CPU 설계 기법

코어와 멀티코어

- 코어: CPU에서 명령어를 실행하는 부품
- 멀티코어: 코어를 여러개 포함하고 있는 CPU. 멀티코어 프로세서 라고도 부름
- CPU의 연산속도가 꼭 코어 수에 비례하여 증가하지는 않음
- Amdahl's law: 병렬화된 작업에서 얻을 수 있는 성능 향상의 한계를 설명하는 수학적 원리. 프로세서를 병렬화 시켜도 병렬처리가 불가능한 부분이 있어서 성능 향상의 한계가 있음
  - $S = \left((1-p) + p/N\right)^{-1} < (1-p)^{-1}$
  - S는 전체 작업 속도 향상 비율, N은 코어의 수, p는 병렬화 가능한 부분의 비율
  - p/N은 병렬처리했을 때 성능이 향상되는 부분을 의미하고, 1-p는 병렬처리가 불가능한 부분을 말함. 수식에 따르면 프로세서를 늘려도 병렬처리가 불가능한 부분을 처리하는 시간보다 적게 성능을 향상시킬 수 없다.
  - 예) 동일한 가중치를 갖는 4단계 중 1단계만 병렬화가 가능한 경우 p = 1/4이므로 $S < (1 - 0.25)^{-1} \simeq 1.3$

CPU의 언어와 CPU의 주요 설계 방식

ISA (Instruction Set Architecture) - 명령어 집합 구조

- CPU의 언어이자 하드웨어가 소프트웨어를 어떻게 이해할지에 대한 약속
- CPU가 이해할 수 있는 명령어들의 모음
- CPU마다 서로 다른 ISA를 가질 수 있다: 소스 코드가 같아도 실행 언어 (어셈플리어)가 달라짐
- 현대 ISA에는 CISC과 RISC가 있음

CISC vs RISC

CISC (Complex Instruction Set Computer)	RISC (Reduced Instruction Set Computer)
복잡하고 다양한 명령어	단순하고 적은 명령어
가변 길이 명령어	고정 길이 명령어
메모리 공간을 절약할 수 있음	메모리 접근을 단순화, 최소화하고 레지스터 활용
다양한 주소 지정 방식	적은 주소 지정 방식
프로그램을 이루는 명령어 수 적음	프로그램을 이루는 명령어 수 많음
여러 클럭에 걸쳐 명령어 수행	1클럭 내외로 명령어 수행
파이프라이닝 하기 어려움	파이프라이닝하기 쉬움

Cashe Memory

Cashe Memory

- RAM (주 메모리)와 CPU 사이에 위치
- CPU의 연산 속도와 메모리 접근 속도의 차이를 줄이기 위한 SRAM 기반의 저장 장치
- SRAM: 시간이 지나도 저장된 데이터가 사라지지 않는 RAM (반대로, DRAM은 시간이 지나면 저장된 데이터가 사라짐)
- 코어와 가까운 순서대로 L1, L2, L3 캐시라고 부름. 일반적으로 L1, L2는 코어 내부에 위치

Cashe miss

- 캐시 메모리가 CPU가 사용할 법한 대상을 예측하여 저장했는데, 예측이 틀려서 메모리에서 필요한 데이터를 직접 가져와야 하는 경우.
- 프로세서가 참조하려는 데이터가 캐시 메모리에 없을 때 발생함
- Compulsory (cold) miss (강제적 미스): 프로그램이 처음 실행될 때 발생. 데이터를 캐시에 저장할 수 있는 새로운 캐시 블록이 필요한 상황에서 발생함
  감소 방법: 프로그램이 시작될 때 캐시를 초기화하고 데이터를 미리 채워넣는 방법
- Capacity miss (용량 미스): 캐시가 필요한 데이터를 저장할 용량이 부족하여 발생.
  감소 방법: 캐시의 크기를 늘려서 더 많은 데이터를 저장, 중요도에 따라 저장
- Conflict miss (충돌 미스): 여러 데이터가 동일한 캐시 세트에 접근하려고 할 때 발생
  감소 방법: 캐시 연관성을 높여서 충돌을 줄이는 방법, 해싱 사용

