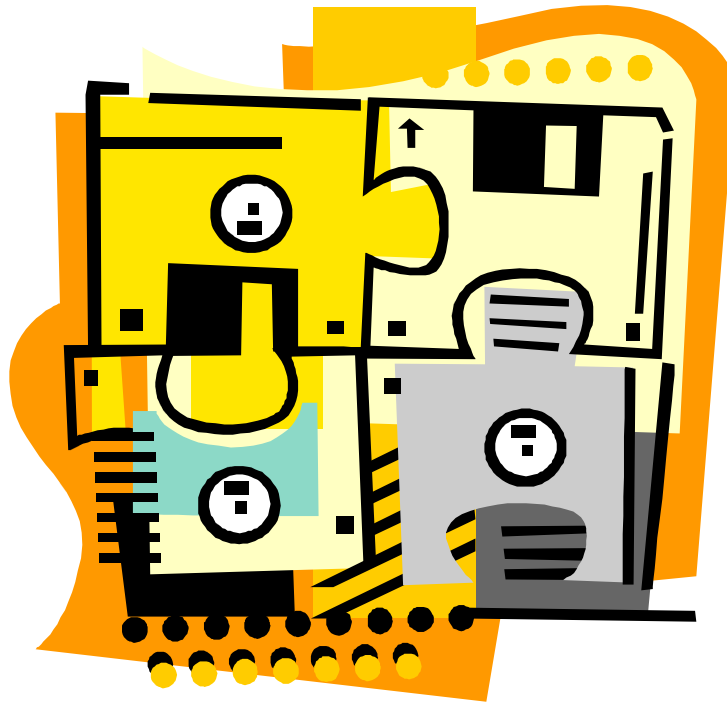
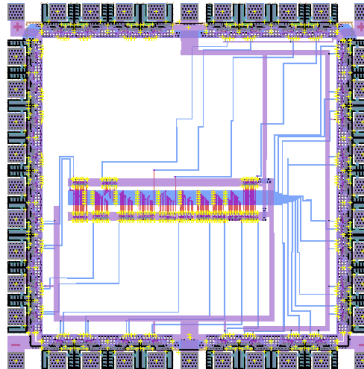


Verilog HDL Intro.



1. Overview



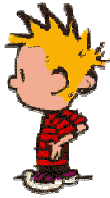


1.1.2 HDL

ÇHDL (hardware description language) : 하드웨어를 기술하고
시뮬레이션, 합성을 하기 위해 고안된 프로그래밍 언어
ex.) Verilog HDL, VHDL

ÇAdvantages of HDL

- Easy to **describe hardware system**
- Easy to convert **designs** to **implementations**
(**Automatic hardware implementation** by computer)
- Easy to debug (**Simulation** by computer)
- Easy for **rapid prototyping**



1.1.4 Simulation & Synthesis

ÇSimulation : 모의 실험

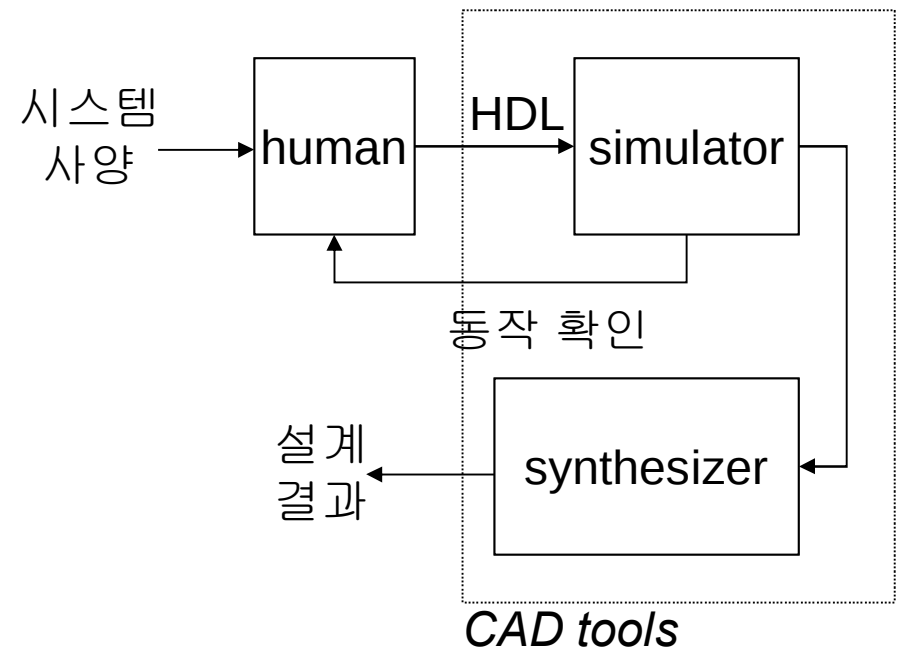
- 만들지 않으니까 돈이 들지 않는다.
- 제대로 만들었는지 확인하기 쉽다.
- 설계가 틀렸을 경우, 고쳐서 다시 시뮬레이션 하여 확인하기 쉽다.
- 모든 경우를 다 테스트할 수 없으므로, 실수의 가능성이 항상 상존한다.
- HDL을 써서 simulation과 synthesis를 함께 진행하면 시간과 노력이 절약된다.

멍청한 컴퓨터가 계산한 결과를 그대로 믿다가는
쫄딱 망하는 경우도 있다!

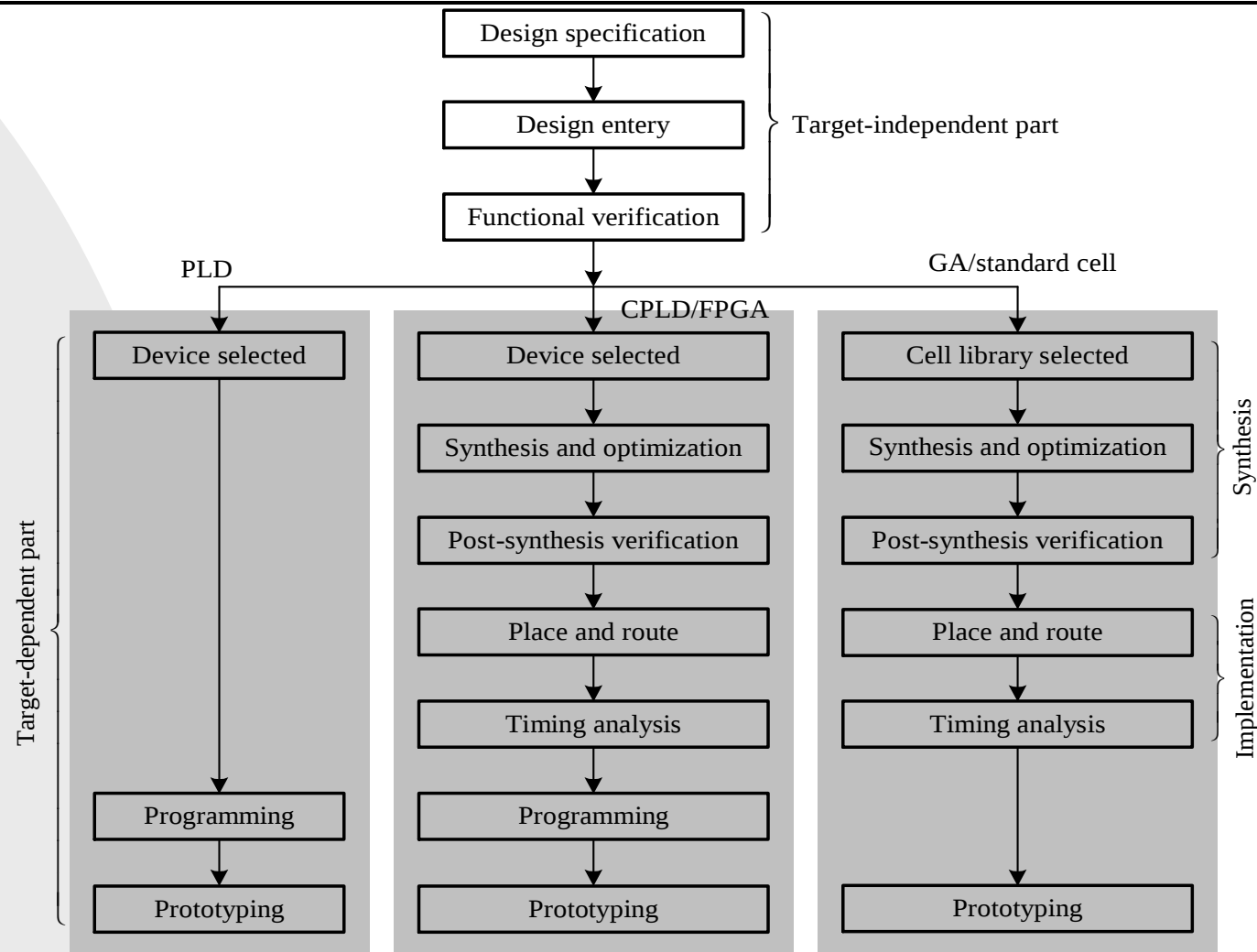


ÇSynthesis : 자동화된 디자인 합성

- C 프로그래밍처럼 behavior만 기술하면 나머지는 컴퓨터가 알아서 합성한다.
- 사람 손으로 할 때보다 최적화된 결과



HDL-Based Design Flow

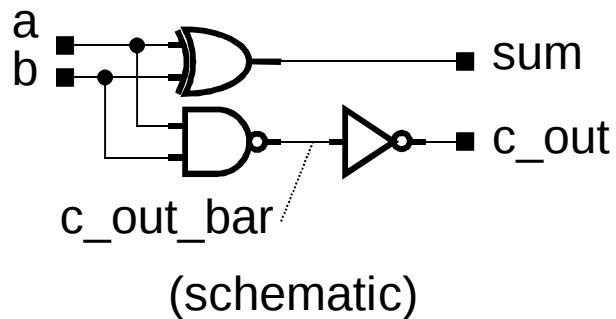
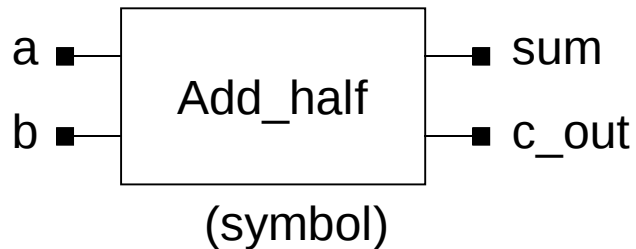




1.2.1 Structural Description

Structural Description in Verilog HDL

- 주어진 블록을 다른 primitives의 연결로 나타냄.
- Symbol, schematic, HDL code로 구성.
- 설계자는 symbol과 schematic을 작성하고, HDL code는 schematic으로부터 자동적으로 생성됨.



```
module Add_half (sum, c_out, a, b);  
  
  input    a, b ;  
  output   sum, c_out ;  
  wire     c_out_bar ;  
  
  xor(sum, a, b) ;  
  nand(c_out_bar, a, b) ;  
  not(c_out, c_out_bar) ;  
  
endmodule
```

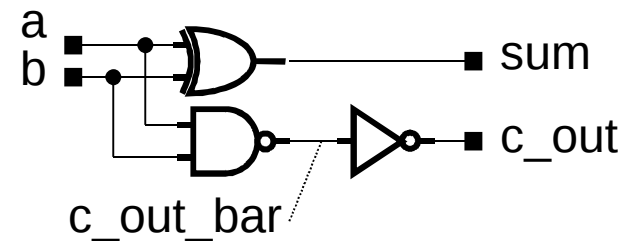
(HDL code)



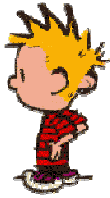
1.2.1 Structural Description

CHDL code

```
module module_name(module_port_list);  
  
// declaration of port modes  
input    input_mode_module_port_list ;  
output   output_mode_module_port_list ;  
// declaration of internal signals  
wire     internal_signal_list ;  
// design primitives  
primitive_name(signal_list) ;  
primitive_name(signal_list) ;  
primitive_name(signal_list) ;  
ë  
  
endmodule
```



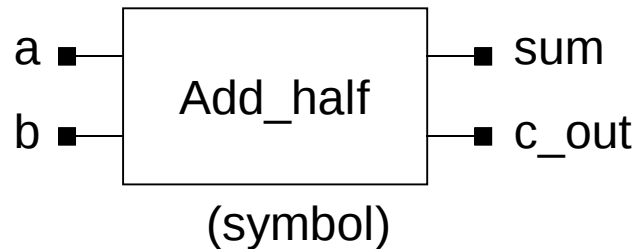
```
module Add-half (sum, c_out, a, b);  
  
// declaration of port modes  
input    a, b ;  
output   sum, c_out ;  
// declaration of internal signals  
wire     c_out_bar ;  
// design primitives  
xor (sum, a, b) ;  
nand (c_out_bar, a, b) ;  
not (c_out, c_out_bar) ;  
  
endmodule
```



1.2.2 Behavioral Description

Behavioral Description in Verilog HDL

- input이 어떠한 값일 때 output이 어떠한 값을 갖는지를 기술.
- Symbol, HDL code로 구성되고 schematic은 없음.
- 설계자는 symbol과 HDL code만을 작성.



```
module Add_half (sum, c_out, a, b);  
  
  input      a, b ;  
  output    sum, c_out ;  
  reg       sum, c_out ;  
  
  always@(a or b)  
  begin  
    sum = a ^ b ;  
    c_out = a & b;  
  end  
  
endmodule
```

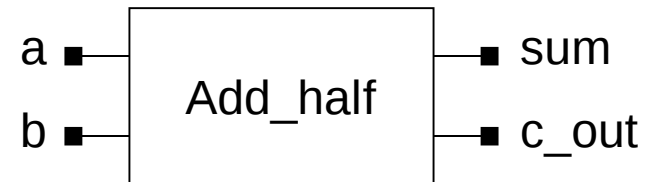
(HDL code)



1.2.2 Behavioral Description

CHDL code

```
module module_name(module_port_list);  
  
// declaration of port modes  
input    input_mode_module_port_list ;  
output   output_mode_module_port_list ;  
// declaration of abstract memory variables  
reg      output_mode_module_port_list ;  
// behavioral function  
always@(event_list)  
  begin  
    output = f(input) ;  
    output = f(input) ;  
    output = f(input) ;  
  ÷  
  end  
  
endmodule
```

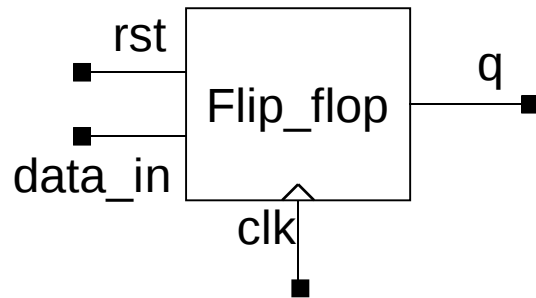


```
module Add_half (sum, c_out, a, b);  
  
// declaration of port modes  
input    a, b ;  
output   sum, c_out ;  
// declaration of abstract memory variables  
reg      sum, c_out ;  
// behavioral function  
always@(a or b)  
  begin  
    sum = a ^ b ;  
    c_out = a & b ;  
  end  
  
endmodule
```



1.2.3 Example of Verilog HDL

Ç Behavioral Description of D-type flip-flop

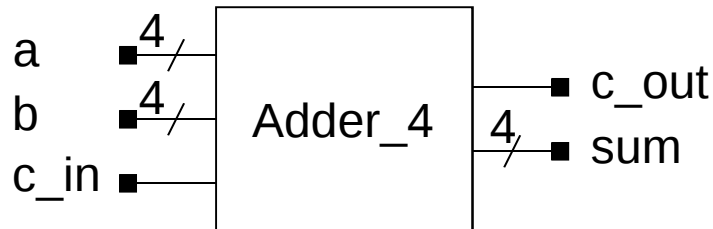


```
module Flip_flop (q, data_in, clk, rst);  
  
  // declaration of port modes  
  input    data_in, clk, rst ;  
  output   q ;  
  // declaration of abstract memory variables  
  reg      q ;  
  // behavioral function  
  always@(posedge clk)  
  begin  
    if (rst==1)  
      q = 0 ;  
    else  
      q = data_in ;  
    end  
  
endmodule
```



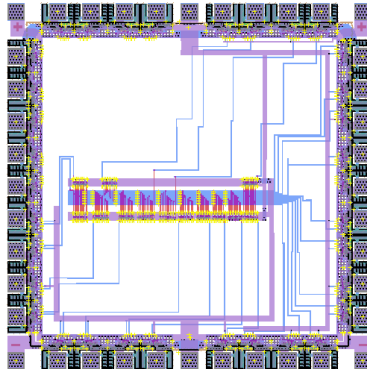
1.2.3 Example of Verilog HDL

Ç Behavioral Description of 4-bit adder



```
module Adder_4 (sum, c_out, a, b, c_in);  
  
    // declaration of port modes  
    input    [3:0]    a, b ;  
    input    c_in ;  
    output    [3:0]    sum ;  
    output    c_out ;  
  
    // behavioral function  
    assign {c_out, sum} = a + b + c_in ;  
  
endmodule
```

2. Syntax





2.1.1 Module

ÇModule: **Basic unit** of design in Verilog HDL

ÇModule is never declared **within another module**.

ÇStructural description and behavioral descriptions can be **mixed together** in a single module.

ÇA module consists of:

(1) **Module port** : **input**, **output**, **inout**

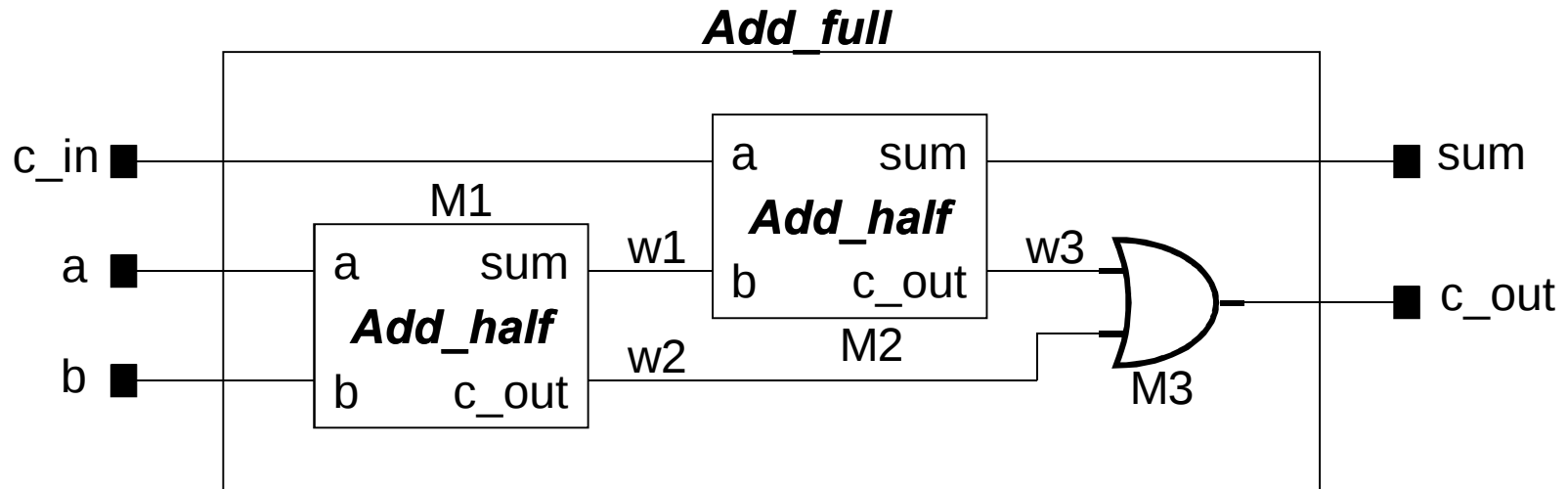
(2) **Module implementation**

(2-1) Sub-modules and connections (Structural description)

(2-2) Behavioral functions (Behavioral description)



2.1.2 Module Instantiation



=

```
module Add_full (sum, c_out, a, b);  
  input      a, b, c_in ;  
  output     sum, c_out ;  
  wire       w1, w2, w3 ;  
  
  Add_half M1(w1, w2, a, b) ;  
  Add_half M2(sum, w3, w1, c_in) ;  
  or M3(c_out, w2, w3) ;  
endmodule
```

+

```
module Add_half (sum, c_out, a, b);  
  input      a, b ;  
  output     sum, c_out ;  
  wire       c_out_bar ;  
  
  xor (sum, a, b) ;  
  nand (c_out_bar, a, b) ;  
  not (c_out, c_out_bar) ;  
endmodule
```



2.2.1 Primitives

ÇPrimitive: **Predefined** module

(=Predefined structural/functional element)

ÇModule $\supset$ Primitive

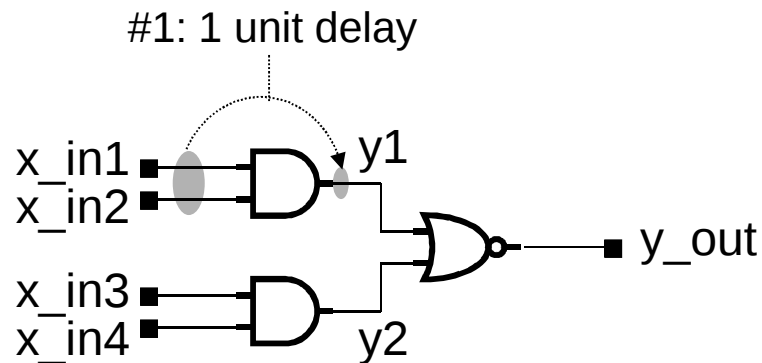
ÇBuilt-in Verilog Primitives

Combinational Logic	Three State	MOS Gate	CMOS Gate	Bi-directional Gate	Pull Gate
and nand or nor xor xnor buf not	bufif0 bufif1 notif0 notif1	nmos pmos rnmos rpmos	cmos rcmos	tran tranif0 tranif1 rtran rtranif0 rtranif1	pullup pulldown



2.2.2 Delay Modeling

Ç # *N*: 입력과 출력 사이의 지연 시간을 *N* unit delay로 설정



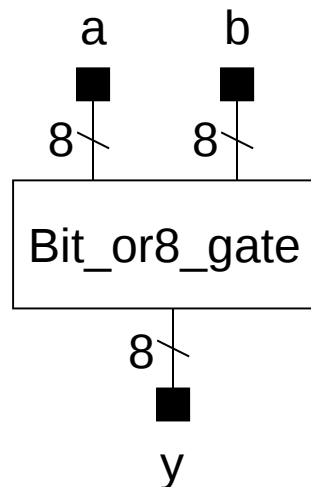
```
module AOI_4_unit (y_out, x_in1, x_in2, x_in3, x_in4);  
  
  input    x_in1, x_in2, x_in3, x_in4 ;  
  output   y_out ;  
  wire     y1, y2 ;  
  
  and #1 (y1, x_in1, x_in2) ;  
  and #1 (y2, x_in3, x_in4) ;  
  nor #1 (y_out, y1, y2) ;  
  
endmodule
```



2.3.2 Behavioral Description

ÇContinuous Assignment:

- operator를 사용하여 combinational logic을 기술



```
module Bit_or8_gate1 (y, a, b) ;  
  
  input    [7:0] a, b ;  
  output   [7:0] y ;  
  
  assign   y = a | b ;  
  
endmodule
```



2.3.2 Operator

ÇUseful Operators

arithmetic operation	
+	addition
-	subtraction

bitwise operation	
~	negation
&	and
	or
^	exclusive-or
~&	nand
~	nor
~^	exclusive-nor

```
module Nand2_RTL (y, x1, x2) ;
```

```
input    x1, x2 ;
```

```
output   y ;
```

```
assign   y = ~(x1 & x2) ;
```

```
endmodule
```

```
module And2_8bit_RTL (y, x1, x2) ;
```

```
input    [7:0] x1 ;
```

```
input    [7:0] x2 ;
```

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

```
assign   y = x1 & x2 ;
```

```
endmodule
```



2.3.2 Operator

ÇUseful Operators

shift operation	
<<	shift left
>>	shift right

reduction operation	
~	negation
&	and
	or
^	exclusive-or
~&	nand
~	nor
~^	exclusive-nor

```
module Shl_4bit_RTL (y, x, c) ;
```

```
input      [3:0]x ;
```

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

```
output     [3:0]y ;
```

```
assign     y = x << c;
```

```
endmodule
```

```
module And4_RTL (y, x) ;
```

```
input      [3:0]x ;
```

```
output     y ;
```

```
assign     y = &x ;
```

```
endmodule
```



2.3.2 Operator

ÇUseful Operators

logical operation	
!	negation
&&	and
	or
==	equal
!=	inequal

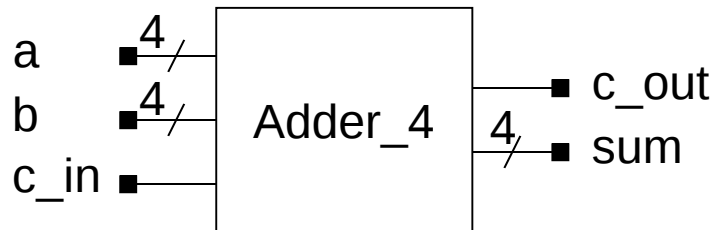
relational operation	
<	less
<=	less or equal
>	greater
>=	greater or equal

```
module And2_algo (y, x1, x2) ;  
  
  input      x1, x2 ;  
  output    y ;  
  reg       y ;  
  
  always@(x1 or x2)  
  begin  
    if ((x1==1)&&(x2==1))  
      y = 1 ;  
    else  
      y = 0 ;  
    end  
  
  endmodule
```



2.3.2 Operator

Concatenation: { } 기호를 사용하여 두개 이상의 signal을 하나의 binary number처럼 취급



```
module Adder_4 (sum, c_out, a, b, c_in) ;  
  
// declaration of port modes  
input    [3:0]    a, b ;  
input    c_in ;  
output   [3:0]    sum ;  
output   c_out ;  
  
// behavioral function  
assign {c_out, sum} = a + b + c_in ;  
  
endmodule
```

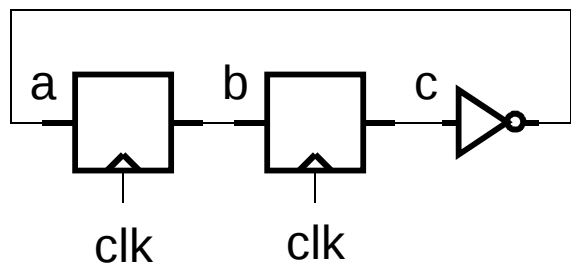


2.5.3 Sequential & Concurrent Execution of Statements

Module 내의 모든 statement는
기본적으로 병렬 수행

always (ë)
begin
ë
end 의 내부에서는

그 내부에서만 순차적으로 수행



두 개의 **FF**은 병렬적으로 수행



```
module Ring_osc2 (a, clk) ;  
  
  input      clk ;  
  output    a ;  
  reg       b, c ;  
  
  // inverter  
  not (a,c) ;  
  
  // 1st flip-flop  
  always@(posedge clk)  
  begin  
    b = a ;  
  end  
  
  // 2nd flip-flop  
  always@(posedge clk)  
  begin  
    c = b ;  
  end  
  
endmodule
```



2.5.3 Sequential & Concurrent Execution of Statements

ÇStructural Description & RTL Description (assign)

: 입력 신호가 바뀔 때마다 출력 신호가 바뀜.

```
// Structural description  
not (b,a) ;  
  
// RTL description  
assign c = ~b ;
```

ÇAlgorithm-based Description (always, if)

: always 문의 @(signal)이 바뀔때마다 beginë end가 수행됨.

