**CSED-311 LAB5 REPORT**

김모세

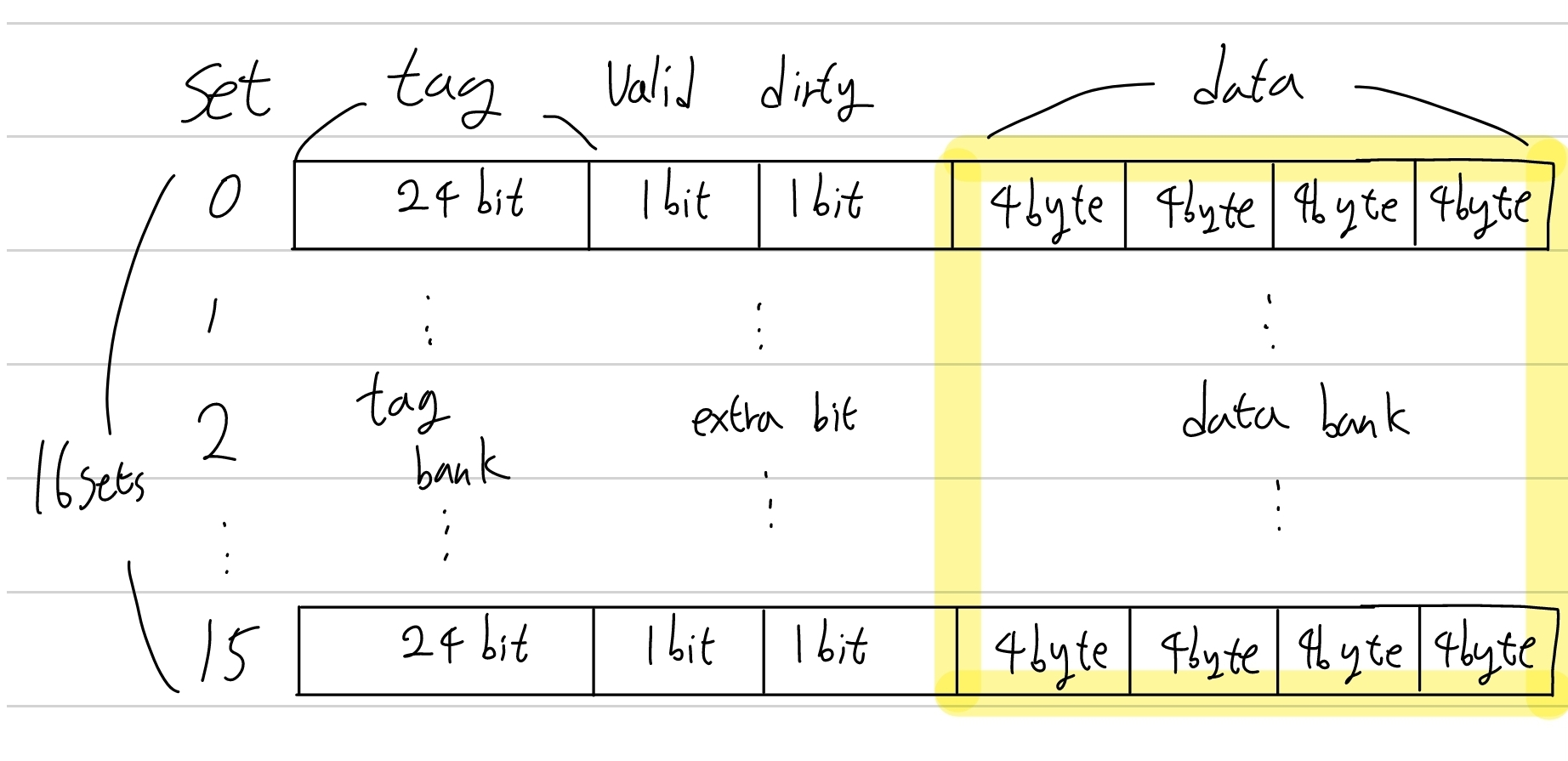
표승현

**Introduction**

verilog 언어를 이용해 direct-mapped cache를 구현하는 것을 목표로 한다. CPU는 memory의 값에 접근하기 위해 cache를 먼저 접근한 후, miss가 발생한 경우 data memory에서 값을 얻을 수 있다. cache는 자주 쓰이는 데이터를 임시적으로 보관하는 공간으로 메모리 접근에 소요되는 시간을 줄이는 역할을 한다.

