




5. 조합논리회로

논리회로 실습

부경대 컴퓨터·인공지능공학부 최필주

- 가산기/감산기
- MUX
- Decoder



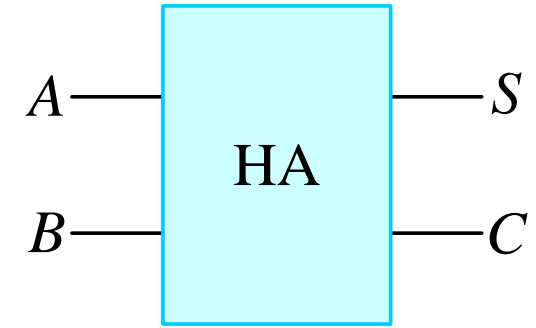
가산기/감산기

가산기(Adder)

- 반가산기(HA, half-adder)

- 두 비트의 덧셈 수행 → 출력: Sum, Carry

$$\begin{array}{r} A \\ + B \\ \hline C \ S \end{array} \quad \begin{array}{r} 0 \\ + 0 \\ \hline 0 \ 0 \end{array} \quad \begin{array}{r} 0 \\ + 1 \\ \hline 0 \ 1 \end{array} \quad \begin{array}{r} 1 \\ + 0 \\ \hline 0 \ 1 \end{array} \quad \begin{array}{r} 1 \\ + 1 \\ \hline 1 \ 0 \end{array}$$



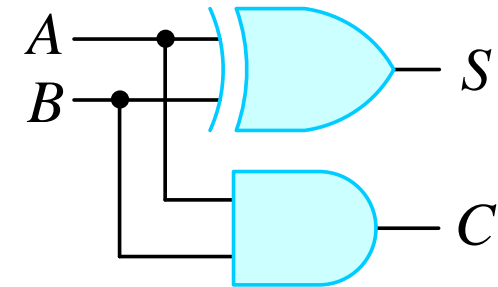
<논리기호>

- 진리표, 논리식, 논리회로

입력		출력	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



$$S = \bar{A}B + A\bar{B} = A \oplus B$$
$$C = A \cdot B$$

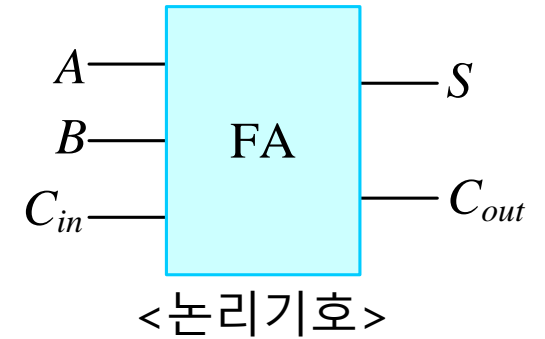


가산기(Adder)

● 전가산기(FA, full-adder)

- 세 비트의 덧셈 수행 → 출력: Sum, Carry

C_{in}	0	1	0	1	0	1	0	1
A	0	0	1	1	0	0	1	1
+ B	+ 0	+ 0	+ 0	+ 0	+ 1	+ 1	+ 1	+ 1
$C_{out} S$	0 0	0 1	0 1	1 0	0 1	1 0	1 0	1 1



- 진리표, 논리식

입력			출력	
A	B	C_{in}	S	C_{out}
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



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

$$S = A \oplus B \oplus C_{in}$$

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

$$C_{out} = AB + BC_{in} + C_{in}A$$

가산기(Adder)

- HA(half-adder) vs. FA(full-adder)

$$\begin{array}{r} 1 \\ + 0 \\ \hline 0 1 \end{array} \quad \begin{array}{r} 1 \\ + 1 \\ \hline 1 0 \end{array}$$

<한 자리 2진수를 더할 때>

$$\begin{array}{r} 1 1 1 \\ 0 0 1 1 \\ + 1 1 0 1 \\ \hline 1 0 0 0 0 \end{array}$$

<두 자리 이상의 2진수를 더할 때>

가산기(Adder)

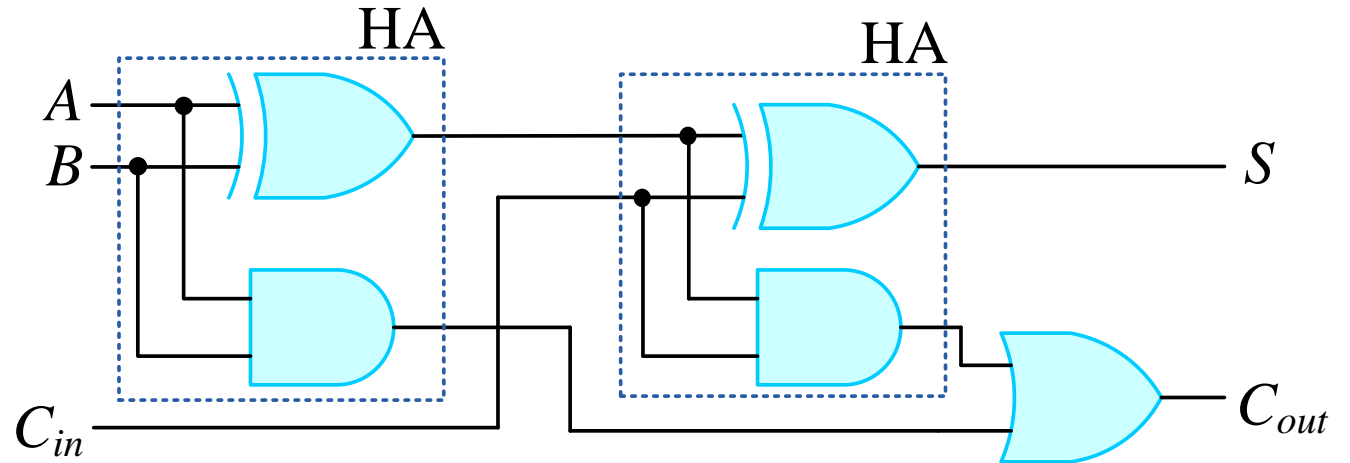
- 전가산기(FA, full-adder)

- 논리식

- $S = A \oplus B \oplus C_{in}$
- $C_{out} = AB + BC_{in} + C_{in}A = AB + C_{in}(A + B) = AB + C_{in}(A \oplus B)$

- 논리회로 구현 2: HA \times 2, 2-input OR

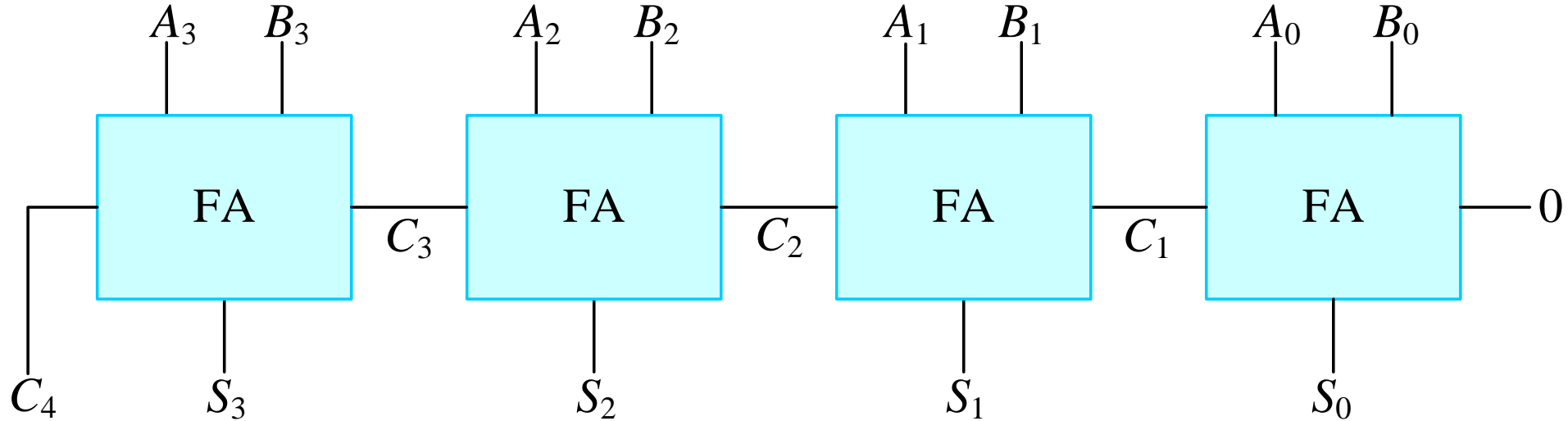
- 1st HA
 - $S' = A \oplus B$
 - $C' = AB$
- 2nd HA
 - $S = S' \oplus C_{in} = (A \oplus B) \oplus C_{in}$
 - $C'' = S' \cdot C_{in} = (A \oplus B) \cdot C_{in}$
- OR
 - $C' + C'' = AB + C_{in}(A \oplus B)$



가산기(Adder)

- 병렬가산기(Parallel-adder)