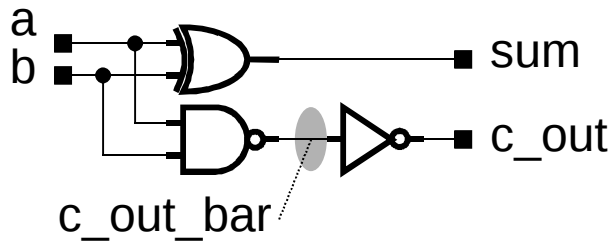
```
always@(posedge clk)  
begin  
    b = a ;  
end
```



2.5.4 Wire & Reg

Wire

- Structural Description에 쓰임
- module과 primitive를 연결하는 signal 중에서
input, output을 제외한 모든 **signal**을 **wire**로 선언



```
module Add_half (sum, c_out, a, b);  
  
  input    a, b ;  
  output   sum, c_out ;  
  wire     c_out_bar ;  
  
  xor (sum, a, b) ;  
  nand (c_out_bar, a, b) ;  
  not (c_out, c_out_bar) ;  
  
endmodule
```



2.5.4 Wire & Reg

QReg

- Algorithm-based Description에 쓰임
- 대입문(A = B)의 A에 해당하는 모든 signal을 reg로 선언

```
module Flip_flop (q, data_in, clk, rst) ;  
  input      data_in, clk, rst ;  
  output     q ;  
  reg        q ;  
  
  always@(posedge clk)  
  begin  
    if (rst==1)  
      q = 0 ;  
    else  
      q = data_in ;  
    end  
  endmodule
```



2.9.1 Synthesis Methods

ÇSynthesis : HDL description → Physical gates and connections

ÇSynthesis for Various Description Methods

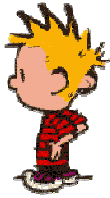
- Structural description : primitive를 gate로 치환한 다음
logic optimization을 실시
- RTL description : 각각의 assign문을 gate로 변환한 다음
logic optimization을 실시
- Algorithm-based description : always...beginë end문 안의
내용을 Verilog HDL이 분석한 다음, 그 내용을 수행할
수 있는 논리회로를 생성해서 gate로 바꾼 다음 logic
optimization을 실시



2.9.2 Synthesizability

ÇSynthesizability of Various Description Methods

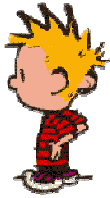
- Structural Description : 각 모듈이 하드웨어 요소와 1대1로 대응되므로 합성이 매우 쉬움
- RTL Description : behavioral description이지만 assign문을 손쉽게 하드웨어 요소로 변환할 수 있으므로 합성이 비교적 쉬움
- Algorithm-based Description : 프로그래밍 언어로 기술하므로 합성이 비교적 어려움
- Synthesizability: Structural > RTL > Algorithm-based



2.9.3 Flexibility & Design Time

Flexibility and Design Time of Various Description Methods

- Structural Description : 설계를 수정, 확장하기 번거롭고 복잡한 시스템을 설계하기 불편하다.
- RTL Description : Structural 과 Algorithm-based의 중간
- Algorithm-based Description : 사람이 생각하는 방식대로 프로그래밍 언어를 사용하여 기술하므로, 수정, 확장이 용이하고 설계 시간이 짧음
- Flexibility: Structural < RTL < Algorithm-based
- Design Time: Structural > RTL > Algorithm-based



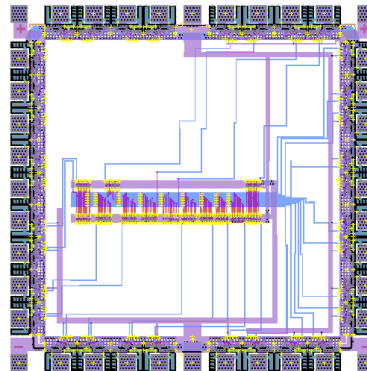
2.11.1 Representation of Numbers

Numbers in Verilog HDL

- `<size><base_format><number>`
- b: binary (2)
- d: decimal (10)
- o: octal (8)
- h: hexadecimal (16)
- number가 “x”나 “z”로 시작하면, 자릿수를 맞추고 남은 부분을 모두 “x”나 “z”로 채움.

number	value	stored binary pattern
2'b10	2	10
4'd10	10	1010
4'o10	8	0100
8'hA	10	00001010
5'bz1x	-	zzz1x
12'hxa	-	xxxxxxxx1010
32'bz	-	z(32bits)
3'b5	Invalid!	Invalid!
2'da	Invalid!	Invalid!

3. Simulation





3.2.1 Verilog HDL Simulation

ÇDesign Unit Testbench (DUTB)

: Verilog HDL에서 시뮬레이션을 하기 위해서 만드는 module

ÇUnit under Test (UUT) : 시뮬레이션 당하는 module

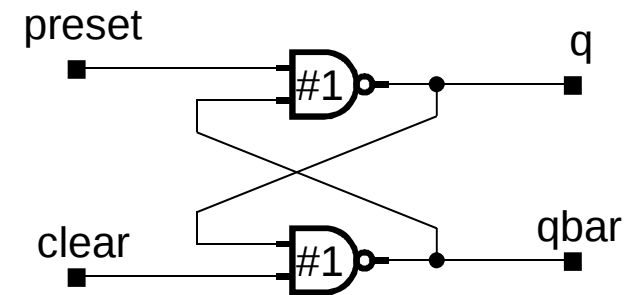
ÇDesign Unit Testbench = 시뮬레이션해야 할 module +
stimulus generator + response monitor

ÇStimulus generator, response monitor는 Verilog HDL로
기술되지만, 실제 설계할 하드웨어와는 전혀 관계가 없다.



3.2.2 Active-Low NAND Latch

```
module Nand_Latch_1 (q,qbar,preset,clear);  
  output q, qbar;  
  input preset, clear;  
  
  nand #1 G1 (q, preset, qbar), G2 (qbar, clear, q);  
endmodule
```



Preset	Clear	q(n+1)	qbar(n+1)
0	0	1	1
0	1	1	0
1	0	0	1
1	1	q(n)	qbar(n)

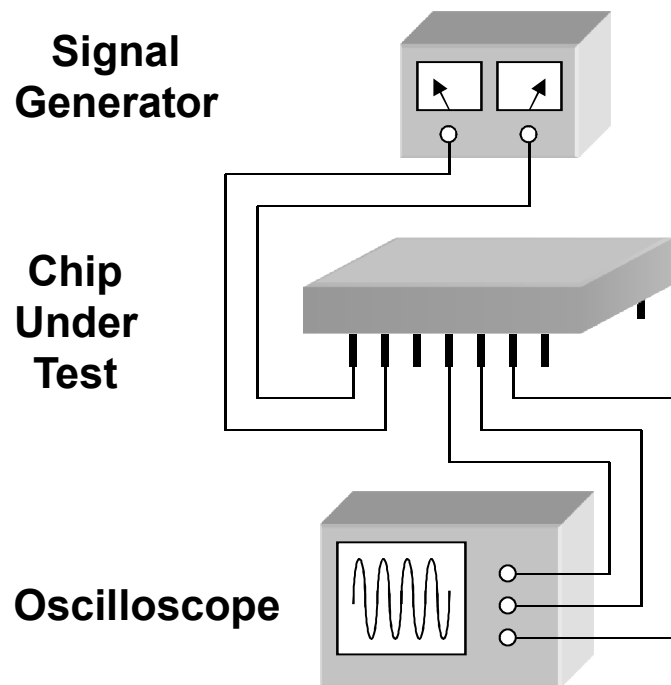
active low라 함은
Preset ,Clear가 0일
때에 각각 preset latch,
clear latch의 동작을
수행한다는 뜻이다.



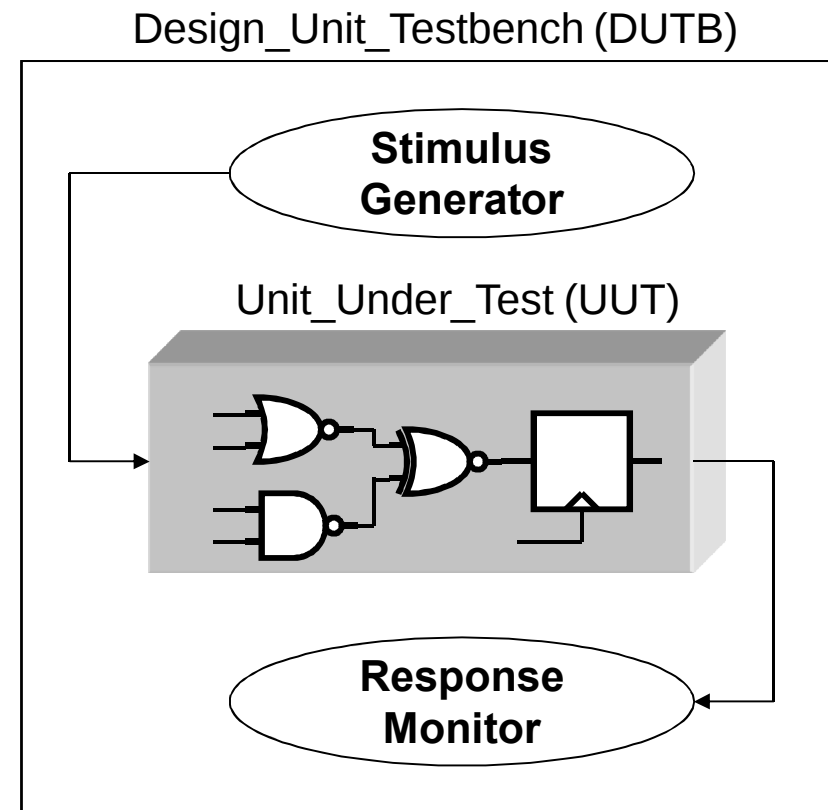


3.2.3 Design Unit Testbench (DUTB)

실제 하드웨어 테스트



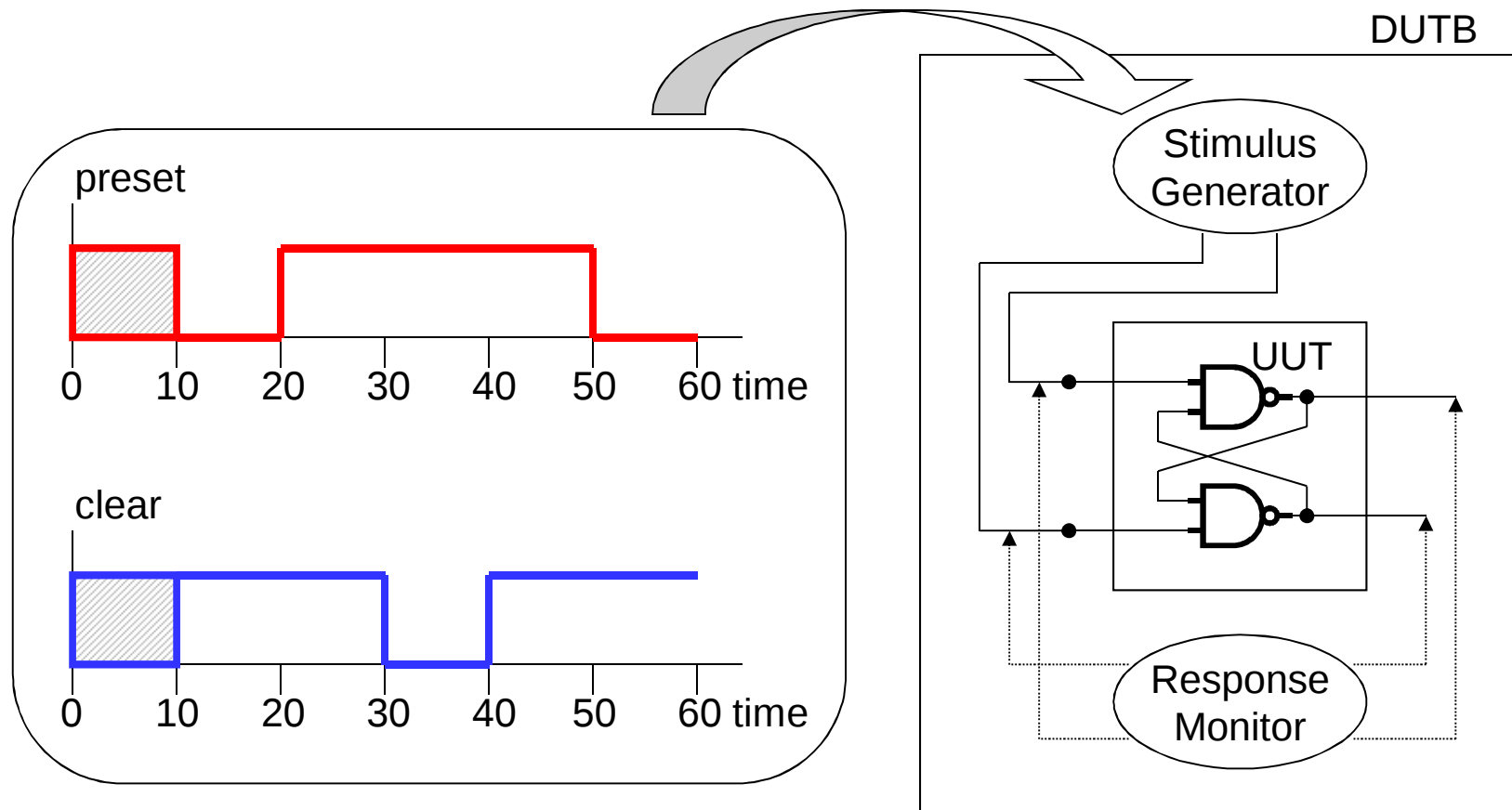
Verilog HDL Simulation



3.2 Testbench

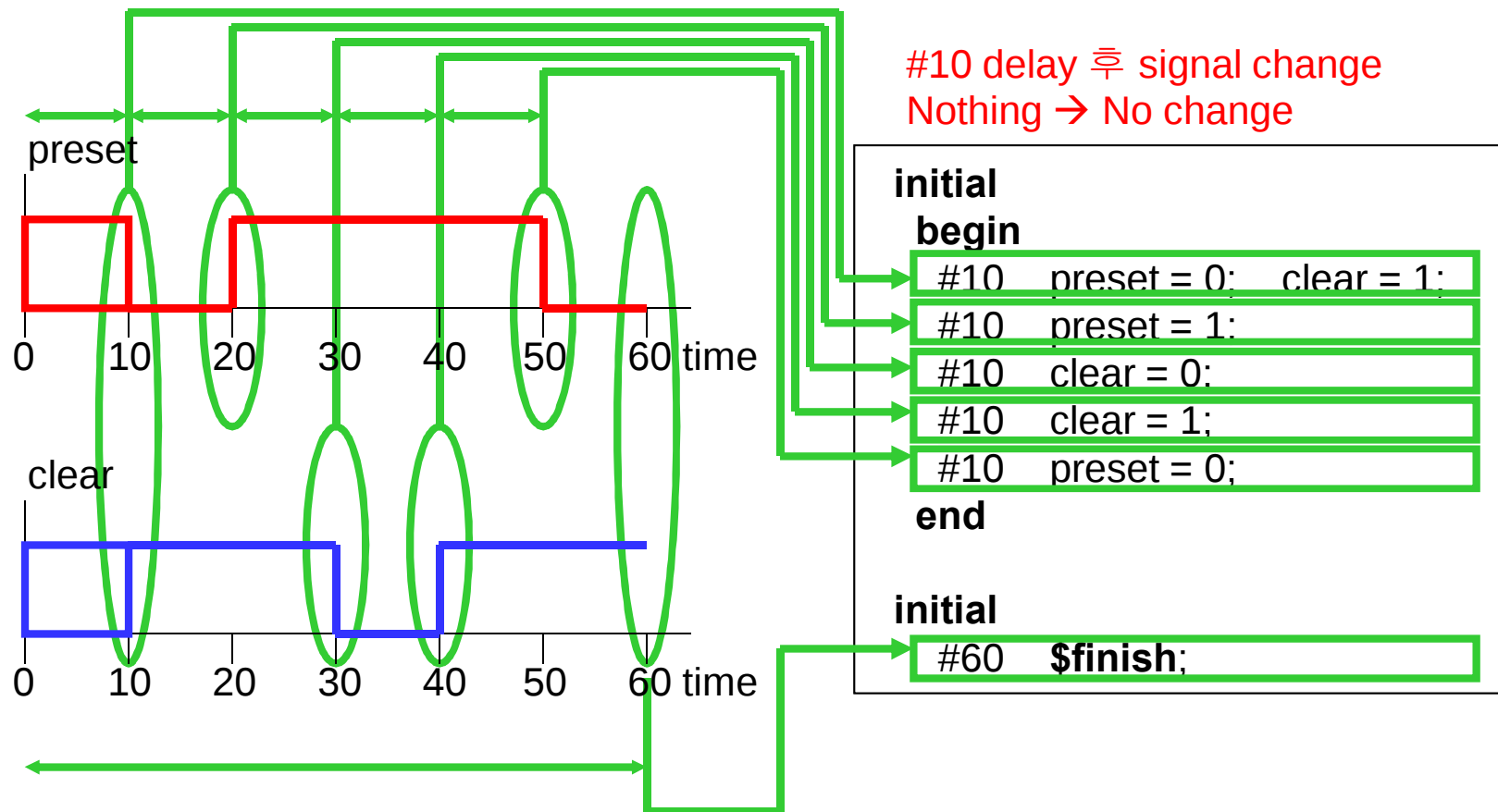


3.2.3 Design Unit Testbench (DUTB)





3.2.4 Stimulus Generator





3.2.5 Response Monitor

```
initial
begin
  $monitor(@time=%2d preset=%1b clear=%1b
  q=%1b qbar=%1b@ $time,preset,clear,q,qbar);
end
```

C++ Analogy

%b:2진수, %o:8진수,
%d:10진수, %h:16진수

\$time: simulation time을
나타내는 함수

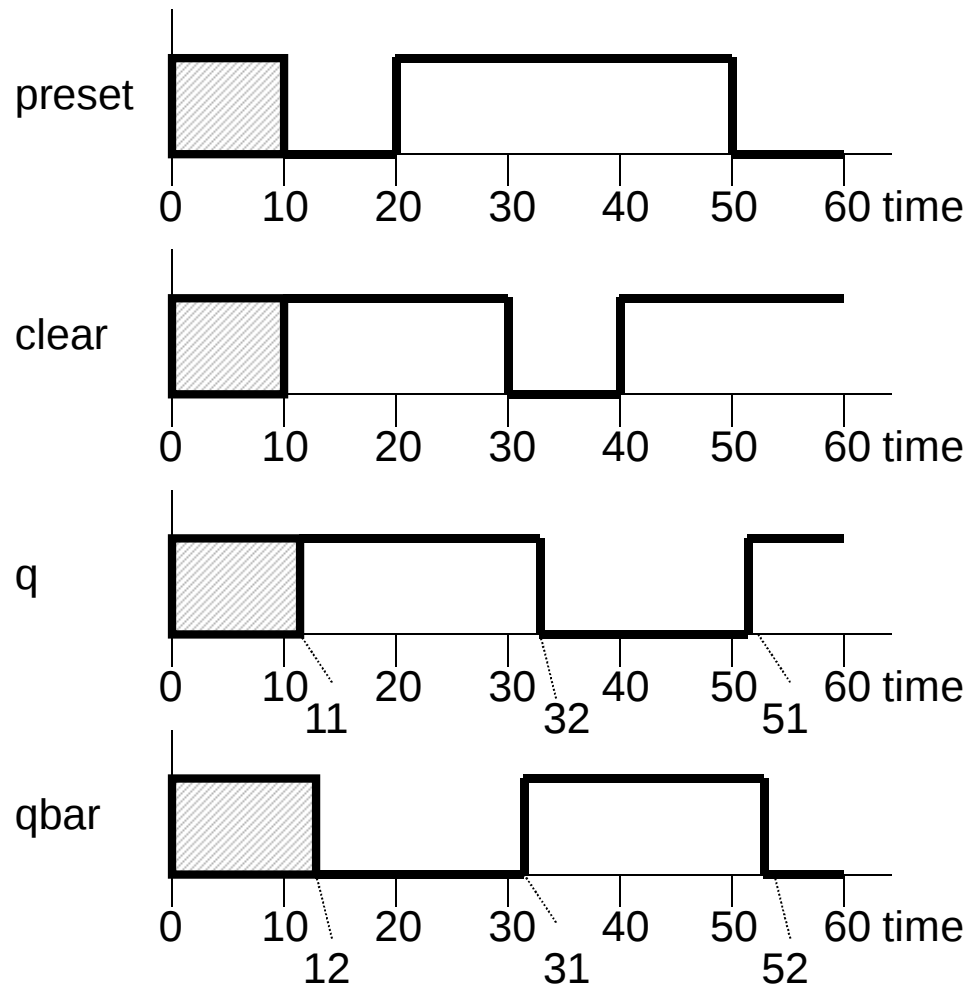
C++ Analogy

```
printf("@time=%2d preset=%1d clear=%1d
q=%1d qbar=%1d@ time(),preset,clear,q,qbar);
```



3.2.6 Monitor Output

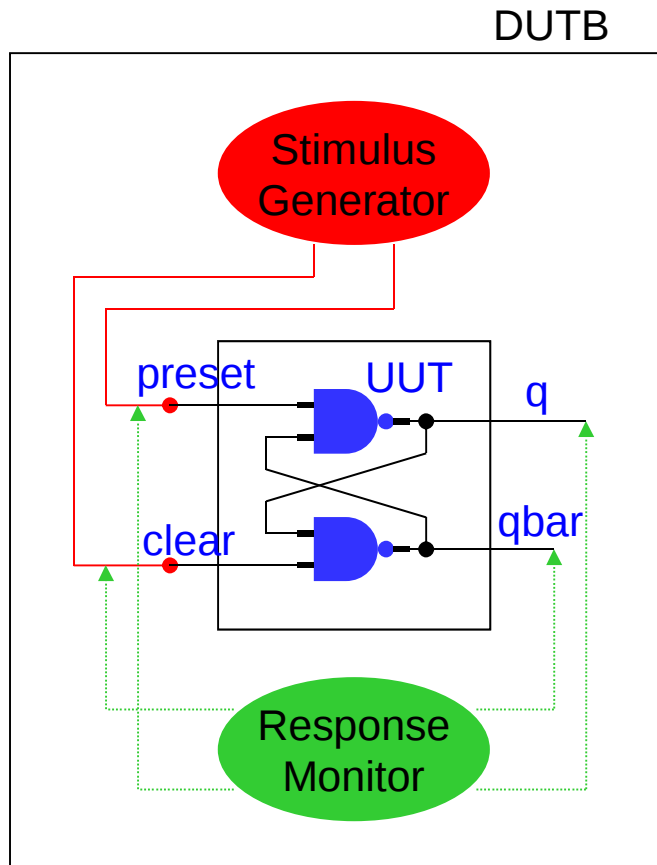
When there is an event.



```
time=0  preset=x clear=x q=x qbar=x
time=10 preset=0 clear=1 q=x qbar=x
time=11 preset=0 clear=1 q=1 qbar=x
time=12 preset=0 clear=1 q=1 qbar=0
time=20 preset=1 clear=1 q=1 qbar=0
time=30 preset=1 clear=0 q=1 qbar=0
time=31 preset=1 clear=0 q=1 qbar=1
time=32 preset=1 clear=0 q=0 qbar=1
time=40 preset=1 clear=1 q=0 qbar=1
time=50 preset=0 clear=1 q=0 qbar=1
time=51 preset=0 clear=1 q=1 qbar=1
time=52 preset=0 clear=1 q=1 qbar=0
```



3.2.7 DUTB Codes



```
module test_Nand_latch_1;
```

```
  reg      preset, clear;  
  wire     q, qbar;
```

```
// Unit under Test
```

```
Nand_Latch_1 M1 (q,qbar,preset,clear);
```

```
// Stimulus Generator
```

```
initial
```

```
begin
```

```
  #10    preset = 0;    clear = 1;
```

```
  #10    preset = 1;
```

```
  #10    clear = 0;
```

```
  #10    clear = 1;
```

```
  #10    preset = 0;
```

```
end
```

```
initial
```

```
  #60    $finish;
```

```
// Response Monitor
```

```
initial
```

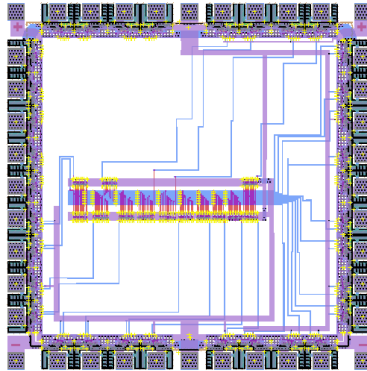
```
begin
```

```
  $monitor(@time=%2d preset=%1b clear=%1b  
           q=%1b qbar=%1b@,$time,preset,clear,q,qbar);
```

```
end
```

```
endmodule
```

4. Advanced Syntax





4.3.1 Net and Register

ÇNet

- data types: wire, supply0, supply1 etc
- structural description에 주로 쓰임
- physical connection을 의미

ÇRegister

- data types: reg, integer, real ...
- behavioral description에 주로 쓰임
- program variable (FF out, 기억의 의미)을 의미



4.3.2 How to Find Out Net and Register

Register

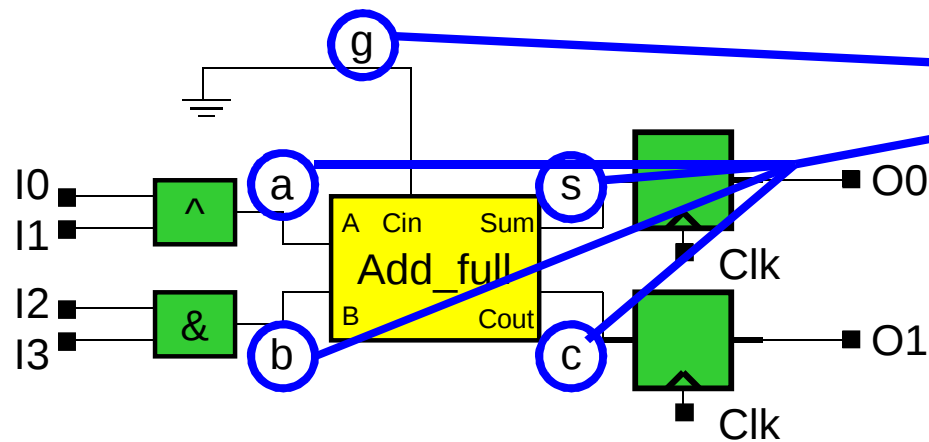
- 대입문 (**A=B**)에서 **A**에 해당하는 모든 **signal**을 **register**로 선언
- **assign** 명령어가 들어가 있는 **continuous assignment**에서는 대입문이 있더라도 **register**로 선언되지 않는다.

Net

- module과 primitive를 연결하는 signal 중에서 input, output, register를 제외한 모든 signal을 net로 선언



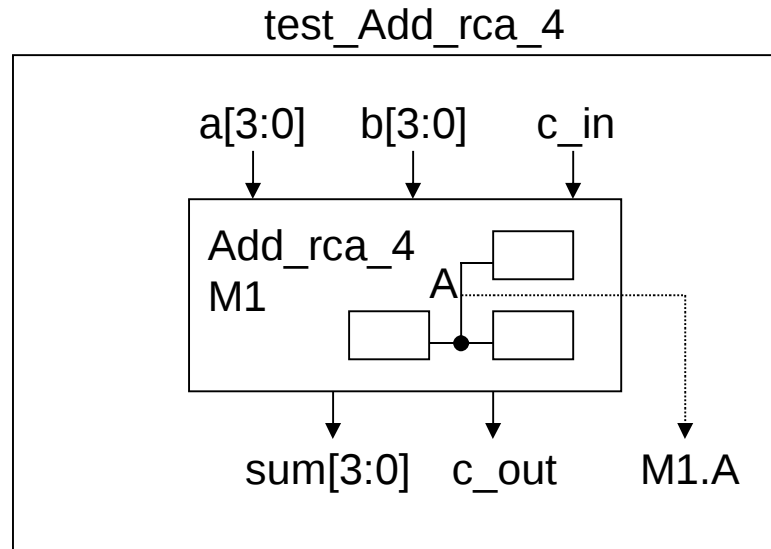
4.3.3 Example of Net and Register



```
module A(O0, O1, I0, I1, I2, I3, Clk);  
  
  input    I0, I1, I2, I3, Clk ;  
  output   O0, O1 ;  
  reg      O0, O1;   
  supply0  g ;  
  wire     a, b, s, c ;  
  
  assign    a = I0 ^ I1 ;  
  assign    b = I2 & I3 ;  
  
  Add_full M0 (s,c,a,b,g) ;  
  
  always@(posedge clk)  
  begin  
    O0 = s ;  
  end  
  
  always@(posedge clk)  
  begin  
    O1 = c ;  
  end  
endmodule
```



4.3.7 Hierarchical Referencing



```

module test_Add_rca_4;

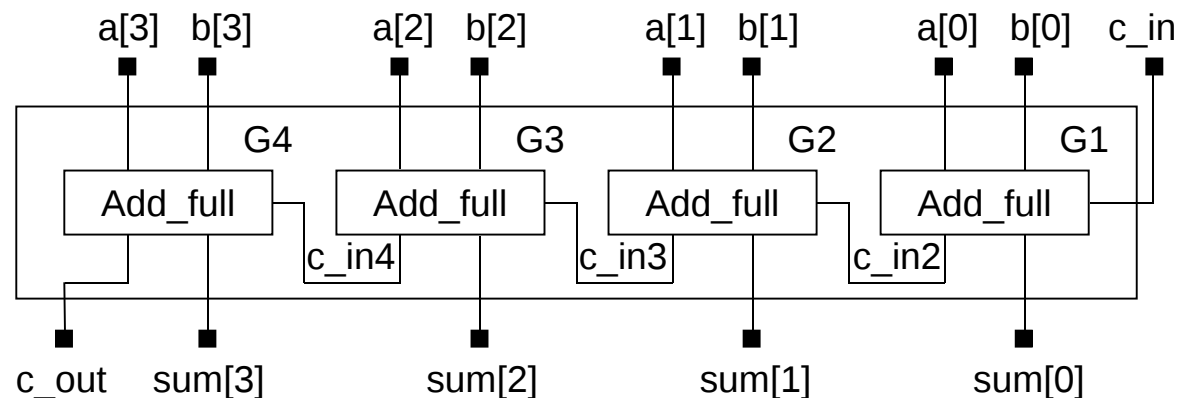
    reg [3:0]      a, b;
    reg           c_in;
    wire [3:0]     sum;
    wire          c_out;

    initial
    begin
        $monitor ($time, &c_out=%b c_in4=%b
                  c_in3=%b c_in2=%b c_in=%b &c_out,
                  M1.c_in4, M1.c_in3, M1.c_in2, c_in);
    end

    Add_rca_4 M1(sum,c_out,a,b,c_in);

endmodule

```





4.6.1 Operator

arithmetic operator	
+	addition
-	subtraction
*	multiplication
/	division
%	modulus

reduction operator	
&	reduction AND
~&	reduction NAND
	reduction OR
~	reduction NOR
^	reduction XOR
~^ ^~	reduction XNOR

bitwise operator	
~	bitwise negation
&	bitwise AND
~&	bitwise NAND
	bitwise OR
~	bitwise NOR
^	bitwise XOR
~^ ^~	bitwise XNOR

logical operator	
!	logical negation
&&	logical and
	logical or
==	logical equality
!=	logical inequality



4.6.1 Operator

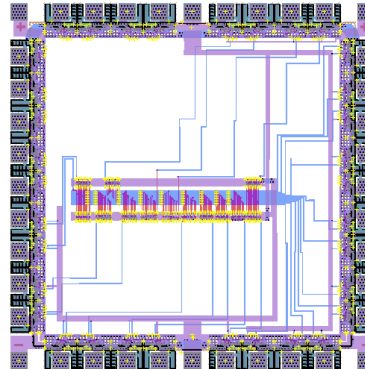
relational operator	
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

shift operator	
<<	left shift
>>	right shift

concatenation operator	
{ A,B }	concatenate A and B
$a[4:0] = \{c_out, c_in[3:0]\};$ The diagram illustrates the concatenation of two 5-bit signals. On the left, there is a single box labeled 'c_out' and a row of four boxes labeled 'c_in[3:0]'. Arrows from each of these five boxes point to a row of five boxes on the right labeled 'a[4:0]'. This shows that 'a[4:0]' is the concatenation of 'c_out' and 'c_in[3:0]'.	

conditional operator	
A ? B : C	if A then B else C
$a = (select == 0) ? b : c;$ The diagram shows a conditional selection. Two input boxes, 'b' and 'c', feed into a selection node. The node has two control inputs: 'select==0' and 'select!=0'. An arrow from the 'select==0' input points to the output box 'a'. This represents the expression 'a = (select == 0) ? b : c;'.	

