컴퓨터 구조

3장 컴퓨터 산술과 논리 연산1

안형태

anten@kumoh.ac.kr

디지털관 139호

컴퓨터 산술과 논리 연산

학습목표

- 산술 및 논리연산장치인 ALU의 내부 구성 이해
- 논리 연산의 원리 이해
- 정수 및 부동소수점 수의 표현 방법과 산술 연산 이해

학습내용

- ALU의 구성 요소
- 정수의 표현
- 논리 연산
- 시프트 연산
- 정수의 산술 연산
- 부동소수점 수의 표현
- 부동소수점 산술 연산

컴퓨터 산술과 논리 연산

1. ALU의 구성 요소

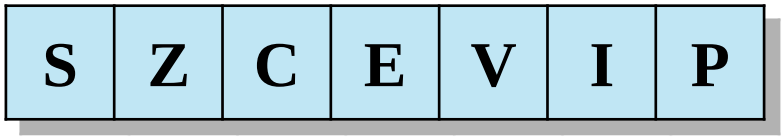
ALU의 구성 요소

산술논리연산장치(Arithmetic and Logical Unit, ALU)

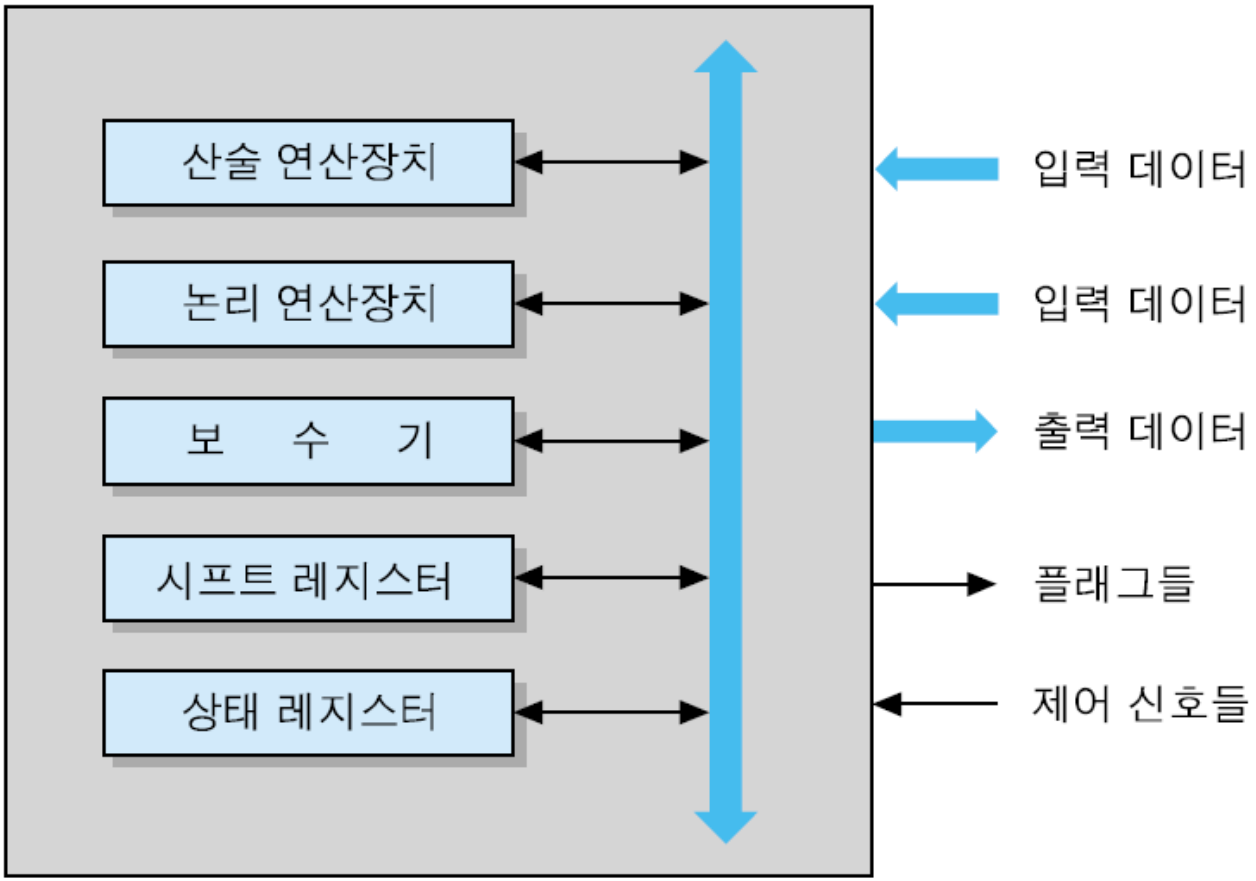
- CPU 내부의 핵심 구성요소로서, 산술 연산과 논리 연산을 수행하는 하드웨어 모듈

