컴퓨터구조

류욱재

skyroom7@knou.ac.kr

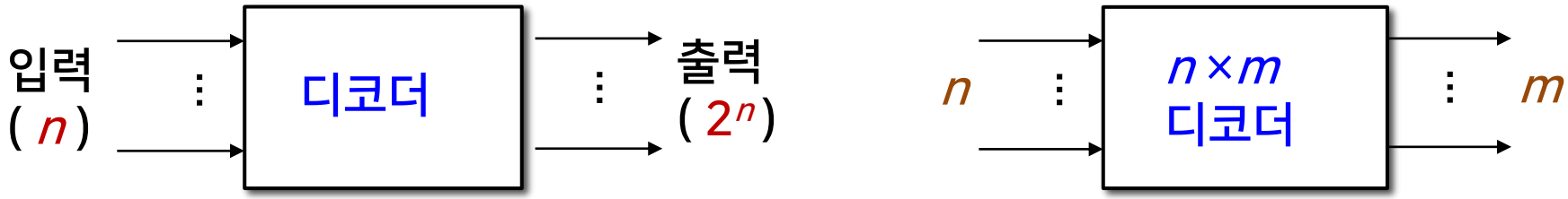
(1) 인코더/디코더

② 디코더

➢ 디코더(decoder) : 부호화된 입력을 받아서
부호화되지 않은 출력을 내보내는 복호화기

(예) 기억장치에서 특정 번지(address)를 선택할 때나
컴퓨터 명령어를 해독하는 데 사용되는 조합논리회로

➢ n 비트의 2진 코드를 최대 2^n 개의 서로 다른 정보로
바꾸어 주는 조합논리회로

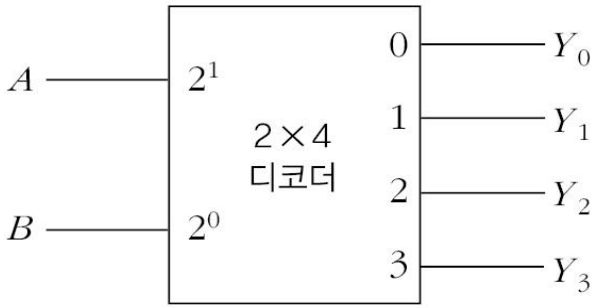


(1) 인코더/디코더

② 디코더

(예) 2×4 디코더

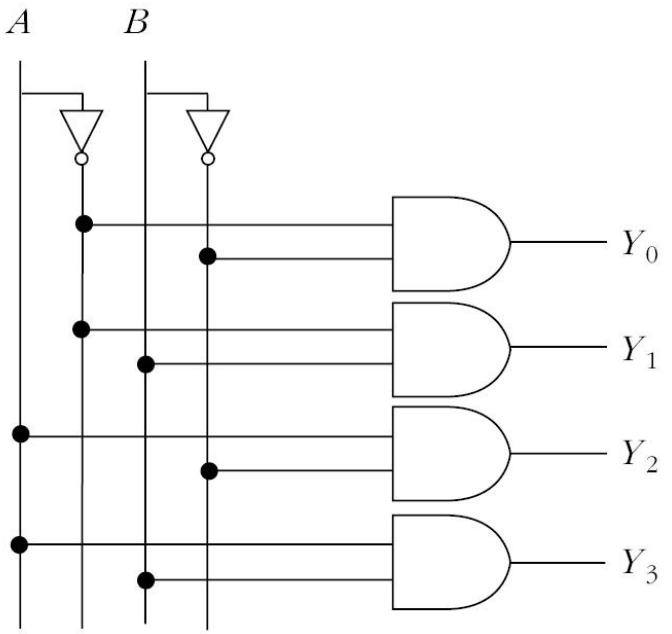
① 블럭도



② 진리표

입력		출력			
A	B	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

③ 내부회로도



(2) 멀티플렉서/디멀티플렉서

① 멀티플렉서

- 멀티플렉서(multiplexer) : 여러 개의 입력선 중에서 하나를 선택하여 단일의 출력을 내보내는 조합논리회로
- 특정 입력선을 선택하기 위해서 선택변수를 사용
- 즉, 2^n 개의 입력선 중에서 특정 입력선을 선택하기 위해서는 n 개의 선택변수가 있어야 한다.
- 이 n 개의 선택변수의 조합에 의해 특정 입력선이 선택
- 데이터 선택기(data selector)라고도 하며, 약어로 MUX 로 표현

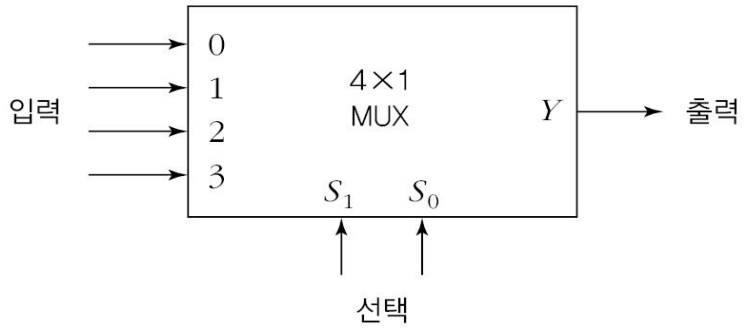
※ 컴퓨터 시스템에서 공통 버스 시스템을 구성하거나 여러 개의 레지스터 중 하나를 선택하는 데 사용

(2) 멀티플렉서/디멀티플렉서

① 멀티플렉서

(예) 4×1 멀티플렉서

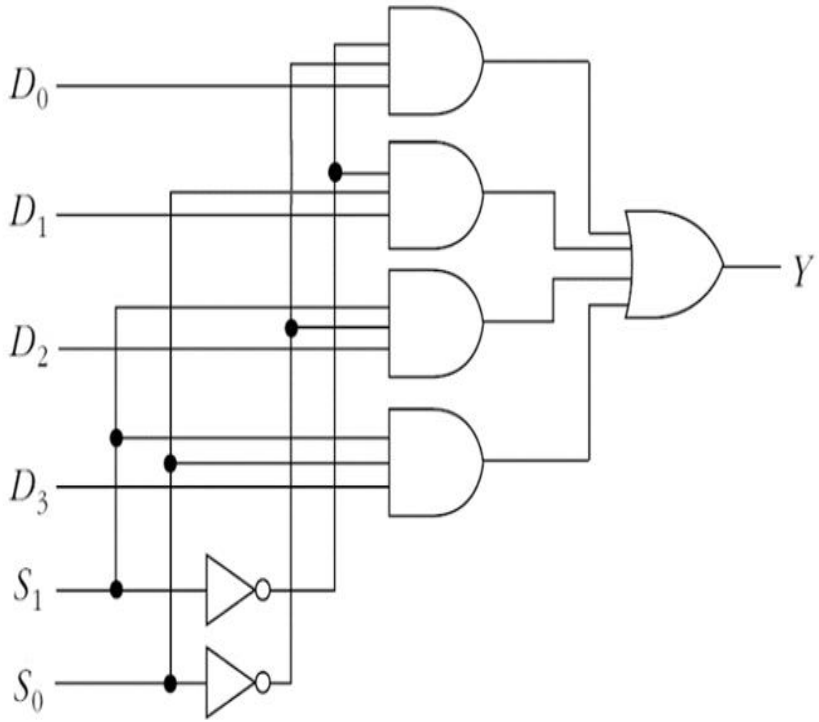
① 블럭도



② 진리표

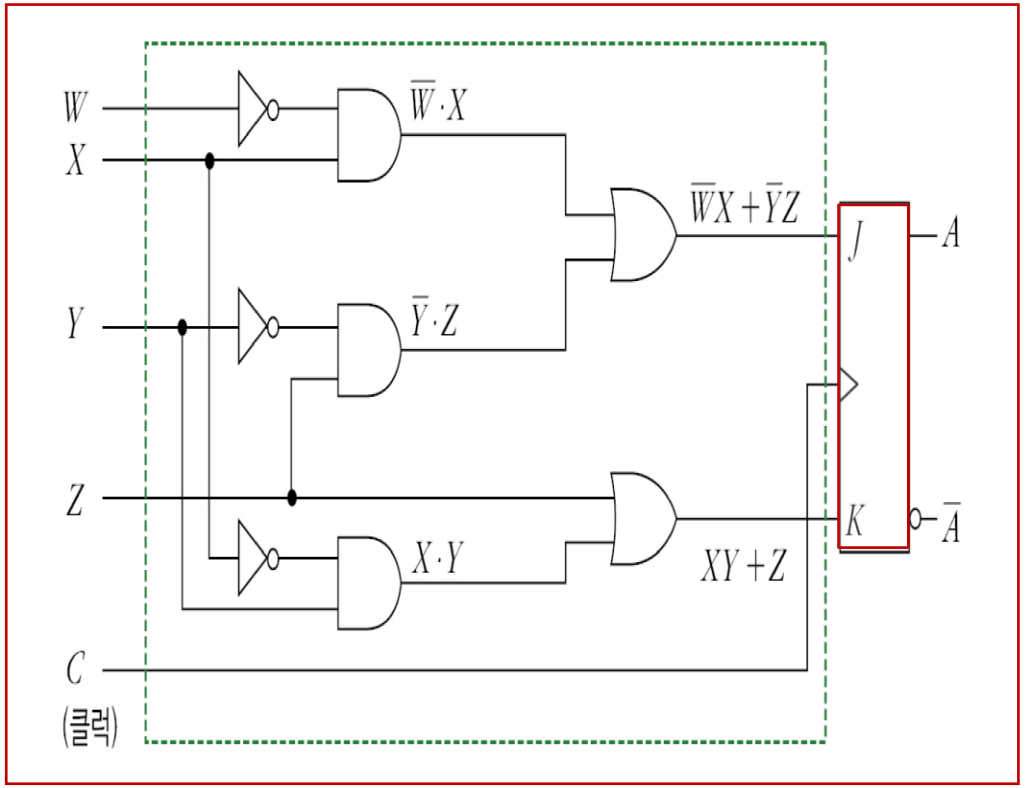
S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

③ 내부회로도(S_1, S_0: 선택변수)



▶ 플립플롭(F/F : Flip Flop)

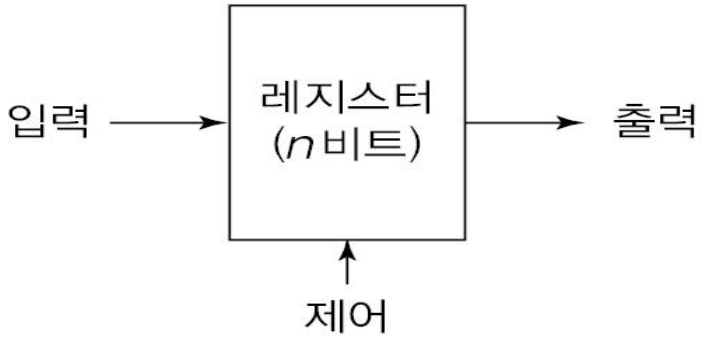
- 입력신호에 의해 상태를 바꾸도록 지시가 있을 때까지 현재의 2진 상태를 유지하는 논리소자
- 한 비트의 2진 정보를 저장할 수 있는 장치
- 클럭 신호에 의해 출력상태를 바꾼다.



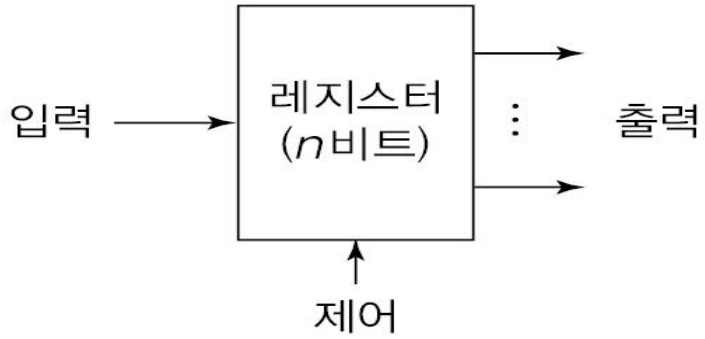
▶ 레지스터

- 데이터를 일시 저장하거나 전송하는 장치
- 여러 개의 플립플롭을 연결하여 구성
- *n* 비트 레지스터는
  - ✓ *n* 개의 플립플롭으로 구성되며,
  - ✓ *n* 비트의 2진 정보를 저장
- 결국 레지스터는 여러 비트를 일시적으로 저장하거나,
  배열된 비트를 좌,우로 자리이동을 시키는데 사용

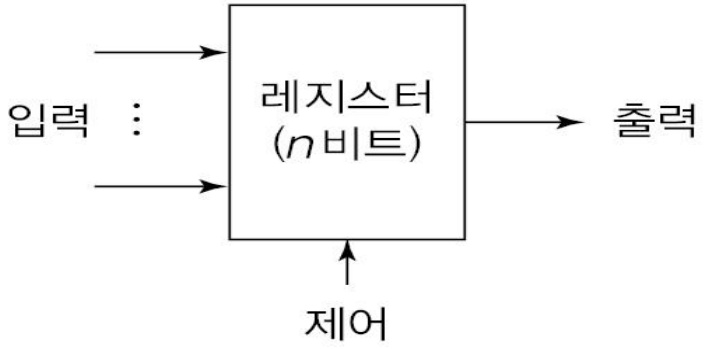
▶ 레지스터의 기본 형태



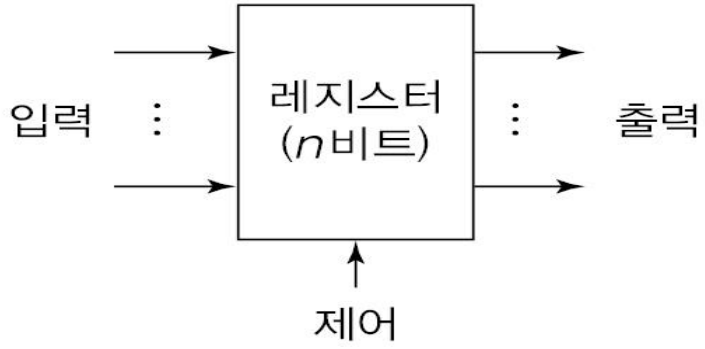
(a) 직렬입력-직렬출력



(b) 직렬입력-병렬출력



(c) 병렬입력-직렬출력



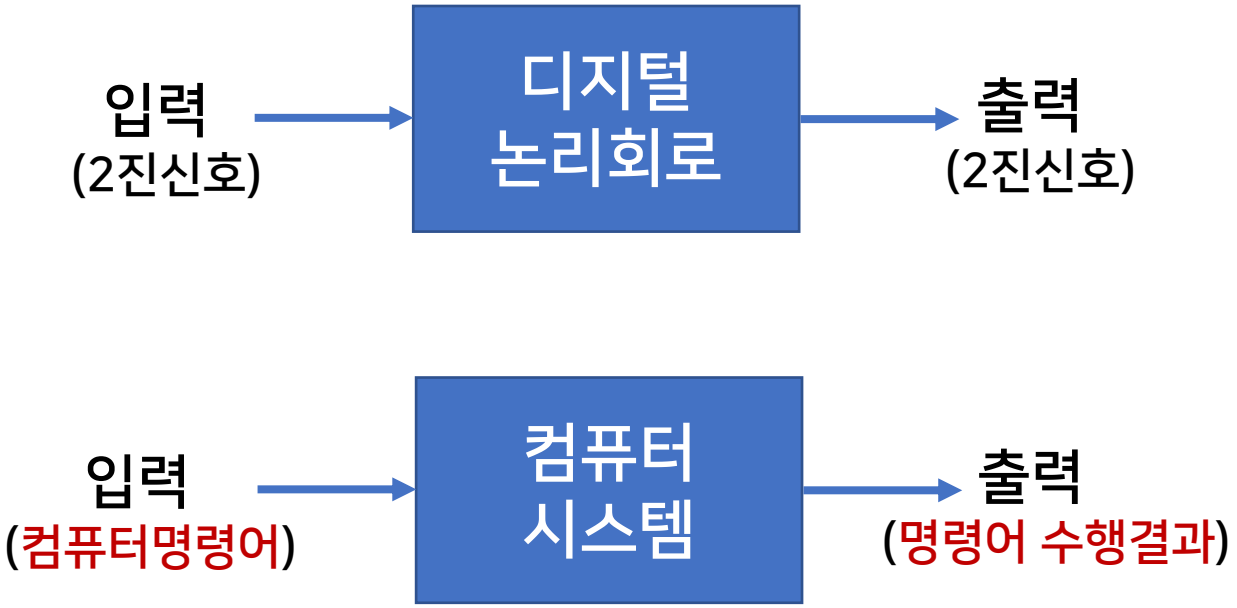
(d) 병렬입력-병렬출력

▶ 카운터의 개요

- 플립플롭을 사용해 만든 순서논리회로로서,
- 입력되는 클록 펄스의 적용에 따라 미리 정해진 순서를 밟아 가는 특수한 형태의 레지스터
- 외부의 입력이나 출력이 없으며,
- 상태변화는 클럭펄스에 의해 수행
- 일반적으로 T F/F 이나 JK F/F 이 사용된다.

