Week 2.

04 CPU의 작동원리

ALU와 제어장치

- **CPU** : 메모리에 저장된 명령어를 읽고 해석하고 실행하는 장치
- **ALU** : 계산 담당
- **제어장치** : 명령어 읽고 해석
- **레지스터** : 작은 임시 저장 장치

✔️ ALU

1. **받아들이는 정보**
   → 레지스터를 통해
   피연산자
   → 제어장치를 통해
   제어 신호
2. **내보내는 정보**

   → 연산 결과
   →
   플래그 : 연산에 대한 추가적인 정보

💡 플래그는 (거의) 0과 1로 표현

💡 ALU 내부에는 계산을 위한 회로가 다양하게 존재 : 가산기, 보수기, 시프터, 오버플로우 검출기 등 ..

▼ **Q. ALU 결과값을 레지스터에 저장하는 이유는?**

CPU가 메모리에 직접 접근하려면 속도가 오래 걸리고 이로 인해 프로그램 속도까지 늦어지는 영향이 있음 → 레지스터는 CPU 내부에 있기 때문에 레지스터로 관리하면 속도가 빨라짐

✔️ 제어장치

➡️ 제어 신호를 내보내고, 명령어를 해석하는 부품

*제어 신호 : 컴퓨터 부품들을 관리하고 작동시키기 위한 일종의 전기 신호

[제어장치가 받아들이는 정보]

- **클럭 신호** : 컴퓨터의 모든 부품을 일사분란하게 움직일 수 있게 하는 시간 단위
- 현재 수행할 명령어 → 명령어 레지스터에 저장
- 플래그 (플래그 레지스터)
- 제어 신호

▼ **Q. 컴퓨터의 모든 부품이 한 클럭 마다 작동한다. (O, X)**

X
컴퓨터 부품들이 클럭이라는 박자에 맞춰 작동할 뿐 한 박자 마다 작동하는 것은 아님!

레지스터

[반드시 알아야 할 레지스터]

- 프로그램 카운터 (=명령어 포인터)
- 명령어 레지스터
- 메모리 주소 레지스터
- 메모리 버퍼 레지스터 (=메모리 데이터 레지스터)
- 플래그 레지스터
- 범용 레지스터 : 데이터 + 주소 모두 저장 가능

✔️ **프로그램 카운터가 꾸준히 증가**하기 때문에 프로그램이 순차적으로 계속 실행

💡 순차적인 실행이 끊기는 경우를 **'인터럽트'가 발생**했다고 함

ex. 주소를 강제적으로 이동시키는 명령어를 사용한 경우

[특정 레지스터를 이용한 주소 지정 방식]

1. **스택 주소 지정 방식**
   : 스택 + 스택 포인터 이용
   *스택 포인터 : 스택의 꼭대기를 가리키는 레지스터 (스택이 어디까지 채워졌는지 표시)
   ✔️ 스택이
   LIFO 방식이라는 것을 알아두기

2. **변위 주소 지정 방식**

   : 오퍼랜드 필드의 값(변위) + 특정 레지스터의 값 → 유효 주소 get.

   (1) 상대 주소 지정 방식
   → 오퍼랜드 + 프로그램 카운터 = 유효 주소 get
   : 분기하여 특정 주소의 코드를 실행할 때 사용

   (2) 베이스 레지스터 주소 지정 방식
   → 오퍼랜드 + 베이스 레지스터 값 = 유효 주소 get
   베이스 레지스터 : 기준 주소 / 오퍼랜드 : 기준 주소로부터 떨어진 거리
   : 얼마나 떨어진 주소에 접근할 것인지 연산하여 유효 주소 get

명령어 사이클과 인터럽트

- 명령어 사이클 : 하나의 명령어를 처리하는 정형화된 흐름
- 인터럽트 : 명령어 사이클 흐름이 끊어지는 상황

1. **명령어 사이클**
   → 프로그램 속 각각의 명령어들은 일정한 주기가 반복되며 실행

   인출 사이클 - 실행 사이클 (+ 간접 사이클)
   : 프로그램을 이루는 수많은 명령어는 인출, 실행 사이클을 반복하면서 실행
   : 간접 사이클은 명령어 실행을 위해 메모리에 한 번 더 접근해야하는 경우 실행하는 단계

2. **인터럽트**

   → CPU가 수행 중인 작업을 방해하는 신호

- **동기 인터럽트 (=예외 Exception)** : CPU에 의해 발생
- **비동기 인터럽트 (하드웨어 인터럽트)** : 입출력장치에 의해 발생

