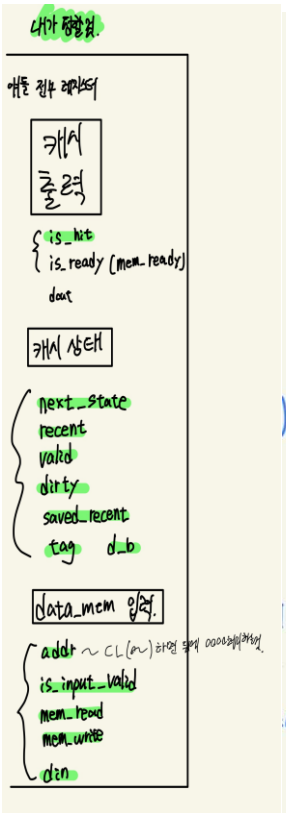
Cache

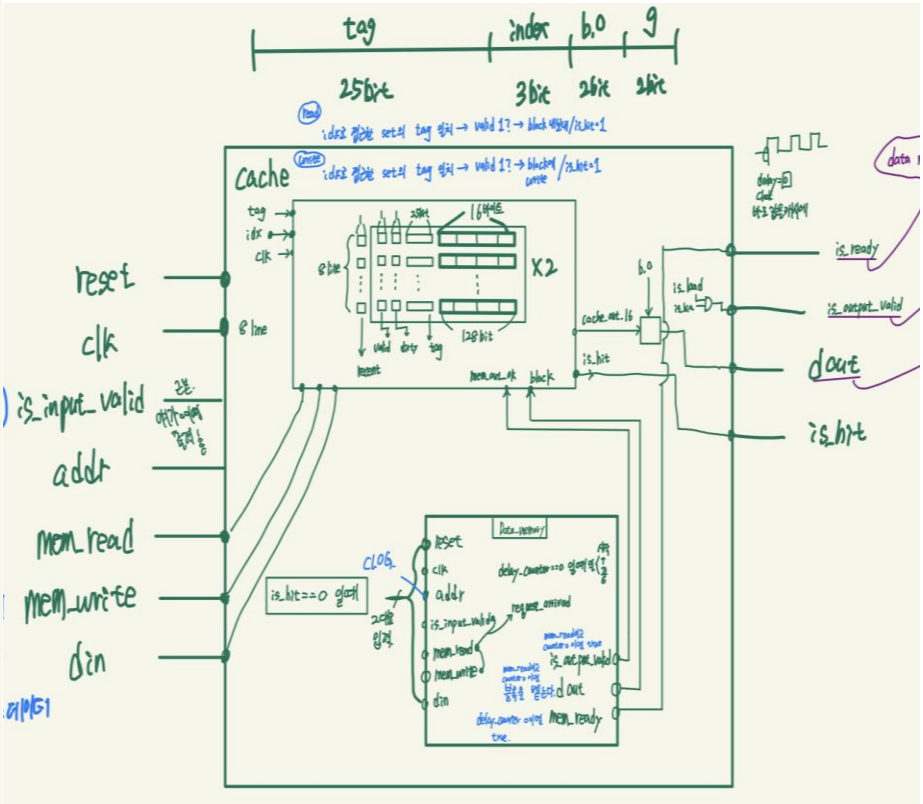
20210115 이세규 20210706 이지원

1. introduction

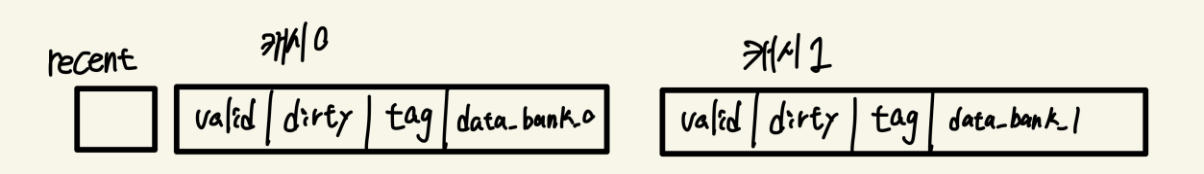
이번 과제의 목표는 캐시를 디자인하고 구현하는 것이다. 지금까지 사용했던 memory는 magic memory로 메모리 접근을 한 사이클만에 할 수 있었다. 하지만 이번 과제에서는 메모리 접근을 위해 50사이클이 소요된다. 대신에 캐싱을 통해서는 한 사이클 만에 메모리에 접근하는 것을 허용한다. 캐시의 크기는 256바이트이며 한 라인의 크기는 16바이트이다. 따라서 총 16개의 라인이 존재하고 우리 팀은 2-way associative cache를 구현하였다. 따라서 캐시의 index는 총 8개가 있으며 하나의 index에는 2개의 line이 할당되게 된다. 캐시 내부적인 동작이 어떻게 되는지, 캐시 miss가 생겼을 때 캐시 내부와 cpu에서는 각각 어떤 동작을 하는지 등 고려해야 할 사항이 적지는 않지만 차근차근 설명해보겠다.

