**Computer Architecture and Mobile Process**

**Cache implementation on**

**My MIPS simulator**

**Project4**

**2022년 6월 8일**

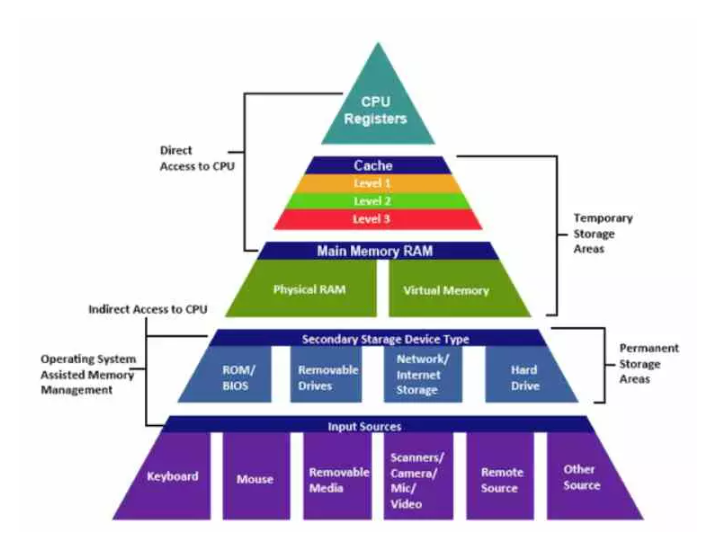
**Left day 5**

**모바일시스템공학과**

**32184939 허찬용**

* 캐시란?

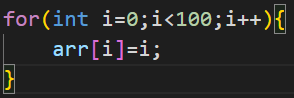
컴퓨터 구조에서 캐시의 사전적 의미는 자주 사용하는 데이터를 저장하는 임시 장소를 의미합니다. 해당 레포트에서 설명할 캐시는 메모리 계층에서 레지스터 바로 아래 계층입니다. 메모리 계층이 높을수록 접근 속도가 빠른데 캐시 메모리는 아래 그림과 같이 레지스터 다음으로 빠릅니다. 캐시 메모리와 메인 메모리를 비교를 하면 캐시메모리가 대략 1000배 정도 빠릅니다.



그 이유 중 하나가 물리적 거리가 먼 메모리에 비해서 캐시는 CPU와의 거리가 가깝다는 것과 Locality라는 개념이 있기 때문입니다. 물리적 거리가 가깝다면 물리적 거리가 먼 메모리에서 데이터를 가져오는 것보다 시간이 적게 걸린다는 말입니다. 하지만 빠르다는 이유로 캐시 메모리의 크기를 늘려CPU에 붙일 수는 없습니다. 그 이유는 비용이 많이 들기 때문입니다. 그래서 비용을 고려하여 최적의 성능을 낼 수 있는 만큼의 크기로 메모리를 붙입니다.

* + Spatial Locality

Spatial Locality는 특정 메모리에 있는 데이터를 사용할 때 해당 메모리 주변의 데이터들을 사용하는 경향이 높다는 것을 의미합니다. 간단한 예시를 들어보겠습니다. 아래 코드를 볼 때 배열의 인덱스에 맞게 메모리에 순차적으로 접근을 합니다. 해당 메모리 주소를 알고 그 주변 주소를 알고 있다면 메모리 접근에 더욱 빨라질 것입니다.



* + Temporal Locality

Temporal Locality는 최근 사용되었던 주소가 다시 사용될 경향이 있다는 것을 의미합니다. 만약 변수를 선언을 하고 해당 변수를 계속해서 Initialize를 하게 된다면 해당 변수의 주소에 계속해서 접근을 하게 될 것입니다. 이와 같은 경향을 Temporal Locality라고 합니다.

* 캐시 구성

캐시를 구현하기 위해서는 여러가지 부분을 생각을 해야 합니다. 사용할 L1 캐시의 크기를 생각하고 해당 캐시의 캐시라인은 얼마로 할 것인지 정해야 합니다. L1 캐시는 캐시 중에서 메모리 계층이 가장 높은 캐시입니다. 계층이 가장 높기 때문에 물리적인 거리도 가깝고 데이터를 가져오는 것도 다른 캐시메모리에 비해 빠르다고 볼 수 있습니다. L1 캐시의 크기는 8KB ~ 64KB입니다.

