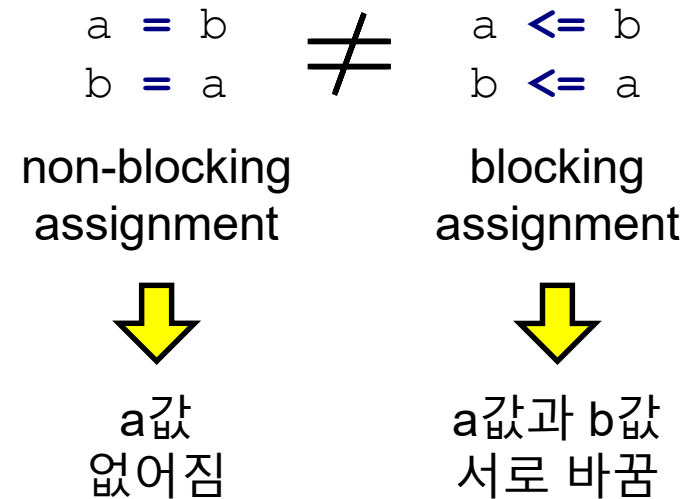


Lecture 03~04

# 조합회로 설계

# 조합회로

- 메모리(또는 상태)를 포함하지 않음
- 같은 입력 → 같은 출력
- Verilog 설계 시 사용 가능 문장
  - assign
  - always @\*  
begin  
    non-blocking assignment;  
    non-blocking assignment;  
    ...  
end



# 연산 공유(operator sharing)



## 0.55- $\mu\text{m}$ CMOS standard-cell technology

비트 크기	자주 쓰는 연산자									
	nand	xor	$>_a$	$>_d$	=	$+1_a$	$+1_d$	$+_a$	$+_d$	MUX
8	8	22	25	68	26	27	33	51	118	21
16	16	44	52	102	51	55	73	101	265	42
32	32	85	105	211	102	113	153	203	437	85
64	64	171	212	398	204	227	313	405	755	171
8	0.1	0.4	4.0	1.9	1.0	2.4	1.5	4.2	3.2	0.3
16	0.1	0.4	8.6	3.7	1.7	5.5	3.3	8.2	5.5	0.3
32	0.1	0.4	17.6	6.7	1.8	11.6	7.5	16.2	11.1	0.3
64	0.1	0.4	35.7	14.3	2.2	24.0	15.7	32.2	22.9	0.3

}

Area  
(gate count)

}

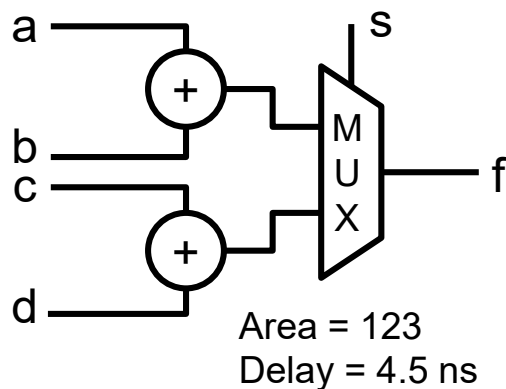
Delay  
(ns)

# 연산 공유(operator sharing)



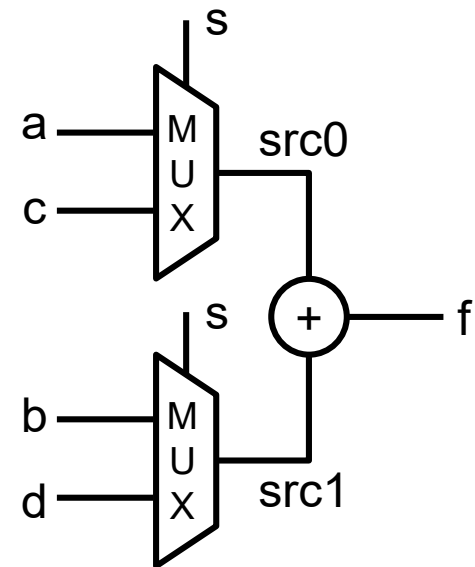
- 같은 연산을 여러 번 사용하면 → 연산 공유
  - 회로 크기 줄임
  - 전파 지연 시간(propagation delay time) 줄일 수 있음

```
if (s == 1)
    f = a + b
else
    f = c + d
```



```
if (s == 1) begin
    src0 = a;
    src1 = b;
end
else begin
    src0 = c;
    src1 = d;
end
```

```
assign f = src0 + src1;
```

