



DeepL

Subscribe to DeepL Pro to translate larger documents.  
Visit [www.DeepL.com/pro](https://www.DeepL.com/pro) for more information.

# 디지털 디자인 및 컴퓨터 아키텍처

사라 해리스 & 데이비드 해리스

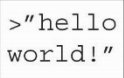


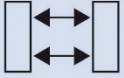
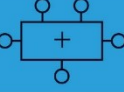

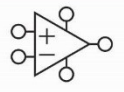


## 5장:

# 디지털 빌딩 블록

*수정 됨 유영환, 2023*

# 5장 :: 주제

- 소개
- 산술 회로
- 번호 시스템
- 순차적 빌딩 블록
- 메모리 어레이
- 논리 배열

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

# 소개

- **디지털 빌딩 블록:**

- 게이트, 멀티플렉서, 디코더, 레지스터, 산술 회로, 카운터, 메모리 어레이, 논리 어레이

- **빌딩 블록은 계층 구조, 모듈성 및 규칙성을 보여줍니다:**

- 더 간단한 구성 요소의 계층 구조
  - 잘 정의된 인터페이스 및 기능
  - 일반 구조로 다양한 크기로 쉽게 확장 가능

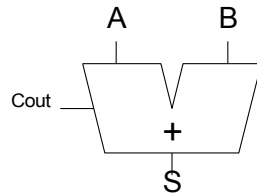
- 7장에서 이러한 빌딩 블록을 사용하여 마이크로프로세서를 빌드하겠습니다.

## 5장: 디지털 빌딩 블록

# 가산기

# 1비트 가산기

## 하프 애더

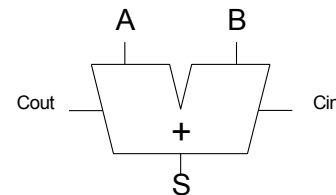


A	B	Cout	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$

$$Cout = A \cdot B$$

## 전체 애더



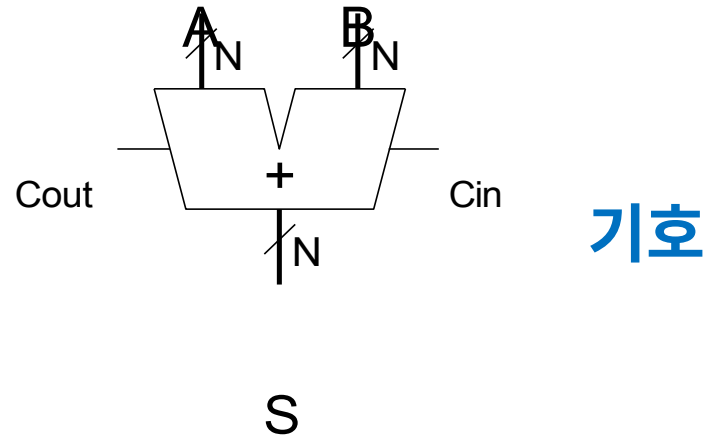
Cin	A	B	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

$$Cout = (A \cdot B) \vee (A \cdot BC_{in}) \vee (B \cdot BC_{in})$$

# 멀티비트 가산기: CPA

- 멀티비트 가산기



- 캐리 전파 가산기(CPA)의 유형:

- 리플 캐리 (느림)
- 캐리 룩어헤드 (빠른)
- 접두사 (더 빠른)

- 캐리 룩어헤드 및 접두사 가산기는 대형 가산



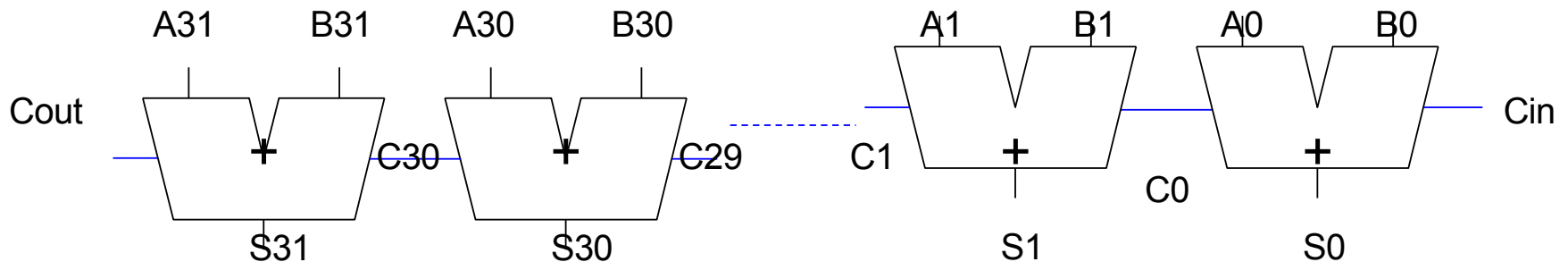
기의 경우 더 빠르지만 더 많은 하드웨어가  
필요합니다.

5장: 디지털 빌딩 블록

# 리플 캐리 추 가

# 리플 캐리 애더

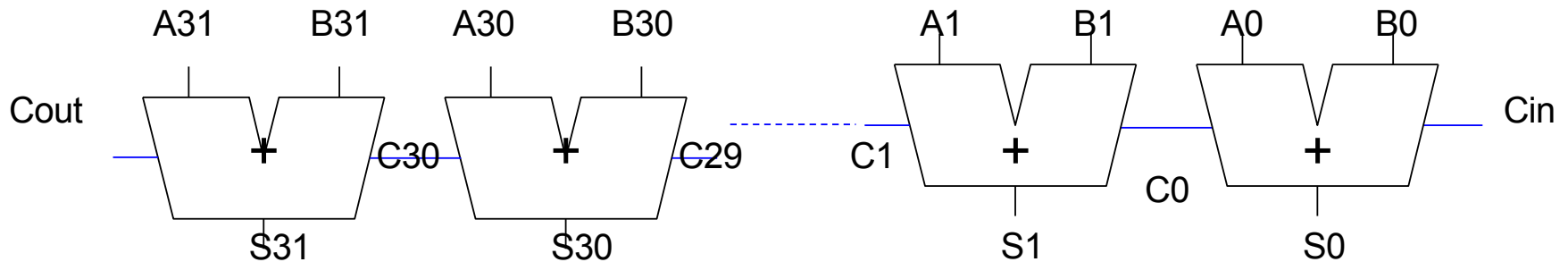
- 1비트 가산기를 함께 연결
- 전체 체인에 파급 효과 전달
- 단점: **느림**



# 리플 캐리 애더 지연

$$3배 = NtFA$$

여기서  $t_{FA}$ 는 1비트 풀 가산기의 지연 시간입니다.



## 5장: 디지털 빌딩 블록

# 캐리 룩어헤드 추 가

# 캐리-룩어헤드 애더

신호 생성 및 전파를 사용하여 k비트 블록에 대한  $C_{out}$  계산 몇 가지 정의

가 있습니다:

- 열  $i$ 는 캐리 아웃을 **생성하여** 캐리 아웃을 생성하거나 반입을 반출로 **전파하기**
- 각 열에 대한 생성( $G_i$ ) 및 전파( $P_i$ ) 신호를 계산합니다:
  - **생성합니다:** 열  $i$ 는  $A_i$ 와  $B_i$ 가 모두 1이면 수행을 생성합니다.

$G_i = A_i \cdot B_i$

- **전파:** 열  $i$ 가  $A_i$  또는  $B_i$ 가 1이면 캐리 인을 캐리 아웃으로 전파합니다.

$P_i = A_i + B_i$

- **수행합니다:** 수행: 열  $i$ 의 수행은 다음과 같습니다:

$$C_i = A_i \cdot B_i + (A_i + B_i)C_{i-1} = G_i + P_i \cdot C_{i-1}$$

# 신호 전파 및 생성

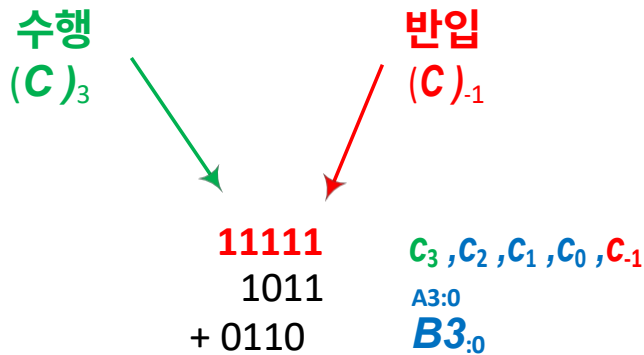
예시: 열은 신호를 전파하고 생성합니다:

열 전파:

$$P_i = A_i + B_i$$

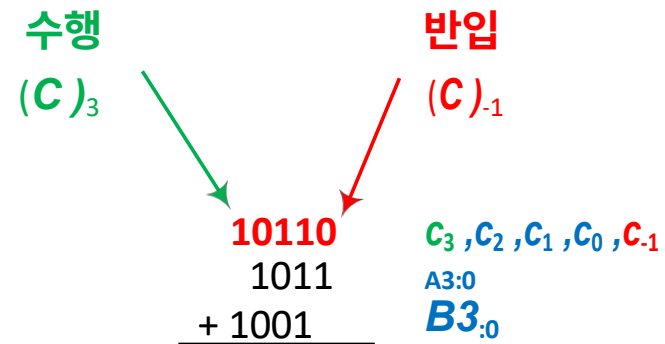
열 생성:

$$G_i = A_i B_i$$



1111  
0010

$p_3, p_2, p_1$   
 $, p_0, g_3, g_2, g_1$   
 $, g_0$



1011  
1001

$p_3, p_2, p_1$   
 $, p_0, g_3, g_2, g_1$   
 $, g_0$

$$C_i = G_i + P_i C_{i-1}$$

# 전파 및 생성 차단

이제 전파 및 생성 신호 열을 사용하여  $k$ 비트 블록에 대한 **블록 전파** 및 **블록 생성** 신호를 계산합니다:

- **k비트 그룹이** (블록의) 캐리 인을 (블록의) 캐리 아웃으로 **전파할지** 계산합니다.
- **k비트 그룹이** (블록의) 캐리아웃을 **생성할지** 계산합니다.



# 전파 및 생성 차단

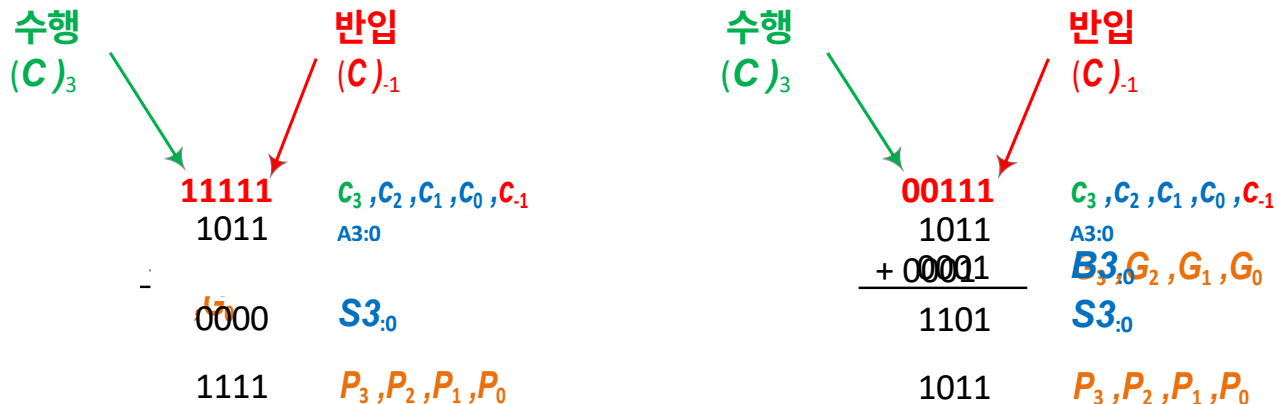
## • 예시: 4비트 블록

### • 블록 전파 신호: $P_{3:0}$ (단일 비트 신호)

- 반입은 블록의 4비트 모두를 통해 전파됩니다:

$$P_{3:0} = P_3 P_2 P_1 P_0$$

### • 예시:



# 전파 및 생성 차단

- **예시: 4비트 블록**

- **블록 전파 신호:  $P3_{:0}$** (단일 비트 신호)

- 반입은 블록의 4비트 모두를 통해 전파됩니다:

$$P3_{:0} = P3P2P1P0$$

- **블록 생성 신호:  $G3_{:0}$** (단일 비트 신호)

- 캐리가 생성됩니다:
  - 를 열 3에 **입력하거나**
  - 를 열 2에 추가하고 열 3을 통해 **전파하거나**
  - 를 열 1에 배치하고 열 2와 3을 통해 **전파하거나**
  - 열 0에서 시작하여 열 1-3을 통해 전파됩니다.

$$G3_{:0} = G3 + G2P3 + G1P2P3 +$$

$$G0P1P2P3 \quad G3_{:0} = G3 + P3 [G2 + P2 (G1 +$$



# 전파 및 생성 차단

## • 예시: 4비트 블록

### • 블록 생성 신호: $G_{3:0}$ (단일 비트 신호)

- 캐리는 열 3에서 생성되거나 열 2에서 생성되어 열 3을 통해 전파되거나 ...

$$G_{3:0} = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3$$

수행  
(C)<sub>3</sub>

$$\begin{array}{r} \downarrow \\ \textcolor{red}{10011} \\ 1001 \\ + 1100 \\ \hline 0110 \end{array}$$

$C_3, C_2, C_1, C_0, C_{-1}$

A<sub>3:0</sub>  
 $\textcolor{blue}{B}_{3:0}$   
 $\textcolor{blue}{S}_{3:0}$

1101  $P_3, P_2, P_1, P_0$   
1000  $G_3, G_2, G_1, G_0$

$$\boxed{G_{3:0} = 1}$$

수행  
(C)<sub>3</sub>

$$\begin{array}{r} \downarrow \\ \textcolor{red}{11000} \\ 1110 \\ + 0100 \\ \hline 0010 \end{array}$$

$C_3, C_2, C_1, C_0, C_{-1}$

A<sub>3:0</sub>  
 $\textcolor{blue}{B}_{3:0}$   
 $\textcolor{blue}{S}_{3:0}$

1110  $P_3, P_2, P_1, P_0$   
0100  $G_3, G_2, G_1, G_0$

$$\boxed{G_{3:0} = 1}$$

수행  
(C)<sub>3</sub>

$$\begin{array}{r} \downarrow \\ \textcolor{red}{01101} \\ 0110 \\ + 0010 \\ \hline 1000 \end{array}$$

$C_3, C_2, C_1, C_0, C_{-1}$

A<sub>3:0</sub>  
 $\textcolor{blue}{B}_{3:0}$   
 $\textcolor{blue}{S}_{3:0}$

0110  $P_3, P_2, P_1, P_0$   
0010  $G_3, G_2, G_1, G_0$

$$\boxed{G_{3:0} = 0}$$

# 전파 및 생성 차단

- 예시: 4비트 블록

- 블록 전파 신호:  $P_{3:0}$  (단일 비트 신호)

- 반입은 블록의 4비트 모두를 통해 전파됩니다:

$$P_{3:0} = P_3 P_2 P_1 P_0$$

- 블록 생성 신호:  $G_{3:0}$  (단일 비트 신호)

- 캐리가 생성됩니다:
  - 를 열 3에 입력하거나
  - 를 열 2에 추가하고 열 3을 통해 전파하거나
  - 를 열 1에 추가하고 열 2와 3을 통해 전파하거나
  - 열 0에서 시작하여 열 1-3을 통해 전파됩니다.

$$G_{3:0} = G_3 + G_2 P_3 + G_1 P_2 P_3 +$$

$$G_0 P_1 P_2 P_3 \quad G_{3:0} = G_3 + P_3 [G_2 + P_2 (G_1 + P_1 G_0)]$$

$$C_3 = G_{3:0} + P_{3:0} C_{-1}$$

# 전파 및 생성 차단

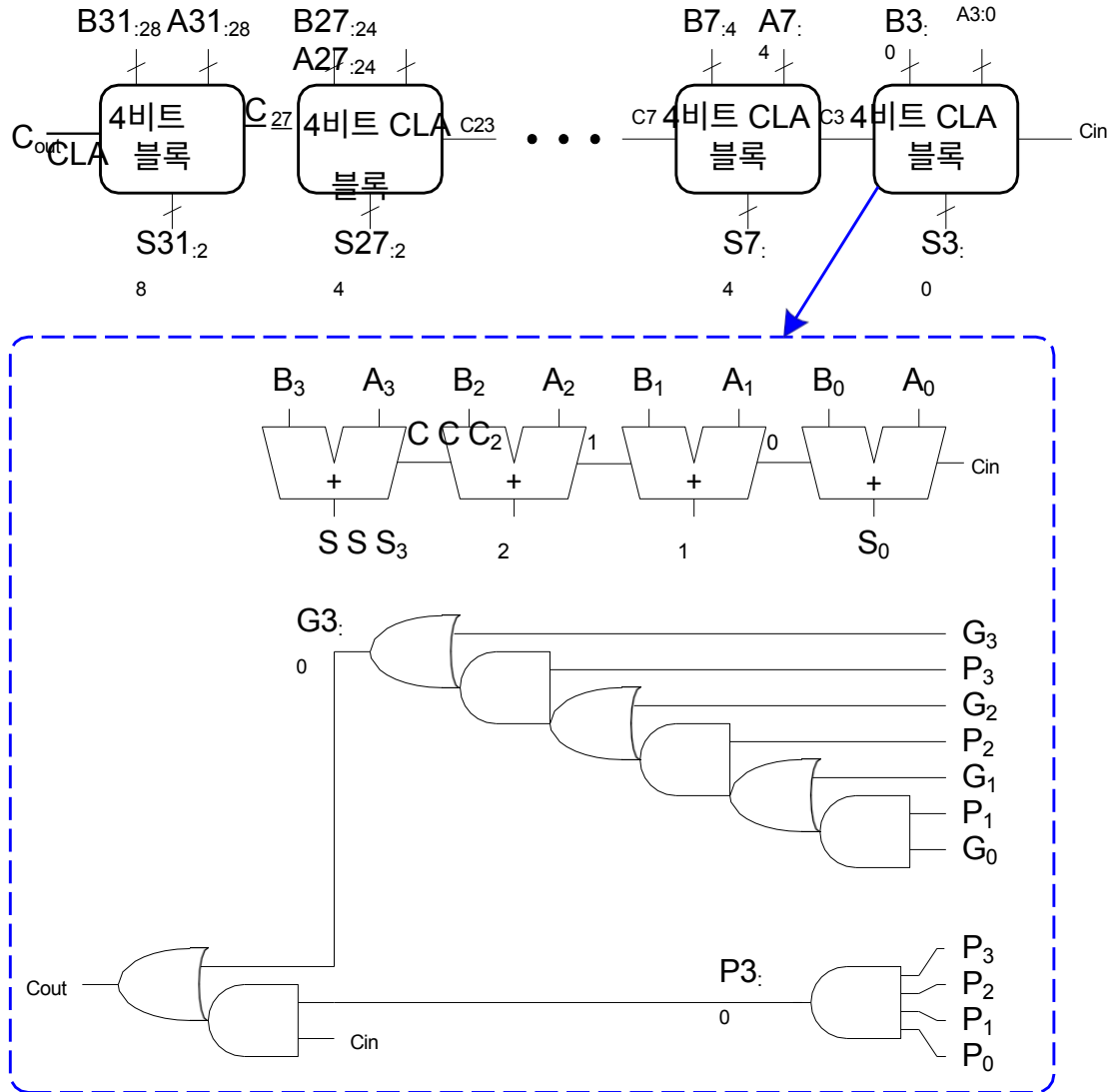
- 예시: 4비트 블록( $P3_{:0}$  및  $G3_{:0}$ )에 대한 블록 전파 및 신호 생성:

$$P3_{:0} = P3P2P1P0$$

$$G3_{:0} = G3 + P3 (G2 + P2 (G1 + P1G0))$$

$$C3 = G3_{:0} + P3_{:0} C-1$$

# 4비트 블록이 포함된 32비트 CLA





# 캐리-룩어헤드 추가

- **1단계:** 모든 열에 대해  $G_i$ 와  $P_i$  계산하기
- **2단계:**  $k$ 비트 블록에 대한  $G$ 와  $P$  계산하기
- **3단계:** 각  $kH/T$  전파/생성 로직을 통해  $C_{in} O_i$  전파됩니다(합계를 계산하는 동안).
- **4단계:** 가장 중요한  $k$ -비트 블록의 합계를 계산합니다.

