Computer Architecture (CSED 311), Spring 2025
Lab 5: Cache

구현웅 (20210940), 김민서 (20220826)
5/27, 2025

1. Introduction

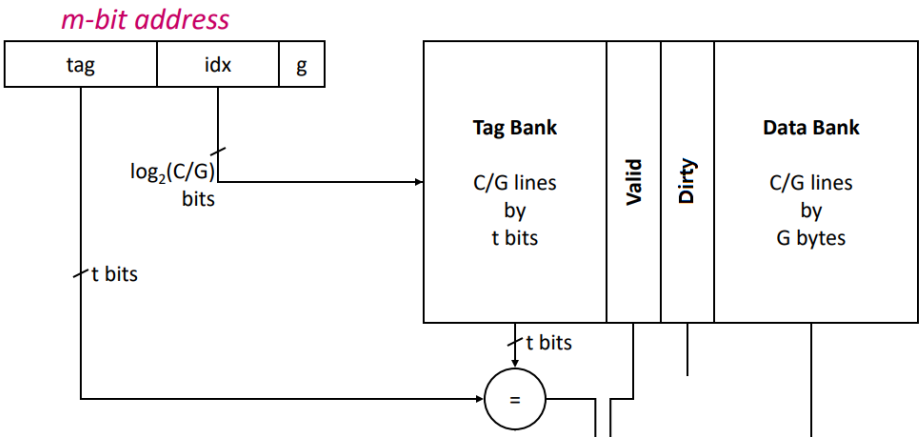
본 랩에서는 pipelined CPU를 바탕으로 메모리 접근에 대해 cache를 구현하는 것을 목표로 한다. 한 사이클에 메모리 접근이 가능한 magic memory를 cache로 대체함으로 생기는 딜레이 문제는 stall과 유사한 방식으로 아키텍쳐 수정을 가해 해결하며, cache는 하나의 cache index에 하나의 block만을 가지는 direct-mapped cache로 구현한다.

2. Design

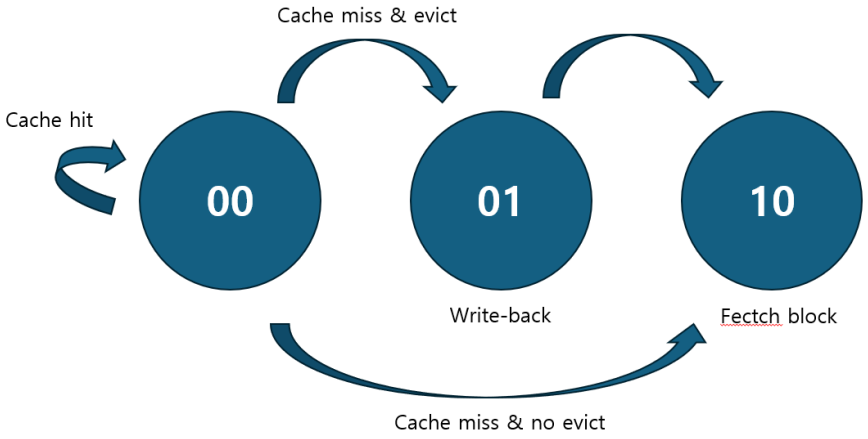
Cache 구현에 있어 크게 두 가지 사항을 설계해야한다. 첫 번째는 cache의 내부 디자인이며, 또 다른 하나는 한 사이클에 cache가 데이터 접근을 보장해 줄 수 없기에 내부적으로 cache의 데이터가 준비될 때까지 기다리는 동작이 필요하다.

2.1. Cache Design

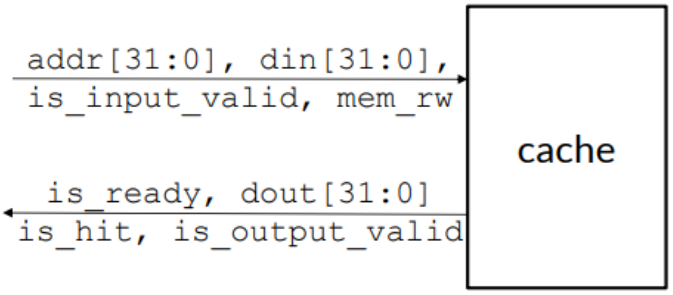
본 랩에서 먼저 구현할 cache는 write-back의 direct-mapped로 cache의 하나의 index에는 tag와 data block 그리고 컨트롤을 위한 valid bit와 dirty bit를 가진다. 각각의 memory address는 tag, cache index, block offset으로 나누어진다.