7. Behavioral Description





7.2.1 Behavioral Statement

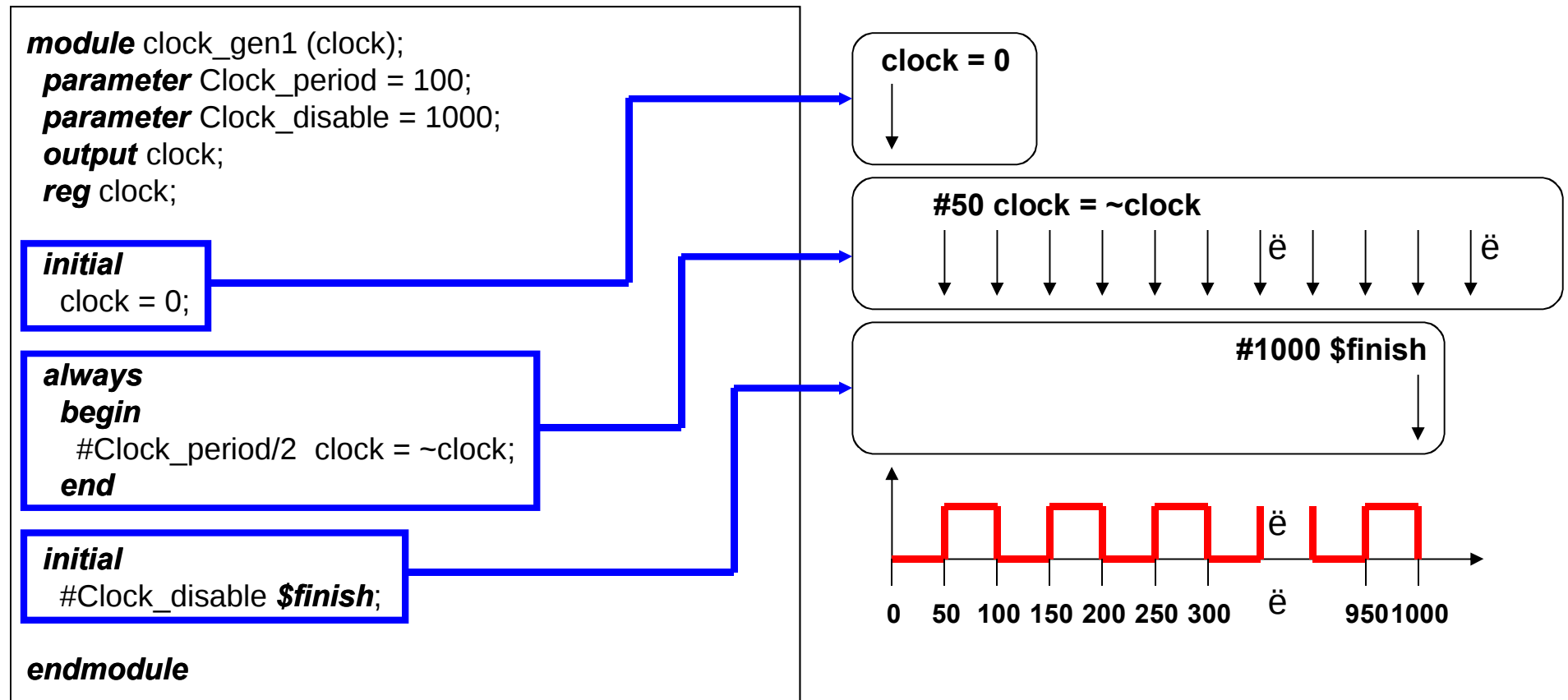
Behavioral Statement

- initial 또는 always로 시작
- initial로 시작되는 behavioral statement는 1번만 수행됨
- always로 시작되는 behavioral statement는 반복 수행됨
- 모든 behavioral statement는 각각 병렬로 수행됨
- behavioral statement 내부의 begin ÷ end 사이에서는 순차적으로 수행됨



7.2.1 Behavioral Statement

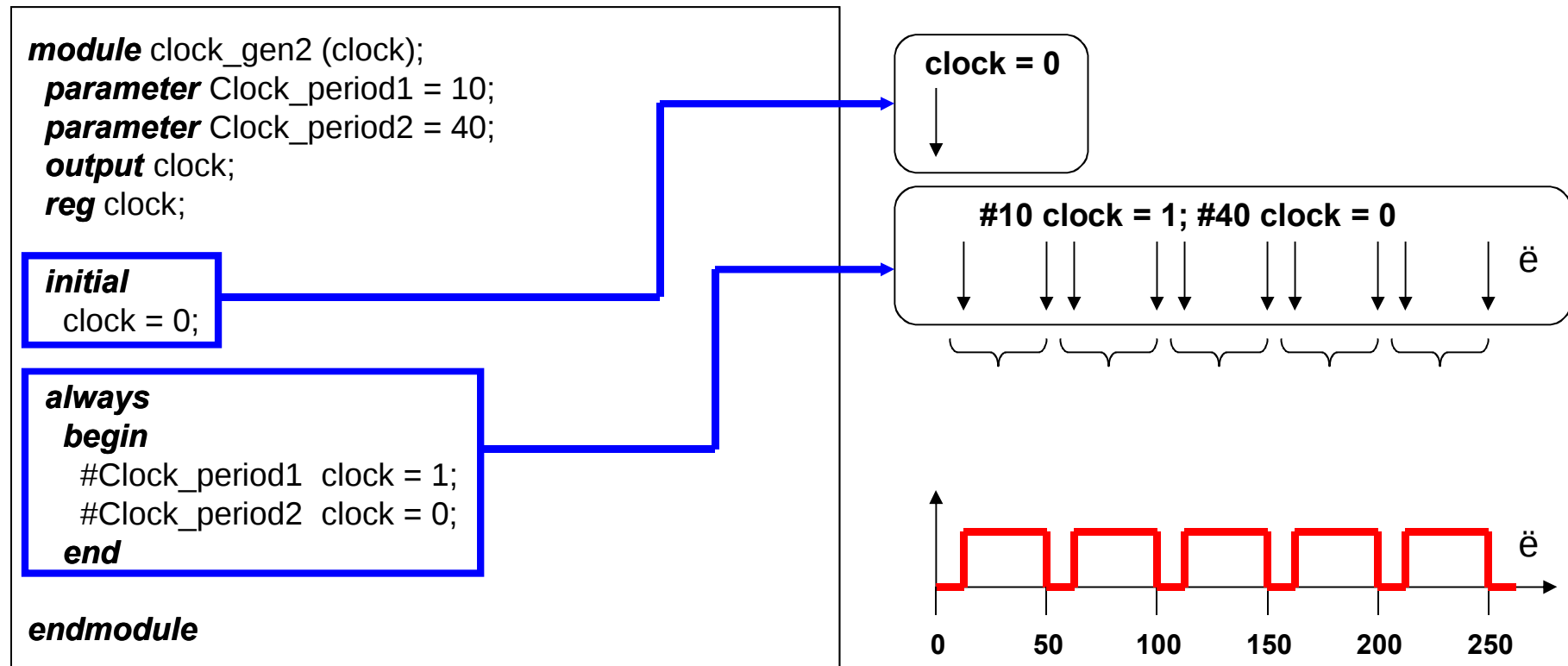
ÇClock Generator Example #1





7.2.1 Behavioral Statement

ÇClock Generator Example #2





7.5.5 WAIT

ÇWAIT

- event: 정해진 **signal**의 값이 변화하거나 정해진 **event**가 발생할 때 수행됨.
- WAIT: 정해진 조건이 만족될 때까지 실행문의 수행을 멈추고 기다림

```
module df1(q,q_bar,clk,set,reset,data);  
  output q,q_bar;  
  input clk,set,reset,data;  
  reg q;  
  
  assign q_bar = ~q;  
  
  always @(posedge clk)  
  begin  
    if (reset==0) q=0;  
    else if (set==0) q=1;  
    else q = data;  
  end  
endmodule
```

(D flip-flop)

```
module latch1(q,q_bar,clk,set,reset,data);  
  output q,q_bar;  
  input clk,set,reset,data;  
  reg q;  
  
  assign q_bar = ~q;  
  
  always  
  begin  
    wait (clk==1);  
    if (reset==0) q=0;  
    else if (set==0) q=1;  
    else q = data;  
  end  
endmodule
```

(D Latch)



7.12.1 ... ? ... : ...

```
module mux(y, sel, a, b);  
  input          sel;  
  input [15:0]    a, b;  
  output [15:0]   y;  
  reg [15:0]      y;  
  
  always @(a or b or sel)  
    y = (sel) ? a : b;  
  
endmodule
```



7.12.2 CASE

```
module mux(y, sel, a, b);  
  input                sel;  
  input [15:0]          a, b;  
  output [15:0]         y;  
  reg [15:0]            y;  
  
  always @(a or b or sel)  
  begin  
    case (sel)  
      1 : y = a;  
      0 : y = b;  
      default y = 1'bx;  
    endcase  
  end  
endmodule
```



7.12.3 IF ... ELSE

```
module mux(y, sel, a, b);  
  input                sel;  
  input [15:0]          a, b;  
  output [15:0]         y;  
  reg [15:0]            y;  
  
  always @(a or b or sel)  
  begin  
    if (sel==1)  
      y = a;  
    else  
      y = b;  
    end  
  endmodule
```



7.12.4 REPEAT

```
module count_1s(a, b);  
  input [15:0]      a;  
  output [4:0]      b;  
  reg [4:0]         b;  
  integer          i;  
  
  initial  
    begin  
      b = 0;  
      i = 0;  
      repeat (16)  
        begin  
          if (a[i]==1)  
            b = b + 1;  
            i = i + 1;  
        end  
      end  
    endmodule
```



7.12.5 FOR

```
module count_1s(a, b);  
  input [15:0]      a;  
  output [4:0]      b;  
  reg [4:0]         b;  
  integer          i;  
  
  initial  
  begin  
    b = 0;  
    for (i=0; i<16; i=i+1)  
      if (a[i]==1)  
        b = b + 1;  
  end  
  
endmodule
```



7.12.6 WHILE

```
module count_1s(a, b);  
  input [15:0]      a;  
  output [4:0]      b;  
  reg [4:0]         b;  
  integer          i;  
  
  initial  
    begin  
      b = 0;  
      i = 0;  
      while (i<16)  
        begin  
          if (a[i]==1)  
            b = b + 1;  
            i = i + 1;  
          end  
        end  
      end  
    endmodule
```



7.12.7 REPEAT and WHILE

```
module count_1s(a, b);  
  input [15:0]      a;  
  output [4:0]      b;  
  reg [4:0]         b;  
  integer          i;  
  
  initial  
  begin  
    b = 0;  
    i = 0;  
    repeat (16)  
    begin  
      if (a[i]==1)  
        b = b + 1;  
      i = i + 1;  
    end  
  end  
endmodule
```

repeat문은 고정된
횟수만큼 loop를
수행하고, while은
주어진 조건을
만족하는 동안
loop를 수행한다.



```
module count_1s(a, b);  
  input [15:0]      a;  
  output [4:0]      b;  
  reg [4:0]         b;  
  integer          i;  
  
  initial  
  begin  
    b = 0;  
    i = 0;  
    while (i<16)  
    begin  
      if (a[i]==1)  
        b = b + 1;  
      i = i + 1;  
    end  
  end  
endmodule
```

System Tasks for Simulation

- `$display` displays values of variables, string, or expressions
 - `$display(ep1, ep2, ..., epn);`
ep1, ep2, ..., epn: quoted strings, variables, expressions.
- `$monitor` monitors a signal when its value changes.
 - `$monitor(ep1, ep2, ..., epn);`
- `$monitoton` enables monitoring operation.
- `$monitotoff` disables monitoring operation.
- `$stop` suspends the simulation.
- `$finish` terminates the simulation.

Time Scale for Simulations

- Time scale compiler directive

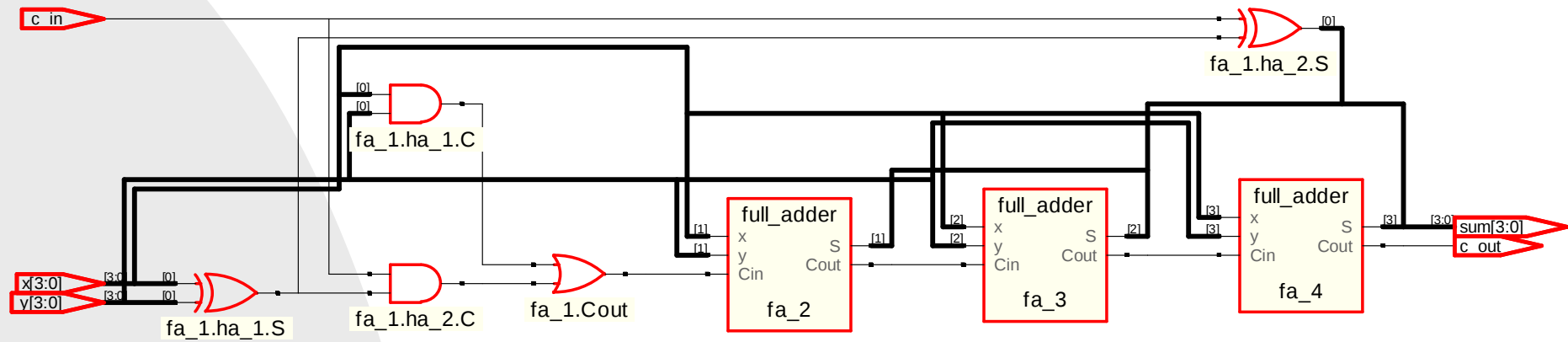
``timescale time_unit / time_precision`

- The `time_precision` must not exceed the `time_unit`.
- For instance, with a timescale 1 ns/1 ps, the delay specification #15 corresponds to 15 ns.
- It uses the same time unit in both behavioral and gate-level modeling.
- For FPGA designs, it is suggested to use `ns` as the time unit.

Modeling and Simulation Example (A 4-bit adder)

```
// Gate-level description of 4-bit adder
module four_bit_adder (x, y, c_in, sum, c_out);
input  [3:0] x, y;
input  c_in;
output [3:0] sum;
output c_out;
wire  C1,C2,C3; // Intermediate carries
// -- four_bit adder body--
// Instantiate the full adder
    full_adder fa_1 (x[0],y[0],c_in,sum[0],C1);
    full_adder fa_2 (x[1],y[1],C1,sum[1],C2);
    full_adder fa_3 (x[2],y[2],C2,sum[2],C3);
    full_adder fa_4 (x[3],y[3],C3,sum[3],c_out);
endmodule
```

Modeling and Simulation Example (A 4-bit adder)



After dissolving one full adder.

Modeling and Simulation Example (A Test Bench)

```
`timescale 1 ns / 100 ps // time unit is in ns.
```

```
module four_bit_adder_tb;
```

```
//Internal signals declarations:
```

```
reg [3:0] x;
```

```
reg [3:0] y;
```

```
reg c_in;
```

```
wire [3:0] sum;
```

```
wire c_out;
```

```
// Unit Under Test port map
```

```
    four_bit_adder UUT (.x(x), .y(y), .c_in(c_in), .sum(sum), .c_out(c_out));
```

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

```
initial begin // for use in post-map and post-par simulations.
```

```
// $sdf_annotate ("four_bit_adder_map.sdf", four_bit_adder);
```

```
// $sdf_annotate ("four_bit_adder_timesim.sdf", four_bit_adder);
```

```
end
```

Tools used before or after Verilog-XL in the design process, such as pre-layout and postlayout tools, produce Standard Delay Format (SDF) files. These files can include timing information for the following: Delays for module io paths, devices, ports, and interconnect delays
Timing checks Timing constraints Scaling, environmental, technology, and user-defined parameters
SDF files are the input for the SDF annotator, which uses the PLI as an interface to backannotate timing information into Verilog HDL designs.

Modeling and Simulation Example (A Test Bench)

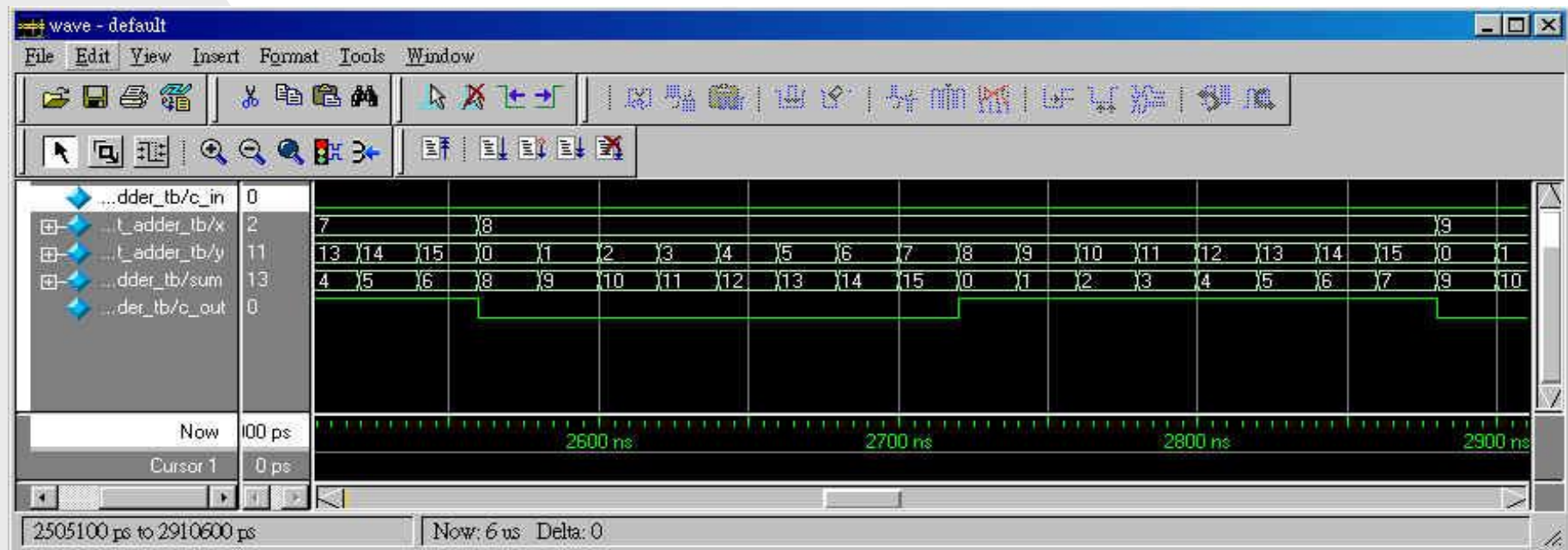
```
initial
  for (i = 0; i <= 255; i = i + 1) begin
    x[3:0] = i[7:4]; y[3:0] = i[3:0]; c_in = 1'b0;
    #20 ; end
initial #6000 $finish;
initial
  $monitor($realtime, "ns %h %h %h %h", x, y, c_in, {c_out, sum});
endmodule
```

Modeling and Simulation Example (Simulation Results)

```
# 0ns 0 0 0 00
# 20ns 0 1 0 01
# 40ns 0 2 0 02
# 60ns 0 3 0 03
# 80ns 0 4 0 04
# 100ns 0 5 0 05
# 120ns 0 6 0 06
# 140ns 0 7 0 07
# 160ns 0 8 0 08
# 180ns 0 9 0 09
# 200ns 0 a 0 0a
# 220ns 0 b 0 0b
# 240ns 0 c 0 0c
# 260ns 0 d 0 0d
```

```
# 280ns 0 e 0 0e
# 300ns 0 f 0 0f
# 320ns 1 0 0 01
# 340ns 1 1 0 02
# 360ns 1 2 0 03
# 380ns 1 3 0 04
# 400ns 1 4 0 05
# 420ns 1 5 0 06
# 440ns 1 6 0 07
# 460ns 1 7 0 08
# 480ns 1 8 0 09
# 500ns 1 9 0 0a
# 520ns 1 a 0 0b
# 540ns 1 b 0 0c
```

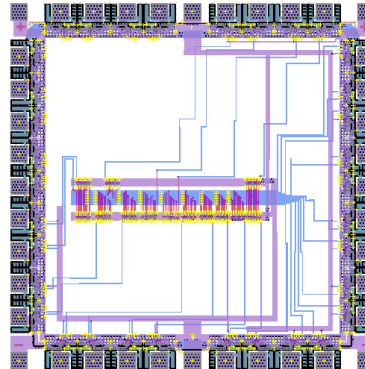
Modeling and Simulation Example



Verilog Example

(또 한개의 파일의 설명 후
Desktop Calculator 중점
Project 추진)

실험 1. Adder/Subtractor





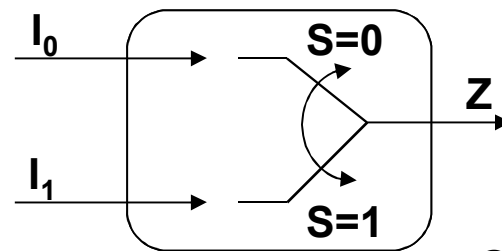
1.1 MUX

ÇMultiplexer

2^n data inputs, n control inputs, 1 output

used to connect 2^n points to a single point

control signals = binary index of input connected to output



Choose I_0 if $S = 0$

Choose I_1 if $S = 1$



1.1 MUX

Ç2 to 1 Multiplexer (2:1 MUX)

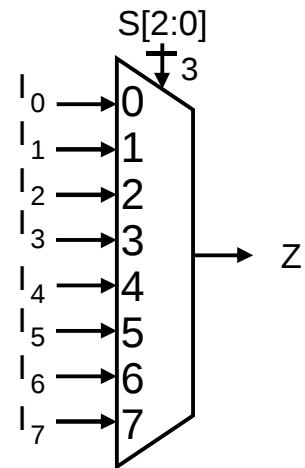
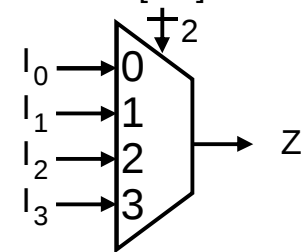
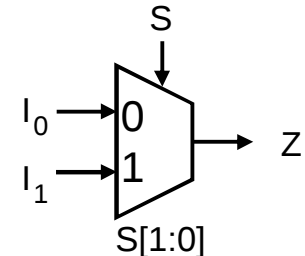
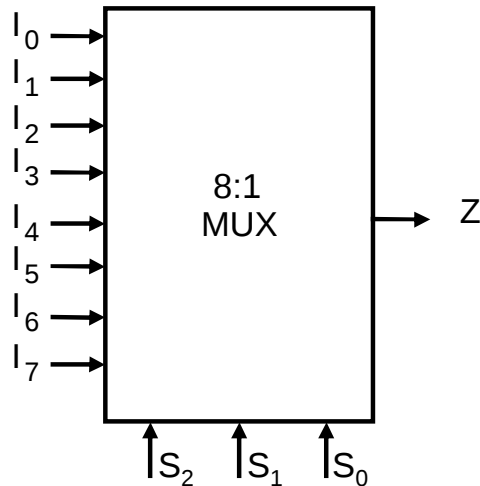
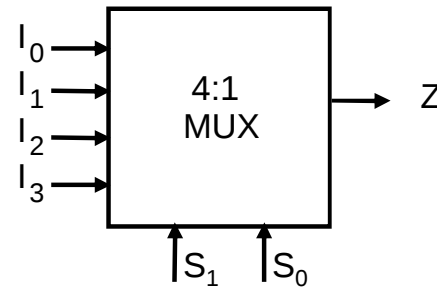
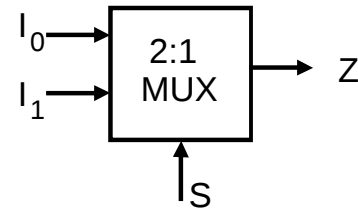
$$Z = S' I_0 + S I_1$$

Ç4 to 1 Multiplexer (4:1 MUX)

$$Z = S_1' \cdot S_0' \cdot I_0 + S_1' \cdot S_0 \cdot I_1 + S_1 \cdot S_0' \cdot I_2 + S_1 \cdot S_0 \cdot I_3$$

Ç8 to 1 Multiplexer (8:1 MUX)

$$Z = S_2' \cdot S_1' \cdot S_0' \cdot I_0 + S_2' \cdot S_1' \cdot S_0 \cdot I_1 + S_2' \cdot S_1 \cdot S_0' \cdot I_2 + S_2' \cdot S_1 \cdot S_0 \cdot I_3 + S_2 \cdot S_1' \cdot S_0' \cdot I_4 + S_2 \cdot S_1' \cdot S_0 \cdot I_5 + S_2 \cdot S_1 \cdot S_0' \cdot I_6 + S_2 \cdot S_1 \cdot S_0 \cdot I_7$$



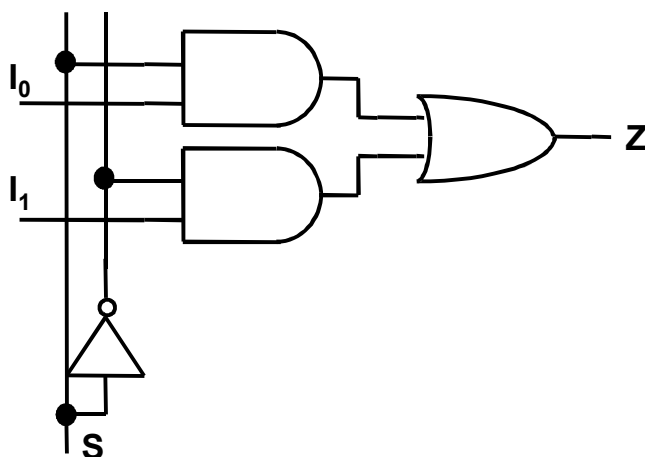
1.1 MUX



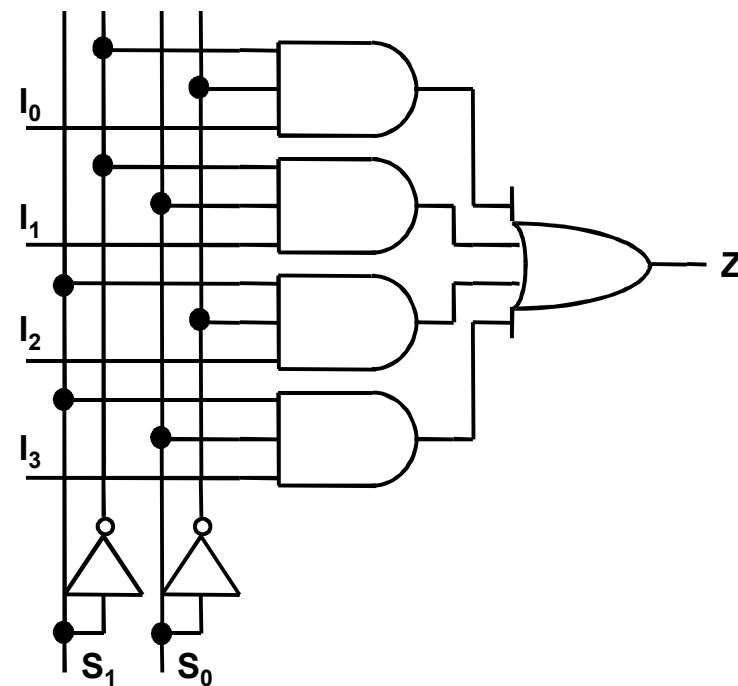
1.1 MUX

ÇGate implementation of MUX

S	AND input	Z
0	$\overline{S}$	I0
1	S	I1



S1	S0	AND input	Z
0	0	$\overline{S1} \overline{S0}$	I0
0	1	$\overline{S1} S0$	I1
1	0	$S1 \overline{S0}$	I2
1	1	$S1 S0$	I3

