컴퓨터 구조 (CSED311)

<Lab5 - Cache>

Team ID : 33
Student 1 : 곽민성 (Gwak Minseong, 20230840)
Student 2 : 김재환 (Kim Jaehwan, 20230499)

1. Introduction

이번 랩의 목표는 Lab 4에서 구현한 Pipelined CPU에, Magic Memory로 구성되어 있던 Data Memory Module을 Blocking Data Cache를 직접 구현해서 대체하는 것이다. 이 랩을 성공적으로 진행함으로써 Direct-mapped cache와 Set-associative cache의 세부 구조와 동작을 정확하게 이해할 수 있었다. 또, Cache Eviction 규칙인 LRU에 대해 더 잘 이해할 수 있었다.

2. Design

우리는 아래 Cache의 구조를 사용하여 구현하였다.

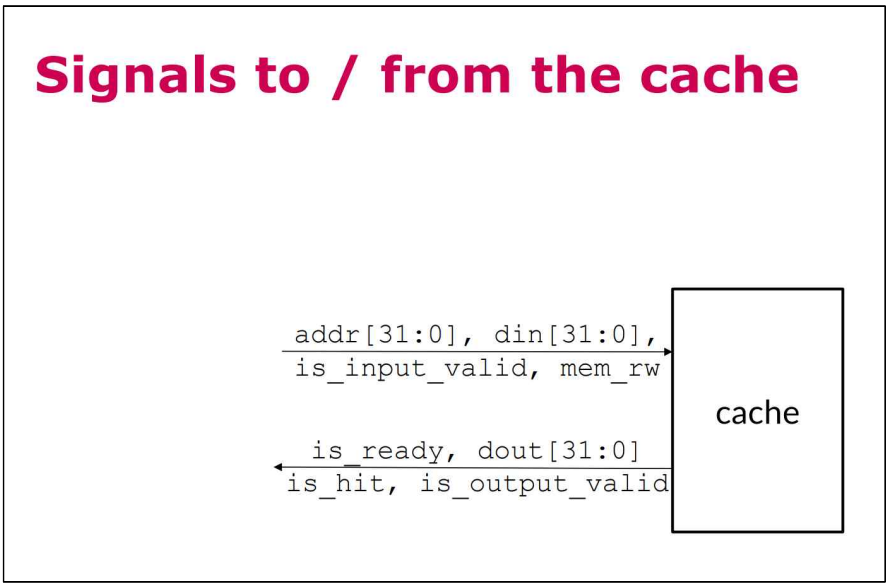

그림 1 Cache의 기본 구조

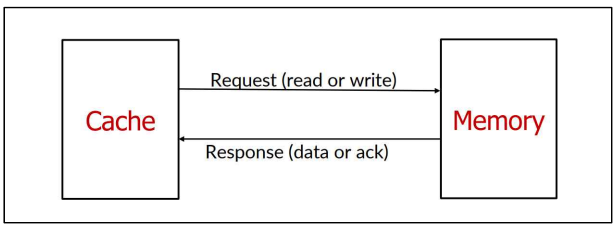

그림 2 Cache와 Memory의 관계. Cache Miss가 나면, Memory (또는 하위 Cache)에 Miss가 난 주소의 값을 요청하여야 한다.

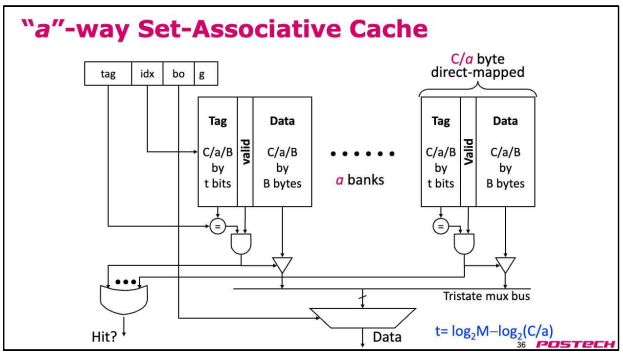

그림 3 "a"-way Set-Assosiative Cache의 기본 구조.

