

Lab6. Cache

Introduction

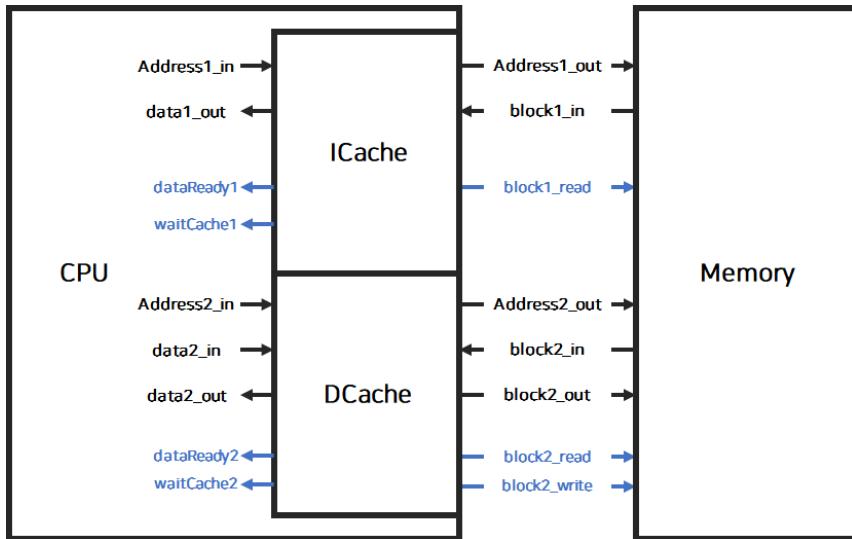
이번 과제의 목표는 기존에 구현했던 Pipelined CPU에 set associative cache를 구현하는 것이다. Cache는 Locality를 통해 CPU와 Memory 사이의 Latency를 줄이기 위해 이용하기 임시로 data를 저장해두는 저장공간을 말한다.

cache는 주소값을 tag와 index로 저장하고 주소에 해당하는 값을 block으로 저장하여 cpu가 원하는 값을 제공할 수 있다. 하지만 cache의 용량은 memory보다 작기 때문에 cache에 원하는 데이터가 없는 miss가 발생한다. miss가 발생한 경우에는 memory로부터 데이터를 가져와야 하며 cache에 있던 데이터 중 하나를 evict해야 한다. 이 때 FIFO, LRU, Random 등의 policy를 사용하여 evict line을 결정한다. cache와 memory는 같은 값을 가져야 하는데 cpu가 data의 값을 변경하는 경우는 다른 값을 가지게 된다. 이를 막기 위해서 올바른 write policy를 이용해야 한다. 이번 Lab에서 어떤 Write Policy를 사용하였는지는 아래의 Design에서 설명하도록 하겠다.

이전까지의 Lab에서는 Magic Memory를 사용하여, 1 clock만에 data를 가져올 수 있게 하였지만, 실제로는 그렇게 일어날 수 없다. 따라서 이번 Lab에서는 Memory 접근에 2 cycles을 사용하고, Cache가 없는 Baseline CPU와, Hit일때 1cycle이지만 miss일때는 6 cycles을 사용하여 data를 가져오는 Cache와 이를 포함하는 Cache CPU를 구현하여, Cache의 동작 원리를 이해하고, Cache를 사용하였을 때 CPU의 성능 향상을 수치를 통해 확인하는 것이 목표이다.

Design

1. Structure



위 구조도는 이번 Lab의 전체 CPU와 Memory의 구조를 그린 것이다.

(기존 CPU구현에서 사용했던 Signal들에 대한 설명은 생략하겠다.)

