* Amdahl’s Laws
* MIPS는 성능 비교에 적합하지 않음 <= 각 instruction 별 clock cycle 과 clock cycle time을 반영하지 않기 때문
* Assembly는 High-level code와 기계어 사이의 인터페이스 역할
* 프로세서와 메모리 사이의 데이터 이동은 비용이 많이 듦 => 메모리에서 데이터를 가져와 처리한 후 결과물만 다시 메모리로 이동하는 방식(폰 노이만 구조)
* Load, Store 등 메모리에 접근하는 instruction은 비용이 큼 => 컴파일러는 레지스터를 최대한 활용해야 함
* Immediate를 쓰는 이유? => 비용이 큰 Load instruction을 줄일 수 있기 때문. 변수 대신 Immediate를 많이 쓸수록 성능 향상을 노릴 수 있다.
* Code와 Data: 메모리 내에서 분리된 영역. Code 영역의 명령어들을 순차적으로 실행(pc value는 0부터 시작). 명령어 실행에 필요한 resource들을 Data 영역에서 가져옴(Load). pc값의 흐름을 제어하는 것이 Control Transfer Operations (ex. Branch).
* Target Addressing
  + 각 instruction들은 16비트 혹은 32비트의 크기를 가짐 => 즉, 명령어의 주소는 항상 짝수 => 주소의 LSB는 항상 0 => 굳이 저장할 필요가 있을까?
  + 따라서 12비트의 공간을 마치 13비트처럼 사용할 수 있음.
  + Target Address를 구할 때는 instruction 내 immediate 값 끝에 다시 0을 붙이고 Sign Ext하여 사용함.
* 같은 High-Level 코드를 짜도 컴파일 방식에 따라 성능이 달라짐.
  + Ex) 반복문을 컴파일할 때, 반복문 안에 들어 있는 instruction의 개수에 따라 성능이 크게 달라짐. => 이 점을 인지하고 코드를 짜면 성능 향상
* Procedure Call (함수 호출)
  + Passing Control: pc값을 procedure code 위치로 옮겼다가 subfunction의 수행이 끝나면 다시 원래 위치(return address)로 돌아와야 함.
  + Passing Arguments: subfunction에 arguments들을 넘겨주고, 그로부터 수행 결과 값(return value)을 반환 받아야 함.
  + Memory Management: procedure execution 동안 메모리 공간 할당 => 수행 끝난 후 Deallocate해야 함.
  + 메모리 영역은 스택 자료구조에 의해 관리됨.
    - Procedure Call 발생 시, Caller가 가지고 있던 data들과 return address의 pc값을 스택에 저장. => 나중에 호출된 함수부터 차례로 수행되기 때문(LIFO) => 이때 함수 단위로 스택에 저장된 pc값과 data들을 하나의 frame으로 묶음 => 각 frame의 첫 부분을 가리키는 fp 레지스터.
    - RISC-V의 스택은 위에서 아래로 쌓임 => 아래로 내려갈수록 address는 작아짐 => 가장 아래 address (동시에 스택의 top)을 가리키는 레지스터가 stack pointer(sp).
    - RISC-V는 argument passing을 위해 8개의 레지스터 할당 => argument가 8개를 넘어갈 경우 나머지는 스택에 저장해 두었다가 꺼내 씀 => 메모리와 데이터 이동이 일어나므로 성능 저하
  + Register Saving Problem: 같은 레지스터를 사용하는 Procedure Call 발생 시, Caller가 백업할 것인가 Calle가 백업할 것인가? => Caller와 Calle는 서로가 어떤 데이터를 사용하는지 알지 못함. => 자신이 가진 모든 데이터를 백업. => 데이터를 더 적게 생성하는 쪽이 백업하는 게 이득.
    - RISC-V의 경우: Temporary는 Caller가, Variable은 Calle가 백업 => 일반적으로, Caller는 Variable을, Calle는 Temporary를 더 많이 생성한다고 보았기 때문.
* Array vs. Pointer: Array에 대한 반복문 코드를 짤 때, 인덱스를 사용하는 것보다 포인터를 사용하는 것이 성능적으로 이득 <= 인덱스를 사용할 경우, 매 루프마다 인덱스를 계산하는 연산이 추가적으로 발생하기 때문. => 의도적으로 포인터를 사용함으로써 성능을 개선할 수 있음.
* Digital System = Combinational Elements + Sequential Elements + Clock signals
* 레지스터는 Sequential Elements로, Clock signal에 의해 업데이트됨. => 레지스터가 업데이트되는 데 걸리는 Clock cycle은 1.
* Single Cycle Processor
  + 모든 종류의 instruction에 대해 하나의 레지스터를 갖는 회로 사용 => 모든 instruction이 하나의 clock cycle 사용 => CPI = 1
  + 여러 Control signal들에 의해 하나의 프로세서로 서로 다른 명령어 수행 가능
  + 장점: Multicycle에 비해 단순함.
  + 하나의 clock cycle에 모든 instruction이 수행 가능하므로 clock cycle time은 propagation delay(critical path, Load instruction)에 의해 결정됨. => Clock cycle time 증가, 비효율적.
* Multi Cycle Processor
  + 하나의 Clock cycle을 갖던 Single Cycle Processor를 여러 단계로 분리함. => 여러 번의 Clock cycle 발생(Clock cycle time은 감소) <= 여러 개의 레지스터를 사용해야 함.
  + instruction마다 CPI가 다름 <= 여러 개의 단계로 나누고, 각 instruction들은 필요한 만큼의 단계만 거치면 되기 때문. => CPI 자체만으로는 Single Cycle 보다 높지만, 전체적인 성능으로는 Multi Cycle이 더 뛰어남.
  + Memory와 ALU는 1개씩만 사용. <= 각 단계마다 따로 사용할 수 있기 때문 => Reusable => 성능 향상
  + Branch type Instruction이 3 cycle만에 수행될 수 있는 이유
    - Instruction Decoding과 Register Fetch가 진행되는 중에 동시에 Target Address를 미리 계산해두기 때문 => 추가적인 레지스터 A, B, ALUOut을 사용하기에 가능한 방식.
    - 단, 이 단계까지는 아직 Decoding이 끝나지 않았기 때문에 Branch Instruction인지는 모르는 상황. (어차피 Register Fetch가 오래 걸리니 일단 계산해두고 보기)
  + Type 별 Clock cycle number
    - Load: 5
    - Store: 4
    - R-type: 4
    - Branch: 3
  + 단점: 추가적인 레지스터를 필요로 함, Single Cycle에 비해 복잡함.
* Design Principle
  + Simplicity favors regularity.
    - Regularity makes implementation simpler.
    - 더 적은 비용으로 더 높은 성능
      * Ex) 모든 instruction의 Opcode는 같은 위치에 존재 => decoding에 드는 비용 절감 (다른 위치였다면 decoding을 위해 모든 비트를 살펴봐야 됨)
  + Smaller is faster