Cashe hit ratio (캐시 적중률)

- 캐시가 히트되는 비율을 말함 = 캐시 히트 횟수 / (캐시 히트 횟수 + 캐시 미스 횟수)
- 캐시 히트: 캐시 메모리가 CPU가 사용할 법한 대상을 예측하여 저장하였는데 실제로 활용될 경우
- 참조 지역성의 원리: 캐시 메모리가 예측을 위해 따르는 원칙
  - CPU는 최근에 접근했던 메모리 공간에 다시 접근하려는 경향이 있다.
  - CPU는 접근한 메모리 공간 근처를 접근하려는 경향이 있다. (공간 지역성)

Cashe Associativity (캐시 연관성)

- 주기억장치(RAM)의 데이터 블록을 캐시 메모리에 매핑하는 방식
- 캐시 연관성이 높아질 수록 캐시 적중률이 높아짐
- 캐시의 성능과 하드웨어 복잡성 (충돌) 간에 트레이드 오프가 있음
- Comprarator (비교기): 캐시 메모리의 각 연관성 레벨에서 주소를 검색하고 찾는데 사용되는 하드웨어 구성요소. 캐시 내의 주소와 메모리에서 읽어온 주소를 비교하여 해당 데이터가 캐시에 존재하는지 확인함

Direct-Mapped Cache (직접 연관성)	Set-Associative Cache (세트 연관성)	Fully-Associative Cache (완전 연관성)
주소가 하나의 위치에만 할당	주소를 어떤 세트에 할당할지 비교하여 선택	주소를 어떤 위치에 할당할지 모든 블록을 비교하여 선택
비교기 1개	비교기 n개 (n-way)	비교기 N개 (N개 블록)
검색 속도가 빠름	중간 검색 속도	검색 속도가 느림
충돌 문제가 많음	충돌 완화	충돌 문제가 적음

- Cashe Coherence (캐시 일관성)

- 다중 캐시 시스템에서 공유 데이터의 일관성을 유지하기 위한 원칙과 기술
- Read Coherence (읽기 일관성): 동일한 메모리 위치에서 데이터를 읽을 때 동일한 값을 얻어야함
- Write Coherence (쓰기 일관성): 데이터를 쓰면 변경 사항이 모든 캐시와 메모리에 반영되어야함
- Global System Coherence (전역 시스템 일관성): 모든 프로세서와 캐시 간 데이터 일관성이 유지

MESI protocol

- 다중 캐시 시스템에서 데이터 일관성을 관리하기 위한 프로토콜 중 하나
- 별도의 플래그 (flag)를 할당하여 플래그의 상태로 데이터 유효성 여부를 판단
- 메모리가 가지는 4가지 상태 정의: Modified, Exclusive, Shared, Invalid
  1. Modified (수정): 데이터가 수정된 상태
  2. Exclusive (배타): 유일한 복사본이며, 주기억장치의 내용과 동일한 상태
  3. Shared (공유): 데이터가 두 개 이상의 프로세서 캐시에 적재되어 있는 상태
  4. Invalid (무효): 데이터가 다른 프로세스에 의해 수정되어 무효화된 상태
- 캐시의 내용과 주기억장치의 내용이 같으면 유효, 다르면 무효 상태
- 상태 전이

  (a) I -> E

  프로세서1: 메모리에서 데이터 읽어와서 로컬 캐시에 저장함: E-state

  프로세서2: 동일한 데이터를 수정하여 메모리에 쓰기 작업 수행: I-state

  프로세서1: 메모리에서 데이터 읽어와서 로컬 캐시에 저장함: E-state

  (b) E -> S

  프로세서1: 데이터를 읽어와서 로컬 캐시에 저장: E-state로 변경

  프로세서2: 동일한 데이터를 메모리에서 읽어서 로컬 캐시로 복사: S-state로 전이

  (c) S -> M

  프로세서1: 데이터를 읽어와서 로컬 캐시로 복사: S-state

  프로세서1: 데이터를 수정하여 메모리에 쓰기 작업 수행: M-state로 전이

  (d) M -> S

  프로세서1: 데이터를 수정하여 메모리에 쓰기 작업 수행: M-state

  프로세서2: 프로세서1이 수정한 데이터를 읽어와서 로컬 캐시로 복사: S-state로 전이

입출력 방법

Programmed I/O (프로그램 입출력)

