CSED311: Lab5 Cache

컴퓨터공학과 20220302 김지현

컴퓨터공학과 20220455 윤수인

1. Introduction

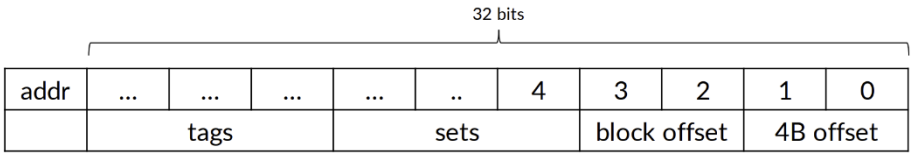
- 본 과제는 Cache 의 구조를 이해하고, Verilog 를 사용하여 Magic memory 대신 Blocking data cache 를 사용하는 형태의 cpu 를 직접 구현하는 것을 목적으로 한다.
- Cache 를 사용하여 데이터를 가져오는 데 드는 latency 를 줄이고, 자주 사용하는 데이터를 빠르게 가져올 수 있도록 한다.
- Verilator 환경에서 실행하며, 레지스터 값과 사이클 수, 각 시뮬레이션에 대한 hit ratio 를 관찰한다.
- Naïve matrix multiplication 과 Opt matrix multiplication 의 차이를 분석한다.

2. Design

Cache 는 자주 사용하는 데이터를 저장하여 memory 접근 시의 성능을 높이며, 주 메모리와 프로세서 사이의 속도 차이를 줄이는 데 중요한 역할을 한다.

본 과제에서는 이전의 Lab4 까지 사용하였던 Data memory 모듈에 직접 접근하는 방식 대신, Cache 를 사용하여 Data memory 에 접근하는 방식으로 구현하여 보다 효율적으로 데이터를 다룰 수 있도록 하였다. Cache 내에서 데이터를 처리한 방식은 다음과 같다.

- 데이터: data_bank 배열을 사용하여 Cache line 의 실제 데이터 블록을 저장한다. 각 Cache line 은 LINE_SIZE(16)만큼의 데이터를 포함한다.
- 태그: tag_bank 배열을 사용하여 각 Cache line 의 태그 정보를 저장한다. 입력 주소의 상위 비트(addr[31:8])를 사용하여 태그를 생성한다.
- Valid bit: valid_table 배열을 사용하여 각 Cache line 의 유효성, valid bit 를 저장한다. Cache line 이 invalid 하다면, 해당 Cache line 의 데이터는 사용할 수 있다.
- Dirty bit: dirty_table 배열을 사용하여 각 Cache line 의 수정 여부를 나타낸다. Dirty bit 가 설정된 경우, Cache line 의 데이터는 Data memory 의 값과 일치하지 않은 상태이다.



# sets	# ways	Block offset 0	Block offset 1	Block offset 2	Block offset 4
Set 0	Way 0	4B	4B	4B	4B
	Way 1				
	...				

<그림 1: Data cache design>

위 그림의 형태를 참고하여, Cache line 의 크기가 16 bytes 인 Direct-mapped cache 의 경우 NUM_SETS 는 16 이므로, set_index 의 크기는 4 bit, block_offset 의 크기는 2 bit, tag 의 크기는 24 bit 이다. 이에 따라 각각 tag 는 addr[31:8], set_index 는 addr[7:4], block_offset 은 addr[3:2]로 설정하였다.

Direct-mapped cache 의 형태로 구현하였으며, 그렇기 때문에 별 다른 Replacement policy 를 설정하는 대신, 1 way 에 있는 데이터에 접근하여 replace 하는 방법을 사용하였다. IDLE, COMPARE_TAG, ALLOCATE, WRITE_BACK 의 4 가지 state 를 사용하여 각 state 에서 적절한 동작을 수행하도록 하였다.

Cache 의 보다 자세한 구현은 Implementation 파트에서 추가로 설명한다.

Cache 가 추가됨에 따라, 기존에 존재하였던 다른 모듈 pc.v 와 cpu.v 에서 변경점이 발생하였다. Cache 를 통해 데이터를 가져오는 동안 사이클이 멈춰야 하므로, cpu.v 모듈에서 is_not_cache_stall 레지스터를 선언하여 이를 관리하였다.

is_not_cache_stall 은 캐시가 데이터를 가져오는 동안 파이프라인이 멈춰야 할지 여부를 나타내는 신호로, is_not_cache_stall = !((EX_MEM_mem_read | EX_MEM_mem_write) & !(is_ready & is_output_valid & is_hit));와 같이 할당된다.

여기서 EX_MEM_mem_read, EX_MEM_mem_write 는 각각 현재 명령어가 메모리 읽기 또는 쓰기 명령인지 나타내며, is_ready, is_output_valid, is_hit 는 Cache 모듈의 output 으로, 현재 cache 의 상태를 나타낸다. 따라서, Cache 가 준비되지 않았거나 Cache miss 가 발생한 경우 is_not_cache_stall 이 0 이 되어 파이프라인이 정지된다.

cpu.v 에서는 파이프라인 레지스터를 업데이트하는 과정에서 is_not_cache_stall 을 확인하여, 해당 시그널이 활성화된 경우 파이프라인을 정지시킨다.

pc.v 에서도 PCWrite 신호와 함께 is_not_cache_stall 을 확인하여 PC 의 업데이트를 지연할지 여부를 결정한다. 이렇게 함으로써 Cache 가 준비되지 않았거나, miss 가 발생하여 데이터를 가져오는 동안 지연이 발생하면 PC 가 갱신되지 않도록 한다. 이러한 변경으로 인해, 캐시를 통해 데이터를 가져오는 동안 CPU 는 파이프라인을 멈추고, 캐시가 준비되면 정상적으로 동작을 재개할 수 있게 된다.

