**Report : Project #3**

**MIPS Emulator with the Cache**

**컴퓨터공학과 32191597 박민규**



**과목 : 컴퓨터구조론**

**담당교수 : 유시환 교수님**

Contents

1. About Project -------------------------------- 2p

1-1. 프로젝트 설명 및 요구사항

1-2. 기본 개념

2. About Code --------------------------------- 12p

2-1. 환경 설명

2-2. 예외 처리

2-3. 파일 설명

2-4. 구조 및 주요 코드 설명

3. About Sample Program -------------------- 19p

4. About Conclusion -------------------------- 23p

**1. About Project**

**1-1 프로젝트 설명 및 요구사항**

본 프로젝트는 이전 프로젝트에서 구현했던 파이프라인을 이용한 MIPS 시뮬레이터 프로그램을 캐시 메모리를 이용하여 구현하는 것입니다. 캐시 메모리를 이용함으로써 이전 프로젝트에서 구현했던 파이프라인 혹은 싱글 사이클 보다 메모리에 접근을 더 효율적이고 빠르게 할 수 있기 때문에 더 좋은 성능을 가진 시뮬레이터를 구현할 수 있습니다. 본 프로젝트를 통해 다양한 캐시 메모리를 이해하고 캐시 사이즈에 따른 hit rate를 이용해 성능을 비교해보는 것이 목표입니다.

프로젝트의 요구사항은 다음과 같습니다.

**1) 기본 요구사항 및 캐시 메모리**

- 이전 프로젝트에서 구현했던 파이프라인을 이용한 MIPS 시뮬레이터와 동일하게 작동합니다.

- 다른점은 메모리에 접근할 때 캐시 메모리에 접근하여 값을 가져오거나 채웁니다.

- 명령어가 전부 실행되면, 다음과 같은 값들을 추가로 출력해야합니다.

- 캐시에 접근한 횟수

- 캐시 hit한 횟수

- 캐시 miss한 횟수



**1-2 기본 개념**

**1) 캐시 메모리에 대한 이해**

캐싱은 컴퓨터의 처리 성능을 높이기 위한 기법입니다. CPU는 데이터를 처리하기 위해 메모리와 끊임없이 데이터를 주고받는데 이때, CPU에 비해 메모리는 속도가 느립니다. 그렇기 때문에 메모리에 접근할 때 CPU는 효율적으로 사용되지 못합니다. 캐시 메모리는 CPU와 메모리의 속도 차이로 인한 병목 현상을 완화하기 위해 사용됩니다. 캐시 메모리는 메모리와 CPU 사이에 위치해 있으며, <그림 1>과 같이 메모리 계층 구조에서 레지스터 다음으로 상위에 위치합니다. 자주 사용하는 데이터를 CPU와 가까운 위치에 저장하여 필요할 때마다 빠르게 꺼내어 쓸 수 있습니다.

<그림 1 - memory hierarchy>

캐시 메모리를 사용하면 CPU가 메모리에 접근하는 횟수를 줄여 성능 향상의 효과를 볼 수 있습니다.

**2) Hit 와 Miss**

캐시 메모리가 있는 컴퓨터 시스템은 CPU가 메모리에 접근하기 전에 캐시 메모리에 원하는 데이터가 존재하는지 확인합니다. 이때 필요한 데이터가 있는 경우를 **Hit**, 실패를 **Miss**라고 합니다.

요청한 데이터를 캐시 메모리에서 찾을 확률을 **Hit ratio**라고 합니다. 캐시 메모리의 성능은 Hit ratio에 의해 결정됩니다.

\* Hit ratio = Hit / (Hit+Miss)

Hit라면 캐시 메모리의 데이터를 CPU 레지스터에 복사합니다. Miss라면 메모리의 데이터를 캐시 메모리에 복사하고, 캐시 메모리의 복제된 내용을 CPU 레지스터에 복사합니다. 캐시 메모리에 데이터를 최대한 많이 넣는다면 CPU에 필요한 대부분의 데이터를 빠르게 얻어올 수 있지만, 용량이 적기 때문에 일부분만 적재할 수 있습니다.

