<프로젝트4 보고서>

컴퓨터구조와 모바일프로세서

싱글 사이클 MIPS with Cache 리포트

학　　과 : 모바일시스템공학과
학　　번 : 32164809
이　　름 : 탁준석
담당교수 : 유시환
제출일자 : 21.06.20
남은 프리데이 : 0일

목 차

1. 개요

가. 실험 목적

우리는 저번 프로젝트3까지 프로세서가 명령어를 처리함에 있어서 이상적인 메모리를 사용한다고 가정하였다. 그러나 현실적인 메모리는 용량이 제한되어 있고 레지스터에 비해 상대적으로 느린 접근 시간을 가지고 있다. 그렇다면 메모리의 접근 시간을 줄여 프로세서의 성능을 향상시키려면 어떻게 해야 할까?

이번 프로젝트4에서는 캐시가 도입된 싱글 사이클 MIPS 프로그램을 설계한다. 메모리가 가진 한계를 이해하고 메모리 성능을 개선하기 위해 캐시가 도입된 배경을 살펴본다. 메모리의 상충 관계 통해 기본적인 캐시의 역할을 이해하고 캐시의 기본 구조를 설계해본다. 캐시의 성능을 향상시키기 위한 세 가지 캐시 사상 방식을 학습하고 캐시 쓰기 정책을 분석하여 즉시 쓰기(write-through)와 나중 쓰기(write-back) 방식을 이해한다. 또한 캐시 실패가 발생하였을 때 LRU와 SCR 교체 정책을 고려하여 효율적인 캐시 블록 교체 방법을 생각해본다. 최종적으로 캐시 사상 방식, 쓰기 정책, 그리고 교체 정책을 고려하여 효율적인 캐시 구조를 설계함으로써 프로세서의 성능을 향상시킨다.

나. 실험 목표

이번 프로젝트4 Single Cycle with Cache의 실험 목표는 다음과 같다.

- 첫 번째, 캐시가 도입된 싱글 사이클 MIPS 프로그램을 개발한다.
- 두 번째, 클럭 사이클마다 변화하는 데이터를 출력한다.
- 세 번째, 마지막 클럭 사이클에 최종 결과, 클럭 사이클 수, 메모리 접근 횟수, 분기한 횟수, 점프한 횟수, 최초 시작 실패(Cold Miss), 대립 실패(Conflict Miss), 캐시 적중률, 캐시 실패율, 그리고 평균 메모리 접근 시간(AMAT)을 출력한다.
- 네 번째, 캐시 사상 방식, 캐시 크기, 그리고 캐시 라인 크기를 능동적으로 바꾸어 캐시의 성능을 서로 비교해보고 평균 메모리 접근 시간을 줄이는 방법을 도출한다.

캐시 사상 방식	종류	캐시 / 캐시 라인 크기
직접 사상 (Direct mapping)	① Direct mapping cache	4KB, 8KB, 32KB / 64B
집합 연관 사상 (Set associative mapping)	① 2-way Set associative cache ② 4-way Set associative cache	4KB, 8KB, 32KB / 64B

<표 1.1> 프로젝트4 Single Cycle MIPS with Cache에서 구현할 캐시 크기와 그 사상 방식.

2. 이론

가. 캐시의 기본(The Basics of Cache)

컴퓨터 프로그래머들은 용량 제한이 없는 빠른 메모리를 원한다. 즉, 거의 무한한 용량을 가지고 있고 접근 시간이 0에 수렴하며 비용이 발생하지 않는 메모리를 사용하고 싶어 한다. 그러나 실제 메모리는 그렇지 않다. 게다가 폰 노이만 컴퓨터 구조는 모든 명령어와 데이터를 메모리에 저장하기 때문에 메모리 접근 횟수가 매우 많아 심각한 병목 현상을 일으킨다. 쉽게 말해, 프로세서가 메모리에 데이터를 요청할 때마다 명령어 처리를 멈추고 메모리를 계속 기다려야 하는데 이것은 곧 프로세스의 성능이 저하되는 상황으로 이어진다.

따라서 메모리에 보다 더 빨리 접근하기 위해 '캐시'라는 개념이 도입되었다. 캐시는 메모리와 프로세서 사이에 데이터를 미리 저장해놓겠다는 생각에서 출발한다. 프로세서가 자주 사용하는 데이터 묶음을 프로세서와 아주 가까운 곳, '캐시'에 임시 저장하면 프로세서가 요구하는 데이터를 더욱 빠르게 제공할 수 있는데 이것은 메모리 접근 시간(Latency)을 획기적으로 줄여 프로세서의 성능을 개선한다. 그렇다면 캐시란 무엇이고 어떤 방법으로 설계해야할까? 캐시와 메모리 계층구조를 설명하기에 앞서, 우선 메모리 접근의 규칙성과 관련된 지역성의 원칙을 먼저 알아보도록 하자.

1) 지역성의 원칙(Principle of Locality)

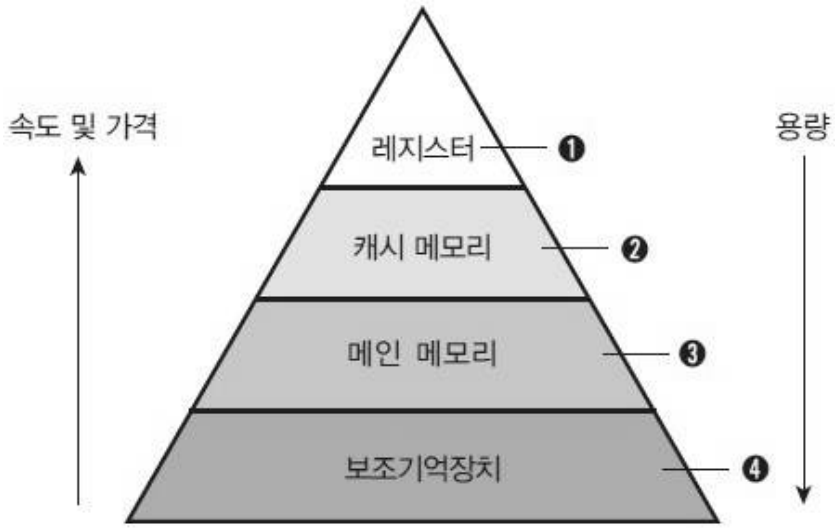
프로세서가 데이터를 읽거나 쓰기 위해 메모리에 접근하는 상황을 생각해보자. 프로세서는 넓은 메모리 사이에서 과연 어디에 위치한 데이터를 참조할까? 흔히 메모리 전역을 골고루 참조한다고 오해하기 쉬우나, 프로세서는 어느 순간에 집중적으로 데이터를 참조하는 규칙성을 보이는데 이것을 지역성의 원칙이라고 한다. 지역성의 원칙은 시간적 지역성과 공간적 지역성, 두 종류가 있다. 시간적 지역성은 어떤 데이터가 참조되면 곧바로 다시 참조될 가능성이 높다는 원칙이고, 공간적 지역성은 어떤 데이터가 참조되면 곧바로 그 주변의 데이터가 참조될 가능성이 높다는 원칙이다. 프로그램의 순환문과 순차 진행을 생각하면 지역성의 원칙을 이해하기 쉽다. 거의 대부분의 프로그램은 순환문 구조를 갖고 있고 순환문이 실행되는 동안 같은 명령어와 데이터에 반복적으로 접근한다. 때문에 시간적으로 큰 관련이 생긴다. 또한 프로그램의 명령어는 대개 순차적으로 처리되어 데이터에 접근하기 때문에 공간적으로도 관련이 생긴다.

지역성의 원칙	설명
시간적 지역성 (Temporal Locality)	프로그램은 최근에 접근했던 데이터들을 다시 사용하려는 경향이 강하다.
공간적 지역성 (Spatial Locality)	프로그램은 최근에 접근했던 데이터에 인접해있는 데이터에 접근하려는 경향이 강하다.

<표 2.1> 지역성의 원칙 중 시간적 지역성과 공간적 지역성.

2) 메모리 계층구조(Memory Hierarchy)

컴퓨터 메모리를 여러 계층으로 구분하여 구현함으로써 지역성의 원칙을 매우 적절하게 이용할 수 있다. 메모리 계층구조는 메모리의 상충 관계에 따라서 메모리 시스템을 단계별로 구현한 구조를 말한다. <그림 2.1>은 메모리의 상충 관계를 잘 나타내는데 프로세서와 가까운 상위 계층일수록 메모리의 용량이 작고 비싸지만 속도가 빠르다. 이와 반대로 프로세서와 먼 하위 계층일수록 메모리의 용량이 크고 싸지만 속도가 느리다. 메모리 계층구조는 속도가 빠른 메모리는 프로세서와 가깝게 두고, 속도가 느린 메모리는 프로세서와 멀게 배치한다. 따라서 상위 계층에서 데이터 접근에 성공하면 명령어를 매우 빠르게 처리할 수 있고, 상위 계층에서 데이터 접근에 실패하면 느린 하위 계층으로 내려가 데이터에 접근하여 명령어를 수행한다. 적중률이 충분히 크다면 메모리 계층의 접근 속도는 최상위 계층과 비슷하고 그 크기는 최하위 계층 메모리와 같아질 것이다.



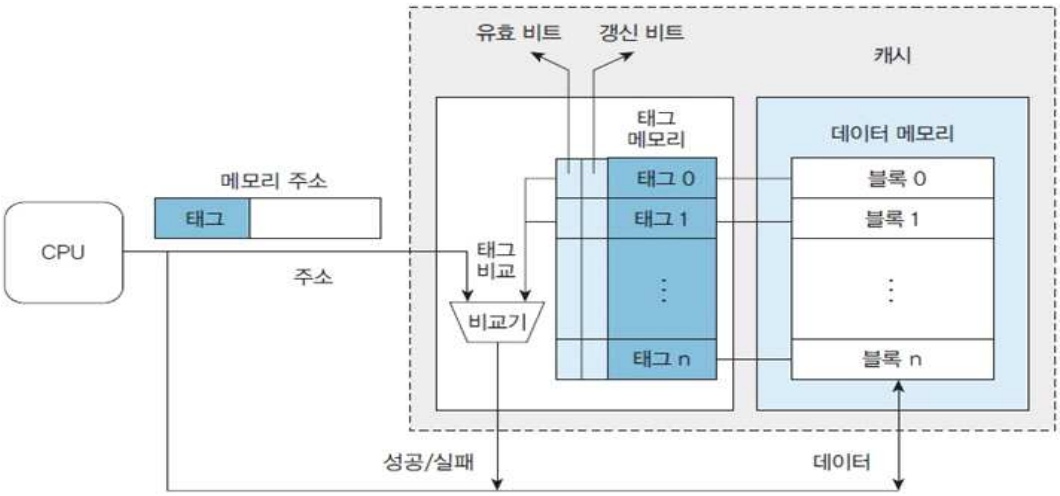
<그림 2.1> 레지스터, 캐시, 메모리, 보고기억 장치의 상충 관계. 캐시에 지역성이 높은 데이터를 미리 저장하면 캐시를 사용하지 않을 때보다 더 빠른 속도로 마이크로프로세서에게 데이터를 제공할 수 있다.

3) 캐시의 기본 구조(Basic Structure of Cache)

우리는 메모리 계층 구조에 따라서 메인 메모리와 프로세서 사이에 캐시를 도입할 것임을 밝혔다. 그렇다면 캐시는 어떤 구조를 가지고 있으며, 메모리 주소는 어떤 방식으로 캐시에 접근하는 것일까? 캐시는 메인 메모리보다 작은 용량을 가지고 있기 때문에 각 캐시에는 여러 블록들이 적재되어야 하고 근본적으로 메모리와 상이한 구조를 갖는다. 그럼 지금부터 캐시의 기본 구조와 캐시 접근 방법에 대해 알아보도록 하자.

먼저 메모리 주소를 분석해보자. 캐시에 접근하기 위해 메모리 주소는 태그, 인덱스, 오프셋으로 분리된다. 인덱스는 특정 계층에서 해당 블록의 위치를 알려주는 주소 정보를 담고 있다. 즉, 인덱스를 통해 특정 블록이 캐시에서 몇 번째 엔트리에 위치하는지 알 수 있다. 태그는 특정 계층에서 해당 블록이 요청한 워드와 일치하는지를 알려주는 주소 정보이다. 인덱스를 통해 요청한 블록이 사상되는 엔트리를 특정했으면 태그를 비교하여 요청한 블록과 일치하는지 확인할 수 있다. 태그는 인덱스로 사용하지 않은 주소의 상위 비트로 구성된다.

다음으로 캐시의 기본 구조를 분석해보자. 캐시는 크게 블록의 위치에 대한 정보를 담고 있는 태그 메모리(tag store)와 블록의 데이터를 담고 있는 데이터 메모리(data store) 부분으로 나누어진다. 태그 메모리의 각 엔트리는 데이터 메모리 블록과 서로 쌍을 이룬다. <그림 2.2>를 살펴보면 태그 0번은 블록 0번에 대응되고, 태그 1번은 블록 1번에 대응되는 것을 확인할 수 있다. 태그는 메모리 주소의 태그와 비교하여 캐시 적중에 성공 여부를 판단하는 역할을 한다. 캐시 적중에 성공하면 해당 태그와 쌍을 이루는 데이터를 불러오고, 캐시 적중에 실패하면 메모리로부터 데이터를 불러와야 한다. 한편, 캐시 블록이 유효한 정보를 가지고 있는지를 알아내는 방법도 필요하다. 예를 들어 프로세서가 맨 처음 작업을 시작하면 캐시는 비어있을 것이며, 태그 필드는 아무런 의미가 없을 것이다. 엔트리가 타당한 주소를 포함하는지를 표기하기 위해 유효 비트(valid)를 캐시에 부착하고 유효 비트가 0이면 캐시는 비어있는 것으로 간주한다. 또한 캐시 블록이 쓰기 이력이 있는지를 알아내는 방법이 역시 필요한데 유효 비트와 마찬가지로 캐시에 갱신 비트(dirty)를 추가한다. 갱신 비트가 1이면 해당 블록이 쓰기가 수행되었고 새로운 데이터가 저장되었음을 알려준다.

