Lab5

## 1. Introduction

이전 과제에서 구현했던 RISC-V 5-stage pipelined CPU 에서는 memory access delay 가 1 이라고 가정한 이른 바 magic memory 개념을 기반하여 설계했다. 본 Lab 5 에서는 실제 CPU 와 연관성을 높이기 위해 magic memory 를 blocking data cache 로 대체하며 이를 통해 캐시를 어떻게 디자인할 지, 그 방식에 따라 hit ratio 가 어떻게 달라지는 지 그리고 캐시 친화적인 알고리즘을 무엇인지에 대해 학습한다.

## 2. Design

### 2.1 explain the design of the cache

이번 과제에서 direct-mapped cache 구조로 cache 를 구현하였다. 문제의 조건에서 캐시 사이즈는 256 바이트, 한 라인 당 사이즈는 16 바이트라고 주어졌으므로, 캐시의 총 엔트리 수는 16 개이다. 따라서 캐시에 접근하는 address 의 구성은 다음과 같다. 캐시의 라인에는 4 byte word 4 개가 저장되므로, 하위 2 비트는 granularity bits, address [3:2]는 block offset bits, 16 개의 엔트리가 캐시에 존재하므로 index bits 는 4 비트, 그리고 나머지 24 비트는 tag bits 에 해당한다. 이때 배열 같은 연속되는 메모리 블록들에 접근할 때 cache hit ratio 를 높이기 위해 address 의 중간 부분을 index bits 으로 사용하고, 상위 부분을 tag bits 로 사용한다.

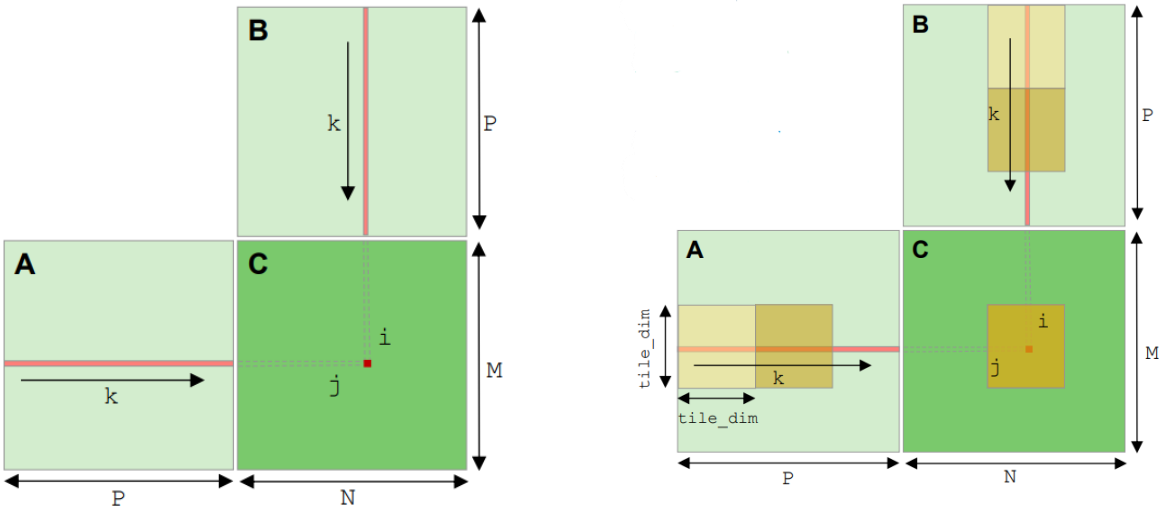
구현한 direct-mapped cache 는 16 개의 엔트리를 지녔으므로, 1 비트 valid bit 레지스터 16 개, 1 비트 dirty bit 레지스터 16 개, 24 비트 tag bank 레지스터 16 개, 128 비트 data bank 레지스터 16 개로 구성되었다. direct-mapped cache 를 사용하였으므로 replacement policy bit 레지스터는 정의하지 않았다.

**2.2 explain your replacement policy.**

direct mapped cache 를 사용하였기에 어떤 블록을 교체할 지 결정할 필요가 없다. 하나의 인덱스에 하나의 캐시 블록이 존재하므로, miss 가 발생한 index 에 해당하는 블록을 교체하면 된다.