**2-1) Locality**

캐시 메모리의 성공 여부는 locality의 원리에 달려있습니다.

- temporal locality : CPU가 한 번 참조한 데이터를 다시 참조할 가능성이 높습니다.

- spatial locality : CPU가 참조한 데이터와 인접한 데이터를 참조할 가능성이 높습니다.

- sequential locality : 분기가 발생하지 않는 한 명령어는 메모리에 저장된 순서대로 실행됩니다.

**3) 캐시 메모리의 구성과 동작**

CPU는 데이터를 요청할 때마다 캐시 메모리에서 해당 데이터의 존재를 확인하기 때문에 캐시 메모리는 요청받은 데이터를 탐색하는 기능이 필요합니다. 캐시 메모리는 데이터를 관리하기 위해 data 메모리와 tag 메모리를 사용합니다. 다음은 용어에 대한 설명입니다.

- block : data의 기본 단위인 word의 집합

- data memory : memory의 data들이 저장된 block으로 구성

- tag memory : data memory의 block을 탐색할 정보를 포함

tag memory의 entry는 data memory block과 쌍을 이루면서 tag, valid bit, dirty bit를 포함합니다.

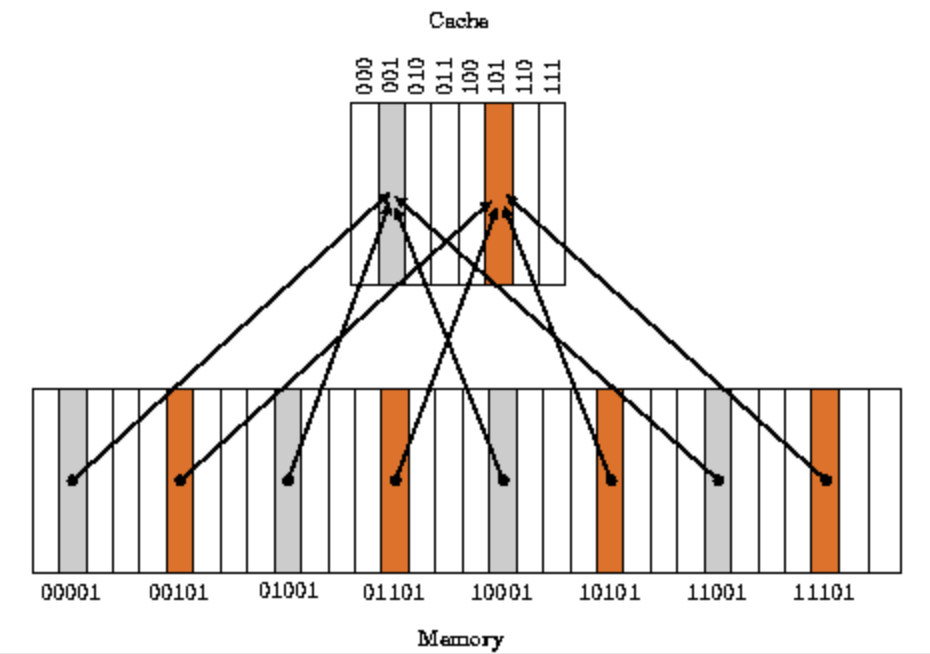
- tag : CPU가 요청한 데이터를 탐색하는데 사용할 주소의 일부. 캐시 block 주소에서 index로 사용되지 않는 부분

- valid bit : 캐시 block이 유효한 데이터인지 나타냅니다.

- dirty bit : 캐시로 block을 가져온 후 CPU가 block을 수정했는지 나타냅니다.

