기

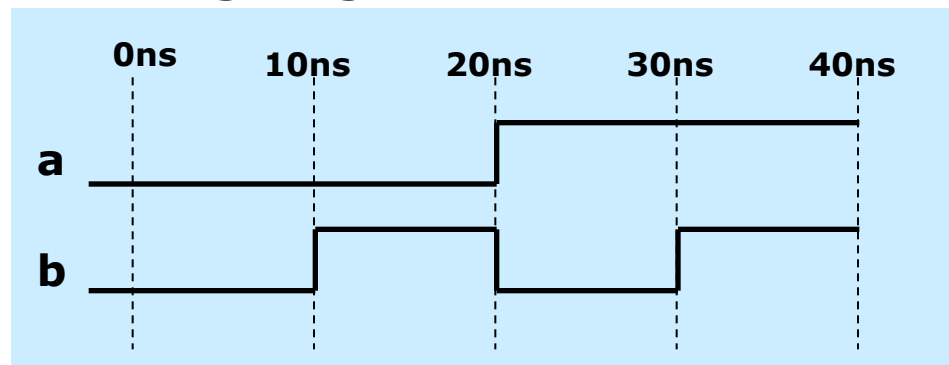
```
tb_hav
ln #
1  `timescale 1ns/10ps
2  module tb_ha;
3  reg a, b;
4  wire cout, sum;
5
6  ha u0(a, b, cout, sum);
7
8  initial
9  begin
10     a = 1'b0;
11     b = 1'b0;
12
13     #10 b = 1'b1;
14
15     #10 a = 1'b1;
16     b = 1'b0;
17
18     #10 b = 1'b1;
19
20     #10 $stop;
21 end
22 endmodule
```

**1ns(단위시간), 10ps(시뮬시간)**

**source code**와 구분할 수 있도록 **tb\_ha**로 저장

입력은 **reg**로 선언하고  
출력은 **wire**로 선언한다.  
**module**을 호출한다.