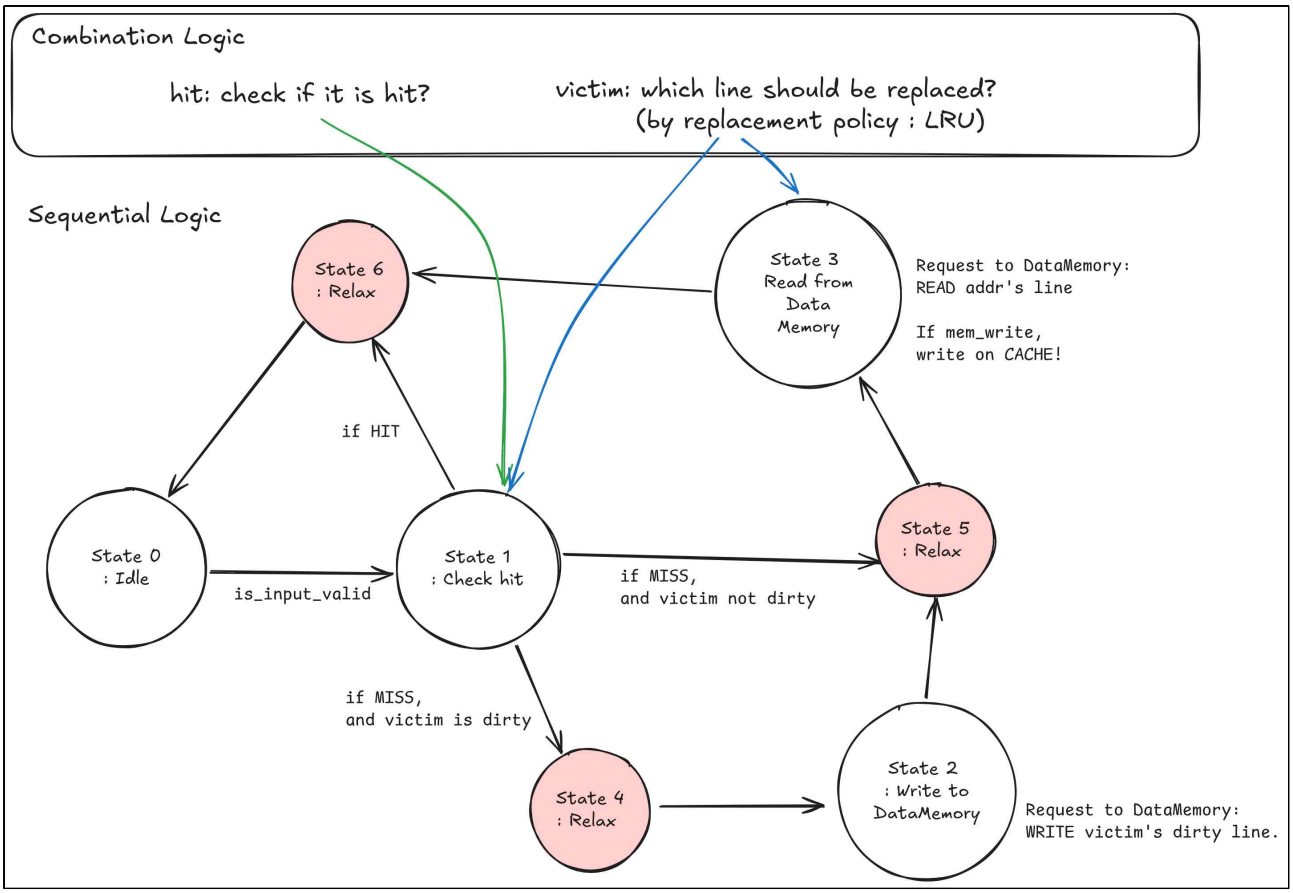

그림 4 Our Cache's State Transition Diagram. 화살표는 clk의 posedge에 따라 이동한다. ready된 상태(state 0, idle)에서 valid한 input을 받게 되면, state 1로 전이해서 cache hit를 확인한다. 만약 hit 한다면, state 6으로 전이해서 그 결과를 출력하고, 다시 state 0으로 돌아온다. 만약 cache miss가 발생했을 때는, evict할 cache line을 LRU 규칙에 따라 찾고, 만약 그 line의 dirty bit가 켜져 있다면 state 4 -> state 2로 이동하면서 write-back한다. 그러고 나서 state 5 -> state 3으로 이동하면서 memory에서 읽거나 쓸 부분을 read한 후 cache에 추가한다. 그 결과를 state 6으로 전이하면서 그 결과를 출력하고, 다시 ready 상태 (state 0)으로 전이한다.

또, 전체 Pipeline은 Cache에서 Data Memory를 받아올 동안 완전히 정지하게 된다.