**4) 캐시 메모리의 구조**

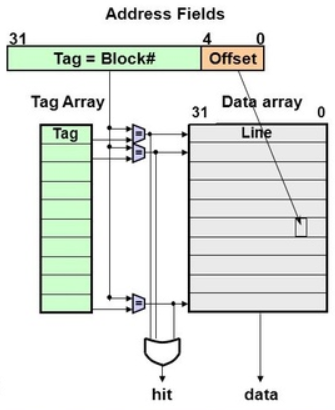
**4-1) Direct Mapped Cache**



<그림 2 - Direct Mapped Cache>

캐시와 메모리 간에 <그림 2>와 같이 mapping 되어 있는 것을 direct mapped cache 구조라고 합니다. 캐시가 메모리의 특정 영역에 access 하고 싶으면 위와 같이 캐시의 한 영역이 여러 개의 memory로 공유된 형태를 띌 것입니다.

address를 받으면 우선 tag, index, offset으로 나누어 쪼갭니다. index와 offset에 해당하는 캐시가 valid라면 해당 캐시의 tag와 주소의 tag를 비교하여 일치하면 hit, 일치하지 않으면 miss입니다. 간단하고 빠르지만, miss가 일어날 때 같은 index에서 tag가 달라 발생하는 conflict miss가 발생할 수 있습니다.

**4-2) Fully Associative Cache**

모든 block을 탐색하면서 비어있는 캐시 메모리가 있으면 바로 access하여 저장하는 방식을 fully associative cache 구조라고 합니다. 모든 block을 탐색하므로 index bit가 필요 없습니다. 따라서 address를 tag와 offset으로만 나누어 쪼갭니다. 이로인해 conflict miss를 피할 수 있지만, 속도가 느리다는 단점이 있습니다.

**4-3) Set Associative Cache**

 direct mapped cache와 fully associative cahce의 장점을 조합한 방식입니다. 하나의 주소 영역이 서로 다른 태그를 갖는 여러 개의 집합으로 이루어지는 방식입니다. 집합의 개수가 N개라면 N-way Set Associative Cache라고 합니다. 특정 행을 지정하여 해당 행안의 어떤 열이든 비어있으면 저장합니다. direct mapped cache에 비해 탐색은 오래 걸리지만, 저장이 빠르며, fully associative cache에 비해 저장이 느리지만 탐색이 빠릅니다. cache line 개수를 way의 개수로 나눈 값을 log2를 취해준 값이 index(set) bit가 되므로 way가 증가할수록 index bit가 감소하고 tag bit가 증가합니다. index bit로 특정 set으로 이동하여 모든 way를 비교하여 알맞은 곳에 접근하는 방식입니다. 이에 따라 way가 증가할수록 miss rate가 줄고, 같은 way에서는 block이 클수록 miss rate가 줄어듭니다.

<그림 3 - Fully Associative Cache>

**5) Cache Miss**

위에서 설명한 것과 같이 tag값이 일치하지 않으면 miss 입니다. 이때 miss의 종류는 다음과 같이 나눌 수 있습니다.

- Cold Miss

처음 캐시 메모리에 접근한다면 캐시 메모리는 초기화 상태로 모두 비어있습니다. 비어있는 캐시 메모리에 접근할 때는 무조건 miss가 나는데 이것을 cold miss라고 합니다.

- Capacity Miss

캐시 메모리의 용량이 부족하여 발생하는 miss로 접근하는 데이터의 양이 캐시의 사이즈를 넘어갈 경우 발생합니다.

- Conflict Miss

Direct mapped cache에서는 같은 index에 다른 tag가 들어왔을 때, set associative cache에서는 set에 way가 부족하여 발생하는 miss입니다.

**6) Replacement Policy**

Miss가 발생한 경우 메모리에서 명령어를 가져와야 하는데 어떤 line을 선택하여 replacement를 할지 정하는 정책입니다. Direct mapped cache는 index bit가 정해져있으므로 해당 index에서만 replacement를 하면 되므로 정책이 필요없습니다. 하지만, 다른 두 구조의 경우 정책을 정해야합니다.

