컴퓨터 구조 Lab6

20190782 컴퓨터공학과 최서영
20190439 컴퓨터공학과 오승훈

Introduction

이번 Lab은 기존의 만들어 놓았던 Pipelined cpu에서 cache를 구현하여 이를 이용하여 cpu가 작동하도록 구현하는 것이다. 기존에는 magic memory를 이용하여 memory 데이터를 가져오는데 걸리는 delay가 한 사이클 내였기 때문에 cache가 없어도 잘 작동하였지만 실제로는 magic memory처럼 작동하지 않는다. 따라서 실제 memory 접근을 구현하기 위해 한번 데이터를 접근하는 데에 2cycle씩 걸리는 cpu를 구현해본다. 그리고, cache의 유용성을 보기 위해 cache를 이용하는 cpu를 구현하여 두 cpu를 비교해보고자 한다.

Design

이번 lab에서는 pipelined cpu에서 사용하였던 adder, alu, alu_control_unit, control_unit, cpu, datapath, register_file, util, sign_extend, forwarding_unit, branch_predictor 모듈과 더불어 cache를 구현한 I_cache와 D_cache가 추가되었고, Memory 모듈이 수정되어 사용되고 있다.

control_unit의 경우 clock의 posedge에서 input으로 들어온 opcode와 func_code(instruction의 정보)를 이용하여 해당 instruction에 알맞은 signal들을 결정하고, 해당 signal을 output으로 내보내 준다. 이때, 만약 flush나 혹은 stall등의 이유로 input으로 들어오는 instruction이 z값으로 들어온다면 read_m1(instruction을 읽어오는 signal)을 제외하곤 모든 signal을 0으로 초기화 시켜준다.

MUX2_to_1와 MUX4_to_1은 기존의 mux의 기능이 같다. 이러한 mux들은 memory를 읽을 때 어떤 address를 사용할지, WB state에서 register에 update할 data는 무엇일지, ALU에 들어갈 input을 결정할 때, pc값을 update할 때 어떤 값으로 update할지 등을 결정할 때 사용된다. 이러한 mux에 서 값을 고르는 signal은 control_unit, forwarding unit 등 다른 모듈에서 오게 된다.

Alu_control_unit은 alu에 넣어주는 funcCode, opcode, branchType을 결정해주는 module이다. 이때, input으로 받는 opcode와 funct(각 instruction의 정보)에 맞는 funcCode를 set하고, opcode를 이용하여 branch인 instruction을 걸러내어 branchTyp과 funcCode를 set해준다.

Register_file은 register 값들을 불러오고 수정할 수 있도록 구현한 모듈이다. 값을 불러오는 경우 rs, rt에 해당하는 port에 들어온 번호에 맞게 register 내부의 값을 output으로 보내주고 값을 수정하는 경우 rd의 값에 대응하는 register에 값을 대입한다. 이때, register 값을 수정하는 것은

오직 negative edge clock일 때, reg_write signal이 1일때만 수정이 가능하다.

Sign_extend는 immediate 값의 bit수를 맞춰주기 위한 모듈로 imm 값을 input으로 받아 Imm_extend를 output으로 내놓고 이 값을 ALU의 input을 결정해주는 MUX로 들어간다.

Adder는 말 그대로 input으로 들어간 두개의 값을 더해주는 역할을 한다.

ALU는 기존의 ALU 연산과 더불어 bcond를 판단하는 기능을 수행한다. ALU는 combinational logic이고, non-blocking으로 계산된다. input으로 들어온 data 2와 func_code, branch_type을 이용하여 input에 알맞은 output을 결정한다.

Hazard는 해당 instruction이 stall이 발생하는지를 확인하여 output으로 stall이 발생하면 1, 아니면 0을 내보 내준다. ID_EX에서 memory를 접근하는 load 혹은 store 일 때, IF_ID 단계에서 사용하는 register와 겹치는지를 확인하여 겹친다면 stall이 필요하다.