3. Implementation

```
module Cache #(parameter LINE_SIZE = 16,
               parameter NUM_SETS = 4, /* Your choice */
               parameter NUM_WAYS = 4 /* Your choice */) (
    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
);

  localparam                  SET_BIT          = `CLOG2(NUM_SETS);
  localparam                  BLK_OFFSET_BIT   = `CLOG2(LINE_SIZE);
  localparam                  TAG_BIT          = 32 - SET_BIT - BLK_OFFSET_BIT;
  localparam                  WAY_BIT          = `CLOG2(NUM_WAYS);

  reg [31:0]                  _dout;
  reg                         _is_output_valid;
  reg                         _is_hit;
  assign                      is_output_valid  = _is_output_valid;
  assign                      is_hit           = _is_hit;
  assign                      dout             = _dout;
```

```
  wire [TAG_BIT - 1:0]        tag              = addr[31:32 - TAG_BIT];
  wire [SET_BIT - 1:0]        set_index        = addr[31 - TAG_BIT:32 - TAG_BIT - SET_BIT];
  wire [BLK_OFFSET_BIT - 1:0] block_offset     = addr[31 - TAG_BIT - SET_BIT:0];

  reg [LINE_SIZE * 8 - 1:0]   cache_mem        [0:NUM_SETS-1][0:NUM_WAYS-1];
  reg                         valid_bits       [0:NUM_SETS-1][0:NUM_WAYS-1];
  reg [TAG_BIT - 1:0]         tags             [0:NUM_SETS-1][0:NUM_WAYS-1];
  reg                         dirty_bits       [0:NUM_SETS-1][0:NUM_WAYS-1];
  integer                     lru_bits         [0:NUM_SETS-1][0:NUM_WAYS-1];
  integer                     lru_counter;

  integer i, j, k;

  wire                        dmem_is_output_valid;
  wire [LINE_SIZE * 8 - 1:0]  dmem_dout;
  wire                        dmem_mem_ready;

  reg hit;
  reg [WAY_BIT - 1:0]         hit_way;

  assign is_ready = (state == 0);
```

위는 기본적인 내부 구조와 wire의 정의이다.

```
always @(*) begin
  hit = 0;
  hit_way = 0;
  if(state == 1) begin
    for (i = 0; i < NUM_WAYS; i = i + 1) begin
      if (valid_bits[set_index][i] && tags[set_index][i] == tag) begin
        hit = 1;
        hit_way = i[WAY_BIT - 1:0];
        break;
      end
    end
  end
end

integer victim;
reg flag;

always @(*) begin
  victim = 0;
  flag = 0;
  for (i = 0; i < NUM_WAYS; i = i + 1) begin
    if (!valid_bits[set_index][i]) begin
      victim = i;
      flag = 1;
      break;
    end
  end
  if (!flag) begin
    for (i = 0; i < NUM_WAYS; i = i + 1) begin
      if (lru_bits[set_index][victim] > lru_bits[set_index][i]) begin
        victim = i;
      end
    end
  end
end
```

위가 combinational한 hit/miss 판정, 아래가 combinational한 evict victim를 찾는 코드이다.

```
reg [2:0] state;

always @(posedge clk) begin
  if(reset) begin
    for (i = 0; i < NUM_SETS; i = i + 1) begin
      for (j = 0; j < NUM_WAYS; j = j + 1) begin
        valid_bits[i][j] <= 0;
        dirty_bits[i][j] <= 0;
        tags[i][j] <= 0;
        cache_mem[i][j]<= 0;
        lru_bits[i][j] <= 0;
      end
      lru_counter <= 0;
    end

    state <= 0;

    _dout <= 0;
    _is_hit <= 0;
    _is_output_valid <= 0;

    dmem_is_input_valid <= 0;
    dmem_addr <= 0;
    dmem_mem_read <= 0;
    dmem_mem_write <= 0;
    dmem_din <= 0;
  end
```

reset에서 cache를 clear한다.

```
  else begin
    if(state == 0) begin // IDLE
      dmem_is_input_valid <= 0;
      if(is_input_valid) state <= 1;
    end
    else if(state == 1) begin // COMPARE TAG
      begin
        /////////////// HIT ///////////////
        if(hit && mem_read) begin
          _dout <= cache_mem[set_index][hit_way][block_offset * 8 +: 32];
          _is_hit <= 1;
          _is_output_valid <= 1;

          lru_bits[set_index][hit_way] <= lru_counter;
          lru_counter <= lru_counter + 1;

          state <= 6; // Go back to IDLE state
        end
        else if(hit && mem_write) begin
          _is_hit <= 1;
          _is_output_valid <= 1;

          cache_mem[set_index][hit_way][block_offset * 8 +: 32] <= din;
          dirty_bits[set_index][hit_way] <= 1;

          lru_bits[set_index][hit_way] <= lru_counter;
          lru_counter <= lru_counter + 1;

          state <= 6; // Go back to IDLE state
        end
```