▶ 컴퓨터 명령어의 필요성

※ 디지털시스템의 분석



컴퓨터 명령어(instruction)

- 컴퓨터가 수행해야 하는 일을 나타내기 위한 비트들의 집합
- 일정한 형식을 가짐

명령어 집합(instruction set)

- 컴퓨터에서 사용할 수 있는 명령어의 세트(set)
- 모든 컴퓨터는 자신의 명령어 집합을 가지고 있음
- 명령어 집합은 그 컴퓨터의 구조적인 특성을 나타내는 가장 중요한 정보
- 동일 계열의 컴퓨터는 같은 명령어 집합이 사용
- 따라서 명령어 집합을 이용하여 컴퓨터 시스템의 구조를 살펴볼 수 있음

- 명령어는 필드(field)라는 비트그룹으로 이루어지며, 연산코드와 오퍼랜드 필드로 구성
  - 연산코드 필드: 처리해야 할 연산의 종류
  - 오퍼랜드 필드: 처리할 대상 데이터 또는 데이터의 주소

<명령어의 구성 형태>

▶ 컴퓨터 명령어의 수행 기능

› 함수연산 기능

– 덧셈, 시프트, 보수 등의 산술연산과 AND, OR, NOT 등의 논리연산 수행 기능

› 정보전달 기능

– 레지스터들 사이의 정보전달 기능과 중앙처리장치와 주기억장치 사이의 정보전달 기능

› 순서제어 기능

– 조건 분기와 무조건 분기 등을 통해 명령어의 수행 순서를 제어하는 기능

› 입출력 기능

– 주기억장치와 입출력장치 사이의 정보 이동 기능

명령어 형식

- 명령어를 구성하는 필드들의 수와 배치 방식 및 각 필드들의 비트 수를 말한다.
- 명령어는 컴퓨터의 내부구조에 따라 여러 가지 형식이 있음

명령어 형식의 분류

- 오퍼랜드의 기억장소에 따른 명령어 형식
- 오퍼랜드의 수에 따른 명령어 형식

3.3.1 기억장소에 따른 명령어 형식

▶ 명령어 형식의 분류

- 오퍼랜드의 기억장소에 따른 명령어 형식
- 오퍼랜드의 수에 따른 명령어 형식

※ 오퍼랜드가 기억되는 장소에 따라

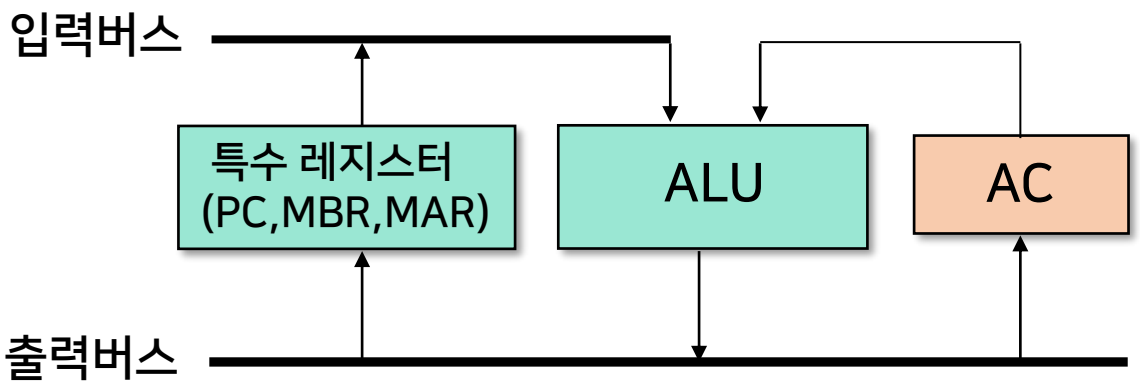
1. 누산기를 이용하는 명령어 형식
2. 다중 레지스터를 이용하는 명령어 형식
3. 스택 구조를 이용하는 명령어 형식

(1) 누산기를 이용하는 명령어 형식

- 누산기를 가진 컴퓨터 구조에서 사용되는 형식

※ 누산기 (AC: accumulator)

– 누산기를 가진 컴퓨터 구조에서 중앙처리장치에 있는 유일한 데이터 레지스터로서 명령어가 수행될 때 오퍼랜드를 기억시키는 레지스터



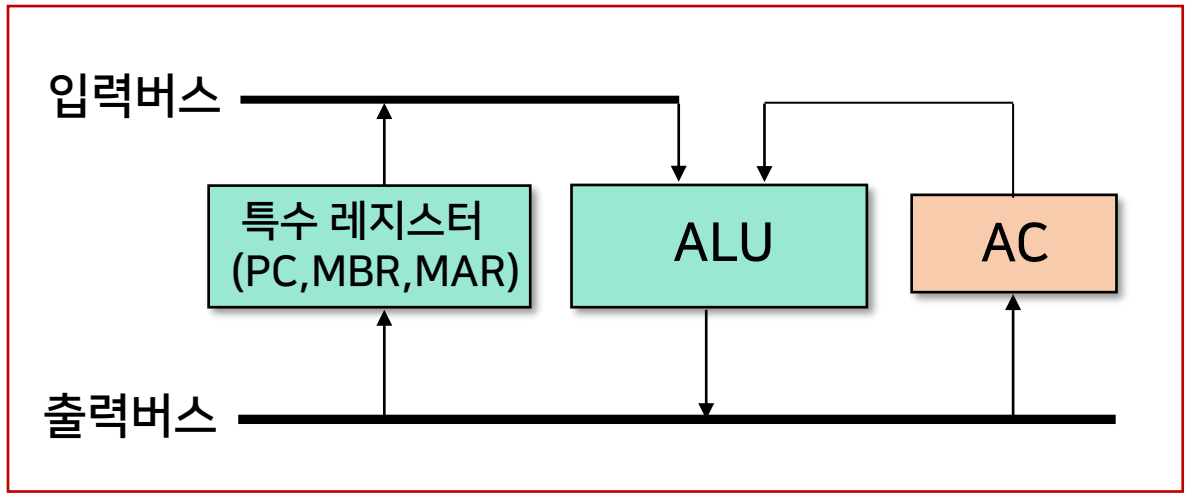
<누산기를 가진 컴퓨터 구조>

(1) 누산기를 이용하는 명령어 형식

<예1>

ADD X ; $AC \leftarrow AC + M[X]$

<의미> '누산기(AC)에 있는 내용과 기억장치 X번지에 있는 내용을 더해서 누산기(AC)로 전송하라'



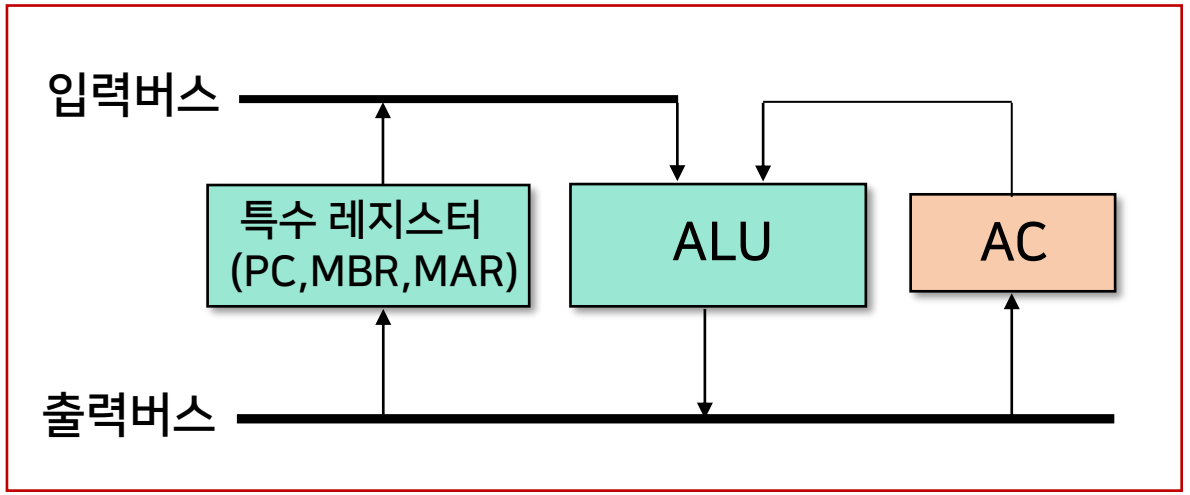
(1) 누산기를 이용하는 명령어 형식

<예2> LOAD X ; *AC ← M[X]*

<의미> '기억장치 X번지에 있는 내용을 누산기로 적재하라'

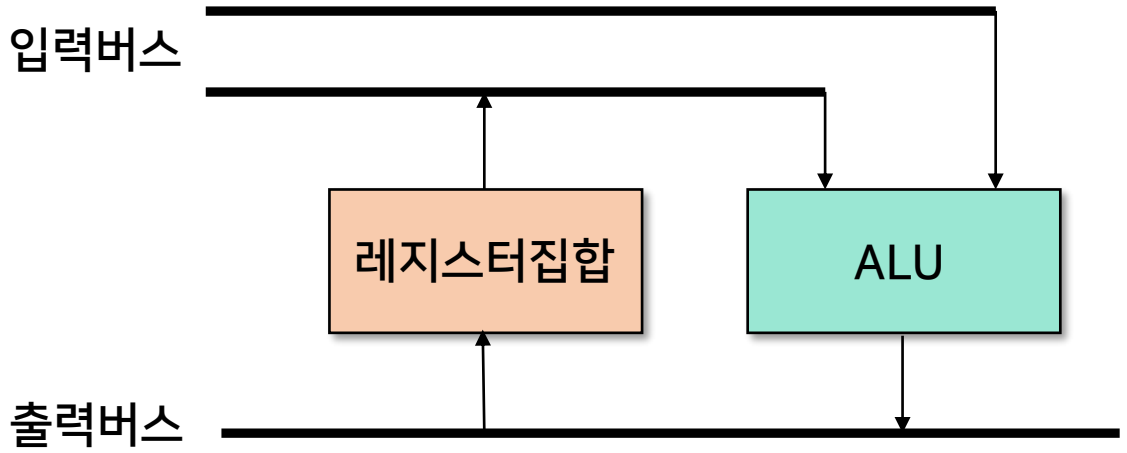
<예3> STORE X ; *M[X] ← AC*

<의미> '누산기의 내용을 기억장치 X번지에 저장하라'



(2) 다중 레지스터를 이용하는 명령어 형식

- 다중 레지스터를 가진 컴퓨터 구조는 중앙처리장치 내에 여러 개의 레지스터를 가지고 있는 컴퓨터이다.



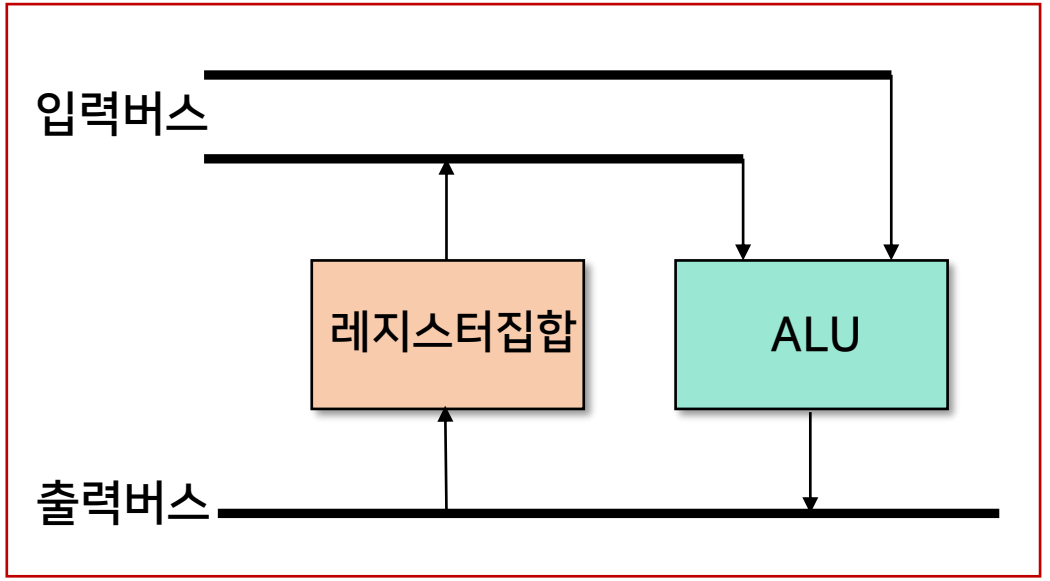
<다중 레지스터를 가진 컴퓨터 구조>

(2) 다중 레지스터를 이용하는 명령어 형식

<예1> 세 개의 레지스터를 사용하는 경우

ADD R1, R2, R3 ; *R3 ← R1 + R2*

<의미> '레지스터 R1의 내용과
레지스터 R2의 내용을 더해서
레지스터 R3로 전송하라'

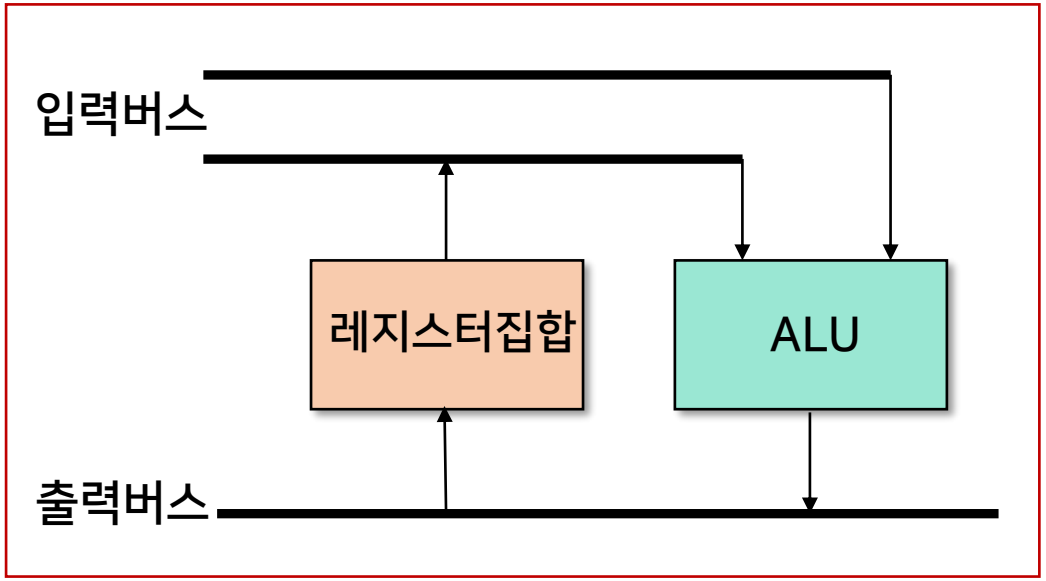


<예2> 두 개의 레지스터를 사용하는 경우

ADD R1, R2 ; *R2 ← R1 + R2*

<의미> '레지스터 R1의 내용과 레지스터 R2의 내용을 더해서
레지스터 R2로 전송하라'

(2) 다중 레지스터를 이용하는 명령어 형식



<예3> 전달기능을 가진 명령어인 경우

MOVE R1, R2 ; *R2 ← R1*

<의미> '레지스터 R1의 내용을 레지스터 R2로 전송하라'

<예4> 주소필드 중 하나가 기억장치 주소필드인 경우

1) LOAD X, R1 ; *R1 ← M[X]*

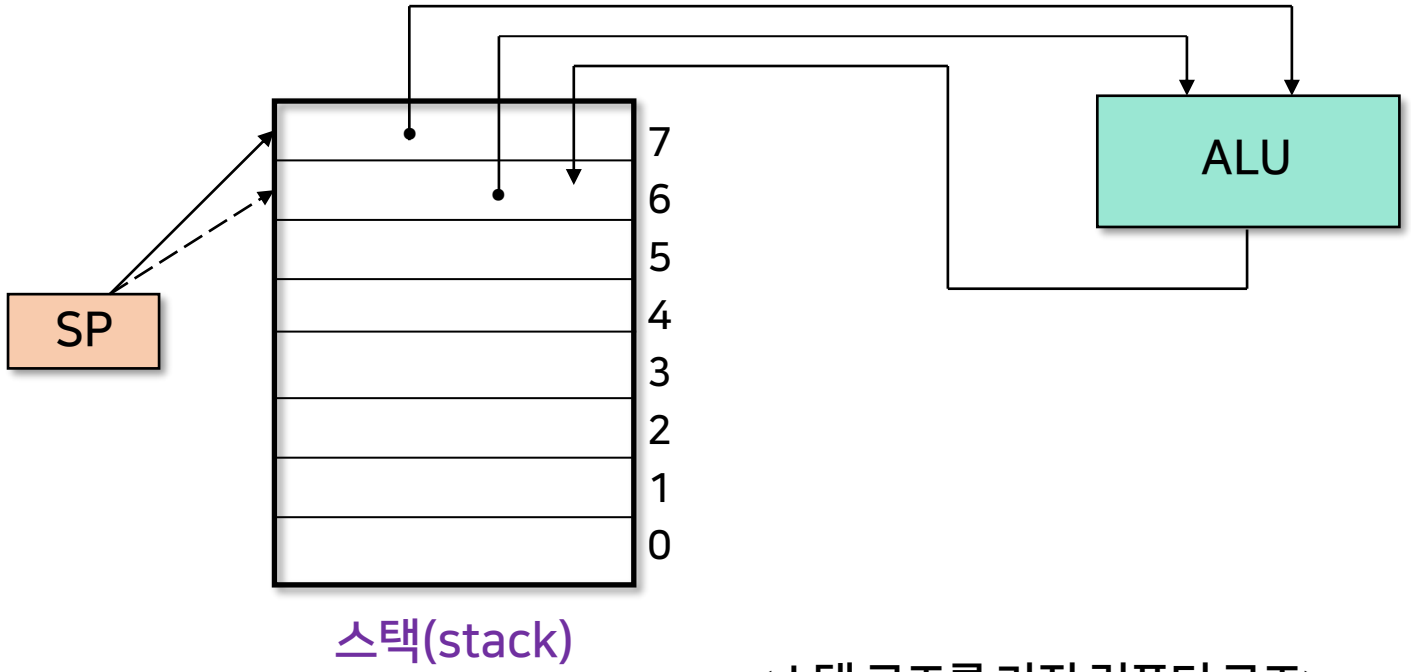
2) STORE R1, X ; *M[X] ← R1*

<의미 1> '기억장치 X번지의 내용을 레지스터 R1에 적재하라'

<의미 2> '레지스터 R1의 내용을 기억장치 X번지에 저장하라'

(3) 스택 구조를 이용하는 명령어 형식

- 스택 구조 컴퓨터는 연산에 필요한 오퍼랜드들을 기억장치 스택에 기억시켜야 하고, 연산의 결과도 스택에 기억시키는 구조이다.