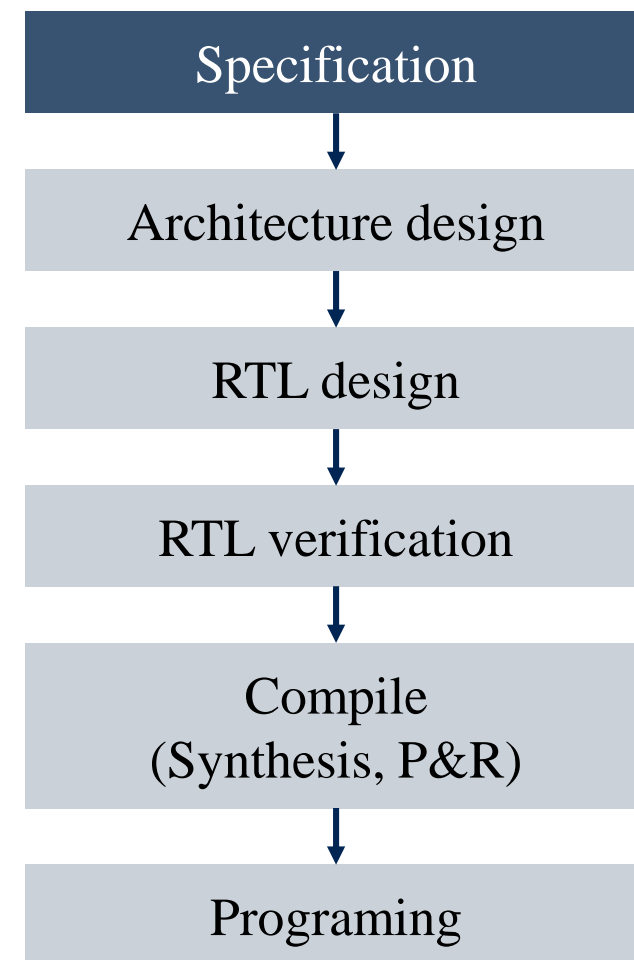
- FA를 여러 개 병렬로 연결
- 2비트 이상의 수를 더할 때 사용
- 예) 4비트 덧셈 $\{C_4S_3S_2S_1S_0\} = \{A_3A_2A_1A_0\} + \{B_3B_2B_1B_0\}$



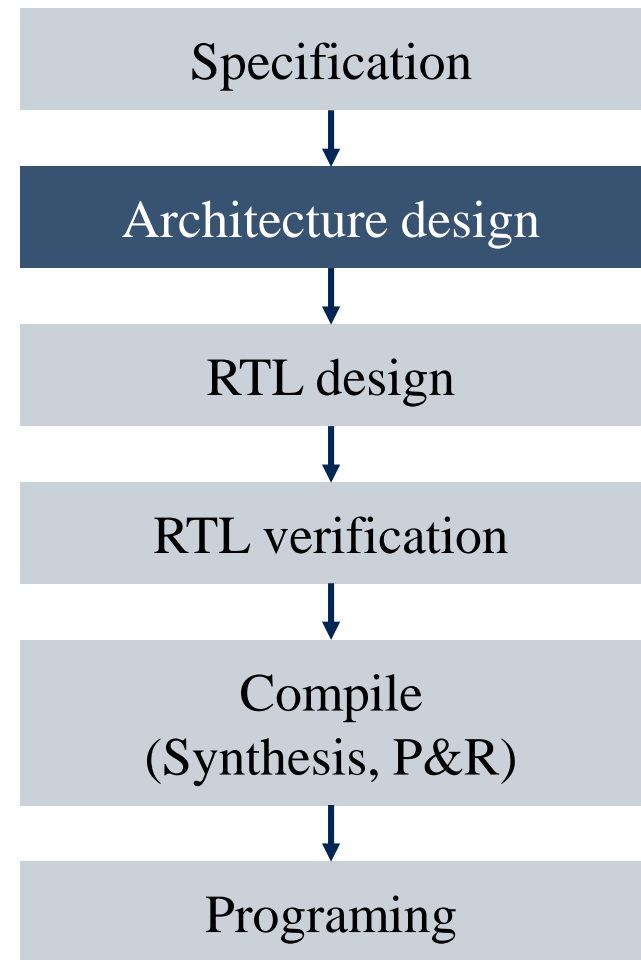
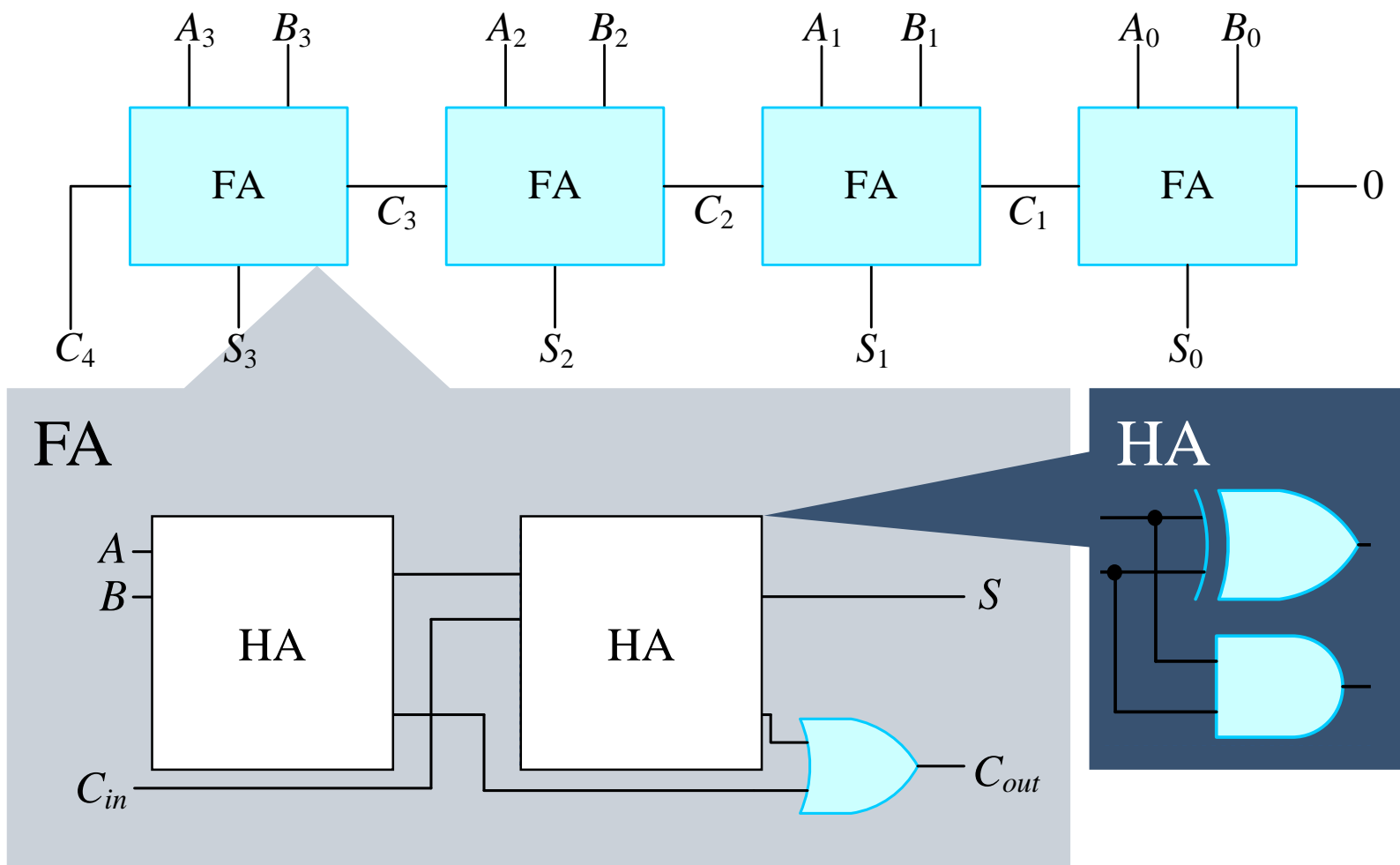
- 수행하려는 기능
 - 4비트 값 두 개를 더함

● 명세

- 입력: $i_A(4), i_B(4) \rightarrow SW$
- 출력: $o_S(4), o_C(1) \rightarrow LED$
- 수행 기능
 - 두 입력을 더한 값을 출력
- 모듈명: Add4b

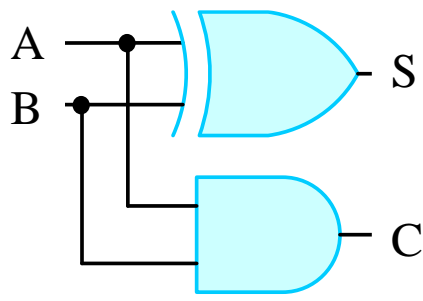


● 구조 설계

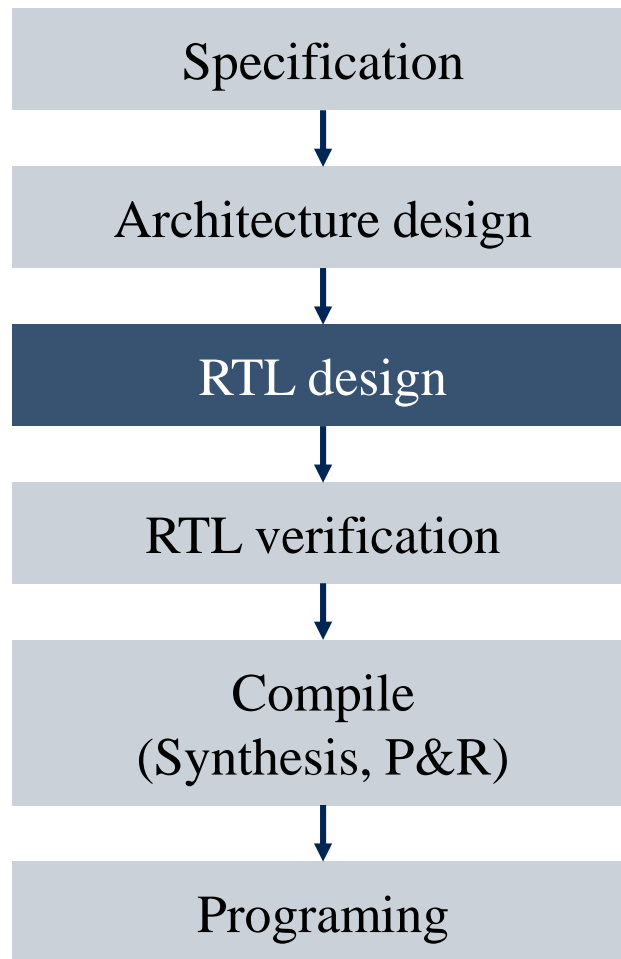


● RTL 설계

■ HA.v

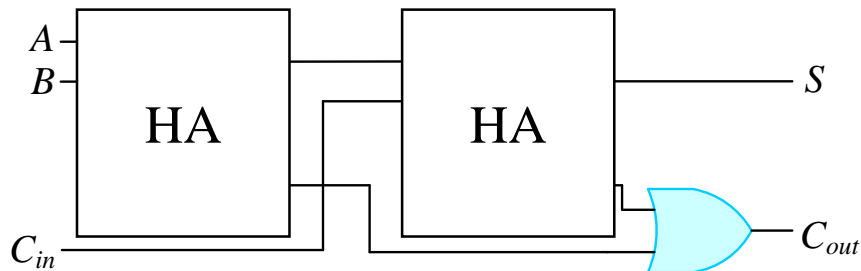


```
module HA(i_A, i_B, o_S, o_C);  
  input i_A, i_B;  
  output o_S;  
  output o_C;  
  
  assign o_S = i_A ^ i_B;  
  assign o_C = i_A & i_B;  
  
endmodule
```



