컴퓨터구조
Computer Architecture

5장 - **제어 장치**

김현석

Contents

학습목표

- 제어 장치의 기능과 종류를 학습한다.
- 명령어 실행 사이클을 이해하고 동작 원리를 이해한다.
- 프로세서의 제어 순서와 실행을 이해한다.
- 파이프 라이닝에 대해 학습하고 장점과 해저드를 이해한다.
- 슈퍼 스칼라 구조에 대해 학습한다.

내용

01 제어 장치의 기능

❑ 컴퓨터의 기본 구성

- **제어 장치** : 컴퓨터의 모든 동작을 제어하는 CPU의 핵심 장치
- ALU, I/O 장치에 프로세서가 전송한 명령어 수행
- 주기억 장치의 명령어를 읽어 CPU의 명령 레지스터 IR로 가져오고,
- 명령 레지스터의opcode를 해독하여 제어 신호를 발생

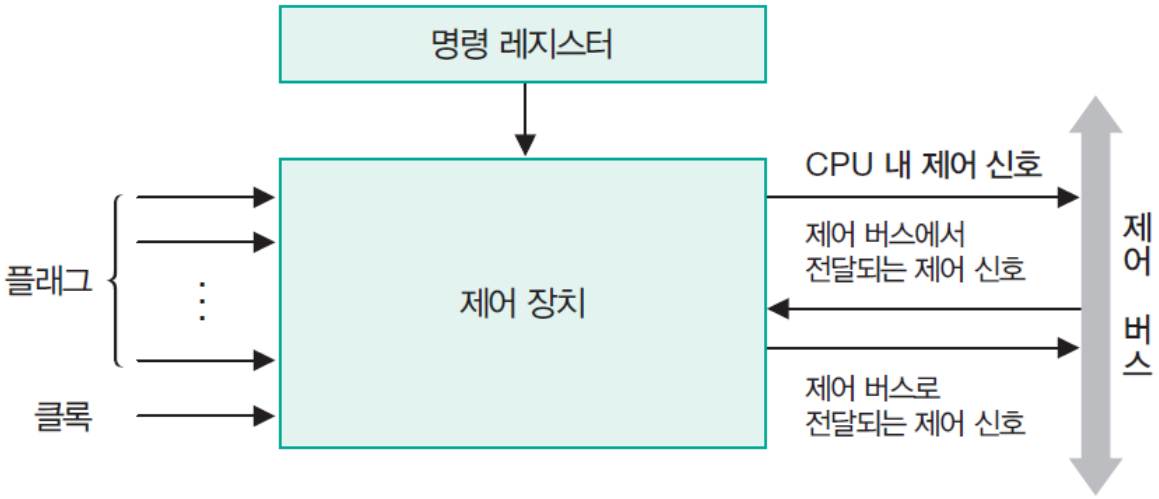
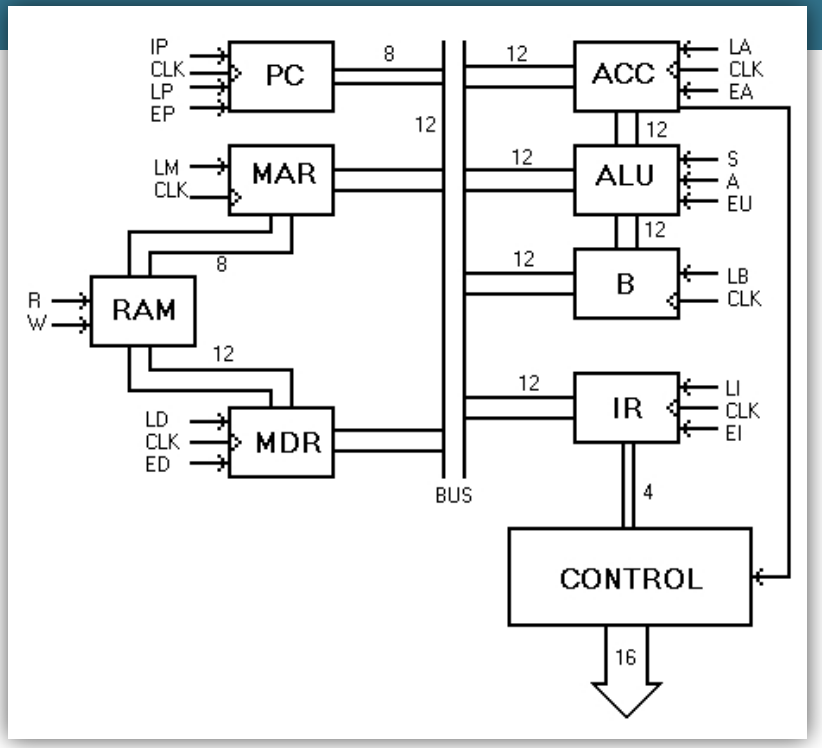
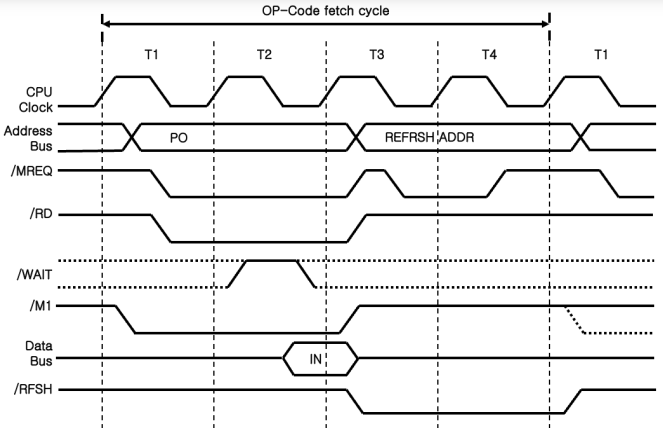


그림 5-1 제어 장치의 기능





01 컴퓨터 시스템의 구성

❑ 제어 장치의 기본 기능

- CPU에 접속된 장치들에 대한 데이터 이동 순서 조정
- 명령어 해독
- CPU 내 데이터 흐름 제어
- 외부 명령을 받아 **일련의 제어 신호 생성**
- 실행 장치(예를 들어 ALU, 데이터 버퍼, 레지스터) 제어
- 명령어 인출, 명령어 해독, 명령어 실행 등을 순서에 맞추어 처리

일련의 제어신호 동작

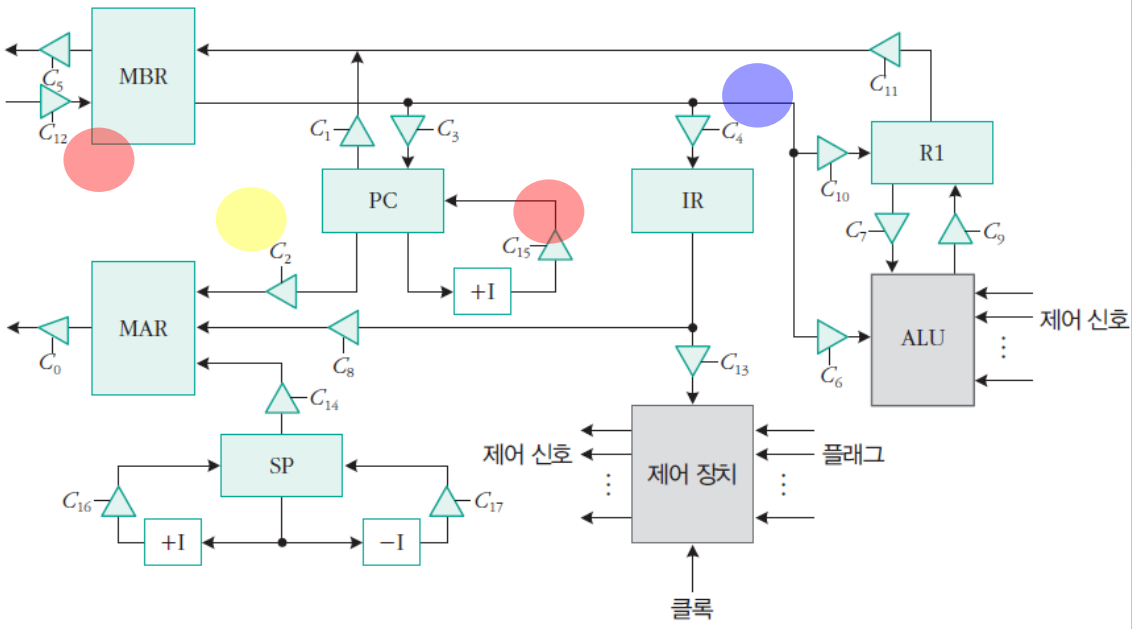


그림 5-21 데이터 경로와 제어 신호

명령어 사이클	마이크로 연산	제어 신호 동작
명령어 인출	t_1: MAR ← (PC)	C_2
	t_2: MBR ← M[MAR], PC ← (PC) + I	C_{12} CR, C_{15}
	t_3: IR ← (MBR)	C_4
명령어 해독	t_1: CU ← (IR:opcode)	C_{13}
	t_2: 명령어 해독(CU)	제어 신호들
명령어 실행: 서브루틴 호출	t_1: MAR ← (SP), MBR ← (PC)	C_{14}, C_1
	t_2: M[MAR] ← (MBR), SP ← SP − I	C_5, CW, C_{17}
	t_3: PC ← (IR:X)	C_3
인터럽트	t_1: MAR ← (SP), MBR ← (PC)	C_{14}, C_1
	t_2: M[MAR] ← (MBR), SP ← (SP) − I	C_5, CW, C_{17}
	t_3: PC ← ISR_Address	C_3

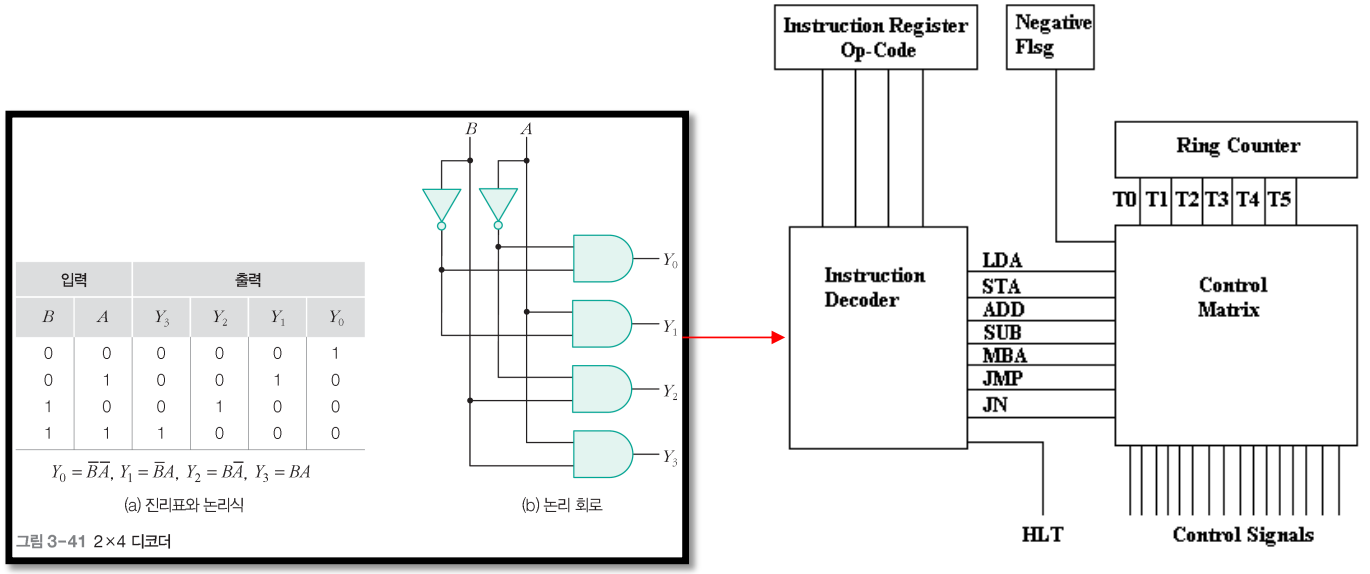
CR: Read control signal to system bus, CW: Write control signal to system bus
ISR: Interrupt Service Routine, SP: Stack Pointer, CU: Control Unit

하드와이어 제어 장치
(hardwired control)

02 제어 장치의 종류

1 하드와이어 제어 장치 (hardwired control)

- **논리 회로**로 만들어진 **하드웨어**로 명령어 실행 제어에 필요한 제어 신호 발생
- 회로 구조를 물리적으로 변경하지 않고는 신호 생성 방법을 수정할 수 없음
- 명령어의 opcode에 제어 신호를 생성하는 기본 데이터 포함
- 명령 디코더에서 명령 코드가 해독
- 명령 디코더는 명령어 opcode에 정의된 여러 필드를 해독하는 많은 해독기 세트로 구성



입력		출력			
B	A	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

$Y_0 = \overline{B}\overline{A},\ Y_1 = \overline{B}A,\ Y_2 = B\overline{A},\ Y_3 = BA$

(a) 진리표와 논리식

(b) 논리 회로

그림 3-41 2×4 디코더

02 제어 장치의 종류

- 제어 하드웨어는 **상태 기계**(state machine)처럼 클록 사이클이 진행됨에 따라 상태가 변함
- 명령 레지스터, 상태 코드, 외부 입력 등에 따라 다르게 변함
- 프로그래밍 가능한 논리 배열(PLA; Programmable Logic Array)과 유사한 방식으로 구성
  - 논리식으로 설계한 고정된 논리 회로에 의해 제어 신호 생성
  - 하드와이어 제어 장치는 마이크로 프로그램 제어 장치보다 빠름
  - 고속으로 작동

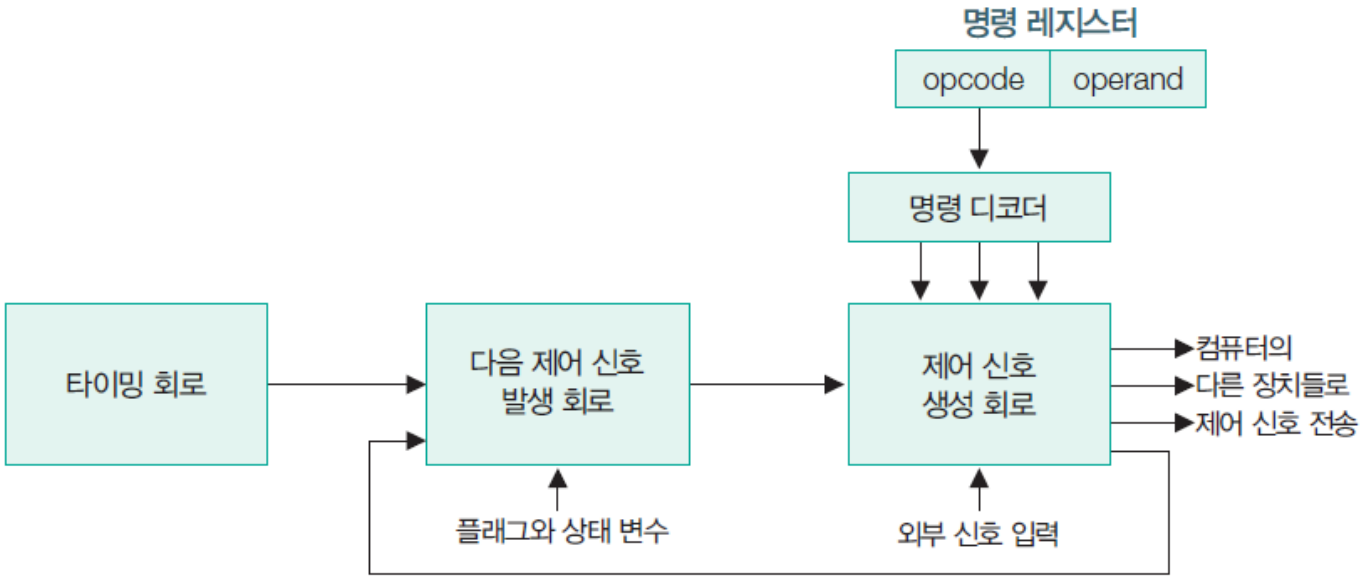
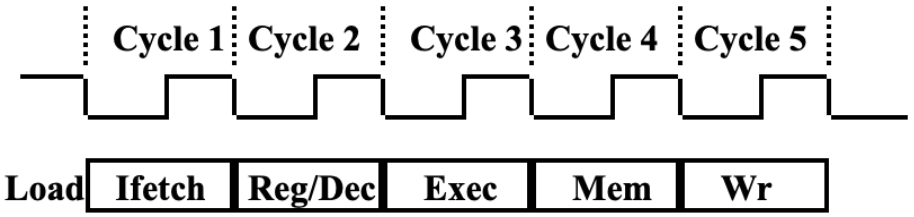