반면에 set-associative, fully-associative 방식을 사용했다면 miss 가 발생한 set 에서 어떤 블록을 교체할 지 결정하는 작업이 필요하다. 대표적인 방식 중 하나는 LRU(Least Recently Used) 방식이다. 가장 오랫동안 사용되지 않은 블록을 교체 대상으로 결정한다.

**2.3 explain naive_matmul vs opt_matmul and why is the cache hit ratio different between two matmul algorithms?**




naïve_matmul 은 일반적인 행렬 곱 연산이다. 이러한 연산 방식은 cache-friendly 하지 못하는데 왜냐하면 B 행렬을 접근할 때, 한 column 에서 row 를 건너가며 element 에 접근하기 때문에 spatial locality 가 떨어진다.

opt_matmul 은 tiling matrix multiplication 이다. 흔히 타일링이라 부르는데 이는 행렬 곱셈을 부분행렬의 곱으로 나누는 기법을 말한다. 타일링 기법을 적용하면 타일 내의 요소들을 순차적으로 접근한다. 따라서 타일 내의 데이터가 캐시에 로드 되면 cache hit 할 확률이 높아진다. 결과적으로 블록 내의 데이터들은 메모리에서 fetch 하지 않고 여러 번 재사용할 수 있게 된다. 하지만 타일링 방식은 행렬, 타일, 캐시의 사이즈에 따라서 성능 향상의 편차가 존재한다.

**2.4 what happens to the cache hit ratio if you change the # of sets and # of ways?**

set 과 way 수를 증가시키면 cache hit ratio 가 증가한다. set 의 수를 증가시키면 같은 set 에 여러 개의 address 가 접근하는 cache conflict 의 발생 수를 줄여주기 때문이다. way 의 수를 증가시키면, 한 set 에 존재하는 cache block 이 많아지므로 마찬가지로, 동일한 index 로 접근하여도 여분의 공간이 있으면 miss 가 발생하지 않아 cache hit ratio 가 증가한다. 따라서 cache size 가 동일할 때 associative cache 의 경우, direct-mapped 보다 cache hit ratio 가 증가하지만, cache hit 의 상황에서 동일한 set 내에서 tag 를 병렬적으로 비교해야 하기 때문에 searching time 이 증가하고 hardware complexity 가 증가한다.

### 3. Implementation: Design of the cache (code level)

```
//cache control signals
wire cache_stall;

wire cache_access;
wire cache_ready;
wire chace_out_valid;
wire cache_hit;
```

이번 랩에서 캐시를 구현하기 위해 추가된 시그널 와이어들이다.

```
//----------chace control---------
Cache_controller cache_control_unit(
  .mem_read(EX_MEM_mem_read),
  .mem_write(EX_MEM_mem_write),
  .cache_ready(cache_ready),
  .chace_out_valid(chace_out_valid),
  .cache_hit(cache_hit),
  .cache_access(cache_access),
  .cache_stall(cache_stall)
);

Cache cache(
  .reset(reset),
  .clk(clk),
  .is_input_valid(cache_access),
  .addr(EX_MEM_alu_out),
  .mem_read(EX_MEM_mem_read),
  .mem_write(EX_MEM_mem_write),
  .din(EX_MEM_dmem_data),
  .is_ready(cache_ready), //dmem ready
  .is_output_valid(chace_out_valid), //valid bit of chace block
  .dout(mem_read_data),
  .is_hit(cache_hit) //is tag same cache block
);
```

캐시로 인한 stall 및

캐시 사용 여부 신호 등은

캐시 컨트롤 유닛에 의해

제어되며,

캐시 ready/valid/hit 은

캐시의 상태 output 이다.

```
module Cache_controller (
    input mem_read,
    input mem_write,
    input cache_ready,
    input chace_out_valid,
    input cache_hit,
    output wire cache_access,
    output wire cache_stall);

    assign cache_access = mem_read||mem_write;
    assign cache_stall = (cache_access && !(cache_ready && chace_out_valid && cache_hit)) ? 1 : 0;

endmodule
```

