Lab 6 - Cache

20160595 수학과 최규민
20160169 물리학과 최수민

1. Introduction

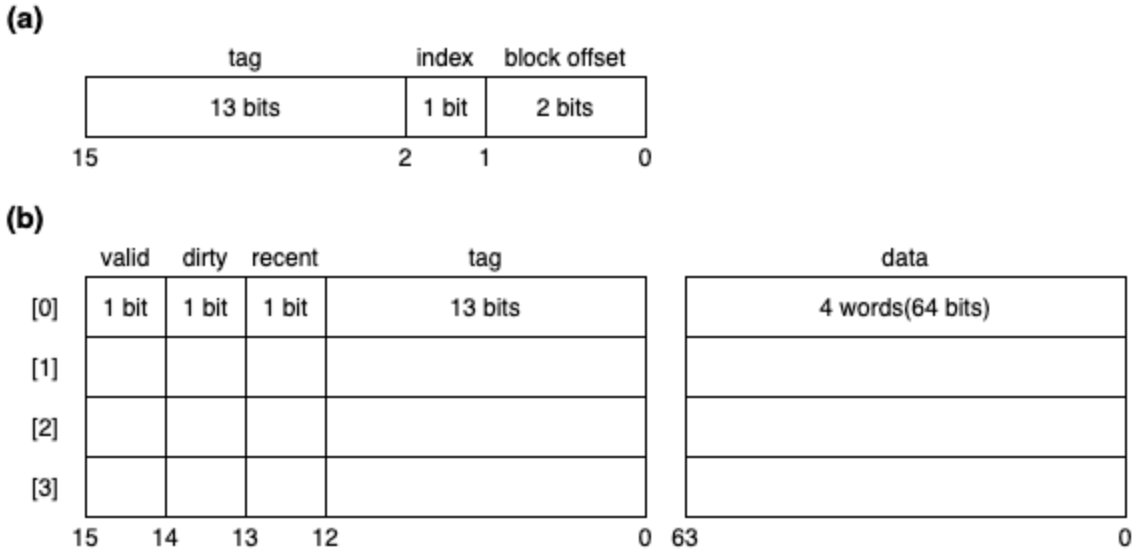
이번 과제에서는 set-associative 캐시를 구현하고 이로 인한 성능 향상을 평가한다. 캐시는 CPU와 메모리 사이에 위치하여 두 장치의 속도 차이로 인한 병목 현상을 완화하기 위해 사용된다. 일반적으로 한 번 사용한 데이터는 다시 사용할 가능성이 높고 그 주변의 데이터도 곧 사용할 가능성이 높아 데이터 지역성을 가지고 있는데, 이를 이용하여 캐시는 메모리에 저장된 프로그램과 데이터의 일부를 불러오고, CPU에서 필요한 데이터를 캐시에서 먼저 찾도록 하여 성능의 향상을 가져온다. 이러한 캐시는 set associativity, replacement policy, write policy, separate I/D cache 등의 다양한 디자인 선택 요소가 있는데, 이러한 요소들을 고려하여 캐시를 구현하고, 캐시가 없을 때와 비교해 성능이 얼마나 향상되는지를 비교한다.
이번 과제를 통해 배워야 하는 것은 다음의 두 가지로 요약할 수 있다.

1. 캐시 설계 시 고려해야 할 디자인 사항들을 파악하고, 이를 적절하게 선택하여 설계한다.
2. 캐시 구현을 통해 어떻게 캐시가 성능의 향상을 가져올 수 있는지 이해한다.

2. Design

먼저 주어진 캐시의 구성은 2-way set associative, single-level cache로, 용량은 32 words, 각 캐시 라인의 크기는 4 words로 주어졌다. 이로부터 캐시 전체의 line의 개수는 8개이고, 메모리에 요청하는 주소 16비트 중 Block offset은 2 bits로 정해졌다.

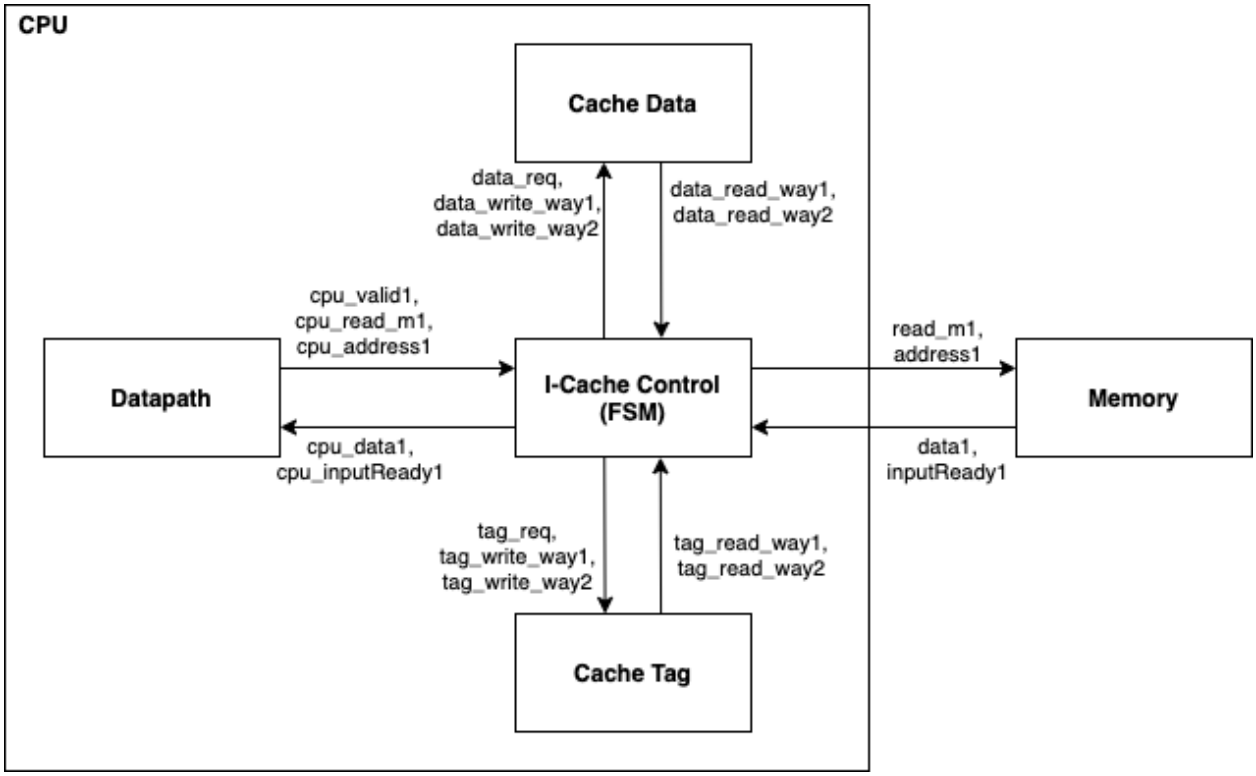
주어진 구성 외의 디자인 결정사항은 다음과 같이 정했다. 먼저 Instruction cache(I-Cache)와 Data cache(D-Cache)를 구분하기로 했다. 따라서 두 캐시는 각각 4 lines씩 가지며, 2-way set associative이므로 하나의 set 당 2개의 line을 가진다. 따라서 index bit는 1 bit로 결정되었다. Replacement policy는 2-way set associative이므로 LRU(Least recently used) 구현이 그렇게 어렵지 않아 LRU로 결정했고, Recent bit를 추가로 저장하도록 하여 가장 최근에 사용된 tag가 1, 그렇지 않은 tag가 0을 저장하도록 했다. Write policy는 Write-back, Write allocate로 결정했다. 이로부터 각 tag별로 Dirty bit를 추가하였다. 그림 1에 이렇게 결정된 주소와 캐시의 구성을 나타냈다.



