
REPORT



Project #04 : Pipelined MIPS Emulator with Cache

과목명	컴퓨터구조및 모바일프로세서	담당교수	유시환 교수님
학 번	32204292	전 공	모바일시스템공학과
이 름	조민혁	제 출 일	2024/06/23

목 차

1. Introduction	1
2. Requirements	2
3. Concepts	3
3-1. Memory Hierarchy	3
3-2. Memory Locality	5
3-2-1. Temporal Locality	6
3-2-2. Spatial Locality	7
3-3. Hierarchical Latency Analysis	8
3-4. Cache	9
3-4-1. Direct Mapped Cache(DMC)	10
3-4-2. Set Associative Cache(SAC)	11
3-4-3. Fully Associative Cache(FAC)	11
3-5. Replacement Policy	12
3-5-1. LRU(Least Recently Used)	13
3-5-2. SCA(Second Chance Algorithm)	13
3-6. Write Policy	14
3-6-1. Write Through	15
3-6-2. Write Back	16
4. Implements	17
4-1. Build Environment	20
4-2. Implement Cache	21
4-2-1. Direct-Mapped-Cache	21
4-3. Implement Replacement Policy	22
4-3-1. SCA(Second Chance Algorithm)	23
4-4. Implement Write Policy	20
4-4-1. Write Through	21
4-4-2. Write Back	22
5. Results	25
5-1. simple.bin	25
5-2. simple2.bin	25
5-3. simple3.bin	26
5-4. simple4.bin	27
5-5. fib.bin	27

5-6. gcd.bin.....	28
6. Analyze Cache.....	28
7. Conclusion & Feelings	30

1. Introduction

이번 프로젝트에서 구현해볼 것은 'Pipelined MIPS Emulator with Cache'이다. 즉, 기존에 구현하였던 파이프라인 구조에 캐시를 구현하여 파이프라인과 캐시를 연결시키는 것이다. 캐시까지 구현을 하여야 현대 컴퓨터 구조의 기본을 완성했다고 말할 수 있다. 컴퓨터의 핵심은 빠르고 정확하게 Input을 처리하여 어떤 Output으로 내놓는 것을 목표로 한다. 컴퓨터의 많은 정보가 메모리에 저장되어 있으며 메모리의 주소를 참조하여 데이터를 가져오기도 하고, 명령어를 가져오기도 한다. 즉, 필요한 수많은 정보들이 메모리에 있다는 소리이다.

그러나 메모리는 기본적으로 CPU와 분리되어 존재한다. 즉, CPU가 어떠한 연산을 하기 위해서는 메모리 접근이 필요한데, 메모리는 앞서 언급한대로 CPU와 분리되어 있기에 이에 접근하는 시간이 걸린다. 프로그램이 커지고, 데이터의 양이 많아질수록 이에 대한 접근 시간은 길어질 것이다. 따라서 우리는 분리되어 있는 곳에서 필요한 정보를 일일이 가져오는 것이 아닌 CPU와 동일한 위치에서 필요한 정보를 가져오는 것을 목표로 하게 된다. 여기서 나온 개념이 'Cache'이다. 우리가 자주 쓰는 것은 책상 위에 두듯 캐시의 아이디어도 그러하다. 자주 참조하는 메모리의 주소와 데이터를 캐시에 미리 가져와 CPU가 메모리까지 접근하는 것이 아닌 캐시에서 바로바로 가져와 사용하는 것이다.

우리는 이러한 캐시를 구현함으로써 컴퓨터 구조의 기본을 완성함과 동시에 캐시의 유용함을 알게 될 것이다.

2. Requirements

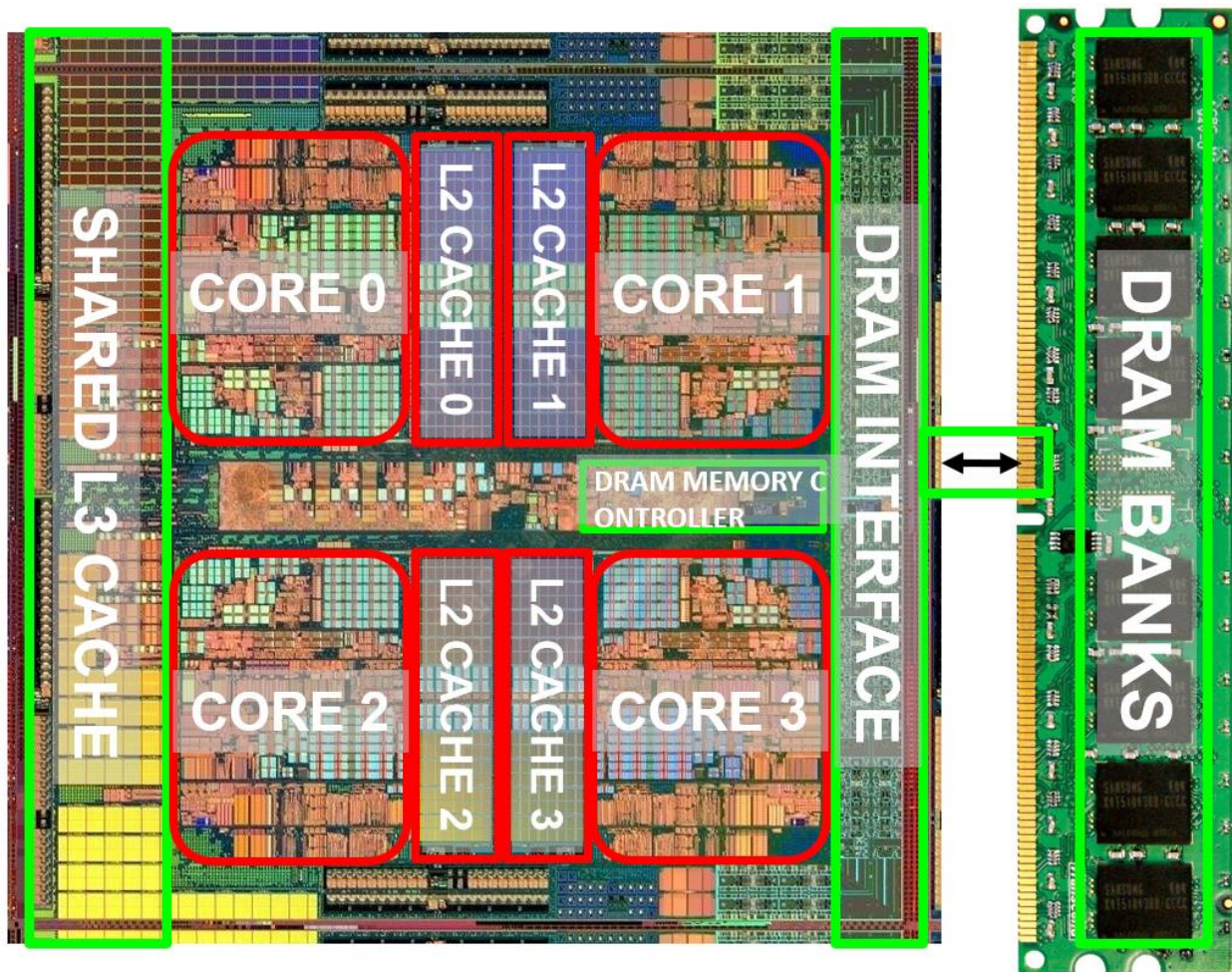
Index	Requirements
1	<p>Cache Structure 를 이해하고 Cache Performance 를 분석할 것</p> <p>A. Program 은 반드시 실행되어 Correct Output 을 출력해야함</p> <p>B. Cache Hit/Miss 와 Average Memory Access Time(AMAT)를 비교할 것</p>
2	<p>Machine Initialization</p> <p>A. Program 실행 전, Binary File 은 Memory 에 load 되어야함</p> <p>B. File Content 를 Memory 에 모두 Read 할 것</p> <p>C. RA Register (R[31]) -> 0xFFFF:FFFF 로 가정 -> PC 가 0xFFFF:FFFF 되면 Machine 은 실행을 끝내고, Program 은 Halt 됨</p> <p>D. Stack Pointer Register (R[29]) -> 0x1000000 으로 가정</p> <p>E. 그 외 Register 값들은 모두 0 으로 초기화할 것</p> <p>F. 결과는 R[2](v0)에 저장</p>
3	<p>Example Assumptions :</p> <p>A. Cache Line Size 는 64 Bytes, Set Associativity 는 Direct-Mapped, 2-way, 4-way, 8-way 등 가능하며 프로그램 실행 전에 정할 것. 또한 Cache Size 는 64, 128, 256 Bytes 가능하며 프로그램 실행전에 정할 것</p> <p>B. Cache 는 적절한 Replacement Algorithm 을 사용할 것</p> <p>C. Cache 는 적절한 Write Policy 를 사용할 것</p> <p>D. Cache Access Latency 는 1 CPU clock cycle time 이며 Memory Access Latency 는 1000 CPU Clock Cycle Time 이다.</p>
4	<p>Prints Out</p> <p>A. 실행한 Cycle 의 총 개수</p> <p>B. Memory Operation(LW/SW)의 개수</p> <p>C. Register Operation 의 개수</p> <p>D. Branch(Total/Taken)의 개수</p> <p>E. Cache Hit/Miss (Cold Miss OR Conflict Miss) 개수</p>