2. design

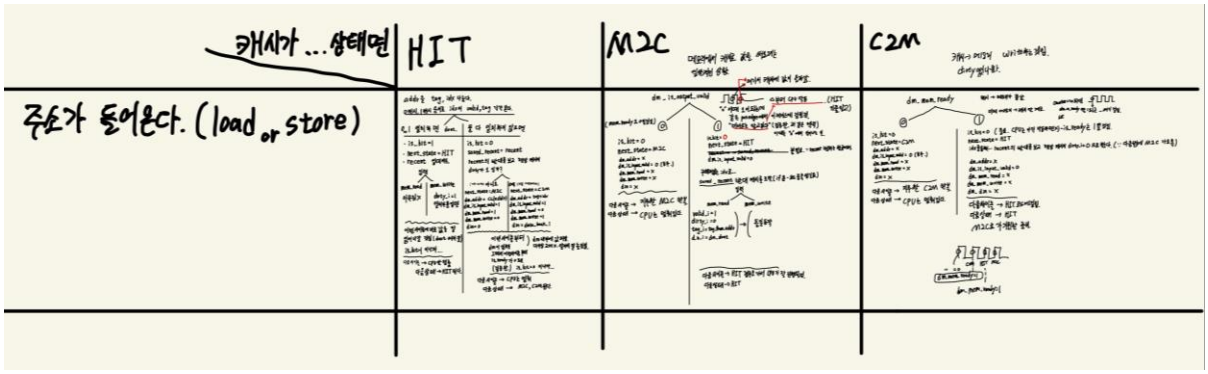




[그림1] 캐시 디자인할 때 고려할 것. [그림2] 캐시 모듈 그림



[그림3] 캐시 한 index의 구성



[그림4] 캐시 상태표

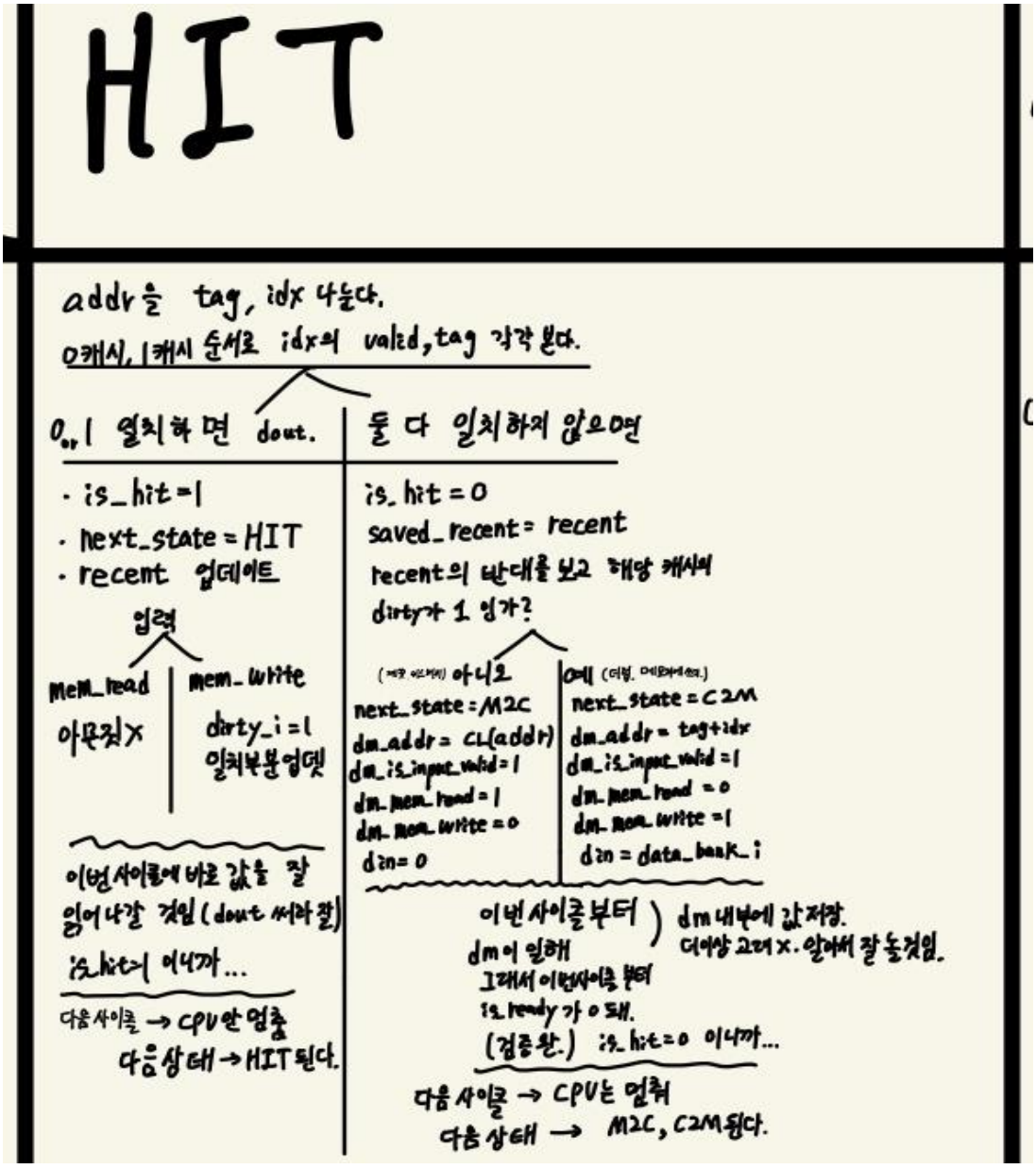
우선, 캐시의 형태는 그림2와 같은 모습이다. 대략적으로 설명해보자면, 캐시 모듈로 기존의 data memory에 들어왔어야 했던 입력이 들어오는데 우선 tag와 index 등을 통해 캐시 line에 해당 정보가 존재하는지 확인할 것이다. hit이라면 해당 라인에서 read 또는 write 작업을 할 것이고 miss라면 data memory에서 값을 읽어오는 단계로 갈 것이다. 이 단계 이전에, write allocate, write back 캐시이므로 miss가 발생하여 새로운 값을 cache line으로 들여와야 할 경우 evict되는 line이 생긴다면 이 line의 dirty의 값에 따라 캐시에서 메모리로 값을 쓰는 단계가 추가될 수도 있다. 이를 종합적으로 고려하여 그림4의 캐시 상태표를 구성하였다. 그림4에 대한 설명은 아래에서 더 하도록 하겠다.

우리 팀은 캐시를 디자인할 때 2-way associative cache를 사용하였기에 replacement policy를 정해야 했는데, 우리 팀은 캐시에서 가장 오랫동안 사용되지 않은 라인을 evict하는 방식으로 구현하였다.(LRU) 가령 동일한 index에 있는 line0과 line1이 있다고 해보자. (A)두 라인이 모두 비어있다면 line 1에 값을 먼저 저장한다. (B)그 후 이 index로 tag가 다른 주소의 값이 저장되어야 할 경우 line 0에 값을 쓴다. (C)또 이 이후에 이 index에 두 line에 저장된 것과 다른 tag를 가진 주소가 오게 되면 이번에는 가장 오랫동안 사용되지 않은 라인인 line 1을 evict한다. 만약에 (B)단계와 (C)단계 사이에 (D)이 index로 line 1의 tag와 동일한 주소가 입력되어 hit이 나온 상황이 추가된다면 (C)단계에서는 line 0이 evict될 것이다. 왜냐하면 line1이 가장 최근에 접근되어 line0이 가장 오랫동안 사용되지 않은 line이 되었기 때문이다. 이를 디자인하기 위해 각 line마다 recent를 두어 최근에 사용된 line을 기록하도록 하였다.