- LRU(Least Recently Used) : 가장 최근에 사용된 적이 없는 것, 즉 가장 오랫동안 사용되지 않은 것을 제거하여 대체하는 알고리즘입니다.

- FIFO(First In First Out) : 적재된 시간을 기준으로 교체될 line을 선택하는 알고리즘입니다.

- LFU(Least Frequently Used) : 가장 적게 참조되었던 line을 교체하는 알고리즘입니다.

- SCA(Second Chance Algorithm) : recency와 frequency를 고려하여 교체될 line을 찾아내는 알고리즘입니다. 한 번이라도 참조된 적이 있다면 sca bit를 1로 바꿔주어 한 번 더 기회를 주는 것으로 가장 오래된 것을 교체될 line으로 선택하되 sca bit가 1인 경우 0으로 바꿔주며 그 다음으로 오래된 것을 교체될 line으로 선택합니다.

**7) Write Policy**

데이터 변경 시 캐시 메모리와 메인 메모리에 데이터 저장 시점에 관한 정책입니다.

- Write Through

캐시에 쓰기 명령이 이루어질 때 마다 캐시와 메모리에 동시에 데이터를 저장하는 정책입니다. 항상 메모리에 접근하기 때문에 쓰기 명령에 걸리는 시간이 길어져 효율이 낮아집니다.

- Write Back

캐시에 쓰기 명령이 이루어지는 동안 캐시에만 내용을 저장하고 캐시의 내용이 제거될 때 메모리에 복사됩니다. 이때 dirty bit를 사용하여 잘못된 접근을 방지합니다.

<그림 4 - 2-Way Set Associative Cache>

**2. About Code**

**2-1 환경 설명**

사용 언어: C언어

개발 환경: Windows (editor: visual studio code)

파일 정보: main.c fetch.c decode.c execute.c memory\_access.c write\_back.c MUX.c load.c DMcache.c cache.h

컴파일 및 실행 방법:

1. 위의 파일들이 모두 있는 디렉토리에 위치합니다.

2. 원하는 C파일의 binary 파일을 생성합니다.

3. 컴파일을 다음과 같이 합니다.

gcc main.c fetch.c decode.c execute.c memory\_access.c write\_back.c MUX.c load.c DMcache.c -o cache

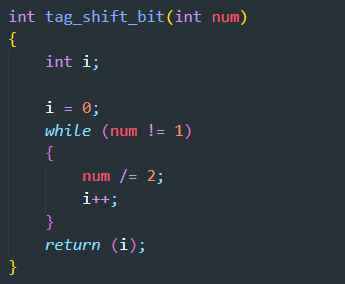
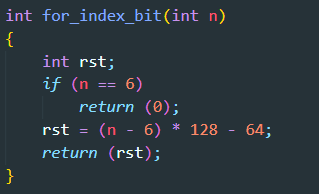
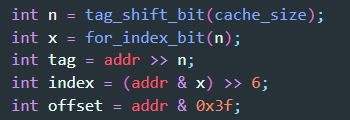
4. 컴파일이 잘 되었다면 cache 실행 파일이 생성되었을 것입니다. ./cache 에 원하는 binary 파일명을 입력하여 실행한 후, 결과를 확인합니다.

4-1) 메인 함수에서 input으로 받는 파일을 인자로 추가해주었기 때문에 파일명과 같이 실행해야 합니다.

**2-2 구조 설명**

 우선 캐시 사이즈는 cache.h 에서 글로벌로 define 하였습니다. 현재 코드에서는 256으로 고정되어 있습니다.

