Mobile processor Programming

assignment #4: S MIPS emulator with Cache

모바일시스템공학과 김태경 (32211203)

목차

Freeday used 5
Freeday left 0

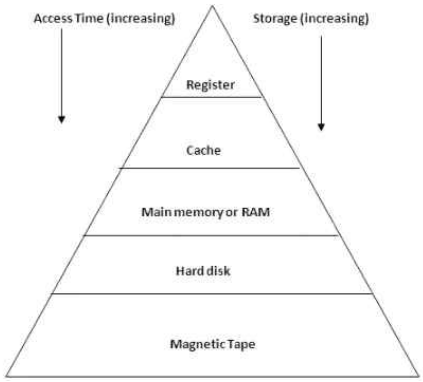
2024.06.01.

1. 프로젝트 목표

이 프로젝트의 목표는 파이프라인(또는 single cycle) MIPS 에뮬레이터와 캐시 시스템을 구현하여 캐시 구조와 성능을 분석하는 것이다. 학생들은 바이너리 파일을 메모리에 로드하고 초기 레지스터 값을 설정하며, 다양한 캐시 설정과 정책을 구현한다. 또한, 캐시와 메모리 접근 지연 시간을 고려하여 성능을 최적화하고, 실행 후 성능 통계를 출력하여 분석한다. 최종적으로, 이론적 지식을 실제로 적용하고 분석하는 능력을 기르는 것이다.

Cache에 대해

캐시 메모리는 CPU와 주기억장치(RAM) 사이에 위치하여, 자주 사용되는 데이터나 프로그램을 고속으로 접근할 수 있도록 저장하는 고속 버퍼 메모리다. 캐시 메모리는 CPU의 처리 속도와 RAM의 접근 속도 차이로 인해 발생하는 병목 현상을 줄이기 위해 사용된다. 이를 통해 CPU가 데이터를 빠르게 받아 처리할 수 있어 전체 시스템 성능이 향상된다.



캐시 메모리의 특징

- 고속 접근: 캐시 메모리는 주기억장치보다 훨씬 빠른 속도로 데이터를 읽고 쓸 수 있다.
- 작은 용량: 캐시 메모리는 일반적으로 수십 KB에서 수 MB 정도로, 주기억장치에 비해 용량이 매우 작다.
- 비용: 캐시 메모리는 빠른 속도를 제공하기 위해 고가의 메모리 기술(SRAM)을 사용하므로 비용이 높다.
- 지역성(Locality): 캐시 메모리는 시간적 지역성과 공간적 지역성을 활용하여 자주 사용되는 데이터에 빠르게 접근한다.

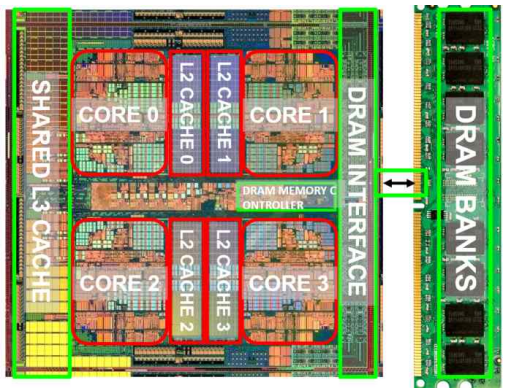
캐시의 지역성

- 시간적 지역성(Temporal Locality): 최근에 접근한 데이터는 다시 접근될 가능성이 높다.
- 공간적 지역성(Spatial Locality): 최근에 접근한 데이터 근처의 데이터도 접근될 가능성이 높다.
- 순차적 지역성(sequential locality) : 분기가 발생하지 않는 한 명령어는 메모리에 저장된 순서대로 인출/실행된다.

캐시의 계층 구조

: 시스템에 장착된 캐시의 용량과 성능이 점점 증가하면서 캐시의 캐시로 사용되는 메모리가 추가되었는데, 이것을 적용된 순서대로 L(Level) 1, L2, L3 라고 부른다.

- L1 캐시 (Level 1 Cache)
  위치: CPU 코어 내부
  속도: 가장 빠름
  용량: 수십 KB (일반적으로 32KB ~ 64KB)
  역할: 자주 접근하는 데이터와 명령어를 저장하여 빠른 응답 속도를 제공함



- L2 캐시 (Level 2 Cache)
  위치: CPU 코어 외부, CPU 다이에 통합
  속도: L1 캐시보다 느림
  용량: 수백 KB ~ 몇 MB (일반적으로 256KB ~ 8MB)
  역할: L1 캐시의 보조 역할을 하며, 더 많은 데이터를 저장함
- L3 캐시 (Level 3 Cache)
  위치: CPU 다이에 통합, 여러 CPU 코어가 공유
  속도: L2 캐시보다 느림
  용량: 수 MB ~ 수십 MB (일반적으로 4MB ~ 32MB)
  역할: 여러 CPU 코어 간에 데이터 공유 및 일관성을 유지하고, L2 캐시의 보조 역할을 함

2. 프로젝트 조건

기본요구사항

1. 초기화
- 바이너리 파일을 메모리에 로드한다.
- 메모리는 큰 배열로 정의되며 파일의 모든 내용을 읽어들인다.
- 레지스터 값은 모두 0으로 초기화되며, 리턴 어드레스 레지스터(RA, r31)는 0xFFFF로, 스택 포인터(SP, r29)는 0x100000으로 설정한다.
- 프로그램 카운터(PC)는 0x0에서 시작한다.
- 캐시 메모리는 모두 0으로 초기화한다.