# 캐리-룩어헤드 추가

- 1단계: 모든 열에 대해  $G_i$ 와  $P_i$  계산하기

$$G_i = A_i \vee B_i$$

$$P_i = A_i \oplus B_i$$

# 캐리-룩어헤드 추가

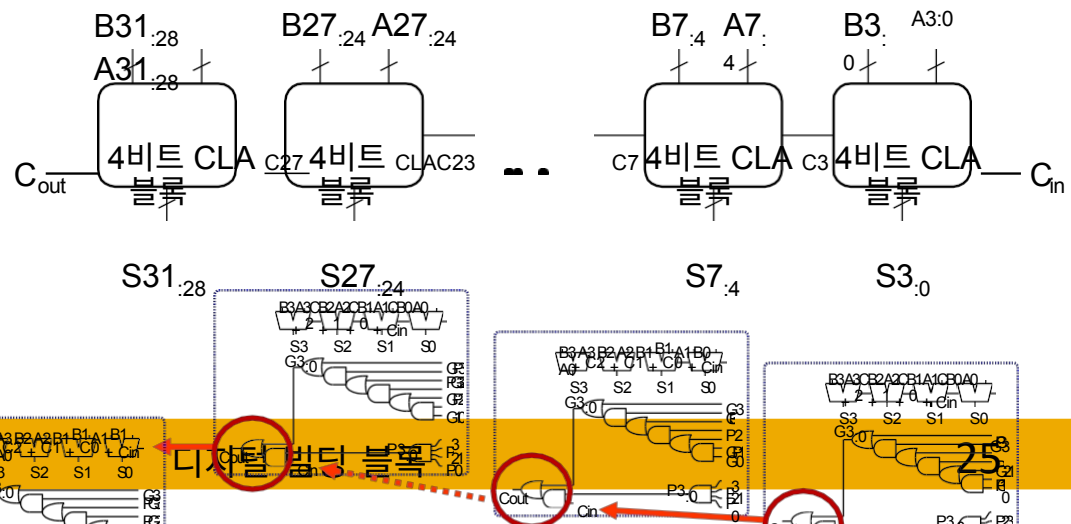
- 1단계: 모든 열에 대해  $G_i$ 와  $P_i$  계산하기
- 2단계: k비트 블록에 대한  $G$ 와  $P$  계산하기

$$P_{3:0} = P_3 P_2 P_1 P_0$$

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$$

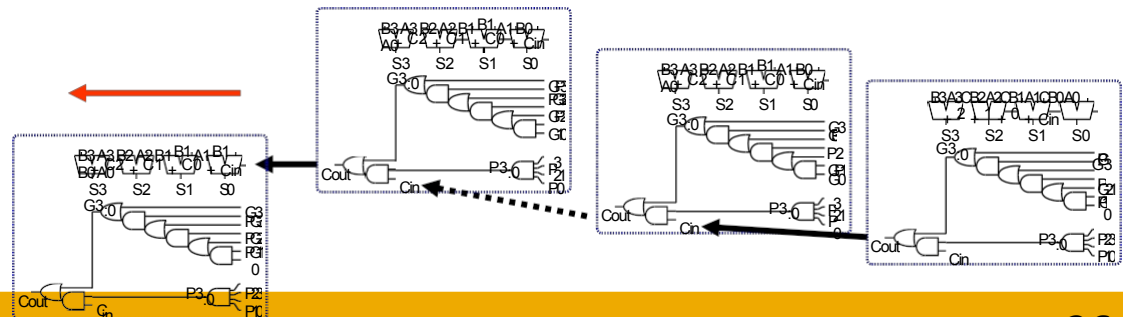
# 캐리-룩어헤드 추가

- 1단계: 모든 열에 대해  $G_i$ 와  $p_i$  계산하기
- 2단계:  $k$ 비트 블록에 대한  $G$ 와  $P$  계산하기
- 3단계:  $C_{in}$ 은 각  $k$ 비트 전파/생성 로직을 통해 전파됩니다(그 사이 계산 합계)

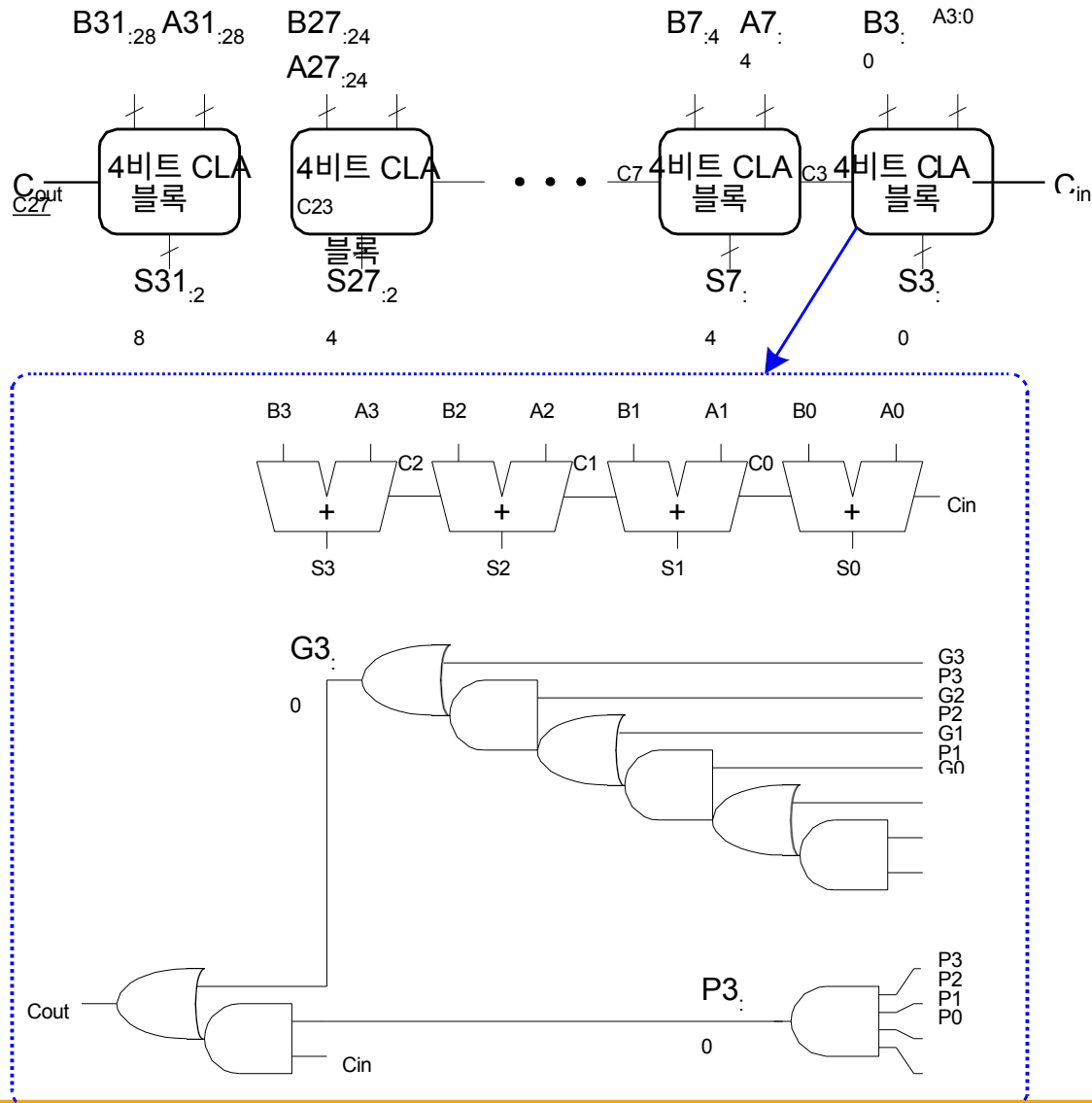


# 캐리-룩어헤드 추가

- 1단계: 모든 열에 대해  $G_i$ 와  $p_i$  계산하기
- 2단계:  $k$ 비트 블록에 대한  $G$ 와  $P$  계산하기
- 3단계: 각  $kH/\tau$  전파/생성 로직을 통해  $C_{in} 0/$  전파됩니다(합계를 계산하는 동안).
- 4단계: 가장 중요한  $k$ -비트 블록의 합계를 계산합니다.



# 4비트 블록이 포함된 32비트 CLA



# 캐리-룩어헤드 애더 지연

k비트 블록이 있는 N비트 CLA의 경우:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

- $t_{pg}$ : 모든  $P_i, G_i$  생성 지연
- $t_{pg\_block}$ : 모든  $P_{i:j}, G_{i:j}$  생성 지연 시간
- $t_{AND\_OR}$ : k비트 CLA 블록에서 최종 AND/OR 게이트의  $C_{in}$ 에서  $C_{out}$ 까지의 지연 시간

N비트 캐리 룩어헤드 가산기는 일반적으로  $N > 16$ 의 경우 리플 캐리 가산기보다 훨씬 빠릅니다.

5장: 디지털 빌딩 블록

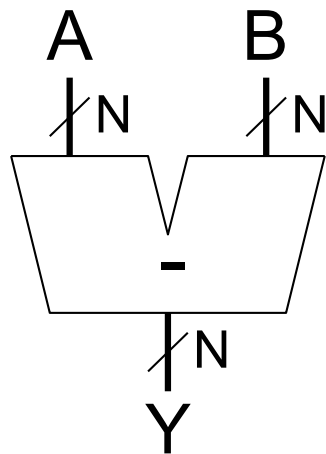
# 배기 및 비교 기



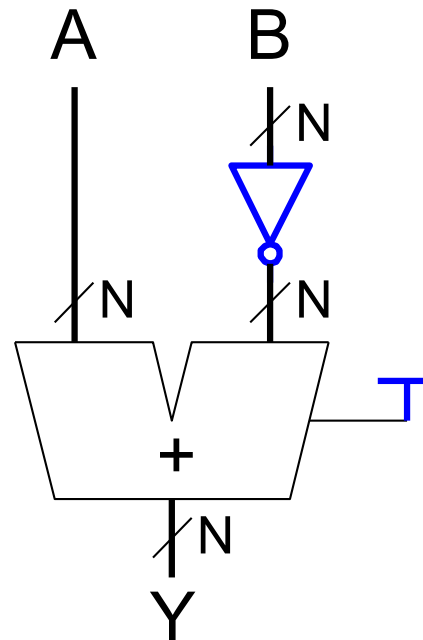
# 배기

$$a - b = a + \overline{b} + 1$$

기호

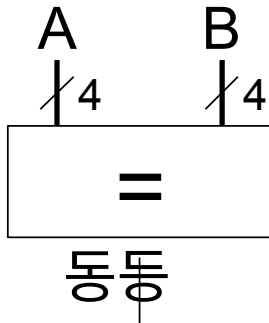


구현

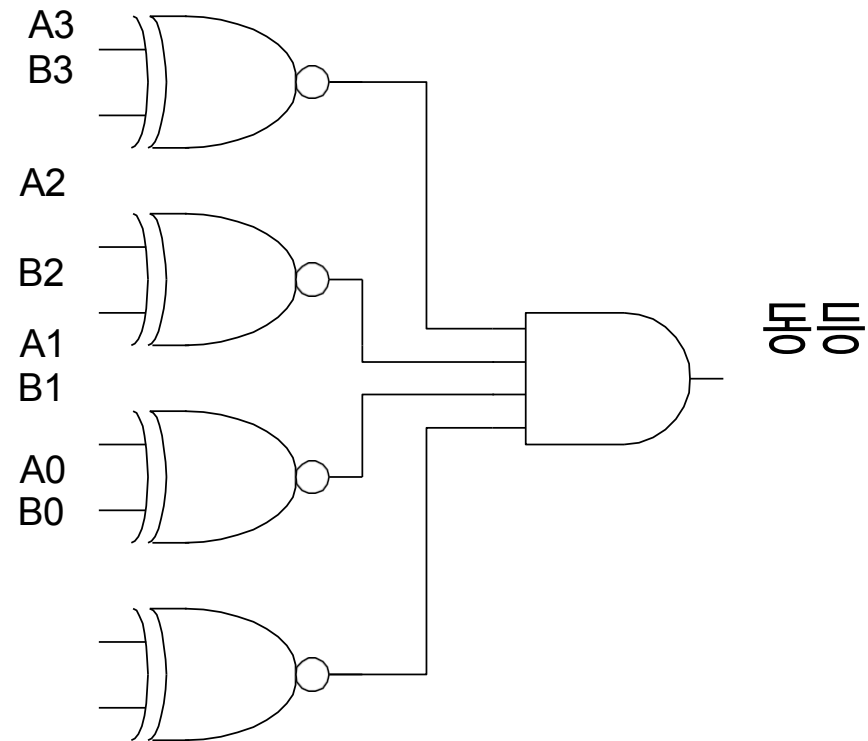


# 비교기: 평등

기호

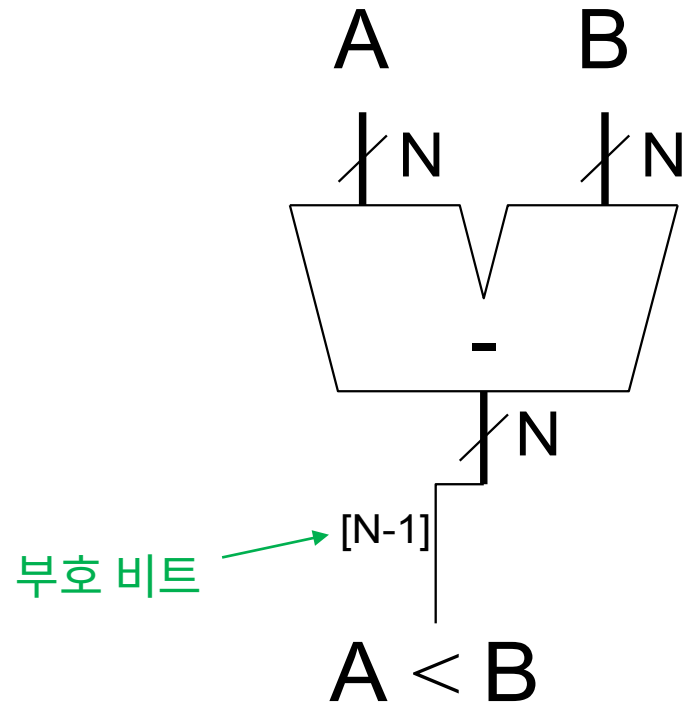


구현



# 비교기: 미만 서명

A-B가 음수인 경우  $A < B$   
오버플로 주의



# 5장: 디지털 빌딩 블록

**ALU:**

**산술 논리 단위**

# ALU: 산술 논리 단위

**ALU가 수행해야 합니다:**

- 추가
- 빼기
- AND
- 또는

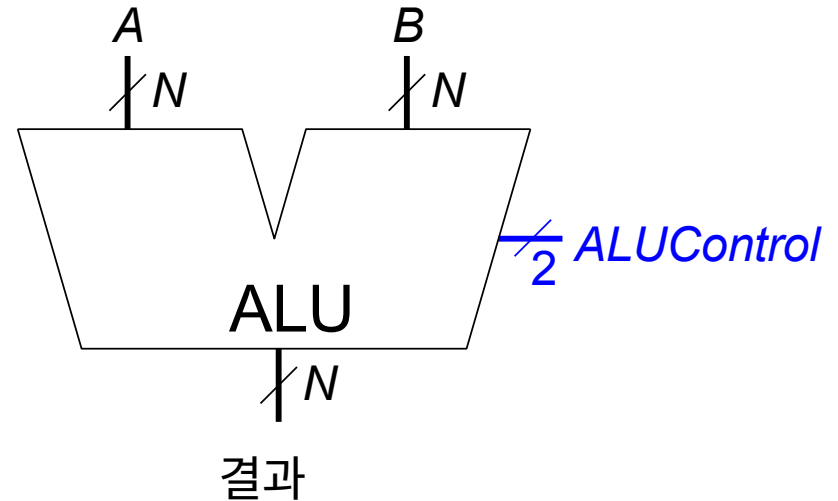
# ALU: 산술 논리 단위

ALUControl <sub>1:0</sub>	기능
00	추가
01	빼기
10	AND
11	또는

**예시:**  $A$  또는  $B$  수행

$ALUControl_{1:0} = 11$

**결과** =  $A$  또는  $B$



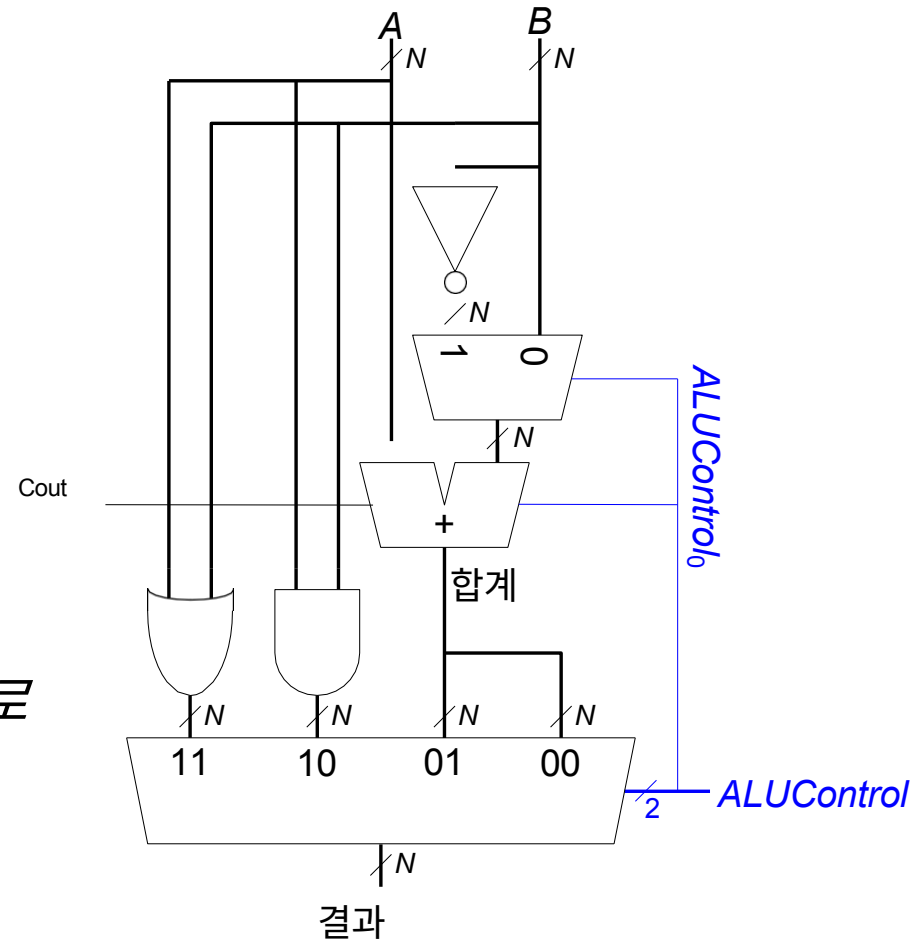
# ALU: 산술 논리 단위

ALUControl <sub>1:0</sub>	기능
00	추가
01	빼기
10	AND
11	A 또는 B 수행

$ALUControl_{1:0} = 11$

Mux는 OR 게이트의 출력을 *결과값으로* 선택합니다,  
그래서:

