**Chapter 5 CPU 성능 향상 기법**

***5-1*** **빠른 CPU를 위한 설계 기법**

**<클럭>**

* 컴퓨터 부품들은 ‘클럭 신호’에 맞춰 일사불란하게 움직인다
* CPU는 ‘명령어 사이클’이라는 정해진 흐름에 맞춰 명령어들을 실행한다

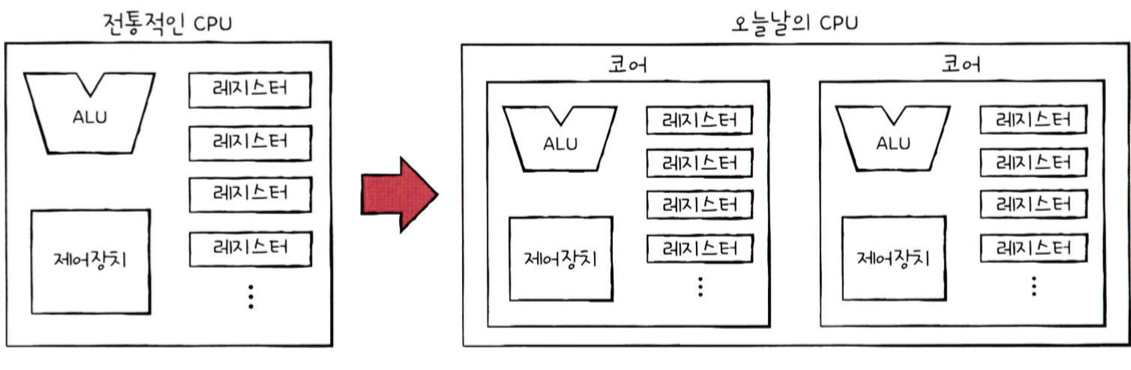
클럭속도가 높아지면 CPU는 명령어 사이클을 더 빠르게 반복 -> CPU 성능이 좋음

하지만 클럭 속도를 무작정 높이면 발열문제 발생 -> 클럭 속도만으로 CPU의 성능을 올리는 것에 한계가 있음

**<코어와 멀티코어>**

클럭 속도를 높이는 방법 외에 CPU의 성능을 높이는 방법 -> CPU의 코어와 스레드 수를 늘림

**코어**: 명령어를 실행하는 부품



코어를 여러 개 포함하고 있는 CPU -> **멀티코어 CPU** or **멀티코어 프로세서**

* 멀티코어는 단일코어보다 속도 빠름
* 코어가 몇 개 포함되어 있는지에 따라 프로세서 명칭이 달라짐 ex) 싱글코어, 듀얼코어, 트리플 코어 등
* CPU의 연산 속도가 꼭 코어 수에 비례하여 증가하지 않음

-> 코어마다 처리할 연산이 적절히 분배되지 않거나 처리하고자 하는 작업량보다 코어 수가 지나치게 많을 때

**<스레드와 멀티스레드>**

**스레드**: 사전적 의미는 실행 흐름의 단위. CPU에서 사용되는 스레드와 프로그래밍에서 사용되는 스레드는 용례가 다름

* CPU에서 사용되는 하드웨어적 스레드
* 프로그램에서 사용되는 소프트웨어적 스레드

**하드웨어적 스레드**

* 하나의 코어가 동시에 처리하는 명령어 단위
* 하나의 코어로 여러 명령어를 동시에 처리하는 CPU -> **멀티스레드 프로세서** or **멀티스레드 CPU**