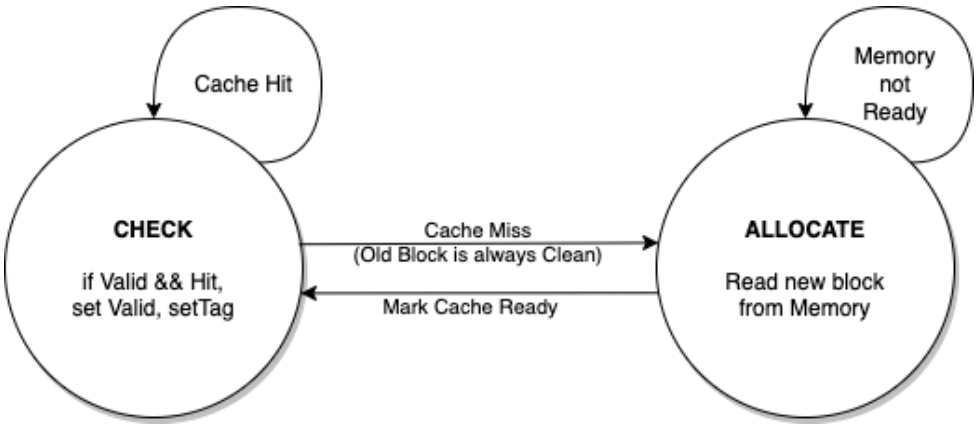
[그림 1] (a) 16비트 주소의 Field 구성. LSB부터 차례로 2비트는 Block offset, 1비트는 Index, 나머지 13비트는 Tag가 된다. (b) 각 캐시의 구성. I-Cache와 D-Cache를 분리해 각 캐시별로 4 lines를 저장하게 된다. 이를 왼쪽의 [0]~[3]까지의 인덱스를 가지는 것으로 나타냈다. [0]과 [2]에 해당하는 line이 하나의 set이 되고, [1]과 [3]에 해당하는 line이 하나의 set이 된다. 각 캐시 line은 4 words로 구성되며, 각 line별로 tag 13비트, recent bit 1비트, dirty bit 1비트, valid bit 1비트를 가진다.

설계 선택 사항들을 결정한 후, 캐시와 메모리, CPU간의 interface와 캐시의 컨트롤을 하기 위한 FSM(Finite state machine)을 설계하였다. 먼저 I-Cache의 경우, 이번 구현에서 instruction을 메모리에 쓰는 경우는 없으므로 메모리의 address1과 관련된 포트들만 사용하고 메모리에서 데이터를 읽어올 수 있으면 된다. 따라서 캐시는 datapath로부터 요청을 받고 이를 Cache Tag에 저장되어 있는 tag들과 비교한 뒤, 일치하는 유효한 데이터를 찾으면 그 데이터를 바로 리턴한다. 만약 일치하는 유효한 데이터가 없다면 메모리에 데이터를 요청하고, 메모리로부터 데이터를 읽어온 뒤 캐시 데이터와 태그를 업데이트함과 동시에 Datapath에도 데이터를 보내준다. I-Cache와 메모리, datapath에서 주고 받는 데이터(혹은 포트)는 그림 2에 나타냈다.

I-Cache에서 요청한 데이터가 있는지 확인하고 없으면 Memory에 요청하는 일련의 동작을 수행하기 위해 FSM을 설계하였다. FSM은 크게 CHECK와 ALLOCATE의 두 단계로 구성되어 있다. CHECK 상태에는 Datapath로부터 들어온 요청에 대해 캐시에 해당 데이터가 있는지 확인하여 Cache hit이 되면 해당 데이터를 리턴하고 계속 CHECK 상태로 남는다. 만약 Cache miss가 발생하면 메모리에 데이터 요청을 보내고 ALLOCATE 상태로 전환한다. ALLOCATE 상태에서는 메모리에서 데이터를 보낼 때까지 기다리며, 메모리에서 데이터를 보내면 Datapath에 알맞은 데이터를 반환하고, 캐시를 업데이트한 뒤 CHECK 상태로 전환한다. 이 때 Instruction은 메모리에 쓰는 동작이 없기 때문에 모든 캐시 line이 항상 Clean한 상태로, Dirty line을 메모리에 쓰는 state는 없다. I-Cache의 FSM은 그림 3에 나타냈다.



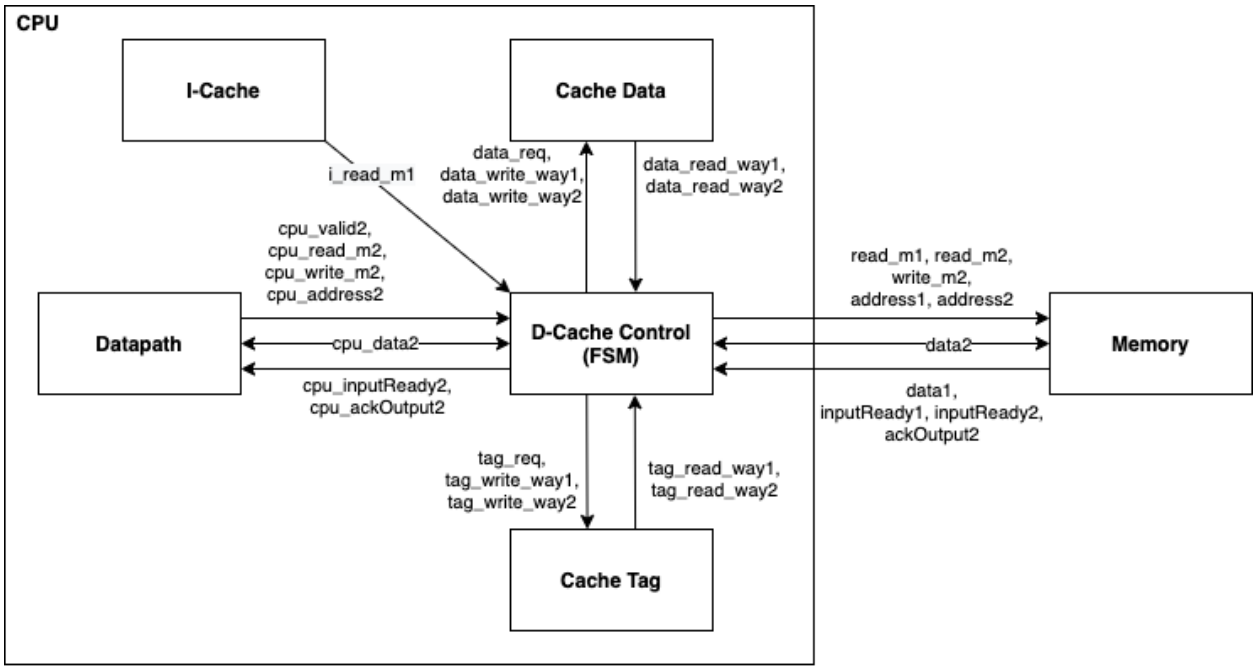
[그림 2] Datapath 모듈과 I-Cache 모듈, Memory 모듈간의 interface를 나타낸 것이다. 중앙의 I-Cache Control은 Datapath에서 요청한 주소를 바탕으로 캐시에 데이터가 있는지 확인하기 위한 tag_req, data_req 등의 요청 내용을 작성해 Cache Data, Cache Tag 모듈에 요청한다. Cache Data, Cache Tag에서의 응답을 통해 캐시에 요청한 데이터가 있는지 확인하고, 없으면 Memory 모듈에 데이터를 요청한다.