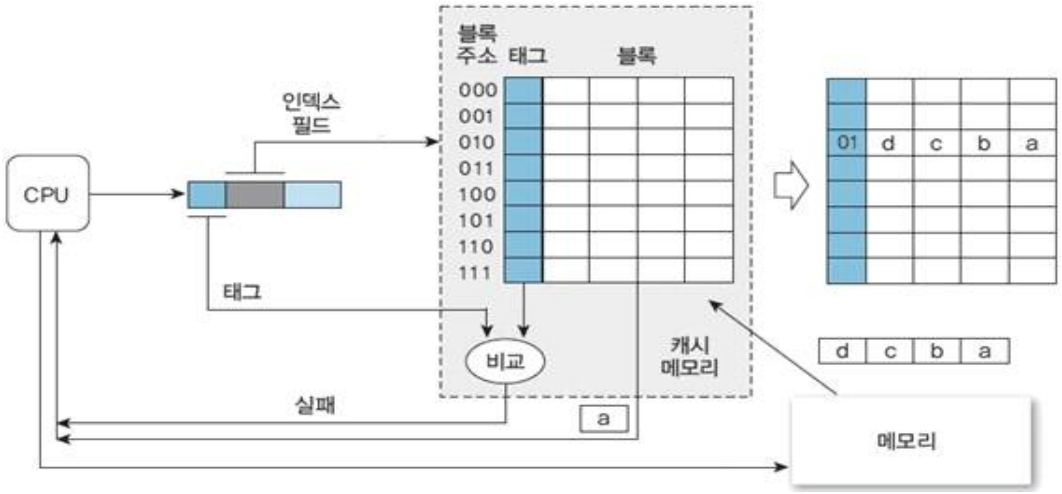


<그림 2.2> 캐시의 기본 구조와 캐시 적중 및 실패 매커니즘. 캐시는 주소의 태그와 비교하기 위한 태그 메모리(Tag store)와 데이터가 저장된 데이터 메모리(Data store)로 나누어진다.

나. 캐시의 사상 방식(Cache Mapping Methods)

1) 직접 사상 방식(Direct Mapping)

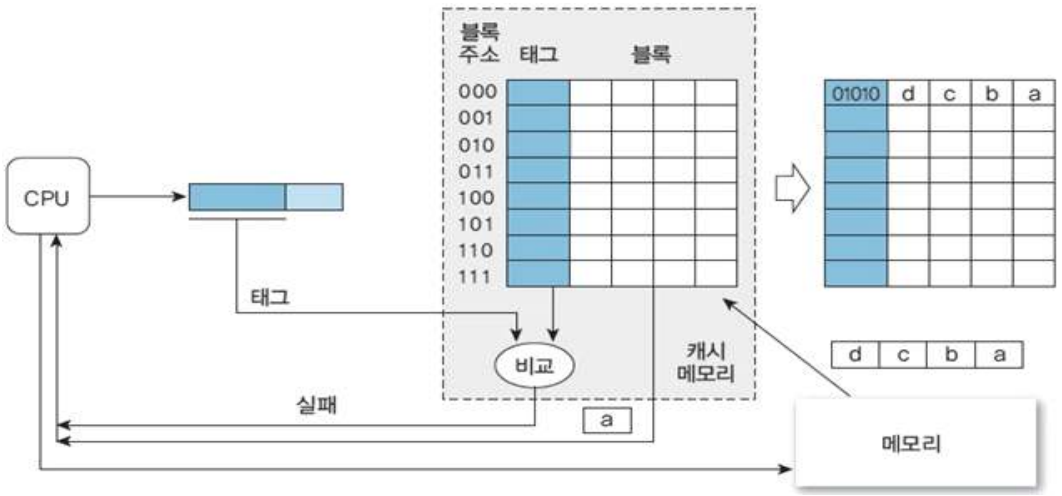
직접 사상 방식은 각 메모리의 위치가 캐시 내의 정확히 한 곳에만 사상되는 캐시 구조를 말한다. 직접 사상 방식은 태그의 길이가 상대적으로 짧고, 주소의 태그를 오직 하나의 태그와 비교하기 때문에 블록이 교체될 때 교체 정책이 따로 필요하지 않다. 직접 사상 방식은 하드웨어 구현이 단순하고 접근 속도가 빠르다는 장점이 있으나, 캐시의 한 곳에 사상되기 때문에 대립 충돌이 자주 발생하여 적중률이 저조하다는 단점이 있다. 직접 사상 방식에서 블록을 찾는 방법은 다음과 같다. 주소의 인덱스 필드를 이용하여 엔트리에 위치한 블록을 선정하고, 선정된 블록의 태그를 주소의 태그와 비교한다. 태그가 같다면 캐시에 블록을 읽거나 쓰기를 곧바로 수행할 수 있다. 그러나 태그가 다르면 메모리에서 새로운 데이터를 불러와 캐시에 저장한 다음 읽거나 쓰기를 수행해야 한다.



<그림 2.3> 직접 사상 방식 캐시의 캐시 적중 및 실패 매커니즘. 2비트 태그, 4비트 인덱스, 3비트 오프셋으로 구성된 8비트 주소가 캐시에 접근하는 과정을 나타낸다.

2) 완전 연관 사상 방식(Fully Associative Mapping)

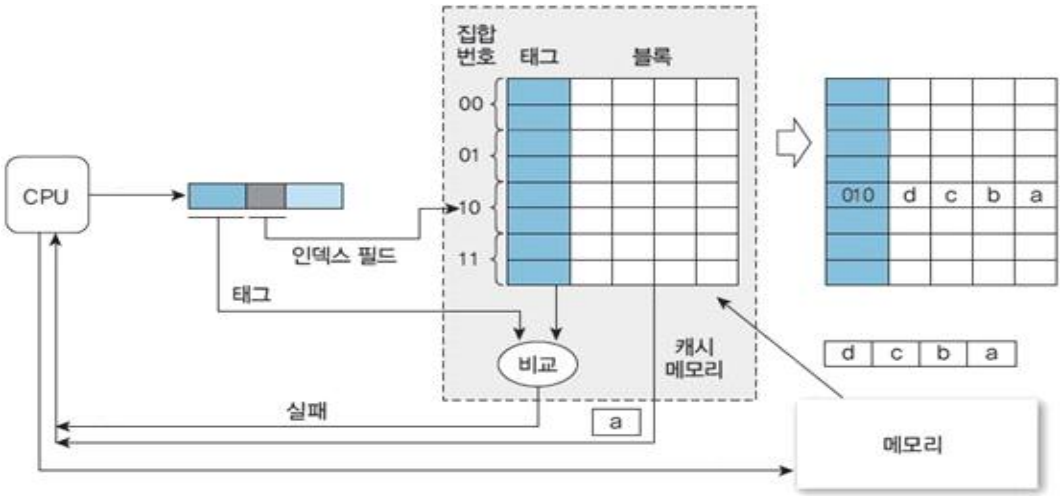
완전 연관 사상 방식은 블록이 캐시의 어느 곳에나 위치할 수 있는 캐시 구조를 말한다. 캐시 블록이 어느 곳에나 위치할 수 있기 때문에 완전 연관 캐시에서 주어진 블록을 찾기 위해서는 캐시 내의 모든 엔트리를 검색해야 한다. 즉 인덱스가 따로 존재하지 않고 오직 태그만으로 블록을 구분한다. 완전 연관 사상 방식에서 블록을 찾는 방법은 다음과 같다. 주소의 태그 필드를 이용하여 태그의 모든 엔트리를 전부 비교한다. 태그가 같다면 캐시에 블록을 읽거나 쓰기를 곧바로 수행할 수 있다. 그러나 태그가 다르면 캐시 교체 정책에 따라서 캐시 블록을 추방하고, 메모리에서 새로운 데이터를 불러와 캐시에 저장한 다음 읽거나 쓰기를 수행해야 한다.



<그림 2.4> 완전 연관 사상 방식 캐시의 캐시 적중 및 실패 매커니즘. 2비트 태그, 4비트 인덱스, 3비트 오프셋으로 구성된 8비트 주소가 캐시에 접근하는 과정을 나타낸다.

3) 집합 연관 사상 방식(Set Associative Mapping)

집합 연관 사상 방식은 각 블록이 배치될 수 있는 위치의 개수가 고정되어 있는 캐시 구조를 말한다. 직접 사상과 완전 연관 사상의 중간 방식으로 한 캐시 블록이 들어갈 수 있는 자리의 개수가 정해져 있으며 각 블록당 n개의 배치 가능한 집합을 갖는 집합 연관 캐시를 n-way 집합 연관 캐시라고 부른다. 집합 연관 사상 방식에서 블록을 찾는 방법은 다음과 같다. 주소의 인덱스 필드를 이용하여 필요한 주소를 가지고 있는 집합을 선정하고, 선정된 집합 내 모든 블록의 태그를 주소의 태그와 비교한다. 태그가 같다면 캐시에 블록을 읽거나 쓰기를 곧바로 수행할 수 있다. 그러나 태그가 다르면 여러 집합이 존재하므로 캐시 교체 정책에 따라서 캐시 블록을 추방하고, 메모리에서 새로운 데이터를 불러와 캐시에 저장한 다음 읽거나 쓰기를 수행해야 한다.



<그림 2.5> 집합 연관 사상 방식 캐시의 캐시 적중 및 실패 매커니즘. 2비트 태그, 4비트 인덱스, 3비트 오프셋으로 구성된 8비트 주소가 캐시에 접근하는 과정을 나타낸다.

다. 캐시 접근(Accessing Cache)

1) 캐시 실패(Cache Miss)

캐시 실패는 프로세서가 요구한 데이터가 상위 계층의 어떤 블록에 없을 때를 말하는데 크게 최초 시작 실패(cold miss), 용량 실패(capacity miss), 그리고 대립 실패(conflict miss)로 나누어진다. 최초 시작 실패는 캐시가 아직 완전히 활용되지 않은 상황에서 캐시에 접근할 때 발생하는 실패이다. 프로세서가 맨 처음 작업을 시작하면 캐시는 비어있는데 이때 캐시 블록은 아무런 정보를 가지고 있지 않다. 캐시에 valid 비트를 도입하여 valid가 0이면 최초 시작 실패가 발생했음을 알 수 있다. 용량 실패는 캐시의 한정된 크기로 인해 발생하는 실패이다. 캐시의 크기가 프로그램보다 작아서 메모리 블록을 교체한 다음에 다시 그 블록을 가져올 때 필연적으로 발생한다. 대립 실패는 여러 블록들이 하나의 집합을 놓고 서로 사상할 때 발생하는 실패이다. 캐시가 충분히 활용된 상황에서 직접 사상 캐시와 집합 연관 사상 캐시는 엔트리를 두고 여러 블록들이 교체되는 경우가 발생한다. 캐시의 인덱스와 태그를 모든 집합에 대해 비교하여 valid가 모두 1이면 대립 실패가 발생했음을 알 수 있다.

캐시 실패	설명
최초 시작 실패 (Cold Miss)	최초 시작 실패 또는 필수 실패(Compulsory)는 캐시에 존재하지 않는 블록에 처음으로 접근할 때 발생하는 캐시 실패이다.
용량 실패 (Capacity Miss)	캐시가 프로그램 수행 중 요청되는 모든 블록을 포함할 수 없어서 블록이 교체되고 나중에 다시 그 블록을 가져올 때 발생하는 실패이다.
대립 실패 (Conflict Miss)	직접 사상, 집합 연관 사상에서 여러 블록들이 하나의 집합에 대해 경쟁을 벌일 때 발생하는 캐시 실패이다.

<표 2.2> 캐시 실패의 대표적인 세 가지 종류.

2) 교체 정책(Replacement Policy)

캐시에 더 이상 빈 블록이 없는 상황에서 새로운 데이터를 읽거나 쓰기 위해 캐시에 접근할 때 대립 실패(conflict miss)가 발생한다. 대립 실패가 발생하였으면 캐시는 캐시 블록을 교체하기 위해 추방할 캐시 블록을 선택해야 한다. 직접 사상 캐시에서 블록은 지정된 딱 한 곳에 들어갈 수 있으므로 그 자리를 차지하고 있는 블록을 교체하면 된다. 그러나 집합 연관 캐시에서 블록은 선정된 집합 중에서 블록을 어디에 위치시킬지 선택한 다음 블록을 교체해야 한다. 마찬가지로 완전 연관 캐시에서 블록은 전체 블록 중에서 어디에 위치시킬지 선택한 다음 블록을 교체해야 한다. 그렇다면 캐시에서 블록을 교체할 때 어느 블록을 선택하고 교체해야 할까? 캐시 블록을 교체하는 정책에는 여러 가지가 있는데 가장 대표적인 Least Recently Used(LRU)와 Second Chance Algorithm(SCA) 이렇게 두 가지 알고리즘을 알아보도록 하자.

가) LRU 알고리즘 (Least Recently Used, LRU)

LRU 알고리즘은 캐시 실패가 발생했을 때 가장 오래된 블록을 버리고 새로운 블록으로 교체하는 알고리즘이다. LRU 알고리즘은 데이터의 최신성(Recency)을 고려한 것으로 가장 오래 전에 써진 캐시 블록을 기억하여 캐시 블록을 교체해야 할 때 캐시에서 추방시킨다.