Address_in / out

1은 ICache로 보내는 instruction을 읽기 위한 Address

2는 DCache로 보내는 data를 읽고 쓰기 위한 Address

data_in : Cache/Memory에 쓰기 위해 CPU로부터 받은 Data

data_out : CPU에서 사용하기 위해 Cache에서 주는 Data

block_in : Memory로부터 Cache가 받은 Block line

block_out : Memory에 쓰기 위해 Memory에 주는 Block line

*Signal

block_read : Memory의 값을 읽을 것이라는 signal이다. (이전까지의 lab에서 readm과 유사)

block_write : Memory의 값에 쓸 것이라는 signal이다. (이전까지의 lab에서 writem과 유사)

waitCache : miss가 일어날 경우 6 cycle이 사용되기 때문에 그동안 멈춰야 할 CPU의 기능이 돌아가지 않도록 wait시키는 Signal이다.

dataReady : CPU에서 요청하느 data가 Cache에서 준비가 되었을때 보내는 Signal이다.

a. Cache Associativity

Cache는 Associativity에 따라 Direct-mapped, n-way associative, fully associative로 나눌 수 있으며, 어떤 방식을 선택하는지에 따라 Tradeoff가 존재한다.

Direct-mapped는 associativity가 1이라고 생각할 수 있다. 즉 index가 가르키는 공간이 direct하게 하나로 결정되는 경우이며, 이는 처리가 빠르지만, miss가 자주 발생한다. Fully associative의 경우는 최대 associativity를 가진 경우라 생각하면 된다. 즉, index가 가르키는 공간이 모든 공간일 경우이며, 이는 miss가 적지만, 모든 block을 탐색해야 하므로 느리다.

이번 Lab에서 구현할 Cache는 2-way set associative Cache이며, 즉 index가 가르키는 공간이 2개인 경우를 말한다.

b. Unified or Separate Cache

Cache를 구현하는 방식은 크게 두가지로 구분할 수 있다.

첫째는, ICache와 DCache가 Unified된, 32 words의 8Lines으로 구성된 하나의 Cache로 구현하는 방법이고, 둘째는 ICache와 DCache를 Separate하여 각각이 16words씩 4lines으로 구성된 두개의 Cache로 구현하는 방법이다. 이 중, 우리는 후자인 Separate된 방식을 선택하였다.

