

컴퓨터 구조

Chapter 04. 중앙 처리 장치

학습목표

- 컴퓨터 프로세서의 기본 구조와 명령 실행 과정을 이해한다.
- ALU 구조를 이해하고 프로세서에서의 산술 및 논리 연산을 학습한다.
- 프로세서 내의 레지스터의 종류와 용도를 이해한다.
- 프로세서 명령어 형식의 종류를 구분하고 명령의 동작을 이해한다.
- 명령의 주소 지정 방식을 이해하고 동작 원리를 학습한다.

내용

- 01 프로세서 구성과 동작
- 02 산술 논리 연산 장치
- 03 레지스터
- 04 컴퓨터 명령어
- 05 주소 지정 방식
- 06 CISC와 RISC

01 프로세서 구성과 동작

1 컴퓨터 기본 구조와 프로세서

- 컴퓨터의 3가지 핵심 장치 : 프로세서(Processor, CPU), 메모리, 입출력 장치
- 버스(Bus) : 장치 간에 주소, 데이터, 제어 신호를 전송하기 위한 연결 통로(연결선)

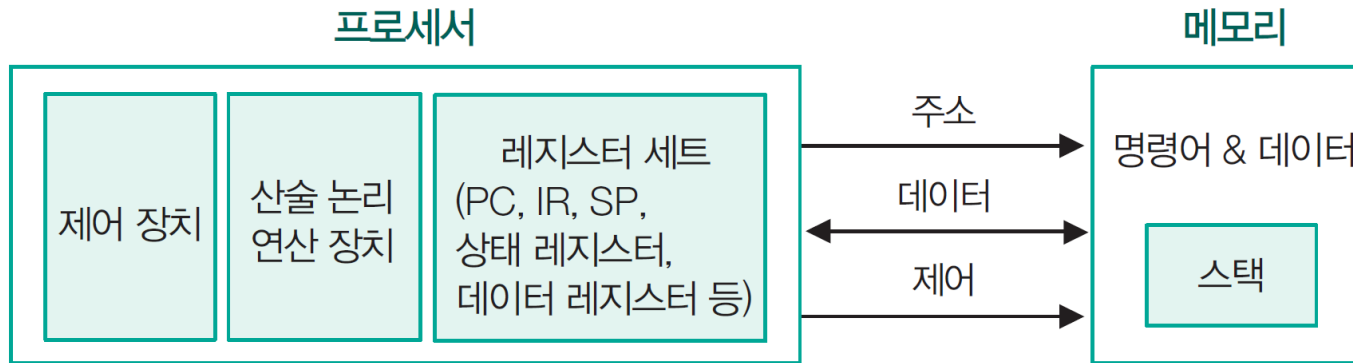


그림 4-1 폰 노이만 컴퓨터의 기본 구조

01 프로세서 구성과 동작

- **버스(Bus)** : 장치간에 주소, 데이터, 제어 신호를 전송하기 위한 연결 통로(연결선)
 - **내부 버스(internal bus)** : 프로세서 내부의 장치 연결
 - **시스템 버스(system bus)** : 핵심 장치 및 주변 장치 연결