ex) 2코어 4스레드 -> 명령어를 실행하는 부품을 두 개 포함하고, 한 번에 네 개의

명령어를 처리할 수 있는 CPU를 의미

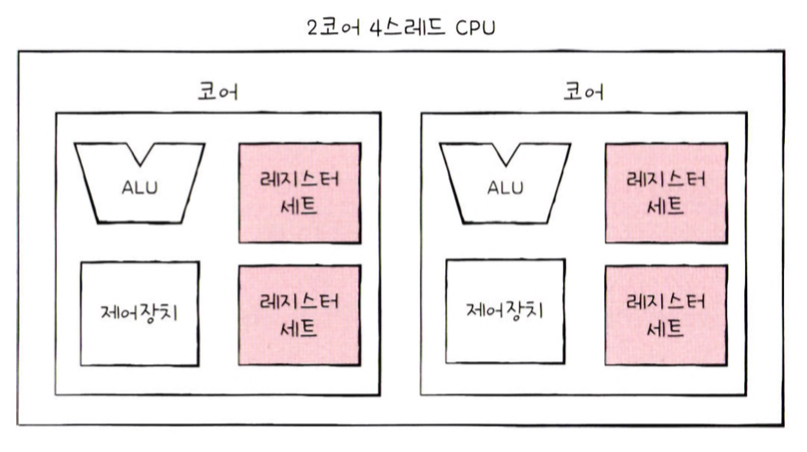
**소프트웨어적 스레드**

* 하나의 프로그램에서 독립적으로 실행되는 단위
* 하나의 프로그램은 실행되는 과정에서 한 부분만 실행될 수도 있지만 여러 부분이 동시에 실행될 수도 있음

ex) 워드 프로세서에서 화면에 출력하는 기능, 맞춤법 검사 기능, 자동 저장 기능 등을 각각의 스레드로 만들면 동시에 실행 가능

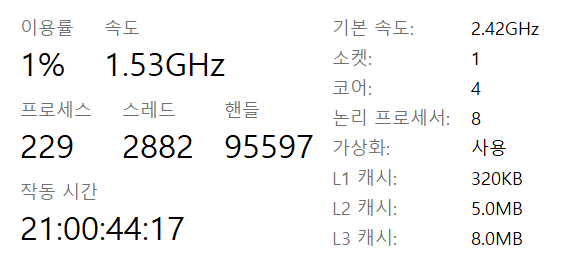
**멀티스레드 프로세서**

* 하나의 코어로 여러 명령어를 동시에 처리하는 CPU
* 실제로 설계하는 일은 매우 복잡하지만, 가장 큰 핵심은 **레지스터**
* 하나의 코어로 여러 명령어를 동시에 처리하도록 만들려면 프로그램 카운터, 스택 포인터, 메모리 버퍼 레지스터, 메모리 주소 레지스터와 같이 하나의 명령어를 처리하기 위해 꼭 필요한 레지스터를 여러 개 가지고 있으면 됨



**논리 프로세서**: 하드웨어 스레드를 논리 프로세서라고 부르기도 함

-> 2코어 4스레드 CPU는 한 번에 네 개의 명령어를 처리가능한데, 프로그램 입장에서는 한 번에 하나의 명령어를 처리하는 CPU가 네 개 있는 것처럼 보임



4코어 8스레드의 논리 프로세서는 8

***5-2*** **명령어 병렬 처리 기법**

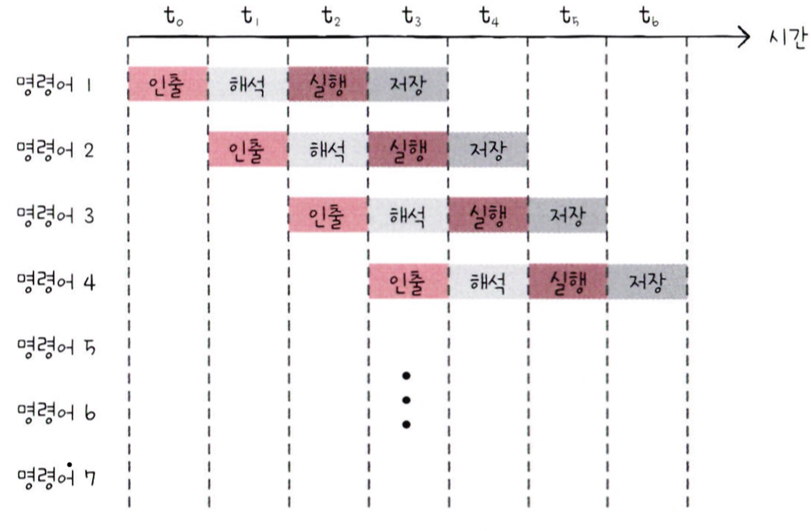
**명령어 병렬 처리 기법**: 명령어를 동시에 처리하여 CPU를 한시도 쉬지 않고 작동시키는 기법