2. 프로그램 완료 조건
- 실행이 끝나면 계산된 결과 값을 출력해야 한다. PC가 0xFFFF로 이동하면 실행이 완료되며, 계산된 결과는 v0(또는 r2) 레지스터에 저장된다.

3. 캐시 구성
- 캐시 라인 크기: 64바이트
- 세트 연관도: 직접 매핑(Direct-mapped), 2-way 등으로 구성 가능
- 캐시 크기: 64바이트, 128바이트, 256바이트 등으로 구성 가능
- 교체 알고리즘: LRU(Least Recently Used) 알고리즘을 사용하여 적절한 교체 알고리즘을 구현한다.
- 쓰기 정책: Write-back 또는 Write-through 정책을 구현한다.
- 사이클: 캐시 접근 지연 시간은 1 CPU 클럭 사이클이며, 메모리 접근 지연 시간은 1000 CPU 클럭 사이클이다.

4. 실행 결과 출력
- 총 실행 사이클 수 / 레지스터 작업 수 / 메모리 (로드/스토어) 작업 수.
- 분기 수 (총 분기 수 / 성공한 분기 수).
- 캐시 히트/미스 통계 (콜드 미스 및 충돌 미스 포함).

3. 프로그램 빌드 환경

컴파일 환경: Microsoft Visual Studio
프로젝트 언어: C언어
실행 파일: pipeline.c / simple , 2, 3, 4, 5, gcd, fib, input.bin

3. 프로젝트에 사용된 개념

캐시의 주요 개념

1. 캐시 히트(Cache Hit) & 캐시 미스(Cache Miss): CPU가 요청한 데이터가 캐시에 존재하는 경우를 말하며, 데이터는 캐시에서 즉시 제공되어 메모리 접근 시간을 줄인다. CPU가 요청한 데이터가 캐시에 존재하지 않는 경우를 말하며, 데이터는 메인 메모리에서 가져와야 하며, 이후 캐시에 저장된다.

2. 태그(Tag): 캐시 블록이 메모리의 어느 주소에 해당하는지를 나타내는 정보이다. 태그는 블록의 주소를 식별하는 데 사용된다.

3. 캐시 블록(Cache Block): 캐시는 블록 단위로 데이터를 저장하며, 각 블록은 여러 개의 연속된 메모리 주소를 포함한다. 이를 통해 공간 효율성을 높이고 데이터 지역성을 활용한다.

4. Set-Associative Cache: 캐시 블록을 여러 개의 셋으로 나누고, 각 셋에는 여러 개의 라인(line)이 있다. 이 방식은 블록을 특정 셋 내의 어느 라인에 저장할지 선택할 수 있어 충돌 미스를 줄인다. 세트 연관도에 따라 다음과 같은 방식으로 구분된다.

캐시의 작동 원리

주소 분해(Address Decomposition):
CPU가 메모리 주소를 요청하면, 주소는 태그, 인덱스, 오프셋으로 분해된다.
태그는 블록의 식별자 역할을 하고, 인덱스는 캐시의 특정 셋을 가리키며, 오프셋은 블록 내의 특정 데이터를 가리킨다.

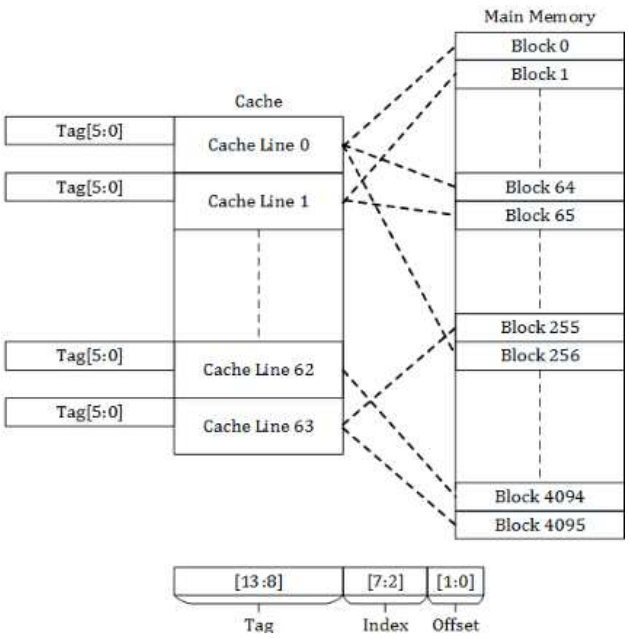
캐시 조회(Cache Lookup)
: 인덱스를 사용하여 해당 셋을 조회하고, 태그를 비교하여 캐시 히트 여부를 판단한다.
- 캐시 히트일 경우, 요청한 데이터는 캐시에서 즉시 제공한다.
- 캐시 미스일 경우, 메인 메모리에서 데이터를 가져와 캐시에 저장하고, 이를 CPU에 제공한다.

교체 알고리즘
LRU (Least Recently Used)
: 가장 오랫동안 사용되지 않은 데이터를 교체하는 방식이다.

