컴퓨터 구조

2장 CPU의 구조와 기능2

안형태

anten@kumoh.ac.kr

디지털관 139호

CPU의 구조와 기능

2. 명령어 실행

명령어 실행

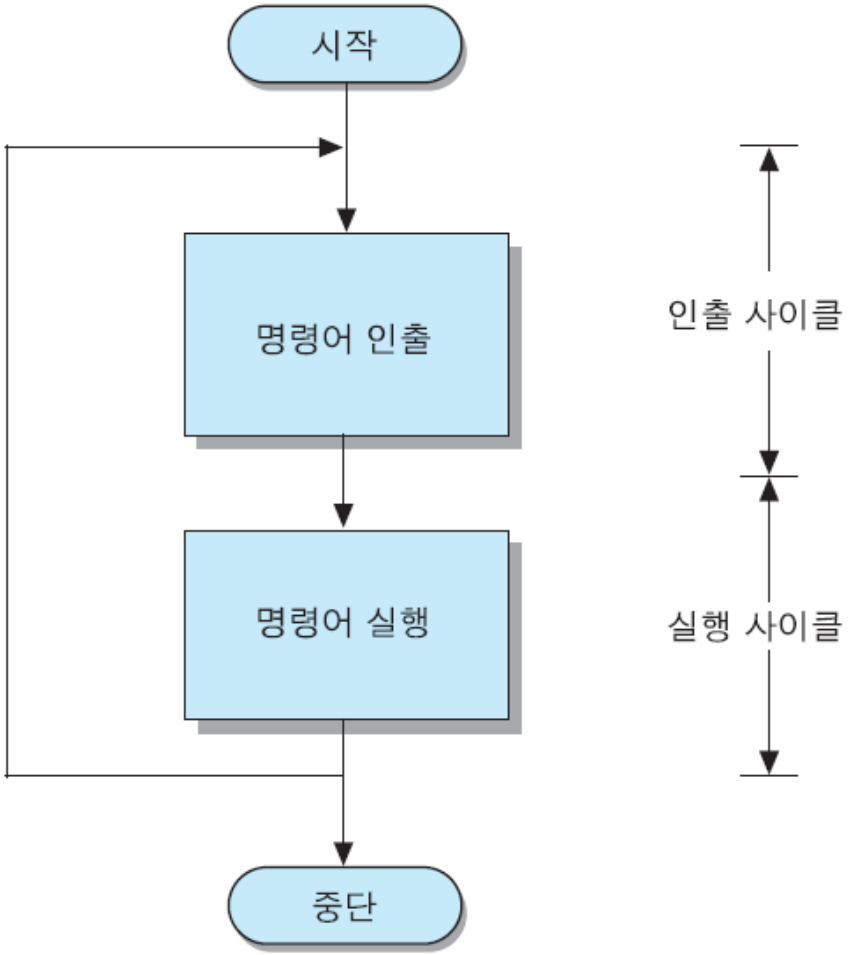
- CPU는 기억장치에 저장되어 명령어들을 인출하여 실행함으로써 실제적인 작업을 수행

- **명령어 사이클(instruction cycle)**
  - CPU가 한 개의 명령어를 실행하는 데 필요한 전체 처리 과정
    - CPU가 프로그램 실행을 시작한 순간부터 전원을 끄거나 회복 불가능한 오류가 발생하여 중단될 때까지 반복

- 두 개의 부사이클(sub-cycle)들로 분리
  - **인출 사이클(fetch cycle):** CPU가 기억장치로부터 명령어를 읽어오는 단계
  - **실행 사이클(execution cycle):** 명령어를 해독하고 실행하는 단계

명령어 실행

- 기본 명령어 사이클



명령어 실행에 필요한 CPU 내부 레지스터들

- **프로그램 카운터(Program Counter, PC)**
  - 다음에 인출할 명령어의 주소를 가지고 있는 레지스터
  - 명령어가 인출된 후에는 자동적으로 일정 크기(한 명령어 길이)만큼 증가
  - 분기(branch) 명령어가 실행되는 경우에는 목적지 주소로 갱신
- **누산기(Accumulator, AC)**
  - 데이터를 일시적으로 저장하는 레지스터
  - ALU의 산술 연산과 논리 연산 과정에 사용
  - 레지스터의 길이는 CPU가 한 번에 처리할 수 있는 데이터 비트 수(단어 길이)와 동일
- **명령어 레지스터(Instruction Register, IR)**
  - 가장 최근에 인출된 명령어 코드가 저장되어 있는 레지스터
  - 제어 장치는 IR에서 명령어를 읽어 와서 해독하고 명령을 수행하기 위해 컴퓨터의 각 장치에 제어신호 전송

명령어 실행에 필요한 CPU 내부 레지스터들

- **기억장치 주소 레지스터(Memory Address Register, MAR)**
  - PC에 저장된 명령어 주소가 시스템 주소 버스로 출력되기 전에 일시적으로 저장되는 주소 레지스터
- **기억장치 버퍼 레지스터(Memory Buffer Register, MBR)**
  - 기억장치 데이터 레지스터(Memory Data Register, MDR)라고도 불림
  - 기억장치에 쓰여질 데이터 혹은 기억장치로부터 읽혀진 데이터를 일시적으로 저장하는 버퍼 레지스터

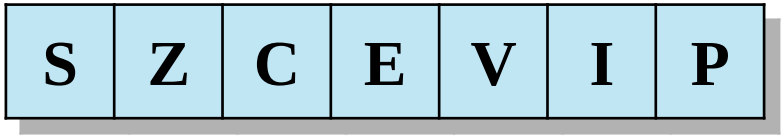
기타 CPU 내부 레지스터들

- **데이터 레지스터(Data Register, 범용 레지스터)**
  - CPU내의 데이터를 일시적으로 저장하기 위한 레지스터
- **입출력 주소 레지스터(I/O Address Register, I/O AR)**
  - 특정 I/O 장치의 주소를 지정하는 데 사용
- **입출력 버퍼 레지스터(I/O Buffer Register, I/O BR)**
  - I/O 모듈과 CPU 간에 데이터를 교환하는 데 사용
- **스택 제어 레지스터(Stack Control Register, Stack Pointer, SP)**
  - 메모리의 한 블록이며, 데이터는 후입 선출(Last In-First Out, LIFO)
  - 복귀할 주소 정보를 저장

기타 CPU 내부 레지스터들

❑ 상태 레지스터(Status Register, SR)

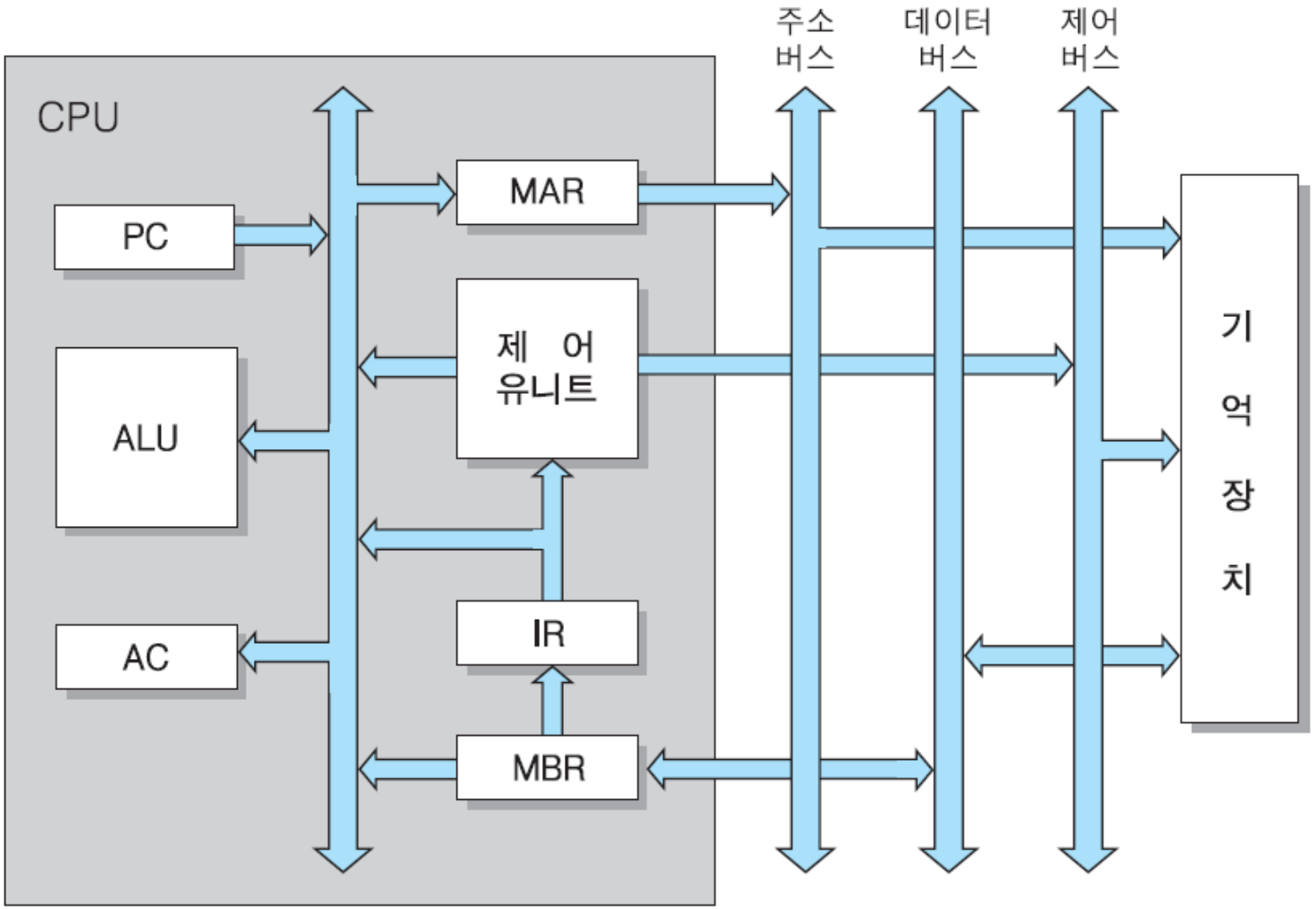
- CPU가 작동하는 동안 특정 조건 발생을 표시하는 데 사용
- 1Byte 또는 2Bytes인 특수 목적용 레지스터
- 명령어 실행 결과에 따른 **조건 플래그(condition flag)**들 저장
  - **[예]** 산술 연산 또는 비교 결과로 모든 비트가 0인 제로(zero) 값이 누산기(AC)에 입력되면, 제로 플래그(Z)를 1로 설정
- 플래그 레지스터(Flag Register, FR), 프로그램 상태 워드(Program Status Word, PSW)라고도 함