이렇게 2-way associative cache를 사용하였기에 direct-mapped cache와 비교하였을 때 hit ratio

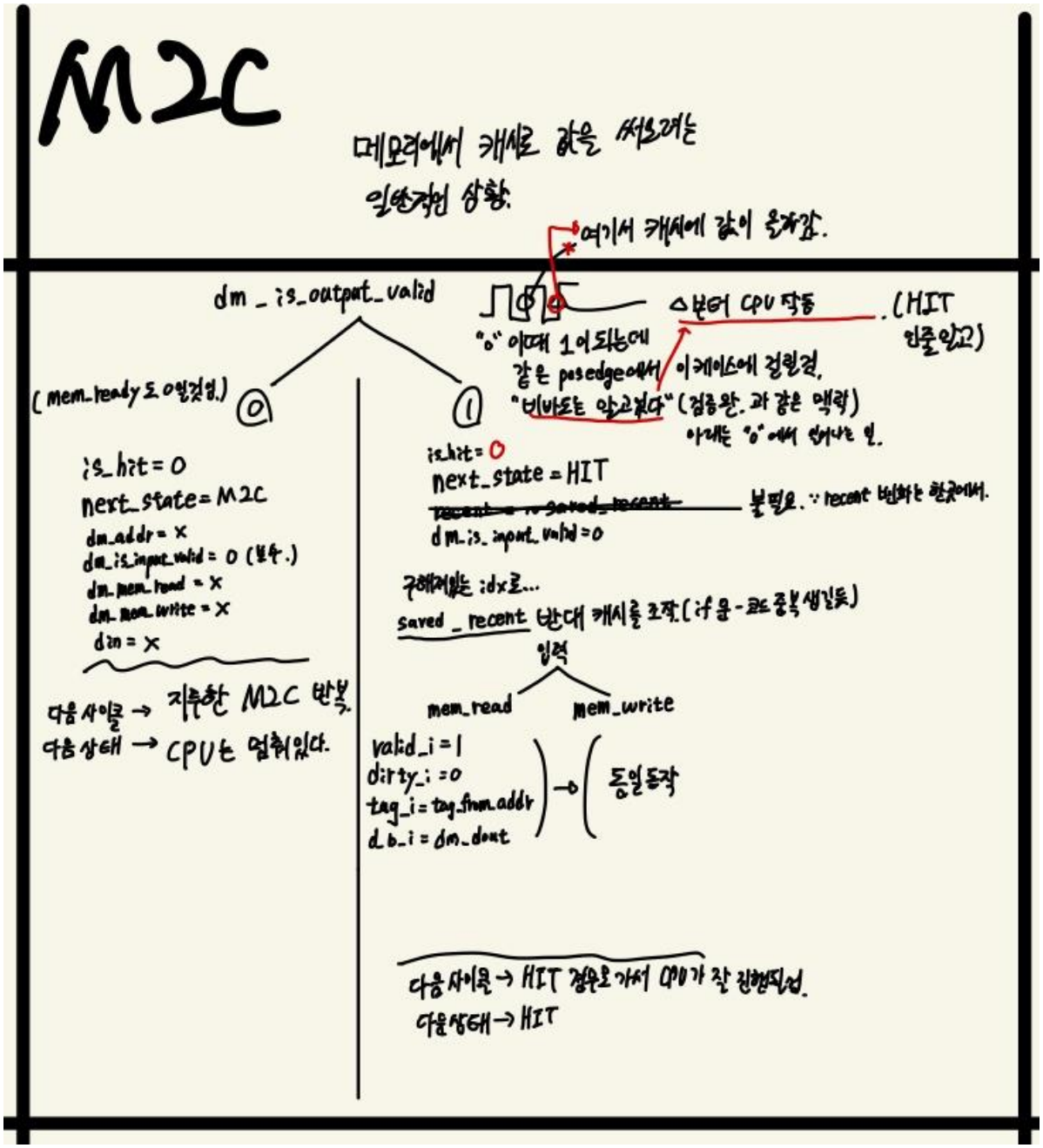
측면에서 이득을 볼 수 있다. direct mapped cache의 경우 naïve는 메모리 접근 2499번 중 miss가 812번(0.675), opt는 메모리 접근 2499번 중 miss가 894번(0.642) 생기고 2 way associative cache의 경우 naïve는 메모리 접근 2499번 중 miss가 613번(0.754), opt는 메모리 접근 2499번 중 miss가 635번(0.745) 생긴다. 따라서 2 way associative cache는 direct mapped cache보다 miss rate가 적으므로 hit ratio가 높은 것이다. 괄호 안에 hit ratio로 써두었다.