ALU의 구성 요소

- **산술 연산장치:** 산술 연산들(+, -, ×, ÷)을 수행
- **논리 연산장치:** 논리 연산들(AND, OR, XOR, NOT 등)을 수행
- **시프트 레지스터(shift register):** 비트들을 좌측 혹은 우측으로 이동시키는 기능을 가진 레지스터
- **보수기(complementer):** 2진 데이터를 2의 보수로 변환(음수화)
- **상태 레지스터(status register):** 연산 결과의 상태를 나타내는 플래그(flag)들을 저장하는 레지스터



ALU의 구성 요소


산술 연산장치
논리 연산장치
보 수 기
시프트 레지스터
상태 레지스터
입력 데이터
입력 데이터
출력 데이터
플래그들
제어 신호들

컴퓨터 산술과 논리 연산

2. 정수의 표현

정수의 표현

- **2진수 체계:** 0, 1, 부호 및 소수점으로 수를 표현
  - **[예]** $-13.625_{10} = -1101.101_2$
- 부호 없는 정수(unsigned integer) 표현
  - **[예]** 10진수 부호 없는 정수를 8-bit 길이의 2진수로 표현

57 = 00111001

0 = 00000000

1= 00000001

128 = 10000000

255= 11111111

- n-비트 2진수를 부호 없는 정수 A로 변환하는 방법

$$A = a_{n-1} \times 2^{n-1} + a_{n-2} \times 2^{n-2} + \cdots + a_1 \times 2^1 + a_0 \times 2^0$$

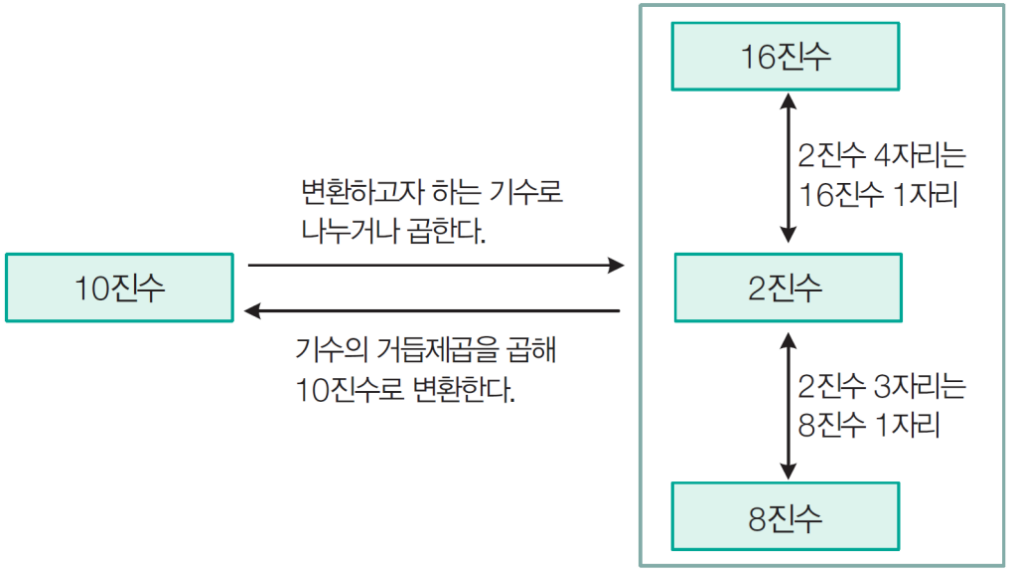
$$110_2 \rightarrow 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6_{10}$$

- n-비트 부호 없는 정수의 표현 가능 범위: $0 \sim 2^n - 1$

진법과 진법 변환

❑2진수, 8진수, 10진수, 16진수의 진법 변환

10진수	2진수	8진수	16진수
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F



소수와 음수의 표현

- 최상위 비트인 a_{n-1}의 좌측에 소수점이 있는 소수(실수에서 정수 부분을 제외한, 소수점 이하의 수 부분)의 10진수 변환 방법

$$A = a_{n-1} \times 2^{-1} + a_{n-2} \times 2^{-2} + \cdots + a_1 \times 2^{-(n-1)} + a_0 \times 2^{-n}$$

