컴퓨터구조및모바일프로세서 과제4 보고서

과목: 컴퓨터구조및모바일프로세서 2분반
학과: 모바일시스템공학과
학번: 32224020
이름: 전은지
**Rest Freedays: 0

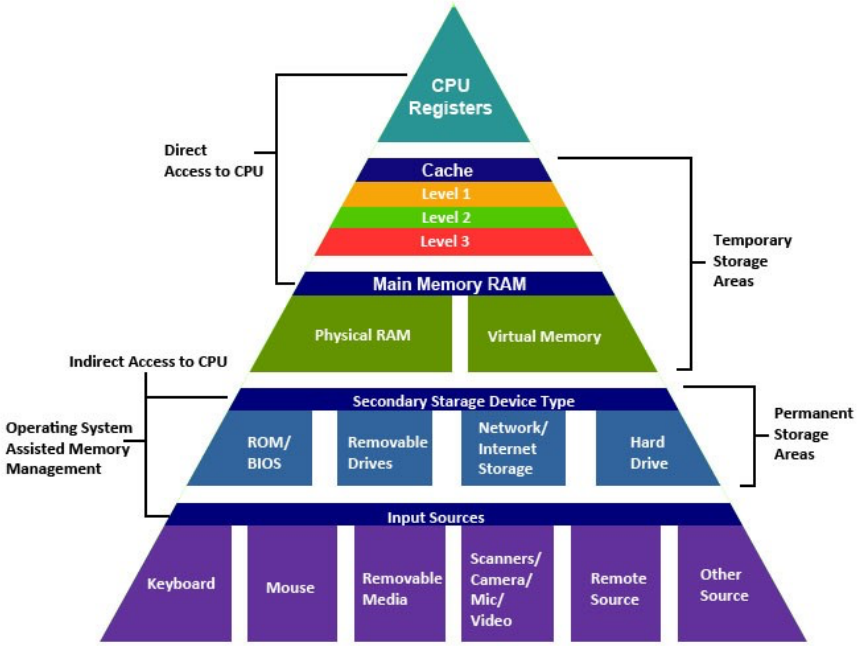
Index

1. Introduction

이 보고서 MIPS 에뮬레이터와 캐시 구현을 목표로 하는 프로그래밍 과제에 대해 다룬다. 현대 컴퓨터 구조에서 메모리 접근 지연 시간과 대역폭은 성능 병목 현상을 유발하는 주요 요인으로 작용한다. 폰 노이만 구조의 컴퓨터는 모든 명령어와 데이터를 메모리에 저장하여 빈번한 메모리 접근을 요구하며, 이는 파이프라인에 대기 사이클을 초래한다. 이를 완화하기 위해 현대 CPU는 빠른 속도의 캐시를 사용하여 메모리 접근 시간을 단축시킨다. 캐시는 메모리 접근 패턴의 공간적 및 시간적 지역성을 활용하여 성능을 향상시키며, 가까운 메모리 주소나 최근에 액세스된 메모리 영역을 다시 액세스할 가능성이 높다는 개념에 기반한다. 본 과제의 목표는 이러한 캐시 구조를 이해하고 성능을 분석하는 것이다. 이를 위해 다양한 캐시 크기와 세트 연관성을 설정하고 교체 및 쓰기 정책을 구현하여 캐시 히트/미스 비율과 평균 메모리 접근 시간을 비교 분석할 것이다. 이 보고서를 통해 MIPS 에뮬레이터와 캐시 구현을 통해 메모리 접근 지연 시간을 줄이고 전체 시스템 성능을 향상시키는 방법을 탐구하며, 캐시의 중요성과 최적화 기법을 습득하는 것을 목표로 한다.

2. Background Information

2.1 Memory Hierarchy



메모리 계층 구조는 컴퓨터 시스템에서 메모리를 여러 계층으로 구성하여 각 계층의 특성과 성능을 최적화하는 개념이다. 이는 메모리 접근 시간을 줄이고 전체 시스템 성능을 향상시키기 위해 설계되었다. 메모리 계층 구조는 주로 다음과 같은 계층으로 구성된다:

1. 레지스터 (Registers)
레지스터는 CPU 내부에 위치한 가장 빠르고 작은 메모리이다. 각 레지스터는 매우 제한된 크기를 가지지만, 명령어 실행에 직접 사용되므로 지연 시간이 거의 없다. CPU는 연산을 수행할 때 주로 레지스터에 저장된 데이터를 사용한다. 레지스터는 나노초(ns) 단위의 접근 시간을 가지며, CPU와 직접 연결되어 있어 매우 높은 속도를 제공한다.

2. 캐시 메모리 (Cache Memory)
캐시는 CPU와 주 메모리(RAM) 사이에 위치한 고속 메모리이다. 캐시는 주로 자주 사용되는 데이터와 명령어를 저장하여 CPU가 빠르게 접근할 수 있도록 한다. 캐시는 일반적으로 L1, L2, L3의 세 가지 레벨로 구분된다:
L1 캐시: CPU 코어에 가장 가까운 위치에 있으며, 매우 빠르지만 크기가 작다. 접근 시간은 수 나노초(ns) 이내이다.
L2 캐시: L1 캐시보다 크지만 약간 느리다. 보통 수십 나노초(ns) 이내의 접근 시간을 가진다.
L3 캐시: 여러 CPU 코어가 공유하는 캐시로, L2 캐시보다 크지만 더 느리다. 접근 시간은 수십에서 수백 나노초(ns) 사이이다.

3. 주 메모리 (Main Memory, RAM)
주 메모리 또는 RAM은 현재 실행 중인 프로그램과 데이터를 저장하는 중간 속도의 메모리이다. RAM은 휘발성 메모리로, 전원이 꺼지면 데이터가 사라진다. 주 메모리는 접근 시간이 수백 나노초(ns)에서 마이크로초(μs) 단위이다. 용량은 수 기가바이트(GB)에서 수십 기가바이트에 이른다.

4. 보조 저장 장치 (Secondary Storage)
보조 저장 장치는 하드 디스크 드라이브(HDD)나 솔리드 스테이트 드라이브(SSD)와 같은 비휘발성 메모리를 포함한다. 이들은 데이터와 프로그램을 영구적으로 저장하며, 접근 시간이 밀리초(ms) 단위로 RAM보다 훨씬 느리다. 그러나 용량이 매우 크고, 데이터를 영구적으로 저장할 수 있다는 장점이 있다.

5. 외부 저장 장치 (Tertiary and Off-line Storage)
외부 저장 장치는 백업 및 대용량 데이터 저장을 위한 장치로, 테이프 드라이브, 광학 디스크(CD/DVD), USB 드라이브 등이 포함된다. 이러한 장치는 주로 데이터 백업 및 장기 보관에 사용되며, 접근 시간이 수 초에서 수 분 단위로 매우 느리다.