- 프로그램 속 명령어로 입출력장치를 제어하는 방법.
- 입출력 장치에 연결된 장치 컨트롤러와 CPU가 상호작용하며 입출력 작업을 수행

Memory-mapped I/O (메모리 맵 입출력)	Isolated I/O (고립형 입출력)
메모리와 입출력장치는 같은 주소 공간 사용	메모리와 입출력장치는 분리된 주소 공간 사용
메모리 주소 공간이 축소됨	메모리 주소 공간이 축소되지 않음
같은 명령어 사용 가능	입출력 전용 명령어 사용

Interrupt-based I/O vs Polling-based I/O

Interrupt-based I/O (인터럽트 기반 입출력)	Polling-based I/O (폴링 기반 입출력)
이벤트 기반: 입출력 장치에서 데이터를 전송할 준비가 됐을 때, 장치 컨트롤러가 CPU에게 인터럽트를 발생시켜 작업을 알림	주기적인 확인: CPU가 주기적으로 입출력 장치에 대한 상태를 확인
효율적인 자원 활용: CPU가 인터럽트 발생 전까지 다른 작업 수행 가능	CPU 자원 낭비
대기 시간 감소 (프로세스가 입출력 완료를 기다리지 않고 다른 작업 처리 가능)	대기 시간 증가 (프로세스가 입출력 완료를 기다려야 함)
프로그래밍 복잡성	단순한 구현

DMA I/O (Direct Memory Access I/O)

- 프로그램 기반 입출력과 인트럽트 기반 입출력의 단점: 입출력장치와 메모리 간의 데이터 이동을 CPU가 주도하고, 이동하는 데이터도 CPU를 거친다
- DMA I/O: DMA 컨트롤러를 통해 직접 메모리에 접근할 수 있는 입출력 기능
  1. CPU가 DMA 컨트롤러에 입출력 작업 명령
  2. DMA 컨트롤러는 CPU 대신 장치 컨트롤러와 상호작용. 메모리에 직접 접근하여 정보 읽거나 쓰기
  3. 입출력 작업이 끝나면 DMA 컨트롤러가 CPU에 인터럽트를 걸어 작업이 끝났음을 알림
- 장점:
  1. 프로세서 개입 없이 데이터를 전송하여 읽기-쓰기 작업속도가 빠름. 속도와 성능향상
  2. 프로세서의 오버헤드가 줄어듦

ILP: Instruction-Level Parallelism (명령어 병렬 처리 기법)

Instruction pipelining (명령어 파이프라이닝)

- 동시에 여러개의 명령어를 겹쳐 실행하는 기법
- 명령어 처리과정: 같은 단계가 겹치지 않으면 CPU는 각 단계를 동시에 실행할 수 있음
  1. IF: Instruction Fetch (명령어 인출)
  2. ID: Instruction Decode (명령어 해석)
  3. EX: Execute Instruction (명령어 실행)
  4. MEM: Memory Access (메모리 액세스)
  5. WB: Write Back (결과 저장)
- 장점: 하나의 명령어를 모두 처리되기 전에 다음 명령어가 처리될 수 있어서 효율적임

Pipeline hazards (파이프라인 위험)

Data hazard (데이터 위험)	**Control hazard** (제어 위험)	**Structural hazard** (구조적 위험)
명령어 간 데이터 의존성에 의해 발생	프로그램 카운터의 갑작스러운 변화에 의해 발생	명령어들이 동시에 CPU 부품을 사용하려고 할 때 발생
어떤 명령어가 이전 명령어를 끝까지 실행 후 실행되어야함	프로그램 카운터는 현재 실행중인 명령어의 다음 주소로 갱신되는데, 프로그램 실행 흐름이 바뀌면 미리 실행되던 명령어가 쓸모없어짐	자원 위험 (resource hazard) 이라고도 함
비순차적명령어처리(OoOE)	분기 예측 (분기 명령어 결과를 예측)	하드웨어 리소스 추가, 슈퍼파이프라이닝, 명령어 스케줄링