* 대표적으로 명령어 **파이프라이닝**, **슈퍼스칼라**, **비순차적 명령어** 처리가 있음

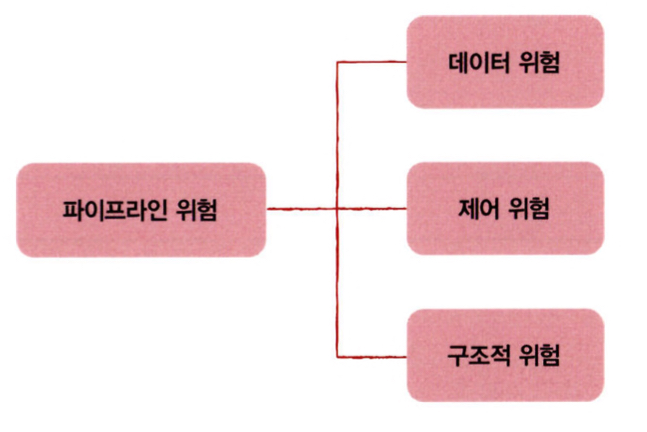
**<명령어 파이프라인>**

**명령어 처리 과정**: 명령어 인출 -> 명령어 해석 -> 명령어 실행 -> 결과 저장

* 같은 단계가 겹치지 않는다면 CPU는 각 단계를 동시에 실행 가능

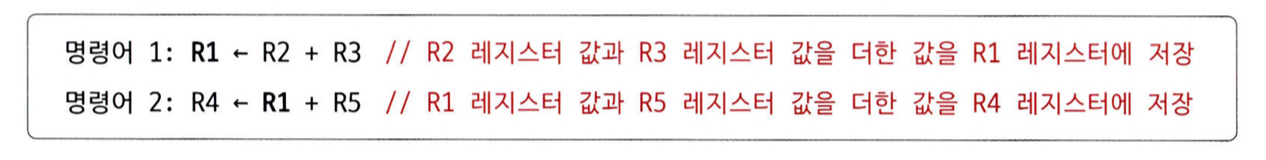


* 공장 생산 라인과 같이 명령어들을 명령어 파이프라인에 넣고 동시에 처리하는 기법을 **명령어 파이프라이닝**이라고 함
* 파이프라이닝은 높은 성능을 가져오기는 하지만, 특정 상황에서는 성능향상에 실패하는 경우도 있고 이러한 상황을 **파이프라인 위험**이라고 부름



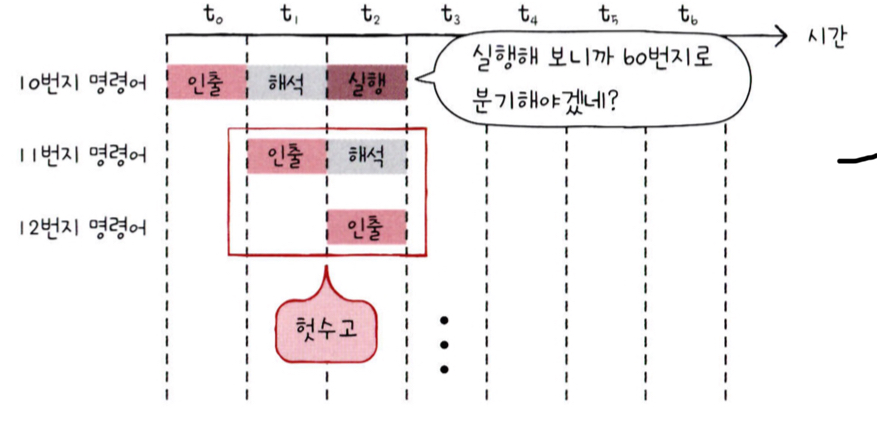
**데이터 위험**

* 명령어 간 데이터 의존성에 의해 발생
* 어떤 명령어는 이전 명령어를 끝까지 실행해야만 비로소 실행할 수 있음
* 데이터 의존적인 두 명령어를 무작정 동시에 실행하려고 하면 파이프라인이 제대로 작동하지 않는 것을 **데이터 위험**이라고 함



**제어 위험**

* 주로 분기 등으로 인한 프로그램 카운터의 갑작스러운 변화에 의해 발생
* 프로그램 카운터는 현재 실행 중인 명령어의 다음 주소로 갱신되지만 프로그램 실행 흐름이 바뀌어 프로그램 카운터 값에 갑작스러운 변화가 생긴다면 명령어 파이프라인에 미리 가지고 와서 처리 중이었던 명령어들은 아무 쓸모가 없어짐



