CSED311 Computer Architecture

Assignment 5 Report: Cache

TEAM 28

20180154 전제민 jeminjeon@postech.ac.kr

20180916 임경빈 kbini@postech.ac.kr

2023/05/29

1. Introduction

이번 Lab5을 수행하며 Cache가 포함된 CPU를 RTL로 구현하였다. 이전까지는 메모리로부터 데이터를 가져오는데 단 한 사이클만 걸리는 "magic memory"를 사용하였지만, 이번 Lab에서는 data cache를 만들어 메모리 데이터를 fetch하는데 드는 긴 latency를 커버하고 캐시에 다시 사용할 만한 데이터를 적재하여 사용하였다. 이번 보고서에서는 CPU Cache의 논리 회로 설계를 다루고자 한다. 특히 Directed-Mapped Cache와 2-way set associated Cache를 둘 다 구현하여 이 둘을 비교하였다.

- **Design 파트에서는** 각 Cache의 structure와 replacement policy를 설명하였다.
- **Implementation 파트에서는** Valid bit, Dirty bit 등을 포함하여 각 Cache의 구체적인 구현을 설명하였다.
- **Discussion 파트에서는** Direct-mapped cache와 Associative cache의 구현을 비교하고 cache hit ratio를 분석하였다. 또한 Naive_matmul 알고리즘과 Opt_matmul 알고리즘의 Cache hit ratio를 분석하고 차이가 나는 이유를 설명하였다.
- **Conclusion 파트에서는** 이번 구현을 통해 새롭게 알거나 깨닫게 된 점을 간단히 정리하였다.

2. Design

(1) Design of Each Modules

- **CPU** : CPU를 이루는 각 하위 모듈들을 연결시키는 모듈이다. 이전 Lab과 구현이 크

게 달라지지는 않았으나, “Cache_stall”이라는 레지스터를 도입하여 만약 Cache가 데이터를 아직 준비하지 못했을 경우 Cache_stall을 1로 set하여 파이프라인이 진행되는 것을 막는 로직이 추가되었다.

- **DataMemory** : 데이터 메모리 모듈이다. 캐시 구현을 위해 미리 제공된 코드를 사용하였다. Cache 모듈 안에 들어가 만약 캐시 미스가 발생했을 경우 Data Memory로부터 필요한 데이터를 fetch하는 형식으로 구성된다. 이를 구현하기 위해 is_input_valid, is_output_valid, mem_ready라는 input, output 이 추가되었다. 또한 Magic Memory가 아니라 데이터를 fetch하는데 50 cycle이라는 delay가 설정되어 있다. Block_size는 16으로 설정되어 있어 한 블록에 16byte의 데이터가 포함된다.
  - **is_input_valid** : Cache miss가 발생하여 data memory로부터 데이터를 가져와야 할 경우 또는 데이터를 메모리에 write back해야 할 경우 addr와 din 등 필요한 정보를 주고 is_input_valid를 1로 set하여 data memory가 적절한 동작을 수행할 수 있도록 하는 input port이다.
  - **is_output_valid** : mem_read이고 delay counter가 0이 되었을 경우 is_output_valid를 1로 set하여 캐시가 output으로 나온 data를 사용해도 좋다는 표시를 하기 위한 signal output이다.
  - **mem_ready** : data memory 내부에서 설정된 delay counter가 0이 되었을 경우 mem_ready를 1로 set하여 data memory를 사용해도 되는지 여부를 나타낸다.
- **Cache** : Direct-Mapped Cache이다. 만약 is_input_valid signal이 들어온다면, mem_read, mem_write 등의 signal과 addr, din을 이용해 필요한 data 제공하는 역할을 수행한다. **Asynchronous하게 valid, is_hit, data를 read하고 Synchronous하게 cache line에 데이터를 write한다.** 좀 더 구체적인 설계는 아래 (2)에서 설명하였다.
- **Cache2** : 2-way Associative Cache이다. 만약 is_input_valid signal이 들어온다면, mem_read, mem_write 등의 signal과 addr, din을 이용해 필요한 data 제공하는 역할을 수행한다. **Asynchronous하게 valid, is_hit, data를 read하고 Synchronous하게 cache line에 데이터를 write한다.** 좀 더 구체적인 설계는 아래 (3)에서 설명하였다.
- 이외에 PC, ALU, BranchPredictor, ImmediateGenerator, RegisterFile, InstMemory 등의 모듈은 이전 Lab과 달라지지 않았으므로 이와 관련된 Design 설명은 생략한다.

(2) Design of Direct-Mapped Cache

- **Structure :** Direct Mapped Cache의 경우 요청한 주소의 [7:4]에 해당하는 index마다 하나

의 캐시 엔트리가 대응 된다. 32비트의 주소에서 앞에서부터 24비트, 4비트, 2비트, 2비트가 각각 태그, 인덱스, Block offset, Byte offset으로 사용된다.