- 파이프라인 위험 예제 코드와 실행 다이어그램:

```
1) ADD R1, R2, R3       IF ID EX MEM WB
2) SUB R4, R1, R5          IF ID EX  MEM WB
3) BEG R6, R7, Label.         IF ID  EX  MEM WB
4) ADD R8, R9, R10               IF  ID  EX  MEM WB
5) (Label). ADD R11, R1, R12.       IF  ID  EX  MEM WB
```

  - Data hazard: 1)에서 R1이 WB되기 전에 2)에서 사용됨
  - Contral hazard: 3)에서 EX단계가 이후 4)보다 5)가 먼저 실행될 수도 있다

Superscalar

- CPU 내부에 여러 개의 명령어 파이프라인을 포함.
- 장점: 명령어 수행 속도와 처리량을 향상시킬 수 있음
- 이론적으로는 파이프라인 개수에 비례하여 프로그램 처리 속도가 빨라짐
- 일반적으로 이상적인 성능을 나타내지 않는 이유: 파이프라인 위험도 증가

Superpipelining

- 파이프라인의 명령어 인출, 명령어 해석, 명령어 실행, 결과 저장을 더 세분화하여 더 많은 단계로 분할한 기법.
- CPU에서 각 명령어를 처리하는데 걸리는 시간을 줄여 전체적인 명령어 처리량과 성능 향상
- 장점:
  1. 높은 성능: 작업을 여러 단계로 나누기 때문에 시스템 성능을 향상시킴. 동시에 여러작업 처리
  2. 빠른 클럭 속도: 파이프라인의 단계를 세분화해서 각 단계를 빠르게 수행
- 단점:
  1. 복잡성: 하드웨어 복잡성이 증가하여 설계와 구현이 어려움
  2. 파이프라인 지연: 전체 작업에는 추가적인 지연이 발생할 수 있음

IoOE: In-of-order Execution

- 명령어를 위에서 아래로 순차적으로 처리하는 명령어 병렬처리 기법
- 단점
  1. 자원 낭비: 명령어가 정해진 순서대로 실행되므로 사용 가능한 자원이 효율적으로 활용되지 않을 수 있음
  2. 성능 저하: 명령어 간의 데이터 종속성이 있을 때 명령어가 대기해야하므로 처리량과 성능이 떨어질 수 있음
  3. 제어 흐름 제한: 명령어의 제어 흐름을 변경하기가 어려움

OoOP: Out-of-order Processor

- 명령어 파이프라인과 명령어 스케줄링 기술을 사용하여 명령어의 완료 순서를 최적화하는 목적
- 명령어를 프로그램에서 나타나는 순서와 다르게 실행함
- 동작방식
  - 명령어 스케줄링: 명령어를 실행가능한 스테이지로 스케줄링. 명령어 간의 의존성과 충돌 고려
  - OoOE: 명령어 스케줄러의 결정에 따라 실행
  - Reorder: 명령어가 실행을 완료하면 결과를 명령어 버퍼에 보관하고 명령어 스케줄러의 결정에 따라 결과가 재정렬됨. 명령어 실행 순서가 프로그램에서의 순서와 바뀌기도 함
- Precise exception: 예외처리가 완료된 후 예외 발생 지점으로 돌아가 다시 프로그램을 동작하는 것이 쉽게 가능한 경우를 말함
- Tomasulo Algorithm (out-of-order processor without reorder buffer): reorder buffer는 명령어 실행이 완료된 후에 결과를 저장하는 공간을 말하는데, 예외가 발생했을 때 buffer가 없으면 예외 발생 명령어를 찾을 수 없기 때문에 precise exception을 지원할 수 없음

OoOE: Out-of-order Execution (비순차적 명령어 처리)

- Out-of-order processor의 동작 방식을 나타냄
- 명령어를 순차적으로만 실행하지 않고 순서를 바꿔 실행해도 무방한 명령어를 먼저 실행하여 명령어 파이프라인이 멈추는 것을 방지하는 기법