mem read/write 신호가 들어오면 캐시 access 를 활성화시키며,

캐시 access 가 켜져 있는데, 캐시가 준비가 안 되었다면 (ready 가 아니거나 miss 거나…) 캐시가 메모리 로부터 맞는 값을 allocate 할 때까지 기다려야 하므로, 캐시 stall 신호를 내보낸다.

```
// ---------- Update program counter ----------
// PC must be updated on the rising edge (positive edge) of the clock.
PC pc(
  .reset(reset),       // input (Use reset to initialize PC. Initial value must be 0)
  .clk(clk),         // input
  .next_pc(next_pc),     // input
  .PC_stall(PC_stall||cache_stall),
  .current_pc(curr_pc)   // output
);
```

```
 end
 else if (!IF_ID_stall && !cache_stall) begin
    IF_ID_IR <= imem_to_IR;
    IF_ID_pc <= curr_pc;
    IF_ID_predict_branch_taken <= predict_branch_taken;
    IF_ID_branch_predict_target <= btb_next_pc;
 end
```

```
 end
 else if(!cache_stall)begin
   ID_EX_mem_read <= mem_read_con;
   ID_EX_mem_to_reg <= mem_to_reg_con;
   ID_EX_pc_to_reg <= pc_to_reg_con;
   ID_EX_is_jump <= is_jump_con;
```

```
input reset,
input clk,

input is_input_valid,
input [31:0] addr,
input mem_read,
input mem_write,
input [31:0] din,

output is_ready,
output is_output_valid,
output [31:0] dout,
output is_hit);
```

캐시 stall 신호는 위와 같이 PC 와 파이프라인 레지스터의 업데이트를 전부 막음으로써 전체 CPU pipeline 의 진행을 막는다.

캐시 모듈의 I/O

```
// Reg declarations
reg [127:0] data_bank [15:0];
reg [23:0] tag_bank [15:0];
reg valid_bit_table [15:0];
reg dirty_bit_table [15:0];
reg waiting_for_evict; //when evict dirty cache block
reg waiting_for_allocate; //waiting for allocate
reg waiting_for_modify;//when write miss, after bring
```

캐시의 저장소 목록이다. 16 개의 Block 으로 구성되어 있으며,

각 Block 마다 128 비트(= 32*4 = 4 word)의 데이터 저장소와, 24 비트 태그, valid, dirty bit 각각 하나씩 저장한다.

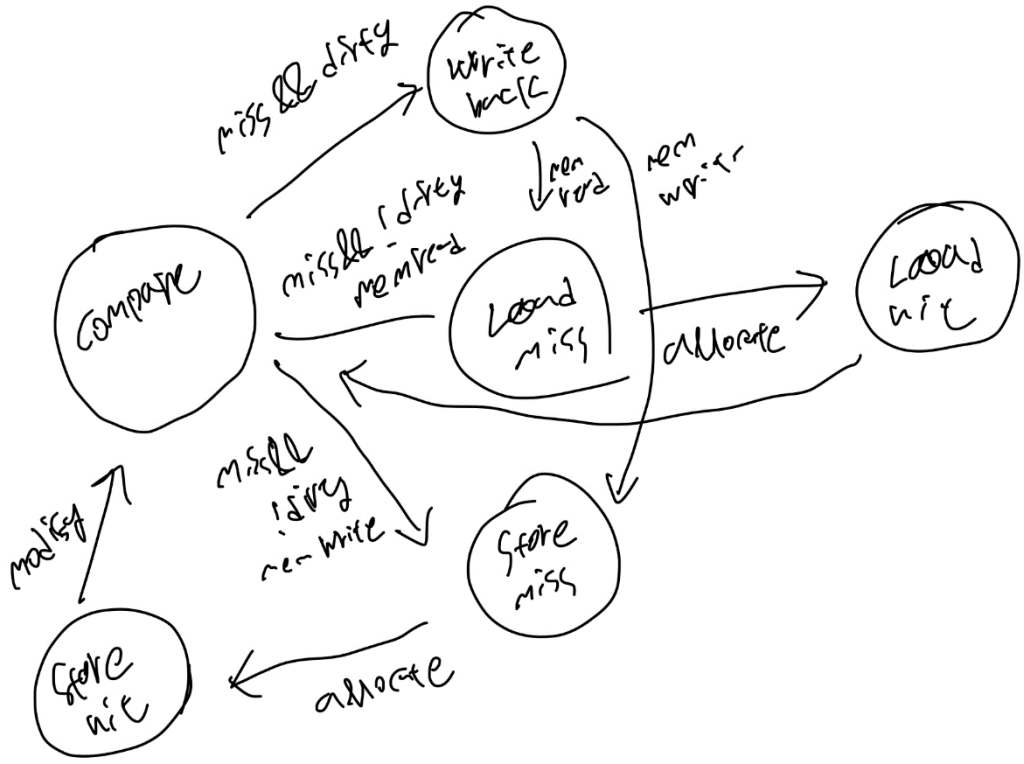
```
//Cache wire declerations
assign is_ready = !waiting_for_evict && !waiting_for_modify;
assign is_output_valid = valid_bit_table[index];
assign is_hit = tag_bank[index] == tag;
assign is_dirty = dirty_bit_table[index];

assign tag = addr[31:8];
assign index = addr[7:4];
assign block_offset = addr[3:2];

assign bit_offset = block_offset*WORD_BIT_SIZE;
assign dout = data_bank[index][bit_offset+:31];
```