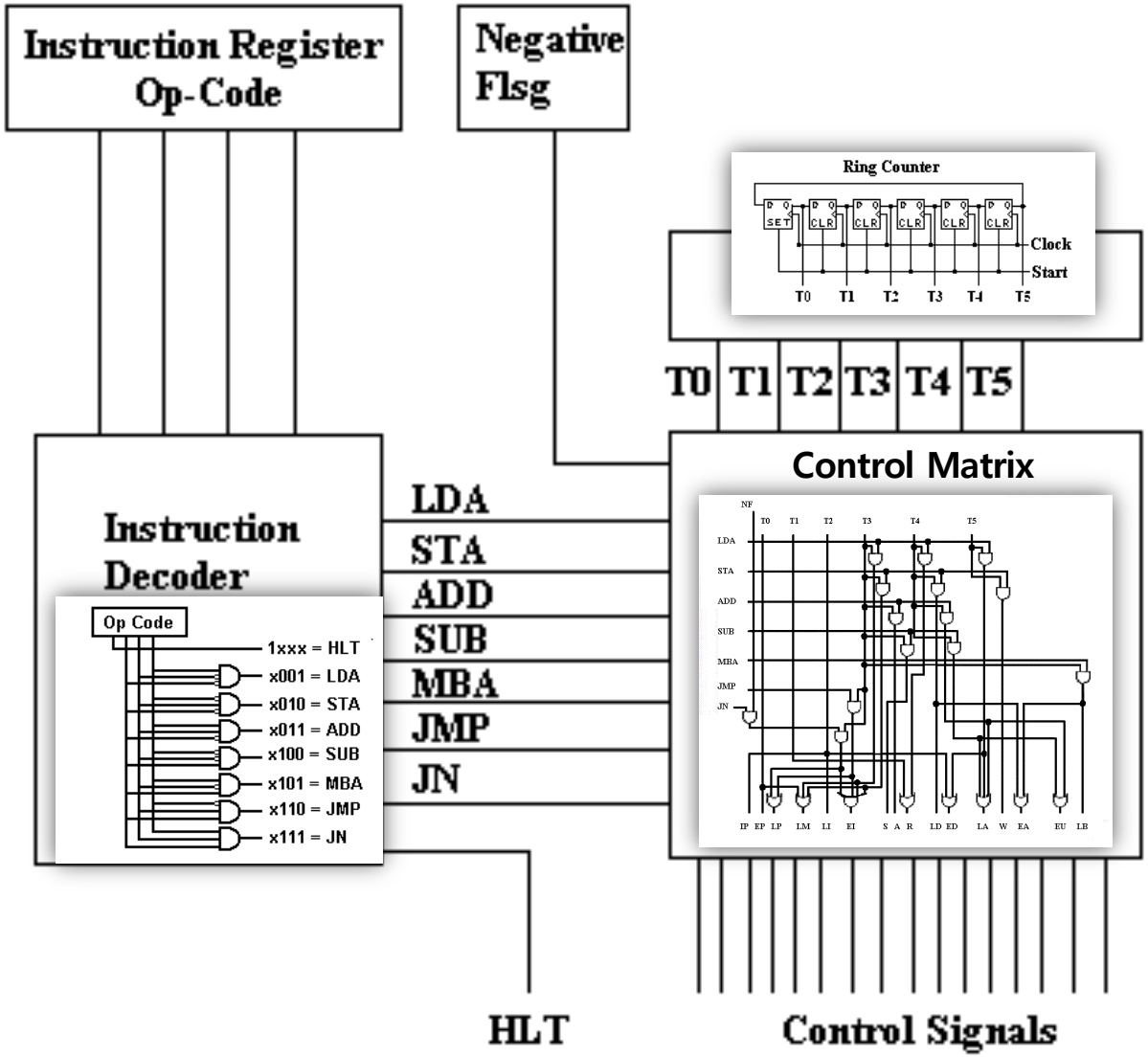
그림 5-2 하드와이어 제어의 개념

02 제어 장치의 종류

- 명령어 실행을 위한 제어 신호는 명령어 실행 사이클 전 구간에서 끊임없이 생성
- 인출 초기 : 새 명령이 제어 장치에 도착
- 명령어 해독 : 새 명령 실행과 관련된 첫 번째 상태 시작
  - 컴퓨터의 플래그 및 상태 정보가 변경되지 않은 상태로 유지되는 한 지속
  - 이들 신호 중 어떤 것이라도 변경이 일어나면 제어 장치 상태도 변경이 일어남
- 제어 신호 발생 회로에 대해 각각의 새 입력이 생성
  - 외부 신호가 나타날 때(예: 인터럽트) 외부 신호에 대한 반응(예: 인터럽트 처리)과 관련된 상태가 됨
  - 컴퓨터의 플래그와 상태 변수 값은 명령어 실행 주기에 적합한 상태를 선택하는 데 사용
- 사이클의 마지막 상태 : 프로그램의 다음 명령을 가져오기 시작하는 제어 상태
  - 프로그램 카운터의 내용을 MAR로 보내고
  - 컴퓨터의 명령 레지스터에 대한 명령어를 읽음
  - 프로그램 실행 종료 명령 : 운영체제 상태로 돌아감



하드와이어 제어장치



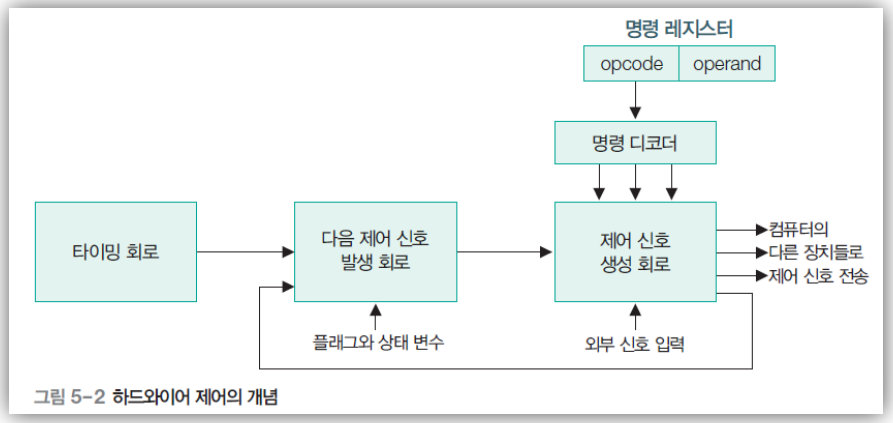


그림 5-2 하드와이어 제어의 개념

Hardwired Control Unit

논리식으로 설계한 고정된 논리 회로에 의해 제어 신호를 생성한다.

하드와이어 제어 장치는 마이크로 프로그램 제어 장치보다 빠르다.

하드와이어 제어 장치는 고속으로 작동한다.

마이크로 프로그램 제어 장치
(micro-programmed control)

02 제어 장치의 종류

2 마이크로 프로그램 제어 장치 (micro-programmed control)

- 하드와이어 제어 장치와 근본적인 차이: 제어 메모리 control memory의 존재 여부
- 각 명령 실행 순서에 해당하는 일련의 해독된 제어 신호를 비트 패턴으로 만들어 제어 메모리에 저장
- 명령 인출과정은 하드와이어 제어 장치와 동일
- 그러나 각 명령의 opcode가 제어 신호를 생성하기 위해 제어 메모리에서 해당 마이크로 프로그램의 시작 주소 지시
  - 명령 레지스터의 opcode가 제어 메모리의 제어 주소 레지스터 CMAR로 전송되고,
  - 마이크로 프로그램의 첫 번째 마이크로 명령이 마이크로 명령 레지스터로 읽힌다.
  - 마이크로 명령 : 인코딩된 형태로 연산코드 제공
  - 마이크로 명령 해독기에서 연산 코드 필드 해독

Micro-Programmed Control Unit

일종의 CPU용 펌웨어(Firmware)
명령어에 해당되는 CPU 제어 소프트웨어 로직
OS 업데이트시 BIOS 업데이트와 함께 제공

Microroutine Name (Mnemonic)	Address-ROM Address (Op-code)	Micro-Instruction Address	Control Signals: ILELRWLELELEASEL PPPM DDIIAA UB
"Fetch"	0	00	0011000000000000
		01	0000100000000000
		02	1000000110000000
LDA	1	03	0010000001000000
		04	0000100000000000
		05	0000000100100000

Instruction Mnemonic	Op-Code	Execution Action	Register Transfers	Ring Pulse	Active Control Signals
LDA (Load ACC)	1	ACC<--(RAM)	1. MAR <-- IR	3	EI, LM
			2. MDR <-- RAM(MAR)	4	R
			3. ACC <-- MDR	5	ED, LA

02 제어 장치의 종류

- 마이크로 명령 : 다음 마이크로 명령의 주소 포함
- 하나의 제어 필드는 마이크로 명령 주소 발생 장치를 제어하는 신호로 사용
- 마이크로 명령 주소 발생 장치 : 마이크로 명령 루틴의 시작 주소 생성
- 주소 발생 장치는 사상 함수에 opcode를 연결하여 루틴의 시작 주소를 생성
  - 사상 함수는 명령어 개수가 적고 체계적일 때는 효율적이지만, 명령어 개수가 많아지고 이전 버전 명령 세트에 계속적으로 추가될 때는 점점 더 복잡해짐
  - 주소 발생 장치는 사상 함수뿐만 아니라 덧셈이나 기타 다른 연산을 추가하여 주소를 찾아냄

02 제어 장치의 종류

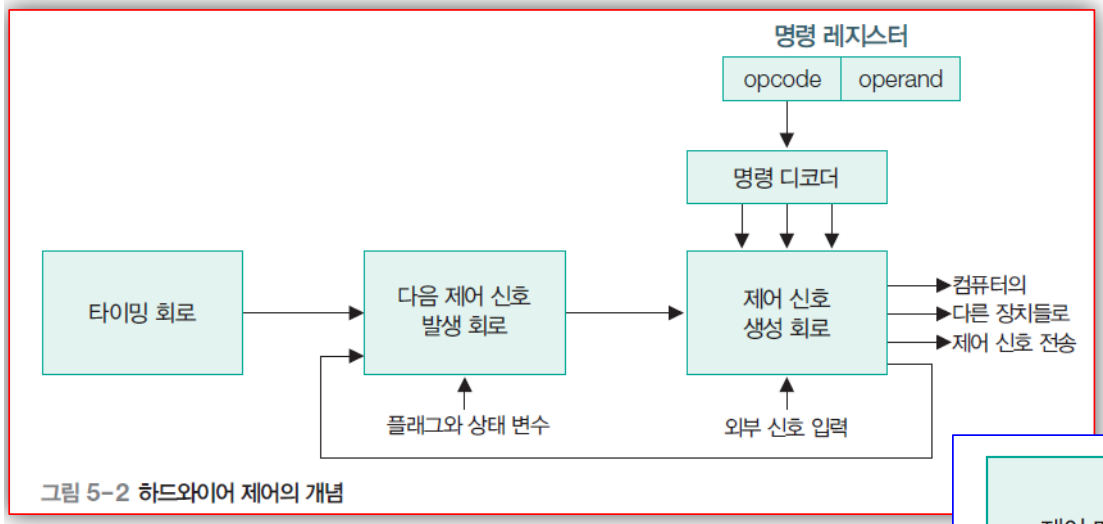


그림 5-2 하드와이어 제어의 개념

하드와이어 제어

마이크로 프로그램 제어

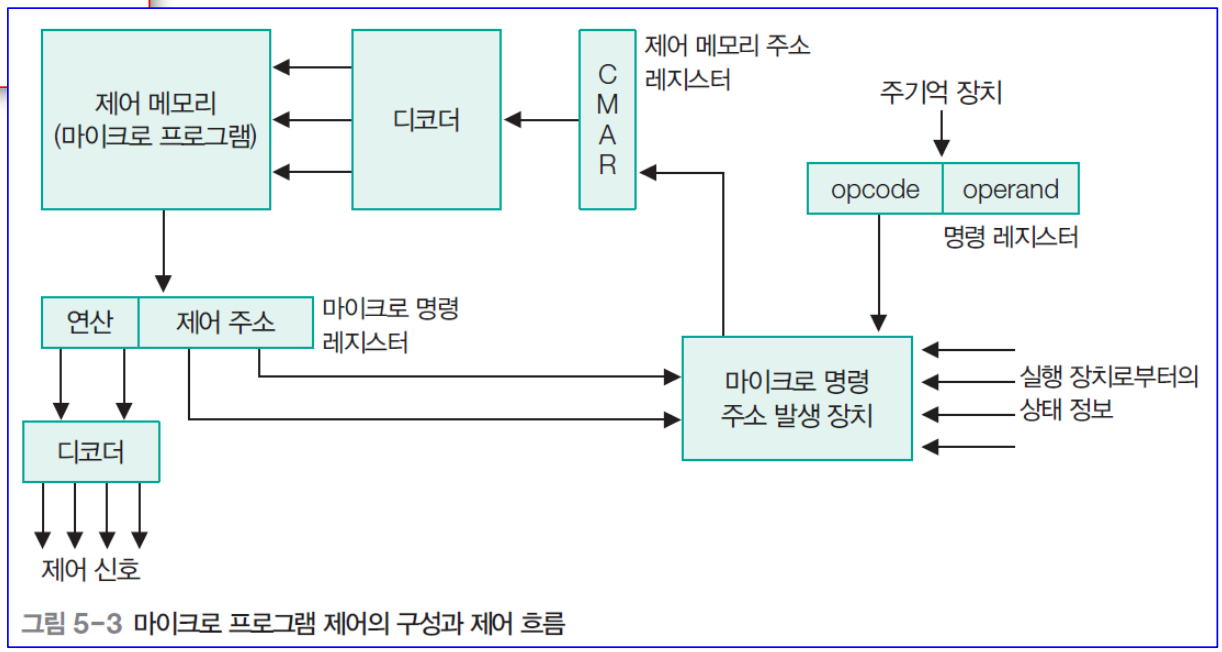


그림 5-3 마이크로 프로그램 제어의 구성과 제어 흐름

02 제어 장치의 종류

❖ 마이크로 명령 설계 순서

❶ 필요한 제어 신호의 목록 작성

❷ 유사한 의미를 가진 제어 신호를 그룹화하여 필드 구성

❸ 각 필드에 논리적인 순서 부여 : 예를 들어 ALU 연산과 ALU 피연산자를 앞에 두고 이어서 실행될 마이크로 명령의 제어 메모리 주소를 뒤에 둔다

❹ 마이크로 명령 형식에 대한 기호 범례를 작성하여 필드 값의 이름과 제어 신호 설정 방법 작성

❺ 필드 폭을 최소화하기 위해 동시에 사용하지 않는 작업을 분리하여 인코딩

- 마이크로 명령을 설계한 후 이를 기반으로 명령어 사이클인 명령어 인출, 명령어 해독, 명령어 실행 각각에 대한 마이크로 프로그램 루틴 작성
- 명령어 실행(연산) 사이클은 모든 기계어 명령마다 다르므로 일일이 정의

02 제어 장치의 종류

- 제어 신호는 프로그래머가 접근할 수 없는 제어 메모리에 제어 워드로 저장
- 프로그램으로 생성되는 제어 신호는 기계어와 유사
- 제어 기억 장치에서 마이크로 명령을 읽어 오기 때문에 속도가 느림

표 5-1 주요 용어

용어	설명
제어 워드	개별 비트가 다양한 제어 신호를 나타내는 단어다.
마이크로 루틴	기계어 제어 순서에 해당하는 일련의 제어 워드는 해당 명령의 마이크로 루틴을 구성한다.
마이크로 명령	마이크로 루틴 내의 개별 제어 워드를 마이크로 명령이라고 한다.
마이크로 프로그램	일련의 마이크로 명령을 마이크로 프로그램이라 하고, ROM 또는 RAM에 저장되며 이 장소를 제어 기억 장치Control Memory, CM라고 한다.
제어 기억 장치	모든 기계어 명령에 대한 마이크로 루틴을 저장하는 특수 기억 장치를 제어 기억 장치라고 한다.