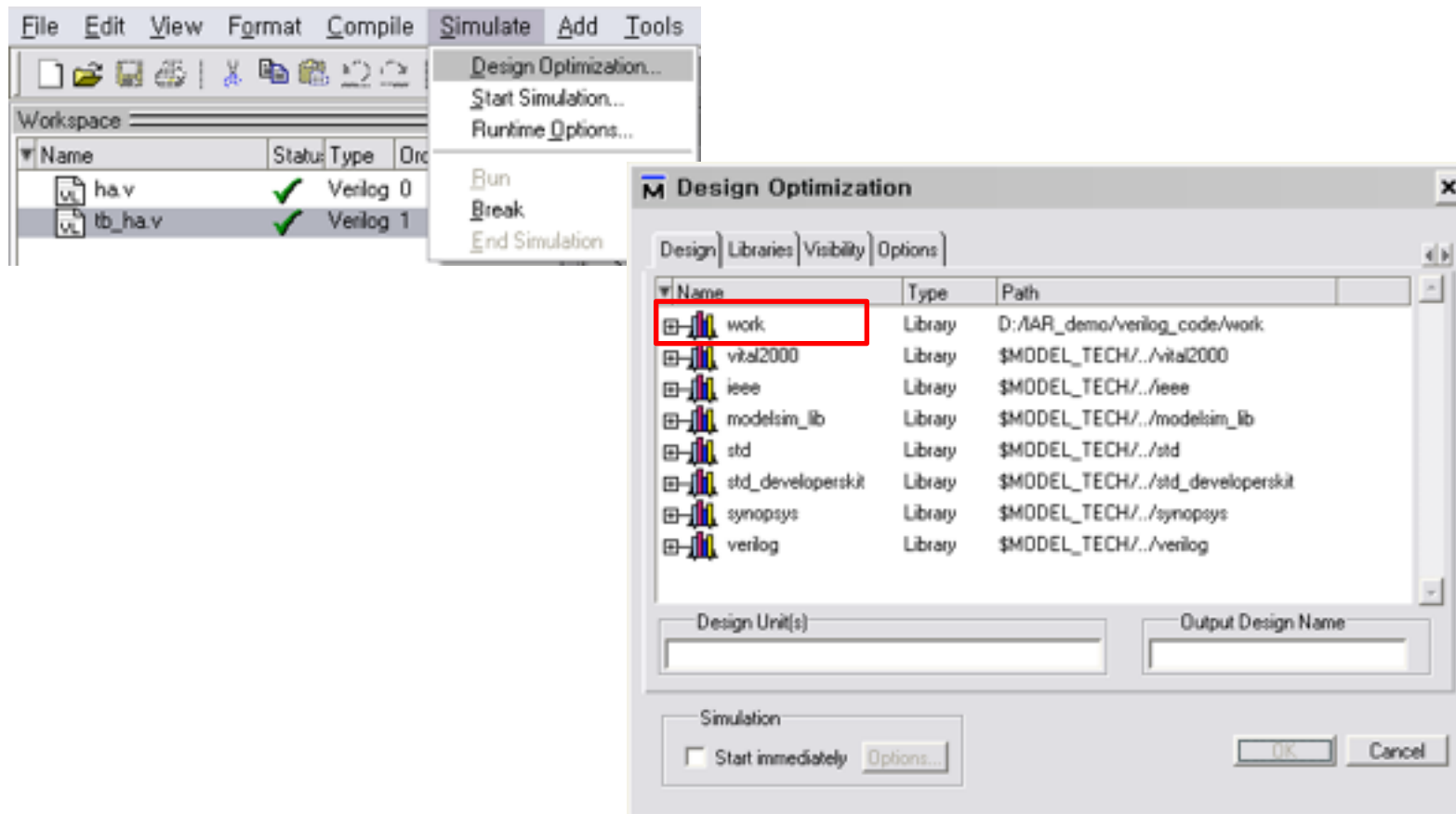
### • Timing Diagram





## Simulation (2)

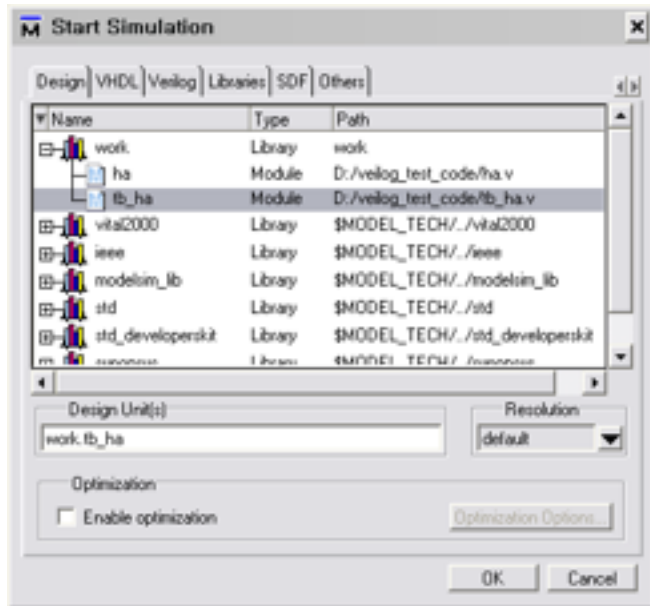
- ❖ 주 메뉴의 Simulate-Design Optimization을 누르거나  아이콘을 눌러 Design Optimization창을 띄운다.



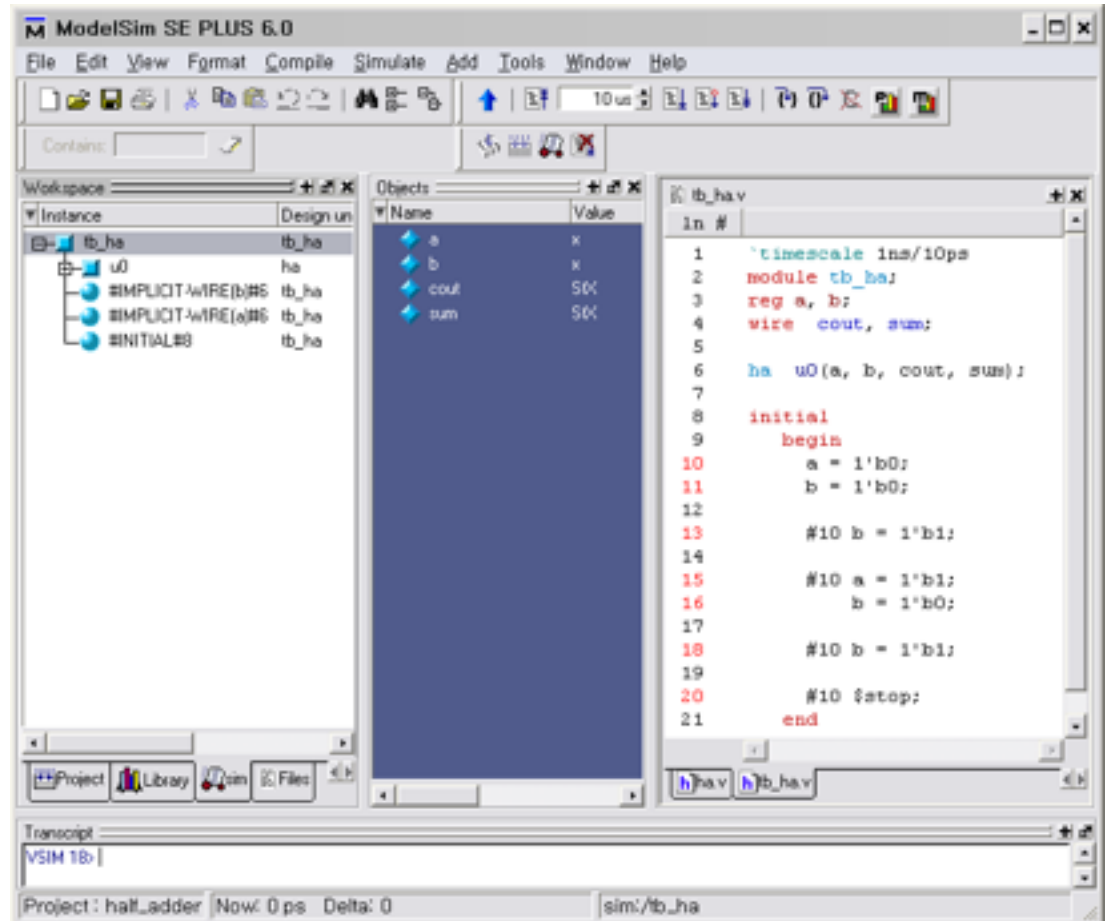


# Simulation (3)

work.tb\_ha를 선택한다.



새로운 창이 생긴다.