state 0, state 1 on hit 상태의 전이이다.

```
            /////////////// MISS ///////////////
            else if(!hit) begin
              _is_hit <= 0;
              _is_output_valid <= 0;

              if(valid_bits[set_index][victim] && dirty_bits[set_index][victim]) begin
                dmem_is_input_valid <= 1;
                dmem_addr <= {tags[set_index][victim], set_index, block_offset} >> BLK_OFFSET_BIT;
                dmem_mem_read <= 0;
                dmem_mem_write <= 1;
                dmem_din <= cache_mem[set_index][victim];

                state <= 4;
              end
              else begin
                dmem_is_input_valid <= 1;
                dmem_addr <= {tag, set_index, block_offset} >> BLK_OFFSET_BIT;
                dmem_mem_read <= 1;
                dmem_mem_write <= 0;
                dmem_din <= 0; // No data to write

                state <= 5;
              end
            end
          end
          else if(state == 4) state <= 2;
          else if(state == 5) state <= 3;
```

state 1에서 miss, state 4와 state 5의 전이를 보여준다.

```
      else if(state == 2) begin
        // Valid + evict + dirty --> memwrite
        dmem_is_input_valid <= 0;
        if(dmem_mem_ready) begin
          dmem_is_input_valid <= 1;
          dmem_addr <= {tag, set_index, block_offset} >> BLK_OFFSET_BIT;
          dmem_mem_read <= 1;
          dmem_mem_write <= 0;
          dmem_din <= 0; // No data to write

          state <= 5;
        end
      end
      else if(state == 3) begin // READ
        dmem_is_input_valid <= 0;
        if(dmem_is_output_valid) begin
          cache_mem[set_index][victim] <= dmem_dout;
          valid_bits[set_index][victim] <= 1;
          dirty_bits[set_index][victim] <= 0;
          tags[set_index][victim] <= tag;

          _dout <= dmem_dout[block_offset * 8 +: 32];
          _is_output_valid <= 1;
          _is_hit <= 0;

          lru_bits[set_index][victim] <= lru_counter;
          lru_counter <= lru_counter + 1;

          state <= 6;
          if(mem_write) begin
            cache_mem[set_index][victim][block_offset * 8 +: 32] <= din;
            dirty_bits[set_index][victim] <= 1;
          end
        end
      end
      else if(state == 6) begin
        _is_output_valid <= 0;
        state <= 0;
      end
    end
  end
```

state 2, state 3, state 6의 전이를 보여준다. 각각 dmem에 요청해야 할 부분이 있기에, 그 부분을

잘 설정하는 것을 볼 수 있다.

```
reg dmem_is_input_valid;
reg [31:0] dmem_addr;
reg dmem_mem_read;
reg dmem_mem_write;
reg [LINE_SIZE * 8 - 1:0] dmem_din;

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

  .is_input_valid(dmem_is_input_valid),
  .addr(dmem_addr),        // NOTE: address must be shifted by CLOG2(LINE_SIZE)
  .mem_read(dmem_mem_read),
  .mem_write(dmem_mem_write),
  .din(dmem_din),

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

dmem 모듈로 들어가는 포트를 정의하고, 그 값들을 이어준다.

4. Discussion

- Cache Design

- 주어진 파라미터에 따라 set index, tag, block-offset의 비트 수를 계산해서 캐시 설정을 동적으로 변경할 수 있도록 구현하였다.

- 따라서 파라미터를 조절함으로써 Direct-mapped cache와 Associative cache를 둘 다 구현할 수 있다.

- set index, tag, block-offset의 비트 수를 로컬 파라미터로 선언하여, 매개변수에 따라 동적으로 캐시를 생성할 수 있도록 하였다. 일반 레지스터 또는 와이어의 경우, 상수 식이 아니므로 다른 와이어나 레지스터의 비트 수를 결정하는데 사용하면 에러가 발생한다.

- 본 보고서에서는 LINE_SIZE = 16, NUM_SETS = 16, NUM_WAYS = 1로 설정하여 Direct-mapped cache를 구현하였고, LINE_SIZE = 16, NUM_SETS = 16, NUM_WAYS = 4로 설정하여 Associative cache를 구현하였다.

- Replacement Policy

이번 과제에서는 교체 정책으로 LRU(Least Recently Used)를 사용하였다.

3. Implementation의 구현과 같이, 조합 논리를 사용하여 victim 캐시 라인을 선택하는 로직을 구현하였다.

- LRU 정책에 따라서, 전체 WAY중에서 유효하지 않은 캐시 라인이 있는 경우 해당 라인을 victim으로 선택한다. 만약 모든 캐시 라인이 유효한 경우에는 LRU 값이 가장 작은 캐시 라인을 victim으로 선택한다.

- 어떤 캐시 라인에 접근이 발생하면, 해당 캐시 라인의 값을 lru_counter로 업데이트하고 lru_counter를 증가시킨다. 즉, 캐시 라인의 LRU 값이 크다는 것은 해당 캐시 라인이 가장 최근에 접근된 캐시 라인이라는 것을 의미한다.

- hit ratio 비교