Data Hazard (데이터 위험)

- 파이프라인 위험 중 하나로 명령어 간 데이터 의존성에 의해 발생된다.
- OoOP에서 고려해야 할 데이터 위험의 세 가지 유형
  1. Read After Write (쓰기 후 읽기): 이전 명령어가 저장한 연산 결과를 다음 명령어가 읽을 때
     ```
     // R1
     ADD R1, R2, R3
     ADD R4, R1, R5
     ```
  2. Write After Read (읽기 후 쓰기): 이전 명령어가 읽고 있는 데이터를 다음 명령어가 변경할 때
     ```
     // R1
     ADD R2, R1, R3
     ADD R1, R4, R5
     ```
  3. Write After Write (쓰기 후 쓰기): 이전 명령어가 쓴 값이 다음 명령어가 쓴 값으로 덮어 씌어짐
     ```
     // R1
     ADD R1, R2, R3
     ADD R1, R4, R5
     ```

Secondary storage (보조기억장치) 활용 방법

RAID (Redandant Array of Independent Disks)

- 데이터의 안정성 혹은 성능을 높이기 위해 여러개의 물리적 보조기억장치를 하나의 논리적 보조기억장치처럼 사용하는 기술
- RAID 0: 여러 보조기억장치에 데이터를 단순히 나누어 저장. 각 하드디스크에 번갈아가며 저장
  장점: 분산저장 (striping)으로 저장된 데이터를 읽고 쓰는 속도가 빠름
  단점: 저장된 정보가 안전하지 않음. 디스크 하나에 문제가 생기면 모든 정보를 읽는데 문제 발생
- RAID 1: 완전한 복사본을 만듦 (Mirroring).
  장점: 똑같은 디스크가 두 개 있는 것과 동일하여 하나에 문제가 생겨도 복구가 간단함
  단점: 사용 가능한 용량이 적어지고 비용이 증가. 쓰기 속도가 RAID 0보다 느림
- RAID 4: 오류를 검출하고 복구하기 위한 정보 (parity bit)를 저장
  장점: RAID 1보다 적은 하드 디스크로도 데이터를 안전하게 보관 가능
  단점: Parity를 저장하는 장치에 병목 현상이 발생
- RAID 5: 각 하드디스크에 pairty를 분산하여 저장
  장점: RAID 4의 병목 현상을 해소
  단점: 쓰기 속도는 RAID 0보다 느림
- RAID 6: RAID5와 기본 구성은 같으나 서로 다른 두개의 parity를 저장함
  장점: RAID 4, RAID 5보다 안전함
  단점: 쓰기 속도는 RAID 5보다 느림
- Nested RAID: 여러 RAID 레벨을 혼합한 방식

Virtual Memory 와 Paging

가상 메모리 관리

- 가상 메모리: 실행하고자 하는 프로그램을 일부만 메모리에 적재하여 실제 물리 메모리 크기보다 더 큰 프로세스를 실행할 수 있게 하는 기술
- 프레임: 메모리 물리 주소 공간을 페이지와 동일한 크기로 자른것
- 가상 주소 크기: 가상 주소 공간에서 사용 가능한 주소의 비트 수
- 가상 메모리 크기: 가상 주소 공간이 가질 수 있는 모든 주소의 조합 = $2^{\text{size of virtual address (bit)}}$ bytes
- 물리 주소 크기: 물리 주소 공간에서 사용 가능한 주소의 비트 수
- 물리 메모리 크기: 물리 주소 공간이 가질 수 있는 모든 주소의 조합 = $2^{\text{size of physical address (bit)}}$ bytes
  예) 가상 주소 크기 = 40 bits, 물리 주소 크기 = 32 bits 일 때, 가상 메모리 크기 = 2^{40} byes 이고 물리 메모리 크기 = 2^{32} bytes 이다.