  - **[예]** 2^3 2^2 2^1 2^0 2^{-1} 2^{-2} 2^{-3} : 자리수(weight)

    1 1 0 1 . 1 0 1

    $= 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}$

    $= 8 + 4 + 1 + 0.5 + 0.125 = 13.625$

- 음수 표현 방법: 2진수의 맨 좌측(최상위) 비트를 **부호 비트(sign bit)**로 활용 → 부호 비트가 0이면 양수, 1이면 음수
  - 부호화-크기 표현(signed-magnitude representation)
  - 1의 보수 표현(1's complement representation)
  - 2의 보수 표현(2's complement representation)

부호화–크기 표현

❑ 부호화-크기 표현

- 맨 좌측 비트(Most Significant Bit, **MSB**)는 부호 비트, 나머지 $(n-1)$개의 비트들은 수의 크기(magnitude)를 나타내는 표현 방식
- **[예]** + 9 = 0 0001001 + 35 = 0 0100011

  - 9 = 1 0001001 - 35 = 1 0100011

- 부호화-크기로 표현된 2진수$(a_{n-1}a_{n-2}\cdots a_1a_0)$를 10진수로 변환

$$A = (-1)^{a_{n-1}}(a_{n-2}\times 2^{n-2} + a_{n-3}\times 2^{n-3} + \cdots + a_1\times 2^1 + a_0\times 2^0)$$

[예] $0\ 0100011 = (-1)^0(0\times2^6+1\times2^5+0\times2^4+0\times2^3+0\times2^2+1\times2^1+1\times2^0)$

$= (32 + 2 + 1) = 35$

$1\ 0001001 = (-1)^1(0\times2^6+0\times2^5+0\times2^4+1\times2^3+0\times2^2+0\times2^1+1\times2^0)$

$= -(8 + 1) = -9$

부호화–크기 표현

- **장점:** 음수 표현 방법들 중에서 가장 간단함

- **단점:** 덧셈과 뺄셈을 수행하기 위해서는 부호 비트와 크기 부분을 별도로 처리하는 복잡한 과정 필요
  - ① 두수의 부호를 비교
  - ② 부호가 같은 경우에는 크기 부분들을 더하고, 다른 경우에는 크기 부분의 차이를 구함
  - ③ 크기 부분의 절댓값이 더 큰 수의 부호가 부호로 세트

  - **0에 대한 표현이 두 개 존재**

$$0\ \ 0000000\ =\ +0$$

$$1\ \ 0000000\ =\ -0$$

    - 데이터가 '0'인지 검사하는 과정이 복잡함
    - n-비트 단어로 표현할 수 있는 수들이 2^n 개가 아닌, $(2^n - 1)$개로 감소

보수 표현

- **보수 표현(complement representation):** 음수를 2진수로 표현하는 방법

- **1의 보수(1's complement) 표현**
  - 모든 비트들을 반전 (0 → 1, 1 → 0)

- **2의 보수(2's complement) 표현**
  - 모든 비트들을 반전하고, 결과값에 1을 더함

- **[예]**

+ 9 = 0000 1001

- 9 = 1111 0110 (1의 보수)

- 9 = 1111 0111 (2의 보수)

+ 35 = 0010 0011

- 35 = 1101 1100 (1의 보수)

- 35 = 1101 1101 (2의 보수)

보수 표현

- 8-비트 2진수로 표현할 수 있는 10진수의 범위
  - 1의 보수: $-(2^7 - 1) \sim (2^7 - 1)$; $-(2^{n-1} - 1) \sim (2^{n-1} - 1)$
  - **2의 보수:** $-(\mathbf{2^7}) \sim (\mathbf{2^7 - 1})$**;** $-(2^{n-1}) \sim (2^{n-1} - 1)$

10진수	1의 보수	2의 보수
127	01111111	01111111
126	01111110	01111110
⋮	⋮	⋮
2	00000010	00000010
1	00000001	00000001
+0	00000000	00000000
−0	11111111	—
−1	11111110	11111111
−2	11111101	11111110
⋮	⋮	⋮
−126	10000001	10000010
−127	10000000	10000001
−128	—	10000000