캐시의 크기는 물론 크면 클수록 좋지만 가격면에서 비싸기 때문에 크게 붙이기에는 현실적으로 어렵다는 것을 알 수 있습니다.

캐시의 크기가 정해지면 이제 캐시라인을 생각해야합니다. 캐시라인은 메인 메모리에서 데이터를 가져올 때 Locality를 통해서 얼마나 많은 데이터를 캐시에 저장할지에 대한 용량입니다. 캐시라인이 크면 메인 메모리에서 사용된 데이터의 주변 데이터가 해당 캐시라인의 크기만큼 가져오게 됩니다. 그렇게 되면 Spatial Locality 와 Temporal Locality를 모두 충족하게 됩니다.

캐시라인의 크기가 정해지면 자연스럽게 캐시라인의 개수가 정해지게 됩니다. 예를 들어 16kb의 캐시에 64byte의 캐시라인이 있다면 캐시라인의 개수는 2^14 / 2^6 총 2^8 즉 256개의 캐시라인이 16kb의 캐시에 존재하는 것을 확인할 수 있습니다. 그렇게 되면 총 인덱스를 확인하기 위해 8bit가 필요하고 offset을 확인하기 위해서 6bit가 필요합니다. 이를 통해 Tag는 32bit-14bit = 18bit가 사용되어야 하는 것을 알 수 있습니다.

앞서 구한 Tag는 해당 인덱스의 캐시라인이 우리가 원하는 주소의 데이터가 맞는지 확인하는 역할을 한다. Index와 Tag 가 같다면 같은 주소이기 때문이다.

* 캐시 종류
  + Direct Mapping

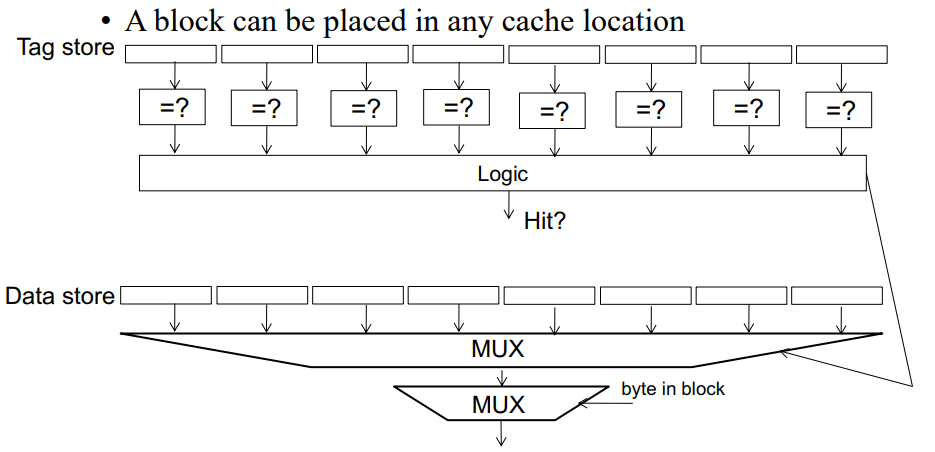
메모리에 접근을 하게 되면 해당 주소의 근처 데이터 값들이 캐시라인의 크기에 맞게 저장이 된다. 캐시에 저장이 될 때 접근하는 주소를 이용하여 tag, index, offset을 구한다. Index를 통해 해당 캐시라인에 접근을 한다. 해당 캐시라인에 맞는 tag를 확인하여 tag가 맞으면 offset을 이용하여 캐시라인에 접근하여 데이터를 가져온다. 만약 tag가 다르다면 교체를 하던 아니면 메모리에서 데이터를 가져오는 방법을 사용해야한다.