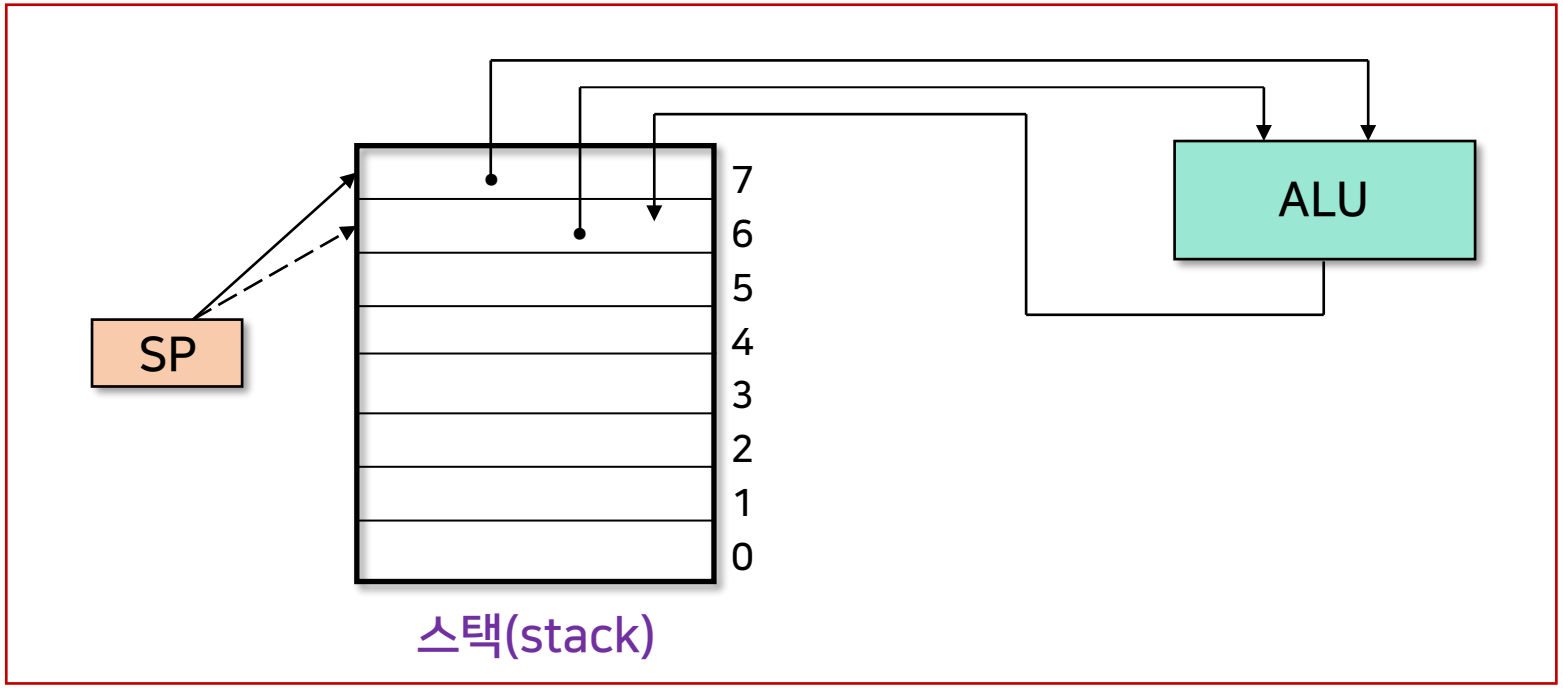
<스택 구조를 가진 컴퓨터 구조>

(3) 스택 구조를 이용하는 명령어 형식

<예1> 주소필드를 사용하지 않는 경우

ADD ; $TOS \leftarrow TOS + TOS_{-1}$

<의미> '기억장치 스택의 맨 위(*TOS*)의 내용과 그 아래(TOS_{-1})의 내용을 더해서 스택의 맨 위(*TOS:* Top Of Stack)로 전송하라'

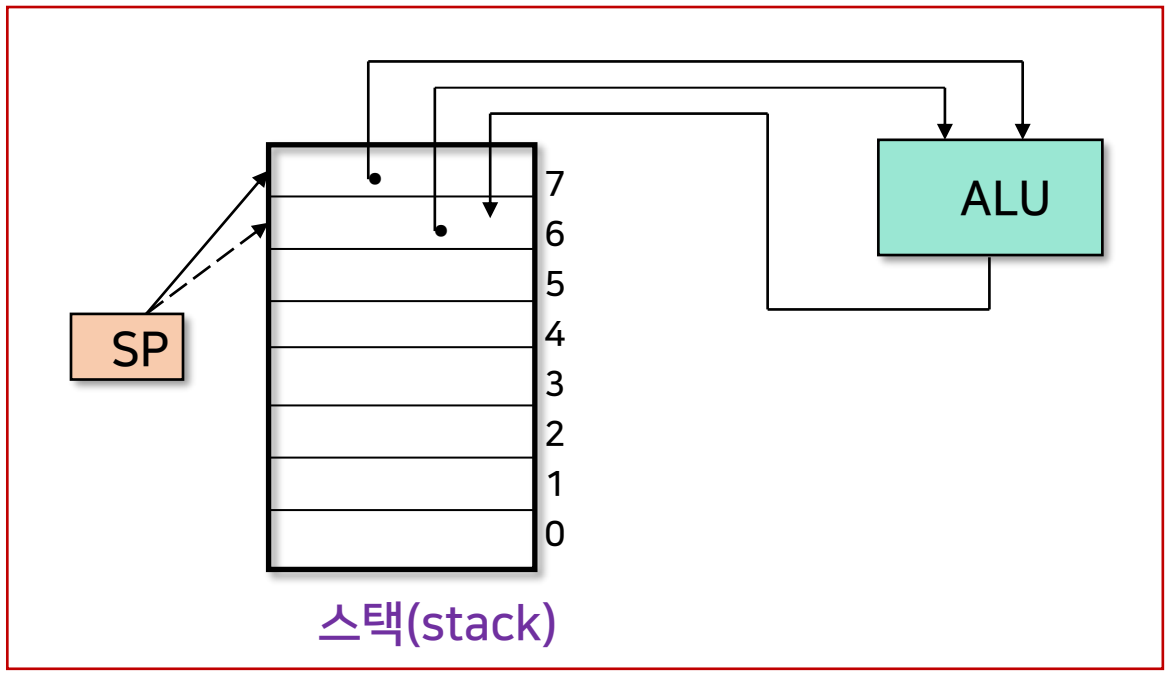


(3) 스택 구조를 이용하는 명령어 형식

<예2> 주소필드를 사용하는 경우

PUSH X ; *TOS ← M[X]* <의미> '기억장치 주소 X의 내용을 기억장치 스택의 맨 위(TOS)로 전송하라'

POP X ; *M[X] ← TOS* <의미> ' 기억장치 스택의 맨 위(*TOS*)의 내용과 그 아래(*TOS-1*)의 내용을 더해서 스택의 맨 위(*TOS:* Top Of Stack)로 전송하라 '



3.3.2 오퍼랜드 수에 따른 명령어 형식

▶ 명령어 형식의 분류

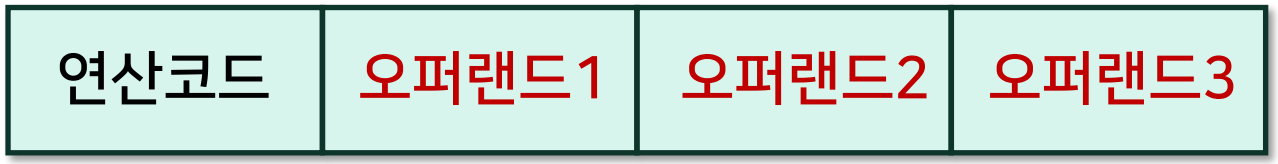
- 오퍼랜드의 기억장소에 따른 명령어 형식
- 오퍼랜드의 수에 따른 명령어 형식

※ 오퍼랜드의 개수에 따라

1. 3-주소 명령어(three-address instruction)
2. 2-주소 명령어(two-address instruction)
3. 1-주소 명령어(one-address instruction)
4. 0-주소 명령어(zero-address instruction)

(1) 3-주소 명령어

- 명령어 오퍼랜드의 개수가 세 개인 명령어 형식



<3-주소 명령어의 형식>

(1) 3-주소 명령어

예

산술식 X = (A+B)×C 에 대해 3-주소 명령어를 이용한 프로그램

```
ADD   A, B, R1    ;   R1 ← M(A) + M(B)
MUL   R1, C, X    ;   M(X) ← R1 × M(C)
```

- ❖ 장점 : 산술식을 프로그램화하는데 있어서 프로그램의 길이가 짧아짐
- ❖ 단점 : 3-주소명령어를 2진 코드화 했을 때 세 개의 오퍼랜드를 나타내기 위한 비트 수가 다른 주소 명령어 형식보다 많이 필요하다.

(2) 2-주소 명령어

- 오퍼랜드의 개수가 두 개인 명령어 형식
- 상업용 컴퓨터에서 가장 많이 사용

〈2-주소 명령어의 형식〉

(2) 2-주소 명령어

예

산술식 X = (A+B)×C 에 대해 3-주소 명령어를 이용한 프로그램

```
LOAD    A, R1   ;   R1 ← M(A)
ADD     B, R1   ;   R1 ← R1 + M(B)
MUL     C, R1   ;   R1 ← R1 × M(C)
STORE   R1, X   ;   M(X) ← R1
```

❖ 장점 : 3-주소 명령어에 비해 명령어의 길이는 짧아짐

❖ 단점 : 같은 내용을 수행하기 위해 수행해야 하는 명령어의 수는 증가됨

(3) 1-주소 명령어

- 오퍼랜드의 개수가 하나인 명령어 형식
- 기억장치로부터 오퍼랜드를 가져오거나 연산결과를 저장하기 위한 임시적인 장소로 누산기 레지스터를 사용한다.

<1-주소 명령어의 형식>

(3) 1-주소 명령어

예

산술식 X = (A+B)×C 에 대해 3-주소 명령어를 이용한 프로그램

```
LOAD    A   ;   AC ← M(A)
ADD     B   ;   AC ← AC + M(B)
STORE   X   ;   M(X) ← AC
LOAD    C   ;   AC ← M(C)
MUL     X   ;   AC ← AC  × M(X)
STORE   X   ;   M(X) ← AC
```

❖ 장점 : 모든 연산은 누산기 레지스터와 기억장치에 저장된 오퍼랜드를 대상으로 수행

❖ 단점 : 프로그램을 수행하기 위해 사용되는 명령어의 수는 더 증가

(4) 0-주소 명령어

- 스택 구조에서 사용되는 형식
- 주소필드를 사용하지 않는다.

<0-주소 명령어의 형식>

(4) 0-주소 명령어

예

산술식 X = (A+B)×C 에 대해 3-주소 명령어를 이용한 프로그램

```
PUSH   A   ;   TOS ← M(A)
PUSH   B   ;   TOS ← M(B)
ADD        ;   TOS ← TOS + TOS-1
PUSH   C   ;   TOS ← M(C)
MUL        ;   TOS ← TOS × TOS-1
POP    X   ;   M(X) ← TOS
```

- ❖ 장점 : 명령어의 길이가 매우 짧아서 기억공간을 적게 차지
- ❖ 단점 : 특수한 경우를 제외하고는 많은 양의 정보가 스택과 기억장치 사이를 이동하게 되어 비효율적

▶ 명령어 주소지정방식(addressing mode)

- 프로그램 수행 시 오퍼랜드를 지정하는 방식
- 명령어의 주소 필드를 변경하거나 해석하는 규칙을 지정하는 형식
- 주소지정방식을 사용하면 명령어의 수를 줄일 수 있는 효과적인 프로그래밍 가능

※ 유효주소

- 주소지정방식의 각 규칙에 의해 정해지는 오퍼랜드의 실제 주소

※ 별도의 주소지정방식 필드를 가진 명령어 형식

- 연산코드 필드
  - 수행할 연산의 종류를 지정
- 주소지정방식 필드(addressing mode field)
  - 연산에 필요한 오퍼랜드의 주소를 알아내는 데 사용
- 주소 혹은 오퍼랜드 필드
  - 기억장치주소 혹은 레지스터를 나타낸다.

※ 주소지정방식의 종류

1. 의미 주소지정방식
2. 즉치 주소지정방식
3. 직접 주소지정방식
4. 간접 주소지정방식
5. 레지스터 주소지정방식
6. 레지스터 간접 주소지정방식
7. 상대 주소지정방식
8. 인덱스된 주소지정방식

▶ 의미주소지정방식(implied mode)

- 명령어 형식에서 주소 필드를 필요로 하지 않는 방식
- 연산코드 필드에 지정된 묵시적 의미의 오퍼랜드를 지정

<예> ADD ; $TOS \leftarrow TOS + TOS_{-1}$

☞ 기억장치 스택에서 ADD와 같은 명령어는 스택의 맨 위 항목과 그 아래 항목을 더하여 스택의 맨 위에 저장하는 명령어로서, 오퍼랜드가 스택의 맨 위에 있다는 것을 묵시적으로 가정함

즉치 주소지정방식(immediate mode)

- 명령어 자체 내에 오퍼랜드를 지정하고 있는 방식
- 오퍼랜드 필드의 내용이 실제 사용될 데이터
- 레지스터나 변수의 초기화에 유용

<예> LDI 100, R1 ; *R1 ← 100*

직접 주소지정방식(direct-addressing mode)

- 명령어의 주소필드에 직접 오퍼랜드의 주소를 저장시키는 방식
- 기억장치에의 접근이 한번에 이루어짐

<예> LDA ADRS ; *AC ← M[ADRS]*

간접 주소지정방식(indirect-addressing mode)

- 명령어의 주소필드에 유효주소가 저장되어있는 기억장치 주소를 기억시키는 방식

<예> LDA [ADRS] ; *AC←M[M[ADRS]]*

3.4.4 레지스터 주소지정과 레지스터 간접주소지정

▶ 레지스터 주소지정방식(register mode)

- 오퍼랜드 필드에 레지스터가 기억되는 방식
- 레지스터에 오퍼랜드가 들어있음(유효주소가 없음)

<예> LDA R1 ; *AC ← R1*

▶ 레지스터 간접 주소지정방식(register-indirect mode)

- 레지스터가 실제 오퍼랜드가 저장된 기억장치의 주소 값을 갖고 있는 방식

<예> LDA (R1) ; *AC ← M[R1]*

▶ 상대 주소지정방식(relative addressing mode)

- 유효주소를 계산하기 위해 처리장치 내에 있는 특정 레지스터의 내용에 명령어 주소필드 값을 더하는 방식
- 특정 레지스터로 프로그램 카운터(PC)가 주로 사용

<예> LDA $ADRS ; *AC←M[ADRS+PC]*

☞ 유효주소 = 명령어 주소부분의 내용 + PC의 내용

▶ 인덱스된 주소지정방식(indexed addressing mode)

- 인덱스 레지스터의 내용을 명령어 주소 부분에 더해서 유효주소를 얻는 방식

  <예> LDA ADRS(R1) ; *AC←M[ADRS+R1]*

  ☞ 유효주소 = 명령어 주소부분의 내용 + 인덱스 레지스터의 내용

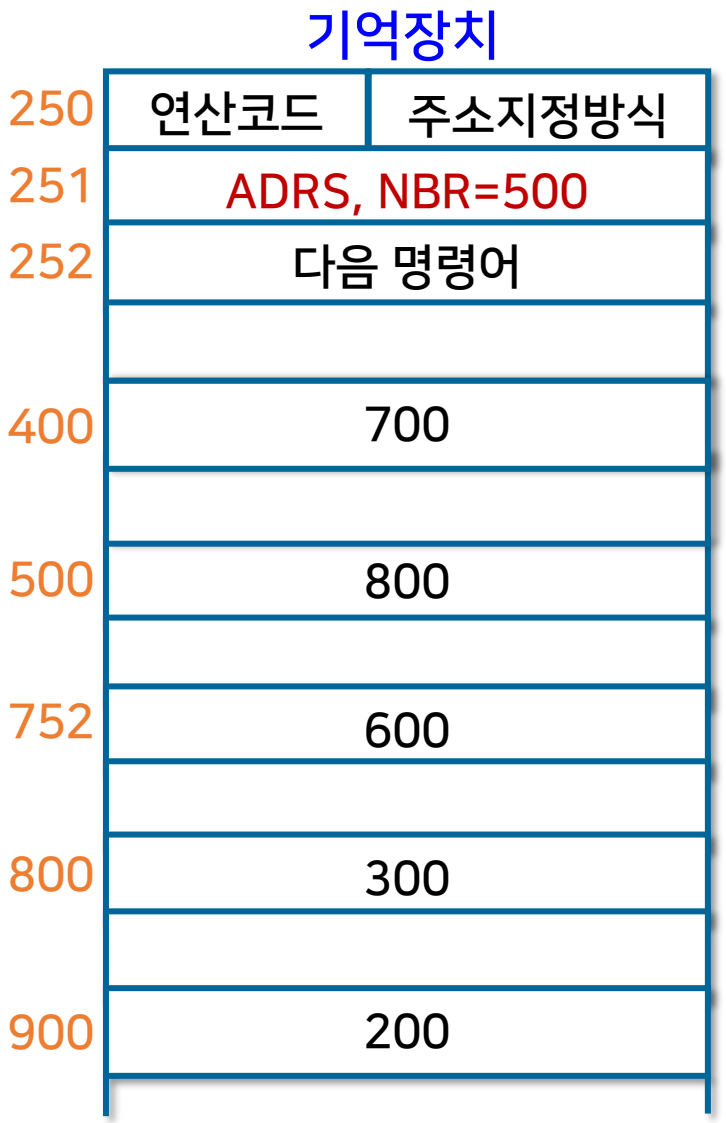
3.4.7 주소지정방식의 요약

PC=250

R1=400

AC

연산코드 : LDA
(AC에 적재하라)