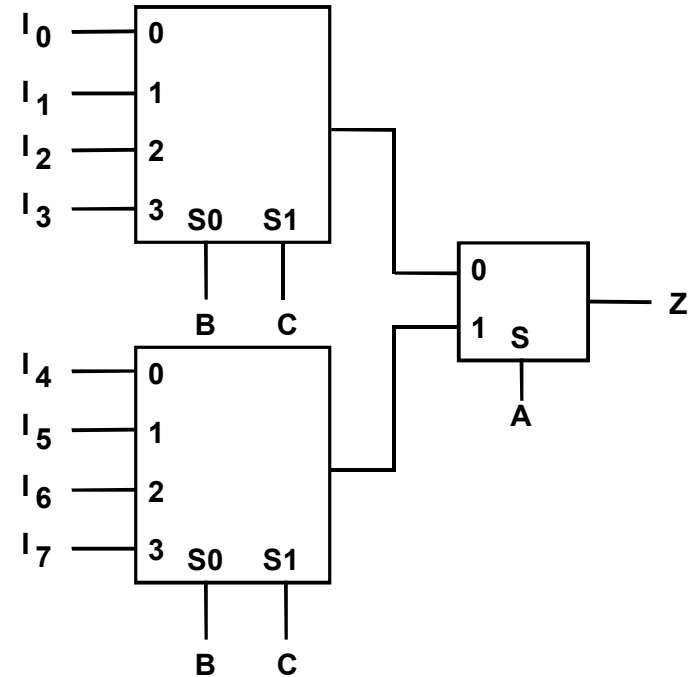
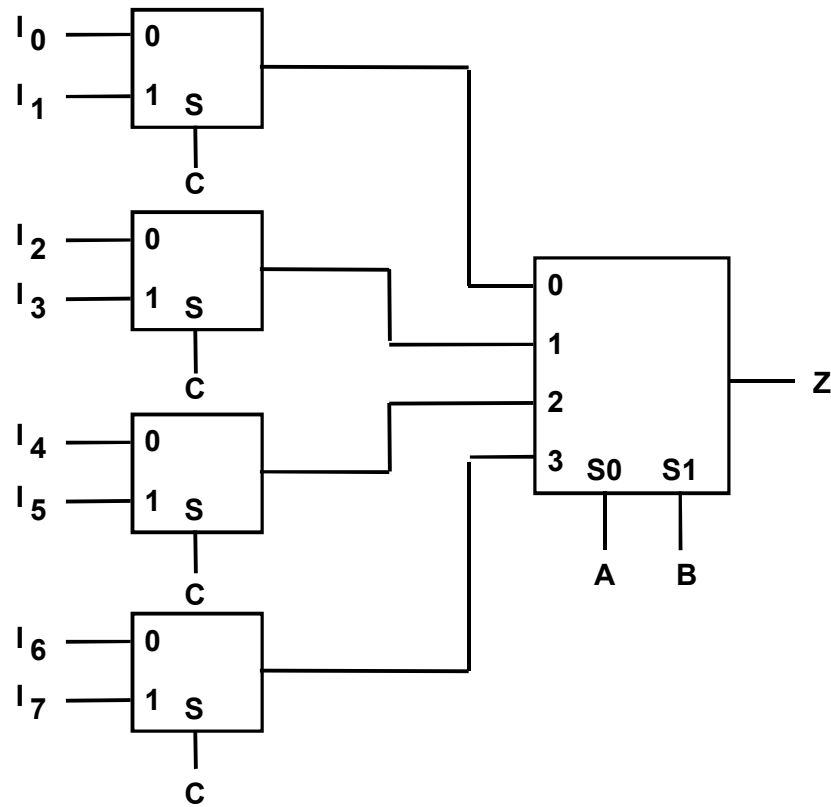


1.1 MUX



1.1 MUX

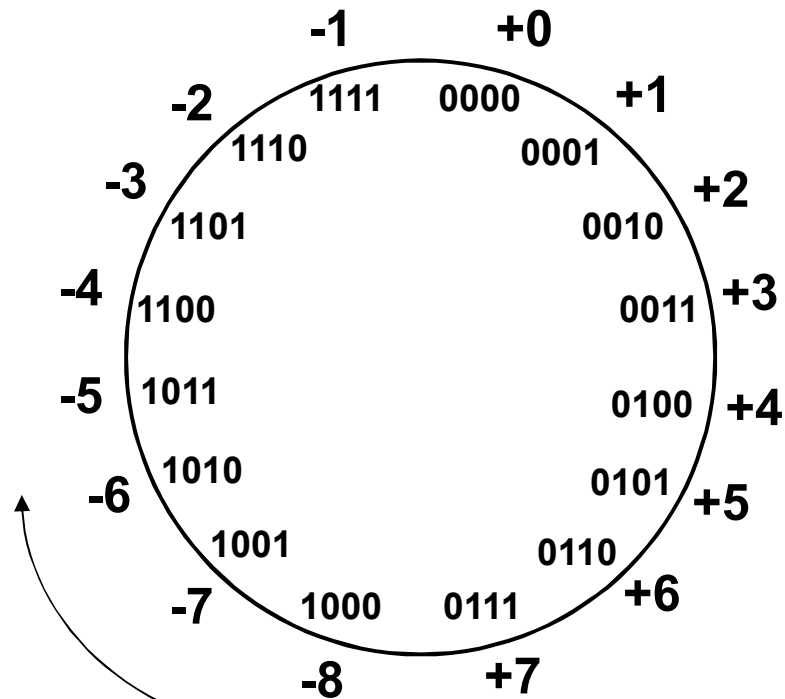
Ç8:1 MUX from 4:1 MUX and 2:1 MUX



1.1 MUX



1.2 2's Complement



*like 1's comp
except shifted
one position
clockwise*

2's complement = bitwise complement + 1

0011 $\rightarrow$ 1100 + 1 $\rightarrow$ 1101 (representation of -3)

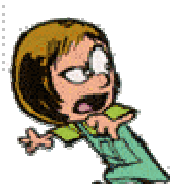
1101 $\rightarrow$ 0010 + 1 $\rightarrow$ 0011 (representation of 3)

+
0 011 = + 3

1 101 = - 3
-

Number range for n bits: $-2^{n-1} \sim (2^{n-1} - 1)$

- Only one representation of 0
- One more negative number than positive number
- Easy addition / subtraction



1.2 Addition and Subtraction

2's Complement

2's complement =
bitwise complement + 1

0111 $\rightarrow$ 1000 + 1 $\rightarrow$ 1001
(representation of -7)

1001 $\rightarrow$ 0110 + 1 $\rightarrow$ 0111
(representation of 7)

덧셈

① 부호없는 덧셈을 수행한다.

뺄셈

① **2's complement**를 사용하여
덧셈으로 변환한 후,
② 부호없는 덧셈을 수행한다.

$$\begin{array}{r} 4 \quad 0100 \\ + (+3) \quad 0011 \\ \hline 7 \quad 0111 \end{array}$$

$$\begin{array}{r} -4 \quad 1100 \\ + (-3) \quad 1101 \\ \hline -7 \quad 11001 \end{array}$$

$$\begin{array}{r} 4 \quad 0100 \\ - (+3) \quad 1101 \\ \hline 1 \quad 10001 \end{array}$$

$$\begin{array}{r} -4 \quad 1100 \\ - (-3) \quad 0011 \\ \hline -1 \quad 1111 \end{array}$$

2's complement는 덧셈, 뺄셈이 간단하기 때문에 대부분의 디지털 회로에 사용된다.



1.2 Addition and Subtraction

$$\begin{array}{r} 4 \\ + 3 \\ \hline 7 \end{array} \quad \underline{\hspace{2cm}}$$

$$\begin{array}{r} -4 \\ + (-3) \\ \hline -7 \end{array} \quad \underline{\hspace{2cm}}$$

$$\begin{array}{r} 4 \\ - (+3) \\ \hline 1 \end{array} \quad \underline{\hspace{2cm}}$$

$$\begin{array}{r} -4 \\ - (-3) \\ \hline -1 \end{array} \quad \underline{\hspace{2cm}}$$

$$\begin{array}{r} 5 \\ + 2 \\ \hline 7 \end{array} \quad \underline{\hspace{2cm}}$$

$$\begin{array}{r} -5 \\ + (-2) \\ \hline -7 \end{array} \quad \underline{\hspace{2cm}}$$

$$\begin{array}{r} 5 \\ - (+2) \\ \hline 3 \end{array} \quad \underline{\hspace{2cm}}$$

$$\begin{array}{r} -5 \\ - (-2) \\ \hline -3 \end{array} \quad \underline{\hspace{2cm}}$$

$$\begin{array}{r} 4 \\ + 2 \\ \hline 6 \end{array} \quad \underline{\hspace{2cm}}$$

$$\begin{array}{r} -4 \\ + (-2) \\ \hline -6 \end{array} \quad \underline{\hspace{2cm}}$$

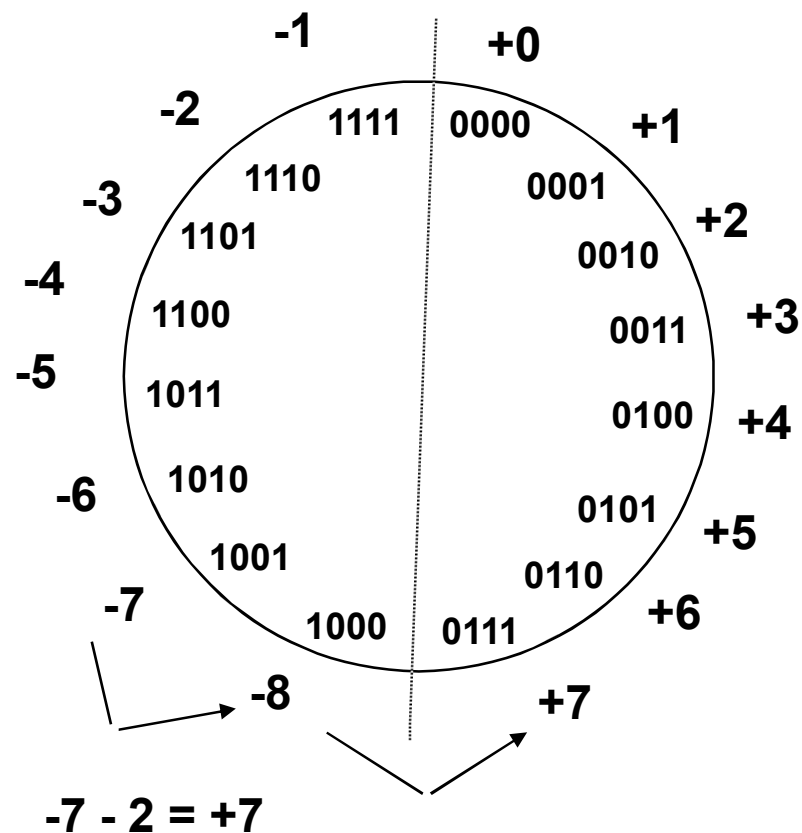
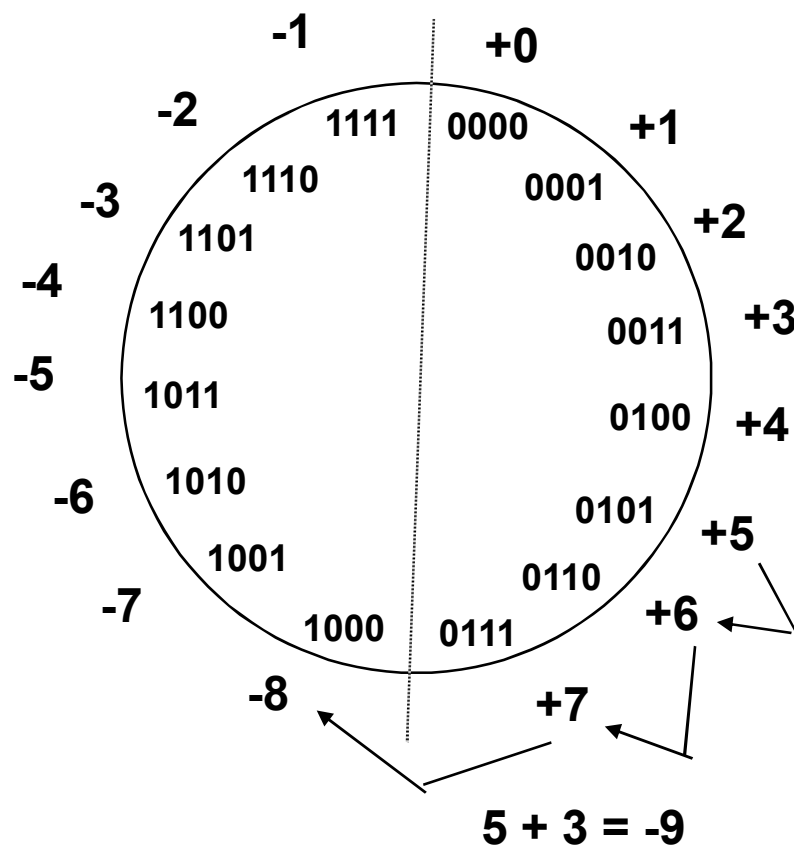
$$\begin{array}{r} 4 \\ - (+2) \\ \hline 2 \end{array} \quad \underline{\hspace{2cm}}$$

$$\begin{array}{r} -4 \\ - (-2) \\ \hline -2 \end{array} \quad \underline{\hspace{2cm}}$$



1.2 Overflow

Add two positive numbers to get a negative number
or two negative numbers to get a positive number

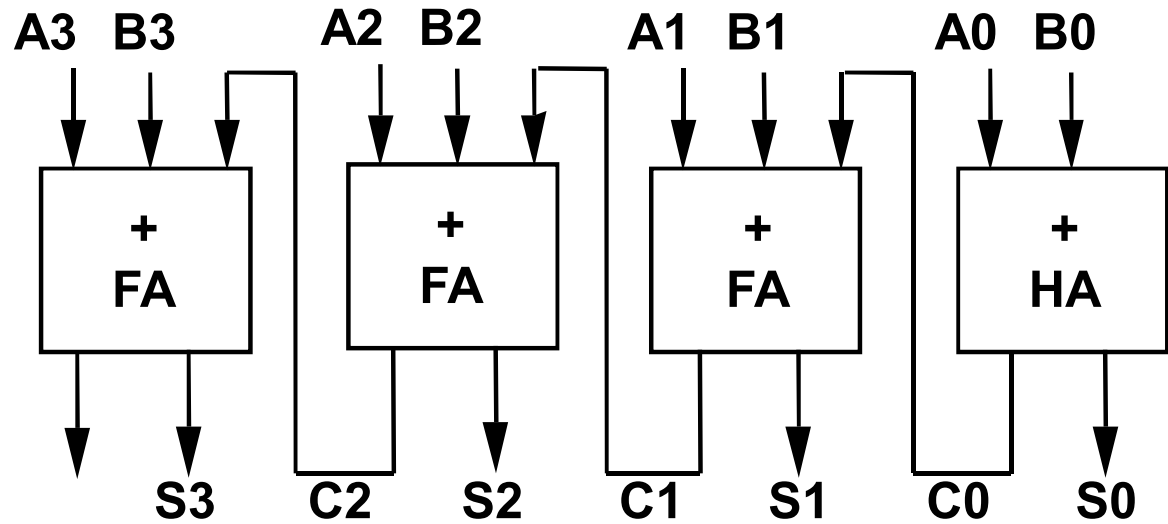
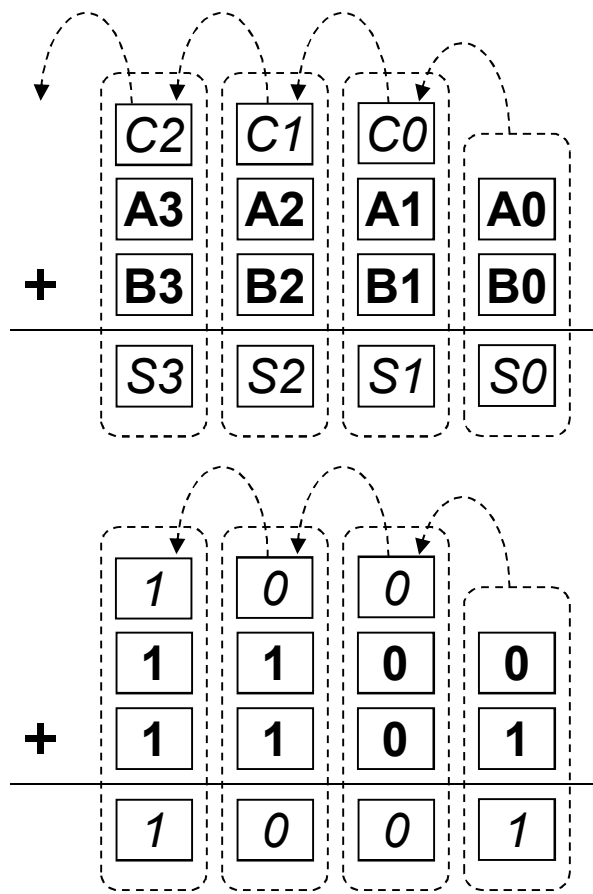


1.2 Addition / Subtraction



1.3 Adder

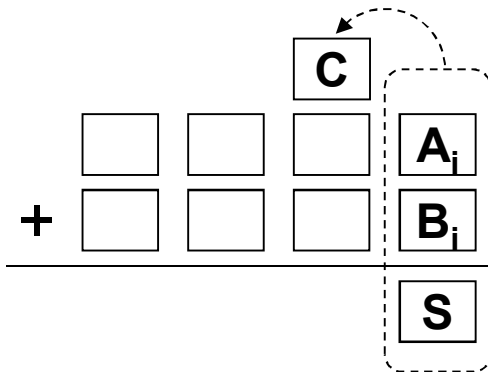
Cascaded Multi-bit Adder





1.3 Half Adder

Half Adder



A _i	B _i	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

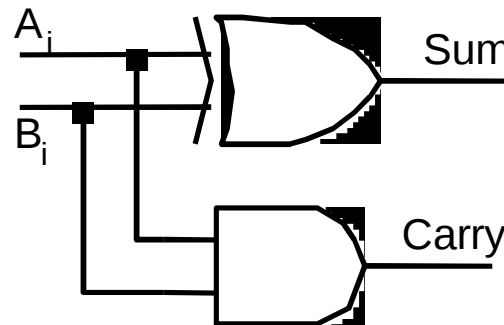
A _i \ B _i	0	1
0	0	1
1	1	0

$$\text{Sum} = \overline{A_i} B_i + A_i \overline{B_i}$$

$$= A_i \oplus B_i$$

A _i \ B _i	0	1
0	0	0
1	0	1

$$\text{Carry} = A_i B_i$$

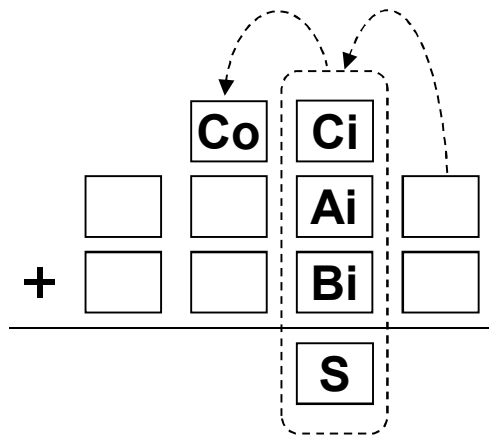


Half-adder Schematic



1.3 Full Adder

Full Adder



A	B	Ci	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

S

	A B			
Ci	00	01	11	10
0	0	1	0	1
1	1	0	1	0

Co

	A B			
Ci	00	01	11	10
0	0	0	1	0
1	0	1	1	1

$$S = Ci \text{ xor } A \text{ xor } B$$

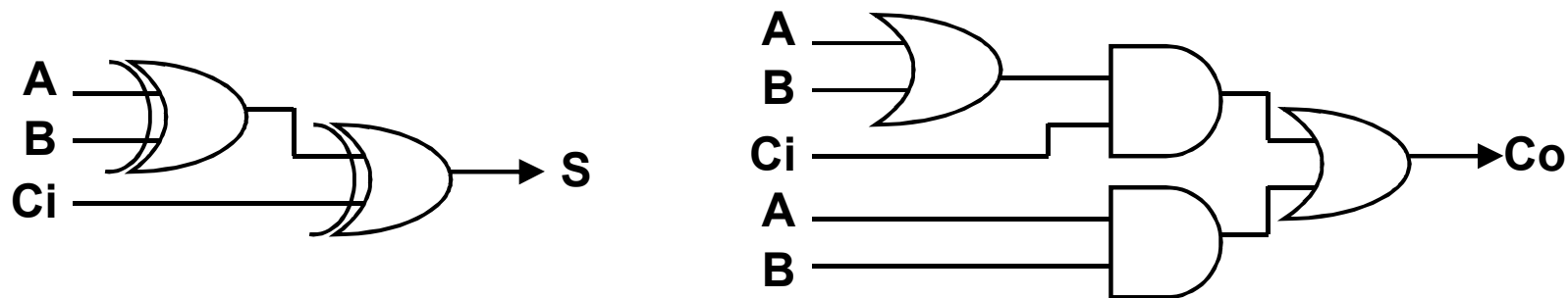
$$Co = B Ci + A Ci + A B = Ci (A + B) + A B$$

1.3 Adder / Subtractor

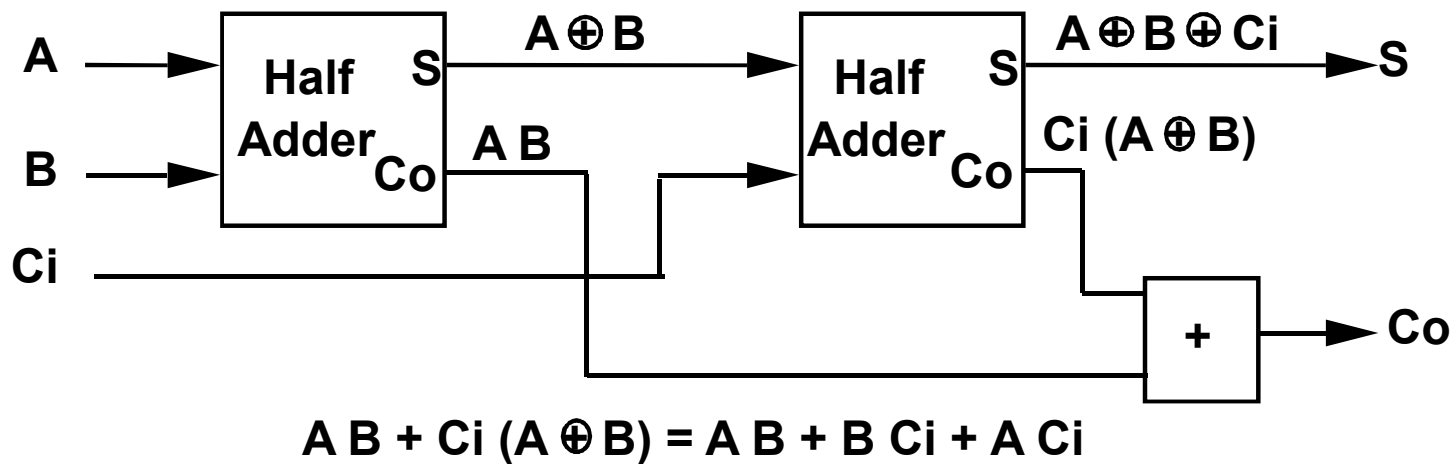


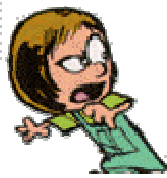
1.3 Full Adder

Standard Approach: 6 Gates



Alternative Implementation: 5 Gates





1.3 Adder / Subtractor

Bitwise complement of B

$$A - B = A + (-B) = A + (\overline{B} + 1) = A + \overline{B} + 1$$

2's complement of B



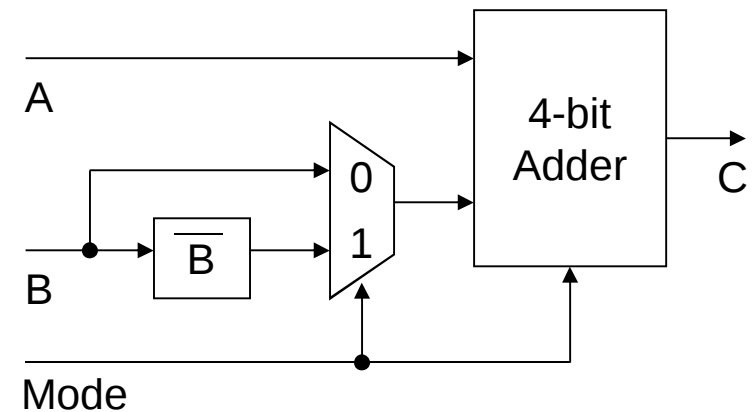
(Addition) $C = A + B$

(Subtraction) $C = A + \overline{B} + 1$



(Addition) $C = A + B + \text{Mode}$ if Mode=0

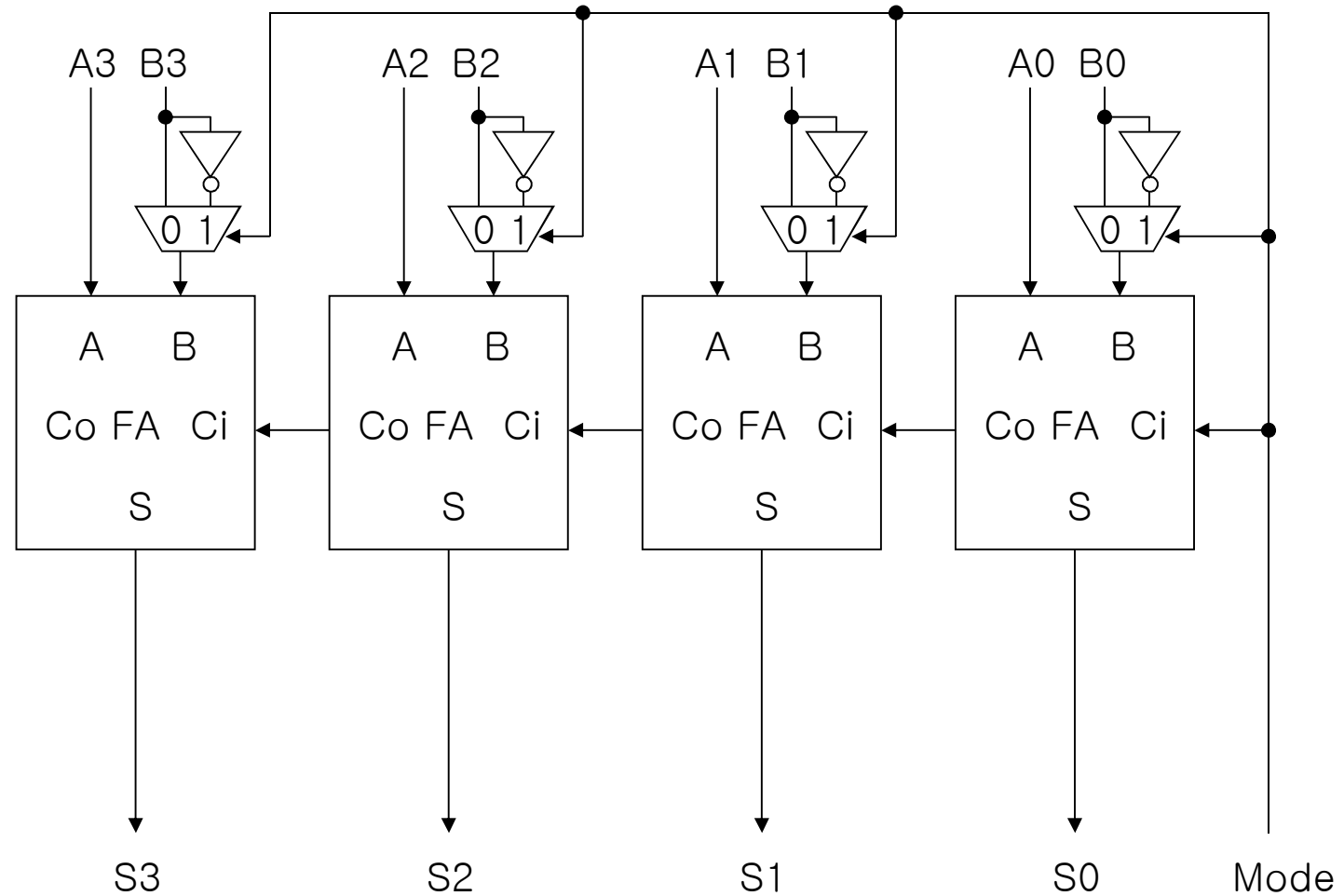
(Subtraction) $C = A + \overline{B} + \text{Mode}$ if Mode=1



1.3 Adder / Subtractor



1.3 Adder / Subtractor

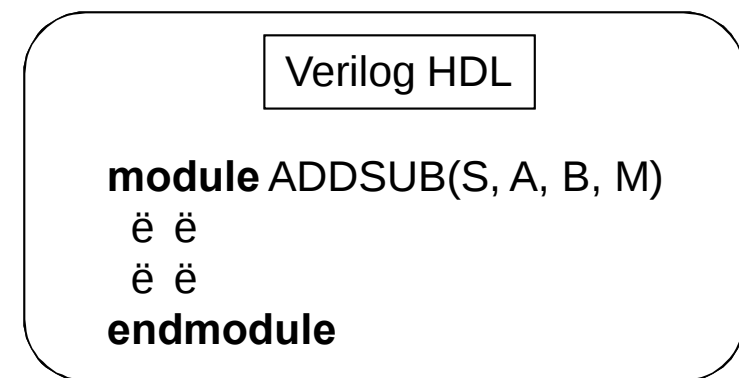
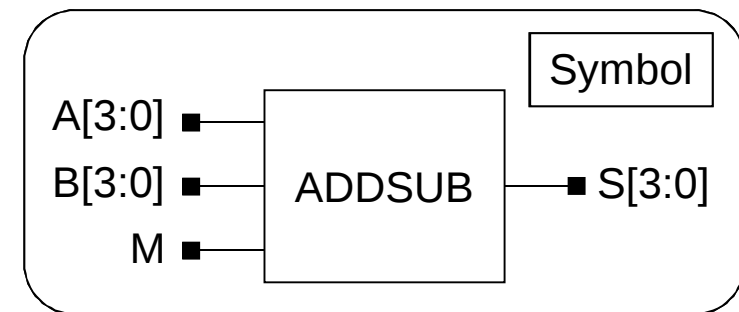
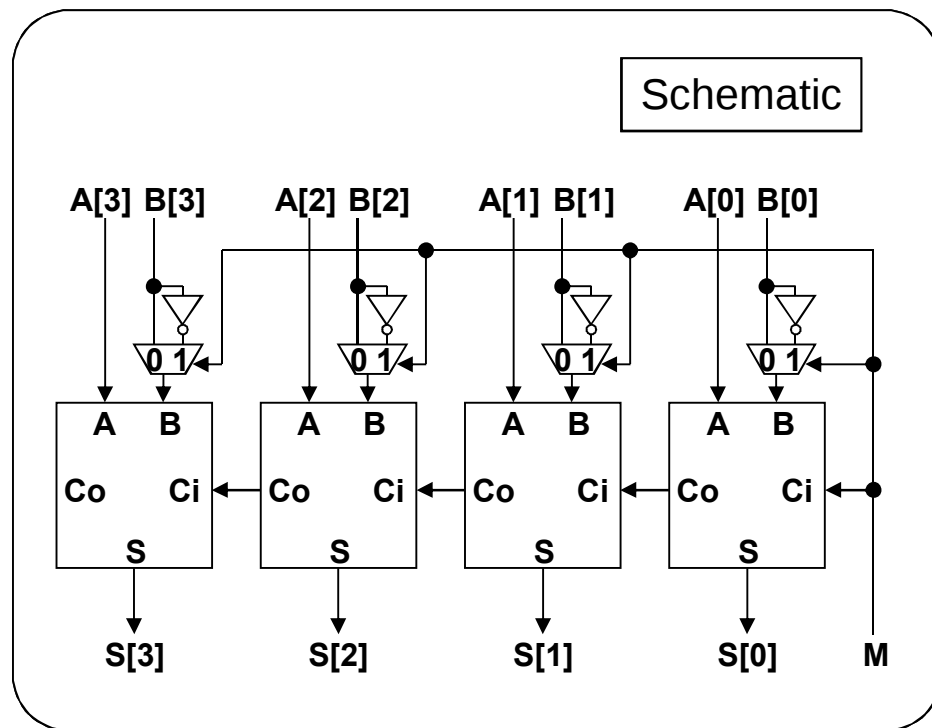


1.3 Adder / Subtractor



1.4 Adder/Subtractor 실험

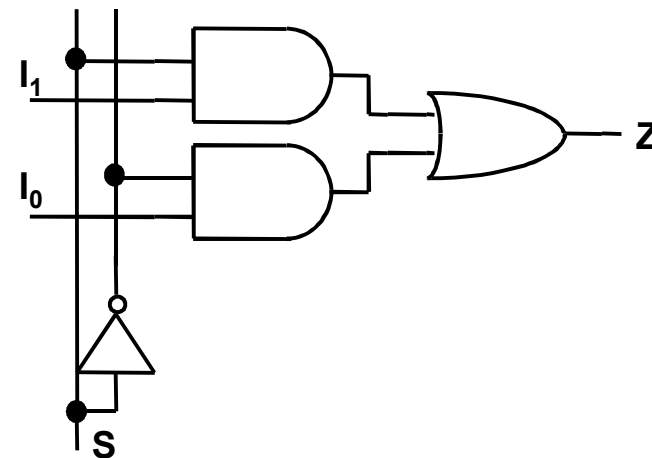
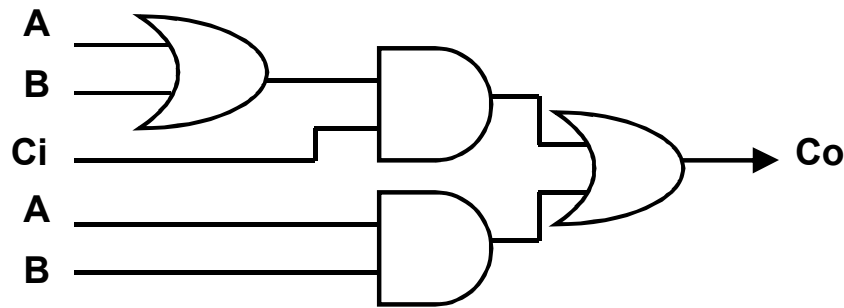
(1) p.15의 회로를 Verilog HDL 모듈로 표현하세요.