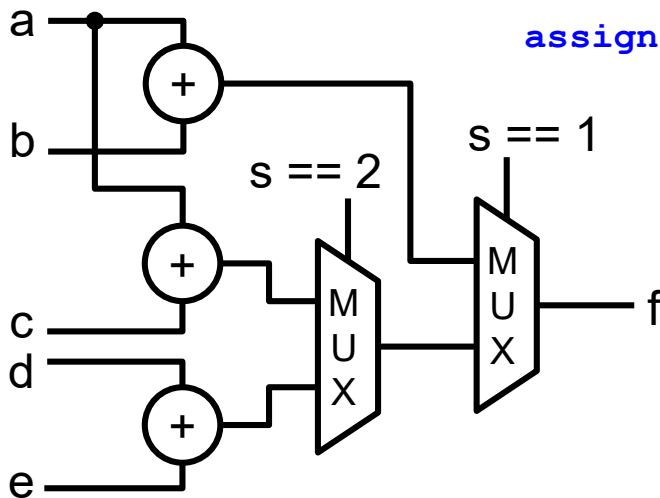


# 연산 공유(operator sharing)



```
if (s == 1)
  f = a + b;
else if (s == 2)
  f = a + c;
else
  f = d + e;
```

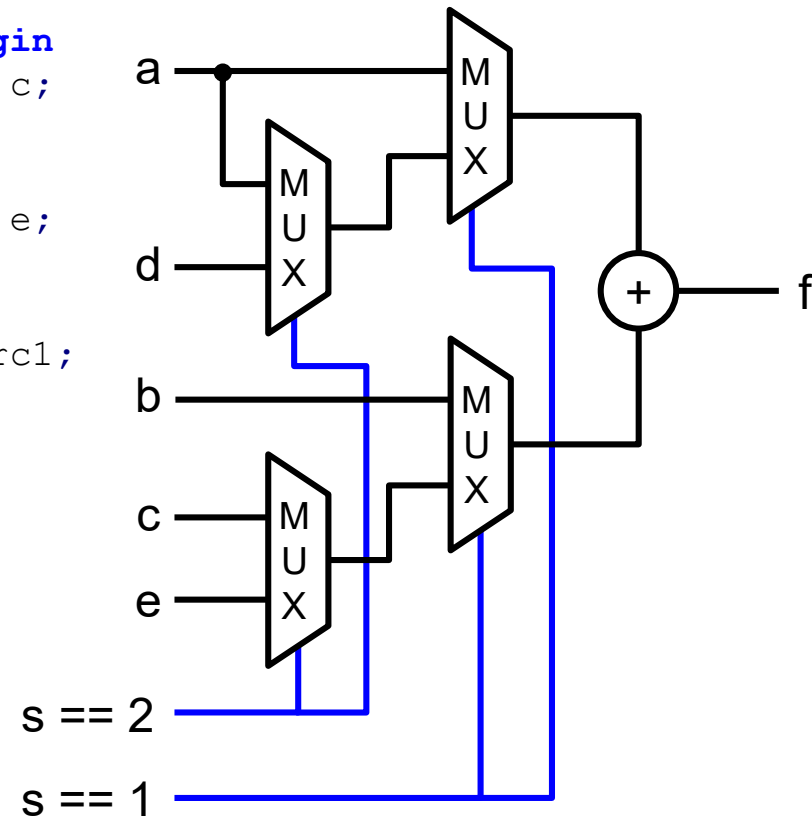
Area = 195  
Delay = 4.8 ns



```
if (s == 1) begin
  src0 = a; src1 = b;
end
else if (s == 2) begin
  src0 = a; src1 = c;
end
else begin
  src0 = d; src1 = e;
end

assign f = src0 + src1;
```

Area = 135  
Delay = 4.8 ns



# 연산 공유(operator sharing)

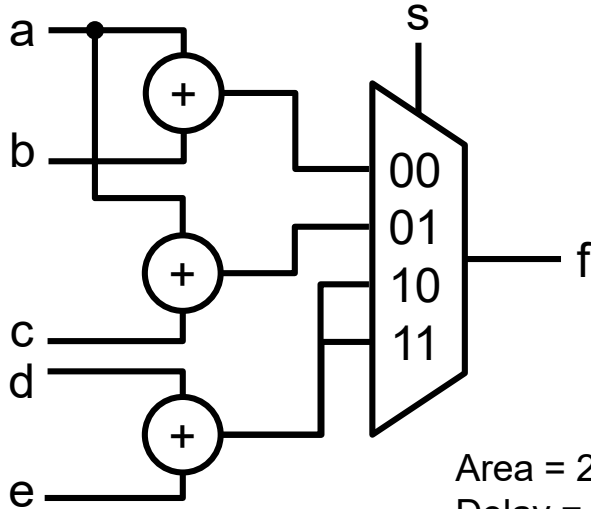


```
case (s)
  2'b00: f = a + b;
  2'b01: f = a + c;
  default: f = d + e;
endcase
```

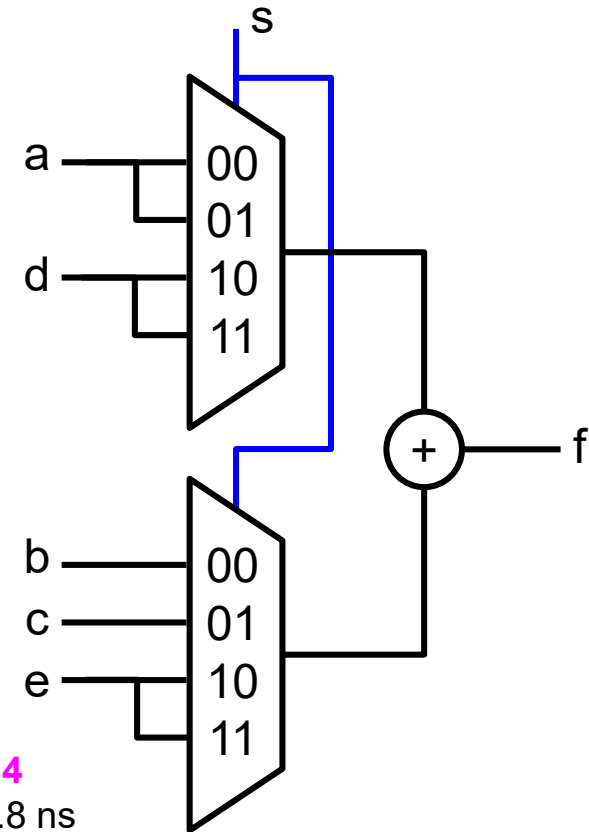


```
case (s)
  2'b00: begin
    src0 = a; src1 = b;
  end
  2'b01: begin
    src0 = a; src1 = c;
  end
  default: begin
    src0 = d; src1 = e;
  end
endcase
```

```
assign f = src0 + src1;
```