02 제어 장치의 종류

3 마이크로 프로그램 제어 장치의 종류

❑ 수평적 마이크로 프로그램

- 제어 신호가 제어 신호당 1비트로 해독된 2진 형식으로 표현
  - 예를 들어 프로세서에 제어 신호가 50개 있다면 50비트의 제어 신호 필요
  - 한번에 2개 이상의 제어 신호를 활성화할 수 있음
- 수평적 마이크로 프로그램의 특징
  - 제어 워드가 더 길다.
  - 병렬 처리 응용에 사용된다.
  - 더 높은 수준의 병렬 처리 가능 : 병렬성이 n이면 한 번에 n개의 제어 신호 활성화 된다.
  - 추가 하드웨어(디코더)가 필요치 않음 ⇨ 수직적 마이크로 프로그래밍보다 빠름

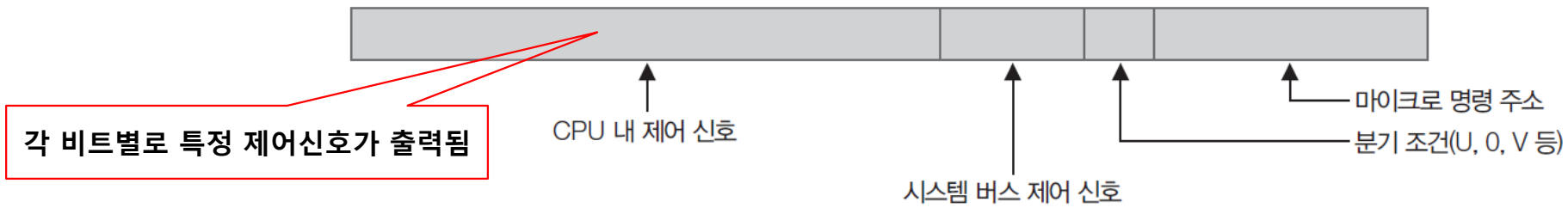


그림 5-4 수평적 마이크로 명령어의 구조 예

02 제어 장치의 종류

❑ 수직적 마이크로 프로그램

- 수직적 마이크로 프로그램에서는 제어 신호가 인코딩된 2진 형식으로 표시
- N개 제어 신호가 필요할 경우 $\lceil \log_2 N \rceil$비트 필요
- 수직적 마이크로 프로그램의 특징
  - 제어 워드가 더 짧다.
  - 유연하므로 새로운 제어 신호를 추가하기 용이
  - 낮은 수준의 병렬화 허용
  - 제어 신호를 생성하는 추가적인 하드웨어 필요 ⇨ 수평적 마이크로 프로그래밍보다 느림

예) 3비트면 2^3 = 8가지 제어신호

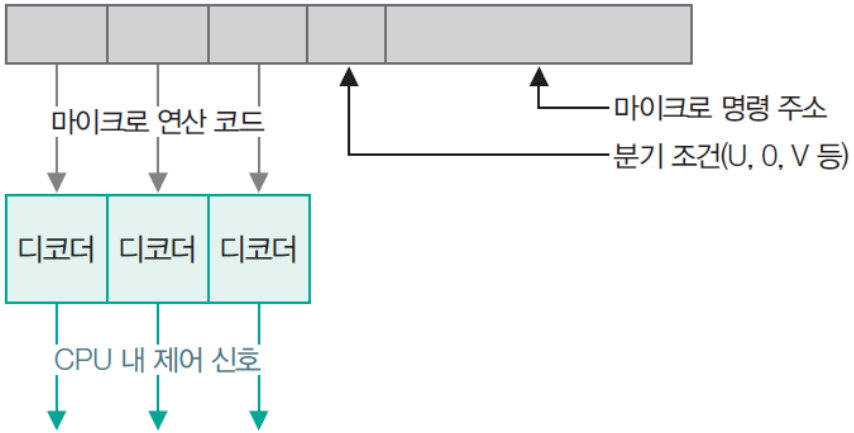


그림 5-5 수직적 마이크로 명령어의 구조 예

02 제어 장치의 종류

4 하드와이어 제어와 마이크로 프로그램 제어 비교

표 5-2 하드와이어 제어와 마이크로 프로그램 제어 비교

하드와이어 제어	마이크로 프로그램 제어
회로 기반 기술을 사용한다.	소프트웨어 기반 기술을 사용한다.
플립플롭, 게이트, 디코더 등을 사용하여 구현한다.	마이크로 명령이 명령의 실행을 제어하는 신호를 발생시킨다.
고정 명령 형식이다.	가변 명령 형식이다(명령당 16~64비트).
레지스터 기반 명령이다.	레지스터 기반이 아닌 명령이다.
ROM이 사용되지 않는다.	ROM이 사용된다.
RISC에서 주로 사용된다.	CISC에서 주로 사용된다.
해독이 빠르다.	해독이 느리다.
변경이 어렵다.	변경이 쉽다.
칩 영역이 작다.	칩 영역이 크다.

02 제어 장치의 종류

- 하드와이어 제어와 마이크로 프로그램 제어를 적절히 혼용하여 hybrid 제어 장치 구성

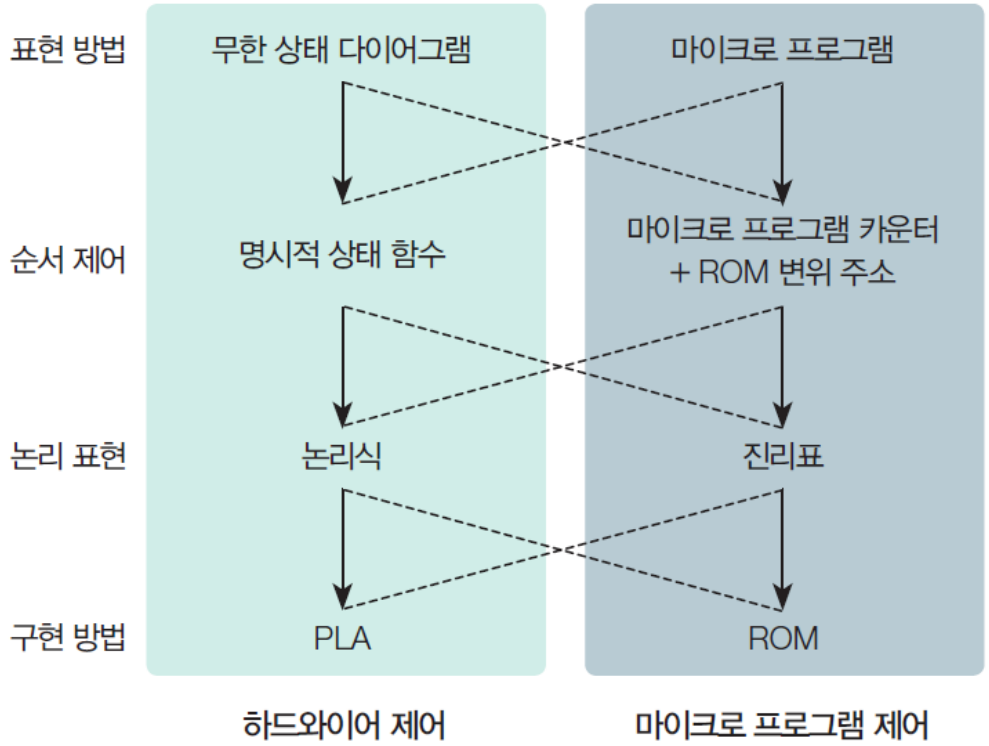
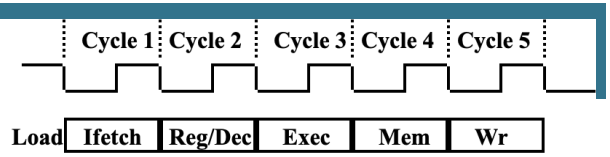


그림 5-6 하이브리드 제어의 구성 방법

명령어 사이클
(Instruction Cycle; Fetch→Decode→Execute)

03 명령어 사이클



❑ 명령어 사이클

- **명령어 인출** → **명령어 해독** → **명령어 실행** 사이클로 진행
- 명령어 사이클: 데이터 경로(data-path) 사이클이라고도 함
- 인터럽트 사이클
  - 인터럽트 사이클은 매 명령어 사이클이 끝나고 인터럽트 유무를 점검
  - 인터럽트가 있으면 인터럽트 처리 루틴 실행

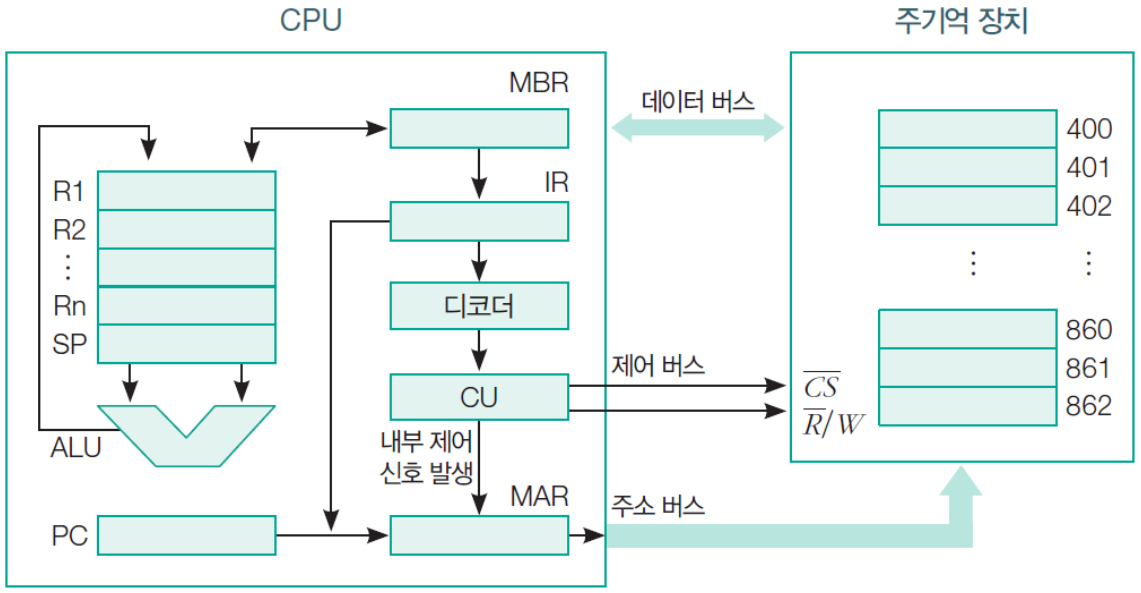


그림 5-7 CPU와 기억 장치의 제어 및 데이터 흐름

03 명령어 사이클

❑ 명령어 사이클(계속)

- 하나의 기계어 명령 : 일련의 마이크로 명령으로 구성된 명령어 사이클을 이루어 실행
- 각 명령어 사이클은 여러 개의 작은 단위로 구성
  - 명령어 인출, 명령어 해독, 명령어 실행, 인터럽트로 구별
- 제어 장치를 설계하려면 더 작은 단위의 마이크로 연산으로 분할(그림 5-8)
  - 각 명령은 더 짧은 하위 사이클(예 : 명령어 인출, 명령어 해독, 명령어 실행, 인터럽트)로 구성된 명령 주기 동안 실행
  - 각 하위 사이클은 하나 이상의 마이크로 연산을 가짐
  - 마이크로 연산은 프로세서의 가장 작은 동작

03 명령어 사이클

❖ 명령어 사이클 흐름도

Instruction Mnemonic	Op-Code	Execution Action	Register Transfers	Ring Pulse	Active Control Signals
LDA (Load ACC)	1	ACC<--(RAM)	1. MAR <-- IR	3	EI, LM
			2. MDR <-- RAM(MAR)	4	R
			3. ACC <-- MDR	5	ED, LA
STA (Store ACC)	2	(RAM) <--ACC	1. MAR <-- IR	3	EI, LM
			2. MDR <-- ACC	4	EA, LD
			3. RAM(MAR) <-- MDR	5	W
ADD (Add B to ACC)	3	ACC <-- ACC + B	1. ALU <-- ACC + B	3	A
			2. ACC <-- ALU	4	EU, LA
SUB (Sub. B from ACC)	4	ACC <-- ACC - B	1. ALU <-- ACC - B	3	S
			2. ACC <-- ALU	4	EU, LA
MBA (Move ACC to B)	5	B <-- ACC	1. B <-- A	3	EA, LB
JMP (Jump to Address)	6	PC <-- RAM	1. PC <-- IR	3	EI, LP
JN (Jump if Negative)	7	PC <-- RAM if negative flag is set	1. PC <-- IR if NF set	3	NF: EI, LP
HLT	8-15	Stop clock			
"Fetch"		IR <-- Next Instruction	1. MAR <-- PC	0	EP, LM
			2. MDR <-- RAM(MAR)	1	R
			3. IR <-- MDR	2	ED, LI, IP