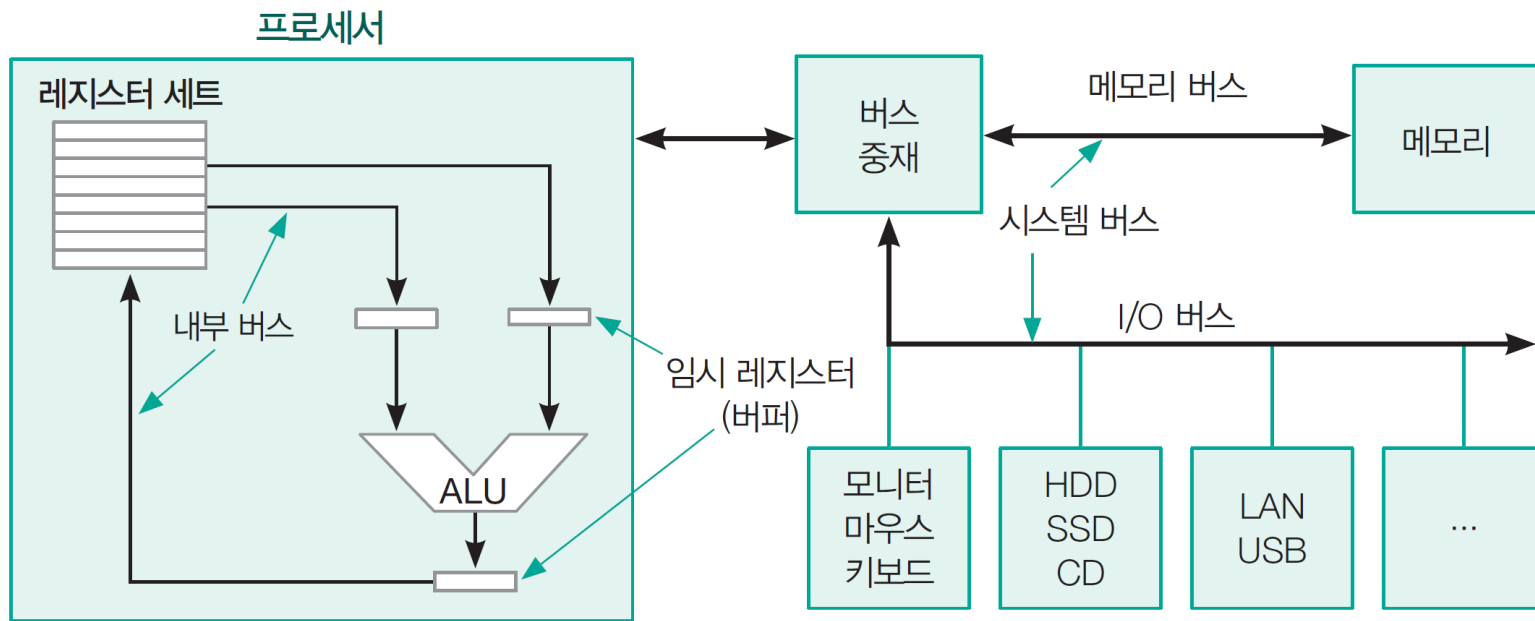


그림 4-2 버스 기반 컴퓨터 구조

01 프로세서 구성과 동작

2 프로세서 구성 요소

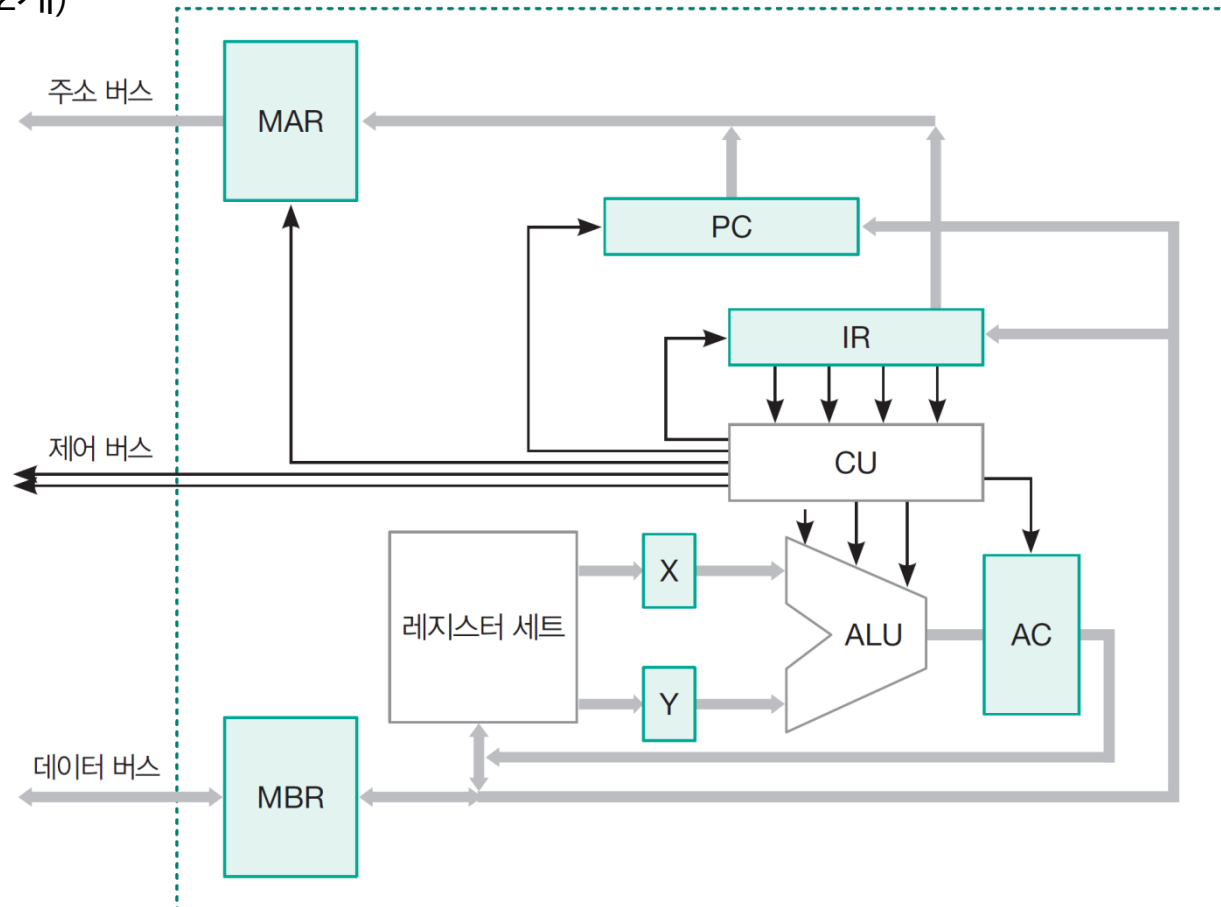
❖ 프로세서 3가지 구성 필수 구성요소

- 산술 논리 연산 장치(Arithmetic Logic Unit, ALU) : 산술 및 논리 연산 등 기본 연산을 수행
- 제어 장치 (Control Unit, CU) : 메모리에서 명령어를 가져와 해독하고 실행에 필요한 장치들을 제어하는 신호를 발생
- 레지스터 세트(register set) : 프로세서 내에 존재하는 용량은 작지만 매우 빠른 메모리, ALU의 연산과 관련된 데이터를 일시 저장하거나 특정 제어 정보 저장
 - 목적에 따라 특수 레지스터와 범용 레지스터로 분류
- 현재는 온칩 캐시(on-chip cache), 비디오 컨트롤러(video controller), 실수보조연산 프로세서(FPU) 등 다양한 장치 포함