```
› ./check (4-way Associative cache)
Naive Matmul
TOTAL CYCLE : 40916 (Answer : 71567)
Correct output : 32/32
Cache Access Count: 2499
Cache Hit Count: 1958
Cache Hit Rate: 0.783513
Opt Matmul
TOTAL CYCLE : 27656 (Answer : 76800)
Correct output : 32/32
Cache Access Count: 2499
Cache Hit Count: 2258
Cache Hit Rate: 0.903561
```

```
❯ ./check (Direct-mapped cache)
Naive matmul
TOTAL CYCLE : 74508 (Answer : 71567)
Correct output : 32/32
Cache Access Count: 2499
Cache Hit Count: 1687
Cache Hit Rate: 0.675070
Opt matmul
TOTAL CYCLE : 79760 (Answer : 76800)
Correct output : 32/32
Cache Access Count: 2499
Cache Hit Count: 1605
Cache Hit Rate: 0.642257
```

- Instruction 관점에서 Hit ratio를 비교하였다.

- 4-way associative cache의 경우, Naive matmul의 hit ratio는 0.783513, Optimized matmul의 hit ratio는 0.903561로 나타났다.

- 테스트 코드는 8 * 8 크기의 두 행렬을 곱하는 코드인데, Naive의 경우 단순히 행렬의 각 원소를 접근하는 방식으로 구현되어 있는 반면 opt matmul의 경우 `TILE=4`를 사용하여 캐시의 크기를 고려하여 한 번의 캐시 miss를 통해 가져온 값을 최대한 재사용하는 방법으로 구현되어 있다. 이러한 최적화가 캐시 설정과 잘 맞아 떨어져서 Naive matmul에 비해 Optimized matmul의 hit ratio가 더 높게 나타났다.

- 반면, Direct-mapped cache의 경우 Naive matmul의 hit ratio는 0.675070, Optimized matmul의 hit ratio는 0.642257로 나타났다.

- 먼저 4-way associative cache와 비교했을 때, Direct-mapped cache의 경우 hit ratio가 낮게 나타났다. 이는 Direct-mapped cache의 경우 각 캐시 라인이 오직 하나의 set에만 매핑되기 때문에, 동일한 set에 접근하는 다른 캐시 라인과 충돌이 발생할 가능성이 높아지기 때문이다.

- 또한 Directed-mapped cache의 경우에서는 Naive matmul의 hit ratio가 Optimized matmul의 hit ratio보다 낮게 나타났다.

- Limitation of this implementation
  - `NUM_SETS = 1, NUM_WAYS = 16`으로 설정하여 Fully Associative cache를 테스트하려 했으나, 의도된 구현 상에서 물리적으로 set index의 비트 수를 0으로 지정할 수 없었다.

5. Conclusion

위에서 설명한 구조대로 CPU를 구현하여 제공된 모든 테스트 벤치에 대해서 정답과 같은 시뮬레이션 레지스터 결과를 얻었다.

	Test Bench
Register Result (naive_matmul_unroll)	### SIMULATING ### TEST END SIM TIME : 81834 TOTAL CYCLE : 40916 (Answer : 71567) FINAL REGISTER OUTPUT 0 00000000 (Answer : 00000000) 1 00000000 (Answer : 00000000) 2 00002ffc (Answer : 00002ffc) 3 00000000 (Answer : 00000000) 4 00000000 (Answer : 00000000) 5 00000000 (Answer : 00000000) 6 00000000 (Answer : 00000000) 7 00000000 (Answer : 00000000) 8 00000000 (Answer : 00000000) 9 00000000 (Answer : 00000000) 10 00000000 (Answer : 00000000) 11 00000000 (Answer : 00000000) 12 00000000 (Answer : 00000000) 13 0000007e (Answer : 0000007e) 14 000004f3 (Answer : 000004f3) 15 000005f0 (Answer : 000005f0) 16 00000000 (Answer : 00000000) 17 0000000a (Answer : 0000000a) 18 00000000 (Answer : 00000000) 19 00000000 (Answer : 00000000) 20 00000000 (Answer : 00000000) 21 00000000 (Answer : 00000000) 22 00000000 (Answer : 00000000) 23 00000000 (Answer : 00000000) 24 00000000 (Answer : 00000000) 25 00000000 (Answer : 00000000) 26 00000000 (Answer : 00000000) 27 00000000 (Answer : 00000000) 28 00000000 (Answer : 00000000) 29 00000000 (Answer : 00000000) 30 00000000 (Answer : 00000000) 31 00000000 (Answer : 00000000) Correct output : 32/32