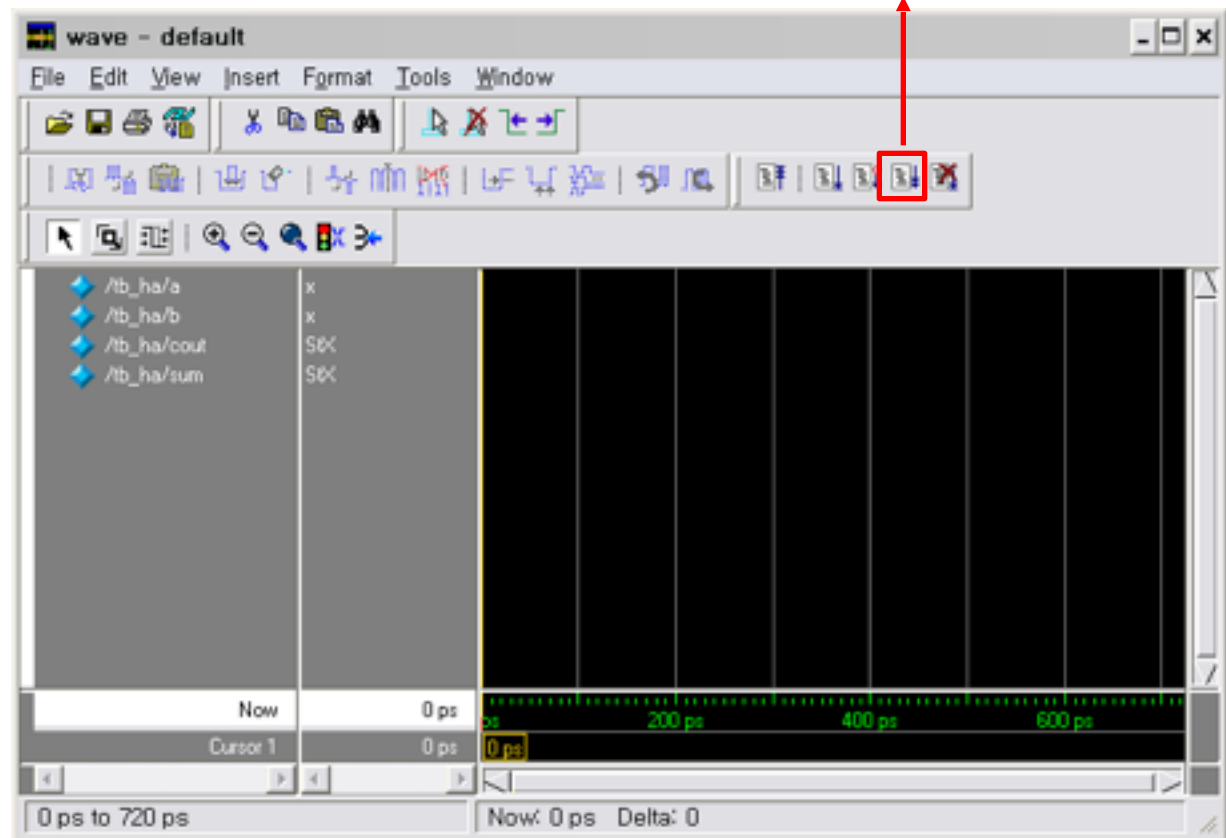


# Simulation (4)

- ❖ Object창에서 팝업창을 띄워 Signals in Region을 선택한다.