01 프로세서 구성과 동작

3 프로세서 기본 구조

- 레지스터 세트(일반적으로 1~32개)
- ALU
- CU
- 이들 장치를 연결하는 버스로 구성



MAR Memory Address Register: 메모리 주소 레지스터 PC Program Counter: 프로그램 카운터 CU Control Unit: 제어 장치
AC Accumulator: 누산기 MBR(MDR) Memory Buffer(Data) Register: 메모리 버퍼(데이터) 레지스터
IR Instruction Register: 명령 레지스터 ALU Arithmetic Logic Unit: 산술 논리 연산 장치

그림 4-3 프로세서 기본 구조

01 프로세서 구성과 동작

❖ ALU

- 덧셈, 뺄셈 등 연산을 수행하고, 그 결과를 **누산기**(Accumulator, AC)에 저장

❖ 프로세서 명령 분류

레지스터-메모리 명령	<ul style="list-style-type: none">• 메모리 워드를 레지스터로 가져올(LOAD) 때• 레지스터의 데이터를 메모리에 다시 저장(STORE)할 때
레지스터-레지스터 명령	<ul style="list-style-type: none">• 레지스터에서 오퍼랜드 2개를 ALU의 입력 레지스터로 가져와 덧셈 또는 논리 AND 같은 몇 가지 연산을 수행하고• 그 결과를 레지스터 중 하나에 다시 저장

01 프로세서 구성과 동작

4 프로세서 명령 실행

- 프로세서는 각 명령을 더 작은 **마이크로 명령**(microinstruction)들로 나누어 실행
 - 1단계** 다음에 실행할 명령어를 메모리에서 읽어 명령 레지스터(IR)로 가져온다.
 - 2단계** 프로그램 카운터(PC)는 그다음에 읽어올 명령어의 주소로 변경된다.
 - 3단계** 제어 장치는 방금 가져온 명령어를 해독(decode)하고 유형을 결정한다.
 - 4단계** 명령어가 메모리에 있는 데이터를 사용하는 경우 그 위치를 결정한다.
 - 5단계** 필요한 경우 데이터를 메모리에서 레지스터로 가져온다.
 - 6단계** 명령어를 실행한다.
 - 7단계** 1단계로 이동하여 다음 명령어 실행을 시작한다.
- 이 단계를 요약하면 **인출**(fetch)-**해독**(decode)-**실행**(execute) 사이클로 구성 – 주 사이클(main cycle)

02 산술 논리 연산 장치

❖ 산술 논리 연산 장치(Arithmetic Logic Unit, ALU) : 산술 연산과 논리 연산

- 주로 정수 연산을 처리
- 부동 소수(Floating-point Number) 연산 : FPU(Floating-Point Unit)
- 최근에는 ALU가 부동 소수 연산까지 처리

❖ 산술 연산 : 덧셈, 뺄셈, 곱셈, 나눗셈, 증가, 감소, 보수

❖ 논리 연산 : AND, OR, NOT, XOR, 시프트(shift)