순서	1	2	3	4	5	6	7	8	9	10	11	12
패턴	1	2	3	4	5	6	3	4	0	6	3	2
태그	태그	태그	태그	태그	태그	태그	태그	태그	태그	태그	태그	태그
엔트리	1	1	1	1	5	5	5	5	5	5	5	2
		2	2	2	2	6	6	6	6	6	6	6
			3	3	3	3	3	3	0	0	0	0
				4	4	4	4	4	4	4	3	3
		Cold miss			Capacity miss				Capacity miss		Capacity miss	

<그림 2.6> 4 엔트리, 1워드 블록, 완전 연관 캐시의 LRU 알고리즘. 패턴이 1, 2, 3, 4, 5, 6, 3, 4, 0, 6, 3, 2 순서대로 나아갈 때 엔트리의 변화를 나타낸 것으로 빨간 배경색을 가진 엔트리가 가장 오래된 엔트리이다. 패턴이 12번 바뀔 때 최초 시작 실패는 4번, 용량 실패는 5번 발생하였다.

나) SCA 알고리즘 (Second Chance Algorithm SCA)

SCA 알고리즘은 캐시 실패가 발생했을 때 가장 오래된 블록을 버리되, 이전에 적중된 이력이 있는 블록은 기회를 한 번 더 주어 다음으로 가장 오래된 블록을 교체하는 알고리즘이다. SCA 알고리즘은 데이

터의 최신성(Recency)과 빈번성(Frequency)을 고려한 것으로 가장 오래 전에 써진 캐시 블록을 기억하여 캐시 블록을 교체해야 할 때, 적중 이력을 살펴본다. 적중 이력이 있다면 캐시에서 추방시키지 않고 다음 오래된 캐시 블록을 살펴본다. 다음 오래된 캐시 블록의 적중 이력이 없다면 캐시에서 추방시킨다.

순서	1		2		3		4		5		6		7		8		9		10		11		12	
패턴	1		2		3		4		5		6		3		4		0		6		3		2	
태그/sca	태그	sca	태그	sca	태그	sca	태그	sca	태그	sca	태그	sca	태그	sca	태그	sca	태그	sca	태그	sca	태그	sca	태그	sca
엔트리	1	0	1	0	1	0	1	0	5	0	5	0	5	0	5	0	0	0	0	0	0	0	0	0
		0	2	0	2	0	2	0	2	0	6	0	6	0	6	0	6	0	6	1	6	1	6	0
		0		0	3	0	3	0	3	0	3	0	3	1	3	1	3	0	3	0	3	1	3	0
		0		0		0	4	0	4	0	4	0	4	0	4	1	4	0	4	0	4	0	2	0

Cold miss Capacity miss Capacity miss Capacity miss

<그림 2.7> 4 엔트리, 1워드 블록, 완전 연관 캐시의 SCA 알고리즘. 패턴이 1, 2, 3, 4, 5, 6, 3, 4, 0, 6, 3, 2 순서대로 나아갈 때 엔트리의 변화를 나타낸 것으로 빨간 배경색을 가진 엔트리가 가장 오래된 엔트리, 노란색 배경을 가진 sca는 적중된 이력을 나타낸다. 패턴이 12번 바뀔 때 최초 시작 실패는 4번, 용량 실패는 5번 발생하였다.

3) 캐시 쓰기 정책(Write Cache Policy)

캐시 쓰기는 조금 다른 방식으로 작동한다. 캐시가 있는 컴퓨터라면 데이터는 메모리와 캐시, 이 두 곳에 동시에 존재한다. 그런데 sw 명령어로 데이터 쓰기를 할 때 데이터를 캐시에만 쓰고 메인 메모리에 쓰지 않을 경우, 메인 메모리는 캐시와 서로 다른 데이터를 갖게 된다. 즉, 캐시는 최신 데이터를 갖고 메모리는 더 이상 사용하지 않는 오래된 데이터를 갖게 된다. 이런 불일치 문제를 해결하기 위해 몇 가지 캐시 쓰기 정책이 도입되었는데 캐시의 데이터를 메모리와 일치시키는 시점에 따라서 즉시 쓰기와 나중 쓰기로 나누어진다.

첫 번째는 즉시 쓰기(Write-through)이다. 즉시 쓰기는 새로운 데이터 쓸 때 캐시와 메모리에 항상 같이 쓰는 방식이다. 두 저장 공간에 항상 데이터를 같이 쓰기 때문에 안정적이고 구현이 쉽지만, 쓰기를 수행하기 위해 계속해서 메인 메모리를 방문해야 하므로 시간이 오래 걸려 좋은 성능을 제공하기 어렵다. 두 번째는 나중 쓰기(Write-back)이다. 나중 쓰기는 새로운 데이터 쓸 때 캐시 블록에 먼저 쓰고 나중에 그 캐시 블록이 교체될 때 쓰기에 의해 데이터가 바뀐 블록이면 그때 메모리에 쓰는 방식이다. 캐시에 dirty 비트를 도입하여 쓰기 이력을 기록해두면 가까운 과거에 새로운 데이터가 캐시에 써졌고 메모리에는 아직 써지지 않은 상태라는 것을 알 수 있다. 나중 쓰기는 메인 메모리 접근 횟수를 줄여 속도가 빨라지기 때문에 좋은 성능을 제공하지만 즉시 쓰기보다 설계 및 구현이 더 어렵다.

쓰기 처리 방식	설명
즉시 쓰기 (Write-through)	쓰기가 항상 캐시 블록과 메모리를 동시에 갱신한다.
나중 쓰기 (Write-back)	쓰기가 캐시 블록을 먼저 갱신하고, 캐시 블록이 교체될 때 메모리에 갱신한다.

<표 2.3> 캐시 쓰기 정책 중 즉시 쓰기와 나중 쓰기.

3. Single Cycle MIPS with Cache

제 3장, Single Cycle MIPS with Cache는 제 2장에서 학습한 내용을 바탕으로 싱글 사이클에 캐시를 구현한 DirectMappingCache.c와 SetAssociativeCache.c 프로그램을 분석한다.

가. 개발 환경

프로젝트4 Single Cycle MIPS with Cache 프로그램은 Windows 10 운영 체제에서 Visual Studio 2019를 통해 C언어로 작성되었다. simple.bin, simple2.bin, simple3.bin, simple4.bin, fib.bin, gcd.bin, input4.bin 파일은 DirectMappingCache.c, SetAssociativeCache.c와 같은 폴더 안에 있어야 한다.

나. 프로그램 분석

프로젝트4 프로그램은 DirectMappingCache.c와 SetAssociativeCache.c 두 가지 c파일로 작성되었다. DirectMappingCache.c는 캐시 크기가 4KB이고 캐시 라인 크기가 64B인 직접 사상 캐시를 구현한 것이고 SetAssociativeCache.c는 캐시 크기가 4KB이고 캐시 라인 크기가 64B인 4-집합 연관 캐시를 구현한 것이다. 제시된 것 외에 다른 캐시 크기나 다른 n-집합 연관의 결과를 확인하고 싶으면 앞으로 설명할 방법에 따라서 일부 변수를 수정하길 바란다. 참고로 제 5장, 결과에서 다양한 캐시 크기와 직접 사상 캐시, 2-집합 연관 캐시, 4-집합 연관 캐시의 결과를 확인할 수 있다. 이번 프로그램 분석은 캐시 구현과 사상 방식을 중심으로 서술하려고 한다. 따라서 싱글 사이클에서 이미 다루었거나, 이해하기 어렵지 않은 코드는 분석에서 제외하였으니 이점 참고하길 바란다.

1) DirectMappingCache.c

가) 구조체와 전역 변수(Structures and Global Variables)

```
(헤더 파일 선언 생략)
(구조체 선언 생략)

struct CACHE {…… ①
  unsigned int tag;
  unsigned int valid;
  unsigned int dirty;
  unsigned int sca; // second chance algorithm
  int DATA[16];      // 64byte cache line size
};

typedef struct {…… ②
  int tag;
  int index;
  int offset;
} Address;
```

```
struct CACHE* DirectCACHE; // Direct … ③
//struct CACHE* SetCACHE[2]; // 2-Way
//struct CACHE* SetCACHE[4]; // 4-Way
struct CACHE* Walker;…… ④
int* oldField;

(함수 선언 생략)
(전역 변수 선언 생략)

int setCacheSize, setCachelineSize;…… ⑤
int indexLength, tagLength, offsetLength;
int countColdMiss, countConflictMiss;…… ⑥
int countHit;
float hitRate, missRate, AMAT;
bool hit, cold, conflict;
```

① 구조체 CACHE는 캐시 블록을 저장하는 구조체로 주소의 태그와 비교할 tag, 유효 비트를 저장할 valid, 나중 쓰기 이력을 저장할 dirty, 캐시 적중 이력을 저장할 sca, 데이터를 저장할 DATA[16] 변수가 선언되어 있다.

② 구조체 Address는 주소의 구조체로 캐시 블록의 태그와 비교할 tag, 캐시 블록의 엔트리를 가리

킬 index, 캐시 블록의 워드를 구분하는데 사용할 offset 변수가 선언되어 있다.

③ 직접 사상 캐시, 2-집합 연관 캐시, 그리고 4-집합 연관 캐시를 위한 구조체 포인터 DirectCACHE, SetCACHE[2], SetCACHE[4]를 선언한다.

④ 캐시 블록을 탐색하기 위한 구조체 포인터 Walker와 가장 오래된 블록의 집합을 찾기 위한 정수 포인터 oldField를 선언한다.

⑤ 프로젝트4 기본구현 목록에 해당하는 최초 시작 실패 횟수, 대립 실패 횟수, 적중 횟수, 적중률, 실패율, 평균 메모리 접근 시간을 변수로 선언한다.

나) SetCache 함수

```
void SetCache(void) {
  setCacheSize = 12.0;·····  ①
  setCachelineSize = 6.0;
  indexLength = setCacheSize - setCachelineSize;
  offsetLength = setCachelineSize;
  tagLength = 32 - (indexLength + offsetLength);
  DirectCACHE = malloc(sizeof(struct CACHE) * pow(2.0, (double)indexLength));·····  ②

  (중략)
  for (int i = 0; i < pow(2.0, (double)indexLength); i++) {·····  ③
    memset(&DirectCACHE[i], 0, sizeof(struct CACHE));
  }

  (중략)
}
```

① DirectCACHE의 캐시 크기와 캐시 라인 크기를 설정한다. setCacheSize의 숫자 12는 4KB(2^12KB)를 의미하고, setCachelineSize의 숫자 6은 64B(2^6B)를 의미한다. 다른 캐시 크기를 확인하고자 하면 <표 3.1>을 참고하여 숫자를 변경하길 바란다.

캐시 크기	4KB	8KB	32KB
setCacheSize	12	13	15

<표 3.1> 캐시 크기에 따른 setCacheSize 변수 설정. 4KB는 2^12KB, 8KB는 2^13KB, 32KB는 2^15KB이므로 승수에 해당하는 12, 13, 15를 변수에 대입하면 된다.

② 엔트리 수(2^6)만큼 구조체 DirectCACHE를 동적 할당한다.

③ 동적 할당한 구조체 DirectCACHE의 모든 변수를 0으로 초기화한다.

다) readMem, writeMem 함수

```
int readMem(int address) {·····  ①
  ADR.tag =  (address >> (indexLength + offsetLength)) & 0x000fffff;·····  ②
  ADR.index =  (address >> offsetLength) & 0x0000003f;
  ADR.offset =  address & 0x0000003f;

  checkCache();·····  ③
  Read_updateCache(address);·····  ④
```

```
  return Walker->DATA[ADR.offset / 4];····· ⑤
}

void writeMem(int address, int writeData) {
  ADR.tag =  (address >> (indexLength + offsetLength)) & 0x000fffff;····· ②
  ADR.index =  (address >> offsetLength) & 0x0000003f;
  ADR.offset = address & 0x0000003f;

  checkCache(); ····· ③
  Write_updateCache(address); ····· ⑥
  Walker->DATA[ADR.offset / 4] = writeData;····· ⑦
  return;
}
```

① 인출된 주소를 매개 변수로 전달받아 비트 연산을 통해 태그, 인덱스, 오프셋으로 구분한다. 캐시 크기가 4KB, 캐시 라인 크기가 64B일 때 태그, 인덱스, 오프셋의 크기는 각각 20, 6, 6이다.

② 태그는 인덱스와 오프셋의 크기를 합한 만큼 오른쪽으로 비트 연산을 하고, 상위 비트가 1일 경우를 제외하기 위해 0x000fffff와 and 연산한다. 인덱스는 오프셋의 크기만큼 오른쪽으로 비트 연산을 하고, 상위 비트가 1일 경우를 제외하기 위해 0x0000003f와 and 연산한다. 오프셋은 상위 비트가 1일 경우를 제외하기 위해 0x00000003f와 and 연산한다.

③ checkCache 함수를 호출하여 캐시 적중과 실패를 결정한다.

④ Read_updateCache 함수를 호출한다. 자세한 내용은 Read_updateCache 함수에서 설명한다.

⑤ Walker 포인터가 가리키는 데이터를 읽는다.

⑥ Write_updateCache 함수를 호출한다. 자세한 내용은 Write_updateCache 함수에서 설명한다.

⑦ Walker 포인터가 가리키는 데이터에 매개변수로 전달받은 새로운 writeData를 쓴다.

라) checkCache 함수

```
void checkCache(void) {
  Walker = &DirectCACHE[ADR.index];····· ①

  switch (Walker->valid) {
  case 0:····· ②
    hit = false;
    cold = true;
    conflict = false;
    countColdMiss++;
    break;

  case 1:
    if (Walker->tag == ADR.tag) {····· ③
      hit = true;
      cold = false;
      conflict = false;
      countHit++;
    }
    else {····· ④
      hit = false;
      cold = false;
      conflict = true;
      countConflictMiss++;
    }
    break;

  default:
    break;
  }
  return;
}
```

① 구조체 포인터 Walker가 주소의 index번째 캐시 블록을 가리킨다.

② Walker가 가리키는 캐시 블록의 valid가 0인 경우, 최초 시작 실패(cold miss)에 해당한다. 따라서 변수 cold를 true로 바꾸고 countColdMiss에 1을 더한다.