텍스트, 영수증, 스크린샷이(가) 표시된 사진

자동 생성된 설명

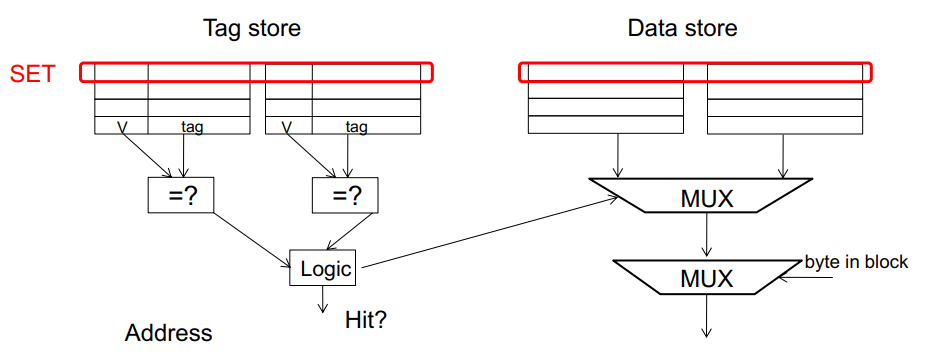
* + Fully Associate

Fully Associate 방법은 비워져 있는 캐시라인이 있다면 해당 메모리 근처 데이터들을 저장을 합니다. 그래서 tag가 맞는지 모든 캐시라인에 가서 확인을 해야하기 때문에 찾는 시간이 오래 걸린다는 단점이 있습니다.



* + Set Associate

Set Associate는 Direct Mapping 기법과 Associate 기법을 합친 기법입니다. Index를 통해 각 way의 캐시라인에 접근을 합니다. Tag가 같은 way가 있다면 해당 way로 가서 데이터를 가져옵니다. 인덱스가 동일한 데이터들을 다양하게 저장할 수 있다는 장점이 있습니다. 만약 모든 way의 index의 tag가 다르다면 해당 Replace를 하거나 메모리에 접근을 하여 데이터를 가져옵니다.



* Replace Policy

Replace에는 여러가지 방법들이 있습니다. 기회를 한번 주는 SCA 기법, 가장 예전에 사용된 캐시라인을 비우는 LRU 기법, 해당 index의 랜덤한 way에 접근하는 Random 기법 등이 있습니다.

* SCA 기법

SCA 기법은 말 그대로 기회를 한 번 더 주는 기법입니다. 캐시에 데이터가 저장이 될 때 SCA는 0으로 저장이 됩니다. 그리고 해당 Index의 캐시라인이 사용이 된다면 SCA는 1로 업데이트가 됩니다. SCA가 1이 된 이후로 해당 캐시라인이 사용이 된다고 하여도 SCA는 커지지 않습니다. 오직 0과 1로만 표현이 됩니다. SCA가 0 인 경우에 동일 Index에 접근을 하였으나 Tag가 다르다면 해당 캐시라인은 교체가 될 것입니다. 하지만 SCA가 1인 경우에는 기회를 한 번 더 주어 SCA가 0이 되고 캐시라인은 보존이 됩니다. 이와 같은 이유는 SCA를 통해 해당 캐시라인에 접근 가능성이 높다는 것을 확인을 하고 기회를 주는 것입니다. 만약 이와 같은 기회를 주지 않으면 Cache Miss가 발생하는 경우가 많아질 수 있기 때문입니다.

* LRU 기법

LRU 기법은 가장 예전에 사용이 된 캐시라인을 교체하는 것입니다. 해당 기법은 Direct mapping에서는 사용이 되지 못합니다. 하지만 Set Associative 기법이나 Fully Associate 기법일 때 사용이 가능한 것입니다. 각 Way에 같은 Index의 캐시라인 중 가장 예전에 사용이 된 캐시라인을 교체를 하는 것입니다. Way가 4인 Set Associative 기법인 경우에는 4 way 중 가장 예전에 사용이 된 캐시라인을 교체를 하면 좋은 점이 가장 사용이 되지 않은 캐시라인이라고 생각을 할 수 있기 때문입니다. 해당 기법은 Cycle 번호를 확인을 하여 가장 작은 Cycle 번호를 가진 cache line을 저장하여 비교를 하면 됩니다.

* Random

Random 기법 같은 경우에는 SCA와 같은 기법을 함께 사용하는 것도 좋은 방법입니다. 물론 해당 Index의 캐시라인이 모두 차있는 경우에 랜덤한 Way에 접근하여 교체할 수도 있습니다. 하지만 SCA를 사용하는 경우 SCA가 0인 Way에 접근을 하여 해당 Way들만 모아 랜덤하게 캐시라인을 Replace할 수 있습니다. 그럼 특정 way에 치중되지 않고 평균적인 성능을 낼 수 있습니다. 해당 기법도 Set Associative 기법이나 Fully Associative 기법에서 사용할 수 있습니다.