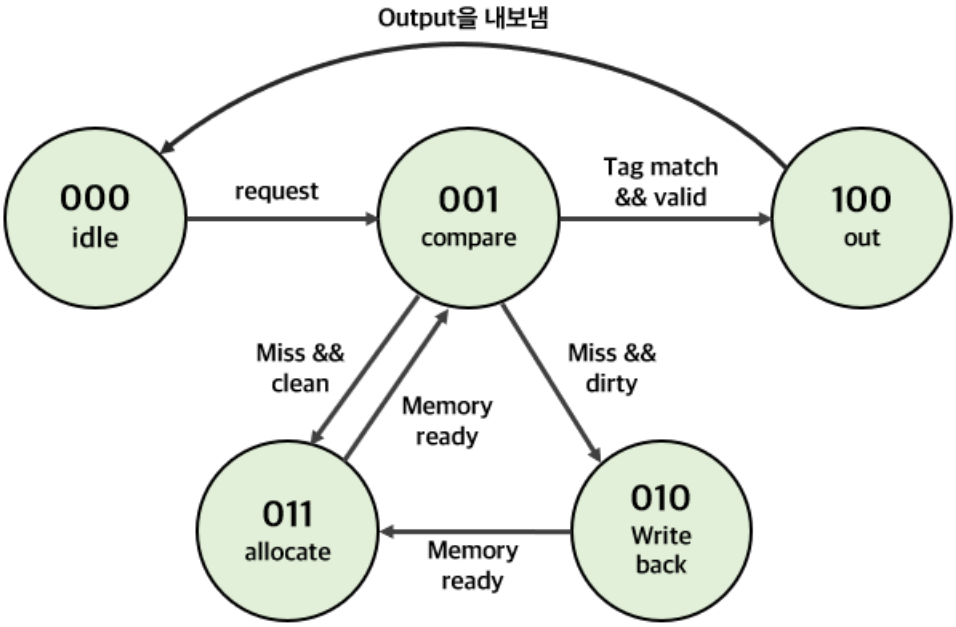
캐시는 총 16개의 라인으로 구성되어 있다. Direct Mapped 방식이므로 하나의 인덱스마다 하나의 캐시라인으로 매핑된다. 하나의 캐시 라인은 155 비트로 구성되어 있으며 앞에서부터 24비트, 1비트, 1비트, 1비트, 128비트가 각각 태그, replacement, valid, dirty, 데이터를 저장한다.

- **Tag :** 인덱스에 대응되는 캐시라인의 태그와 주소의 태그 비트를 비교하여 Hit 또는 Miss 여부를 알아내기 위해 사용된다.
- **Replacement :** Replace가 필요한 way를 표시하기 위해 필요한 bit이다. 사실 associative cache를 구현하기 위해 필요하여 direct mapped cache의 경우 하나의 인덱스마다 하나의 캐시라인만 매핑 되기 때문에 사용될 일이 없다.
- **Valid :** 캐시가 초기화될 때는 Valid를 0으로 초기화한다. Valid가 0일 경우 해당 캐시라인은 사용된 적이 없으므로 Miss를 일으키고 필요한 데이터를 가져와야 한다. 사용된 캐시라인은 Valid bit가 1로 set된다.
- **Dirty :** 만약 Write를 통해 캐시 라인에 write가 이루어졌을 경우, 아직 메모리에는 업데이트가 반영되지 않았으므로 해당 캐시 라인을 evict해야 할 때 메모리에 write back 되어야함을 표시하기 위해 Dirty bit을 set한다.
- **Data :** 하나의 캐시 라인에 16바이트의 데이터가 저장된다. address에서 Block Offset을 통해 4개의 블록 중 필요한 블록을 먼저 찾고, Byte Offset을 통해 블록 내 4개 바이트 중 필요한 바이트를 찾는다.

31:8	7:4	3:2	1:0
Tag (24bits)	Index (4bits)	Block Offset (2b)	Byte Offset (2b)

Index	154:131	130	129	128	127:96	95:64	63:32	31:0
0	Tag	Repl.	Valid	Dirty	Data 3	Data 2	Data 1	Data 0
1	Tag	Repl.	Valid	Dirty	Data 3	Data 2	Data 1	Data 0
…								
14	Tag	Repl.	Valid	Dirty	Data 3	Data 2	Data 1	Data 0
15	Tag	Repl.	Valid	Dirty	Data 3	Data 2	Data 1	Data 0

- **Cache State :** Cache 모듈에 들어온 input에 대한 여러 상황에 따라 output을 내보내거나 data 메모리에 접근하는 동작을 수행하기 위해 Cache의 상태를 나타내는 Cache_state 레지스터를 3bit로 정의하고 아래와 같은 transition이 일어나도록 하여 관리하였다.