● RTL 설계

■ FA.v



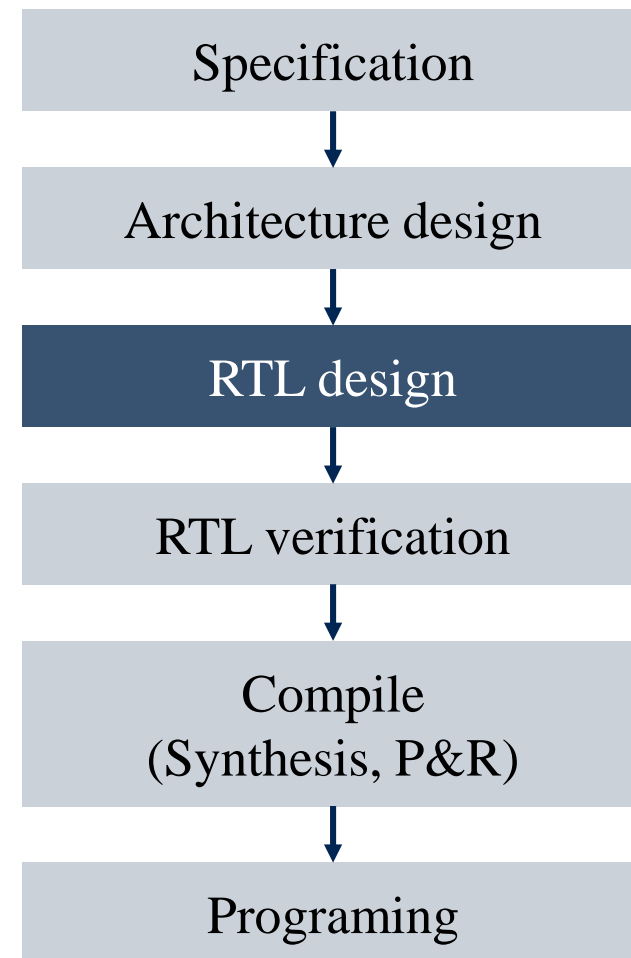
```

module HA(i_A, i_B, i_C, o_S, o_C);
input i_A, i_B, i_C;
output o_S, o_C;
wire HA0_o_S, HA0_o_C;
wire HA1_o_S, HA1_o_C;

HA HA0(i_A, i_B, HA0_o_S, HA0_o_C);
HA HA1(HA0_o_S, i_C, HA1_o_S, HA1_o_C);

assign o_S = HA1_o_S,
       o_C = HA0_o_C | HA1_o_C;

endmodule
    
```



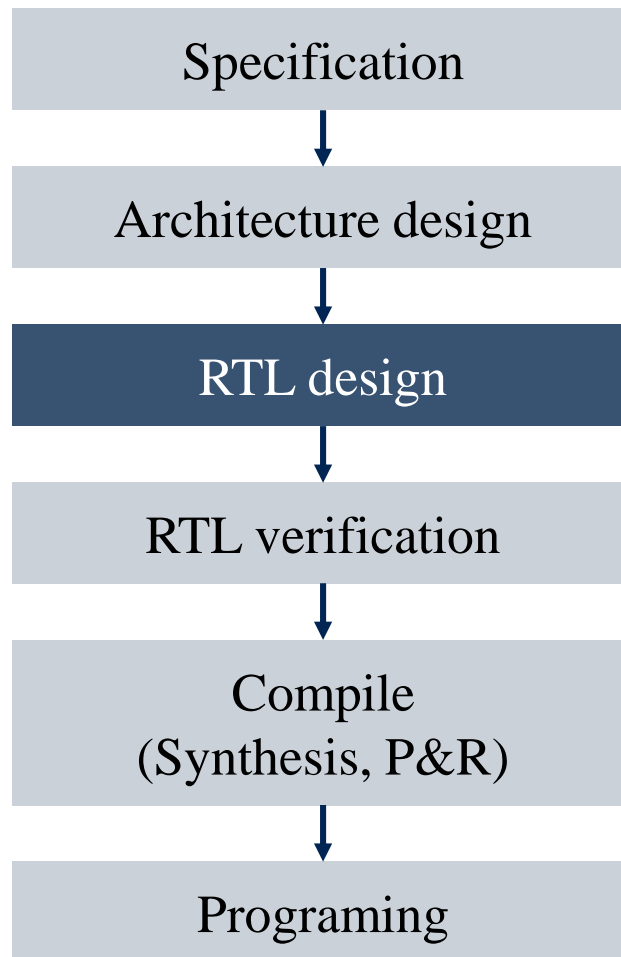
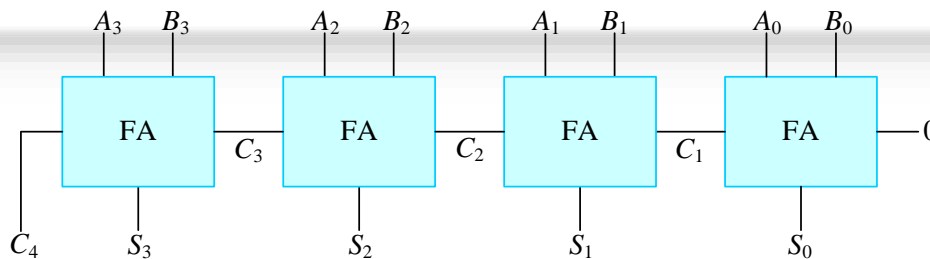
● RTL 설계

■ Add4b.v

```
module Add4b(i_A, i_B, o_S, o_C);
input [3:0] i_A, i_B;
output wire [3:0] o_S;
output wire o_C;
wire [2:0] cout;

FA FA0(i_A[0], i_B[0], 1'b0, o_S[0], cout[0]);
FA FA1(i_A[1], i_B[1], cout[0], o_S[1], cout[1]);
FA FA2(i_A[2], i_B[2], cout[1], o_S[2], cout[2]);
FA FA3(i_A[3], i_B[3], cout[2], o_S[3], o_C);

endmodule
```



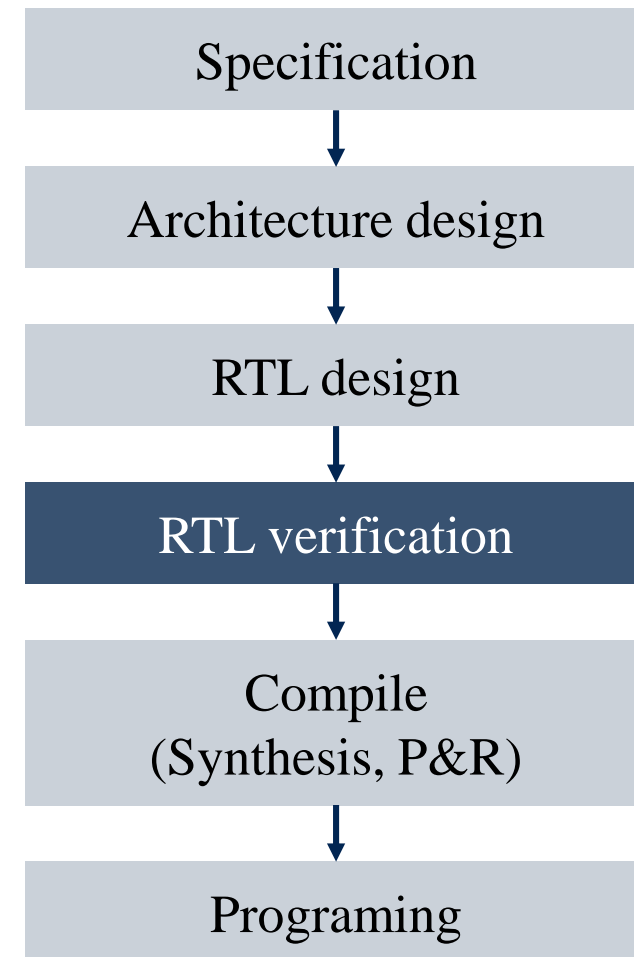
- RTL 검증 - testbench module

- tb_Add4b.v

```
module tb_Add4b;
reg      [3:0]      Add_i_A;
reg      [3:0]      Add_i_B;
reg                      Add_i_C;
wire[3:0]  Add_o_S;
Wire      Add_o_C

Add4b U0(Add_i_A, Add_i_B, Add_o_S, Add_o_C);

initial
Begin
    Add_i_C = 0;
    Add_i_A = 4'b1010; Add_i_B = 4'b1100;
    #10 Add_i_A = 5; Add_i_B = 7;
    #10 Add_i_A = 9; Add_i_B = 8;
endmodule
```