1.4 Adder/Subtractor 실험

(2) p.16에 사용된 full adder와 mux를 p.13과 p.4를 참조하여 Verilog HDL 모듈로 표현하세요.





1.4 Adder/Subtractor 실험

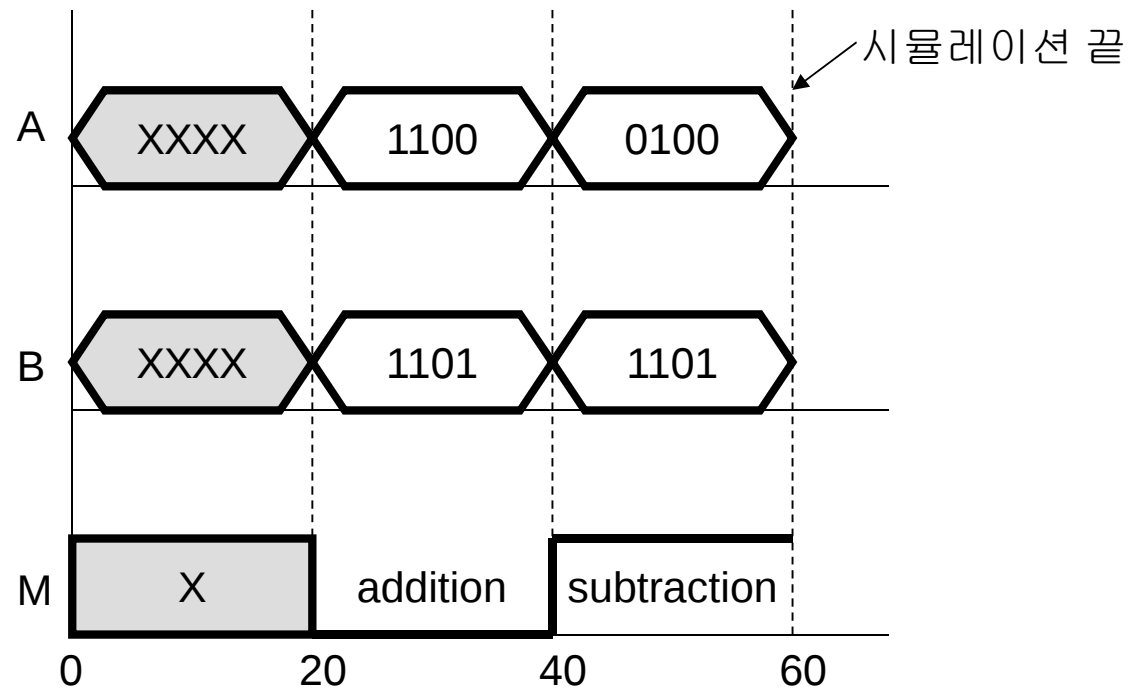
(3) 다음과 같은 연산을 검증할 수 있도록 A[3:0], B[3:0], M을 생성하는 stimulus generator를 Verilog HDL 코드로 표현하세요.

(가)

$$\begin{array}{r} -4 \quad 1100 \\ + (-3) \quad 1101 \\ \hline -7 \quad 11001 \end{array}$$

(나)

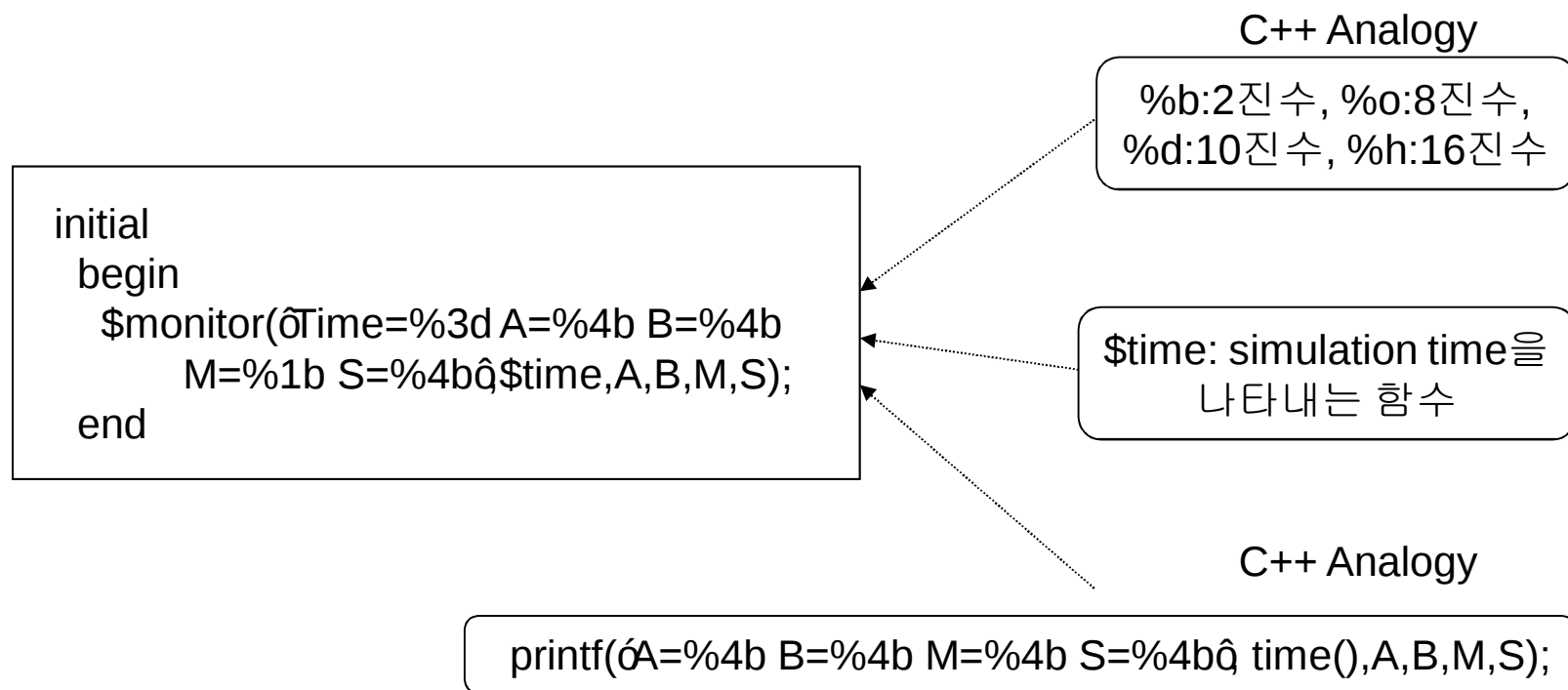
$$\begin{array}{r} 4 \quad 0100 \\ - (-3) \quad 1101 \\ \hline 7 \quad 00111 \end{array}$$





1.4 Adder/Subtractor 실험

(4) 시간 \$time, 입력 신호 A, B, M, 출력 신호 S를 측정하는 response monitor를 Verilog HDL 코드로 표현하세요.



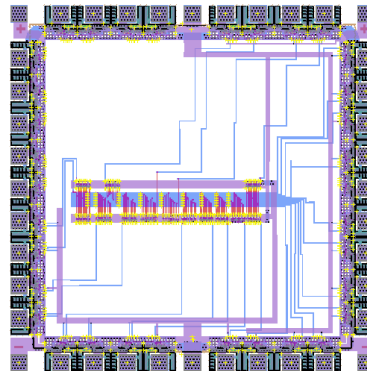


1.4 Adder/Subtractor 실험

(5) 앞의 (1)-(4)에서 작성한 Verilog HDL 코드를 조합하여, (1)의 회로를 테스트하는 DUTB의 Verilog HDL 코드를 작성하세요.

(6) SILOS 프로그램을 사용하여 (5)에서 작성한 DUTB를 시뮬레이션하고, 출력 신호 S가 시간에 따라 어떻게 변하는지 관찰하세요.

실험 2. Decoder





2.1 Decoder

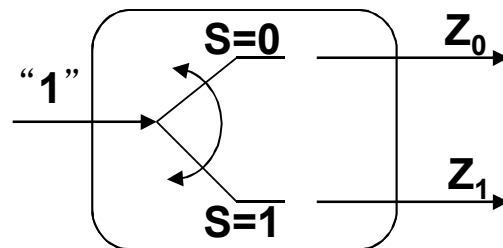
ÇDecoder

n control inputs, 2^n outputs

used to make one point to “1” out of 2^n points

used as a “minterm generator”

control signals = binary index of minterm



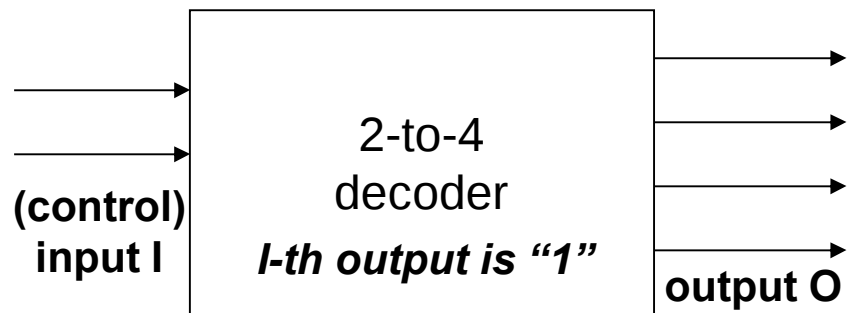
$Z_0 = \text{“1”}$ if $S = 0$

$Z_1 = \text{“1”}$ if $S = 1$

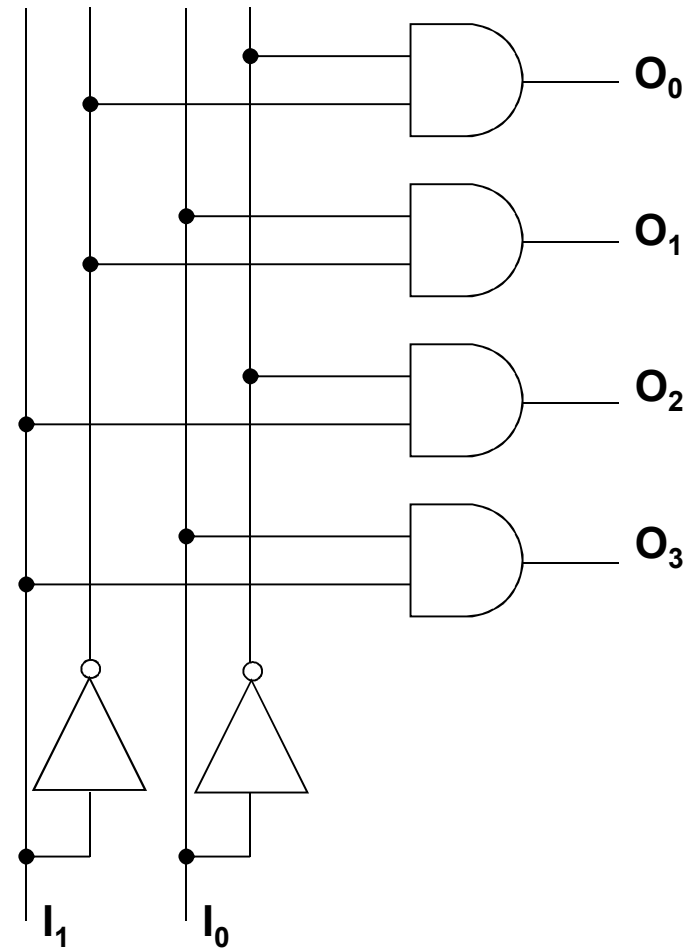


2.1 Decoder

ÇDecoder



I	I_1	I_0	O_0	O_1	O_2	O_3
0	0	0	1	0	0	0
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	1	1	0	0	0	1



2.1 Decoder



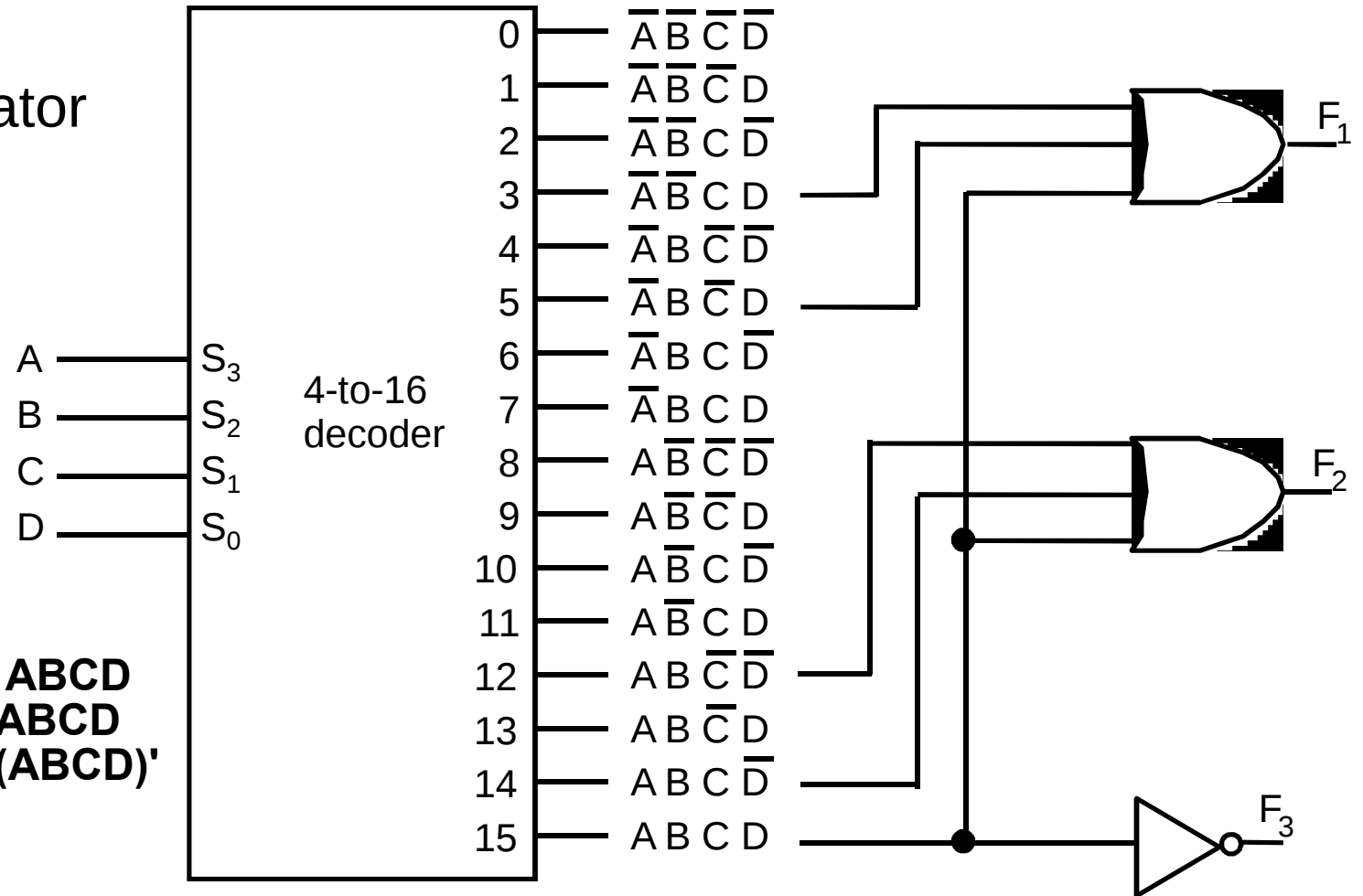
2.1 Decoder

ÇDecoder as a
minterm generator

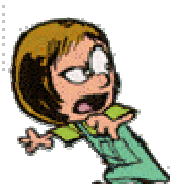
$$F_1 = A'B'CD + A'BC'D + ABCD$$

$$F_2 = ABC'D' + ABCD' + ABCD$$

$$F_3 = A' + B' + C' + D' = (ABCD)'$$

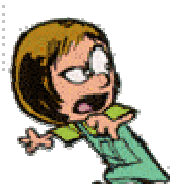


2.1 Decoder



2.2 CASE 문

```
module mux(y, sel, a, b);  
  input                sel;  
  input [15:0]          a, b;  
  output [15:0]         y;  
  reg [15:0]            y;  
  
  always @(a or b or sel)  
  begin  
    case (sel)  
      1 : y = a;  
      0 : y = b;  
      default y = 1'bx;  
    endcase  
  end  
endmodule
```



2.2 CASE 문

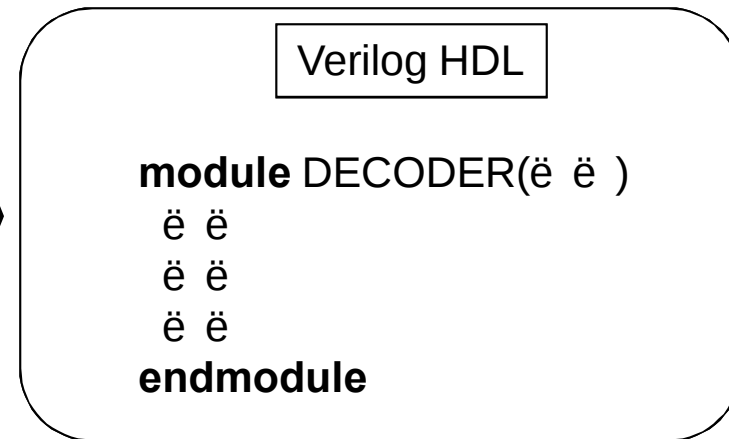
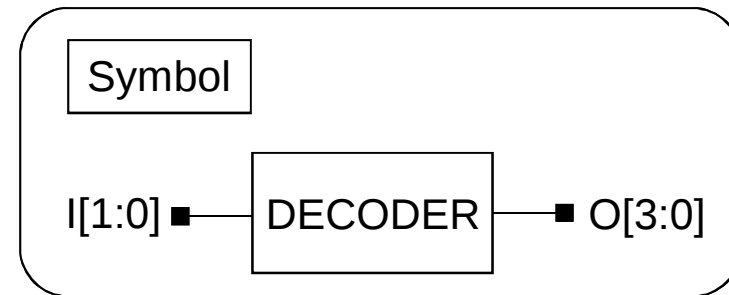
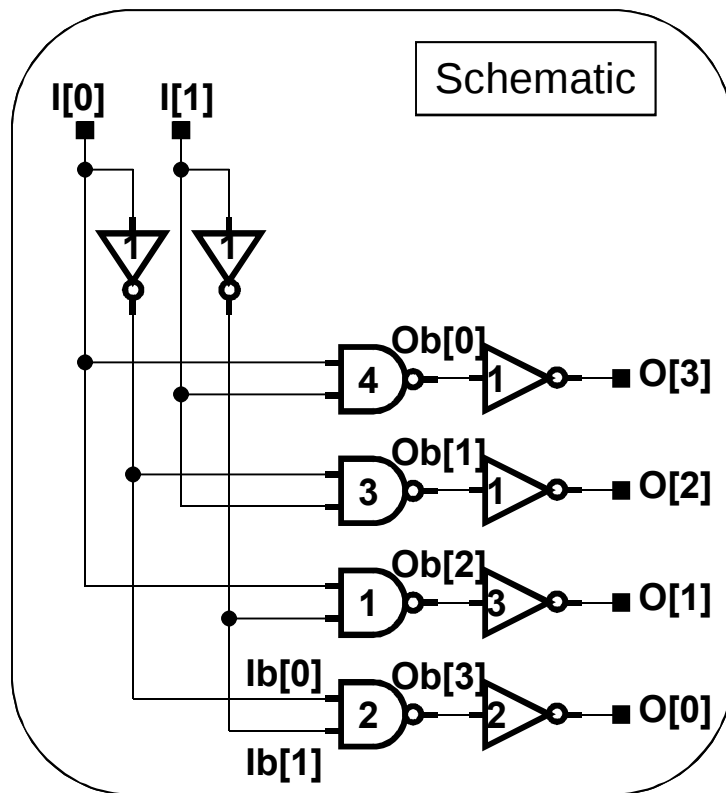
I	I ₁	I ₀	O ₀	O ₁	O ₂	O ₃
0	0	0	1	0	0	0
1	0	1	0	1	0	0
2	1	0	0	0	1	0
3	1	1	0	0	0	1

```
module dec2to4(o, i);  
  input [1:0]      i;  
  output [3:0]     o;  
  reg [3:0]        o;  
  
  always @(i)  
  begin  
    case (i)  
      2'b00 : o = 4'b0001;  
      2'b01 : o = 4'b0010;  
      2'b10 : o = 4'b0100;  
      2'b11 : o = 4'b1000;  
      default y = 4'bxxxx;  
    endcase  
  end  
endmodule
```



2.3 Decoder 실험 (1)

(1) p.23의 회로를 아래 그림과 같이 수정한 후, Verilog HDL 모듈로 표현하세요.

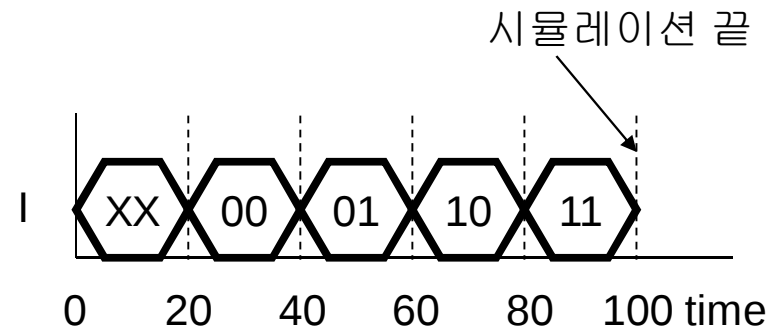




2.3 Decoder 실험 (1)

(2) 2-to-4 Decoder를 검증할 수 있도록 다음과 같은 파형의 입력 신호 I를 생성하는 stimulus generator를 Verilog HDL 코드로 표현하세요.

I[1]	I[0]	O[0]	O[1]	O[2]	O[3]
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1





2.3 Decoder 실험 (1)

(3) 시간 \$time, 입력 신호 I, 출력 신호 O를 측정하는 response monitor를 Verilog HDL 코드로 표현하세요.

(4) 앞의 (1)-(3)에서 작성한 Verilog HDL 코드를 조합하여, (1)의 회로를 테스트하는 DUTB의 Verilog HDL 코드를 작성하세요.

(5) SILOS 프로그램을 사용하여 (4)에서 작성한 DUTB를 시뮬레이션하고, 출력 신호 O가 시간에 따라 어떻게 변하는지 관찰하세요.



2.4 Decoder 실험 (2)

- (1) p.26을 참조하여 2-to-4 Decoder를 CASE문을 사용하여 Verilog HDL 모듈로 표현하세요. 이때, 입력이 들어가서 출력이 나오기까지의 Delay는 50이라고 가정하세요.
- (2) p.28을 참조하여 2-to-4 Decoder를 검증할 수 있는 입력 신호 I를 생성하는 stimulus generator를 Verilog HDL 코드로 표현하세요.



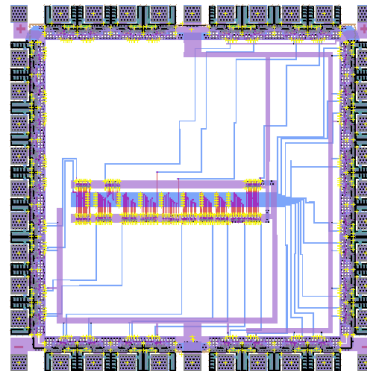
2.4 Decoder 실험 (2)

(3) 시간 \$time, 입력 신호 I, 출력 신호 O를 측정하는 response monitor를 Verilog HDL 코드로 표현하세요.

(4) 앞의 (1)-(3)에서 작성한 Verilog HDL 코드를 조합하여, (1)의 회로를 테스트하는 DUTB의 Verilog HDL 코드를 작성하세요.

(5) SILOS 프로그램을 사용하여 (4)에서 작성한 DUTB를 시뮬레이션하고, 출력 신호 O가 시간에 따라 어떻게 변하는지 관찰하세요.