- **idle :** 캐시 요청을 대기하고 있는 상태이다. mem_read와 mem_write를 0으로 reset 하여 데이터 메모리가 데이터를 가져오지 않도록 한다.
- **compare :** idle 상태에서 캐시 요청이 들어오면 compare로 넘어오게 된다. 이때 들어온 주소를 이용해 해당하는 캐시 인덱스의 태그와 매치하여 Hit 여부를 판단한다.
- **write_back :** 만약 Compare 단계에서 cache miss가 발생하고 해당하는 라인이 dirty 일 경우 캐시에 저장되어 있는 데이터를 데이터 메모리에 write하게 된다. 가져와야 할 데이터가 데이터 메모리로부터 준비가 되면 allocate로 넘어가게 된다.
- **allocate :** cache miss가 발생했을 경우 요청 받은 주소에 해당하는 데이터를 데이터 메모리로부터 가져온다. 만약 데이터가 준비되었을 경우 Compare 단계로 이동한다.
- **out :** 만약 요청받은 주소의 태그와 해당 인덱스에 저장된 태그가 일치하고 Valid 도 1로 set 되어 있을 경우 Cache Hit가 발생하여 해당 데이터를 output으로 내보낸다.

(3) Design of 2-way Set Associative Cache

- **Structure :** 2-way Associative Cache의 경우 요청한 주소의 [6:4]에 해당하는 index마다 2개의 캐시라인이 대응된다. 32비트의 주소에서 앞에서부터 25비트, 3비트, 2비트, 2비트

가 각각 태그, 인덱스, Block offset, Byte offset으로 사용된다.

캐시는 총 16개의 라인으로 구성되어 있다. 2-way Associative 방식이므로 하나의 인덱스마다 2개의 캐시라인으로 매핑된다. 하나의 캐시 라인은 156 비트로 구성되어 있으며 앞에서부터 1비트, 1비트, 1비트, 25비트, 128비트가 각각 Valid, Dirty, Replacement, Tag, 데이터를 저장한다.

- Valid, Dirty, Replacement 비트 등에 대한 설명은 위 directed mapped cache와 동일하므로 생략한다.

31:7	6:4	3:2	1:0
Tag (25bits)	Index (3bits)	Block Offset (2b)	Byte Offset (2b)

	Way 1					Way 0				
Index	311	310	309	308:284	283:156	155	154	153	152:128	127:0
0	V	D	R	Tag	Data (3+2+1+0)	V	D	R	Tag	Data (3+2+1+0)
1	V	D	R	Tag	Data (3+2+1+0)	V	D	R	Tag	Data (3+2+1+0)
…										
6	V	D	R	Tag	Data (3+2+1+0)	V	D	R	Tag	Data (3+2+1+0)
7	V	D	R	Tag	Data (3+2+1+0)	V	D	R	Tag	Data (3+2+1+0)

- **Cache State :** Cache State의 동작 과정은 위의 Direct Mapped Cache와 동일하므로 생략한다.

(4) Replacement Policy

- 2-way Associative의 경우 하나의 index에 두개의 캐시라인이 대응되므로 둘 중 어떤 라인을 사용할 것인지 결정할 Replacement policy가 필요하다.
- LRU(Least Recently Used) 방식을 베이스로 사용하여, 두 캐시라인 중 최근에 사용하지 않은 캐시 라인을 Evict하는 방식으로 Replacement Policy를 구현하였다. Cache Hit, Cache Miss를 판단하는 Compare 단계에서 두 라인 중 사용할 라인을 Cache_select라는 레지스터에 저장하는데, 사용하지 않은 Cache line의 Replacement 비트를 1로 두어 나중에 Eviction이 필요할 경우 이 캐시라인을 제거하는 방식이다. 구체적인 로직은 아래와 같이 구성된다.

(1) Cache Hit, Cache Miss를 판단하는 Compare 단계에서, 만약 Cache Hit가 발생했을 경우 Cache_Select를 Hit가 발생한 (태그가 일치하고 Valid가 1인) 라인으로 설정한다.

(2) 만약 Replacement bit가 1인 캐시라인이 있을 경우, Cache_Select를 해당 캐시라인으로 설정한다.

(3) 만약 두 캐시라인 모두 Replacement bit가 0인 경우, Cache_select를 0으로 두어 첫번째 라인에 대해 작업을 수행한다 (두 라인 모두 Replacement bit가 1인 경우는 존재하지 않는다.)

(4) 이후 Cache 라인에 데이터를 할당하는 Allocate 단계에서, Cache_select에 해당하지 않는 라인의 Replacement bit를 1로 설정한다. 데이터를 할당한 캐시라인은 Replacement bit를 0으로 설정한다.

(5) 위와 같은 로직으로 캐시 라인의 데이터를 교체할 경우, 직전에 사용된 (데이터가 할당된) 캐시라인은 Replacement bit가 0, 그렇지 않은 캐시라인은 Replacement bit가 1로 설정되기 때문에 **결과적으로 LRU policy에 따라 캐시를 교체하게 된다.** (만약 두 캐시라인 모두 Replacement bit가 0인 경우는 두 라인 모두 사용된 적이 없는 경우로, 이 때는 0번째 캐시라인을 처음으로 사용하게 된다.)