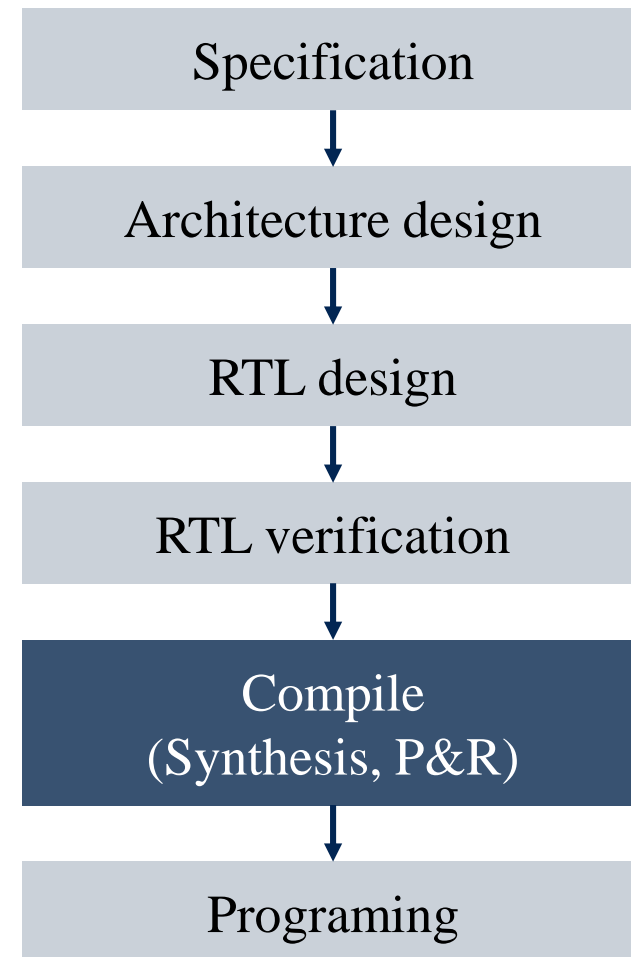
● FPGA 구현 – pin 설정

Table 3-6 Pin Assignments for Slide Switches

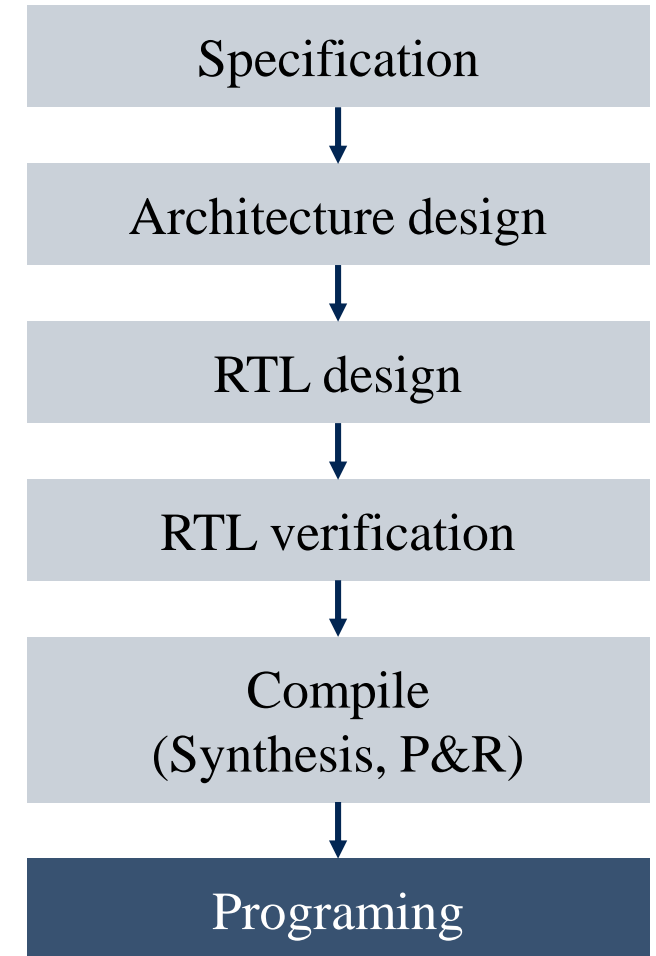
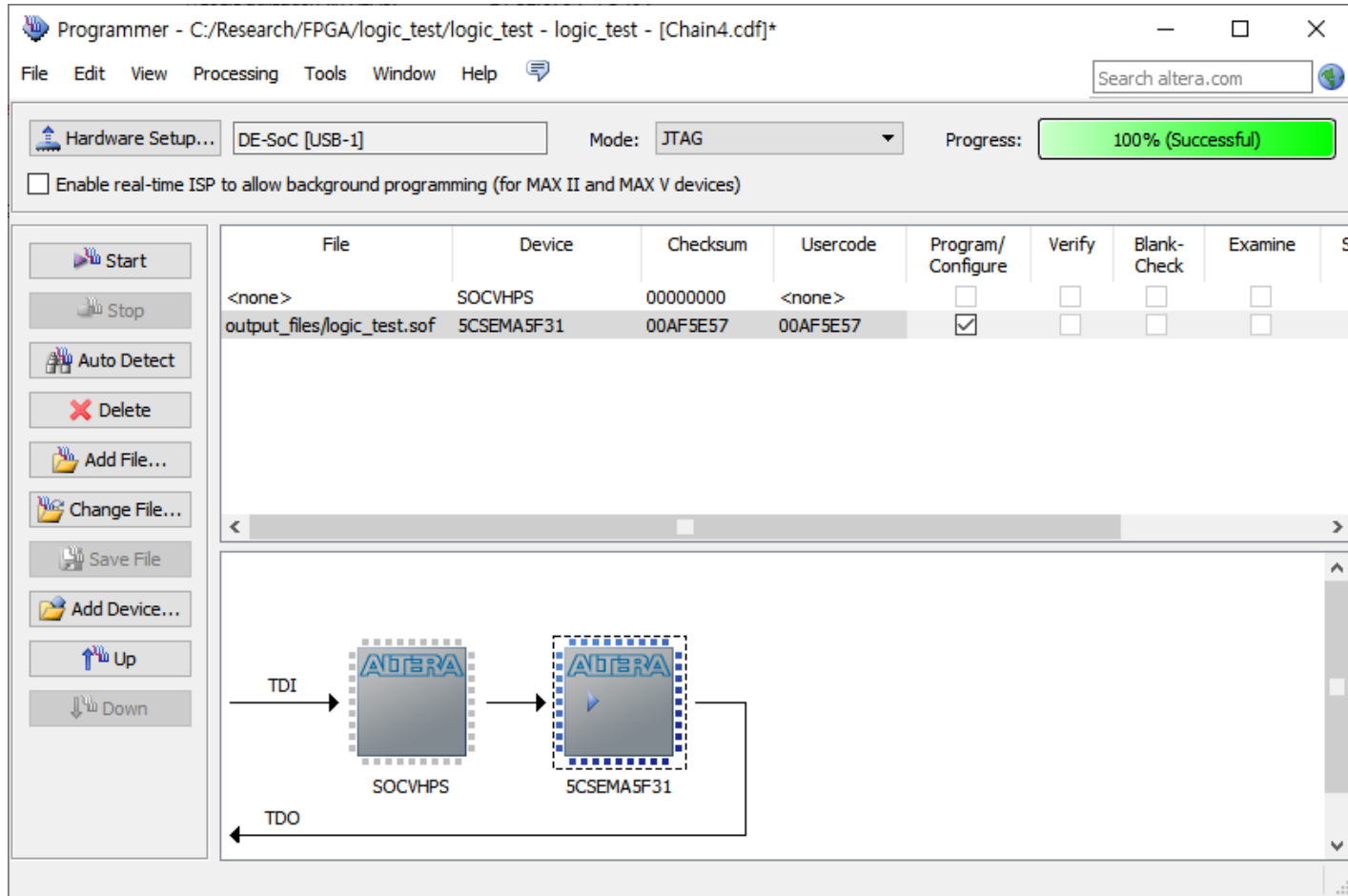
Signal Name	FPGA Pin No.	Description	I/O Standard
SW[0]	PIN_AB12	Slide Switch[0]	3.3V
SW[1]	PIN_AC12	Slide Switch[1]	3.3V
SW[2]	PIN_AF9	Slide Switch[2]	3.3V
SW[3]	PIN_AF10	Slide Switch[3]	3.3V
SW[4]	PIN_AD11	Slide Switch[4]	3.3V
SW[5]	PIN_AD12	Slide Switch[5]	3.3V
SW[6]	PIN_AE11	Slide Switch[6]	3.3V
SW[7]	PIN_AC9	Slide Switch[7]	3.3V
SW[8]	PIN_AD10	Slide Switch[8]	3.3V
SW[9]	PIN_AE12	Slide Switch[9]	3.3V

Table 3-8 Pin Assignments for LEDs

Signal Name	FPGA Pin No.	Description	I/O Standard
LEDR[0]	PIN_V16	LED [0]	3.3V
LEDR[1]	PIN_W16	LED [1]	3.3V
LEDR[2]	PIN_V17	LED [2]	3.3V
LEDR[3]	PIN_V18	LED [3]	3.3V
LEDR[4]	PIN_W17	LED [4]	3.3V
LEDR[5]	PIN_W19	LED [5]	3.3V
LEDR[6]	PIN_Y19	LED [6]	3.3V
LEDR[7]	PIN_W20	LED [7]	3.3V
LEDR[8]	PIN_W21	LED [8]	3.3V
LEDR[9]	PIN_Y21	LED [9]	3.3V