상태 레지스터(status register)

- 조건 플래그의 종류
  - **부호(S) 플래그:** 직전에 수행된 산술연산 결과값의 부호 비트를 저장 (양수: 0, 음수: 1)
  - **제로(Z) 플래그:** 연산 결과값이 0이면, 1로 세트
  - **올림수(C) 플래그:** 덧셈이나 뺄셈에서 올림수(carry)나 빌림수(borrow)가 발생한 경우에 1로 세트
  - **동등(E) 플래그:** 두 수를 비교한 결과가 같게 나왔을 경우에 1로 세트
  - **오버플로우(V) 플래그:** 산술 연산 과정에서 오버플로우가 발생한 경우에 1로 세트
  - **인터럽트(I) 플래그**
    - 인터럽트 가능(interrupt enabled) 상태이면, 0으로 세트
    - 인터럽트 불가능(interrupt disabled) 상태이면, 1로 세트
  - **슈퍼바이저(P) 플래그**
    - CPU의 실행 모드가 슈퍼바이저(커널) 모드(supervisor mode)이면, 1로 세트
    - 사용자 모드(user mode)이면, 0으로 세트

데이터 통로가 표시된 CPU 내부 구조



CPU 클럭 주파수와 클럭 주기

❑ 클럭(Clock)

- CPU를 비롯한 컴퓨터의 모든 부품이 일정한 속도로 작동하기 위한 전기적 **진동(pulse)**
  - 클럭 발생기가 만들며, 일반적으로 클럭 수가 클수록 컴퓨터의 처리 속도가 빠름
- 클럭 주파수(Hz 단위)는 1초에 클럭이 몇 번 발생하는지를 의미
  - 1초에 1번 클럭이 발생하면 클럭 주파수는 1Hz
  - 1초에 10^9번 클럭이 발생하면 클럭 주파수는 1GHz
- 클럭 주기는 한 신호 뒤에서 다음 신호가 올 때까지의 간격
  - 클럭 주기 = $\frac{1}{\text{클럭 주파수}}$

❑ 마이크로 연산(micro-operation)

- CPU 클럭의 각 주기 동안 수행되는 기본 단위의 동작

인출 사이클

- 인출 사이클의 마이크로 연산

t_0 : **MAR ← PC**

t_1 : **MBR ← M[MAR], PC ← PC + 1**

t_2 : **IR ← MBR**

단, t_0 , t_1 및 t_2 는 CPU 클럭(clock)의 주기

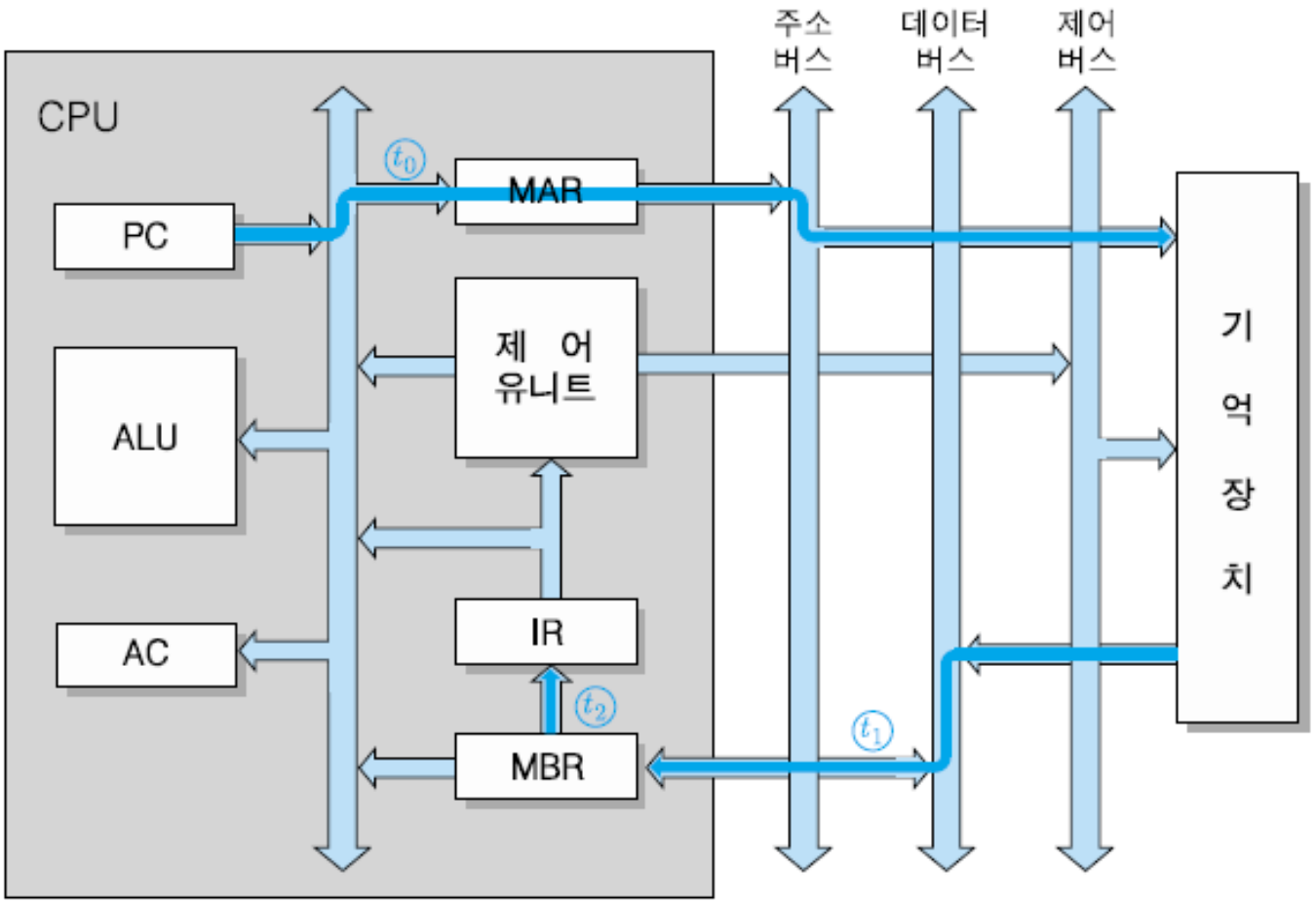
- 첫번째 주기(t_0): 현재의 PC가 지정하는 명령어의 주소가 CPU 내부 버스를 통하여 MAR로 이동
- 두번째 주기(t_1): MAR에 저장된 명령어의 주소가 기억장치로 이동되고, 기억장치는 해당 주소의 명령어를 읽어서 데이터 버스를 통하여 MBR에 적재하며, PC의 내용에 1을 더함
  - 만약, 주소 지정 단위가 1Byte이고, 명령어 크기가 2Bytes인 경우, t_1 에서 PC ← PC + 2로 증가
- 세번째 주기(t_2): MBR에 있는 명령어가 명령어 레지스터인 IR로 이동

인출 사이클의 주소 및 명령어 흐름도

t_0 : MAR ← PC

t_1 : MBR ← M[MAR], PC ← PC + 1

t_2 : IR ← MBR



실행 사이클

- CPU는 실행 사이클 동안에 **명령어 코드를 해독(decode)**하고, 그 결과에 따라 필요한 **연산들을 수행**

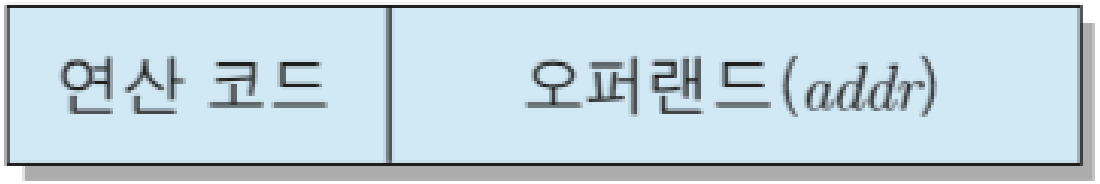
- CPU가 수행하는 연산들의 종류
  - **데이터 이동:** CPU와 기억장치 간 혹은 I/O장치 간에 데이터를 이동
  - **데이터 처리:** 데이터에 대하여 산술 혹은 논리 연산을 수행
  - **데이터 저장:** 연산결과 데이터 혹은 입력장치부터 읽어드린 데이터를 기억장치에 저장
  - **프로그램 제어:** 프로그램의 실행 순서를 결정

- 실행 사이클에서 수행되는 마이크로-연산들은 명령어의 **연산코드(OP code)**에 따라 결정됨

기본적인 명령어 형식의 구성

- **연산 코드(OPeration code, OP code)**
  - CPU가 수행할 연산을 지정

- **오퍼랜드(operand)**
  - 명령어 실행에 필요한 데이터가 저장된 주소(***addr***)



실행사이클 – LOAD, *addr* 명령어

- 기억장치에 저장되어 있는 데이터를 CPU 내부 레지스터인 AC로 이동하는 명령어

  t_0 : **MAR ← IR(*addr*)**

  t_1 : **MBR ← M[MAR]**

  t_2 : **AC ← MBR**

  - 첫번째 주기(t_0): 명령어 레지스터(IR)에 있는 명령어의 주소 부분을 MAR로 전송
  - 두번째 주기(t_1): 해당 주소가 지정한 기억장소로부터 데이터를 인출하여 MBR로 전송
  - 세번째 주기(t_2): 해당 데이터를 AC에 적재
- **[예]** CPU 클럭이 2GHz 인 경우 클럭 주기 및 LOAD 명령어 수행 시간
  - 클럭의 주기 = $\mathbf{1s/(2 \times 10^9) = 0.5}\boldsymbol{ns}$
  - 인출 및 실행 사이클: **0.5ns × (3+3) = 3.0*ns***