3. Implementation

(1) Exceptions of Identical Modules

이전 Lab 4-2와 구현이 동일한 모듈인 PC, ALU, ALUControlUnit, ForwardingUnit, , HazardControlUnit, ImmediateGenerator, RegisterFile에 대한 내용은 제외하였다.

(2) Data Memory implementation

Data Memory는 skeleton code에 주어진 것을 그대로 사용하였다. Cache miss가 나서 data memory에서 데이터를 read 해야하는 경우 또는 cache conflict 가 발생하여 기존 dirty data를 write back 해야 하는 경우에만 is_input_valid를 1로 줘서 Data Memory에서 Delay 만큼 시간이 지난 뒤 data를 write 하거나 read 할 수 있도록 연결을 구현하였다.

```
module DataMemory #(parameter MEM_DEPTH = 16384,
                    parameter DELAY = 50,
                    parameter BLOCK_SIZE = 16) (
    input reset,
    input clk,

    // Inputs from the cache
    input is_input_valid,                  // is request valid?
    input [31:0] addr,                     // address of the memory
    input mem_read,                        // is read signal driven?
    input mem_write,                       // is write signal driven?
    input [BLOCK_SIZE * 8 - 1:0] din,      // data to be written

    // outputs from the data memory
    output is_output_valid,                // is output valid?
    output [BLOCK_SIZE * 8 - 1:0] dout,    // output data
    output mem_ready);
```

(3) Two-way Set Associative Cache implementation

Two-way Set Associative cache의 구현은 다음과 같이 구현하였다. Two way cache를 indexing 하는데

는 cache module에 input으로 주어진 addr의 [6:4] 3bit를 사용하였고 tag part는 addr의 [31:7] 25bit 를 사용하도록 구현하였다. Index와 tag를 저장하기 위한 레지스터, two-way cache를 다음과 같은 크기로 구현하였다.

```
33    reg [311:0] TWOWAYCACHE[0:7];
34    reg [2:0]index;
35    reg [24:0]tag;
```

Data Memory와의 연결은 아래와 같이 구현하였다.

```
DataMemory #(.BLOCK_SIZE(LINE_SIZE)) data_mem(
  .reset(reset),
  .clk(clk),

  .is_input_valid(is_dmem_input_valid),
  .addr({wire_addr>>4}),        // NOTE: address must be shifted by CLOG2(LINE_SIZE)
  .mem_read(wire_mem_read),
  .mem_write(wire_mem_write),
  .din(wire_din),

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

Two-way Set Associative cache와 state 관련 레지스터들의 초기화는 아래와 같이 구현하였다.

```
always @(posedge clk)begin
    //######## initialize ########################
    if(reset) begin
     for(i = 0; i< 8 ; i = i + 1)
        TWOWAYCACHE[i] = 0;
        cache_state <= 0;

        index[3:0]<=0;
        tag[23:0]<=0;
        wire_mem_read<=0;
        wire_mem_write<=0;
        wire_din[127:0]<=0; // 128bit
        wire_addr[31:0]<=0; // 32bit
        cache_tag[23:0]<=0; // 24bit
        cache_data[127:0]<=0; //128bit
        wire_is_data_mem_ready<=0;
        total_count<=0;
        hit_count<=0;
        is_hit<=0;
        is_dmem_input_valid<=0;
        output_ready<=0;
        write_check<=0;
        read_check <=0;
        hit_check <=0;
        cache_way <=0;
        cache_select<=0;
    end
```

캐쉬에 유효한 입력이 들어오고 mem_read가 1이거나 mem_write가 1인 경우 cache_state에 따라 다음과 같은 동작을 수행하도록 구현하였다.

1) 000 state

아래는 cache_state가 000일때의 구현이다. 000state는 유효한 입력이 들어올 때까지 대기하다가

유효한 입력이 들어오면 cache_state를 001로 바꾸어 주는 state이다.

```
else if(is_input_valid&&(mem_read||mem_write)) begin
        case(cache_state)
                3'b000: begin

                    if(is_input_valid)begin
                    cache_state <= 3'b001;
                    cache_select <=0;
                    end
```

2) 001 state

아래는 001 state의 구현이다. 001state에서는 valid, tag, dirty, replacement bit를 검사하고 그에 따라 다른 state로 변하도록 구현하였다 CACHE 모듈에 입력으로 들어온 addr[6:4]를 인덱스로 사용하여 TWOWAYCACHE의 해당 인덱스의 2개의 way 중 해당 valid가 1이고 tag도 일치하는게 있다면 100 state로 보내고 없다면 replacement bit 가 1인 way로 dirty bit가 1이면 010 state에서 write back하도록 state를 변화시키고 dirty bit가 0이면 바로 011state로 변화시켜 DataMemory로 부터 데이터를 받아오도록 구현하였다. Replacement 여부는 wire를 통해 구현하였는데 is_replace_1은 TWOWAYCACHE[index][155:0]을 replace해야 할 때 1이고 is_replace_2은 TWOWAYCACHE[index][311:156]을 replace해야 할 때 1이다. 100 state에서 사용하는 부분을 replacement bit를 0으로 주고 사용하지 않는 way 부분을 replacement bit를 1로 주도록 LRU replacement policy로 구현하였다.