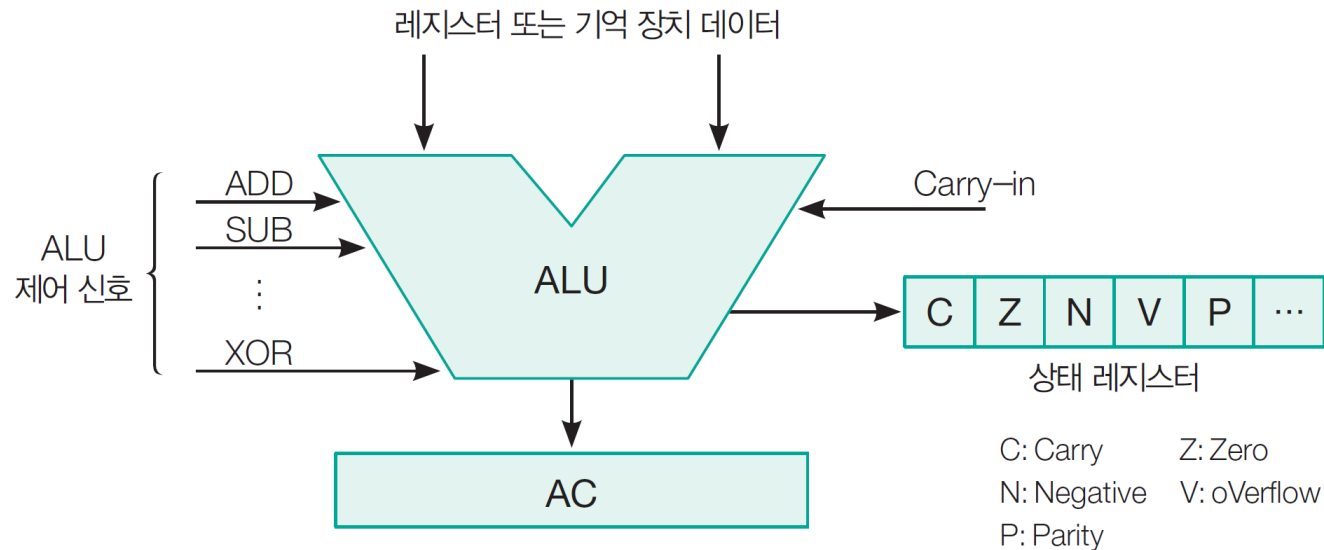


그림 4-4 ALU의 동작

02 산술 논리 연산 장치

1 산술 연산

표 4-1 산술 연산

연산	8비트 연산	
	동작	설명
ADD	$X \leftarrow A + B$	A와 B를 더한다.
SUB	$X \leftarrow A + (\sim B + 1)$	A + (B의 2의 보수)
MUL	$X \leftarrow A * B$	A와 B를 곱한다.
DIV	$X \leftarrow A / B$	A와 B를 나눈다.
INC	$X \leftarrow A + 1$	A를 1 증가시킨다.
DEC	$X \leftarrow A - 1(0xFF)$	A를 1 감소시킨다.
NEG	$X \leftarrow \sim A + 1$	A의 2의 보수다.

❖ Booth Algorithm

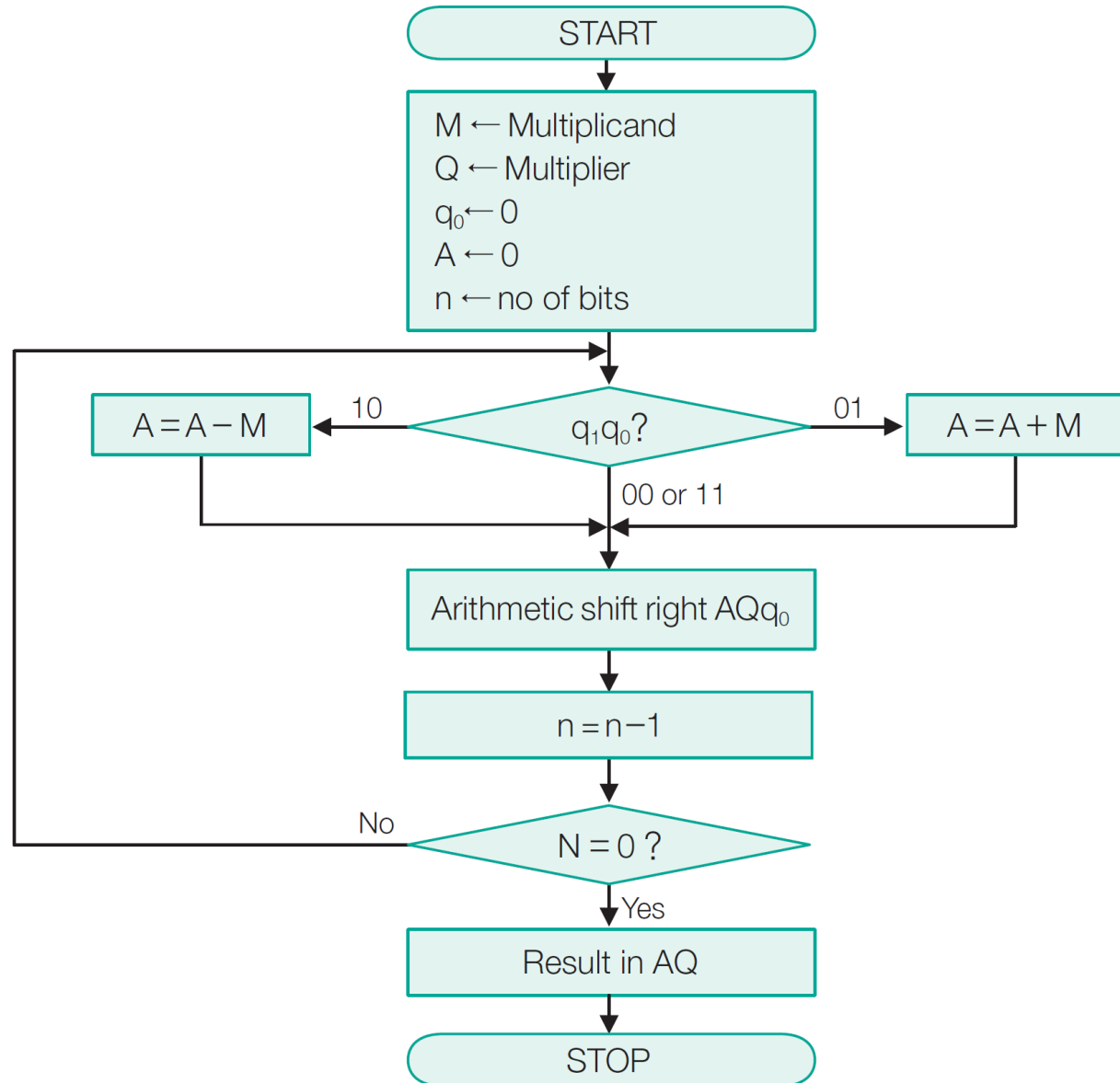


그림 4-5 부스 알고리즘 순서도

02 산술 논리 연산 장치

❖ Booth Algorithm 예 : $(-7) * (+3)$

n	A	Q($q_4q_3q_2q_1$)	q_0	설명
4	0000	0011	0	초기 상태
	0111	0011	0	q_1q_0 이 10이므로 $A=A-M=0000+0111$
3	0011	1001	1	AQ q_0 을 오른쪽 산술 시프트
2	0001	1100	1	q_1q_0 이 11이므로 연산 없이 AQ q_0 을 오른쪽 산술 시프트
	1010	1100	1	q_1q_0 이 01이므로 $A=A+M=0001+1001$
1	1101	0110	0	AQ q_0 를 ASR
0	1110	1011	0	q_1q_0 가 00이므로 연산 없이 AQ q_0 을 오른쪽 산술 시프트