Area = 216  
Delay = 4.8 ns



Area = 114  
Delay = 4.8 ns

# 연산 공유(operator sharing)



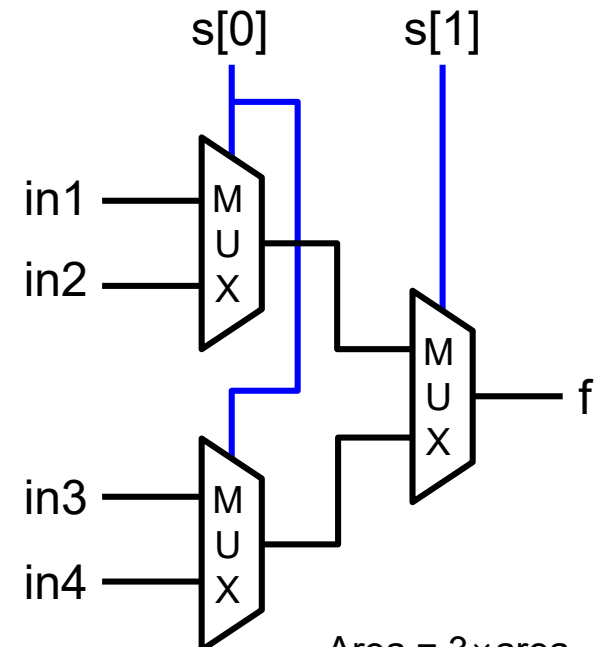
- 4-to-1 MUX의 area 및 delay 계산?

s[1]	s[0]	f
0	0	in1
0	1	in2
1	0	in3
1	1	in4

2-to-1 MUX



s[1]	f
0	MUX1의 출력
1	MUX2의 출력



$$\text{Area} = 3 \times \text{area}_{\text{MUX}}$$

$$\text{Delay} = 2 \times \text{delay}_{\text{MUX}}$$

# 연산 공유(operator sharing)

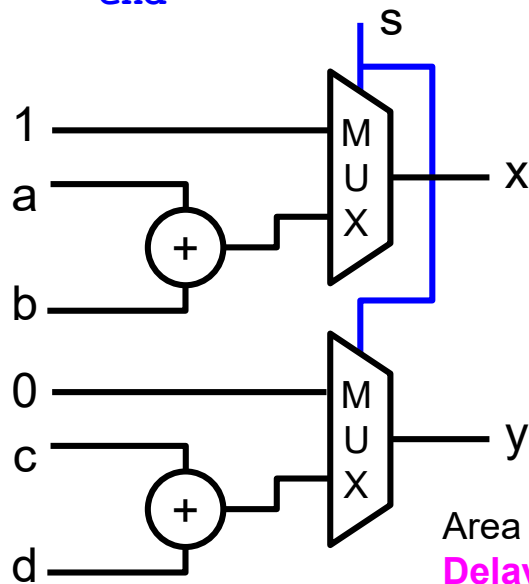


```
if s begin
  x = a + b;
  y = 0;
end
else begin
  x = 1;
  y = c + d;
end
```

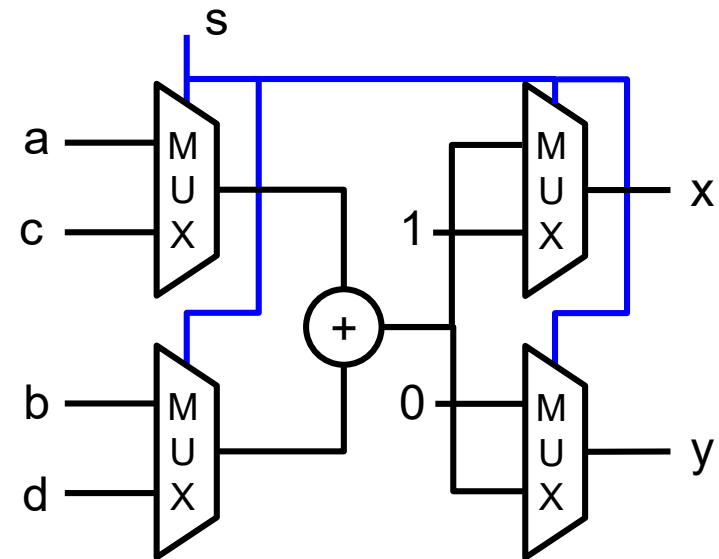


```
if s begin
  src0 = a;
  src1 = b;
  x = sum;
  y = 0;
end
else begin
  src0 = c;
  src1 = d;
  x = 1;
  y = sum;
end
```