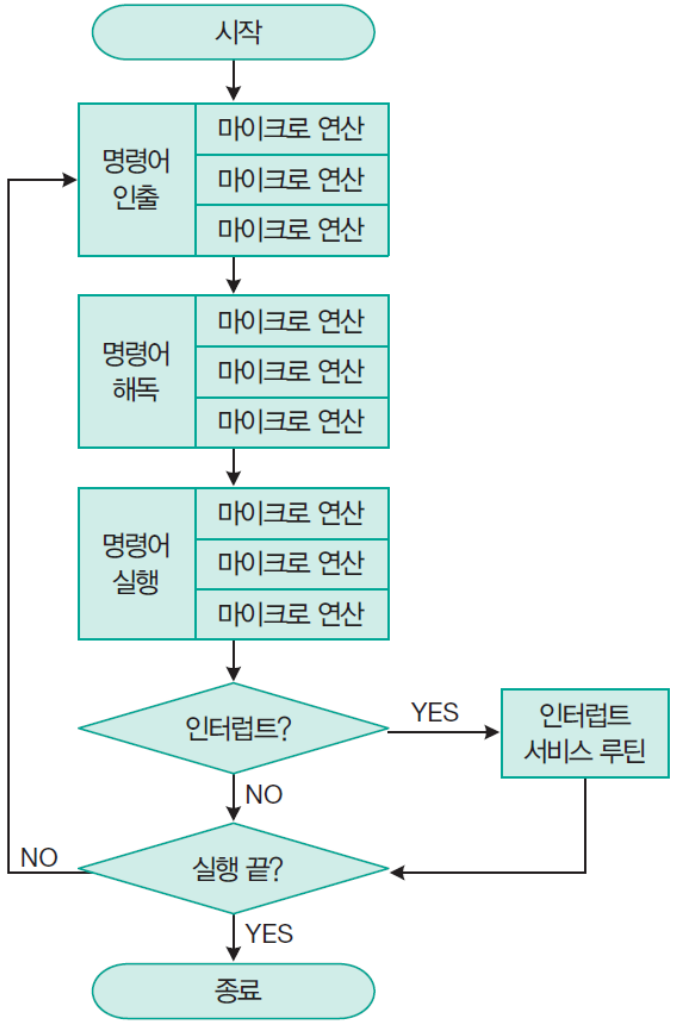
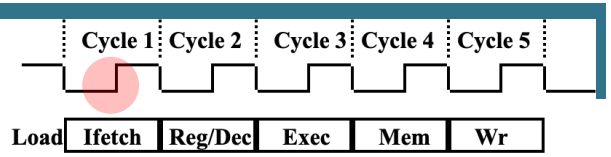


그림 5-8 명령어 사이클 흐름도

03 명령어 사이클



1 명령어 인출 사이클: instruction fetch

- 명령어 인출 사이클은 모든 명령어 실행의 첫 번째 단계
- 다음에 실행할 명령어를 주기억 장치에서 읽어 오는 과정

t_1 : PC ➔ MAR ➔ 주소 버스

t_2 : M[MAR] ➔ 데이터 버스 ➔ MBR

t_3 : MBR ➔ 명령 레지스터(IR)

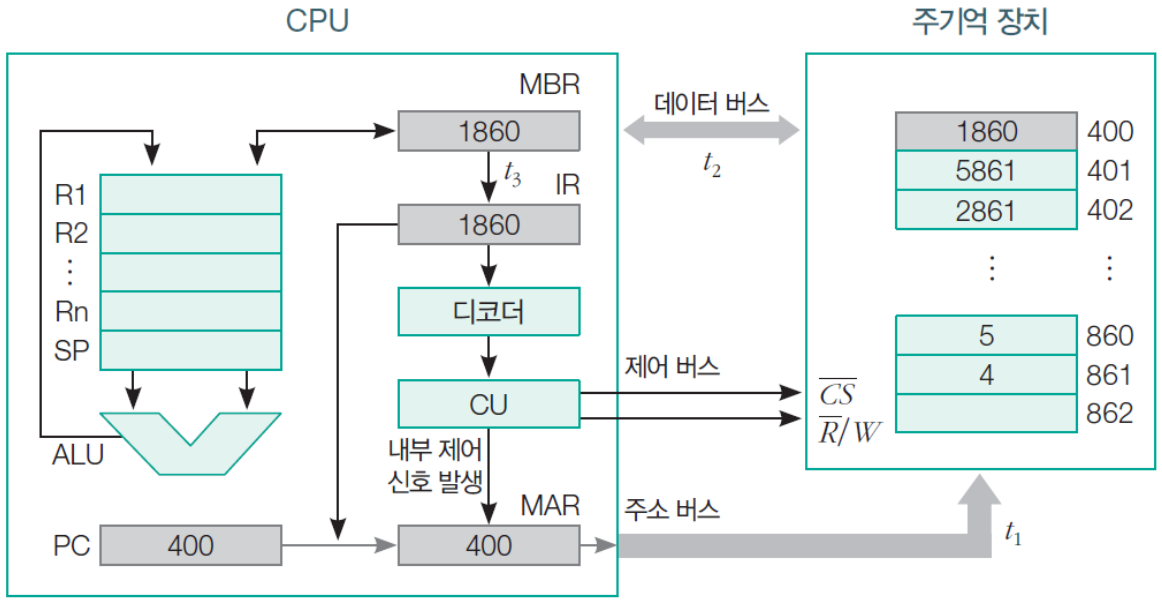


그림 5-9 명령어 인출 사이클의 마이크로 명령 실행 과정

03 명령어 사이클

- 명령어 인출 사이클이 진행되는 동안 프로세서의 레지스터 변화 과정

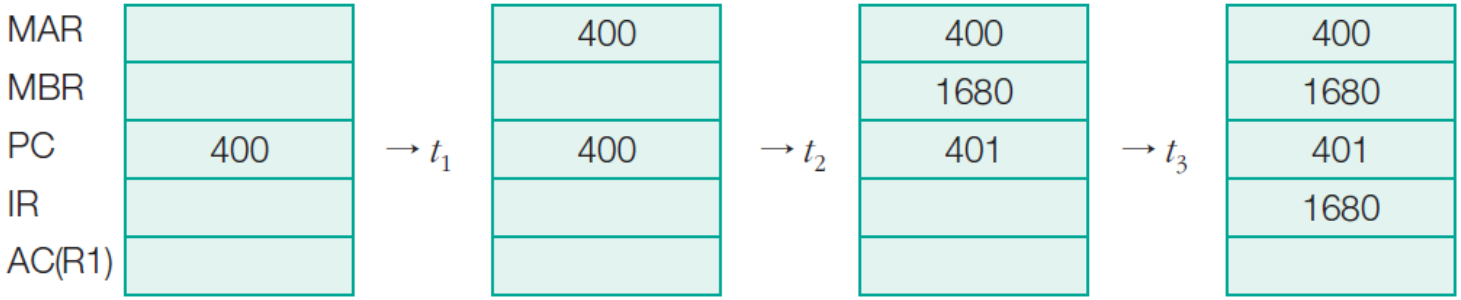


그림 5-10 명령어 인출 사이클의 진행에 따른 레지스터 변화

t_1: MAR ← (PC)

t_2: MBR ← M[MAR], PC ← (PC) + I

t_3: IR ← (MBR)

- I : 명령어의 크기
- 바이트 단위 주소 지정이고, 명령어의 기본 크기가 2 바이트라면 I는 2가 됨

03 명령어 사이클

- 마이크로 연산 $PC \leftarrow (PC) + I$는 t_2, t_3 중 어느 것과도 출동이 발생하지 않으므로 둘 중 어디에서 실행되어도 무관
- 마이크로 연산을 그룹으로 묶을 때는 다음 두 가지 간단한 규칙을 따라야 한다.

  ❶ 연산의 순서 준수

  (MAR ← (PC))는 반드시 MAR의 주소를 사용하기 때문에 (MBR ← 주기억 장치) 앞에 와야 함

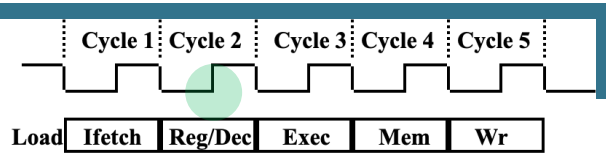
  ❷ 충돌을 피해야 함

  동시에 동일한 레지스터에서 읽고 쓰려고 해서는 안 됨

  예: 마이크로 연산 (MBR ← 주기억 장치)와 (IR ← MBR)은 동시에 실행되지 않아야 함

- 마이크로 연산 중 하나가 덧셈 연산 수행 : ALU의 기능과 프로세서 구조에 따라 덧셈 마이크로 연산을 ALU가 수행할 수도 있음

03 명령어 사이클



2 명령어 해독 사이클: instruction decode

- 명령어 해독 사이클 : 명령 레지스터IR의 내용 중 opcode만 해독기로 전달
- 해독기는 제어 기억 장치에서 명령 연산에 해당되는 마이크로 루틴을 찾아 해독
- 해독된 명령어에 대한 후속 마이크로 연산 발생

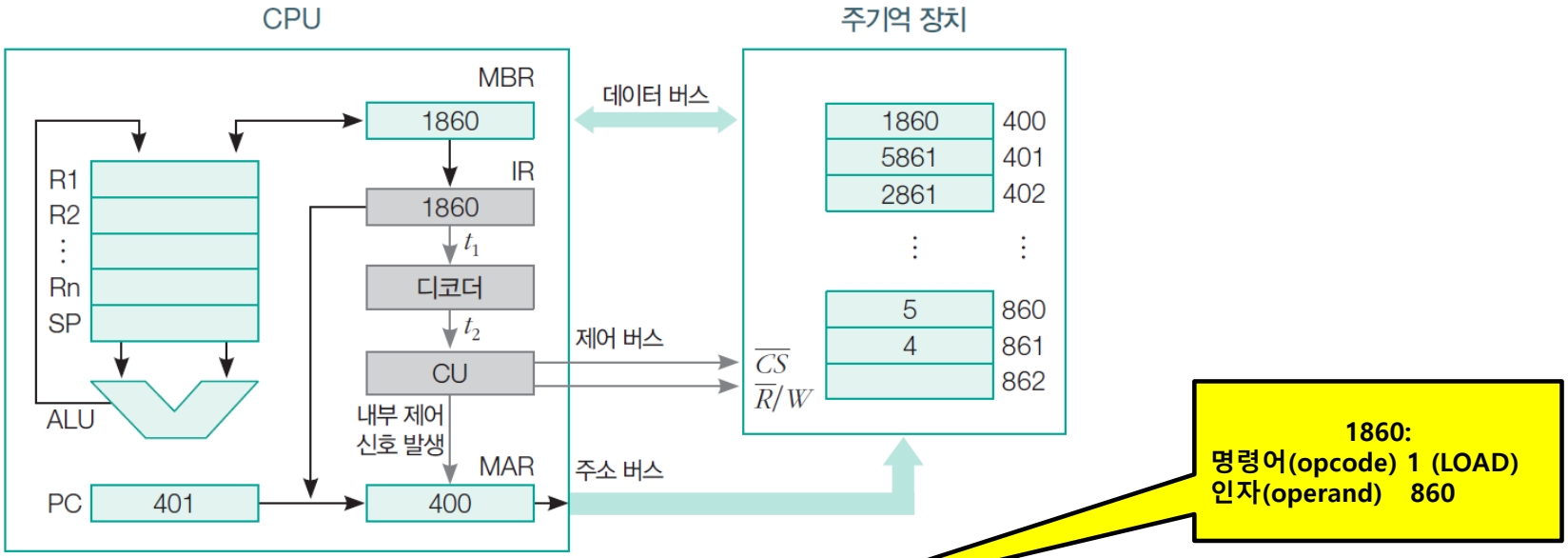
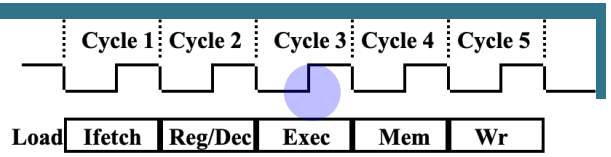


그림 5-11 명령어 해독 사이클의 마이크로 명령어 실행 과정

t_1: Decoder ← (IR:opcode) ; 여기서 opcode는 1이라고 가정한다.

t_2: Instruction Decoding ; CU가 명령어를 해독하여 제어 신호를 발생한다.

03 명령어 사이클



3 명령어 실행 사이클: instruction execute

- 명령어 실행 사이클: 해독된 명령어 실행 사이클
- 예: 데이터를 읽어서 레지스터 R1에 저장하는 명령어 실행 사이클
  - IR : operand ➔ MAR ➔ 주소 버스
  - M[MAR] ➔ MBR ➔ 데이터 버스
  - MBR ➔ R1

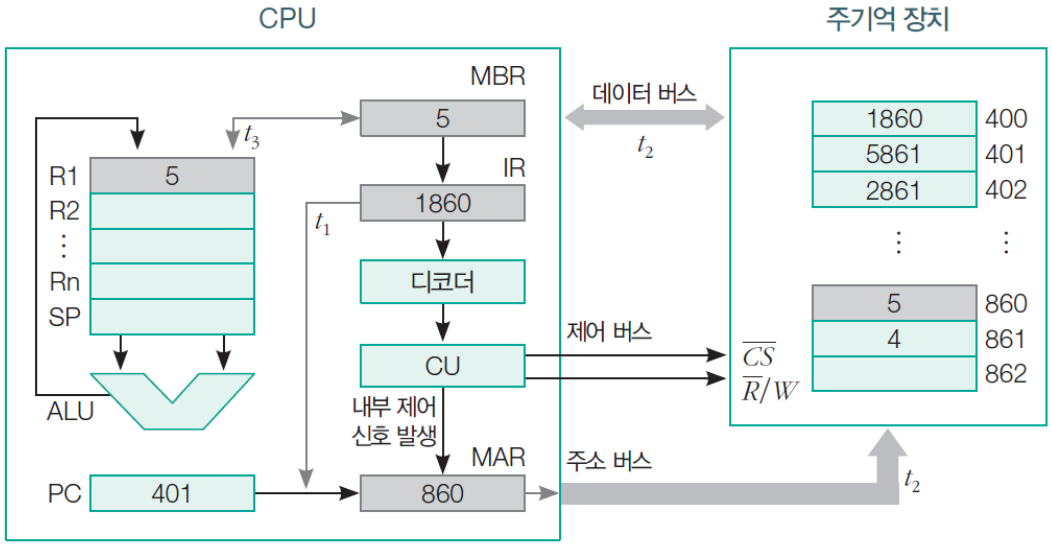


그림 5-12 명령어 실행 사이클의 마이크로 명령 실행 과정

t_1: MAR ← (IR:operand) ; 명령 레지스터의 오퍼랜드(860)를 MAR로

t_2: MBR ← M[MAR] ; 메모리에서 피연산자를 읽어 MBR로

t_3: R1 ← (MBR) ; MBR에서 R1(누산기 역할)로

03 명령어 사이클

- 명령어 인출 사이클이 진행되는 동안 프로세서의 레지스터 변화 과정

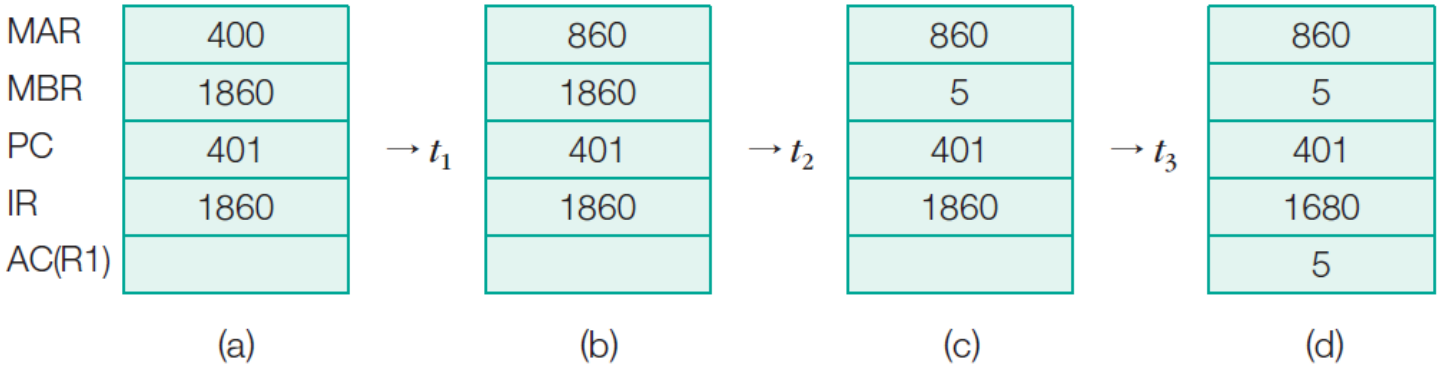


그림 5-13 명령어 실행 사이클의 진행에 따른 레지스터 변화

03 명령어 사이클

- 명령어 실행 사이클은 명령어 세트의 개수만큼 아주 다양하게 존재
- 명령 인출과정은 동일
- 레지스터 R1의 데이터와 메모리 X번지의 데이터를 ALU에서 더해 다시 R1에 저장