실험 3. BCD Adder





3.1 BCD (Binary-Coded Decimal)

숫자	BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

합	BCD
0	0 0000
1	0 0001
2	0 0010
3	0 0011
4	0 0100
5	0 0101
6	0 0110
7	0 0111
8	0 1000
9	0 1001
10	1 0000
11	1 0001
12	1 0010
13	1 0011
14	1 0100
15	1 0101
16	1 0110
17	1 0111
18	1 1000

	숫자	BCD
X	1	0001
+Y	2	0010
Z		0 0011

	숫자	BCD
X	7	0111
+Y	3	0011
Z	10	1 0000

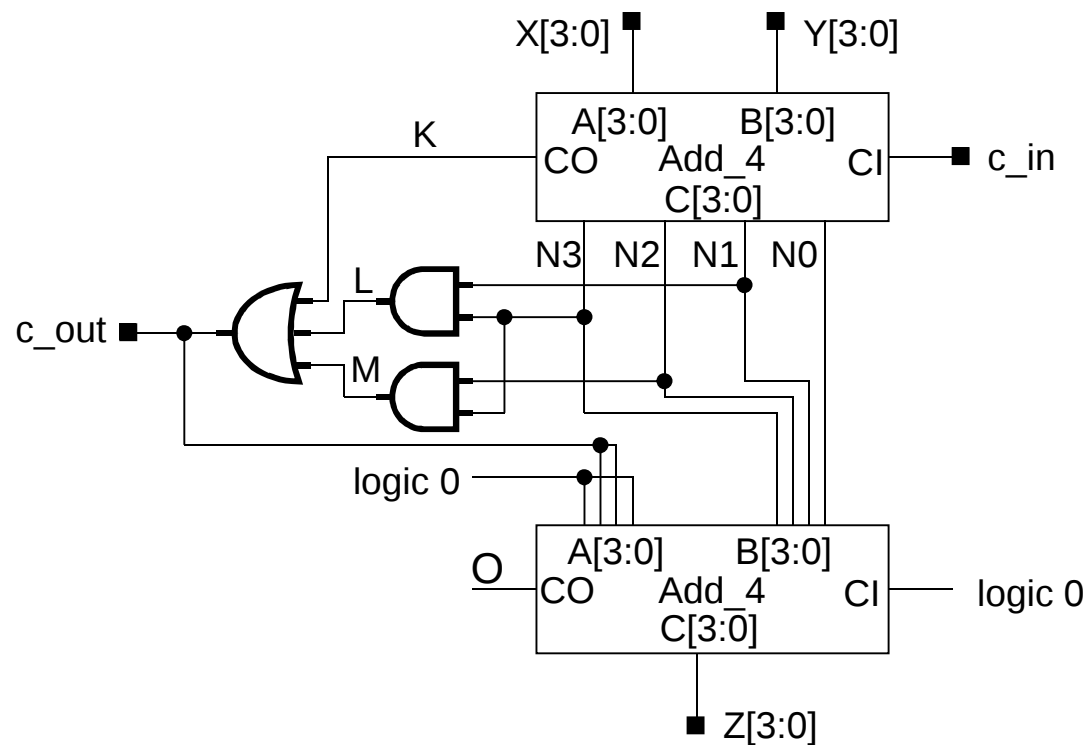
	숫자	BCD
X	8	1000
+Y	5	0101
Z	13	1 0011

	숫자	BCD
X	9	1001
+Y	9	1001
Z	18	1 1000



3.2 BCD Adder 실험

(1) 다음 그림과 같은 4bit BCD (binary-coded decimal) adder를 Verilog HDL 모듈로 표현하세요.





3.2 BCD Adder 실험

(2) p.34에 사용된 Add_4를 Verilog HDL 모듈로 표현하세요.
(structural description이든 behavioral description이든 관계
없으며, delay는 0이라고 가정하세요.)



3.2 BCD Adder 실험

(3) 시간에 따른 X, Y 값이 다음과 같도록 하는 stimulus generator를 Verilog HDL 코드로 표현하세요.

시간이 10 눈금 지난 후에
X[3:0]이 숫자 6(=0110)의
값을 가지게 하는 HDL 코드는
#10 X = 4'b0110
가 된다.



시간	X	Y
10	1	2
20	7	3
30	8	5
40	9	9
50	시뮬레이션 끝	



3.2 BCD Adder 실험

(4) 시간 \$time, 입력 신호 X, Y, 출력 신호 X+Y를 측정하는 response monitor를 Verilog HDL 코드로 표현하세요. 이때, c_out, Z[3:0]이 하나로 묶여서 5bit 값으로 출력되는 것에 주의하세요.

```
time=10      X=0001  Y=0010  X+Y=00011
ë
ë
ë
```



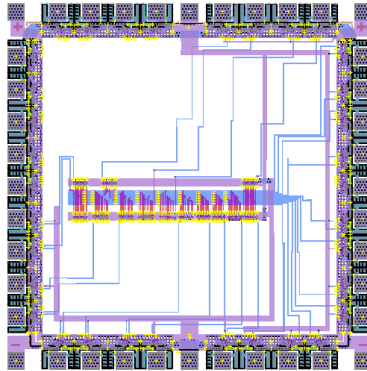
3.2 BCD Adder 실험

(5) 앞의 (1)-(4)에서 작성한 Verilog HDL 코드를 조합하여, (1)의 회로를 테스트하는 DUTB의 Verilog HDL 코드를 작성하세요.

(6) SILOS 프로그램을 사용하여 (5)에서 작성한 DUTB를 시뮬레이션하고, 출력 신호 $X+Y$ 가 시간에 따라 어떻게 변하는지 관찰하세요.

(7) 이 실험에서 만든 모든 파일을 저장해서 보관하세요. 다음 실험에 다시 사용됩니다.

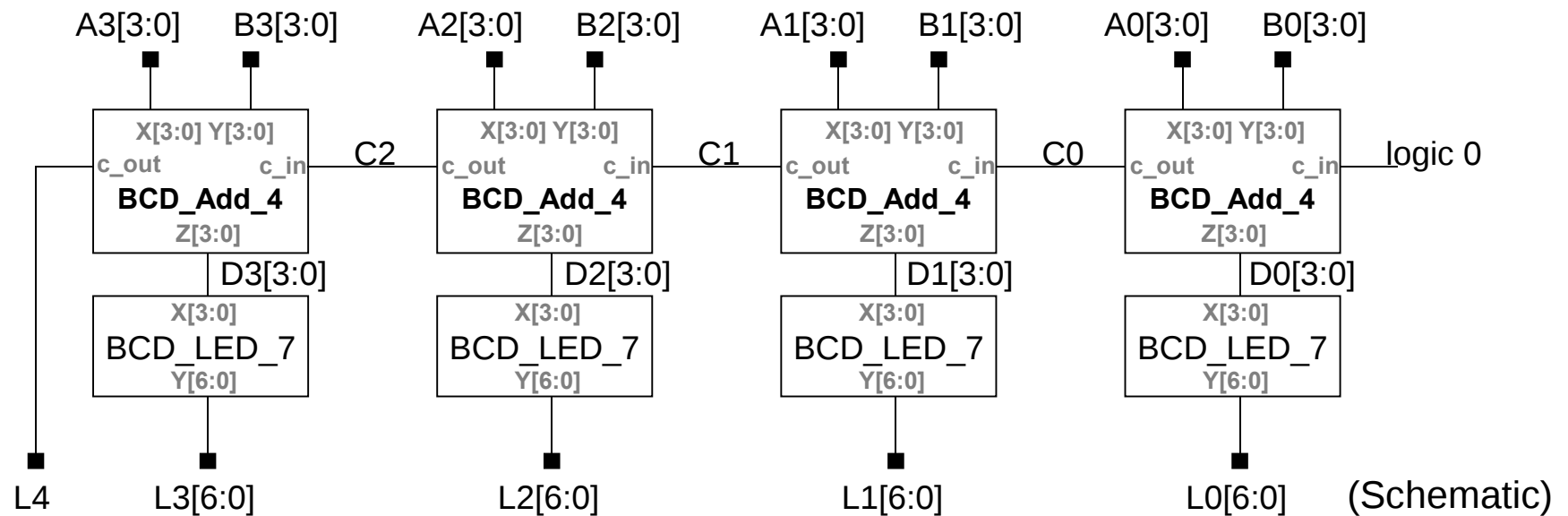
실험 4. Desktop Calculator





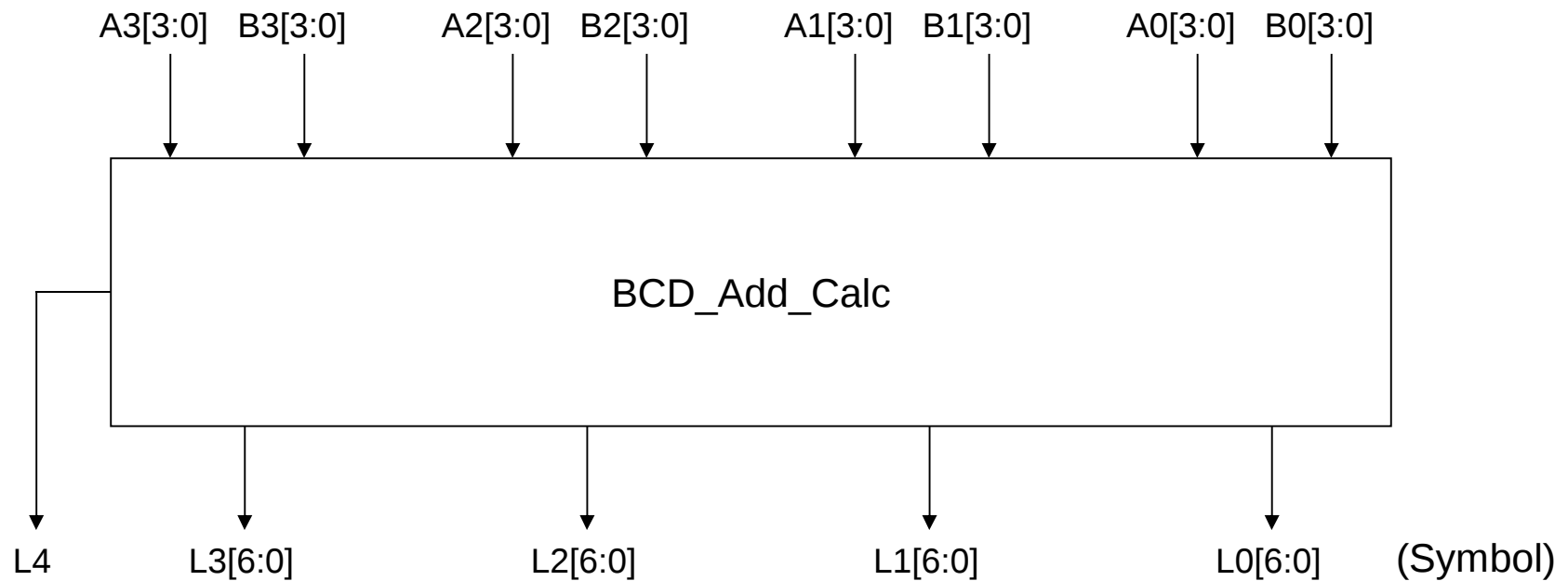
4.1 Desktop Calculator 실험

(1) 다음 그림은 4자리의 10진수로 표시되는 16비트 BCD 숫자 두개를 더해서 5자리의 7-segment LED에 표시하는 계산기의 블록도입니다. 이 블록을 Verilog HDL 모듈로 표현하세요.





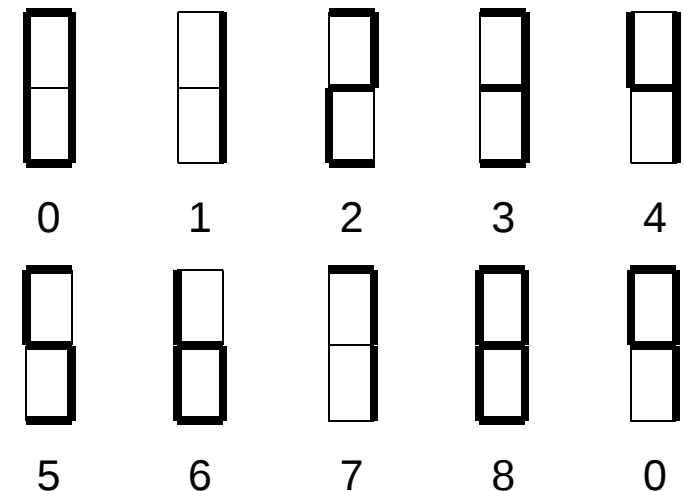
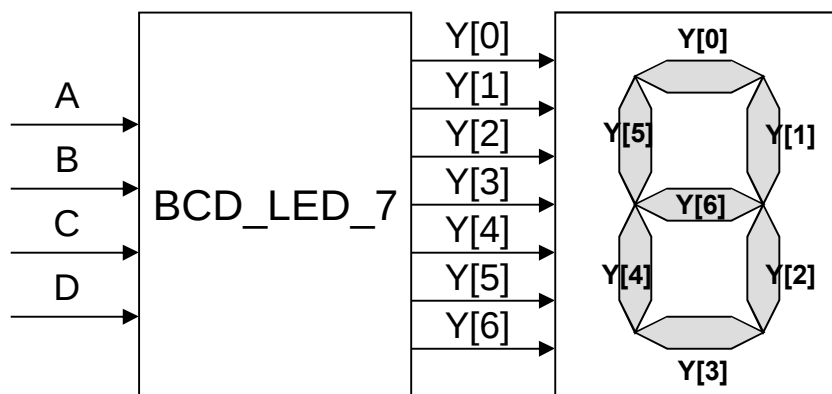
4.1 Desktop Calculator 실험





4.1 Desktop Calculator 실험

(2) 다음은 각각의 4비트 BCD 숫자를 7-segment LED에 표시하는 BCD_LED_7 블록을 나타낸 것입니다. 이 블록을 Verilog HDL 모듈로 표현하세요. 이때, case문을 사용하면 편리합니다.





4.1 Desktop Calculator 실험

(3) 시간에 따른 A0~A3, B0~B3의 숫자 값이 다음과 같을 때
입력 신호 A0[3:0], A1[3:0], A2[3:0], A3[3:0], B0[3:0], B1[3:0],
B2[3:0], B3[3:0]를 생성하는 stimulus generator를 Verilog HDL
코드로 표현하세요.

시간이 10 눈금 지난 후에
X[3:0]이 숫자 6(=0110)의
값을 가지게 하는 HDL 코드는
#10 X = 4'b0110
가 된다.



시간	A3	A2	A1	A0	B3	B2	B1	B0
10	1	2	3	4	4	3	2	1
20	7	2	1	8	0	3	6	9
30	0	5	0	4	9	8	2	6
40	5	3	4	7	2	1	4	8
50	시뮬레이션 끝							



4.1 Desktop Calculator 실험

(4) 시간 \$time, 입력 신호 A3~A1, B3~B1, 출력 신호 L4~L0를 측정하여 다음과 같이 출력하는 response monitor를 Verilog HDL 코드로 표현하세요.

```
time=10 A=1234 B=4321 L=0 1101101 1101101 1101101 1101101  
ë  
ë  
ë
```

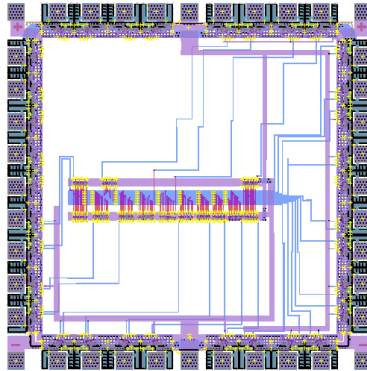


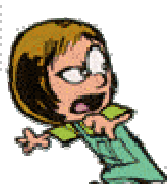
4.1 Desktop Calculator 실험

(5) 앞의 (1)-(4)에서 작성한 Verilog HDL 코드와, 이전 실험에서 작성했던 BCD_Add_4 모듈을 조합하여, (1)의 회로를 테스트하는 DUTB의 Verilog HDL 코드를 작성하세요.

(6) SILOS 프로그램을 사용하여 (5)에서 작성한 DUTB를 시뮬레이션하고, 출력 신호 L4~L0가 시간에 따라 어떻게 변하는지 관찰하세요.

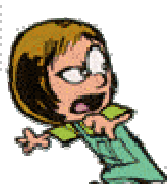
실험 5. Barrel Shifter





5.1 CASE 문

```
module mux(y, sel, a, b);  
  input                sel;  
  input [15:0]          a, b;  
  output [15:0]         y;  
  reg [15:0]            y;  
  
  always @(a or b or sel)  
  begin  
    case (sel)  
      1 : y = a;  
      0 : y = b;  
      default y = 1'bx;  
    endcase  
  end  
endmodule
```



5.2 FOR 문

```
module count_1s(a, b);  
  input [15:0]      a;  
  output [4:0]      b;  
  reg [4:0]         b;  
  integer          i;  
  
  initial  
  begin  
    b = 0;  
    for (i=0; i<16; i=i+1)  
      if (a[i]==1)  
        b = b + 1;  
  end  
  
endmodule
```



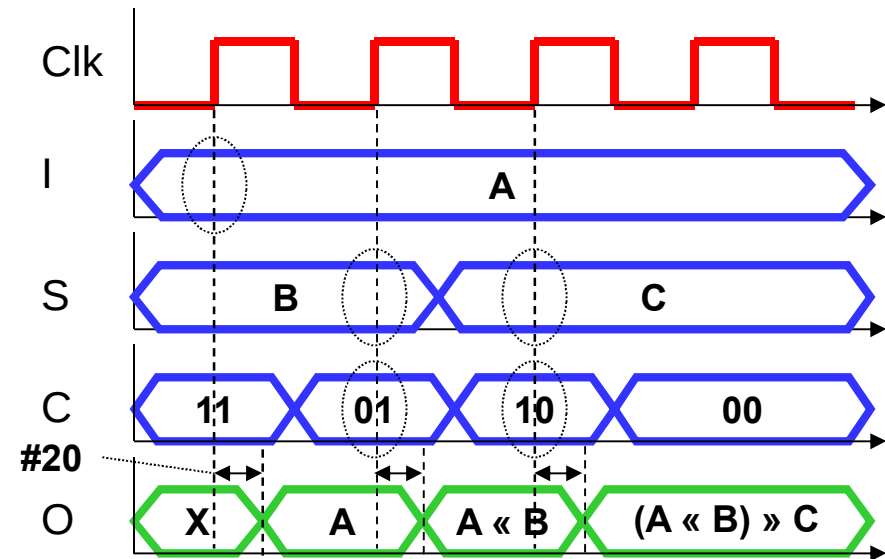
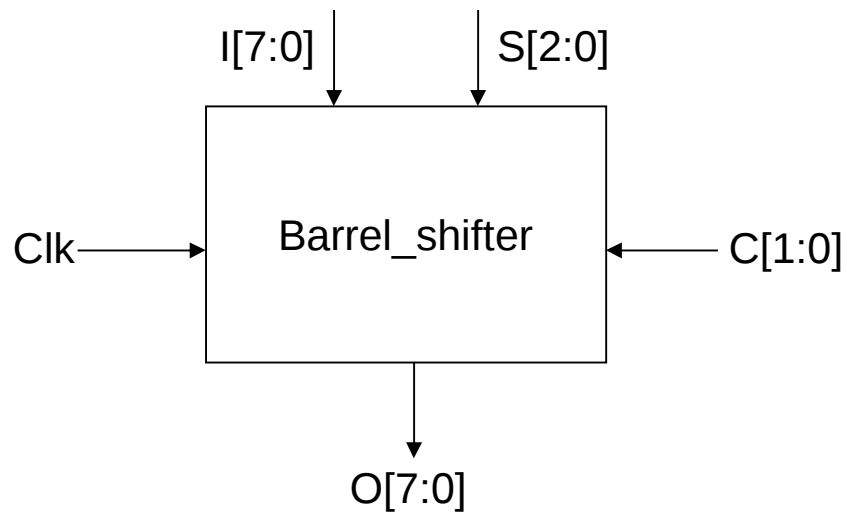
5.3 Barrel Shifter 실험

(1) Barrel Shifter는 입력 신호 $I[7:0]$ 와 제어 신호 $C[1:0]$, $S[2:0]$ 을 받아서, $C=11$ 일 때에는 출력 신호 $O[7:0] = I$ (load), $C=10$ 일 때에는 $O = O \gg S$ (shift right), $C=01$ 일 때에는 $O = O \ll S$ (shift left), $C=00$ 일 때에는 O 값을 그대로 유지시키는 블록입니다. 이때, 출력 신호는 클럭 신호 Clk 의 positive edge에서 #20 만큼의 시간이 지난 후에 바뀌며, 쉬프트시킨 빈자리에는 '0'이 채워집니다.



5.3 Barrel Shifter 실험

Clk	C	O
	00	O
	01	$I \ll S$
	10	$I \gg S$
	11	I



I[7:0]	C[1:0]	S[2:0]	O[7:0]
10101010	01	010	10101000
11100111	10	010	00111001
00011010	01	001	00110100
01010101	10	011	00001010
00011100	01	100	11000000
11001100	10	110	00000011

5.3 Barrel Shifter 실험



5.3 Barrel Shifter 실험

(2) p.50의 Barrel Shifter를 case문과 for문을 사용하여 Verilog HDL 모듈로 표현하세요. 이때, shift operator $\ll$ 과 $\gg$ 를 사용하지 마세요.

(힌트) C값에 따라서 load, shift left, shift right 등의 동작을 수행하도록 하는 부분은 case 문으로, I[7:0]을, S[2:0]비트만큼 쉬프트 시켜서 O[7:0]을 생성하는 부분은 for 문으로 기술하세요.



5.3 Barrel Shifter 실험

(3) 다음과 같은 입력 신호를 생성하는 stimulus generator를 Verilog HDL 코드로 표현하세요.

time	I[7:0]	C[1:0]	S[2:0]
0	10101010	11	010
100		01	
200	11100111	11	010
300		10	
400	00011010	11	001
500		01	
600	01010101	11	011
700		10	
800	00011100	11	100
900		01	
1000	11001100	11	110
1100		10	
1200	시뮬레이션 끝		



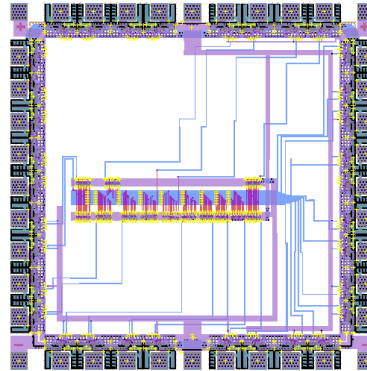
5.3 Barrel Shifter 실험

(4) 시간 \$time, 입력 신호 I, C, S, 출력 신호 O를 측정하여 다음과 같이 출력하는 response monitor를 Verilog HDL 코드로 표현하세요.

```
time=0 I=10101010 C=11 S=010 O=XXXXXXXXX  
ë  
ë  
ë
```

(5) (2)에서 기술한 Barrel_shifter 모듈을 테스트할 수 있도록 DUTB를 완성한 후, SILOS 프로그램을 사용하여 시뮬레이션하고, 출력 O를 시간에 따라 관찰하세요.

실험 6. ALU

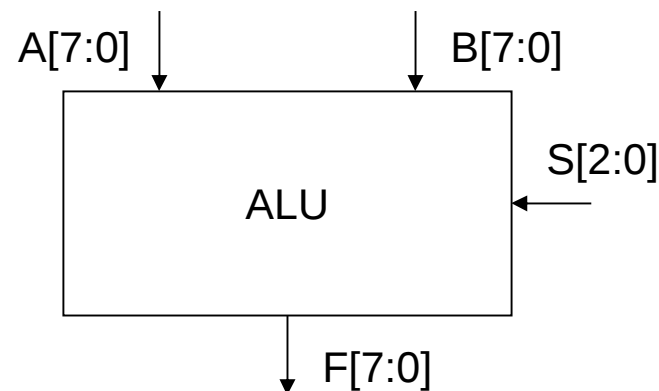




6.1 ALU 실험

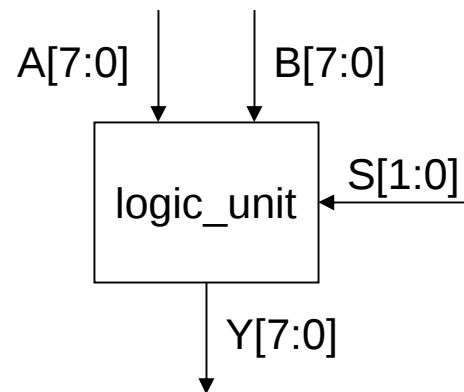
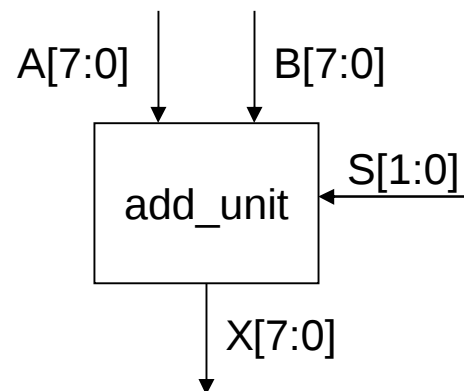
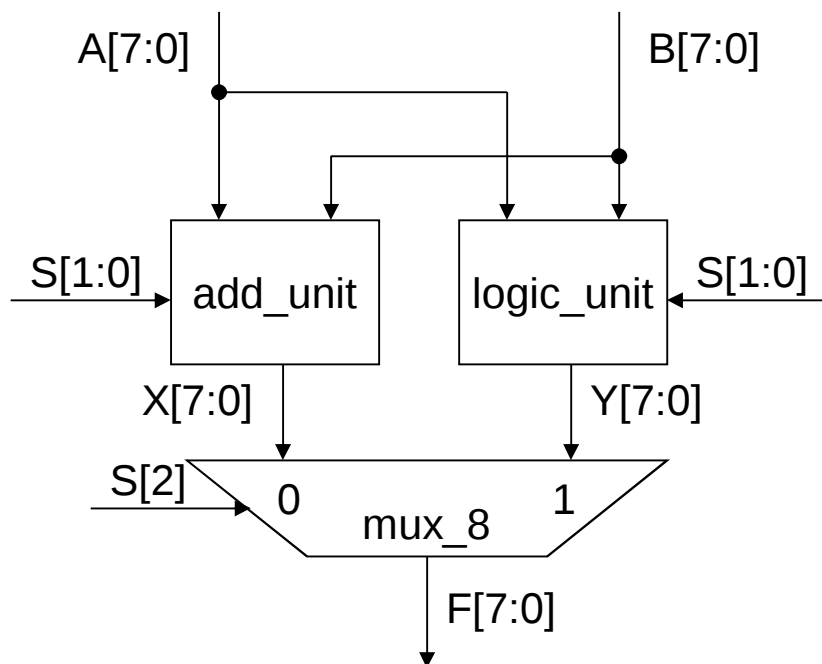
(1) ALU는 산술 연산 (arithmetic operation)과 논리 연산 (logic operation)을 담당하는 블록으로, 마이크로프로세서의 핵심 블록입니다. 이번 실험에서 설계할 8비트 ALU는 아래와 같은 구조와 기능을 가집니다.

S[2:0]	Output	Function
0 0 0	$F = A$	Transfer
0 0 1	$F = A + B$	Addition
0 1 0	$F = A - B$	Subtraction
0 1 1	$F = -A$	Negation
1 0 0	$F = A \mid B$	OR
1 0 1	$F = A \& B$	AND
1 1 0	$F = A \wedge B$	XOR
1 1 1	$F = \sim A$	NOT





6.1 ALU 실험



6.1 ALU 실험



6.1 ALU 실험

(2) 8비트 덧셈기 `add_8(O,A,B)`, 8비트 뺄셈기 `sub_8(O,A,B)`, 8비트 MUX인 `mux_8(O,A,B,S)`을 behavioral description 방식으로 Verilog HDL로 표현하세요. 이때, 문제에는 carry와 borrow가 표시되어 있지 않으므로, 이를 포함하여 기술하세요.

(3) p.56의 `add_unit` 모듈을 Verilog HDL 코드로 표현하세요. 이때, behavioral description을 사용하지 말고, (2)에서 만든 `add_8`, `sub_8`, `mux_8`만을 사용하여 structural description으로 기술하세요. (힌트) $F=A$, $F=A+B$ 의 계산은 `add_8`과 `mux_8`로, $F=A-B$, $F=-A$ 의 계산은 `sub_8`로 구현할 수 있습니다. `mux_8`은 모두 4개가 필요합니다.



6.1 ALU 실험

(4) p.56의 logic_unit를 Verilog HDL로 표현하세요. 이때, behavioral description을 사용하지 말고, Verilog HDL primitives인 or, and, xor, not와 (2)에서 만든 mux_8만을 사용하여 structural description으로 기술하세요.

(힌트) mux_8은 모두 3개가 필요합니다.

(5) p.55의 ALU를 add_unit와 logic_unit를 사용하여 Verilog HDL 모듈로 표현하세요.



6.1 ALU 실험

(6) 다음과 같은 입력 신호를 생성하는 stimulus generator를 Verilog HDL 코드로 표현하세요.

time	A[7:0]	B[7:0]	S[2:0]
0	10101010	01010101	000
100	00000011	00111111	001
200	01111101	00000011	010
300	01100111	11100111	011
400	00110011	00001111	100
500	11110000	11001100	101
600	11110000	10011001	110
700	01100111	11100111	111
800	시뮬레이션 끝		



6.1 ALU 실험

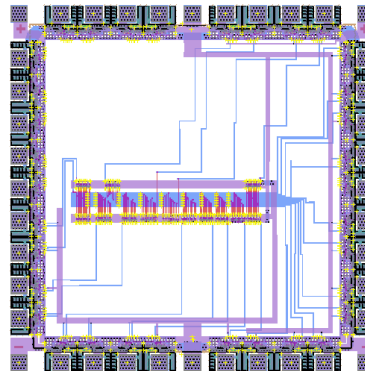
(7) 시간 \$time, 입력 신호 A, B, S, 출력 신호 F를 측정하여 다음과 같이 출력하는 response monitor를 Verilog HDL 코드로 표현하세요.

```
time=0 A=10101010 B=01010101 S=000 F=XXXXXXXXX  
ë  
ë  
ë
```

(8) (5)에서 기술한 ALU 모듈을 테스트할 수 있도록 DUTB를 완성하세요.

(9) SILOS 프로그램을 사용하여 시뮬레이션하고, 출력 F를 시간에 따라 관찰하세요.