실행사이클 – STA(STOR), *addr* 명령어

- AC 레지스터의 내용을 기억장치에 저장하는 명령어

$$t_0 : \mathbf{MAR} \leftarrow \mathbf{IR}(addr)$$

$$t_1 : \mathbf{MBR} \leftarrow \mathbf{AC}$$

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

  - 첫번째 주기(t_0): 데이터를 저장할 기억장치의 주소를 MAR로 전송
  - 두번째 주기(t_1): 저장할 데이터를 버퍼 레지스터인 MBR로 이동
  - 세번째 주기(t_2): MBR의 내용을 MAR이 지정하는 기억장소에 저장

실행사이클 – ADD, *addr* 명령어

- 기억장치에 저장된 데이터를 AC의 내용과 더하고, 그 결과는 다시 AC에 저장하는 명령어

$$t_0 : \mathbf{MAR} \leftarrow \mathbf{IR}(addr)$$

$$t_1 : \mathbf{MBR} \leftarrow \mathbf{M[MAR]}$$

$$t_2 : \mathbf{AC} \leftarrow \mathbf{AC} + \mathbf{MBR}$$

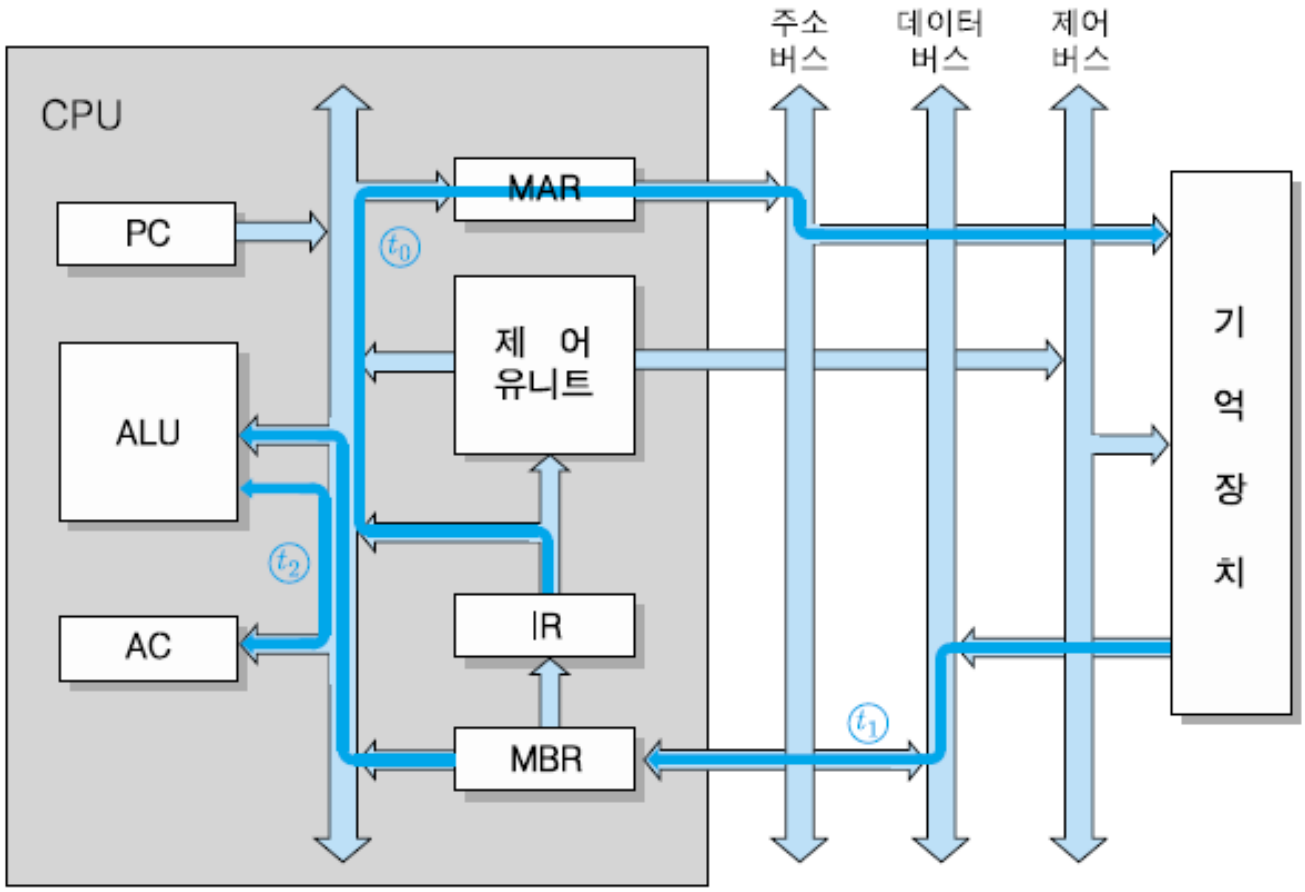
  - 첫번째 주기(t_0): 명령어 레지스터(IR)에 있는 명령어의 주소 부분을 MAR로 전송
  - 두번째 주기(t_1): 해당 주소가 지정한 기억장소로부터 데이터를 인출하여 MBR로 전송
  - 세번째 주기(t_2): 해당 데이터와 AC의 내용을 더하고, 결과값을 다시 AC에 저장

ADD 명령어 실행 사이클 동안의 정보 흐름

t_0 : MAR ← IR(*addr*)

t_1 : MBR ← M[MAR]

t_2 : AC← AC + MBR



실행사이클 – JUMP, *addr* 명령어

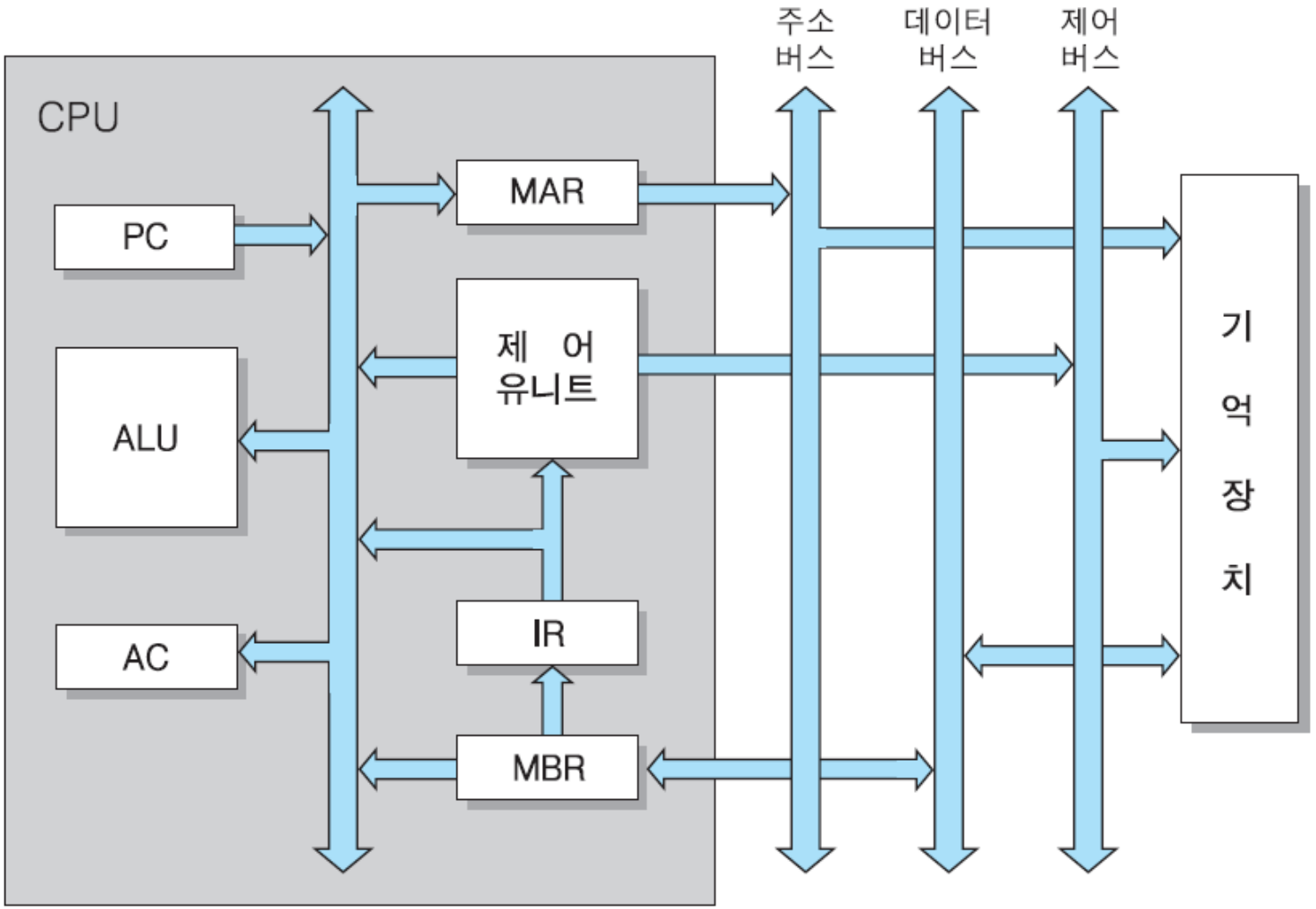
- 오퍼랜드(***addr***)가 가리키는 위치의 명령어로 실행 순서를 변경하는 **분기(branch)** 명령어(제어 명령어)

$$t_0 : \mathbf{PC} \leftarrow \mathrm{IR}(addr)$$

  - 명령어의 오퍼랜드(**분기할 목적지 주소**)를 PC에 저장
  - 다음 명령어 인출 사이클에서 해당 주소의 명령어가 인출되므로, 분기가 발생