구현한 cache를 naive matmul, optimal matmul algorithm으로 각각 실행해보며 hit rate를 분석해본다.

**Design**

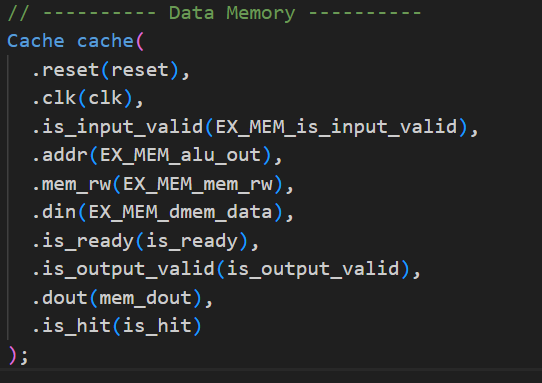
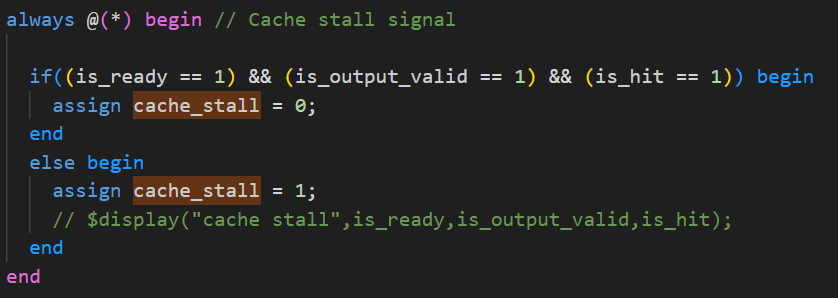