캐시 사이즈는 **64B**로 고정하였고 이에 따라 offset은 address의 제일 왼쪽부터 6bit로 고정하였습니다. 캐시는 Direct mapped cache 구조를 선택하였고 이에 따라 address를 tag, index, offset 비트로 나누었습니다. tag bit를 계산하기 위한 shift bit를 위에서 define 했던 cache size를 인자로 취해 값을 계산했고 계산된 tab shift bit를 인자로 index 계산에서 쓰이는 값을 계산했습니다.



또한, 쓰기 정책은 **write back**을 선택했기 때문에 dirty bit를 추가했습니다. SW일 때 접근하는 MemWrite함수에서 hit 거나 miss일 때 항상 dirty bit를 1로 해주었고 캐시에 접근할 때 conflict miss가 발생한다면 dirty 가 발생했던 메모리 주소에 값을 채워주어야 하므로 dirty address를 구해서 메모리에 값을 채워주고 dirty bit를 0으로 해주었습니다.

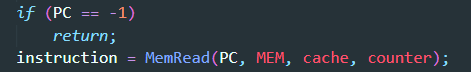


이때, n은 위의 tag\_shift\_bit와 같은 값입니다.

**2-3 수정한 파일 설명**

기존 코드에서 fetch 단계와 memory\_access 단계에서 다음과 같은 수정을 하였습니다.

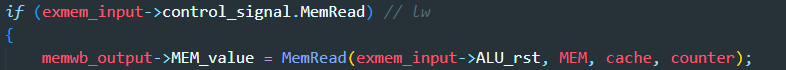
ㅁ fetch.c



이전 명령어가 jr ra라 PC가 -1일 경우 캐시에 접근할 때 memory 배열 범위를 초과하는 값에 접근하므로 segmentation fault 에러가 났습니다. 때문에 fetch 단계에서 PC가 -1일 때는 그냥 return을 해줌으로 함수를 끝냈습니다. 그것이 아니라면 기존에 메모리에 바로 접근하여 명령어를 가져오는 것 대신 MemRead라는 캐시 메모리에 접근하여 값을 읽어오는 함수에 접근하여 명령어를 가져옵니다.

ㅁ memory\_access.c

memory access 단계에서 역시 메모리에 접근하는 대신 캐시 메모리에 접근하였습니다.



SW일 때는 MemRead 함수에서 리턴하는 값을 MEM\_value에 저장했고



SW일 때는 MemWrite 함수에 접근하여 캐시에 값을 저장합니다.

**3. About Sample Programs**

다음은 예제 소스코드에 대한 설명입니다.

1. simple.c : 오직 리턴만하는 함수

2. simple2.c : 100을 리턴하는 함수

3. simple3.c : 1부터 100까지의 합을 리턴하는 함수 (반복문)

4. simple4.c : 1부터 10까지의 합을 리턴하는 함수 (재귀)

5. gcd.c : 0x1298과 0x09387의 최대공약수를 리턴하는 함수 (재귀)

6. fib.c : 피보나치 수열의 10번째 숫자를 리턴하는 함수 (재귀)

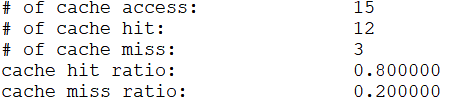
7. input4.c : input\_data.h에 저장되어 있는 값들 중 102번째 작은 값을 리턴하는 함수 (반복문)

다음은 출력에 대한 설명입니다. 각 예제 소스코드에 대한 마지막 출력 결과와 hit rate를 보여주고 cache size를 다르게 하면서(64B, 128B, 256B) 결과를 비교하여 분석하겠습니다.