2.2 Locality

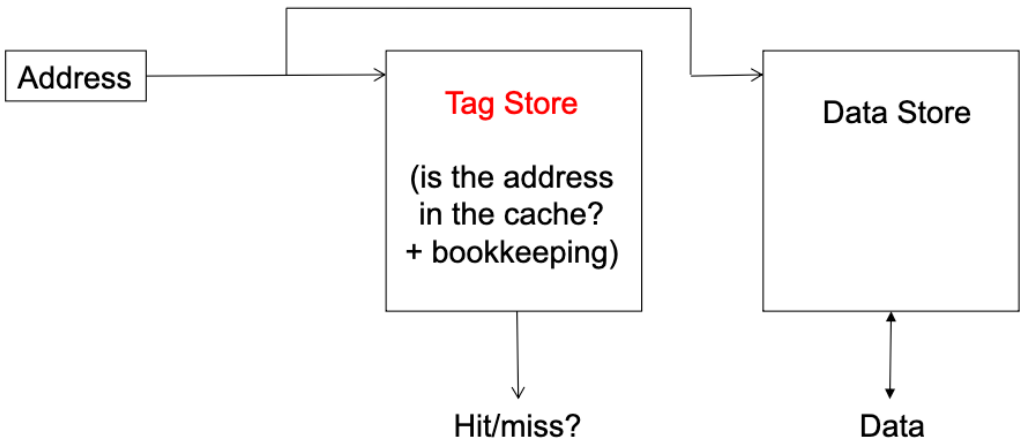
지역성(Locality)은 프로그램이 실행되는 동안 메모리 접근 패턴이 특정한 경향을 보인다는 개념이다. 이는 캐시 메모리의 효율성을 높이는 핵심 요소로, 메모리 시스템 성능 향상에 중요한 역할을 한다. 지역성은 크게 시간적 지역성과 공간적 지역성으로 나눌 수 있다.

시간적 지역성(Temporal Locality)은 최근에 접근한 데이터나 명령어가 곧 다시 접근될 가능성이 높다는 개념이다. 예를 들어, 반복문에서 동일한 변수가 여러 번 사용되는 경우 시간적 지역성이 나타난다. 이는 프로그램이 특정 메모리 위치를 자주 재사용하는 경향을 보이며, 캐시는 이러한 패턴을 활용하여 최근에 사용된 데이터를 캐시에 저장함으로써 메모리 접근 시간을 줄인다.

공간적 지역성(Spatial Locality)은 한 번 접근한 데이터나 명령어의 근처에 있는 데이터나 명령어가 곧 접근될 가능성이 높다는 개념이다. 예를 들어, 배열의 요소를 순차적으로 접근하는 경우 공간적 지역성이 나타난다. 이는 프로그램이 메모리의 연속된 위치를 자주 접근하는 경향을 보이며, 캐시는 이러한 패턴을 활용하여 인접한 메모리 블록을 한 번에 캐시에 저장함으로써 메모리 접근 효율을 높인다.

시간적 지역성과 공간적 지역성을 활용한 캐시 메모리 설계는 메모리 접근 시간을 단축시키고, 시스템 전반의 성능을 향상시키는 데 중요한 역할을 한다. 시간적 지역성을 활용하기 위해 최근에 접근한 데이터를 캐시에 저장하여 재사용 가능성을 높이고, 공간적 지역성을 활용하여 인접한 데이터를 함께 캐시에 저장함으로써 연속적인 메모리 접근 효율을 극대화한다. 이러한 접근 방식은 메모리 시스템의 효과적인 관리와 최적화를 가능하게 한다.

2.3 Cache



캐시(Cache)는 프로세서 내부에 위치한 고속 메모리로, 메모리의 일부를 복사하여 CPU 가까이에 저장함으로써 자주 사용되는 데이터나 명령어에 대한 접근 속도를 높여 메모리 접근 시간을 단축하고 시스템 성능을 향상시키는 데 중요한 역할을 한다.

캐시 구조는 데이터 저장소와 태그 저장소로 구성되며, 각각 캐시 라인, 캐시 인덱스 및 캐시 태그를 포함한다. 캐시 라인은 캐시에 저장되는 데이터 블록이다. 각 캐시 라인은 메모리의 연속된 여러 바이트를 포함하며, 한 번에 로드되고 저장된다. 캐시 라인의 크기는 시스템에 따라 다를 수 있지만, 일반적으로 32바이트, 64바이트 또는 그 이상이다. 캐시 인덱스는 메모리 주소의 일부 비트를 사용하여 캐시에서 데이터가 저장된 위치를 지정한다. 캐시 인덱스를 통해 특정 데이터가 캐시의 어느 라인에 저장되어 있는지 빠르게 찾을 수 있다. 캐시 태그는 캐시 라인에 저장된 데이터가 메모리의 어느 주소에서 왔는지 식별하는 데 사용된다. 태그는 메모리 주소의 상위 비트로 구성되며, 캐시 인덱스와 함께 사용되어 메모리 주소를 완전히 식별할 수 있다. 캐시가 데이터 요청을 받을 때, 캐시 인덱스와 캐시 태그를 사용하여 요청된 데이터가 캐시에 있는지 확인한다.