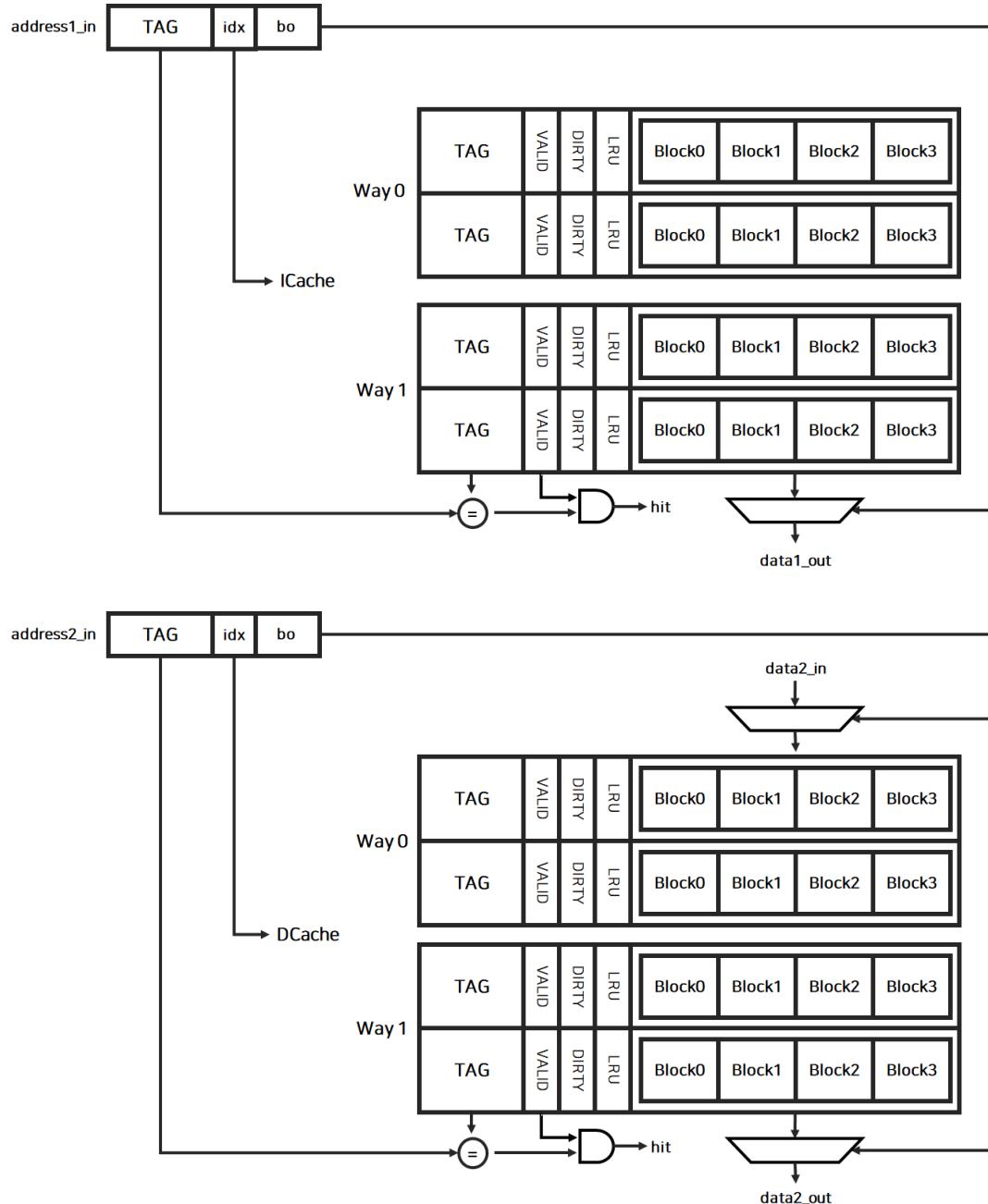
c. Replacement Policy

Set Associative Cache에서 같은 index에서 어떤 line을 Evict할 때 어떤 기준으로 이를 결정할 것인지에 대한 Policy를 뜻한다. 일반적으로 Cache의 성질을 더 극대화하기 위해 LRU를 사용하기 때문에, 이번 Lab에서 LRU(Least Recently Use)를 사용하여 구현할 것이다. 이는 가장 자주 사용하지 않은 line을 선택하는 방식이다.

d. Write Policy

Cache에 Write하는 정책은 Write-Through와 Write-Back으로 두가지가 있다. Write-Through의 경우, 항상 Cache의 값은 Memory에도 존재한다. 즉, CPU에서 Memory에 Write를 할 때에, 항상 Memory에 이를 반영하는 Policy이다. 반면, Write-Back의 경우, Memory에 Write를 하면 이를 일단 Cache에만 Write를 해 두었다가 이후, Memory에 저장되지 않은 line 즉, Dirty한 Data를 Evict해야 하는 경우, Write를 하는 방식을 의미한다. 이번 Lab에서는 후자인 Write-Back 방식을 사용하였다.

2. Cache



위의 두 구조도는 이번 Lab에서 작동하는 ICache와 DCache의 작동형태를 간단하게 나타낸 것이다. Cache는 앞서 말했듯, ICache와 DCache로 구분되어있으며 각각의 input인 address_in에 따라 분리되어 작동한다. 전체 Cache는 32 words로 각 ICache와 DCache가 16 words를 사용한다. 이때 각 line이 4 words이므로 각 Cache는 4 lines씩 갖게 된다. 또한 2-way set associative cache이므로 각 way별로 2 lines을 가진다. 이를 그림으로 표현하면 위와 같이 표현할 수 있다. way에 따라 index는 0 또는 1이므로 idx는 1bit가 필요하고, block은 총 4개가 있으므로 block offset은 2bit가 필요하다. 따라서 총 16bits의 address는 Tag(13bit), Idx(1 bit), bo(2 bit)로 나뉘어 사용된다. 또한 Cache의 한 Line은 Tag(13), Valid(1), Dirty(1), LRU(1), Blocks(4*16)로 총 80bits가 사용된다.