실험 7. BCD to Excess-3 Code Converter

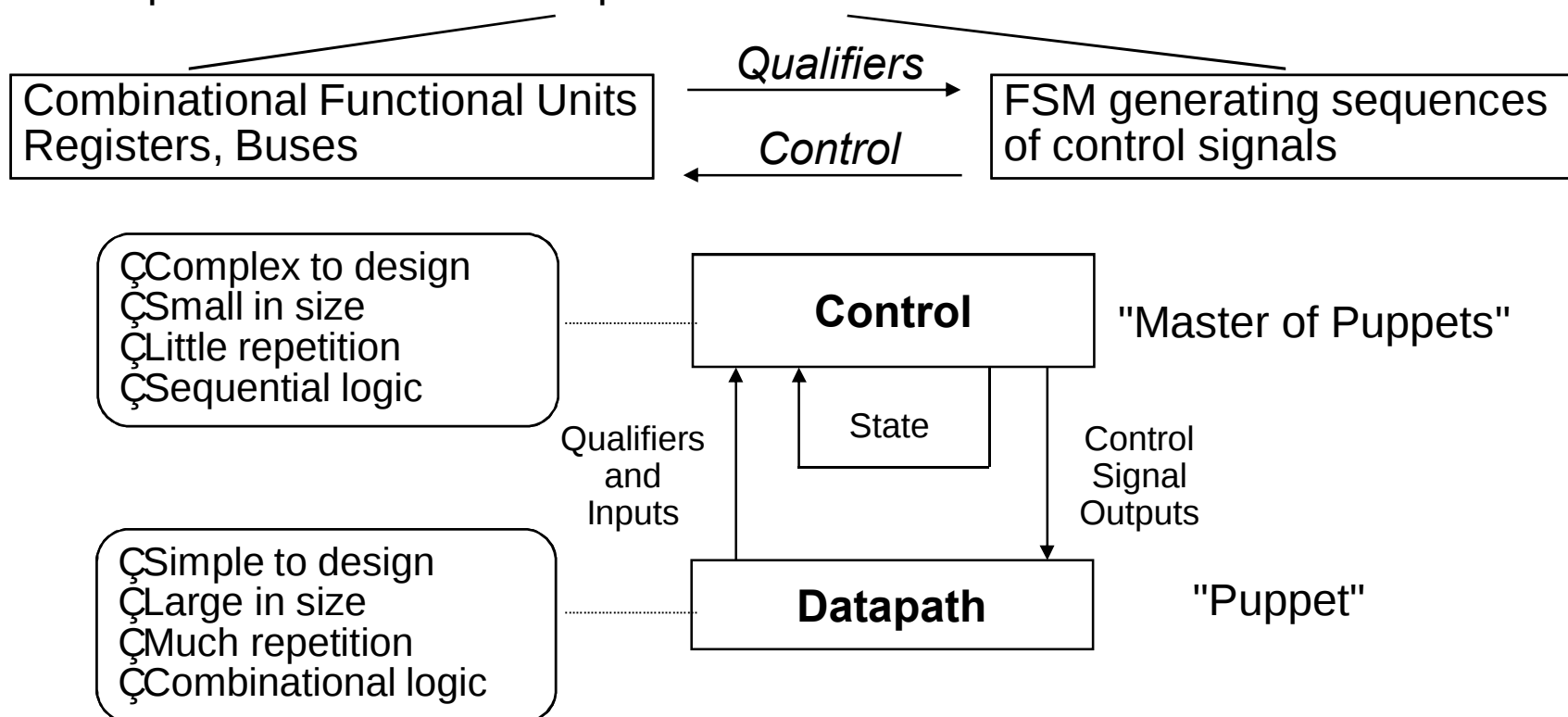




7.1 State Machine

ÇDatapath/Control Approach: Simpler to Design

Computer Hardware = Datapath + Control

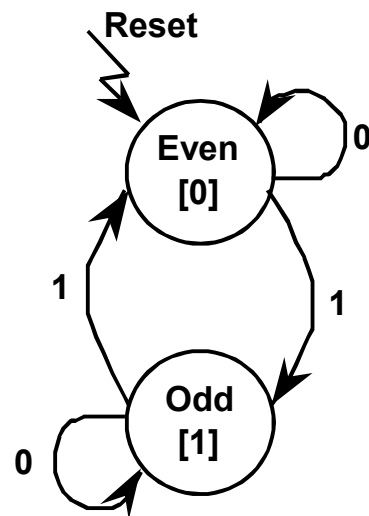




7.1 State Machine

Parity Checker:

assert output whenever input bit stream has odd # of 1's



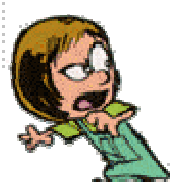
State
Diagram

Present State	Input	Next State	Output
Even	0	Even	0
Even	1	Odd	0
Odd	0	Odd	1
Odd	1	Even	1

Symbolic State Transition Table (Symbol)

Present State	Input	Next State	Output
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Encoded State Transition Table (Binary Number)

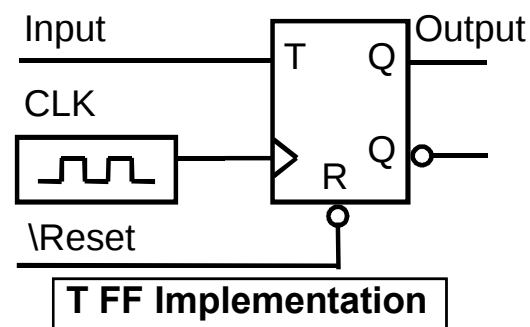
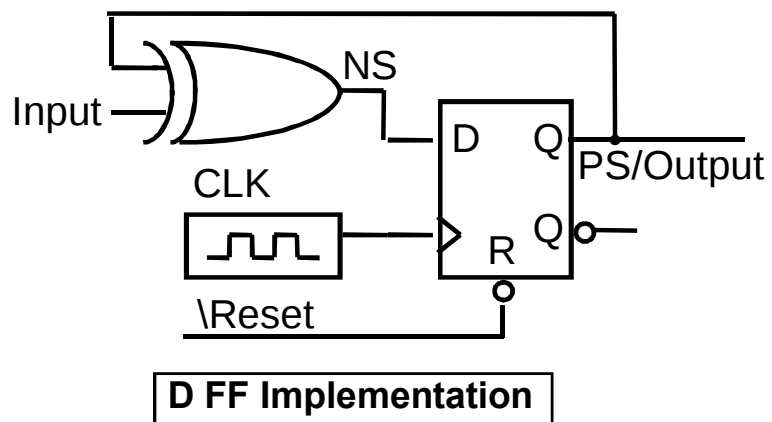


7.1 State Machine

Implementation

Next State/Output Functions

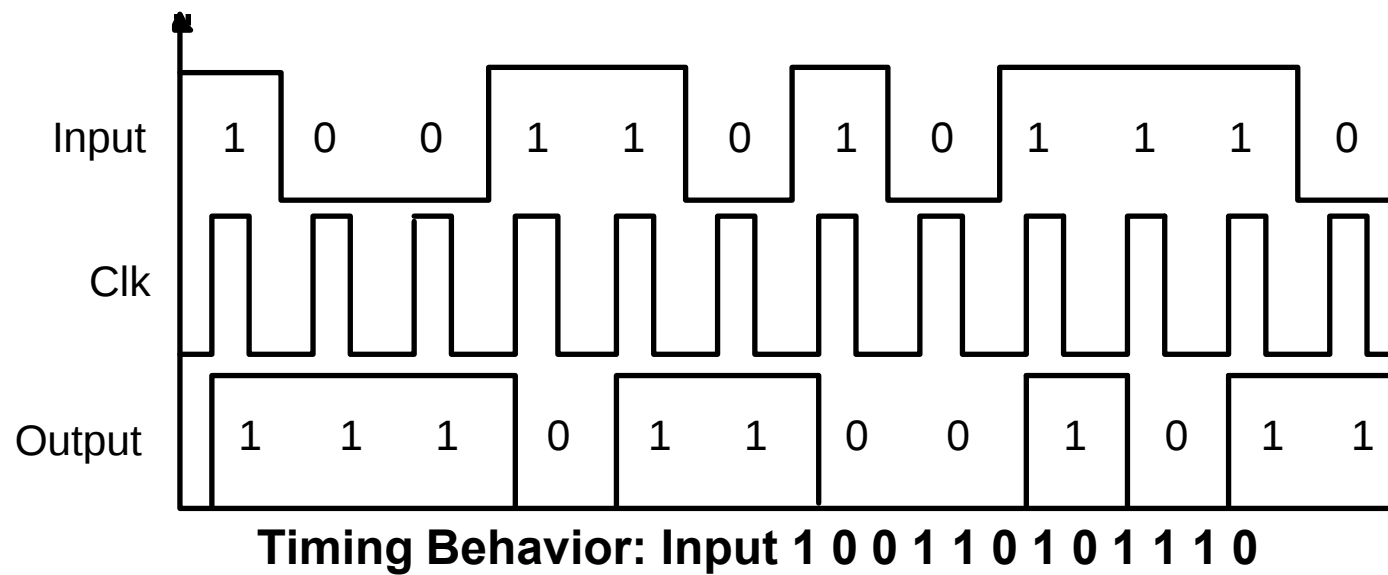
$$NS = PS \text{ xor } PI; \quad OUT = PS$$





7.2 Timing

ÇTiming Behavior





7.2 Timing

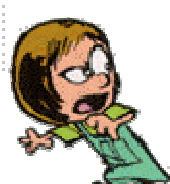
ÇTiming: When are inputs sampled,
next state computed, outputs asserted?

State Time: Time between clocking events

- Clocking event causes state/outputs to transition, based on inputs
- After propagation delay, Next State entered, Outputs are stable

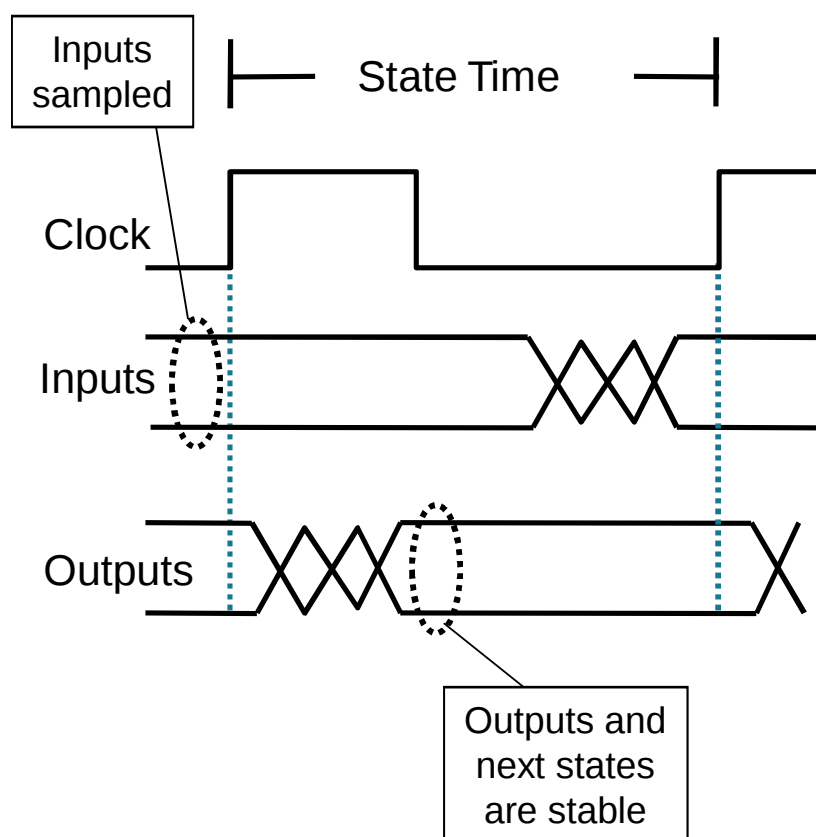
NOTE: Asynchronous signals take effect immediately
Synchronous signals take effect at the next clocking event

ex) tri-state enable: effective immediately
sync. counter clear: effective at next clock event
async. counter clear: effective immediately



7.2 Timing

Example: Positive Edge Triggered Synchronous System



On rising edge, inputs sampled
outputs, next state computed

After propagation delay, outputs and
next state are stable

IMPORTANT:

**Inputs to FSM should be settled down
before clock edge.**



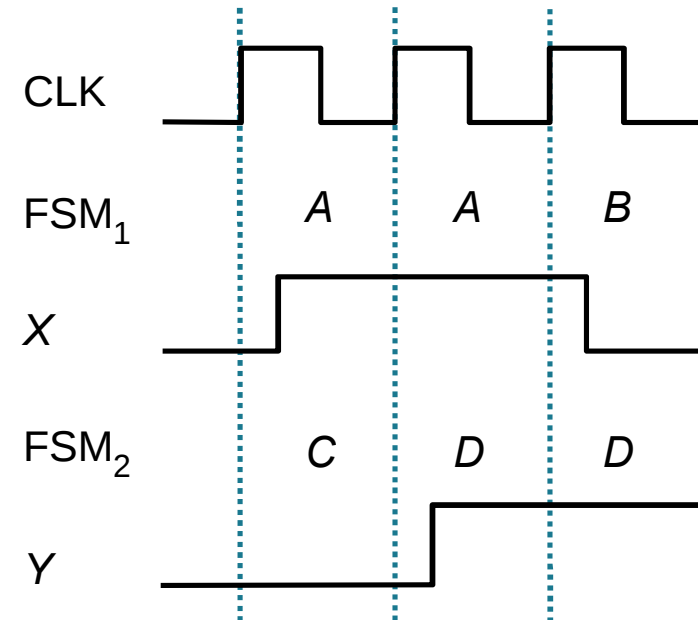
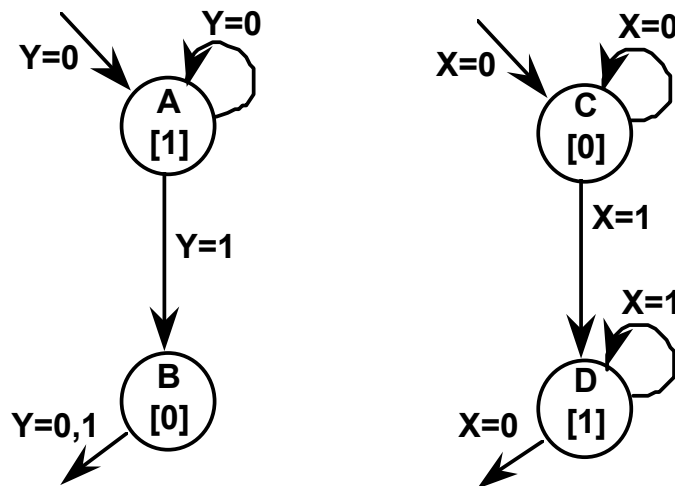
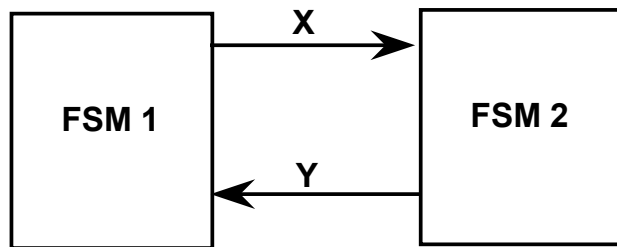
말로는 쉽지만 굉장히
헛갈리고 잘 까먹게 된다.
유일한 방법은 많이
디자인해보는 것 뿐!

7.2 Timing



7.2 Timing

ÇExample: Positive Edge Triggered Synchronous System



- Initial inputs/outputs: X = 0, Y = 0

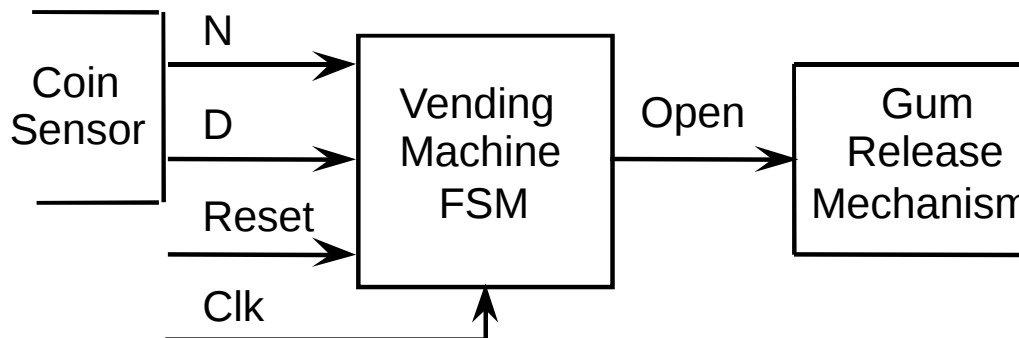


7.3 Vending Machine Example

ÇGeneral Machine Concept

- deliver package of gum after 15 cents deposited
- single coin slot for dimes (10 cents), nickels (5 cents)
- no change

ÇStep 1: Understand the problem: draw a block diagram





7.3 Vending Machine Example

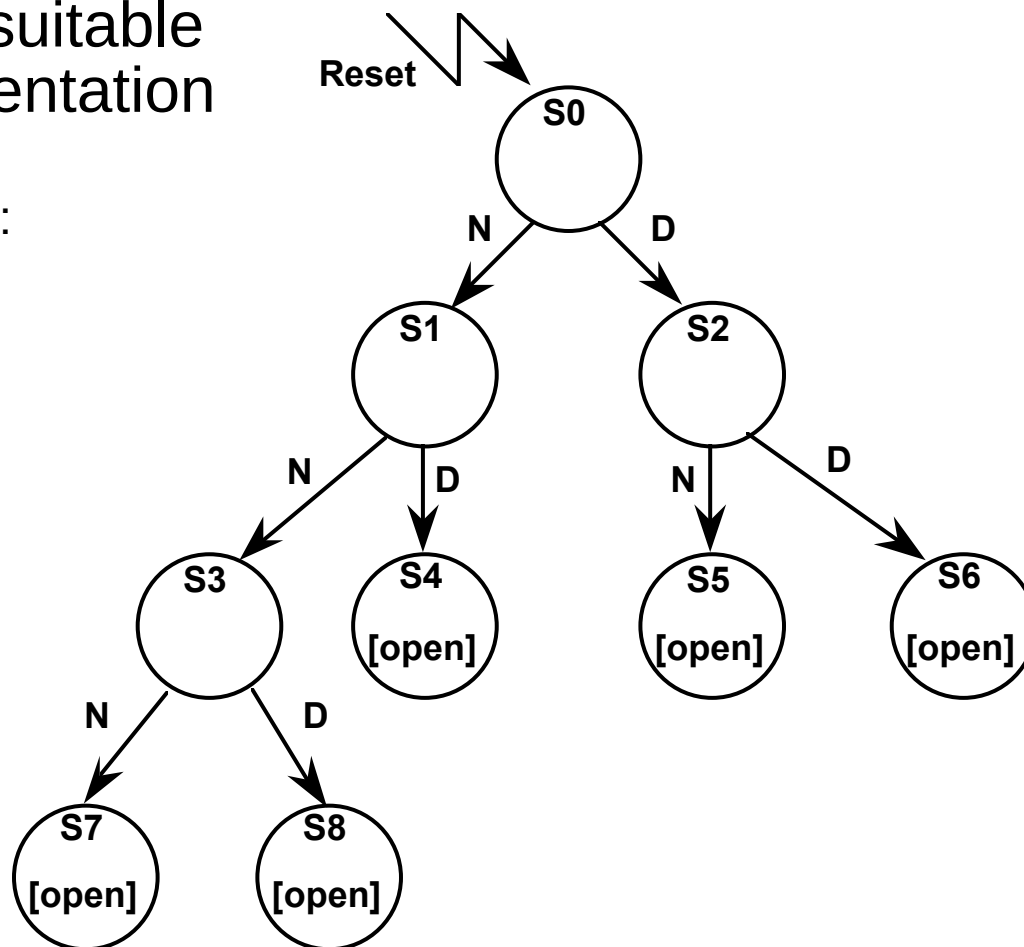
ÇStep 2: Map into more suitable abstract representation

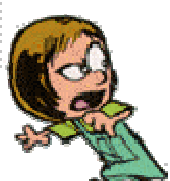
Tabulate typical input sequences:

- three nickels
- nickel, dime
- dime, nickel
- two dimes
- two nickels, dime

Draw state diagram:

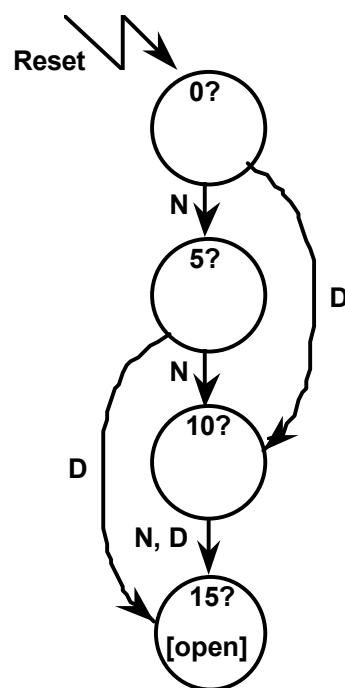
- Inputs: N, D, reset
- Output: open





7.3 Vending Machine Example

ÇStep 3: State Minimization



reuse states
whenever possible

Present State	Inputs		Next State	Output Open
	D	N		
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	X	X
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	X	X
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	X	X
15¢	X	X	15¢	1

Symbolic State Table



7.3 Vending Machine Example

ÇStep 4: State Assignment

Present State		Inputs		Next State		Output
Q_1	Q_0	D	N	D_1	D_0	Open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	X	X	X
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	X	X	X
1	0	0	0	1	0	0
		0	1	1	1	0
		1	0	1	1	0
		1	1	X	X	X
1	1	0	0	1	1	1
		0	1	1	1	1
		1	0	1	1	1
		1	1	X	X	X



7.3 Vending Machine Example

ÇStep 5: State Register Implementation

K-map for D1

		Q0	Q1	Q1		
		00	01	11	10	
D	N	00	0	0	1	1
	01	0	1	1	1	
	11	X	X	X	X	
	10	1	1	1	1	
		Q0		Q1		

K-map for D0

A Karnaugh map for the function D0. The map is a 4x4 grid with rows labeled D (00, 01, 11, 10) and columns labeled Q0 (00, 01, 11, 10). The map contains the following values: 0, 1, 1, 0 in the first row; 1, 0, 1, 1 in the second row; X, X, X, X in the third row; and 0, 1, 1, 1 in the fourth row. The map is annotated with several groupings and labels: a bracket labeled Q1 above the top two columns (01 and 11); a bracket labeled Q1 above the bottom two columns (11 and 10); a bracket labeled Q0 below the bottom two columns (01 and 11); a bracket labeled D on the right side of the map; and a bracket labeled N on the right side of the map. The map is also labeled with Q0 and Q1 at the top and D and N on the left.

		Q1			
		00	01	11	10
D	00	0	1	1	0
	01	1	0	1	1
	11	X	X	X	X
	10	0	1	1	1

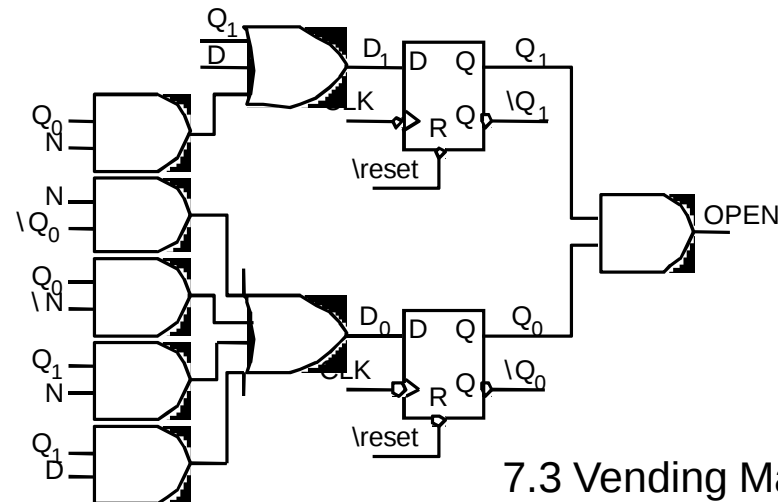
K-map for OPEN

		Q0	Q1	Q1		
		00	01	11	10	
D	N	00	0	0	1	0
	01	0	0	1	0	N
	11	X	X	X	X	
	10	0	0	1	0	
		Q0				

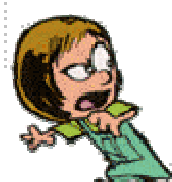
$$D1 = Q1 + D + Q0 N$$

$$D0 = N \overline{Q0} + Q0 \overline{N} + Q1 N + Q1 D$$

$$OPEN = Q1 Q0$$

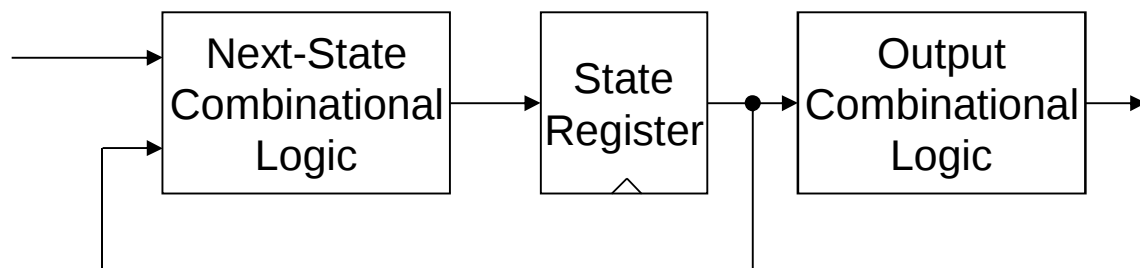


7.3 Vending Machine Example



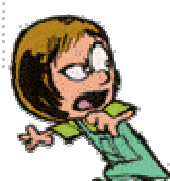
7.4 Moore Machine and Mealy Machine

Moore Machine



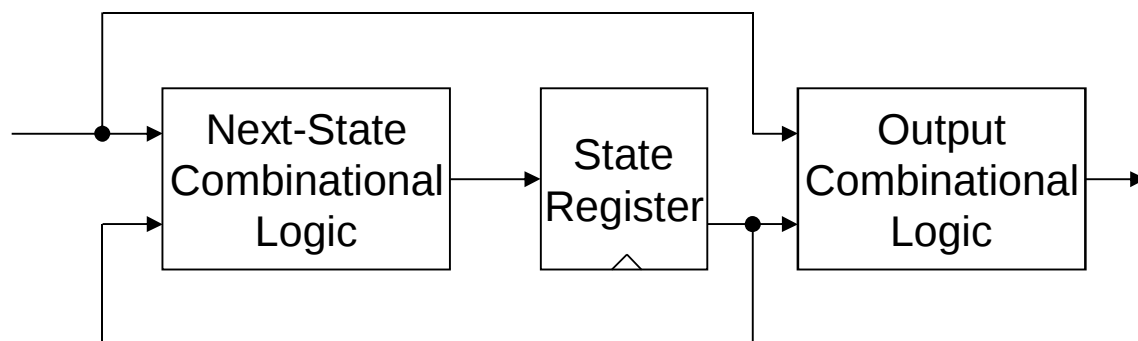
Outputs are function
solely of the current state

Outputs change
**synchronously with
state changes**



7.4 Moore Machine and Mealy Machine

Mealy Machine



Outputs depend on
state AND **inputs**

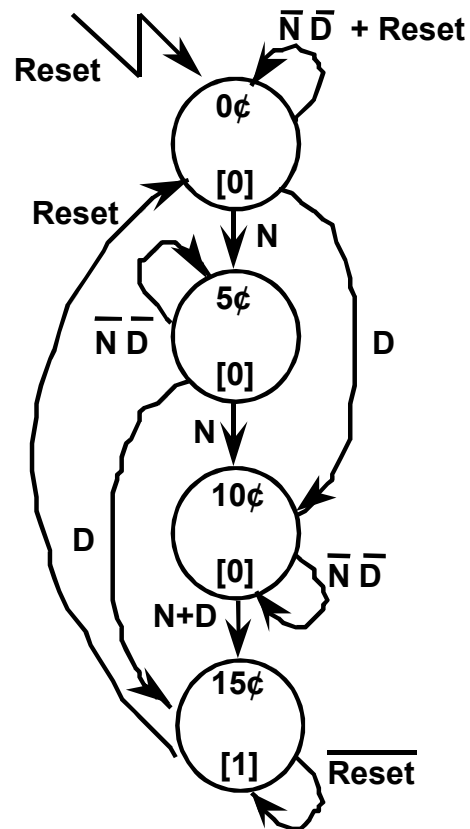
Input change causes an
immediate output change

Asynchronous outputs

7.4 Moore Machine and Mealy Machine

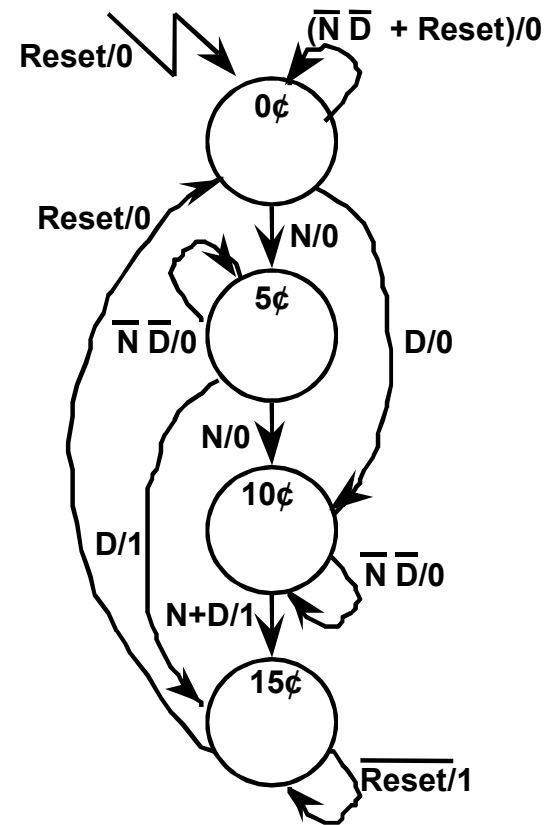


Moore Machine

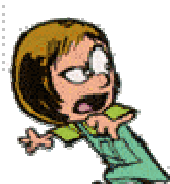


Outputs are associated with **State**

Mealy Machine

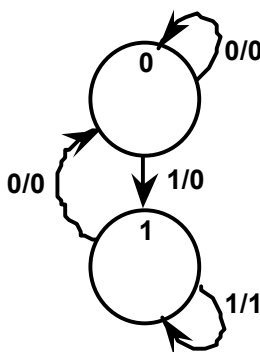
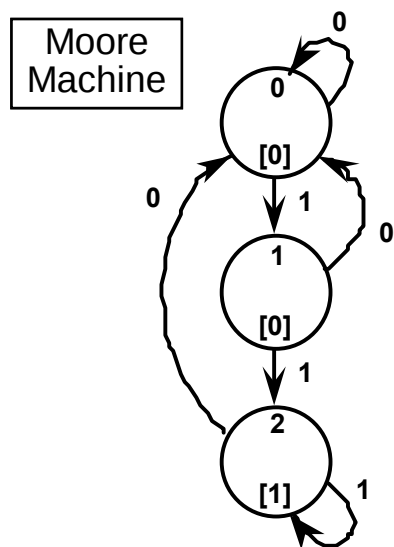


Outputs are associated with **Transitions**



7.4 Moore Machine and Mealy Machine

- Mealy Machine typically has **fewer states** than Moore Machine for same output sequence
- **Timing** of Mealy Machine is **more complex** than Moore Machine



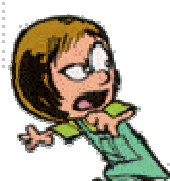
Same I/O behavior
but
Different # of states

Mealy Machine

Mealy Machine은
state의 수가 적기 때문에
전문 **designer**들은
복잡해도 그냥 사용한다.