* Write Policy
* Write-Through

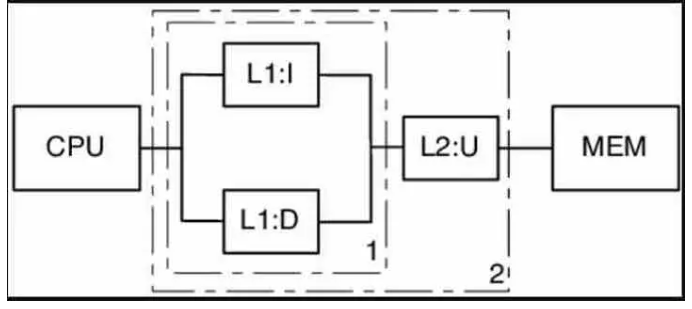
해당 기법은 캐시 메모리와 메인 메모리에 업데이트를 같이 합니다. 이와 같은 이유로 유지성이 안정적입니다. 하지만 속도가 느린 메인 메모리인 경우 cpu에 딜레이를 주어야하기 때문에 성능이 떨어질 수 있습니다.

* Write-back

해당 기법은 캐시 메모리에 해당 접근해야 하는 메모리의 주소가 있으면 캐시에 접근하여 캐시 메모리를 업데이트 합니다. 하지만 교체가 되어야하는 경우 캐시라인을 그냥 교체를 하는 것이 아니라 캐시메모리에 있는 데이터 값들을 원래 메모리로 가서 데이터 값들을 업데이트 시켜주어야 합니다. 물론 캐시 메모리의 변화가 없었다면 메인 메모리로 가서 업데이트할 필요는 없습니다. 위와 같은 이유로 구현이 복잡합니다. 하지만 메모리에 접근을 여러 번 하지 않기 때문에 더욱 편합니다.

* 캐시가 필요한 경우

캐시가 필요한 경우는 2가지가 있습니다. Fetch 단계와 Memory Access 단계입니다. Fetch 단계에서 명령어를 가지고 올 때 메모리에 접근을 해야 합니다. 그리고 Memory Access 단계일 때 메모리에 접근을 해야 합니다. 캐시는 두 가지로 나뉘어져 있어야 합니다. 명령어를 담는 캐시와 데이터를 담는 캐시는 나뉘어져야 하기 때문입니다. 이와 같이 L1 캐시가 나누어지는 이유는 프로세서가 명령어를 병렬적으로 처리할 때 발생할 수 있는 캐시의 경합을 줄이기 위해서입니다.



* 캐시 HIT / MISS

해당 인덱스에 맞는 캐시라인에 접근을 한다면 TAG가 일치하는지를 확인을 해야 한다. 만약 TAG가 일치한다면 HIT 한다는 것이기 때문에 해당 캐시라인의 offset에 접근하여 데이터를 가져오면 됩니다.

Miss인 경우를 생각해보겠습니다. 가장 먼저 캐시라인이 비워져 있을 때 Cold miss가 발생합니다. 해당 cold miss가 발생하면 해당 index의 캐시라인으로 가서 캐시 라인 크기에 맞게 데이터를 메모리로부터 가져오면 됩니다. 하지만 해당 인덱스의 캐시라인이 채워져 있는 경우라면 조금 달라집니다. 해당 캐시라인을 replace해야하기 때문입니다. Replace를 할 때 생각해야할 것이 몇 가지 있습니다. 먼저 Fetch인 경우에는 instruction이 들어가 있는 Cache가 Replace 될 때 아무런 조치없이 업데이트를 시키면 됩니다. 하지만 Memory Access 단계에서는 그냥 Replace를 하면 문제가 생깁니다. 왜냐하면 SW라는 명령어를 통해서 캐시라인의 데이터가 업데이트가 되면 메모리에도 업데이트 시켜주어야 하기 때문입니다. 물론 Write Through 방법을 사용하게 되면 상관이 없겠지만 Write Back 기법을 사용을 하면 dirty라는 변수를 사용해서 해당 캐시라인이 업데이트 되었는지 확인을 해야 합니다. 만약 dirty가 1이라면 먼저 해당 캐시라인의 주소였던 곳으로 가서 메모리에 캐시라인 값들을 저장을 해줍니다. 그 뒤 메모리에서 Replace할 데이터들을 가져와 캐시라인을 업데이트 시켜야 합니다.