즉치 주소
LDA #NBR ; *AC←NBR*

직접 주소
LDA ADRS ; *AC←M[ADRS]*

간접 주소
LDA [ADRS] ; *AC←M[[ADRS]]*

상대 주소
LDA $ADRS ; *AC←M[ADRS+PC]*

인덱스 주소
LDA ADRS(R1) ; *AC←M[ADRS+R1]*

레지스터 주소
LDA R1 ; *AC←R1*

레지스터 간접주소
LDA (R1) ; *AC←M[R1]*

3.4.7 주소지정방식의 요약

방 식	기호표기	전 송 문	유효 주소	AC 내용
즉치주소	LDA #NBR	*AC←NBR*	251	500
직접주소	LDA ADRS	*AC←M[ADRS]*	500	800
간접주소	LDA [ADRS]	*AC←M[M[ADRS]]*	800	300
상대주소	LDA $ADRS	*AC←M[ADRS+PC]*	752	600
인덱스주소	LDA ADRS(R1)	*AC←M[ADRS+R1]*	900	200
레지스터주소	LDA R1	*AC←R1*	-	400
레지스터간접	LDA (R1)	*AC←M[R1]*	400	700

- 데이터 전송 명령어
- 데이터 처리 명령어
- 프로그램 제어 명령어

▶ 데이터 전송 명령어

- 한 장소에서 다른 장소로 단지 데이터를 전송하는 명령어
- 레지스터와 레지스터 사이, 레지스터와 기억장치 사이,
  또는 기억장치와 기억장치 사이에 데이터를 이동하는 기능
- 입출력 명령어가 포함

▶ 데이터 전송 명령어

전송명령어	니모닉	기 능
Load	*LD*	기억장치로부터 레지스터로의 전송
Store	*ST*	레지스터로부터 기억장치로의 전송
Move	*MOVE*	레지스터로부터 다른 레지스터로의 전송
Exchange	*XCH*	두 레지스터 간 또는 레지스터와 기억장치 간의 데이터 교환
Push	*PUSH*	기억장치의 스택과 레지스터 간의 데이터 전송
Pop	*POP*	
Input	*IN*	레지스터와 입출력장치 간의 데이터 전송
Output	*OUT*	

▶ 데이터 처리 명령어

- 데이터에 대한 연산을 실행하고 컴퓨터에 계산능력을 제공

① 산술 명령어

② 논리와 비트 처리 명령어

③ 시프트 명령어

(1) 산술 명령어

- 사칙연산에 대한 명령어

산술 명령어	니모닉	기 능
Increment	*INC*	1 증가
Decrement	*DEC*	1 감소
Add	*ADD*	덧셈
Subtract	*SUB*	뺄셈
Multiply	*MUL*	곱셈
Divide	*DIV*	나눗셈
Add with carry	*ADDC*	캐리를 포함한 덧셈
Subtract with borrow	*SUBB*	빌림을 포함한 뺄셈
Negate	*NEG*	2의 보수

(2) 논리와 비트 처리 명령어

- 레지스터나 기억장치에 저장된 단어에 대한 2진 연산
- 주로 2진 부호화 정보를 표현하는 비트 그룹이나 개별 비트를 처리하는데 사용
- 비트 값을 0으로 만들거나, 기억장치 레지스터에 저장된 오퍼랜드에 새로운 비트 값을 삽입하는 것 등이 가능

(2) 논리와 비트 처리 명령어

논리 명령어	니모닉	기 능
Clear	*CLR*	모든 비트를 0으로 리셋
Set	*SET*	모든 비트를 1로 셋
Complement	*COM*	모든 비트를 반전
AND	*AND*	비트별 AND 연산
OR	*OR*	비트별 OR연산
Exclusive-OR	*XOR*	비트별 XOR 연산
Clear carry	*CLRC*	캐리 비트의 리셋
Set carry	*SETC*	캐리 비트의 셋
Complement carry	*COMC*	(반전)보수

(3) 시프트 명령어

- 오퍼랜드의 비트를 왼쪽이나 오른쪽으로 이동시키는 명령어
- 논리적 시프트와 산술적 시프트, 회전형 시프트 연산 등이 있음

시프트 명령어	니모닉	기 능
Logical shift right	*SHR*	오른쪽 시프트(왼쪽의 남는 비트는 0으로 채움)
Logical shift left	*SHL*	왼쪽 시프트(오른쪽의 남는 비트는 0으로 채움)
Arithmetic shift right	*SHRA*	부호비트는 고정(왼쪽의 남는 비트는 부호비트로 채움
Arithmetic shift left	*SHLA*	부호비트는 고정(오른쪽의 남는 비트는 0으로 채움)
Rotate right	*ROR*	오른쪽으로 순환(버려지는 비트는 다시 왼쪽비트로)
Rotate left	*ROL*	왼쪽으로 순환(버려지는 비트는 다시 오른쪽비트로)
Rotate right with carry	*RORC*	캐리를 포함한 오른쪽 순환
Rotate left with carry	*ROLC*	캐리를 포함한 왼쪽 순환

▶ 프로그램 제어 명령어

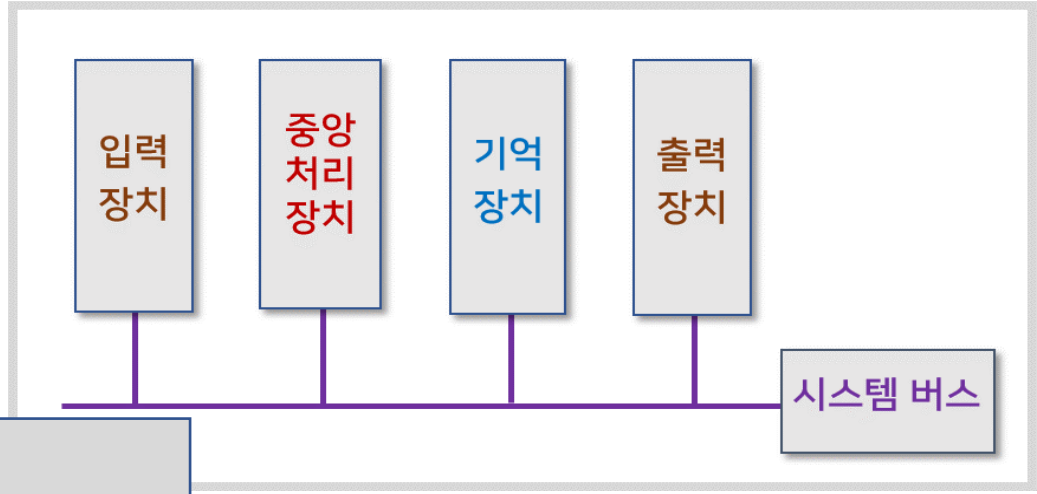
- 프로그램 수행의 흐름을 제어
- 다른 프로그램의 세그먼트 (segment)로 분기

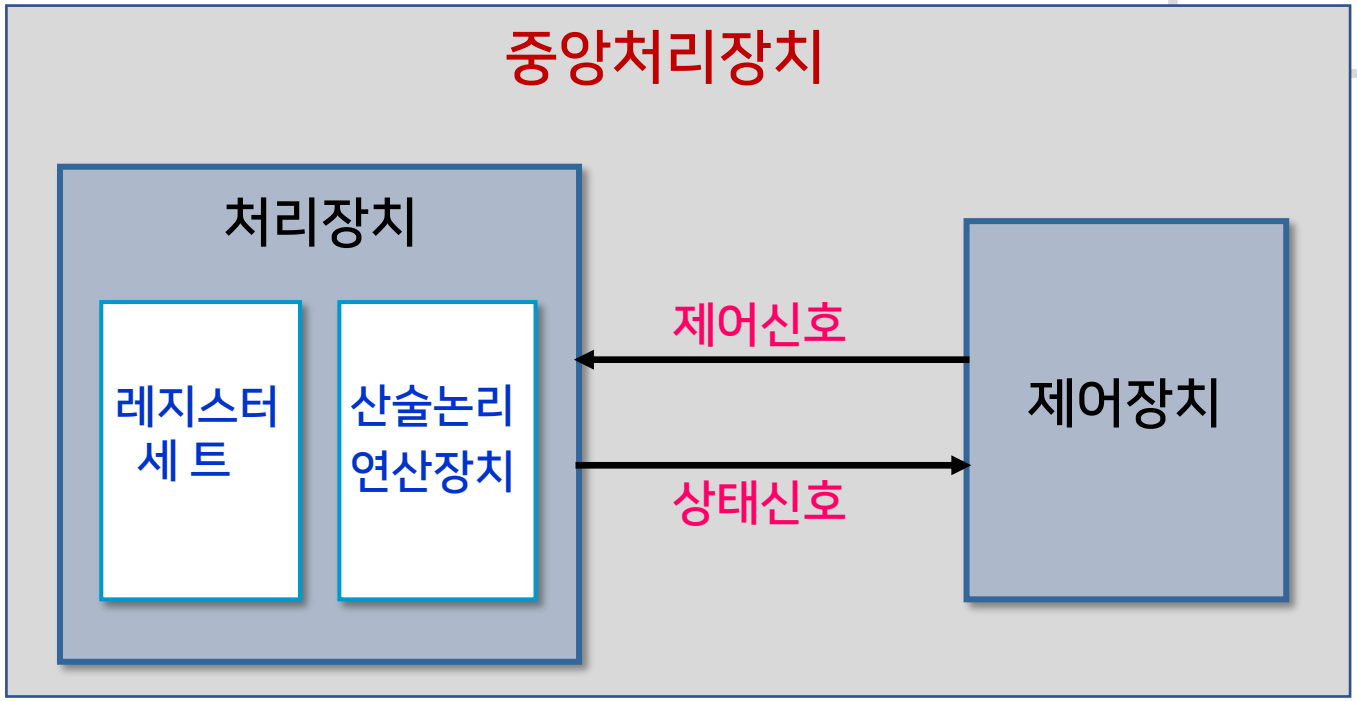
제어 명령어	니모닉	기 능
Branch	*BR*	조건 혹은 무조건적으로 유효주소로 분기
Jump	*JMP*	
Skip next instruction	*SKP*	조건이 만족되면 다음 명령어를 수행하지 않고 넘어감
Call procedure	*CALL*	서브루틴 호출
Return from procedure	*RET*	서브루틴 실행 후 복귀
Compare(by subtraction)	*CMP*	두 오퍼랜드의 뺄셈을 통해 상태 레지스터의 값을 변환
Test (by ANDing)	*TEST*	논리 AND 연산만 구현

▶ 중앙처리장치(CPU: Central Processing Unit)

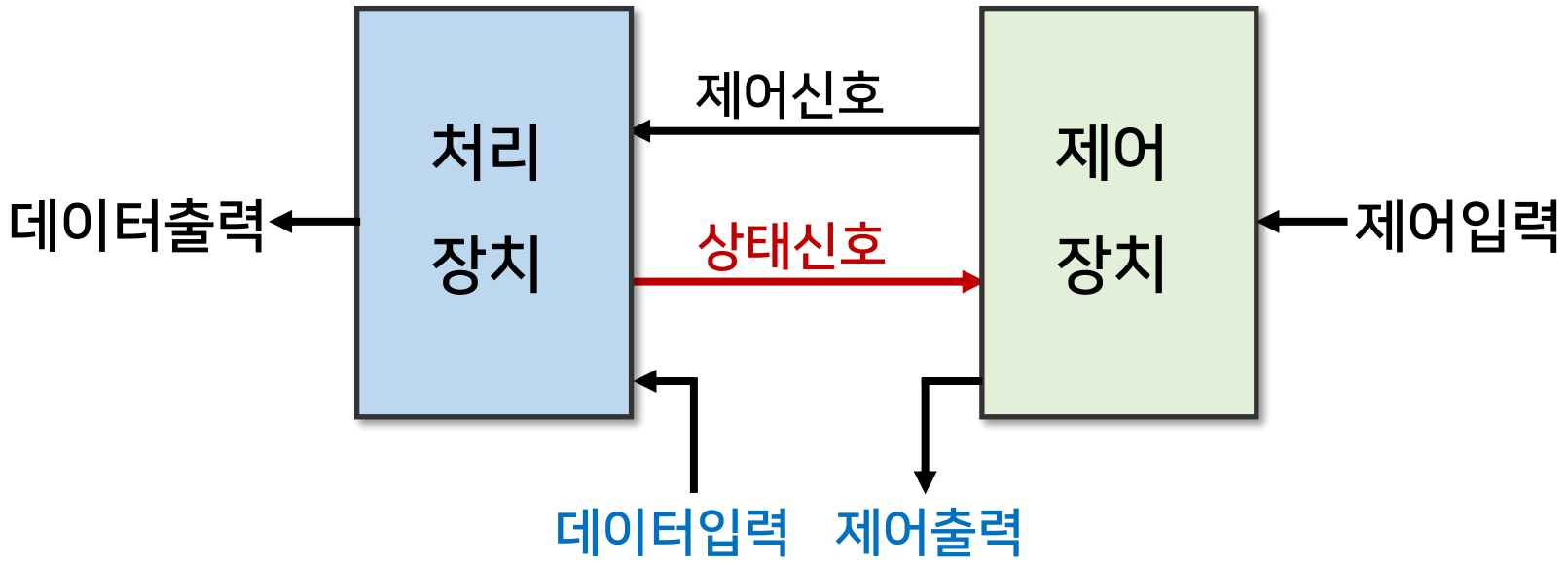
처리 장치와 제어장치가 결합된 형태

- 처리장치 : 데이터를 처리하는 연산을 실행
- 제어장치 : 연산의 실행순서를 결정





※ 처리장치와 제어장치의 관계



▶ 처리장치의 구성

› 산술논리연산장치와 레지스터들로 구성

- 산술논리연산장치(ALU: Arithmetic and Logic Unit)

  : 산술, 논리, 비트연산 등의 연산을 수행

- 레지스터(Register)

  : 연산에 사용되는 데이터나 연산의 결과를 저장

❖ 산술논리연산장치(ALU)는 독립적으로 데이터를 처리하지 못하며, 반드시 레지스터들과 조합하여 데이터를 처리

▶ 마이크로 연산

› 레지스터에 저장되어 있는 데이터에 대해 이루어지는 기본적인 연산

- 한 레지스터의 내용을 다른 레지스터로 옮기는 것
- 두 레지스터의 내용을 합하는 것
- 레지스터의 내용을 1만큼 증가시키는 것 등

❖ 처리장치의 동작원리를 이해하기 위해서는 마이크로 연산을 이해해야 함

▶ 마이크로 연산의 종류

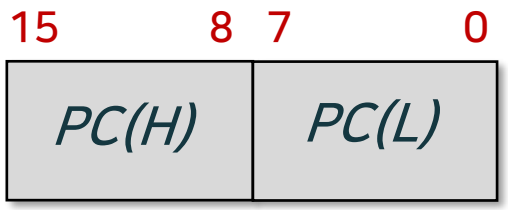
- 레지스터 전송 마이크로 연산(register transfer micro-operation)
- 산술 마이크로 연산(arithmetic micro-operation)
- 논리 마이크로 연산(logic micro-operation)
- 시프트 마이크로 연산(shift micro-operation)

※ 레지스터의 표현

<레지스터 R>

<8비트 레지스터의 개별 비트>

<16비트 레지스터의 순서 표시>



<16비트 레지스터의 분할>

▶ 레지스터 전송 마이크로 연산

› 한 레지스터에서 다른 레지스터로 2진 데이터를 전송하는 연산

: 레지스터 사이의 데이터 전송은 연산자 '←' 로 표시

<예> $R2 \leftarrow R1$

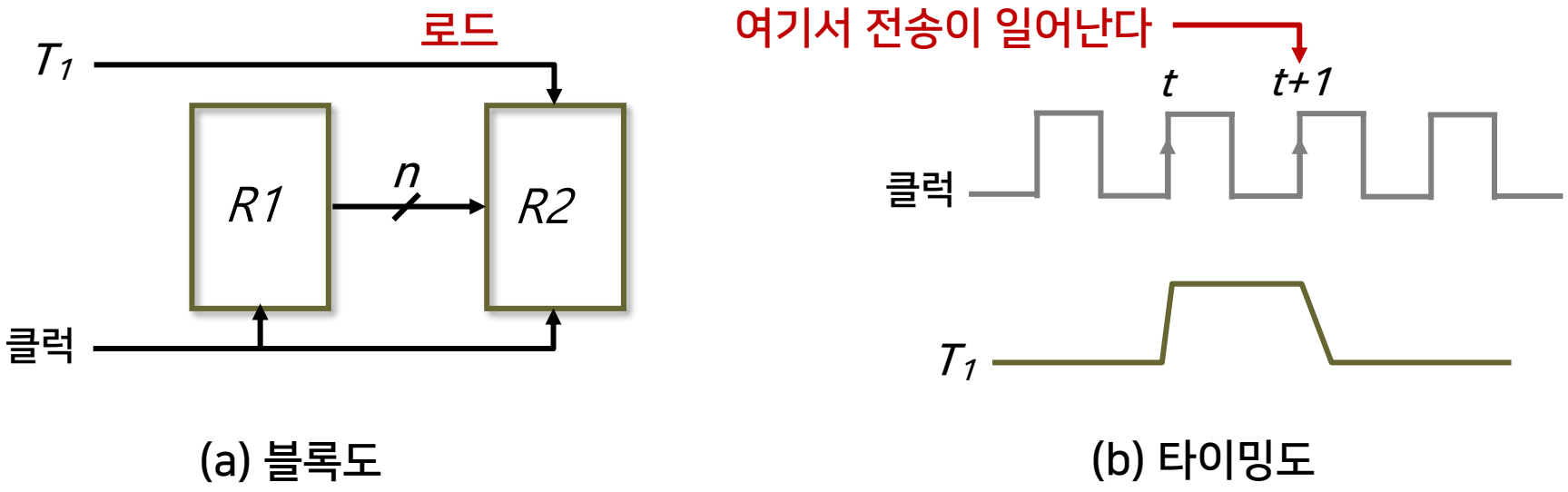
<의미 > 레지스터 *R1* 의 내용이 레지스터 *R2* 로 전송

☞ 여기서 *R1* : 출발 레지스터(source register)