[하드웨어 인터럽트 처리 순서]

1. 인터럽트 요청 신호 전송 - (입출력장치)
2. 인터럽트 여부 확인 - (CPU)
3. 인터럽트 수용 가능 여부 확인 - (인터럽트 플래그)
4. 수용 가능할 시, 작업 백업 - (CPU, 스택에 백업)
5. 인터럽트 서비스 루틴 실행 - (인터럽트 벡터)
6. 완료 후, 백업해둔 작업 실행 재개

05 CPU 성능 향상 기법

> 컴퓨터 부품들은 **클럭 신호** 에 맞춰 일사불란하게 움직임
>
> CPU는 **명령어 사이클** 이라는 정해진 흐름에 맞춰 명령어들을 실행

✔️ 빠른 CPU를 위한 설계 기법

1. **클럭**
   → 클럭 속도가 높아지면 CPU는 명령어 사이클을 더 빠르게 반복
   → 일반적으로 클럭 속도가 높은 CPU는 성능이 좋음

- **클럭 속도** : 1초에 클럭이 몇 번 반복되는지 나타냄 (Hz헤르츠 단위)

⭐ 클럭 속도를 높인다고 무조건 CPU가 빨라지는 것 X

⭐ 클럭 속도만으로 CPU의 성능을 올리는 것은 한계

2. **코어와 멀티코어**

→ CPU의 코어, 스레드 수를 늘리면 성능 향상

[코어 늘리는 방법]

코어 : '명령어를 실행하는 부품'

과거 CPU의 정의였지만 현대에는 '코어'가 그 의미를 상징
오늘 날의 CPU는 '명령어를 실행하는 부품을 여러 개 포함하는 부품'으로 확장

✔️ **멀티코어 CPU(멀티코어 프로세서)** : 코어를 여러 개 포함하고 있는 CPU

> 💡 **CPU의 연산 속도는 코어 수에 비례하여 증가 X**
> → 코어마다 처리할 연산이 적절히 분배되어야 속도가 증가
> ex. 조별 과제 ..

3. **스레드와 멀티스레드**

- **하드웨어적 스레드** (CPU 입장에서 정의)
  → 하나의
  코어가 동시에 처리하는 명령어 단위

  ⭐ 멀티스레드 CPU(멀티스레드 CPU)
  : 하나의 코어로 여러 명령어를 동시에 처리하는 CPU
  ex. 8코어 16스레드

- **소프트웨어적 스레드** (소프트웨어적으로 정의)
  → 하나의
  프로그램에서 독립적으로 실행되는 단위

- **멀티스레드 프로세서**
  → 하나의 코어로 여러 명령어를 동시에 처리하는 CPU
  ⭐ 멀티스레드 프로세서의 가장 큰 핵심은
  레지스터

[정리]

> 코어 : 명령어를 실행할 수 있는 하드웨어 부품
>
> 스레드 : 명령어를 실행하는 단위
>
> 멀티코어 프로세서 : 명령어를 실행할 수 있는 하드웨어 부품이 CPU 안에 두 개 이상 있는 CPU
>
> 멀티스레드 프로세서 : 하나의 코어로 여러 개의 명령어를 동시에 실행할 수 있는 CPU