- FPGA 구현
 - Program



- FA를 HA 없이 직접 구현하기

- FA의 논리식

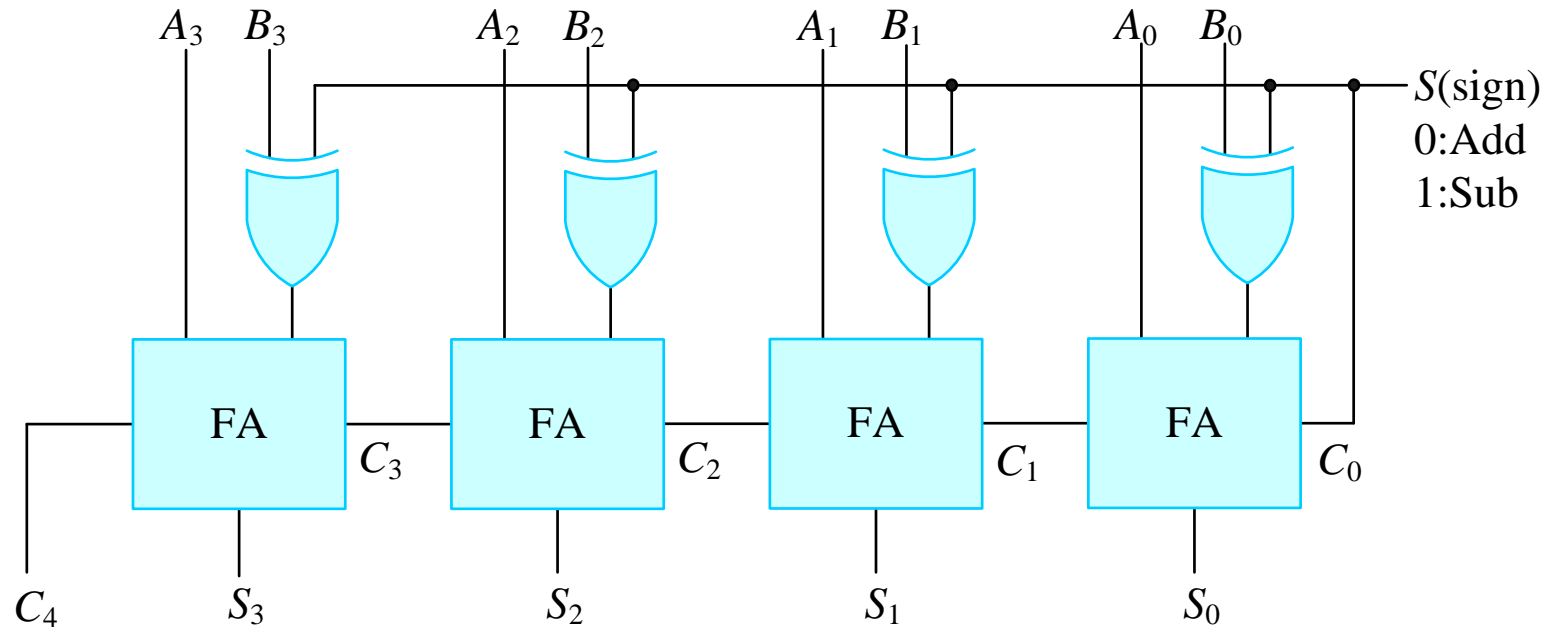
- $S = A \oplus B \oplus C_{in}$

- $C_{out} = AB + BC_{in} + C_{in}A = AB + C_{in}(A + B) = AB + C_{in}(A \oplus B)$

- HA를 사용한 것과 없이 사용한 것 중 어느 것이 더 효과적일까?

감산기(Subtractor)

- 병렬가감산기(Parallel-adder/subtractor)
 - 병렬 가산기에 XOR를 추가하여 감산(뺄셈) 기능 추가



- $S = 0$ 일 때: $\{C_4 S_3 S_2 S_1 S_0\} = \{A_3 A_2 A_1 A_0\} + \{B_3 B_2 B_1 B_0\} + 0$
- $S = 1$ 일 때: $\{C_4 S_3 S_2 S_1 S_0\} = \{A_3 A_2 A_1 A_0\} + \{\overline{B_3} \overline{B_2} \overline{B_1} \overline{B_0}\} + 1$
 $= \{A_3 A_2 A_1 A_0\} + (-\{B_3 B_2 B_1 B_0\})$

● RTL 설계

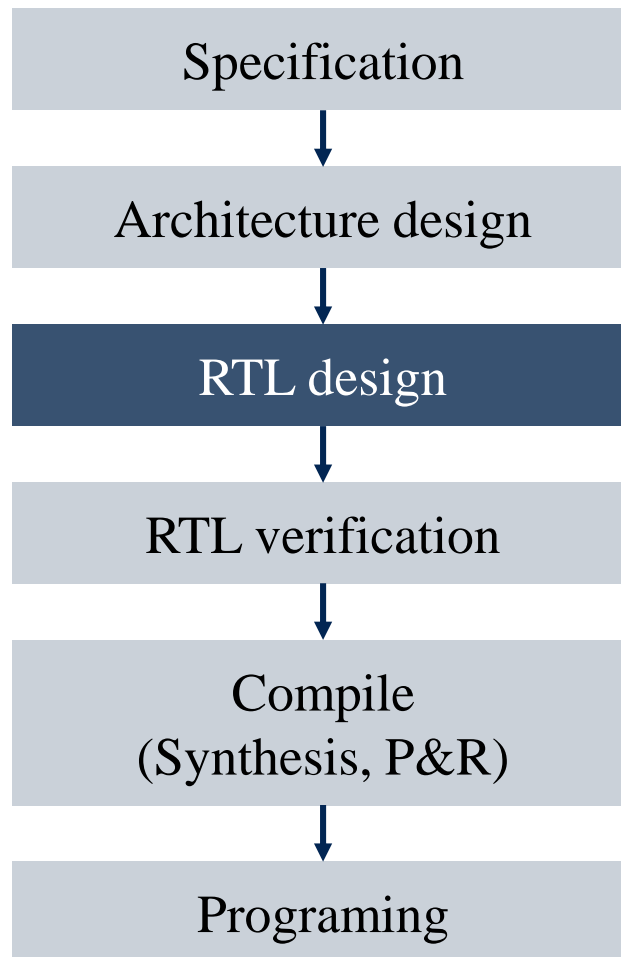
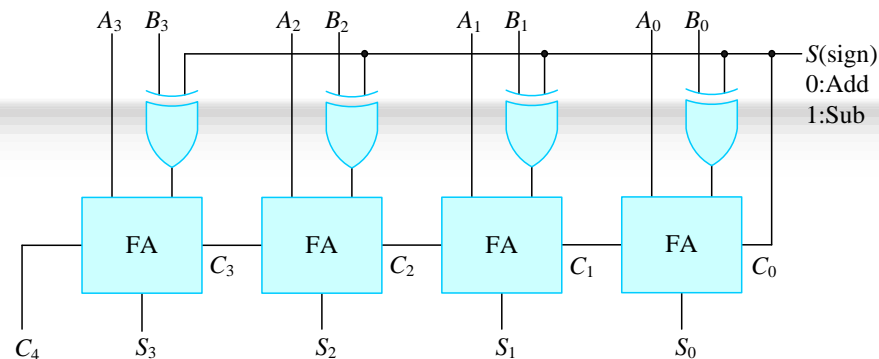
■ AddSub4b.v

```

module AddSub4b(i_A, i_B, i_fSub, o_S, o_C);
input [3:0] i_A, i_B;
input i_fSub;
output wire [3:0] o_S;
output wire o_C;
wire [2:0] cout;

FA HA0(i_A[0], i_B[0] ^ i_fSub, i_fSub, o_S[0], cout[0]);
FA HA1(i_A[1], i_B[1] ^ i_fSub, cout[0], o_S[1], cout[1]);
FA HA2(i_A[2], i_B[2] ^ i_fSub, cout[1], o_S[2], cout[2]);
FA HA3(i_A[3], i_B[3] ^ i_fSub, cout[2], o_S[3], o_C);

endmodule
    
```



산술 연산자

- 산술 연산의 필요성
 - 더하거나 뺄 때마다 가산기/감산기를 설계해야 할까?
- 산술 연산자의 종류

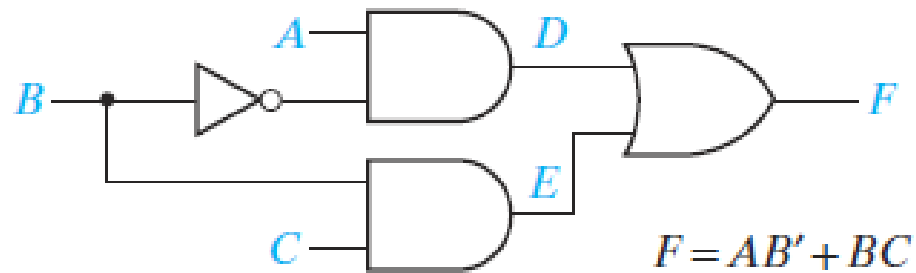
연산자	의미	사용 예	비고
+	덧셈	$\text{add} = a + b$	
-	뺄셈	$\text{sub} = a - b$	
*	곱셈	$\text{mul} = a * b$	FPGA에서는 DSP로 구현됨(* 사용 가능)
/	나눗셈	$\text{div} = a / b$	시뮬레이션에서만 사용 가능
%	나머지(modulus)	$\text{rem} = a \% b$	시뮬레이션에서만 사용 가능

- 가감산기를 산술 연산자를 이용하여 나타내보기



멀티플렉서(Multiplexer, MUX)

- 여러 개의 값 중 하나를 선택하여 출력하는 회로
 - 2×1 MUX: 2개의 값 중 하나를 선택하여 출력



- 조건 연산자로 표현 가능 $\rightarrow F = B ? C : A$

멀티플렉서(Multiplexer, MUX)