2.3.1 Cache Hit / Miss

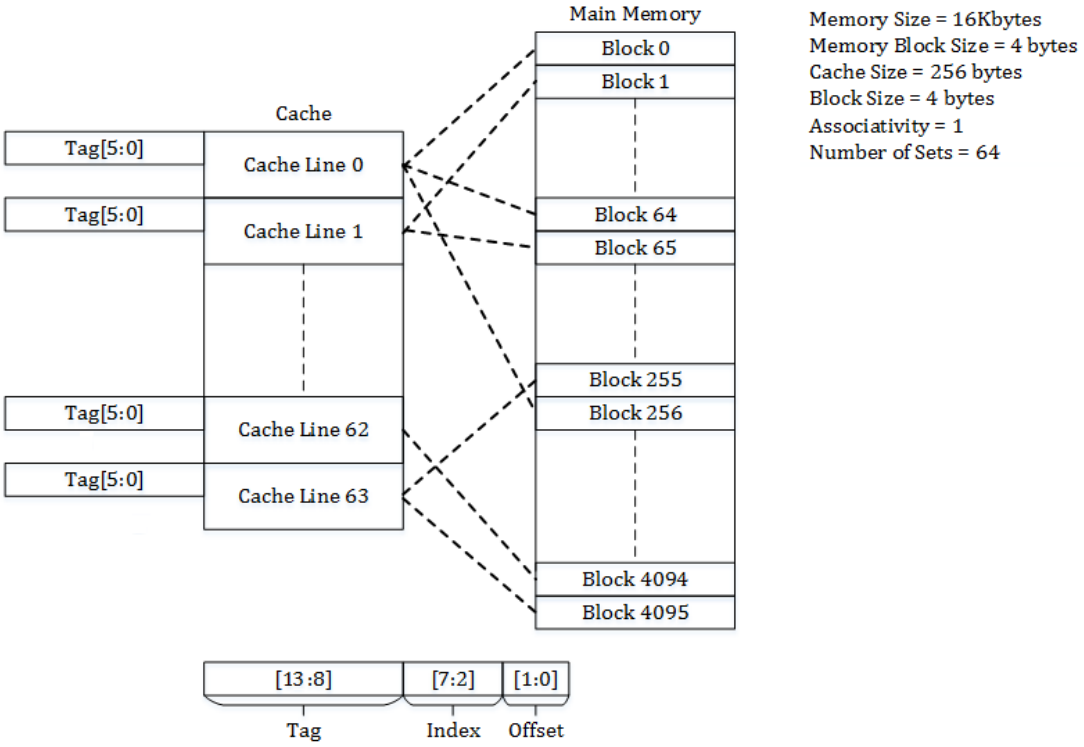
캐시 히트(Cache Hit)는 CPU가 요청한 데이터가 캐시 메모리에 존재하여 빠르게 접근할 수 있는 경우로, 이는 메모리 접근 시간을 크게 줄여 시스템 성능을 향상시키는 역할을 한다. 캐시 히트율(Cache Hit Rate)은 전체 메모리 접근 중 캐시 히트가 발생한 비율을 나타내며, 히트율이 높을수록 캐시의 성능이 우수하다는 의미이다. 반대로 캐시 미스(Cache Miss)는 요청한 데이터가 캐시에 존재하지 않아 메인 메모리에서 데이터를 불러와야 하는 경우로, 이는 메모리 접근 시간을 증가시키고 시스템 성능을 저하시킨다. 캐시 미스율(Cache Miss Rate)은 전체 메모리 접근 중 캐시 미스가 발생한 비율을 나타내며, 미스율이 높을수록 성능이 저하된다는 것을 의미한다.

캐시 미스는 콜드 미스(Cold Miss), 캐피시티 미스(Capacity Miss), 컨플릭트 미스(Conflict Miss) 세 가지 종류로 분류된다. 콜드 미스는 캐시가 비어있거나 처음으로 데이터를 불러올 때 발생하며, 컴펄서리 미스(Compulsory Miss)라고도 한다. 캐피시티 미스는 캐시의 용량이 부족하여 데이터를 저장할 공간이 충분하지 않을 때 발생한다. 컨플릭트 미스는 집합 연관 매핑이나 직접 매핑 방식에서 여러 데이터가 동일한 캐시 라인이나 세트에 매핑되어 발생하는 경우이다. 이러한 미스를 최소화하기 위해 다양한 최적화 기법을 적용할 수 있다.

캐시 성능을 최적화하기 위해서는 캐시 크기를 증가시켜 캐피시티 미스를 줄이고, 집합 연관도를 높여 컨플릭트 미스를 줄이는 방법이 있다. 또한, LRU(Least Recently Used), FIFO(First In, First Out), SCA(Second Chance Algorithm) 등의 효율적인 교체 정책을 사용

하여 캐시 미스를 최소화할 수 있다. 프리페칭 기법을 통해 데이터를 미리 캐시에 로드하여 캐시 미스를 줄이는 것도 하나의 방법이다. 캐시 미스가 발생하면 데이터 불러오기, 캐시 업데이트, 데이터 반환의 과정을 거쳐야 하므로 추가적인 시간 지연(캐시 미스 페널티)이 발생하여 성능 저하를 초래한다. 캐시 히트와 미스를 효과적으로 관리함으로써 전체 메모리 접근 시간을 줄이고 시스템의 효율성을 높일 수 있다.

2.4 Direct-Mapped Cache

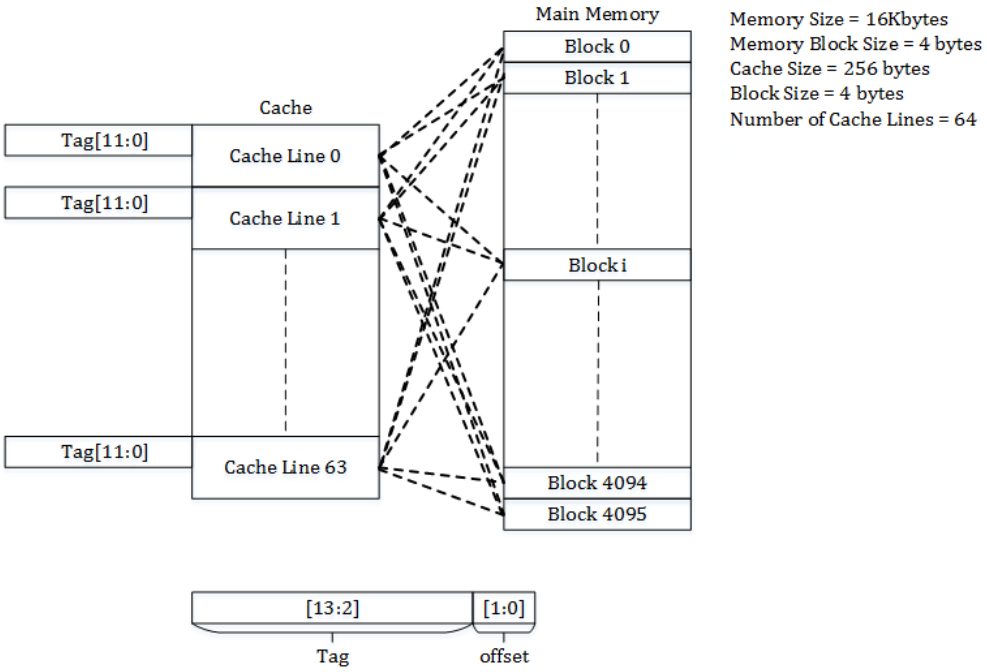


Direct-Mapped Cache는 메모리 블록이 캐시의 특정 위치에 고정적으로 매핑 되는 방식이다. 각 메모리 블록은 그림에서 볼 수 있듯이 오직 하나의 캐시 라인에만 저장될 수 있어, 구조가 단순하고 빠르며 하드웨어 구현이 용이하다. Direct-Mapped Cache의 주소는 태그(tag), 인덱스(index), 캐시 라인 오프셋(cache line offset)으로 구성된다. 인덱스는 캐시의 특정 위치를 가리키며, 해당 위치에 대한 태그를 확인하여 데이터가 존재하는지 여부를 판단한다.