[그림 3] I-Cache의 FSM. 캐시 데이터를 확인하여 Cache hit면 데이터를 리턴하는 CHECK 상태와, 메모리로부터 데이터를 기다려서 받은 뒤 캐시를 업데이트하고 데이터를 리턴하는 ALLOCATE 상태로 구성된다.

D-Cache의 경우, 데이터를 메모리에 쓰는 경우도 고려해주어야 한다. 이 때 Cache Miss이고 evict되는 cache line이 dirty인 경우, dirty line을 메모리에 작성한 뒤 원래 읽거나 쓰고자 했던 데이터를 메모리로부터 받아와야 하므로 총 2번의 메모리 요청을 수행해야 한다. 이 때 메모리의 address2와 관련된 포트들만 사용한다면 한 번의 메모리 요청 당 6 cycles를 기다려야 해서 총 12 cycles동안 파이프라인이 stall되게

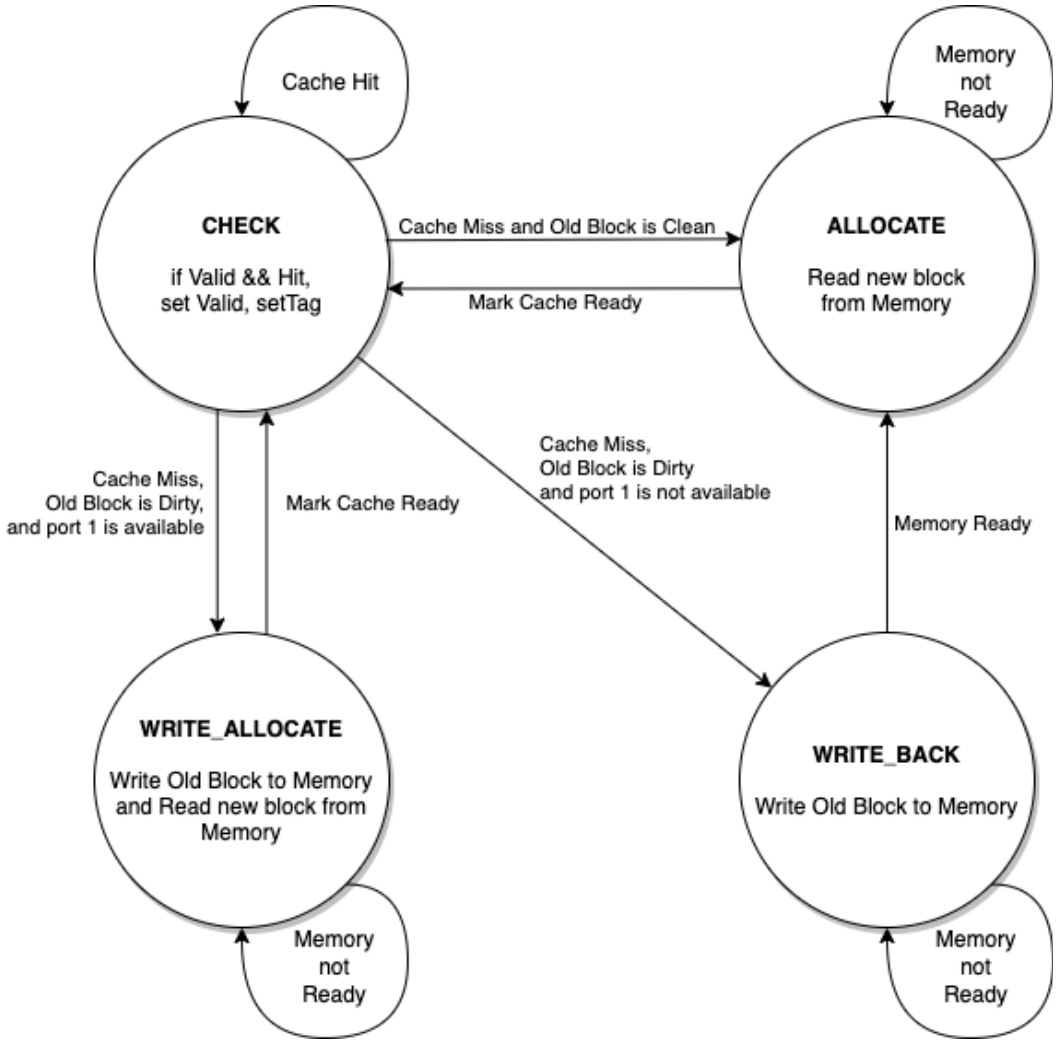
된다. 이를 줄이기 위해 I-cache에서 memory read 요청을 하지 않은 경우엔 메모리의 address1과 관련된 포트에 읽어올 데이터를 요청하고, address2와 관련된 포트에 dirty line의 쓰기 요청을 하면 두 메모리 요청을 병렬적으로 수행할 수 있으므로 6 cycles만 기다려도 된다. 따라서 D-cache는 메모리의 address1, address2와 관련된 포트를 모두 사용할 수 있어야 하고, 메모리에서 데이터를 읽고 쓸 수 있어야 한다.
캐시는 datapath로부터 요청을 받았을 때, 이를 Cache Tag에 저장되어 있는 tag들과 비교한 뒤 Cache hit이 되면 데이터를 바로 리턴하거나 쓴다. 만약 Cache miss가 되면 evict될 cache line이 dirty인지 확인하여 dirty인 경우 해당 dirty line을 메모리에 쓰기 요청을 하고, 이후 필요한 데이터를 메모리에 요청한다. 메모리에서 필요한 요청이 모두 수행된 이후, 캐시 데이터와 태그를 업데이트함과 동시에 Datapath에 필요한 데이터를 보내주거나 데이터 쓰기를 수행한다. D-Cache와 메모리, datapath에서 주고받는 데이터(혹은 포트)는 그림 4에 나타냈다.



[그림 4] Datapath 모듈과 D-Cache 모듈, Memory 모듈간의 interface를 나타낸 것이다. 중앙의 D-Cache Control은 Datapath에서 요청한 주소를 바탕으로 캐시에 데이터가 있는지 확인하기 위한 tag_req, data_req 등의 요청 내용을 작성해 Cache Data, Cache Tag 모듈에 요청한다. Cache Data, Cache Tag에서의 응답을 통해 캐시에 요청한 데이터가 있는지 확인하고, 없으면 Memory 모듈에 데이터를 요청한다. 이 때 Dirty line을 evict해야 하는 경우, I-Cache로부터 메모리에 읽기 요청을 보냈는지(i_read_m1)에 대한 정보를 받아 Memory 모듈의 두 포트를 모두 사용할 것인지 여부를 결정한다.