**결과 = A 또는 B**



# ALU: 산술 논리 단위

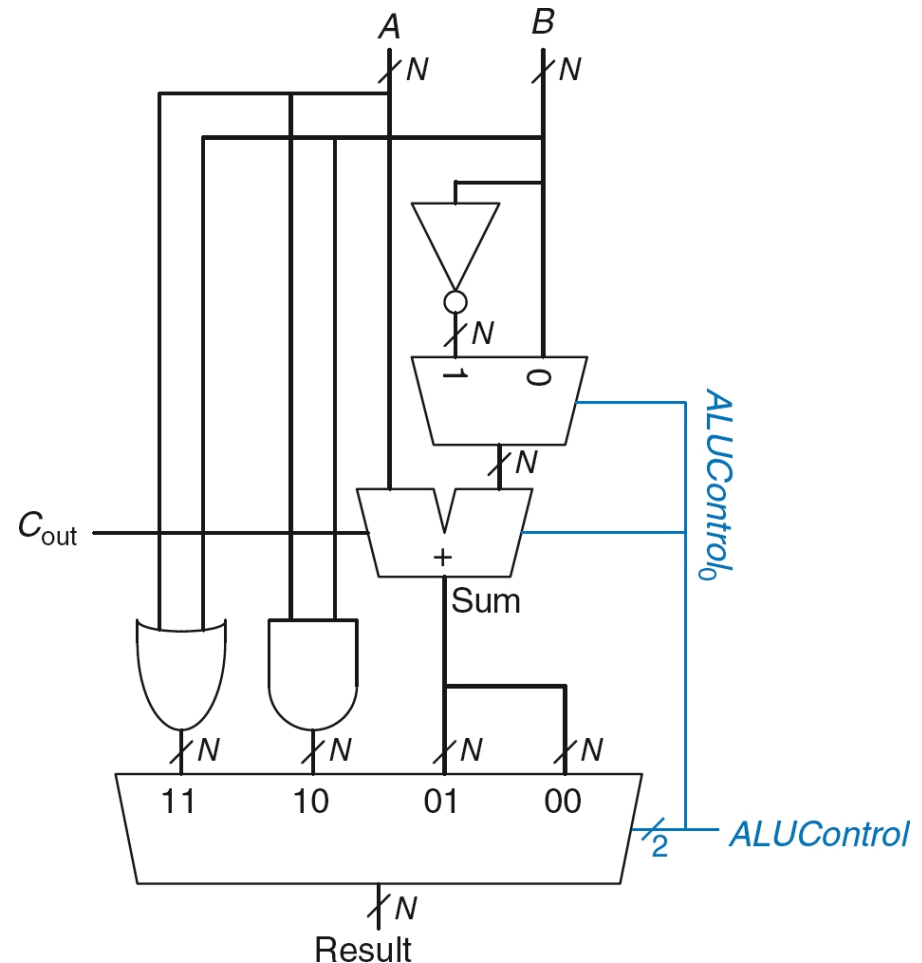
ALUControl <sub>1:0</sub>	기능
00	추가
01	빼기
10	AND
11	또는

**예제 예:  $A + B$  수행**

$ALUControl_{1:0} = 00$

$ALUControl_0 = 00$ 이므로:

$C_{in}$ 에서 가산기 = 0





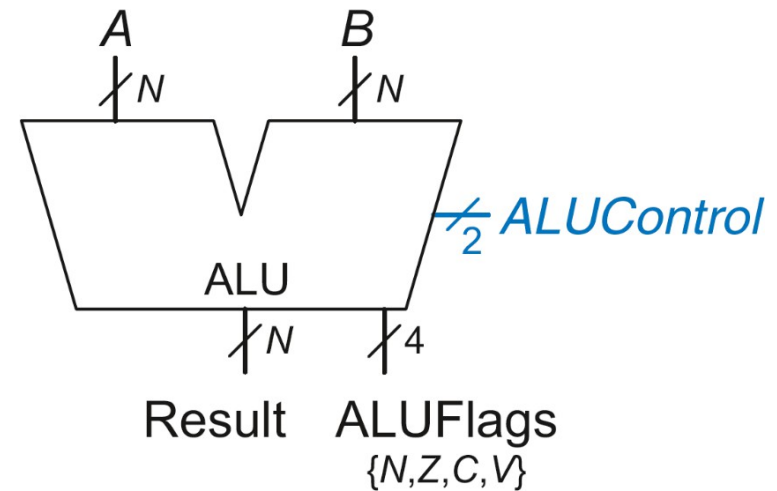
2<sup>nd</sup> 가산기 입력은  $B$

Mux는  $Sum$ 을 **결과값**으로 선택하므로

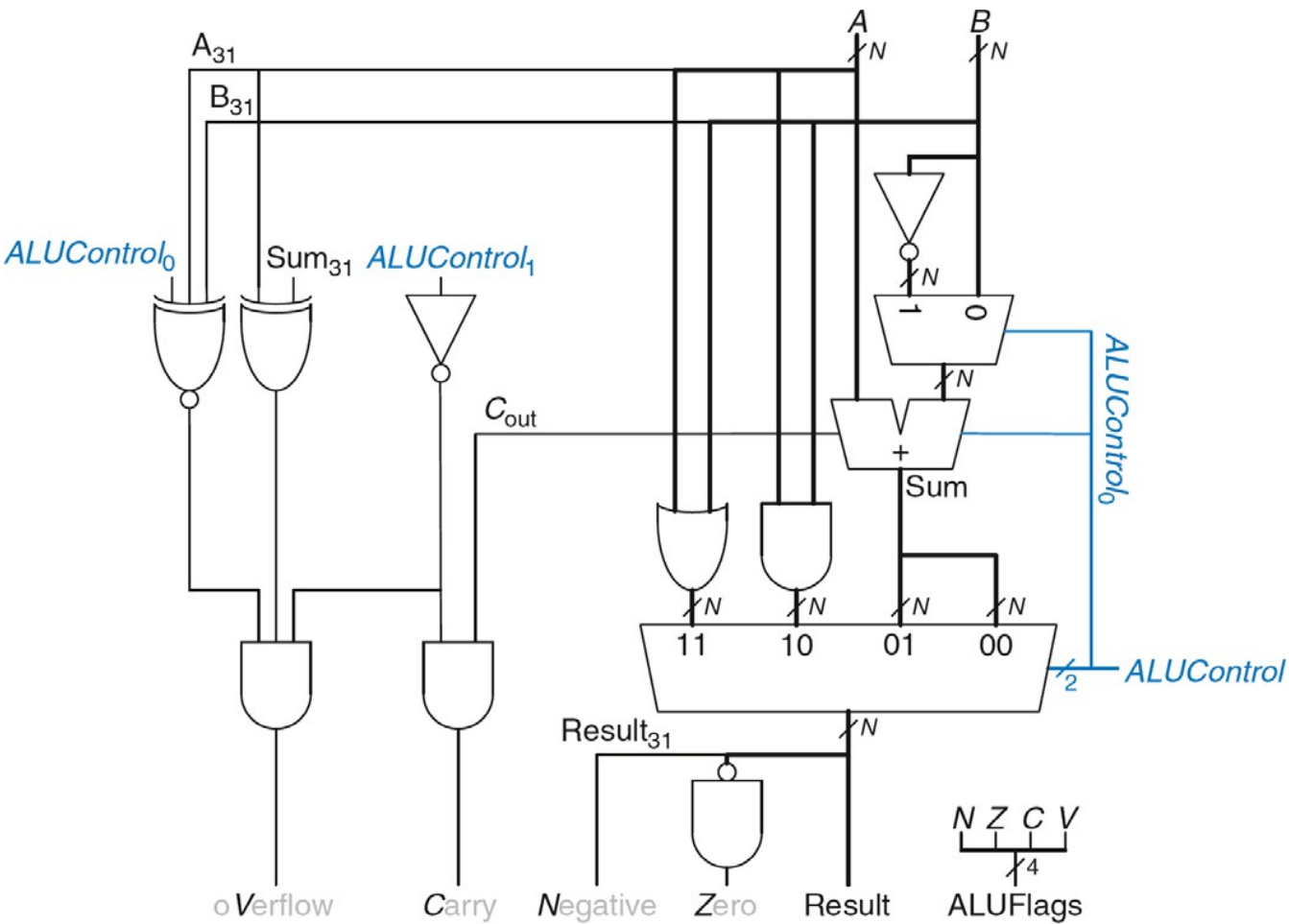
$$\text{결과} = A + B$$

# 상태 플래그가 있는 ALU

깃발	설명
$N$	결과는 음수입니다.
$Z$	결과는 0입니다.
$C$	애더 생산 수행
$V$	오버플로 추가



## 상태 플래그가 있는 ALU



# 상태 플래그가 있는 ALU: 부정

$N = 10$ 이면

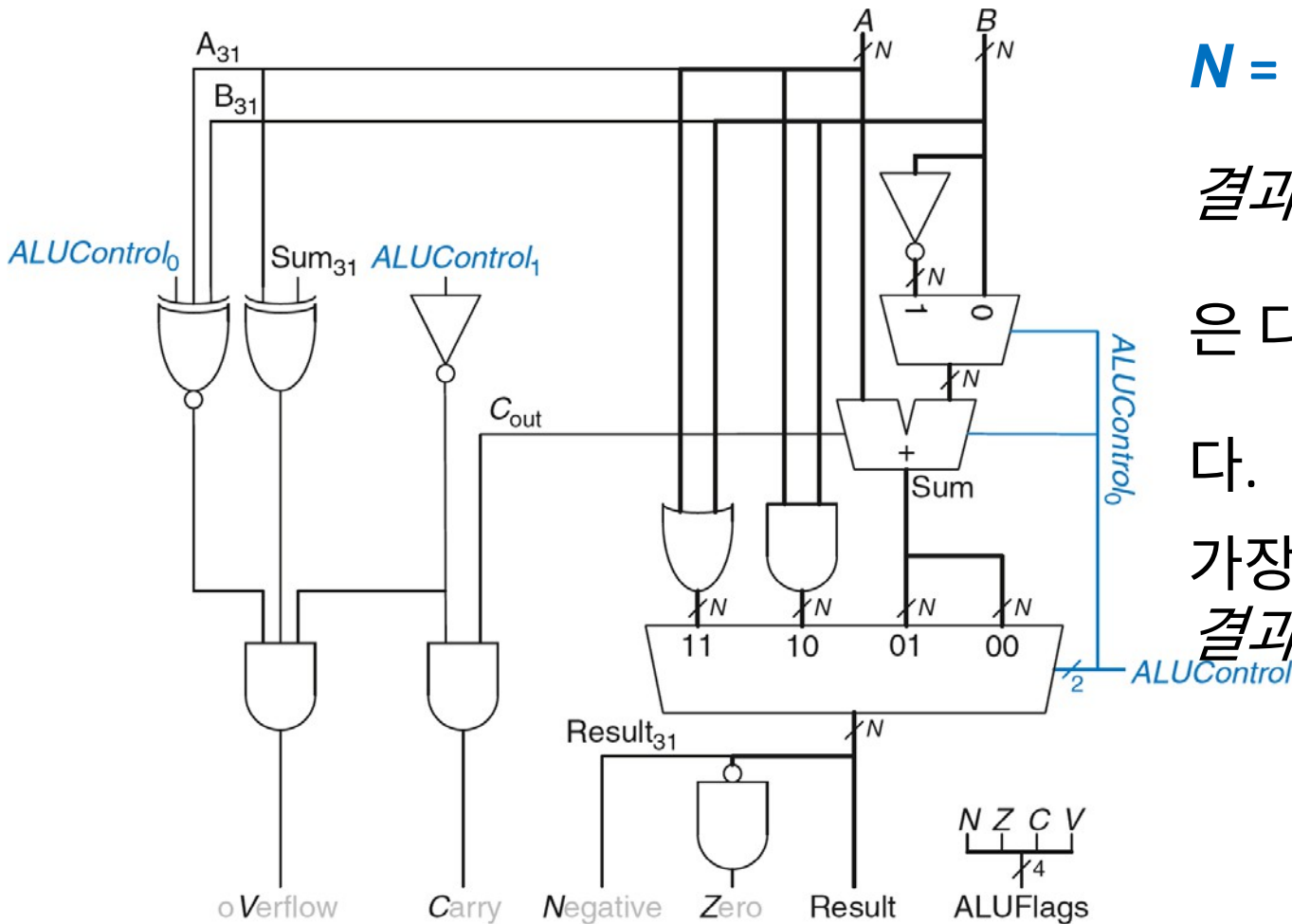
결과가 음수이므로  $N$

은 다음에 연결됩니

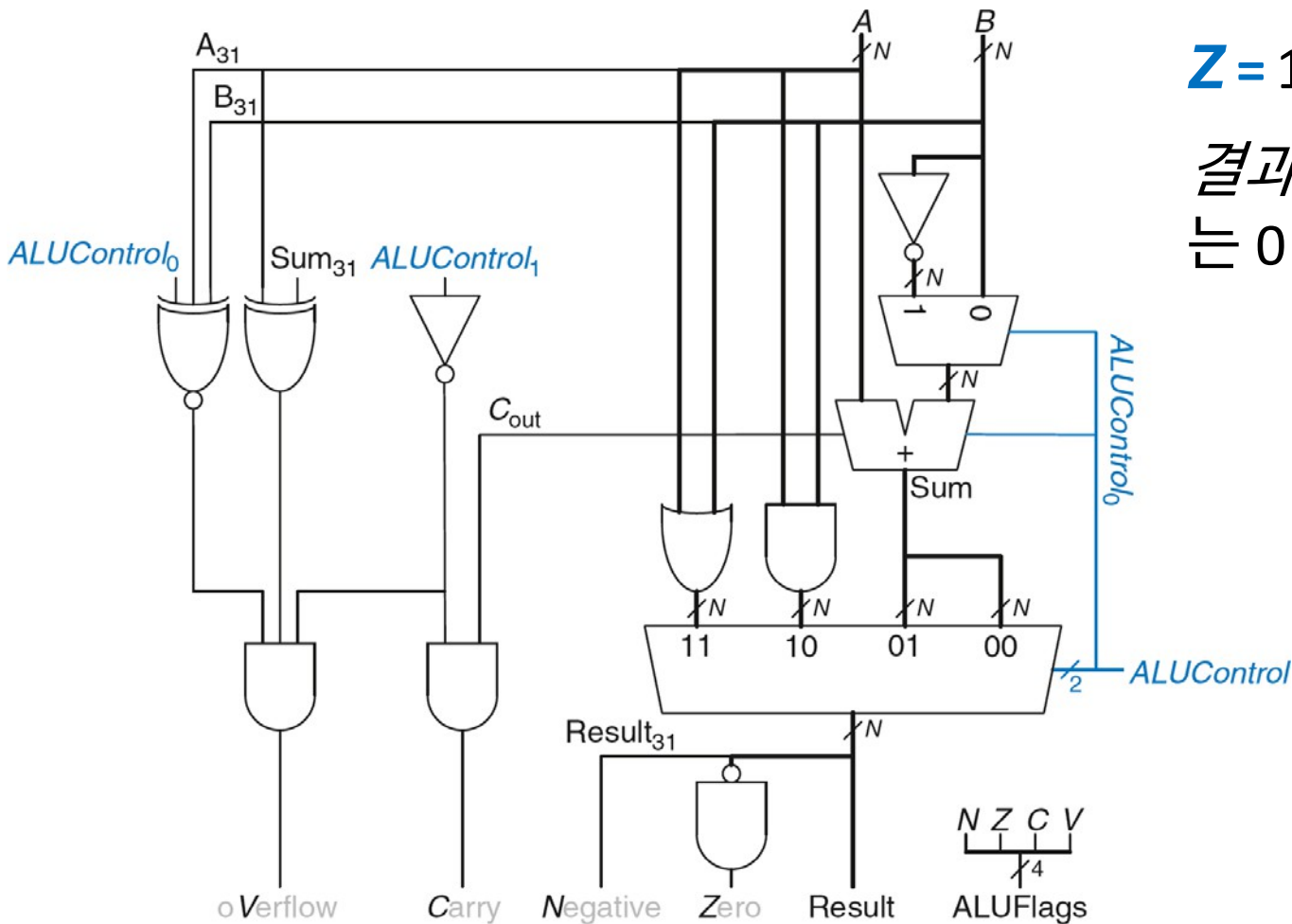
다.