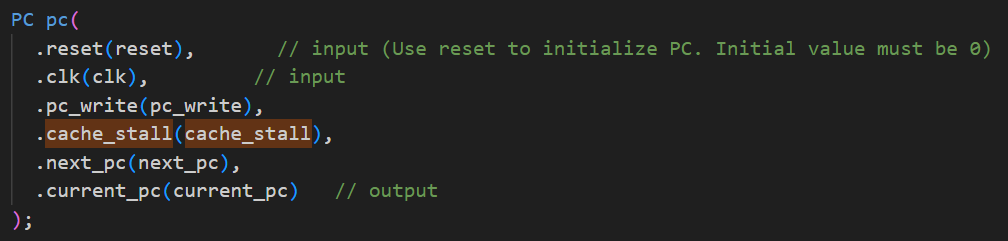
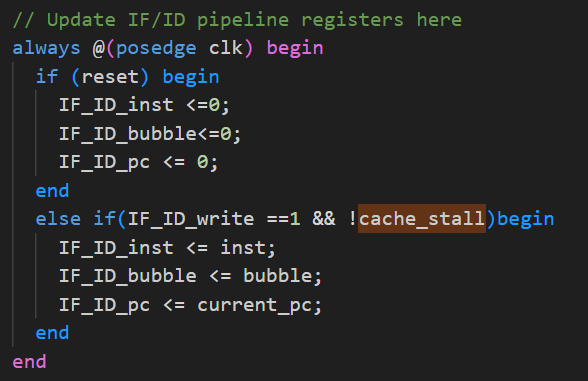
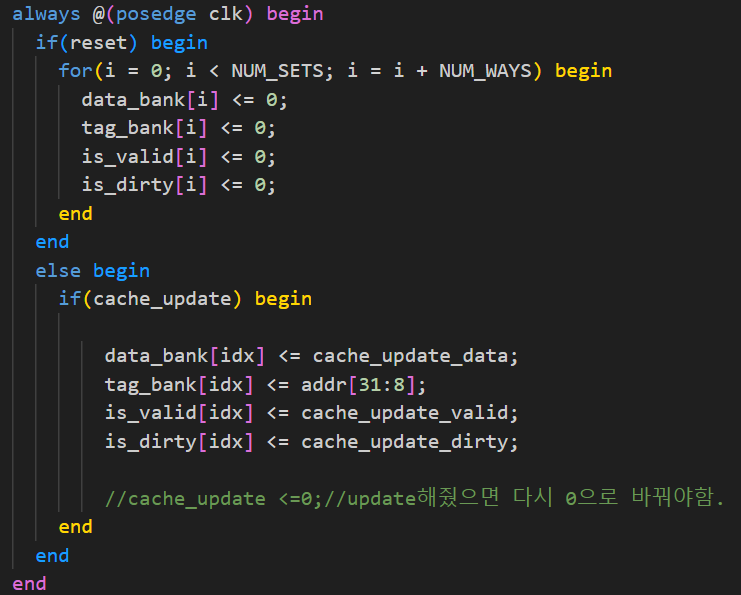
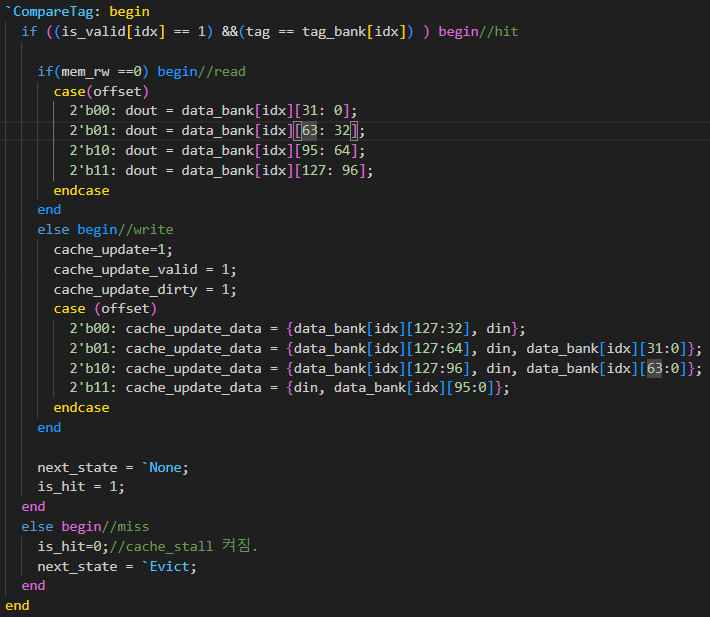