- **[예]** CPU 클럭이 4GHz 인 경우 클럭 주기 및 JUMP 명령어 수행 시간
  - 클럭 주기 = $\mathbf{1s/(4\times10^9) = 0.25}\boldsymbol{ns}$
  - 인출 및 실행 사이클: $\mathbf{0.25ns\times(3+1) = 1.0}\boldsymbol{ns}$

기본 명령어 사이클 전체 흐름



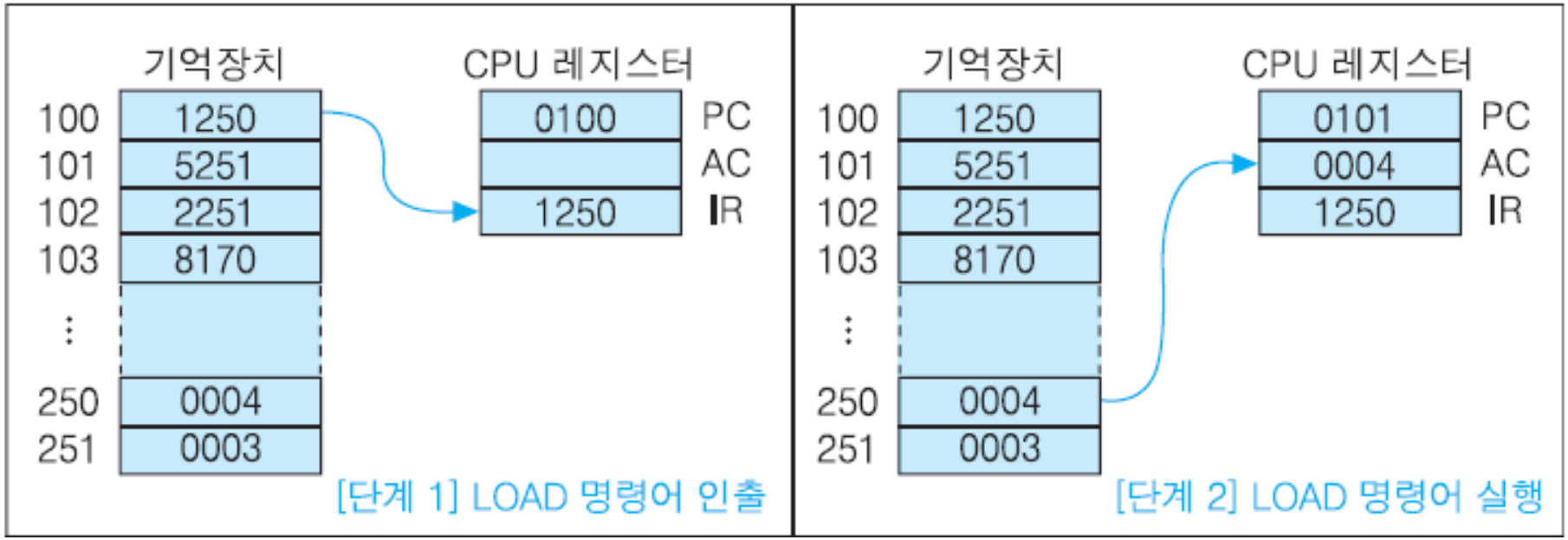
어셈블리 프로그램의 실행 과정

- 연산 코드에 임의의 정수(10진수) 배정
  - **LOAD:** **1**
  - **STA:** **2**
  - **ADD:** **5**
  - **JUMP:** **8**
  - **[예]** 어셈블리 프로그램

주소	명령어	기계 코드
100	LOAD 250	1250
101	ADD 251	5251
102	STA 251	2251
103	JUMP 170	8170

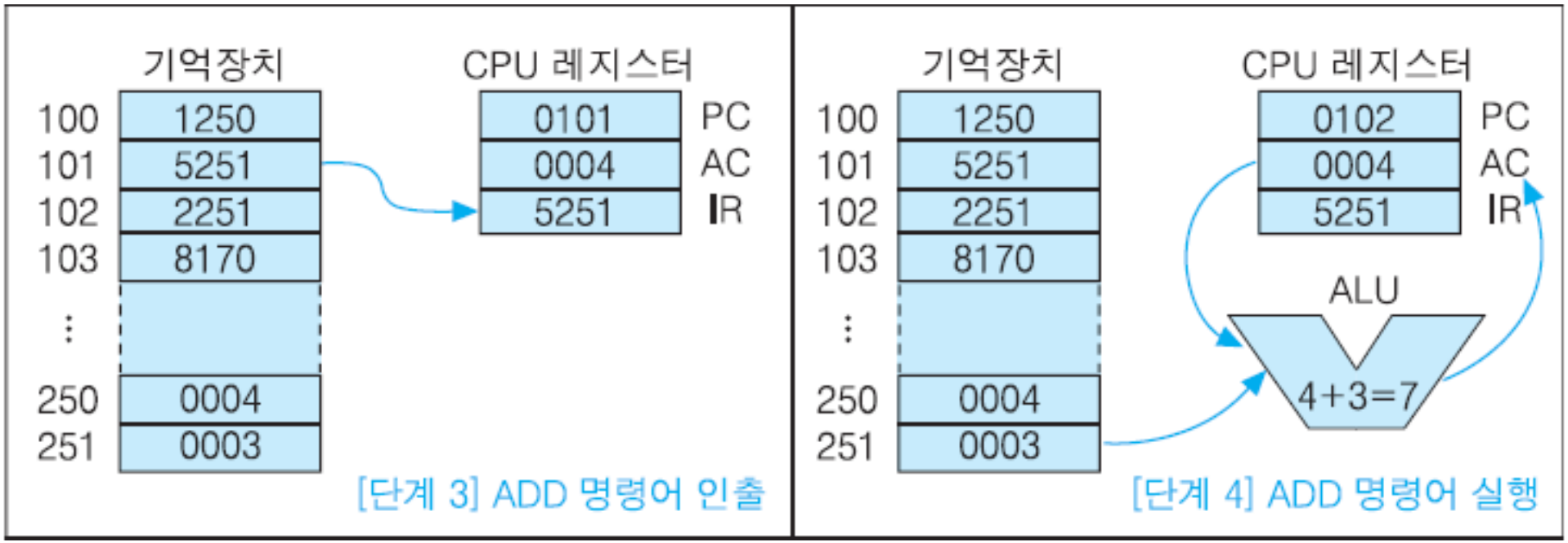
어셈블리 프로그램의 실행 과정

- **LOAD(1) 250, PC → 100번지**
  - 100번지의 첫 번째 명령어 코드가 인출되어 IR에 저장
  - 250번지의 데이터를 AC로 이동
  - PC = PC + 1 = 101



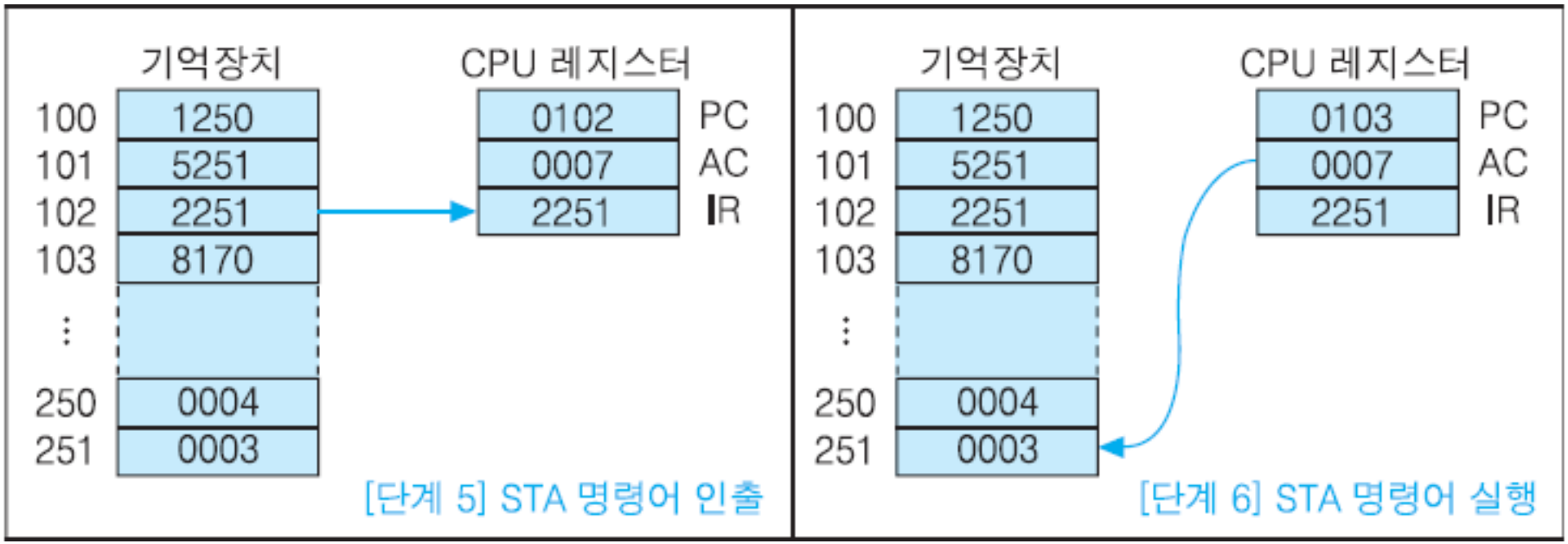
어셈블리 프로그램의 실행 과정

- **ADD(5) 251, PC → 101번지**
  - 두 번째 명령어가 101번지로부터 인출되어 IR에 저장
  - AC의 내용과 251번지의 내용을 더하고, 결과를 AC에 저장
  - PC의 내용은 102로 증가