Paging

- 페이징: 메모리와 프로세스를 일정한 단위로 자르고, 이를 메모리에 불연속적으로 할당하는 것
- 페이지: 프로세스의 논리 주소 공간을 일정한 단위로 자른것
- 페이지 테이블: 페이지 번호와 프레임 번호를 짝지어준 표
- 페이지 엔트리의 수 = 가상 메모리 크기 / 페이지 크기
- 페이지 테이블의 크기 = 페이지 엔트리의 수 * 페이지 엔트리의 크기
  예) 가상 메모리 크기 = 2^{40} bytes, 페이지 크기 = 4 KB, 페이지 엔트리의 크기 = 4 byte 일 때, 페이지 크기 = 2^12 byte로 쓰면 페이지 엔트리의 수 = 2^{40} / 2^{12} = 2^{28} 개. 페이지 테이블의 크기 = 2^{28} * 2^{2} byte = 2^{30} bytes = 1GB

Page Replacement Algorithm (페이지 교체 알고리즘)

- 기존에 메모리에 적재된 불필요한 페이지를 선별하여 보조기억장치로 내보내는 작업
- Demand paging (요구 페이징): 실행에 요구되는 페이지만 적재하는 기법
- Page fault: 프레임에 참조하고자하는 페이지가 존재하지 않는 상태. 보조기억장치로부터 필요한 페이지를 가져와야함
- Page reference string (페이지 참조열): CPU가 참조하는 페이지들 중 연속된 페이지를 생략한 페이지열
- FIFO 페이지 교체 알고리즘: 메모리에 가장 먼저 올라온 페이지부터 내쫓는 방식
  단점: 프로그램 실행 초기에 사용된 페이지가 실행 내내 사용될 내용을 포함할 수도 있음
- Second chance (2차 기회) 페이지 교체 알고리즘: FIFO 알고리즘을 쓰되 참조비트를 확인하여 1일 경우 내쫓지 않고 참조비트를 0으로 만든 뒤 현재 시간을 적재 시간으로 설정. 참조비트는 CPU가 참조할 때 1로 바뀜.
- Optimal (최적) 페이지 교체 알고리즘: 앞으로의 사용 빈도가 가장 낮은 페이지를 교체하는 알고리즘. 가장 낮은 페이지 폴트율을 보장함. 실제 구현이 어려워서 페이지 교체 알고리즘의 이론상 성능을 평가하기 위한 목적으로 사용됨
- LRU (Least Recently Used) 페이지 교체 알고리즘: 가장 오랫동안 사용되지 않은 페이지를 교체하는 알고리즘.

Page Table

- PTBR (Page Table Base Register): 각 프로세스의 페이지 테이블이 적재된 주소를 가리키는 레지스터. TLB (Translation Lookaside Buffer): 페이지 테이블의 캐시 메모리. 페이지 테이블을 메모리에 두면 메모리 접근시간이 두배로 늘어나는데 TLB로 이를 해결할 수 있음
- 참조 지역성에 근거하여 주로 최근에 사용된 페이지 위주로 가져와 저장함.
- 예측이 성공하면 TLB hit, 실패하면 TLB miss

Q15.

Categorize processor types using Flynn’s taxonomy. List example processors that are included in each category. (Flynn의 분류 체계(Flynn's taxonomy)를 사용하여 프로세서 유형을 분류하세요. 각 범주에 포함된 예제 프로세서를 나열하세요.)

Flynn's taxonomy (Fynn의 분류체계)

- SISD (Single Instruction, Single Data Stream): 단일 명령어로 단일 데이터처리
  - 하나의 명령 스트림이 하나의 데이터 스트림을 처리하는, 즉 명령과 데이터에 병렬성이 없는 순차적인 시스템을 말한다. 싱글코어 프로세서 하나가 탑재된 PC가 여기에 해당한다.
  - 인텔의 x86, AMD의 Ryzen
- SIMD (Single Instruction, Multiple Data Stream): 단일 명령어로 다중 데이터 처리
  - 단일 명령을 복수의 연산기에 전달하더라도 각각의 연산기는 서로 다른 데이터를 가지고 연산하는 시스템을 말한다. 벡터 연산기라고 불리는 일부의 슈퍼 컴퓨터가 대표적인 예다.
  - 인텔의 SSE, ARM의 NEON
- MISD (Multiple Instruction, Single Data Stream): 다중 명령어로 단일 데이터 처리
  - 복수의 명령 스트림이 단일 데이터 스트림을 처리하는 시스템을 말한다. 이론상으로만 존재할 뿐 쓸모가 없으므로 이 분류에 해당하는 실제 시스템은 거의 존재하지 않는다.
  - Not Common.
- MIMD (Multiple Instruction, Multiple Data Stream): 다중 명령어로 다중 데이터 처리
  - 복수의 연산기가 가진 각기 다른 데이터 스트림을 서로 다른 명령 스트림에서 처리하는 시스템.
  - 인텔의 Xeon, AMD의 EPYC