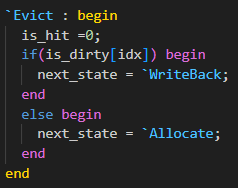
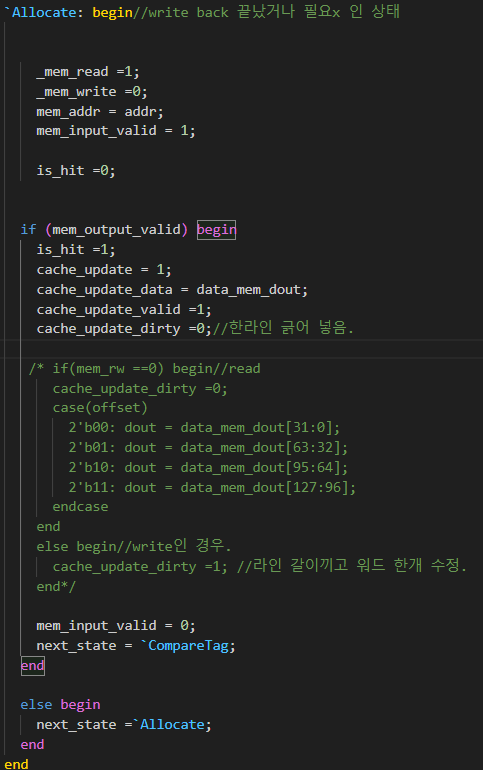
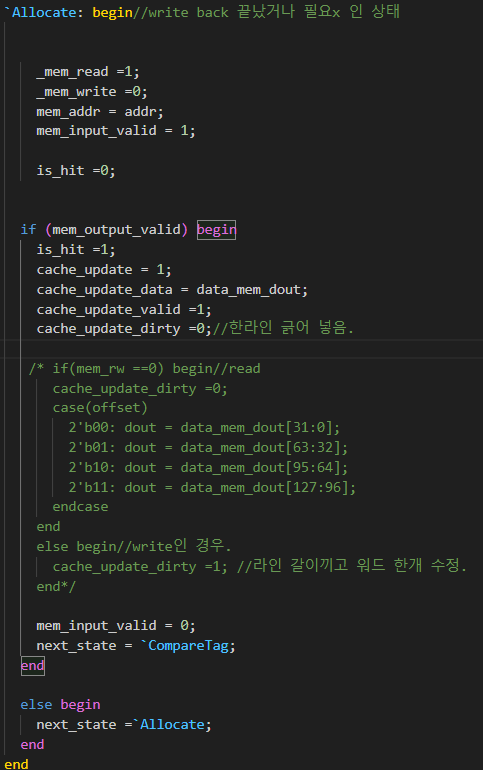
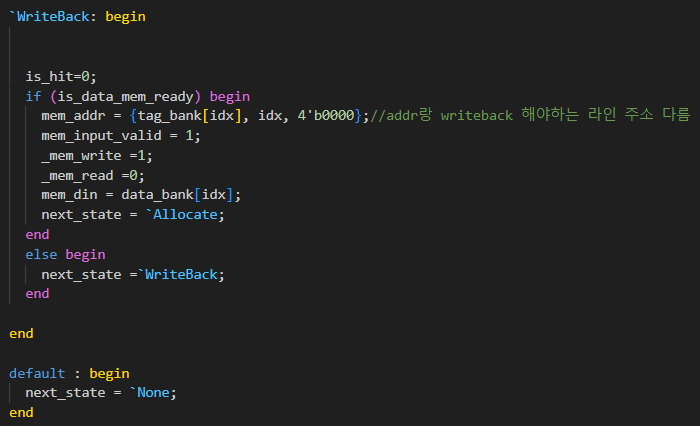