③ 캐시 블록의 valid가 1이고 주소의 태그와 블록의 태그가 같은 경우, 적중(hit)에 해당한다. 따라서 변수 hit를 true로 바꾸고 countHit에 1을 더한다.

④ 캐시 블록의 valid가 1이고 주소의 태그와 블록의 태그가 다른 경우, 대립 실패(conflict miss)에 해당한다. 따라서 변수 conflict를 true로 바꾸고 countConflictMiss에 1을 더한다.

마) Read_updateCache 함수

```
void Read_updateCache(int address) {
  if (hit) {·····①
    Walker = &(DirectCACHE[ADR.index]);
    return;
  }

  if (cold) {·····②
    Walker = &(DirectCACHE[ADR.index]);
    int newAddress = address & 0xfffffff0;
    Walker->tag = ADR.tag;
    Walker->valid = 1;
    Walker->dirty = 0;

    for (int i = 0; i < 16; i++) {
      Walker->DATA[i] = MEMORY[newAddress / 4 + i];
    }
    return;
  }

  if (conflict) {·····③
    Walker = &(DirectCACHE[ADR.index]);
    int oldAddress = ((Walker->tag << indexLength) + ADR.index) << offsetLength;
    int newAddress = address & 0xfffffff0;

    if (Walker->dirty == 1) {·····④
      for (int i = 0; i < 16; i++) {
        MEMORY[oldAddress / 4 + i] = Walker->DATA[i];
      }
    }

    for (int i = 0; i < 16; i++) {·····⑤
      Walker->DATA[i] = MEMORY[newAddress / 4 + i];
    }

    Walker->tag = ADR.tag;
    Walker->valid = 1;
    Walker->dirty = 0;
    return;
  }
}
```

① 캐시 적중일 때, 해당 캐시 블록을 바로 읽을 수 있으므로 Walker가 주소 index번째 캐시 블록을 가리킨다.

② 최초 시작 실패일 때, 해당 캐시 블록은 비어있으므로 메모리에서 데이터를 불러와 저장한 다음 캐시 블록을 읽어야 한다. Walker가 주소 index번째 캐시 블록을 가리키고, 매개변수로 전달받은 주소의 [tag+index+0x000000] 부분부터 offset-2 만큼 메모리에서 데이터를 불러와 캐시 블록의 DATA[16]에 저장한다. 캐시 블록의 tag에 주소의 tag를 저장하고, valid를 1로 바꾼다.

③ 대립 실패일 때, 해당 캐시 블록을 교체한 다음 캐시 블록을 읽어야 한다.

④ dirty가 1인 경우, 나중 쓰기를 해야 하는 캐시 블록이므로, Walker의 [tag+index+0x000000] 부분

부터 offset-2 만큼 캐시 블록 DATA[16]에서 데이터를 불러와 메모리에 저장한다. 나중 쓰기를 완료했으면 dirty를 0으로 바꾼다.

⑤ Walker가 주소 index번째 캐시 블록을 가리키고, 매개변수로 전달받은 주소의 [tag+index+0x000000] 부분부터 offset-2 만큼 메모리에서 데이터를 불러와 캐시 블록의 DATA[16]에 저장한다. 캐시 블록의 tag에 주소의 tag를 저장하고, valid를 1로 바꾼다.

바) Write_updateCache 함수

```
void Write_updateCache(int address) {
  Walker = &(DirectCACHE[ADR.index]);
  if (hit) {·····①
    Walker->dirty = 1;
    return;
  }

  if (cold) {·····②
    Walker->tag = ADR.tag;
    Walker->valid = 1;
    Walker->dirty = 1;
    return;
  }

  if (conflict) {·····③
    int oldAddress = ((Walker->tag << indexLength) + ADR.index) << offsetLength;

    if ((Walker->dirty == 1)) {·····④
      for (int i = 0; i < 16; i++) {
        MEMORY[oldAddress / 4 + i] = Walker->DATA[i];
      }
    }

    Walker->tag = ADR.tag;
    Walker->valid = 1;
    Walker->dirty = 1;
    return;
  }
}
```

① 캐시 적중일 때, 해당 캐시 블록에 바로 쓸 수 있으므로 Walker가 주소 index번째 캐시 블록을 가리킨다. 가까운 미래에 나중 쓰기를 해야 하므로 dirty를 1로 바꾼다.

② 최소 시작 실패일 때, 해당 캐시 블록에 바로 쓸 수 있으므로 Walker가 주소 index번째 캐시 블록을 가리킨다. 캐시 블록의 tag에 주소의 tag를 저장하고 valid를 1로 바꾼다. 또한 가까운 미래에 나중 쓰기를 해야 하므로 dirty를 1로 바꾼다. 이후 writMem 함수에서 writeData를 전달받아 데이터를 캐시에 쓴다.

③ 대립 실패일 때, 해당 캐시 블록을 교체한 다음 캐시 블록에 써야 한다.

④ dirty가 1인 경우, 나중 쓰기를 해야 하는 캐시 블록이므로, Walker의 [tag+index+0x000000] 부분부터 offset-2 만큼 캐시 블록 DATA[16]에서 데이터를 불러와 메모리에 저장한다.

⑤ Walker가 주소 index번째 캐시 블록을 가리키고, 캐시 블록의 tag에 주소의 tag를 저장하고 valid를 1로 바꾼다. 또한 가까운 미래에 나중 쓰기를 해야 하므로 dirty를 1로 바꾼다. 이후 writeMem 함수에서 writeData를 전달받아 데이터를 캐시에 쓴다.

2) SetAssociativeCache.c

가) SetCache 함수

```
void SetCache(void) {
  setCacheSize = 12.0; ····· ①
  setCachelineSize = 6.0;
  indexLength = setCacheSize - setCachelineSize;
  offsetLength = setCachelineSize;
  tagLength = 32 - (indexLength + offsetLength);

  oldField = malloc(sizeof(struct CACHE) * pow(2.0, (double)indexLength)); ····· ②

  (중략)
  for (int i = 0; i < pow(2.0, (double)indexLength); i++) {
    memset(&oldField[i], 0, sizeof(struct CACHE));
  }

  for (int i = 0; i < 4; i++) { ····· ③
    SetCACHE[i] = malloc(sizeof(struct CACHE) * pow(2.0, (double)indexLength));

    (중략)
    for (int j = 0; j < pow(2.0, (double)indexLength); j++) { ····· ④
      memset(&SetCACHE[i][j], 0, sizeof(struct CACHE));
    }
  }

  (중략)
  return;
}
```

① SetCACHE의 캐시 크기와 캐시 라인 크기를 설정한다. setCacheSize의 숫자 12는 4KB(2^12KB)를 의미하고 setCachelineSize의 숫자 6은 64B(2^6B)를 의미한다. 다른 캐시 크기를 확인하고자 하면 <표 3.1>을 참고하여 숫자를 변경하길 바란다.

② 엔트리 수(2^6)만큼 정수 포인터 oldField를 동적 할당하고 oldField의 모든 변수를 0으로 초기화한다.

④ 엔트리 수(2^6)만큼 구조체 DirectCACHE[0], DirectCACHE[1], DirectCACHE[2] 그리고 DirectCACHE[3]을 동적 할당한다.

⑤ 동적 할당한 구조체 DirectCACHE[0], DirectCACHE[1], DirectCACHE[2] 그리고 DirectCACHE[3]의 모든 변수를 0으로 초기화한다.

나) checkCache 함수

```
void checkCache(void) {
  for (int i = 0; i < 4; i++) {····················· ①
    Walker = &SetCACHE[i][ADR.index];

    switch(Walker->valid) {
      case 0:······································ ②
        hit = false;
        cold = true;
        conflict = false;
        field = i;
        countColdMiss++;
        return;
      break;

      case 1:
        if (Walker->tag == ADR.tag) {····· ③
          Walker->sca = 1;
          hit = true;
          cold = false;
          conflict = false;
          field = i;
          countHit++;
          return;
        }
        else if (Walker->tag != ADR.tag) {
          if(i == 3) {····························· ④
            hit = false;
            cold = false;
            conflict = true;
            countConflictMiss++;
            return;
          }
        }
        else;
      break;

      default:
      break;
    }
  }
  return;
}
```

① 구조체 포인터 Walker가 i번째 집합의 주소 index번째 캐시 블록을 가리킨다. i번째 집합에서 캐시 적중 또는 실패를 알 수 없으면 Walker는 다음 i+1번째 집합의 주소 index번째 캐시 블록을 가리킨다.

② Walker가 가리키는 캐시 블록의 valid가 0인 경우, 최초 시작 실패(cold miss)에 해당한다. 따라서 변수 cold를 true로 바꾸고 countColdMiss에 1을 더한다. 또한 캐시 실패가 발생한 집합을 field에 저장해 이 집합에서 캐시 읽기와 쓰기를 수행할 수 있도록 한다.

③ 캐시 블록의 valid가 1이고 주소의 태그와 블록의 태그가 같은 경우, 적중(hit)에 해당한다. 따라서 변수 hit를 true로 바꾸고 countHit에 1을 더한다. 또한 캐시 적중이 발생한 집합을 field에 저장해 이 집합에서 캐시 읽기와 쓰기를 수행할 수 있도록 하고, 적중 이력이 발생했으므로 sca를 1로 바꾼다.

④ Walker가 가리키는 모든 집합의 캐시 블록이 valid가 1이고 주소의 태그와 블록의 태그가 다른 경우, 대립 실패(conflict miss)에 해당한다. 따라서 변수 conflict를 true로 바꾸고 countConflictMiss에 1을 더한다.

다) Read_updateCache 함수

```
void Read_updateCache(int address) {
  if (hit) {····· ①
    Walker = &SetCACHE[field][ADR.index];
    return;
  }

  if (cold) {····· ②
    Walker = &SetCACHE[field][ADR.index];
    int newAddress = address & 0xfffffffc0;

    Walker->tag = ADR.tag;
    Walker->valid = 1;
    Walker->dirty = 0;
    Walker->sca = 0;

    for (int i = 0; i < 16; i++) {
      Walker->DATA[i] =
      MEMORY[(newAddress / 4) + i];
    }
    return;
  }

  if (conflict) {····· ③
    bool escape = false;

    while (!escape) {····· ④
  Walker = &SetCACHE[oldField[ADR.index]]
           [ADR.index];

      if (Walker->sca == 0) {····· ⑥
        field = oldField[ADR.index];
        oldField[ADR.index]++;
        escape = true;

        if (oldField[ADR.index] > 3)
          oldField[ADR.index] = 0;
      }
      else if (Walker->sca == 1) {····· ⑤
        Walker->sca = 0;
        oldField[ADR.index]++;

        if (oldField[ADR.index] > 3)
          oldField[ADR.index] = 0;
      }
      else;
    }

    Walker = &SetCACHE[field][ADR.index];

    int  oldAddress = ((Walker->tag <<
        indexLength) + ADR.index) <<
        offsetLength;

    int newAddress = address & 0xfffffffc0;

    if (Walker->dirty == 1) {····· ⑦
      for (int i = 0; i < 16; i++) {
        MEMORY[oldAddress / 4 + i] =
        Walker->DATA[i];
      }
    }

    for (int i = 0; i < 16; i++) {····· ⑧
      Walker->DATA[i] =
      MEMORY[(newAddress / 4) + i];
    }

    Walker->tag = ADR.tag;
    Walker->valid = 1;
    Walker->dirty = 0;
    Walker->sca = 0;
    return;
  }
}
```

① 캐시 적중일 때 해당 집합의 캐시 블록을 바로 읽을 수 있으므로 Walker가 field 집합의 주소 index번째 캐시 블록을 가리킨다.

② 최초 시작 실패일 때, 해당 집합의 캐시 블록은 비어있으므로 메모리에서 데이터를 불러와 저장한 다음 캐시 블록을 읽어야 한다. Walker가 field 집합의 주소 index번째 캐시 블록을 가리키고, 매개 변수로 전달받은 주소의 [tag+index+0x000000] 부분부터 offset-2 만큼 메모리에서 데이터를 불러와 캐시 블록의 DATA[16]에 저장한다. 캐시 블록의 tag에 주소의 tag를 저장하고, valid를 1로 바꾼다.

③ 대립 실패일 때, SCA 알고리즘에 따라 선택된 집합의 캐시 블록을 교체한 다음 캐시 블록을 읽어야 한다.

④ SCA 알고리즘에 따라서 교체할 집합의 캐시 블록을 선택한다.

⑤ sca 비트가 1일 경우 적중 이력이 있으므로 해당 캐시 블록은 교체하지 않는다. 단, 교체 대상에 해당되었으므로 sca를 0으로 바꾸고, 다음 오래된 집합을 탐색하기 위해 oldField에 1을 더한다.

⑥ sca 비트가 0일 경우, 적중 이력이 없으므로 해당 캐시 블록을 교체한다. 교체할 블록의 집합을 field에 저장하고 다음 오래된 집합을 가리키기 위해 oldField에 1을 더한다. 교체할 집합의 블록을 찾았기 때문에 반복문을 빠져나온다.

⑦ dirty가 1인 경우, 나중 쓰기를 해야 하는 캐시 블록이므로, Walker의 [tag+index+0x000000] 부분부터 offset-2 만큼 field 집합의 캐시 블록 DATA[16]에서 데이터를 불러와 메모리에 저장한다. 나중 쓰기를 완료했으면 dirty를 0으로 바꾼다.

⑧ Walker가 field 집합의 주소 index번째 캐시 블록을 가리키고, 매개변수로 전달받은 주소의 [tag+index+0x000000] 부분부터 offset-2 만큼 메모리에서 데이터를 불러와 캐시 블록의 DATA[16]에 저장한다. 캐시 블록의 tag에 주소의 tag를 저장하고, valid를 1로 바꾼다.

