[CS Study] 1주차

: CPU 레지스터 및 명령어 사이클

0. 목차

---

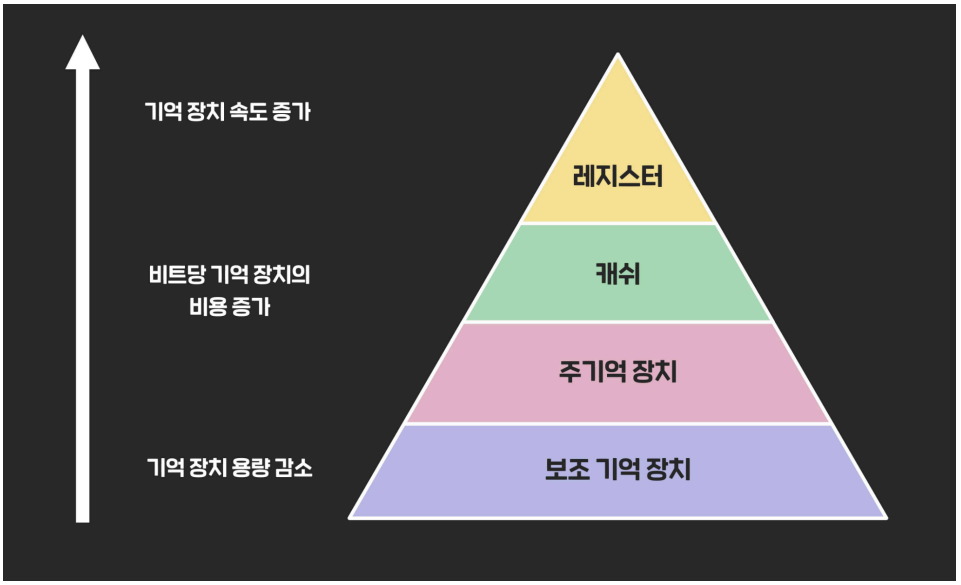
1. 레지스터(Register) 개요

레지스터는 CPU 내부에 존재하는 최고속 기억장치이다. CPU가 연산 과정에서 필요한 데이터와 상태 정보를 임시 저장하며, 메모리 계층 구조의 최상단에 위치한다.

1.1 메모리 계층 구조의 목적

메모리 계층 구조(Memory Hierarchy)는 '속도, 용량, 비용' 사이의 트레이드오프를 해결하여 시스템의 전체 성능을 최적화하는 데 목적이 있다.

- 성능 극대화: CPU의 처리 속도와 메인 메모리(RAM)의 접근 속도 사이에는 커다란 간극이 존재한다. 레지스터와 캐시를 통해 이 속도 차이를 보완함으로써 CPU가 데이터 대기 시간 없이 연산을 지속할 수 있게 한다.
- 경제성 확보: 속도가 빠른 저장장치는 비싸고 용량이 적으며, 느린 장치는 저렴하고 용량이 크다. 자주 사용하는 데이터는 상위 계층(레지스터)에, 그렇지 않은 데이터는 하위 계층(HDD/SSD)에 배치하여 비용 대비 효율적인 저장 시스템을 구축한다.



1.2 레지스터의 역할

CPU는 연산 시 RAM에 직접 접근하지 않고, 실행에 즉각 필요한 데이터를 레지스터로 인출하여 처리한다. 이는 프로세서가 하드웨어 수준에서 낼 수 있는 최대의 연산 효율을 이끌어내기 위한 필수적인 과정이다.

---

2. 레지스터의 종류와 역할

CPU는 용도에 따라 다양한 레지스터를 사용하여 데이터를 처리한다. 각 레지스터의 구체적인 명칭과 실제 사용 사례는 다음과 같다.

2.1 프로그램 카운터 **(PC, Program Counter)**

- 역할: 다음에 인출하여 실행할 명령어의 메모리 주소를 보관한다.
- 사용법: CPU가 메모리에서 명령어를 가져올 때 이 레지스터를 참조한다. 일반적으로는 순차적으로 주소가 증가하나, `if`문이나 `while`문 같은 조건문, 또는 함수 호출(`CALL`)이 발생하면 실행 흐름을 바꾸기 위해 목적지 주소로 값이 직접 갱신된다.