캐시의 output signal 들은 위와 같은 logic 으로 결정된다.

tag, index, block offset 으로 나뉜 input addrs 는 hit 비교를 위해 쓰인다.




캐시의 state 를 나타낸 그림.

캐시는 무조건 위와 같은 경로를 따라 작동한다.

1. Compare: input addr 를 tag, index, block offset 으로 분해하여 지금 캐시에 hit 인지 확인한다.
2. 1.에서 hit 라면 Load hit 이면 valid 하고, Store hit 이면, store 할 값으로 해당 캐시 내용이 변조될 때 까지 기다렸다가 valid 해짐.
3. 만약 1. 에서 miss 라면, miss 된 block 의 dirty bit 을 확인한다. 만약 dirty bit 가 0 이면, 그냥 evict 시키고 바로”load/store miss state”로 가서 dmem 에서 필요한 정보를

allocate 하면 되나, 만약 dirty bit 가 1 이면, 캐쉬로 write back 을 끝내기 전에는 allocate 시작조차 못 하므로 한 번 더 기다려야 한다.

4. 3.에서 allocate 를 했다면, “Load/ Store hit” state 가 되게 된다. 이제 필요한 작업을 하면 된다.

위 과정을 보면, 아직 writeback 중 이거나 (waiting_for_evict), allocate 중 이거나 (waiting_for_allocate), store instruction 이 hit 하긴 했지만 아직 그 정보를 쓰지는 못한 상태 (waiting_for_modify)일 때는 캐시 값을 읽으면 안 된다.

따라서 세 신호중 하나라도 켜져 있으면 is_ready 신호가 꺼진다.

캐시가 위와 같은 state 를 따르는 동안,

```
//I/O for data memory
assign dmem_read = (!is_hit || !is_output_valid) && !dmem_out_valid; //Cache r
assign dmem_write = (!is_hit) && is_output_valid && is_dirty; //Conflict occur &&
assign dmem_input = data_bank[index];
//If write to dmem, get addr from cache tag, If read from dmem use addr
assign dmem_addr = dmem_write ? {tag_bank[index],index,4'b0000} : addr ;
```

위와 같은 logic 으로 dmem 에 신호를 보내게 된다.

1. 모종의 이유로 miss 발생 시, dmem 으로부터 해당 정보를 읽어와야 함.

2. miss 가 발생하고, dirty bit 이 켜져 있으면 dmem 으로 write back 해야함

```
// Instantiate data memory
DataMemory #(.BLOCK_SIZE(LINE_SIZE)) data_mem(
  .reset(reset),
  .clk(clk),
  .is_input_valid(is_input_valid),
  .addr(dmem_addr>>>(`CLOG2(LINE_SIZE))), // NOTE: address mus
  .mem_read(dmem_read),
  .mem_write(dmem_write),
  .din(dmem_input),
  .is_output_valid(dmem_out_valid), //Load output valid
  .dout(dmem_out),
  .mem_ready(is_data_mem_ready) //mem ready for knew inst
);
```

그리고 위의 신호들을 이렇게 dmem 에 연결시켜 주면 된다.