라) Write_updateCache 함수

```
void Write_updateCache(int address) {
  if (hit) {····· ①
    Walker = &SetCACHE[field][ADR.index];
    Walker->dirty = 1;
    return;
  }

  if (cold) {····· ②
    Walker = &SetCACHE[field][ADR.index];
    Walker->tag = ADR.tag;
    Walker->valid = 1;
    Walker->dirty = 1;
    Walker->sca = 0;
    return;
  }

  if (conflict) {····· ③
    bool escape = false;

    while (!escape) {····· ④
  Walker = &SetCACHE[oldField[ADR.index]]
           [ADR.index];

      if (Walker->sca == 0) {····· ⑥
        field = oldField[ADR.index];
        oldField[ADR.index]++;
        escape = true;

        if (oldField[ADR.index] > 3)
          oldField[ADR.index] = 0;
      }
      else if (Walker->sca == 1) {····· ⑤
        Walker->sca = 0;
        oldField[ADR.index]++;

        if (oldField[ADR.index] > 3)
          oldField[ADR.index] = 0;
      }
      else;
    }

    Walker = &SetCACHE[field][ADR.index];
    int oldAddress = ((Walker->tag <<
        indexLength) + ADR.index) <<
        offsetLength;

    if (Walker->dirty == 1) {····· ⑦
      for (int i = 0; i < 16; i++) {
        MEMORY[oldAddress / 4 + i] =
        Walker->DATA[i];
      }
    }

    Walker->tag = ADR.tag;····· ⑧
    Walker->valid = 1;
    Walker->sca = 0;
    Walker->dirty = 1;
    return;
  }
}
```

① 캐시 적중일 때, 해당 집합의 캐시 블록에 바로 쓸 수 있으므로 Walker가 field 집합의 주소 index번째 캐시 블록을 가리킨다. 가까운 미래에 나중 쓰기를 해야 하므로 dirty를 1로 바꾼다.

② 최초 시작 실패일 때, 해당 집합의 캐시 블록에 바로 쓸 수 있으므로 Walker가 field 집합의 주소 index번째 캐시 블록을 가리킨다. 캐시 블록의 tag에 주소의 tag를 저장하고 valid를 1로 바꾼다. 또한 가까운 미래에 나중 쓰기를 해야 하므로 dirty를 1로 바꾼다.

③ 대립 실패일 때, SCA 알고리즘에 따라 선택된 집합의 캐시 블록을 교체한 다음 캐시 블록에 써야 한다.

④ SCA 알고리즘에 따라서 교체할 집합의 캐시 블록을 선택한다.

⑤ sca 비트가 1일 경우 적중 이력이 있으므로 해당 캐시 블록은 교체하지 않는다. 단, 교체 대상에 해당되었으므로 sca를 0으로 바꾸고, 다음 오래된 집합을 탐색하기 위해 oldField에 1을 더한다.
⑥ sca 비트가 0일 경우, 적중 이력이 없으므로 해당 캐시 블록을 교체한다. 교체할 블록의 집합을 field에 저장하고 다음 오래된 집합을 가리키기 위해 oldField에 1을 더한다. 교체할 집합의 블록을 찾았기 때문에 반복문을 빠져나온다.
⑦ dirty가 1인 경우, 나중 쓰기를 해야 하는 캐시 블록이므로, Walker의 [tag+index+0x000000] 부분부터 offset-2 만큼 field 집합의 캐시 블록 DATA[16]에서 데이터를 불러와 메모리에 저장한다.
⑧ Walker가 field 집합의 주소 index번째 캐시 블록을 가리키고, 캐시 블록의 tag에 주소의 tag를 저장하고, valid를 1로 바꾼다. 또한 가까운 미래에 나중 쓰기를 해야 하므로 dirty를 1로 바꾼다. 이후 writeMem 함수에서 writeData를 전달받아 데이터를 캐시에 쓴다.

4. 문제 및 해결

제 4장, 문제 및 해결에서는 프로그램을 작성하면서 발생했던 몇 가지 문제와 그 해결 과정을 담았다. 디버깅 과정에서 문제가 됐던 부분을 하나하나 분석하고 문제를 해결하는 과정을 살펴보도록 하자.

가. 디버깅

1) DirectMappingCache.c

가) Write_updateCache 함수와 dirty 비트 누락

input4.bin의 결과가 출력되지 않고 무한루프를 도는 오류가 발생했다. 그 외의 다른 바이너리 파일의 결과는 올바르게 출력되는데 input4.bin의 결과가 출력되지 않았고, 오류의 원인을 밝히기 위해 몇 가지 가능성 있는 가설을 세웠다.

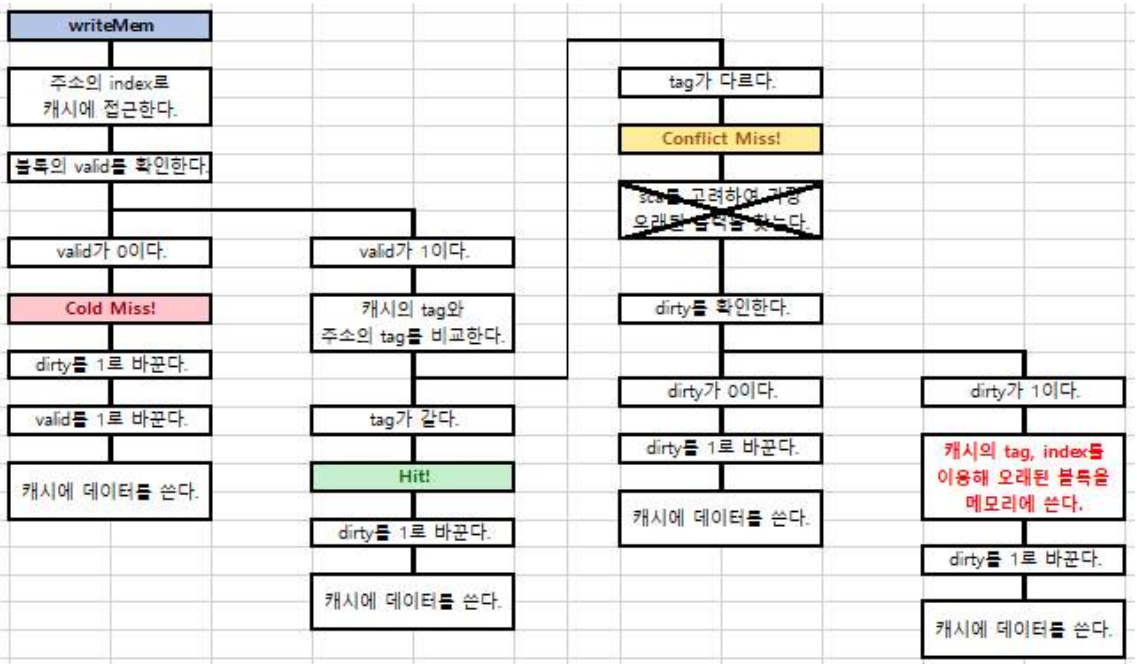
1. 캐시 읽기에서 문제가 발생했다.
➜ 캐시 읽기는 모든 바이너리 파일에서 수행하므로 캐시 읽기가 input4.bin에 문제를 일으켰다면 다른 바이너리 파일도 마찬가지로 문제가 생겼을 것이다.

2. 캐시 쓰기에서 문제가 발생했다.
가. 캐시 쓰기 과정에서 hit, miss, conflict를 잘못 구분하였다.
➜ 문제를 일으킬 가능성이 있으므로 확인해보자.

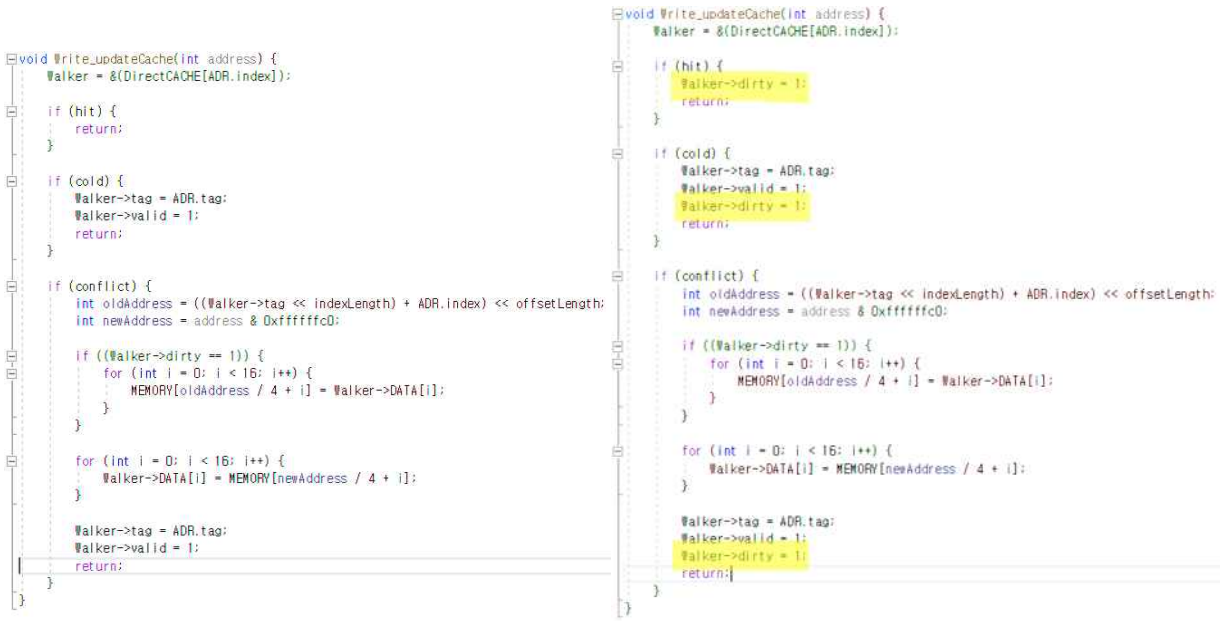
나. hit, miss, conflict에 따라서 수행해야할 쓰기 메커니즘이 잘못되었다.
➜ 문제를 일으킬 가능성이 있으므로 확인해보자.

(나)에 해당하는 조건 'hit, miss, conflict에 따라서 수행해야할 쓰기 메커니즘이 잘못되었다.'를 확인하기 위해 <그림 4.1>과 같이 쓰기 정책의 순서도를 다시 한 번 작성해보았다. 직접 사상 캐시의 쓰기 정책에 따르면 캐시 적중(hit)과 최초 시도 실패(cold miss)이 발생한 블록은 바로 캐시에 데이터를 쓰고, 대립 실패(conflict miss)가 발생한 블록은 dirty 비트를 확인하여 dirty가 0이면 캐시에 데이터를 쓰고, dirty가 1이면 오래된 블록을 메모리에 쓴 다음 캐시에 데이터를 써야했다. 또한 캐시 쓰기를 수행했으므로 캐시 적중과 실패에 관계없이 dirty 비트를 1로 바꾸어야 했다. 순서도를 모두 작성하고 프로그램 코드를 확인해본 결과, 캐시 쓰기 역할을 하는 Write_updateCache 함수에서 dirty 비트를 업데이트하는

코드가 누락되어있다는 것을 알게 되었다. dirty 비트를 1로 인가해주자 올바른 교체가 이루어졌고 프로그램이 원활하게 작동하였다.



<그림 4.1> 직접 사상 캐시의 캐시 쓰기 순서도. 나중 쓰기 정책이 도입된 캐시 쓰기를 하므로 dirty 비트를 1로 바꾸어야 한다.



```
void Write_updateCache(int address) {
    Walker = &(DirectCACHE[ADR.index]);

    if (hit) {
        return;
    }

    if (cold) {
        Walker->tag = ADR.tag;
        Walker->valid = 1;
        return;
    }

    if (conflict) {
        int oldAddress = ((Walker->tag << indexLength) + ADR.index) << offsetLength;
        int newAddress = address & 0xffffffc0;

        if ((Walker->dirty == 1)) {
            for (int i = 0; i < 16; i++) {
                MEMORY[oldAddress / 4 + i] = Walker->DATA[i];
            }
        }

        for (int i = 0; i < 16; i++) {
            Walker->DATA[i] = MEMORY[newAddress / 4 + i];
        }

        Walker->tag = ADR.tag;
        Walker->valid = 1;
        return;
    }
}
```