3. Concepts

Cache 를 구현하기 전에 먼저 Memory 의 개념, Cache 의 개념 및 종류, Replacement Policy, Write Policy 에 대한 개념을 알아야한다. 따라서 앞서 언급한 것들에 대한 설명을 진행한 후 구현을 하도록 하겠다.

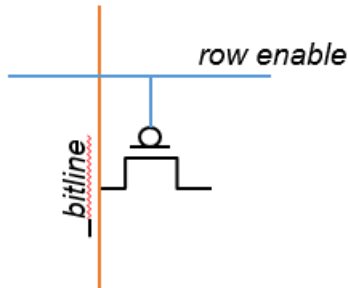
3-1. Memory Hierarchy

위의 사진은 현대 컴퓨터 시스템의 메모리 구조이다. 그림에서 보여지는 것과 같이 컴퓨터는 L1,

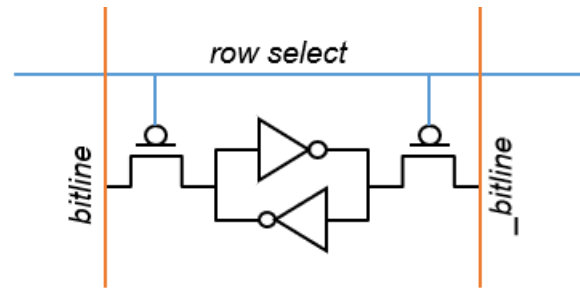


L2, L3 Cache, Core, DRAM 으로 이루어졌다. 메모리는 기본적으로 DRAM 과 SRAM 의 종류가 있는데 메모리 영역에서는 DRAM 이 사용되고 CPU 내의 영역에서는 SRAM 이 사용되는 것이다. 이러한 메모리 계층 구조는 컴퓨터 시스템에서 데이터를 저장하고 액세스하는데 효율적인 장점을 가져다준다. 각 메모리 계층은 크기, 속도, 가격이 모두 상이하며 이상적인 메모리를 생각하면, 크기는 크면서 속도는 빠르고, 가격은 저렴한 메모리를 생각할 것이다. 하지만 그런 이상적인

메모리를 현실적으로 구현하기에는 쉽지 않다. 따라서 3 가지 측면에 대해서 적절한 절충점이 필요한데 그것을 메모리 계층적 구조가 효율성에 대해 기여를 해주는 것이다. 그러면 이제 이러한 메모리를 구성하는 DRAM 과 SRAM 을 살펴보자.



<DRAM>



<SRAM>

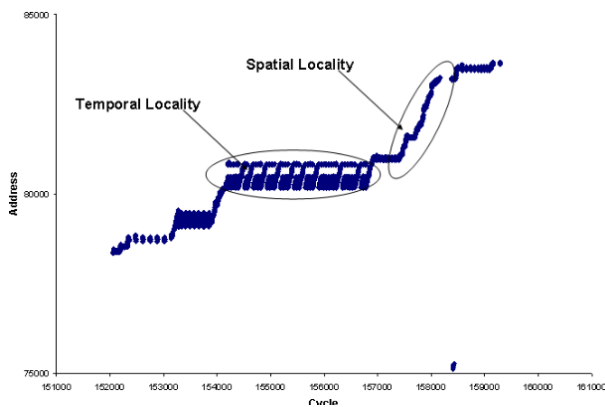
왼쪽 그림은 DRAM에 대한 구조이고, 오른쪽 그림은 SRAM에 대한 구조이다. 먼저 DRAM부터 살펴보면 Dynamic Random Access Memory란 뜻으로 이름에 Dynamic이 존재하는 이유는 DRAM은 데이터를 저장하기 위해 Capacitor를 사용한다. 즉, Capacitor가 Charge 상태일 때는 값이 1인 상태이고, Non-Charge 상태일 때는 값이 0인 상태이다. 그러나 Capacitor의 RC path의 특성에 의해 DRAM은 자연적으로 전하가 손실된다. 이에 따라 전하 손실을 방지하기 위해 주기적으로 Refresh 작업이 필요하다. DRAM은 1개의 Capacitor와 1개의 Access Transistor로 이루어진다. 이에 따라 비교적 작은 크기로 많은 양의 데이터를 저장할 수 있으며, 집적도 또한 높아진다. 따라서 SRAM에 비해 비교적 저렴한 가격에 많은 양의 데이터를 저장할 수 있는 특징이 있다. 그러나 Refresh 시간, Capacitor 충전, 방전 시간에 의해 SRAM에 비해 느린 속도를 가지고 있다.

그러면 SRAM은 어떨까? SRAM은 Static Random Access Memory라는 뜻으로 오른쪽 사진과 같은 구조로 이루어져있다. 그림에서도 보여지듯 저장을 위한 4개의 Transistor, 접근을 위한 2개의 Transistor로 이루어져있는데 Feedback 구조를 통하여 DRAM과는 달리 데이터를 전하 손실없이 저장할 수 있다. 따라서 Refresh 시간이 필요하지 않으며 이에 따라 DRAM에 비해 빠른 속도를 가지게 된다. 그러나 면적이 DRAM보다 크기에 같은 면적에서는 DRAM보다 적은 양의 데이터를 저장하게 된다. 이에 따라 크기가 커질수록 가격이 비싸지는 특징이 있다. 따라서 이러한 SRAM은 대용량 메모리로 사용하기에 적절하지 못하기에 CPU의 캐시 메모리로 사용하며 우리는 SRAM에 대한 구조를 구현할 것이다.

이러한 DRAM과 SRAM의 개념 및 특징을 알아봄으로써 DRAM과 SRAM의 적절한 절충점을 찾음으로써 메모리 계층 구조를 설계할 수 있게 됨을 알 수 있게 되었다.

3-2. Memory Locality

위의 설명에서는 메모리 계층 구조를 알아보았다. 또한 SRAM을 통해 캐시 메모리를 구성하는 것도 알 수 있었다. 그러면 캐시 메모리에는 어떤 데이터를 저장하고, CPU는 메모리의 어느 위치에 접근할까? 현실에서도 사람들이 도서관의 모든 책에 접근하지 않고 특정 관심 분야에만 접근하는 경향이 있다. CPU도 그러하다 메모리의 모든 영역에 접근하지 않으며 특정 메모리 영역에만 자주 접근하는 특성이 있는데 이러한 특징을 Memory Locality라 하며 Temporal Locality, Spatial Locality가 존재한다.