R2 : 도착 레지스터(destination register)

※ 하드웨어적인 측면에서의 레지스터 전송

<예> 레지스터 *R1* 에서 *R2* 로의 전송



(a) 블록도 (b) 타이밍도

<T_1 =1인 상태에서 *R1* 에서 *R2* 로의 데이터 전송>

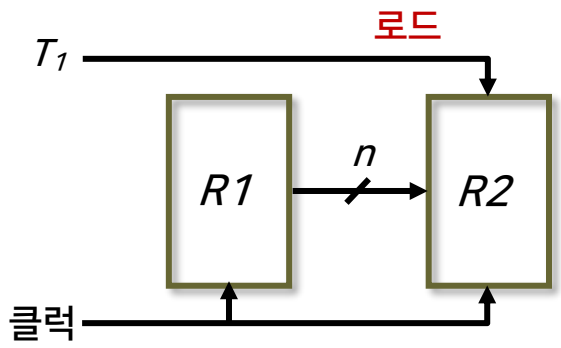
※ 레지스터 전송문

➢ 앞의 그림을 조건문으로 표현하면

$if\ (T_1 = 1)\ then\ (R2 \leftarrow R1)$

➢ 레지스터 전송문으로 표현하면

$T_1 : R2 \leftarrow R1$



(a) 블록도

※ 레지스터 전송문장에서 사용되는 기본적인 기호

기 호	의 미	사 용 예
영문자(숫자와 함께)	레지스터를 표시	*AR, R2, DR, IR*
괄호	레지스터의 일부분	*R2(1), R2(7:0), AR(1)*
화살표	자료의 이동 표시	*R1 ← R2*
쉼표	동시에 실행되는 두 개 이상의 마이크로 연산을 구분	*R1 ← R2, R2 ← R1*
대괄호	메모리에서의 어드레스	*DR ← M[AR]*

▶ 산술 마이크로 연산

› 레지스터 내의 데이터에 대해서 실행되는 산술연산

- 기본적인 산술연산으로는 덧셈, 뺄셈, 1 증가, 1 감소, 그리고 보수연산이 있다.

기 호 표 시	의 미
$R0 \leftarrow R1 + R2$	*R1* 과 *R2* 의 합을 *R0* 에 저장
$R2 \leftarrow \overline{R2}$	*R2* 의 보수(1의 보수)를 *R2* 에 저장
$R2 \leftarrow \overline{R2} + 1$	*R2* 에 2의 보수를 계산 후 저장
$R0 \leftarrow R1 + \overline{R2} + 1$	*R1* 에 *R2* 의 2의 보수를 더한 후 *R0* 에 저장
$R1 \leftarrow R1 + 1$	*R1* 에 1 더함 (상승 카운트)
$R1 \leftarrow R1 - 1$	*R1* 에 1 뺌 (하강 카운트)

▶ 논리 마이크로 연산

› 레지스터 내의 데이터에 대한 비트를 조작하는 연산

- 기본적인 논리연산으로는 AND, OR, XOR, NOT 연산이 있다.

기 호 표 시	의 미
$R0 \leftarrow \overline{R1}$	비트별 논리적 NOT(1의 보수)
$R0 \leftarrow R1 \wedge R2$	비트별 논리적 AND(비트 클리어)
$R0 \leftarrow R1 \vee R2$	비트별 논리적 OR(비트 세트)
$R0 \leftarrow R1 \oplus R2$	비트별 논리적 XOR(비트별 보수)

▶ 시프트 마이크로 연산

› 레지스터 내의 데이터를 시프트(shift) 시키는 연산

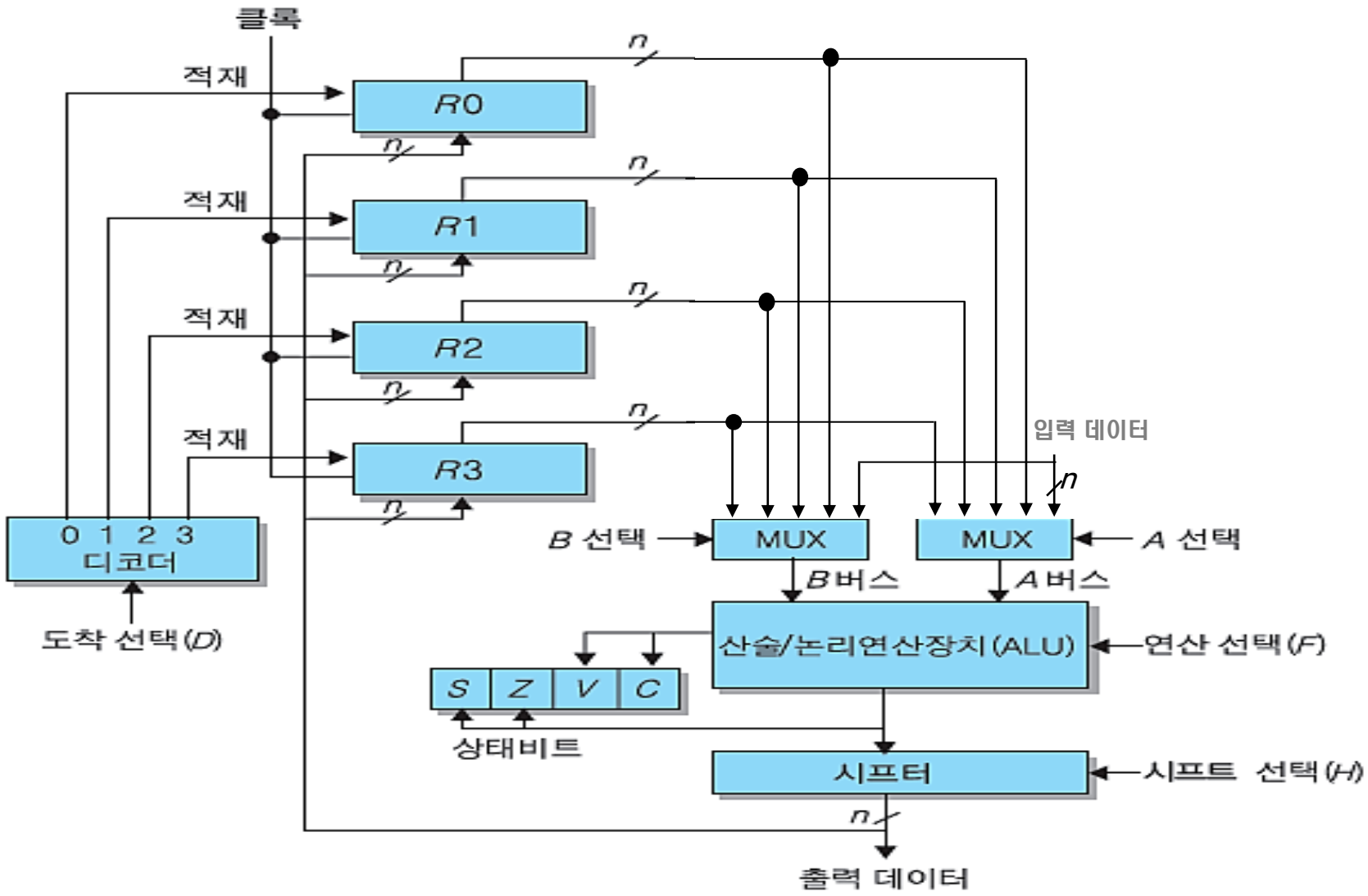
- 데이터의 측면이동에 사용

유 형	기 호 표 시	8비트 데이터의 경우	
		출발지 *R2*	시프트 후: 목적지 *R1*
왼쪽 시프트	$R1 \leftarrow sl\ R2$	1 0 0 1 1 1 1 0	0 0 1 1 1 1 0 0
오른쪽 시프트	$R1 \leftarrow sr\ R2$	1 1 1 0 0 1 0 1	0 1 1 1 0 0 1 0

- 시프트 연산을 수행하더라도 *R2* 의 값은 변하지 않는다.
- *sr* 이나 *sl* 에 대해서 입력 비트는 0으로 가정한다.
- 출력비트의 값은 버려진다.

- 여러 개의 레지스터(레지스터 세트)
- 산술논리연산장치(ALU)
- 내부 버스(internal bus)