기본적으로 memory address가 주어지면 cache index로 cache entry를 참조하며 tag가 서로 같은지를 비교해 block의 데이터가 주어진 address의 것인지 확인한다. 이때 valid bit는 cache의 tag와 block data의 valid 여부를 나타내며, dirty는 data block이 수정되었다는 것을 의미하며 cache entry가 evict 될 때 메모리에 write-back를 위해 준비되었다.



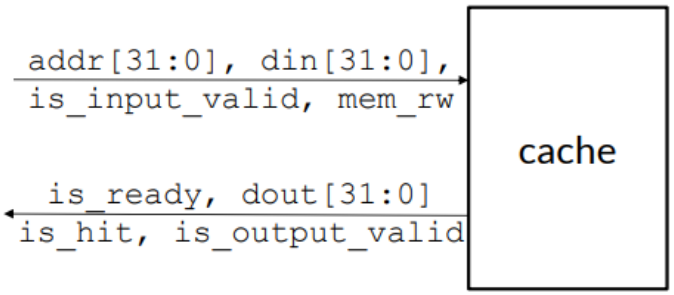
Cache는 내부 동작을 관리하기 위해 3개의 상태를 지닌다. 00은 가장 초기 상태로 입력된 address와 tag를 비교하는 역할을 하며, cache hit에 경우 바로 read, write를 수행한다. 01은 evict된 데이터를 memory에 요청해 write-back하는 단계이며 cache miss로 evict된 entry의 dirty bit가 활성화 되어있다면 이 단계로 들어온다. 10은 memory에 read를 요청하고 cache entry를 설정하는 단계이며, 01 write-back 후에 진입하거나 00에서 cache miss이며 write-back 필요없을 때 이 단계로 바로 넘어온다.



Cache의 인터페이스는 위와 같으며 이전과는 다르게 is_input_valid, is_hit, is_ready, is_output_valid 등의 신호를 주고 받는다. is_input_valid는 다양한 instruction에 의해 read 혹은 write 신호가 활성화 되었을 때 활성화된다. is_ready의 경우 cache를 사용할 수 있다는 의미이며 state 01 처럼 추가적인 내부 동작이 필요할 때는 0으로 외부의 접근을 제한한다. is_outpt_valid의 경우에도 유사하게 cache의 dout이 valid하다는 의미로 cache에서 적절한 block에 대해 read를 수행했을 때만 활성화된다.

2.2. Modify pipelined CPU

Cache는 memory에 write-back과 memory에 데이터를 읽어 fetch를 하는 동작으로 인해 memory가 한 사이클 동작을 지원해도 cache의 동작에는 더 많은 cycle이 요구될 수 있다. 따라서 memory access를 1cycle로 가정하고 제작한 pipelined에도 수정이 필요하다.



is_ready, is_hit, is_output_valid 등으로 cache가 사용불가능하며 요청에 대한 내부적 처리를 수행하는지 판단할 수 있으며, 만약에 cache를 사용할 수 없는 상태라면 pipelined을 멈추고 대기해야한다. Out-of-order를 지원하지 않으므로 단순하게 stall로 cache가 요청을 완료할 때까지 기다리면 된다.

3. Implementation

설계를 바탕으로 구현한 cache와 pipelined CPU 수정된 부분은 다음과 같다.