ADD R1, X

$$t_1: \text{MAR} \leftarrow (\text{IR:X})$$

$$t_2: \text{MBR} \leftarrow \text{M[MAR]}$$

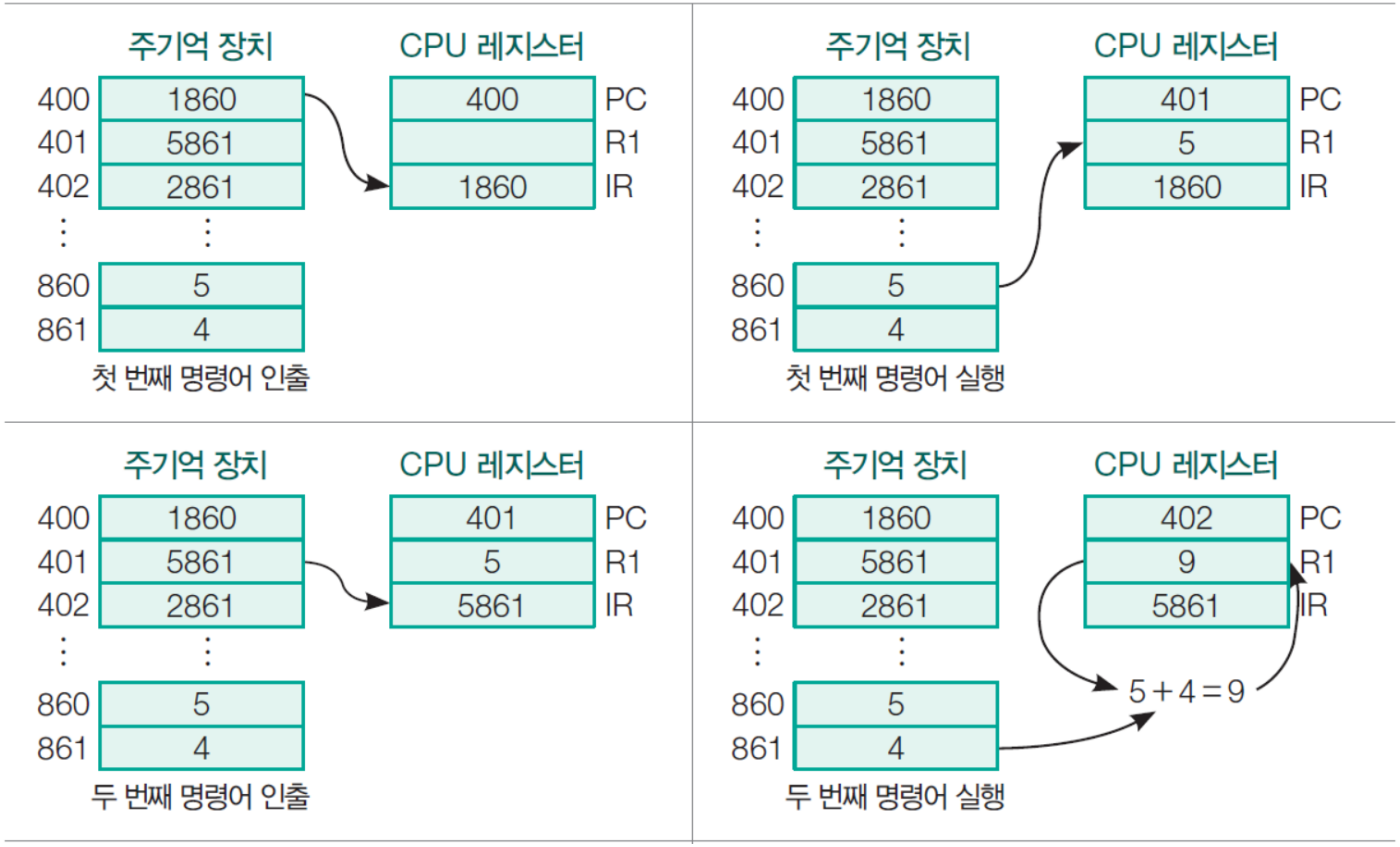
$$t_3: \text{R1} \leftarrow (\text{R1}) + (\text{MBR})$$

03 명령어 사이클

명령어(opcode)
1 – LOAD
5 - ADD
2 - STORE

❑ LOAD(1), ADD(5), STORE(2) 3개의 명령이 실행되는 과정

- CPU내의 PC, R1, IR 등의 레지스터 내용 변화 주목



03 명령어 사이클

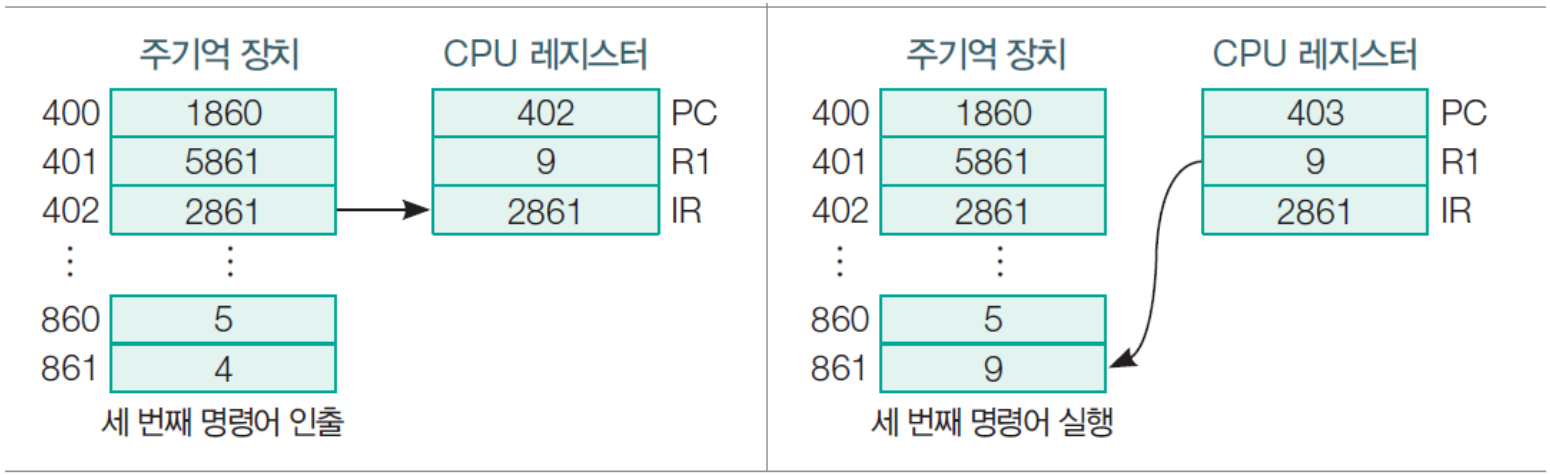


그림 5-14 LOAD, ADD, STORE가 실행되는 과정에서 레지스터 변화

03 명령어 사이클

❑ 예: ISZ(Increment and Skip if Zero)

- X 값을 1 증가시키고 그 결과가 0이면 바로 다음 명령을 건너뜀

ISZ X

$$t_1: \text{MAR} \leftarrow (\text{IR:X})$$

$$t_2: \text{MBR} \leftarrow \text{M[MAR]} + 1$$

$$t_3: \text{M[MAR]} \leftarrow (\text{MBR}),\ \text{If } ((\text{MBR}) = 0) \text{ then } (\text{PC} \leftarrow \text{PC} + \text{I})$$

03 명령어 사이클

❑ 예: BSA(Branch-and-Save-Address)

BSA X

t_1: MAR ← (SP), MBR ← (PC)

t_2: M[MAR] ← (MBR), SP ← SP−I

t_3: PC ← (IR:X)

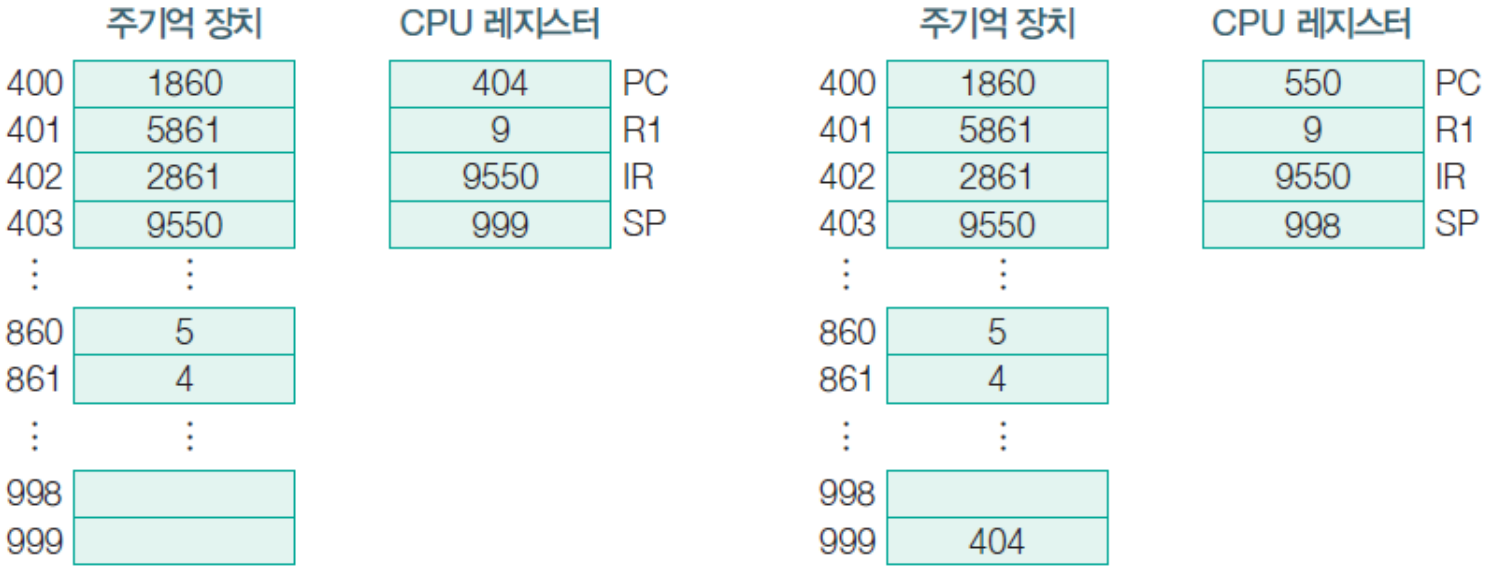


그림 5-15 서브루틴이 호출되는 과정에서 레지스터 변화

03 명령어 사이클

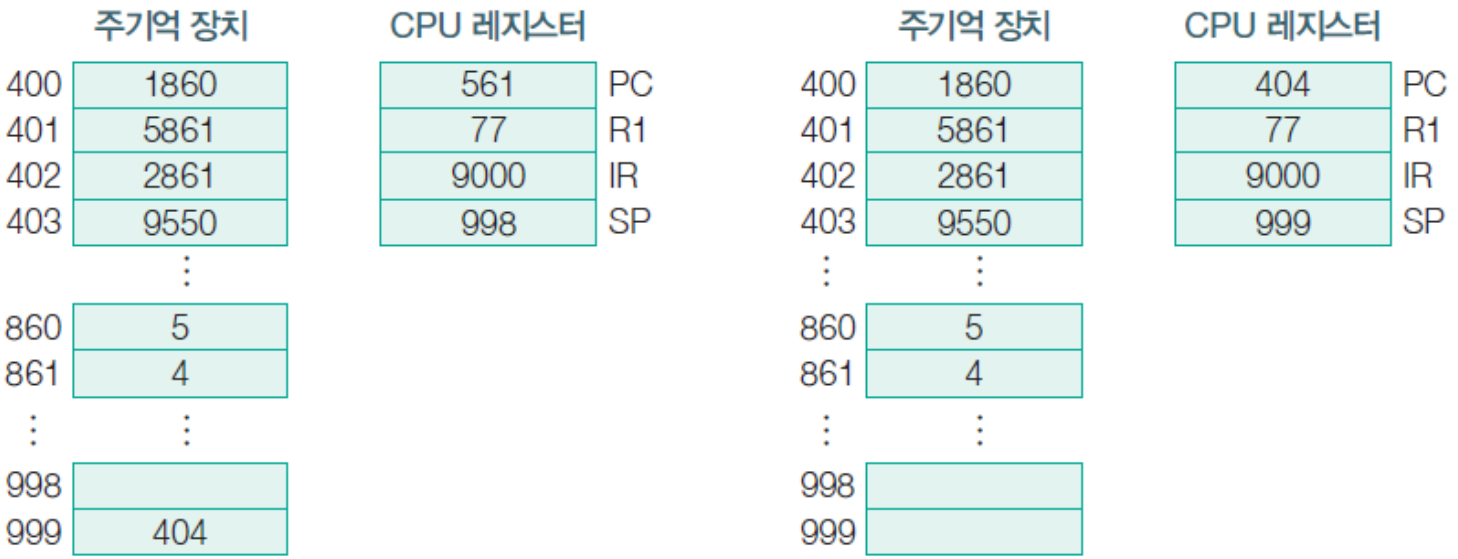
❑ **예: RET(return)**

t_1: SP ← SP + I

t_2: MAR ← (SP)

t_3: MBR ← M[MAR]

t_4: PC ← MBR



(a) 서브루틴 복귀 전 (b) 서브루틴 복귀 후

그림 5-16 서브루틴에서 복귀하는 과정에서 레지스터 변화

03 명령어 사이클

❑ 중첩 서브루틴이나 다중 서브루틴인 경우

- 매 서브루틴 호출마다 스택에 복귀할 주소(PC)를 저장하고,
- 복귀할 때는 스택 값을 꺼내어 PC로 가져온다.