2의 보수 → 10진수 변환

- 2의 보수로 표현된 양수($a_{n-1} = 0$)를 10진수로 변환하는 방법
  - $A = a_{n-2} \times 2^{n-2} + a_{n-3} \times 2^{n-3} + \cdots + a_1 \times 2^1 + a_0 \times 2^0$

- 2의 보수로 표현된 음수($a_{n-1} = 1$)를 10진수로 변환하는 방법
  - $A = -2^{n-1} + (a_{n-2} \times 2^{n-2} + a_{n-3} \times 2^{n-3} + \cdots + a_1 \times 2^1 + a_0 \times 2^0)$

  - **[예]** $10101110 = -128 + (1 \times 2^5 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1)$

    $= -128 + (32 + 8 + 4 + 2) = -82$

  - **[다른 방법]** 10101110 → 01010010 으로 먼저 변환한 후, 음수 표시

    $01010010 = -(1 \times 2^6 + 1 \times 2^4 + 1 \times 2^1)$

    $= -(64 + 16 + 2) = -82$

비트 확장

- **비트 확장 (Bit Extension):** 데이터의 길이(비트 수)를 늘리는 방법
  - 데이터를 더 많은 비트의 레지스터에 저장하거나, 더 긴 데이터와 연산을 수행하기 위해 필요

- **부호화-크기 표현의 비트 확장:** 부호 비트를 새로운 맨 좌측 위치로 이동시키고, 나머지 위치들은 0으로 채움
  - **[예제]** 10진수 '21'과 '-21'에 대한 8-비트 길이의 부호화-크기 표현을 16-비트 길이로 확장하라.
  - **[풀이]**
    - +21 = 00010101 (8-비트 부호화-크기 표현)
    - +21 = 0000000000010101 (16-비트 부호화-크기 표현)
    - -21 = 10010101 (8-비트 부호화-크기 표현)
    - -21 = 1000000000010101 (16-비트 부호화-크기 표현)

비트 확장

- **2의 보수 표현의 비트 확장:** 확장되는 상위 비트들은 부호 비트와 같은 값으로 세트
  - **부호-비트 확장(sign-bit extension)**이라 함
  - **[예제]** 10진수 ‘21’과 ‘-21’에 대한 8-비트 길이의 2의 보수 표현을 16-비트 길이로 확장하라
  - **[풀이]**
    - +21 = 00010101 (8-비트 2의 보수)
    - +21 = 0000000000010101 (16-비트 2의 보수)
    - -21 = 11101011 (8-비트 2의 보수)
    - -21 = 1111111111101011 (16-비트 2의 보수)