02 산술 논리 연산 장치

❖ Booth Algorithm 예 : $5 * (-4)$

n	A	$Q(q_4q_3q_2q_1)$	q_0	설명
4	0000	1100	0	초기 상태
3	0000	0110	0	q_1q_0 이 00이므로 연산 없이 AQq_0 을 오른쪽 산술 시프트
2	0000	0011	0	q_1q_0 이 00이므로 연산 없이 AQq_0 을 A오른쪽 산술 시프트
1	1011	0011	0	q_1q_0 이 10이므로 $A=A-M=0000+1011$
	1101	1001	1	AQq_0 을 오른쪽 산술 시프트
0	1110	1100	1	q_1q_0 이 11이므로 연산 없이 AQq_0 을 오른쪽 산술 시프트

02 산술 논리 연산 장치

2 논리 연산과 산술 시프트 연산

표 4-2 논리 연산

연산	8비트 연산	
	동작	설명
AND	$X \leftarrow A \& B$	A와 B를 비트 단위로 AND 연산한다.
OR	$X \leftarrow A B$	A와 B를 비트 단위로 OR 연산한다.
NOT	$X \leftarrow \sim A$	A의 1의 보수를 만든다.
XOR	$X \leftarrow A \wedge B$	A와 B를 비트 단위로 XOR 연산한다.
ASL	$X \leftarrow A \ll n$	왼쪽으로 n비트 시프트(LSL과 같다.)
ASR	$X \leftarrow A \gg n, A[7] \leftarrow A[7]$	오른쪽으로 n비트 시프트(부호 비트는 그대로 유지한다.)
LSL	$X \leftarrow A \ll n$	왼쪽으로 n비트 시프트
LSR	$X \leftarrow A \gg n$	오른쪽으로 n비트 시프트
ROL	$X \leftarrow A \ll 1, A[0] \leftarrow A[7]$	왼쪽으로 1비트 회전 시프트, MSB는 LSB로 시프트
ROR	$X \leftarrow A \gg 1, A[7] \leftarrow A[0]$	오른쪽으로 1비트 회전 시프트, LSB는 MSB로 시프트
ROLC	$X \leftarrow A \ll 1, C \leftarrow A[7], A[0] \leftarrow C$	캐리도 함께 왼쪽으로 1비트 회전 시프트
RORC	$X \leftarrow A \gg 1, C \leftarrow A[0], A[7] \leftarrow C$	캐리도 함께 오른쪽으로 1비트 회전 시프트

02 산술 논리 연산 장치

❖ 논리 연산 예 1 : $A=46=00101110_{(2)}$, $B=-75=10110101_{(2)}$

A AND B	A OR B	A XOR B
00101110 46	00101110 46	00101110 46
& 10110101 -75	10110101 -75	^ 11111111 -128
00100100 36	10111111 -65	11010001 -47

02 산술 논리 연산 장치

❖ 논리 연산 예 2

A AND B		A OR B	
00101110		00001110	
& 00001111	상위 4비트 삭제	10110000	상위 4비트 값 설정
00001110		10111110	

02 산술 논리 연산 장치

❖ 시프트 연산 예

연산	왼쪽	오른쪽
산술 시프트	<p>MSB LSB</p> <p>ASL</p> <p>0 0 0 1 0 1 1 0</p> <p>0 0 1 0 1 1 0 0 ← 0</p>	<p>MSB LSB</p> <p>ASR</p> <p>1 0 0 1 0 1 1 0</p> <p>1 1 0 0 1 0 1 1</p>
논리 시프트	<p>MSB LSB</p> <p>LSL</p> <p>0 0 0 1 0 1 1 0</p> <p>0 0 1 0 1 1 0 0 ← 0</p>	<p>MSB LSB</p> <p>LSR</p> <p>1 0 0 1 0 1 1 0</p> <p>0 → 0 1 0 0 1 0 1 1</p>
회전 시프트	<p>MSB LSB</p> <p>ROL</p> <p>0 0 0 1 0 1 1 0</p> <p>0 0 1 0 1 1 0 0</p>	<p>MSB LSB</p> <p>ROR</p> <p>1 0 0 1 0 1 1 0</p> <p>0 1 0 0 1 0 1 1</p>
캐리와 함께 회전 시프트	<p>MSB LSB C</p> <p>ROLC</p> <p>0 0 0 1 0 1 1 0 0</p> <p>0 0 1 0 1 1 0 0 0</p>	<p>MSB LSB C</p> <p>RORC</p> <p>0 0 0 1 0 1 1 0 1</p> <p>1 0 0 0 1 0 1 1 1</p>

1 레지스터 동작

- 레지스터는 CPU가 사용하는 데이터와 명령어를 신속하게 읽어 오고 저장하고 전송하는 데 사용
- 레지스터는 메모리 계층의 최상위에 있으며 시스템에서 가장 빠른 메모리
- 매우 단순한 마이크로프로세서는 누산기(AC) 레지스터 1개로만 구성 가능
- 레지스터 용도에 따른 종류
 - 누산기(Accumulator, AC)
 - 프로그램 카운터(Program Counter, PC)
 - 명령 레지스터(Instruction Register, IR)
 - 메모리 데이터 레지스터(Memory Data Register : MDR, Memory Buffer Register : MBR)
 - 메모리 주소 레지스터(Memory Address Register, MAR)
- 데이터(범용) 레지스터는 보통 8~32개 정도, 많으면 128개 이상인 경우도 있음
- 특수 레지스터는 8~16개 정도