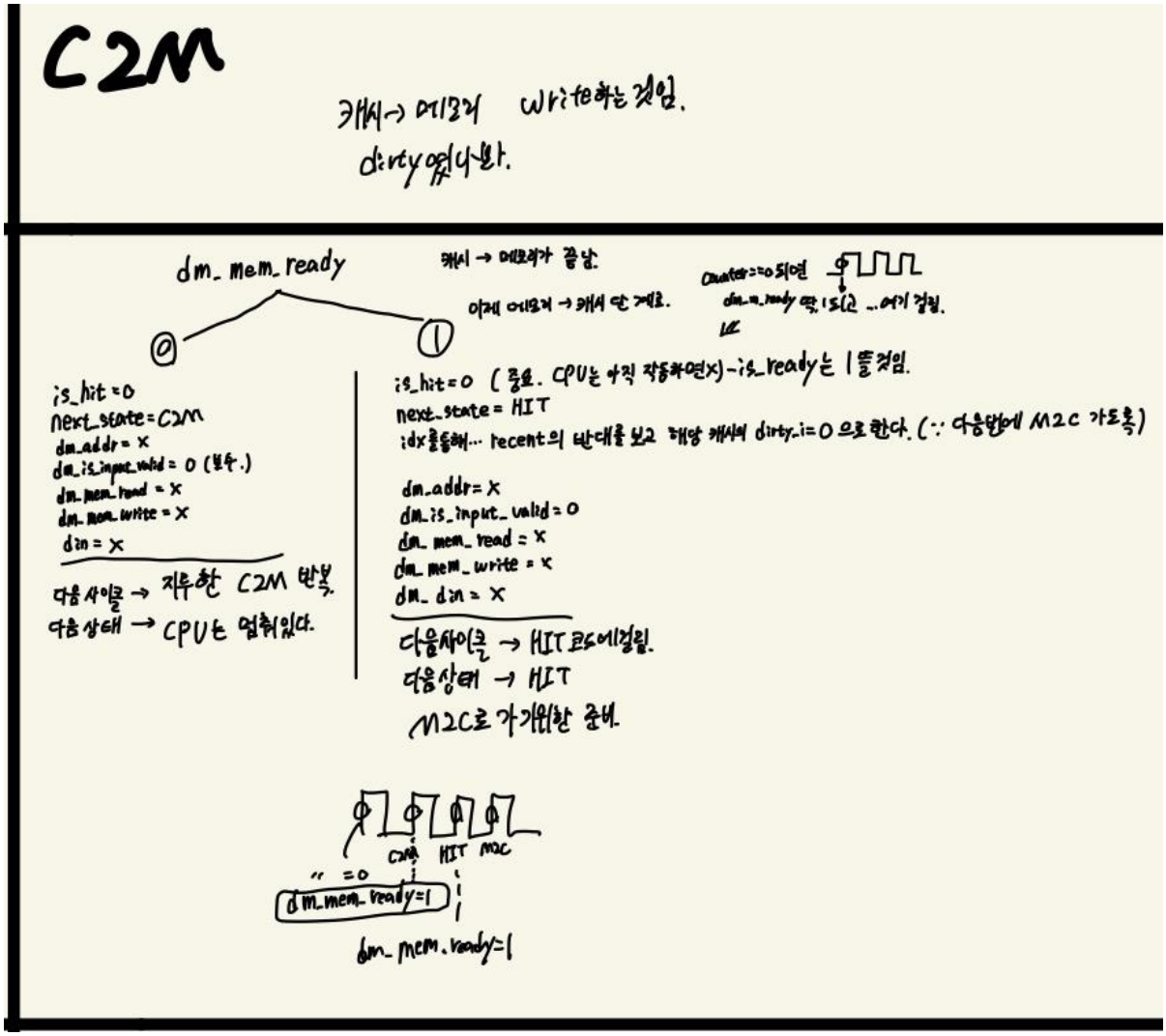
즉, 더 효율적인 구현인 것이다. 이 이유에 대해 일반적인 이유를 간단히 살펴보자. way를 늘림으로써 conflict miss를 줄일 수 있다. 1 way에서는 index가 같고 tag가 다르면 기존에 있던 데이터가 항상 evict되는데 LRU로 구현된 2 way에서는 가장 최근에 사용한 블록은 남겨두는 식이다. 따라서 충돌 미스를 줄였다고 볼 수 있다. 이는 시간 지역성이 개선된 것으로 볼 수도 있다. 왜냐하면 direct mapped cache에 비해 메모리 접근을 조금 더 시간을 갖고 하더라도 2 way라는 이유로 evict되지 않을 가능성이 늘기 때문이다.



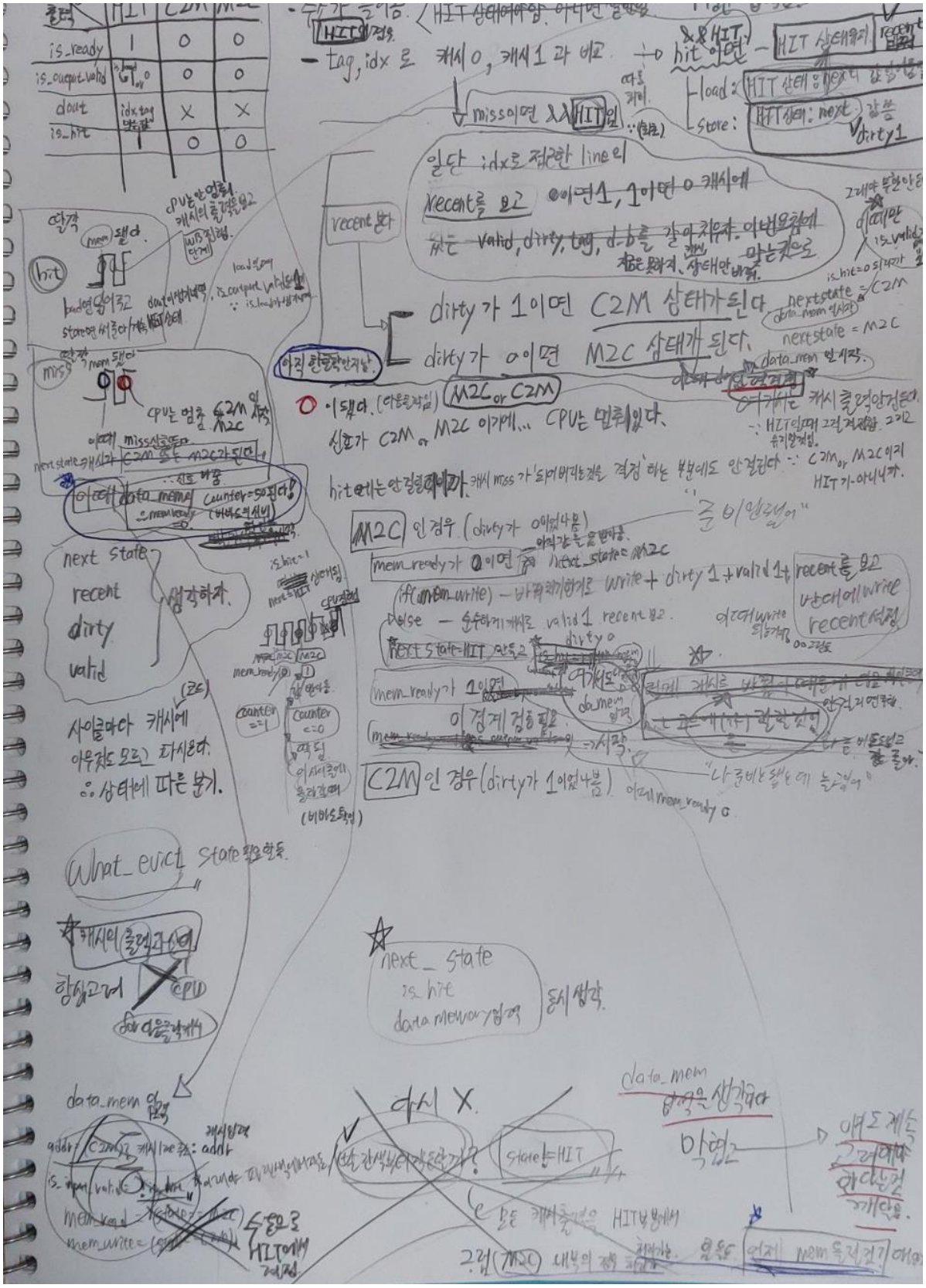
M2C

메모리에서 캐시로 값을 써주는 일련과정 상황.