- 장점: 자주 사용되는 데이터가 캐시에 오래 유지될 수 있어 히트율이 높다.
- 단점: 구현이 복잡하며, 각 데이터의 사용 시간을 추적하고 관리하는 오버헤드가 발생한다.

캐싱 라인의 매핑 방법:

1. 직접 매핑(Direct Mapping)
: 직접 매핑은 메인 메모리를 일정 블록으로 나누어 각 블록을 캐시의 특정 영역에 매핑하는 방식이다.
- 작동 원리: 메인 메모리의 주소는 태그(Tag), 인덱스(Index), 오프셋(Offset)으로 나뉜다. 인덱스는 캐시의 특정 라인을 가리키며, 해당 라인에 데이터가 저장된다. 데이터 접근 시, 태그를 비교하여 적중 여부를 판단한다.



- 장점:
    1. 구현의 용이성: 구조가 단순하여 하드웨어 구현이 쉽다.
    2. 속도: 인덱스를 사용해 특정 위치를 바로 찾을 수 있어 검색 속도가 빠르다.

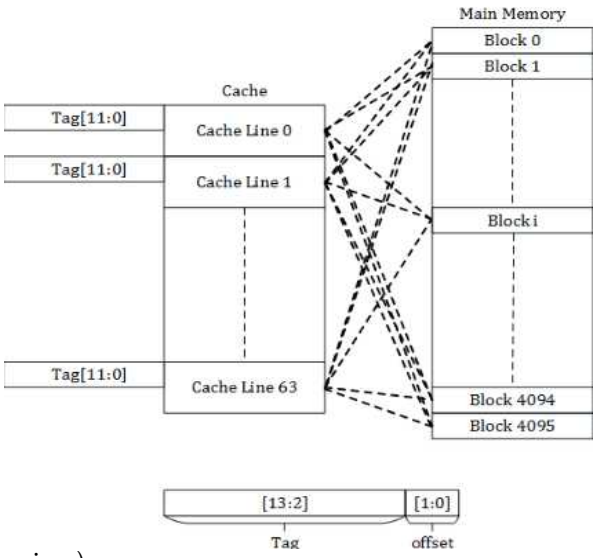
- 단점:
    1. 충돌 문제: 서로 다른 메모리 블록이 동일한 캐시 라인에 매핑될 경우, 충돌이 발생하여 캐시 적중률이 낮아질 수 있다.
    2. 낮은 적중률: 특정 패턴의 데이터 접근이 반복될 경우 캐시 히트율이 크게 떨어질 수 있다.

2. 완전 연관 매핑(Fully Associative Mapping)
: 완전 연관 매핑은 메인 메모리의 데이터를 캐시의 빈 공간 어디에나 자유롭게 저장할 수 있는 방식이다.
- 작동 원리: 메인 메모리의 주소는 태그(Tag)와 오프셋(Offset)으로 나뉜다. 캐시의 모든 라인에 데이터를 저장할 수 있으며, 데이터 접근 시 모든 태그를 비교하여 적중 여부를 판단한다.

- 장점:

= 높은 적중률: 데이터가 어느 위치에든 저장될 수 있어 충돌 문제가 발생하지 않는다.

= 유연성: 데이터가 자주 접근되는 위치에 자유롭게 배치될 수 있어 효율적이다.



- 단점:

= 구현의 복잡성: 모든 태그를 병렬로 비교해야 하므로 하드웨어 구현이 복잡하고 비용이 높다.

= 속도: 태그 비교를 병렬로 수행해야 하므로 검색 속도가 느릴 수 있다.

3. 집합 연관 매핑(Set Associative Mapping)

: 집합 연관 매핑은 직접 매핑과 완전 연관 매핑의 장점을 결합한 방식으로, 데이터의 적중률과 구현의 복잡도 사이의 균형을 맞춘다.

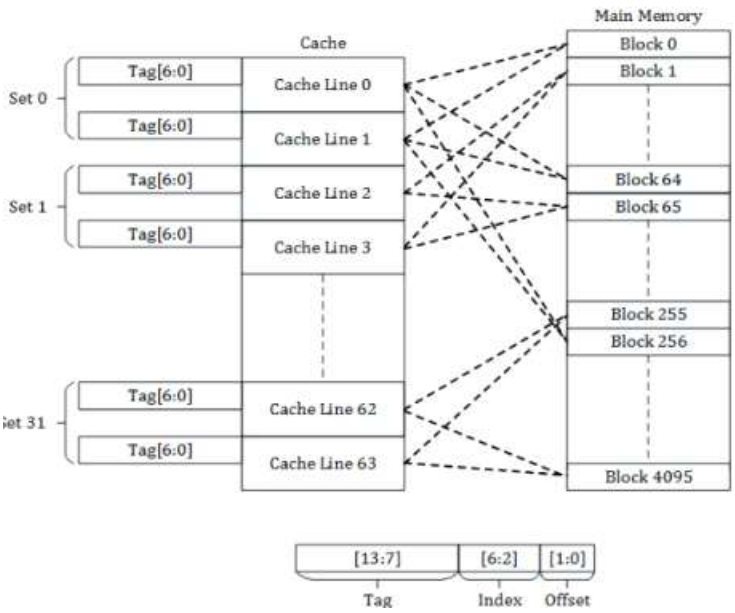
- 작동 원리: 캐시는 여러 개의 세트(set)로 나뉘며, 각 세트는 여러 개의 라인(line)으로 구성된다. 메인 메모리의 주소는 태그(Tag), 세트 인덱스(Set Index), 오프셋(Offset)으로 나뉜다. 세트 인덱스를 사용하여 특정 세트를 선택하고, 해당 세트 내의 라인 중 하나에 데이터를 저장한다. 데이터 접근 시 세트 내의 모든 태그를 비교하여 적중 여부를 판단한다.