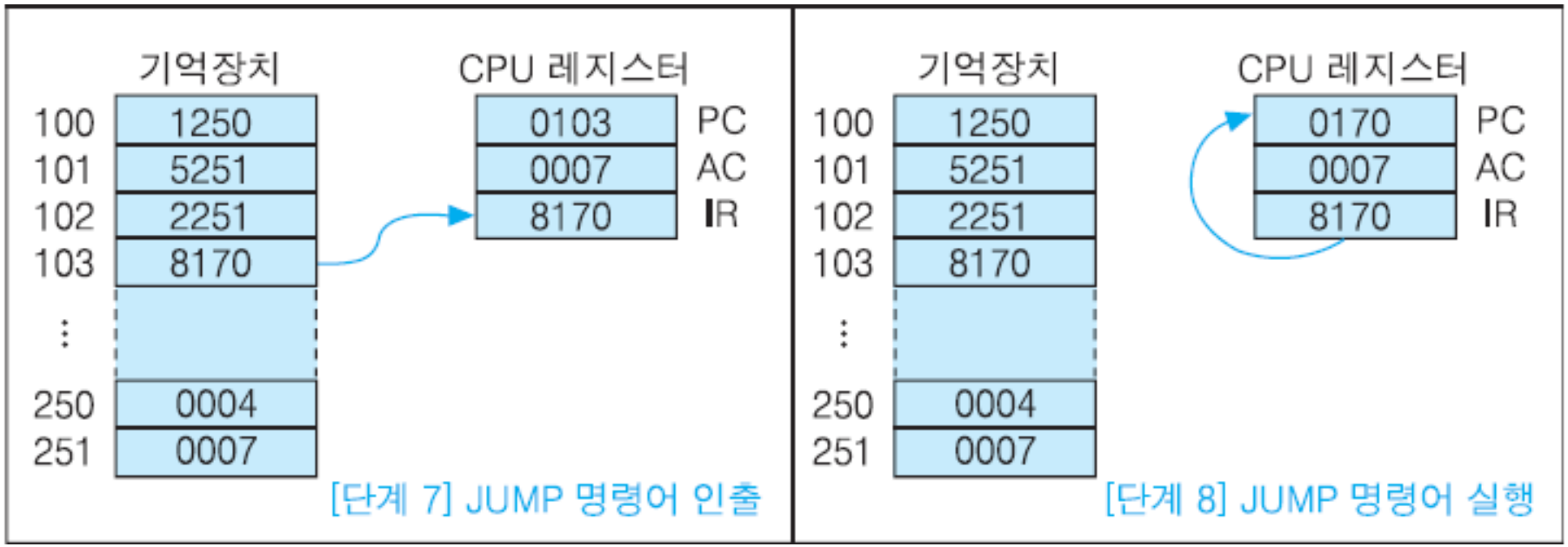
어셈블리 프로그램의 실행 과정

- **STA(2) 251, PC → 102번지**
  - 세 번째 명령어가 102번지로부터 인출되어 IR에 저장
  - AC의 내용을 251번지에 저장
  - PC의 내용은 103으로 증가



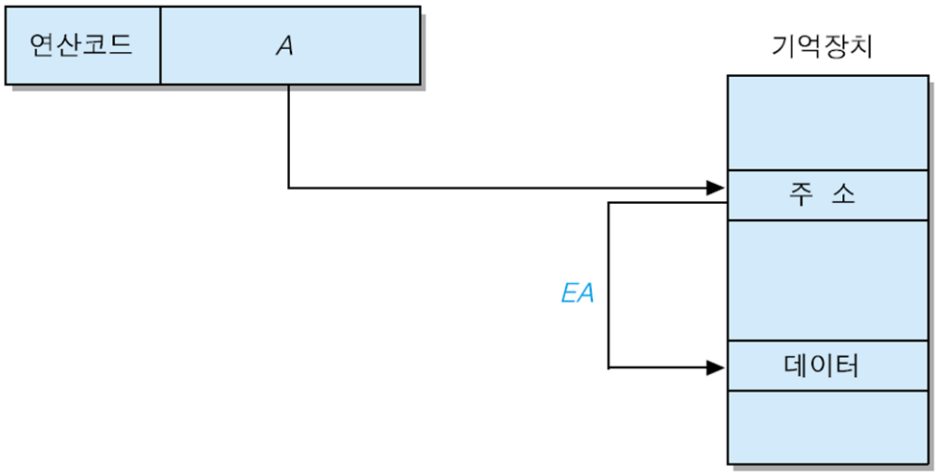
어셈블리 프로그램의 실행 과정

- **JUMP(8) 170, PC → 103번지**
  - 네 번째 명령어가 103번지로부터 인출되어 IR에 저장
  - 분기될 목적지 주소, 즉 IR의 하위 부분(170)이 PC로 적재
    - 다음 명령어 인출 사이클에서는 170 번지의 명령어 인출



간접 사이클(indirect cycle)

- 명령어에 포함되어 있는 주소를 이용하여, 그 명령어 실행에 필요한 **데이터의 주소를 인출**하는 사이클
  - 인출 사이클과 실행 사이클 사이에 위치
  - 간접 주소지정 방식(indirect addressing mode)에서 사용



- 인출된 명령어의 주소 필드 내용을 이용하여 기억장치로부터 데이터의 실제 주소를 인출하여 IR의 주소 필드에 저장

간접 사이클(indirect cycle)

- 간접 사이클에서 수행될 마이크로-연산

$$t_0 : \mathbf{MAR} \leftarrow \mathbf{IR}(addr)$$

$$t_1 : \mathbf{MBR} \leftarrow \mathbf{M[MAR]}$$

$$t_2 : \mathbf{IR}(addr) \leftarrow \mathbf{MBR}$$

  - 첫번째 주기(t_0): 명령어 레지스터인 IR에 있는 명령어의 오퍼랜드(addr) 값을 MAR로 전송
  - 두번째 주기(t_1): 그 주소 값이 지정하는 기억장치 주소로부터 읽혀진 데이터를 데이터 버스를 통하여 MBR에 저장
  - 세번째 주기(t_2): 전송된 MBR의 데이터는 유효주소 정보이기에 그 값을 다시 IR의 주소 필드로 전송
- **[예]** CPU 클럭이 2GHz 인 경우 ADD 명령어 내에 간접 사이클이 포함된 수행 시간
  - 인출, 간접, 실행 사이클: **0.5ns × (3+3+3) = 4.5*ns***

여러 가지 명령어의 종합적인 실행 과정

- **[예제]** 프로그램이 아래 표와 같이 작성되었을 때, 프로그램 실행 시간? (단, CPU 클럭은 2GHz이고, 메모리 지연시간은 없다고 가정)

주소	명령어	간접 사이클(I)	기계코드
100	LOAD 250	1	1250
101	ADD 251	1	5251
102	STORE 251	0	2251
103	JUMP 170	0	8170

- **[풀이]**
  - 클럭 주기 = $\frac{1}{2\times10^9} = 0.5ns$
  - LOAD : 인출(3) + 간접(3) + 실행(3) = 9개 클럭 주기
  - ADD : 인출(3) + 간접(3) + 실행(3) = 9개 클럭 주기
  - STORE : 인출(3) + 간접(0) + 실행(3) = 6개 클럭 주기
  - JUMP : 인출(3) + 간접(0) + 실행(1) = 4개 클럭 주기
  - 프로그램 실행에 총 28개 클럭 주기가 요구되므로, 프로그램 실행시간은 28×0.5ns = 14ns

인터럽트 사이클(interrupt cycle)

인터럽트(interrupt)

- CPU가 정상적인 프로그램 실행 중에 또 다른 프로그램의 실행 요구로 현재 실행 중인 프로그램을 중단시키고 요구된 프로그램을 실행
  - 긴급한 상황 대처나 외부 장치들과 상호 작용을 위하여 필요
- CPU가 프로그램 실행 중에 인터럽트 요구가 들어오면, **원래의 프로그램 수행을 중단**하고, 요구된 **인터럽트를 위한 서비스 프로그램을 먼저 수행**
- 인터럽트 시스템의 기본요소
  - 인터럽트 요청 신호, 인터럽트 처리 루틴, 인터럽트 서비스 루틴

인터럽트 서비스 루틴(Interrupt Service Routine, ISR)

- 인터럽트를 처리하기 위하여 수행되는 프로그램 루틴

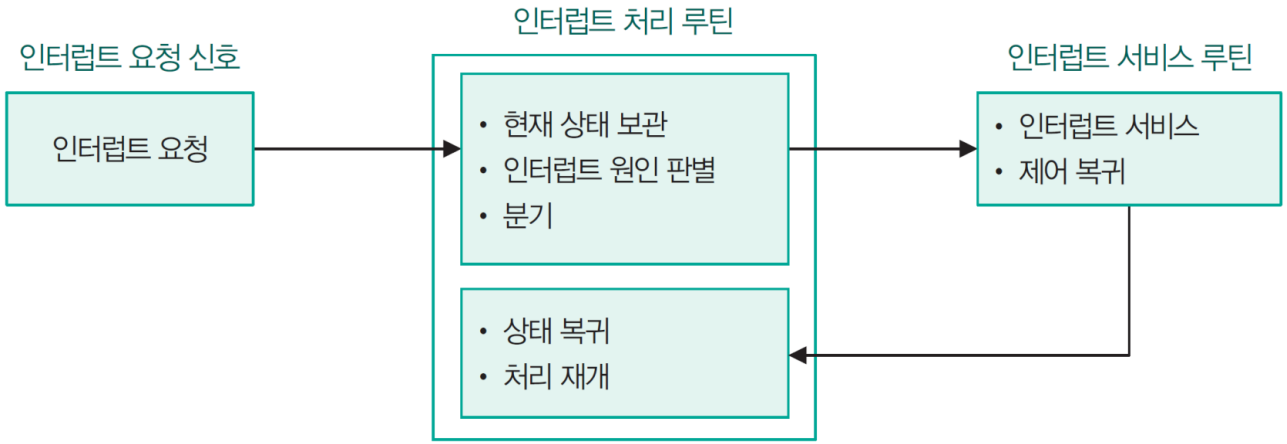
인터럽트 벡터 테이블(interrupt vector table)