**assign** sum = src0 + src1;



Area = 144  
Delay = 4.5 ns



Area = 135  
Delay = 4.8 ns

**이 경우는 연산 공유 할 필요 없음!**



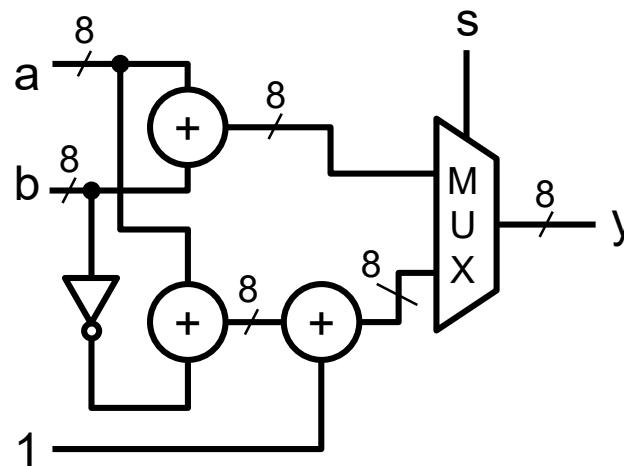
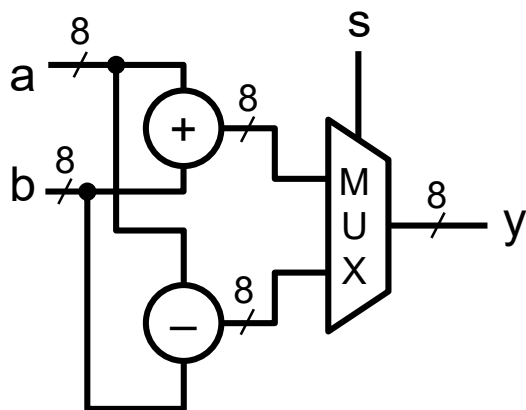
# 함수 공유(function sharing)



s	f
0	$a + b$
1	$a - b$



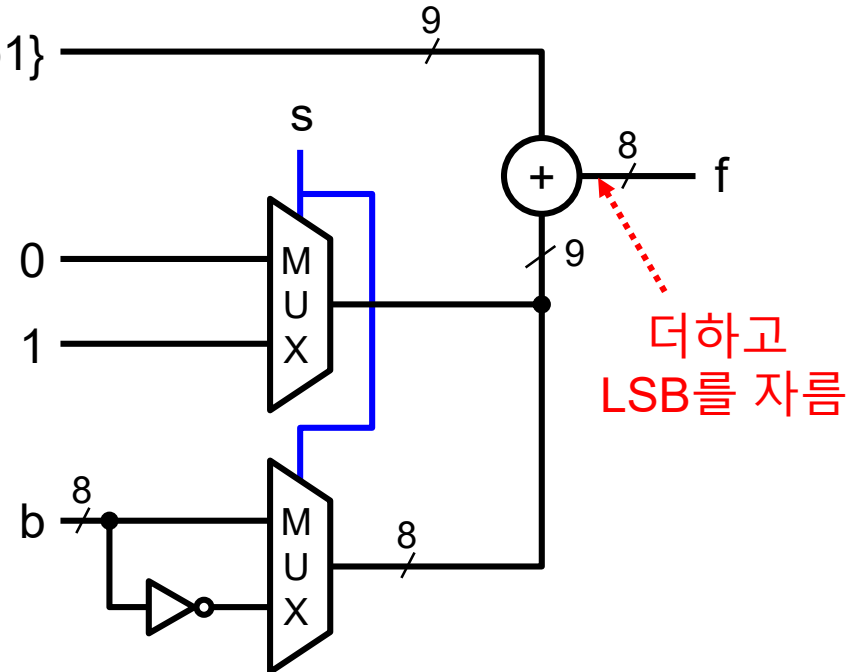
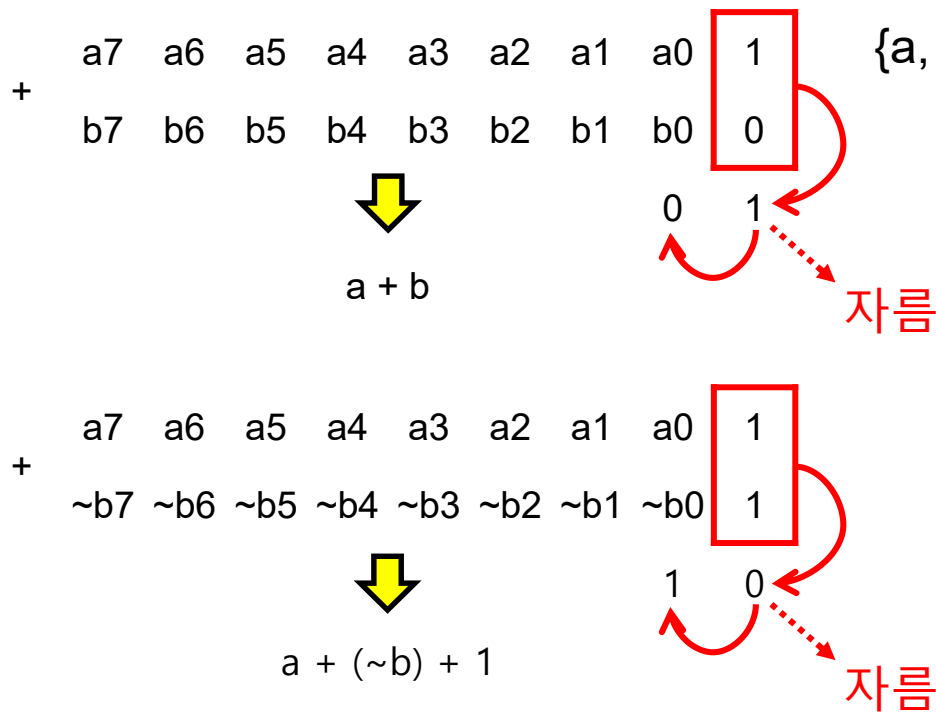
s	f
0	$a + b$
1	$a - b = a + (\sim b + 1)$



더하기 개수를 줄인 방법?