Direct-Mapped Cache의 주요 장점은 다음과 같다. 각 메모리 블록이 고정된 위치에 매핑되므로 캐시 조회가 빠르고 효율적이다. 고정된 인덱스를 사용하여 단일 엔트리의 태그 매칭만 필요하므로 조회 속도가 빠르다. 또한, 구조가 단순하여 하드웨어 구현이 용이하고 비용이 적게 든다. 특정 주소의 데이터가 항상 일정한 캐시 라인에 저장되므로 예측 가능성이 높다.

그러나 Direct-Mapped Cache는 몇 가지 단점도 가지고 있다. 두 개 이상의 메모리 블록이 동일한 인덱스를 가질 경우 충돌이 발생하여 캐시 미스가 증가할 수 있다. 이를 충돌 미스(Conflict Miss)라고 한다. 또한, 특정 메모리 블록이 고정된 위치에만 저장될 수 있어 캐시 활용도가 낮아질 수 있다. 프로그램이 일정한 패턴으로 메모리에 접근할 경우 충돌이 빈번하게 발생하여 성능이 저하될 수 있다.

2.5 Fully Associative Cache



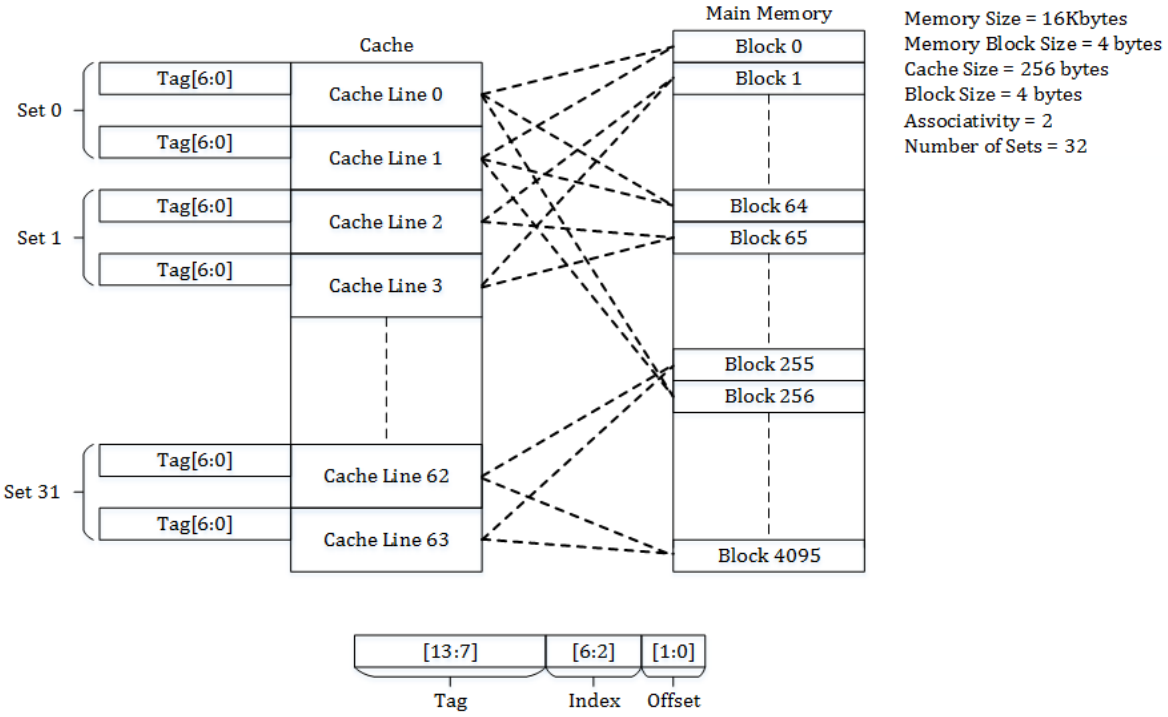
Fully Associative Cache는 메모리 블록이 캐시 내 어느 위치에든 저장될 수 있는 방식이다. 이는 캐시의 모든 라인을 검색하여 데이터의 위치를 찾는 방식으로, 특정 위치에 고정되지 않으므로 더 유연하게 데이터를 저장할 수 있다. Fully Associative Cache의 주소는 태그(tag)와 캐시 라인 오프셋(cache line offset)으로 구성된다. 이 방식에서는 인덱스가 필요하지 않으며, 태그와 오프셋만으로 데이터의 위치를 찾는다.

위 그림은 Fully Associative Cache의 동작 방식을 보여준다. 메모리 블록들이 캐시의 모든 라인에 저장될 수 있음을 나타내고 있다. 그림에서 각 메모리 블록은 캐시 내 임의의 라인에 매핑될 수 있으며, 이는 캐시 활용도를 극대화한다.

Fully Associative Cache의 주요 장점은 데이터가 캐시의 어느 위치에든 저장될 수 있어 캐시 활용도가 높다는 점이다. 메모리 블록이 특정 위치에 고정되지 않으므로 충돌이 거의 발생하지 않는다. 따라서, 캐시 미스율이 낮아질 수 있으며, 캐시의 전반적인 성능이 향상된다. 또한, 자주 사용되는 데이터가 캐시에서 효율적으로 관리될 수 있어 프로그램의 성능을 높일 수 있다.

그러나 Fully Associative Cache는 몇 가지 단점도 가지고 있다. 모든 캐시 라인을 검색해야 하므로 하드웨어 구현이 복잡하고 비용이 많이 든다. 각 캐시 접근마다 모든 라인의 태그를 비교해야 하므로 조회 속도가 느릴 수 있다. 또한, LRU(Least Recently Used)와 같은 교체 정책을 구현하는 데 오버헤드가 발생할 수 있다.

2.6 Set-associative cache



Set-Associative Cache는 Direct-Mapped Cache와 Fully Associative Cache의 특성을 혼합한 형태로, 메모리 블록을 여러 개의 세트(set)로 나누어 저장하는 방식이다. 각 세트는 여러 개의 캐시 라인(cache line)으로 구성되며, 메모리 블록은 특정 세트 내의 어느 라인에든 저장될 수 있다. 이 방식은 캐시의 유연성과 성능을 균형 있게 유지하면서 충돌 미스(Conflict Miss)를 줄이기 위해 설계되었다.

위 그림은 Set-Associative Cache의 구조를 보여준다. 그림에서 각 메모리 블록은 특정 세트에 매핑되며, 세트 내의 여러 캐시 라인 중 하나에 저장될 수 있음을 나타낸다.

Set-Associative Cache의 주소 구조는 태그(tag), 세트 인덱스(set index), 캐시 라인 오프셋(cache line offset)으로 구성된다. 세트 인덱스는 특정 세트를 지정하고, 태그는 해당 세트 내에서 데이터의 유효성을 확인하는 데 사용된다.