가장 중요한 부분

결과.



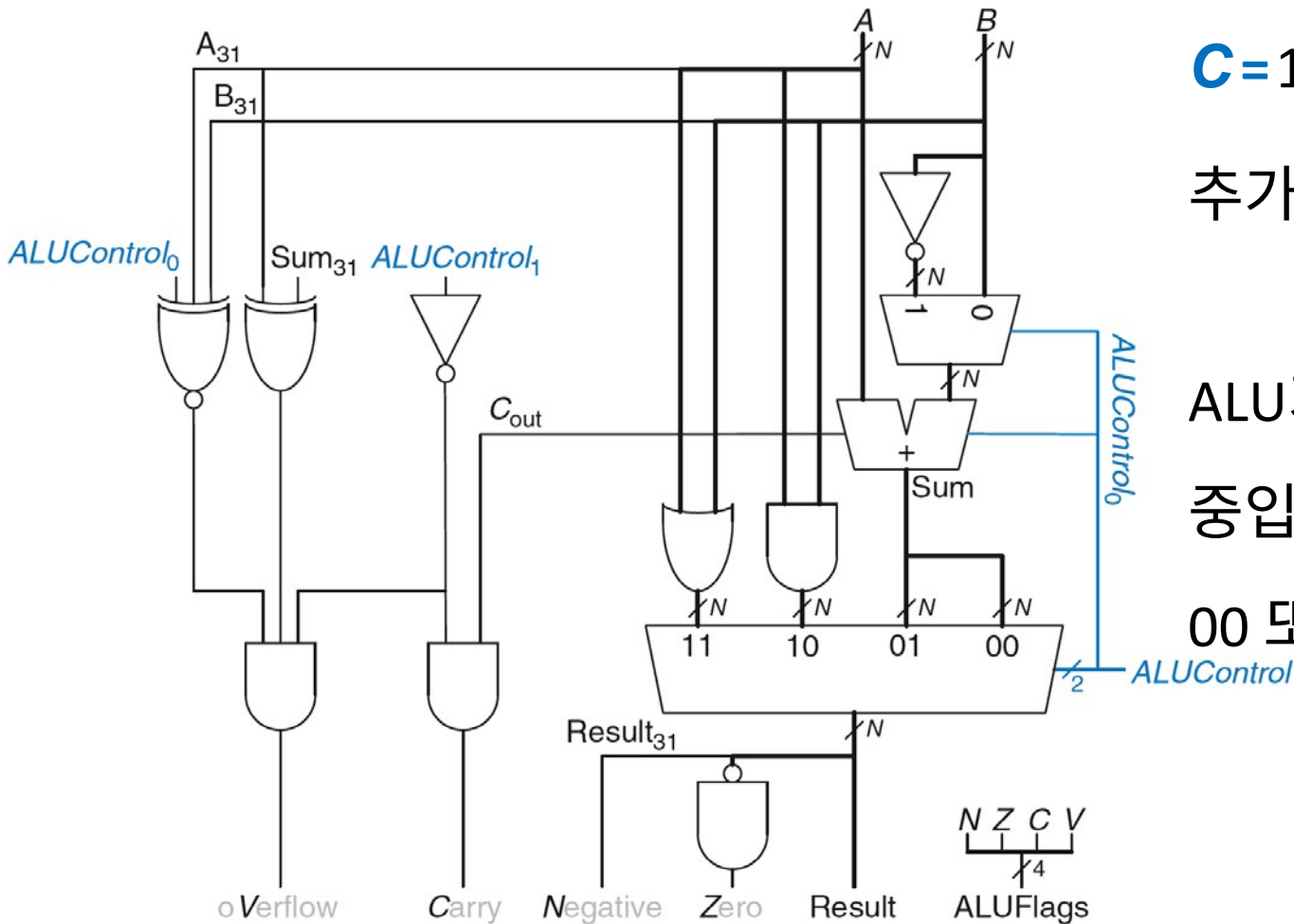
# 상태 플래그가 있는 ALU: 0



**Z = 10**이면

결과**의** 모든 비트  
는 0

# 상태 플래그가 있는 ALU: Carry



**C**=1이면

추가의 Cout은 1

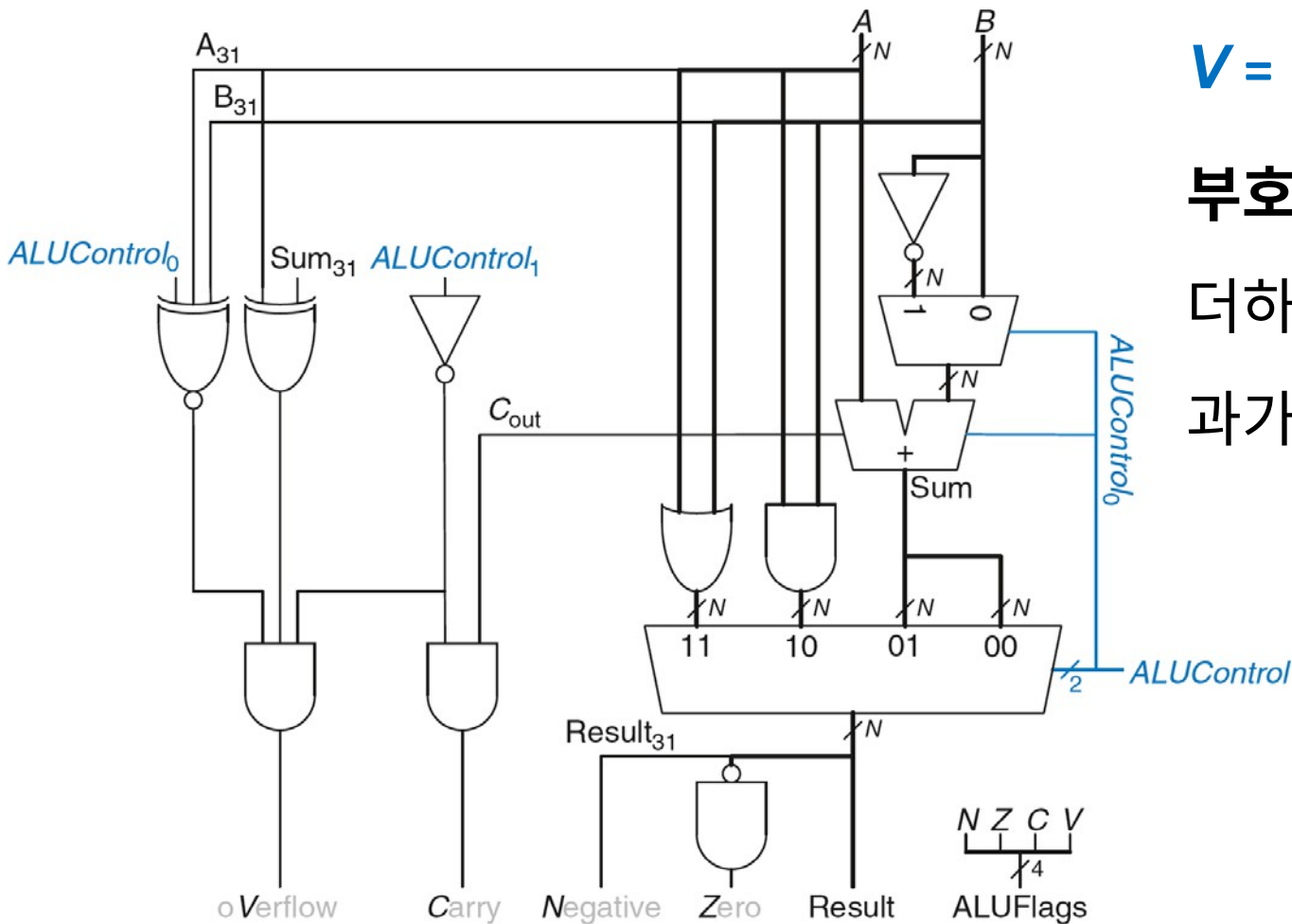
**AND**

ALU가 더하기 또는 빼기

중입니다(ALUControl은

00 또는 01).

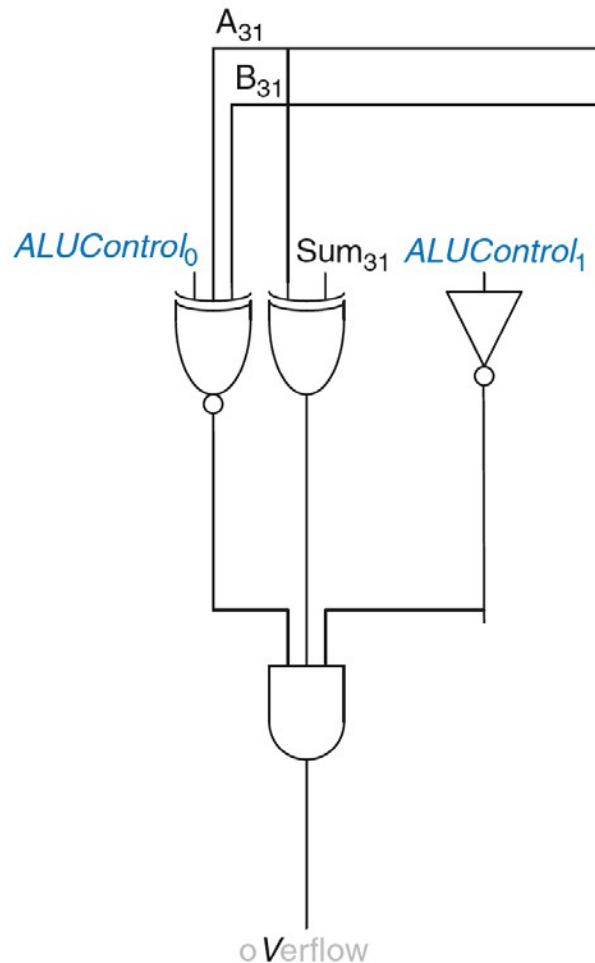
# 상태 플래그가 있는 ALU: oVerflow



$V = 10$ 면

**부호가 같은 숫자 2개를  
더하면 부호가 반대인 결  
과가 생성됩니다.**

# 상태 플래그가 있는 ALU: oVerflow



$V = 1$ 이면

ALU가 덧셈 또는 뺄셈을 수행 중입니다  
( $ALUControl_1 = 0$ ).

**AND**

A와 Sum은 부호가 반대입니다.

**AND**

A와 B는 덧셈에 대해 동일한 부호를 가집니  
다( $ALUControl_0 = 0$ ).

**OR**

A와 B는 뺄셈의 부호가 다릅니다( $ALUControl_0$



= 1).

# 플래그 기준 비교

서명된 플래그와 서명되지 않은 플래그를 빼고 확인  
하여 비교합니다.

비교	서명	서명되지 않음
$==$	$Z$	$Z$
$!=$	$\sim Z$	$\sim Z$
$<$	$N \wedge V$	$\sim C$
$<=$	$Z \mid (N \wedge V)$	$Z \mid \sim C$
$>$	$\sim Z \ \& \ \sim(N \wedge V)$	$\sim Z \ \& \ C$

$\geq$	$\sim(N \wedge V)$	C
--------	--------------------	---

## 5장: 디지털 빌딩 블록

**시프터, 승  
수 및 분배**

기

# 시프트

**논리 시프트:** 값을 왼쪽 또는 오른쪽으로 이동하고 빈 공간을 0으로 채웁니다.

- 예:  $11001 \gg 2 = 00110$
- 예:  $11001 \ll 2 = 00100$

**산술 시프트:** 논리 시프트와 동일하지만 오른쪽 시프트에서 빈 공간을 이전 최댓값 비트(msb)로 채웁니다.

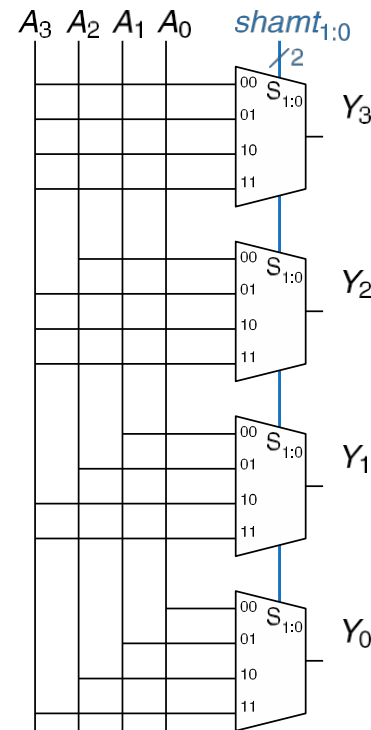
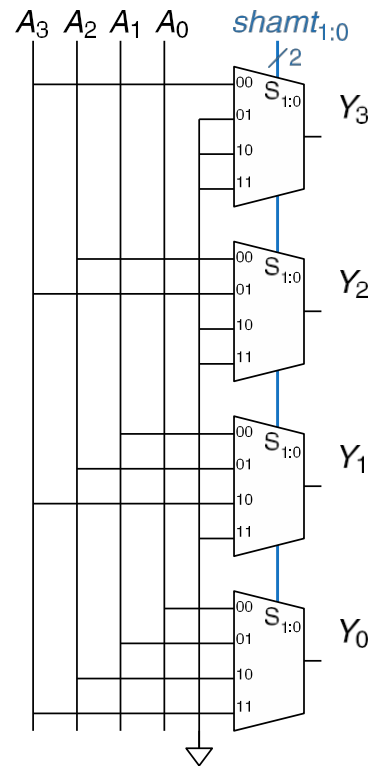
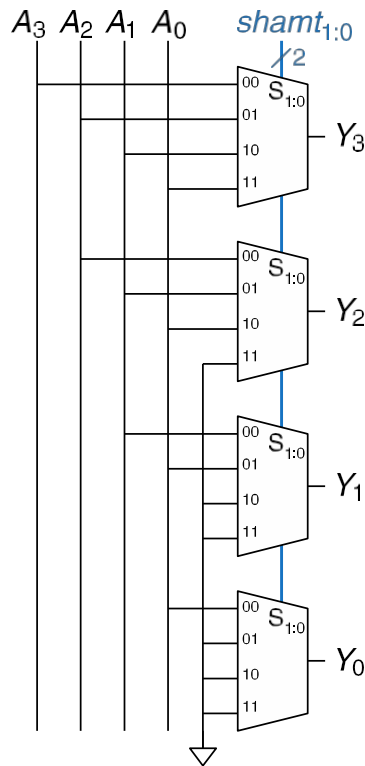
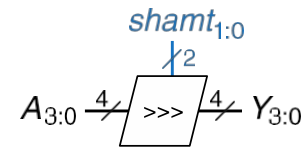
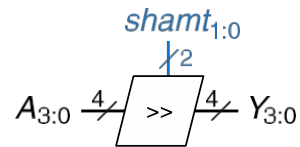
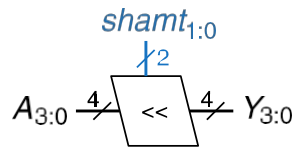
- 예:  $11001 \ggg 2 = 11110$
- 예:  $11001 \lll 2 = 00100$

**로테이터:** 비트가 원을 그리며 회전하여 한쪽 끝에서 이동한 비트가 다른 쪽 끝으로 이동합니다.

- 예: 11001 ROR 2 = 01110
- 예: 11001 ROL 2 = 00111

# 시프트 디자인

왼쪽 시프트논리적 시프트 오른쪽    산술 시프트 오른쪽





# 곱셈기와 나눗셈기로서의 시프터

- $A \ll N = A \times 2^N$

- 예:  $00001 \ll 3 = 01000$  ( $1 \times 2^3 = 8$ )

- 예:  $11101 \ll 2 = 10100$  ( $-3 \times 2^2 = -12$ )

- $a \gg n = a \div 2^N$

- 예:  $01000 \gg 1 = 00100$  ( $8 \div 2^1 = 4$ )

- 예:  $10000 \gg 2 = 11100$  ( $-16 \div 2^2 = -4$ )

## 5장: 디지털 빌딩 블록

# 카운터 및 교 대근무 레지스

터

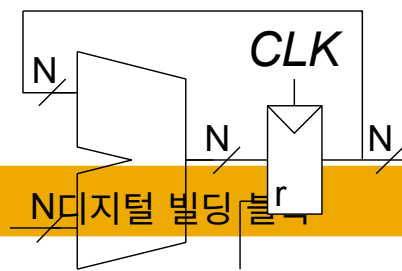
# 카운터

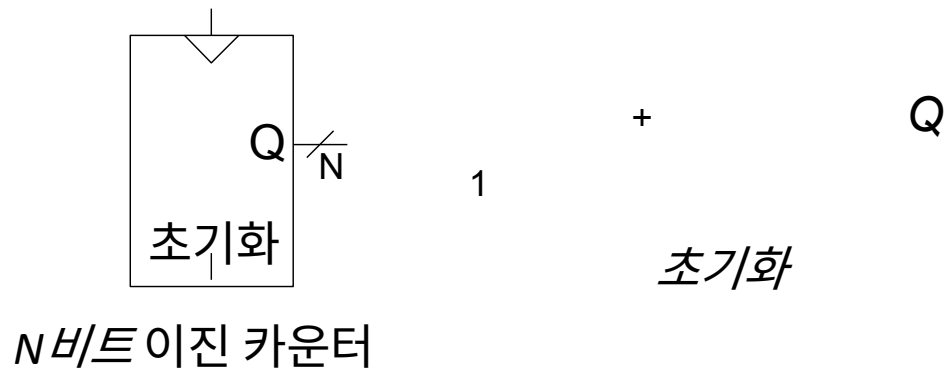
- 각 시계 가장자리의 증분
- 숫자를 순환하는 데 사용됩니다. 예를 들어,  
-000 , 001, 010, 011, 100, 101, 110, 111, 000, 001...
- **사용 예시:**
  - 디지털 시계 디스플레이
  - 프로그램 카운터: 현재 실행 중인 명령어를 추적합니다.