3-2-1. Temporal Locality

Temporal Locality는 시간적 지역성이라는 뜻으로, CPU가 접근한 메모리의 특정 영역을 일정 시간 뒤에 다시 접근한다는 특성이다. 따라서 이러한 특성을 아이디어로 하여 캐시 메모리에 최근에 접근한 데이터를 저장하는 것이다. 이런 특성을 이용하여 캐시 메모리에 저장함으로써 CPU는 메모리에 접근하지 않아도 캐시 메모리에서 자주 사용되는 데이터를 가져올 수 있다. 이에 따라 프로그램의 속도가 향상될 수 있다.

3-2-2. Spatial Locality

Spatial Locality는 공간적 지역성이라는 뜻으로, CPU가 특정 메모리 영역에 접근하며 일정 시간이 흐른 후 그 주변 메모리 영역 또한 접근한다는 특성이다. 프로그래밍에서 배열, 구조체를 떠올릴 수 있으며, 이러한 특성을 아이디어로 캐시 메모리에 접근한 메모리 위치의 주변 메모리 위치를 저장하는 것이다. 이를 통해 캐시 메모리를 Block 단위로 바라볼 수 있으며, 프로그램의 속도 또한 향상될 수 있다.

3-3. Hierarchical Latency Analysis

현대 컴퓨터 구조는 메모리 계층 구조를 사용한다고 언급하였다. 컴퓨터 시스템이 각 계층에 대한 접근 시간과 그에 따른 지연 시간이 발생하는데 그러면 이러한 계층 구조에 대해 성능이 어떠한 지 분석하는 과정이 필요하다.

Hierarchical Latency Analysis

- For a given memory hierarchy level i it has a technology-intrinsic access time of t_i . The perceived access time T_i is longer than t_i .
- Except for the outer-most hierarchy, when looking for a given address there is
 - a chance (hit-rate h_i) you “hit” and access time is t_i
 - a chance (miss-rate m_i) you “miss” and access time $t_i + T_{i+1}$
 - $h_i + m_i = 1$
- Thus

$$T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})$$

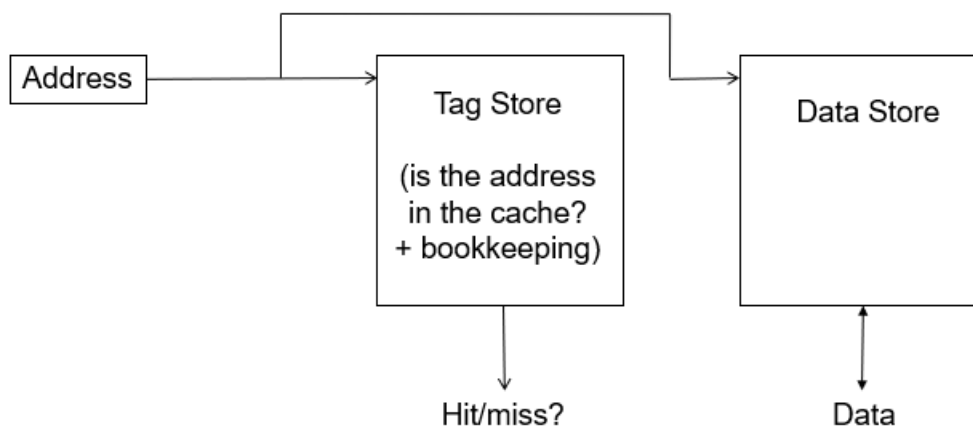
$$T_i = t_i + m_i \cdot T_{i+1}$$

keep in mind, h_i and m_i are defined to be the hit-rate and miss-rate of just the references that missed at L_{i-1}

위의 사진은 메모리 계층 구조의 성능을 수식으로 정리해 놓은 것이다. i 를 메모리 계층의 Level이라고 하고, T 를 계층 접근 시간, h 를 계층에서의 Hit Rate, t 를 계층에 대한 고유 접근 시간, m 을 계층에서의 Miss Rate, T_{i+1} 을 다음 계층의 접근 시간이라고 한다면 계층에 대한 접근 시간 $T = h * t + m * (t + T_{i+1})$ 로 계산된다. 즉, 캐시 메모리에 대해 Hit가 발생하면 계층에 대한 고유 접근 시간만큼이 필요하고, Miss가 발생하면 다음 계층에서 Data를 찾아야하기에 고유 접근 시간에 더해 다음 계층 접근 시간까지 추가로 발생한다. 프로그램에서 발생하는 모든 Hit Rate, Miss Rate를 통해 접근 시간은 위와 같이 정리될 수 있다. 두 번째 수식은 위의 수식을 간소화한 형태이다.

3-4. Cache

앞서 언급한 것들을 통해 메모리의 계층 구조, 메모리 접근 패턴의 특성, 캐시의 필요성, 성능 분석 방법을 알아보았다. 그러면 이제 좀 더 캐시를 자세히 알아볼 필요가 있다. 앞서 언급한 개념들을 종합하여 생각해보면 캐시는 자주 사용되는 데이터를 캐시 메모리에 저장함으로써 반복적으로 메모리에 접근하는 불필요한 지연 시간을 줄이는데 목적이 있다. 그러면 캐시는 어떤 구조로 이루어져 있을까? 아래 그림을 보자



- Cache hit rate = $(\# \text{ hits}) / (\# \text{ hits} + \# \text{ misses}) = (\# \text{ hits}) / (\# \text{ accesses})$
- Average memory access time (AMAT)
$$= (\text{hit-rate} * \text{hit-latency}) + (\text{miss-rate} * \text{miss-latency})$$
- Aside: Can reducing AMAT reduce performance?

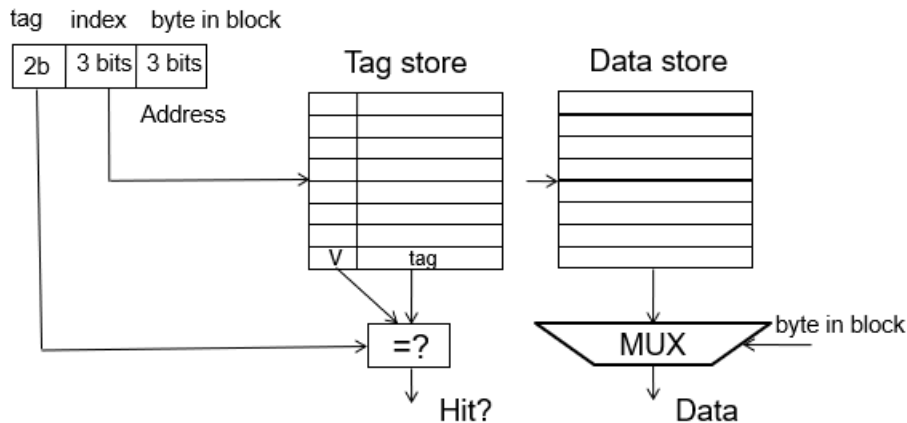
캐시는 기본적으로 CPU로부터 주소를 받는다. 주소를 받아 먼저 캐시에 해당 주소가 존재하는 지 확인을 하는데 이를 Tag에서 확인한다. Tag에 존재하는 경우 Hit이고, 존재하지 않는 경우 Miss가 발생하게 된다. Miss가 발생하는 경우에는 캐시 메모리에 해당 주소에 대한 데이터가 존재하지 않기에 직접 메모리에 접근하거나 다음 계층 구조로 접근하여 데이터를 가져오는 작업이 필요하며, Hit가 발생하는 경우 캐시 메모리의 Data Store에서 해당 주소의 Data를 가져오게 된다.