※ 간단한 처리장치의 내부 구성도

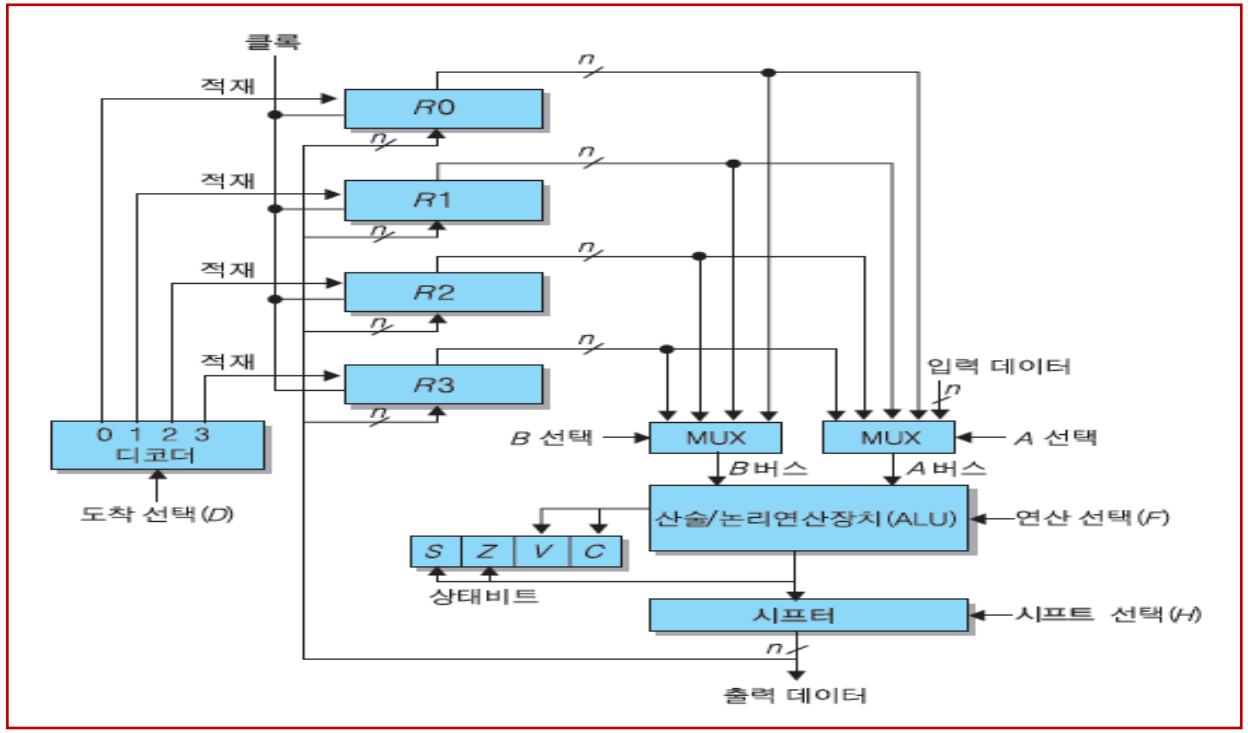


※ 처리장치의 동작

› 마이크로 연산의 수행과정을 통해 처리장치가 동작

※ 마이크로 연산의 수행과정

1) 지정된 출발 레지스터의 내용이 ALU의 입력으로 전달
2) ALU에서 그 연산을 실행
3) 그 결과가 도착 레지스터에 전송



※ 처리장치에서 마이크로 연산의 수행과정

- 처리장치의 구성요소들의 선택신호에 의해 제어됨

<마이크로 연산의 예>

$R0 \leftarrow R1 + R2$

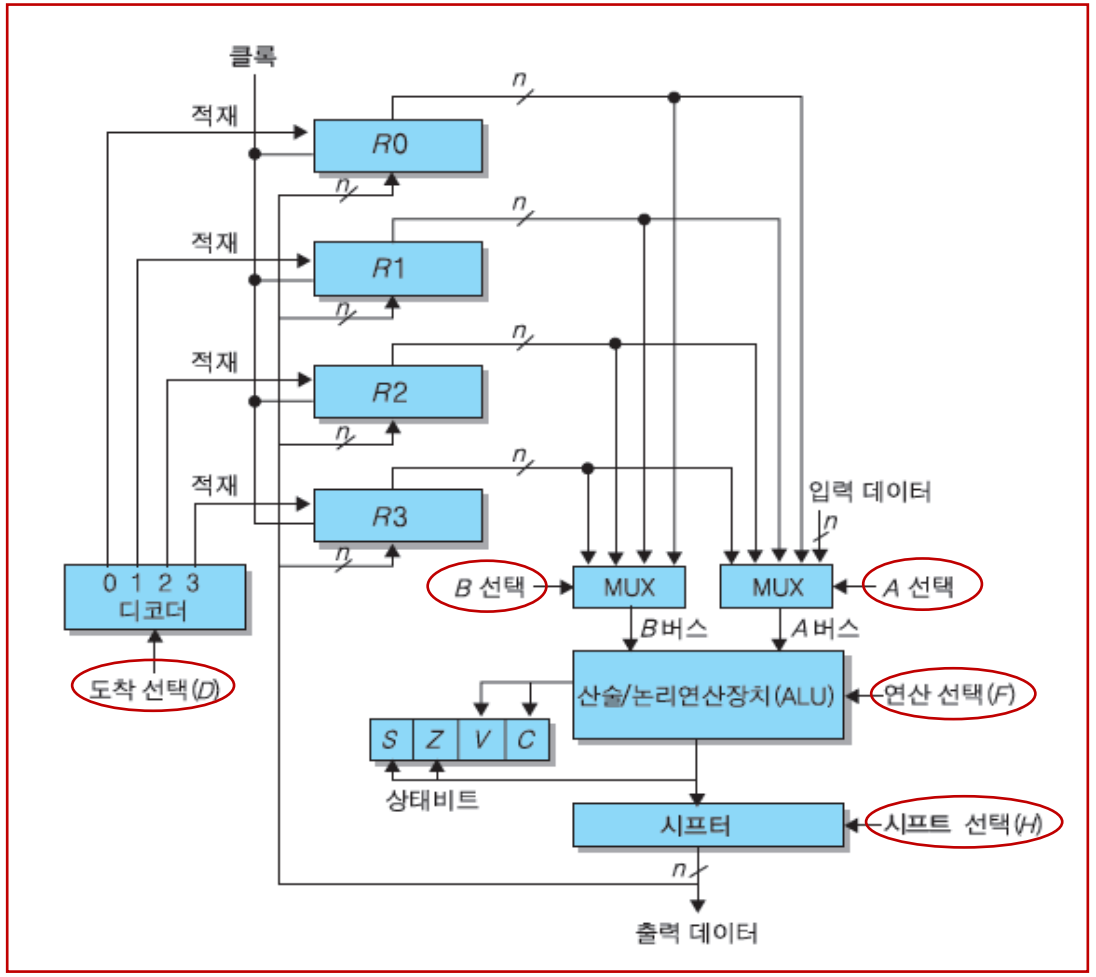
① 선택신호 A 는 *R1* 의 내용을 버스 A 로 적재

② 선택신호 B 는 *R2* 의 내용을 버스 B 로 적재

③ 선택신호 F 는 ALU에서 산술연산 *A+B* 를 수행

④ 선택신호 H 는 시프터에서 시프트 연산을 수행

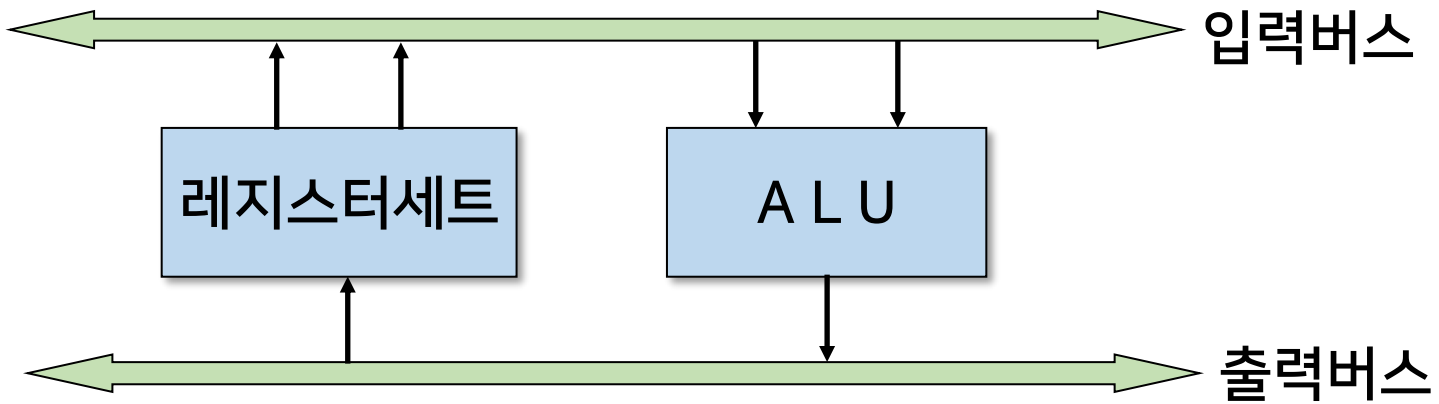
⑤ 선택신호 D 는 연산결과를 *R0* 로 적재



▶ 내부버스

› 레지스터들 간의 데이터 전송을 위한 공통선로의 집합

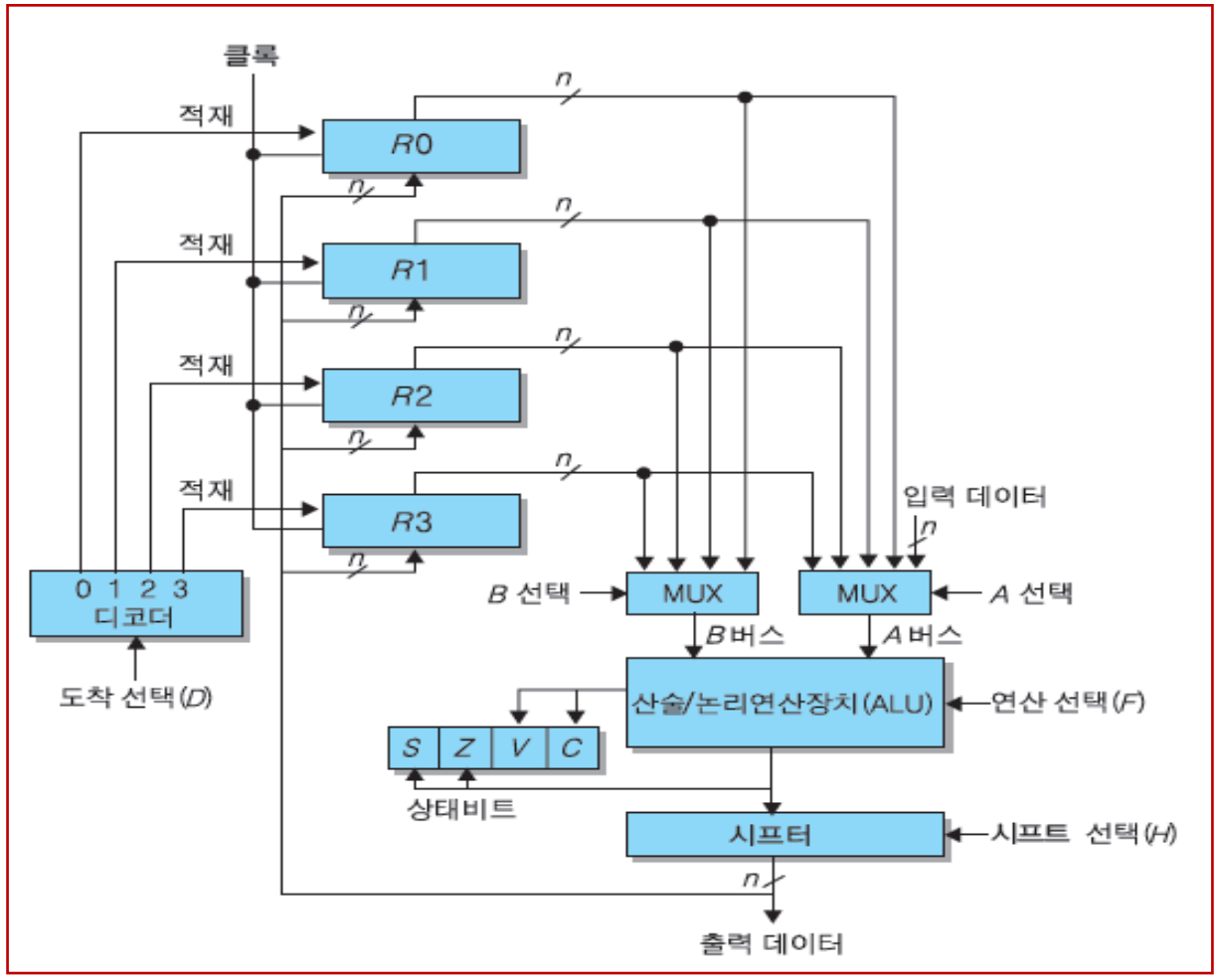
※ 내부버스의 개념도



※ 내부버스를 구성하는 방법

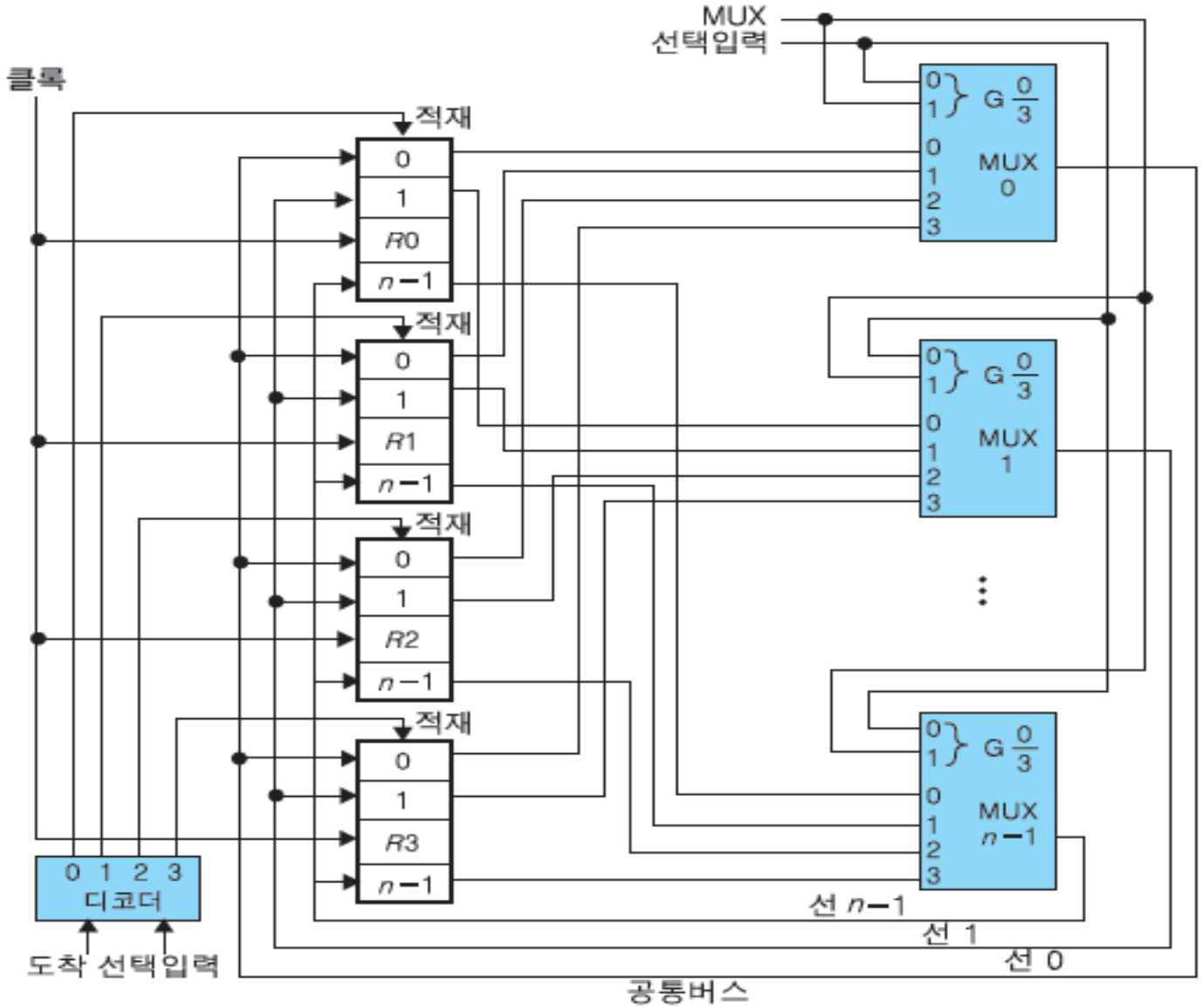
› 멀티플렉서와 디코더를 이용

- 멀티플렉서는 출발 레지스터 선택
- 디코더는 도착 레지스터를 선택



4.3.2 내부버스

※ 네 레지스터의 버스시스템

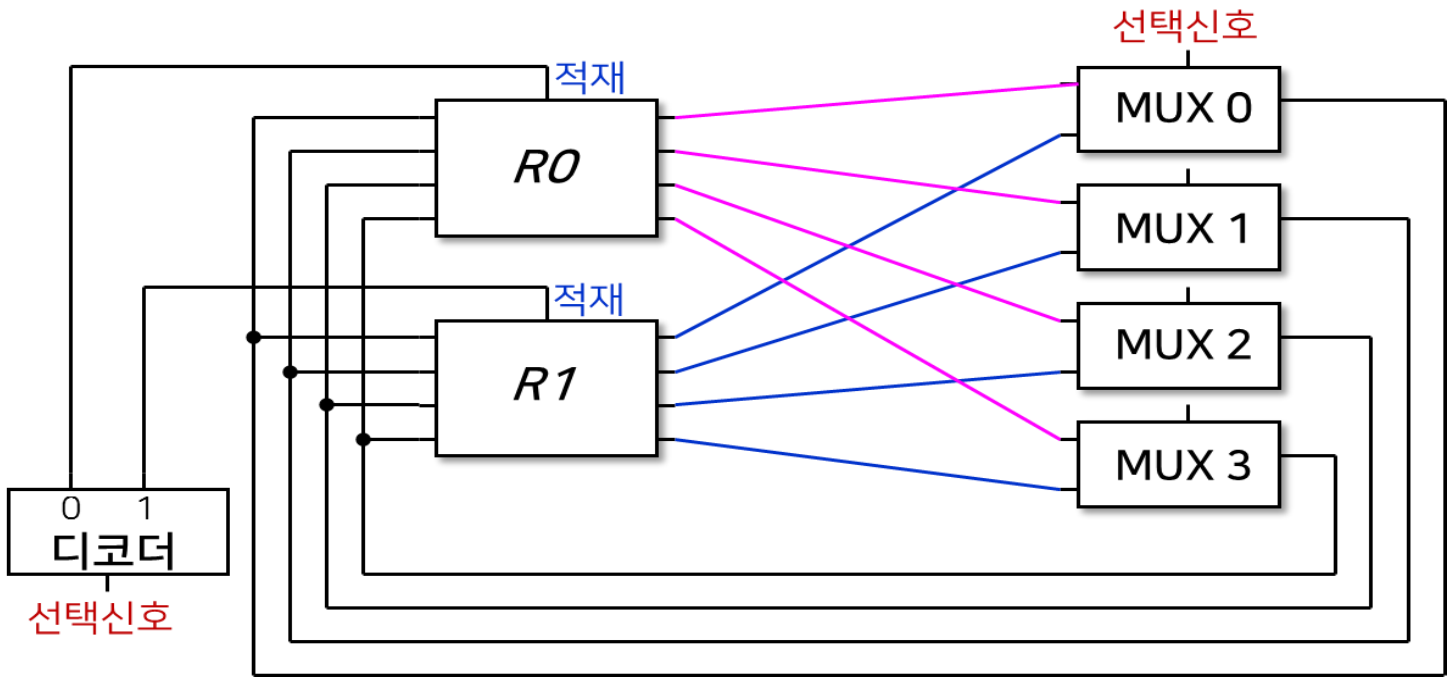


※ 간단한 내부버스의 구성 및 동작 예

☞ 마이크로 연산 : *R1* ← *R0*

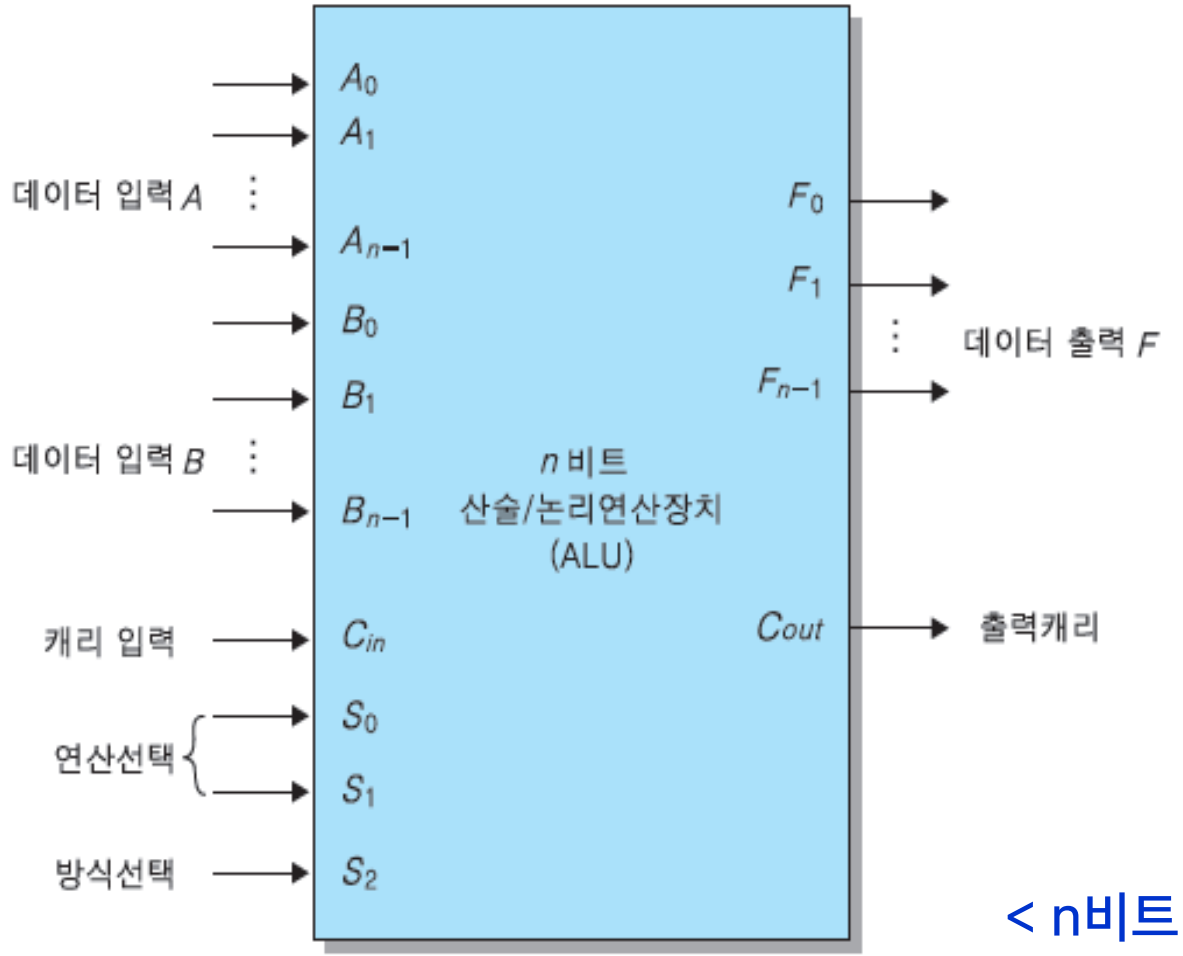
✓ *R0*, *R1* 이 4비트 레지스터 인 경우

- 내부버스 구성을 위해 : 2×1 MUX 4개, 1×2 디코더 1개 필요
- 마이크로 연산을 위해 : MUX 의 선택신호는 0(2진수), 디코더의 선택신호는 1(2진수) 부여



▶ 산술연산과 논리연산을 실행하는 조합논리회로

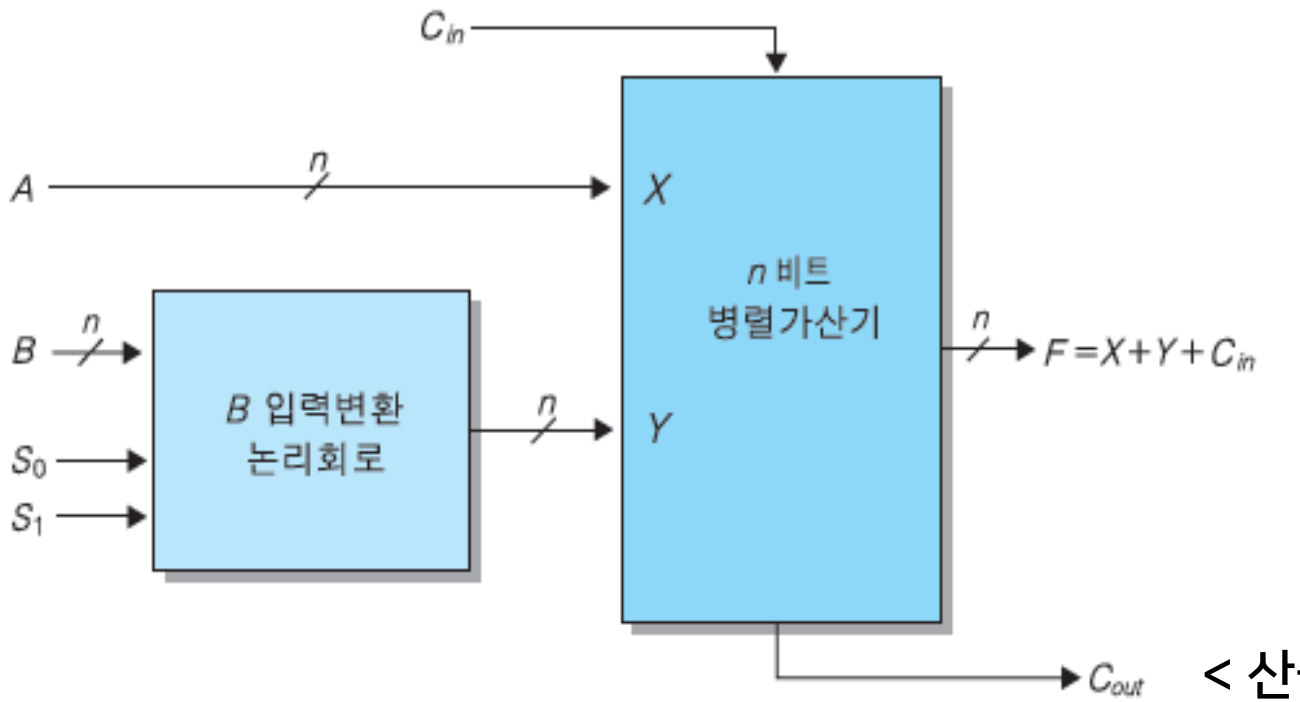
› 산술연산회로와 논리연산회로의 결합



< n비트 ALU의 블록도 >

▶ 산술연산회로

- 여러 개의 전가산기(FA)를 연속적으로 연결한 병렬가산기로 구성
- 병렬가산기로 들어가는 제어입력 값을 선택하여 여러 가지 형태의 산술연산을 실행