Register Result (opt_matmul_unroll)	### SIMULATING ### TEST END SIM TIME : 55314 TOTAL CYCLE : 27656 (Answer : 76800) FINAL REGISTER OUTPUT 0 00000000 (Answer : 00000000) 1 00000000 (Answer : 00000000) 2 00002ffc (Answer : 00002ffc) 3 00000000 (Answer : 00000000) 4 00000000 (Answer : 00000000) 5 00000000 (Answer : 00000000) 6 00000000 (Answer : 00000000) 7 00000000 (Answer : 00000000) 8 00000000 (Answer : 00000000) 9 00000000 (Answer : 00000000) 10 00000000 (Answer : 00000000) 11 00000000 (Answer : 00000000) 12 00000000 (Answer : 00000000) 13 0000007e (Answer : 0000007e) 14 000004f3 (Answer : 000004f3) 15 000005f0 (Answer : 000005f0) 16 00000000 (Answer : 00000000) 17 0000000a (Answer : 0000000a) 18 00000000 (Answer : 00000000) 19 00000000 (Answer : 00000000) 20 00000000 (Answer : 00000000) 21 00000000 (Answer : 00000000) 22 00000000 (Answer : 00000000) 23 00000000 (Answer : 00000000) 24 00000000 (Answer : 00000000) 25 00000000 (Answer : 00000000) 26 00000000 (Answer : 00000000) 27 00000000 (Answer : 00000000) 28 00000000 (Answer : 00000000) 29 00000000 (Answer : 00000000) 30 00000000 (Answer : 00000000) 31 00000000 (Answer : 00000000) Correct output : 32/32

위와 같이 정상 동작하는 모습을 확인할 수 있다.

4-way Associative Cache에서 사이클 수와 cache hit ratio는 다음과 같다.

```
Test basic
TOTAL CYCLE : 161 (Answer : 36)
Correct output : 32/32
Cache Access Count: 11
Cache Hit Count: 9
Cache Hit Rate: 0.818182

Test ifelse
TOTAL CYCLE : 177 (Answer : 44)
Correct output : 32/32
Cache Access Count: 15
Cache Hit Count: 13
Cache Hit Rate: 0.866667

Test loop
TOTAL CYCLE : 710 (Answer : 323)
Correct output : 32/32
Cache Access Count: 140
Cache Hit Count: 138
Cache Hit Rate: 0.985714

Test non-controlflow
TOTAL CYCLE : 166 (Answer : 46)
Correct output : 32/32
Cache Access Count: 8
Cache Hit Count: 6
Cache Hit Rate: 0.750000

Test recursive
TOTAL CYCLE : 2827 (Answer : 1188)
Correct output : 32/32
Cache Access Count: 396
Cache Hit Count: 380
Cache Hit Rate: 0.959596

Test naive_matmul_unroll
TOTAL CYCLE : 40916 (Answer : 71567)
Correct output : 32/32
Cache Access Count: 2499
Cache Hit Count: 1958
Cache Hit Rate: 0.783513

Test opt_matmul_unroll
TOTAL CYCLE : 27656 (Answer : 76800)
Correct output : 32/32
Cache Access Count: 2499
Cache Hit Count: 2258
Cache Hit Rate: 0.903561
```

6. References

[1] Lecture notes.