D-Cache에서 요청한 데이터가 있는지 확인하고 없으면 Memory에 요청하는 일련의 동작을 수행하기 위해 FSM을 설계하였다. FSM은 크게 CHECK, ALLOCATE, WRITE_BACK, WRITE_ALLOCATE의 네 단계로 구성되어 있다. CHECK 상태에는 Datapath로부터 들어온 요청에 대해 캐시에 해당 데이터가 있는지 확인하여 Cache hit이 되면 해당 데이터를 리턴하거나 캐시에 데이터를 쓰고 계속 CHECK 상태로 남는다. 만약 Cache miss가 발생하면 evict할 line을 결정하고, 이 line의 dirty bit를 확인한다. 만약 evict할 line이 dirty하면, I-Cache에서 메모리의 port 1을 사용하는지 확인하고, 만약 port 1을 사용하지 않는다면 메모리

쓰기와 읽기 요청을 병렬적으로 처리하는 WRITE_ALLOCATE 상태로, 그렇지 않다면 메모리 쓰기를 수행하는 WRITE_BACK 상태로 전환하며 알맞은 메모리 요청을 보낸다. 만약 evict할 line이 clean하면 메모리에 데이터 요청을 보내고 ALLOCATE 상태로 전환한다. WRITE_ALLOCATE 상태에서는 Dirty line에 대한 메모리 쓰기와 필요한 데이터에 대한 메모리 읽기 요청을 동시에 수행하며, 이 두 요청이 모두 완료될 때까지 기다렸다가 두 요청이 완료되면 캐시를 업데이트하고 필요 시 적절한 데이터를 리턴한 뒤 CHECK 상태로 전환한다. WRITE_BACK 상태에서는 dirty line에 대한 메모리 쓰기가 끝날 때까지 기다리며, 메모리에서 데이터 쓰기가 완료되었음을 알리면 필요한 데이터에 대한 읽기 요청을 보내고 ALLOCATE 상태로 전환한다. ALLOCATE 상태에서는 메모리에서 데이터를 보낼 때까지 기다리며, 메모리에서 데이터를 보내면 캐시를 업데이트하고 필요 시 적절한 데이터를 리턴한 뒤 CHECK 상태로 전환한다. D-Cache의 FSM은 그림 5에 나타냈다.



[그림 5] D-Cache의 FSM. 캐시 데이터를 확인하여 Cache hit면 데이터를 리턴하거나 캐시를 업데이트하는 CHECK 상태, Dirty line의 메모리 쓰기와 필요한 데이터의 메모리 읽기 동작을 동시에 수행하는 WRITE_ALLOCATE 상태, Dirty line의 메모리 쓰기 동작을 수행하는 WRITE_BACK 상태, 그리고 메모리로부터 데이터를 기다려서 받은 뒤 캐시를 업데이트하고 필요 시 데이터를 리턴하는 ALLOCATE 상태로 구성된다.

캐시를 도입하면서 메모리와 Datapath에서도 수정해야 할 사항들이 생겼다. 먼저 메모리는 이전에 한 cycle만에 요청된 동작을 수행하고, 한 번에 하나의 word만 리턴하거나 쓰기를 수행했다면, 이제는 동작을 수행하기까지 일정한 latency를 가지고 캐시의 line 1개에 해당하는 4 words를 한꺼번에 리턴하거나 쓰기를 수행해야 한다. 이에 따라 메모리 내부적으로 포트 당 counter를 두어 매 clock마다 counter를 증가시키고, 일정 latency만큼 지났을 때 읽기나 쓰기를 수행하도록 수정한다. 또한 data1, data2 포트를 4 words(64 bits) 크기로 수정하고, 읽기의 경우 요청이 들어온 address로부터 차례로 4 words를 읽어 리턴하고 쓰기의 경우 요청이 들어온 address로부터 4 words에 해당하는 메모리를 업데이트하도록 수정한다.

Datapath의 경우 이전에 없던 Memory latency가 생김에 따라, instruction을 메모리에 요청해 받아오는 IF 단계와 데이터를 메모리에 읽거나 쓰기 요청을 하는 MEM 단계에서 한 cycle만에 응답이 오지 않을 경우 파이프라인을 stall해야 한다. IF 단계에서는 instruction이 리턴될 때까지 pc 업데이트를 하지 않고, ID 단계에 bubble을 삽입해야 한다. MEM 단계에서는 메모리에 보낸 요청이 완료될 때까지 IF, ID, EX, MEM 단계를 모두 stall하고, WB 단계에 bubble을 삽입해야 한다.

메인 모듈은 CPU 모듈이며, 서브 모듈은 크게 메모리를 담당하는 Memory 모듈, 캐시에 해당하는 I-Cache, D-Cache, 그리고 dm_cache_tag, dm_cache_data 모듈, 그리고 datapath를 구성하는 datapath 모듈과 그 외의 모듈들로 나눌 수 있다. 이 중 Memory 모듈과 datapath 모듈, 그리고 캐시와 관련된 모듈을 제외한 나머지 모듈은 지난 Lab 5의 구현과 다른 부분이 없어 이들을 제외하고 설계하였다.

2.1. CPU 모듈

CPU 모듈은 input으로 reset signal(reset_n), clock signal(clk), data1을 받고, inout port로 사용하는 data2가 있다. output으로는 read_m1과 read_m2, write_m2, address1, address2, 그리고 num_inst, is_halted, output_port가 있다. read_m1, address1, data1은 메모리에서 instruction을 읽어오는데 사용하고, read_m2, write_m2, address2, data2는 메모리에서 데이터를 읽어오거나 쓸 때 사용한다. 또한 하나의 instruction이 끝날 때마다 num_inst를 증가시켜 출력시켜주고, WWD instruction의 경우 레지스터값을 output_port를 통해 출력해준다. HLT instruction의 경우 is_halted를 출력하여 프로그램이 halt되었음을 알린다.
CPU 모듈에 datapath 모듈과 두 캐시에 해당하는 I_Cache, D_Cache 모듈이 선언되어 있고, Memory 모듈은 CPU 바깥에 선언되어 있다. 이에 따라 두 캐시 모듈과 메모리를 연결하는 wire들은 기존의 CPU 포트와 캐시 모듈의 포트를 바로 연결하고, datapath 모듈과 두 캐시 모듈을 연결하기 위해 새로운 wire들을 선언하고, 이들은 구분을 위해 cpu_ prefix를 사용하기로 했다.

2.2. datapath 모듈

CPU 모듈과 동일한 input, output, inout을 가진다. 캐시와 메모리를 제외한 서브 모듈들에 적절한 input과 output을 연결하여 데이터를 전달하는 것을 담당한다. 메모리에 control signal와 데이터를 전달하고, PC값과 메모리에서 읽어온 값인 instruction과 memory data를 register 변수에 저장하며, register

pipeline에 해당하는 reg 변수들의 업데이트를 담당한다. 또한 wb 단계의 halt, wwd, new_inst 값에 따라 cpu에 적절한 output을 전달한다.