각 모듈에 대한 구현은 Implementation 에서 보다 자세히 설명한다.

3. Implementation

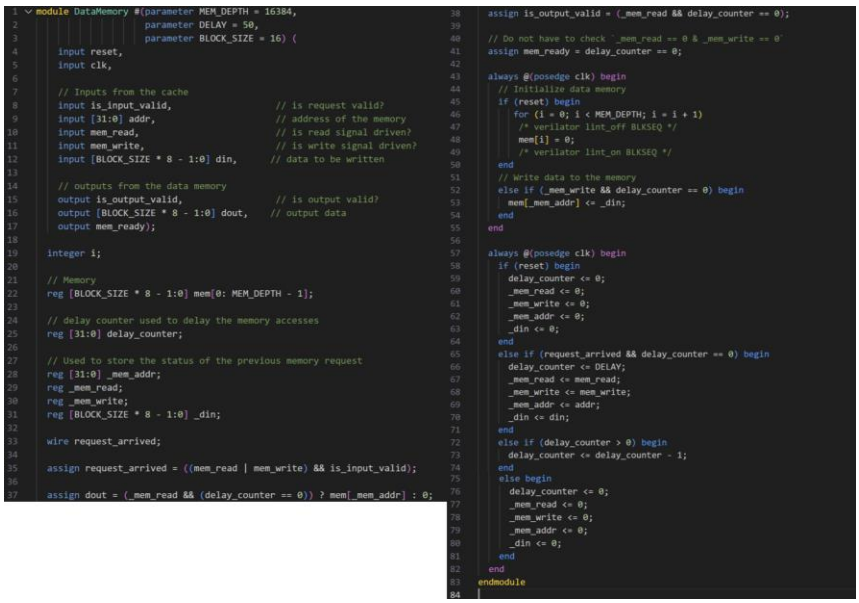
- **pc.v**

```
1   module pc(input reset,
2   input clk,
3   input PCWrite,
4   input is_not_cache_stall,
5   input [31:0] next_pc,
6   output reg [31:0] current_pc);
7
8       always @(posedge clk) begin
9           if(reset) begin
10              current_pc <= 0;
11          end
12          else begin
13              if(PCWrite & is_not_cache_stall) begin
14                  current_pc <= next_pc;
15              end
16          end
17    end
18
19  endmodule
20
```

<그림 2: pc.v 모듈의 구성>

Lab 4 의 pc.v 모듈에서 is_not_cache_stall 관련 부분이 추가되었다. pc.v 모듈에서는 reset 이 1 인 경우 current_pc 의 값을 0 으로 업데이트하고, reset 이 1 이 아닌 경우에는 PCWrite & is_not_cache_stall 의 값에 따라 current_pc 값이 달라진다. Cache stall 상황이 아닌 경우, 즉 Cache 에서 데이터를 성공적으로 가져온 경우나 Cache access 자체가 필요 없는 경우에만 PC 를 업데이트하므로, 기존의 PCWrite 와 함께 is_not_cache_stall 의 값도 함께 고려한다. PCWrite & is_not_cache_stall 이 1 인 경우, 다음 instruction 을 읽어와야 하므로 current_pc 의 값을 next_pc 의 값으로 업데이트한다. 반대로 0 인 경우, hazard 등의 상황에 의해 다음 instruction 을 읽어오지 말아야 할 상황이므로, current_pc 값을 업데이트하지 않는다. Input 으로 들어오는 is_not_cache_stall 에 대한 자세한 설명은 추후 cpu.v 의 변경점 부분을 참고한다. 이 과정은 clock synchronous 하게 이루어진다.

- **DataMemory.v**



```
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);

  integer i;

  // Memory
  reg [BLOCK_SIZE * 8 - 1:0] mem[0: MEM_DEPTH - 1];

  // delay counter used to delay the memory accesses
  reg [31:0] delay_counter;

  // Used to store the status of the previous memory request
  reg [31:0] _mem_addr;
  reg _mem_read;
  reg _mem_write;
  reg [BLOCK_SIZE * 8 - 1:0] _din;

  wire request_arrived;

  assign request_arrived = ((mem_read | mem_write) && is_input_valid);

  assign dout = (_mem_read && (delay_counter == 0)) ? mem[_mem_addr] : 0;

  assign is_output_valid = (_mem_read && delay_counter == 0);

  // Do not have to check `_mem_read == 0 & _mem_write == 0`
  assign mem_ready = delay_counter == 0;

  always @(posedge clk) begin
    // Initialize data memory
    if (reset) begin
      for (i = 0; i < MEM_DEPTH; i = i + 1)
        /* verilator lint_off BLKSEQ */
        mem[i] = 0;
        /* verilator lint_on BLKSEQ */
    end
    // Write data to the memory
    else if (_mem_write && delay_counter == 0) begin
      mem[_mem_addr] <= _din;
    end
  end

  always @(posedge clk) begin
    if (reset) begin
      delay_counter <= 0;
      _mem_read <= 0;
      _mem_write <= 0;
      _mem_addr <= 0;
      _din <= 0;
    end
    else if (request_arrived && delay_counter == 0) begin
      delay_counter <= DELAY;
      _mem_read <= mem_read;
      _mem_write <= mem_write;
      _mem_addr <= addr;
      _din <= din;
    end
    else if (delay_counter > 0) begin
      delay_counter <= delay_counter - 1;
    end
    else begin
      delay_counter <= 0;
      _mem_read <= 0;
      _mem_write <= 0;
      _mem_addr <= 0;
      _din <= 0;
    end
  end
endmodule
```