Q16.

What are two types of data locality observed in typical workloads? List example data accesses that exhibit each type of data locality. (전형적인 작업에서 관찰되는 데이터 지역성의 두 가지 유형은 무엇인가요? 각 데이터 지역성 유형을 나타내는 예제 데이터 접근을 나열하세요.)

- Temporal locality: 접근한 메모리 공간 근처를 접근하려는 경향
  - instruction in loop, induction variables
- Spatial locality : 최근에 접근했던 메모리 공간에 다시 접근하려는 경향
  - sequential instruction access, array data

→ 캐시 메모리는 2개의 참조 지역성의 원리에 입각해 CPU가 사용할 법한 데이터를 예측함.

Q17.

What does “CPI” stand for? Explain why we can say the performance of a computer system can be improved if CPI is decreased. (You should exhibit the equation that calculates execution time.) Show counterexamples (i.e. cases that performance is degraded even if CPI is reduced). (CPI는 무엇을 의미합니까? CPI가 감소하면 컴퓨터 시스템의 성능이 향상될 수 있다고 말할 수 있는 이유를 설명합니다. (실행 시간을 계산하는 식을 표시해야 합니다.) 반례(즉, CPI가 감소해도 성능이 저하되는 경우)를 보여줍니다.)

- CPI: Clock cycle numbers per instruction
- Execution time (실행 시간) = (# of instructions)×(CPI)×(clock cycle time)
- 일반적으로 CPI가 낮아지면 execution time 도 낮아진다.
- 반례) CPI가 낮아도 performance가 낮은 경우
  - 파이프라이닝 오버헤드: 명령어가 파이프라인을 통과하는 데 더 많은 클럭 사이클이 필요할 수 있음
  - 캐시 미스: 데이터를 메인 메모리에서 가져와야 해서 추가적인 클럭 사이클이 필요해짐
  - 분기 예측 실패: 실행된 명령어들이 폐기되고 재실행이 요구되어 실행 시간이 증가

Q19.

What are the advantages of “dynamic branch prediction” compared to “static branch prediction”? Describe limitations of simple 1-bit and 2-bit dynamic branch prediction mechanisms (along with examples). ("동적 분기 예측"과 "정적 분기 예측"을 비교했을 때 "동적 분기 예측"의 장점은 무엇인가요? 간단한 1-bit 및 2-bit 동적 분기 예측 메커니즘의 제한 사항(예와 함께)을 설명하세요)