3.1. Cache

```
// | TAG | VALID | DIRTY | DATA |
localparam integer BankWidth = TagWidth + 2 + BLOCK_SIZE * 8;
localparam integer BankIdxWidth = $clog2(CAPACITY / BLOCK_SIZE);
localparam integer TagWidth = 32 - $clog2(CAPACITY);
localparam integer BlockOffsetWidth = $clog2(BLOCK_SIZE / GRANULARITY);
```

Cache의 한 entry는 tag, valid bit, dirty bit, data로 구성되어 있다.

```
assign {tag, bank_idx, block_offset} = addr[31:$clog2(GRANULARITY)];
assign dmem_din = bank[bank_idx][BLOCK_SIZE*8-1:0];
assign dout = bank[bank_idx][(block_offset*GRANULARITY*8)+:32];
```

입력으로 들어오는 address는 각각의 tag, cache index, block offset으로 파싱되며, dout은 asynchronous read의 방식으로 생성되며 is_output_valid로 유효성을 판단한다.

```
end else if (mem_read) begin
  if (state == 2'b00 && bank[bank_idx][BankWidth-1-:TagWidth] == tag) begin
    // no fetch from memory
    is_output_valid <= 1;
    is_hit <= 1;
    is_ready <= 1;
    is_delay <= 1;
```

read의 입력에 대해 위와 같이 tag가 서로 같다면 이는 hit이며 바로 dout를 출력으로 사용할 수 있다. 이에 대해 신호를 설정해주는 부분이다.

```
end else begin
  if (state == 2'b00) begin
    if (bank[bank_idx][BankWidth-TagWidth-2] == 1) begin  // dirty
      // write back
      is_output_valid <= 0;
      dmem_read <= 0;
      dmem_write <= 1;
      dmem_is_input_valid <= 1;
      dmem_addr <= {
        bank[bank_idx][BankWidth-1-:TagWidth], bank_idx, {$clog2(BLOCK_SIZE) {1'b0}}
      };
      is_hit <= 0;
      is_ready <= 0;
      state <= 2'b01;  // next: fetch_mem
      is_delay <= 1;
```

tag가 서로 다르다면 cache entry는 evict해야하며 만약 dirty bit가 세팅되어 있다면 이를 write-back해야한다. 위의 if문은 이를 표현하고 있으며 write-back를 위한 주소를 설정하고 dmem bit를 세팅해 memory에 write를 요청한다.

```
    end else begin  // not dirty
      // no write back, just fetch from memory
      is_output_valid <= 0;
      dmem_read <= 1;
      dmem_write <= 0;
      dmem_is_input_valid <= 1;
      dmem_addr <= addr & {{(32 - $clog2(BLOCK_SIZE)) {1'b1}}, {$clog2(BLOCK_SIZE) {1'b0}}};
      is_hit <= 0;
      is_ready <= 0;
      state <= 2'b10;
      is_delay <= 1;
    end
```

하지만 dirty bit이 세팅되어 있지 않는다면 write-back를 수행하지 않고 state 10으로 바꾸고 바로 memory에서 cache로 block을 페체할 준비를 한다.

```
end else if (state == 2'b01 && dmem_is_ready) begin
  // fetch from memory
  is_output_valid <= 0;
  dmem_read <= 1;
  dmem_write <= 0;
  dmem_is_input_valid <= 1;
  dmem_addr <= addr & {{(32 - $clog2(BLOCK_SIZE)) {1'b1}}, {$clog2(BLOCK_SIZE) {1'b0}}};
  state <= 2'b10;  // next: end_fetch
  is_delay <= 1;
```

state 01에서는 write-back을 수행되었고 다음 memory에서 cache로 data를 페치하기 위한 준비를 한다.

```
end else if (state == 2'b10 && dmem_is_output_valid && dmem_is_ready) begin
  is_output_valid <= 1;
  bank[bank_idx] <= {tag, 1'b1, 1'b0, dmem_dout};
  dmem_read <= 0;
  dmem_write <= 0;
  dmem_is_input_valid <= 0;
  is_hit <= 1;
  is_ready <= 1;
  state <= 2'b00;  // next: initial (write_back)
  is_delay <= 1;
end
```

마지막으로 state 10에서는 memory에 요청한 cache block data가 준비되었는지 확인하고, bank의 데이터를 세팅하는 작업을 수행한다. 마지막으로 state 00로 되돌리며 cache는 다시 입력을 받을 준비를 한다.