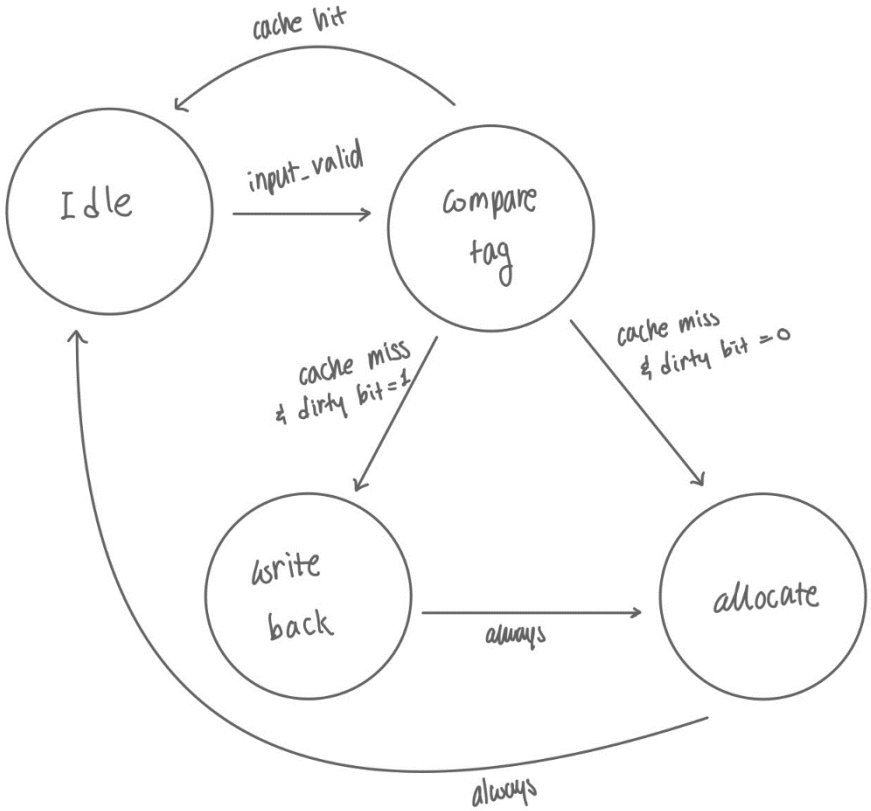
<그림 3: DataMemory.v 모듈의 구성>

DataMemory.v 모듈은 새로 제공된 코드를 사용하였다. cpu 에서 Cache 모듈을 호출하고, 만약 Cache miss 가 발생한 경우 Data Memory 로부터 필요한 데이터를 가져오는 방식으로 사용된다. 기존 Lab4 까지의 DataMemory.v 모듈에 새로운 input 인 is_input_valid 와, 새로운 output 인 is_output_valid, mem_ready 가 추가되었다. Lab5 에서는 Magic memory 대신 Blocking data cache 를 사용하므로, 데이터를 가져오는 데에 걸리는 delay 를 고려하기 위해 delay counter 를 사용한다.

- is_input_valid: Cache miss 발생 시 Data memory 에서 데이터를 가져오거나(allocate), 데이터를 메모리에 write back 해야 한다. 이처럼 Data memory 를 사용할 때, is_input_valid 의 값을 1 로 하여 Data memory 의 input 이 유효한지 확인하고, 원하는 동작을 수행하도록 한다.
- is_output_valid: mem_read 가 활성화되어 있고, delay_counter 가 0 이 된 경우 is_output_valid 의 값을 1 로 하여, Cache 모듈 내에서 Data memory 의 output data 를 사용해도 됨을 나타내는 시그널이다.
- mem_ready: Data memory 를 사용해도 되는지 여부를 나타내는 시그널로, delay counter 가 0 이 되었을 경우, 즉 데이터를 가져올 준비가 된 경우 mem_ready 를 1 로 설정한다.

- **Cache.v**

Direct-mapped cache 로 구현하였으며, 자세한 구현은 Cache.v 코드 파일을 참고한다. IDLE, COMPARE_TAG, ALLOCATE, WRITE_BACK 의 4 가지 state 로 구성하였으며, FSM 은 다음과 같다.



<그림 4: FSM design>

- IDLE 상태
  - input wire 중 is_input_valid 시그널이 활성화되면, COMPARE_TAG 상태로 전환한다. 입력 신호가 비활성화된 경우, IDLE 상태를 유지한다.
- COMPARE_TAG 상태
  - 입력된 주소 addr 은 미리 tag(input_tag), set index(input_set_index), block offset(input_block_offset)으로 분해된다. 현재 set 의 tag(cache_tag_read)와 valid bit(cache_valid)를 확인하여 캐시 히트 여부(is_hit)를 결정한다. **is_hit = (input_tag == cache_tag_read) & cache_valid** 로 설정된다.

- 캐시 히트 시, 캐시의 작업이 끝났음을 나타내는 시그널(cache_is_output_valid)의 값을 1 로 설정한다. Read 작업인 경우 캐시의 output 데이터(cache_dout)를 블록 오프셋을 고려한 data_read 의 일부분으로 설정한다. Write 작업인 경우(mem_rw == 1) 캐시 내의 데이터를 갱신하고 dirty bit 를 설정한다. 작업이 완료된 후 IDLE 상태로 전환된다.
- 캐시 미스 시, dirty bit(cache_dirty)가 0 인지 확인한다. Dirty bit 가 0 인 경우, Data memory 로부터 데이터를 로드하기 위해 ALLOCATE 상태로 전환된다. Dirty bit 가 1 인 경우, 현재 데이터를 Data memory 에 기록하기 위해 WRITE_BACK 상태로 전환된다. 각 상태에 알맞게 dmem_read, dmem_write, dmem_addr 를 적절히 설정한다.