이에 따라 캐시 메모리에 대한 Hit Rate를 평가할 수 있으며, 캐시 메모리를 사용하였을 때 평균적으로 Memory Access Time이 어떠한 지 평가할 수 있다.

캐시 메모리는 기본적으로 Cache Line(Block)들로 이루어진다. 이에 따라 Cache Line을 식별하여 데이터를 가져오는 작업이 필요한데, 식별하는 과정은 index를 통해 이루어지며, 해당 Cache Line에서

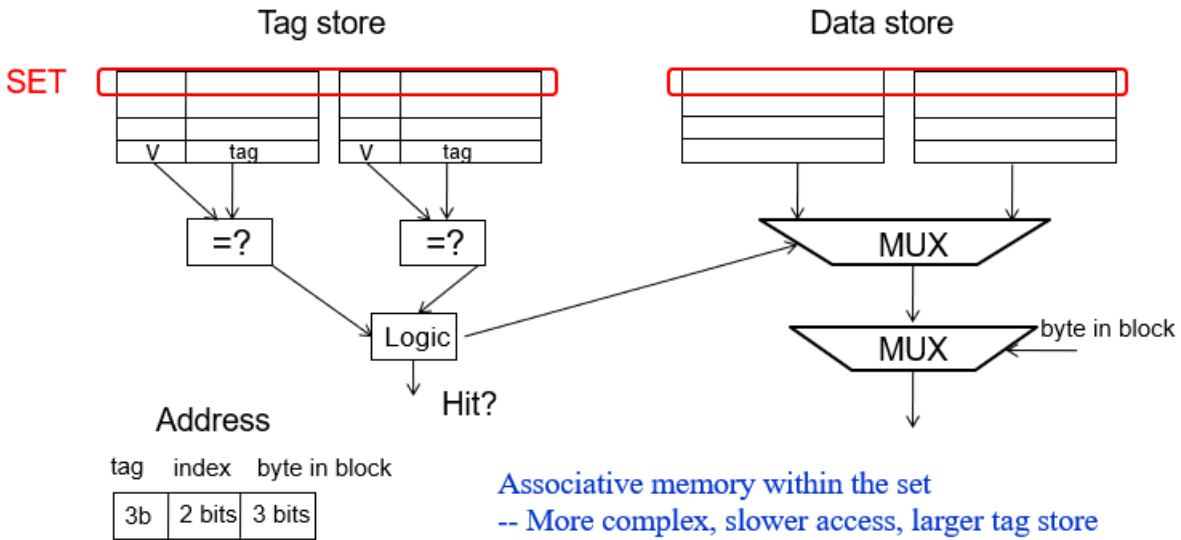
Tag가 존재하는 지를 확인하고 Offset를 통해 Cache Line에서 특정 Data Set을 가져오게된다. 이에 따라 Address는 기본적으로 Tag, Index, Offset으로 분리되며, Cache의 종류에 따라 각 구성 요소들의 Bit 수는 달라지게 된다.

3-4-1. Direct Mapped Cache



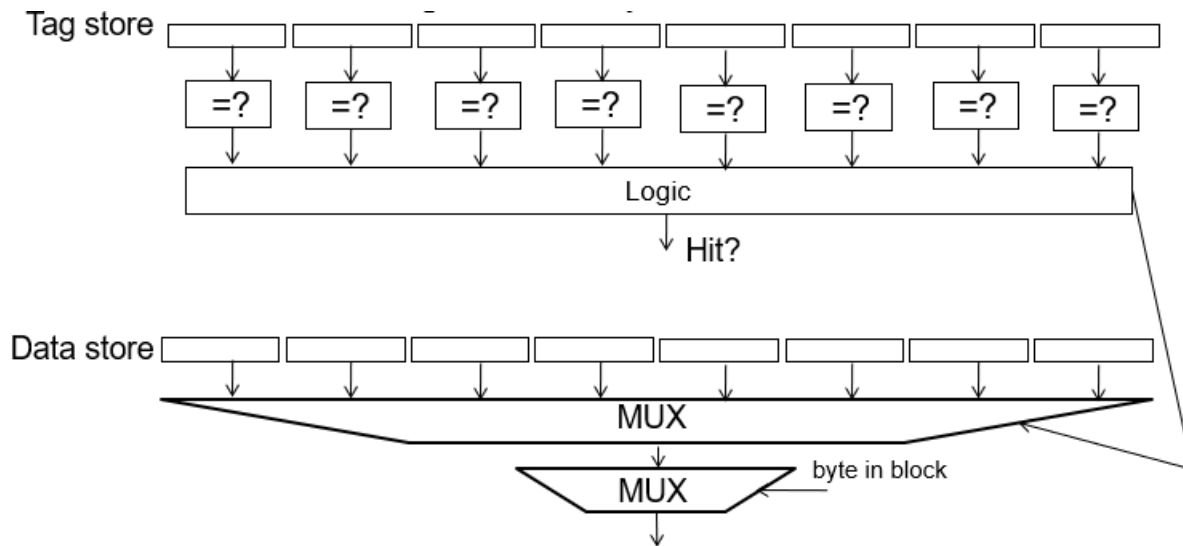
위의 그림은 Direct Mapped Cache의 구조이다. 그림에서 보여지는 것과 같이 Address는 tag, index, offset(byte in block)으로 나뉘게 된다. 그림에서 보여지는 것과 같이 먼저 index를 통하여 Cache Line에 접근해 해당 Tag가 존재하는 지와 Valid한지를 확인한다. Tag가 존재하면 Hit이고, 존재하지 않으면 Miss가 발생한다. Hit가 발생하면 Data Store에서 Cache Line에서 Offset만큼 떨어진 위치에서 데이터를 가져온다. 이것이 Direct Mapped Cache의 기본 동작 방식이다. Direct Mapped Cache는 하나의 Cache Line이 정확하게 메모리 주소에 매핑되는 특징이 있다. 하지만 Conflict Miss가 발생하면 최악의 경우 hit율이 0%가 될 수 있다. 예를 들어 주소 A,B가 존재한다고 하였을 때 A와 B가 같은 index를 가지면 A가 먼저 해당 index의 Cache Line에 접근하였을 때 A의 데이터를 가져올 것이다. 이후 B가 해당 index의 Cache Line에 접근하면 tag가 맞지 않기에 메모리에서 Data를 가져올 것이고 다시 A가 접근하면 B가 했던 행동을 할 것이다. 이에 따라 최악의 경우 Hit Rate가 0%가 될 수 있다. 이에 따라 다른 종류의 Cache가 필요하다.

3-4-2. Set Associative Cache



위의 그림은 Set Associative Cache의 구조이다. 기존의 Direct Associative Cache에서 하나의 Tag Store, Data Store가 여러 개로 나뉜 특징이 있으며, 필요에 따라 Set Associative Cache는 N개로 나눌 수 있기에 N-Way Cache 라고도 불리며, Direct Associative Cache는 1-Way Cache라고 불리는 것이다. 즉, 하나의 Tag Store와 Data Store를 여러 개의 세트로 나눔으로써 같은 index를 가지더라도 각 세트에 따라 다른 Tag 값이 들어갈 수 있기에 충돌 효과를 줄일 수 있는 것이다. 그러나 각 세트를 확인해야하므로 매핑 시간이 증가하여 전체적인 메모리 접근 시간이 증가할 수 있다.

3-4-3. Fully Associative Cache



위의 그림은 Fully Associative Cache의 구조이다. Fully라는 말에서 알 수 있 듯 Fully Associative Cache는 N-Way Associative에서 N이 매우 큰 구조이다. 즉, 모든 Cache Line이 메모리의 주소값에 모두 매핑될 수 있다. 즉, 매핑 제한이 없기에 인덱스 또한 필요가 없다. 또한 Conflict Miss가 가장 적게 발생하는 구조이다. 그러나, 하나의 Tag값을 찾기 위해 모든 Cache Line을 찾아야 하기 때문에 시간이 가장 오래 걸릴 수 있으며 구현 또한 가장 어려울 수 있다.