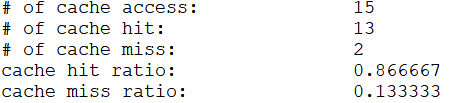


**3-1 simple.c**

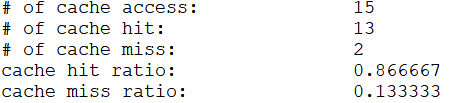
- 64B



- 128B



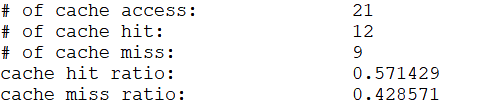
- 256B



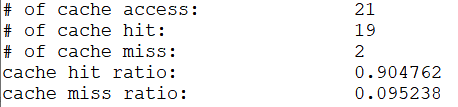
캐시에 접근하는 수가 적다보니 cache size에 따라 값의 변화가 적습니다.

**3-2. simple2.c**

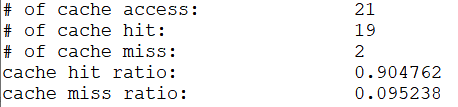
- 64B



- 128B



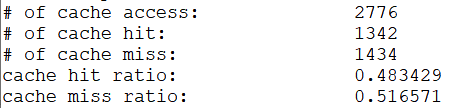
- 256B



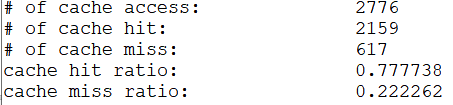
이번 코드역시 캐시에 접근하는 수가 적어 128B와 256B일 때 동일한 ratio를 보입니다. 하지만 64B인 경우 cache line이 한 개이므로 미스율이 상당히 높아지는 것을 알 수 있습니다.

**3-3. simple3.c**

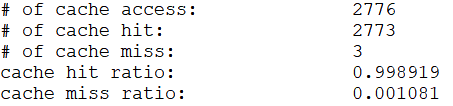
- 64B



- 128B



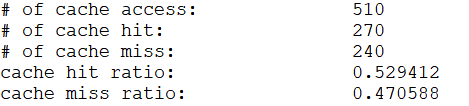
- 256B



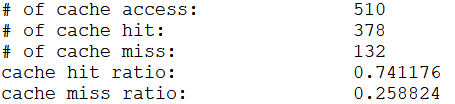
캐시에 접근하는 횟수가 많아진만큼 캐시 사이즈에 따라 ratio가 급격히 다른 것을 알 수 있습니다. 예상했듯이 64B인 경우 미스율이 cache line이 한 개이므로 미스율이 상당히 높습니다. 하지만 256B의 경우 miss가 3번밖에 없어 거의 100%에 가까운 히트율을 보입니다.

**3-4. simple4.c**

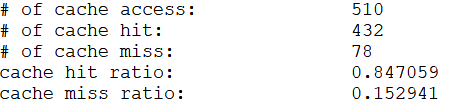
- 64B



- 128B



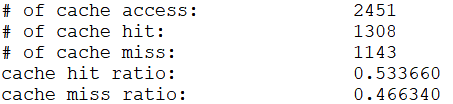
-256B



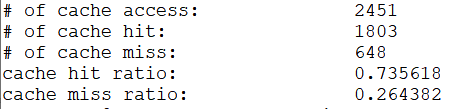
반복문이었던 simple3에 비해 재귀로 함수를 돌린 simple4가 미스율이 더 높은 것을 알 수 있습니다. 이것으로 locality에 영향을 많이 받는다는 것을 알 수 있습니다.

**3-5. gcd.c**

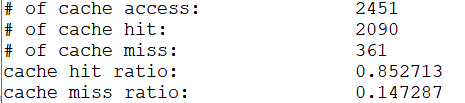
- 64B



- 128B

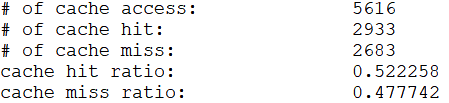


- 256B

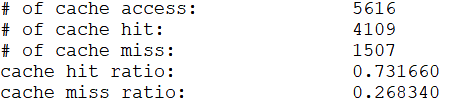


gcd역시 simple4와 마찬가지로 재귀함수를 사용해서 히트율의 상승률이 반복문에 비해 상대적으로 조금 낮은 결과가 나온 것을 알 수 있습니다.