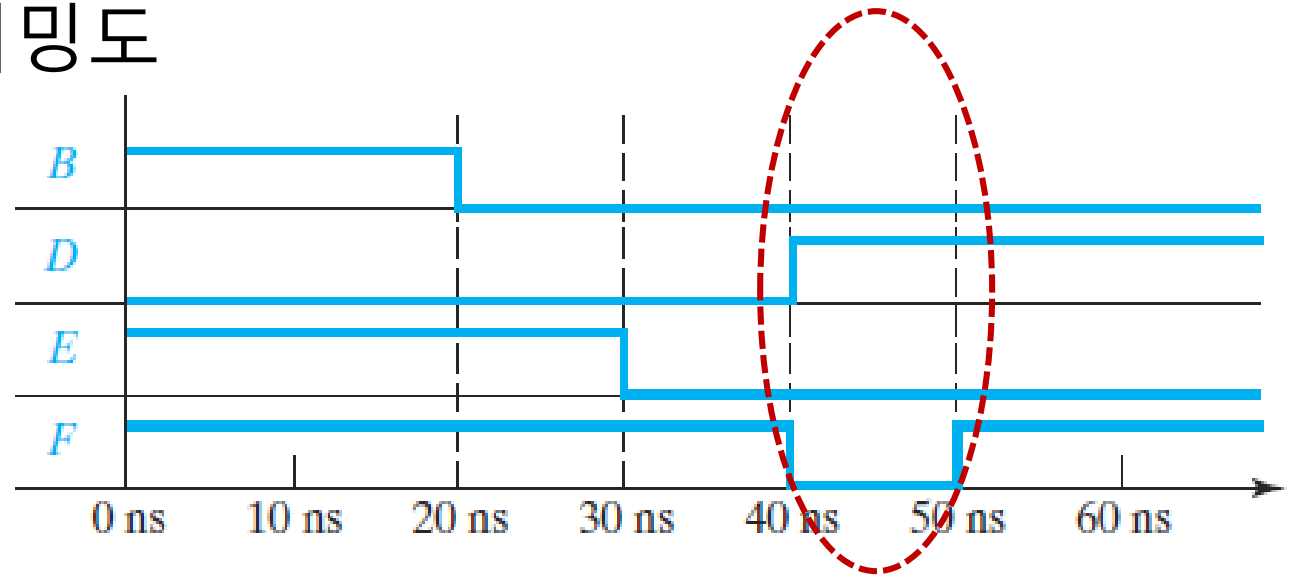
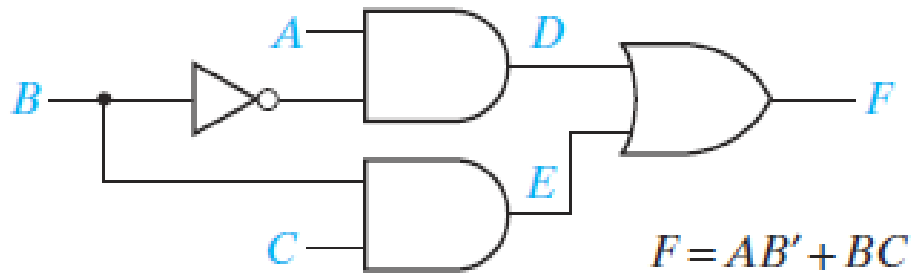
- 고려 사항

- Verilog 표현 $B ? C : A$ 와 $AB' + BC$ 는 동일한가?
 - $B ? C : A$ 로 표현 시: 2x1 MUX 소자로 구성됨
 - $AB' + BC$ 로 표현 시: AND, OR, NOT 게이트들의 조합으로 구성됨

멀티플렉서(Multiplexer, MUX)

- 고려 사항

- Verilog 표현 $B ? C : A$: A와 $AB' + BC$ 는 동일한가?
- $F = AB' + BC$ 의 회로도 및 타이밍도

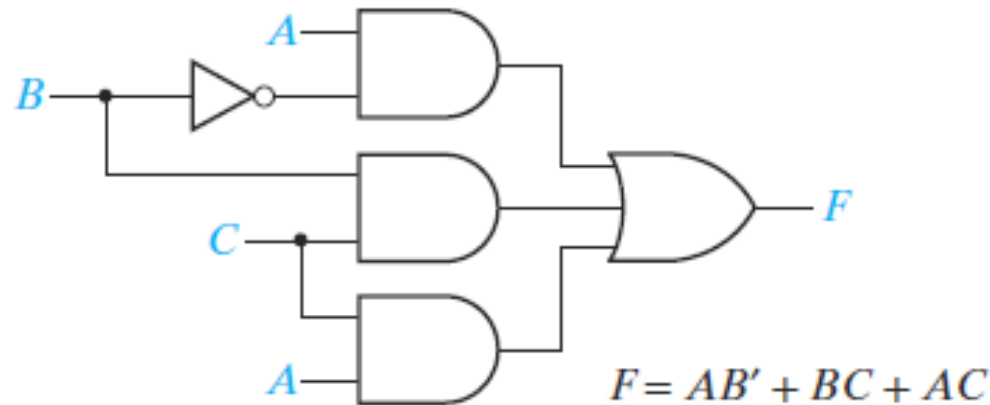


- 타이밍도의 가정
 - 각 gate의 delay는 10ns
 - $A=1, C=1$

멀티플렉서(Multiplexer, MUX)

- 고려 사항

- Verilog 표현 $B ? C : A$ 와 $AB' + BC$ 는 동일한가?
- Glitch를 제거한 MUX



- B와 상관없이 $A=C=1$ 이면 $F=1$ 로 하는 회로 추가

➔ MUX로 표현 가능한 경우(신호 선택)에는 반드시 MUX로 표현

조건 연산자와 if문

- C에서의 표현과 비교

C에서의 비슷한 표현	Verilog 내에서의 표현
result = C ? A : B	assign result = C ? A : B;
if(C) result = A; else result = B;	??

조건 연산자와 if문

- 할당문의 종류에 따른 조건 연산자와 if문 사용법

구분	조건 연산자 예시	동일 표현
연속 할당문 (continuous assignment)	<code>assign result = C ? A : B;</code>	-
절차형 할당문 (procedural assignment)	<code>always@(A, B, C) result = C ? A : B</code>	<code>always@(A, B, C) if(C) result = A; else result = B;</code>

조건 연산자와 if문

- 연속 할당문 vs. 절차형 할당문

구분	예시
연속 할당문 (continuous assignment)	<code>assign A = B;</code> <code>assign A = C;</code>
절차형 할당문 (procedural assignment)	<code>always@(B, C) begin</code> <code>A = B;</code> <code>A = C;</code> <code>End</code>

조건 연산자와 if문

- 연속 할당문 우선 순위 예시

	표현1	표현2
예시1	<pre>always@(B, C) begin A = B; B = C; end</pre>	<pre>always@(B, C) begin B = C; A = B; end</pre>
예시2	<pre>always@(S, B, C) begin A = B; if (S) A = C; end</pre>	<pre>always@(S, B, C) begin if(S) A = C; else A = B; end</pre>

조건 연산자와 if문

- 할당문의 종류에 따른 조건 연산자와 if문 사용법

구분	조건 연산자 예시	동일 표현	
연속 할당문 (continuous assignment)	<code>assign result = C ? A : B;</code>	-	
절차형 할당문 (procedural assignment)	<code>always@(A, B, C) result = C ? A : B</code>	<code>always@(A, B, C) if(C) result = A; else result = B;</code>	<code>always@(A, B, C) begin result = B; if(C) result = A; end</code>

- 2-input MUX

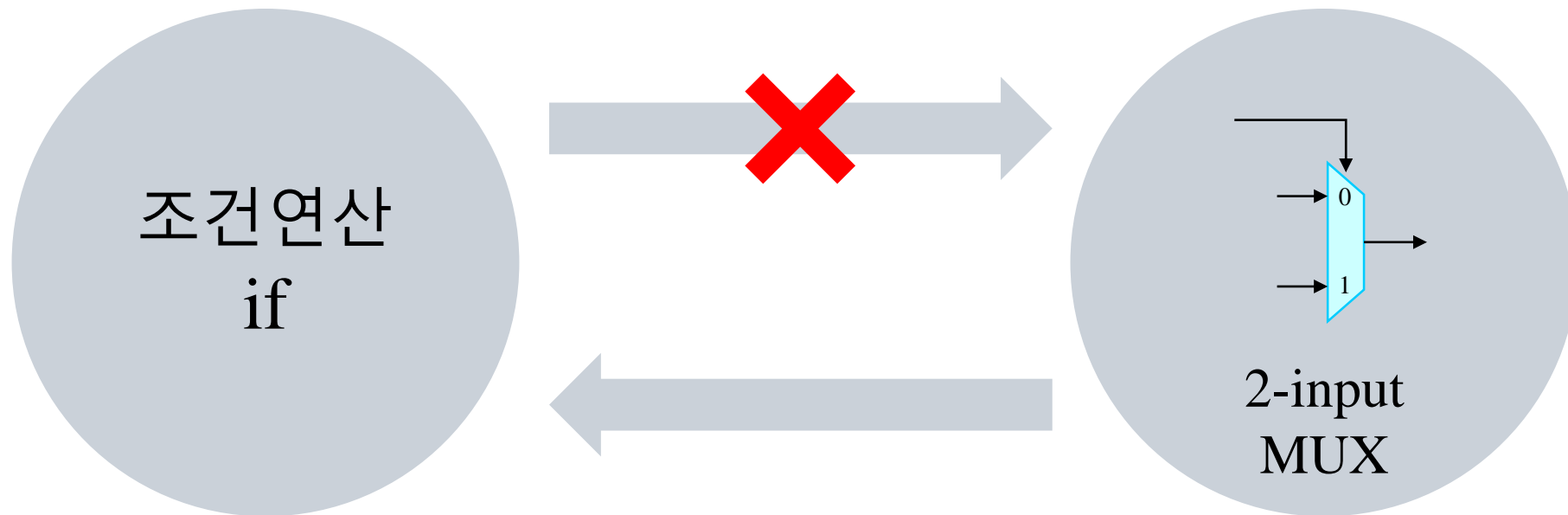
- 연속/절차형 할당문 내에서 조건 연산자로 표현: $\text{result} = C ? A : B;$
- 절차형 할당문 내에서 if문으로 표현: $\text{if}(C) \text{ result} = A; \text{ else result} = B;$

- 4-input, 8-input MUX는?