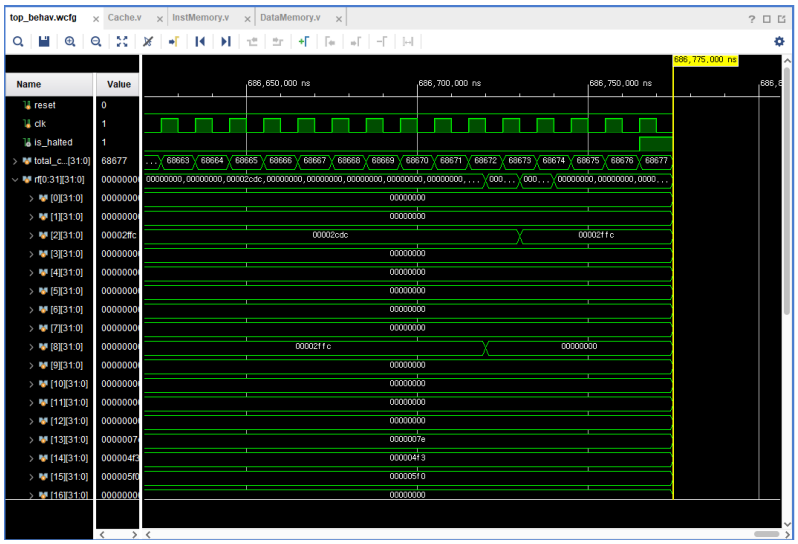
## 4. discussion

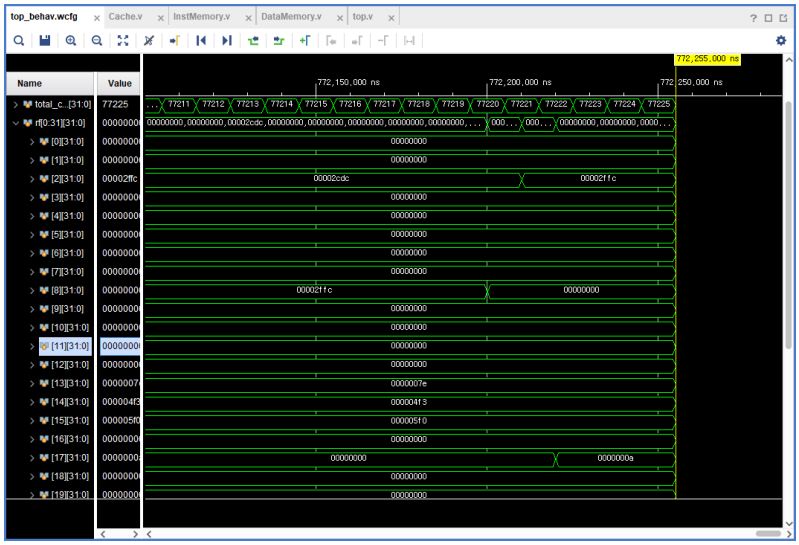
```
assign dmem_addr = dmem_write ? {tag_bank[index],index,4'b0000} : addr ;
```

이 부분을 생각을 못해 오랫동안 디버깅을 하였다. 만약에 dirty 비트가 켜진 block 을 write back 을 해주어야 한다면, 현재 캐시에 접근한 주소로 메모리에 접근하는 것이 아니라, evict 한 블록에 해당하는 tag 와 index 를 조합하여 메모리에 접근해야 한다.

```
assign dmem_read = (!is_hit || !is_output_valid) && !dmem_out_valid;
```

마찬가지로 헷갈렸던 부분인데 cache miss 가 발생했을 때 메모리에 값을 읽어야 하는 건 맞지만 메모리의 delay counter 가 0 이 돼서 dmem_out_valid 신호가 켜졌다면 dmem_read 신호를 꺼주어야 한다.



naïve_mat_mul 의 경우, 68677 사이클이 소요되었고 opt_mat_mul 의 경우 77225 사이클이 소요되었다.

항목 2.3 에서 서술했듯이 타일링 기법이 항상 성능 향상을 이룩하는 것은 아니다. 본 과제에서 direct-mapped cache 로 구현을 했지만 오히려 naïve_mat_mul 을 실행하는데 더 적은 사이클이 소요되었다. 만약 우리가 N-way associative cache 로 구현을 했다면 opt_mat_mul 에서 cache hit ratio 가 더 올라가 사이클 수가 더 적게 걸릴 수 있을 것 같다.

**5. conclusion**

이번 과제를 통해 cache 와 data memory 간에 어떠한 신호를 주고받으며 동작을 하는 지, 또 그에 따라서 cache 는 CPU 에 어떤 신호를 보내주어야 하는 지를 코드를 구현하면서 학습했다. 종강이 다가오면서 여러 일정들이 겹쳐 2-way associative cache 를 구현하지 못한 것이 아쉽다.