위 그림은 우리 팀이 구현한 캐시의 상태에 대한 설명과 상태 이동을 어떤 방식으로 하는지 정리해 둔 것이다. 이해하기 쉽게 수기로 작성하였다. 아래 내용은 디자인의 전 과정을 나타낸 것이다. 이해하는 것에 도움이 될 수 있으니 첨부한다. 디자인 단계에서 염두 할 점은 그림1에 있는 입력, 출력, 그리고 상태와 관련된 레지스터에 적절한 값을 할당하였는지 확인하는 것이다.



3. implementation

- 클락이 올라갈 때 시작이고 다음 클락이 올라갈 때까지 값을 구성해두고 그 다음에 캐시를 업데이트 한다.

메모리의 쓰기 작업은 결국 클락이 올라가는 순간에 진행되어야만 한다. 즉 combinational하게 무엇을 써야할 지 내부 레지스터에 저장을 해두고, 실제로 쓰는 작업은 따로 진행한다. 이를 코드를 보면서 이해해보자.

```
always @(posedge clk) begin
  if(hit_0) begin
    recent[idx_from_addr] <= 0;
    if(write_0) begin
      data_bank_0[idx_from_addr] <= data_bank;
      dirty_0[idx_from_addr] <= 1;
    end
  end
  else if(hit_1) begin
    recent[idx_from_addr] <= 1;
    if(write_1) begin
      data_bank_1[idx_from_addr] <= data_bank;
      dirty_1[idx_from_addr] <= 1;
    end
  end
  else if(m2c_0) begin
    valid_0[idx_from_addr]<=1;
    dirty_0[idx_from_addr]<=0;
    tag_0[idx_from_addr]<=tag;
    data_bank_0[idx_from_addr]<=data_bank;
  end
  else if(m2c_1) begin
    valid_1[idx_from_addr]<=1;
    dirty_1[idx_from_addr]<=0;
    tag_1[idx_from_addr]<=tag;
    data_bank_1[idx_from_addr]<=data_bank;
  end
  else if(c2m_0)
  begin
    dirty_0[idx_from_addr]<=0;
  end
  else if(c2m_1)
  begin
    dirty_1[idx_from_addr]<=0;
  end
end
```

```
case(state)
  `HIT:
  begin
    if( (tag_from_addr == tag_0[idx_from_addr]) && valid_0[idx_from_addr] ) //hit in 0
    begin
      is_hit = 1;
      hit_0=1;
      next_state=`HIT;
      //load
      case (bo_from_addr)
      0: dout = data_bank_0 [idx_from_addr] [31:0];
      1: dout = data_bank_0 [idx_from_addr] [63:32];
      2: dout = data_bank_0 [idx_from_addr] [95:64];
      3: dout = data_bank_0 [idx_from_addr] [127:96];
      endcase
      //store
      if(mem_write)
      begin
          write_0=1;
          data_bank = data_bank_0[idx_from_addr];
          case (bo_from_addr)
          0: data_bank[31:0] = din;
          1: data_bank[63:32]  = din;
          2: data_bank[95:64]  = din;
          3: data_bank[127:96] = din;
          endcase
      end
```

왼쪽 코드는 실제로 캐시가 write 되는 부분이다. 여기서 내부 레지스터인 data_bank와 tag는 combinational하게 미리 정해진다. 이는 오른쪽 코드에서 확인 가능하다. 또한 각 state마다 캐시에서 어떤 부분이 바뀌어야 하는지 다르고, 2-way로 구현하였기 때문에 set(캐시0인지, 캐시1인지)을 정해야 한다. 이를 위해 hit_0, hit_1, wirte_0, write_1, m2c_0, m2c_1, c2m_0, c2m_1 시그널을 정해서 캐시를 변경해야 하는 상황이면 이 시그널 중 하나를 1로 바꾸고, 클락이 올라갈 때 특정 시그널에 해당하는 쓰기 방식을 사용하여 캐시에 쓰게 한다. 이는 왼쪽 코드에 else if 문으로 구분하여 구현하였다.

cache 내부에서 miss로 발생하는 지연은 cache 외부를 완벽하게 멈춰야 한다. cpu에는 memory_stall 시그널이 있고 이는 memory_stall = is_input_valid && (!(is_hit && is_ready && is_output_valid))로 정해진다. 즉 load나 store인 경우 is_hit, is_ready, is_output_valid 중 하나라도 0인 경우 멈춰야 하는 것이다. 이때 memory_stall로 cpu가 완벽하게 멈추게 하기 위해서 파이프라인 레지스터는 memory_stall 신호가 오면 아무 변화도 하지 않도록 구현한다. 또한 pc도