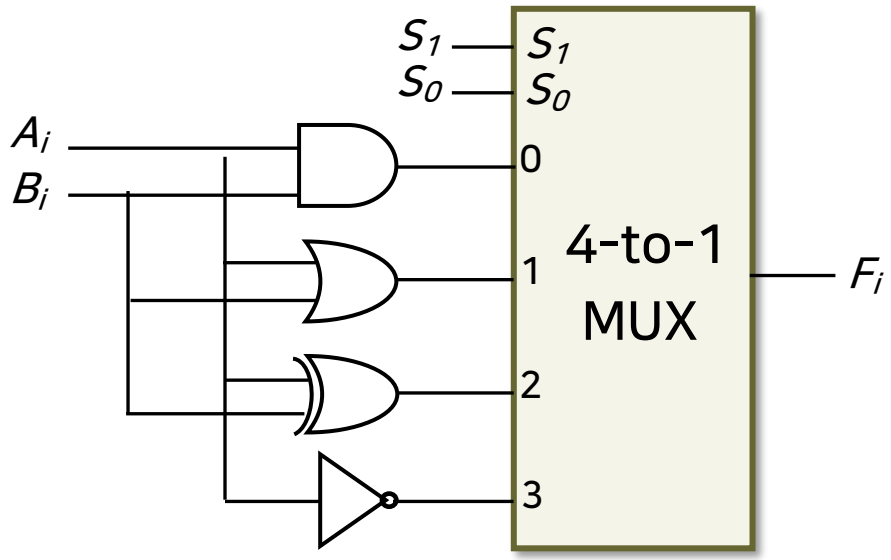
< 산술연산회로의 블록도 >

※ 산술연산의 종류

선택신호		입력값	$F = X + Y + C_{in}$	
S_1	S_0	Y	$C_{in} = 0$	$C_{in} = 1$
0	0	모두 0	$F = A$ (전송)	$F = A + 1$ (증가)
0	1	B	$F = A + B$ (가산)	$F = A + B + 1$
1	0	$\bar{B}$	$F = A + \bar{B}$	$F = A + \bar{B} + 1$ (감산)
1	1	모두 1	$F = A - 1$ (감소)	$F = A$ (전송)

▶ 논리연산회로

- 레지스터에 있는 각 비트를 독립된 2진 변수로 간주하여 비트별 연산을 실행
- AND, OR, XOR, NOT연산 등이 있으며, 이를 이용하여 복잡한 연산을 유도



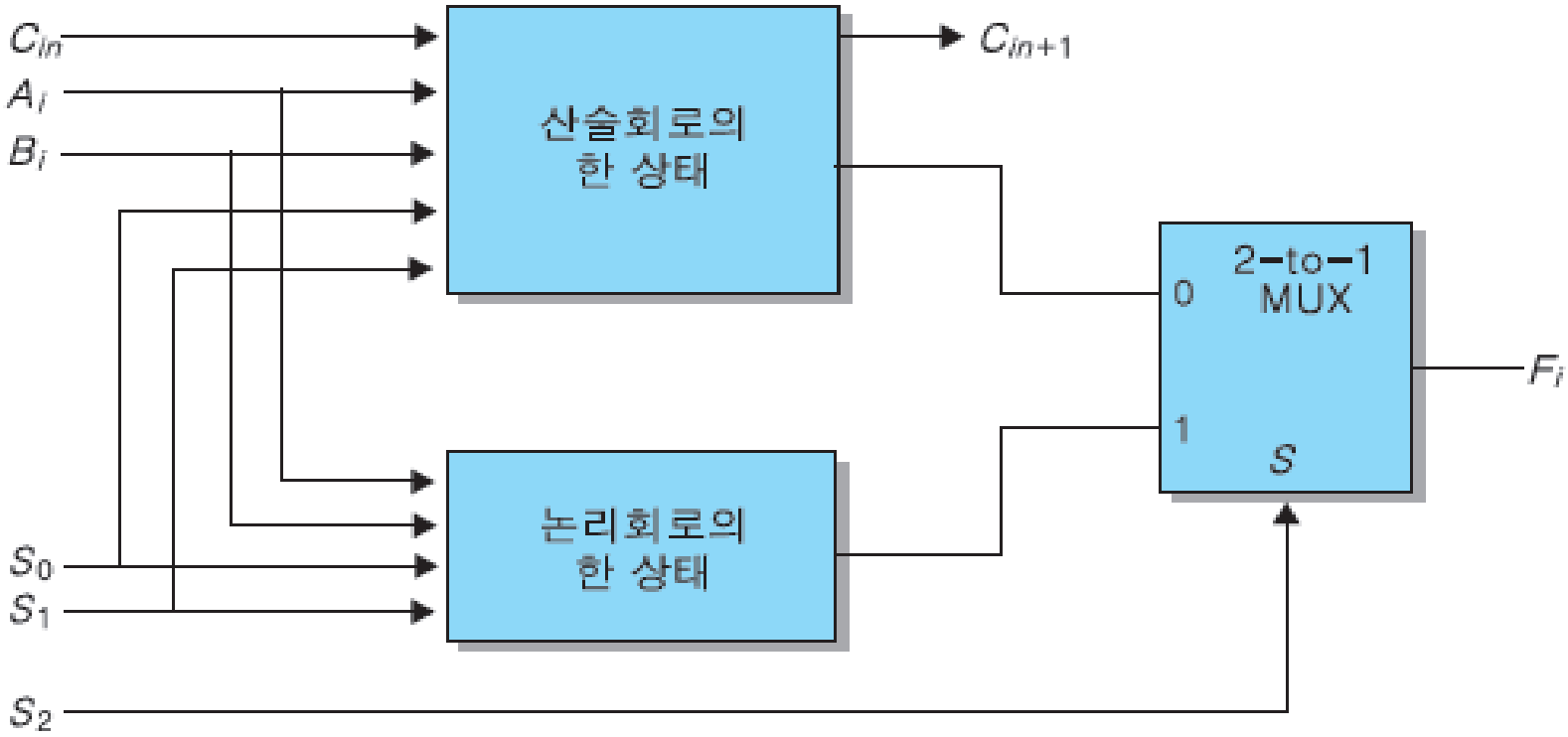
(a) 논리도

S_1	S_0	출 력	연 산
0	0	$F = A \wedge B$	AND
0	1	$F = A \vee B$	OR
1	0	$F = A \oplus B$	XOR
1	1	$F = \bar{A}$	NOT

(b) 함수표

▶ 산술논리연산회로

› 산술연산장치와 논리연산장치를 결합



< 한 단계의 ALU 구성도 >

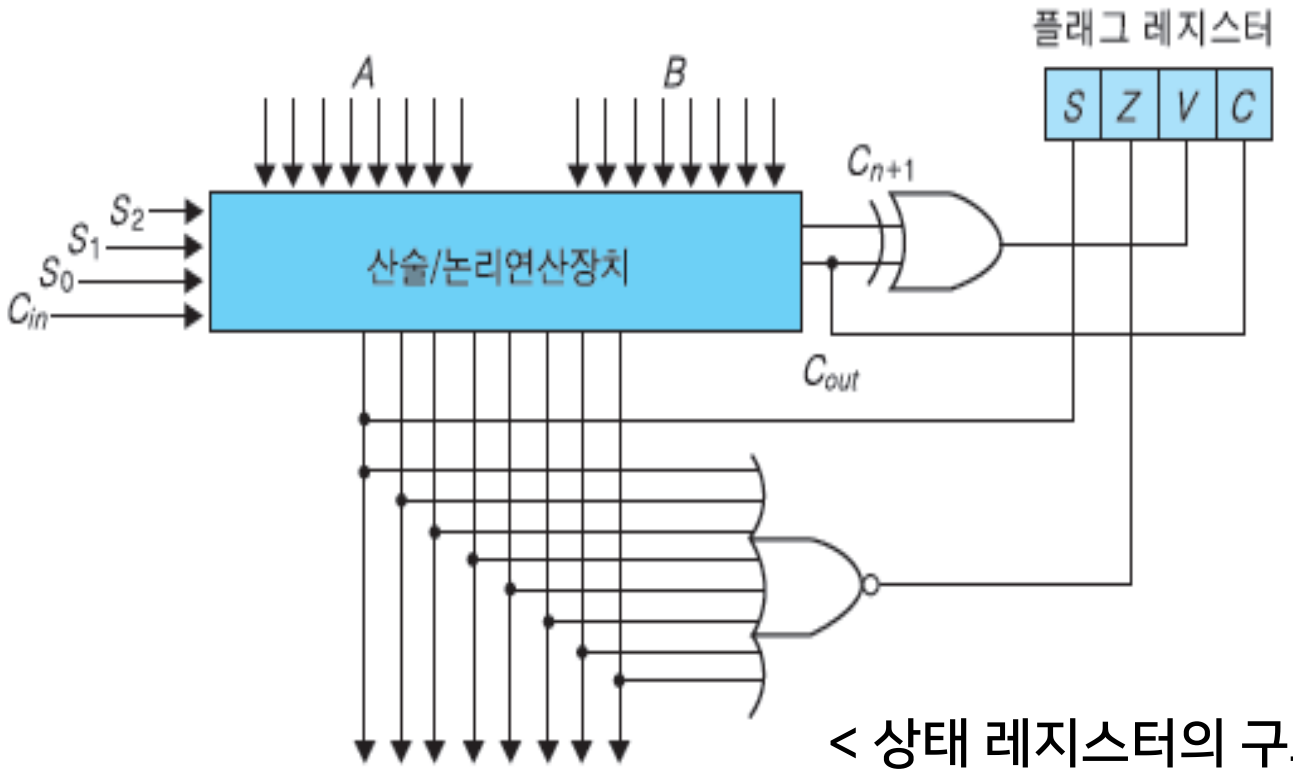
※ ALU에 대한 연산표

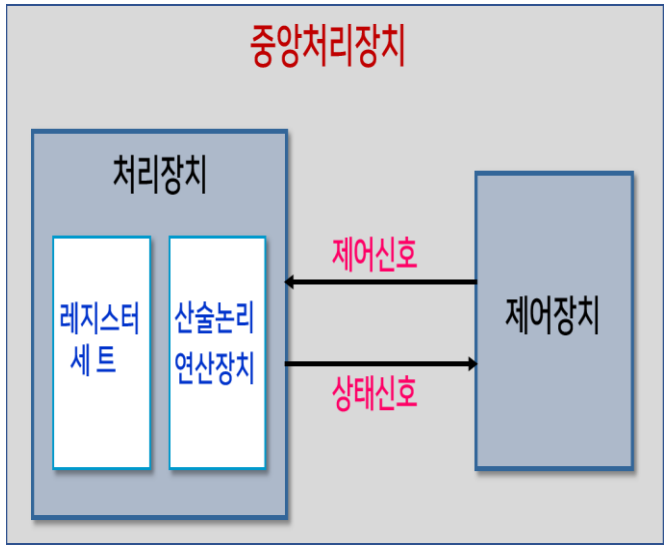
8비트 데이터의 경우				연 산	기 능
S_2	S_1	S_0	C_{in}		
0	0	0	0	$F = A$	A의 전송
0	0	0	1	$F = A + 1$	A에 1 더하기
0	0	1	0	$F = A + B$	덧셈
0	0	1	1	$F = A + B + 1$	캐리 값 1과 더하기
0	1	0	0	$F = A + \bar{B}$	A에 B의 1의 보수 더하기
0	1	0	1	$F = A + \bar{B} + 1$	뺄셈
0	1	1	0	$F = A - 1$	A에서 1 빼기
0	1	1	1	$F = A$	A의 전송
1	0	0	x	$F = A \wedge B$	AND
1	0	1	x	$F = A \vee B$	OR
1	1	0	x	$F = A \oplus B$	XOR
1	1	1	x	$F = \bar{A}$	A의 보수

▶ 상태 레지스터(flag register)

› ALU에서 산술연산이 수행된 후 연산결과에 의해 나타나는 상태 값을 저장

: 상태 레지스터는 C(carry bit), S(sign bit), Z(zero bit), V(overflow bit)로 구성

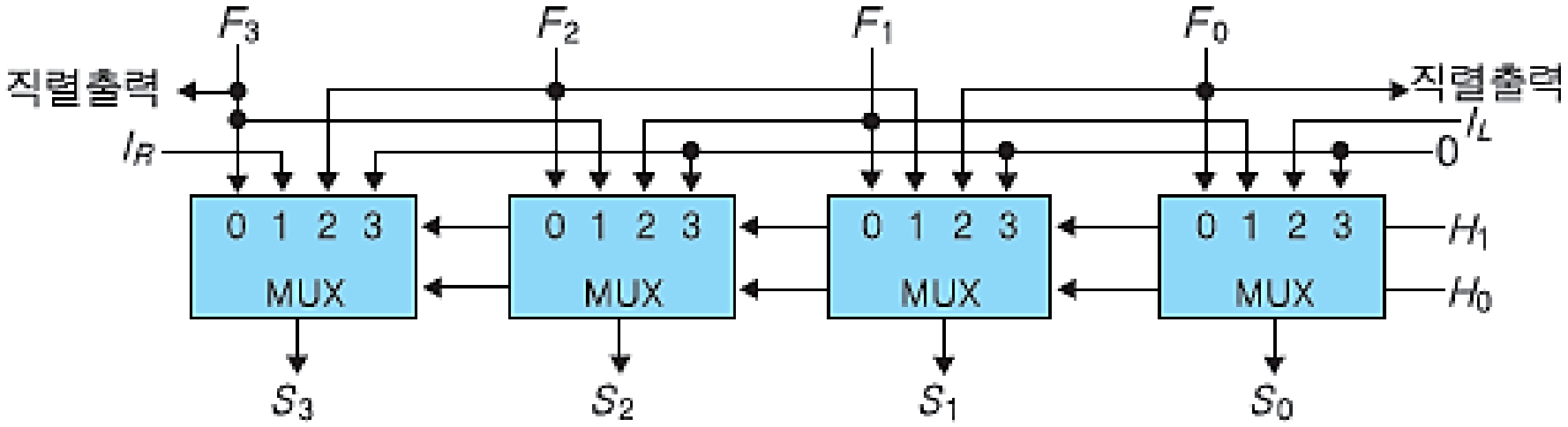




< 상태 레지스터의 구조 >

▶ 시프터(shifter)

› 입력 데이터의 모든 비트들을 각각 서로 이웃한 비트로 자리를 옮기는 시프트 연산을 수행



< 4비트 시프터의 구조 >

※ 시프터 연산의 종류

H_1	H_0	연 산	기 능
0	0	$S \leftarrow F$	시프트 없이 전송
0	1	$S \leftarrow shr\ F$	우측 시프트하여 전송
1	0	$S \leftarrow shl\ F$	좌측 시프트하여 전송
1	1	$S \leftarrow 0$	모든 출력비트에 0을 전송

▶ 제어단어(control word)

- 제어변수(선택신호)들의 묶음

※ 선택신호

- 처리장치내에서 수행되는 마이크로 연산을 선택하는 변수
- 처리장치의 버스, ALU, 쉬프터, 도착 레지스터 등을 제어
- 선택신호 즉, 제어변수가 특정한 마이크로 연산을 선택
- 이러한 제어변수들의 묶음을 제어단어(control word)라 함