```
                3'b001: begin
                    if(is_valid_1 && is_tag_1)begin
                        cache_state <=3'b100;
                        is_hit <=1;
                        is_output_valid<=1;
                        output_ready<=1;
                        cache_select<=0;
                    end
                    else if(is_valid_2 && is_tag_2)begin
                        cache_state <=3'b100;
                        is_hit <=1;
                        is_output_valid<=1;
                        output_ready<=1;
                        /// 2
                        cache_select<=1;
                    end
                    else if(is_replace_1 && is_dirty_1)begin
                        cache_state <= 3'b010;
                        is_hit <=0;
                        index <=addr[6:4];
                        cache_tag <=TWOWAYCACHE[addr[6:4]][152:128];
                        cache_data<=TWOWAYCACHE[addr[6:4]][127:0];
                        wire_din <= TWOWAYCACHE[addr[6:4]][127:0];
                        wire_addr <= {TWOWAYCACHE[addr[6:4]][152:128],addr[6:4], 4'b0000}; // 4bit shift
                        wire_mem_read <= 0;
                        wire_mem_write <= 1;
                        is_dmem_input_valid <=1;
                        write_check <=1;
                    end
                    else if(is_replace_2 && is_dirty_2)begin
                        cache_state <= 3'b010;
                        is_hit <=0;
                        index <=addr[6:4];
                        cache_tag <=TWOWAYCACHE[addr[6:4]][308:284];
                        cache_data<=TWOWAYCACHE[addr[6:4]][283:156];
                        wire_din <=  TWOWAYCACHE[addr[6:4]][283:156];
                        wire_addr <= { TWOWAYCACHE[addr[6:4]][308:284],addr[6:4], 4'b0000}; // 4bit shift
                        wire_mem_read <= 0;
                        wire_mem_write <= 1;
                        is_dmem_input_valid <=1;
                        write_check <=1;
                    end
```

```
279                     else if(is_replace_1&&!is_dirty_1)begin
280                         cache_state <=3'b011;
281                         cache_select <=0;
282                         wire_addr=addr;
283                         wire_mem_read <= 1;
284                         wire_mem_write <= 0;
285                         is_dmem_input_valid <=1;
286                         is_hit <=0;
287                         read_check<=1;
288                     end
289                     else if(is_replace_2&&!is_dirty_2)begin
290                         cache_select <=1;
291                         cache_state <=3'b011;
292                         wire_addr=addr;
293                         wire_mem_read <= 1;
294                         wire_mem_write <= 0;
295                         is_dmem_input_valid <=1;
296                         is_hit <=0;
297                         read_check<=1;
298                     end
299                     else if(!is_replace_1 && !is_replace_2) begin
300                         cache_select <=0;
301                         cache_state <=3'b011;
302                         wire_addr=addr;
303                         wire_mem_read <= 1;
304                         wire_mem_write <= 0;
305                         is_dmem_input_valid <=1;
306                         is_hit <=0;
307                         read_check<=1;
308                     end
```

3) 010 state

아래는 010state의 구현이다. 010 state는 dirty bit가 1일 때 write back 하기 위한 스테이트이다. DataMemory에 유효한 입력, din, 캐시에 있는 해당 교체되어야하는 메모리주소를 줘서 write_back 하도록 구현하였다. Write_back이 끝나면 011state로 변화한다.

```
313                 //################## Write - Back ################
314                 3'b010: begin //#######################################....
315                     if(is_data_mem_ready&&!write_check)begin
316                         is_dmem_input_valid<=1;
317                         read_check<=1;
318                         cache_state <= 3'b011;
319                     end
320                     else begin
321                     cache_state <= 3'b010;
322                     is_dmem_input_valid <=0;
323                     write_check<=0;
324                     end
325                 end
```

4) 011 state

다음은 011 state의 구현이다. 011 state는 DataMemory에서 값을 읽어와 cache에 allocate 하기 위한 스테이트이다. DataMemory에 유효한 입력, addr을 줘서 해당 메모리 주소의 값을 가져와 cache에 할당하도록 구현하였다. Allocate가 끝나면 다시 001state로 변화한다.