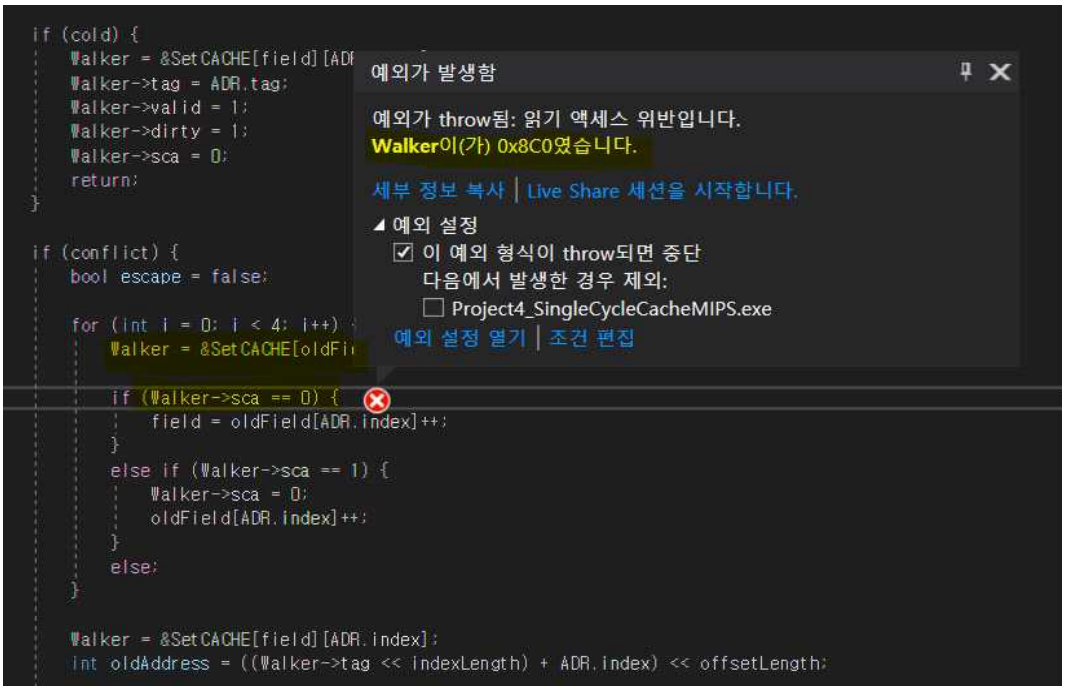
```
void Write_updateCache(int address) {
    Walker = &(DirectCACHE[ADR.index]);

    if (hit) {
        Walker->dirty = 1;
        return;
    }

    if (cold) {
        Walker->tag = ADR.tag;
        Walker->valid = 1;
        Walker->dirty = 1;
        return;
    }

    if (conflict) {
        int oldAddress = ((Walker->tag << indexLength) + ADR.index) << offsetLength;
        int newAddress = address & 0xffffffc0;

        if ((Walker->dirty == 1)) {
            for (int i = 0; i < 16; i++) {
                MEMORY[oldAddress / 4 + i] = Walker->DATA[i];
            }
        }

        for (int i = 0; i < 16; i++) {
            Walker->DATA[i] = MEMORY[newAddress / 4 + i];
        }

        Walker->tag = ADR.tag;
        Walker->valid = 1;
        Walker->dirty = 1;
        return;
    }
}
```

<그림 4.2> dirty 비트를 업데이트 하지 않은 프로그램(왼쪽)과 dirty 비트를 1로 업데이트하는 프로그램(오른쪽). 수정 전인 왼쪽 함수는 dirty 비트가 누락되어 있고 오른쪽 함수처럼 dirty 비트를 1로 바꾸도록 코드를 추가했다.

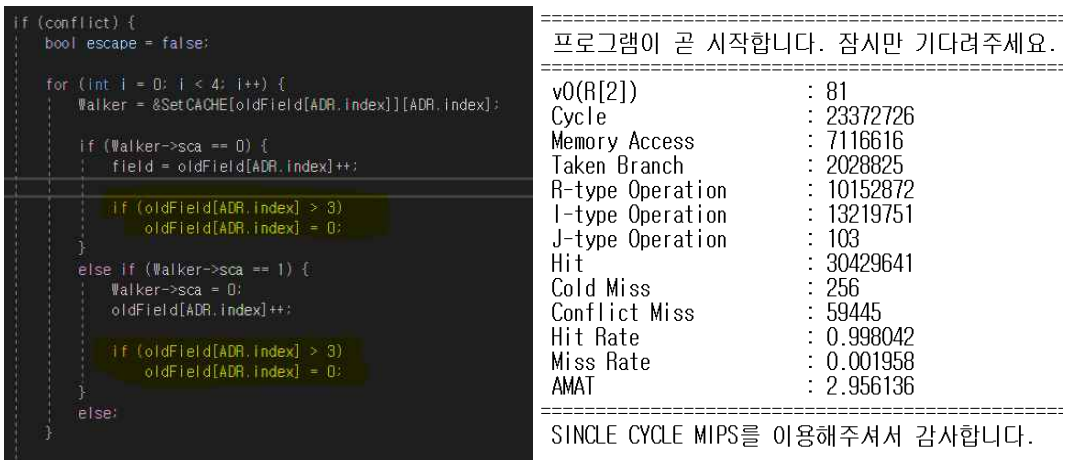
2) SetAssociativeCache.c

가) Conflict miss에 따른 SCA 알고리즘

집합 연관 캐시의 대립 충돌에 따른 교체 정책을 구현하는 부분에서 많은 오류가 발생했고 디버깅을 하는데 시간이 가장 오래 걸렸다. 첫 번째 오류는 <그림 4.3>과 메시지와 함께 발생하였다. 초기 코드에는 for 반복문을 이용하여 가장 오래된 집합을 탐색하고 sca 비트가 1인 경우 한 번 더 기회를 줄 수 있도록 코드를 작성했다. for 반복문을 이용하여 총 4번 루프를 돌면 가장 오래된 집합을 찾을 수 있을 것이라고 기대하였는데 이때 Walker의 액세스 위반이 발생하였다. 캐시 크기가 12고, 캐시 라인 크기가 6인 상황에서 인덱스는 64(2^6)이므로 총 64개의 엔트리를 가질 수 있었다. 그런데 가장 오래된 집합을 가리키기 위해 postfix로 oldField 배열에 1로 더하는 값이 제한이 걸려있지 않았고 존재하지도 않는 4번째, 5번째, 혹은 그 이상의 field를 반환하고 있었다.



<그림 4.3> 대립 충돌에 따른 교체 정책 부분에서 오류가 발생한 코드. 수정 전 코드는 가장 오래된 집합을 가리키기 위해 1만큼 더하는 과정에서 4 이상의 정수를 저장할 가능성이 있다.



<그림 4.4> 대립 충돌에 따른 교체 정책 부분에서 if문을 추가한 코드. 수정 후 코드는 0~3번째 집합만을 가리킬 수 있도록 3을 넘어가면 0번째 집합을 가리키도록 0으로 바꾸어준다.

4-집합 연관 캐시에서 집합의 수는 4개로 제한되어 있기 때문에 가장 오래된 집합을 나타내는 oldField 배열 역시 0부터 3 사이의 값을 가져야 했다. 따라서 가장 오래된 집합을 순환하여 가리키기 위해 순서대로 0 ➜ 1 ➜ 2 ➜ 3 ➜ 0인 구조를 만들어야 했고 따라서 oldField 배열에 4 이상의 값이 저장되면 0으로 바꾸어 주었다. 그 결과 액세스 위반 오류를 해결할 수 있었다.

두 번째 오류는 <그림 4.4>와 같이 input4.bin의 결과가 85가 아닌 81로 출력되는 상황이었다. for 반복문으로 가장 오래된 집합을 찾는 과정에서 무조건 4번 루프를 돌도록 설계했었다. 그러나 for 반복문으로 오래된 집합을 찾는 코드는 두 가지 문제를 가지고 있었다. 4번 루프를 돌기 전에 가장 오래된 집합을 찾을 경우, 남은 루프를 마저 돌면서 원하지 않게 oldFiled 배열의 값이 증가한다는 것과 네 집합에 속한 블록의 sca 비트가 모두 1인 경우, 4번 루프를 도는 것만으로 가장 오래된 집합을 찾을 수 없다는 것이었다. 다시 말해 for 반복문으로 가장 오래된 집합을 찾는 건 적절하지 않았고 while 반복문으로 교체하여 sca 비트가 0인 블록이 발견되는 즉시 반복문을 빠져나오도록 설계 방향을 바꾸기로 했다. 이에 따라 <그림 4.5>와 같이 불리언 형식의 escape 변수를 추가하여 sca 비트가 0인 블록이 발견되면 escape 변수를 true로 바꾸어 while 반복문을 빠져나올 수 있게 하였고, 그 결과 input4.bin의 결과가 85로 올바르게 출력되었다.

```
if (conflict) {
    bool escape = false;

    while (!escape) {
        Walker = &SetCACHE[oldField[ADR.index]][ADR.index];

        if (Walker->sca == 0) {
            field = oldField[ADR.index]++;
            escape = true;

            if (oldField[ADR.index] > 3)
                oldField[ADR.index] = 0;
        }
        else if (Walker->sca == 1) {
            Walker->sca = 0;
            oldField[ADR.index]++;

            if (oldField[ADR.index] > 3)
                oldField[ADR.index] = 0;
        }
        else;
    }
```

```
================================================
 프로그램이 곧 시작합니다. 잠시만 기다려주세요.
================================================
v0(R[2])                : 85
Cycle                   : 23372654
Memory Access           : 7116580
Taken Branch            : 2028843
R-type Operation        : 10152836
I-type Operation        : 13219715
J-type Operation        : 103
Hit                     : 30423112
Cold Miss               : 256
Conflict Miss           : 65866
Hit Rate                : 0.997831
Miss Rate               : 0.002169
AMAT                    : 3.166531
================================================
 SINCLE CYCLE MIPS를 이용해주셔서 감사합니다.
```

<그림 4.5> 대립 충돌에 따른 교체 정책 부분에서 while 반복문으로 수정한 코드. 수정 후 코드는 sca 비트가 0인 블록을 발견하자마자 해당 블록의 집합을 저장하고 즉시 반복문을 빠져나온다.

5. 결과

가. 바이너리 파일 실행

1) simple.bin

- **Direct Mapping Cache**

```
Microsoft Visual Studio 디버그 콘솔
==========================================================
SINGLE CYCLE MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > simple.bin
==========================================================
0. No
1. Yes
==========================================================
모든 실행 결과를 보겠습니까? > 0

==========================================================
프로그램이 곧 시작합니다. 잠시만 기다려주세요.
==========================================================
v0(R[2])              : 0
Cycle                 : 8
Memory Access         : 2
Taken Branch          : 0
R-type Operation      : 4
I-type Operation      : 4
J-type Operation      : 0
Hit                   : 8
Cold Miss             : 2
Conflict Miss         : 0
Hit Rate              : 0.800000
Miss Rate             : 0.200000
AMAT                  : 200.800000
==========================================================
SINCLE CYCLE MIPS를 이용해주셔서 감사합니다.

C:₩Users₩ekrxj₩Desktop₩C Programming₩컴퓨터구조와모바일프로세서₩Project
e MIPS₩Project4_SingleCycleCacheMIPS₩Debug₩Project4_SingleCycleCacheMIP
프로세스 17752개)이(가) 종료되었습니다(코드: 0개).
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅
디버깅이 중지되면 자동으로 콘솔 닫기]를 사용하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

- **4-way Set Associative Cache**

```
Microsoft Visual Studio 디버그 콘솔
==========================================================
SINGLE CYCLE MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > simple.bin
==========================================================
0. No
1. Yes
==========================================================
모든 실행 결과를 보겠습니까? > 0

==========================================================
프로그램이 곧 시작합니다. 잠시만 기다려주세요.
==========================================================
v0(R[2])              : 0
Cycle                 : 8
Memory Access         : 2
Taken Branch          : 0
R-type Operation      : 4
I-type Operation      : 4
J-type Operation      : 0
Hit                   : 8
Cold Miss             : 2
Conflict Miss         : 0
Hit Rate              : 0.800000
Miss Rate             : 0.200000
AMAT                  : 200.799988
==========================================================
SINCLE CYCLE MIPS를 이용해주셔서 감사합니다.

C:₩Users₩ekrxj₩Desktop₩C Programming₩컴퓨터구조와모바일프로세서₩Project
e MIPS₩Project4_SingleCycleCacheMIPS₩Debug₩Project4_SingleCycleCacheMIP
프로세스 22456개)이(가) 종료되었습니다(코드: 0개).
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅
디버깅이 중지되면 자동으로 콘솔 닫기]를 사용하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

2) simple2.bin

- **Direct Mapping Cache**

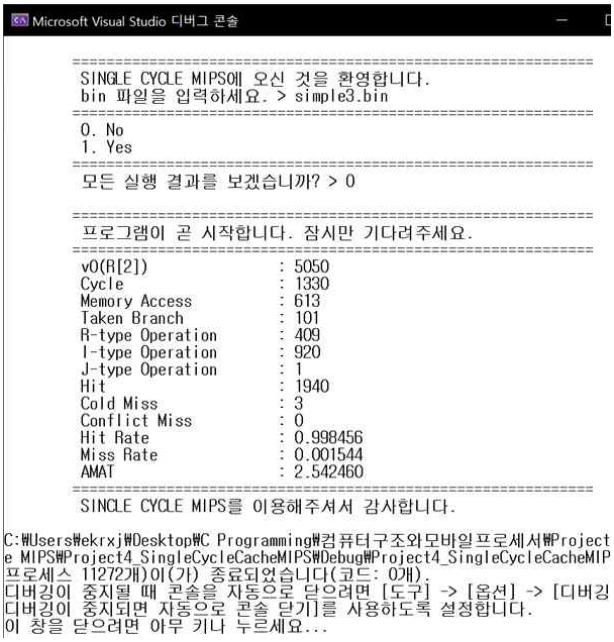
```
Microsoft Visual Studio 디버그 콘솔
==========================================================
SINGLE CYCLE MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > simple2.bin
==========================================================
0. No
1. Yes
==========================================================
모든 실행 결과를 보겠습니까? > 0

==========================================================
프로그램이 곧 시작합니다. 잠시만 기다려주세요.
==========================================================
v0(R[2])              : 100
Cycle                 : 10
Memory Access         : 4
Taken Branch          : 0
R-type Operation      : 3
I-type Operation      : 7
J-type Operation      : 0
Hit                   : 12
Cold Miss             : 2
Conflict Miss         : 0
Hit Rate              : 0.857143
Miss Rate             : 0.142857
AMAT                  : 143.714286
==========================================================
SINCLE CYCLE MIPS를 이용해주셔서 감사합니다.

C:₩Users₩ekrxj₩Desktop₩C Programming₩컴퓨터구조와모바일프로세서₩Project
e MIPS₩Project4_SingleCycleCacheMIPS₩Debug₩Project4_SingleCycleCacheMIP
프로세스 10108개)이(가) 종료되었습니다(코드: 0개).
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅
디버깅이 중지되면 자동으로 콘솔 닫기]를 사용하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

- **4-way Set Associative Cache**

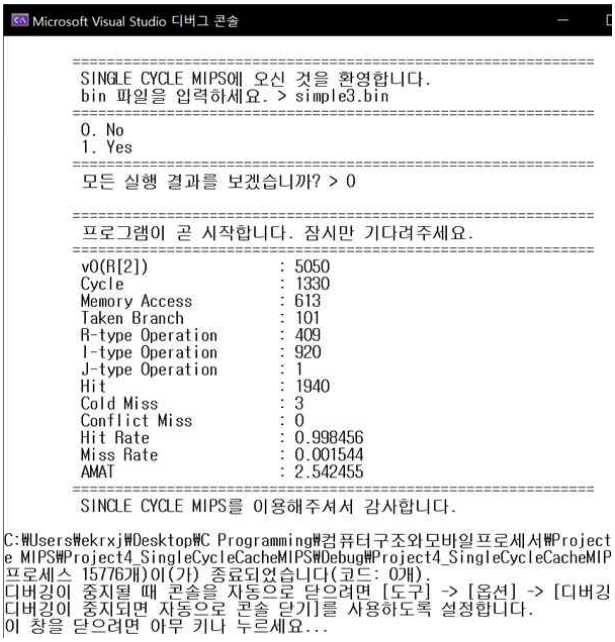
```
Microsoft Visual Studio 디버그 콘솔
==========================================================
SINGLE CYCLE MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > simple2.bin
==========================================================
0. No
1. Yes
==========================================================
모든 실행 결과를 보겠습니까? > 0

==========================================================
프로그램이 곧 시작합니다. 잠시만 기다려주세요.
==========================================================
v0(R[2])              : 100
Cycle                 : 10
Memory Access         : 4
Taken Branch          : 0
R-type Operation      : 3
I-type Operation      : 7
J-type Operation      : 0
Hit                   : 12
Cold Miss             : 2
Conflict Miss         : 0
Hit Rate              : 0.857143
Miss Rate             : 0.142857
AMAT                  : 143.714279
==========================================================
SINCLE CYCLE MIPS를 이용해주셔서 감사합니다.