2 레지스터 종류

❖ 프로그램 카운터(PC)

- 명령 포인터 레지스터라고도 하며, 실행을 위해 인출(fetch)할 다음 명령의 주소를 저장하는데 사용
- 명령어가 인출되면 PC 값이 단위 길이(명령 크기)만큼 증가
- 항상 가져올 다음 명령의 주소 유지

❖ 메모리 주소 레지스터(Memory Address Register, MAR)

- CPU가 읽고 쓰기 위한 데이터의 메모리 주소 저장
- 메모리에 데이터를 저장하거나 읽을 때 필요한 메모리 위치의 주소를 MAR로 전송

❖ 메모리 버퍼 레지스터(Memory Buffer Register, MBR, MDR)

- 메모리에서 데이터를 읽거나 메모리에 저장될 명령의 데이터를 일시적 저장
- 명령어 내용은 명령 레지스터로 전송되고, 데이터 내용은 누산기 또는 I/O 레지스터로 전송

03 레지스터

❖ 명령 레지스터(Instruction Register, IR)

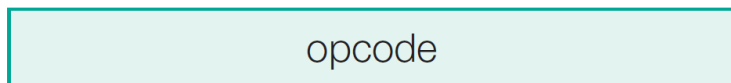
- 주기억 장치에서 인출한 명령어 저장
- 제어 장치는 IR에서 명령어를 읽어 와서 해독하고 명령을 수행하기 위해 컴퓨터의 각 장치에 제어 신호 전송

❖ 누산기(ACcumulator register, AC)

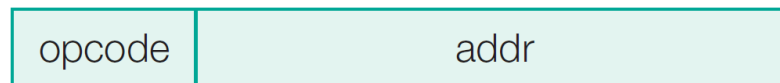
- ALU 내부에 위치하며, ALU의 산술 연산과 논리 연산 과정에 사용
- 제어 장치는 주기억 장치에서 인출된 데이터 값을 산술 연산 또는 논리 연산을 위해 누산기에 저장
- 이 레지스터는 연산할 초기 데이터, 중간 결과 및 최종 연산 결과 저장
- 최종 결과는 목적지 레지스터나 MBR을 이용하여 주기억 장치로 전송

1 명령어 형식

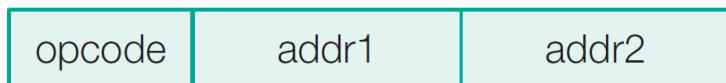
- 연산 코드(opcode), 오퍼랜드(operand), 피연산자 위치, 연산 결과의 저장 위치 등 여러 가지 정보로 구성



(a) 0-주소 명령어



(b) 1-주소 명령어



(c) 2-주소 명령어



(d) 3-주소 명령어

그림 4-9 명령어 형식

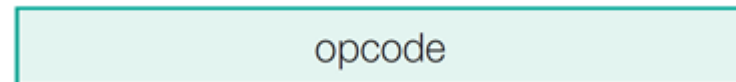
04 컴퓨터 명령어

❖ 0-주소 명령어

- 자료의 주소를 지정하는 Operand부 없이 OP-Code부만으로 구성
- 모든 연산은 Stack 메모리의 Stack Pointer(TOP)가 가리키는 Operand를 이용
- 모든 연산은 스택에 있는 자료를 이용하여 수행하기 때문에 스택 머신 이라고 함.
- 명령어의 길이가 짧아서 수행시간이 짧지만 전체 프로그램의 길이는 길어짐
- 수식을 계산하기 위해 수식을 후위식(역Polish) 형태로 변경해야 함
- 주소의 사용 없이 스택에 연산자와 피연산자를 넣었다 꺼내어 연산한 후 결과를 다시 스택에 넣으면서 연산하기 때문에 원래 자료가 남지 않음

❖ 예제 $D = (A + B) * C$

- PUSH A
- PUSH B
- ADD
- PUSH C
- MUL
- POP D



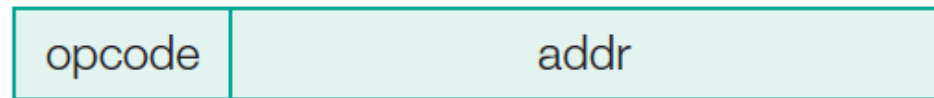
(a) 0-주소 명령어

❖ 1-주소 명령어

- 연산 대상이 되는 2개 중 하나만 표현하고 나머지 하나는 묵시적으로 지정: 누산기(AC)
- 기억 장치 내의 데이터와 AC 내의 데이터로 연산
- 연산 결과는 AC에 저장
- 다음은 기억 장치 X번지의 내용과 누산기의 내용을 더하여 결과를 다시 누산기에 저장
$$\text{ADD } X \quad ; AC \leftarrow AC + M[X]$$
- 오퍼랜드 필드의 모든 비트가 주소 지정에 사용: 보다 넓은 영역의 주소 지정