- ALLOCATE 상태
  - Data Memory 에서 새로운 데이터를 읽어 와 캐시 라인에 저장한다. 데이터가 메모리에서 준비되면, 캐시 라인의 데이터를 갱신하고 tag 업데이트 및, valid bit 설정 작업을 수행한다. 만약 write 작업인 경우 dirty bit 도 설정한다. 작업이 완료되면 IDLE 상태로 전환된다.
- WRITE_BACK 상태
  - dirty bit 가 1 인 캐시 라인의 데이터를 Data Memory 에 기록한 후, 새로운 데이터를 읽어오기 위해 ALLOCATE 상태로 전환된다.

각각의 로직을 코드와 함께 설명하자면 다음과 같다.

```
assign input_tag = addr[31:8];
assign input_set_index = addr[7:4];
assign input_block_offset = addr[3:2];
assign clog2 = `CLOG2(LINE_SIZE);

assign is_ready = is_data_mem_ready;
assign is_output_valid = cache_is_output_valid;
assign dout = cache_dout;
assign is_hit = (input_tag == cache_tag_read) & cache_valid;
assign cache_data_read = data_bank[input_set_index];
assign cache_tag_read = tag_bank[input_set_index];
assign cache_valid = valid_table[input_set_index];
assign cache_dirty = dirty_table[input_set_index];
```

<그림 5: Cache 모듈 구성에 필요한 wire assign>

입력된 주소 addr 을 tag, set_index, block_offset 으로 나누어 할당하는 과정, cache 의 output 으로 사용될 wire 에 각각 레지스터를 할당하는 과정 및 hit 여부를 판단하는 is_hit, 현재 Cache 내의 값을 나타내는 cache_data_read, cache_tag_read, cache_valid, cache_dirty 를 할당하는 과정은 assign 을 사용하여 clock asynchronous 하게 이루어진다.

```
always @(posedge clk) begin
  if(reset) begin
    for(i = 0; i < 16; i = i + 1) begin
      data_bank[i] <= 0;
      tag_bank[i] <= 0;
      valid_table[i] <= 0;
      dirty_table[i] <= 0;
    end
  end
end
always @(posedge clk) begin
    if(cache_data_write_enable) begin
      data_bank[input_set_index] <= cache_write_data;
    end
    if(cache_tag_write_enable) begin
      tag_bank[input_set_index] <= cache_write_tag;
      valid_table[input_set_index] <= cache_valid_bit;
      dirty_table[input_set_index] <= cache_dirty_bit;
    end
end
```

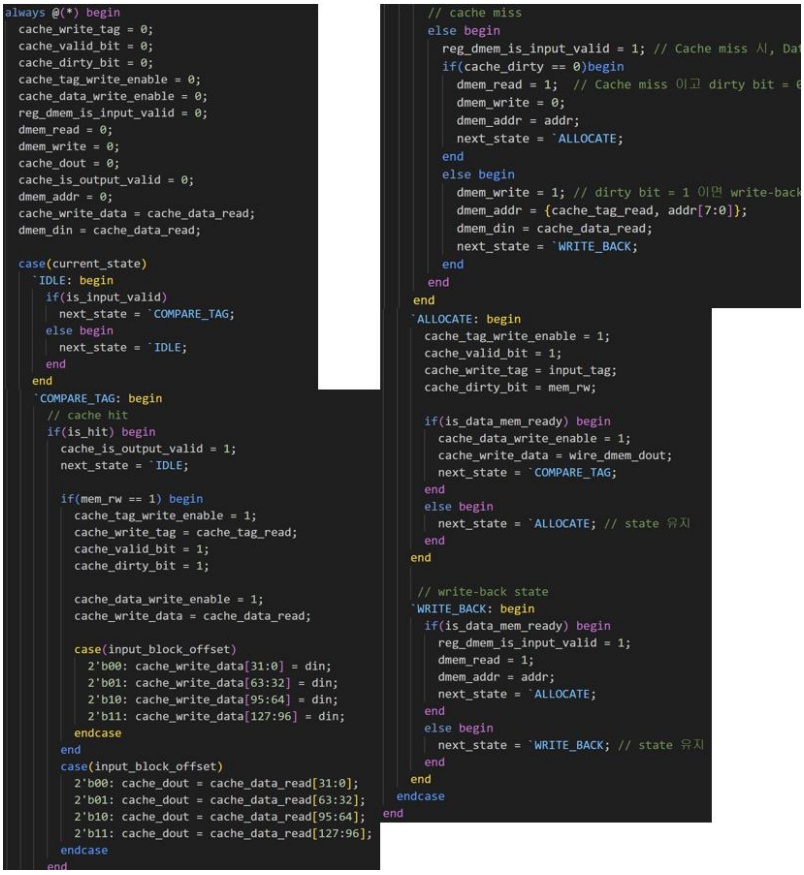
<그림 6: data_bank, tag_bank, valid_table, dirty_table 관리>

Reset 시그널이 활성화되었을 때 data_bank, tag_bank, valid_table 을 초기화하고, 일반적인 상태에서 cache_data_write_enable, cache_tag_write_enable 시그널에 따라 각각 값을 갱신하는 과정은 clock synchronous 하게 이루어진다.

```
always @(posedge clk) begin
  if(reset) begin
    current_state <= `IDLE;
  end
  else begin
    current_state <= next_state;
  end
end
```