Implementation - Baseline CPU

1. Memory

baseline cpu가 memory에게 data를 요청하면 1clock 이후에 data를 사용가능하게 만드는 방식으로 구현했다.

```
always@(posedge clk)
  if(!reset_n)
    begin...
    end
  else
    begin
      if (count == 1) begin isReady <= 1; count <= 0; end
      else begin isReady <= 0; count <= 1; end

      if(read_m1) data1 <= (write_m2 & address1==address2)?data2:memory[address1];
      if(read_m2) output_data2 <= memory[address2];
      if(write_m2) memory[address2] <= data2;
    end
  
```

count와 isReady flag를 이용하여 구현했다. count는 1clock을 기다리는 역할을 하며 clock이 지났을 때 isReady가 ON되면서 cpu에서는 data를 사용할 수 있다.

2. CPU

cpu는 isReady가 ON 되었을 때 pipeline을 진행하고 그렇지 않은 경우는 stall하여 요청한 data를 기다린다.

```
  if (isReady) begin
    num_clock <= num_clock + 1;
    num_inst <= (num_clock >= 5 && !MEM_is_bubble) ? num_inst + 1 : num_inst;

    //MEM
    MEM_data <= data2;
    MEM_alu_out <= EX_alu_out;
    MEM_dest <= EX_dest;

    MEM_mem_to_reg <= EX_mem_to_reg;
    MEM_reg_write <= EX_reg_write;
    MEM_wrd <= EX_wrd;
    MEM_hlt <= EX_hlt;
    MEM_is_bubble <= EX_is_bubble;

    // EX
    EX_B <= ForwardBMuxOut;
    EX_alu_out <= alu_result;
    EX_dest <= ID_dest;
    EX_mem_read <= ID_mem_read;
    EX_mem_write <= ID_mem_write;
    EX_mem_to_reg <= ID_mem_to_reg;
    EX_reg_write <= ID_reg_write;
    EX_wrd <= ID_wrd;
    EX_hlt <= ID_hlt;
    EX_is_bubble <= ID_is_bubble;
  
```

그 외의 module은 pipeline과 동일하기 때문에 설명을 생략한다.

Implementation - Cached CPU

1. Memory

cached cpu에서는 cache miss가 발생하면 memory에게 data request를 한다. memory에서는 요청을 받으면 address가 포함된 4개의 block을 cache에게 전달한다.

```
always@(posedge clk)
  if(!reset_n)
    begin...
  end
  else begin
    if(read_m1) begin
      data1 <= (write_m2 & address1==address2)? data2: { memory[address1+3], memory[address1+2], memory[address1+1], memory[address1] };
    end

    if(read_m2) begin
      dataReady <= 1;
      output_data2 <= { memory[address2+3], memory[address2+2], memory[address2+1], memory[address2] };
    end else dataReady <= 0;

    if(write_m2) begin
      getData2 <= data2;
      writeReady <= 1;
    end else if (writeReady) begin
      writeReady <= 0;
      memory[address2] <= getData2[BLOCK0]; memory[address2+1] <= getData2[BLOCK1]; memory[address2+2] <= getData2[BLOCK2]; memory[address2+3] <= getData2[BLOCK3];
    end
  end
end
```

2. Cache

a. ICache

```
always @(*) begin
  if (read_m1) begin
    hit[0] = i_cache[0][idx][`VALID] && (tag == i_cache[0][idx][`TAG]);
    hit[1] = i_cache[1][idx][`VALID] && (tag == i_cache[1][idx][`TAG]);
    if      (hit[0]) way = 0;          //cache way0 hit
    else if (hit[1]) way = 1;          //cache way1 hit
    else begin                         //cache miss
      if (!i_cache[0][idx][`VALID]) way = 0;
      else if (!i_cache[1][idx][`VALID]) way = 1;
      else if (i_cache[0][idx][`LRU]) way = 0;
      else                           way = 1;
    end
  end
end
```

cpu로부터 cache에 read 요청이 들어오면 address로부터 tag, idx를 추출하여 hit 여부를 판단한다.