자신없으면
Moore Machine으로
설계하는 편이
수월하다.





7.5 Complex Counter Example

ÇProblem Definition

A sync. 3 bit counter has a mode control M. When M = 0, the counter counts up in the binary sequence. When M = 1, the counter advances through the Gray code sequence.

Binary: 000, 001, 010, 011, 100, 101, 110, 111

Gray: 000, 001, 011, 010, 110, 111, 101, 100

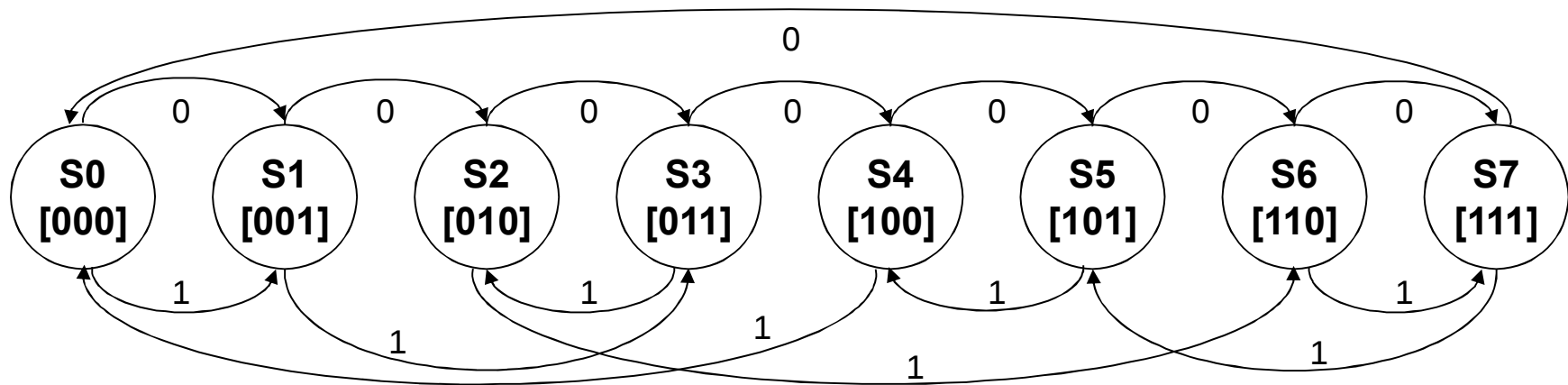
ÇExample of Valid I/O behavior

Mode Input M	Current State	Next State (Z2 Z1 Z0)
0	000	001
0	001	010
1	010	110
1	110	111
1	111	101
0	101	110
0	110	111



7.5 Complex Counter Example

ÇState Diagram of Moore Machine





7.6 Traffic Controller Example

Problem Definition

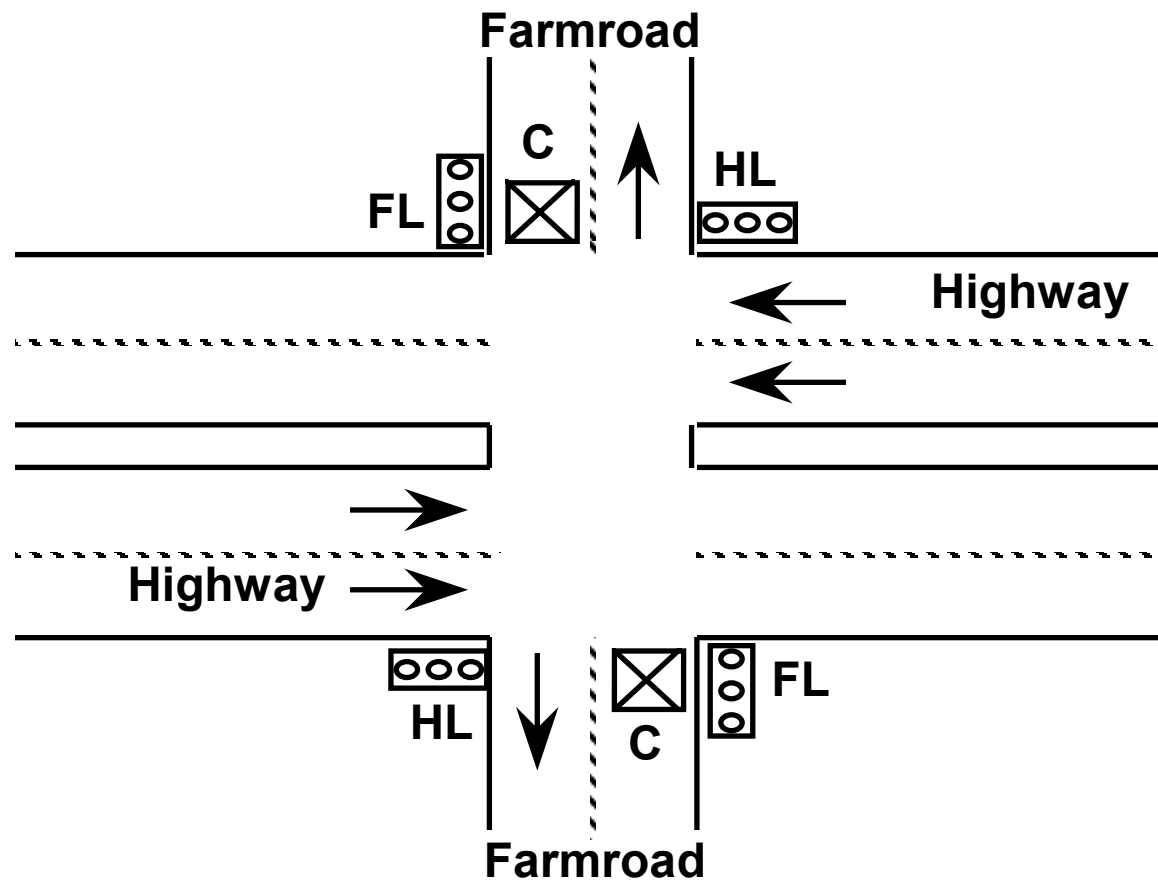
A busy highway is intersected by a little used farmroad. Detectors C sense the presence of cars waiting on the farmroad. With no car on farmroad, light remain green in highway direction. If vehicle on farmroad, highway lights go from Green to Yellow to Red, allowing the farmroad lights to become green. These stay green only as long as a farmroad car is detected but never longer than a set interval. When these are met, farm lights transition from Green to Yellow to Red, allowing highway to return to green. Even if farmroad vehicles are waiting, highway gets at least a set interval as green.

Assume you have an interval timer that generates a short time pulse (TS) and a long time pulse (TL) in response to a set (ST) signal. TS is to be used for timing yellow lights and TL for green lights.



7.6 Traffic Controller Example

Çllustration





7.6 Traffic Controller Example

ÇTabulation of Inputs and Outputs:

Input Signal

reset
C
TS
TL

Description

place FSM in initial state
detect vehicle on farmroad
short time interval expired
long time interval expired

Output Signal

HG, HY, HR
FG, FY, FR
ST

Description

assert green/yellow/red highway lights
assert green/yellow/red farmroad lights
start timing a short or long interval

ÇTabulation of Unique States

State

S0
S1
S2
S3

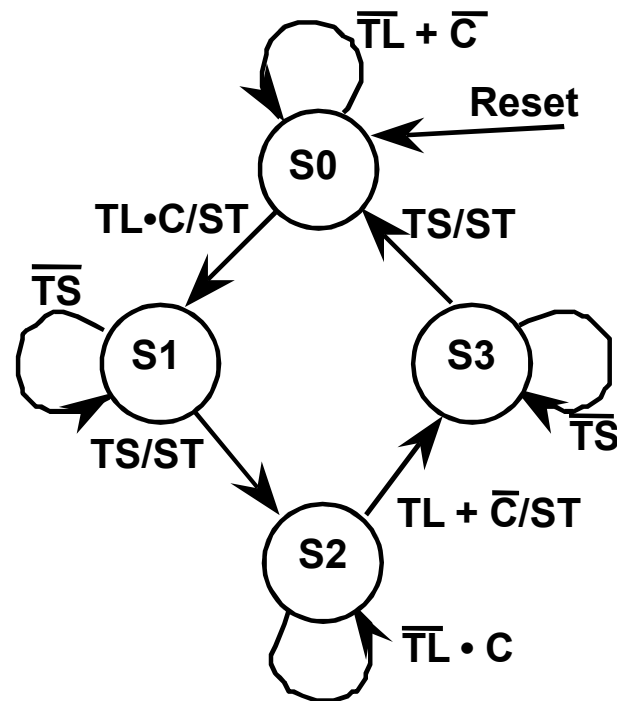
Description

Highway green (farmroad red)
Highway yellow (farmroad red)
Farmroad green (highway red)
Farmroad yellow (highway red)



7.6 Traffic Controller Example

ÇState Diagram of Mealy Machine



S0: HG

S1: HY

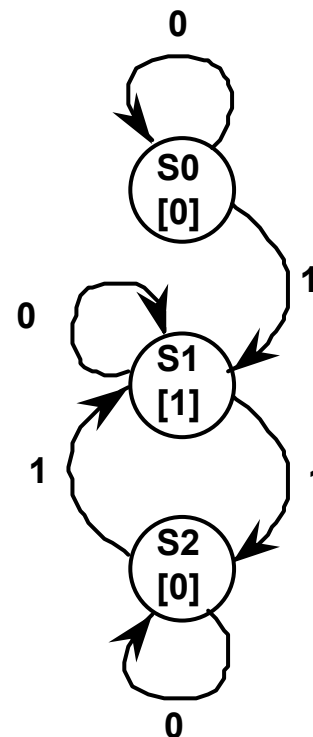
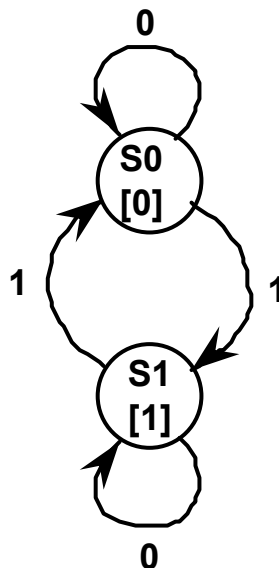
S2: FG

S3: FY



7.7 State Machine Reduction

Parity Checker
Example



- Identical output behavior on all input strings
- FSMs are *equivalent*, but require different implementations



7.7 State Machine Reduction

Implement FSM with fewest possible states

- Least number of flip-flops
- Boundaries are power of two number of states
- Fewest states usually leads to more opportunities for don't cares
- Reduce the number of gates needed for implementation



7.7 State Machine Reduction

ÇGoal

- **Identify and combine states that have equivalent behavior**
- **Equivalent States**: for all input combinations, states transition to the same or equivalent states
- Parity Checker Example: S0, S2 are equivalent states
Both output a 0
Both transition to S1 on a 1 and self-loop on a 0

ÇApproach

- Start with **state transition table**
- **Identify states with same output behavior**
- **If such states transition to the same next state, they are equivalent**
- **Combine into a single new renamed state**
- Repeat until no new states are combined



7.7 State Machine Reduction

4-bit Sequence Detector Example

Single input X, output Z

Taking inputs grouped four at a time, output 1 if last four inputs were the string 1010 or 0110

I/O Behavior:

X = 0010 0110 1100 1010 0011 ...
Z = 0000 0001 0000 0001 0000 ...

Upper bound on FSM complexity

Fifteen states ($1 + 2 + 4 + 8$)

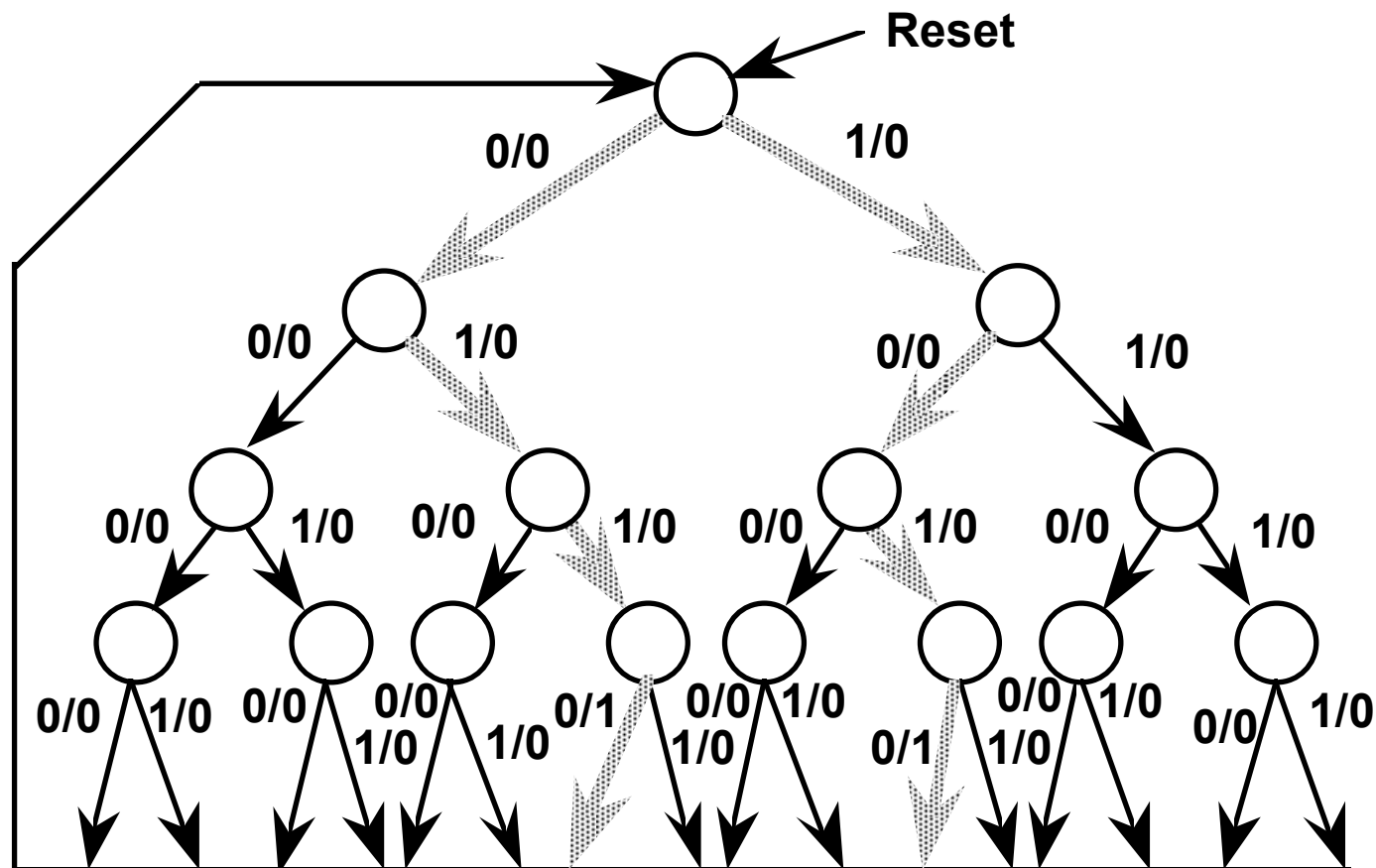
Thirty transitions ($2 + 4 + 8 + 16$)

sufficient to recognize any binary string of length four!



7.7 State Machine Reduction

ÇState Diagram



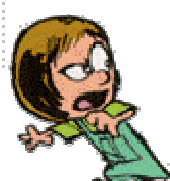
7.7 State Machine Reduction



7.7 State Machine Reduction

ÇState
Transition
Table

Input Sequence	Present State	Next State		Output	
		X=0	X=1	X=0	X=1
Reset	S ₀	S ₁	S ₂	0	0
0	S ₁	S ₃	S ₄	0	0
1	S ₂	S ₅	S ₆	0	0
00	S ₃	S ₇	S ₈	0	0
01	S ₄	S ₉	S ₁₀	0	0
10	S ₅	S ₁₁	S ₁₂	0	0
11	S ₆	S ₁₃	S ₁₄	0	0
000	S ₇	S ₀	S ₀	0	0
001	S ₈	S ₀	S ₀	0	0
010	S ₉	S ₀	S ₀	0	0
011	S ₁₀	S ₀	S ₀	1	0
100	S ₁₁	S ₀	S ₀	0	0
101	S ₁₂	S ₀	S ₀	1	0
110	S ₁₃	S ₀	S ₀	0	0
111	S ₁₄	S ₀	S ₀	0	0



7.7 State Machine Reduction

Input Sequence	Present State	Next State		Output	
		X=0	X=1	X=0	X=1
Reset	S ₀	S ₁	S ₂	0	0
0	S ₁	S ₃	S ₄	0	0
1	S ₂	S ₅	S ₆	0	0
00	S ₃	S ₇	S ₈	0	0
01	S ₄	S ₉	S ₁₀	0	0
10	S ₅	S ₁₁	S ₁₂	0	0
11	S ₆	S ₁₃	S ₁₄	0	0
000	S ₇	S ₀	S ₀	0	0
001	S ₈	S ₀	S ₀	0	0
010	S ₉	S ₀	S ₀	0	0
011	S ₁₀	S ₀	S ₀	1	0
100	S ₁₁	S ₀	S ₀	0	0
101	S ₁₂	S ₀	S ₀	1	0
110	S ₁₃	S ₀	S ₀	0	0
111	S ₁₄	S ₀	S ₀	0	0



7.7 State Machine Reduction

Input Sequence	Present State	Next State		Output	
		X=0	X=1	X=0	X=1
Reset	S ₀	S ₁	S ₂	0	0
0	S ₁	S ₃	S ₄	0	0
1	S ₂	S ₅	S ₆	0	0
00	S ₃	S ₇	S ₈	0	0
01	S ₄	S ₉	S' ₁₀	0	0
10	S ₅	S ₁₁	S' ₁₀	0	0
11	S ₆	S ₁₃	S ₁₄	0	0
000	S ₇	S ₀	S ₀	0	0
001	S ₈	S ₀	S ₀	0	0
010	S ₉	S ₀	S ₀	0	0
011 or 101	S' ₁₀	S ₀	S ₀	1	0
100	S ₁₁	S ₀	S ₀	0	0
110	S ₁₃	S ₀	S ₀	0	0
111	S ₁₄	S ₀	S ₀	0	0



7.7 State Machine Reduction

Input Sequence	Present State	Next State		Output	
		X=0	X=1	X=0	X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S_7	S_8	0	0
01	S_4	S_9	S'_{10}	0	0
10	S_5	S_{11}	S'_{10}	0	0
11	S_6	S_{13}	S_{14}	0	0
000	S_7	S_0	S_0	0	0
001	S_8	S_0	S_0	0	0
010	S_9	S_0	S_0	0	0
011 or 101	S'_{10}	S_0	S_0	1	0
100	S_{11}	S_0	S_0	0	0
110	S_{13}	S_0	S_0	0	0
111	S_{14}	S_0	S_0	0	0



7.7 State Machine Reduction

Input Sequence	Present State	Next State		Output	
		X=0	X=1	X=0	X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S'_7	S'_7	0	0
01	S_4	S'_7	S'_{10}	0	0
10	S_5	S'_7	S'_{10}	0	0
11	S_6	S'_7	S'_7	0	0
not (011 or 101)	S'_7	S_0	S_0	0	0
011 or 101	S'_{10}	S_0	S_0	1	0



7.7 State Machine Reduction

Input Sequence	Present State	Next State		Output	
		X=0	X=1	X=0	X=1
Reset	S_0	S_1	S_2	0	0
0	S_1	S_3	S_4	0	0
1	S_2	S_5	S_6	0	0
00	S_3	S'_7	S'_7	0	0
01	S_4	S'_7	S'_{10}	0	0
10	S_5	S'_7	S'_{10}	0	0
11	S_6	S'_7	S'_7	0	0
not (011 or 101)	S'_7	S_0	S_0	0	0
011 or 101	S'_{10}	S_0	S_0	1	0

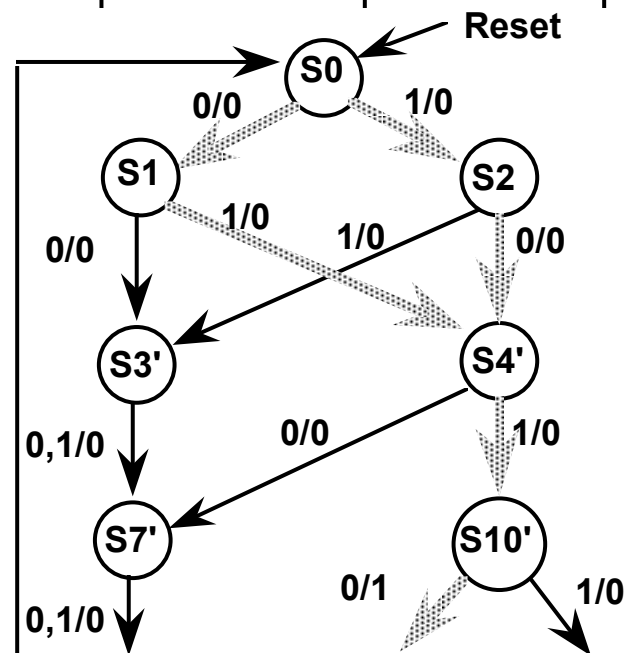


7.7 State Machine Reduction

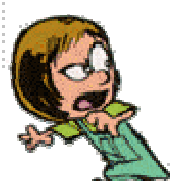
**Final Reduced
State Transition Table**

Input Sequence	Present State	Next State		Output	
		X=0	X=1	X=0	X=1
Reset	S0	S1	S2	0	0
0	S1	S3'	S4'	0	0
1	S2	S4'	S3'	0	0
00 or 11	S3'	S7'	S7'	0	0
01 or 10	S4'	S7'	S10'	0	0
not (011 or 101)	S7'	S0	S0	0	0
011 or 101	S10'	S0	S0	1	0

**Corresponding State
Diagram**



7.7 State Machine Reduction



7.7 State Machine Reduction

Row-Matching Method

- Straightforward to understand and easy to implement
- Problem: does not allow yield the most reduced state table!

Example: 3 State Parity Checker

Present State	Next State		Output
	X=0	X=1	
S_0	S_0	S_1	0
S_1	S_1	S_2	1
S_2	S_2	S_1	0

No way to combine states S_0 and S_2
based on Next State Criterion!



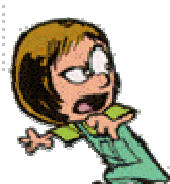
7.8 BCD to Excess-3 Code Converter 실험

(1) BCD Code와 Excess-3 Code는 모두 4비트 코드이며, 다음 테이블과 같습니다.

BCD	Excess-3
0000	0011
0001	0100
0010	0101
0011	0110
0100	0111
0101	1000
0110	1001
0111	1010
1000	1011
1001	1100
1010	xxxx
1011	xxxx
1100	xxxx
1101	xxxx
1110	xxxx
1111	xxxx

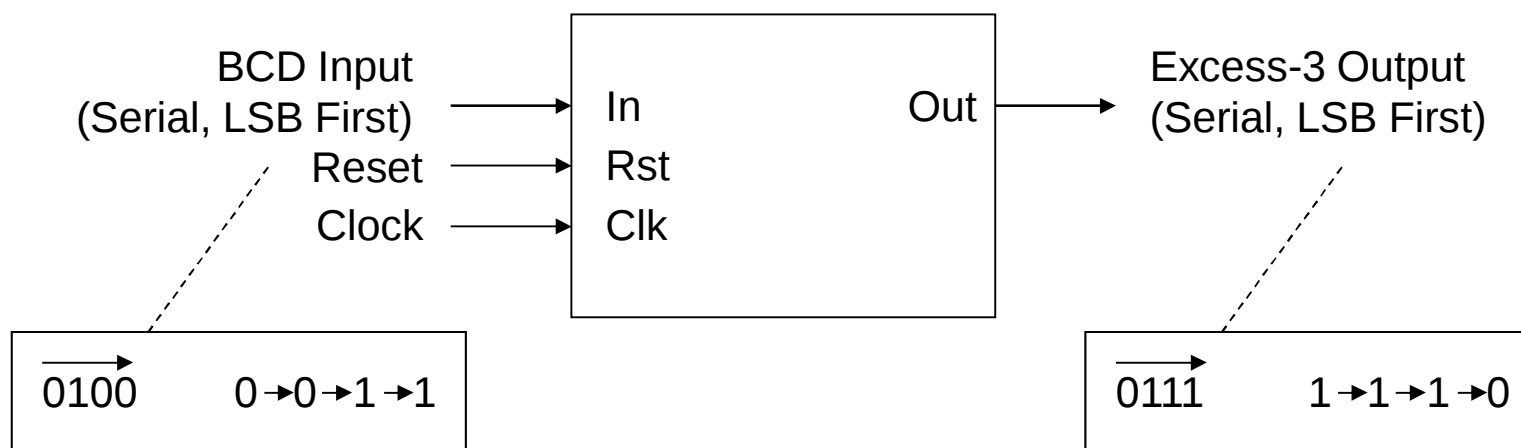
해당되는
BCD Code가
없으므로
출력은
Don't Care

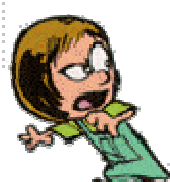
7.8 BCD to Excess-3 Code Converter 실험



7.8 BCD to Excess-3 Code Converter 실험

(2) 이번 실험에서 설계할 BCD to Excess-3 Code는 다음 그림과 같이 4비트의 BCD Code를 1비트씩 입력받아서 4비트의 Excess-3 Code를 1비트씩 출력하는 기능과 구조를 가집니다.





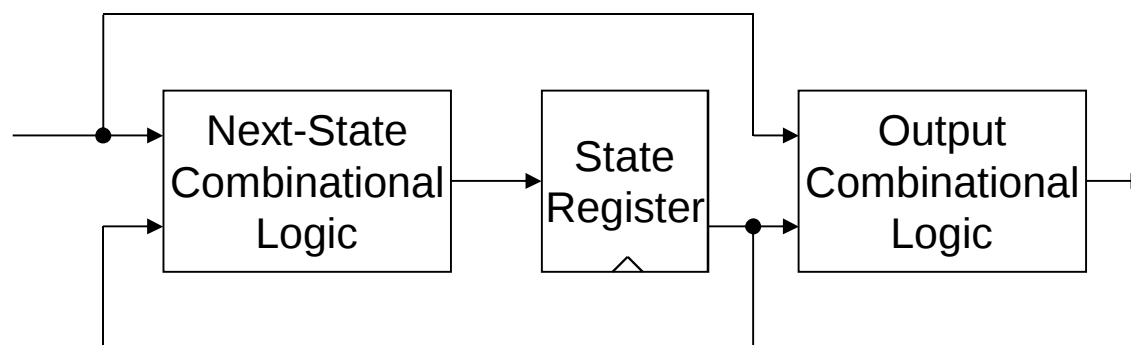
7.8 BCD to Excess-3 Code Converter 실험

- (3) BCD to Excess-3 Code Converter의 State Diagram을 Mealy Machine으로 그리세요. 간략화하지 말고 p.88처럼 15개의 State가 모두 표시되어야 합니다.
- (4) 그려진 State Diagram에서 p.89와 같은 State Transition Table을 그리세요.
- (5) 그려진 State Transition Table을 가지고 p.90~p.95와 같이 State Reduction을 실시하세요.



7.8 BCD to Excess-3 Code Converter 실험

(6) 최종 State Diagram을 Verilog HDL로 표현하세요.
Mealy Machine의 구조는 다음과 같으므로,
always@(...) 블록이 3개가 되어야 합니다.





7.8 BCD to Excess-3 Code Converter 실험

- (7) BCD Code 10개를 입력하여 출력이 제대로 나오는지 확인할 수 있도록 하는 stimulus generator, response monitor, DUTB를 Verilog HDL로 표현하세요.
- (8) SILOS 프로그램을 사용하여 시뮬레이션하고, 출력 F를 시간에 따라 관찰하세요.