(Address = tag + set index + cacheline offset)

Set-Associative Cache의 주요 장점은 다음과 같다. 데이터가 특정 세트 내의 여러 라인에 저장될 수 있어 캐시 활용도가 높다. 메모리 블록이 특정 세트에만 고정되므로 충돌

미스가 Direct-Mapped Cache에 비해 적다. 또한, 전체 캐시를 검색할 필요가 없으므로 Fully Associative Cache에 비해 조회 속도가 빠르다. 이 방식은 캐시의 유연성과 성능을 균형 있게 유지할 수 있다. 예를 들어, 그림에서 세트 0은 두 개의 캐시 라인으로 구성되어 있으며, 블록 0과 블록 1이 이 세트 내의 어느 라인에던 저장될 수 있다. 이는 데이터 접근 시 충돌이 발생할 가능성을 줄여준다.

그러나 Set-Associative Cache는 몇 가지 단점도 가지고 있다. 구현이 Direct-Mapped Cache보다 복잡하여 비용이 더 많이 들 수 있다. 또한, 특정 세트 내에서만 데이터를 저장할 수 있으므로, 세트가 가득 차면 데이터를 교체해야 하는 경우가 발생할 수 있다. 이는 교체 정책의 복잡성을 증가시킨다.

Set-Associative Cache에서 하나의 세트가 2개의 블록으로 이루어져있다면 2-way set-associative cache라고 하고 4개의 블록으로 이루어져있다면 4-way set-associative cache라고 한다. 즉, n-way set-associative cache는 블록들을 n열 종대로 배열한 것이다. 이번 과제에서는 4-way set-associative cache 방법을 이용하였다.

2.7 Replacement Policy

캐시의 교체 정책(Replacement Policy)은 캐시가 가득 찼을 때, 새로운 데이터를 저장하기 위해 어떤 기존 데이터를 제거할지를 결정하는 방법이다. 이는 캐시 성능에 중요한 영향을 미치며, 다양한 교체 정책이 존재한다. 대표적인 교체 정책에는 랜덤(RANDOM), FIFO(First In, First Out), LRU(Least Recently Used), SCA(Second Chance Algorithm)가 있다.

2.7.1 RANDOM (랜덤)

랜덤 교체 정책은 캐시가 가득 찼을 때, 임의의 캐시 라인을 선택하여 교체하는 방식이다. 이 방법은 구현이 간단하며, 특정 패턴에 의한 편향이 없다는 장점이 있다. 그러나 다른 정책에 비해 효율성이 떨어질 수 있으며, 히트율이 낮을 가능성이 있다.

2.7.2 FIFO (First In, First Out)

FIFO 교체 정책은 캐시에 먼저 들어온 데이터를 먼저 내보내는 방식이다. 즉, 가장 오래된 데이터를 교체하는 것이다. 이 방법은 구현이 간단하고, 데이터의 연령에 따라 교체를 결정할 수 있어 일정한 히트율을 유지할 수 있다. 그러나 자주 사용되는 데이터가 오래되었다는 이유로 교체될 수 있어 성능이 저하될 위험이 있다.

2.7.3 LRU (Least Recently Used)

LRU 교체 정책은 가장 오랫동안 사용되지 않은 데이터를 교체하는 방식이다. 이는 최근

에 사용된 데이터가 다시 사용될 가능성이 높다는 원칙에 기반하고 있다. LRU는 높은 히트율을 제공할 수 있지만, 각 캐시 접근마다 사용 정보를 업데이트해야 하므로 구현이 복잡하고, 오버헤드가 발생할 수 있다. 이번 과제에서는 해당 알고리즘을 선택하였다.

2.7.4 SCA (Second Chance Algorithm)

SCA 교체 정책은 FIFO와 유사하지만, 두 번째 기회를 제공하는 방식이다. 각 캐시 라인은 두 번째 기회를 나타내는 비트를 가지고 있다. 교체 시, 두 번째 기회 비트가 설정된 캐시 라인은 비트를 초기화하고 교체하지 않으며, 비트가 설정되지 않은 라인을 교체한다. 이 방법은 자주 사용되는 데이터가 교체되는 것을 방지하여 성능을 향상시킬 수 있다.

2.8 Write Policy

캐시의 쓰기 정책(Write Policy)은 캐시와 메인 메모리 간의 데이터 일관성을 유지하면서 데이터를 어떻게 쓸 것인지를 결정하는 방법이다. 주요 쓰기 정책으로는 Write-Through와 Write-Back이 있다. 각 정책은 성능과 일관성 측면에서 다르게 작동하며, 다양한 시스템 요구 사항에 따라 선택된다.

2.8.1 Write-Through

Write-Through 정책은 데이터를 캐시에 쓰는 동시에 메인 메모리에도 즉시 쓰는 방식이다. 이 정책은 메모리 일관성을 쉽게 유지할 수 있으며, 캐시와 메인 메모리 간의 동기화를 보장한다. Write-Through의 주요 장점은 데이터 일관성을 유지하기 쉽고, 메인 메모리와 캐시의 데이터가 항상 동일하다는 점이다. 이로 인해 시스템 충돌이나 재시작 시 데이터 손실의 위험이 적다. 그러나 모든 쓰기 작업이 캐시와 메인 메모리에 동시에 이루어지므로 메모리 쓰기 횟수가 증가하고, 이로 인해 쓰기 성능이 저하될 수 있다는 단점이 있다.

2.8.2 Write-Back

Write-Back 정책은 데이터를 캐시에만 쓰고, 캐시 라인이 교체될 때 메인 메모리에 데이터를 쓰는 방식이다. 이 정책은 캐시의 쓰기 작업을 줄여 성능을 향상시킬 수 있다. Write-Back의 주요 장점은 데이터가 캐시에만 쓰이므로 쓰기 작업이 빠르게 수행되고, 메인 메모리에 쓰기 작업이 줄어들어 성능이 향상된다는 점이다. 그러나 캐시와 메인 메모리 간의 데이터 일관성을 유지하기 어렵고, 캐시 라인이 교체될 때 데이터가 메모리에 쓰여야 하므로 복잡성이 증가한다는 단점이 있다. 또한, 시스템 충돌이나 재시작 시 데이터 손실의 위험이 존재한다.