컴퓨터 산술과 논리 연산

3. 논리 연산

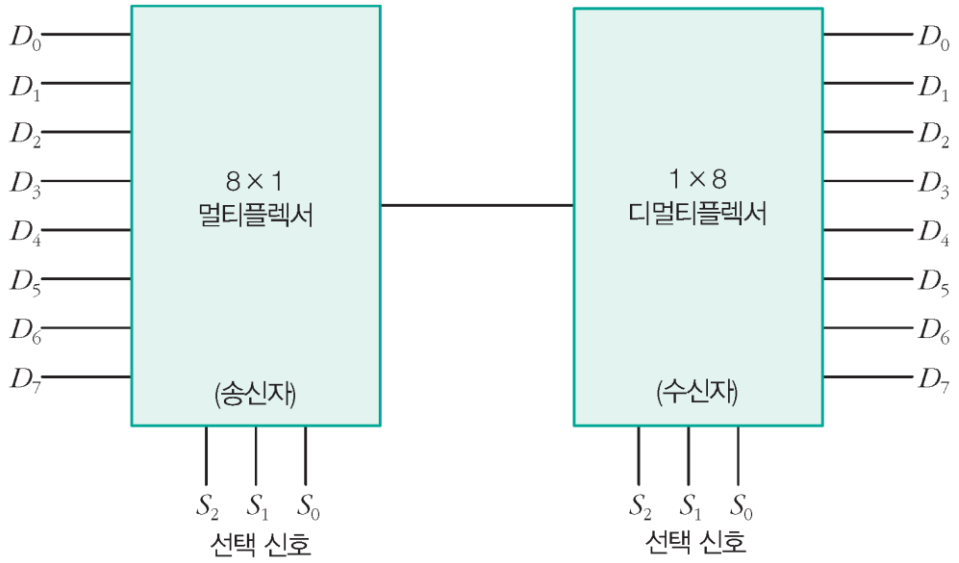
논리 연산

- 기본적인 논리 연산들
  - **[예]** 논리 연산 진리표

A *B*	NOT *A*	NOT *B*	*A* AND *B*	*A* OR *B*	*A* XOR *B*
0 0	1	1	0	0	0
0 1	1	0	0	1	1
1 0	0	1	0	1	1
1 1	0	0	1	1	0

멀티플렉서

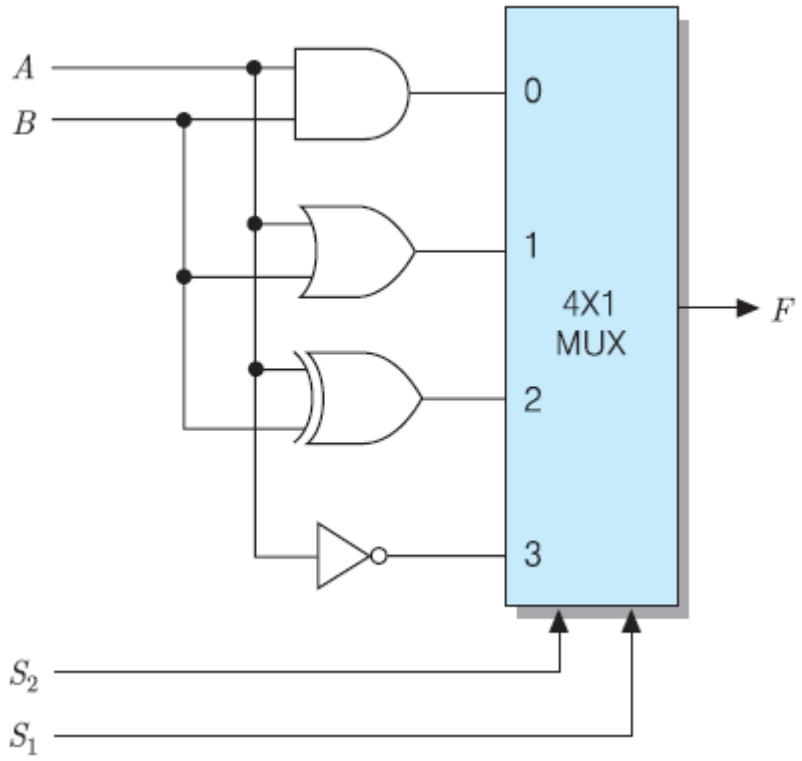
- **멀티플렉서(multiplexer, MUX):** 데이터 선택기, 다중화기
  - 여러 개의 입력 선들 중에서 하나를 선택하여 출력 선에 연결하는 조합논리회로
    - 선택선들의 값에 따라서 입력 선들 중 하나가 선택
- **디멀티플렉서(demultiplexer, DEMUX):** 데이터 분배기, 역다중화기
  - 정보를 한 선으로 받아 2^n 개의 가능한 출력 선들 중 하나를 선택하여, 받은 정보를 전송하는 회로