- 장점:

= 적중률 향상: 동일한 세트 내에서 여러 라인을 사용할 수 있어 충돌 문제가 감소한다.

= 구현 용이성: 완전 연관 매핑보다 구현이 간단하며, 태그 비교도 세트 내에서만 이루어져 병렬 비교의 부담이 줄어든다.

= 유연성: 데이터가 특정 세트 내에서 자유롭게 배치될 수 있어 효율적이다.



- 단점:
  = 복잡성 증가: 직접 매핑보다는 복잡하며, 세트 인덱싱과 태그 비교를 동시에 수행해야 한다.
  = 비용: 하드웨어 구현 비용이 직접 매핑보다 높다.

Cache Miss 유형

: 캐시 미스는 CPU가 요청한 데이터가 캐시에 없을 때 발생하며, 주요 유형으로는 콜드 미스, 용량 미스, 충돌 미스, 일관성 미스가 있다.

1. 콜드 미스 (Cold Miss)

: 데이터가 처음 캐시에 접근될 때 발생
- 원인: 이전에 한 번도 캐시에 로드된 적이 없음
- 해결: 프리페칭으로 일부 콜드 미스를 줄일 수 있음

2. 용량 미스 (Capacity Miss)

: 캐시 용량이 부족하여 필요한 데이터를 모두 저장할 수 없을 때 발생
- 원인: 캐시 크기가 작아 최근 사용 데이터를 유지하지 못함
- 해결: 캐시 크기 증가 또는 효율적인 캐시 관리 알고리즘 사용

3. 충돌 미스 (Conflict Miss)

: 서로 다른 데이터가 동일한 캐시 라인에 매핑되어 발생
- 원인: 특정 인덱스에 데이터가 집중되어 발생
- 해결: 더 높은 집합 연관 매핑 또는 더 나은 매핑 알고리즘 사용

4. 일관성 미스 (Coherency Miss)

: 캐시 일관성을 유지하기 위해 데이터가 무효화되거나 갱신될 때 발생
- 원인: 멀티프로세서 시스템에서 데이터 일관성 유지 과정에서 발생
- 해결: 일관성 프로토콜 최적화 또는 미스 최소화 접근 패턴 사용

캐시 일관성 문제해결

캐시 일관성 문제는 공유 메모리 구조에서 여러 프로세스가 동일한 데이터를 사용할 때 데이터 불일치가 발생하는 문제를 해결하기 위한 방법들이다. 주요 해결 방법으로는 공유 캐시 사용, 버스 감시 메커니즘, 디렉토리 기반 캐시 일관성 유지 방법 등이 있다.

1, 공유 캐시 사용:
모든 프로세스가 하나의 캐시를 공유하여 사용하는 방식이다.
- 장점: 구조가 단순하고, 데이터 일관성을 유지하는 데 추가적인 복잡성이 없다.
- 단점: 프로세스 간 충돌이 심해질 수 있으며, 병목 현상이 발생할 수 있다. 여러 프로세스가 동시에 캐시를 접근하려고 할 때 성능이 저하될 수 있다.

2. 버스 감시 메커니즘 (Bus Snooping Mechanism):
각 캐시가 버스를 통해 다른 캐시의 메모리 접근을 감시하고, 필요 시 캐시를 갱신하는 방식이다. 스누피 제어기(Snoopy Controller)가 이러한 감시 작업을 수행한다.
- 작동 원리: 프로세서가 캐시 블록을 변경하면 해당 변경 사항을 버스를 통해 브로드캐스트한다. 다른 캐시들이 이 신호를 감지하고, 동일한 데이터 블록이 있을 경우 해당 블록을 무효화(invalidate)하거나 갱신(update)한다.
- 장점: 실시간으로 캐시 일관성을 유지할 수 있으며, 구현이 비교적 간단하다.
- 단점: 버스 트래픽이 증가하여 성능이 저하될 수 있다. 특히 많은 프로세서가 존재하는 시스템에서는 확장성이 떨어질 수 있다.

3. 디렉토리 기반 캐시 일관성 유지 방법 (Directory-Based Coherence):
캐시 블록의 상태 정보를 중앙 디렉토리에 저장하여 데이터 일관성을 유지하는 방식이다.
- 작동 원리: 중앙 디렉토리는 각 캐시 블록의 상태와 해당 블록을 캐싱하고 있는 프로세서를 추적한다. 프로세서가 데이터를 읽거나 쓸 때, 디렉토리를 참조하여 일관성을 유지한다. 데이터 변경 시 디렉토리를 갱신하고, 관련 캐시에 변경 사항을 전파한다.
- 장점: 확장성이 뛰어나며, 많은 수의 프로세서를 지원할 수 있다.
- 단점: 디렉토리를 관리하기 위한 추가적인 오버헤드가 발생하며, 디렉토리 접근 지연이 문제가 될 수 있다.