3. Program Description

이번 과제는 싱글 사이클에서 구현하였다. 기존 싱글 사이클 과제에서 캐시를 위해 추가한 부부면 작성하도록 하겠다.

```
#define MEMORY_SIZE 0x4000000 // 64MB memory
#define CACHE_SIZE 256 // 256 bytes
#define CACHE_LINE_SIZE 64 // 64 bytes per cache line
#define CACHE_WAYS 4 // 4-way set associative cache
#define SET_COUNT (CACHE_SIZE / (CACHE_LINE_SIZE * CACHE_WAYS))
#define MEMORY_LATENCY 1000 // Memory access latency in cycles
```

먼저, 캐시 메모리의 크기와 구조에 대한 정의이다. 캐시 메모리의 전체 크기를 256 바이트로 정의하였고, 하나의 캐시 라인 크기를 64 바이트로 설정하였다. 이번 과제는 4-way set-associative cache 방식을 사용하였기 때문에 각 세트가 4개의 캐시 라인을 가지도록 설정하였다. SET_COUNT로 캐시 세트의 개수를 계산하여 정의하였다. 각 세트는 CACHE_LINE_SIZE * CACHE_WAYS 바이트를 차지하므로, 전체 캐시 크기를 나누어 세트의 개수를 계산한다. 이 경우 (256 / (64 * 4)) = 1로, 캐시 세트의 개수는 1개이다. 마지막으로, MEMORY_LATENCY는 메모리 접근 지연 시간을 정의하고 있다. 이 값은 1000 사이클로 설정되어 있으며, 메모리 접근 시 소요되는 시간을 나타낸다.

```
typedef enum { RANDOM, FIFO, LRU, SCA } ReplacementPolicy;
typedef enum { WRITE_BACK, WRITE_THROUGH } WritePolicy;
```

```
ReplacementPolicy replacement_policy = LRU;
WritePolicy write_policy = WRITE_BACK;
```

캐시 메모리에서 사용할 교체 정책과 쓰기 정책에 대하여 enum으로 선언한 부분이다. 4가지의 교체 정책과 2가지의 쓰기 정책을 정의하여 유연하게 구현하려고 하였으나 이번 과제에서는 LRU 교체 정책과 WRITE_BACK 쓰기 정책만 이용하여 구현하였다.

```c
typedef struct {
    uint8_t data[CACHE_LINE_SIZE];
    uint32_t tag;
    int valid;
    int dirty;
    int lru_counter;
    int second_chance;
} CacheLine;
```

```c
typedef struct {
    CacheLine lines[CACHE_WAYS];
    int fifo_index;
} CacheSet;
```

캐시 메모리의 구조를 구조체로 정의하였다. 각각의 CacheLine과 CacheSet 구조체는 캐시 메모리의 라인과 세트를 나타내며, 이를 통해 캐시 데이터와 관련된 다양한 속성을 관리할 수 있다.

먼저, CacheLine 구조체는 캐시의 각 라인을 정의하고 있다. 이 구조체는 캐시 라인에 저장되는 데이터와 다양한 속성을 포함하고 있다. data는 캐시 라인에 저장되는 데이터 배열로, 배열의 크기는 CACHE_LINE_SIZE로 정의되어 있다. tag는 캐시 라인의 태그 값으로 메모리 블록의 주소와 연관된다. valid는 캐시 라인의 유효 비트로 데이터가 유효한지 여부를 나타내며, dirty는 더티 비트로 데이터가 수정되었는지 여부를 나타낸다. lru_counter는 LRU (Least Recently Used) 카운터로, 최근 사용되지 않은 정도를 나타내고, second_chance는 Second Chance 비트로 두 번째 기회를 위한 비트이다.

다음으로, CacheSet 구조체는 여러 개의 캐시 라인으로 구성된 캐시 세트를 정의하고 있다. lines는 캐시 세트 내의 캐시 라인 배열로, 배열의 크기는 CACHE_WAYS로 정의되어 있다. 이는 n-way set associative 캐시를 구현하는 데 사용된다. fifo_index는 FIFO (First In, First Out) 교체 정책을 사용할 때 필요한 인덱스이다.

```c
// Initialize cache
void cacheInitialize() {
    for (int i = 0; i < SET_COUNT; i++) {
        for (int j = 0; j < CACHE_WAYS; j++) {
            cache[i].lines[j].valid = 0;
            cache[i].lines[j].dirty = 0;
            cache[i].lines[j].lru_counter = 0;
            cache[i].lines[j].second_chance = 0;
            memset(cache[i].lines[j].data, 0, CACHE_LINE_SIZE);
        }
        cache[i].fifo_index = 0;
    }
}
```

이 함수는 캐시 메모리를 초기화하는 함수이다. 각 캐시 라인의 valid 비트를 0으로 설정하여, 해당 캐시 라인이 비어 있음을 나타낸다. dirty 비트는 0으로 설정하여, 캐시 라인이 수정되지 않았음을 나타낸다. lru_counter는 0으로 설정하여, LRU (Least Recently Used) 교체 정책에서 최근에 사용되지 않은 상태로 초기화된다. second_chance 비트는 0으로

설정하여, Second Chance 교체 정책에서 두 번째 기회를 받지 않은 상태로 초기화된다. 또한, memset 함수를 사용하여 각 캐시 라인의 data 배열을 0으로 설정한다. 마지막으로, 각 캐시 세트의 fifo_index를 0으로 설정하여, FIFO (First In, First Out) 교체 정책에서 첫 번째 위치로 초기화한다.

```
// Select cache line based on replacement policy
CacheLine* selectCacheLine(CacheSet* set) {
    switch (replacement_policy) {
        case RANDOM:
            return &set->lines[rand() % CACHE_WAYS];
        case FIFO:
            return &set->lines[set->fifo_index++ % CACHE_WAYS];
        case LRU: {
            CacheLine* lru_line = &set->lines[0];
            for (int i = 1; i < CACHE_WAYS; ++i) {
                if (set->lines[i].lru_counter < lru_line->lru_counter) {
                    lru_line = &set->lines[i];
                }
            }
            return lru_line;
        }
        case SCA: {
            CacheLine* sca_line = NULL;
            for (int i = 0; i < CACHE_WAYS; ++i) {
                if (set->lines[i].second_chance == 0) {
                    sca_line = &set->lines[i];
                    break;
                }
                set->lines[i].second_chance = 0;
            }
            if (sca_line == NULL) {
                sca_line = &set->lines[0];
            }
            return sca_line;
        }
        default:
            return &set->lines[0]; // 기본 값
    }
}
```

캐시의 교체 정책에 따라 캐시 라인을 선택하는 함수이다. Replacement_policy에 따라 교체 정책을 선택하지만 이번 과제에서는 LRU로 고정되어 있기 때문에 사실상 LRU를 제외한 나머지 3개의 정책 케이스들은 아무런 기능을 하지 않는다.

```
// Cache access function
int cacheAccess(uint32_t address, uint8_t* data, int write) {
    uint32_t tag = address / CACHE_SIZE;
    uint32_t set_index = (address / CACHE_LINE_SIZE) % SET_COUNT;
    uint32_t offset = address % CACHE_LINE_SIZE;
    CacheSet* set = &cache[set_index];

    for (int i = 0; i < CACHE_WAYS; i++) {
        CacheLine* line = &set->lines[i];
        if (line->valid && line->tag == tag) { // Cache hit
            if (write) {
                memcpy(line->data + offset, data, 4); // Writing 4 bytes
                if (write_policy == WRITE_BACK) {
                    line->dirty = 1;
                } else {
                    uint32_t mem_address = (line->tag * SET_COUNT + set_index) * CACHE_LINE_SIZE + offset;
                    memWrite(mem_address, *((uint32_t*)data));
                }
            } else {
                memcpy(data, line->data + offset, 4); // Reading 4 bytes
            }
            line->lru_counter = instruction_count;
            line->second_chance = 1;
            cache_hit_count++;
            total_cycles += 1; // Cache hit latency
            return 1; // Cache hit
        }
    }
```