<Design of cache>

이번 실습에서 구현한 cache의 모습은 위와 같다. set은 16개, 각 set당 cache line은 1개가 할당된다. 각 cache line은 16byte로 4개 word를 저장할 수 있다. 추가로 cache line의 유효성을 판단하기 위한 valid bit 1개와 수정 여부를 판단하는 dirty bit 1개를 저장할 공간을 추가로 할당한다.

따라서 addr 32개 bit 중에서 4bit가 set를 표현에 할당되고, 4bit가 offset 그리고 나머지 24bit가 tag를 표현하는 데 쓰인다.

**Implementation**

1. cpu module
   1. 메모리 접근
      1. 
      2. 이전 lab과 달리 memory 데이터에 접근하는 경우 data memory가 아닌 cache에 우선적으로 접근한다. cache에서 miss가 일어난 경우, 모듈 내부에서 data memory에 접근할 수 있도록 구현했다.
   2. cache stall
      1. 
      2. cache stall의 조건은 is\_ready와 is\_output\_valid 그리고 is\_hit signal이 1이 아닐 경우이다. cache가 miss를 처리하는 동안 memory에 접근하는 과정에서 추가적인 사이클이 발생한다. 이때 memory 접근이 완전히 처리되지 않았음에도 다음 instruction이 fetch되어 pipeline이 실행되는 것을 막기 위해 cache stall signal을 정의하여 사용한다.
      3. 
      4. pc모듈에 cache stall 시그널을 연결하여 새로운 pc 업데이트를 멈춘다.
      5. 
      6. cache stall이 활성화되어 있는 경우에는 register를 업데이트하지 않는다.
2. Cache module
   1. synchronously write to cache
      1. 
      2. cache\_update가 활성화 되어 있는 경우 cache의 정보를 새롭게 갱신한다. cache\_update 시그널과 다음 cache 정보는 cache 내부에서 state가 진행됨에 따라 적절히 제공된다.
   2. State
      1. cache는 None, CompareTag, Evict, Allocate, WriteBack으로 총 5개의 state로 구성된 FSM이다. 각 state에서 적절한 동작을 수행하며 cache 기능을 구현한다. 보다 자세한 동작은 아래에서 서술한다.
      2. None
         1. cache의 default state이다. is\_input\_valid가 활성화되면 next\_state를 CompareTag state로 갱신하여 cache 동작을 실행한다. 아닌 경우에는 None state에 머물러 있는다.
      3. CompareTag
         1. 
         2. cache가 메모리 접근을 감지하여 addr에 해당하는 data가 cache에 존재하는 지 검사하는 단계이다. Hit인 경우, mem\_rw의 값에 따라 적절한 동작을 수행한다. read인 경우 data bank에 존재하는 data를 dout에 할당하여 내보내준다. write인 경우 해당하는 cache line의 block에서 offset에 해당하는 word를 input 값으로 수정하고, valid 와 dirty 값을 1로 갱신한다. 이후 next\_state를 None으로 보내주어 동작을 마친다.(이렇게 설정한 캐시 업데이트 값들은 synchronous하게 캐시를 업데이트한다.)
         3. 하지만 cache에 addr에 해당하는 data가 없는 경우 miss라고 판단하여 Evict state로 간다.
      4. Evict
         1. 
         2. addr에 해당하는 cache line의 dirty bit가 1인 경우 이를 memory에 write back해줘야 하기 때문에 WriteBack state로 이동한다. 그렇지 않은 경우 cache line을 교체해주기만 하면 되기 때문에 Allocate state로 이동한다.
      5. Allocate
         1. 
         2. 
         3. mem\_addr에 접근하고자 하는 address 주소를 넘겨주고, mem\_input\_valid를 활성화하여 data memory에서 data에 접근한다. 원하는 data가 준비되면 data memory는 mem\_output\_valid를 활성화한다. 이 시그널이 활성화되면 가져온 data를 cache\_update\_data에 할당해주어 cache line의 값을 갱신한다. 이후 다시 memory에 접근하지 않도록 mem\_input\_valid를 비활성화한 후, CompareTag state로 이동한다.
      6. WriteBack
         1. 
         2. dirty bit가 1인 cache line을 evict할 때, 저장된 data를 memory에 업데이트한 후 evict해야 한다. 따라서 해당하는 addr와, data를 data memory에 전달한 후 mem\_input\_valid를 활성화하여 data memory에 write 동작을 수행하도록 한다. 이후 Allocate state로 이동한다.