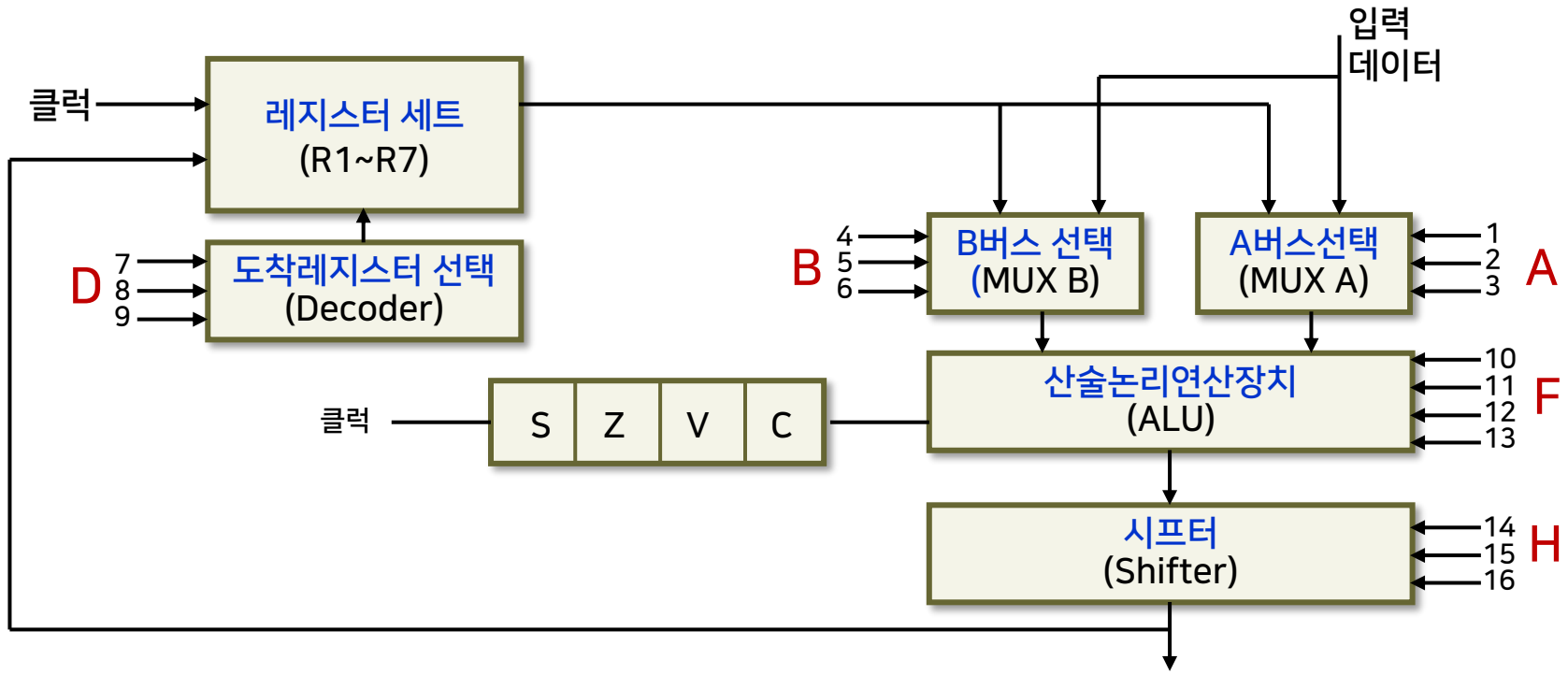
※ 제어단어를 살펴보기 위해

✓ 예를 들어 처리장치의 구성이 다음과 같다면

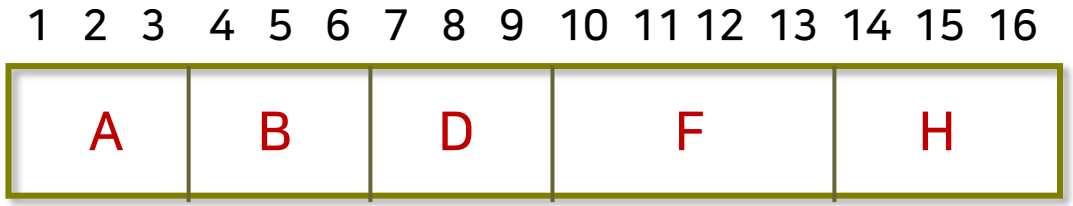
- 레지스터 세트 : 7개의 레지스터(R1 ~ R7)
- 산술논리연산장치 : 12가지 연산을 수행
- 시프터 : 6가지 연산을 수행

✓ 제어단어를 구성하는 선택신호(제어변수)는?

※ 처리장치의 구조에서 선택신호와 제어단어의 구성



(a) 처리장치의 블록도

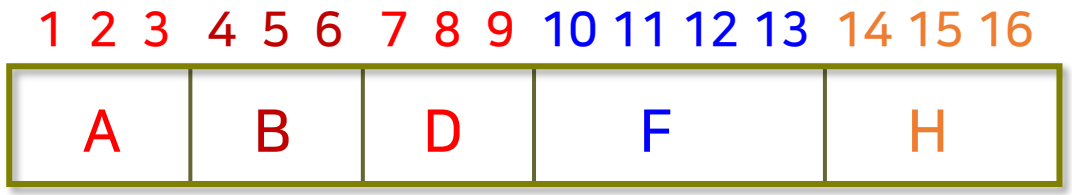


(b) 제어단어

※ 제어단어의 내역

✓ 예를 들어 처리장치의 구성이 다음과 같다면

- ➢ 레지스터 세트 : 7개의 레지스터(R1 ~ R7)
- ➢ 산술논리연산장치 : 12가지 연산을 수행
- ➢ 쉬프터 : 6가지 연산을 수행



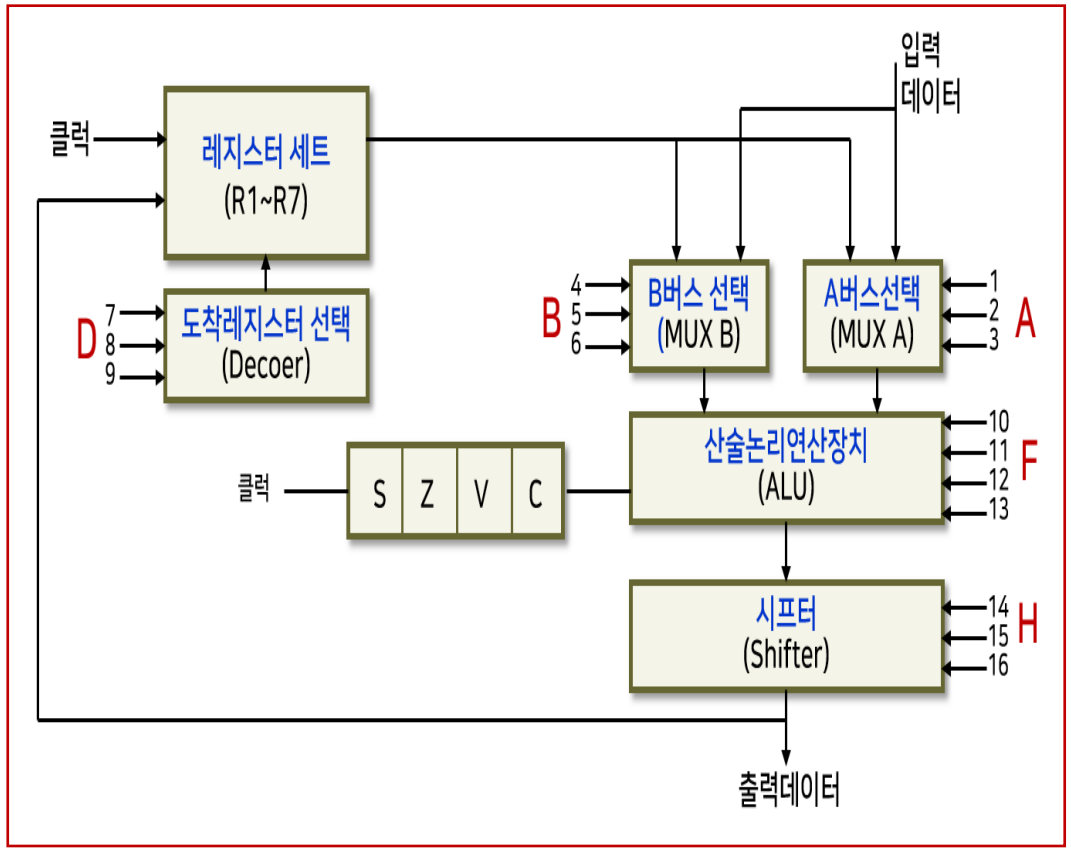
✓ A 필드 : ALU로 입력되는 A 버스 선택(3 비트)

✓ B 필드 : ALU로 입력되는 B 버스 선택(3 비트)

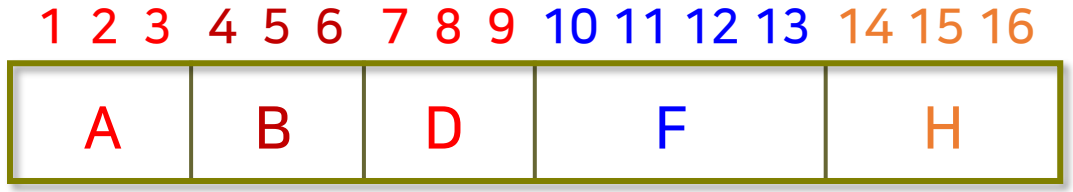
✓ D 필드 : 도착 레지스터 선택(3 비트)

✓ F 필드 : ALU의 연산 선택(4 비트)

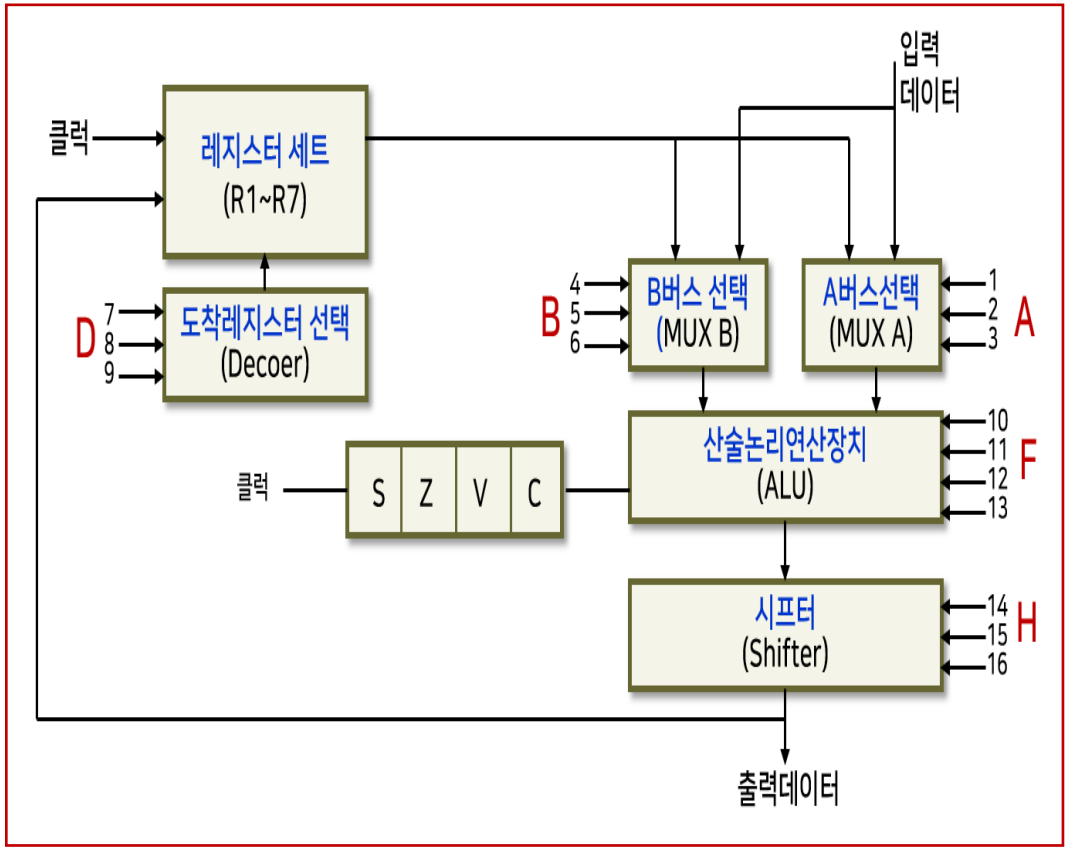
✓ H 필드 : 쉬프터의 연산 선택(3 비트)



※ 제어단어 각 필드의 동작



- A와 B 필드의 3비트
  : ALU로 입력되는 각각의 출발 레지스터를 선택
- D필드의 3비트
  : ALU의 결과가 저장될 도착 레지스터를 선택
- F필드의 4비트
  : ALU에서 이루어지는 12가지 연산 중 하나를 선택
- H필드의 3비트
  : 시프터에서의 시프트 연산 중 하나를 선택



✔ 전체 16비트로 구성된 선택신호들의 모임(제어단어)을 처리장치의 각 구성요소에 가하면

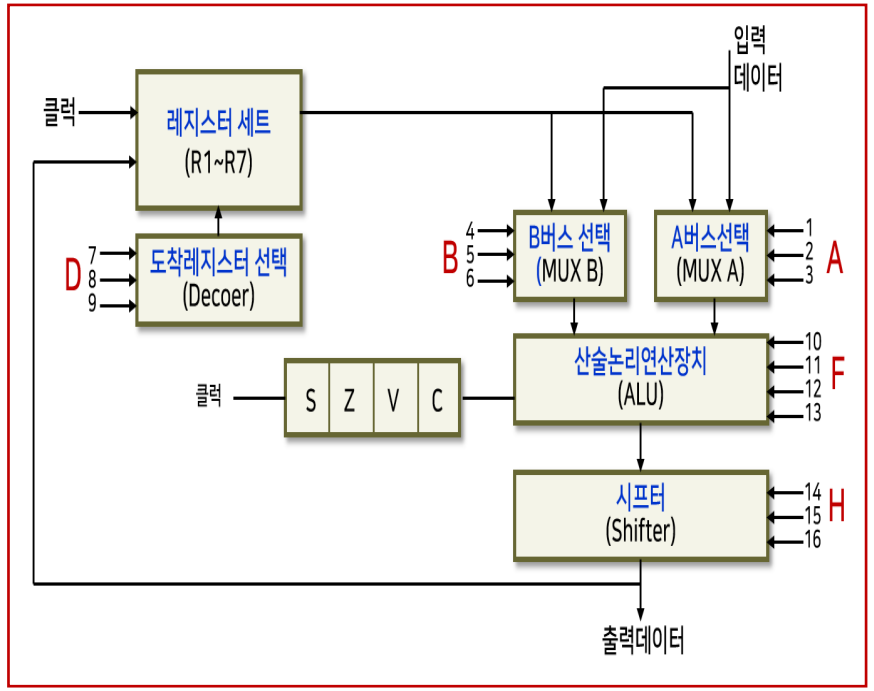
☞ 해당 마이크로 연산이 수행됨

※ 제어단어의 내역표

2진 코드	A	B	D	F		H
				$C_{in}=0$	$C_{in}=1$	
000	외부입력	외부입력	없음	$F=A$	$F=A+1$	시프트 없음
001	$R1$	$R1$	$R1$	$F=A+B$	$F=A+B+1$	*SHR*
010	$R2$	$R2$	$R2$	$F=A+\bar{B}$	$F=A-B$	*SHL*
011	$R3$	$R3$	$R3$	$F=A-1$	$F=A$	$bus=0$
100	$R4$	$R4$	$R4$	$F=A\wedge B$	-	-
101	$R5$	$R5$	$R5$	$F=A\vee B$	-	*ROR*
110	$R6$	$R6$	$R6$	$F=A\oplus B$	-	*ROL*
111	$R7$	$R7$	$R7$	$F=\bar{A}$	-	-

※ 제어단어의 작성 예

$$R1 \leftarrow R2 - R3$$



① A 필드 : ALU의 A 버스 입력으로 *R2* 의 내용을 보낸다.

② B 필드 : ALU의 B 버스 입력으로 *R3* 의 내용을 보낸다.

③ D 필드 : 연산 결과를 도착 레지스터 *R1*으로 보낸다.

④ F 필드 : ALU에서 감산 연산(*F=A-B*)을 수행한다.

⑤ H 필드 : 시프터에서 연산을 수행하지 않는다.(시프트 없음)

※ 제어단어의 작성 방법($R1 \leftarrow R2 - R3$)

<제어단어 내역표>

2진 코드	A	B	D	F		H
				$C_{in} = 0$	$C_{in} = 1$	
000	외부입력	외부입력	없음	$F = A$	$F = A + 1$	시프트 없음
001	$R1$	$R1$	**$R1$**	$F = A + B$	$F = A + B + 1$	*SHR*
010	**$R2$**	$R2$	$R2$	$F = A + \overline{B}$	**$F = A - B$**	*SHL*
011	$R3$	**$R3$**	$R3$	$F = A - 1$	$F = A$	$bus = 0$
100	$R4$	$R4$	$R4$	$F = A \wedge B$	-	-
101	$R5$	$R5$	$R5$	$F = A \vee B$	-	*ROR*
110	$R6$	$R6$	$R6$	$F = A \oplus B$	-	*ROL*
111	$R7$	$R7$	$R7$	$F = \overline{A}$	-	-

<제어단어 작성결과>

필 드	A	B	D	F	H
기 호	*R2*	*R3*	*R1*	*F=A-B*	시프트없음
2진 코드	010	011	001	0101	000

※ 여러 가지 마이크로 연산에 대한 제어단어의 예

마이크로 연산	기호표시					2진 제어단어				
	A	B	D	F	H	A	B	D	F	H
$R1 \leftarrow R2 - R3$	$R2$	$R3$	$R1$	$F = A - B$	시프트없음	010	011	001	0101	000
$R4 \leftarrow shr(R5 + R6)$	$R5$	$R6$	$R4$	$F = A + B$	*SHR*	101	110	100	0010	001
$R7 \leftarrow R7 + 1$	$R7$	-	$R7$	$F = A + 1$	시프트없음	111	000	111	0001	000
$R1 \leftarrow R2$	$R2$	–	$R1$	$F = A$	시프트없음	010	000	001	0000	000
$Output \leftarrow R3$	$R3$	-	$NONE$	$F = A$	시프트없음	011	000	000	0000	000
$R4 \leftarrow rol\ R4$	$R4$	-	$R4$	$F = A$	*ROL*	100	000	100	0000	110
$R5 \leftarrow 0$	-	-	$R5$	-	*bus = 0*	000	000	101	0000	011

▶ 제어단어 생성을 위한 효과적인 방법

› 작성된 제어단어를 기억장치에 저장하고,
기억장치의 출력을 처리장치의 각 구성요소의 선택신호로 연결

❖ 이렇게 하면 기억장치로부터 연속적인 제어단어를 읽음으로써
처리장치에서의 마이크로 연산이 정해진 순서대로,
연속적으로 수행된다. ⇒ 제어장치의 역할