tag가 일치하면 hit, 일치하지 않으면 miss이다. 판별 후에는 way 값을 설정해준다. miss에서는 사용하지 않는 (not valid) line이 있으면 해당 way를 선택하고 그렇지 않으면 LRU bit를 사용하여 판별한다.

```

if (hit[0]) begin           //cache way0 hit
    waitCache <= 0;
    i_cache[0][idx]['LRU'] <= 0;
    i_cache[1][idx]['LRU'] <= 1;

    if (read_m1) begin
        dataReady <= 1;
        case (bo)
            0: data1_out <= i_cache[0][idx]['BLOCK0'];
            1: data1_out <= i_cache[0][idx]['BLOCK1'];
            2: data1_out <= i_cache[0][idx]['BLOCK2'];
            3: data1_out <= i_cache[0][idx]['BLOCK3'];
        endcase
    end
end else if (hit[1]) begin //cache way1 hit
    waitCache <= 0;
    i_cache[1][idx]['LRU'] <= 0;
    i_cache[0][idx]['LRU'] <= 1;

    if (read_m1) begin
        dataReady <= 1;
        case (bo)
            0: data1_out <= i_cache[1][idx]['BLOCK0'];
            1: data1_out <= i_cache[1][idx]['BLOCK1'];
            2: data1_out <= i_cache[1][idx]['BLOCK2'];
            3: data1_out <= i_cache[1][idx]['BLOCK3'];
        endcase
    end

```

hit인 경우 1clock 만에 data를 cpu에 전달해야 한다. 따라서 clk에 맞춰 data1_out에 요청한 data를 넣고 dataReady를 1로 만들어주어 cpu가 곧바로 사용할 수 있게 만들어 준다. 접근한 line은 LRU bit를 0으로 만들고 원래 0이었던 line을 1로 바꾸어 LRU 규칙을 구현했다.

```

if (read_m1) begin
    dataReady <= 0;
    // signal to cpu
    waitCache <= 1;
    counter_read <= 5;
    address1_out <= { i_cache[way][idx]['TAG'], idx, 2'b00 };
end else begin
    if (counter_read == 4) begin block_read <= 1; address1_out <= { tag, idx, 2'b00 }; end
    else begin block_read <= 0; end
    if (counter_read == 1) begin
        dataReady <= 1;
        if(way == 0) begin
            i_cache[0][idx] <= {tag, 1'b1, 1'b0, 1'b0, block1_in };
            i_cache[1][idx]['LRU'] <= 1'b1;
        end else begin
            i_cache[1][idx] <= {tag, 1'b1, 1'b0, 1'b0, block1_in };
            i_cache[0][idx]['LRU'] <= 1'b1;
        end
        case (bo)
            0: data1_out <= block1_in['BLOCK0];
            1: data1_out <= block1_in['BLOCK1];
            2: data1_out <= block1_in['BLOCK2];
            3: data1_out <= block1_in['BLOCK3];
        endcase
    end else begin dataReady <= 0; end
    if (counter_read == 0) waitCache <= 0;
    if (counter_read > 0) begin
        counter_read <= counter_read - 1;
    end
end

```

miss인 경우 6clock을 소모하게 된다. 따라서 dataReady를 0으로, waitCache는 1로 설정하여 cpu가 cache를 기다릴 수 있게 만들어준다.

counter_read는 나머지 5clock을 기다릴 수 있도록 만들어준다. 5clock 동안 cache는 memory에 data를

요청하고 받아와서 icache에 저장한다. 또 data1_out에 요청한 주소의 값을 넣어 cpu에게 전달한 후 counter가 0이되면 waitCache를 0으로 만들어주어 cpu가 다시 작동할 수 있게 만들어준다.

