2024 Spring CSED311 Lab 5 Report

Team ID: 67735
20220312 박준혁, 20220871 홍지우

명예서약 (Honor Code)

나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.
I completed this programming task without the improper help of others.

1 Introduction

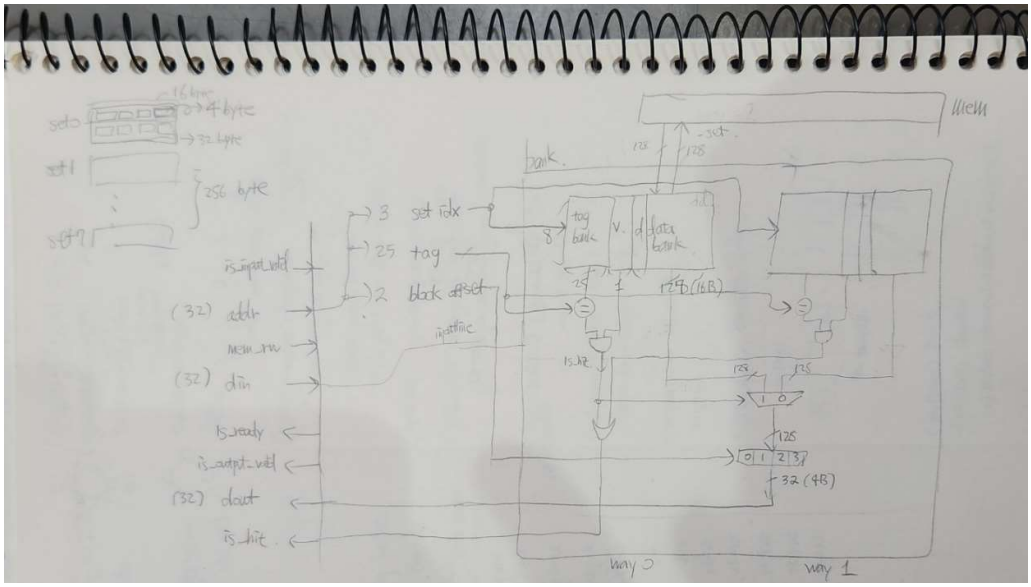
1.1 Direct-mapped Cache를 구현하였다.

2 Design

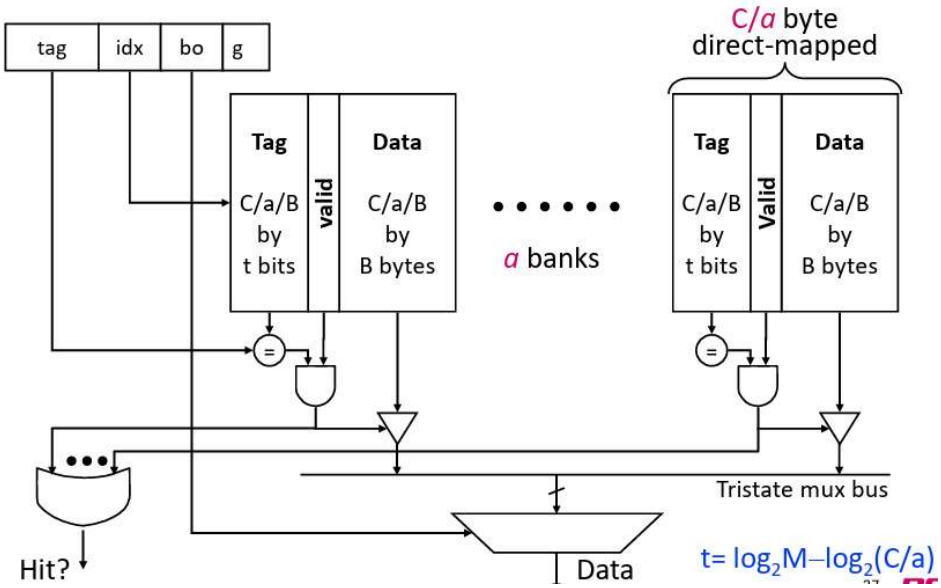
2.1 설계 내용

Associative로 구현하고 싶었으나 시간상 Direct-mapped로 구현하게 되었다

아래는 2-way로 설계한 그림이다. select하는 로직을 통째로 제외하면 본 과제에서 구현한 것과 동일한 구조를 가진다.