```
326                 // ################ Allocate ###################### ..\
327                 3'b011: begin
328                     if(is_data_mem_ready&&wire_is_output_valid&&!read_check&&cache_select==0)begin
329                         cache_state <= 3'b001;
330                         TWOWAYCACHE[addr[6:4]][152:128] <= addr[31:7]; //tag
331                         TWOWAYCACHE[addr[6:4]][127:0]<=dmem_dout[127:0]; //data
332                         TWOWAYCACHE[addr[6:4]][154]<=0; //dirty
333                         TWOWAYCACHE[addr[6:4]][155]<=1; //valid
334                         TWOWAYCACHE[addr[6:4]][153]<=0; // 2WAY 중에 업데이트 되는애만 1.
335                         TWOWAYCACHE[addr[6:4]][309]<=1; // 2WAY 중에 업데이트 되는애만 1.
336                         is_dmem_input_valid <=0;
337                     end
338
339                     else if(is_data_mem_ready&&wire_is_output_valid&&!read_check&&cache_select==1)begin
340                         cache_state <= 3'b001;
341                         TWOWAYCACHE[addr[6:4]][308:284] <= addr[31:7]; //tag
342                         TWOWAYCACHE[addr[6:4]][283:156]<=dmem_dout[127:0]; //data
343                         TWOWAYCACHE[addr[6:4]][310]<=0; //dirty
344                         TWOWAYCACHE[addr[6:4]][311]<=1; //valid
345                         TWOWAYCACHE[addr[6:4]][153]<=1; // 2WAY 중에 업데이트 되는애만 1.
346                         TWOWAYCACHE[addr[6:4]][309]<=0; // 2WAY 중에 업데이트 되는애만 1.
347
348                         is_dmem_input_valid <=0;
349                     end
350
351                     else begin
352                         cache_state <= 3'b011;
353                         is_dmem_input_valid <=0;
354                         read_check<=0;
355                     end
356
357
```

5) 100 state

다음은 100 state의 구현이다. 100 state에서는 mem_read가 1인 경우에는 clk asynchrounously read하고 mem_write가 1인 경우에는 캐시에 clk synchronously write 하도록 구현하였다. Cache_select reg를 통해 어떤 way의 값을 사용해야 하는지 판별한다.

```
3'b100: begin
    output_ready<=0;
    cache_way<=0;
    if(mem_write) begin
    is_dmem_input_valid <=0;
        cache_state<=0;
        is_output_valid <=0;
        finish <=1;
        is_hit <=0;

        //write at cache block
        if(cache_select ==0) begin
        TWOWAYCACHE[addr[7:4]][154]<=1;//dirty
        case(addr[3:2])
            2'b00: begin
                TWOWAYCACHE[addr[6:4]][31:0]<=din[31:0];
//                  CACHE[addr[7:4]][127:0] <=
            end
            2'b01: begin
                TWOWAYCACHE[addr[6:4]][63:32]<=din[31:0];
            end
            2'b10: begin
                TWOWAYCACHE[addr[6:4]][95:64]<=din[31:0];
            end
            2'b11: begin
                TWOWAYCACHE[addr[6:4]][127:96]<=din[31:0];
            end
        endcase
        end
```

```
        if(cache_select ==1) begin
        TWOWAYCACHE[addr[7:4]][310]<=1;//dirty
        case(addr[3:2])
            2'b00: begin
                TWOWAYCACHE[addr[6:4]][187:156]<=din[31:0];
//                  CACHE[addr[7:4]][127:0] <=
            end
            2'b01: begin
                TWOWAYCACHE[addr[6:4]][219:188]<=din[31:0];
            end
            2'b10: begin
                TWOWAYCACHE[addr[6:4]][251:220]<=din[31:0];
            end
            2'b11: begin
                TWOWAYCACHE[addr[6:4]][283:252]<=din[31:0];
            end
        endcase
        end
```

<synchronously write to Cache>

```
        else if(cache_state ==3'b100) begin
            wire_din =4294967295; //NO DIN
            wire_addr = 4294967295; //NO ADDR
            wire_mem_read =0;
            wire_mem_write = 0;
            //is_dmem_input_valid =0;
            if(cache_select==0)begin
            case(addr[3:2])
            2'b00: begin
                dout = TWOWAYCACHE[addr[6:4]][31:0];
            end
            2'b01: begin
                dout = TWOWAYCACHE[addr[6:4]][63:32];
            end
            2'b10: begin
                dout = TWOWAYCACHE[addr[6:4]][95:64];
            end
            2'b11: begin
                dout = TWOWAYCACHE[addr[6:4]][127:96];
            end
            endcase