# 함수 공유(function sharing)

## ■ 더하기의 carry 비트 활용



# Full comparator

## ■ 효율적인 full comparator

비트 크기	관계 연산자			
	$>_a$	$>_d$	$=$	
8	25	68	26	Area (gate count)
16	52	102	51	
32	105	211	102	
64	212	398	204	
8	4.0	1.9	1.0	Delay (ns)
16	8.6	3.7	1.7	
32	17.6	6.7	1.8	
64	35.7	14.3	2.2	

### 단순한 설계

```

assign gt = (a > b) ? 1 : 0;
assign lt = (a < b) ? 1 : 0;
assign eq = (a == b) ? 1 : 0;

```

### 효율적인 설계

```

assign gt = (a > b) ? 1 : 0;
assign eq = (a == b) ? 1 : 0;
assign lt = ~(gt | eq);

```

# Barrel shifter

- 기본 시프트 연산자 (>>, <<, >>>, <<<)
  - Rotation 불가능
- Barrel 시프트
  - Rotation 가능
  - 입출력 포트
    - 입력: in 및 shamt(시프트 량)
    - 출력: out
  - 동작 원리
    - shamt의 비트 크기에 따라 시프트 단계로 나눔
    - 예: 3-bit shamt =  $\text{shamt}[2] \cdot 2^2 + \text{shamt}[1] \cdot 2^1 + \text{shamt}[0] \cdot 2^0$ 
      - 첫 단계: 시프트 하지 않거나 1비트( $2^0$ ) 시프트
      - 둘 단계: 시프트 하지 않거나 2비트( $2^1$ ) 시프트
      - 셋 단계: 시프트 하지 않거나 4비트( $2^2$ ) 시프트

# Barrel shifter

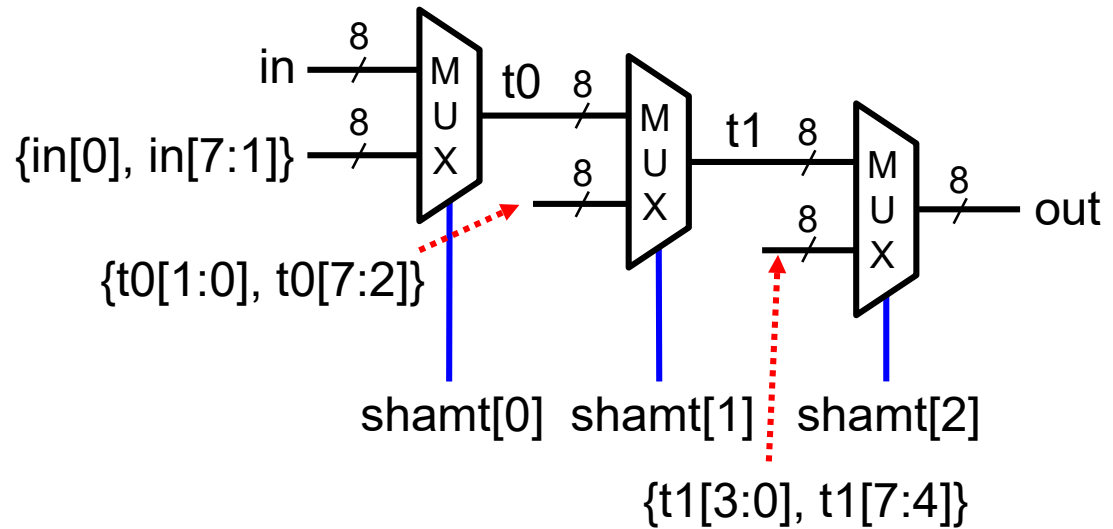
## ■ 코드

```
module barrel_shifter(
    input  [7:0] in,
    input  [2:0] shamt,
    output [7:0] out
);
```

```
    wire [7:0] t0, t1;
```

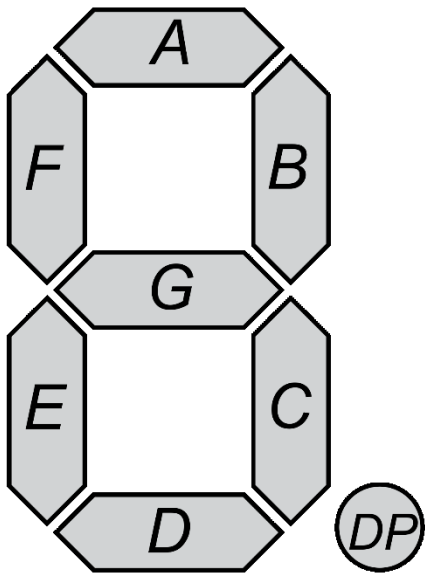
```
    assign t0  = shamt[0] ? {in[0],   in[7:1]} : in; // 1st stage
    assign t1  = shamt[1] ? {t0[1:0], t0[7:2]} : t0; // 2nd stage
    assign out = shamt[2] ? {t1[3:0], t1[7:4]} : t1; // 3rd stage
```

```
endmodule
```



# 7-segment LED decoder

## 7-segment LED 동작 원리



숫자	Active high							Active low						
	A	B	C	D	E	F	G	A	B	C	D	E	F	G
0	1	1	1	1	1	1	0	0	0	0	0	0	0	1
1	0	1	1	0	0	0	0	1	0	0	1	1	1	1
2	1	1	0	1	1	0	1	0	0	1	0	0	1	0
3	1	1	1	1	0	0	1	0	0	0	0	1	1	0
4	0	1	1	0	0	1	1	1	0	0	1	1	0	0
5	1	0	1	1	0	1	1	0	1	0	0	1	0	0
6	1	0	1	1	1	1	1	0	1	0	0	0	0	0
7	1	1	1	0	0	0	0	0	0	0	1	1	1	1
8	1	1	1	1	1	1	1	0	0	0	0	0	0	0
9	1	1	1	1	0	1	1	0	0	0	0	1	0	0