2.2 명령어 레지스터 **(IR, Instruction Register)**

- 역할: 메모리에서 인출한 '현재 실행 중인 명령어'를 임시 보관한다.
- 사용법: 제어장치(CU)는 IR에 담긴 비트 데이터를 해독하여 연산 종류(더하기, 저장하기 등)를 파악하고 각 하드웨어 부품에 제어 신호를 보낸다.

2.3 메모리 주소 레지스터 **(MAR, Memory Address Register)**

- 역할: CPU가 데이터를 읽거나 쓰기 위해 접근하려는 메모리의 주소를 보관한다.
- 사용법: 주소 버스로 값을 내보내기 전의 대기 공간이다. 예를 들어 `0x100` 주소의 데이터를 읽고 싶다면 MAR에 `0x100`을 저장한 뒤 메모리에 읽기 신호를 보낸다.
- 주의점**:** CPU는 하드웨어 설계상 직접 메모리에 접근할 수 없기 때문에 꼭 레지스터와 같은 중간 매개체를 이용해서 접근해야 한다.

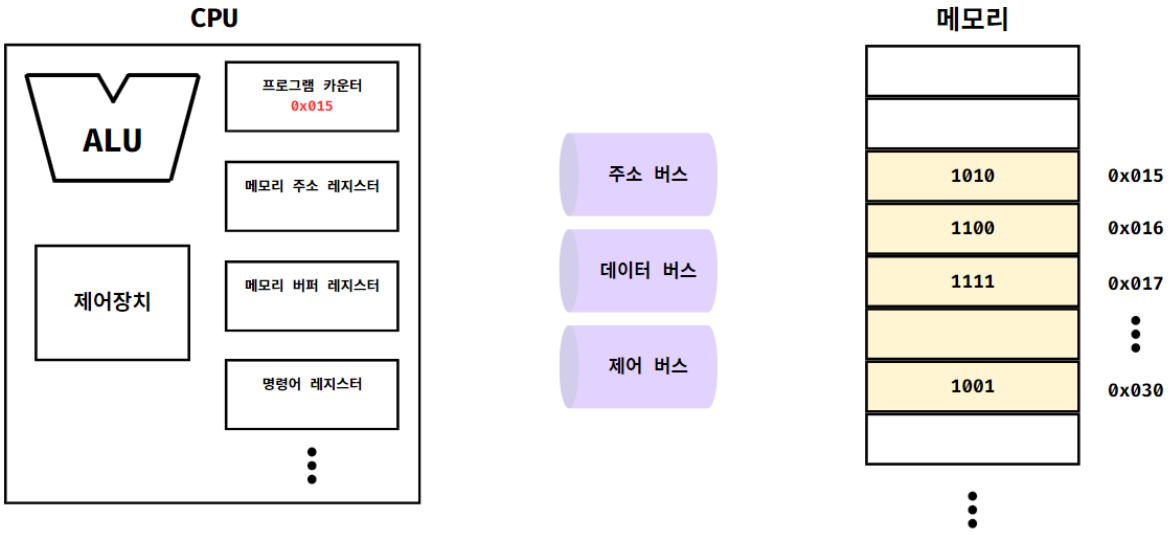
2.4 메모리 버퍼 레지스터 **(MBR, Memory Buffer Register)**

- 역할: 메모리와 주고받는 실제 데이터 또는 명령어를 임시 보관한다.
- 사용법: 데이터 버스의 통로 역할을 한다. 메모리에서 읽어온 값이 CPU 내부로 들어오거나, 연산 결과값을 메모리에 쓰기 위해 나갈 때 반드시 거쳐 가는 공간이다.