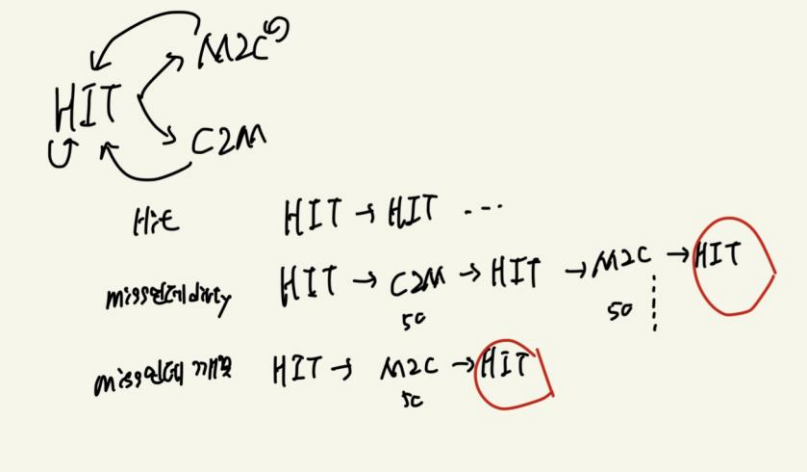
memory_stall 신호가 오면 pc값이 변경이 되지 않도록 한다.

```
// Update EX/MEM pipeline registers here
always @(posedge clk) begin
  if (reset) begin
    EX_MEM_mem_write<=0;
    EX_MEM_mem_read<=0;
    EX_MEM_mem_to_reg<=0;
    EX_MEM_reg_write<=0;
    EX_MEM_alu_out<=0;
    EX_MEM_rs2_data<=0;
    EX_MEM_inst<=0;
    EX_MEM_is_halted<=0;

    EX_MEM_pc_to_reg<=0;
    EX_MEM_current_pc<=0;
  end
  else if(memory_stall) begin end
```

```
// PC must be updated on the rising edge
PC pc(
  .reset(reset),       // input (Use rese
  .clk(clk),         // input
  .next_pc(next_pc),     // input
  .current_pc(current_pc),   // output
  .is_stall(is_stall||memory_stall)
);
```

이는 위의 코드처럼 else if(memory_stall) begin end, .is_stall(is_stall||memory_stall)로 구현한다.



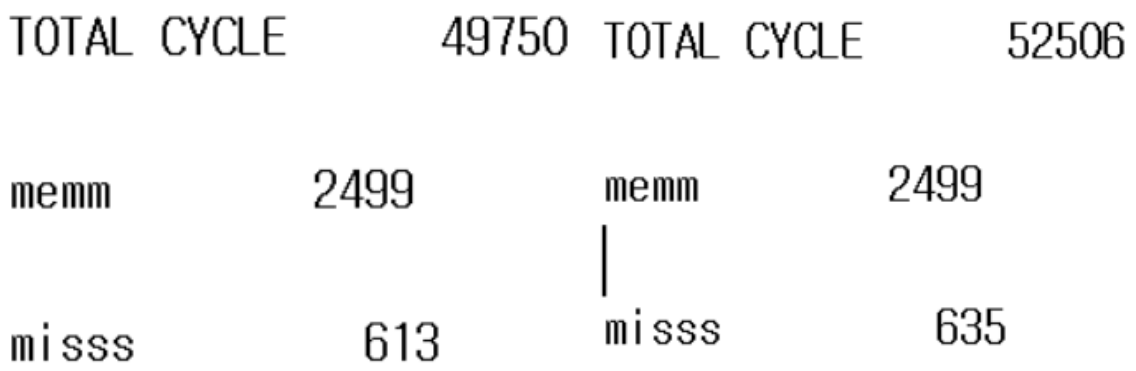
우리의 캐시의 경우, 일반적인 hit는 한 사이클안에서 이루어진다. 또한 miss인데 dirty한 캐시를 memory에 업데이트 해줘야 하는 경우는 HIT 1 + C2M 50 + HIT 1 + M2C 50 + HIT 1 = 103 사이클동안 이루어진다. 이를 자세히 살펴보면, 첫 HIT state에서는 miss가 일어났다는 사실을 인지하고 C2M으로 넘어가기 위해 data memory에 적절한 input을 설정해준다. 이후 memory에서 쓰기 작업이 일어나면서 50 사이클이 경과하고 다시 HIT state로 넘어간다. 두번째 HIT state에서는 dirty가 변경되어 있으므로 memory에서 읽어온 값으로 캐시를 업데이트하고자 M2C로 넘어간다. 이후 M2C에서 읽기 작업을 위해 50 사이클이 경과하고 다시 세 번째 HIT로 와서 잘 업데이트된 캐시에 read 또는 store를 진행한다.

miss인데 dirty하지 않은 캐시를 memory에 업데이트 해줘야 하는 경우는 HIT 1 + M2C 50 + HIT 1 = 52 사이클동안 이루어진다. memory에서 읽어온 값으로 캐시를 업데이트하고자 M2C로 넘어간다. 첫 HIT state에서는 miss가 일어났다는 사실을 인지하고 M2C으로 넘어가기 위해 data

memory에 적절한 input을 설정해준다 이후 M2C에서 읽기 작업을 위해 50 사이클이 경과하고 다시 HIT로 와서 잘 업데이트된 캐시에 read 또는 store를 진행한다.

4. discussion

- naïve_matmul vs opt_matmul



왼쪽 그림은 naïve 경우이고 오른쪽 그림은 optimal 경우이다. naïve의 경우 hit raito는 0.75470188 이고 optimal 0.74589836 이다. 0.00812044 차이가 난다. 두개는 거의 차이가 없다. 왜 그런지 알아보자.