아래는 참고한 사진이다.



3 Implementation

3.1 캐시 구조

3.1.1 입출력

```
Cache cache(
    //input
    .reset (reset),
    .clk (clk),
    .is_input_valid (cache_is_input_valid),
    .addr(EX_MEM_alu_out),
    .mem_rw((EX_MEM_mem_read && !EX_MEM_mem_write) ? 0 :
         ((!EX_MEM_mem_read &&  EX_MEM_mem_write) ? 1 : 0)),
    .din(EX_MEM_dmem_data),
    //output
    .hit_count(hit_count),
    .miss_count(miss_count),
    .is_ready(cache_is_ready),
    .is_output_valid(cache_is_output_valid),
    .dout(dmemOutput),
    .is_hit(cache_is_hit)
);
```

위와 같이 입력과 출력이 구성되어 있다.

MEM stage에서 read, write를 구별하여 캐시에 접근하는 방식이다.

cache_is_input_valid의 경우는 다음과 같다.

```
assign cache_is_input_valid = EX_MEM_mem_read | EX_MEM_mem_write;
```

3.1.2 캐시 뱅크

```
CacheBank bank (
  .bank_active(1),
  .reset(reset),
  .clk(clk),
  .mem_rw(mem_rw),
  .addr(addr),
  .data_ready(is_output_valid),
  .is_input_valid(is_input_valid),
  .data_write_back_complete(data_write_back_complete),
  .input_set(dmem_output_set), //128 bit
  .input_line(din), //32 bit
  .output_set(bank_output_set), //128 bit
  .output_line(bank_output_line), // 32 bit
  .output_addr(bank_write_back_addr), // 32 bit
  .dmem_read(bank_dmem_read),
  .dmem_write(bank_dmem_write),
  .cache_state(bank_state),
  .is_hit(bank_is_hit)
);
```

```
// parse address
wire [24:0] inst_tag;
wire [2:0] set_index;
wire [1:0] block_offset;
assign inst_tag = addr[31:7];
assign set_index = addr[6:4];
assign block_offset = addr[3:2];
// byte offset : [1:0]

// bank registers
reg [24:0] tag_bank [7:0];
reg valid_bank [7:0];
reg dirty_bank [7:0];
reg [127:0] data_bank [7:0];
```

하나의 bank는 위와 같다. bank 안에는 tag, valid, dirty, data를 보관하고 있다

크기는 다음과 같이 정의되어 있다

```
// C = 256 bytes -> logC = 10
// a = 2
// B = 16 byte (128 bit) -> block offset = log(B/G) = 2 bit
// G = 4 byte-> byte offset = 2 bit [1:0]
// C/a/B = 8 -> set index= 3 bit
// t = 25 bit
```

3.1.3 Dmem과의 연결

```
// Instantiate data memory
DataMemory #(.BLOCK_SIZE(LINE_SIZE)) data_mem(
  .reset(reset),
  .clk(clk),

  .is_input_valid(is_input_valid && request_counter<5),
  .addr((dmem_addr>>(`CLOG2(LINE_SIZE)))),        // N
  .mem_read(bank_dmem_read),
  .mem_write(bank_dmem_write),
  .din(bank_output_set),

  // is output from the data memory valid?
  .is_output_valid(is_output_valid),
  .dout(dmem_output_set), //128 bit
  // is data memory ready to accept request?
  .mem_ready(is_data_mem_ready)
);
```

dmem과는 다음과 같이 와이어링 되어 있다.

캐시에서 요청하는 부분을 is_input_valid를 통해 조절한다.

3.2 Hit, Miss Policy

3.2.1 Hit

```
// hit
assign is_hit = (inst_tag == tag_out) && valid_out;
```

Hit 유무는 바로 async하게 알 수 있다.

3.2.2 Miss

Miss관련해서는 state machine을 사용하였다.

4개의 state를 사용, dirty라인을 교체하여 메모리에 쓰고 (1)

메모리의 값을 read해서 캐시에 불러오고 (2)

위 두가지 동작을 를 구분, 연속적으로 작동시키기 위해 그렇다.