b. DCache

instruction을 수정하지 않아 write과정이 없는 ICache와는 달리 Dcache는 write가 일어난다. ICache와 같은 부분은 생략하고 추가된 부분만 설명하도록 하겠다.

```
assign block_read = block_read_r | block_read_w;
assign block_write = block_write_r | block_write_w;
assign dataReady = dataReady_r | dataReady_w;
assign waitCache = waitCache_r | waitCache_w;
```

먼저 모든 flag를 read/write로 나누고 or operation으로 assign하여 사용한다.

```
if (write_m2) begin
    dataReady_w <= 1;
    waitCache_w <= 0;
    case (bo)
        0: d_cache[0][idx][`BLOCK0] <= data2_in;
        1: d_cache[0][idx][`BLOCK1] <= data2_in;
        2: d_cache[0][idx][`BLOCK2] <= data2_in;
        3: d_cache[0][idx][`BLOCK3] <= data2_in;
    endcase
    d_cache[0][idx][`DIRTY] <= 1;
end
```

write_m2가 ON되었으며 hit인 상황에서는 cpu에서 전달된 data값인 data2_in을 해당 block에 넣어서 cache에 write하게 만들어준다. 수정되었으면 dirty bit를 1로 만들어준다.

```
if (write_m2) begin
    dataReady_w <= 0;
    counter_write <= 5;
    waitCache_w <= 1;
    if(d_cache[way][idx][`DIRTY] == 1) block_write_w <= 1;
    address2_out <= { d_cache[way][idx][`TAG], idx, 2'b00 };
    .
    .
    .

assign block2_out = d_cache[way][idx][`BLOCK_SIZE-1:0];
```

miss인 상황에서는 eviction이 일어나는데 기존의 block이 dirty인 경우는 memory에 write back해야 한다. 따라서 dirty bit를 검사해 block_write_w를 1로 할당하여 memory에 data가 write될 수 있게 한다. address2_out에 주소를 block2_out에는 block들의 값을 넣어준다.

```

else begin
block_write_w <= 0;
if (counter_write == 4) begin block_read_w <= 1; address2_out <= { tag, idx, 2'b00 }; end
else begin block_read_w <= 0; end
if (counter_write == 1) begin
dataReady_w <= 1;
if (way == 0) begin
if (bo == 3) d_cache[0][idx] <= { tag, 1'b1, 1'b1, 1'b0, data2_in, block2_in["BLOCK2"], block2_in["BLOCK1"], block2_in["BLOCK0"] };
if (bo == 2) d_cache[0][idx] <= { tag, 1'b1, 1'b1, 1'b0, block2_in["BLOCK3"], data2_in, block2_in["BLOCK1"], block2_in["BLOCK0"] };
if (bo == 1) d_cache[0][idx] <= { tag, 1'b1, 1'b1, 1'b0, block2_in["BLOCK3"], block2_in["BLOCK2"], data2_in, block2_in["BLOCK0"] };
if (bo == 0) d_cache[0][idx] <= { tag, 1'b1, 1'b1, 1'b0, block2_in["BLOCK3"], block2_in["BLOCK2"], block2_in["BLOCK1"], data2_in };

d_cache[1][idx]["LRU"] <= 1'b1;
end
else if (way == 1) begin
if (bo == 3) d_cache[1][idx] <= { tag, 1'b1, 1'b1, 1'b0, data2_in, block2_in["BLOCK2"], block2_in["BLOCK1"], block2_in["BLOCK0"] };
if (bo == 2) d_cache[1][idx] <= { tag, 1'b1, 1'b1, 1'b0, block2_in["BLOCK3"], data2_in, block2_in["BLOCK1"], block2_in["BLOCK0"] };
if (bo == 1) d_cache[1][idx] <= { tag, 1'b1, 1'b1, 1'b0, block2_in["BLOCK3"], block2_in["BLOCK2"], data2_in, block2_in["BLOCK0"] };
if (bo == 0) d_cache[1][idx] <= { tag, 1'b1, 1'b1, 1'b0, block2_in["BLOCK3"], block2_in["BLOCK2"], block2_in["BLOCK1"], data2_in };

d_cache[0][idx]["LRU"] <= 1'b1;
end
end else begin dataReady_w <= 0; end
if (counter_write == 0) begin waitCache_w <= 0; end
if (counter_write > 0) begin
counter_write <= counter_write - 1;
end