캐시 메모리의 성능:
캐시 크기, 인출 방식, 쓰기 정책, 교체 알고리즘에 따라 달라진다.

1. 캐시 크기
캐시가 클수록 더 많은 데이터를 저장할 수 있어 캐시 히트율이 높아지지만, 캐시 메모리의 비용도 증가한다. 또한, 너무 큰 캐시는 관리 오버헤드가 발생할 수 있다.

2. 인출 방식

- 요구 인출 (Demand Fetch): 필요 시 데이터를 인출하는 방식으로, 불필요한 데이터 인출을 피할 수 있으나 지연 시간이 발생할 수 있습니다.
- 선 인출 (Pre-Fetch): 예상되는 데이터를 미리 인출하는 방식으로, 접근 시 지연 시간을 줄일 수 있으나 예측 부정확 시 캐시 공간을 낭비할 수 있습니다.

3. 쓰기 정책
- Write-Through: 데이터를 캐시와 메인 메모리에 동시에 기록하는 방식으로, 데이터 일관성을 유지하지만 쓰기 동작 시 성능 저하가 발생할 수 있습니다.
- Write-Back: 캐시에만 데이터를 쓰고, 캐시에서 제거될 때 메인 메모리에 기록하는 방식으로, 성능이 향상되나 데이터 일관성을 유지하기 위한 관리가 필요합니다.

4. 교체 알고리즘
- FIFO (First In First Out): 먼저 들어온 데이터를 먼저 교체하는 방식으로, 구현이 간단하지만 히트율이 낮을 수 있습니다.
- LRU (Least Recently Used): 가장 오랫동안 사용되지 않은 데이터를 교체하는 방식으로, 히트율이 높으나 구현이 복잡합니다.
- LFU (Least Frequently Used): 사용 빈도가 적은 데이터를 교체하는 방식으로, 사용되지 않는 데이터를 효율적으로 제거할 수 있으나 구현이 복잡합니다.
- Random: 무작위로 데이터를 교체하는 방식으로, 구현이 매우 간단하지만 히트율이 낮을 수 있습니다.
- Optimal (Belady's MIN): 향후 가장 참조되지 않을 데이터를 교체하는 이상적인 방식으로, 실제 구현이 불가능하여 주로 이론적 분석에 사용됩니다.

5. 프로그램 구성

과제 2번인 single cycle MIPS의 구현에 캐시를 추가적으로 구현하였다.

공통

총 사이클 수 확인 용 cycle count

```
int total_cycles = 0;
```

캐시라인 구조체 선언 **direct mapped**

```
typedef struct {
    int tag;
    int valid;
    int dirty;
    int data[16];
}cacheline;
```

fully associate

```
typedef struct {
    int tag;
    int val;
    int state;
    int dirty;
    int data[16];
}cacheline;
```

캐시라인 배열 선언
old: 용량 미스 해결용 변수
캐시의 miss, hit count

```
cacheline cache[64];
int old;
int cache_hit;
int cache_miss;
```

```
cacheline cache[64];
int old;
int cache_hit;
int cache_miss;
```

Direct Mapped Read & Write Memory 함수

Read Memory

주소 계산

```
int tmp = addr - (addr % 64);
int tag = (addr / 4 & 0xfffff800) >> 11;
int index = (addr / 4 & 0x7f0) >> 4;
int offset = (addr / 4 & 0xf);
```

- addr 매개변수를 64바이트 단위로 변환
- tag(12bit), index(14bit), offset(6bit)

캐시 히트

```
//cache hit
if ((cache[index].tag == tag) && (cache[index].valid == 1))
{
    cache_hit++;
    total_cycles++;
    return cache[index].data[offset];
}
```

캐시에서 해당 인덱스의 라인이 유효하고(valid == 1), 태그가 일치하면 캐시 히트로 간주하고, 캐시에서 데이터를 읽어 반환, 캐시 히트 수를 증가시키며, 총 사이클 수를 1 증가시킵니다

캐시 미스

```
//cache cold miss
cache_miss++;
total_cycles += 1000;
if (cache[index].valid == 0)
{
    cache[index].valid = 1;
    cache[index].tag = tag;
    for (int n = 0; n < 16; n++)
    {
        cache[index].data[n] = memory[tmp / 4];
        tmp = tmp + 4;
    }
    return memory[addr / 4];
}
```

```
//cache conflict miss
else {
    if (cache[index].dirty == 1)
    {
        tmp = (cache[index].tag << 13) | (index << 6)
        for (int n = 0; n < 16; n++)
        {
            memory[tmp / 4] = cache[index].data[n];
            tmp = tmp + 4;
        }
    }
    cache[index].tag = tag;
    tmp = addr - (addr % 64);
    for (int n = 0; n < 16; n++)
    {
        cache[index].data[n] = memory[tmp / 4];
        tmp = tmp + 4;
    }
    return memory[addr / 4];
}
```

캐시 라인이 유효하지 않거나 태그가 일치하지 않으면 캐시 미스로 간주하고 캐시 미스 수를 증가시키고 총 사이클 수를 1000 증가시킨다..