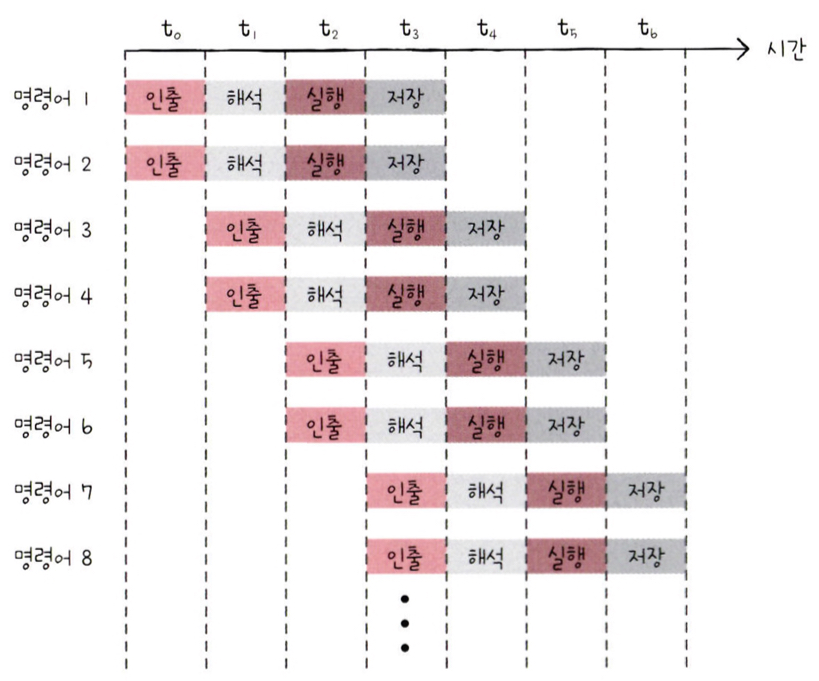
* 이를 위해 사용하는 기술 중 하나가 **분기 예측**
  + **분기 예측**: 프로그램이 어디로 분기할지 미리 예측한 후 그 주소를 인출하는 기술

**구조적 위험**

* 명령어들을 겹쳐 실행하는 과정에서 서로 다른 명령어가 동시에 ALU, 레지스터 등과 같은 CPU 부품을 사용하려고 할 때 발생
* **자원 위험**이라고도 부름

**<슈퍼 스칼라>**

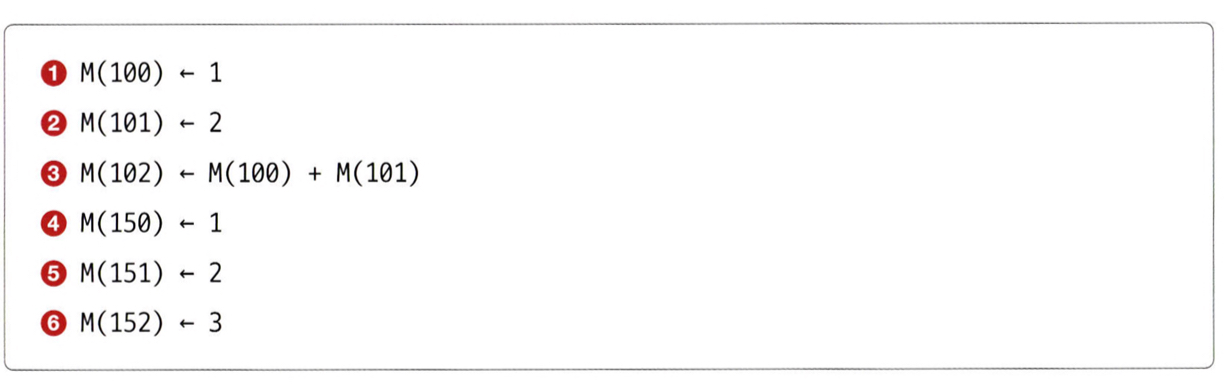
**슈퍼 스칼라**: CPU 내부에 여러 개의 명령어 파이프라인을 포함한 구조