기호

구현

CLK

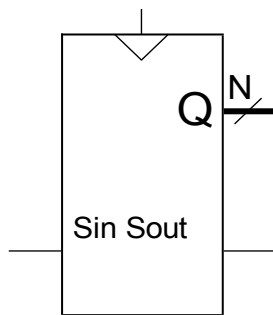




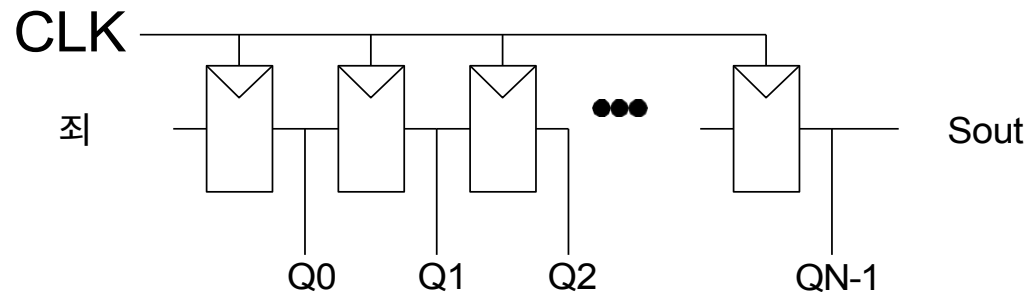
# 교대근무 레지스터

- 각 클럭 가장자리에서 새 비트를 이동합니다.
- 각 시계 가장자리에서 약간 밖으로 이동
- 직렬-병렬 *변환기*: 직렬 입력( $S_{in}$ )을 병렬 출력( $Q0:N-1$ )으로 변환합니다.

기호:

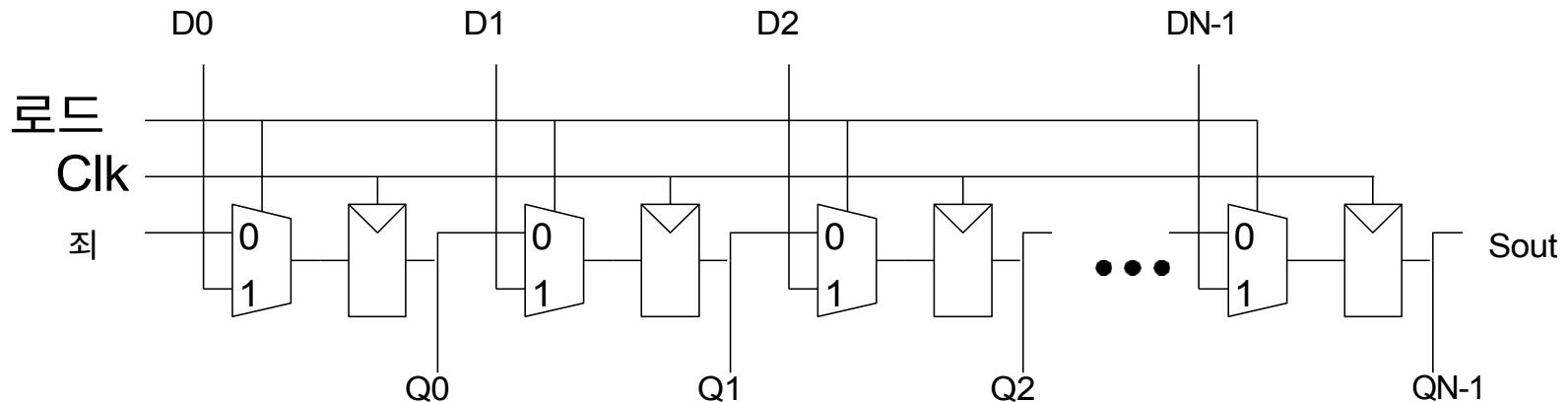


구현:



# 병렬 부하가 있는 시프트 레지스터

- $Load = 1$ 이면 일반 N비트 레지스터로 작동합니다.
- $로드 = 0$ 일 때, 시프트 레지스터 역할을 합니다.
- 이제 직렬-병렬 변환기 역할을 할 수 있습니다(Sin에서  $Q0_{:N-1}$ ) 또는 병렬-직렬 변환기( $D0_{:N-1} \sim Sout$ )



# 5장: 디지털 빌딩 블록

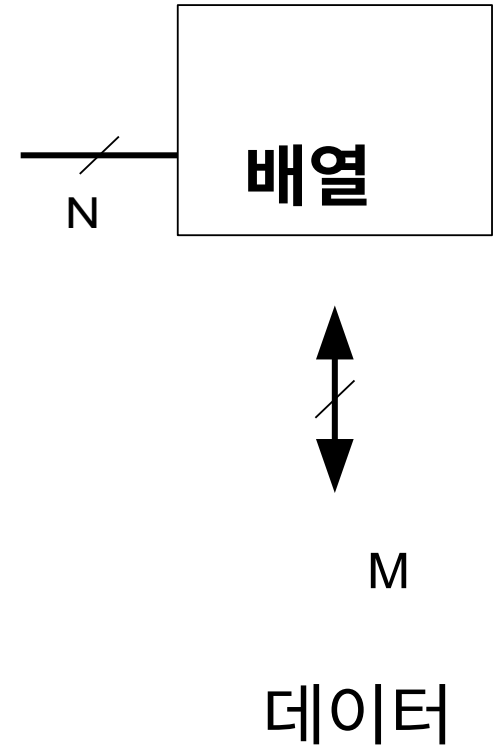
## 메모리

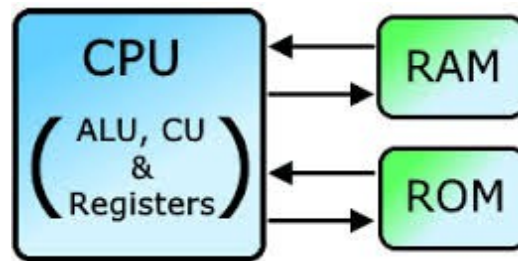


# 메모리 어레이

- 대용량 데이터의 효율적인 저장
- M비트 데이터 값 읽기/쓰기 각각  
고유한 N비트 주소
- 3가지 일반적인 유형  
:
  - 동적 랜덤 액세스 메모리(DRAM)
  - SRAM(정적 랜덤 액세스 메모리)
  - 읽기 전용 메모리(ROM)

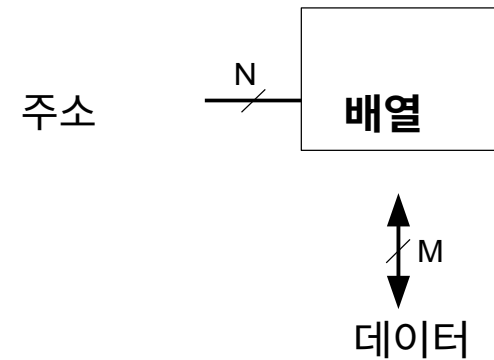
주소





# 메모리 어레이

- 비트 셀의 2차원 배열
- 각 비트 셀은 하나의 비트를 저장합니다.
- $N$ 개의 주소 비트와  $M$ 개의 데이터 비트:
  - $2^N$  행과  $M$  열
  - **깊이**: 행 수(단어 수)
  - **너비**: 열 수(단어 크기)
  - **배열 크기**: 깊이  $\times$  너비  $= 2^N \times M$

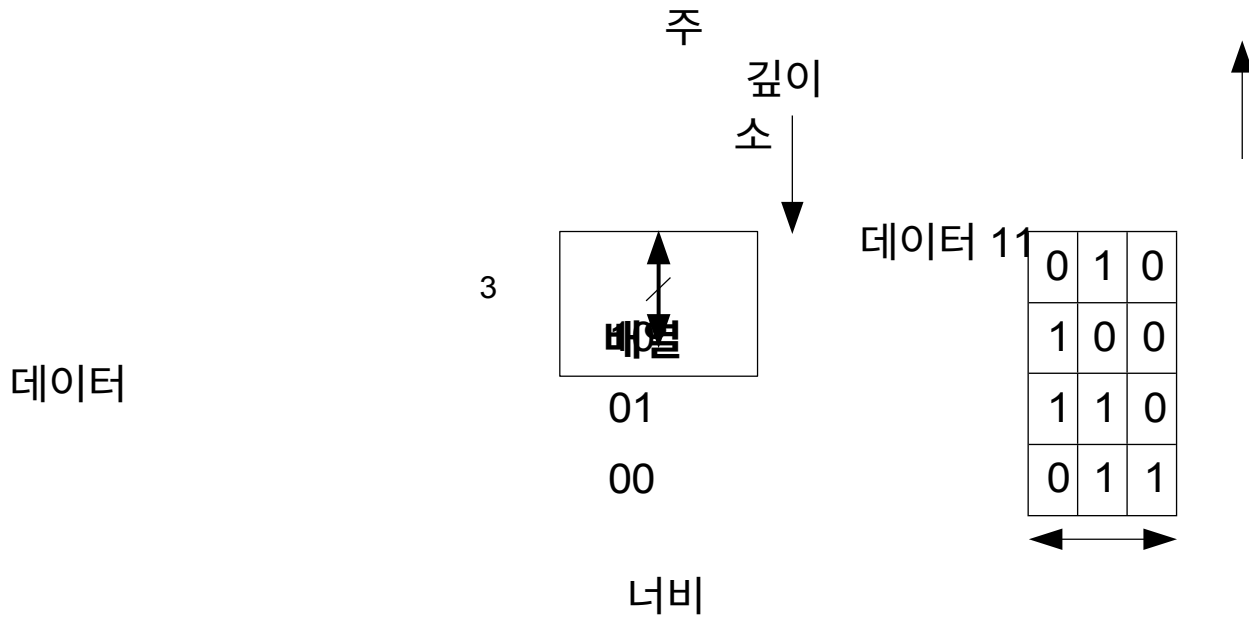


—

주

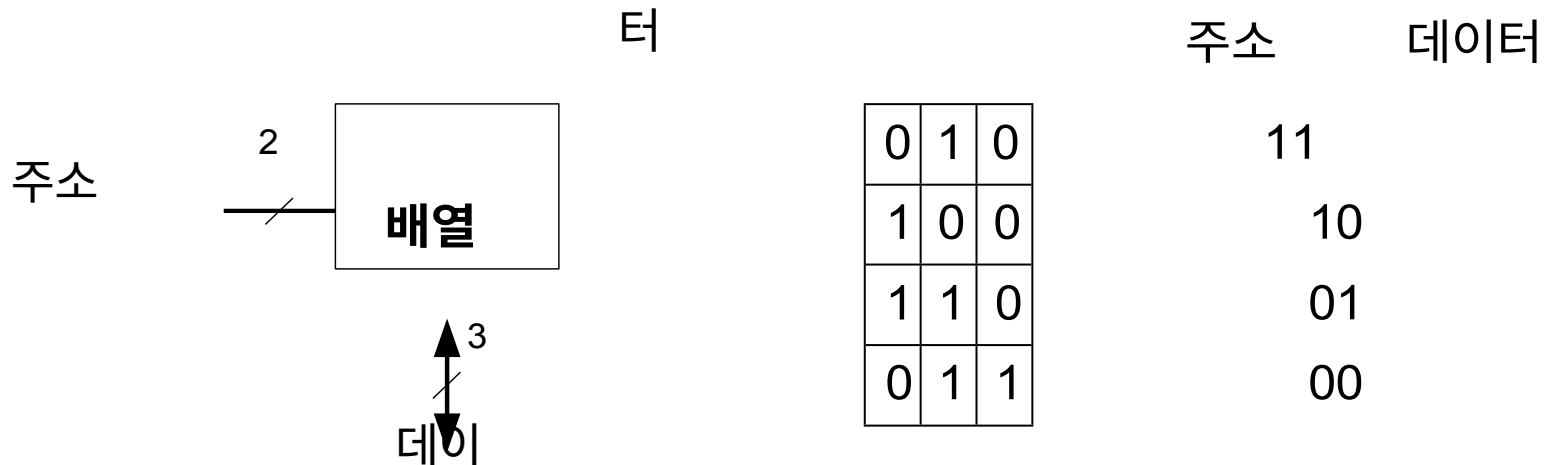
소

2



# 메모리 배열 예제

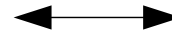
- $2^2 \times 3$ 비트 배열
- 단어 수 4
- 단어 크기: 3비트
- 예를 들어 주소 10에 저장된 3비트 단어는 100입니다.



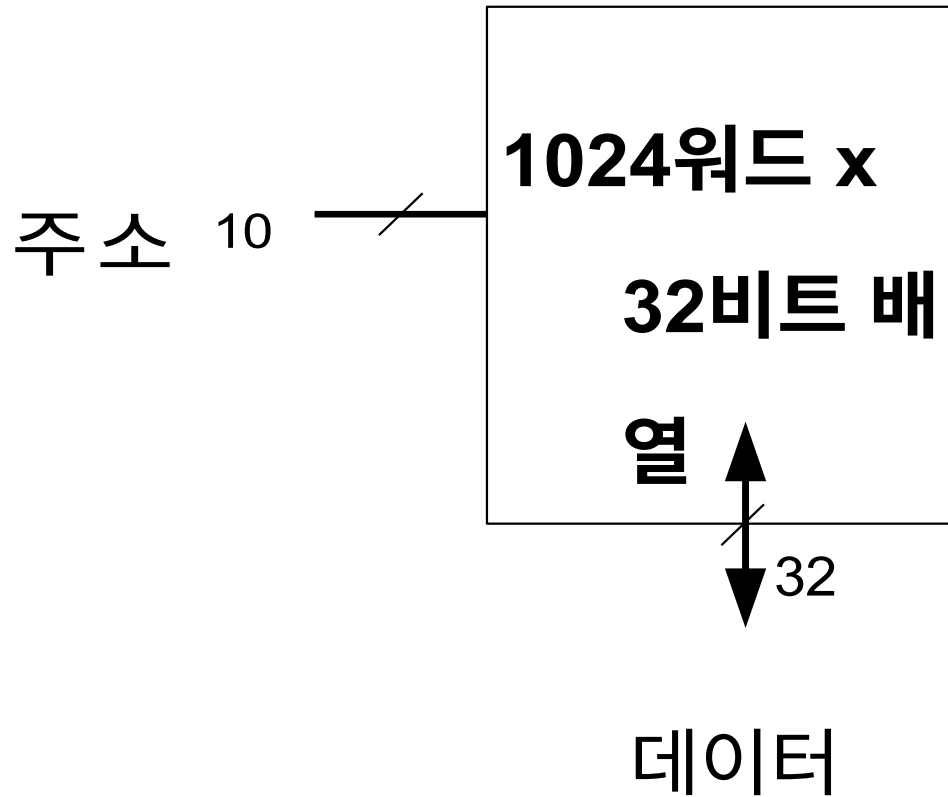
너비

깊이

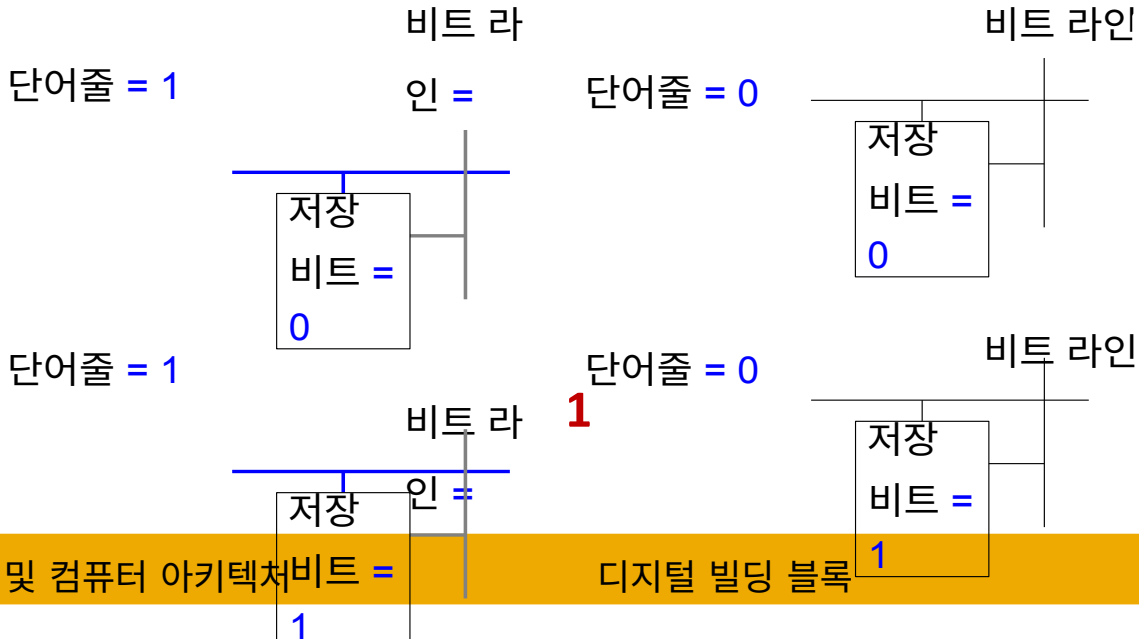
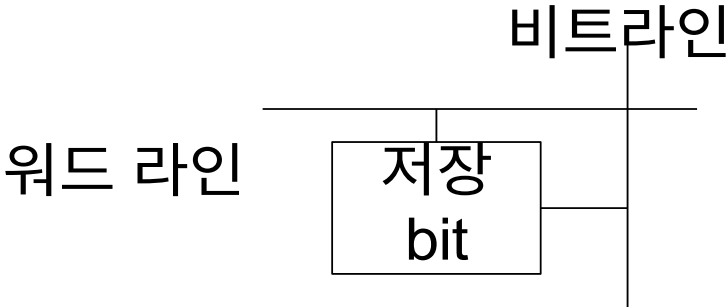
이