명령어 병렬 처리 기법

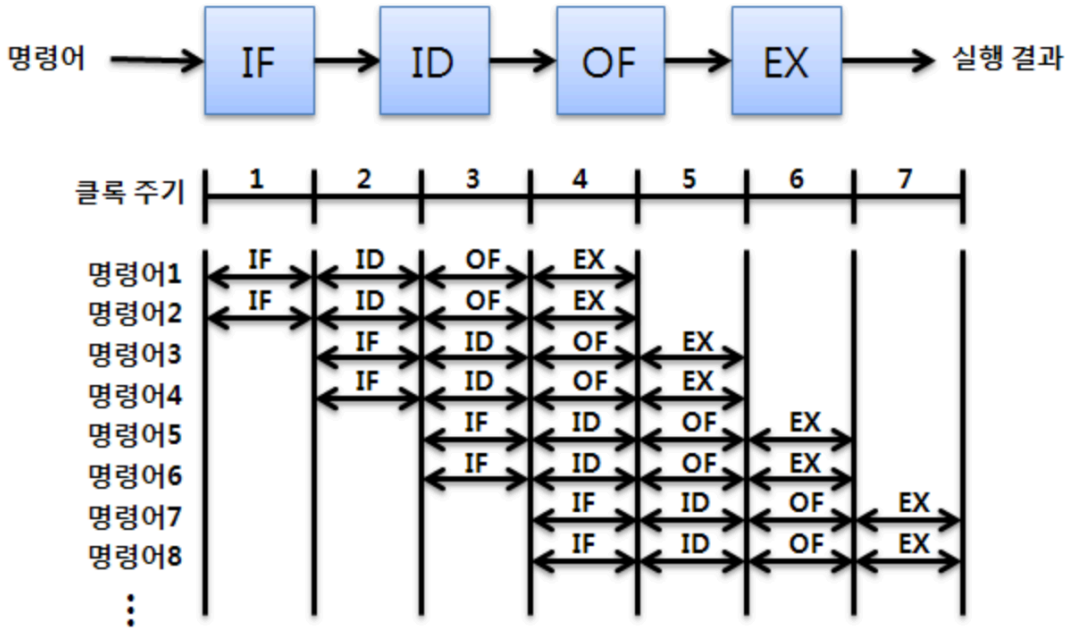
→ 명령어를 동시에 처리하여 CPU를 최대한 활용하는 기법

명령어 파이프 라인

: 클럭 단위로 나눠보기

> 명령어 인출(Instruction Fetch)
>
> 명령어 해석(Instruction Decode)
>
> 명령어 실행(Execute Instruction)
>
> 결과 저장(Write Back)
>
> 중요 : 같은 단계가 겹치지 않으면, CPU는 **'각 단계를 동시에 실행 가능'**

- **명령어 파이프라이닝** : 마치 공장 생산 라인과 같이 명령어들을 명령어 파이프라인에 넣고 동시에 처리하는 기법



⚠️ 파이프라인 위험

- **데이터 위험** : 명령어 간의 의존성에 의해 생기는 문제
- **제어 위험** : 프로그램 카운터의 갑작스러운 변화
- **구조적 위험** : 서로 다른 명령어가 같은 CPU부품(ALU,레지스터)를 쓰려고 할 때

슈퍼스칼라

→ 여러 개의 명령어 파이프라인을 포함한 구조

공장 생산 라인을 여러 개 두는 것이라고 생각하면 됨

💡 이론적으로 파이프라인 개수에 비례하여 프로그램 처리 속도가 빨라짐

but, 파이프라인 위험 등의 예상치 못한 문제로 인해 실제로 반드시 파이프라인 개수에 비례하여 빨라지는 것은 아님 !

➡️ 슈퍼스칼라 방식을 차용한 CPU는 파이프라인 위험 방지를 위해 고도로 설계되어야 함

비순차적 명령어 처리

→ 명령어들을 순차적으로 실행하지 않는 기법 (OooE)

> **명령어의** 합법적인 새치기
> 의존성이 없는 명령어의 순서를 바꿔서 실행

> → 파이프 라인이 멈추는 것을 방지하는 기법

CISC와 RISC

명령어 집합(ISA)
→ CPU가 이해할 수 있는 명령어들의 모음

ISA는 CPU의 언어이자 하드웨어가 소프트웨어를 어떻게 이해할지에 대한 약속

CISC Complex Instruction Set Computer

: 복잡한 명령어 집합을 활용하는 컴퓨터를 의미

가변 길이 명령어 : 명령어의 형태와 크기가 다양

⭐ 프로그램을 실행하는 명령어 수가 적다는 말은 **'컴파일된 프로그램의 크기가 작다'는 것을 의미**

- **명령어 파이프라이닝이 불리하다는 치명적인 단점**
  명령어의 크기와 실행되기 전까지의 시간이 일정 X
  대다수의 복잡한 명령어는 사용 빈도가 낮음

✔️ CISC이 주는 교훈

- 빠른 처리를 위해 명령어 파이프라인을 십분 활용해야 함.
- 원활한 파이프라이닝을 위해 **'명령어 길이와 수행 시간이 짧고 규격화'** 되어 있어야함
- 어차피 자주 쓰이는 명령어만 줄곧 사용
- 복잡한 기능 추가보다는 **'자주 쓰이는 기본 명령어를 작고 빠르게 만드는 것이 중요'**

RISC Reduced Instruction Set Computer

- CISC에 비해 명령어 수가 적음
- 짧고 규격화된 명령어 (1클럭 내외로 실행되는 명령어 지향)

고정 길이 명령어 : RISC는 메모리에 직접 접근하는 명령어를 load, store 두 개로 제한

➡️ **메모리 접근을 단순화하고 최소화를 추구**

➡️ 레지스터 이용 연산 많고, 범용 레지스터 개수도 더 많음