C:₩Users₩ekrxj₩Desktop₩C Programming₩컴퓨터구조와모바일프로세서₩Project
e MIPS₩Project4_SingleCycleCacheMIPS₩Debug₩Project4_SingleCycleCacheMIP
프로세스 9360개)이(가) 종료되었습니다(코드: 0개).
디버깅이 중지될 때 콘솔을 자동으로 닫으려면 [도구] -> [옵션] -> [디버깅
디버깅이 중지되면 자동으로 콘솔 닫기]를 사용하도록 설정합니다.
이 창을 닫으려면 아무 키나 누르세요...
```

3) simple3.bin

- **Direct Mapping Cache**

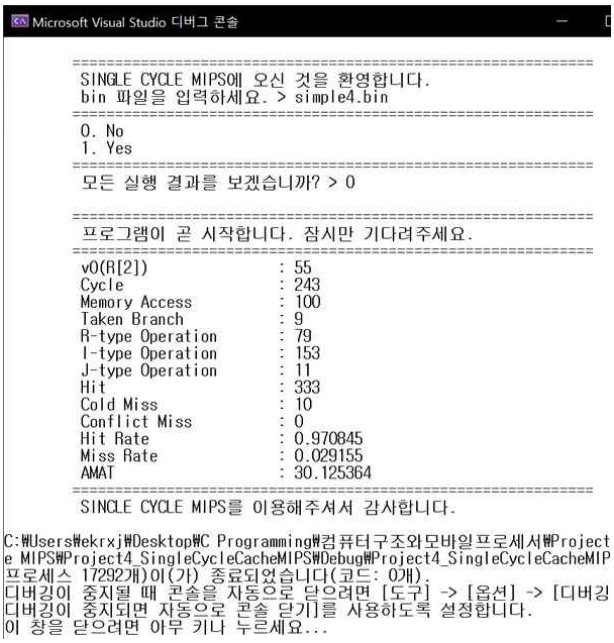


- **4-way Set Associative Cache**

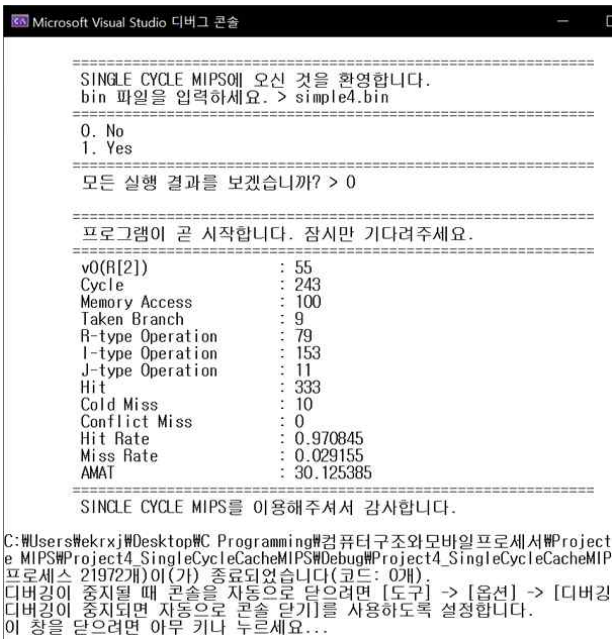


4) simple4.bin

- **Direct Mapping Cache**

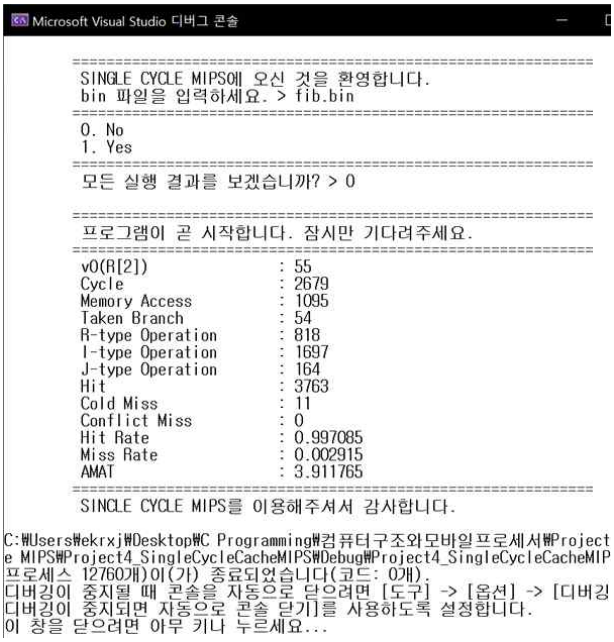


- **4-way Set Associative Cache**

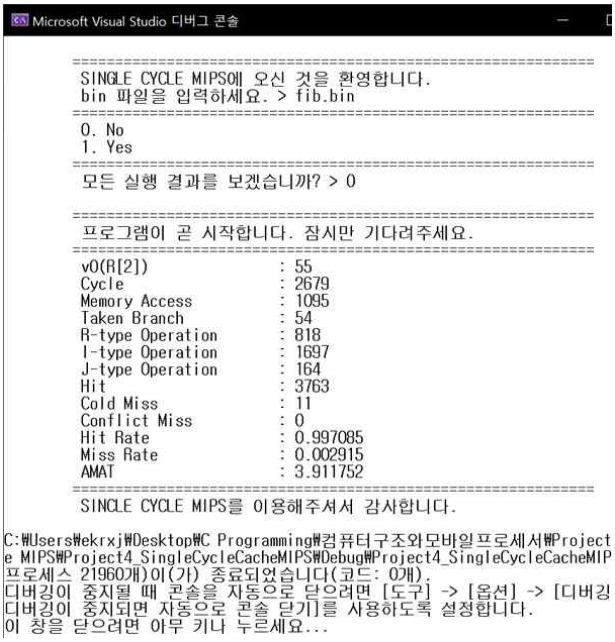


5) fib.bin

- **Direct Mapping Cache**

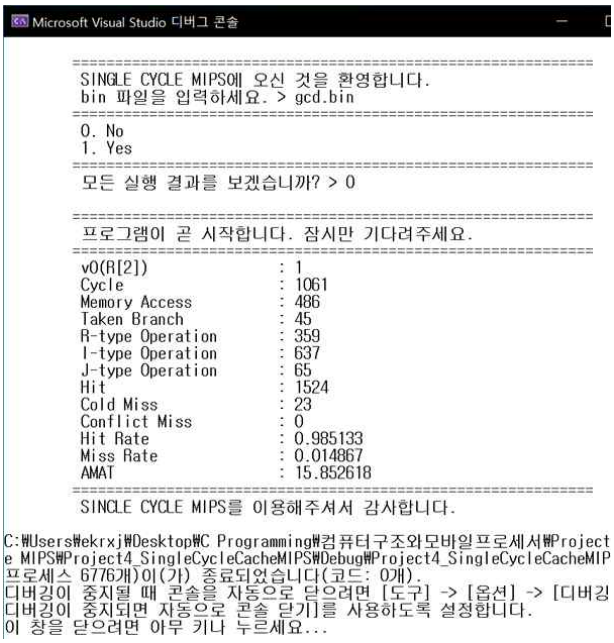


- **4-way Set Associative Cache**

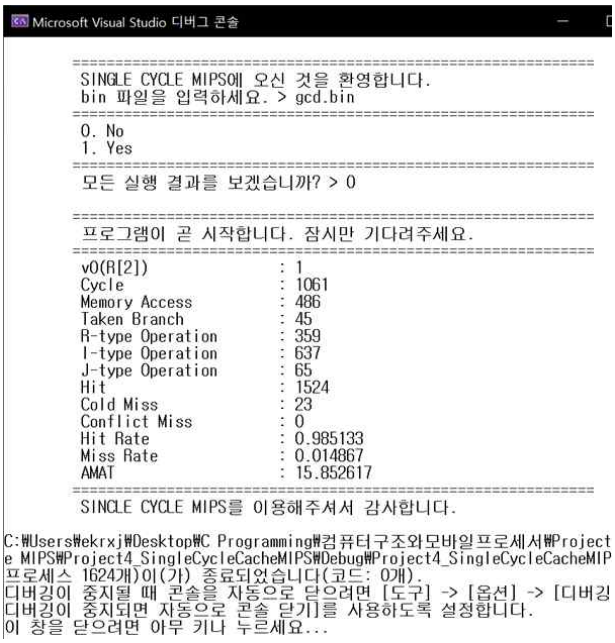


6) gcd.bin

- **Direct Mapping Cache**

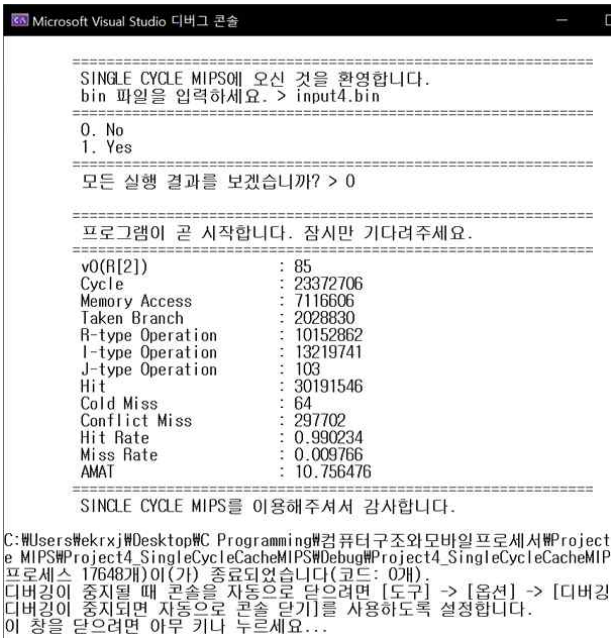


- **4-way Set Associative Cache**

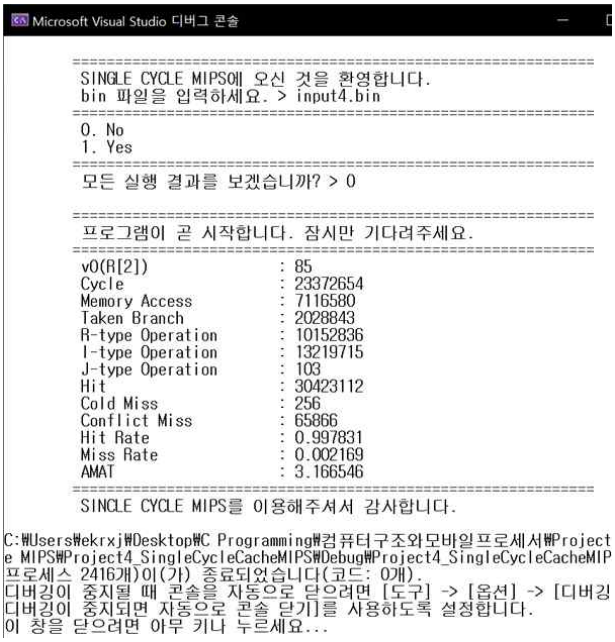


7) input4.bin

- **Direct Mapping Cache**



- **4-way Set Associative Cache**



나. 결과 분석

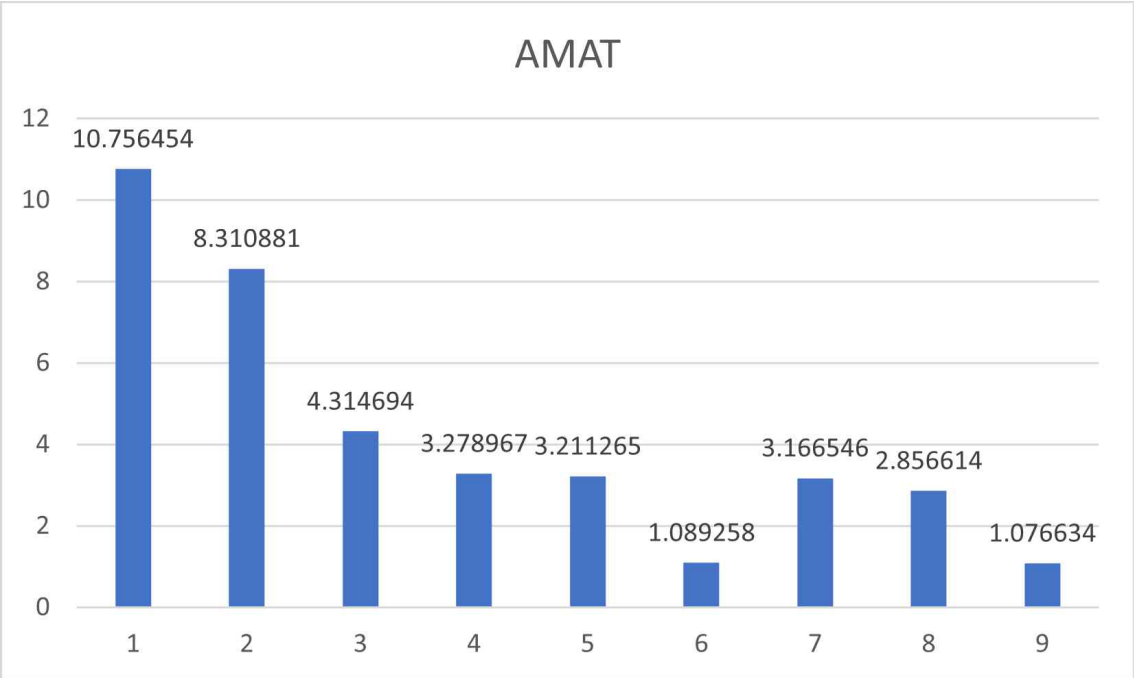
	Direct mapped Cache			2-way set associated Cache			4-way set associated Cache		
	4KB	8KB	32KB	4KB	8KB	32KB	4KB	8KB	32KB
setCacheSize(2^n)	12	13	15	12	13	15	12	13	15
setCachelineSize	6	6	6	6	6	6	6	6	6
tag length	20	19	17	20	19	17	20	19	17
index length	6	7	9	6	7	9	6	7	9
offset length	6	6	6	6	6	6	6	6	6
ADR.tag	0x000f:ffff	0x0007:ffff	0x0001:ffff	0x000f:ffff	0x0007:ffff	0x0001:ffff	0x000f:ffff	0x0007:ffff	0x0001:ffff
ADR.index	0x0000:003f	0x0000:007f	0x0000:01ff	0x0000:003f	0x0000:007f	0x0000:01ff	0x0000:003f	0x0000:007f	0x0000:01ff
v0(R[2])	85	85	85	85	85	85	85	85	85
Cycle	23372706	23372706	23372706	23372730	23372658	23372666	23372654	23372730	23372658
Memory Access	7116606	7116606	7116606	7116618	7116582	7116586	7116580	7116618	7116582
Taken Branch	2028830	2028830	2028830	2028824	2028842	2028840	2028843	2028824	2028842
R-type Operation	10152862	10152862	10152862	10152874	10152838	10152842	10152836	10152874	10152838
I-type Operation	13219741	13219741	13219741	13219753	13219717	13219721	13219715	13219753	13219717
J-type Operation	103	103	103	103	103	103	103	103	103
Hit	30191546	30266185	30388148	30419794	30421752	30486528	30423112	30432685	30486902
Cold Miss	64	128	512	128	256	1024	256	512	2048
Conflict Miss	297702	222999	100652	69426	67232	1700	65866	56151	290
Hit Rate	0.990234	0.992682	0.996682	0.997719	0.997787	0.999911	0.997831	0.998142	0.999923
Miss Rate	0.009766	0.007318	0.003318	0.002281	0.002213	0.000089	0.002169	0.001858	0.000077
AMAT	10.756454	8.310881	4.314694	3.278967	3.211265	1.089258	3.166546	2.856614	1.076634

<표 5.1> 캐시 사상 방식과 캐시 크기에 따른 input4.bin의 기본구현 출력 결과.

<표 5.1>은 직접 사상 캐시, 2-way 집합 연관 캐시 그리고 4-way 집합 연관 캐시가 각각 4KB, 8KB 그리고 32KB의 캐시 크기를 가질 때 input4.bin의 기본구현 출력 결과를 나타낸다. input4.bin의 적중률, 실패율, 평균 메모리 접근 시간을 분석하면 다음과 같은 결과를 도출할 수 있다.

- input4.bin의 결과(v0)는 85로 모두 같다.
- 캐시 크기가 증가할수록 공통적으로 최초 시작 실패는 소폭 증가하지만, 대립 실패는 대폭 감소한다.
- 캐시 크기가 증가할수록 공통적으로 AMAT이 감소한다. 직접 사상 캐시는 10.75 ➜ 8.31 ➜ 4.31, 2-way 집합 연관 캐시는 3.27 ➜ 3.21 ➜ 1.08, 그리고 4-way 집합 연관 캐시는 3.16 ➜ 2.85 ➜ 1.07 순으로 감소한다.
- 캐시 크기가 같을 때, 직접 사상 캐시, 2-way 집합 연관 캐시 그리고 4-way 집합 연관 캐시 순으로 AMAT이 감소한다. 4KB 캐시 크기는 10.75 ➜ 3.27 ➜ 3.16, 8KB 캐시 크기는 8.31 ➜ 3.21 ➜ 2.85, 그리고 32KB 캐시 크기는 4.31 ➜ 1.08 ➜ 1.07 순으로 감소한다.
- 집합 연관 캐시의 집합 n을 증가할수록 공통적으로 AMAT이 감소하나, 그 변화량은 크지 않다.