* 슈퍼스칼라 구조로 명령어 처리가 가능한 CPU -> **슈퍼스칼라 프로세서** or **슈퍼스칼라 CPU**
* 이론적으로 프로그램 처리 속도는 파이프라인 개수에 비례하여 빨라지지만 위험 등의 예상치 못한 문제로 반드시 비례하여 빨라지지는 않음

**<비순차적 명령어 처리>**

* 명령어들을 순차적으로 실행하지 않는 기법



-> 3번 명령어를 실행하기 위해서는 1,2번이 끝날 때까지 기다려야 함

-> 4,5,6번 명령어는 데이터 의존성이 없기 때문에 먼저 실행하여 명령어를 실행해도 괜찮음

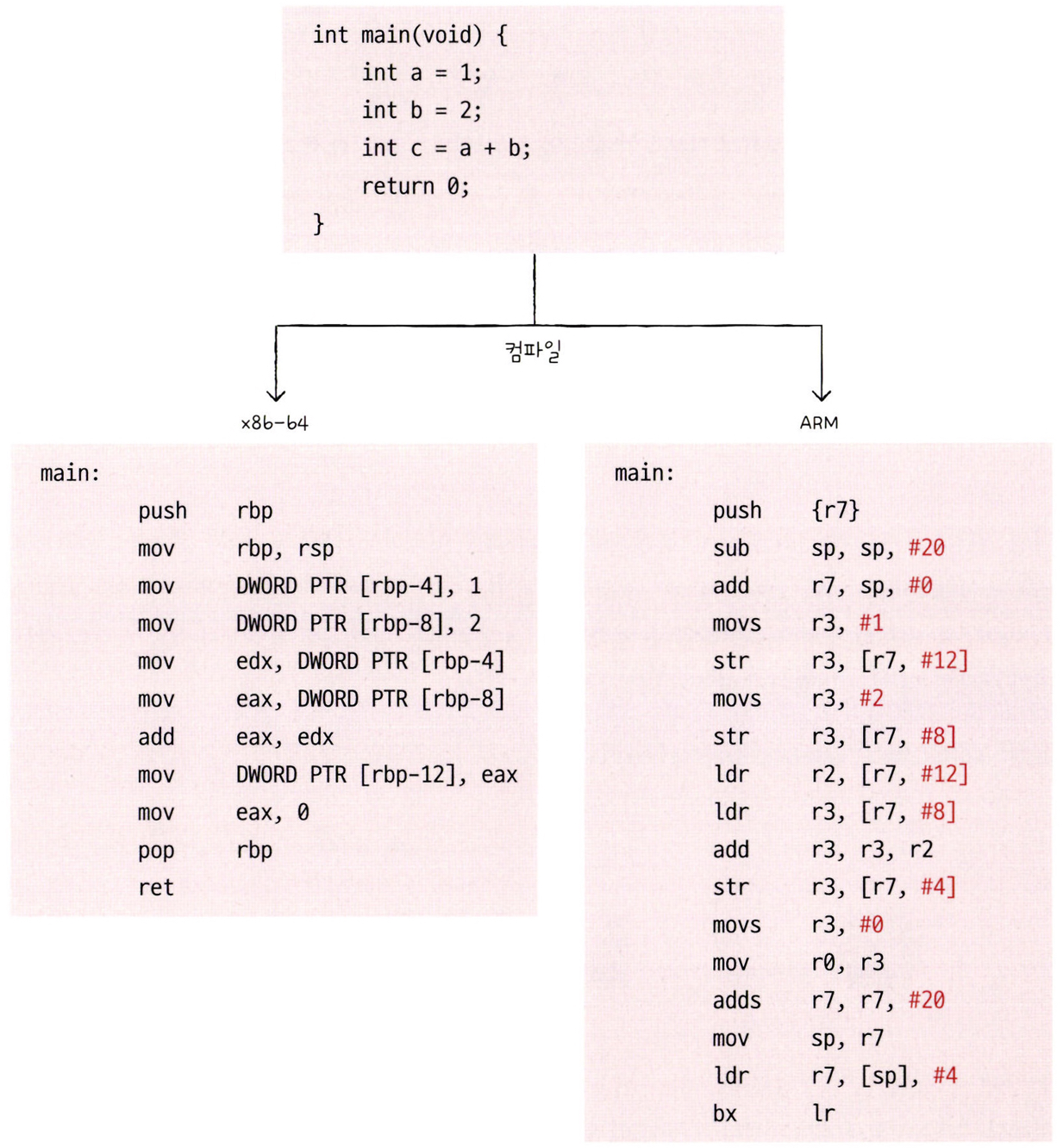
* **비순차적 명령어 처리 기법**: 명령어를 순차적으로만 실행하지 않고 순서를 바꿔 실행해도 무방한 명령어를 먼저 실행하여 명령어 파이프라인이 멈추는 것을 방치하는 기법