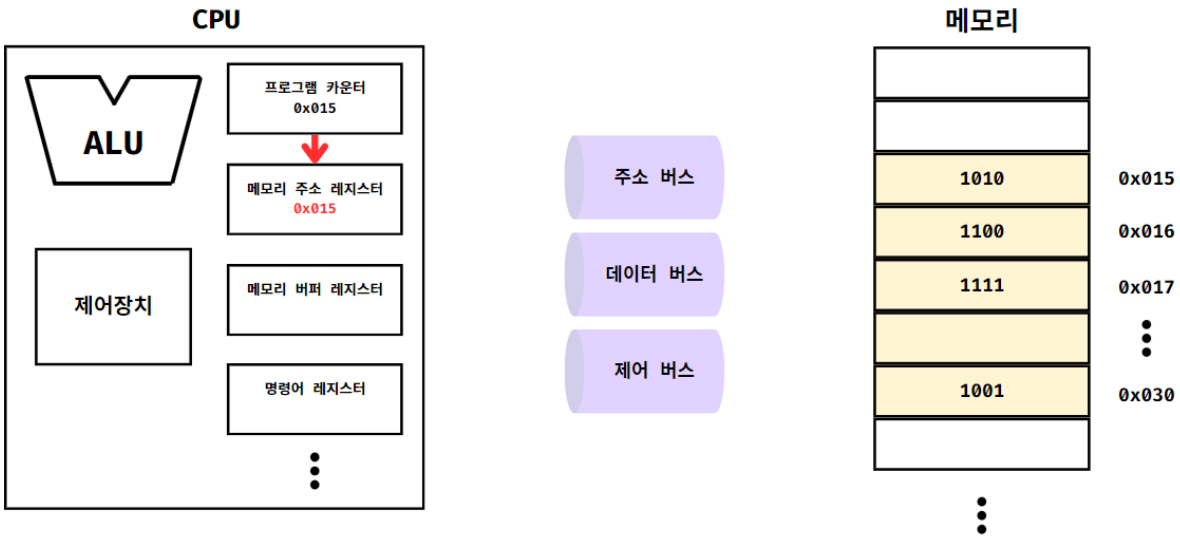
[참고] 실제 프로그램 실행 과정 (데이터 흐름 중심)

위의 2.1~2.4 레지스터들이 실제 명령어 실행 시 움직이는 순서는 다음과 같다.

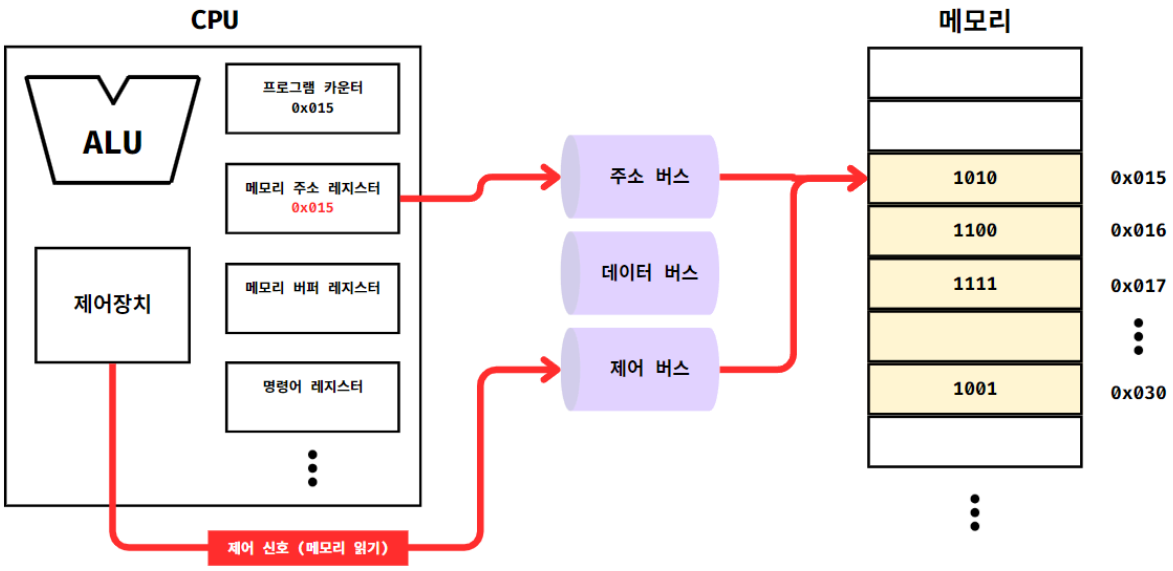
- 단계 **1** (실행 시작점 설정): 프로그램을 실행하기 위해 **PC**에 프로그램의 시작 주소를 저장한다.