* 코드 작업을 통한 결과

1. **Direct Mapping**
   * + Fetch 단계

Fetch 단계에서 캐시를 사용하기 위해서는 먼저 pc를 이용하여 index, tag, offset을 얻어야 합니다. 물론 fully associate 기법을 사용하면 index를 사용하지 않아도 되지만 해당 레포트를 위한 기법으로 direct mapping과 set associate를 사용하였기 때문에 위와 같은 변수들이 필요합니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

해당 코드는 index, tag, offset, start cache를 구하는 방법입니다. 위와 같이 구하는 이유는 캐시의 크기를 16KB로 하고 캐시라인의 크기를 64Byte로 하였기 때문입니다.

* + - * Cold miss 인 경우

Cold miss 인 경우 캐시라인이 비워져 있어 채워야 하는 경우입니다. 캐시라인이 비워져 있는 것은 Valid bit로 확인을 합니다. Valid bit가 0이면 비워져 있다는 것이기 때문에 메모리에 접근하여 캐시라인을 채워줍니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* + - * Hit인 경우

해당 인덱스의 캐시라인이 채워져 있고 Tag가 일치한다면 Hit 했다고 표현을 합니다. HIT 한 경우는 메모리로 접근을 하지 않고 캐시 메모리로 가서 데이터를 가져옵니다. 만약 Replace 방법이 SCA인 경우는 아래와 같이 second bit를 1로 해주어야 합니다. 하지만 그냥 바로 Replace를 한다고 하면 inst\_cache[index].second=1이라는 코드는 할 필요가 없습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* + - * Miss인 경우

만약 캐시라인이 채워져 있으나 Tag가 달라 Miss가 났다면 캐시라인을 Replace 시켜줘야 합니다. Instruction Cache는 Replace 이외에는 캐시라인이 업데이트 되는 경우가 없기 때문에 메모리에 저장할 필요가 없습니다. 하지만 Replace하는 경우는 2가지로 나뉘어집니다. 만약 second bit가 0인 경우에는 캐시라인을 Replace를 해주고 tag만 업데이트 해주면 되지만 만약에 second bit가 1인 경우에는 캐시라인을 Replace를 해주지 않고 instruction만 메모리에서 가지고 옵니다. 그리고 기회를 써버렸기 때문에 second bit가 0으로 됩니다. 만약 Replace 기법이 miss가 나는 경우 무조건 Replace 하는 것이면 캐시라인을 그냥 업데이트 시켜줍니다. 즉 아래의 조건문은 없어질 것이고 반복문과 tag를 업데이트하는 명령어만 있을 것입니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* + - **Memory Access 단계**
      * LW 명령어에서 HIT한 경우

LW 명령어인 경우 hit하게 되면 캐시 메모리에서 데이터를 가져옵니다. HIT 했기 때문에 second bit는 1로 업데이트 됩니다. 만약 SCA를 사용하지 않는다면 second bit 코드는 삭제하면 됩니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* + - * LW 명령어에서 Miss한 경우

LW 명령어인 경우 Miss하는 경우는 2가지가 있다. 캐시메모리가 비워져 있는 경우의 채워야 하는 상황일 때의 Miss 그리고 캐시메모리는 채워져 있지만 Tag를 비교를 하였을 때 Tag가 다른 경우의 Miss가 있습니다.

캐시라인이 비워져 있는 경우의 Miss인 경우는 Tag, second bit과 valid bit를 업데이트 해야 합니다. 그리고 캐시라인 사이즈에 맞게 메모리에서 데이터를 가져와 업데이트 시켜줍니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Data Cache인 경우에는 Instruction Cache인 경우보다 생각해야하는 점들이 더 있습니다. 앞선 fetch cache에서 second bit를 확인한 것과 동일하게 second bit를 확인을 해야 합니다. 확인을 하였을 때 second bit가 1인 경우는 second bit를 0으로 해주고 메모리에서 데이터를 읽어옵니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Second bit가 0인 경우에 Dirty bit를 확인을 해주어야 합니다. Memory Access 단계에서는 캐시라인의 데이터가 SW 명령어를 통해 업데이트 될 수 있기 때문입니다. 만약 Dirty bit가 1인 경우에는 캐시라인의 데이터가 처음 입력될 때와 변하였다는 의미이기 때문에 캐시메모리에 저장된 데이터를 메인 메모리에 저장을 해줍니다. 메인 메모리에 저장을 하고 Replace를 하면 업데이트 된 데이터가 메인 메모리에 저장이 된 뒤 Replace한 것이기 때문에 다음에 LW하거나 메모리에 접근할 때 올바른 값을 가지고 올 수 있습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Dirty bit가 0인 경우에는 해당 캐시라인의 데이터가 처음 입력되었을 때와 비교하여 변하지 않았다는 것이므로 캐시라인을 Replace해주면 됩니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* + - * SW 명령어에서 HIT한 경우