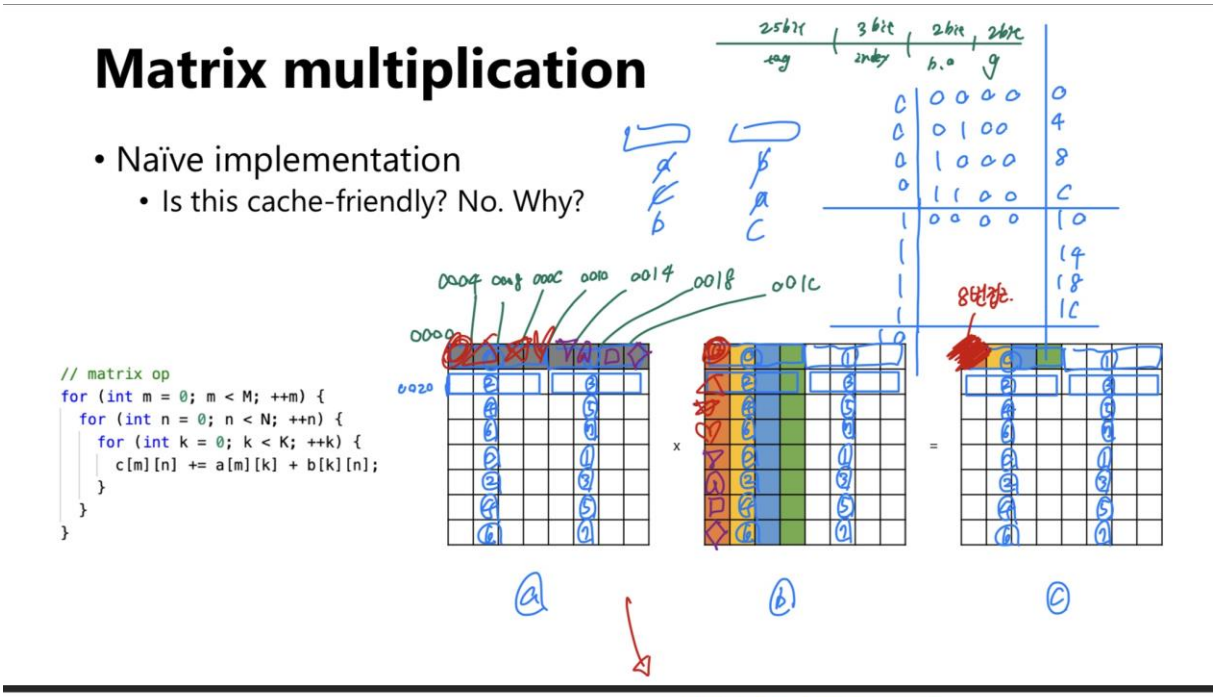
- naïve implementation이 cache friendly한가? opt implementation이 cache friendly한가?에 대한 종합적 답변 + 두 예제의 hit ratio가 다른 이유

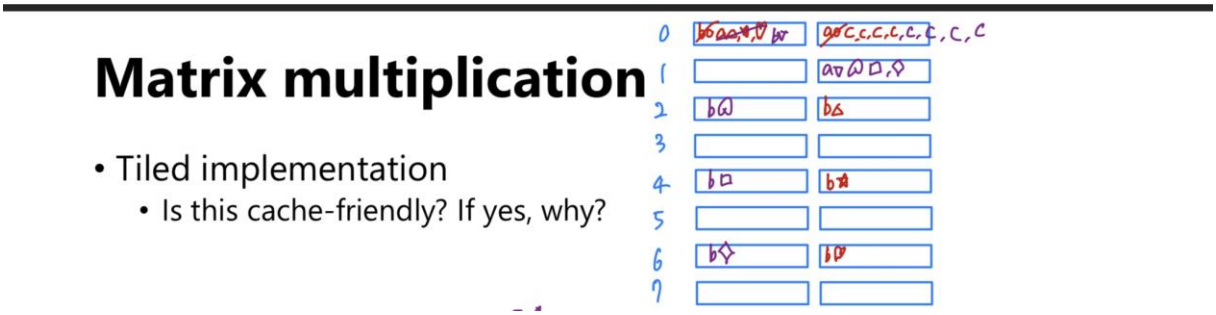
두 예제의 hit ratio가 다른 이유는 어느 쪽이 더 효율적인 것인가에 대한 답과는 무관하게 메모리에 접근하는 순서가 다르기 때문이라고 답할 수 있다. 각각에 대해 더 알아보자.

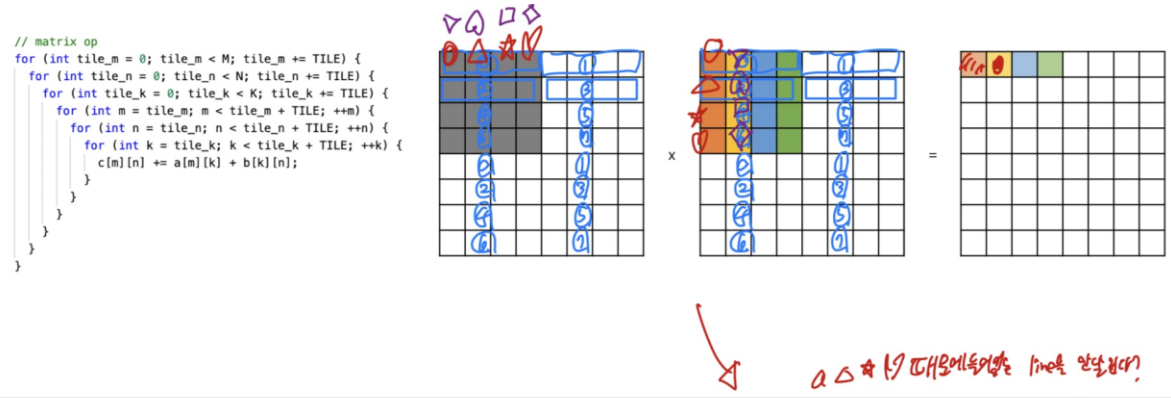
루프 타일링 또는 블로킹을 사용하는 최적화된 구현은 그렇지 않은 구현에 비해 훨씬 캐시친화적이다. 최적화된 구현에서 행렬 곱셈은 TILE상수를 사용하여 더 작은 타일 또는 블록으로 나뉘고, 외곽의 루프는 이러한 타일을 반복하고 내부 루프는 각 타일 내에서 작업을 수행한다. 계산을 더 작은 타일로 나누는 것을 통해 최적화된 구현은 데이터 지역성과 캐시 활용도를 향상시킨다. 내부 루프는 캐시에 보다 효과적으로 맞는 행렬의 하위 집합에서 작동한다. 이런 방식으로 하면 캐시 내에서 데이터를 재사용할 가능성이 높아지기 때문에 캐시 누락이 줄어든다. 캐시 라인의 크기는 16바이트이고 각 캐시 인덱스에는 2개의 라인이 있으므로 타일 크기는 캐시 라인 크기 또는 그 배수와 일치하도록 적절한 수치로 해야 한다. 이렇게 하면 캐시 라인을 더 잘 활용할 수 있고 캐시 충돌의 가능성이 줄어든다. 최적화된 구현에서 타일 크기는 4로 설정된다. 이것은 4x4 부분 행렬이 단일 캐시 라인에 맞을 수 있기 때문에 더 성능이 좋다. 최적화된 구현의 루프는 각 타일에서 계산이 순차적으로 수행되기에 캐시 재사용을 최대화하는 방식으로 구성된다.전반적으로 루프 타일링을 사용해 최적화한 구현은 캐시 내에 잘 맞는 작은 타일을 활용하여 캐시 지역성