<그림 7: state update>

Reset 시그널이 활성화되었을 때 현재 상태(current_state)를 IDLE 로 초기화하고, 일반적인 상태에서는 다음 상태로 전환하는 로직은 clock synchronous 하게 이루어진다.



```
always @(*) begin
  cache_write_tag = 0;
  cache_valid_bit = 0;
  cache_dirty_bit = 0;
  cache_tag_write_enable = 0;
  cache_data_write_enable = 0;
  reg_dmem_is_input_valid = 0;
  dmem_read = 0;
  dmem_write = 0;
  cache_dout = 0;
  cache_is_output_valid = 0;
  dmem_addr = 0;
  cache_write_data = cache_data_read;
  dmem_din = cache_data_read;

  case(current_state)
    `IDLE: begin
      if(is_input_valid)
        next_state = `COMPARE_TAG;
      else begin
        next_state = `IDLE;
      end
    end
    `COMPARE_TAG: begin
      // cache hit
      if(is_hit) begin
        cache_is_output_valid = 1;
        next_state = `IDLE;

        if(mem_rw == 1) begin
          cache_tag_write_enable = 1;
          cache_write_tag = cache_tag_read;
          cache_valid_bit = 1;
          cache_dirty_bit = 1;

          cache_data_write_enable = 1;
          cache_write_data = cache_data_read;

          case(input_block_offset)
            2'b00: cache_write_data[31:0] = din;
            2'b01: cache_write_data[63:32] = din;
            2'b10: cache_write_data[95:64] = din;
            2'b11: cache_write_data[127:96] = din;
          endcase
        end
        case(input_block_offset)
          2'b00: cache_dout = cache_data_read[31:0];
          2'b01: cache_dout = cache_data_read[63:32];
          2'b10: cache_dout = cache_data_read[95:64];
          2'b11: cache_dout = cache_data_read[127:96];
        endcase
      end
      // cache miss
      else begin
        reg_dmem_is_input_valid = 1; // Cache miss 시, Da
        if(cache_dirty == 0)begin
          dmem_read = 1;  // Cache miss 이고 dirty bit = 0
          dmem_write = 0;
          dmem_addr = addr;
          next_state = `ALLOCATE;
        end
        else begin
          dmem_write = 1; // dirty bit = 1 이면 write-back
          dmem_addr = {cache_tag_read, addr[7:0]};
          dmem_din = cache_data_read;
          next_state = `WRITE_BACK;
        end
      end
    end
    `ALLOCATE: begin
      cache_tag_write_enable = 1;
      cache_valid_bit = 1;
      cache_write_tag = input_tag;
      cache_dirty_bit = mem_rw;

      if(is_data_mem_ready) begin
        cache_data_write_enable = 1;
        cache_write_data = wire_dmem_dout;
        next_state = `COMPARE_TAG;
      end
      else begin
        next_state = `ALLOCATE; // state 유지
      end
    end

    // write-back state
    `WRITE_BACK: begin
      if(is_data_mem_ready) begin
        reg_dmem_is_input_valid = 1;
        dmem_read = 1;
        dmem_addr = addr;
        next_state = `ALLOCATE;
      end
      else begin
        next_state = `WRITE_BACK; // state 유지
      end
    end
  endcase
end
```

<그림 8: 다음 state 결정 및 Cache 로직 처리>

Cache 의 상태와 입력 신호에 따라 다음 상태를 결정하고 필요한 제어 신호를 설정하는 역할을 한다. 이 로직은 위에서 서술한 바와 같이 동작하며, clock asynchronous 하게 처리되었다.

- **cpu.v**

기존의 cpu.v 코드에서 Data memory 에 직접적으로 접근하는 대신 Cache 모듈을 호출하여 접근하는 식으로 수정되었다. Input wire 중 mem_rw 는 mem_read 가 활성화되어 있을 때 0, mem_write 가 활성화되어 있을 때 1 로 설정된다. 이는 EX_MEM pipeline register update 과정에서 ID_EX_mem_read, ID_EX_mem_write 의 값에 따라 설정된다.

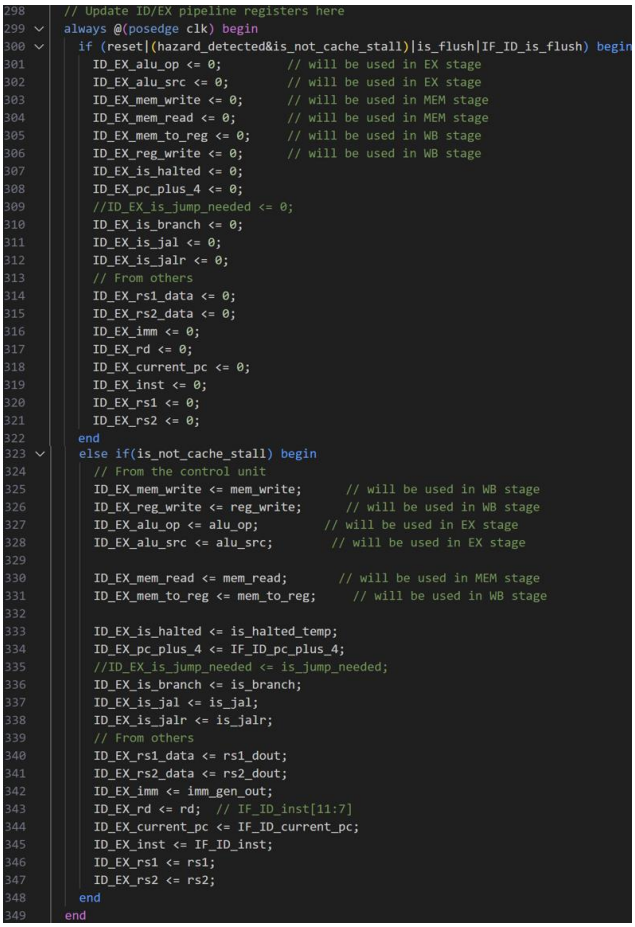
```
449   // ---------- Cache -----------
450   Cache cache(
451     .reset(reset),
452     .clk(clk),
453     .is_input_valid(EX_MEM_mem_write | EX_MEM_mem_read),
454     .addr(EX_MEM_alu_out),
455     .mem_rw(mem_rw),
456     .din(EX_MEM_dmem_data),
457     .is_ready(is_ready),
458     .is_output_valid(is_output_valid),
459     .dout(data_out),
460     .is_hit(is_hit)
461   );
```