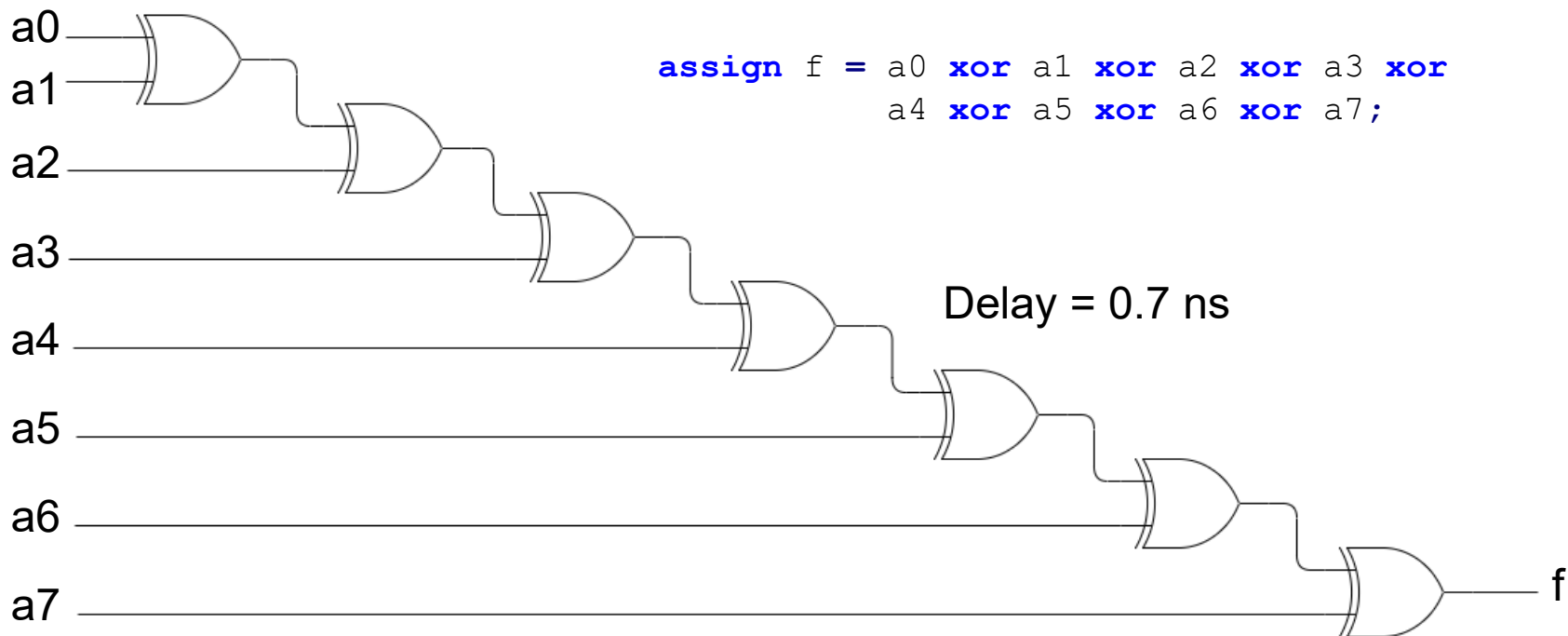
# 7-segment LED decoder

```
module sseg_decoder(  
    input      [3:0] num,  
    input      dp,  
    output reg [7:0] sseg  
);  
  
always @* begin  
    case (num)  
        4'd0: sseg[6:0] = 7'b111_1110;  
        4'd1: sseg[6:0] = 7'b011_0000;  
        4'd2: sseg[6:0] = 7'b110_1101;  
        4'd3: sseg[6:0] = 7'b111_1001;  
        4'd4: sseg[6:0] = 7'b011_0011;  
        4'd5: sseg[6:0] = 7'b101_1011;  
        4'd6: sseg[6:0] = 7'b101_1111;  
        4'd7: sseg[6:0] = 7'b111_0000;  
        4'd8: sseg[6:0] = 7'b111_1111;  
        default: sseg[6:0] = 7'b111_1011; // 4'd9  
    endcase  
    sseg[7] = dp;  
end  
  
endmodule
```