Forwarding_unit에서는 ID_EX에서 사용하려고 하는 register가 뒤의 단계(EX_MEM or MEM_WB)에서 WB을 하려고 하는 register와 같은지를 판별하여 forwarding이 필요한지를 판단하고, 필요하다면 어떤 단계의 data를 사용해야 하는지를 판단하는 모듈이다.

Datapath은 control_unit으로부터 signal을 받아 위에서 소개한 모듈들을 연결하고, 각 stage의 data를 저장하여 각 stage에 필요한 데이터를 전달해주는 역할을 한다. 이때, 데이터는 signal과 instruction, register 값 등이 있다. 또한 stall이나 flush가 발생해야 하는지를 확인하고, 해야 한다면 그에 맞는 data와 signal을 setting하여 다음 stage로 보내는 역할을 한다.

Branch_predictor는 다음 instruction을 받아올 때, 어떤 주소의 instruction을 받아야 하는지를 결정하는 모듈이다. 이번 랩에서는 2-bit global prediction을 사용하였는데, 이 모듈에서는 현재 PC가 BTB_table에서 hit일 때, jump면 table에 저장된 값을, branch라면 hypothesis counter를 이용하여 taken인지 not taken인지를 확인해서 taken인 경우만 table에 저장된 값을 반환한다.

I_cache와 D_cache는 baseline pipelined cpu에서는 쓰이지 않지만 cache를 구현하기 위해 쓰이는 모듈이다. 두 cache 다 single-level cache이며 2-way set associative cache로 구현이 되었다. 각각 4개의 cache line을 가지고 있다. 두 모듈 다 cpu를 통해 읽고자 하는 memory의 address가 input으로 들어오면 해당 address를 이용하여 어떤 set의 몇 번째 index인지를 결정해 해당 cache의 tag값과 valid 값을 비교해서 hit인지 miss인지를 결정한다. 만약 hit이라면 해당 cache line에 존재하는 데이터를 output으로 내보내거나 어떤 data를 적는다. 그후, LRU 값과 dirty bit을 세팅해준다. miss라면 읽고자 하는 address를 포함한 4개의 data를 memory에 요청하여 읽어온다. 읽어온 후, 마찬가지로 해당 data들의 address를 이용하여 cache line에 데이터를 저장하고, valid과 LRU, dirty bit, tag를 알맞게 세팅한다. 이때, 새로운 데이터가 적히기 전 없어지는 데이터의 dirty bit이 1이라면 memory에 없어지는 데이터를 해당 address에 적는다. 이를 위해 memory 모듈에서 내보내는 data의 형식이 기존과 다르다.

만약, baseline pipelined cpu라면 위의 두 모듈과 수정된 memory 모듈을 사용하지 않는다. 대신, memory를 접근하고자 할 때 모두 2cycle씩 걸리게 된다. 이는 cpu에서 counter를 만들어 제어하고 있다.

Implementation

A. Baseline Pipelined cpu

Baseline cpu의 경우 memory를 읽어 오는 instruction이 존재하는 경우 모두 2cycle이 걸리는 특징을 가지고 있기 때문에 다음과 같이 2_bit register를 만들어 read_m1 제어를 해주면 된다.

```
assign read_m1 = read_control_m1 && (pc_value_counter == 1);
```