            end


            else if (cache_select==1)begin
            case(addr[3:2])
            2'b00: begin
//                  dout = CACHE[addr[6:4]][31:0];
                dout = TWOWAYCACHE[addr[6:4]][187:156];
            end
            2'b01: begin
//                  dout = CACHE[addr[6:4]][63:32];
                dout = TWOWAYCACHE[addr[6:4]][219:188];
            end
            2'b10: begin
//                  dout = CACHE[addr[6:4]][95:64];
                dout = TWOWAYCACHE[addr[6:4]][251:220];
            end
            2'b11: begin
//                  dout = CACHE[addr[6:4]][127:96];
                dout = TWOWAYCACHE[addr[6:4]][283:252];
            end
            endcase
            end
```

<asynchronously read from Cache>

(3) Direct-Mapped Cache implementation

Direct-mapped cache의 구현은 2-way Set Associative cache와 유사하게 구현하였다. 다른 점은 tag로 상위24비트를 사용하고 인덱스로 addr[7:4]를 사용하도록 구현하였다. Cache에 way가 하나 뿐이므로 메모리 주소의 4bit로 인덱싱 한 후 어떤 way를 사용할 지 결정하는 부분이 없다는 차이가 있다.

(4) Cache Stall implementation

Cache에서 데이터가 준비되지 않은 경우에는 stall을 하여 pipeline register들의 write가 일어나지 않도록 구현하였다. Cache_stall signal은 MEM stage에서 load 또는 store instruction이 수행 될 때 ready, is_hit, valid 세 가지가 모두 1인 경우를 제외한 나머지 상황에서 cache_stall이 1이 되도록 구현하였다. 즉 MEM_stage에서 load 또는 store 명령어를 실행할 시에는 ready, is_hit, valid가 1인 경우에만 stall 되지 않는다. 아래는 cache_stall signal을 통해 pipeline register의 update를 막는 예시이다.