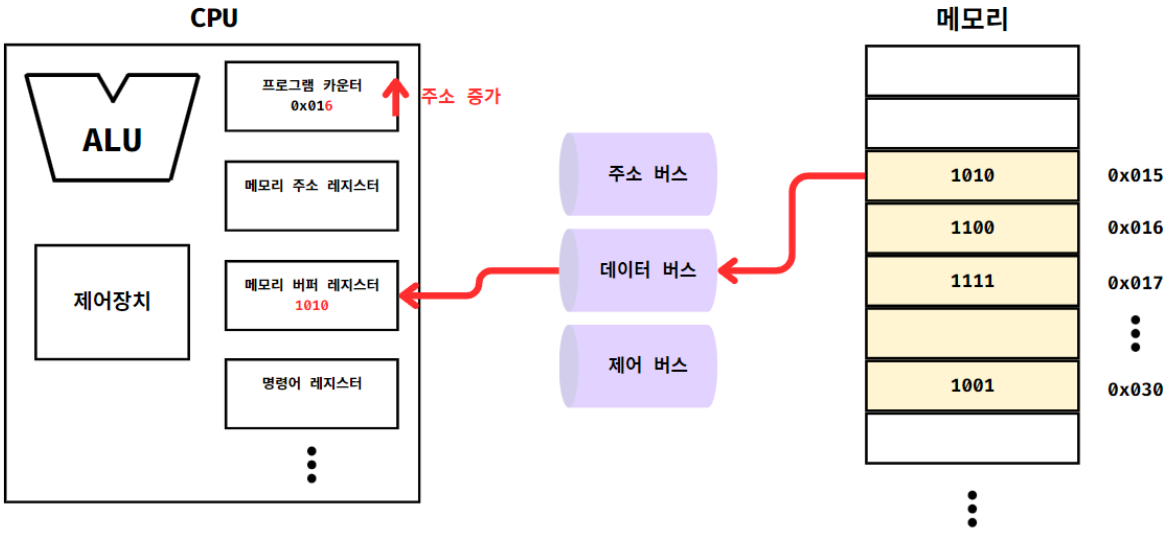
- 단계 **2** (메모리 주소 지정): 주소 `0x015`에 접근하기 위해 **MAR**에 해당 주소를 저장한다.



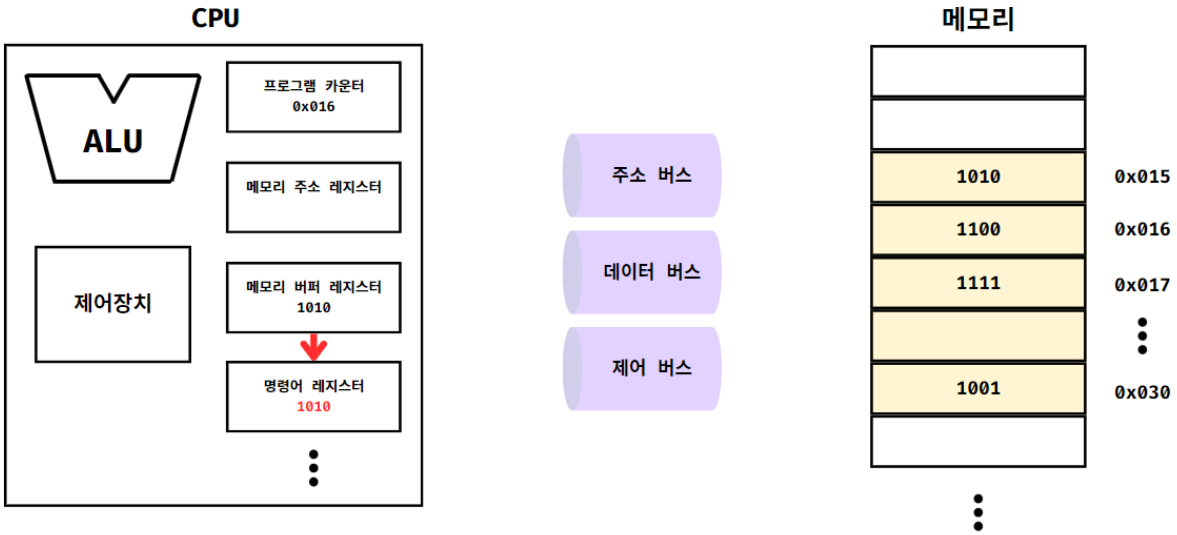
- 단계 **3 (**신호 송출**)**: 제어 신호와 **MAR**의 주소값을 각각 제어 버스와 주소 버스를 통해 메모리로 전달한다.