Cache write의 경우에도 read와 매우 유사하게 동작한다.

```
end else if (mem_write) begin
  if (state == 2'b00 && bank[bank_idx][BankWidth-1-:TagWidth] == tag) begin
    is_output_valid <= 1;
    bank[bank_idx][BankWidth-TagWidth-2] <= 1;  // set dirty
    bank[bank_idx][(block_offset*GRANULARITY*8)+:32] <= din;
    is_hit <= 1;
    is_ready <= 1;
    is_delay <= 1;
```

tag 값이 일치해 hit가 난 경우에 위와 같이 bank data를 업데이트하며 추가로 dirty bit를 세팅한다.

```
end else if (state == 2'b10 && dmem_is_output_valid && dmem_is_ready) begin
  // write to the bank line
  is_output_valid <= 1;
  bank[bank_idx] <= {tag, 1'b1, 1'b1, dmem_dout};  // dirty
  bank[bank_idx][(block_offset*GRANULARITY*8)+:32] <= din;
  dmem_read <= 0;
  dmem_write <= 0;
  dmem_is_input_valid <= 0;
  is_hit <= 1;
  is_ready <= 1;
  state <= 2'b00;
  is_delay <= 1;
end
```

memory에 write-back를 수행하고 cache에 들어온 write 요청을 처리하는 state 10에서도 bank에 데이터를 세팅하고 dirty bit를 활성화하는 것을 볼 수 있다.

Cache는 hit가 발생한다면 state 00를 순회하며 바로 바로 값을 읽거나 쓰는 반면, miss가 발생하면 write-back를 수행하는 state 01, memory의 값을 cache에 fetch하는 state 10를 돌아가며 miss난 entry를 세팅하고 읽기, 쓰기를 수행한다.

3.2. Modify pipelined CPU

Cache를 지원하기 위해 CPU에 가한 수정은 매우 단순하다.

```
always @(*) begin
  if (cache_is_output_valid && cache_is_ready) begin
    cache_is_input_valid = (ID_EX_mem_read || ID_EX_mem_write);
  end else begin
    cache_is_input_valid = (EX_MEM_mem_read || EX_MEM_mem_write);
  end
end

always @(*) begin
  if (EX_MEM_mem_read || EX_MEM_mem_write) begin
    is_cache_stall = !(cache_is_output_valid && cache_is_hit && cache_is_ready);
  end else begin
    is_cache_stall = 0;
  end
end
```

먼저 cache의 입력으로 들어가는 is_input_valid의 경우 memory 접근하는 적절한 stage에서 read와 write의 활성 여부로 판단 할 수 있다.

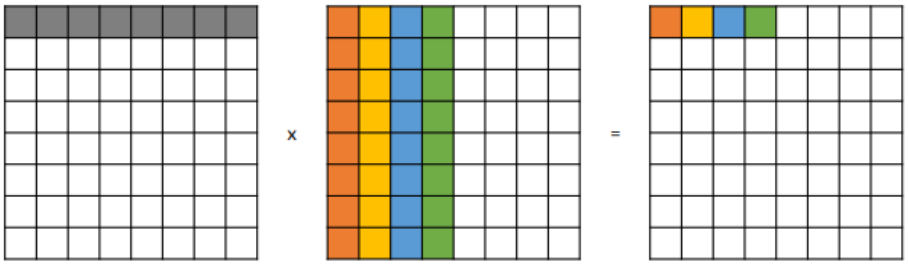
Cache가 요청에 대해 내부 동작을 수행하고 있어 pipeline이 멈춰야하는 경우는 is_output_valid와 is_hit, is_ready를 통해서 판단할 수 있으며 is_cache_stall로 생성한다. is_cache_stall은 data hazard의 stall과 같은 방식으로 register를 제어함으로 pipeline을 멈추고 cache의 동작을 기다린다.