❖ 예제 $D = (A + B) * C$

- LOAD A
- ADD B
- MUL C
- STORE D



(b) 1-주소 명령어

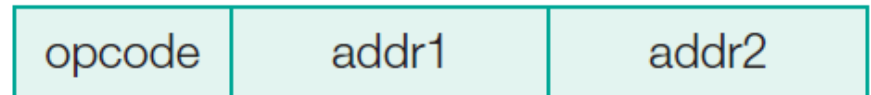
❖ 2-주소 명령어

- 오퍼랜드부가 두 개로 구성되는, 가장 일반적으로 사용되는 명령어 형식
- 연산에 필요한 두 오퍼랜드 중 하나가 결과 값 저장
- 레지스터 R1과 R2의 내용을 더하고 그 결과를 레지스터 R1에 저장
- R1 레지스터의 기존 내용은 지워짐

ADD R1, R2 ; $R1 \leftarrow R1 + R2$

❖ 예제 $D = (A + B) * C$

- MOVE R1, A
- ADD R1, B
- MUL R1, C
- MOVE D, R1



(c) 2- 주소 명령어

❖ 3-주소 명령어

- 연산에 필요한 오퍼랜드 2개와 결과 값의 저장 장소가 모두 다름
- 레지스터 R2와 R3의 내용을 더하고 그 결과 값을 레지스터 R1에 저장하는 명령어다.
- 연산 후에도 입력 데이터 보존
- 다른 형식의 명령어 보다 프로그램의 전체 길이가 짧아짐 / 명령어 한 개의 길이가 길어짐
- 명령어 해독 과정이 복잡해짐

ADD R1, R2, R3 ; $R1 \leftarrow R2 + R3$

- 하나의 명령을 수행하기 위해 최소한 4번 기억장소에 접근해야 하므로 수행시간이 길어짐

❖ 예제 $D = (A + B) * C$

- ADD R1, A, B
- MUL D, C, P



(d) 3-주소 명령어

04 컴퓨터 명령어

❖ 0-주소, 1-주소, 2-주소, 3-주소 명령을 사용하여 $Z=(B+C) \times A$ 를 구현한 예

- 니모닉(mnemonic)

ADD : 덧셈

MUL : 곱셈

MOV : 데이터 이동(레지스터와 기억 장치 간)

LOAD : 기억 장치에서 데이터를 읽어 누산기에 저장

STOR : AC의 내용을 기억 장치에 저장

0-주소	1-주소	2-주소	3-주소
PUSH B	LOAD B	MOV R1, B	ADD R1, B, C
PUSH C	ADD C	ADD R1, C	MUL Z, A, R1
ADD	MUL A	MUL R1, A	
PUSH A	STOR Z	MOV Z, R1	
MUL			
POP Z			

1 즉시 주소 지정(immediate addressing mode, 즉시 주소 지정)

- 오퍼랜드를 지정하는 가장 간단한 방법
 - 명령어 자체에 오퍼랜드를 포함
 - 오퍼랜드가 포함되어 명령어가 인출될 때 오퍼랜드도 자동으로 인출
 - 즉시(즉치) 오퍼랜드 : 즉시 사용 가능
- 레지스터 R1에 상수 4를 저장하는 즉시 주소 지정 명령어의 예

MOVE	R1	4
------	----	---

그림 4-13 즉시 주소 지정

- 장점 : 오퍼랜드를 인출을 위한 메모리 참조가 필요 없음
CPU에서 곧바로 자료를 이용할 수 있어서 실행속도 빠름
- 단점 : 상수만 가능, 상수 값의 크기가 필드 크기(오퍼랜드 길이)로 제한
- 작은 값의 정수를 지정하는 데 많이 사용

2 직접 주소 지정(direct addressing mode)

- 메모리에 위치한 오퍼랜드의 전체 주소 지정
- 주소 부분에 실제 사용할 데이터의 유효 주소를 적기 때문에 주소 길이에 제약을 받음
- 기억 용량이 2^n 개의 Word인 메모리 시스템에서 주소를 표현하려면 nBit의 오퍼랜드부가 필요함
- 장점 : 데이터 인출을 위하여 한 번의 기억장치 액세스만 필요
- 단점 : 연산 코드를 제외하고 남은 비트들만 주소 비트로 사용될 수 있기 때문에 직접 지정할 수 있는 기억장소의 수가 제한

유효주소

컴퓨터가 실제로 사용할 데이터 (피연산자)가 저장된 주소, 즉 직접 주소지정방식은 주소부에 유효 주소를 표현

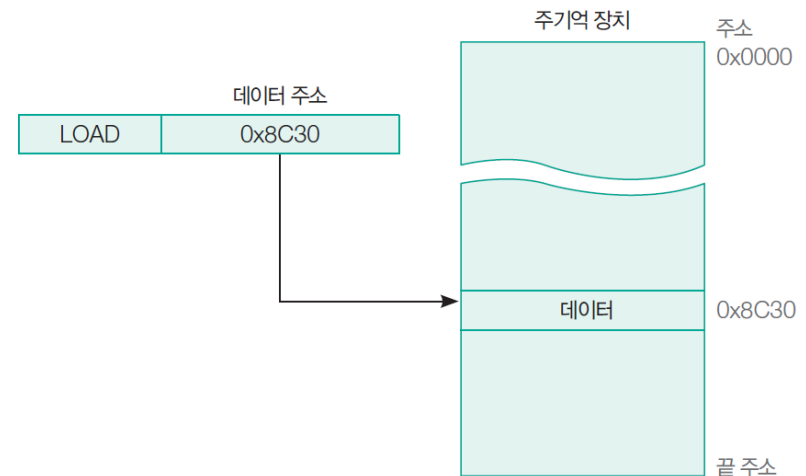


그림 4-14 직접 주소 지정

05 주소 지정 방식

CPU 내부 레지스터들과 주기억장치에 그림 2-24와 같은 값들이 저장되어 있다고 가정하자. 여기서, CPU 레지스터 및 각 기억 장소의 폭(width)은 16비트이며, 그림에서 모든 값들은 편의상 10진수로 표시하였다(그림 2-24는 [예제 2-7] 및 [예제 2-9]에서도 공통적으로 사용됨).