● 4-input mux의 표현

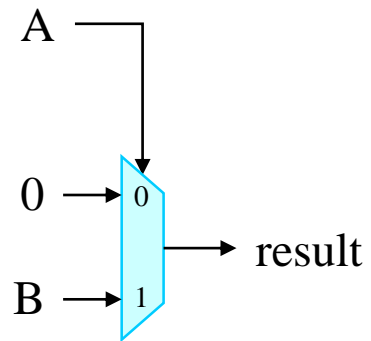
Verilog 표현 예시	생성되는 하드웨어 로직
<pre> wire [1:0] i0, i1, i2, i3; wire [1:0] sel; reg [1:0] out; always@(i0, i1, sel) case(sel) 2'b00: out = i0; 2'b01: out = i1; 2'b10: out = i2; default: out = i3; endcase </pre>	<p>The diagram shows two 4-input multiplexers. The top multiplexer has four inputs labeled i0[0], i1[0], i2[0], and i3[0], and a select line labeled sel. Its output is labeled out[0]. The bottom multiplexer has four inputs labeled i0[1], i1[1], i2[1], and i3[1], and a select line labeled sel. Its output is labeled out[1].</p>

- MUX와 조건연산/if문



- MUX는 조건연산이나 if문으로 표현해야 함
- 조건연산이나 if문은 MUX가 아닌 다른 회로를 의미할 수도 있음

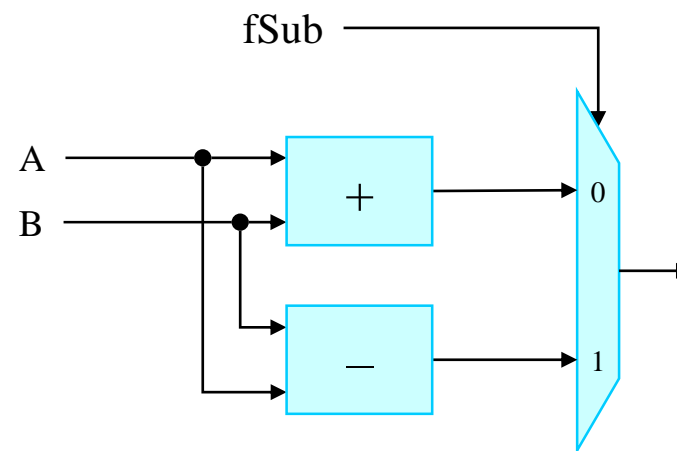
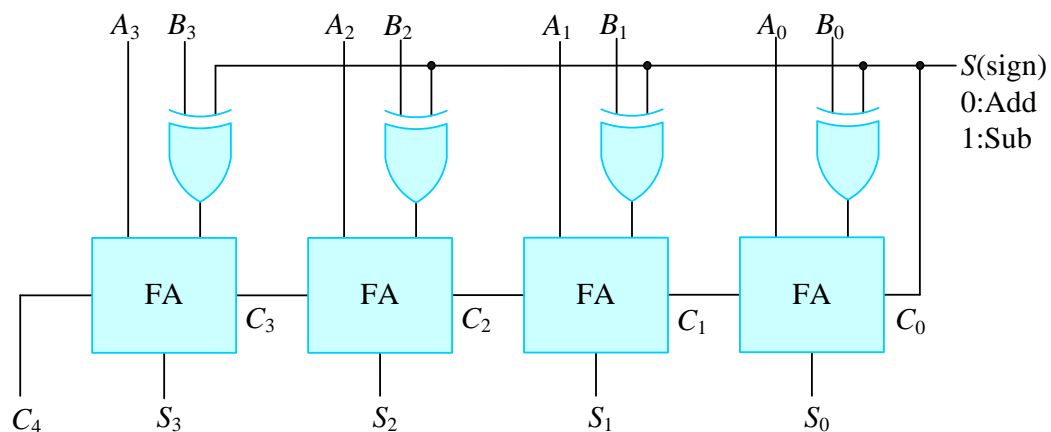
- 예시 - gating
 - Verilog 표현: $\text{result} = A ? B : 0$
 - MUX를 이용해서 구현될까?



- 예시 - 가감산기

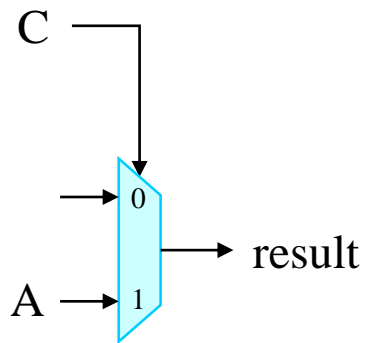
- Verilog 표현: $S = fSub ? A - B : A + B$

- 가감산기 하나로 구현될까? MUX를 이용해서 구현될까?



- 예시 - latch

- Verilog 표현: `if(C) result = A;`
 - else에 대해서는 정의X
- 다음과 같이 MUX로 구현될까?



● 예시 - latch

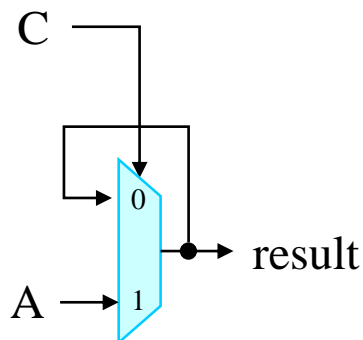
- Verilog 표현: `if(C) result = A;`
 - else에 대해서는 정의X → 값을 유지하는 것으로 간주
 - 동일한 표현

```
always@(A, C)
  if(C) result = A;
```



```
always@(A, C)
  if(C) result = A;
  else result = result;
```

- 다음과 같이 MUX로 구현될까?



● 예시 - latch

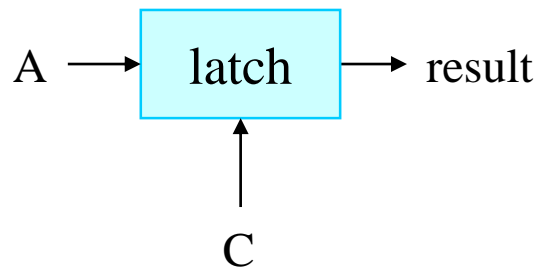
- Verilog 표현: `if(C) result = A;`
 - else에 대해서는 정의X → 값을 유지하는 것으로 간주
 - 동일한 표현

```
always@(A, C)  
  if(C) result = A;
```



```
always@(A, C)  
  if(C) result = A;  
  else result = result;
```

- 실제 구현: latch로 구현됨 ... enable 신호가 1일 때에만 값 변화



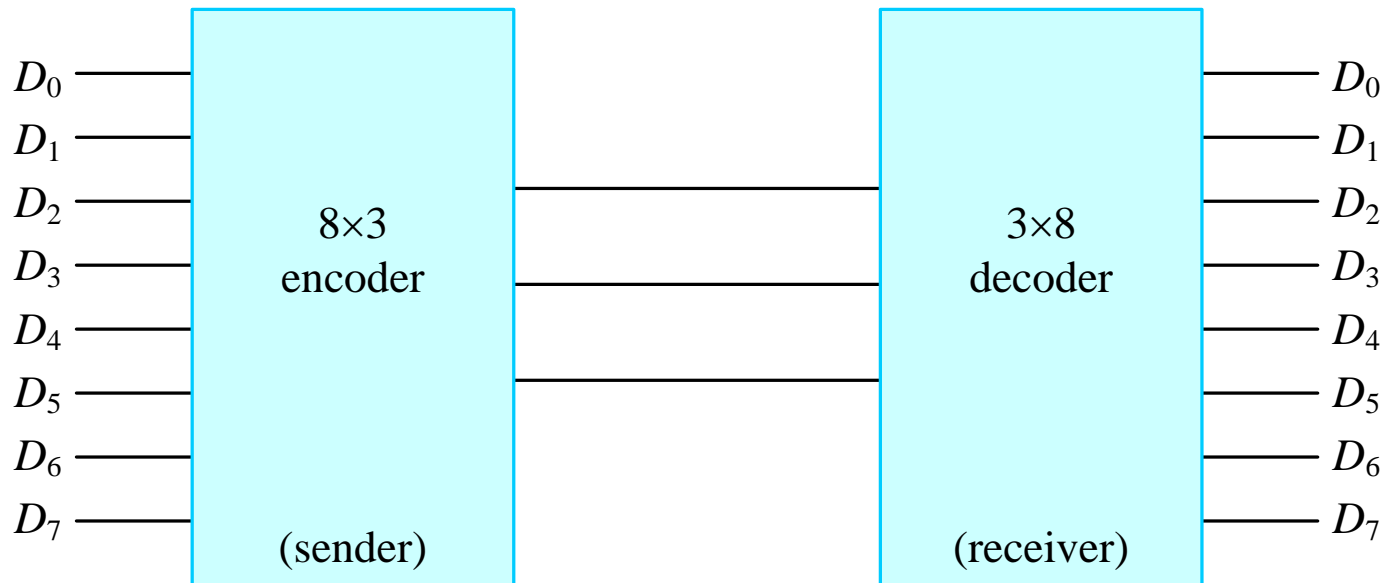


Decoder

인코더(Encoder)와 디코더(Decoder)

- n 비트 2진 코드 $\leftrightarrow 2^n$ 개의 정보

- Encoder vs. Decoder



- Encoder: $D_0 \sim D_7$ 중 on된 bit의 위치를 2진수로 출력
- Decoder: $D_0 \sim D_7$ 중 2진수 입력이 가리키는 곳을 on

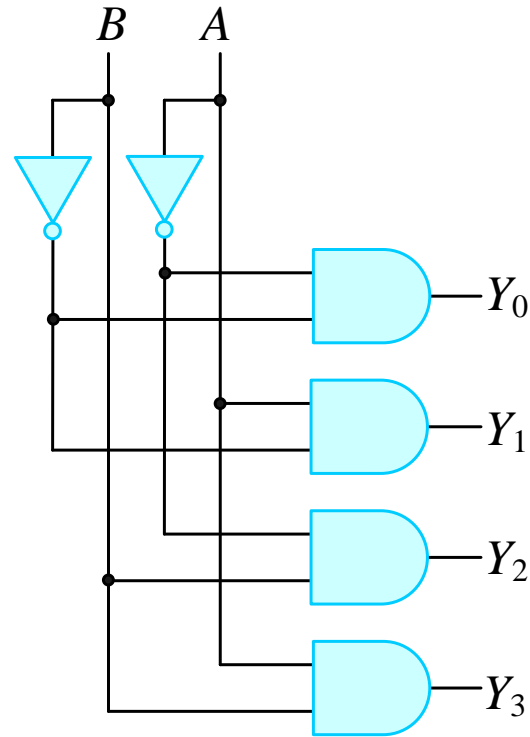
디코더(Decoder)

- 2×4 디코더

- 논리식: $Y_0 = \bar{B}\bar{A}$, $Y_1 = \bar{B}A$, $Y_2 = B\bar{A}$, $Y_3 = BA$

- 진리표와 논리회로

입력		출력			
B	A	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0



디코더(Decoder)