메모리에 요청할 때, instruction의 경우 항상 read-only port(data1)을 이용해 요청하며, data의 경우 항상 read-write port(data2)를 이용해 요청한다. 또한 메모리 요청에 대한 latency에 따라 파이프라인을 적절하게 stall한다.

2.3. Memory 모듈

Memory 모듈은 받은 요청에 따라 메모리에서 읽은 값을 리턴하거나 메모리에 값을 쓴다. input으로 reset signal(reset_n), clock signal(clk)와 read-only 포트에 대한 요청인 read_m1, address1, read와 write 둘 다 가능한 포트에 대한 요청인 read_m2, write_m2, address2를 받는다. output으로 read-only 포트에 대한 응답인 data1, inputReady2가 있고, read-write 포트에 대한 응답인 inputReady2, ackOutput2와 inout port로 사용하는 data2가 있다. 두 포트는 각각 counter를 가지고, 각 요청이 들어오면 counter가 0에서부터 시작하여 증가하고, latency만큼 기다린 이후 적절한 응답(inputReady 혹은 ackOutput 시그널과 data1, data2)을 반환한다.

2.4. dm_cache_tag 모듈

dm_cache_tag 모듈은 input으로 reset signal(reset_n), clock signal(clk)과 함께 요청의 내용 및 요청한 태그 index 정보가 들어있는 tag_req, 그리고 태그에 쓸 데이터에 해당하는 tag_write_way1, tag_write_way2를 받고, output으로 요청받은 태그 데이터인 tag_read_way1, tag_read_way2를 전달한다. dm_cache_tag 모듈은 태그 정보를 저장하는 4개의 reg 변수를 가지고 있으며, 유효한 요청인지에 대한 1비트와 인덱스를 가리키는 1비트, 총 2비트로 구성된 tag_req에 따라 요청받은 index의 2-way tag를 모두 리턴하거나 모두 업데이트한다.

2.5. dm_cache_data 모듈

dm_cache_data 모듈은 input으로 reset signal(reset_n), clock signal(clk)과 함께 요청의 내용 및 요청한 line의 index 정보가 들어있는 data_req, 그리고 각 line에 쓸 데이터에 해당하는 data_write_way1, data_write_way2를 받고, output으로 요청받은 cache line인 data_read_way1, data_read_way2를 전달한다. dm_cache_data 모듈은 태그 정보를 저장하는 4개의 reg 변수를 가지고 있으며, 유효한 요청인지에 대한 1비트와 인덱스를 가리키는 1비트, 총 2비트로 구성된 data_req에 따라 요청받은 index의 2-way data를 모두 리턴하거나 모두 업데이트한다.

2.6. I_Cache 모듈

I_Cache 모듈은 instruction에 대한 캐시로, 기본적인 input으로 reset signal(reset_n), clock signal(clk)을 받으며, datapath에서의 메모리 요청에 해당하는 cpu_read_m1, cpu_address1, cpu_valid1을 input으로 받고 요청에 대한 응답으로 cpu_data1, cpu_inputReady1을 output으로 가진다. Memory에 데이터 요청을 하기 위해 read_m1, address1을 output으로 가지고, 메모리의 응답을

받기 위해 data1, inputReady1을 input으로 가진다. I_Cache 모듈에 dm_cache_tag 모듈과 dm_cache_data 모듈을 인스턴스화하여 tag와 data를 저장할 수 있도록 하고, FSM을 구현하여 datapath로부터의 instruction과 관련된 메모리 요청이 캐시를 거쳐 이루어지도록 한다.

2.7. D_Cache 모듈

D_Cache 모듈은 memory data에 대한 캐시로, 기본적인 input으로 reset signal(reset_n), clock signal(clk)을 받으며, datapath에서의 메모리 요청에 해당하는 cpu_read_m2, cpu_write_m2, cpu_address2, cpu_valid2을 input으로 받고 요청에 대한 응답으로 cpu_data2, cpu_inputReady2, cpu_ackOutput2을 output으로 가진다. Memory에 데이터 요청을 하기 위한 read_m1, read_m2, write_m2, address1을 output으로 가지고, 메모리의 응답을 받기 위해 data1, inputReady1, inputReady2, ackOutput2를 input으로 가지며 data2를 inout port로 가진다. 또한 I_Cache 모듈에서 메모리의 read-only 포트를 사용하는지 확인하기 위한 i_read_m1을 input으로 가진다. D_Cache 모듈에 dm_cache_tag 모듈과 dm_cache_data 모듈을 인스턴스화하여 tag와 data를 저장할 수 있도록 하고, FSM을 구현하여 datapath로부터의 data와 관련된 메모리 요청이 캐시를 거쳐 이루어지도록 한다.

3. Implementation

3.1. Cache 모듈 (dm_cache_data, dm_cache_tag)

```
/* 2-way set associative, 4 lines, data memory*/
module dm_cache_data(clk, reset_n, data_req, data_write_way1, data_write_way2, data_read_way1, data_read_way2);

    input clk;
    input reset_n;
    input [`CACHE_REQ_SIZE-1:0] data_req;      // data request/command, e.g. RW, valid
    input [`CACHE_DATA_SIZE-1:0] data_write_way1;   // write port
    input [`CACHE_DATA_SIZE-1:0] data_write_way2;   // write port
    output [`CACHE_DATA_SIZE-1:0] data_read_way1;   // read port way1
    output [`CACHE_DATA_SIZE-1:0] data_read_way2;   // read port way2

    reg [`CACHE_DATA_SIZE-1:0] data_mem[3:0];

    initial begin
        data_mem[0] = 0;
        data_mem[1] = 0;
        data_mem[2] = 0;
        data_mem[3] = 0;
    end

    assign data_read_way1 = data_mem[data_req[`CACHE_REQ_INDEX] + 0];
    assign data_read_way2 = data_mem[data_req[`CACHE_REQ_INDEX] + 2];

    always @(posedge clk) begin
        if(!reset_n) begin
            data_mem[0] = 0;
            data_mem[1] = 0;
            data_mem[2] = 0;
            data_mem[3] = 0;
        end
        else begin
            if (data_req[`CACHE_REQ_WE]) begin
                data_mem[data_req[`CACHE_REQ_INDEX] + 0] <= data_write_way1;
                data_mem[data_req[`CACHE_REQ_INDEX] + 2] <= data_write_way2;
            end
        end
    end
endmodule
```

[그림 6] Cache 모듈 중 데이터를 저장하는 모듈. data_mem에 데이터를 저장하며, write_way1(way2)를 통해 캐시를 업데이트하고, read_way1(way2)를 이용해 캐시 데이터를 읽어온다. data_req을 통해 RW인지 여부와 요청 index를 전달한다.
Cache_tag 모듈도 Cache_data모듈과 동일하게 구성된다.

Cache 모듈은 Cache FSM의 컨트롤을 통해 실제로 tag와 data를 저장하는 모듈이다. Index bit 0에 해당하는 way1은 mem[0], mem[2]에 저장되어 있고, way2는 mem[1], mem[3]에 저장되어있다. Cache FSM에서 요청할 index와 RW여부를 data_req에 저장하여 넘겨주고, data_req에 따라 clk synchronous하게 Cache tag와 Data를 업데이트해준다.