3-5. Replacement Policy

만약 어떤 주소에 대해 Tag 값을 찾으려 할 때 모든 Cache Line에 해당 Tag 값이 존재하지 않을 경우 메모리에서 직접 해당 주소에 대한 데이터를 가져와야한다. 또한 이 과정에서 캐시에 가져온 데이터에 대한 저장에 필요한데 캐시 메모리의 Cache Line이 모두 차지 되어 있는 경우에 Cache Line에 대한 교체가 필요한데 아무거나 교체해서는 안되고, 적절한 교체가 이루어져야 한다. 그에 대한 해결책을 제시해주는 것이 Replacement Policy이며 대표적인 알고리즘으로는 LRU(Least Recently Used)와 SCA(Second Chance Algorithm)이다.

3-5-1. LRU(Least Recently Used)

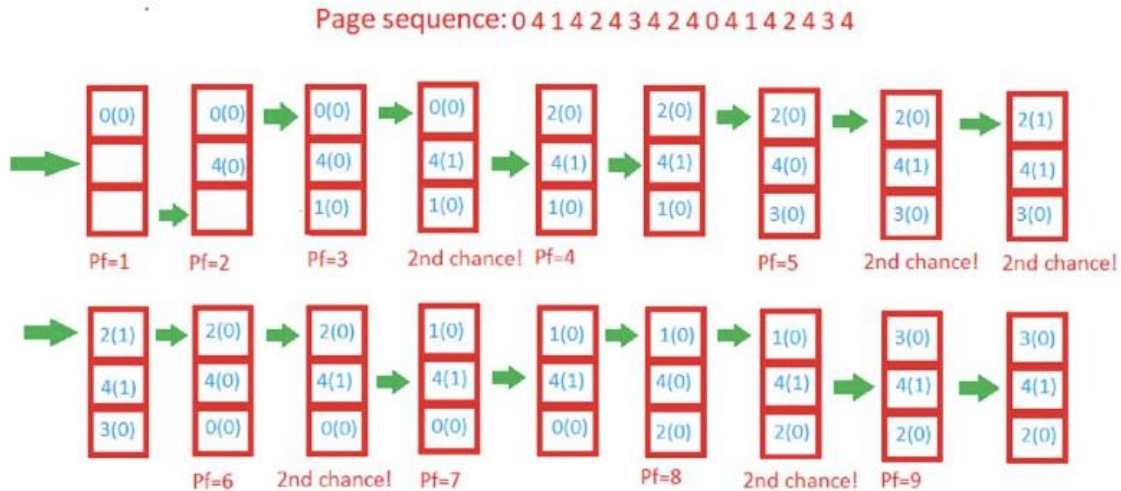


LRU Algorithm은 캐시 메모리에서 가장 오랫동안 사용되지 않은 Cache Line을 교체하는 방식이다. 위에서 Memory Locality에서 알 수 있었듯 컴퓨터 시스템은 최근에 사용한 데이터나 그 주변 데이터를 자주 사용하는 경향이 있다. 따라서 가장 오랫동안 사용되지 않은 데이터가 캐시 메모리에 존재한다면 이에 해당하는 Cache Line을 교체해주는 게 최선의 방법일 수 있다.

위의 그림에서 보여지듯 처음에 캐시 메모리에 3, 1, 2 순서대로 데이터가 채워지고 6에서 Cache Miss가 발생하게 된다. 이에 따라 Cache Line에서 교체되는데 3이 가장 오랫동안 사용되지 않았으므로 교체해주게 된다. 그 후 들어오는 데이터에 대해서도 같은 방식으로 진행해준다.

그러나 이러한 LRU Algorithm은 구현적인 측면에서 어려울 수 있다. 따라서 이에 따른 완화 정책이 등장하는데 그것이 바로 SCA(Second Chance Algorithm)이다.

3-5-2. SCA(Second Chance Algorithm)



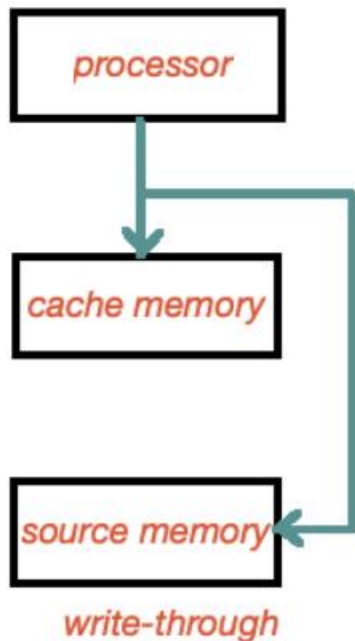
Second Chance Algorithm은 이름에서 알 수 있듯 두 번의 기회를 준다는 것인데, 만약 Conflict Miss가 발생하면 Cache Line 교체가 필요하다. 이 때 가장 오래된 것을 교체하는 것이 아닌 sca_bits 라는 것을 추가 정보 비트로 두어서 한번도 접근되지 않은 Cache Line을 찾는다. 만약 특정 Cache Line에 접근이 한번이라도 있었을 경우 sca_bits는 1로 설정된다. 이후 새로운 Data가 들어오려할 때 가장 참조된지 오래된 Data가 sca_bits가 1이라면 해당 Cache Line을 바로 교체하는 것이 아닌 한 번의 기회를 주는 것이다. 이후 sca_bits는 0으로 state가 변화되고, 이후 다시 해당 Cache Line을 검사한다면 해당 Cache Line이 교체되는 구조이고, 그 전에 sca_bits가 0인 Cache Line을 만났다면 그 Cache Line이 교체된다.

위의 그림에서 알 수 있는 처음 0, 4, 1, 3개의 데이터가 캐시 메모리로 들어오게되고 이후 4가 들어오려 할 때 캐시 메모리에 해당 데이터가 존재하므로 Hit가 발생하여 sca_bits는 1로 설정되고 포인터가 해당 Cache Line을 가르키고 있는 상태에서 새로운 데이터가 접근하면 sca_bits를 0으로 설정해 주면서 다음 Cache Line으로 넘어가게 된다. 이것이 Second Chance Algorithm이다.

3-6. Write Policy

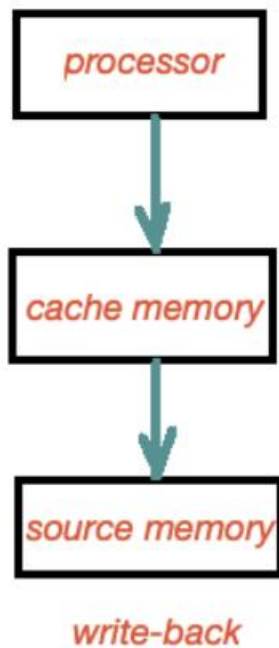
Write Policy는 이름 그대로 쓰기 정책이다. 만약 SW와 같은 메모리에 접근하여 데이터를 써야할 경우 캐시 메모리에 먼저 데이터를 쓰게된다. 이 때 데이터가 캐시 메모리에 쓰여지면 캐시 메모리의 Data와 메모리의 Data가 달라지게 된다. 이러면 캐시 메모리와 메모리의 일관성이 깨지기 때문에 이에 따른 적절한 해결책이 필요하다. 이 때 등장한 개념이 Write Policy이며 Write Policy에는 Write Through, Write Back 두가지 방식이 존재한다.