- 다양한 인터럽트 신호를 처리하는 ISR의 시작 주소를 포함

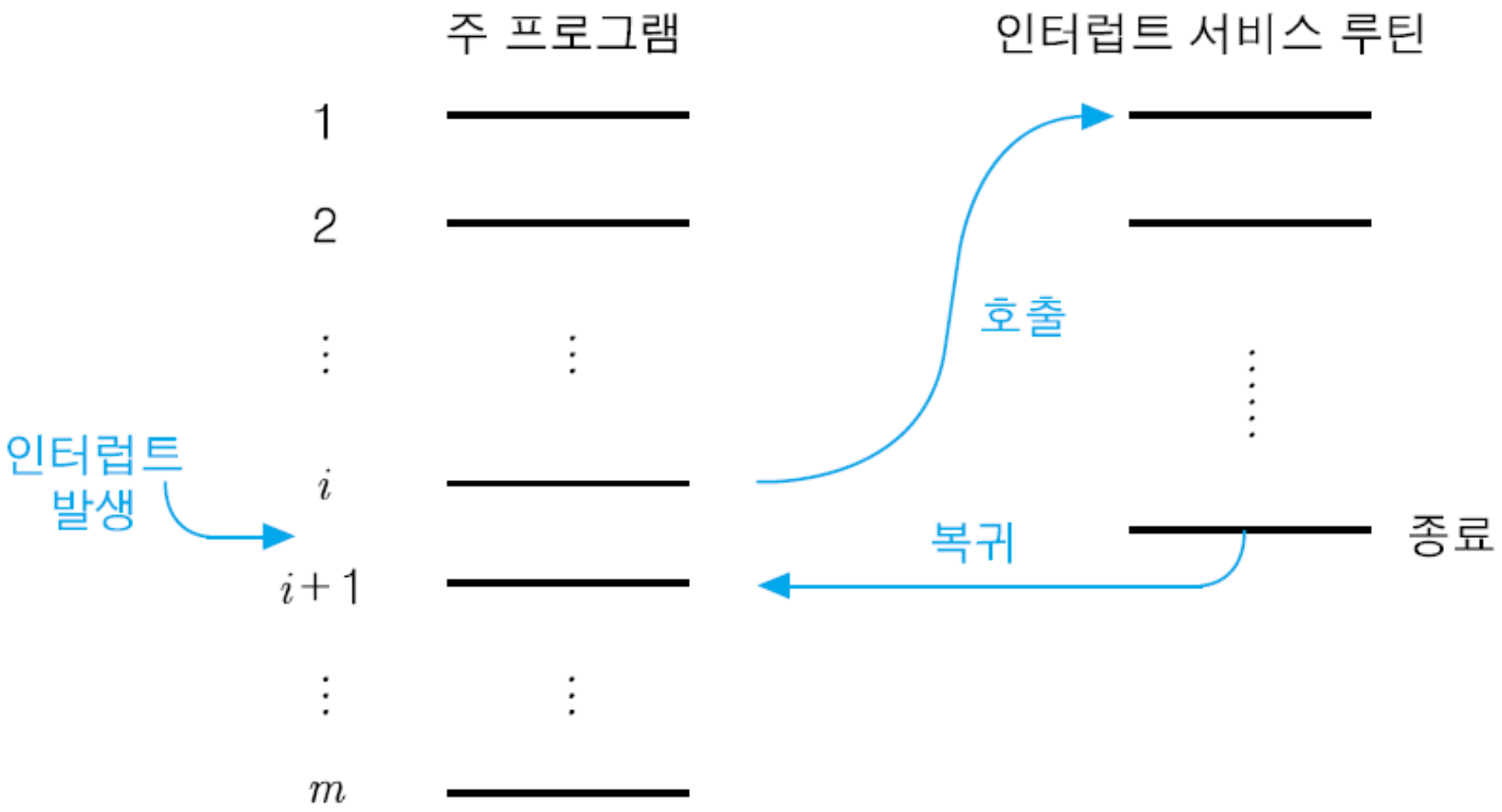
인터럽트 처리 과정

❑ 인터럽트 처리 과정

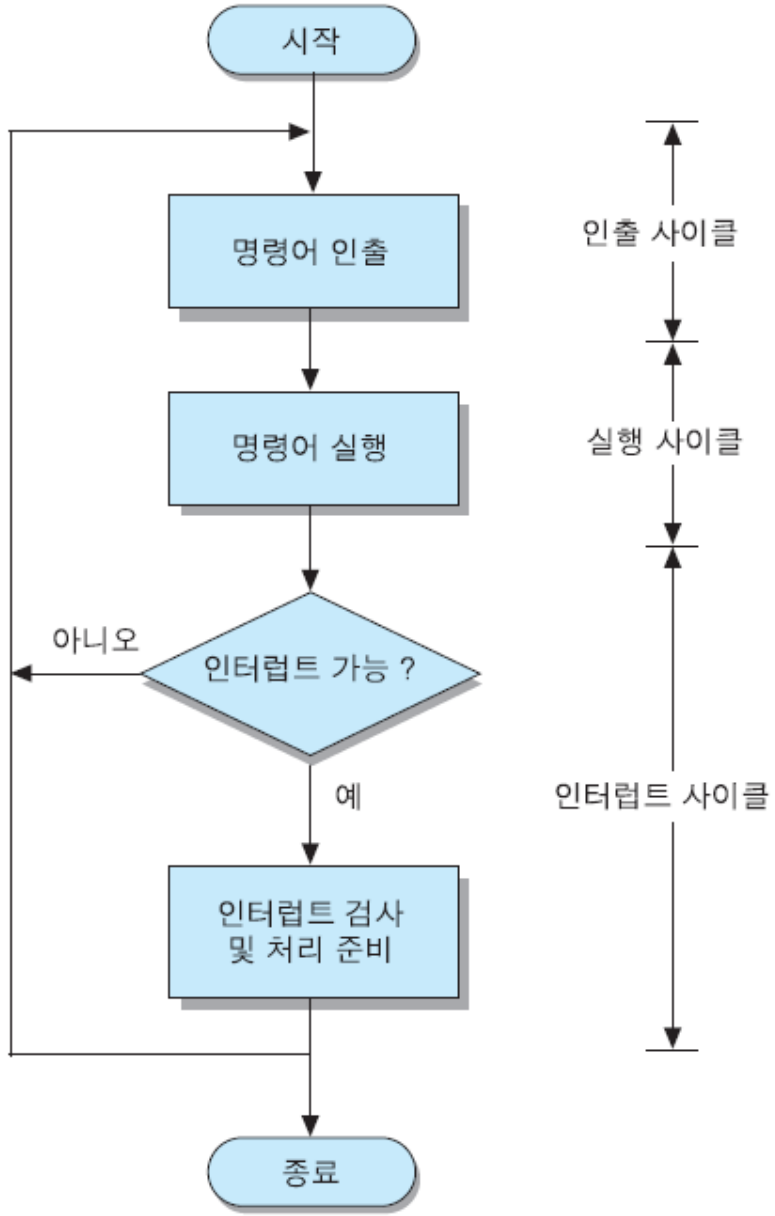
- ① 인터럽트 요청 신호가 발생
- ② CPU는 현재 수행 중인 명령어까지만 완료한 후, 수행 중인 프로그램을 일시 중지
- ③ 수행 중인 프로그램 상태(PC의 내용 등)를 안전한 장소([예] stack)에 보관
  - 일반적으로 스택은 주기억장치의 끝 부분을 사용
- ④ 인터럽트의 원인을 찾아 해당 인터럽트 서비스 루틴을 실행
  - 해당 인터럽트 서비스 루틴의 시작 주소를 PC에 적재
- ⑤ 인터럽트 서비스 루틴이 끝나면 보관해 두었던 레지스터 내용과 PC 내용을 복구하여 인터럽트 당한 프로그램을 중단된 곳부터 다시 수행



인터럽트에 의한 제어의 이동



인터럽트 사이클이 추가된 명령어 사이클



인터럽트 사이클의 마이크로 연산

- 인터럽트 사이클의 마이크로 연산

  t_0 : MBR ← PC

  t_1 : MAR ← SP, PC ← ISR의 시작 주소

  t_2 : M[MAR] ← MBR, SP ← SP - 1

  - 스택 포인터(Stack Pointer, SP)는 스택의 최상위 주소(top of stack, TOS)를 저장하는 레지스터로, 저장 후에는 1 감소)
  - 첫번째 주기(t_0): PC의 내용을 MBR로 전송
  - 두번째 주기(t_1): SP의 내용을 MAR로 전송하고, PC의 내용은 인터럽트 서비스 루틴의 시작 주소로 변경
    - 만약 주소지정 단위가 1Byte이고 저장될 주소는 16bits라면, t_2 에서 SP ← SP – 2 로 변경
  - 세번째 주기(t_2): MBR에 저장되어 있던 원래 PC의 내용을 스택에 저장