- 콜드 미스: 캐시 라인이 유효하지 않으면, 메모리에서 16개의 데이터를 읽어와 캐시 라인을 갱신하고, 해당 데이터를 반환한다.
- 컨플릭트 미스: 캐시 라인이 유효하지만 태그가 다르면, 캐시 라인이 더티(dirty == 1)한 경우 기존 데이터를 메모리에 저장하고, 새 데이터를 읽어와 캐시 라인을 갱신한 후 데이터를 반환한다.

Wirte Memory

주소 계산: Read Memory와 동일하다.

캐시 히트

```
//cache hit
if ((cache[index].tag == tag) && (cache[index].valid == 1))
{
    cache[index].dirty = 1;
    cache[index].data[offset] = val;
    cache_hit++;
    total_cycles++;
    return;
}
```

캐시에서 해당 인덱스의 라인이 유효하고 태그가 일치하면 캐시 히트

캐시 미스:

캐시 미스 수를 증가시키고 총 사이클 수를 1000 증가시킨다.

- 콜드 미스: 캐시 라인이 유효하지 않으면, 메모리에서 16개의 데이터를 읽어와 캐시 라인을 갱신하고, 새로운 데이터를 캐시에 저장한 후 더티 비트를 설정합니다.
- 컨플릭트 미스: 캐시 라인이 유효하지만 태그가 다르면, 캐시 라인이 더티한 경우 기존 데이터를 메모리에 저장한 후 새 데이터를 읽어와 캐시 라인을 갱신하고 새로운 데이터를 캐시에 저장한 후 더티 비트를 설정합니다.

```
//cache cold miss
cache_miss++;
total_cycles += 1000;
if (cache[index].valid == 0)
{
    cache[index].valid = 1;
    cache[index].tag = tag;
    for (int n = 0; n < 16; n++)
    {
        cache[index].data[n] = memory[tmp / 4];
        tmp = tmp + 4;
    }
    cache[index].data[offset] = val;
    cache[index].dirty = 1;
    return;
}
```

```
//cache conflict miss
else {
    if (cache[index].dirty == 1)
    {
        tmp = (cache[index].tag << 13) | (index << 6);
        for (int n = 0; n < 16; n++)
        {
            memory[tmp / 4] = cache[index].data[n];
            tmp = tmp + 4;
        }
    }
    cache[index].tag = tag;
    tmp = addr - (addr % 64);
    for (int n = 0; n < 16; n++)
    {
        cache[index].data[n] = memory[tmp / 4];
        tmp = tmp + 4;
    }
    cache[index].data[offset] = val;
    cache[index].dirty = 1;
    return;
}
```

Fully associate Read & Write Memory 함수

Read Memory

주소 계산

```
int tmp = Addr - (Addr % 64);
int tag = (Addr / 64) & 0xffffffff;
int offset = (Addr % 64) / 4;
```

- addr 매개변수에서 64바이트 단위의 주소 변환
- tag(26bit), offset(6bit)

캐시 히트:

```
//cache hit
for (int i = 0; i < 64; i++)
{
    if ((cache[i].tag == tag) && (cache[i].val == 1))
    {
        cache[i].state = 1;
        cache_hit++;
        total_cycles++;
        return cache[i].data[offset];
    }
}
```

캐시의 모든 라인(총 64개)을 검사하여, 태그가 일치하고 유효 비트(val)가 1인 경우 캐시 히트로 간주한다.

- 캐시의 데이터를 반환하고, 캐시 히트 수를 증가시키며, 총 사이클 수를 1 증가시킨다.

캐시 미스:

캐시의 모든 라인을 검사하여 유효 비트(val)가 0인 라인을 찾으면, 이를 캐시 콜드 미스로 간주한다. 캐시 미스 수를 증가시키고 총 사이클 수를 1000 증가시킨다.

- 캐시 콜드 미스의 경우, 메모리에서 16개의 데이터를 읽어와 캐시 라인을 갱신하고 해당 데이터를 반환한다.
- 캐시 라인이 유효하지만 태그가 다르면, 캐시 컨플릭트 미스로 간주하고 LRU(Least Recently Used) 알고리즘에 따라 캐시 라인을 교체한다.

```
//cache cold miss
cache_miss++;
total_cycles += 1000;
for (int i = 0; i < 64; i++)
{
    if (cache[i].val == 0)
    {
        cache[i].val = 1;
        cache[i].tag = tag;
        for (int n = 0; n < 16; n++)
        {
            cache[i].data[n] = memory[tmp / 4];
            tmp = tmp + 4;
        }
        return memory[Addr / 4];
    }
}
```

```
//cache conflict miss
while (1) {
    if (cache[old].state == 0)
    {
        if (cache[old].dirty == 1)
        {
            tmp = cache[old].tag << 6;
            for (int n = 0; n < 16; n++)
            {
                memory[tmp / 4] = cache[old].data[n];
                tmp = tmp + 4;
            }
        }
        cache[old].tag = tag;
        tmp = Addr - (Addr % 64);
        for (int n = 0; n < 16; n++)
        {
            cache[old].data[n] = memory[tmp / 4];
            tmp = tmp + 4;
        }
        old = old + 1;
        if (old == 64)
        {
            old = 0;
        }
        return memory[Addr / 4];
    }
    else {
        cache[old].state = 0;
        old = old + 1;
        if (old == 64)
        {
            old = 0;
        }
    }
}
```

Wirte Memory

주소 계산: Read Memory와 동일하다.