(CACHE_IDLE, CACHE_WRITE_BACK_REQUEST,

CACHE_WRITE_ALLOCATE_REQUEST, CACHE_DATA_RECIEVED)

3.2.2.1 Write-Back

```
else begin
  if (dirty_bank[set_index] == 1) begin
    // write-back
    cache_state<=`CACHE_WRITE_BACK_REQUEST;
    output_set <= data_bank[set_index];
    output_addr <={tag_out, set_index, 4'b0000};
    dmem_read <= 0;
    dmem_write <= 1;
  end
```

Miss 발생시, Dirty가 있다면 교체해주는 로직이다.

Dmem의 원래의 주소(output_addr)에 바뀐 set을 통째로 전달한다

3.2.2.2 Allocate

```
    else begin
      // allocate
      cache_state<=`CACHE_WRITE_ALLOCATE_REQUEST;
      dmem_read <= 1;
      dmem_write <= 0;
    end
    if(data_write_back_complete==1) begin
      dirty_bank[set_index] <= 0;
      cache_state<=`CACHE_IDLE;
    end
    if (data_ready) begin
      // get data from mem
      cache_state<=`CACHE_DATA_RECIEVED;
      data_bank[set_index] <= input_set;
      tag_bank[set_index] <= inst_tag;
      valid_bank[set_index] <= 1;
      dirty_bank[set_index] <= 0;
      dmem_write <= 0;
    end
  end
```

우선, write_back이 있었던 경우를 대비, 쓰기 완료 여부를 입력으로 받아서

write_back -> write_allocate로 넘어가는 로직을 추가해준다.

(IDLE로 만들고 dirty를 0으로 바꾸면 다음 사이클에서 알아서 넘어감)

allocate가 끝나면 (data_ready=1) 뱅크의 데이터들을 업데이트해준다.

3.2.2.3 Write

Read의 경우는 위 단계에서 종료하면 되고, write면 지정된 위치에 쓰고 dirty를 1로 바꾸는 작업까지 해준다.

```
if (mem_rw == 1) begin // write
  // write-allocate
  if(cache_state==`CACHE_DATA_RECIEVED||cache_state==`CACHE_IDLE) begin
    dirty_bank[set_index] <= 1;
    case (block_offset)
      0: begin
        data_bank[set_index][31:0] <= input_line;
      end
      1: begin
        data_bank[set_index][63:32] <= input_line;
      end
      2: begin
        data_bank[set_index][95:64] <= input_line;
      end
      3: begin
        data_bank[set_index][127:96] <= input_line;
      end
    endcase
  end
end
```

3.3 Hit Ratio

```
### SIMULATING ###
TEST END
SIM TIME : 193046
TOTAL CYCLE : 96522 (Answer : 71567)
FINAL REGISTER OUTPUT
Total: 2499
Hit: 1247
Miss:1252 Ratio: 0.499
```

Naive

```
### SIMULATING ###
TEST END
SIM TIME : 175132
TOTAL CYCLE : 87565 (Answer : 76800)
FINAL REGISTER OUTPUT
Total: 2499
Hit: 1434
Miss:1065 Ratio: 0.57383
```

Opt

optimal에서 더 많은 hit이 나왔다.

4 Discussion

- Analyze cache hit ratio
  - If you implement associative cache, compare it with direct-mapped cache
  - Explain your replacement policy
  - Naïve matmul vs optimized matmul
  - Why is the cache hit ratio different between two matmul algorithms?
  - What happens to the cache hit ratio if you change the # of sets and # of ways?

4.1 본 과제에서는 direct mapped로 구현했다

4.2 replacement policy 또한 associative cache에 해당하는 항목이다.

원래는 LRU로 하려고 했다.

4.3 optimal에서 더 많은 hit이 나왔다.

4.4 optimal의 경우 캐시의 사이즈 만큼을 단위로 iterate해, 그냥 순서대로 하는 것보다 캐시-친화적이며 conflict miss를 줄일 수 있다.

4.5 way의 수를 늘리면 Conflict Miss가 줄어 hit ratio가 증가한다.