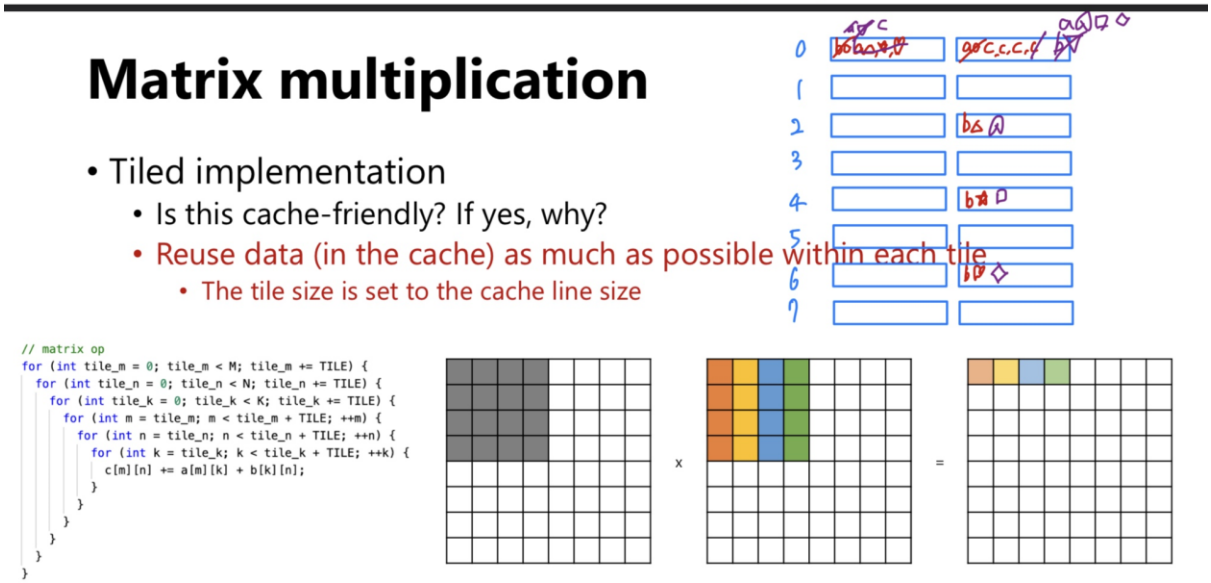
을 좋게 한다. 캐시 미스를 줄이고 캐시 성능을 향상시켜 최적화되지 않은 구현에 비해 캐시 친화적이다.

메모리 접근 패턴은 a 메모리에 접근, b 메모리에 접근, c 메모리에 접근 순이고 a는 row major하게, b는 column major하게, c는 column major하게 행렬을 반복한다. 캐시 라인의 크기는 16바이트로 행렬의 한 행의 크기인 크기인 4바이트*8 인 32바이트에 비해 작다. 이는 각 캐시 라인이 매트릭스의 모든 연속된 요소를 저장하고 있을 수는 없음을 의미한다. 행렬의 특정 요소에 액세스할 때 캐시는 해당 요소를 포함하는 전체 캐시 라인을 로드하는데 다른 요소가 캐시 라인을 없애기 전에 동일한 캐시 라인 내의 인접 요소에 대한 후속 액세스가 이루어지면 캐시 지역성의 이점을 얻을 수 있다. 하지만 동일한 캐시 라인 내에 있지 않은 요소에 대한 후속 액세스가 이루어지면 evict도 발생하고 추가적인 메모리 액세스도 생긴다. naive implementation에서 루프는 캐시 라인의 크기를 고려하지 않은 방식으로 행렬에 접근한다. 따라서 캐시 미스가 자주 발생할 가능성이 높으며, 특히 매트릭스가 더 크고 캐시 크기가 작다면 이러한 효과는 더 커질 것이다. 하지만 이 예시는 matrix가 충분히 크지 않고 캐시의 크기가 작은 상황이기 때문에 naive가 optimized implementation보다 더 높은 hit ratio가 나온 것이다. 이해를 위해 아래 그림을 첨부한다. 순서대로 naïve implementation, opt implementation이다.









- 캐시 set 수와 캐시의 way 수를 바꿨을 때 일어나는 일

캐시의 각 set는 메모리의 고유한 부분을 보유할 수 있다. 위 그림2의 윗부분의 bit구성을 보면 알 수 있듯이 기본적으로 index를 기준으로 캐시에 접근하기 때문이다. set이 클 겨우 conflict miss가 날 가능성이 줄어들고 hit ratio가 올라간다.

way 수는 동일 index에 값을 저장할 수 있는 추가적인 다른 공간의 수이기 때문에 conflict를 날 것을 n번 막을 수 있게 된다.(n way라면) 즉, 동일한 index가 캐시에 저장될 수 있게 된 것이므로 way 수가 커지면 conflict 수가 줄어 hit ratio가 올라간다.

요약하면, set 수든, way 수든 크게 하면 캐시의 hit ratio는 올라간다. 하지만 실제 구현에서는 캐시의 용량 문제, 캐시 접근 속도 문제와 타협하여 적절한 수의 set 수와 way수를 결정하여야 한다.

5. conclusion

opt_matmul_unroll.mem: 52506 사이클

naive_matmul_unroll.mem: 49750 사이클

recursive_mem.txt: 2019 사이클

non-controlflow_mem.txt: 150 사이클

basic_mem.txt: 139 사이클

loop_mem.txt: 426 사이클

non_control_flow_mem.txt: 163 사이클

ifelse_mem.txt: 147 사이클

위는 이번 구현의 결과이다.

기존 magic memory를 사용하던 것에서 50사이클씩 걸리는 memory를 사용함에 따라 전체적으로 cpu가 굉장히 느려지게 된다. 우리의 hit ratio는 최종적으로 약 75%가 나왔다. 즉 50사이클 이상 걸릴 명령어들 4개 중 3개를 더욱 빨리 해결한 것이다. 이는 캐시 사용의 중요성을 직접 구현을 통해 증명한 것이다.

또한 지난 5개의 랩과제를 통해 ALU부터 한 단계씩 cpu를 업그레이드해가며 성능을 개선시켰고 캐시를 사용하는 파이프라인 cpu까지 구현에 성공하였다. 뿌듯하다!?