- 단계 **4 (**데이터 수신**)**: 메모리에서 읽어온 값을 데이터 버스를 통해 **MBR**에 저장하고, **PC** 값을 증가시킨다.



- 단계 **5 (**해석 및 제어**)**: **MBR**의 값을 **IR**로 이동시킨 후, **제어장치(CU)**가 명령어를 해석하고 제어 신호를 발생시킨다.



2.5 플래그 레지스터 (FR, Flag Register)

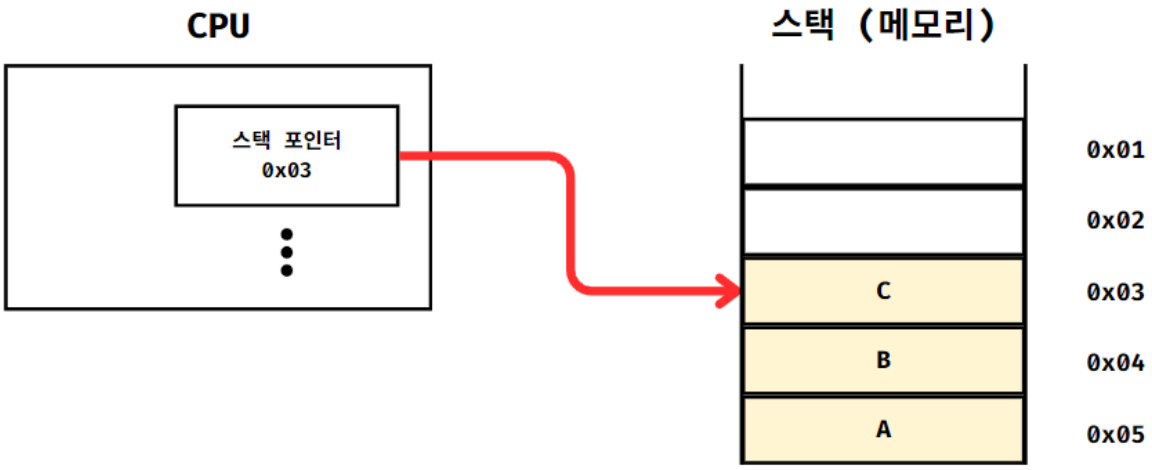
- 역할: ALU 연산 결과에 대한 상태 정보나 CPU의 현재 상태를 비트 단위로 저장한다.
- 사용법: 연산 결과가 0이면 '제로 플래그(ZF)'를 1로 세팅하고, 음수가 나오면 '부호 플래그(SF)'를 세팅한다. 이후의 분기 명령어는 이 플래그 값을 보고 다음 동작을 결정한다.

2.6 범용 레지스터 (General-Purpose Register)

- 역할: 데이터와 주소를 모두 저장할 수 있는 다목적 저장 공간이다.
- 예시: x86 아키텍처의 `EAX`, `EBX`, `ECX` 등
- 사용법: x86 구조의 `EAX`, `EBX` 등이 대표적이다. 함수 내에서 덧셈 연산을 수행할 때 변수 값을 잠시 담아두거나, 연산 결과를 저장하여 다른 연산에 전달하는 용도로 가장 빈번하게 사용된다.

2.7 스택 포인터 (SP, Stack Pointer)

- 역할: 메모리의 스택 영역에서 가장 최근에 데이터가 쌓인 지점(최상단)을 가리킨다.
- 사용법: 함수가 호출될 때 복귀 주소를 저장하거나, 지역 변수를 위한 메모리 공간을 확보할 때 SP 값을 조정하여 현재 스택의 위치를 관리한다.