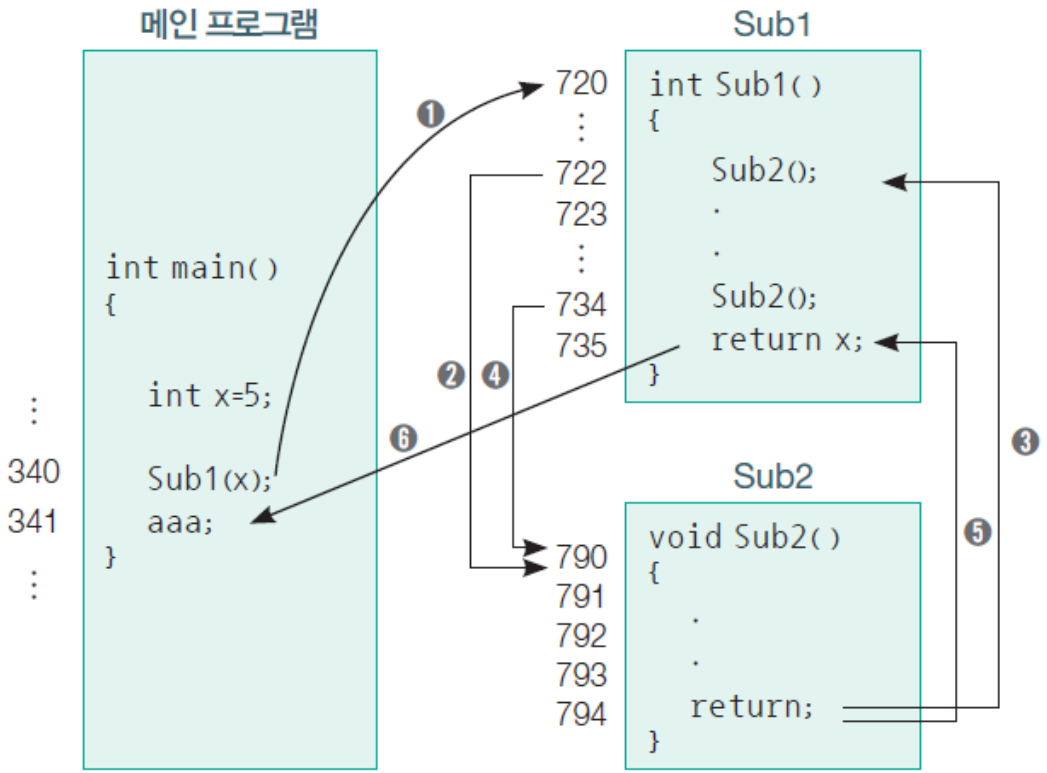


그림 5-17 다중 서브루틴이 호출되는 상황

03 명령어 사이클

❑ 다중 서브루틴 예에서 레지스터 변화

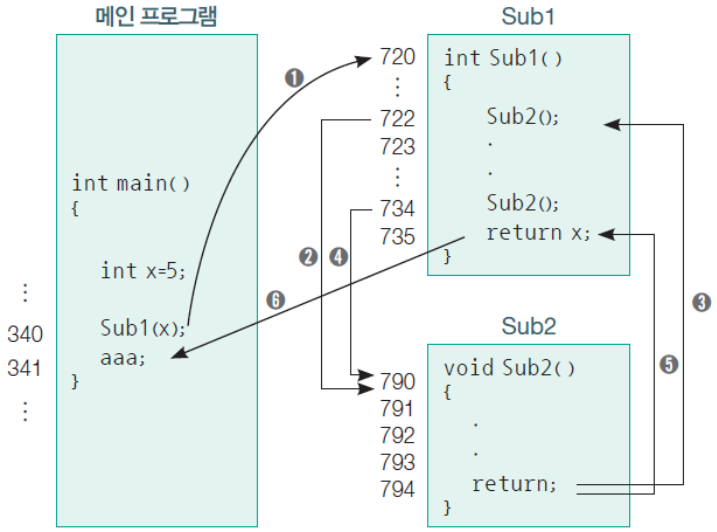


그림 5-17 다중 서브루틴이 호출되는 상황

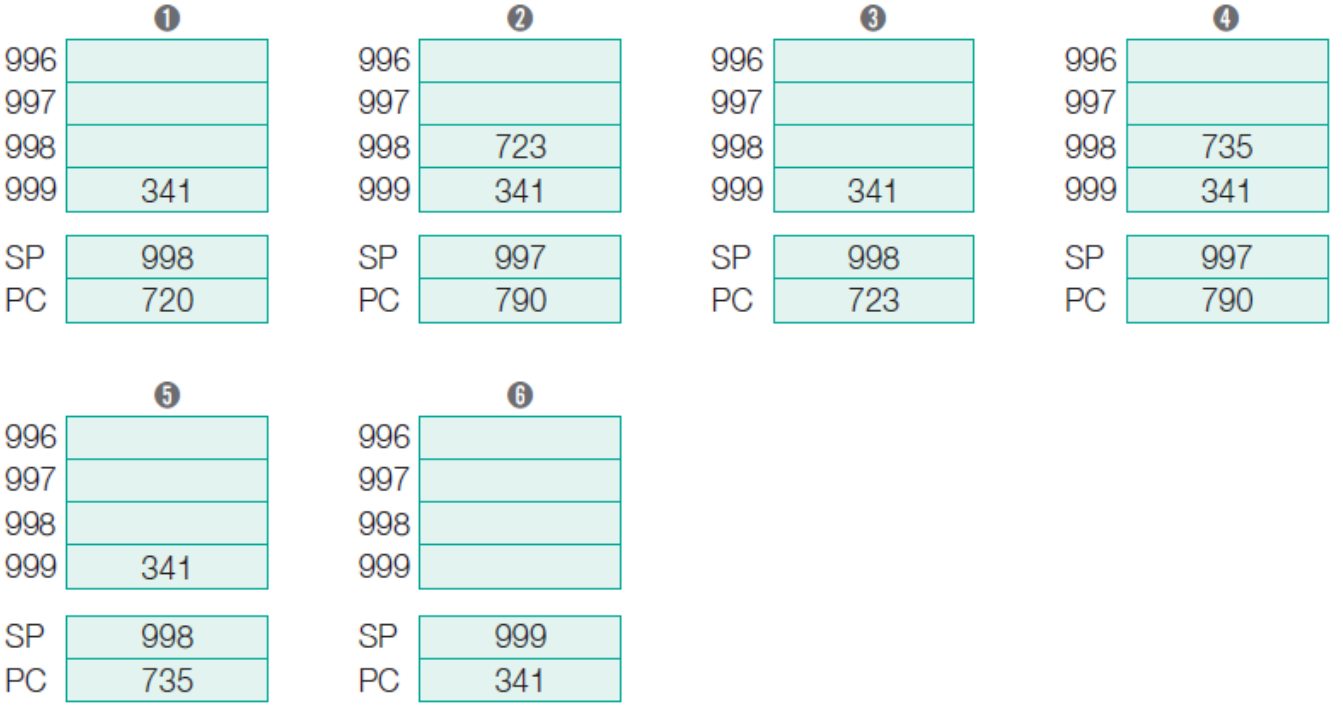


그림 5-18 다중 서브루틴이 호출되는 상황에서 레지스터 변화

t_1: MAR ← (SP), MBR ← (PC)

t_2: M[MAR] ← (MBR), SP ← SP − I

t_3: PC ← (IR:X)

BSA
(Branch-and-Save-Address)

t_1: SP ← SP + I

t_2: MAR ← (SP)

t_3: MBR ← M[MAR]

t_4: PC ← MBR

RET(return)

03 명령어 사이클

4 인터럽트 사이클: interrupt

- 실행 주기가 완료되면 인터럽트가 발생했는지 여부 점검
- 인터럽트가 발생했다면 인터럽트 사이클 실행
- 인터럽트 사이클은 통상적으로 다음과 같다.

$$t_1: \text{MAR} \leftarrow (\text{SP}),\ \text{MBR} \leftarrow (\text{PC})$$

$$t_2: \text{M[MAR]} \leftarrow (\text{MBR}),\ \text{SP} \leftarrow \text{SP} - \text{I}$$

$$t_3: \text{PC} \leftarrow \text{Interrupt\_Service\_Routine\_Address}$$

03 명령어 사이클

5 명령어 사이클

- 항상 명령어 인출, 명령어 해독, 명령어 실행 사이클 순서로 실행
- 인터럽트 사이클은 항상 명령어 실행이 끝난 후
  인터럽트가 있으면 실행하고,
  그렇지 않으면 다음 명령어 인출 사이클로 진행

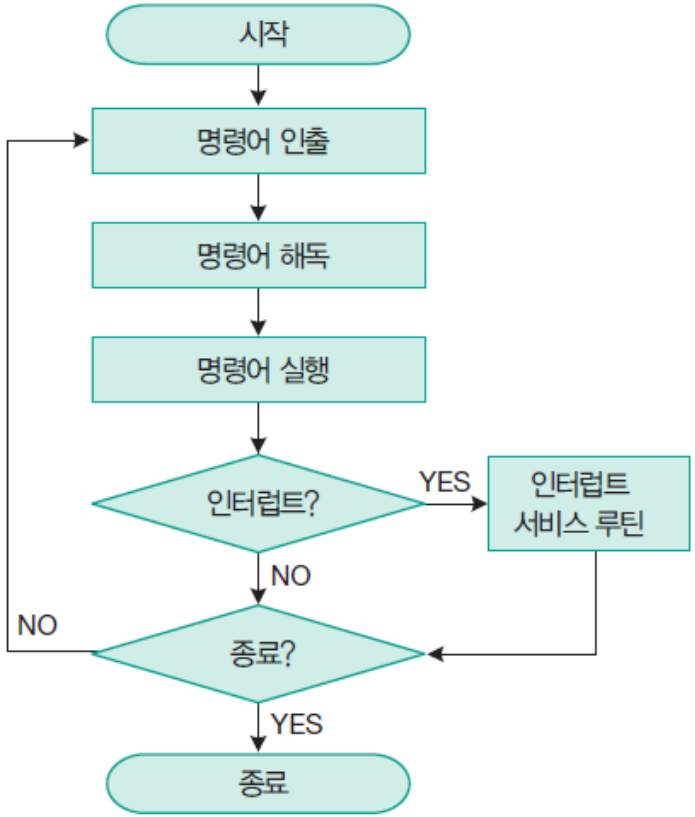


그림 5-19 명령어 사이클

04 프로세서 제어

❑ 제어 장치의 특성

❶ 프로세서의 기본 장치 정의

❷ 프로세서가 수행하는 마이크로 연산 나열

❸ 마이크로 연산을 할 수 있도록 제어 장치가 수행해야 할 기능 결정

❖ 모든 마이크로 연산은 다음 범주 중 하나임

- 한 레지스터에서 다른 레지스터로 데이터 전송
- 레지스터에서 외부 인터페이스(예 : 시스템 버스)로 데이터 전송
- 외부 인터페이스에서 레지스터로 데이터 전송
- 입력 및 출력 레지스터를 사용하여 산술 또는 논리 연산 수행

04 프로세서 제어

❖ **제어 장치는 다음 두 가지 기본 작업 수행**

- **순서** : 프로그램에서 정해진 순서와 그에 해당하는 마이크로 연산을 적절한 순서로 진행
- **실행** : 제어 장치는 각 마이크로 연산 수행

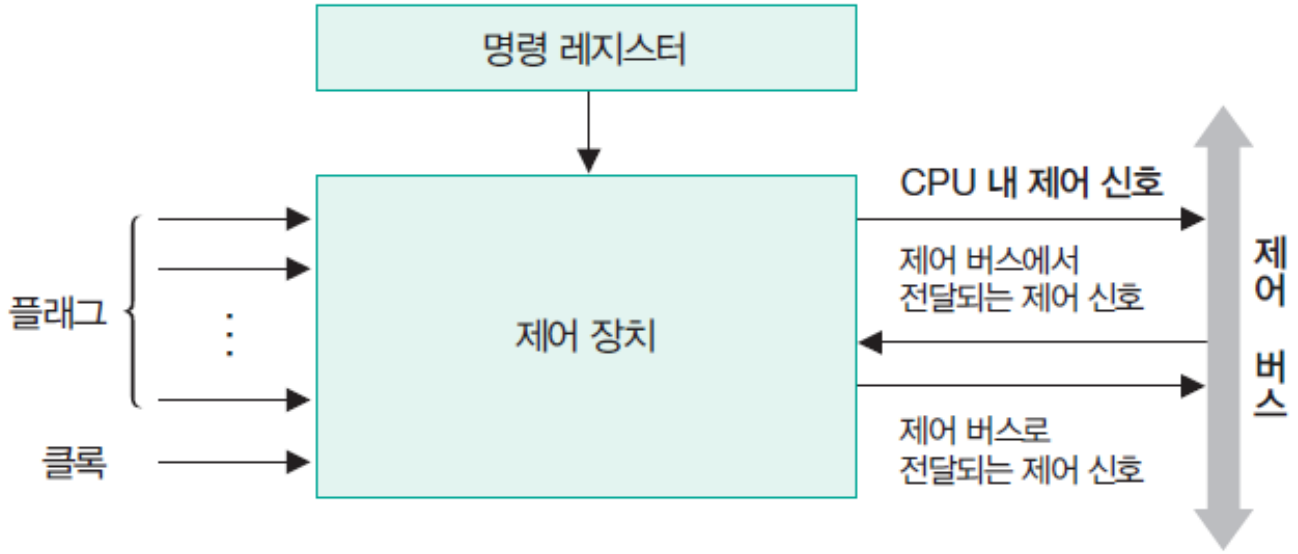


그림 5-20 제어 장치의 개념

04 프로세서 제어

❖ **제어 장치는 다음 제어 신호를 동시에 전송하여 명령 흐름 제어**

- 제어 신호는 MAR 내용을 주소 버스에 전달
- 제어 버스에 메모리 읽기 제어 신호 전송
- 데이터 버스 내용을 MBR에 저장할 수 있도록 제어 신호 전송
- PC 내용에 I를 더해 PC에 다시 저장하는 제어 신호

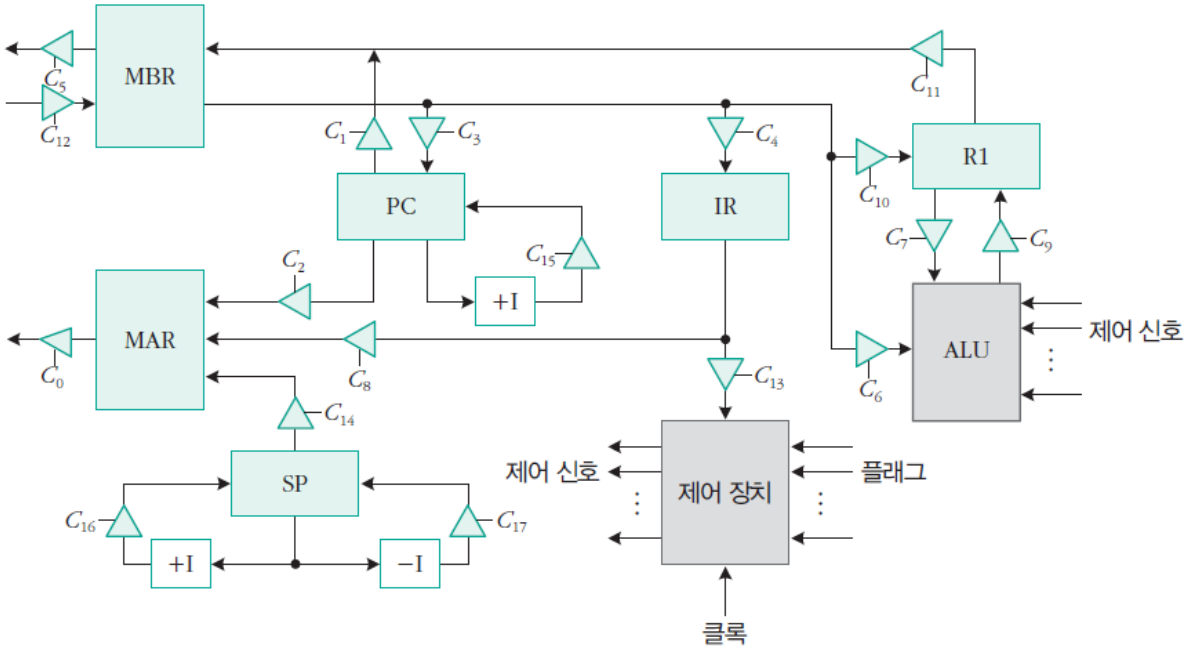


그림 5-21 데이터 경로와 제어 신호

04 프로세서 제어

❑ 데이터 경로

- 제어 장치는 내부 데이터 흐름 제어: 예) 명령어 인출을 할 때 MBR내용이 IR로 전송
- 제어할 각 경로에는 스위치 존재([그림 5-21]에 삼각형으로 표시).
- 제어 장치에서 나오는 제어 신호는 일시적으로 게이트를 열어 데이터 통과