- 2×4 디코더

- 논리식: $Y_0 = \bar{B}\bar{A}$, $Y_1 = \bar{B}A$, $Y_2 = B\bar{A}$, $Y_3 = BA$

- 진리표와 논리회로

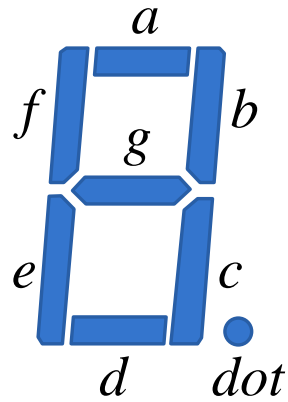
입력		출력			
<i>B</i>	<i>A</i>	<i>Y</i> ₃	<i>Y</i> ₂	<i>Y</i> ₁	<i>Y</i> ₀
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

verilog

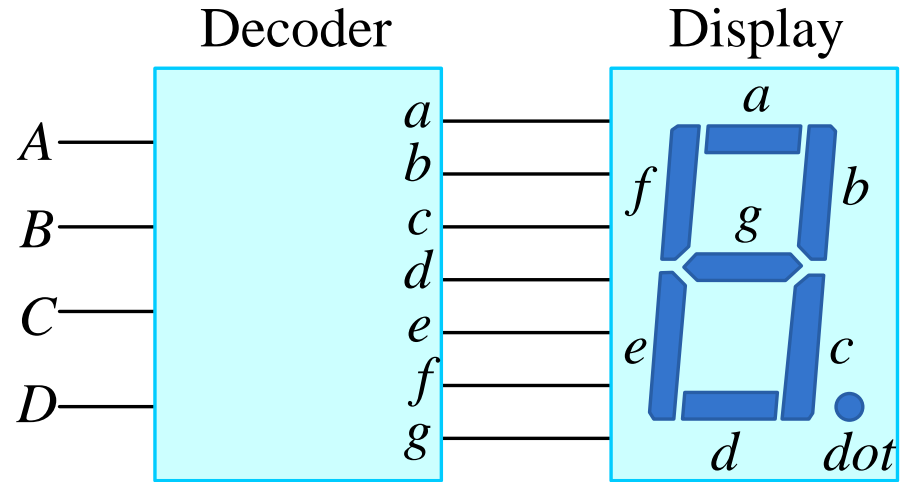
```
wire [1:0] sel;  
reg [3:0] out;  
always@(i0, i1, sel)  
    case(sel)  
        2'b00: out = 4'b0001;  
        2'b01: out = 4'b0010;  
        2'b10: out = 4'b0100;  
        default: out = 4'b1000;  
    endcase
```

디코더(Decoder)

- 7-Segment 디코더



<7-세그먼트 구성>



<7-세그먼트와 디코더의 연결>

0	1	2	3	4	5	6	7	8	9

디코더(Decoder)

● 7-Segment 디코더

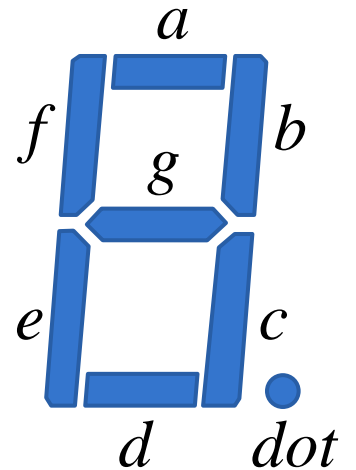
■ 진리표

입력				출력						
D	C	B	A	\bar{a}	\bar{b}	\bar{c}	\bar{d}	\bar{e}	\bar{f}	\bar{g}
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0
0	0	1	1	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	1	0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1	1	0	0
1	0	1	0	x	x	x	x	x	x	x
1	0	1	1	x	x	x	x	x	x	x
1	1	0	0	x	x	x	x	x	x	x
1	1	0	1	x	x	x	x	x	x	x
1	1	1	0	x	x	x	x	x	x	x
1	1	1	1	x	x	x	x	x	x	x

```

wire [3:0] sel;
reg [6:0] out;
always@*
    case(sel)
        4'h0 : o_FND = 7'b1000000;
        4'h1 : o_FND = 7'b1111001;
        4'h2 : o_FND = 7'b0100100;
        4'h3 : o_FND = 7'b0110000;
        4'h4 : o_FND = 7'b0011001;
        4'h5 : o_FND = 7'b0010010;
        4'h6 : o_FND = 7'b0000010;
        4'h7 : o_FND = 7'b1011000;
        4'h8 : o_FND = 7'b0000000;
        4'h9 : o_FND = 7'b0011000;
    endcase

```



<DE1-SoC에서의 번호>

디코더(Decoder)

● 7-Segment 디코더

■ 진리표

입력				출력						
D	C	B	A	\bar{a}	\bar{b}	\bar{c}	\bar{d}	\bar{e}	\bar{f}	\bar{g}
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0
0	0	1	1	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	1	0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1	1	0	0
1	0	1	0	x	x	x	x	x	x	x
1	0	1	1	x	x	x	x	x	x	x
1	1	0	0	x	x	x	x	x	x	x
1	1	0	1	x	x	x	x	x	x	x
1	1	1	0	x	x	x	x	x	x	x
1	1	1	1	x	x	x	x	x	x	x

BA DC	00	01	11	10
00		1		
01	1			1
11	x	x	x	x
10			x	x

$$\bar{a} = \overline{DCBA} + C\bar{A}$$

BA DC	00	01	11	10
00				
01		1		1
11	x	x	x	x
10			x	x

$$\bar{b} = \overline{CBA} + C\bar{B}\bar{A} = C(B \oplus A)$$

BA DC	00	01	11	10
00				1
01				
11	x	x	x	x
10			x	x

$$\bar{c} = \overline{CBA}$$

BA DC	00	01	11	10
00		1		
01	1		1	
11	x	x	x	x
10		1	x	x

$$\bar{d} = \overline{CBA} + \overline{CBA} + CBA$$

디코더(Decoder)

● 7-Segment 디코더

■ 진리표

입력				출력						
D	C	B	A	\bar{a}	\bar{b}	\bar{c}	\bar{d}	\bar{e}	\bar{f}	\bar{g}
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0
0	0	1	1	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	1	0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1	1	0	0
1	0	1	0	x	x	x	x	x	x	x
1	0	1	1	x	x	x	x	x	x	x
1	1	0	0	x	x	x	x	x	x	x
1	1	0	1	x	x	x	x	x	x	x
1	1	1	0	x	x	x	x	x	x	x
1	1	1	1	x	x	x	x	x	x	x

BA DC	00	01	11	10
00		1	1	
01	1	1	1	
11	x	x	x	x
10		1	x	x

$$\bar{e} = A + C\bar{B}$$

BA DC	00	01	11	10
00		1	1	1
01			1	
11	x	x	x	x
10			x	x

$$\bar{f} = BA + \bar{C}\bar{B} + \bar{D}\bar{C}A$$

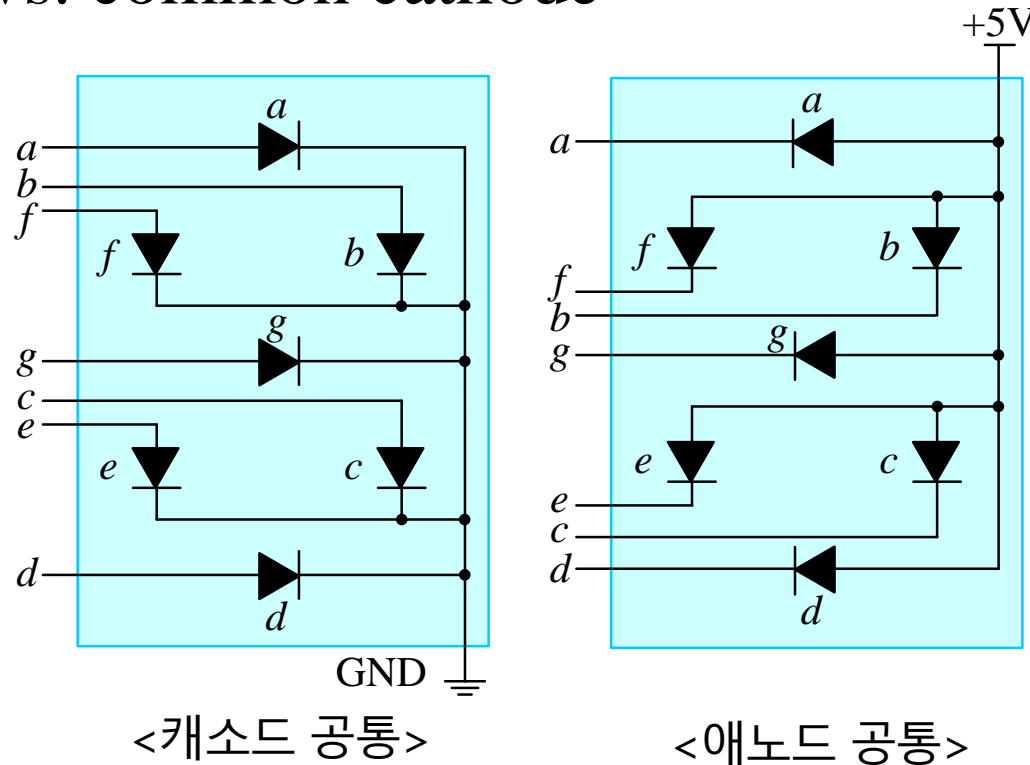
BA DC	00	01	11	10
00	1	1		
01			1	
11	x	x	x	x
10			x	x

$$\bar{g} = \bar{D}\bar{C}\bar{B} + CBA$$

디코더(Decoder)

- 7-Segment 디코더

- Common anode vs. common cathode



- DE1-SoC는 common anode 방식: FPGA에서 0 신호를 주어야 켜짐

- 구현하고자 하는 기능
 - 스위치 4개로 표현한 2진수를 Seven-segment로 표시하기
 - 입력: i_Num(3)
 - 출력: o_FND(6)
 - 모듈명: FND
 - 파일명: FND.v
- 제출 파일
 - FND.v, tb_FND.v, 시뮬레이션 파형, FPGA 동작 동영상