2.8 베이스 레지스터 **(BP, Base Pointer)**

- 역할: 현재 실행 중인 함수의 스택 프레임 시작 주소를 보관하거나, **"변위 주소 지정 방식"**에서 기준 주소 역할을 수행한다.
- 예시: 배열의 시작 주소를 베이스 레지스터에 담고, 인덱스(변위)를 더해 특정 요소에 접근하는 상황.
- 사용법:
    - 변위 주소 지정 방식**(Displacement Addressing Mode)**: 오퍼랜드 필드의 값(변위)과 베이스 레지스터의 값을 더하여 실제 데이터가 위치한 유효 주소를 결정한다.
    - 함수 내에서 매개변수나 지역 변수에 접근할 때 기준점으로 활용하며, 함수 실행 중에는 값이 변하지 않아 안정적인 주소 계산을 돕는다.

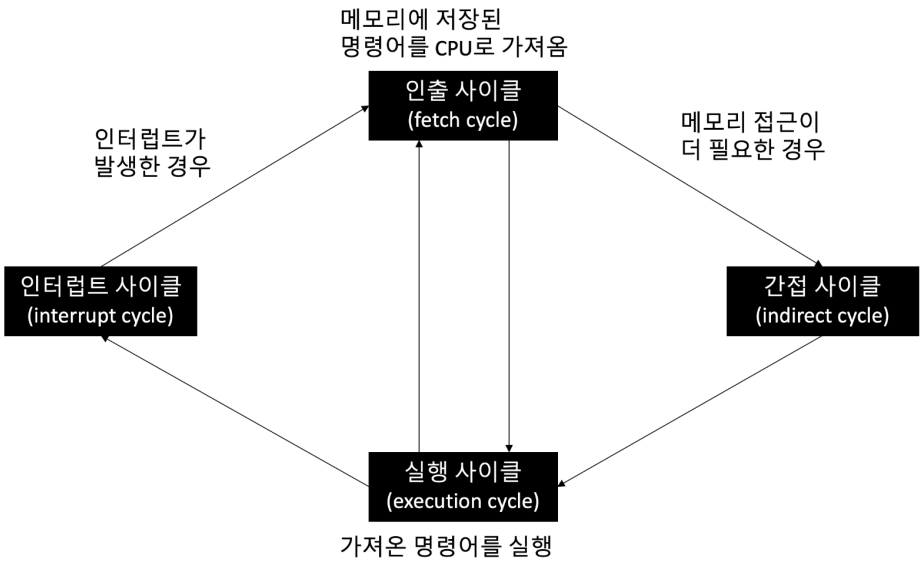
구분	상대 주소 지정법 **(Relative)**	베이스 레지스터 주소 지정법 **(Base)**
기준 레지스터	**PC** (Program Counter)	**BP** (Base Pointer) / 범용 레지스터
계산 공식	유효 주소 = PC + 변위	유효 주소 = Base Register + 변위
주요 용도	실행 흐름 제어 (Jump, Branch)	데이터 구조 접근 (Array, Local Variables)
주요 이점	코드 재배치가 용이함 (위치 독립적 코드)	데이터 영역 분리 및 관리가 용이함

출처 : https://nijy.tistory.com/153

---

3. 명령어 사이클 (Instruction Cycle)

명령어 사이클은 CPU가 하나의 명령어를 처리하는 전체 과정을 의미하며, 다음의 4가지 세부 사이클로 구성된다.