❑ ALU

- 제어 장치는 일련의 제어 신호로 ALU 제어
- 이러한 신호는 ALU 내 다양한 논리 회로와 게이트 활성화

❑ 시스템 버스

- 제어 장치는 제어 신호를 시스템 버스의 제어선(예: 메모리R EAD)으로 전송

04 프로세서 제어

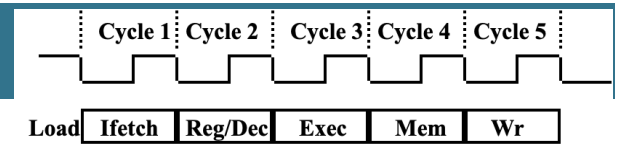
❖ 마이크로 연산에 필요한 제어 신호

명령어 사이클	마이크로 연산	제어 신호 동작
명령어 인출	t_1: MAR ← (PC)	C_2
	t_2: MBR ← M[MAR], PC ← (PC) + I	C_{12} CR, C_{15}
	t_3: IR ← (MBR)	C_4
명령어 해독	t_1: CU ← (IR:opcode)	C_{13}
	t_2: 명령어 해독(CU)	제어 신호들
명령어 실행: 서브루틴 호출	t_1: MAR ← (SP), MBR ← (PC)	C_{14}, C_1
	t_2: M[MAR] ← (MBR), SP ← SP – I	C_5, CW, C_{17}
	t_3: PC ← (IR:X)	C_3
인터럽트	t_1: MAR ← (SP), MBR ← (PC)	C_{14}, C_1
	t_2: M[MAR] ← (MBR), SP ← (SP) – I	C_5, CW, C_{17}
	t_3: PC ← ISR_Address	C_3

CR: Read control signal to system bus, CW: Write control signal to system bus
ISR: Interrupt Service Routine, SP: Stack Pointer, CU: Control Unit

파이프 라이닝;
동시에 여러 개의 명령어를 처리하는 기술

05 파이프 라이닝

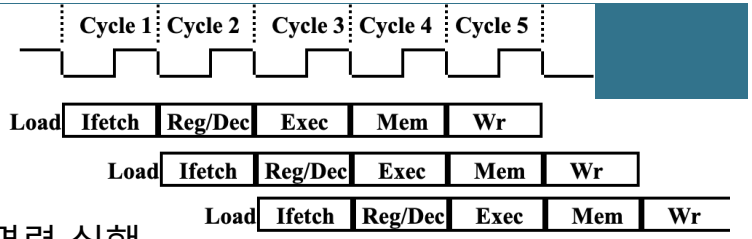


❑ 명령어 단계 병렬 처리

- 프로세서의 제어 장치는 기본적으로 '명령어 인출 → 명령어 해독 → 명령어 실행' 순서로 명령 수행
- 전통적인 프로세서에서는 다음과 같이 순차적으로 실행

그림 5-22 전통적 PC에서 명령어 실행(순차적 실행)

05 파이프 라이닝



❑ 명령어 단계 병렬 처리

- 현대 대부분의 프로세서에서는 파이프 라이닝(pipelining) 기술로 명령 실행
- 파이프 라이닝은 그림과 같이 명령 하나를 여러 단계로 나누어 각각을 독립적인 장치에서 동시에 실행하는 기술 → N단계 파이프라인
- 하나의 명령을 3단계로 나누어 실행 하는 예

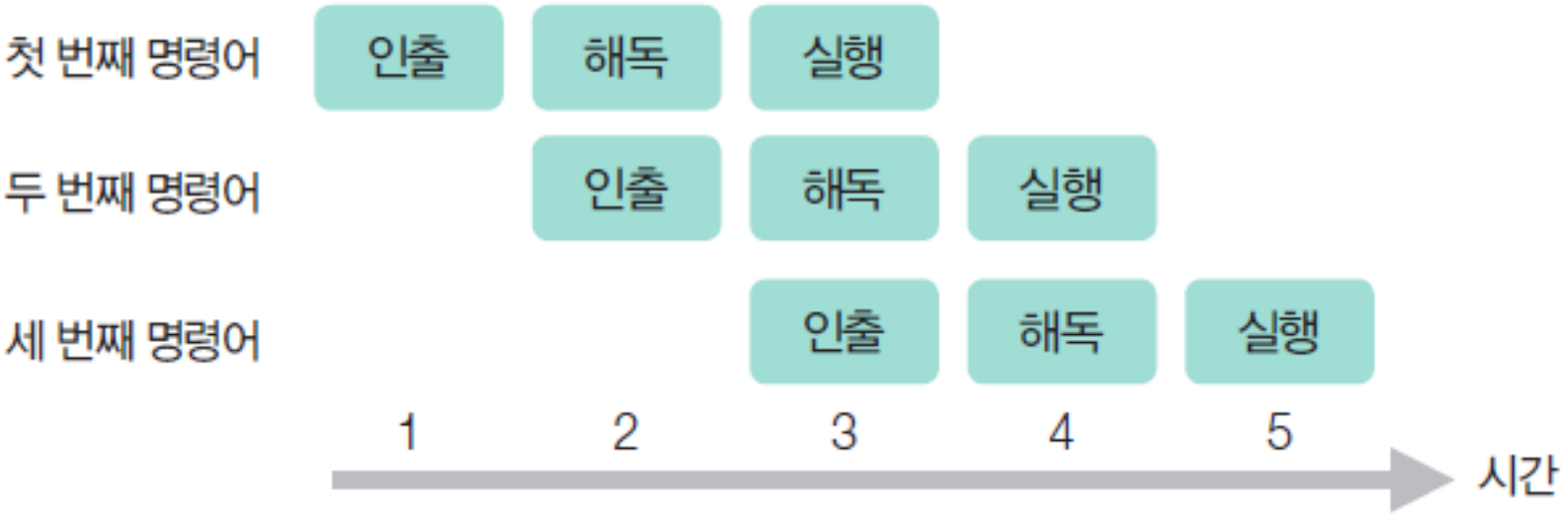


그림 5-23 파이프 라인닝 개념

05 파이프 라이닝

❑ 5단계 파이프라인

- 단계가 S1~S5로, 5단계 파이프 라인
- 1단계 : 메모리에서 명령어를 인출
- 2단계: 명령어를 해독하고 명령어 형태를 결정하며 필요한 피연산자 결정
- 3단계: 레지스터 또는 메모리에서 피연산자 결정
- 4단계: 명령어 연산 수행
- 5단계: 결과를 레지스터에 저장

- 5개의 장치가 서로 독립적으로 작동하고 각각의 명령이 순서대로 각 장치를 이동하며 실행된다면 실행 시간은 훨씬 단축

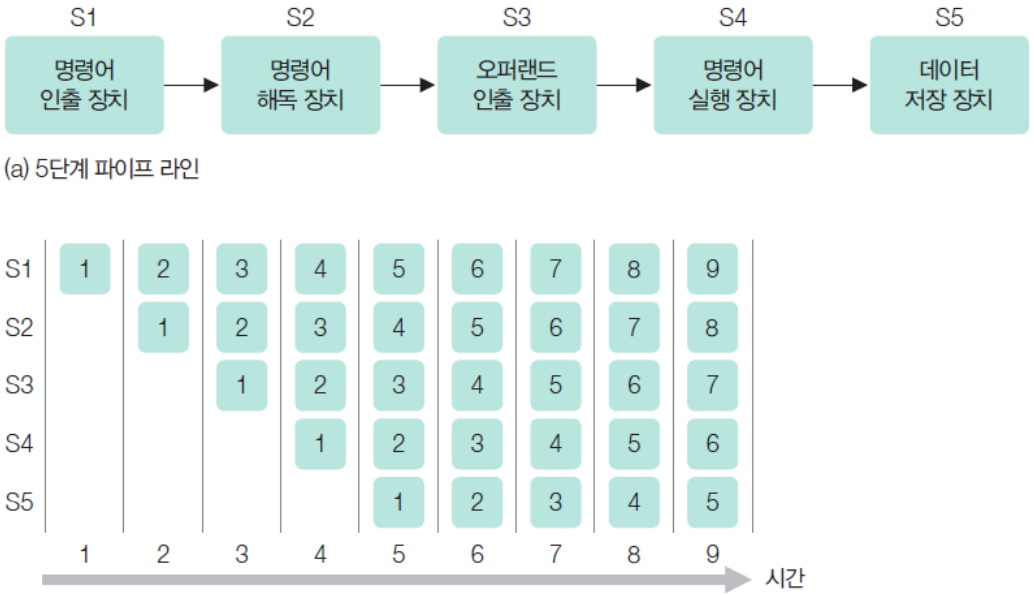


그림 5-24 5단계 파이프 라인에서 프로그램 실행

05 파이프 라이닝

❑ 5단계 파이프라인 (계속)

그림 5-24 5단계 파이프 라인에서 프로그램 실행

- 예: 각 단계가 2ns 소요
  - 전통적인 시스템 : 명령 하나가 완전히 실행되는 데는 10ns 소요 (= 5단계 x 2ns)
  - 파이프라인 시스템 : 매 클록 사이클(2ns)마다 명령 5개가 동시에 실행되므로 시간은 1/5로 단축
  - 파이프 라인이 없는 컴퓨터에서는 100MIPS
  - 5단계 파이프 라인을 가진 컴퓨터는 500MIPS의 처리 속도
- 파이프 라이닝을 사용하면 지연 시간(명령어를 실행하는 데 걸리는 시간)과 프로세서 대역폭 (CPU의 MIPS 수)간 균형 유지
  - 한 사이클에 소요되는 시간 : Tns
  - 파이프 라인 : n단계
  - 각 명령은 n 단계를 거치므로 전체 걸리는 시간: nTns
  - 하지만 파이프라인 구조이기 때문에 이후부터는 명령어 하나당 Tns 가 걸리게 됨
- 클록 사이클이 초당 10^9/T라면 초당 실행되는 명령의 개수는 $10^9/T$개
  - 예를 들어 T=2ns인 경우 매초 5억 건의 명령이 실행
  - MIPS(Million Instructions Per Second) : 초당 실행되는 명령 개수를 100만으로 나눔 $(10^9/T)/10^6=1000/T$ MIPS (보통 1000밉스짜리 CPU라고 부름)
  - 현재는 MIPS 대신 GIPS(Billion Instructions Per Second, BIPS)를 쓰는 것이 더 타당

05 파이프 라이닝

hazard

1. 위험 2. 해이 3.

1 데이터 해저드(data hazards)

- 데이터 의존성(data dependency) : 파이프 라인에서 앞서가는 명령의 ALU 연산 결과를 레지스터에 기록하기 전에 다른 명령에 이 데이터가 필요한 상황
- 앞의 명령 결과가 다음 명령 입력으로 사용될 때 파이프 라인 시스템에서 문제 발생
- 해결 방법
  - 레지스터에 저장되기 전에 ALU 결과를 직접 다음 명령에 직접 전달하는 데이터 포워딩
  - 또는 버블(**NOP**)을 명령 사이에 끼워 넣어 프로그램 실행을 1단계 또는 2단계 지연
- WAW(Write After Write)와 WAR(Write After Read) 해저드
  - 레지스터 재명명함(register renaming) : 관련 없는 레지스터로 바꾸어 사용함으로써 쉽게 해결가능
- 따라서 참 의존(True dependency)는 RAW(Read After Write)뿐임

WAW (Write After Write):
앞 명령어 write,
뒤 명령어 write;
R2: 데이터 덮어 쓰기됨

1. $R2 \leftarrow R1 + R7$
2. $R2 \leftarrow R1 + R3$

WAR (Write After Read):
앞 명령어 read,
뒤 명령어 write;
R7: 변경된 값을 읽을 수 있음

1. $R2 \leftarrow R1 + R7$
2. $R7 \leftarrow R1 + R3$

RAW (Read After Write):
앞 명령어 write,
뒤 명령어 read;
R2: 쓰기전에 읽을 수 있음

1. $R2 \leftarrow R1 + R3$
2. $R4 \leftarrow R2 + R3$

05 파이프 라이닝

- 예(RAW: read after write): 첫 번째 연산 ADD의 결과(r3)가 두 번째 연산 SUB의 입력으로 사용
  - 1번 명령이 S5단계에서 레지스터에 저장되는데
  - 2번 명령은 S3단계에서 데이터 요구

```
1. ADD    r3, r2, r1    /* r3 = r2 + r1 */
2. SUB    r4, r3, r5    /* r4 = r3 - r5 (HAZARD: r3이 아직 저장되지 않음!) */
```

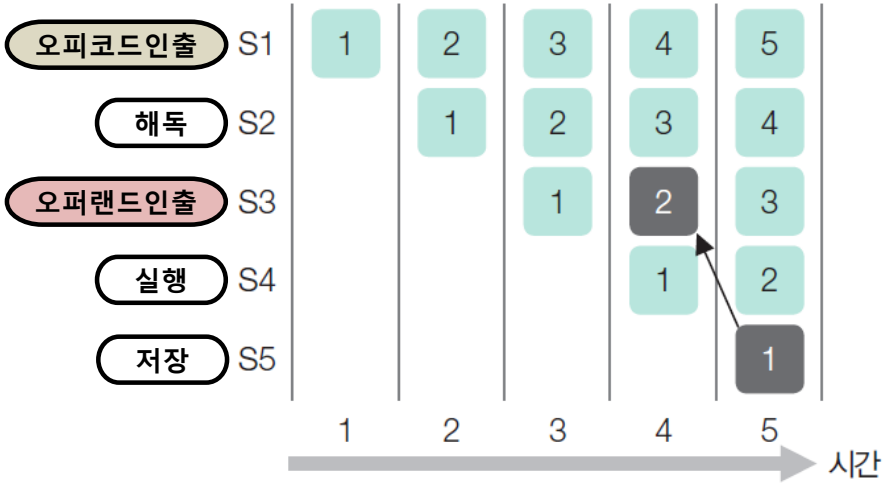


그림 5-25 데이터 의존성(종속)으로 인한 데이터 해저드

S5단계에서 1번 명령어의 마지막 저장(R3);
S3단계에서 2번 명령어가 R3을 읽어오려고 함;
이때 R3는 1번 명령어의 결과를 저장하지 않은 상태임

2 제어 해저드(control hazards)

❖ 파이프 라인 CPU 구조의 분기 명령이 실행될 때 발생

- 이미 파이프 라인에 적재되어 실행되고 있는 이어지는 다른 명령들이 더 이상 필요가 없어지므로 발생
- [그림 5-26]과 같이 3번 명령에서 15번 명령으로 분기가 일어난다면 이미 파이프 라인 단계에 들어와 실행되고 있는 4, 5, 6, 7번 명령은 더 이상 필요가 없으므로 전체 프로그램의 속도 저하 요인이 됨

❖ 제어 해저드 해결 방법

- 지연 슬롯(delay slot)을 넣고 분기 목적지 주소를 계산하는 과정을 파이프 라인 속에 넣는 것
- 지연 슬롯이란 NOP나 분기 명령과 무관한 명령을 끼워 넣는 것
- 이 방법은 컴파일러나 프로그래머가 프로그램 순서를 바꾸는 것
- 또는 분기 예측 알고리즘을 이용하기도 함