# 메모리 어레이



# 메모리 어레이 비트 셀





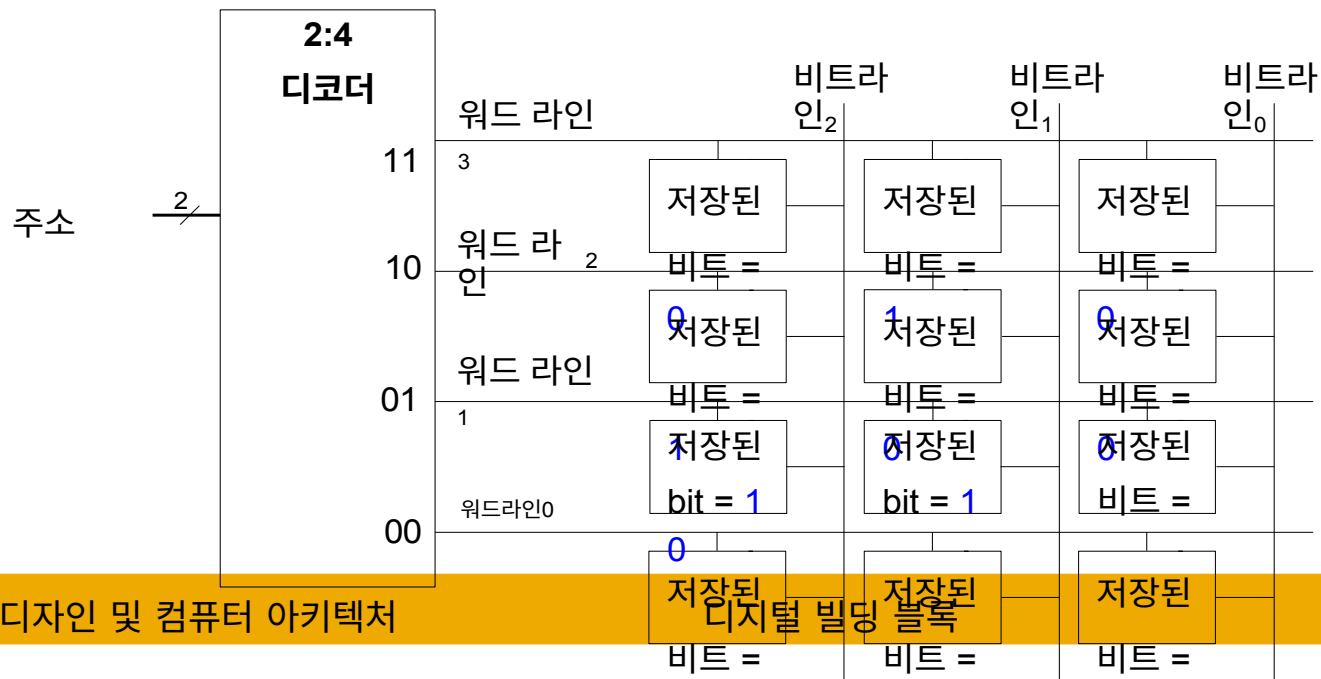
(a)

(b)

# 메모리 배열

## • 워드 라인:

- 활성화처럼
- 메모리 배열의 단일 행 읽기/쓰기
- 는 고유 주소에 해당합니다.
- 한 번에 한 단어 라인만 **HIGH**

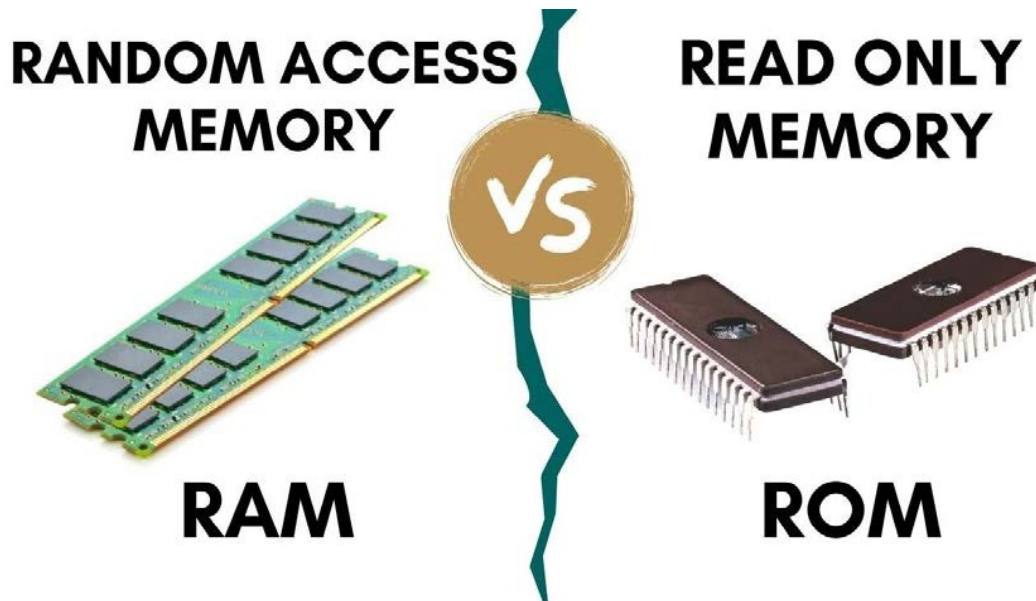


데이터 데이터<sub>21</sub>

데이터<sub>0</sub>

# 메모리 유형

- RAM(랜덤 액세스 메모리): **휘발성**
- 읽기 전용 메모리(ROM): **비휘발성**



# RAM: 랜덤 액세스 메모리

- **휘발성:** 전원이 꺼지면 데이터가 손실됨
- 빠른 읽기 및 쓰기
- 컴퓨터의 주 메모리는 RAM(DRAM)입니다.

(테이프 레코더와 같은 순차 액세스 메모리와 달리) 모든 데이터 단어가 다른 데이터 단어와 동일한 지연 시간으로 해당 주

소를 사용하여 액세스되기 때문에 역사적으로 **랜덤 액세스** 메모리라고 불립니다.

# ROM: 읽기 전용 메모리

- **비휘발성:** 전원이 꺼져도 데이터 유지
- 빠르게 읽지만 쓰기가 불가능하거나 느림
- 카메라, 썸 드라이브, 디지털 카메라의 플래시 메모리는 모두 ROM입니다.

ROM은 제작 시 또는 퓨즈를 태워 기록했기 때문에 과거에는 

**기 전용** 메모리라고 불렸습니다. ROM은 한 번 구성되면 다시 쓸 수 없었습니다. 하지만 플래시 메모리와 다른 유형의 ROM은 더 이상 그렇지 않습니다.



# 5장: 디지털 빌딩 블록

## RAM

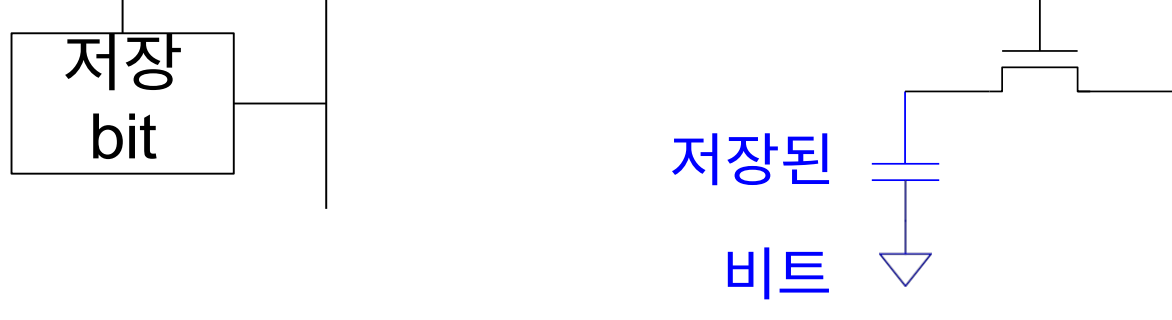
# RAM 유형

- **DRAM**(동적 랜덤 액세스 메모리)
- **SRAM**(정적 랜덤 액세스 메모리)
- 데이터를 저장하는 방식이 다릅니다:
  - DRAM은 커패시터를 사용합니다.
  - SRAM은 크로스 커플링 인버터를 사용합니다.

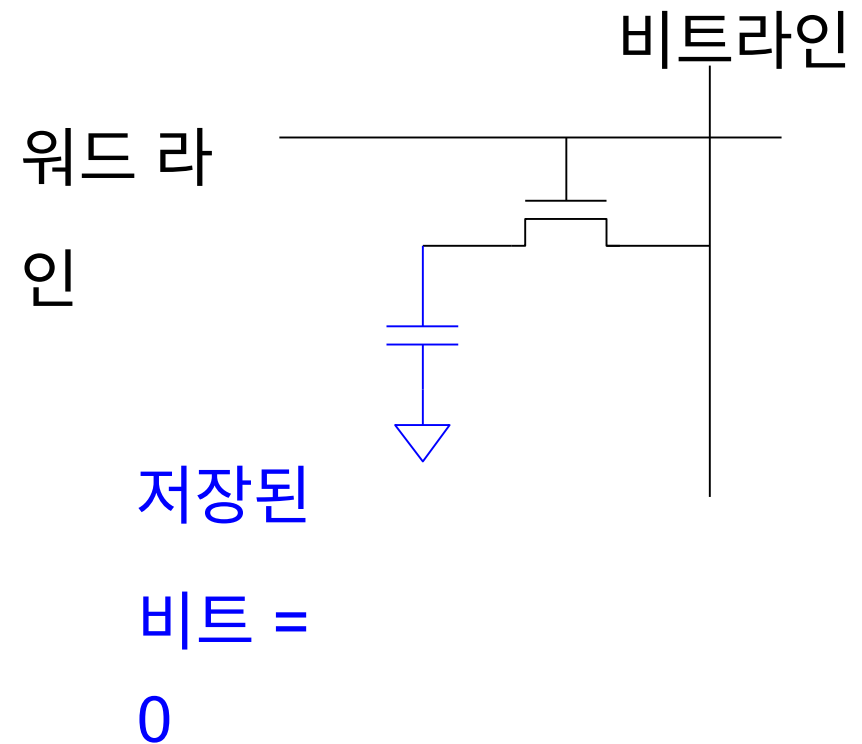
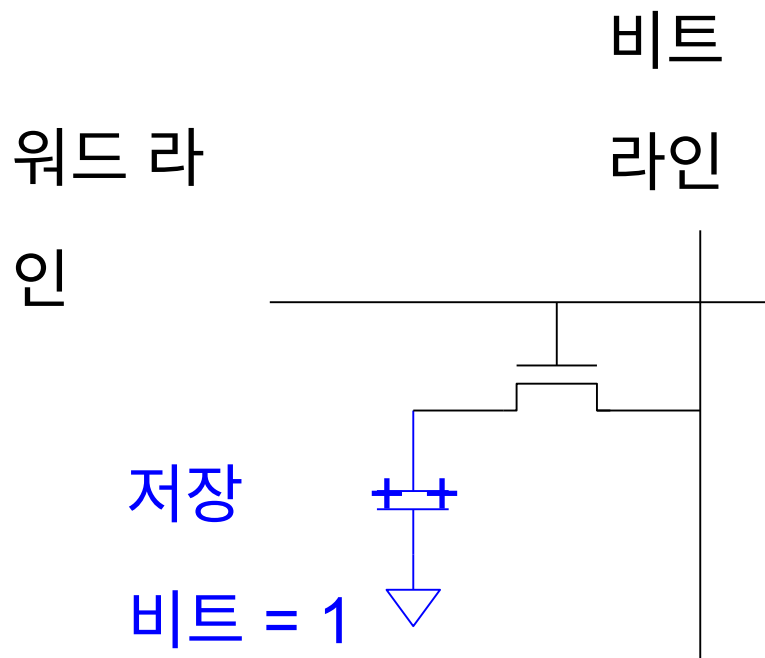
# DRAM

- **커패시터에** 저장된 데이터 비트
- **동적**: 값을 **새로 고쳐야** 합니다.  
(재작성) 이후 주기적으로
  - 읽기는 커패시터에 저장된 값을 파괴합니다.
  - 커패시터에서 전하가 누설되면 값이 저하됩니다.

비트		비트라인	
워드 라인	라인	워드 라인	
인		인	

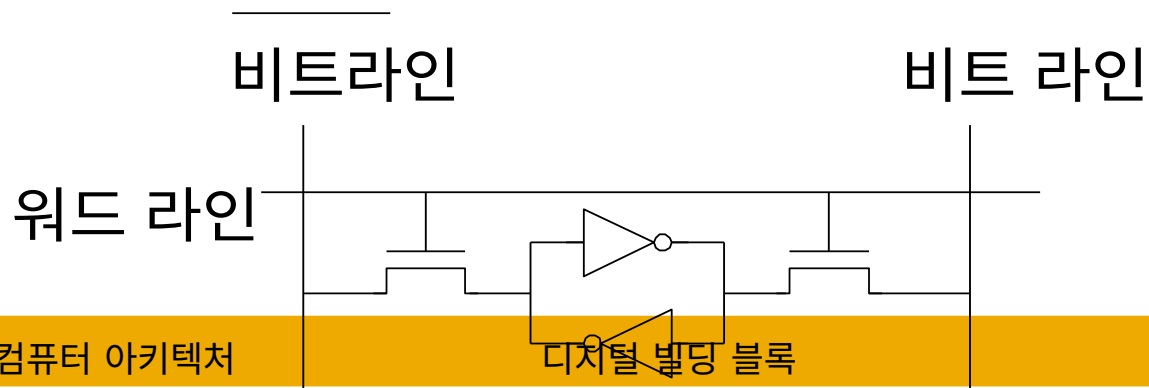
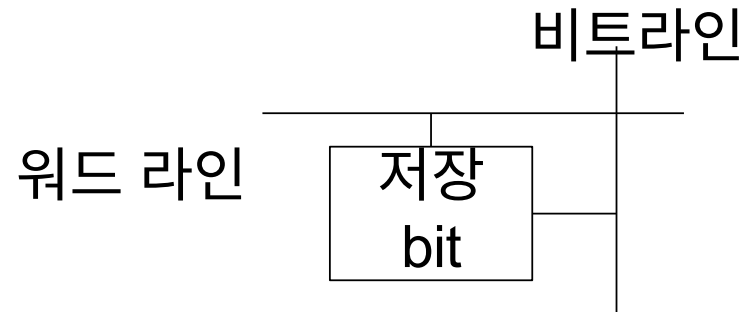


# DRAM

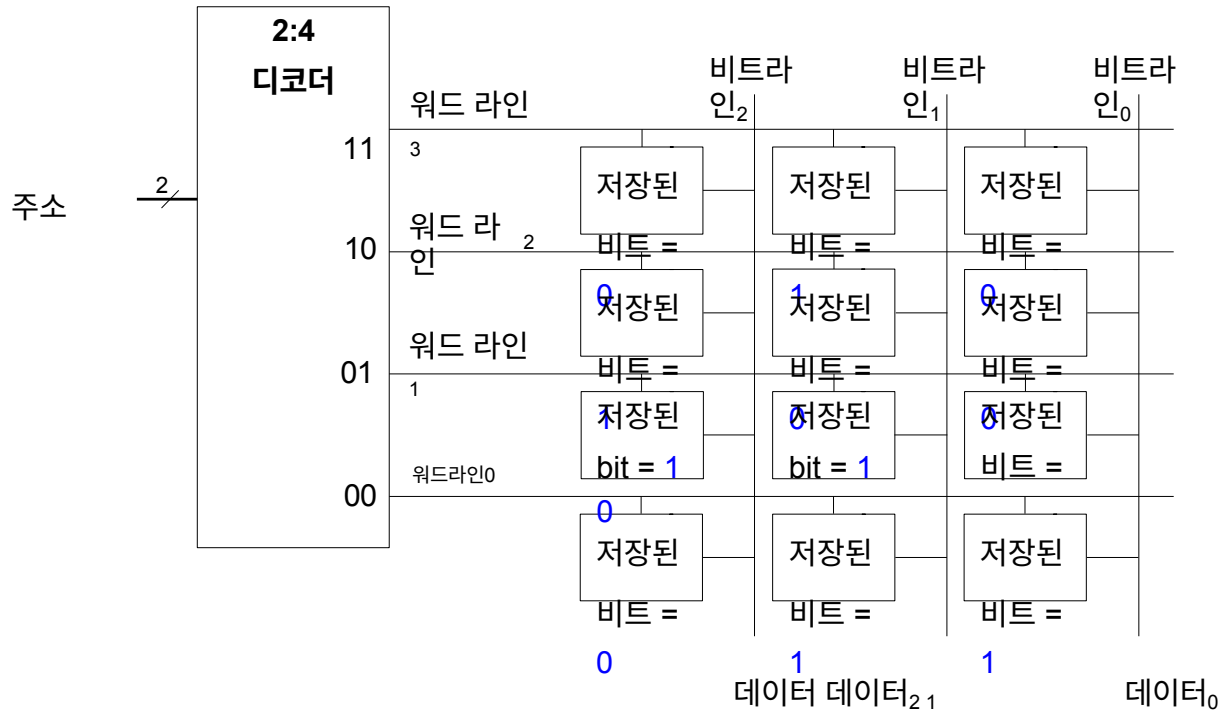


# SRAM

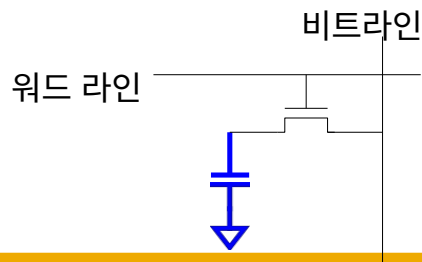
- 정적: 값을 **새로 고칠** 필요가 없습니다.