3.2. I_Cache모듈

1) CHECK state

```
// Check Cache hit? miss?
CHECK: begin

    // no memory request
    mem_req_readl = 0;

    // Initialize UPDATE reg
    UPDATE_WAY1 = 0;
    UPDATE_WAY2 = 0;

    // If no CPU request, maintain at CHECK state
    if(!cpu_validl) begin
        vstate = CHECK;
    end else begin
        // cache hit (tag match and cache entry is valid)
        if(CACHE_HIT) begin
            cpu_res_inputReady1 = 1;

            if((HIT_way1 && VALID_way1)) begin

                // Way1 HIT : Update Recent bit => way1 : 1, way2 = 0
                tag_write_way1[`CACHE_TAG_RECENT] = 1;
                tag_write_way2[`CACHE_TAG_RECENT] = 0;

            end else begin

                // Way2 HIT :Update Recent bit => way1 : 0, way2 = 1
                tag_write_way1[`CACHE_TAG_RECENT] = 0;
                tag_write_way2[`CACHE_TAG_RECENT] = 1;
            end

            // update tag
            tag_req[`CACHE_REQ_WE] = 1;

            // finished
            vstate = CHECK;
        end
```

[그림 7] I_Cache 모듈에서 Cache Hit일 때 동작

I_Cache 모듈은 Instruction만 관리하기 때문에 Write Request를 고려할 필요가 없다. 따라서 I_Cache의 FSM은 CHECK, ALLOCATE state만 존재하고, Cache Hit일 때는 CHECK state에 머물러 있으면서 Cache Miss가 발생했을 때만 ALLOCATE state로 전환한다. Cache Hit일 때에는 cpu_inputReady1을 인가시켜 CPU 쪽에 valid 시그널을 주고, Way1과 Way2 중 tag가 일치하는 블록(사용한 블록)을 확인해서 RECENT bit을 변경해준다. 가장 최근에 사용된 블록의 RECENT bit가 1, 그렇지 않은 쪽을 0으로 설정한다.

```
// cache miss
else begin

    if(!RECENT_way1 && !RECENT_way2) begin
        // if both way is not used, allocate to way 1
        UPDATE_WAY1 = 1;
        UPDATE_WAY2 = 0;
    end

    else if (!RECENT_way1) begin
        // evict way 1
        UPDATE_WAY1 = 1;
        UPDATE_WAY2 = 0;
    end

    else begin
        // evict way 2
        UPDATE_WAY1 = 0;
        UPDATE_WAY2 = 1;
    end

    // generate memory request on miss
    mem_req_read1 = 1;

    // memory request address (sampled from CPU request)
    mem_req_addr1 = {cpu_address1[15:2], 2'b0};

    // wait until new block allocated
    vstate = ALLOCATE;
end
```

[그림 8] I_Cache 모듈에서 Cache Miss일 때 동작

Cache Miss가 났을 때는 LRU 교체 방식에 따라 ALLOCATE state에서 가장 오래전에 사용된 블록을 evict시키기 위해 UPDATE reg를 1로 설정해주고, Memory 쪽에 read 요청을 보내준 후, ALLOCATE state로 전환한다.

2) ALLOCATE state

```
// wait for allocating a new cache line
ALLOCATE: begin

    // memory responded
    if (inputReady1 && mem_req_addr1 == {cpu_address1[15:2], 2'b0}) begin
        // read correct word from cache (way 1)
        case(ADDRESS_BO)
            2'b00: data_way1 = data1[`BLOCK_WORD_1];
            2'b01: data_way1 = data1[`BLOCK_WORD_2];
            2'b10: data_way1 = data1[`BLOCK_WORD_3];
            2'b11: data_way1 = data1[`BLOCK_WORD_4];
        endcase

        // read correct word from cache (way 2)
        case(ADDRESS_BO)
            2'b00: data_way2 = data1[`BLOCK_WORD_1];
            2'b01: data_way2 = data1[`BLOCK_WORD_2];
            2'b10: data_way2 = data1[`BLOCK_WORD_3];
            2'b11: data_way2 = data1[`BLOCK_WORD_4];
        endcase
```

[그림 9] ALLOCATE state, 메모리에서 Data를 받아오면
Cache Update와 별개로 데이터를 바로 cpu쪽에 넘겨준다.

```
        else if (UPDATE_WAY1) begin // update way1
            data_write_way1 = data1;
            cpu_res_data1 = data_way1;

            tag_write_way1[`CACHE_TAG_RECENT] = 1;
            tag_write_way1[`CACHE_TAG_VALID] = 1;
            tag_write_way1[`CACHE_TAG] = ADDRESS_TAG;

            tag_write_way2[`CACHE_TAG_RECENT] = 0;

            cpu_res_inputReady1 = 1;
        end

        else begin // update way 2
            data_write_way2 = data1;
            cpu_res_data1 = data_way2;

            tag_write_way1[`CACHE_TAG_RECENT] = 0;

            tag_write_way2[`CACHE_TAG_RECENT] = 1;
            tag_write_way2[`CACHE_TAG_VALID] = 1;
            tag_write_way2[`CACHE_TAG] = ADDRESS_TAG;

            cpu_res_inputReady1 = 1;
        end

        // update cache line data
        data_req[`CACHE_REQ_WE] = 1;

        // update tag
        tag_req[`CACHE_REQ_WE] = 1;

        // re-compare tag for write miss
        vstate = CHECK;
    end
```

[그림 10] ALLOCATE state, Memory에서 받아온 data를 Cache Module에 업데이트해준다.

ALLOCATE state에서는 CHECK state에서 LRU 교체 방식에 따라 결정한 UPDATE reg에 따라서 알맞은 way에 받아온 data와 address의 tag를 업데이트해준다. 업데이트하는 tag에는 VALID bit, RECENT bit이 포함되어 있다. 업데이트되지 않는 way에는 RECENT bit을 0으로 만들어 업데이트 해준다. 캐시 업데이트가 종료되면, 다시 CHECK state로 돌아간다.