<그림 9: cpu.v 모듈 내 Cache 사용>

또한 위에서 언급한 바와 같이, is_not_cache_stall 레지스터를 선언하고 !((EX_MEM_mem_read|EX_MEM_mem_write)&!(is_ready & is_output_valid & is_hit));로 할당하여 파이프라인 stall 여부를 판단하였다. 각 파이프라인 레지스터 업데이트 과정에서 is_not_cache_stall 의 값을 확인한다. 해당 과정은 clock synchronous 하게 이루어진다.

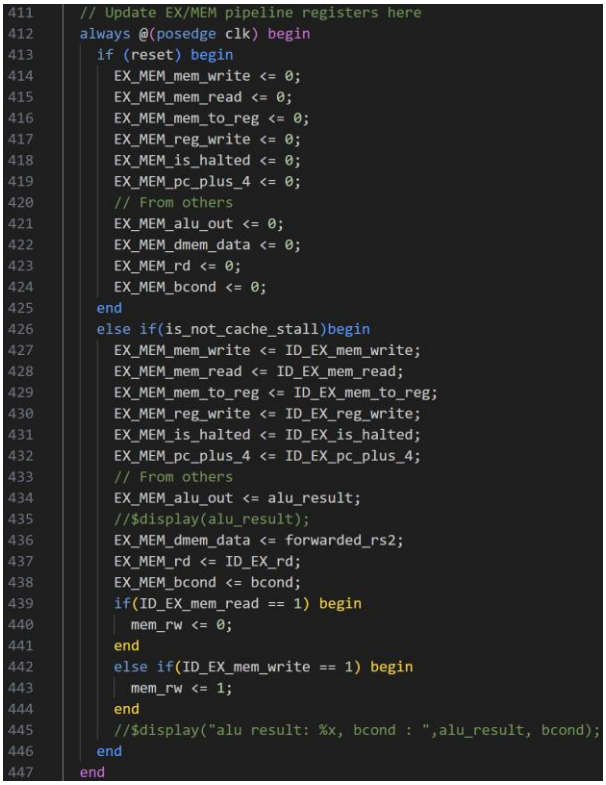
```
// Update IF/ID pipeline registers here
always @(posedge clk) begin
  if (reset) begin
    IF_ID_inst <= 0;
    IF_ID_current_pc <= 0;
    IF_ID_pc_plus_4 <= 0;
    IF_ID_is_flush <= 0;
  end
  else begin
    if((!hazard_detected)&is_not_cache_stall) begin
      IF_ID_inst <= inst_out;
      IF_ID_current_pc <= current_pc;
      IF_ID_pc_plus_4 <= pc_plus_4;
      IF_ID_is_flush <= is_flush;
      //$display("%h", IF_ID_inst);
    end
  end
end
```

<그림 10: IF_ID pipeline register update 시 is_not_cache_stall 확인>



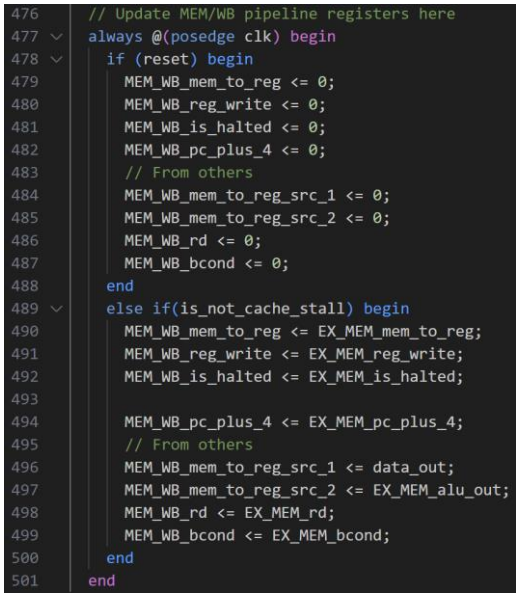
```
// Update ID/EX pipeline registers here
always @(posedge clk) begin
  if (reset|(hazard_detected&is_not_cache_stall)|is_flush|IF_ID_is_flush) begin
    ID_EX_alu_op <= 0;         // will be used in EX stage
    ID_EX_alu_src <= 0;        // will be used in EX stage
    ID_EX_mem_write <= 0;      // will be used in MEM stage
    ID_EX_mem_read <= 0;       // will be used in MEM stage
    ID_EX_mem_to_reg <= 0;     // will be used in WB stage
    ID_EX_reg_write <= 0;      // will be used in WB stage
    ID_EX_is_halted <= 0;
    ID_EX_pc_plus_4 <= 0;
    //ID_EX_is_jump_needed <= 0;
    ID_EX_is_branch <= 0;
    ID_EX_is_jal <= 0;
    ID_EX_is_jalr <= 0;
    // From others
    ID_EX_rs1_data <= 0;
    ID_EX_rs2_data <= 0;
    ID_EX_imm <= 0;
    ID_EX_rd <= 0;
    ID_EX_current_pc <= 0;
    ID_EX_inst <= 0;
    ID_EX_rs1 <= 0;
    ID_EX_rs2 <= 0;
  end
  else if(is_not_cache_stall) begin
    // From the control unit
    ID_EX_mem_write <= mem_write;      // will be used in WB stage
    ID_EX_reg_write <= reg_write;      // will be used in WB stage
    ID_EX_alu_op <= alu_op;         // will be used in EX stage
    ID_EX_alu_src <= alu_src;        // will be used in EX stage

    ID_EX_mem_read <= mem_read;       // will be used in MEM stage
    ID_EX_mem_to_reg <= mem_to_reg;     // will be used in WB stage

    ID_EX_is_halted <= is_halted_temp;
    ID_EX_pc_plus_4 <= IF_ID_pc_plus_4;
    //ID_EX_is_jump_needed <= is_jump_needed;
    ID_EX_is_branch <= is_branch;
    ID_EX_is_jal <= is_jal;
    ID_EX_is_jalr <= is_jalr;
    // From others
    ID_EX_rs1_data <= rs1_dout;
    ID_EX_rs2_data <= rs2_dout;
    ID_EX_imm <= imm_gen_out;
    ID_EX_rd <= rd;  // IF_ID_inst[11:7]
    ID_EX_current_pc <= IF_ID_current_pc;
    ID_EX_inst <= IF_ID_inst;
    ID_EX_rs1 <= rs1;
    ID_EX_rs2 <= rs2;
  end
end
```