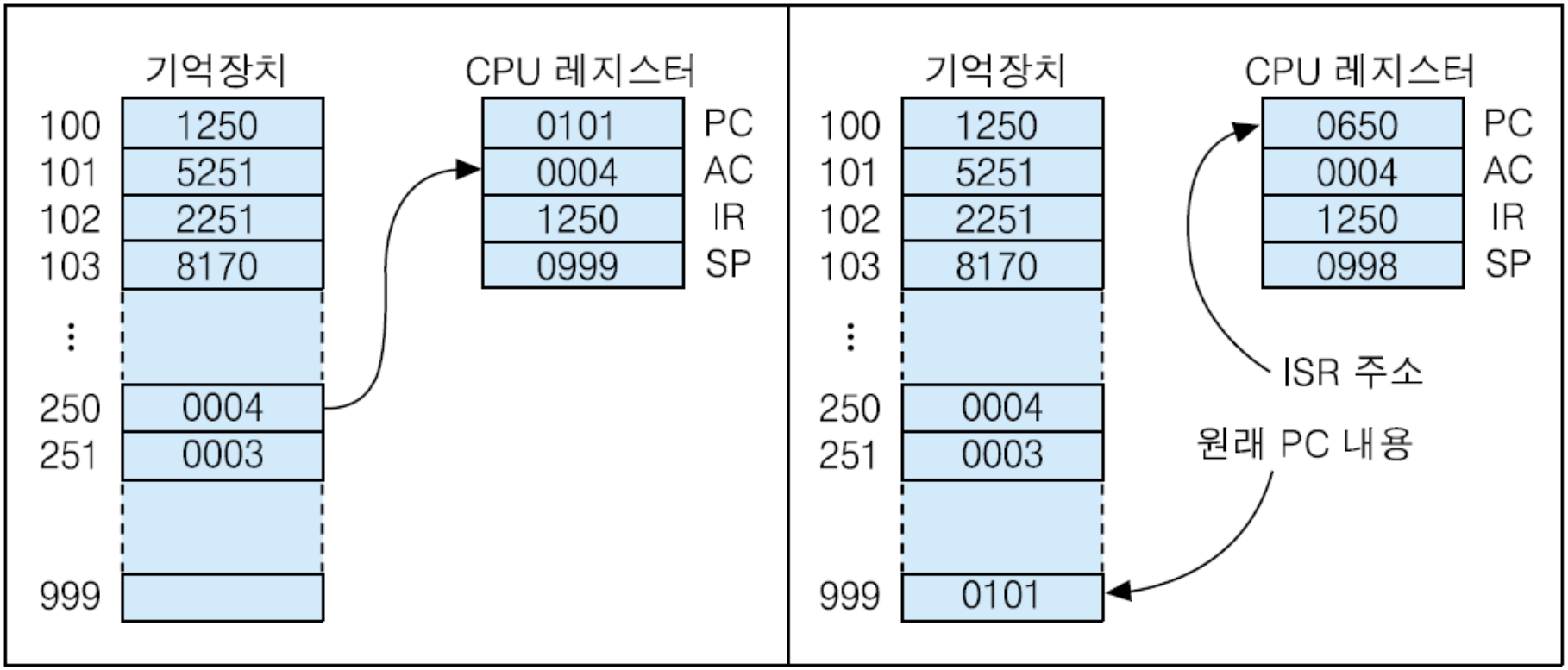
인터럽트 사이클의 마이크로 연산

- **[예]** 인터럽트 사이클의 마이크로 연산
  - 프로그램의 첫 번째 명령어인 LOAD 250 명령어가 실행되는 동안에 인터럽트가 들어왔으며, 현재 SP = 999이고, 인터럽트 서비스 루틴의 시작 주소는 650 번지라고 가정

100 LOAD 250
101 ADD 251
102 STA 251
103 JUMP 170

인터럽트 요구가 들어온 경우의 상태 변화

- SP = 999, ISR 시작 주소 = 650번지
  - ISR의 시작단계에서 레지스터들의 내용을 스택에 저장
  - ISR의 마지막 단계에서 레지스터들의 내용 복원



(a) LOAD 명령어의 실행 사이클이 종료된 상태 (b) 인터럽트 사이클이 종료된 상태

다중 인터럽트

다중 인터럽트(multiple interrupt)

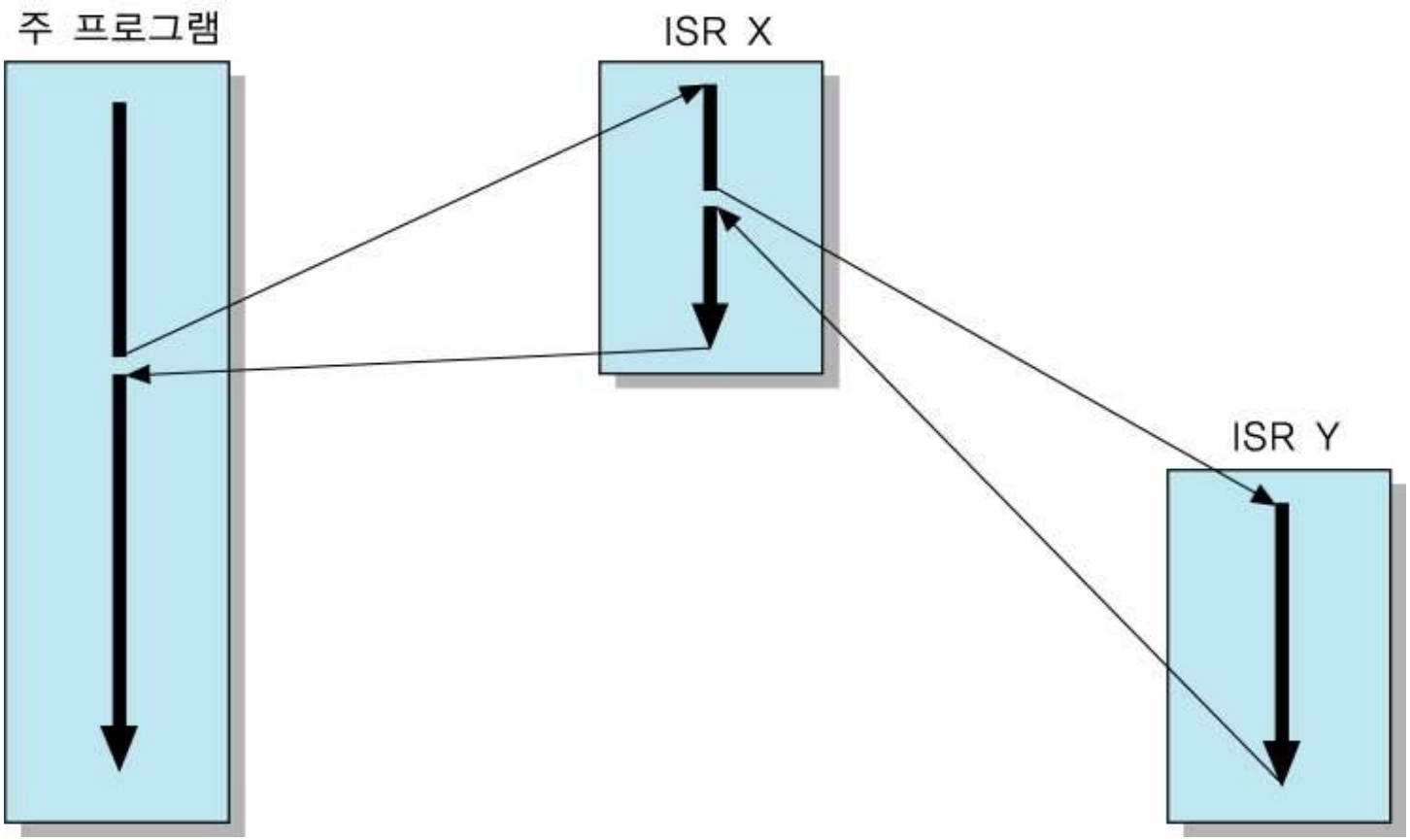
- 인터럽트 서비스 루틴을 수행하는 동안에 다른 인터럽트 발생

다중 인터럽트의 처리방법

- **[1]** CPU가 인터럽트 서비스 루틴을 처리하고 있는 도중에는 새로운 인터럽트 요구가 들어오더라도 인터럽트 사이클을 **수행하지 않는 방법**
  - 인터럽트 플래그(interrupt flag) ← 인터럽트 불가능(interrupt disabled) 상태
  - 시스템 운영상 중요한 프로그램이나 도중에 중단할 수 없는 데이터 입출력 동작 등을 위한 인터럽트를 처리하는데 사용
- **[2]** 인터럽트의 **우선순위 기반**
  - 우선순위가 낮은 인터럽트가 처리되고 있는 동안에 우선순위가 더 높은 인터럽트가 들어온다면, 현재의 인터럽트 서비스 루틴의 수행을 중단하고 새로운 인터럽트를 먼저 처리

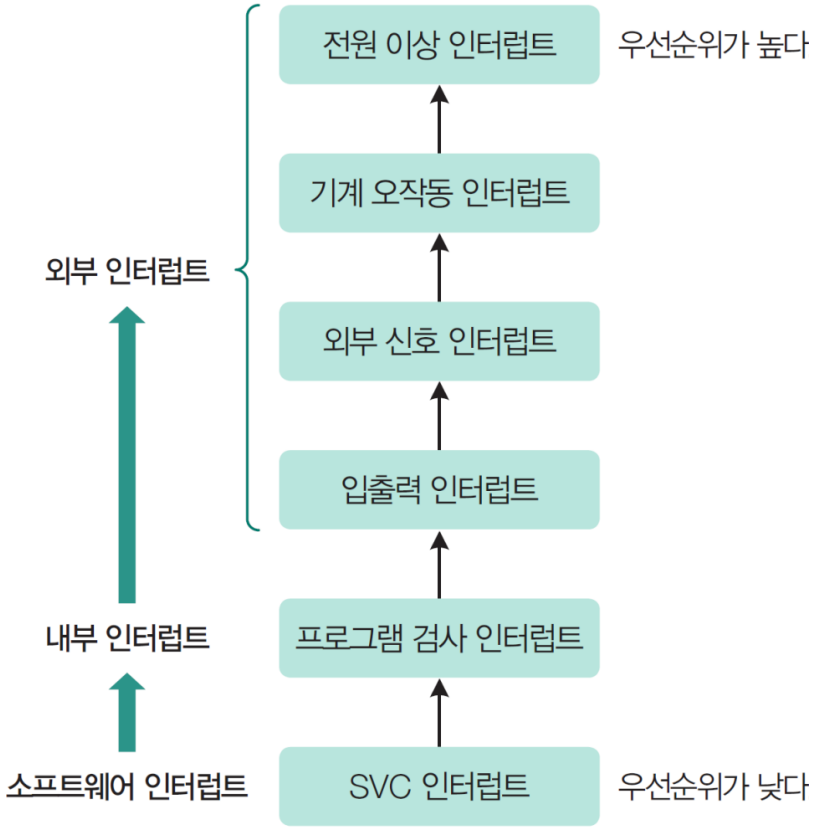
다중 인터럽트 처리 방법

- 장치 X를 위한 ISR X를 처리하는 도중에 우선 순위가 더 높은 장치 Y로부터 인터럽트 요구가 들어와서 먼저 처리되는 경우에 대한 제어의 흐름