- Static branch prediction (정적 분기 예측): 컴파일러 도움으로 분기 예측 Predict-Taken or Predict-Not-Taken 을 그대로 수행하되, 만약 mispredict가 일어나면 해당 instruction을 flush 시키고 다시 retry하는 구조로 되어 있음.
- Dynamic branch prediction (동적 분기 예측): runtime 내 mispredict가 발생하면 유동적으로 수행 절차를 바꿀 수 있는 방식.
  1. 적응성: 실행시간에 분기 행동을 모니터링하고 학습하여 실행 상황에 따라 예측을 조정할 수 있음
  2. 다양한 패턴 처리: 다양한 분기 패턴을 인식하고 대응하는 예측 매커니즘을 사용할 수 있음
  3. 성능 향상: 프로그램의 실행 흐름을 더 정확하게 예측하여 불필요한 명령어 실행을 줄임
- 1-bit 동적 분기 예측: 1-bit 레지스터로 추적. 맞으면 레지스터 값 유지, 틀리면 변경
  - 제한사항: 두 가지 예측 결과만 다루므로 정확도가 제한됨. 패턴이 바뀌면 정확한 예측이 어려움. 예) 분기 명령어가 예측이 맞을 경우 계속 실행되거나 틀릴경우 분기로 이동하는데, 무작위로 참과 거짓이 번갈아 나오는 경우에는 정확한 예측이 어려움
- 2-bit 동적 분기 예측: 2-bit 카운터를 사용하여 추척. 이전 예측을 기억하고 패턴이 변경될 때 빠르게 적응
  - 제한사항1: 예측 정보를 2-bit로 유지해야하므로 추가적인 하드웨어 비용이 듦
  - 제한사항2: 예측의 역전과 관련된 상황에서 한계에 도달할 수 있음. 예) 반복적으로 참과 거짓을 번갈아 내는 패턴이 있는 경우, 2-bit은 예측이 맞는 경우에도 계속해서 잘못된 예측을 수행할 수 있음

Q20.

When we increase the block size of a cache, the miss rate initially decreases. But if the block size becomes a large fraction of the cache size, the miss rate increases again. Explain the reasons. (캐시의 블록 크기를 증가시킬 때 미스 비율은 처음에 감소합니다. 그러나 블록 크기가 캐시 크기의 큰 일부분이 되면 다시 미스 비율이 증가하는 이유를 설명하세요.)

- 블록 사이즈가 커지면, 데이터가 한 블록에 더 많아져서 처음에는 miss rate가 줄어듬. Spatial locality 활용 가능
- 블록 사이즈가 더 커지면, block 갯수가 줄어들어 block이 쫓겨날 확률이 높아지기 때문에 miss rate가 늘어남.

Q10. (2022)

Why do we need TLB (Translation Lookahead Buffer). In case of page fault, explain the operation of the TLB? (왜 우리는 TLB (Translation 미리보기 버퍼)가 필요한지에 대해 설명하고, 페이지 폴트가 발생하는 경우 TLB의 작동을 설명하세요.)

- 속도 증가 : 가상 메모리 주소를 물리적인 주소로 변환하는 속도 (매번 메모리에 접근하지 않아도 주소 변환 결과를 빠르게 얻을 수 있음)를 높이기 위해 사용되는 캐시
- 메모리 효율성**:** TLB는 메모리 접근을 줄이는 데 도움을 줍니다. 메모리에 직접 접근하는 대신 TLB를 통해 주소 변환을 수행하면 메모리 대역폭을 절약할 수 있으며, 이는 메모리 효율성을 높임
- 페이지 테이블의 일부 내용, 최근에 사용된 페이지 위주로 저장
- 페이지 폴트(page fault)가 발생하는 경우 TLB의 작동
  1. TLB의 내용은 무효화되고, 운영 체제는 페이지 테이블을 검색하여 가상 주소를 실제 물리 주소로 변환.

2. TLB를 업데이트하여 향후 주소 변환에 사용될 정보를 갱신하고, 데이터를 로드하여 프로세스가 해당 주소에 접근할 수 있게함.
3. 페이지 폴트 후 재시도할 때 TLB를 통해 주소 변환을 수행하므로 메모리 액세스 지연을 최소화함

Q13. (2022)

Explain the relation between processor frequency and clock cycle time. Additionally, in case of 2GHz CPU, what is the clock cycle time in terms of ps(pico second)? (프로세서 주파수와 클럭 주기 시간 사이의 관계를 설명하고, 2GHz CPU의 경우 클럭 주기 시간을 피코초(pico second) 단위로 어떻게 계산하는지 설명하세요.)

- 클럭 주기 시간(초) = 1 / 클럭 주파수(헤르츠)
- 클록 주기 (clock cycle time) : 1 클록에 걸리는 시간 (1ps = 10^{-12} s)
- 클록 frequency : CPU의 1초당 클록 사이클 수
  - 4GHz = 4000MHz = 4×10^9Hz
  - 클록 주기 = 1s / (2×10^9)Hz * 10^{12} ps/s = 500 ps