CPU 레지스터		주소	기억장치
PC	450	⋮	
		150	1234
IX	003	151	5678
BR	500		
R0		172	0202
R1	203	173	—
R2	151	⋮	
R3		⋮	

- (1) 직접 주소지정 방식을 사용하는 명령어의 주소 필드(A)에 저장된 내용이 150일 때, 유효 주소(EA) 및 그에 의해 인출되는 데이터를 구하라.
- (2) 명령어 길이가 16비트이고 연산 코드가 5비트라면, 이 명령어에 의해 직접 주소 지정 될 수 있는 기억장치 용량은 얼마인가?

3 간접 주소 지정(indirect addressing mode)

- 메모리 참조가 두 번 이상 일어나는 경우
- 데이터를 가져오는 데 많은 시간 소요
- 명령어에 나타낼 주소가 명령어 내에서 데이터를 지정하기 위해 할당된 비트(Operand 부의 비트) 수로 나타낼 수 없을 때 사용하는 방식이다.
- 명령어 형식에 간접비트(I) 필요
 - 만약 $I = 0$ 이면, 직접 주소지정 방식
 - 만약 $I = 1$ 이면, 간접 주소지정 방식
- 장점 : 최대 기억장치 용량이 Word의 길이에 의하여 결정
- 단점 : 실행 사이클 동안에 두 번의 기억장치 액세스가 필요
 - 첫 번째 액세스: 주소 인출
 - 두 번째 액세스: 실제 데이터 인출

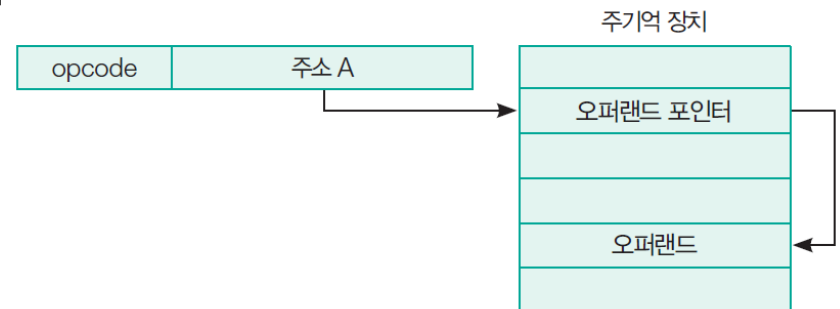
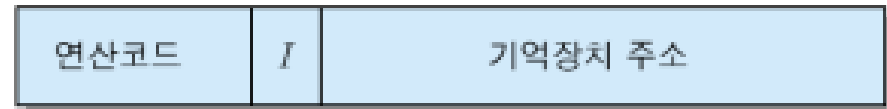


그림 4-21 간접 주소 지정

05 주소 지정 방식

CPU 내부 레지스터들과 주기억장치에 그림 2-24와 같은 값들이 저장되어 있다고 가정하자. 여기서, CPU 레지스터 및 각 기억 장소의 폭(width)은 16비트이며, 그림에서 모든 값들은 편의상 10진수로 표시하였다(그림 2-24는 [예제 2-7] 및 [예제 2-9]에서도 공통적으로 사용됨).

CPU 레지스터		주소	기억장치
PC	450	⋮	
		150	1234
IX	003	151	5678
BR	500		
R0		172	0202
R1	203	173	—

CPU 레지스터들과 주기억장치에 그림 2-24와 같은 값들이 저장되어 있을 때 아래 물음에 답하라.

- (1) 간접 주소지정 방식을 사용하는 명령어의 주소 필드(A)에 저장된 내용이 '172'라고 가정했을 때, 유효 주소(EA) 및 그에 의해 인출되는 데이터를 구하라.
- (2) 이 명령어에 의해 주소지정 될 수 있는 기억장치 용량은 얼마인가?

4 묵시적 주소 지정(implied addressing mode)

- 오퍼랜드의 소스나 목적지를 명시하지 않음
- 암묵적으로 그 위치를 알 수 있는 주소 지정 방식
- 예를 들어 서브루틴에서 호출한 프로그램으로 복귀할 때 사용하는 RET 명령
 - 명령어 뒤에 목적지 주소가 없지만 어디로 복귀할지 자동으로 알 수 있음
- PUSH, POP 등 스택 관련 명령어
 - 스택이라는 목적지나 소스가 생략
 - PUSH R1 : 레지스터 R1의 값을 스택에 저장
 - POP : 스택의 TOP에 있는 값을 AC로 인출
- 누산기를 소스나 목적지로 사용하는 경우도 생략 가능



Thank You