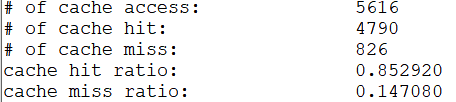
**3-6. fib.c**  
- 64B



- 128B



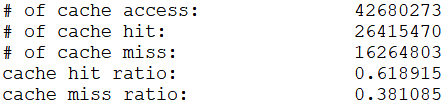
- 256B



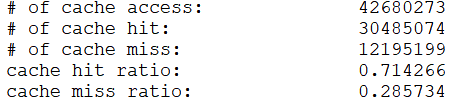
gcd와 마찬가지로 fib역시 재귀함수를 사용해서 비슷한 히트율을 보입니다.

**3-7. input4.c**

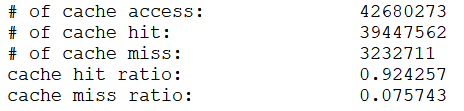
- 64B

****

- 128B

****

- 256B



반복 횟수가 아무리 많아도 반복문을 사용함으로써 256B 기준 90% 이상의 hit rate을 기록했습니다. 실제로 동작한다면 이전 메모리에서 접근했던 pipeline과 비교했을 때 92%를 캐시 메모리에서 값을 가져옴으로써 압도적인 시간 차이를 기록할 것으로 예상됩니다.

전체적인 결과를 종합해보았을 때 히트율에 대해 캐시 액세스 횟수가 지나치게 적으면 캐시 라인이 한 개일 때를 제외하고 거의 차이가 없습니다. 하지만, 캐시 액세스 횟수가 충분히 크다면 캐시 사이즈가 커질수록 히트율이 증가하는 것을 알 수 있습니다. 또한, 재귀함수에서 보다 loop를 사용한 반복문에서 같은 캐시 사이즈에 대해 더 높은 히트율을 기록한 것을 알 수 있습니다.

**4. About Conclusion**

**배운 점과 어려웠던 점**

이번 프로젝트를 진행하면서 캐시의 작동 원리와 개념에 대해 배웠고 replacement policy 에 쓰이는 LRU, SCA, FIFO 등의 알고리즘을 공부하면서 다방면에서 많은 것을 배웠습니다. 완성하고 보니 코드가 몇 줄 되지 않았지만, 이전 프로젝트와 마찬가지로 개념을 명확히 알고 정리해야 그에 따른 알맞은 코드를 작성할 수 있어서 예상했던 것 보다 더 오랜 시간이 걸렸습니다.

- 구조체 배열 선언 (segmentation error)

처음에 개념 정리 하는 부분 역시 오래 걸렸지만, 개념 정리한 것을 토대로 코드를 작성하고 처음 디버깅을 했을 때 segmentation fault 에러가 발생했는데 이유를 알기까지가 오래걸렸습니다. 이것은 구조체를 cache\_size인 256만큼 선언함으로써 발생했던 오류로 글로벌로 선언하면서 해결할 수 있었습니다.

- miss 로직

특히 miss가 발생했을 때 캐시에 값을 채워 넣을 때의 메모리 주소(tmpAddr) 로직과 conflict miss가 발생했을 때 메모리로 write back 하는 로직을 생각 하는게 까다로웠습니다. dirty bit가 발생한 주소에 대해서 write back을 해야 하는데 계속해서 위에서 말한 메모리 주소(tmpAddr)에 write back을 하여 에러가 계속해서 발생했습니다. 계속해서 공부하고 찾아보고 디버깅해본 결과 깨닫게 되었고 dirty bit를 구하는 로직을 생각해내어 해결할 수 있었습니다.