이 결과를 바탕으로 input4.bin에 한정하여 효율적인 캐시를 정의해보자. 직접 사상 캐시보다 집합 연관 캐시가 최소 1.3배에서 최대 9배까지 효율이 높으므로 집합 연관 캐시를 사용하는 게 더 바람직하다. 또한 캐시 크기를 증가시키는 비용이 크다고 가정하면 32KB 2-way 집합 연관 캐시보다 8KB 4-way 집합 연관 캐시를 사용하는 게 가격 대비 성능이 좋다.



<표 5.2> 직접 사상 캐시(1~3), 2-way 집합 연관 캐시(4~6), 4-way 집합 연관 캐시(7~9)의 캐시 크기별 input4.bin의 평균 메모리 접근 시간(AMAT). 캐시 크기가 증가할수록, n-way 집합 연관 사상으로 갈수록 평균 메모리 접근 시간이 감소하는 경향을 보인다.

6. 고찰 및 느낀 점

이번 프로젝트4를 진행하기에 앞서 중요한 네 가지 단계별 목표를 정했다. 첫 번째 목표는 "캐시 용어를 정확하게 이해하자.", 두 번째 목표는 "캐시의 세 가지 사상 방식을 이해하자.", 세 번째 목표는 "캐시 쓰기 정책을 이해하고 SCA 교체 정책을 도입하자."였다. 첫 번째 목표를 달성하기 위해서 수업 강의 자료와 전공 서적간의 용어를 비교하고 인터넷 검색을 통해 용어의 정의를 정확하게 파악하려고 노력했다. 프로젝트4에서 새롭게 등장한 캐시 용어들은 프로젝트2, 3에서 사용한 익숙한 용어들이 아니었다. 이러한 용어의 정의를 제대로 이해하지 못한 상태로 캐시 쓰기 정책과 교체 정책을 섣불리 구현하는 것은 상당히 큰 혼란을 야기할 것으로 예상되었다. 따라서 강의 자료에서 사용된 용어를 중심으로 전체적인 캐시의 기본을 이해하려고 했고 이것은 캐시 이론의 기초를 튼튼하게 하여 프로그램을 작성하는데 큰 도움이 되었다. 두 번째 목표를 달성하기 위해서 사상 방식에 대해 설명한 전공 서적과 인터넷 자료를 특히 많이 참고했다. 이번 프로젝트는 사상 방식을 정확하게 구현하는 것이 중요한 목표 중 하나였고 사상 방식이 올바르게 구현되어야 이후 쓰기 정책과 교체 정책도 그 위에 올릴 수 있는 형태였다. MIP는 32비트 주소를 사용하지만 처음 이해하기에는 비트 수가 너무 많아 다소 직관적이지 않았고 따라서 8비트 주소를 갖는 간단한 캐시를 먼저 이해해보기로 했다. 종이에 8비트 주소를 갖는 캐시의 기본 구조를 그렸고 사상 방식에 맞춰 기본 구조를 조금씩 바꾸어 나갔다. 기본 구조에 대한 이해를 바탕으로 먼저 4, 8, 32KB 직접 사상 캐시를 큰 어려움 없이 코드로 구현할 수 있었다. 이번 프로젝트4에서는 마지막 목표를 달성하는 게 가장 어려웠다. 캐시 쓰기 정책과 sca 교체 정책은 강의 자료로 충분히 이해할 수 있었으나, 코드로 이것을 구현하기가 난해했다. 특히 집합 연관 사상 캐시의 sca 교체 정책을 구현하는 게 가장 어려웠는데 직접 사상 캐시와 다르게 가장 오래된 집합을 선별할 방법이 쉽게 떠오르지 않았다. 게다가 교체 정책에 해당하는 코드 작성을 마치고 디버그 과정을 거치면서 오류를 수정하고 부실하게 작성된 코드를 보완하는데 시간이 많이 들었다. 다행스럽게도 코드를 나름 견고하게 작성한

덕분에 디버깅을 성공적으로 마쳐 꽤 만족스러운 결과를 얻을 수 있었고 큰 성취감을 느낄 수 있었다. 한편, 마지막 프로젝트4는 여러 다른 수업과 일정이 겹쳐 시간이 많이 부족했다. 수업 시간에 배운 내용을 충분히 생각해볼 시간이 없었고, 여러 가지 조건을 갖는 캐시들을 모두 구현할 만큼 일정이 여유롭지 못했다. 이번 프로젝트4에서 다른 캐시 크기나 캐시 라인 크기를 설정할 수 있도록 하여 AMAT을 다양하게 확인하지 못한 것과, 완전 연관 사상 방식을 구현하지 못한 것이 큰 아쉬움으로 남는다.

이번 컴퓨터구조 및 모바일프로세서의 프로젝트4를 마지막으로 종강을 맞이했다. 이 수업은 그동안의 전공 중에서 가장 많은 과제와 리포트를 받았다고 해도 과언이 아니다. 하지만 덕분에 이 수업을 통해 여러 가지 큰 배움을 얻을 수 있었다. 지금부터는 컴퓨터구조 및 모바일프로세서를 수강하며 스스로 느낀 점에 대해 말하고자 한다. 먼저, 간단한 계산기, 싱글 사이클, 파이프라인, 그리고 캐시까지 전체적인 컴퓨터구조의 핵심을 이해할 수 있었다. 컴퓨터 프로세서의 근본적인 작동 원리를 이해하고 성능을 향상시키기 위해 제시된 여러 메커니즘을 배우는 건 즐거운 일이었다. 처음 컴퓨터가 발명된 이래로 더 성능 좋은 프로세서와 메모리를 설계하기 위한 사람들의 노력과 그 노력이 녹아든 결과를 한 눈에 배울 수 있던 의미 있는 시간이었다. 두 번째, 프로그램 언어 작성에 대한 두려움을 많이 없앴다. 3년의 휴학 이후, 학생으로서 수업에 대한 감각이 많이 떨어져 있었고 기초적인 C언어 코딩 방법도 많이 잊어버린 상태였다. 혹시 내 수준이 너무 낮아 수업을 따라가는데 지장이 있을까 염려도 하였으나, 다행히 교수님의 배려로 프로젝트1 간단한 계산기를 구현하며 감을 잡을 수 있었고 잘 기억나지 않거나 모호했던 C언어 코딩 방법도 다시 복기할 수 있었다. 무엇보다 간단한 계산기를 시작으로 순차적으로 더 많은 체계적인 코드와 복잡한 체계를 요구하는 싱글 사이클, 파이프라인을 C언어로 설계함으로써 막연하게 가지고 있었던 코드에 대한 두려움도 많이 없앨 수 있었다. 세 번째, 글을 쓰는 방법에 대해 배웠다. 이번 수업에서 네 개의 리포트를 작성하는 동안 내가 얻은 결과를 독자가 가장 효과적으로 읽게 할 방법에 대해 끊임없이 고민했다. 전체적인 흐름이 이어지고 오해의 소지가 없으며 누구나 이해하기 쉬운 글을 작성하는 건 정말 어려웠다. 가끔은 특정한 목표가 정해진 코드를 작성하는 것보다 어떻게 표현할지에 대해 더 고뇌하기도 했다. 물론 글쓰기 실력이라는 게 정량적으로 측정할 수 있는 것도 아니고 아직 내 실력이 전문적인 논문에 비할 바도 아니지만, 적어도 이번 수업에 제출한 리포트는 이전 학기의 다른 리포트에서 찾아볼 수 없는 그런 고민이 많이 정말 묻어난다고 생각한다. 마지막으로, 할 수 있다는 자신감을 얻었다. 비록 한 학기 동안 여러 개의 프로젝트를 진행하는 게 몸과 마음이 고된 일이기는 했지만, 돌아보면 전체적으로 꽤 만족스러운 결과를 도출할 수 있었다. 덕분에 의지를 갖추고 실천하면 어려운 일도 곧 해낼 수 있다는 자신감을 얻었다. 가장 많은 시간과 노력을 쏟아 부은 만큼 나에게 큰 자산이 될 수 있어서 기쁘다. 짜임새 있는 수업으로 내외적으로 큰 성장의 기회를 제공해주신 교수님께 감사의 말씀을 전한다.

7. 참고 문헌

David A. Patterson, John L. Hennessy.(1994) 컴퓨터구조 및 설계 하드웨어/소프트웨어 인터페이스(고려대학교 박명순, 하순회, 장훈 옮김). 한티 미디어