캐시 히트:

```
//cache hit
for (int i = 0; i < 64; i++)
{
    if ((cache[i].tag == tag) && (cache[i].val == 1))
    {
        cache[i].state = 1;
        cache[i].dirty = 1;
        cache[i].data[offset] = valid;
        cache_hit++;
        total_cycles++;
        return;
    }
}
```

캐시의 모든 라인(총 64개)을 검사하여, 태그가 일치하고 유효 비트(val)가 1인 경우 캐시 히트로 간주한다.

- 캐시 데이터를 갱신하고 더티 비트(dirty)를 설정한 후, 캐시 히트 수와 총 사이클 수를 각각 증가시킨다.

캐시 미스:

- 캐시 미스 수를 증가시키고 총 사이클 수를 1000 증가시킵니다.
- 캐시 콜드 미스의 경우, 메모리에서 16개의 데이터를 읽어와 캐시 라인을 갱신하고, 새로운 데이터를 캐시에 저장한 후 더티 비트를 설정합니다.
- 캐시 컨플릭트 미스의 경우, LRU 알고리즘에 따라 캐시 라인을 교체하고 새로운 데이터를 캐시에 저장한 후 더티 비트를 설정합니다.

```
//cache cold miss
cache_miss++;
total_cycles += 1000;
for (int i = 0; i < 64; i++)
{
    if (cache[i].val == 0)
    {
        cache[i].val = 1;
        cache[i].tag = tag;
        for (int n = 0; n < 16; n++)
        {
            cache[i].data[n] = memory[tmp / 4];
            tmp = tmp + 4;
        }
        cache[i].dirty = 1;
        cache[i].data[offset] = valid;
        return;
    }
}
```

```
//cache conflict miss
while (1) {
    if (cache[old].state == 0)
    {
        if (cache[old].dirty == 1)
        {
            tmp = (cache[old].tag << 6);
            for (int n = 0; n < 16; n++)
            {
                memory[tmp / 4] = cache[old].data[n];
                tmp = tmp + 4;
            }
        }
        cache[old].tag = tag;
        tmp = Addr - (Addr % 64);
        for (int n = 0; n < 16; n++)
        {
            cache[old].data[n] = memory[tmp / 4];
            tmp = tmp + 4;
        }
        cache[old].dirty = 1;
        cache[old].data[offset] = valid;
        old = old + 1;
        if (old == 64)
        {
            old = 0;
        }
        return;
    }
    else
    {
        cache[old].state = 0;
        old = old + 1;
        if (old == 64)
        {
            old = 0;
        }
    }
}
```

Read & Write Memory의 적용
// Direct, Fully 동일하게 적용됨

instruction Fetch

```
// Instruction Fetch
int inst = Read_Memory(pc);
total_cycles++; // fetch cycle count
```

Load Word

```
case 0x23: // Load Word
    AluResult = Regs[rs] + s_imm;
    printf("@ 0x%x LW:  M[ R[%d]:0x%x + 0x%x] -> R[%d]: 0x%x\n",
        pc, rs, Regs[rs], s_imm, rt, AluResult);
    Regs[rt] = Read_Memory(AluResult);
    pc = pc + 4;
    M_count = M_count + 1;
    I_inst = I_inst + 1;
    total_cycles += 3;  // cycle Ex, Mem, WB count
    break;
```

Store Word

```
case 0x2b: // Store Word
    AluResult = Regs[rs] + s_imm;
    printf("@ 0x%x SW:  R[%d]: 0x%x -> M[ R[%d]:0x%x + s_imm 0x%x ]\n",
        pc, rt, Regs[rt], rs, Regs[rs], s_imm);
    Write_Memory(AluResult, Regs[rt]);
    pc = pc + 4;
    M_count = M_count + 1;
    I_inst = I_inst + 1;
    total_cycles += 3;  // cycle Ex, Mem, WB count
    break;
```

cycle 카운트

Fetch cycle +1

```
// Instruction Fetch
int inst = Read_Memory(pc);
total_cycles++; // fetch cycle count
```

Decode cycle +1

```
// Instruction Decode
int opcode = (inst >> 26) & 0x3f;
int rs = (inst >> 21) & 0x1f;
int rt = (inst >> 16) & 0x1f;
int rd = (inst >> 11) & 0x1f;
int shamt = (inst >> 6) & 0x1f;
int func = inst & 0x0000003f;
int imm_ = inst & 0x0000ffff;
int addr = inst & 0x03ffffff;
int s_imm = (imm_ & 0x8000) ? (imm_ | 0xffff0000) : (imm_);
int z_imm = imm_ & 0x0000ffff;
int b_addr = s_imm * 4;
int j_addr = ((pc + 4) & 0xf0000000) | (addr << 2);
int test = inst;
total_cycles++; // decode cycle count
```

Execution, Memory Access, Write Back cycle +3

ex): switch-case문의 case별 적용

```
case 0x8: // Add Imm
    AluResult = Regs[rs] + s_imm;
    printf("@ 0x%x Addi :  R[%d]: 0x%x + 0x%x -> R[%d]: 0x%x₩n",
        pc, rs, Regs[rs], s_imm, rt, AluResult);
    Regs[rt] = AluResult;
    pc = pc + 4;
    I_inst = I_inst + 1;
    total_cycles += 3;  // cycle Ex, Mem, WB count
    break;
```