4. Discussion

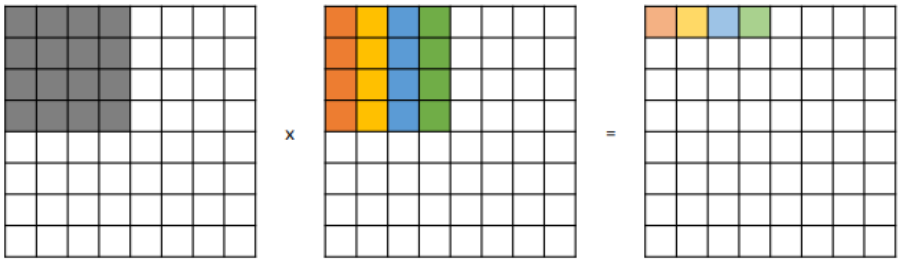
4.1. Analyze cache hit ratio

본 단락에서는 주어진 행렬 테스트 케이스에서 구현한 cache 설계의 hit rate를 분석해보고자 한다. 먼저 cache의 사이즈는 256 bytes, block size는 16 bytes이고 granularity는 4 bytes다. 각각의 행렬의 원소인 int는 4 bytes 데이터로 block 하나에는 총 4개의 int가 저장될 수 있다. 각각의 address는 28bit의 tag, 4bit의 cache index, 4bit의 block offset를 가진다고 하자.

Naïve algorithm의 경우에는 각각의 행과 열을 곱하고 특정 레지스터에 더한다고 하고, 각각의 행렬은 겹치지 않는 cache index를 가진다고 가정해보자.



이 경우에는 행렬의 크기 N에 대해 N=4 정도로 충분히 작아 각각의 행렬이 독립적인 cache 공간을 할당받는다면 cold miss외에는 miss가 거의 나지 않는다. 하지만 N=64로 크기가 커진다면 행렬의 행은 모두 같은 tag를 가지게 되므로, 특히 두 번째 행렬의 열 참조로 인해 매번 hit miss가 발생할 것이며 1/3이하로 hit rate가 감소할 것이다.



조금 더 cache 친화적인 알고리즘의 경우에는 N이 충분히 작은 경우에는 마찬가지로 높은 hit rate를 보여줄 것이며, 가장 왼쪽과 오른쪽의 행렬의 경우에는 N이 커져도 한 번에 연산하는 행의 수가 낮으므로 조금 더 높은 hit rate를 보여줄 것이다. 하지만 가운데의 열 참조 행렬의 경우에는 마찬가지로 N=16인 경우 매번 참조마다 miss가 발생하므로 전반적인 hit rate는 개선이 없을 것이다. 즉, direct-mapped cache이므로 개선된 알고리즘으로도 높은 hit rate를 기대하기는 힘든 것이다.

하지만 4-way associated의 cache를 사용했다고 가정한다면 최적화 없는 알고리즘의 경우 4번째 열까지는 동시에 tag가 저장되지만 그 다음부터는 계속 miss와 evict이 발생할 것이다. 하지만 최적화 알고리즘에서는 각각의 계산 블록 마다 4개의 서로 다른 tag를 cache에 저장할 수 있으므로 계산 단위내에서는 열 참조에 대해 cache miss가 발생하지 않을 것으로 기대할 수 있다. 따라서 n-way associated로 cache를 설계했다면 캐쉬 친화적 알고리즘의 효과를 크게 볼 수 있을 것이다.

5. Conclusion

본 랩에서는 direct-mapped cache를 구현하고 이를 지원하기 위해 pipelined를 수정했다. Cache는 제공된 사양의 인터페이스를 통해 CPU와 상호작용하며, 내부적으로 memory에서 cache block를 fetch하고 conflict의 경우에는 dirty bit를 확인하고 write-back를 수행한다. 그리고 행렬 연산에 대해 캐시 친화적인 알고리즘의 hit rate를 분석함으로써 associative cache의 유용성도 확인할 수 있었다. 이 과정을 통해 memory hierarchy를 더욱 깊이 이해하고, cache등의 memory unit이 architecture에서 결합되고 사용되는지 배울 수 있는 기회가 되었다.