3.3. D_Cache모듈

1) CHECK state

```
// cache hit and write -> write data to cache with dirty bit and update recent bit
else if (cpu_write_m2) begin
    cpu_res_ackOutput2 = 1;

    if((HIT_way1 && VALID_way1)) begin
        // Way1 HIT: Write on way 1, set dirty bit
        UPDATE_WAY1 = 1;
        UPDATE_WAY2 = 0;

        // write correct word using data2
        case(ADDRESS_BO)
            2'b00: write_data = {cpu_data2, data_read_way1[`BLOCK_WORD_1_C]};
            2'b01: write_data = {data_read_way1[`BLOCK_WORD_1], cpu_data2, data_read_way1[`BLOCK_WORD_2_C]};
            2'b10: write_data = {data_read_way1[`BLOCK_WORD_3_C], cpu_data2, data_read_way1[`BLOCK_WORD_4]};
            2'b11: write_data = {data_read_way1[`BLOCK_WORD_4_C], cpu_data2};
        endcase

        data_write_way1 = write_data;

        tag_write_way1[`CACHE_TAG_DIRTY] = 1;

        // Update Recent bit => way1 : 1, way2 = 0
        tag_write_way1[`CACHE_TAG_RECENT] = 1;
        tag_write_way2[`CACHE_TAG_RECENT] = 0;

    end else begin
        // Way2 HIT: Write on way 2, set dirty bit
        UPDATE_WAY1 = 0;
        UPDATE_WAY2 = 1;

        // write correct word using data2
        case(ADDRESS_BO)
            2'b00: write_data = {cpu_data2, data_read_way2[`BLOCK_WORD_1_C]};
            2'b01: write_data = {data_read_way2[`BLOCK_WORD_1], cpu_data2, data_read_way2[`BLOCK_WORD_2_C]};
            2'b10: write_data = {data_read_way2[`BLOCK_WORD_3_C], cpu_data2, data_read_way2[`BLOCK_WORD_4]};
            2'b11: write_data = {data_read_way2[`BLOCK_WORD_4_C], cpu_data2};
        endcase

        data_write_way2 = write_data;

        tag_write_way2[`CACHE_TAG_DIRTY] = 1;

        // Update Recent bit => way1 : 0, way2 = 1
        tag_write_way1[`CACHE_TAG_RECENT] = 0;
        tag_write_way2[`CACHE_TAG_RECENT] = 1;
```

[그림 11] D_Cache 모듈에서 Cache Hit && Write Request일 때 동작

D_Cache 모듈은 I_Cache와 달리 write request가 있다. Cache Hit일 때, Read일 때의 동작은 I_Cache와 동일하다. write일 때는 cpu_res_ackOutput2를 인가시켜 CPU 쪽에 valid 시그널을 주고, offset 값에 따라서 적절한 위치에 data를 업데이트 해준 후, Dirty Bit을 1로 만들어 이후 Cache Miss일 때 Write-back 해줄 블록을 표시해준다. 다른 동작은 I_Cache와 동일하다.

```
else if (!RECENT_way1) begin
    // evict way 1
    UPDATE_WAY1 = 1;
    UPDATE_WAY2 = 0;

    // if evict line is dirty, write back it to memory
    if (VALID_way1 && DIRTY_way1) begin
        // generate memory write request for dirty line
        mem_req_write2 = 1;
        // memory request address (sampled from cache tag)
        mem_req_addr2 = {tag_read_way1[`CACHE_TAG], ADDRESS_IDX, 2'b00};
        mem_req_data2 = data_read_way1;
        if (!i_read_m1) begin
            // port 1 is not using. use port1 to read data
            mem_req_addr1 = {cpu_address2[15:2], 2'b0};
            mem_req_read1 = 1;

            vstate = WRITE_ALLOCATE;
        end else begin
            // wait until write back done
            vstate = WRITE_BACK;
        end
    end

    else begin
        // generate memory request on miss
        mem_req_read2 = 1;
        // memory request address (sampled from CPU request)
        mem_req_addr2 = {cpu_address2[15:2], 2'b0};
        // wait until new block allocated
        vstate = ALLOCATE;
    end
end
```

[그림 12] D_Cache 모듈에서 Cache Miss일 때 동작

Cache Miss가 났을 때는 LRU 교체 방식에 따라 ALLOCATE state에서 가장 오래전에 사용된 블록을 evict시킨다. 이 때 Dirty Data는 메모리에 저장하고, 새로운 데이터를 Cache에 저장해야 하는데, 이 때 data1 port를 사용하기 위해 Write와 Read를 동시에 수행하는 WRITE_ALLOCATE 상태로 전환한다. Dirty Data가 없을 때는 ALLOCATE state로 전환하여 I_Cache에서와 동일한 동작을 수행한다.

2) WRITE_ALLOCATE state

```
// wait for write back and allocating a new cache line
WRITE_ALLOCATE: begin
    // memory responded
    if (ackOutput2 && inputReady1) begin
        if (cpu_write_m2) begin
            case(ADDRESS_BO)
                2'b00: write_data = {cpu_data2, data1[`BLOCK_WORD_1_C]};
                2'b01: write_data = {data1[`BLOCK_WORD_1], cpu_data2, data1[`BLOCK_WORD_2_C]};
                2'b10: write_data = {data1[`BLOCK_WORD_3_C], cpu_data2, data1[`BLOCK_WORD_4]};
                2'b11: write_data = {data1[`BLOCK_WORD_4_C], cpu_data2};
            endcase
        end else begin
            write_data = data1;
        end
```

[그림 13] D_Cache 모듈에서 WRITE_ALLOCATE state일 때 동작

WRITE_ALLOCATE 는 data1 port를 사용하여 Read를 하고, data2 port를 이용해 Write를 동시에 수행한다. Memory로부터 불러온 data1을 Cache에 업데이트하며, 업데이트가 끝나면 CHECK state로 전환한다. 나머지 동작은 ALLOCATE state와 동일하게 수행한다.

3.4. Memory 모듈

```
if(read_m1) begin
        if (count1 == 0 && requested_address1 == address1 && inputReady1 == 1) begin
                // data already given but address is not changed. do nothing
        end else if (count1 < `MEM_STALL_COUNT - 1) begin
                // increase count
                count1 <= count1 + 1;
                inputReady1 <= 0;
                if (count1 == 0) begin
                        requested_address1 <= address1;
                end
        end else begin
                // count is full. return data and reset count
                count1 <= 0;
                inputReady1 <= 1;
                data1[`BLOCK_WORD_1] <= memory[requested_address1+`WORD_SIZE'b00];
                data1[`BLOCK_WORD_2] <= memory[requested_address1+`WORD_SIZE'b01];
                data1[`BLOCK_WORD_3] <= memory[requested_address1+`WORD_SIZE'b10];
                data1[`BLOCK_WORD_4] <= memory[requested_address1+`WORD_SIZE'b11];
        end
end
```

[그림 14] Memory 모듈은 카운터 숫자가 설정한 값에 이르면 data를 fetching해준다.

Memory 모듈은 한번의 Memory Access에 4 word를 6 cycle만에 return 한다. 이를 위해 Memory에 counter를 사용하여 설정해 둔 6 count가 지나면, 읽기의 경우 inputReady 신호와 data를 넘겨주고 쓰기의 경우 ackOutput 신호를 넘겨주고 메모리에 data를 쓰도록 구현하였다.