```
521     if (reset) begin
522         MEM_WB_mem_to_reg <=0;
523         MEM_WB_reg_write <=0;
524 
525         MEM_WB_mem_to_reg_src_1 <=0;
526         MEM_WB_mem_to_reg_src_2 <=0;
527         MEM_WB_rd <=0;
528         MEM_WB_HALT <=0;
529 
530         MEM_WB_pc <= 0;
531         MEM_WB_pc_to_reg <= 0;
532 
533     end
534     else if(cache_stall ==0) begin
535         MEM_WB_mem_to_reg <=wire_MEM_WB_mem_to_reg;
536         MEM_WB_reg_write <=wire_MEM_WB_reg_write;
537 
538         MEM_WB_mem_to_reg_src_1 <=wire_MEM_WB_mem_to_reg_src_1;
539         MEM_WB_mem_to_reg_src_2 <=wire_MEM_WB_mem_to_reg_src_2;
540         MEM_WB_rd <= wire_MEM_WB_rd;
541         MEM_WB_HALT <= wire_MEM_WB_HALT;
542 
543         MEM_WB_pc <= wire_MEM_WB_pc;
544         MEM_WB_pc_to_reg <= wire_MEM_WB_pc_to_reg;
545     end
```

4. Discussion

(1) Hit ratio of Direct-Mapped Cache and 2-way Set Associative Cache

우리가 구현한 Direct-Mapped Cache, 2-way Set Associative Cache를 이용하여 test bench를 수행하였을 때 hit rate, total cycle은 다음과 같다.

2-way Set Associative Cache를 이용하여 테스트 벤치를 실행한 결과이다.

Naïve_matmul_unroll.mem	Opt_matmul_unroll.mem
TOTAL CYCLE 60661 0 00000000 1 00000000 2 00002ffc 3 00000000	TOTAL CYCLE 62284 0 00000000 1 00000000 2 00002ffc 3 00000000

4 00000000 5 00000000 6 00000000 7 00000000 8 00000000 9 00000000 10 0000000d 11 00000000 12 00000000 13 0000007e 14 000004f3 15 000005f0 16 00000000 17 0000000a 18 00000000 19 00000000 20 00000000 21 00000000 22 00000000 23 00000000 24 00000000 25 00000000 26 00000000 27 00000000 28 00000000 29 00000000 30 00000000 31 00000000 Total : 2499 Hit : 1839	4 00000000 5 00000000 6 00000000 7 00000000 8 00000000 9 00000000 10 0000000d 11 00000000 12 00000000 13 0000007e 14 000004f3 15 000005f0 16 00000000 17 0000000a 18 00000000 19 00000000 20 00000000 21 00000000 22 00000000 23 00000000 24 00000000 25 00000000 26 00000000 27 00000000 28 00000000 29 00000000 30 00000000 31 00000000 Total : 2499 Hit : 1828

Direct-Mapped Cache를 이용하여 테스트 벤치를 실행한 결과이다.

Naïve_matmul_unroll.mem	Opt_matmul_unroll.mem
TOTAL CYCLE 75321 0 00000000 1 00000000 2 00002ffc 3 00000000 4 00000000 5 00000000 6 00000000 7 00000000 8 00000000 9 00000000 10 0000000d 11 00000000 12 00000000 13 0000007e 14 000004f3 15 000005f0 16 00000000 17 0000000a 18 00000000 19 00000000 20 00000000 21 00000000 22 00000000	TOTAL CYCLE 80655 0 00000000 1 00000000 2 00002ffc 3 00000000 4 00000000 5 00000000 6 00000000 7 00000000 8 00000000 9 00000000 10 0000000d 11 00000000 12 00000000 13 0000007e 14 000004f3 15 000005f0 16 00000000 17 0000000a 18 00000000 19 00000000 20 00000000 21 00000000 22 00000000

23 00000000 24 00000000 25 00000000 26 00000000 27 00000000 28 00000000 29 00000000 30 00000000 31 00000000 Total : 2499 Hit : 1687	23 00000000 24 00000000 25 00000000 26 00000000 27 00000000 28 00000000 29 00000000 30 00000000 31 00000000 Total : 2499 Hit : 1605

cycle 수를 비교하였을 때 2-way Set Associative Cache는 naïve implementation에 대하여 60661 cycle, opt implementation에 대하여 62284 cycle이 소요되었고 Direct-Mapped Cache는 naïve implemantion에 대하여 75321 cycle, opt implementation에 대하여 80655 cycle이 소요되었다. 두 matmul 방식 모두 2-way Set Associative Cache가 Direct-Mapped Cache 보다 더 적은 사이클이 소요되었다.

Hit ratio를 비교하면 2-way Set Associative Cache 는 naïve implementation이 73.59%, opt implementation이 73.15 %로 결과가 도출 되었고 Direct-Mapped Cache는 naïve implementation이 67.51%, opt implementation이 64.23%로 결과가 도출 되었다. 2-way Set Associative와 Direct Mapped Cache를 비교하면 2-way Set Associative Cache가 Direct Mapped Cache 보다 더 높은 hit ratio를 보였고 두 방식의 캐시 모두 naïve implementation이 opt implementation 보다 더 높은 hit ratio를 보였다.

(2) Comparision of Direct-Mapped Cache and 2-way Set Associative Cache

Direct-Mapped Cache, 2-way Set Associative Cache를 사용하여 test bench 를 수행한 결과 naïve matmul, opt matmul 두 방식의 테스트 벤치 모두 2-way Set Associative Cache가 Direct-Mapped Cache보다 더 높은 hit ratio를 보였고 더 적은 사이클수가 소요되었다. Direct-mapped cache에서는 캐시 인덱싱에 4bit를 사용하였고 2-way Set Associative Cache에서는 3bit를 사용하였는데 Direct mapped cache에서 이 부분이 일치하는 부분이 많아 cache conflict 상황이 더 많이 발생하여 이러한 결과를 나타내었다고 해석하였다. 만약 way의 수를 더 늘린다면 cache conflict 상황이 줄어들어 더 높은 hit-ratio를 나타낼 것으로 예상된다. 반대로 way의 수를 줄이고 set의 수를 늘릴 경우

(3) Comparision of naïve implementation to opt implementation

Direct-Mapped Cache, 2 way Set Associative Cache 모두 Naïve matmul 방식이 opt matmul 방식보다 더 높은 hit rate를 보였고 사이클 수는 더 적게 나타났다. Opt matmul 방식이 naïve matmul 방식보다 더 높은 hit ratio를 나타낼 것으로 예상하였으나 Direct-mapped Cache, 2-way Set

Associative Cache 모두 예상과 다른 결과를 보였다. 이는 opt matmul 방식에서는 tiled implementation 방식을 사용하기 때문에 더 효율적이여야 한다. Matmul이 matrix C =matrix A * matrix B 라고 둘 때 A, B, C 모두 n by n matrix라면 A에서 block을 읽고 B에서 block을 읽고 C에서 block을 읽고 쓰는 과정은 총 $n^3 + 2n^2$ 만큼 일어난다. 하지만 opt(tiled) matmul 방식에서는 n by n matrix를 k by k 의 Tile들(총 타일의 개수 = k^2) 로 나눈다면 $2kn^2 + 2n^2$ 만큼 타일을 읽고 쓴다. 따라서 만약 $n^3 > 2kn^2$ 즉, $n > 2k$ 여서 tile의 크기를 너무 작게 설정하게 되면 메모리에서 읽고 쓰는 과정이 오히려 더 많이 걸리게 된다. 우리의 test bench에서는 이처럼 tile size를 너무 작게 설정하여 tile의 개수가 많아져서 위 결과처럼 낮은 hit ratio를 나타내었다고 해석하였다.

5. Conclusion

이번 Lab에서는 Verilog를 이용해 Cache를 이용하는 CPU를 구현하였다. Directed Mapped Cache와 Set Associative Cache를 모두 설계하고 직접 만들어 성능을 비교하였다. 이전 Lab에 비해 디버깅을 하기 쉽지 않았지만 무사히 성공적으로 마무리할 수 있었다. 메모리로 인해 발생하는 Latency를 재현했을 뿐인데 이렇게 구현이 복잡해진다면 실제 CPU를 만드는 과정은 얼마나 디테일하고 고차원적일지 새삼 실감이 나게 되는 계기였다.

지금까지 한 학기 동안 6번에 걸친 Lab을 통해 Single-cycle CPU에서 시작하여 Cache가 포함된 5-stage Multi-cycle pipeline CPU를 완성하였다. 어렵고 힘든 과정이었지만 컴퓨터 아키텍쳐와 프로세서를 심도 있게 이해할 수 있었던 시간이었다.