SW 명령어에서 HIT한 경우에는 second bit을 1로 해주고 메인 메모리에 원래 저장할 값을 캐시 메모리에 저장을 해줍니다. 그리고 캐시 메모리의 값이 변하였기 때문에 Dirty bit는 1로 해줍니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* + - * SW 명령어에서 Miss한 경우

SW 명령어를 수행할 때 Miss가 나게 되면 reg[ins->rt] 값을 메모리에 저장을 해준 뒤 Second bit를 확인을 합니다. 만약 second bit가 1이면 second bit을 0으로 해준 뒤 메인 메모리에 접근하여 데이터를 직접 저장합니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

해당 인덱스의 캐시 라인이 채워져 있지 않은 경우는 tag, valid bit, dirty bit을 업데이트 시켜줍니다. 그 뒤 캐시라인 사이즈에 맞게 메인 메모리에서 데이터를 가져와 저장합니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

만약 해당 인덱스의 캐시라인이 채워져 있지만 TAG가 달라 Replace 해주어야 하는 경우는 Dirty bit를 확인을 해야 합니다. Dirty bit가 1이면 캐시라인에 저장된 데이터의 원래 메인 메모리 주소로 가서 업데이트 된 데이터들을 저장을 한 뒤 캐시라인을 교체해줍니다. 만약 Dirty bit가 0이면 처음 들어온 캐시라인 데이터와 같다는 의미이기 때문에 따로 메인 메모리에 저장을 하지 않고 메인 메모리에서 데이터를 가져와 캐시라인에 저장을 해줍니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

1. **Set Associate 기법**
   * + Fetch 단계

Fetch 단계는 Direct Mapping 기법으로 사용하여 앞서 설명한 방법과 동일합니다.

* + - Memory Access 단계
      * Random 기법을 사용하는 경우

가장 먼저 해당 Set의 캐시라인이 비워져 있는지 확인을 합니다. 확인을 하고 set\_index 변수가 -1이라면 해당 인덱스의 모든 Set들의 캐시라인이 채워져 있다는 것을 의미합니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

캐시라인이 모두 채워져 있다면 Miss 이거나 Hit라는 의미이기 때문에 먼저 HIT인지 확인을 해야 합니다. 모든 set을 돌면서 같은 tag를 가지는 set을 찾습니다. 만약 찾지 못하였다면 그 때는 Replace가 되어야 한다는 것을 의미합니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Replace를 하기 위해서는 먼저 Second bit가 0인지 확인을 합니다. 그 이유는 second bit가 0인 것을 replace하면 되기 때문입니다. 만약 모두 second bit가 1이라면 second bit를 0으로 만들 set 인덱스를 랜덤으로 정해줍니다. 만약 second bit가 0인 것들이 있으면 배열을 만들어 1로 업데이트 시켜줍니다. 해당 배열에 1인 값들을 확인하여 랜덤 값으로 replace할 set의 index를 정해줍니다.

텍스트이(가) 표시된 사진

자동 생성된 설명

앞서 구한 index 값들을 사용하여 캐시라인을 채울지, 아니면 HIT하여 캐시라인에서 가져올지, 그것도 아니면 Replace할지를 정해줍니다.

텍스트, 실내이(가) 표시된 사진

자동 생성된 설명

캐시라인이 비워져 있어 캐시라인을 채우는 것이나 HIT하여 데이터를 가져오는 것, 그리고 Replace하는 경우는 앞서 설명한 Direct mapping 과 동일합니다.