3-6-1. Write Through



Write Through 방식은 SW와 같은 메모리의 쓰기 작업이 발생할 경우 Cache Memory에 데이터를 업데이트 해주고, 메모리에도 데이터를 업데이트를 해주면서 일관성을 유지해주는 것이다. 이러한 방식은 구현이 가장 간단하며 개념 자체도 쉽지만, SW와 같은 메모리 쓰기 작업이 발생할 때마다 메모리에 접근 해야하는 과정이 필요하다. 이에 따라 지연 시간이 증가하기에 프로그램의 속도가 저하될 수 있다는 단점이 있다.

3-6-2. Write Back



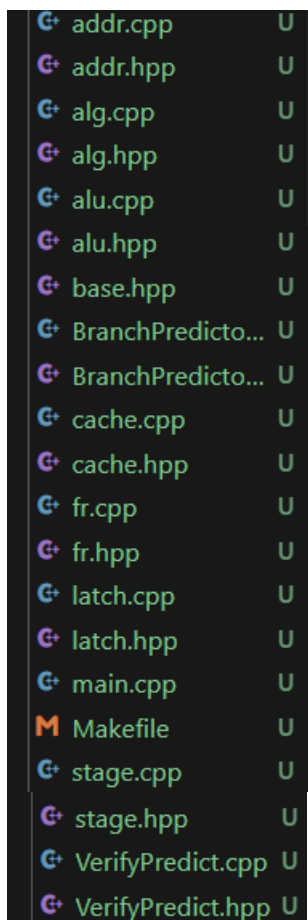
Write Back 방식은 Write Through와는 다르게 SW와 같은 메모리 쓰기 작업이 발생했을 때 바로 캐시 메모리에는 바로 데이터를 업데이트 해주지만 메모리에는 바로 데이터를 업데이트 해주지 않는다. 메모리의 데이터 업데이트 시기는 캐시 메모리의 Cache Line이 교체될 때 메모리의 값을 업데이트 해주는 것이다. 이에 따라 SW와 같은 메모리 쓰기 작업이 발생할 때마다 메모리에 접근하는 것이 아닌 Cache Line 교체할 때만 메모리에 접근하여 데이터를 업데이트 해주므로 지연 시간이 Write Through에 비해 줄어들 수 있다. 그러나 Write Through에 비해 구현 과정에 있어서는 어려울 수 있다.

4. Implements

4-1. Build Environments

먼저 구현한 코드 설명에 앞서 해당 프로그램을 개발한 IDE, 프로그램을 실행하기 위한 Build 방식을 설명하겠다.

- ✓ 개발 환경 : Ubuntu 22.04, VSCode
- ✓ 프로그래밍 언어 : C++
- ✓ 프로그램 구성 :



The screenshot shows a file explorer with the following files and folders:

- addr.cpp U
- addr.hpp U
- alg.cpp U
- alg.hpp U
- alu.cpp U
- alu.hpp U
- base.hpp U
- BranchPredicto... U
- BranchPredicto... U
- cache.cpp U
- cache.hpp U
- fr.cpp U
- fr.hpp U
- latch.cpp U
- latch.hpp U
- main.cpp U
- Makefile U
- stage.cpp U
- stage.hpp U
- VerifyPredict.cpp U
- VerifyPredict.hpp U

✓ 프로그램 실행 방식 : Makefile 이용 (아래 사진 참고)

- 위에서부터 순차적으로 터미널에 입력

```
make main
```

```
./main
```

4-2. Implement Cache

이번 프로젝트에서 Cache를 구현할 때 Direct Mapped Cache, Set Associative Cache, Fully Associative Cache 중에 Direct Mapped Cache를 이용하여 구현하였다.

4-2-1. Direct Mapped Cache

```
1  #ifndef _CACHE_HPP_
2  #define _CACHE_HPP_
3  #include <cstdint>
4
5  /* Direct Mapped Cache, 64Byte*/
6  class DMC {
7  private:
8      uint32_t DMC_tag;
9      uint32_t sca;
10     uint32_t valid = 0;
11     uint32_t dirty;
12     int data[16];
13 public:
14     DMC();
15     void DMC_Init();
16     void Fill_DMC(uint32_t tag_, uint32_t sca_, uint32_t valid_, uint32_t dirty_, uint32_t addr_);
17     uint32_t DMC_Read(uint32_t addr_);
18
19     int DMC_Write_Through(uint32_t addr_, int data_);
20     int DMC_Write_Back(uint32_t addr_, int data_);
21
22     uint32_t get_sca() const;
23
24     void set_tag(uint32_t tag_);
25     void set_sca(uint32_t sca_);
26     void set_valid(uint32_t valid_);
27     void set_dirty(uint32_t dirty_);
28     void set_data(uint32_t addr_);
29 };
30
31
32 #endif
```

위의 사진은 Direct Mapped Cache의 헤더파일 코드이다. 클래스를 이용하여 Direct Mapped Cache를 구현하였으며 구성 요소로 tag, sca, valid, dirty bits를 두었으며 64 Byte라는 가정하에 int가 4 Byte Data Type이므로 16개의 Data 배열을 만들어주었다. 또한 해당 Cache Line에서 Read와 Write가 이루어지도록 적절한 함수를 두어 해당 기능이 동작하도록 하였다.

```
/* Decompose Direct Mapped Cache */
tag = (EXMEM_Latch[1].get_addr() >> 13) & 0x0007ffff;
idx = (EXMEM_Latch[1].get_addr() >> 6) & 0x0000007f;
offset = EXMEM_Latch[1].get_addr() & 0x0000003f;
```

위의 사진은 Direct Mapped Cache에 접근할 때 Address를 tag, idx, offset으로 분리하는 과정을 표현한 것이다.

```

void DMC_::Fill_DMC(uint32_t tag_, uint32_t sca_, uint32_t valid_, uint32_t dirty_, uint32_t addr_){
    DMC_tag = tag_;
    sca = sca_;
    valid = valid_;
    dirty = dirty_;
    addr_ = addr_ & 0xffffffc;
    for(int i = 0; i<16; i++){
        data[i] = memory[addr_ / 4 + i];
    }
}

```

위의 사진은 Direct Mapped Cache가 비어있는 상태 즉, Cold Miss가 발생했을 경우 앞서 언급한 사진에서 분리된 주소를 바탕으로 채워지는 과정이다. 또한 64 Byte 시스템에서 하위 6 bits는 반복되어 사용되므로 주소를 0xffff:fffc로 Bit AND 연산을 통해 memory로부터 적절한 값을 받아오도록 구현하였다.

```

uint32_t DMC_::DMC_Read(uint32_t addr_){
    if(valid == 0){
        cache_miss++; // Cold Miss
        Fill_DMC(tag, 0, 1, 0, addr_);
        return data[offset / 4];
    }
    else{
        if(DMC_tag == tag){ // valid = 1 & hit
            cache_hit++;
            set_sca(1);
            return data[offset / 4];
        }
        else{ // Conflict Miss
            cache_miss++;
            return MISS;
        }
    }
}

```

위의 사진은 LW와 같은 명령어를 처리할 때 Direct Mapped Cache에서 Data를 Read하는 과정을 담은 것이다. 먼저, valid가 0이면 유효하지 않은 Cache Line이라 판단하여 Cold Miss가 발생했다고 판단하였다. 따라서 이에 따라 miss의 count를 증가시켜주고 그에 따른 Cache를 채워주는 작업을 진행하였다. 이후 Cache Line을 채운 뒤에 offset만큼의 Data를 Return 하도록 하였다.

만약 valid가 1이고, 입력받은 tag와 Direct Mapped Cache의 tag값이 일치하다면 Cache Line에 적절한 값이 존재하기에 Cache Hit가 발생했다고 판단하였다. 이후, hit count를 증가시켜주고 Cache Line에 접근하였기에 sca를 1로 설정해주었고, offset만큼의 Data를 Return 해주었다.