# 레이아웃 위한 설계



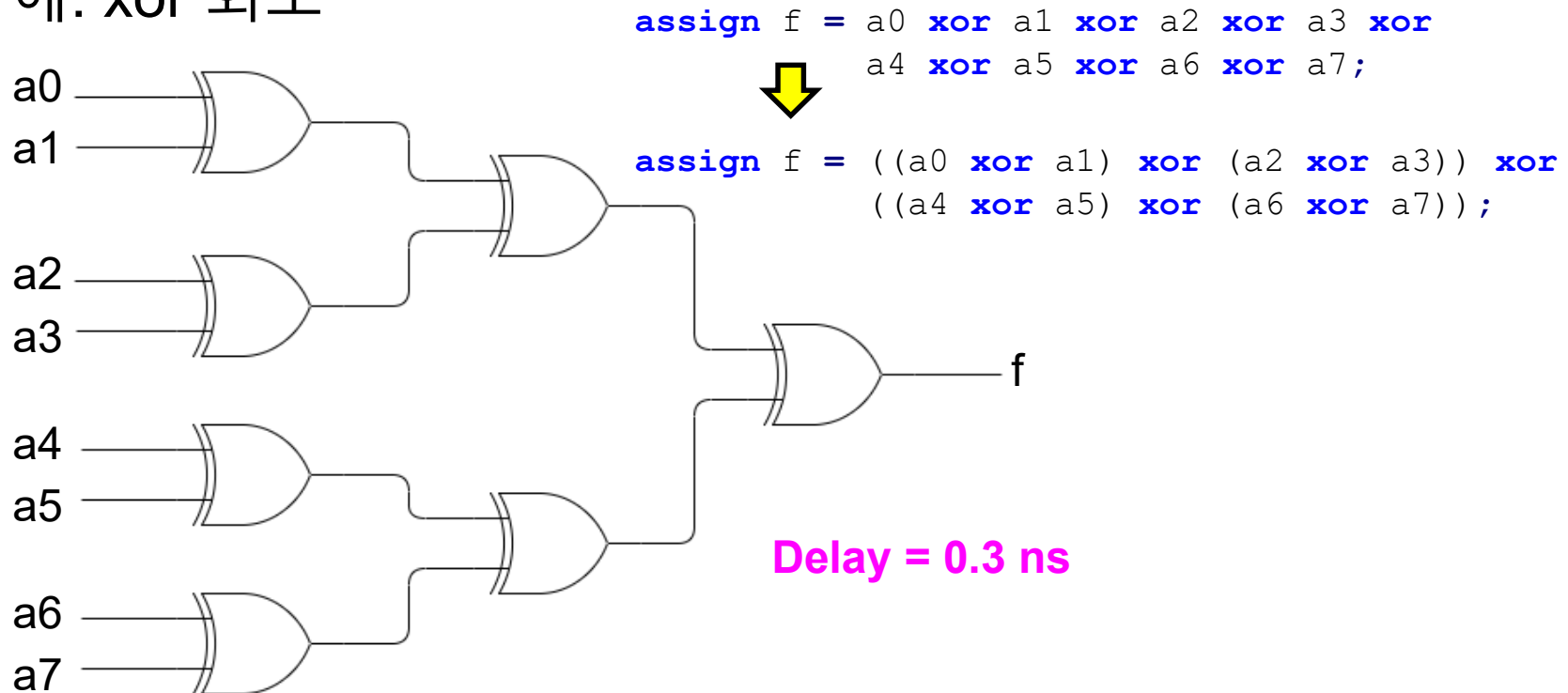
- 괄호 사용 → 회로 레이아웃 조절 가능
- 예: xor 회로





# 레이아웃 위한 설계

- 괄호 사용 → 회로 레이아웃 조절 가능
- 예: xor 회로



# 레이아웃 위한 설계

- 예: 4-to-2 priority encoder

in	code
1xxx	11
01xx	10
001x	01
0001	00

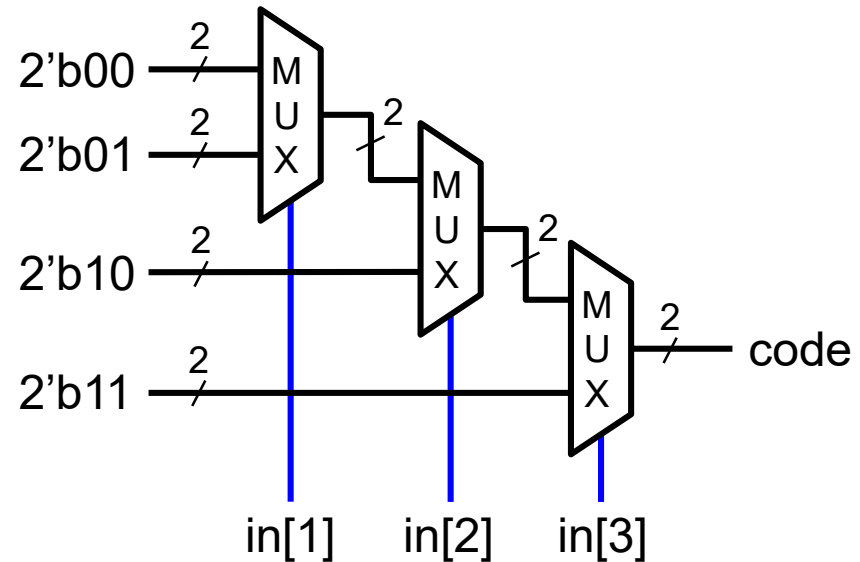
```

module pri_encoder(
    input  [3:0] in,
    output [1:0] code
);

    assign code = (in[3]) ? 2'b11 :
                  (in[2]) ? 2'b10 :
                  (in[1]) ? 2'b01 :
                  2'b00;

endmodule