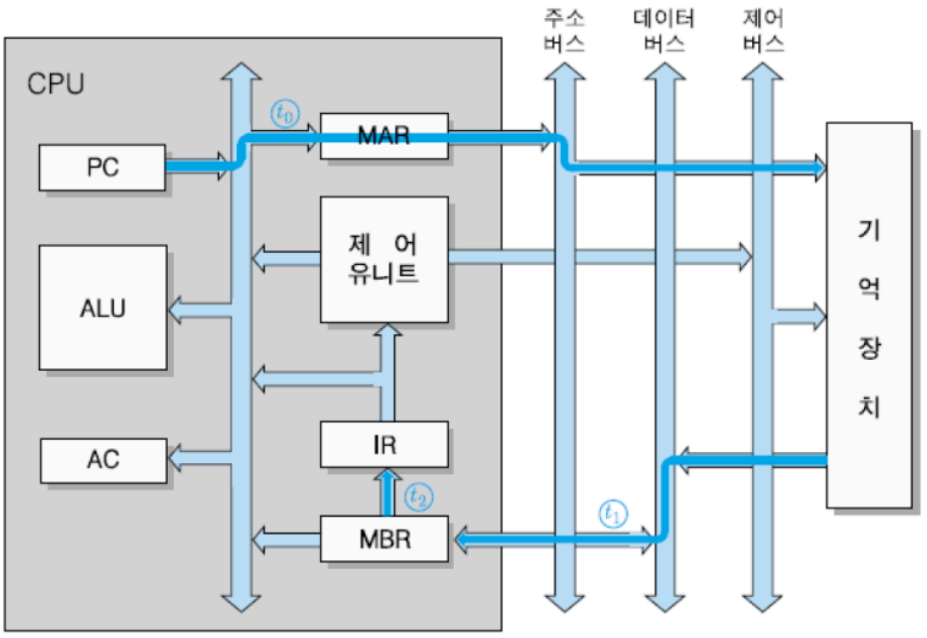


3.1 인출 사이클 (Fetch Cycle)

PC의 주소를 **MAR**로 전달한다.

메모리의 해당 주소에서 명령어를 읽어 **MBR**에 저장한다.

MBR의 내용을 **IR**로 이동시키고 **PC** 값을 증가시킨다.



3.2 간접 사이클 **(Indirect Cycle)**

- 명령어의 주소 지정 방식이 간접 방식일 때 수행된다.
- 유효 주소를 얻기 위해 메모리에 추가적으로 접근하는 단계이다.

3.3 실행 사이클 **(Execution Cycle)**

- **IR**에 저장된 명령어를 제어장치가 해독(Decode)하고 연산을 수행한다.
- 데이터 이동, 산술/논리 연산 등이 실제로 일어나는 단계이다.

3.4 인터럽트 사이클 **(Interrupt Cycle)**

- 실행 완료 후 인터럽트 발생 여부를 확인한다.
- 발생 시 현재 상태(PC 등)를 스택에 저장하고 인터럽트 서비스 루틴으로 분기한다.

---

4. 명령어 인터럽트 사이클 **(Interrupt Cycle)**

CPU는 하나의 명령어를 실행 완료할 때마다 외부 혹은 내부에서 발생한 중단 신호를 확인하며, 이를 인터럽트 사이클이라 한다. 이 과정은 시스템의 유연한 대응과 멀티태스킹 구현을 위한 핵심 기법이다.

4.1 인터럽트**(Interrupt)**의 정의

CPU가 프로그램을 수행 중인 도중, 예기치 못한 상황이나 외부 장치의 요청으로 인해 현재 처리 순서를 잠시 중단시키도록 하는 신호를 의미한다. CPU는 이 신호를 받으면 현재 작업을 안전하게 보관한 뒤, 우선순위가 높은 긴급 작업을 먼저 처리한다.

4.2 인터럽트의 주요 종류

발생 원인과 CPU 동작과의 동기 여부에 따라 다음과 같이 분류한다.

- 예외 **(Exception,** 동기 인터럽트**)**
  - 정의: CPU 내부에서 명령어들을 수행하다가 예상치 못한 상황이 발생했을 때 나타나는 인터럽트이다.
  - 특징: 명령어 실행 흐름과 동기적으로 발생하며, 주로 프로그래밍상의 오류와 관련이 깊다.
  - 예시: 0으로 나누기(Divide-by-Zero), 존재하지 않는 메모리 주소 참조, 권한 위반 등.
- 인터럽트 **(Interrupt,** 비동기 인터럽트**)**
  - 정의: 주로 입출력(I/O) 장치 등 CPU 외부에서 발생하는 알림 성격의 인터럽트이다.
  - 특징: CPU의 현재 실행 흐름과 관계없이 비동기적으로 발생한다.
  - 예시: 키보드/마우스 입력 알림, 타이머 종료, 하드웨어 오류 알림 등.
- 소프트웨어 인터럽트 **(Software Interrupt)**
  - 정의: 프로그램 코드 내에서 특정 명령을 실행하여 의도적으로 발생시키는 인터럽트이다.
  - 용도: 사용자 모드에서 커널 모드의 기능을 사용하기 위한 시스템 콜**(System Call)** 호출 시 주로 사용된다.