만약 valid가 1이지만 tag값이 일치하지 않다면 Conflict Miss가 발생했다고 판단하여 miss의 count를 증가시켜주고 Miss를 Return하여 Return된 곳에서 적절한 처리가 일어나도록 하였다.

```

if(EXMEM_Latch[1].MEM.get_MemRead()){ // lw
    int res = 0;
    res = DMC[idx].DMC_Read(EXMEM_Latch[1].get_addr());

    if(res == MISS){ // miss 발생
        int idx_ = SCA(DMC);
        DMC[idx_].set_tag(tag);
        DMC[idx_].set_valid(1);
        DMC[idx_].set_dirty(0);
        DMC[idx_].set_data(EXMEM_Latch[1].get_addr());
        DMC[idx_].set_sca(0);

        MEMWB_Latch[0].mem_data = memory[EXMEM_Latch[1].get_addr() / 4];
    }
    else{
        MEMWB_Latch[0].mem_data = res;
    }
    memory_access_count++;
}

```

위의 사진은 MEM Stage에서 Cache에 접근하는 과정을 표현한 것이다. 만약 LW 명령어가 들어왔을 경우 Cache에서 일차적으로 값을 읽어와 하기 때문에 먼저 Cache에서 Read해온다. 만약 res에 적절한 값이 반환되면 바로 Latch에 저장해주어 WB Stage로 넘어가면 되지만 앞서 언급한 Conflict Miss가 발생한 경우 적절한 처리가 필요하다. 후에 서술한 SCA Algorithm을 통해 먼저 교체해야할 Cache Line들의 index를 가져오고 가져온 index에 교체될 값들을 입력해준다. 이후 memory에서 값을 가져와 Latch에 값을 넣어주는 작업을 통해 Conflict Miss를 해결해주었다.

4-3. Implement Replacement Policy

이번 프로젝트에서 사용한 Replacement Policy는 Second Chance Algorithm(SCA)이다. 아래 설명을 통해 SCA 알고리즘을 어떻게 구현하였는가를 서술하도록 하겠다.

4-3-1. SCA(Second Chance Algorithm)

```
uint32_t SCA(DMC_ * DMC){
    int ret = -2;
    while(1){
        for(int i = 0; i<128; i++){
            if(DMC[i].get_sca() == 0){
                ret = i;
                break;
            }
            else{
                DMC[i].set_sca(0);
            }
        }
        if(ret != -2) break;
    }
    return ret;
}
```

위의 사진은 구현한 SCA Algorithm이다. 먼저 Direct Mapped Cache가 128 크기로 이루어졌기 때문에 해당 Direct Mapped Cache를 0번부터 시작하여 127번까지 검사하도록하고 만약 sca_bits가 0인 index를 발견하였다면 ret에 해당 index를 담아두고 break를 하고 ret의 값 검사를 통해 while문을 빠져나가도록 하며 해당 값을 return 해주도록 하였다. 이렇게 Return된 값들은 위에서 설명한 LW 명령어의 Conflict Miss를 해결할 수 있으며 아래에서 설명할 Write Policy의 SW 명령어에 대한 Conflict Miss를 해결할 수 있다.

4-4. Implement Write Policy

이번 프로젝트에서 Write Policy는 Write Through 방식과 Write Back 방식으로 구현하였다. 아래 사진에서 어떻게 구현 하였는지 설명하도록 하겠다.

4-4-1. Write Through

```
int DMC_::DMC_Write_Through(uint32_t addr_, int data_){
    /* Write through */

    if(valid == 0){
        cache_miss++;
        Fill_DMC(tag, 0, 1, 0, addr_);
        data[offset / 4] = data_;
        memory[addr_ / 4] = data_;
        set_dirty(1);
        return 1;
    }
    else{
        if(DMC_tag == tag){ // valid & hit
            cache_hit++;
            set_sca(1);
            data[offset / 4] = data_;
            memory[addr_ / 4] = data_;
            set_dirty(1);
            return 1;
        }
        else{
            cache_miss++;
            return MISS;
        }
    }
}
```

위의 사진은 Write Through를 구현한 코드이다. Read할 때와 마찬가지로 먼저 valid 검사를 통해 Cold Miss가 발생 하였는지를 판단한다. 만약 Cold Miss가 발생했다면 Cache Line을 채워주고 Write할 Data를 Cache에 Write하고 Memory에 역시 해당 Data를 Write를 해준다. 그리고, Hit가 발생했다면 sca_bits를 1로 설정해주고 Cache Line의 Data와 Memory에 Data를 적어주는 작업을 하고 dirty_bits를 1로 설정해준다. 마지막으로 Conflict Miss가 발생했다면 Miss를 Return해줌으로써 Conflict Miss에 대한 적절한 처리가 이루어지도록 해준다.

```

else if(EXMEM_Latch[1].MEM.get_MemWrite()){ // SW
    if(DMC[idx].DMC_Write_Through(EXMEM_Latch[1].get_addr(), EXMEM_Latch[1].get_write_data()) == MISS){
        int idx_ = SCA(DMC);
        DMC[idx_].set_tag(tag);
        DMC[idx_].set_valid(1);
        DMC[idx_].set_dirty(0);
        DMC[idx_].set_data(EXMEM_Latch[1].get_addr());
        DMC[idx_].set_sca(0);
        DMC[idx_].DMC_Write_Through(EXMEM_Latch[1].get_addr(), EXMEM_Latch[1].get_write_data());
    }
    else{
        cout<<"CACHE SW OK"<<endl;
    }
    memory_access_count++;
}
}

```

위의 사진은 Conflict Miss가 발생했을 경우 처리해주는 과정이다. 역시 SCA Algorithm을 통하여 먼저 교체할 index를 가져와주고 교체 작업을 진행해준다. 교체가 완료되었을 경우 다시 하면 교체된 Index에 대해 Write Through 함수를 실행해줌으로써 SW에 대한 데이터 쓰기 작업이 이루어지도록 하였다.

4-4-2. Write Back

```

int DMC_::DMC_Write_Back(uint32_t addr_, int data_){
    /* Write Back */

    if(valid == 0){
        cache_miss++;
        Fill_DMC(tag, 0, 1, 0, addr_);
        data[offset / 4] = data_;
        set_dirty(1);
        return 2;
    }
    else{
        if(DMC_tag == tag){ // valid & hit
            cache_hit++;
            set_sca(1);
            data[offset / 4] = data_;
            set_dirty(1);
            return 2;
        }
        else{
            cache_miss++;
            return MISS;
        }
    }
}
}

```

마지막으로 Write Back에 대한 구현이다. Write Back이라고 Write Through와 큰 차이는 없이 구현하였다. 하지만 차이점이라고 한다면 Write Back에서의 Data 쓰기 작업에서 Memory에도 동시에 쓰기 작업이 이루어지지 않는다는 것이다. 따라서 Cache Line의 Data영역에만 값을 써주고 함수를 종료하고 Conflict Miss가 발생하는 경우 적절한 작업을 통해 해당 값을 Memory에 값을 써주는 작업을 진행하였다.

```
/* Write Back Policy */
if(DMC[idx].DMC_Write_Back(EXMEM_Latch[1].get_addr(), EXMEM_Latch[1].get_write_data()) == MISS){
    int idx_ = SCA(DMC);
    DMC[idx_].set_tag(tag);
    DMC[idx_].set_valid(1);
    DMC[idx_].set_dirty(0);
    DMC[idx_].set_data(EXMEM_Latch[1].get_addr());
    DMC[idx_].set_sca(0);
    memory[EXMEM_Latch[1].get_addr() / 4] = EXMEM_Latch[1].get_write_data();
}
else{
    cout<<"CACHE SW OK"<<endl;
}
```

위의 사진은 Conflict Miss가 발생할 경우 Write Back 방식으로 처리해주는 과정이다. 역시나 SCA Algorithm을 통해 교체할 index를 찾아준다. 그 이후 써야할 Data를 Memory에 적어줌으로써 Write Back 방식이 이루어지도록 하였다.