논리 연산을 위한 하드웨어 모듈

하드웨어의 구성

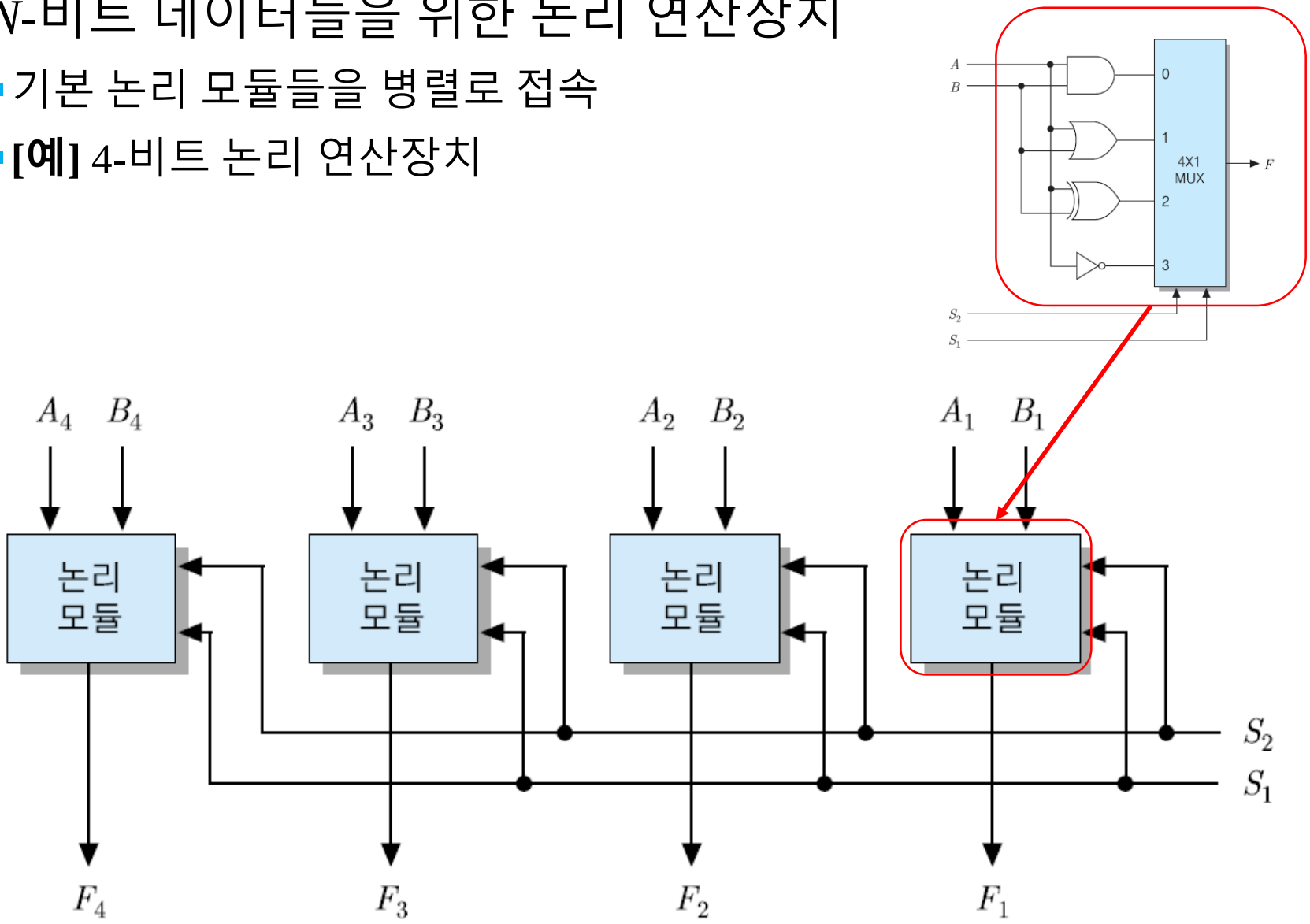
- 입력 비트들은 모든 논리 게이트들을 통과
- 4×1 멀티플렉서: 4(=2^2)개의 입력들 중의 하나를 선택선 S2과 S1에 입력된 값에 따라서 출력으로 보내주는 조합논리회로
  - **[예]** 4×1 멀티플렉서를 이용하여 두 입력 A, B에 대해 AND, OR, XOR, NOT 논리 연산을 수행하는 하드웨어 모듈



S_2	S_1	출력	연산
0	0	$F=A\wedge B$	AND
0	1	$F=A\vee B$	OR
1	0	$F=A\oplus B$	XOR
1	1	$F=\overline{A}$	NOT

N-비트 논리 연산장치

- N-비트 데이터들을 위한 논리 연산장치
  - 기본 논리 모듈들을 병렬로 접속
  - **[예]** 4-비트 논리 연산장치



논리 연산을 활용한 비트 변경

- 논리 연산들은 레지스터에 저장된 데이터 단어의 전체가 아닌 **일부 비트들에 대해서만 수행**될 수 있음
  - 일부 비트들의 값을 변경하는데 유용하게 사용

- **기본적인 논리연산:** AND 연산, OR 연산, XOR 연산, NOT 연산

- **응용된 논리연산:** 선택적-세트 연산, 선택적-보수 연산, 마스크 연산, 삽입 연산, 비교 연산

AND 연산과 OR 연산

- **AND 연산:** 두 데이터 단어들의 대응되는 비트들 간에 AND 연산을 수행

```
A = 10110101
B = 00111011
-----------------------
    00110001   (AND 연산 결과)
```

- **OR 연산:** 두 데이터 단어들의 대응되는 비트들 간에 OR 연산 수행

```
A = 10010101
B = 00111011
-----------------------
    10111111   (OR 연산 결과)
```