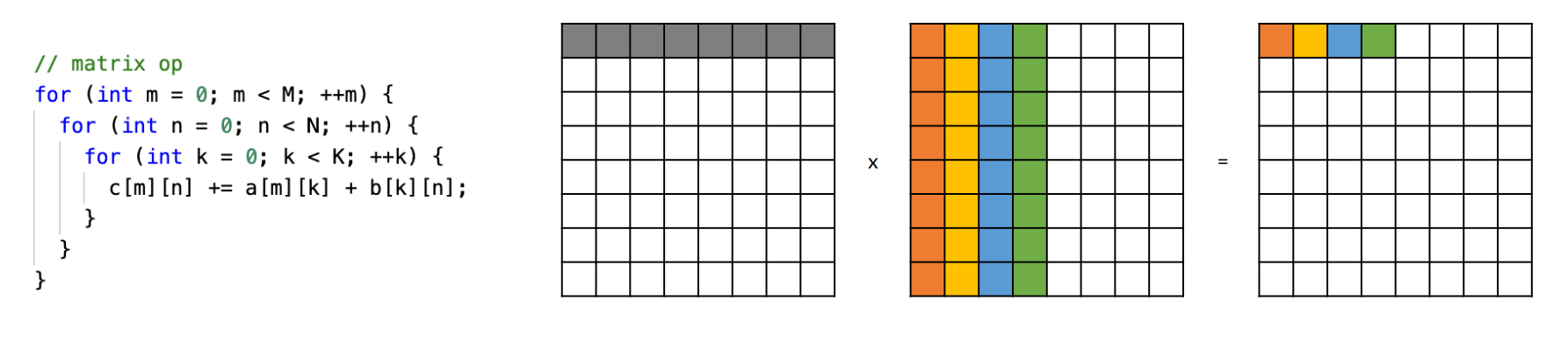
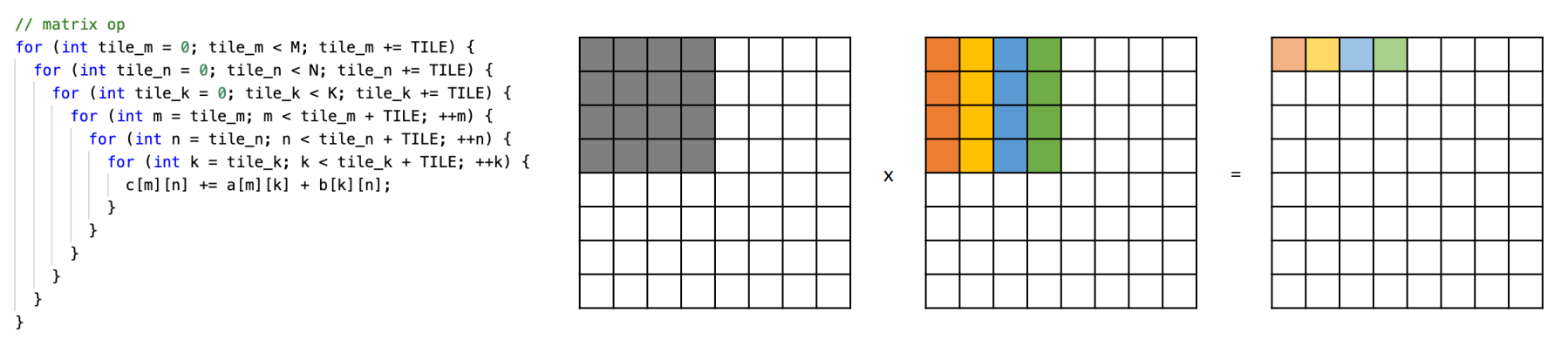
\*) hit\_ratio 계산

hit\_ratio는 top 모듈의 출력으로 hit,miss signal을 새로 연결해 tb에서 계산하였다. cache 모듈의 Comparetag state에서 hit이 날때마다 hit 시그널을 1로 켜주고 miss가 날때마다 miss 시그널을 1로 켜주었다. 이때 우리는 Allocate 이후 Comparetag state로 가 캐시를 읽기 때문에 hit 시그널이 모든 메모리 조회에 대해 켜진다. 그러므로 hit signal이 켜질때마다 tb의 total\_count 변수를 1씩 더하고 miss signal이 켜질때마다 tb의 miss\_count 변수를 1씩 더했다.

total\_count-miss\_count가 hit\_count이다.

1. **replacement policy**
   1. 이번 실습에서 구현한 cache는 set당 하나의 cache line을 가지고 있다. 따라서 conflict가 발생하는 경우, 겹치는 cache line이 하나뿐이므로 tag가 겹치는 cache line을 바로 evict 해주면 된다. 따라서 별도의 replacement policy 알고리즘을 구현할 필요가 없었다.