<그림 11: ID_EX pipeline register update 시 is_not_cache_stall 확인>



```
411   // Update EX/MEM pipeline registers here
412   always @(posedge clk) begin
413     if (reset) begin
414       EX_MEM_mem_write <= 0;
415       EX_MEM_mem_read <= 0;
416       EX_MEM_mem_to_reg <= 0;
417       EX_MEM_reg_write <= 0;
418       EX_MEM_is_halted <= 0;
419       EX_MEM_pc_plus_4 <= 0;
420       // From others
421       EX_MEM_alu_out <= 0;
422       EX_MEM_dmem_data <= 0;
423       EX_MEM_rd <= 0;
424       EX_MEM_bcond <= 0;
425     end
426     else if(is_not_cache_stall)begin
427       EX_MEM_mem_write <= ID_EX_mem_write;
428       EX_MEM_mem_read <= ID_EX_mem_read;
429       EX_MEM_mem_to_reg <= ID_EX_mem_to_reg;
430       EX_MEM_reg_write <= ID_EX_reg_write;
431       EX_MEM_is_halted <= ID_EX_is_halted;
432       EX_MEM_pc_plus_4 <= ID_EX_pc_plus_4;
433       // From others
434       EX_MEM_alu_out <= alu_result;
435       //$display(alu_result);
436       EX_MEM_dmem_data <= forwarded_rs2;
437       EX_MEM_rd <= ID_EX_rd;
438       EX_MEM_bcond <= bcond;
439       if(ID_EX_mem_read == 1) begin
440         mem_rw <= 0;
441       end
442       else if(ID_EX_mem_write == 1) begin
443         mem_rw <= 1;
444       end
445       //$display("alu result: %x, bcond : ",alu_result, bcond);
446     end
447   end
```

<그림 12: EX_MEM pipeline register update 시 is_not_cache_stall 확인>



```
476   // Update MEM/WB pipeline registers here
477   always @(posedge clk) begin
478     if (reset) begin
479       MEM_WB_mem_to_reg <= 0;
480       MEM_WB_reg_write <= 0;
481       MEM_WB_is_halted <= 0;
482       MEM_WB_pc_plus_4 <= 0;
483       // From others
484       MEM_WB_mem_to_reg_src_1 <= 0;
485       MEM_WB_mem_to_reg_src_2 <= 0;
486       MEM_WB_rd <= 0;
487       MEM_WB_bcond <= 0;
488     end
489     else if(is_not_cache_stall) begin
490       MEM_WB_mem_to_reg <= EX_MEM_mem_to_reg;
491       MEM_WB_reg_write <= EX_MEM_reg_write;
492       MEM_WB_is_halted <= EX_MEM_is_halted;
493
494       MEM_WB_pc_plus_4 <= EX_MEM_pc_plus_4;
495       // From others
496       MEM_WB_mem_to_reg_src_1 <= data_out;
497       MEM_WB_mem_to_reg_src_2 <= EX_MEM_alu_out;
498       MEM_WB_rd <= EX_MEM_rd;
499       MEM_WB_bcond <= EX_MEM_bcond;
500     end
501   end
```

<그림 13: MEM_WB pipeline register update 시 is_not_cache_stall 확인>

이외 모듈들은 Lab4 에서와 동일하므로 생략한다.