XOR 연산과 NOT 연산

- **XOR 연산:** 두 데이터 단어들의 대응되는 비트들 간에 exclusive-OR(XOR) 연산을 수행

```
A = 1 0 0 1 0 1 0 1
B = 0 0 1 1 1 0 1 1
-----------------------
    1 0 1 0 1 1 1 0    (XOR 연산 결과)
```

- **NOT 연산:** 데이터 단어의 **모든 비트들을 반전(invert)**

```
A = 1 0 0 1 0 1 0 1    (연산 전)
-----------------------
    0 1 1 0 1 0 1 0    (NOT 연산 결과)
```

선택적–세트 연산

- **응용된 논리연산:** 원래 데이터를 특정 비트들을 적절히 세트된 다른 데이터와 연산을 수행함으로써 원래 데이터의 **일부 비트들을 변경**

- **선택적-세트(selective-set) 연산:** 데이터의 일부 비트들을 1로 세트해주는 논리적 연산
  - A 레지스터의 비트들을 중 일부를 1로 세트하기 위해서 B 레지스터의 비트들 중에서 대응되는 위치에 있는 비트를 1로 세트하여 연산을 수행
  - **[예]** 레지스터 A에 '10010010'이 저장되어 있을 때, 하위 네 비트를 1로 세트하라.
    - **OR 연산 이용**

A = 1 0 0 1 0 0 1 0 (연산 전)
B = 0 0 0 0 1 1 1 1

A = 1 0 0 1 1 1 1 1 (연산 결과)

선택적–보수 연산

- **선택적-보수(selective-complement) 연산:** 데이터의 일부 비트들을 보수화(반전)시키는 논리적 연산
  - B 레지스터의 비트들 중에서 1로 세트 된 비트들에 대응되는 A 레지스터의 비트들을 보수로 변환
  - **[예]** A 레지스터에 '10010101'이 저장되어 있을 때, 하위 네 비트의 값을 반전시켜라.
    - **XOR 연산 이용**

A = 1 0 0 1 0 1 0 1 (연산 전)
B = 0 0 0 0 1 1 1 1

A = 1 0 0 1 1 0 1 0 (연산 결과)

마스크 연산

- **마스크(mask) 연산:** 데이터 일부 비트들을 0으로 리셋(reset) 시키기 위한 논리적 연산
  - B 레지스터의 비트들 중에서 값이 0인 비트들과 같은 위치에 있는 A 레지스터의 비트들을 0으로 바꾸는 연산(clear)
  - **[예]** A 레지스터에 '11110101'이 저장되어 있을 때, 상위 세 비트를 0으로 세트하라.
    - **AND 연산 이용**

      A = 1 1 0 1 0 1 0 1 (연산 전)

      B = 0 0 0 1 1 1 1 1

      A = 0 0 0 1 0 1 0 1 (연산 결과)

삽입 연산

- **삽입(insert) 연산:** 데이터의 일부 비트들을 새로운 값들로 대체시키기 위한 논리적 연산
  - ① 삽입할 비트 위치들에 대하여 **마스크(AND) 연산** 수행
    - 삽입할 위치의 비트들을 모두 0으로 리셋
  - ② 새롭게 삽입할 비트들과 **OR 연산**을 수행
  - **[예]** A 레지스터에 '10010101'이 저장되어 있을 때, 상위 네 비트를 '1110'으로 대체하라.

A = 1 0 0 1 0 1 0 1
B = 0 0 0 0 1 1 1 1 **마스크 (AND 연산)**

A = 0 0 0 0 0 1 0 1 **첫 단계 결과**
B = 1 1 1 0 0 0 0 0 **삽입 (OR 연산)**

A = 1 1 1 0 0 1 0 1 **최종(삽입) 결과**

비교 연산

- **비교(compare) 연산:** A와 B 레지스터의 내용을 비교
  - **XOR 연산** 활용
  - 만약 대응되는 비트들의 값이 같으면 → A 레지스터의 해당 비트를 '0'으로 세트
  - 만약 서로 다르면 → A 레지스터의 해당 비트를 '1'로 세트
  - 모든 비트들이 같으면(A = 00000000) → Z(zero) 플래그를 1로 세트

A = 1 1 0 1 0 1 0 1

B = 1 0 0 1 0 1 1 0

A = 0 1 0 0 0 0 1 1 (연산 결과)

Q & A

End!