[TMI] 인터럽트의 우선순위

- 외부 인터럽트 > 내부 인터럽트 > 소프트웨어 인터럽트
  - 외부 인터럽트: 전원, I/O 장치, 타이머 등 컴퓨터 외부 요인으로 발생
  - 내부 인터럽트: 잘못된 명령어나 데이터를 사용할 때, CPU 내부에서 발생
    - **[예]** 0으로 나누기, 오버플로우(overflow), 언더플로우(underflow), 잘못된 메모리 접근
  - 소프트웨어 인터럽트: 프로그램이 작업을 수행하기 위해 운영체제의 특정 기능을 요청할 때 발생
    - **[예]** SuperVisor Call(SVC): 사용자 모드에서 실행되는 응용 프로그램이 특정 자원에 접근하기 위해, 커널(kernel) 모드로 잠깐 전환을 요청하는 인터럽트

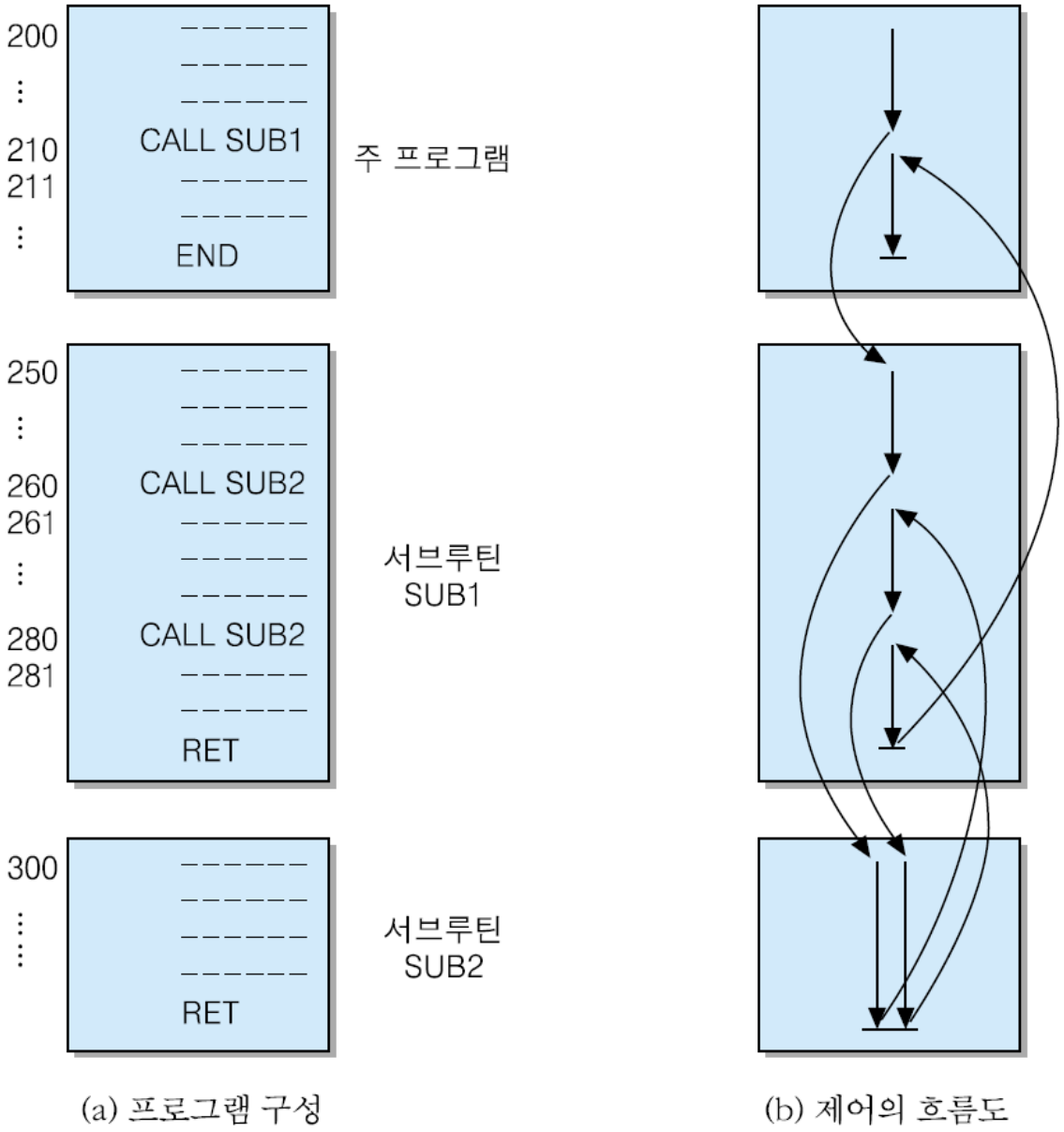


서브루틴 호출(subroutine call)

- **서브루틴 호출(subroutine call)**
  - 한 블록으로 구성된 명령어 실행 중에 또 다른 블록으로 구성된 명령어를 삽입하여 실행하기를 원하는 형태로 프로그램을 구성할 때 사용하는 명령어로서, 호출(CALL)과 복귀(RET) 명령어가 함께 사용
  - **CALL 명령어:** 현재의 PC 내용을 스택에 저장하고 서브루틴의 시작 주소로 분기하는 명령어
  - **RET 명령어:** CPU가 원래 실행하던 프로그램으로 복귀(return)시키는 명령어

- 단순히 실행 순서만 변경할 때 사용할 때는 분기(branch) 명령어 사용

서브루틴이 포함된 프로그램이 수행되는 순서



CALL/RET 명령어의 마이크로 연산

- **CALL X 명령어에 대한 마이크로 연산**

  t_0 : **MBR ← PC**

  t_1 : **MAR ← SP, PC ← X**

  t_2 : **M[MAR]← MBR, SP ← SP - 1**

  - 현재의 PC 내용(서브루틴 수행 완료 후에 복귀할 주소)을 SP가 지정하는 스택의 최상위(Top Of Stack, TOS)에 저장

- **RET 명령어의 마이크로 연산**

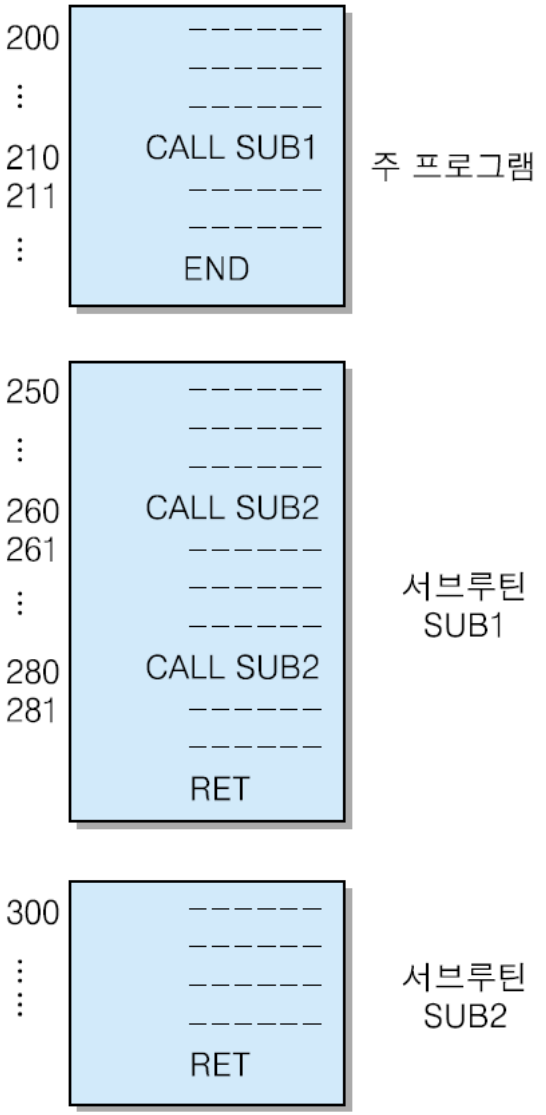
  t_0 : **SP ← SP + 1**

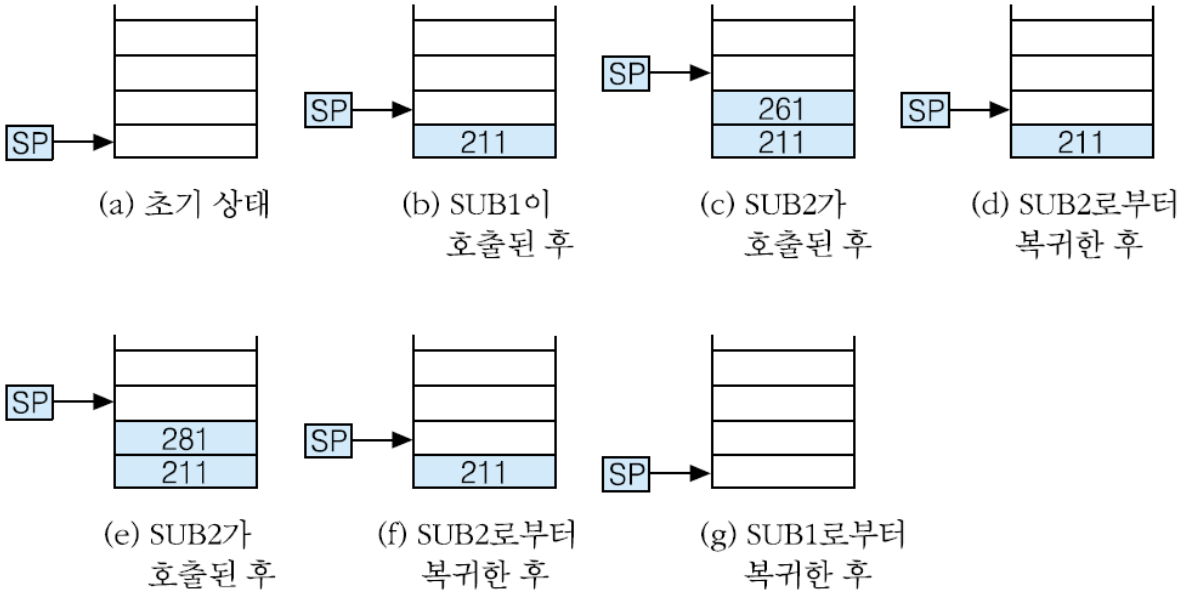
  t_1 : **MAR ← SP**

  t_2 : **MBR ← M[MAR]**

  t_3 : **PC ← MBR**

프로그램 수행 과정에서 스택의 변화





Q & A

End!