해당 코드는 캐시 접근 함수로 주어진 메모리 주소에서 데이터를 읽거나 쓰는 작업을 수행하고, 캐시 히트 또는 캐시 미스 여부를 판단한다. 위 그림에서는 캐시 히트를 판단하고 있다. 먼저, 함수는 입력된 메모리 주소로 태그, 세트 인덱스, 오프셋을 계산한다. 이를 통해, 함수는 세트 인덱스에 해당하는 캐시 세트를 찾는다. 이후, 세트의 모든 캐시 라인을 순회하며 valid 비트가 1이고 태그가 일치하는지 확인한다. 이 조건이 만족하면 캐시 히트가 발생한다. 캐시 히트가 발생하면 함수는 쓰기 연산과 읽기 연산을 구분하여 처리한다. 쓰기 연산의 경우, memcpy를 사용하여 캐시 라인의 data 배열에 4바이트 데이터를 쓴다. 만약 쓰기 정책이 WRITE_BACK으로 설정되어 있으면, dirty 비트를 1로 설정한다. 반면, WRITE_THROUGH인 경우, 메모리 주소를 계산하여 memWrite 함수를 호출하여 메모리에 데이터를 쓴다. 정책이 이미 WRITE_BACK으로 정해져 있기 때문에 WRITE_THROUGH의 경우가 작동하지 않는다. 읽기 연산의 경우, memcpy를 사용하여 캐시 라인의 data 배열에서 4바이트 데이터를 읽어온다. 캐시 히트 시, 해당 캐시 라인의 lru_counter를 현재 명령어 카운터 값으로 업데이트하고, second_chance 비트를 1로 설정한다. 또한, 캐시 히트 카운트를 증가시키고, 사이클 수에 1을 더한다. 마지막으로, 캐시 히트가 발생했음을 나타내기 위해 1을 반환한다.

```
// Cache miss
CacheLine* line = selectCacheLine(set);
if (line->dirty) {
    uint32_t mem_address = (line->tag * SET_COUNT + set_index) * CACHE_LINE_SIZE;
    for (int i = 0; i < CACHE_LINE_SIZE; i += 4) {
        uint32_t value = *((uint32_t*)(line->data + i));
        memWrite(mem_address + i, value);
    }
}

line->valid = 1;
line->tag = tag;
line->lru_counter = instruction_count;
line->second_chance = 1;
line->dirty = write_policy == WRITE_BACK ? write : 0;

uint32_t mem_address = (tag * SET_COUNT + set_index) * CACHE_LINE_SIZE;
for (int i = 0; i < CACHE_LINE_SIZE; i += 4) {
    uint32_t value = memAccess(mem_address + i, 0, 0);
    *((uint32_t*)(line->data + i)) = value;
}

if (write) {
    memcpy(line->data + offset, data, 4);
} else {
    memcpy(data, line->data + offset, 4);
}

printf("Cache miss: address=%08X, set_index=%d, tag=%d\n", address, set_index, tag);

cache_miss_count++;
total_cycles += MEMORY_LATENCY; // Cache miss latency
return 0; // Cache miss
```

이 코드는 위의 캐시 히트 코드에 이어져 있는 캐시 미스 판단 코드이다. 캐시 히트와 마찬가지로 cacheAccess 함수에 포함되어 있다. 먼저, selectCacheLine 함수로 교체할 캐시 라인을 선택한다. 선택된 캐시 라인의 dirty 비트를 검사하여, 더티 비트가 설정되어 있는 경우 해당 캐시 라인의 데이터를 메모리에 기록한다. 그 다음, 메모리 주소를 계산하고, memWrite 함수를 사용하여 캐시 라인의 데이터를 메모리에 기록한다. 이후, 메모리 주소룰 계산하고 데이터를 메모리에서 읽어온다. 읽어온 데이터를 캐시 라인의 data 배열에 저장하고 쓰기 연산과 읽기 연산인 경우를 각각 나누어 계산한다. 마지막으로, 캐시 미스 카운트를 증가시키고 사이클 수에 메모리 접근 지연 시간을 더한다. 캐시 미스가 발생했음을 나타내기 위해 0을 반환하고 함수가 마무리 된다.

```
float calculateAMAT() {
    float hit_time = 1.0f; // Cache hit time in cycles
    float miss_penalty = MEMORY_LATENCY; // Cache miss penalty in cycles
    float miss_rate = (float)cache_miss_count / (cache_hit_count + cache_miss_count);
    return hit_time + miss_rate * miss_penalty;
}
```

캐시의 평균 메모리 접근 시간인 AMAT를 계산하는 코드이다. AMAT는 캐시의 성능을 평가하는 중요한 지표로, 캐시 히트 시간과 캐시 미스 패널티를 고려하여 계산된다.

```
uint32_t fetch() {
    uint8_t data[4];
    cacheAccess(pc, data, 0);
    instruction = (data[3] << 24) | (data[2] << 16) | (data[1] << 8) | data[0];
    printf("Fetched instruction at PC: %08X, Instruction: %08X\n", pc, instruction);
    total_cycles++;
    return instruction;
}
```

마지막으로 변경된 fetch 함수이다. 기존 fetch 함수에서 캐시가 추가 되었기 때문에 캐시에 접근해서 현재 pc에 해당 되는 명령어를 가져온다. 가져온 4바이트의 명령어를 시프트하여 32비트의 명령어로 조합하고 메인함수를 실행한다.