이렇게 하여 instruction을 2cycle 당 한 번씩 읽어올 수 있게 제어를 해주고, pc_value_counter와 pc_value를 밑의 그림과 같이 제어를 해주면 된다.

```
always @(posedge clk) begin
        if(!reset_n) begin
                pc_value <= pc_value;
                pc_value_counter <= 0;
        end
        else begin
                if((id_ex_jalr && !(jalr_check))) begin
                        if(next_PC == pc_value_alu_out) begin
                                pc_value <= pc_value_alu_out;
                                pc_value_counter <= !pc_value_counter;
                        end
                        else begin
                                pc_value <= pc_value_alu_out;
                                pc_value_counter <= 0;
                        end
                end
                else if((!(b_j_check) && is_BJ_type_update)) begin
                        if(next_PC == if_id_pc_next) begin
                                pc_value <= if_id_pc_next;
                                pc_value_counter <= !pc_value_counter;
                        end
                        else begin
                                pc_value <= if_id_pc_next;
                                pc_value_counter <= 3;
                        end
                end
                else if(mem_counter) begin
                        pc_value <= next_PC;
                        pc_value_counter <= pc_value_counter;
                end
                else begin
                        pc_value <= next_PC;
                        if(pc_value_counter == 0 ) begin
                                pc_value_counter <= 1;
                        end
                        else begin
                                pc_value_counter <= 0;
                        end
                end
        end
end
```

위와 같을 때 jalr과 branch에서 pc_value_counter 값의 조절에 있어서 차이점이 조금 있는데 jalr의 경우 다음 instruction의 pc 값이 EX stage에서 나와진다는 특성상

pc_value_counter 값이 1일 때 pc_value가 나오게 되므로 그러한 점을 고려하여 flush와 pc_value 그리고 pc_value_counter 값을 조절해주면 된다. 그리고 jump와 branch 일 때는 pc_value 값이 0일 때 다음 instruction의 pc_value가 나오게 되므로 예측한 값과 실제 다음 값이 같은 경우 즉 next_PC와 if_id_pc_next가 같은 경우에는 현재 counter 값이 바뀌면서 바로 instruction을 읽어오면 되지만, 아닌 경우에는 pc_value_counter를 3으로 이동시킨 후 다시 한 번 더 pc_value_counter가 0과 1로 돌게 할 수 있도록 한다. 이러한 부분은 이미 요청된 instruction이 요청된 pc_value 값에 대해서는 instruction을 2cycle만에 읽어와서 Memory 접근을 완료해야 한다는 Teams 답변에 기반을 한 부분이고, 또한 맞는 pc_value로 instruction을 Memory에서 요청한 경우에는 flush를 하지 않아도 된다는 점을 구현하기 위해서 다음과 같이 구현을 하였다.

또한 Memory 접근에 한해서는

```
if(id_ex_read_m2 || id_ex_write_m2) begin
        mem_counter <= 1;
end
else begin
        mem_counter <= 0;
end
```

위와 같이 ex stage에 read_m2나 write_m2 즉 memory에 load 또는 store를 진행하는 instruction에 한해서는 mem_counter를 1로 변경을 해주고, 나머지 instruction을 정상적으로 받는다.

```
ex_mem_alu_value <= id_ex_alu_output;
ex_mem_reg2 <= id_ex_alu_mux2_out;
ex_mem_instruction <= id_ex_instruction;
ex_mem_rd_index <= id_ex_rd_index;
ex_mem_write_m2 <= id_ex_write_m2;
ex_mem_read_m2 <= id_ex_read_m2;
ex_mem_reg_write <= id_ex_reg_write;
ex_mem_halt <= id_ex_halt;
ex_mem_wwd <= id_ex_wwd;
ex_mem_S <= id_ex_S;
ex_mem_pc_reg <= id_ex_pc_reg;
ex_mem_pc_to_reg <= id_ex_pc_to_reg;
ex_mem_jalr <= id_ex_jalr;
```