```

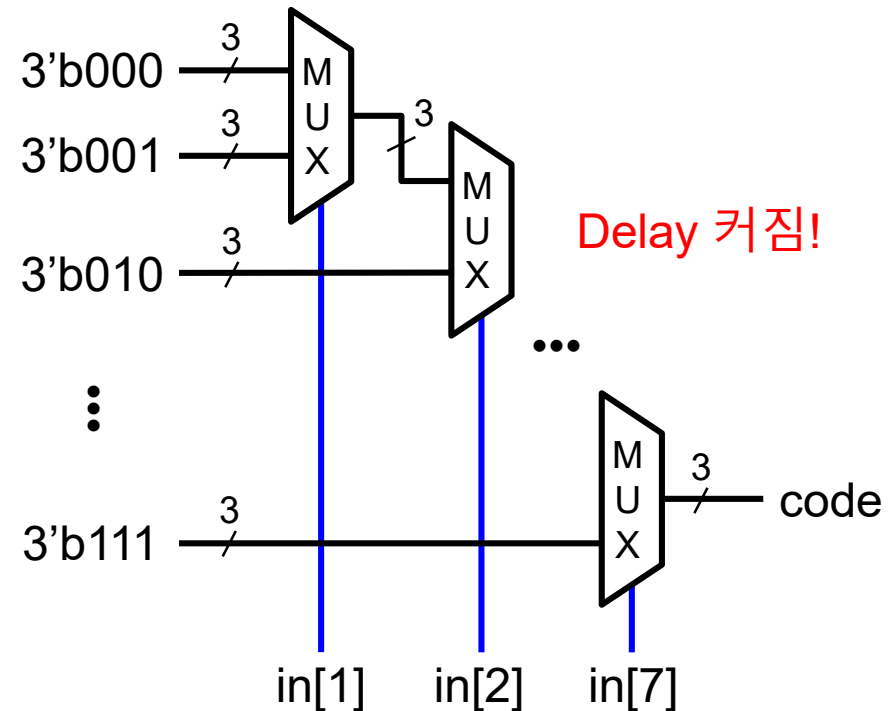


# 레이아웃 위한 설계



- 예: 8-to-3 priority encoder

```
module pri_encoder83(  
    input  [7:0] in,  
    output [2:0] code  
);  
  
assign code = (in[7]) ? 3'b111 :  
              (in[6]) ? 3'b110 :  
              (in[5]) ? 3'b101 :  
              (in[4]) ? 3'b100 :  
              (in[3]) ? 3'b011 :  
              (in[2]) ? 3'b010 :  
              (in[1]) ? 3'b001 :  
              3'b000;  
  
endmodule
```



# 레이아웃 위한 설계



- 예: 4-to-2 priority encoder로 8-to-3 priority encoder

in	code
1xxx xxxx	111
01xx xxxx	110
001x xxxx	101
0001 xxxx	100
0000 1xxx	011
0000 01xx	010
0000 001x	001
0000 0001	000



in	code
1xxx xxxx	111
01xx xxxx	110
001x xxxx	101
0001 xxxx	100
0000 1xxx	011
0000 01xx	010
0000 001x	001
0000 0001	000

4-to-2 priority encoder

- 입력:  $\text{in}[7:4]$
- 출력:  $\text{code}[2] = |\text{in}[7:4]$

4-to-2 priority encoder

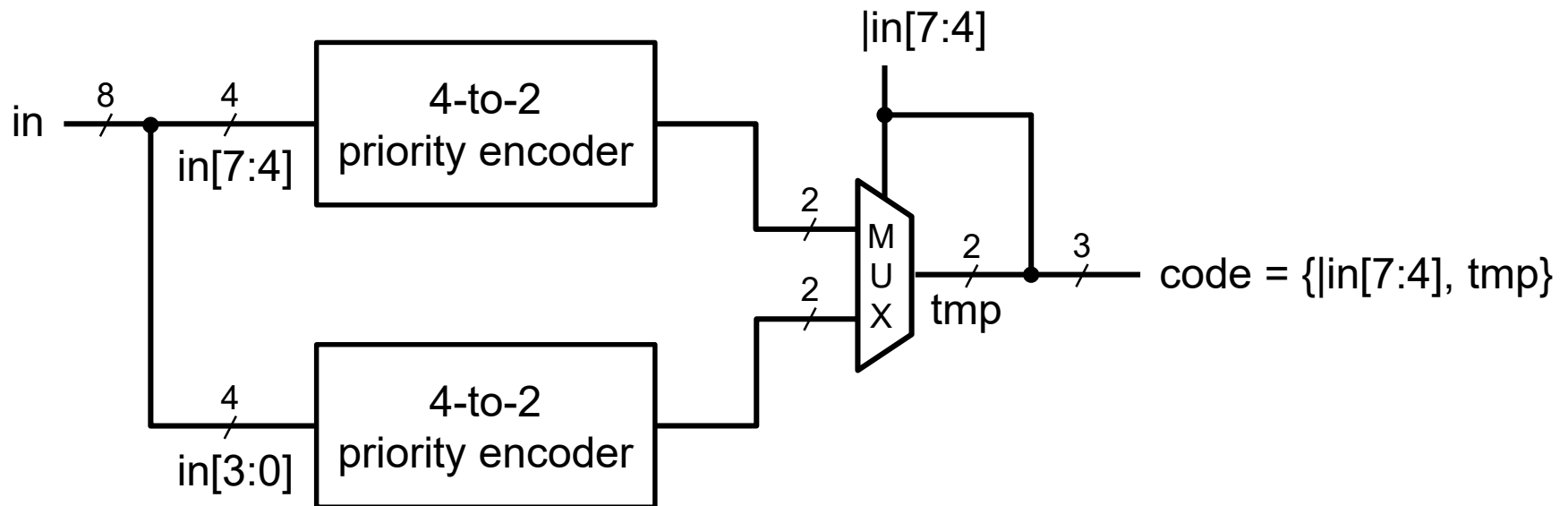
- 입력:  $\text{in}[3:0]$
- 출력:  $\text{code}[2] = |\text{in}[3:0]$

2-to-1 MUX

# 레이아웃 위한 설계



- 예: 4-to-2 priority encoder로 8-to-3 priority encoder



# 레이아웃 위한 설계



- 예: 4-to-2 priority encoder로 8-to-3 priority encoder

```
module pri_encoder83(  
    input  [7:0] in,  
    output [2:0] code  
);  
  
    wire [1:0] code1, code2, tmp;  
  
    pri_encoder42 u1(.in(in[7:4]), .code(code1));  
    pri_encoder42 u2(.in(in[3:0]), .code(code2));  
  
    assign tmp = (!in[7:4]) ? code1 : code2;  
    assign code = {!in[7:4], tmp};  
  
endmodule
```