- sca 로직 (LRU 로직)

처음에는 세 가지 구조를 모두 구현 하는게 목표였습니다. 하지만, sca 로직 중 LRU 로직을 꽤 오랜 시간 생각했고 나름대로 잘 짰다고 생각했지만, 계속해서 에러가 났고 여러가지 for문을 통해 구현에 성공했지만 코드가 너무 지저분했고 비효율적이라 생각이 들어 결국 구현에 포기했습니다. LRU에 대해 찾아본 결과 linked list로 구현이 가능했지만 해당 자료구조에 익숙치 않아 단기간에 사용할 수 없었습니다. 이번 계기를 통해서 이번 방학 동안 자료구조와 알고리즘 공부를 단단히 해야겠다고 생각했습니다.

- 글로벌 변수 입력

제가 cache size를 인자로 하여 tag shift bit 및 index 계산에 쓰이는 값을 구한 이유는 input으로 cache size를 받으려는 목적에 있었습니다. 하지만, 위의 구조체 배열 선언과 같은 이유로 cache\_size 변수를 global에서 사용했습니다. 이에 따라 cache\_size를 main에서 입력 받아도 오류가 났고 해결방법을 찾지 못해 결국 헤더에서 define을 통해 수동으로 크기를 수정했습니다.

**잘한 점과 아쉬웠던 점**

**- 스스로 구현에 성공**

물론 개념에 대한 부분이나 로직의 에러가 났던 부분에 대한 도움은 몇몇 학우 들과의 토론을 통하여 해결을 했지만, 첫 과제였던 계산기부터 이번 cache 과제까지 기간안에 스스로 구현에 성공했다는 점은 잘했다고 생각합니다. 하지만 시간 부족으로 다른 두 구조를 구현하지 못한 것은 상당히 아쉬웠다고 생각했습니다. 하지만, 그만큼 개념 정리 및 로직에 시간을 많이 썼고 그것에서 느껴진 공부량 및 코드 구현력에 대해서 능력이 상승했다고 느껴져 뿌듯합니다.

**- 성능 분석을 못한 점**

이번 과제에서 아쉬웠던 가장 큰 부분이 성능 분석을 완벽히 해내지 못한 것입니다. 처음 목표는 세가지 구조를 모두 구현한 후 캐시 사이즈에 따라 AMAT 분석을 하는 것이었습니다. 물론 구현한 Direct mapped cache에서 캐시 사이즈를 다르게 하면서 히트율을 비교해보았지만, latency (penalty) 에 대한 이해가 부족하여 AMAT 분석을 하지 못한 것이 아쉬움이 남습니다.

이번 학기를 다니는 동안 가장 기억에 남는 수업을 떠올리면 바로 이번 컴퓨터구조 수업을 떠올릴 것 같습니다. 사실 저번 겨울방학에 42seoul 이라는 프로그램에 참여하면서 C언어에 대해 자신감이 가득했고 그에 따라 한 학기동안 과제를 모두 C언어로 구현했습니다. 그럼에도 스스로 부족함을 많이 느꼈고 많이 발전하고 싶다는 느낌도 받았습니다. 아키텍처 및 CPU와 같은 로우레벨 분야를 계속해서 공부하면서 흥미롭다는 생각도 했고 나아가 운영체제 분야까지 스스로 찾아보면서 내용을 파고들면서 공부했습니다. 이렇듯 스스로 전공 분야에 온전히 집중하며 몇 날 며칠을 공부하고 지루하지 않게 코딩을 할 수 있는 기회가 처음 있어서 좋았습니다. 특히 이전까지는 코딩이 그렇게 즐겁지 않고 자신도 없어서 진로에 대해서도 불투명하게 생각하고 있었는데, 이번을 계기로 스스로 개발자를 해도 될 것 같다는 생각이 들 정도로 자신감이 올랐습니다. 여러모로 아쉽기도 하고 뿌듯하기도 했습니다. 마지막으로 이런 기회를 주시고 한 학기동안 고생해주신 교수님께 감사하단 말씀 전하고 싶습니다. 감사합니다.