4. Result

```
********** Result **********
Total number of cycles of execution: 2016
Number of memory (load/store) operations: 2
Number of register operations: 6
Number of branches (total/taken): 0/0
Cache hit/miss: 8/2
Average Memory Access Time (AMAT): 201.00 cycles
**********
```

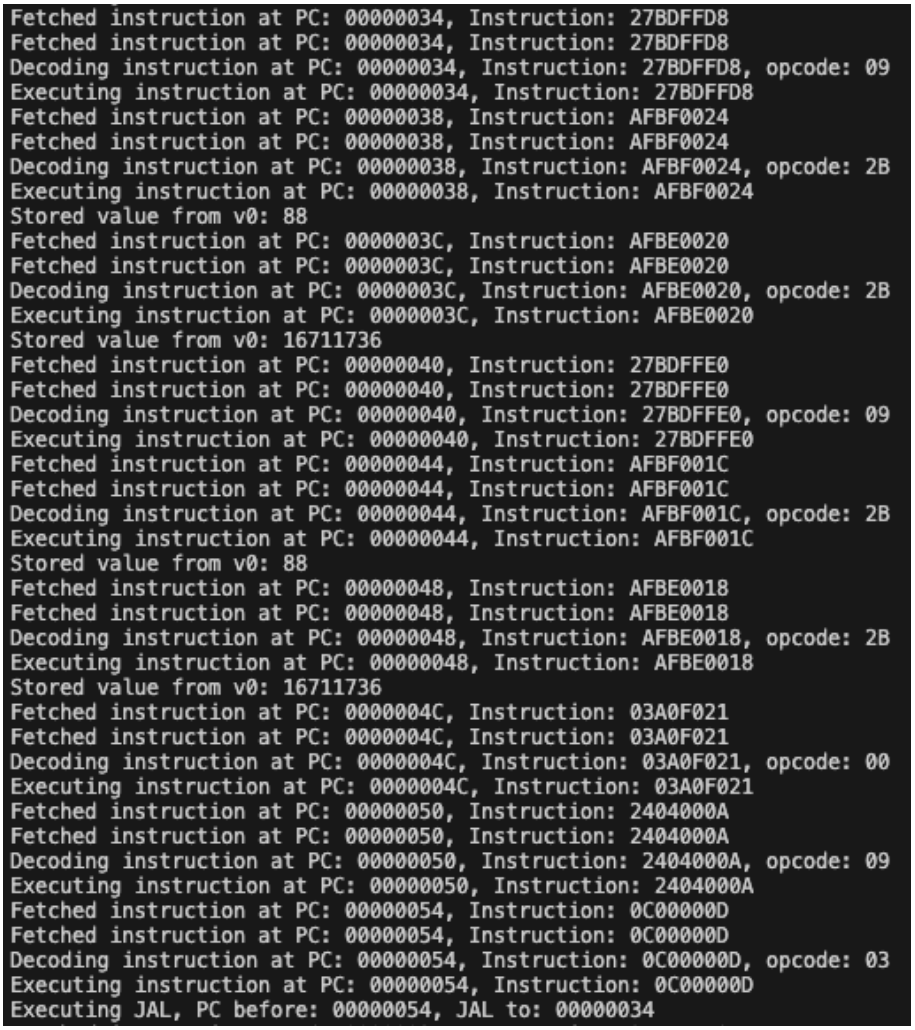
simple.bin

```
********** Result **********
Total number of cycles of execution: 2022
Number of memory (load/store) operations: 4
Number of register operations: 6
Number of branches (total/taken): 0/0
Cache hit/miss: 12/2
Average Memory Access Time (AMAT): 143.86 cycles
**********
```

simple2.bin

```
Fetched instruction at PC: 00000044, Instruction: AFBE0014
Fetched instruction at PC: 00000044, Instruction: AFBE0014
Decoding instruction at PC: 00000044, Instruction: AFBE0014, opcode: 2B
Executing instruction at PC: 00000044, Instruction: AFBE0014
Stored value from v0: 16777192
Fetched instruction at PC: 00000048, Instruction: 03A0F021
Fetched instruction at PC: 00000048, Instruction: 03A0F021
Decoding instruction at PC: 00000048, Instruction: 03A0F021, opcode: 00
Executing instruction at PC: 00000048, Instruction: 03A0F021
Fetched instruction at PC: 0000004C, Instruction: AFC00008
Fetched instruction at PC: 0000004C, Instruction: AFC00008
Decoding instruction at PC: 0000004C, Instruction: AFC00008, opcode: 2B
Executing instruction at PC: 0000004C, Instruction: AFC00008
Stored value from v0: 0
Fetched instruction at PC: 00000050, Instruction: AFC0000C
Fetched instruction at PC: 00000050, Instruction: AFC0000C
Decoding instruction at PC: 00000050, Instruction: AFC0000C, opcode: 2B
Executing instruction at PC: 00000050, Instruction: AFC0000C
Stored value from v0: 0
Fetched instruction at PC: 00000054, Instruction: AFC00008
Fetched instruction at PC: 00000054, Instruction: AFC00008
Decoding instruction at PC: 00000054, Instruction: AFC00008, opcode: 2B
Executing instruction at PC: 00000054, Instruction: AFC00008
Stored value from v0: 0
Fetched instruction at PC: 00000058, Instruction: 08000011
Fetched instruction at PC: 00000058, Instruction: 08000011
Decoding instruction at PC: 00000058, Instruction: 08000011, opcode: 02
Executing instruction at PC: 00000058, Instruction: 08000011
Executing J, PC before: 00000058, J to: 00000044
```

simple3.bin



simple4.bin

2 번째 프로젝트로 진행했던 싱글 사이클 과제를 조금 보수하고 해당 과제에 캐시를 붙여넣어 이번 과제를 진행하였다. 싱글 사이클 과제를 진행할 때 심플 3 파일을 실행시키면 무한루프가 돌아 메모리 크기를 키우는 방법으로 해결하였었는데 이번 프로젝트에서는 심플 3 의 무한 루프를 해결하지 못하였다. 메모리 크기를 키우고 캐시 사이즈와 캐시 라인 사이즈를 키우는 등 크기를 조정하는 것만으로는 해결되진 않았다. 여러 디버깅 코드들을 넣어가며 어느 부분이 문제인지 찾아보려고 하였지만 무한루프가 찍히는 상황에서 pc 값에 맞는 명령어들이 실행되지 않는 것을 보고 어떤 부분이 문제인지 찾기 어려웠다. 심플 3 의 경우 점프와 SW 가 반복되고 있었고 심플 4 의 경우 ADDIU 와 SW 가 반복되고 있었다. pc 업데이트가 잘못되고 있는 부분을 찾아 수정하였으나 결국 무한루프는 해결하지 못하였다.

심플 1 과 심플 2 의 결과를 통해 캐시 히트 수가 증가하여 캐시 효율성이 높아졌고, AMAT 가 감소한 사실을 확인 할 수 있었다. 총 실행 사이클 수는 약간 증가하였지만, 메모리 접근 시간이 개선되었음을 알 수 있다.

마지막으로, 이번 학기 컴퓨터구조론 수업을 통해 평소 관심을 갖기 어려운 부분에 대해 깊게 파고들 수 있었던 것 같아 좋았다. 컴퓨터 언어로 소프트웨어를 개발하는 것이 아닌 하드웨어에 접근하여 컴퓨터가 어떻게 사용자의 요구를 처리하는지 눈으로 확인 할 수 있었다. 생소한 부분이 많아 어렵기도 했지만 이론과 개발 측면에서 많은 것을 얻을 수 있었기 때문에 매우 유익한 수업이었다.