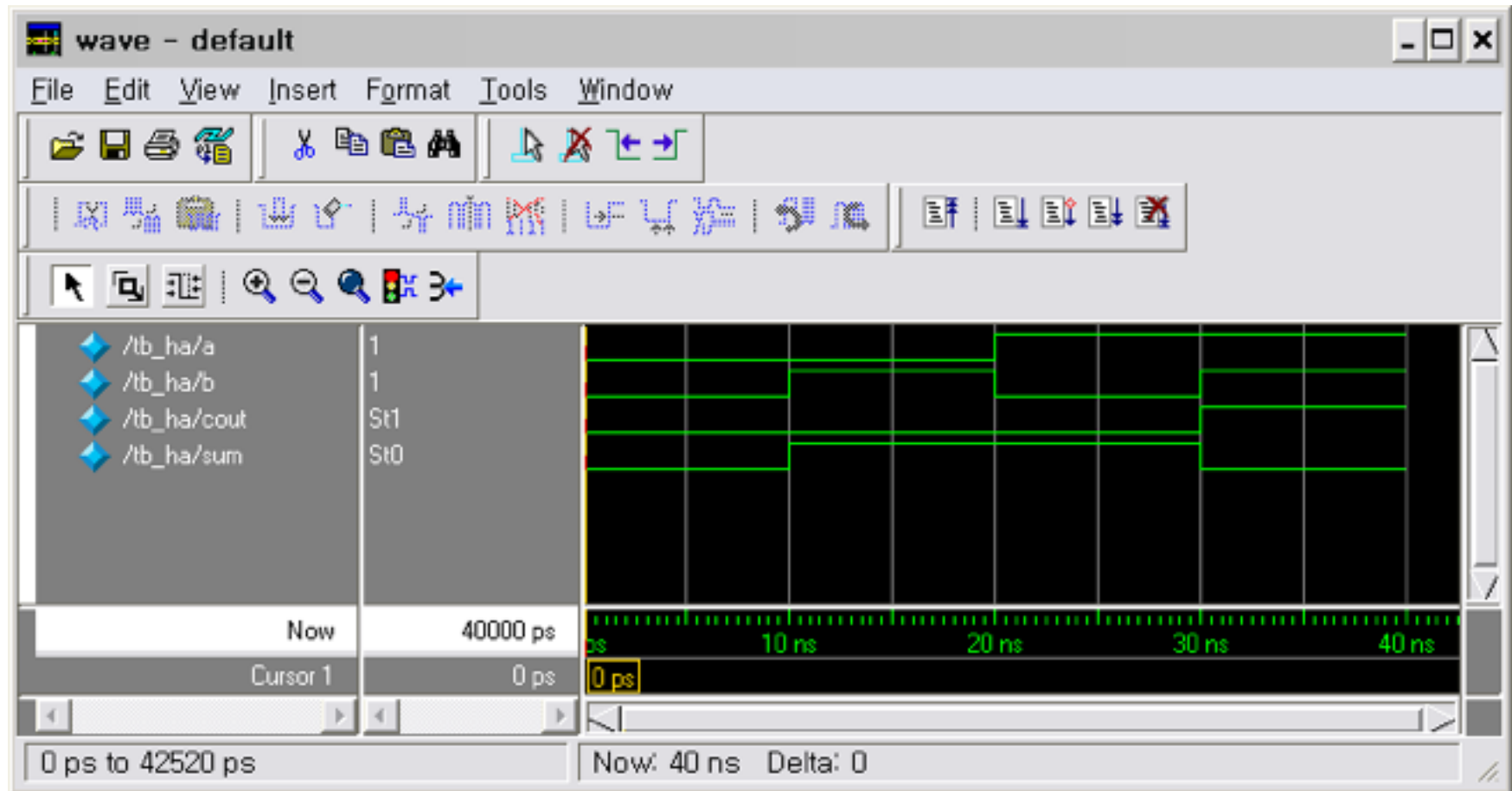
Wave창에서 Run -all을 선택





# Result

- ❖ 의도한 결과와 같은지 확인한다.





# 4-bit Adder

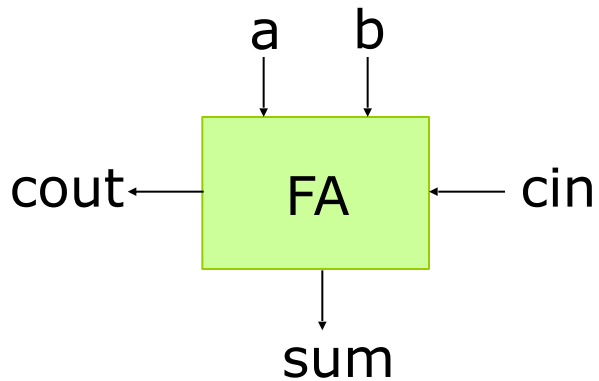




# Full Adder(1)

## ❖ Full adder cell (FA) revisited

### • Block Diagram



### • K-Map

ab	00	01	11	10
cin				
0			1	
1		1	1	1

$$\begin{aligned} \text{cout} &= a \& b \mid b \& \text{cin} \mid a \& \text{cin} \\ &= a \& b \mid \text{cin} \& (a \mid b); \end{aligned}$$

### • Truth Table

a	b	cin	cout	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

ab	00	01	11	10
cin				
0		1		1
1	1		1	

$$\text{sum} = a \wedge b \wedge \text{cin};$$



# Full Adder(2)

## ❖ Verilog code

```
module fa(a, b, cin, cout, sum);  
  input a, b, cin;  
  output cout, sum;  
  
  assign cout = a&b | a&cin | b&cin;  
  assign sum = a ^ b ^ cin;  
  
endmodule
```

Annotations for the Verilog code:

- 시작 (Start) → module fa(a, b, cin, cout, sum);
- 입력 포트 (Input Port) → input a, b, cin;
- 출력 포트 (Output Port) → output cout, sum;
- assign cout = a&b | a&cin | b&cin;
- assign sum = a ^ b ^ cin;
- 끝 (End) → endmodule

save name : fa.v

입출력 포트 모두 (All input/output ports)



The diagram shows a central blue rectangular block labeled **ripple\_adder\_4**. Two inputs, **a** and **b**, enter the top of the block from above. Each input line has a diagonal slash with the number **4** next to it, indicating a 4-bit bus. An output line labeled **cout** exits the left side of the block. Another output line labeled **sum** exits the bottom of the block, also marked with a diagonal slash and the number **4**, indicating a 4-bit bus.

The diagram illustrates a 4-bit ripple-carry adder circuit. It consists of four Full Adders (FA) arranged in a chain. The inputs are two 4-bit numbers,  $a$  and  $b$ , represented by bits  $a_0, a_1, a_2, a_3$  and  $b_0, b_1, b_2, b_3$  respectively. The carry-in to the rightmost FA is  $1'b0$ . The carry-out of each FA is connected to the carry-in of the next FA to its left, labeled  $c_0, c_1, c_2$ . The final carry-out is labeled  $c_{out}$ . The sum outputs are labeled  $sum_0, sum_1, sum_2, sum_3$ . The entire circuit is enclosed in a yellow box, and a pink box labeled `ripple_adder_4` is shown in the top left corner.



# 4-bit Ripple Carry Adder (2)

## ❖ Verilog code

```
module ripple_adder_4(a, b, cout, sum);  
    4-bit ← input [3:0] a, b;  
    output [3:0] sum;  
    output cout;  
    입출력 외. ← wire [2:0] c;  
    module 호출 ← fa u0(.a(a[0]), .b(b[0]), .cin(1'b0), .cout(c[0]), .sum(sum[0]) );  
    instance 이름 ← fa u1(.a(a[1]), .b(b[1]), .cin(c[0]), .cout(c[1]), .sum(sum[1]) );  
    fa u2(.a(a[2]), .b(b[2]), .cin(c[1]), .cout(c[2]), .sum(sum[2]) );  
    fa u3(.a(a[3]), .b(b[3]), .cin(c[2]), .cout(cout), .sum(sum[3]) );  
    endmodule  
    현재module의 port name, wire
```



# 4-bit Adder

## ❖ Behavioral model

```
module adder_4(a, b, cout, sum);  
input [3:0] a, b;  
output [3:0] sum;  
output cout;
```

'+' library에 있는 **adder**를 사용하게 된다.

```
assign {cout, sum} = a + b ;
```

→ 합쳐서 결과를 내보낼 수 있도록 한다.

```
endmodule
```

모델링이 쉽지만, **library**에 있는 **adder**의 종류를 모르기 때문에 **gate-level**보다 빠를 수도, 늦을 수도 있다.