4.3 인터럽트 서비스 루틴 **(ISR, Interrupt Service Routine)**

인터럽트를 처리하기 위해 수행하는 특수한 프로그램 루틴으로, '인터럽트 핸들러(Interrupt Handler)'라고도 불린다.

- 동작 원리: 인터럽트가 발생하면 CPU는 현재 실행 중인 프로그램의 상태를 스택에 보존하고, 해당 인터럽트에 대응하는 ISR로 점프한다.
- 인터럽트 벡터**(Interrupt Vector)**: 여러 종류의 인터럽트가 존재하므로, 각 인터럽트마다 실행해야 할 ISR의 시작 주소를 담고 있는 테이블을 참조하여 정확한 루틴을 찾아간다.

4.4 인터럽트 사이클의 실행 단계

전체 명령어 사이클 내에서 인터럽트 처리는 다음과 같은 순서로 진행된다.

1. 인터럽트 확인: 실행 사이클(Execution Cycle) 종료 직후, CPU가 인터럽트 요청 신호를 검사한다.
2. 상태 저장 **(Context Save)**: 인터럽트가 감지되면, 현재 **PC(Program Counter)** 값과 주요 상태 정보를 스택에 저장한다.
3. **ISR** 분기: 인터럽트 벡터를 통해 해당 **ISR**의 시작 주소를 **PC**에 로드한다.
4. 루틴 실행: 해당 인터럽트를 처리하는 코드를 실행한다.
5. 복구 **(Context Restore)**: 처리가 완료되면 스택에 보관했던 주소를 다시 **PC**로 복원하여 중단된 지점부터 실행을 재개한다.

---

5. CPU 설계 아키텍처: CISC vs RISC

CPU가 명령어를 해석하고 실행하는 방식에 따라 크게 두 가지 설계 철학으로 나뉜다.

5.1 CISC (Complex Instruction Set Computer)

- 철학: "명령어 집합을 복잡하고 강력하게 만들어, 소프트웨어의 작성을 편리하게 하자."
- 특징:
    - 가변 길이 명령어: 명령어의 크기가 제각각이다.
    - 복잡한 주소 지정: 하나의 명령어로 메모리 접근과 연산을 동시에 수행할 수 있다.
    - 낮은 레지스터 의존도: 메모리에 직접 접근하는 명령어가 많아 레지스터 개수가 상대적으로 적다.
- 장점: 컴파일된 코드의 크기가 작아 메모리를 절약할 수 있다. (하위 호환성이 뛰어남)
- 단점: 명령어 구조가 복잡해 파이프라이닝**(Pipelining)** 효율을 높이기 어렵고, 전력 소모가 크다.
- 대표 모델: Intel x86 계열.

5.2 RISC (Reduced Instruction Set Computer)

- 철학: "명령어를 단순화하고 규격화하여, 하드웨어 실행 속도를 극한으로 높이자."
- 특징:
    - 고정 길이 명령어: 모든 명령어의 크기가 일정하여 해석 속도가 빠르다.
    - **Load/Store** 구조: 메모리 접근은 전용 명령어로만 수행하며, 모든 연산은 레지스터에서 이루어진다.
    - 많은 레지스터: 연산 효율을 위해 다수의 범용 레지스터를 보유한다.
- 장점: 명령어 구조가 단순하여 파이프라이닝 최적화에 유리하며, 전력 효율이 매우 높다.
- 단점: CISC보다 코드의 줄 수가 늘어나며, 컴파일러의 역할이 매우 중요하다.
- 대표 모델: ARM(Apple Silicon), RISC-V.

5.3 현대 아키텍처의 융합 (CISC의 RISC화)

현대의 Intel/AMD CPU는 겉으로는 CISC 방식을 유지하지만(호환성), 내부적으로는 명령어를 **μ-ops(마이크로 연산)**라는 작은 RISC 형태로 쪼개서 처리한다.

- 목적: CISC의 풍부한 소프트웨어 생태계를 유지하면서, RISC의 강력한 파이프라이닝 성능을 얻기 위함이다.
- 한계: 명령어를 쪼개는 디코더**(Decoder)** 하드웨어가 추가로 필요하므로, 이 과정에서 발생하는 발열과 전력 소모가 ARM(RISC) 대비 불리한 요인이 된다.