***5-3*** **CISC와 RISC**

**<명령어 집합>**

**명령어 집합**: CPU가 이해할 수 있는 명령어들의 모음. **명령어 집합구조**(**ISA**)라고 함

* CPU마다 ISA가 다를 수 있음
* 인텔의 노트북 속 CPU는 x86 혹은 x86-64 ISA를 이해하고, 애플의 아이폰 속 CPU는 ARM ISA를 이해함
* ISA가 다르다는 건 CPU가 이해할 수 있는 명령어가 다르다는 뜻이고 명령어가 달라지면 어셈블리어도 달라짐



* ISA가 다르면 제어장치가 명령어를 해석하는 방식, 사용되는 레지스터의 종류와 개수, 메모리 관리 방법등 많은 것이 달라짐
* 명령어 병렬 처리 기법을 적용하기에 용이한 ISA가 있고, 그렇지 못한 ISA도 있음

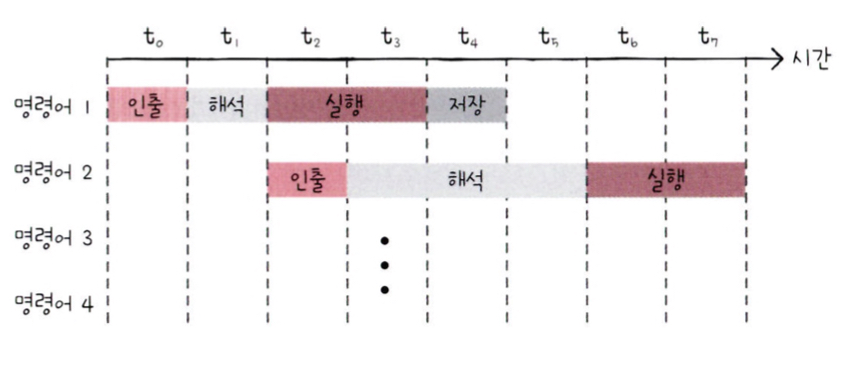
**<CISC>**

* Complex Instruction Set Computer의 약자 -> 복잡한 명령어 집합을 활용하는 컴퓨터
* 다양한 명령어들을 활용하는 CPU 설계 방식
* 명령어의 형태와 크기가 다양한 **가변 길이 명령어**를 활용
* 메모리에 접근하는 주소 지정 방식도 다양
* 다양하고 강력한 명령어를 활용한다는 말은 상대적으로 적은 수의 명령어로도 프로그램을 실행할 수 있다는 것을 의미

ex) x86-64

* 메모리 공간을 절약할 수 있다는 장점
* 단점은 명령어의 크기와 실행되기까지의 시간이 일정하지 않고, 복잡한 명령어 때문에 명령어 하나를 실행하는 데에 여러 클럭 주기를 필요로 함

-> 명령어 파이프라인을 구현하는 데 큰 걸림돌



* 대다수의 복잡한 명령어는 사용 빈도가 낮음
* 위의 단점들로 CISC 기반 CPU는 성장에 한계

**<RISC>**

* Reduced Instruction Set Computer의 약자
* CISC에 비해 명령어의 종류가 적음
* 짧고 규격화된 명령어, 되도록 1클럭 내외로 실행되는 명령어를 지향
* **고정 길이 명령어**를 활용 -> 명령어 파이프라이닝에 최적화
* 메모리에 직접 접근하는 명령어를 load, store 두 개로 제한할 만큼 메모리 접근을 단순화하고 최소화를 추구 -> CISC보다 주소 지정 방식의 종류가 적은 경우가 많음
* 메모리 접근을 단순화, 최소화하는 대신 레지스터를 적극적으로 활용
* 사용 가능한 명령어 개수가 CISC보다 적기 때문에 보다 많은 명령으로 프로그램을 작동 ex) ARM