# **4-bit Adder-Subtractor**

---



# 2의 보수의 표현

## ❖ 2의 보수

- 양수 A의 word length를 W, A의 2의 보수(-A)를  $A^c$ 라 할 때,  
 $A^c = 2^W - A$
- Ex> W=4인 2의 보수  
7 = 0111, -7 = 10000 - 0111 = 1001  
- 다른 관점 : 10000 - 0111 = (1111+0001) - 0111  
= (1111-0111) + 0001  
- “1111”에서 A를 빼는 것은 A의 각 비트를 complement 취하는 것과 같으며,  
그 결과에 1만 더하면 2's complement가 얻어짐.

-8   4   2   1

$$1 \ 1 \ 0 \ 0 = -8 + 4 = -4$$

$$1 \ 1 \ 0 \ 1 = -8 + 4 + 1 = -3$$



## 2의 보수의 덧셈

❖ 양, 음수에 무관하게 두수를 더한 후, carry-out을 무시

$$\begin{array}{r} 1 \quad 1 \quad 0 \quad 0 \quad (-4) \\ + \quad 1 \quad 1 \quad 0 \quad 1 \quad (-3) \\ \hline \end{array}$$

$$\begin{array}{r} \boxed{1} \quad 1 \quad 0 \quad 0 \quad 1 \quad (-7) \end{array}$$

↓  
**carry-out** 발생

Carry를 무시하면 원하는  
결과인 (-7)을 얻음

$$\begin{array}{r} 1 \quad 1 \quad 0 \quad 0 \quad (-4) \\ + \quad 0 \quad 0 \quad 1 \quad 1 \quad (3) \\ \hline \end{array}$$

$$\begin{array}{r} \boxed{\phantom{0}} \quad 1 \quad 1 \quad 1 \quad 1 \quad (-1) \end{array}$$

↓  
**carry-out** 발생하지 않음

Carry를 발생없이 원하는  
결과인 (-1)을 얻음





# Overflow (1)

## ❖ Hardware의 표현 가능한 수의 범위를 벗어나는 것

- 양수와 양수의 합의 결과가 음수로 표현
- 음수와 음수의 합의 결과가 양수로 표현

$$\text{Ex : } (+5) + (+4) = (+9)$$

$$\begin{array}{rcccccc} & 0 & 1 & 0 & 1 & (5) \\ + & 0 & 1 & 0 & 0 & (4) \\ \hline & 1 & 0 & 0 & 1 & (-7) \end{array}$$

$$(-6) + (-5) = (-11)$$

$$\begin{array}{rcccccc} & 1 & 0 & 1 & 0 & (-6) \\ + & 1 & 0 & 1 & 1 & (-5) \\ \hline & \boxed{1} & 0 & 1 & 0 & 1 & (5) \end{array}$$

↓  
**carry-out** 발생

4-bit 는 **-8 ~ 7**까지 표현된다.

계산 결과가 이 범위를 넘어서면 **Overflow**가 발생한다.



# Adder-Subtractor (1)

## ❖ Overflow에 무관한 adder-subtractor

-8	4	2	1	
1	1	0	0	$= -8 + 4 = -4$
0	1	0	1	$= 4 + 1 = 5$

-16	8	4	2	1	
1	1	1	0	0	$= -16 + 8 + 4 = -4$
0	0	1	0	1	$= 4 + 1 = 5$

	1	0	1	0	(-6)
+	1	0	1	1	(-5)
<hr/>					
1	0	1	0	1	(-11)
$-16 + 4 + 1 = -11$					

	0	1	1	0	(6)
+	0	1	0	1	(5)
<hr/>					
0	1	0	1	1	(11)
$8 + 2 + 1 = 11$					

같은 부호의 덧셈일 경우에는  
부호 고려하지 않음

	1	1	0	0	(-4)
+	0	0	1	1	(3)
<hr/>					
0	1	1	1	1	(15)
$8 + 4 + 2 + 1 = 15$					

	0	1	0	1	(5)
+	1	1	0	1	(-3)
<hr/>					
1	0	0	1	0	(-14)
$-16 + 2 = -14$					

다른 부호의 덧셈일 경우에는  
부호를 고려해 주어야 한다.



# Adder-Subtractor (2)

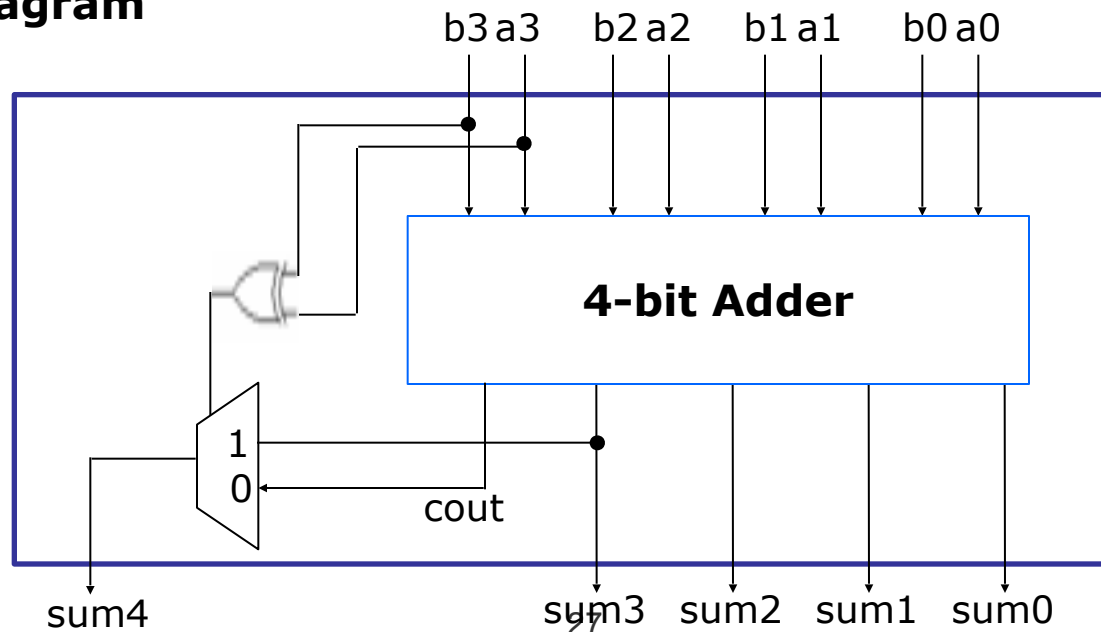
	1	1	0	0	(-4)
+	0	0	1	1	(3)
<hr/>					
	1	1	1	1	(-1)