이로써 Cache를 구현하는데 있어 모든 코드 설명을 마쳤다.

5. Results

5-1. Simple.bin

```
=====PROGRAM RESULT=====
Total Cycle Count of Program : 10
The Result is R[2] : 0
Total Instruction Count : 9
R-Type Instruction Count : 5
I-Type Instruction Count : 4
J-Type Instruction Count : 0
Jump Count : 1
Branch Count : 0
Memory Access Count : 0
Prediction Count : 0
Predict Count : 0
Mispredict Count : 0
Total Cache Access Count : 2
Cache Hit Count : 1
Cache Miss Count : 1
Cache Hit Rate : 50
AMAT : 1010 ns
=====
```

5-2. Simple2.bin

```

=====PROGRAM RESULT=====
Total Cycle Count of Program : 12
The Result is R[2] : 100
Total Instruction Count : 11
R-Type Instruction Count : 4
I-Type Instruction Count : 7
J-Type Instruction Count : 0
Jump Count : 1
Branch Count : 0
Memory Access Count : 0
Prediction Count : 0
Predict Count : 0
Mispredict Count : 0
Total Cache Access Count : 4
Cache Hit Count : 3
Cache Miss Count : 1
Cache Hit Rate : 75
AMAT : 1030 ns
=====

```

5-3. Simple3.bin

```

=====PROGRAM RESULT=====
Total Cycle Count of Program : 1535
The Result is R[2] : 5050
Total Instruction Count : 1534
R-Type Instruction Count : 512
I-Type Instruction Count : 920
J-Type Instruction Count : 1
Jump Count : 2
Branch Count : 101
Memory Access Count : 0
Prediction Count : 102
Predict Count : 1
Mispredict Count : 101
Total Cache Access Count : 613
Cache Hit Count : 612
Cache Miss Count : 1
Cache Hit Rate : 99.8369
AMAT : 7120 ns
=====

```

5-4. Simple4.bin

```
=====PROGRAM RESULT=====
Total Cycle Count of Program : 294
The Result is R[2] : 55
Total Instruction Count : 293
R-Type Instruction Count : 120
I-Type Instruction Count : 153
J-Type Instruction Count : 11
Jump Count : 22
Branch Count : 9
Memory Access Count : 1
Prediction Count : 10
Predict Count : 1
Mispredict Count : 9
Total Cache Access Count : 100
Cache Hit Count : 93
Cache Miss Count : 7
Cache Hit Rate : 93
AMAT : 7930 ns
=====
```

5-5. fib.bin

```
=====PROGRAM RESULT=====
Total Cycle Count of Program : 3171
The Result is R[2] : 55
Total Instruction Count : 3170
R-Type Instruction Count : 1255
I-Type Instruction Count : 1697
J-Type Instruction Count : 164
Jump Count : 274
Branch Count : 54
Memory Access Count : 1
Prediction Count : 109
Predict Count : 55
Mispredict Count : 54
Total Cache Access Count : 1095
Cache Hit Count : 1088
Cache Miss Count : 7
Cache Hit Rate : 99.3607
AMAT : 17880 ns
=====
```

5-6. gcd.bin

```

=====PROGRAM RESULT=====
Total Cycle Count of Program : 3577
The Result is R[2] : 7
Total Instruction Count : 3576
R-Type Instruction Count : 1469
I-Type Instruction Count : 1793
J-Type Instruction Count : 106
Jump Count : 212
Branch Count : 208
Memory Access Count : 1
Prediction Count : 209
Predict Count : 1
Mispredict Count : 208
Total Cache Access Count : 1370
Cache Hit Count : 1317
Cache Miss Count : 53
Cache Hit Rate : 96.1314
AMAT : 66170 ns
=====

```

6. Analyze Cache

그러면 출력된 결과를 바탕으로 Direct Mapped Cache, SCA Algorithm에서 Write Through를 사용했을 때와 Write Back 방식을 사용했을 때를 비교하도록 하겠다. 표로 정리하면 다음과 같다.

Program	Replacement	Write Policy	Cache Total	Cache Hit	Cache Miss	Hit Rate	Memory Access	AMAT
simple.bin	SCA Algorithm	Write Through	2	1	1	50%	0	1010ns
		Write Back	2	1	1	50%	0	1010ns
simple2.bin	SCA	Write	4	3	1	75%	0	1030ns

	Algorithm	Through						
		Write Back	4	3	1	75%	0	1030ns
simple3.bin	SCA Algorithm	Write Through	613	612	1	99.8359%	0	7120ns
		Write Back	613	612	1	99.8369%	0	7120ns
simple4.bin	SCA Algorithm	Write Through	100	93	7	93%	1	7930ns
		Write Back	100	93	7	93%	1	7930ns
fib.bin	SCA Algorithm	Write Through	1095	1088	7	99.3607%	1	17880ns
		iWrite Back	1095	1088	7	99.3607%	1	17880ns
gcd.bin	SCA Algorithm	Write Through	1370	1317	53	93.1314%	1	66170ns
		Write Back	1370	1317	53	96.1314%	1	66170ns

7. Conclusion & Feelings

이번 Cache 과제를 끝으로 모든 컴퓨터 구조 프로젝트 과제가 끝났습니다. 처음 Calculator Project를 진행하면서 생각했던 목표가 객체 지향 언어와 Low Level 언어 사이의 언어인 C++로 컴퓨터 구조를 표현하고 싶었지만 C++의 기능들은 객체 지향에 가까워서 그런지 C언어 만큼 컴퓨터 구조를 표현하기에는 쉽지 않았던 것 같습니다. C언어만큼 컴퓨터의 하드웨어, 구조를 잘 표현할 수 있는 언어는 없는 것 같다고 느꼈습니다. 지금 우리가 당연하게 사용하고 있는 컴퓨터 뒤에는 수많은 구조들이 융합되어 있고, 이번 학기를 거치면서 컴퓨터 구조의 기본을 구현해보았지만 이 또한 쉽게 구현하기에는 난이도가 높았다고 느껴졌습니다. 기존의 프로그래밍 과제들은 헤더파일 분할도 크게 필요없고, 시간을 많이 투자하지 않아도 구현할 수 있는 과제들이었지만 컴퓨터 구조의 4개의 프로젝트들은 이론의 이해는 기본이고, 이해한 개념들을 프로그래밍 하는 것은 또 다른 문제였다고 느꼈습니다. Cache Project는 기말고사 기간과 겹쳐 많은 시간을 투자하지 못하였습니다. 그에 따라 Fully Cache, Set Associative Cache를 구현하지 못했고, Replacement Policy의 SCA를 제외한 다른 Algorithm 또한 구현하지 못한 것은 아직도 아쉽게 느껴지고 있습니다. 다시한번 느낀점이지만 프로그래밍이란 것은 절대 쉬운 것이 아닌 것 같습니다. 정말 많은 노력이 필요하고, 습관처럼 해야하는 것이 프로그래밍인 것 같습니다. 이번 프로젝트들을 진행하면서 정말 많은 정보를 찾아보았고, 수많은 디버깅 또한 진행하였습니다. 이러한 시행착오들과 노력들은 절대 헛되지 않을 것이라고 생각합니다. 이번 프로젝트 기간동안 프로그래밍 측면에 있어서는 정말 많이 성장했다고 본인 스스로 느끼고 있습니다. 프로그래밍 당시에는 너무 고통스럽고 힘들었지만 막상 4개의 프로젝트를 끝내고 나니 후련하기도 하면서 한편으로는 아쉬운 점들도 많이 있다고 생각합니다. 컴퓨터 학과를 다닌다면 한번쯤은 들어보아야 하는 컴퓨터구조 과목, 정말 오랫동안 기억에 남을 것 같습니다. 한 학기동안 좋은 강의 정말 감사했습니다.