```

eviction한 이후에는 실제로 miss를 발생시킨 주소의 값을 읽어야한다. 따라서 counter가 4일 때 address2_out과 block_read_w값을 설정하여 memory에서 원하는 값을 가져온다.

가져온 값을 d_cache에 기록하고 LRU bit를 설정한다.

clock이 모두 지나간 이후에는 waitCache를 0으로 설정하여 cpu가 기다림을 멈출 수 있게 만들어준다.

3. CPU

```

module CacheCPU(clk, reset_n, block1_read, address1_out, block1_in, block2_read, block2_write, address2_out, block2 inout, num_inst, output_port, is_halted);
input clk;
input reset_n;

output block1_read;
output [WORD_SIZE-1:0] address1_out;
input [BLOCK_SIZE-1:0] block1_in;

output block2_read;
output block2_write;
output [WORD_SIZE-1:0] address2_out;
input [BLOCK_SIZE-1:0] block2 inout;

output reg [WORD_SIZE-1:0] num_inst;
output reg [WORD_SIZE-1:0] output_port;
output reg is_halted;

// cache
wire read_m1, read_m2, write_m2;
wire waitCache1, waitCache2, dataReady2, isReady;
wire [WORD_SIZE-1:0] address1_in, data1_out, address2_in, data2_in, data2_out;
wire [BLOCK_SIZE-1:0] block2_in, block2_out;

```

```

ICache i_cache(!clk, read_m1, address1_in, address1_out, data1_out, block1_in, block1_read, dataReady1, waitCache1);
DCache d_cache(!clk, read_m2, write_m2, address2_in, address2_out, data2_in, data2_out, block2_in, block2_out, block2_read, block2_write, dataReady2, waitCache2);

assign read_m1 = waitCache1 ? 0 : 1;
assign read_m2 = waitCache2 ? 0 : (EX_is_bubble ? 0 : EX_mem_read);
assign write_m2 = waitCache2 ? 0 : (EX_is_bubble ? 0 : EX_mem_write);
assign address1_in = PC;
assign address2_in = EX_alu_out;
assign data2_in = EX_B;
assign isReady = waitCache2 ? dataReady2 : 1;

assign block2 inout = block2_write ? block2_out : `BLOCK_SIZE'bz;
assign block2_in = block2_read? block2 inout : block2_in;

```

cache 모듈 중 cache-memory 상호작용에 필요한 port들은 cpu port로 곧바로 연결하여 TB module-memory로 전달하여 cache와 memory는 직접적으로 interaction하게 구현했다.

read_m1, read_m2, write_m2는 cpu가 read/write를 시도했을 때 켜지는 플래그이다. cache에서 miss가 일어나서 waitCache flag가 켜졌을 땐 더 이상 read/write를 시도하지 않는다.

```

num_clock <= num_clock + 1;
num_inst <= (!MEM_is_bubble) ? num_inst + 1 : num_inst;

if (isReady) begin
    //MEM
    MEM_data <= data2_out;
    MEM_alu_out <= EX_alu_out;
    MEM_dest <= EX_dest;

    MEM_mem_to_reg <= EX_mem_to_reg;
    MEM_reg_write <= EX_reg_write;
    MEM_wwd <= EX_wwd;
    MEM_hlt <= EX_hlt;
    MEM_is_bubble <= EX_is_bubble;

    // EX
    EX_B <= ForwardBMuxOut;
    EX_alu_out <= alu_result;
    EX_dest <= ID_dest;
    EX_mem_read <= ID_mem_read;
    EX_mem_write <= ID_mem_write;
    EX_mem_to_reg <= ID_mem_to_reg;
    EX_reg_write <= ID_reg_write;
    EX_wwd <= ID_wwd;
    EX_hlt <= ID_hlt;
    EX_is_bubble <= ID_is_bubble;
end else begin MEM_is_bubble <= 1; end