$$-16+8+4+2+1 = -1$$

	0	1	0	1	(5)
+	1	1	0	1	(-3)
<hr/>					
	0	0	0	1	(2)

두 입력의 부호 비트를 "**xor**"하여  
**1**일 경우에 부호 비트를 확장한다.

- Block Diagram





# Adder-Subtractor (3)

## ❖ Verilog code

```
module add_sub(a, b, sum);  
  input  [3:0] a, b;  
  output [4:0] sum;  
  wire cout;  
  
  adder_4  u0 (.a(a), .b(b), .cout(cout), .sum(sum[3:0]));  
  assign sum[4]=(a[3]^b[3])? cout : cout;  
  
endmodule
```



# Decoder & Encoder

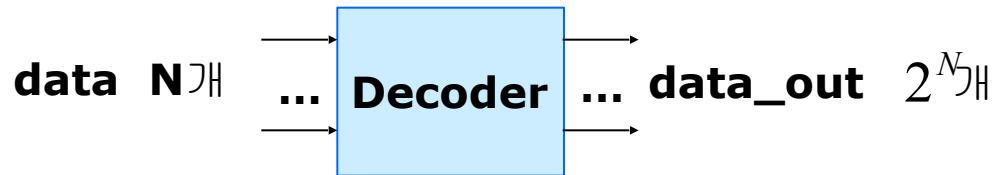
---



# Decoder & Encoder

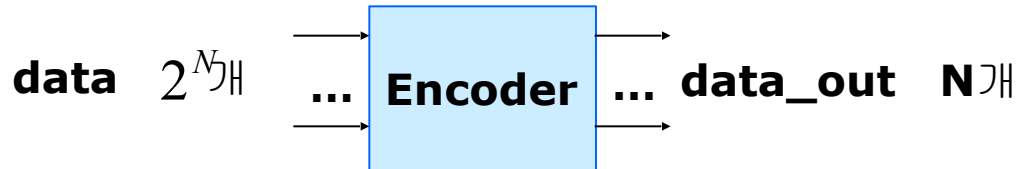
## ❖ Decoder

- N개의 입력에 대하여 최대  $2^N$ 개의 출력을 갖는다.



## ❖ Encoder

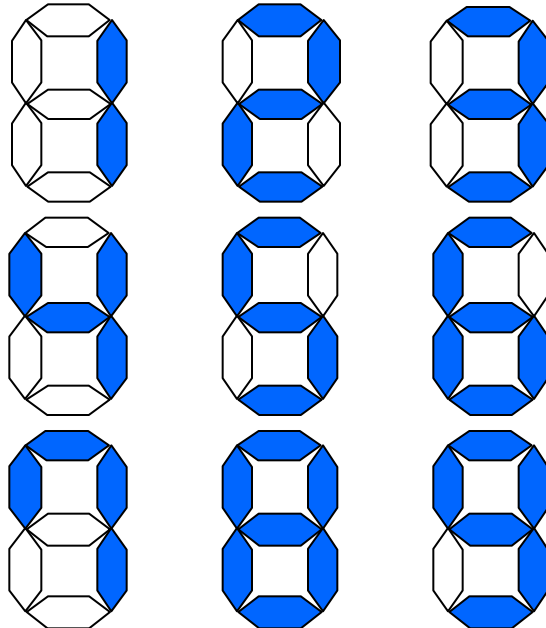
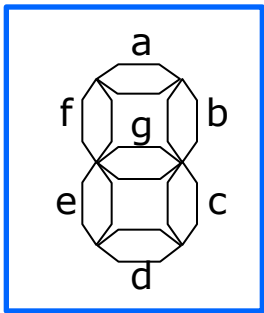
- $2^N$ 개의 입력으로부터 N개의 출력을 만든다.