**Discussion**

1. Naive matmul vs optimized matmul
   1. 같은 행렬의 곱셈을 연산하더라도 메모리 접근 순서에 따라 cache 효율이 달라져 성능에 차이를 보일 수 있다.
   2. 
   3. Naive matmul의 경우 일반적인 행렬곱 연산 방식대로 A의 row 한 줄과 B의 col 한 줄을 차례대로 곱하고 더하는 연산을 수행한다.
   4. 
   5. Optimized matmul의 경우 A와 B를 4\*4 tile 4개로 나누어 tile별로 행렬곱 연산을 수행한다. 순서와 관계 없이 element끼리의 곱을 더하는 연산이기 때문에 보편적인 방법의 행렬곱과 결과가 같다.
2. **Why is the cache hit ratio different between two matmul algorithms?**
   1. Optimized matmul 방식이 cache를 더 효율적으로 사용한다. 구현한 cache의 한 cache line은 16byte로 4개의 word를 담을 수 있다.
   2. Naive의 경우, A의 첫 row에 접근하는 과정을 본다. 첫 element 접근에서 miss가 나온 후 col 0~3의 element를 cache line에 저장하므로 이후 3개의 element 접근에 대해서는 hit이다. 5번째 element는 다른 set로 분류되므로 다시 miss가 발생하고, 똑같이 남은 3개의 element는 hit로 처리된다.  
      B는 col의 첫 줄을 순서대로 접근한다. cache는 인접한 address의 data 4개를 저장하게 되어 있으므로 B의 세로로 된 첫 줄에 대한 접근은 모두 miss가 발생할 것이다.  
      C에서는 첫 element에서 miss가 발생하고, 뒤의 element를 포함하여 cache에 저장된다.  
      이 연산에서 hit rate가 최대가 되기 위해서는 cache에 저장된 B의 좌측 8\*4 정보가 계속 남아 hit 되어야 한다. 하지만 이미 B의 요소가 cache의 set을 많이 차지하고 있어 A에서의 element 접근 과정에서 B의 것이 evict될 가능성이 높다. 하지만 B의 data는 곧 다시 쓰이게 되므로 miss가 필연적으로 발생할 수밖에 없다.
   3. Optimized의 경우 4\*4 tile로 matrix를 나누어 연산을 진행한다. A의 element에 대한 접근은 Naive의 것과 비슷하게 가로 4개씩 cache에 저장된다.  
      B에서 접근할 때 Naive에서처럼 모든 col의 요소를 접근하지 않고, 그 절반만 접근하여 4\*4의 element만을 cache에 저장했다. 저장한 16개의 data에 관한 연산을 우선적으로 진행한 후, 뒤의 연산에서는 이에 대한 접근을 필요로 하지 않기 때문에 cache에 다시 저장해 줄 필요가 없어진다. 연산 과정에서는 B의 element가 차지하고 있는 cache의 set이 Naive에 비해 적기 때문에 evict될 가능성이 적다. 따라서 B의 data는 각각의 tile의 연산 과정에서 tile에 속한 element에 대한 hit rate이 높아진다.
   4. hit ratio 계산할 결과 naive의 경우 0.67507, optimized의 경우0.6422가 나왔다. 위에서 서술했듯이 optimized의 hit ratio가 높은게 일반적인데 근소하지만 그 반대의 나온것은 아마 우리가 처리하는 matrix가 8x8로 매우 작고 캐시 또한 16 line의 direct mapped cache라 optimized code라 지역성을 잘 살리지 못한듯 하다. 아마 캐시의 way를 늘리면 필요한 block이 evict되는 것을 줄여 우리가 예상한 hit ratio가 나올것이다.
3. **What happens to the cache hit ratio if you change the # of sets and # of ways?**
   1. set의 수를 변경하는 경우
      1. way의 수를 늘리면 addr의 index 값이 겹쳐 evict되는 경우는 줄어든다. 이로써 이후에 쓰일 라인을 오랫동안 캐시에 보관할 수 있어 hit ratio를 높일 수 있다. 다만 way가 너무 커질경우 캐시의 bandwidth가 낮아지므로 locality를 제대로 활용하는 코드가 아니라면 추가적인 cost가 더 클 수 있다.
      2. set의 수를 늘리면 블록들이 더 다양한 인덱스에 나뉘어져 들어가므로 이 또한 hit rate를 높이는 방안이 된다. 하지만 way를 늘릴때와 같이 더 많아진 set을 찾기 위한 추가적인 cost가 든다.

**Conclusion**