4. Discussion

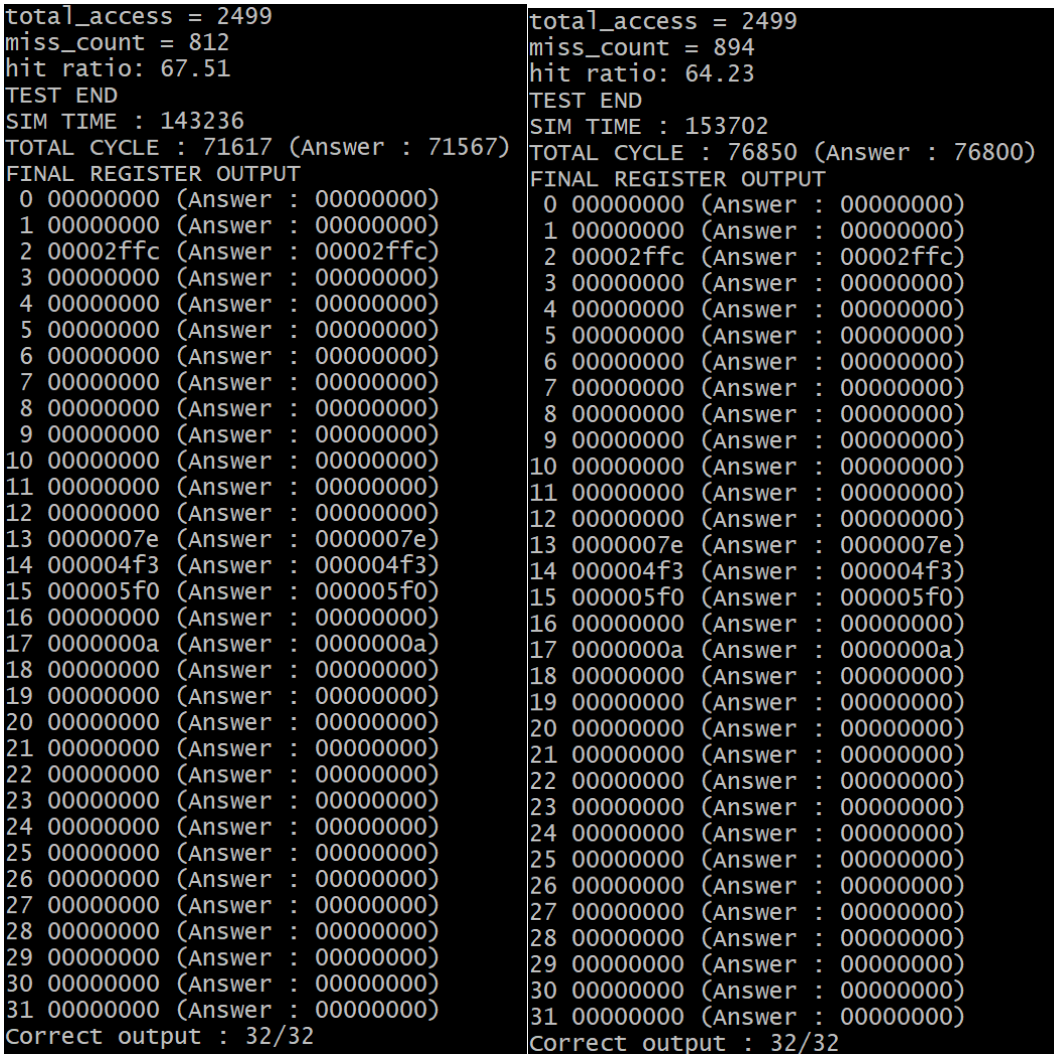
%display 를 이용하여 Cygwin terminal 상에서 Total access 와 miss count, hit ratio 를 출력하였다. 주어진 testbench 에 대한 시뮬레이션 결과 레지스터 값이 올바르게 출력되는 것을 확인할 수 있었다. 좌측이 Naïve matrix multiplication 에 대한 결과, 우측이 Opt matrix multiplication 에 대한 결과이다.

Naïve matrix multiplication 의 hit ratio 는 67.51%, Opt matrix multiplication 의 hit ratio 는 64.23%이다. Naïve 의 경우가 Opt 보다 hit ratio 가 높은 것은 Direct-mapped cache 로 구현하였기 때문으로 추측된다. N-way Set-associative cache 방식으로 구현하여 N 의 값을 증가시키면 Opt matrix multiplication 의 hit ratio 가 Naïve matrix multiplication 보다 높아질 것이다.

이번 랩 과제에서 주어진 naive matrix multiplication 과 opt matrix multiplication 의 차이를 확인해보자면, naive matrix multiplication 을 사용해 A X B 를 구하는 경우 행렬 A 의 i 번째 행과 행렬 B 의 j 번째 열을 순차적으로 곱하고 더하게 되는데 여기서 행렬 B 의 경우 접근이 연속적이지 않아 cache miss 발생률이 높게 나타난다.

하지만 opt matrix multiplication (tile approach)의 경우 행렬을 작은 "tile" 로 나눠서 해당 elements 를 메모리에 로드하여 작업을 하게 된다. Cache miss 를 줄이기 위해서는 locality of reference 를 줄여야하는데 (temporal locality 와 spatial locality), 이렇게 tile 단위로 작업을 하게 되면 각 tile 내부에서 연속적인 메모리 접근이 이루어져서 spatial locality 가 높아짐을 알 수 있다.

Naive approach 에서는 메모리 접근이 연속되진 않지만 메모리를 좀더 분산되게 접근하기 때문에 같은 캐시 라인에 data 가 mapping 되면서 생기는 conflict miss 의 확률이 opt approach 보다 적을 수 있다. 하지만 set associative cache 를 사용한다면 opt approach 에서 가지는 다른 데이터가 같은 캐시라인에 mapping 됨으로서 생기는 cache miss 의 경우를 줄일 수 있을 것이라고 예상된다.



<그림 14: 시뮬레이션 결과 및 Hit ratio>

5. Conclusion

이번 Lab5 에서는 Cache 를 이용한 CPU 를 구현하였다. 2-way set associative cache 를 구현하는 것을 목표로 하였으나, way_index selection 과정에 어려움이 있어 Direct-mapped cache 의 구현에 그친 것이 아쉽지만, 두 방식 모두에 대해 공부하며 Cache 의 동작에 대해 이해할 수 있었다. Write-back & Allocate 방식으로 구현한 이번 과제와 달리, Write-through & No Allocate 방식으로 구현한 Cache 에 대해서도 탐구할 기회가 있으면 좋겠다.