# 7-Segment Decoder(1)

- Block Diagram



- Truth Table

	input				output						
	bcd[3:0]				seg [6:0]						
					a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	1	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	0	0	1	1
A	1	0	1	0	X	X	X	X	X	X	X
B	1	0	1	1	X	X	X	X	X	X	X
C	1	1	0	0	X	X	X	X	X	X	X
D	1	1	0	1	X	X	X	X	X	X	X
E	1	1	1	0	X	X	X	X	X	X	X
F	1	1	1	1	X	X	X	X	X	X	X



# 7-Segment Decoder(2)

## ❖ Verilog code

```
module seg_decoder(bcd, seg);  
input  [3:0] bcd;  
output [6:0] seg;  
reg      [6:0] seg;  
always (bcd) begin  
    case(bcd)  
        0:seg = 7'b111_1110;  
        1:seg = 7'b011_0000;  
        2:seg = 7'b110_1101;  
        3:seg = 7'b111_1001;  
        4:seg = 7'b011_0011;  
        5:seg = 7'b101_1011;  
        6:seg = 7'b101_1111;  
        7:seg = 7'b111_0010;  
        8:seg = 7'b111_1111;  
        9:seg = 7'b111_0011;  
        default:seg = 7'bxxx_xxxx;  
    endcase  
end  
endmodule
```





# Digital Timer

---

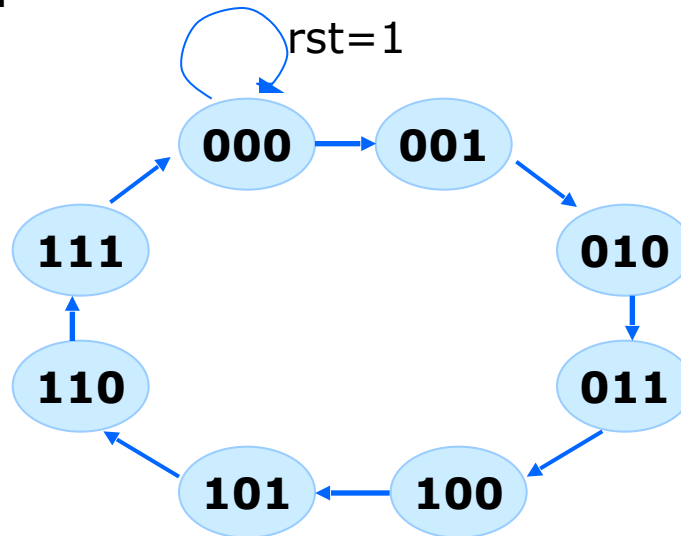


# Counter

## ❖ 펄스(Pulse)를 세는 회로

- 이벤트 카운팅, 순차제어, 주파수 분할등의 응용회로에서 사용

### • State Diagram



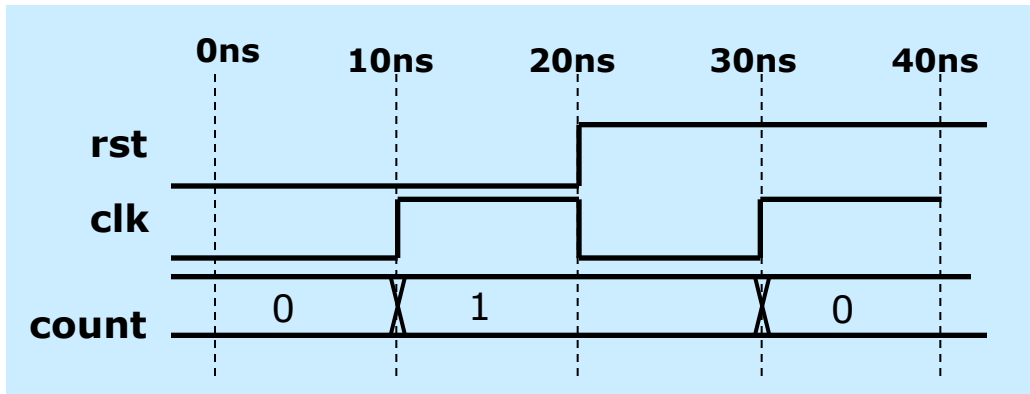
3비트 카운터의 최대 카운터 수는  $8(2^3 = 8)$  이며, mod-8 카운트 순서는 000~111이다.



# 동기식 및 비동기식 카운터

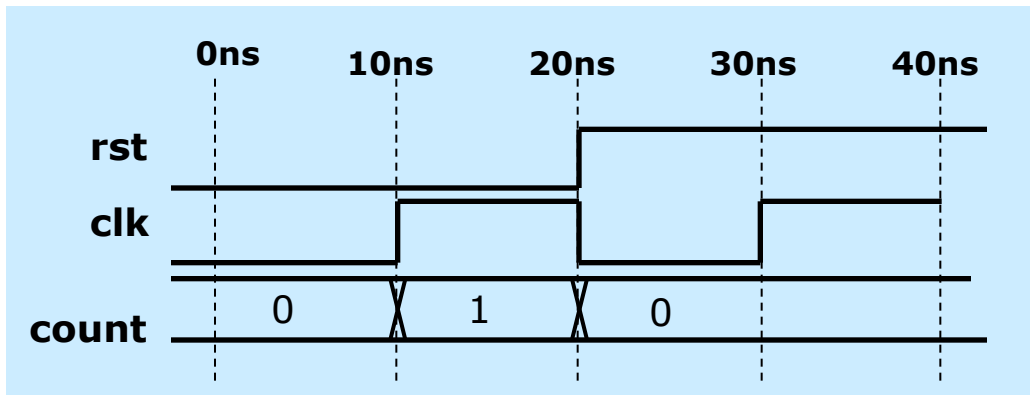
## ❖ 동기식(Synchronous)