```

```
forwarding_unit ForwardingUnit(!waitCache2, rs, rt, ID_read1, ID_read2, ID_dest, EX_dest, MEM_dest, ID_reg_write,
```

`isReady`는 cpu가 Dcache를 기다리고 있을 때 켜지는 플래그이다. Dcache에서 값이 전달되기 전까지 MEM 이하의 pipeline은 중단되어야 한다. 따라서 pipeline register가 업데이트 되지 않도록 만들어주어 stall을 구현했다. 추가로 forwarding unit에서도 `waitCache`인 상황에서는 멈추도록 구현했다.

```

if (!is_stall && dataReady1) begin
    PC <= PC_NXT;
    IF_PC <= PC + 1;
    if (!IF_flush) begin
        IF_inst <= data1_out;
        IF_is_bubble <= 0;
    end
    else begin
        IF_inst <= 0;
        IF_is_bubble <= 1;
    end
end else begin
    IF_is_bubble <= 1;
end

```

IF에서도 `dataReady1`이 켜지기 전까지는 PC값을 update하지 않음으로써 stall을 구현했다.

다른 module들은 pipelined cpu와 구현이 동일하므로 설명을 생략한다.

Discussion

stage를 stall할 때 pipeline register의 업데이트만 막으면 된다고 생각했다. 하지만 forwarding unit이 계속 작동하고 있어 잘못된 값이 전달되어 alu 작동에 오류가 생겼다. 따라서 cache를 기다리는 중에는 forwarding을 멈추도록 구현했다.

Conclusion

baselined cpu의 작동 모습이다.

```
# Clock # 2425
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : //wsl$/Ubuntu/home/seungyeon/CSED311_computer-architecture/Pipeline/lab5/baseline_cpu_TB.v(155)
#      Time: 242650 ns Iteration: 2 Instance: /cpu_TB
```

cached cpu의 작동 모습이다.

```
# Clock # 2089
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : //wsl$/Ubuntu/home/seungyeon/CSED311_computer-architecture/Pipeline/lab5/cache_cpu_TB.v(154)
#      Time: 209050 ns Iteration: 2 Instance: /cache_cpu_TB
```



```
# IF 0017 0028
# ID 0018 00000027
# ID 0019 00000027
# ID 001a 00000027
# ID 001b 00000027
# ID 001c 00000027
# ID 001d 00000027
# IF 001d 0029
# ID 001e 00000028
```

IF 단계에서 memory에서 instruction을 읽어오는 6 clock을 기다리는 모습을 확인할 수 있다.

아래는 ICache와 DCache에서 DataReady가 되는 것을 기준으로 hit과 miss의 개수를 센 결과다.

# Icache : hit	1090 miss	143
# Dcache : hit	240 miss	6

각 cache에 대한 Hit ratio는 다음과 같다.

$$\text{ICache hit ratio} = 1090 / (1090+143) = 0.88$$

$$\text{DCache hit ratio} = 240 / (240+6) = 0.97$$

cache가 있는 CPU가 non cache cpu보다 $2425/2089=1.1610$ 이므로 약 16% 더 빠르다. cache cpu가 baselined cpu에 비해 memory 접근시 지체되는 clock 수가 더 길다는 것을 감안하면 cached cpu가 clock 수가 눈에 띄게 줄어든 것을 볼 수 있다. 우리는 cache를 직접 구현함으로써 cache가 효율적인 cpu 작동에 긍정적인 영향을 준다는 것을 증명했다.