3.5. Datapath 모듈

```
always @(*) begin
        if (!reset_n) begin
                instr_stall = 0;
                mem_data_stall = 0;
        end else begin
                if (read_m1 && !inputReady1) begin
                        instr_stall = 1;
                end else begin
                        instr_stall = 0;
                end

                if ((read_m2 && !inputReady2) || (write_m2 && !ackOutput2)) begin
                        mem_data_stall = 1;
                end else begin
                        mem_data_stall = 0;
                end
        end
end
```

[그림 15] 캐시에 데이터를 요청하고, valid 시그널(inputReady, ackOutput)이 들어올 때까지 Stall해서 pipeline이 진행되지 않도록 한다.

```
// update pc
if(!flush && (stall || instr_stall || mem_data_stall)) begin
        pc <= pc;
end else begin
        pc <= pc_nxt;
end

// update IF/ID pipeline register (instr from data)
if(!flush && !stall && !instr_stall && !mem_data_stall) begin
        pc_id <= pc;
        new_inst_id <= 1'b1;
end else if(!flush && (stall || instr_stall || mem_data_stall)) begin // stall
        pc_id <= pc_id;
        new_inst_id <= new_inst_id;
end     else begin // flush
        pc_id <= ~0;
        new_inst_id <= 0;
end

// update ID/EX pipeline register
if(!flush & !stall & !instr_stall & !mem_data_stall) begin
        target <= instr[11:0]; pc_ex <= pc_id; rf_rs_ex <= rf_rs; rf_rt_ex <= rf_rt; immed_ex <= immed_id;
        rs_ex <= instr[11:10]; rt_ex <= instr[9:8]; rd_ex <= rd_id;
end else if (mem_data_stall) begin
        target <= target; pc_ex <= pc_ex; rf_rs_ex <= rf_rs_ex; rf_rt_ex <= rf_rt_ex; immed_ex <= immed_ex;
        rs_ex <= rs_ex; rt_ex <= rt_ex; rd_ex <= rd_ex;
end else begin
        pc_ex <= ~0;
        target <= 0; rf_rs_ex <= 0; rf_rt_ex <= 0; immed_ex <= 0;
        rs_ex <= 0; rt_ex <= 0; rd_ex <= 0;
end
```

[그림 16] instr_stall, mem_data_stall값에 따라 파이프라인의 진행을 적절하게 stall한다.

datapath 모듈의 경우 다른 서브 모듈과의 연결이나 pipeline register의 업데이트는 Lab 5에서의 구현과 거의 동일하다. 다른 부분은 Memory latency가 추가되면서 바뀐 부분들로, 먼저 캐시에 데이터를 요청한 후

값이 들어오는지를 파악해 instr_stall, mem_data_stall값을 지정한다. 이후 instr_stall, mem_data_stall값에 따라 파이프라인을 stall하거나 기존대로 진행한다.

3.6. CPU 모듈

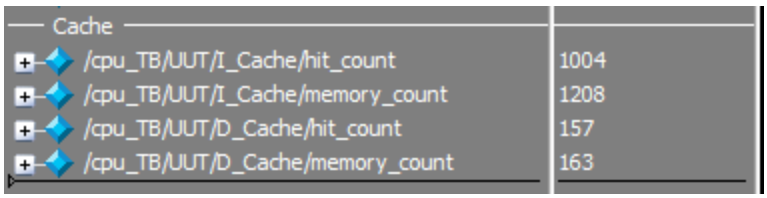
```
wire i_read_m1, d_read_m1;
wire [`WORD_SIZE-1:0] i_address1, d_address1;
wire cpu_valid1;
wire cpu_valid2;

assign read_m1 = i_read_m1 | d_read_m1;
assign address1 = i_read_m1? i_address1: d_address1;
assign cpu_valid1 = cpu_read_m1;
assign cpu_valid2 = cpu_read_m2 | cpu_write_m2;

  datapath Datapath(
          .clk(clk),
          .reset_n(reset_n),
          .read_m1(cpu_read_m1),
          .address1(cpu_address1),
          .data1(cpu_data1),
          .inputReady1(cpu_inputReady1),
          .read_m2(cpu_read_m2),
          .write_m2(cpu_write_m2),
          .address2(cpu_address2),
          .data2(cpu_data2),
          .inputReady2(cpu_inputReady2),
          .ackOutput2(cpu_ackOutput2),
          .num_inst(num_inst),
          .output_port(output_port),
          .is_halted(is_halted)
  );

  instr_cache I_Cache(
          .clk(clk),
          .reset_n(reset_n),
          .cpu_read_m1(cpu_read_m1),
          .cpu_address1(cpu_address1),
          .cpu_data1(cpu_data1),
          .cpu_inputReady1(cpu_inputReady1),

          .read_m1(i_read_m1),
          .address1(i_address1),
          .data1(data1),
          .inputReady1(inputReady1),
          .cpu_valid1(cpu_valid1)
  );
```

[그림 17] CPU 모듈의 코드 중 일부. Datapath와 I_Cache는 cpu_ prefix가 붙은 wire들로 연결되어 있으며, I_Cache 모듈의 cpu_ prefix가 붙지 않은 포트는 대부분 CPU 모듈의 포트들과 바로 연결되어 있다. read_m1과 address_m1은 I_Cache와 D_Cache에서 모두 사용하기 때문에 충돌을 막기 위해 별도의 wire를 통해 값을 지정한다.

CPU 모듈은 datapath 모듈과 I_Cache, D_Cache 모듈이 선언되어 있다. cpu_ prefix가 붙은 wire들은 datapath와 캐시 모듈들을 연결하는 wire이고, 그 외의 경우 CPU의 port를 이용하여 Memory와 캐시 모듈들이 연결된다.

4. Discussion

- **Cache Hit Ratio**



I_Cache의 경우 총 1208개의 memory access 중 1004개는 Cache hit이다. 따라서 I_Cache의 Hit ratio는 1004/1208 ≈ 0.83으로 약 83%의 Hit ratio를 가진다. D_Cache의 경우 총 163개의 memory access 중 157개는 Cache hit이다. 따라서 D_Cache의 Hit ratio는 157/163 ≈ 0.96으로 약 96%의 Hit ratio를 가진다. 전체 memory access에 대한 Hit ratio를 계산하면, 전체 1371개의 memory access 중 1161개는 Cache Hit이므로, 전체 Hit ratio는 1161/1371 ≈ 0.85으로 약 85%의 Hit ratio를 가진다.
I_Cache보다 D_Cache의 Hit ratio가 높은 이유는 테스트벤치에서 대부분의 memory data access를 하는 fibonacci를 계산하는 부분에서, memory data를 상당히 규칙적으로 접근하는 반면, instruction의 경우 JAL, JPR이 포함되어 있어 data에 비해 불규칙적으로 접근하기 때문이라고 추측했다.

- **Cache의 유무에 따른 성능 비교**

Without cache, 2 cycles of memory latency

```
# Clock # 2911
# The testbench is finished. Summarizing...
# All Pass!
```

With cache, 6 cycles of memory latency

```
# Clock # 2270
# The testbench is finished. Summarizing...
# All Pass!
```

Cache가 없고 memory latency가 2 cycles인 경우 2911 Clock이 걸리고, Cache가 있고 memory latency가 6 cycles인 경우 2270 Clock이 걸렸다. Memory latency가 3배 더 긴 것에도 불구하고 Cache가 있는 CPU가 더 빠르다. 이는 Cache에서 데이터 지역성을 잘 활용하여 Memory latency를 잘 감추었기 때문이다. 이를 통해 Cache가 Memory와 CPU 사이의 속도 차이를 잘 메우고, 많은 양의 데이터를 빠르게 접근할 수 있다는 Memory illusion을 효과적으로 제공한다는 것을 확인할 수 있었다.

5. Conclusion

Pipeline CPU에 2-way set associative, LRU, write-back, write-allocate cache를 적용함으로써 높은 데이터 지역성의 활용을 통해 성능을 향상하였다. memory latency가 3배 차이남에도 불구하고

Cache를 사용한 경우의 성능이 더 좋았고, 이로 인해 Cache가 성능에 미치는 영향이 크다는 것을 확인하였다.