05 파이프 라이닝

❖ 분기로 인한 제어 해저드

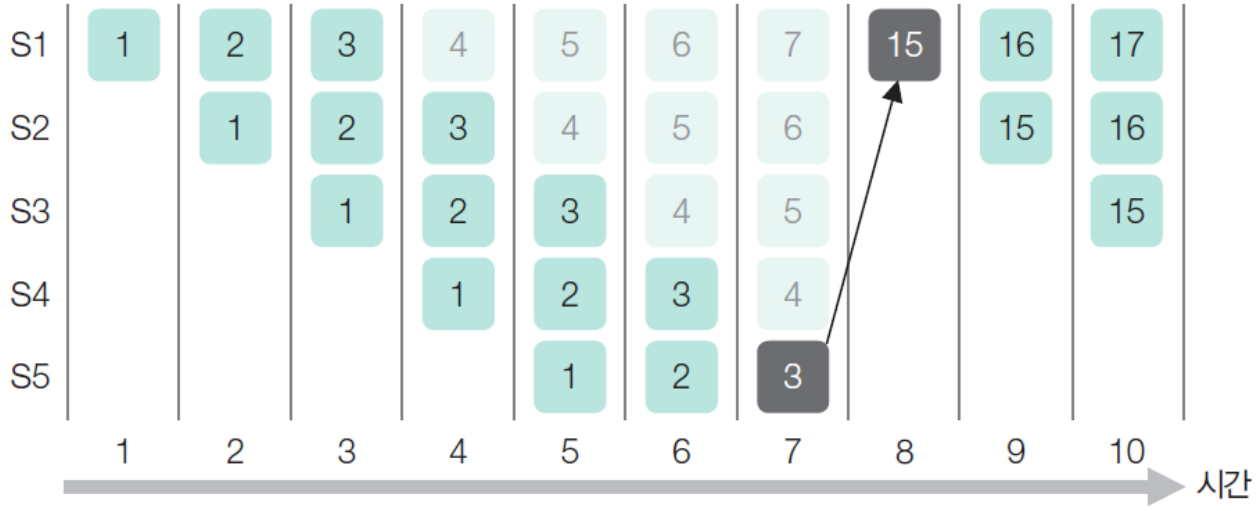
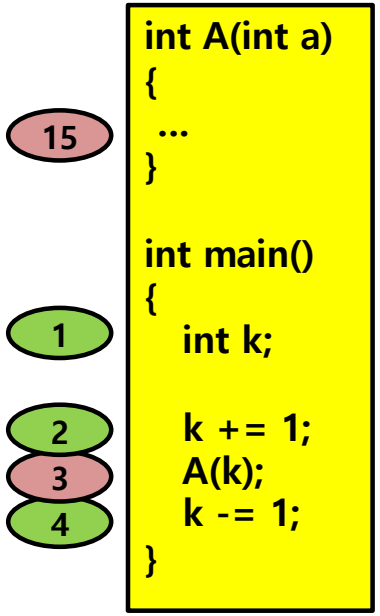


그림 5-26 분기로 인한 제어 해저드



컴파일 직후 파이프라인에 적재 → 실행했는데 Branch 하게 되면 이후 명령들이 소용 없어짐. 컴파일러가 분기예측알고리즘을 통해서 제어 해저드를 회피함

3 구조적 해저드(structural hazards)

- 서로 다른 단계에서 동시에 실행되는 명령이 컴퓨터 내의 장치 하나를 동시에 사용하려고 할 때 발생
- 예1 : 명령어 인출과 오퍼랜드 인출이 동시에 발생하는 경우
  - 명령 2개가 동시에 메모리에 가서 명령과 데이터를 가져와야 하는데,
  - 인출 과정이 모두 같은 장치들(bus, memory 등)을 사용하므로 충돌 발생
- 예2 : CPU 내의 장치인 ALU를 명령 2개가 동시에 사용해야 하는 경우

❖ 인출 과정에서 메모리 충돌을 해결 방법

- 하버드 구조(harvard architecture)를 사용
  - 메모리를 프로그램(명령어) 메모리와 데이터 메모리로 완전 분리시킨 구조
  - 명령어 인출과 데이터 인출 과정에서 충돌을 원천적으로 봉쇄
  - RISC 프로세서에서 많이 사용하는 구조
- 분리 캐시를 사용하여 충돌 회피
  - 인텔 계열 프로세서의 L1 캐시에서 사용하는 구조로, L1 명령어 캐시와 L1 데이터 캐시로 분리
  - 그러나 명령어 2개가 동시에 캐시 미스가 발생하면 충돌이 발생할 수 있음
- 동일한 장치를 동시 사용하여 일어나는 충돌 : 장치를 하나 더 둠
  - 예를 들어 CPU 내에 ALU를 2개 둠

05 파이프 라이닝

4 슈퍼 스칼라

❖ **하나의 프로세서 안에 2개 이상의 파이프 라인 탑재**

- 하나의 명령어 인출 장치가 명령어 쌍을 동시에 가져와서 각 명령어를 다른 파이프 라인에 배치하고 개별 ALU를 가지고 병렬 작업
- 명령 2개는 자원(예를 들어 레지스터)을 사용할 때 충돌하지 않아야 함
- 둘 중 어느 명령도 다른 명령의 결과에 의존하지 않아야 함
- 하드웨어를 추가하여 실행 도중에 충돌을 감지 및 제거

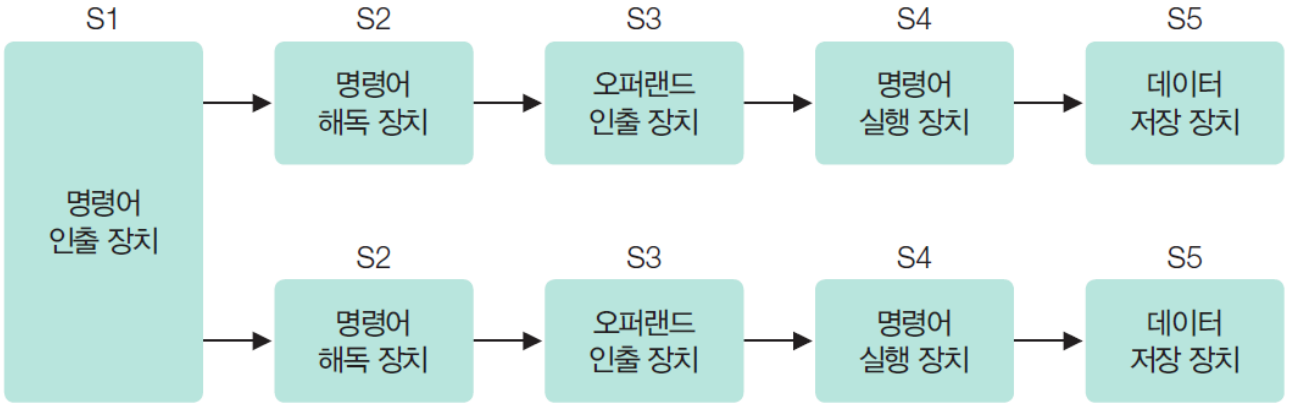


그림 5-27 슈퍼 스칼라에서 명령어 실행

05 파이프 라이닝

❖ 단일 또는 2중 파이프 라인 : RISC 머신이 원조

❖ 486부터는 인텔이 데이터 파이프 라인 도입

- 486은 파이프 라인을 하나
- 펜티엄은 [그림 5-27]과 비슷한 5단계 파이프 라인을 2개 가지고 있음
  - U 파이프 라인(메인 파이프 라인)은 임의의 펜티엄 명령을 실행
  - V 파이프 라인이라고 하는 두 번째 파이프 라인은 단순한 정수 명령어나 간단한 부동 소수점 명령어 하나만 실행

❖ 실행 규칙

- 명령어 한 쌍이 서로 호환되어 병렬로 실행될 수 있는지 여부를 결정
- 명령어 한 쌍이 충분히 단순하지 않거나 호환되지 않는 경우, 첫 번째 명령만 실행
- 두 번째 명령은 멈추어 있다가 다음 명령과 짝을 지어 실행
- 명령어는 항상 순서대로 수행: 따라서 호환 가능한 쌍을 생성한 펜티엄 전용 컴파일러는 구형 컴파일러보다 빠르게 실행되는 프로그램을 생성 - 측정 결과 최적화된 펜티엄 실행 코드는 동일한 클록 속도로 실행되는 486보다 정수 프로그램에서 정확히 2배 빠름 – 속도 향상은 두 번째 파이프 라인에 의한 것

05 파이프 라이닝

- 파이프 라인을 4개 도입하는 것도 가능하지만, 너무 많은 하드웨어가 소요
- 대신 고급 CPU에서는 다른 접근 방식 사용
  - 기본 개념은 파이프 라인은 하나지만 기능 장치를 여러 개 제공
  - 인텔 코어 아키텍처는 아래 그림과 유사한 구조

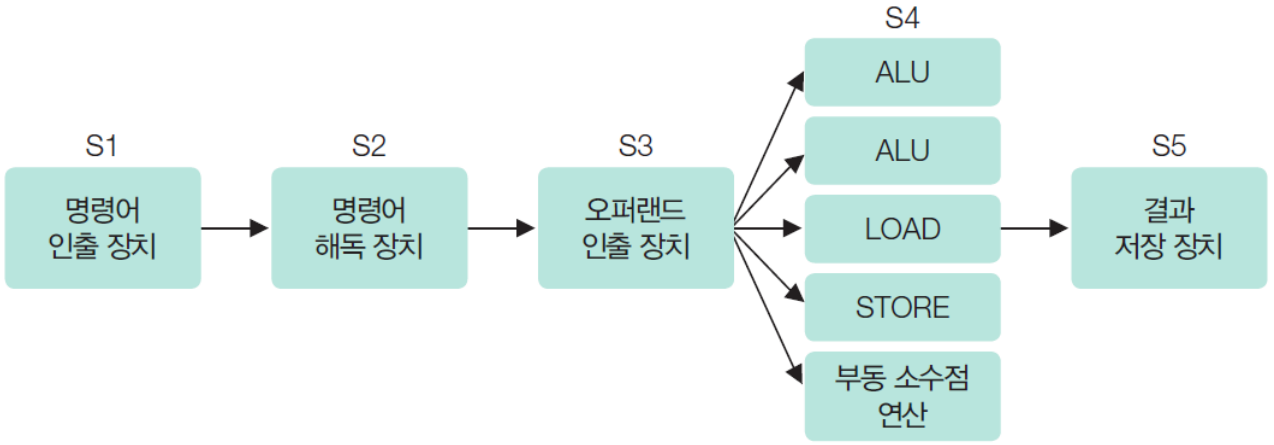


그림 5-28 기능 장치를 여러 개 가진 슈퍼 스칼라

05 파이프 라이닝

- 슈퍼 스칼라 프로세서라는 아이디어: S3이 S4 단계보다 훨씬 빠르게 명령을 실행 가능
- S3 단계에서 매 10ns마다 명령을 수행하고 모든 기능 유닛에서 10ns에 작업을 수행할 수 있다면, 작업을 한번에 하나 이상 수행할 수 없으므로 의미 없음
- S4 단계가 시간이 더 오래 걸린다면 의미 있음
  - 실제로 S4 단계의 기능 장치인 LOAD 및 STORE의 메모리 액세스 및 부동 소수점 연산은 실행에 1클록 사이클보다 오래 걸림
  - 그림에서 알 수 있는 바와 같이 S4 단계에서 다수의 ALU를 가짐

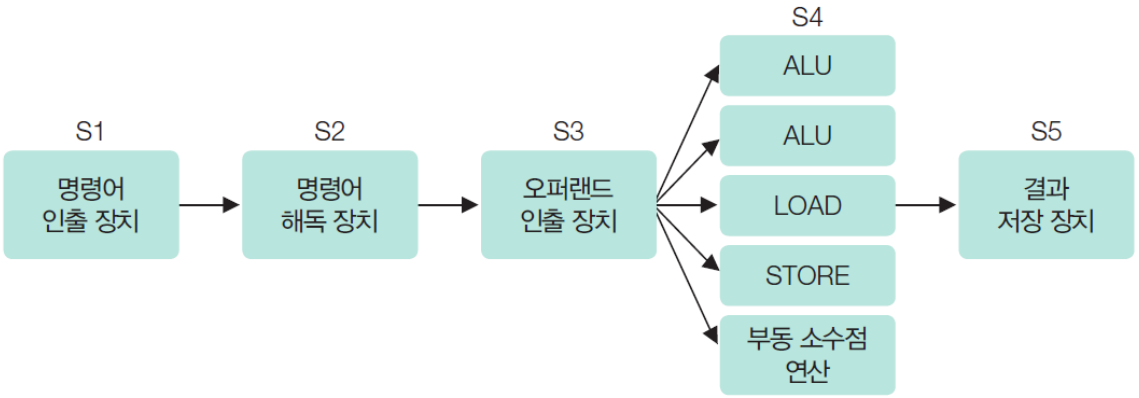
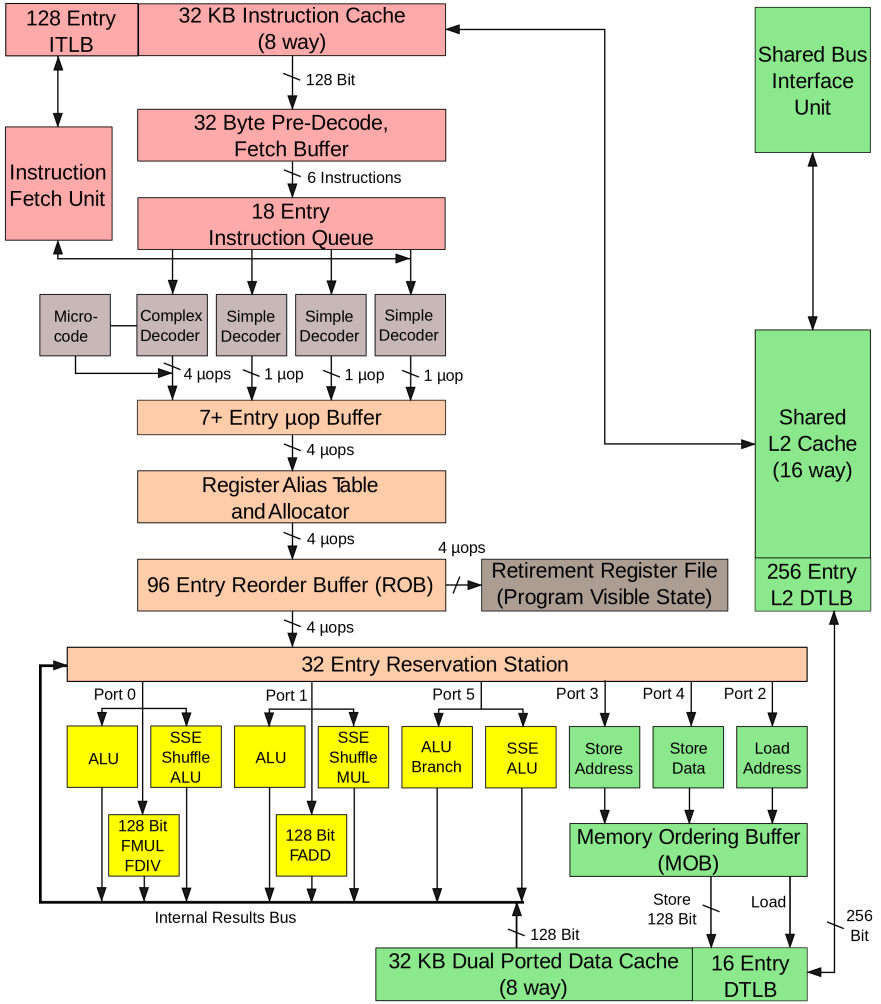


그림 5-28 기능 장치를 여러 개 가진 슈퍼 스칼라



Intel Core 2 Architecture

wiki: intel core series architecture

감사합니다.