Read Memory 중

cache hit cycle +1

cache miss cycle + 1000

Write Memory 중

cache hit cycle +1

cache miss cycle +1000

6. 문제점 및 해결방법

1) Read & Write Memory의 주소 변환 문제

fully의 주소변환

```
int tmp = Addr - (Addr % 64);
int tag = (Addr / 64) & 0xffffffff;
int offset = (Addr % 64) / 4;
```

fully를 먼저 구현하고 direct를 추가로 구현해보았는데, fully에서는 tag와 offset의 선언에서는 64를 나누고 자리수를 잘라내는 방식으로 구현하였는데, direct에서는 동일하게 했을 때 작동하지 않아, GPT를 통해 다음과 같이 바꾸었더니 작동하였다.

```
int tmp = addr - (addr % 64);
int tag = (addr / 4 & 0xfffff800) >> 11;
int index = (addr / 4 & 0x7f0) >> 4;
int offset = (addr / 4 & 0xf);
```

7. 실행결과

direct mapping

	total cycle	inst Ex	Mem acc	cache num	cache hit	cache miss	cache rate
simple.bin	2048	8	2	10	8	2	80
simple2.bin	2062	10	4	14	12	2	85.71
simple3.bin	11590	1330	613	1943	1940	3	99.84
simple4.bin	11548	243	100	343	333	10	97.08
gcd.bin	29829	1961	486	1547	1523	24	98.51
fib.bin	28158	2679	1095	3774	3763	11	99.70

fully associate

	total cycle	inst Ex	Mem acc	cache num	cache hit	cache miss	cache rate
simple.bin	2048	8	2	10	8	2	80
simple2.bin	2062	10	4	14	12	2	85.71
simple3.bin	11590	1330	613	1943	1940	3	99.84
simple4.bin	11548	243	100	343	333	10	97.08
gcd.bin	29829	1961	486	1547	1523	24	98.51
fib.bin	27831	2679	1095	3774	3763	11	99.70

fib.bin 실행 비교

	total cycle	inst Ex	Mem acc	cache rate
direct	28158	2679	1095	99.70
fully	27831	2679	1095	99.70

다른 파일들로 실행하였을 때 유일하게 차이를 보인 값은 fib.bin파일의 cycle 수였다. 약 300에 가까운 사이클 차이가 발생했는데, 구조상 Direct mapping이 fully associate에 비해 비효율적이어서, 이러한 차이를 보였다.

아마 명령어의 수가 적은 파일들 뿐이라 크게 유의미한 차이를 보이지는 못한 것 같고, input4.bin파일을 통해 cache rate의 차이를 보일 수 있을 것 같은데, 이전의 과제에서 주어진 input4.bin의 원본파일에서 명령어를 수정한 후에 정확한 비교를 할 수 있을 것 같다는 생각이다.

8. 느낀점

이 프로젝트를 통해 파이프라인 MIPS 에뮬레이터와 캐시 시스템을 구현하는 과정 중 pipeline에서 구현하지는 못하고 single cycle에서 구현하였지만, 캐시 메모리는 CPU와 주기억장치 사이에서 병목 현상을 줄여 시스템 성능을 크게 향상시키는 중요한 역할을 한다는 것을 깨달았다.

캐시 메모리의 히트 조건을 가지는 구조와, 다양한 Miss 발생 조건을 파악하고, 다양한 매핑 방법(direct mapping cache, fully associate cache 등)에 대해 학습하는 동시에 일부 구현함으로서 miss 발생을 줄일 수 있는 방법에 대해 파악했다. LRU 알고리즘에 대한 이해는 확실히 하였지만 구현하는데 있어서 어떻게 적용할지가 갈피를 잡기 힘들었는데, 인터넷에서 많이 참고하여 구현할 수 있었다. 이 밖에도 다양한 알고리즘(FIFO, LFU, Random, Optimal)이 있는데 이를 적요하면, 어떻게 구현해야 할지도 궁금하여 따로 공부해볼 예정이다.

실행결과에서는 매핑 방법들 간의 비교 중 input4.bin파일으로 실행해야 확실한 차이를 보여줄 수 있을 것 같은데, 이전의 과제 중 input4.bin파일 내부의 잘못된 명령어(instruction)의 수정을 하지 못하여 실행 중 무한 루프만 계속 돌았다. 따라서 input4.bin은 실행 해볼 수 없었고, 다른 파일들로 비교하였을 때는 크게 유의미한 차이를 보이지는 않았기에 아쉬웠다.

9. 참고자료

- 피라미드 구조 사진 및 참고자료 출처
https://merrily-code.tistory.com/235
- 참고자료 출처
https://zion830.tistory.com/46
- 참고차료 출처
https://m.blog.naver.com/techref/222290343550
- 참고자료 출처
https://gguljaem.tistory.com/entry/%EC%BA%90%EC%8B%9C%EB%A9%94%EB%AA%A8%EB%A6%AC%EC%9D%98-%EA%B0%9C%EB%85%90%EA%B3%BC-%EB%A7%A4%ED%95%91%EA%B8%B0%EB%B2%95%EC%97%90-%EB%8C%80%ED%95%9C-%EC%84%A4%EB%AA%85
- 코드 참고자료 출처
https://dailylifeofdeveloper.tistory.com/355