### • Timing Diagram



## ❖ 비동기식(asynchronous)

### • Timing Diagram

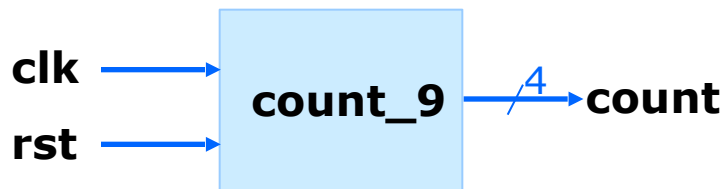




# Counter\_10

## ❖ 0~9까지 동작하는 Counter

- **Block Diagram**



- **Verilog Code**

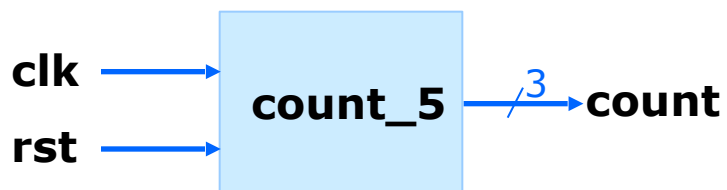
```
module count_9(clk, rst, count);  
  input  clk, rst;  
  output [3:0] count;  
  reg [3:0] count;  
  always @(posedge clk or posedge rst )  
  begin  
    if(rst) count = 4'b0;  
    else begin  
      if(count==4'b1001) count = 4'b0;  
      else count = count +1; end  
    end  
  endmodule
```



# Counter\_6

## ❖ 0~5까지 동작하는 counter

- **Block Diagram**



- **Verilog Code**

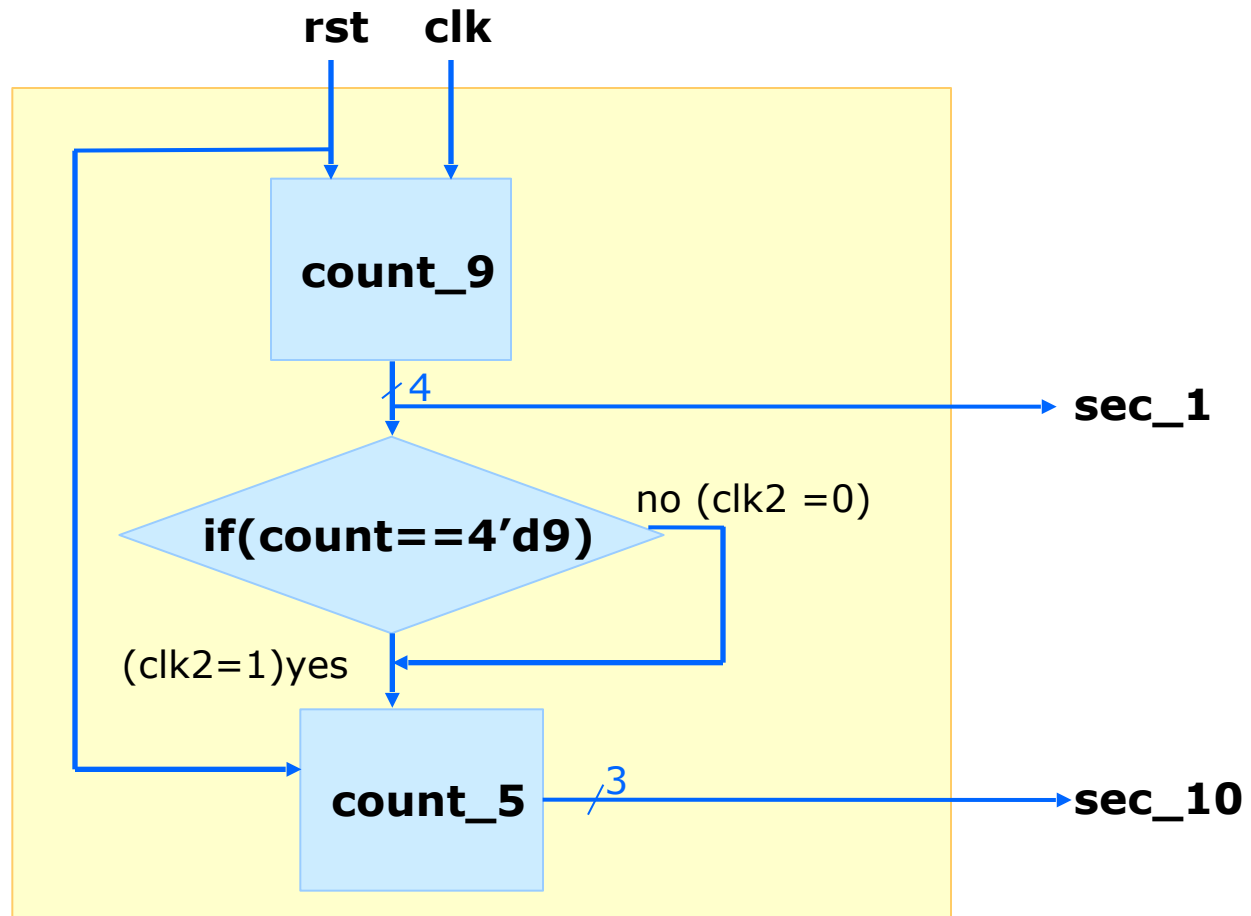
```
module count_5(clk, rst, count);  
  input  clk, rst;  
  output [2:0] count;  
  reg [2:0] count;  
  always @(posedge clk or posedge rst )  
  begin  
    if(rst) count = 3'b0;  
    else begin  
      if(count==3'b101) count = 3'b0;  
      else count = count +1; end  
    end  
  end  
endmodule
```



# Timer

## ❖ 0~59까지 동작하는 Timer

- Block Diagram





```
module timer(clk, rst, sec_1, sec_10);
input clk, rst;
output [3:0] sec_1;
output [2:0] sec_10;
reg clk2;

count_9 u0(clk, rst, sec_1);
count_5 u1(clk2, rst, sec_10);

always@(posedge clk or posedge rst)
Begin
    if(rst)begin
        clk2=1'b0;
    end
    else begin
        if(sec_1==4'd9)
            clk2 = 1'b0;
        else
            clk2 = 1'b1;
        end
    end
end
endmodule
```