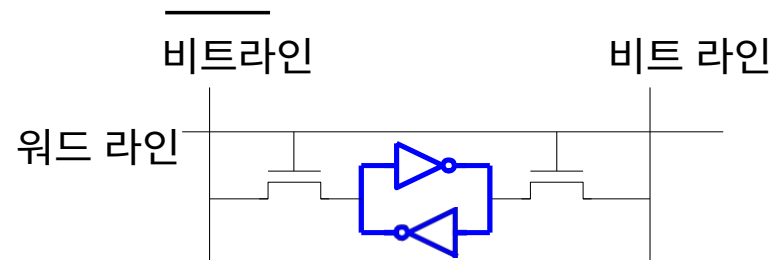
# 메모리 어레이 검토



DRAM 비트 셀:



SRAM 비트 셀:



# SRAM 대 DRAM



	SRAM	DRAM
속도	더 빠르게	느린
밀도	낮음	높음
용량	Small	대형
전력 소비	낮음	높음

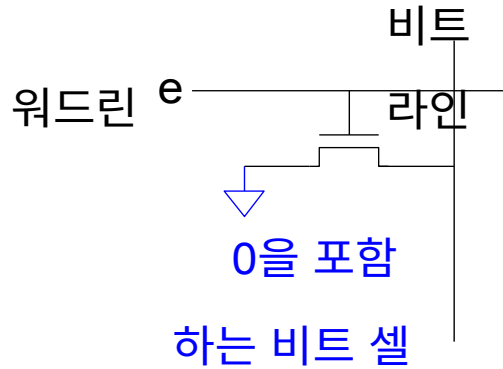


비용	비싸다	저렴한
다음과 같이 사용	캐시 메모리	메인 메모리

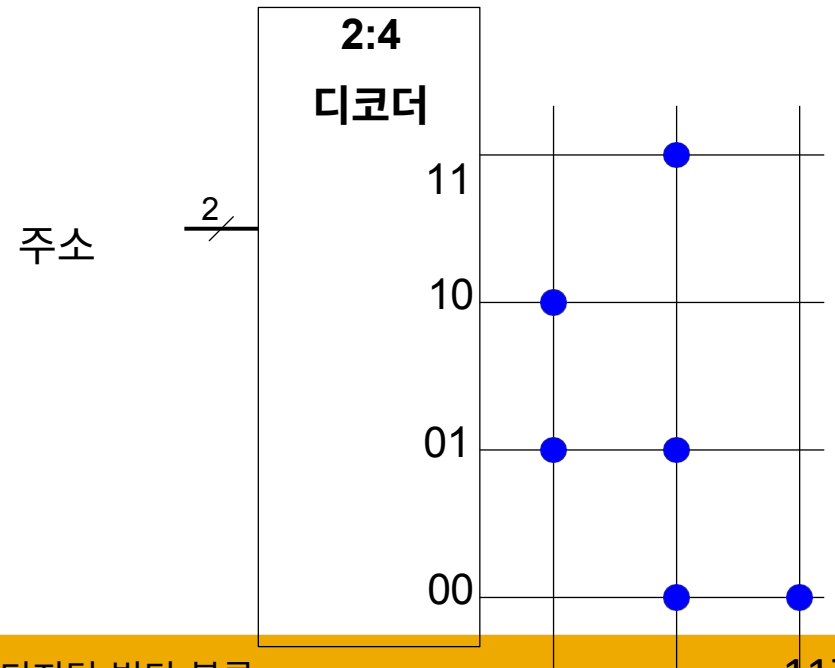
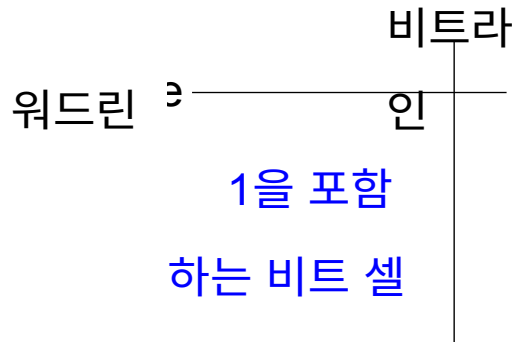
# 5장: 디지털 빌딩 블록

## ROM

# ROM: 도트 표기법



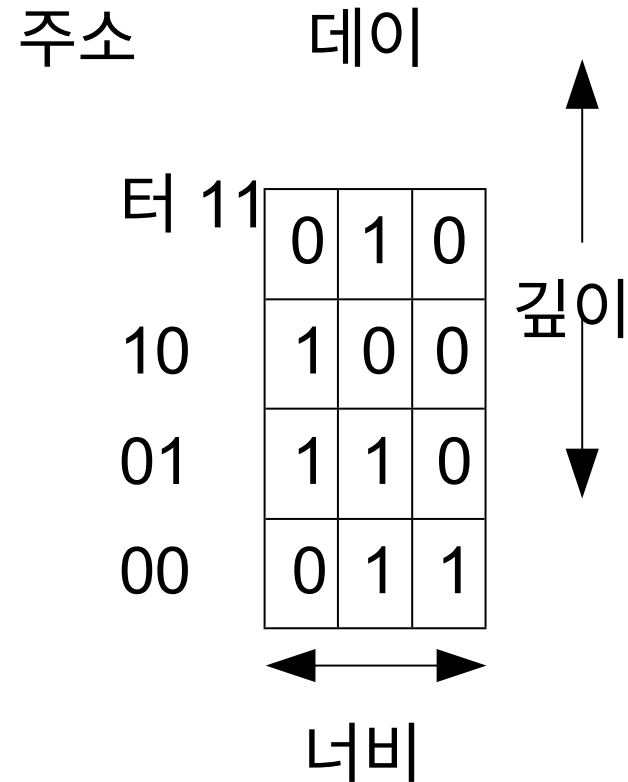
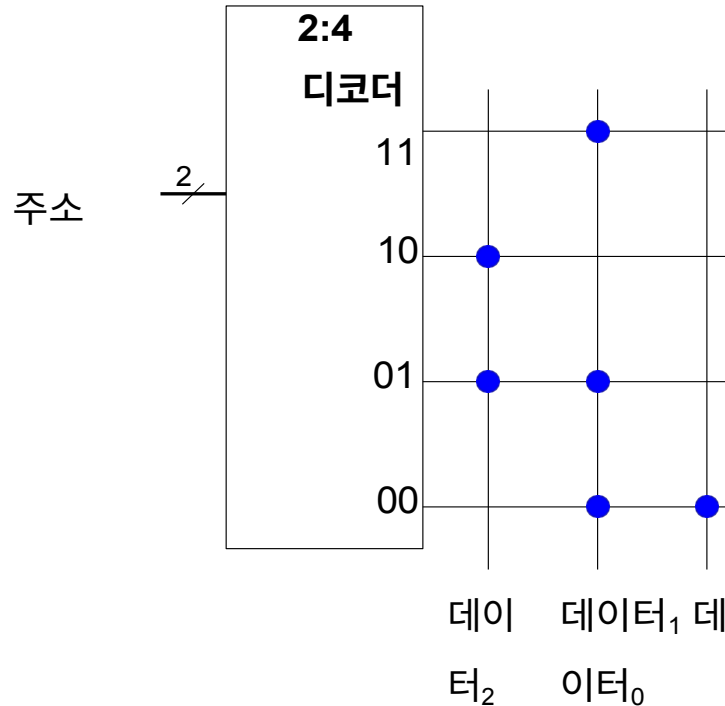
- 트랜지스터의 유무로 비트를 저장합니다.
- 비트라인을 HIGH로 당기고 워드라인을 켜면 됩니다,
  - 트랜지스터가 있으면 비트 라인을 LOW로 당깁니다.
  - 그렇지 않으면, 비트선은 HIGH로 유지됩니다.



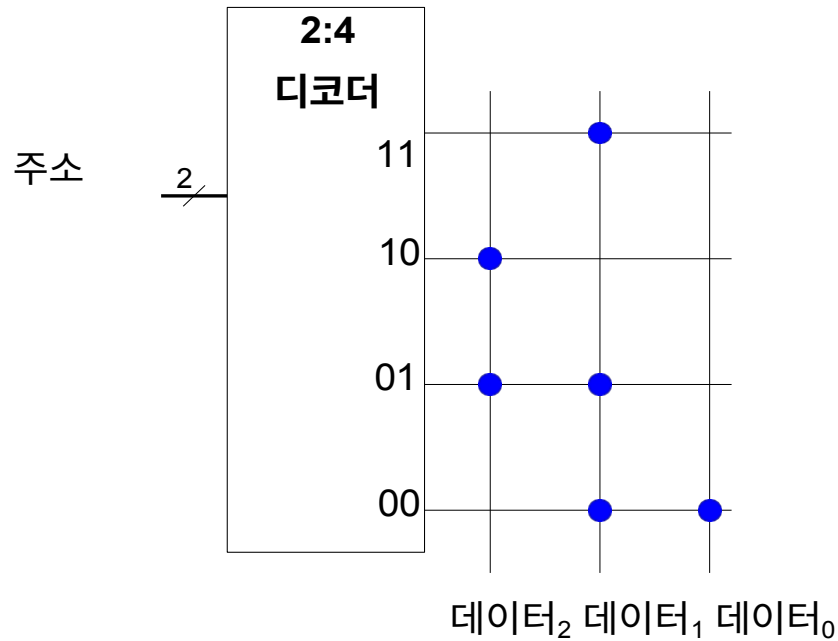
- 행과 열의 교차점에 있는 *점*은 데이터 비트가 1임을 나타냅니다.

데이터<sub>2</sub> 데이터<sub>1</sub> 데이터<sub>0</sub>

# ROM 스토리지



# ROM 로직



$$\text{데이터}_2 = A_1 \oplus A_0$$

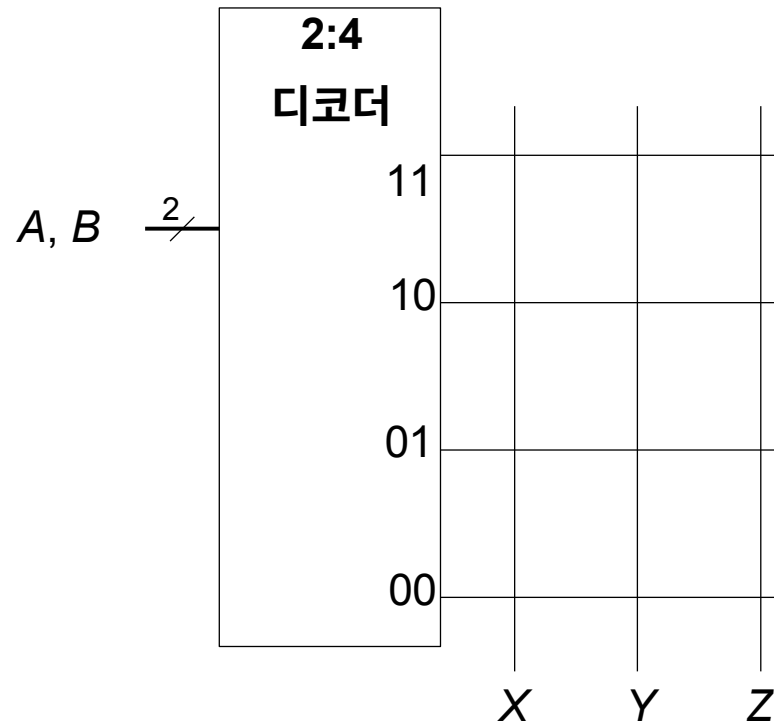
$$\overline{\text{데이터}_1} = \overline{A_1}$$

$$A_0 \text{ 데이터}_0 = \overline{A_1 A_0}$$

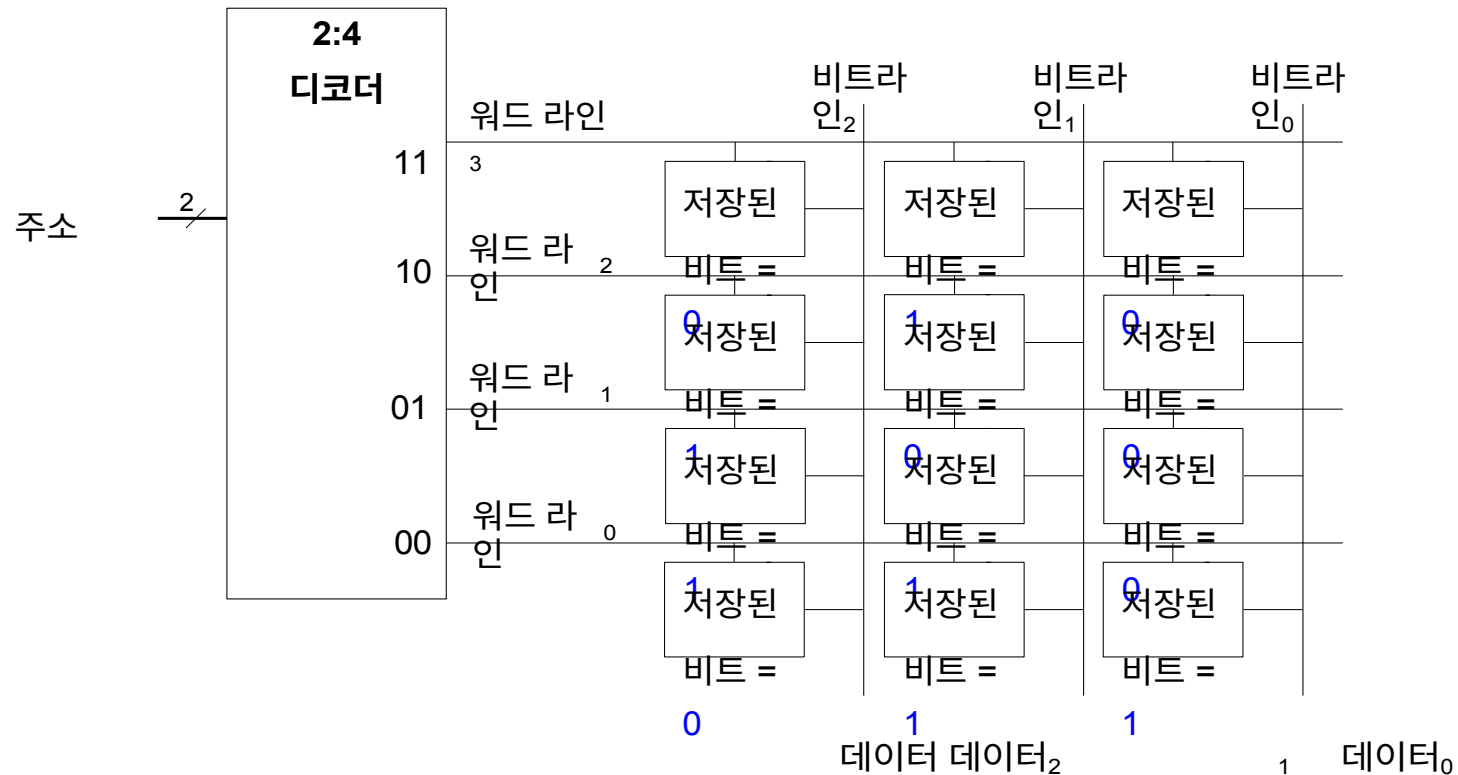
# 예시: ROM을 사용한 로직

$2^2 \times 3$ 비트 ROM을 사용하여 다음 논리 함수를 구현합니다:

- $X = AB$
- $Y = A + B$
- $Z = A \overline{B}$



# 모든 메모리 배열을 사용한 로직



$$\text{데이터}_2 = A_1 \oplus A_0$$

$$\text{데이터}_1 = A_1 +$$

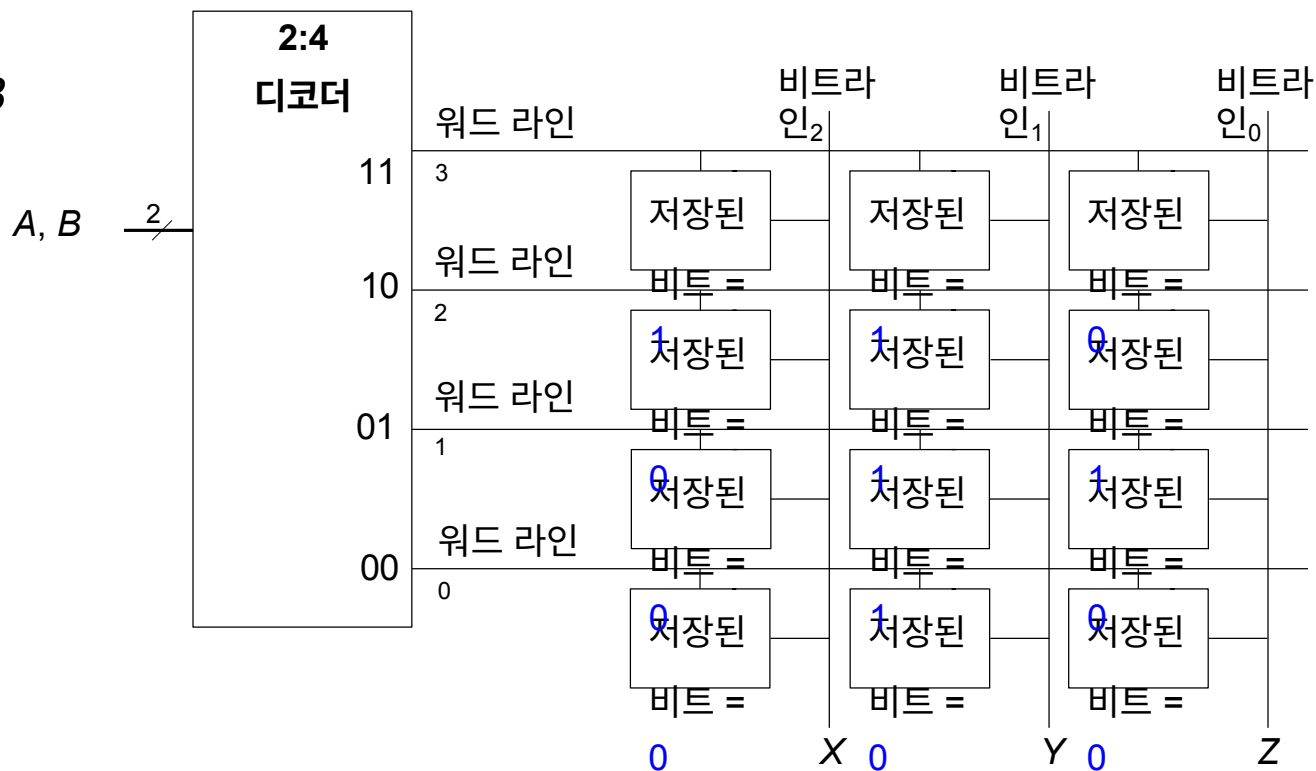


$$A0 \text{ 데이터} = A1A0$$

# 메모리 배열을 사용한 로직

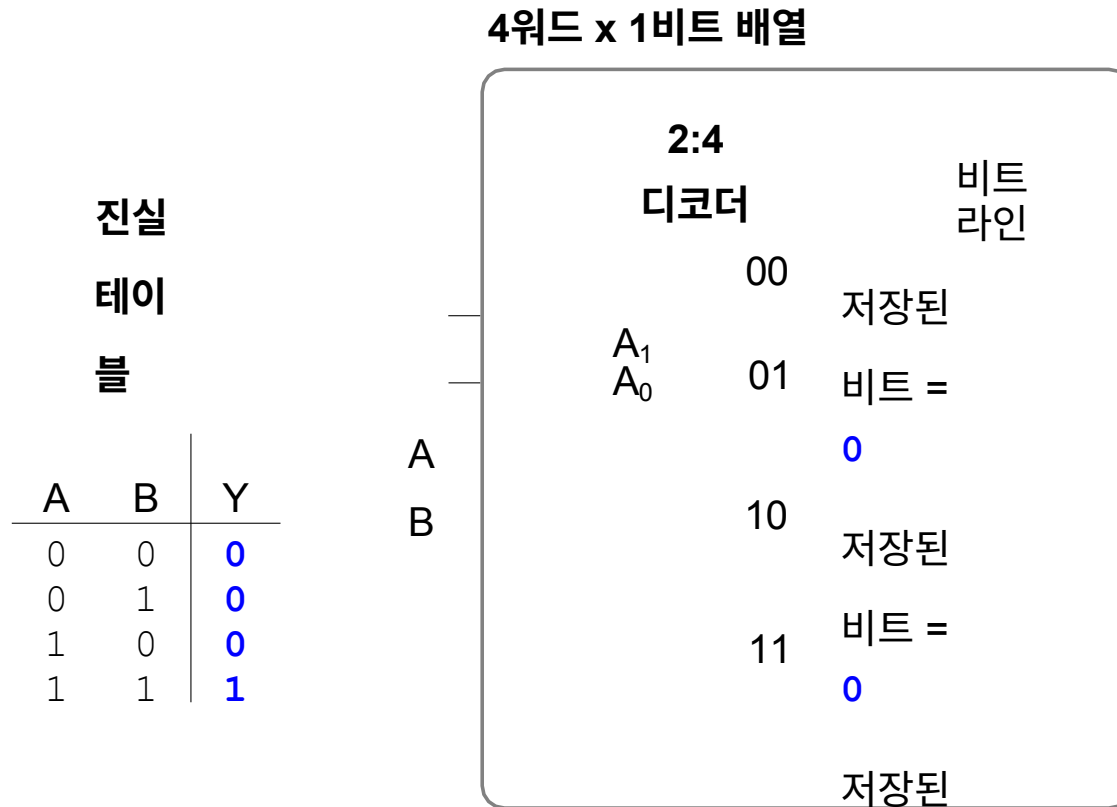
$2^2 \times 3$ 비트 메모리 배열을 사용하여 다음 논리 함수를 구현합니다:

- $X = AB$
- $Y = A + B$
- $Z = A \overline{B}$



# 메모리 배열을 사용한 로직

각 입력 조합(주소)에서 출력을 조회하는 **룩업 테이블(LUT)**로 사용할 수 있습니다.