* 결과 분석

INSTRUCTION CACHE HIT / MISS

테이블이(가) 표시된 사진

자동 생성된 설명

DATA CACHE HIT / MISS

테이블이(가) 표시된 사진

자동 생성된 설명

메인 메모리에 접근하여 데이터를 가져오는 시간을 100ns 라고 가정으로 하고 캐시 메모리에서 데이터를 가져오는 시간을 0.1ns라고 가정을 합니다. 그러면 캐시 메모리에서 가져오는 시간이 약 1000배 빠른 것입니다. 위와 같이 가정을 하고 캐시를 사용하면 각 얼마의 시간적 이득을 얻는지 확인하겠습니다.

**INSTRUCTION CACHE**

테이블이(가) 표시된 사진

자동 생성된 설명

**DATA CACHE**

테이블이(가) 표시된 사진

자동 생성된 설명

캐시를 사용함으로써 시간 단축을 엄청나게 한 것을 확인할 수 있습니다. 해당 표들의 단위는 ns입니다. 실제 얼마나 차이가 나는지 그래프로 확인해보겠습니다.

**INSTRUCTION CACHE**

**DATA CACHE**

위 그래프를 보았을 때 많게는 1000배 적게는 100배의 시간을 단축한 것을 확인할 수 있었습니다. 이를 통해 HIT RATE가 높을수록 효율성은 올라가는 것을 확인할 수 있었습니다.

* Cache 과제를 하며 느낀 점

Cache 과제를 하면서 PIPE LINE에서 구현했던 Branch Prediction처럼 하드웨어의 효율성을 높이기 위해서는 Prediction 성공률과 같은 예측들이 잘 맞아야 한다는 것을 느꼈습니다. Cache 과제에서는 Cache HIT RATE가 PIPE LINE의 Branch Prediction과 같습니다. Replace의 기법에 따라 HIT RATE가 달라지는 것을 확인할 수 있었습니다. Simple3, simple4, gcd, fib파일들은 pc의 범위가 좁았고 Data Memory에 접근하는 범위도 좁았습니다. 그렇다 보니 어떤 Replace 기법을 사용하든지 간에 HIT RATE가 같았습니다. 여기서 주목해야하는 파일은 INPUT4였습니다. 해당 파일은 cache가 Direct mapping이냐 아니면 Set associate 기법이냐에 따라 영향을 많이 받았습니다. 뿐만 아니라 Replace 기법에서도 영향을 받는 것을 확인할 수 있었습니다. 이와 같은 이유들로 조금 더 HIT RATE를 높이기 위해 연구를 하고 그 결과들이 계속해서 나온다는 것을 깨닫았습니다.

* Cache 과제를 끝내며

CACHE 과제를 하면서 역시나 제일 먼저 종이에 의사코드를 적으면서 생각을 정리하였습니다. 컴퓨터 구조라는 수업을 듣기전에는 무턱대고 코드를 짜기 시작하였으나 해당 과목을 들으면서 제가 구현하고 싶은 것들을 생각을 먼저하고 정리를 한 뒤 코드를 적기 시작하게 되었습니다. 어렵고 복잡한 구현일수록 생각을 하고 도안을 만들어야 한다는 것을 뼈저리게 느꼈습니다. 저에게는 너무 유익한 시간이었던 것 같습니다. CACHE 과제뿐 만 아니라 SINGLE CYCLE, PIPE LINE 등 너무 좋은 과제였습니다. 물론 머리에 부화가 많이 걸리고 디버깅하면서 많이 힘든 점도 있었지만 개발자로 가는 길에 한 걸음은 내딛었다고 생각을 합니다. 물론 아쉬운 점도 많았습니다. 특히 CACHE 과제 시간이 더 있었다면 어땠을까? 라는 생각이 들었습니다. 시험이 금요일에 끝나고 며칠 남지 않은 시간동안 원래 만들었던 Direct mapping을 조금 손보니 에러가 터지고 그것을 고치니 토요일 하루가 다 갔습니다. 레포트를 쓰고 코드를 조금 손보니 6월 20일 새벽 5시였습니다. 이제 과제가 끝났다는 행복감도 있지만 시간이 조금만 더 있었다면 LRU나 FIFO 등 다른 알고리즘들을 구현하며 성능 비교를 하였다면 더욱 좋지 않았을까 생각이 들었습니다. 한 학기동안 교수님께 많은 것을 배우고 깨닫는 좋은 시간을 가졌던 것 같습니다. 교수님 감사합니다.