Y

## 5장: 디지털 빌딩 블록

**로직 어레이: PLA**

**및 FPGA**

# 논리 배열

- 모든 기능을 수행하도록 연결을 구성할 수 있는 프로그래밍 가능 게이트 어레이
- 대량으로 생산되므로 저렴합니다.
- 재구성이 가능하여 하드웨어 교체 없이 설계를 수정할 수 있습니다.
  - 재구성 가능성은 개발 과정에서 유용하며, 새로운 구성을 다운로드하기만 하면 시스템을 업그레이드할 수 있기 때문에 현장에서도 유용합니다.

# 로직 배열의 유형

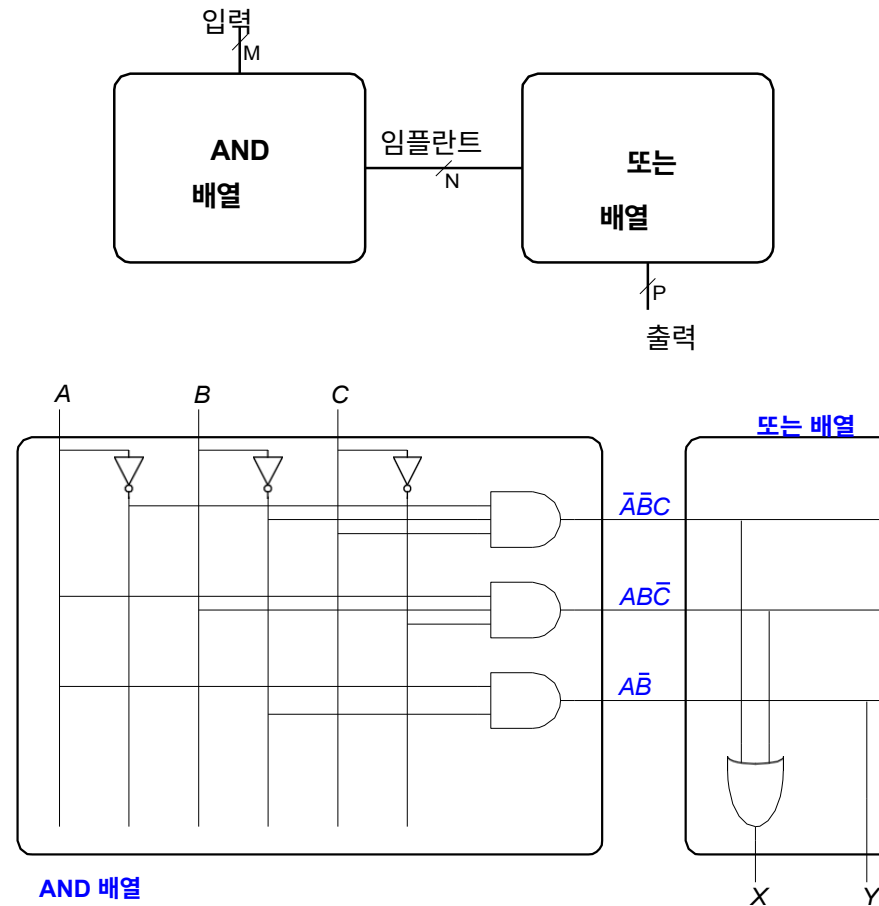
- **PLA**(프로그래머블 로직 어레이)
  - AND 배열 다음에 OR 배열
  - 조합 논리만 해당
  - 내부 연결 수정
- **FPGA**(필드 프로그래머블 게이트 어레이)
  - 논리 요소 배열(LE)
  - 조합 및 순차 논리

## – 프로그래밍 가능한 내부 연결



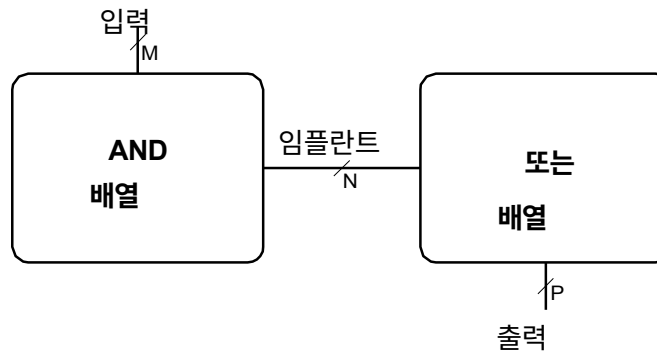
# PLA: 프로그래머블 로직 어레이

- $X = \overline{A}\overline{B}C + AB\overline{C}$
- $Y = A\overline{B}$

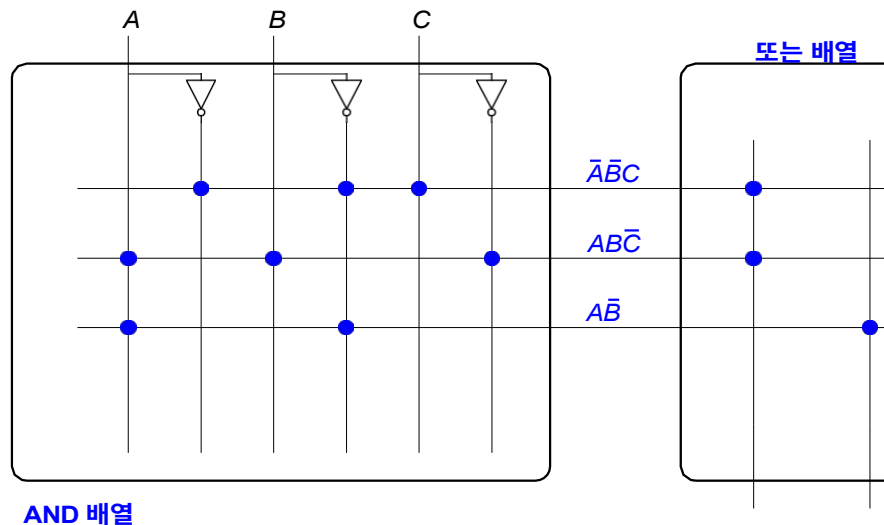


# PLA: 점 표기법

- $X = \overline{A}\overline{B}C + A\overline{B}\overline{C}$
- $Y = A\overline{B}$



- AND 배열의 점: 암시자를 구성하는 리터럴
- OR 배열의 점: 임플란트가 출력 함수의 일부인 경우



X

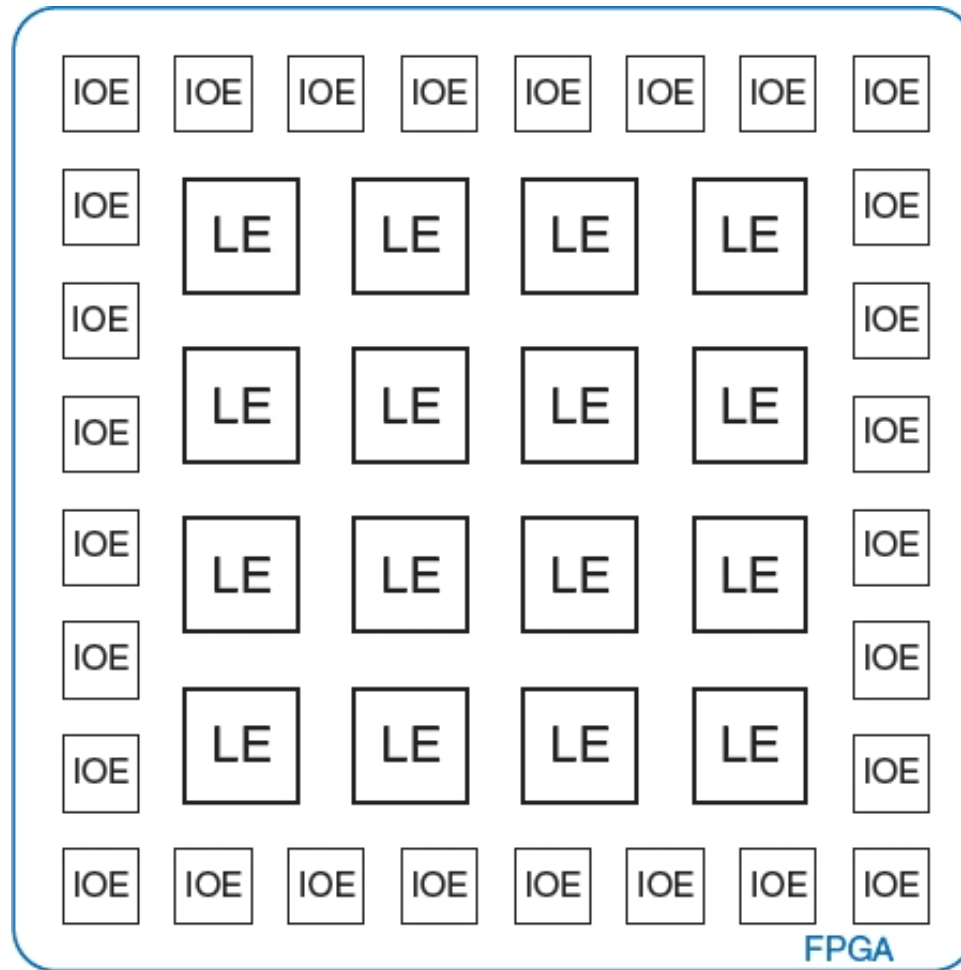
Y

# FPGA: 필드 프로그래머블 게이트 어레이

- 구성

- **LE**(논리 요소): 조합 또는 순차적 함수 수행
- **IOE**(입력/출력 요소): 외부 세계와의 인터페이스
- **프로그래밍 가능한 상호 연결**: LE와 IOE 연결
- 일부 FPGA에는 배율기 및 RAM과 같은 다른 빌딩 블록이 포함되어 있습니다.

# 일반 FPGA 레이아웃

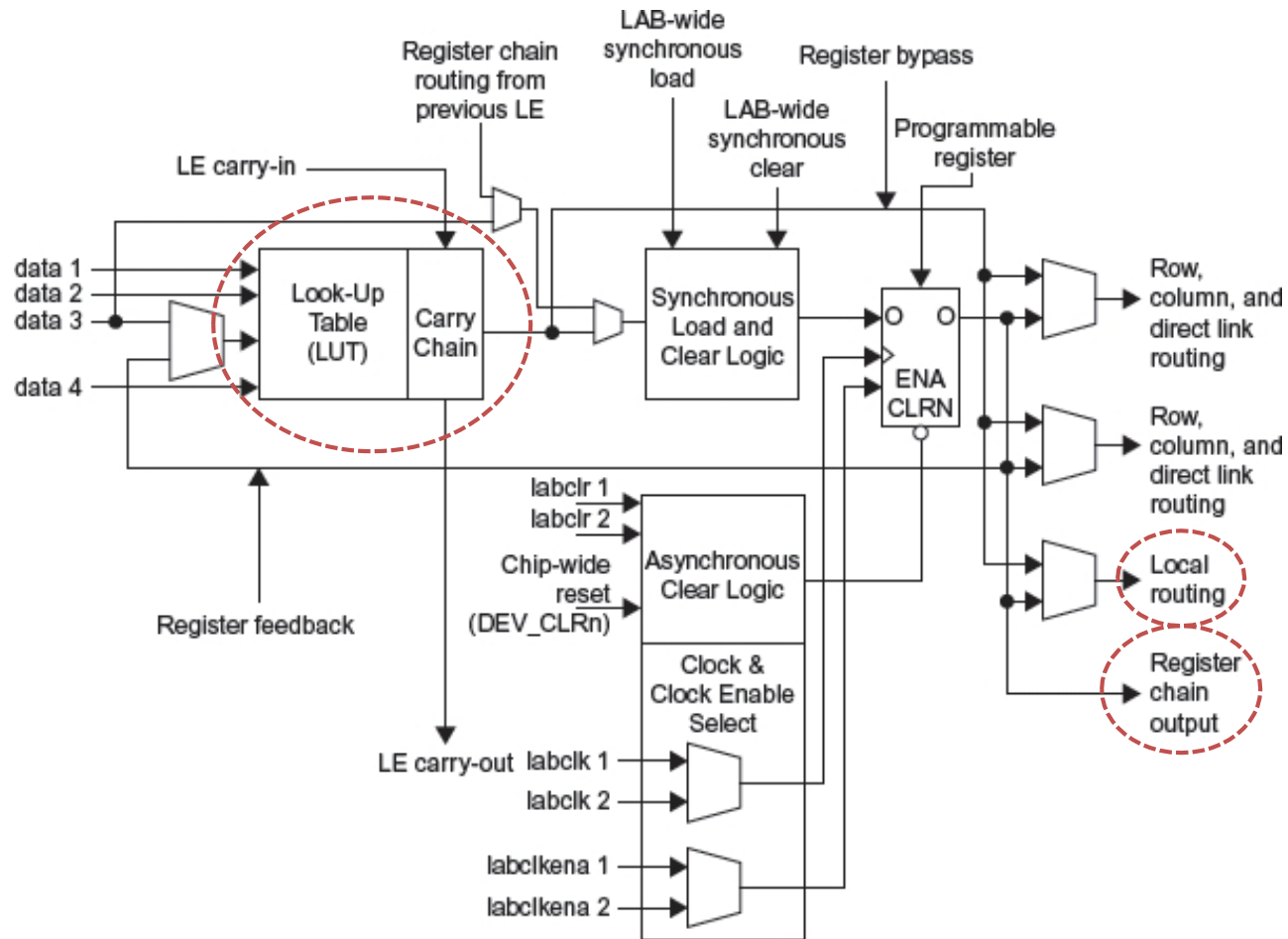


# IF: 논리 요소

- 구성

- **LUT**(룩업 테이블): 조합 논리 수행
- **플립플롭**: 순차 논리 수행
- **멀티플렉서**: LUT 및 플립플롭 연결

# 알테라 사이클론 IV IF



## 사이클론 IV 데이터시트에서



# 알테라 사이클론 IV LE

- **알테라 사이클론 IV LE가 있습니다:**
  - 4입력 LUT 1개
  - 등록된 출력 1개
  - 조합 출력 1개

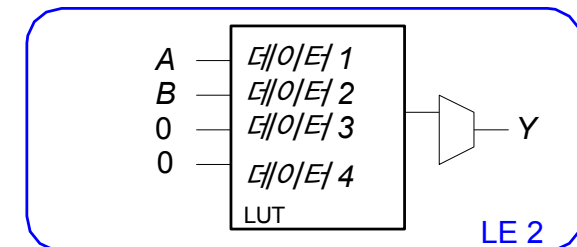
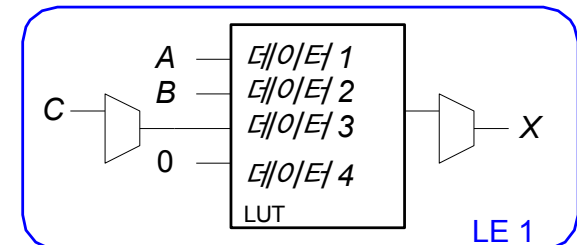
# LE 구성 예시

다음 기능을 수행하도록 Cyclone IV LE를 구성하는 방법을 보여 줍니다:

- $X = \overline{\overline{A}BC} + \overline{ABC}$
- $Y = \overline{AB}$

(A) 데이터 1	(B) 데이터 2	(C) 데이터 3 데이터 4	(X) LUT 출력
0	0	0	X
0	0	1	X
0	1	0	X
0	1	1	X
1	0	0	X
1	0	1	X
1	1	0	X
1	1	1	X

(A) 데이터 1	(B) 데이터 2	(Y) LUT 출력
0	0	X
0	1	X
1	0	X
1	1	X



1      1      x      x      |      0

# 논리 요소 예제 1

구축에 필요한 Cyclone IV LE 수

$$Y = A1 \oplus A2 \oplus A3 \oplus A4 \oplus A5 \oplus A6$$

**솔루션:**

2 LE

먼저  $Y1 = A1 \oplus A2 \oplus A3 \oplus A4$  (4개 변수의 함수)를 계산합니다.

두 번째는  $Y = Y1 \oplus A5 \oplus A6$  (3개 변수의 함수)를 계산합니다.

# 논리 요소 예제 2

구축에 필요한 Cyclone IV LE 수

32비트 2:1 멀티플렉서

**솔루션:**

32 LE

1비트 멀티플렉서는 3개의 변수로 구성되며 하나의 LE에 들어갑니다. 32비트 멀티플렉서는 32개의 복사본이 필요합니다.

# 논리 요소 예제 3

구축에 필요한 Cyclone IV LE 수

2비트 상태, 2입력, 3출력의 임의 FSM

**솔루션:**

5 LE

하나의 LE는 상태 비트와 다음 상태 로직을 보유할 수 있으며, 이는 4개의 변수(상태 비트 2개, 입력 2개)로 이루어진 함수입니다.

하나의 LE는 최대 4개의 변수(상태 2비트, 입력 2비트)의 함수인 비트의 출력을 계산할 수 있습니다.  
따라서 상태에는 2개의 LE가 필요하고 출력에는 3개의 LE가 필요합니다.

# 참고 사항

디지털 디자인 및 컴퓨터 아키텍처 강의 노트

© 2021 사라 해리스와 데이비드 해리스

이러한 노트는 출처를 명시하는 한 교육 및/또는 비상업적 목적으로 사용 및 수정할 수 있습니다.