그 후 mem_counter 값이 1인 경우에 한해서는 pipeline을 stall을 시키기 위해서 현재 instruction이 계속 위치할 수 있도록 한다. ex_mem register와 mem_wb register의 제어만을 예시로 보여드리면

```
if(mem_counter == 1) begin
        ex_mem_alu_value <= ex_mem_alu_value;
        ex_mem_reg2 <= ex_mem_reg2;
        ex_mem_instruction <= ex_mem_instruction;
        ex_mem_rd_index <= ex_mem_rd_index;
        ex_mem_write_m2 <= ex_mem_write_m2;
        ex_mem_read_m2 <= ex_mem_read_m2;
        ex_mem_reg_write <= ex_mem_reg_write;
        ex_mem_halt <= ex_mem_halt;
        ex_mem_wwd <= ex_mem_wwd;
        ex_mem_S <= ex_mem_S;
        ex_mem_pc_reg <= ex_mem_pc_reg;
        ex_mem_pc_to_reg <= ex_mem_pc_to_reg;
        ex_mem_jalr <= ex_mem_jalr;
        ex_mem_F <= ex_mem_F;
        mem_counter <= 0;
end
```

```
if(mem_counter == 1) begin
        mem_wb_memory_value <= 16'bz;
        mem_wb_alu_value <= 16'bz;
        mem_wb_instruction <= 16'bz;
        mem_wb_rd_index <= 0;
        mem_wb_write_m2 <= 0;
        mem_wb_read_m2 <= 0;
        mem_wb_reg_write <= 0;
        mem_wb_halt <= 0;
        mem_wb_wwd <= 0;
        mem_wb_S <= 0;
        mem_wb_F <= 0;
        mem_wb_pc_reg <= 16'bz;
        mem_wb_pc_to_reg <= 16'bz;
end
```

Wb stage 쪽에는 들어올 instruction이 없기 때문에 16'bz로 값을 넣어준다. 위와 같이 mem_counter가 1인 경우에 한해서 모든 datapath에 한해서 stall을 시켜주도록 한다. 그리고 mem_counter가 stall이 된 경우에 한해서 pc_value 값도 제어를 해주기 위해서 다음과 같이 pc_value 제어를 해주면 된다.

```
else if(mem_counter) begin
        pc_value <= next_PC;
        pc_value_counter <= pc_value_counter;
end
```

현재 여기서 next_PC 값은 다음 PC 값이지만

```
assign is_flush = (id_ex_jalr && !(jalr_check)) || (!(b_j_check) && is_BJ_type_update) || (is_stall === 1)
       || (pc_value_counter == 0) || (pc_value_counter == 3) || mem_counter;
```

위와 같이 branch prediction module에 들어가는 is_flush 값을 1로 해줌으로써 next_PC 값이 pc_value(branch prediction module 에서는 PC)로 그대로 유지될 수 있도록 해주면 된다.

```
if(is_flush) begin
        next_PC = PC;
end
```

B. Cache Pipelined cpu

two-way cache 를 구현해야하고, I cache와 D cache를 나눠서 구현을 하였다. I, D cache 모두 LRU 기반 evict 방식을 가지고 있고, D cache에서는 write_back을 기반으로 구현을 하였다. I cache부터 먼저 설명을 하자면

```
reg [63:0]cache_line[1:0][1:0];
reg [12:0]cache_tag[1:0][1:0];
reg cache_LRU[1:0][1:0];
reg cache_valid[1:0][1:0];
reg way;
reg [2:0]counter;
reg counter_signal;
```

위와 같이 cache_line을 저장하는 register cache_tag를 저장해두는 register, cache_LRU 값을 저장하는 register, valid register 등 모든 값을 다 저장을 해둘 수 있도록 하고, I cache

의 경우 write를 하는 경우가 없기 때문에 cache line을 저장을 해두고, evict 하는 경우 write_back을 구현할 필요가 없다는 특징이 있다. 그리고 2 line 이고, 한 line당 4word 이므로 16 bit 이라면 13bit / 1bit / 2bit == tag / line / offset 과 같은 address 구성을 나타낸다. 그러므로 address[2]를 기반으로 line에 접근을 하고, two way 중 tag 값이 같은 것이 있는지를 확인하여 구현을 하면 된다.

```
if(read_m && reset_n && (counter == 0)) begin          //read_m check
        if(cache_valid[memory_address[2]][0] || cache_valid[memory_address[2]][1]) begin
                //$display ("%d", memory_address);
                if(cache_tag[memory_address[2]][0] == memory_address[15:3]) begin
                        hit = 1;
                        miss = 0;
                        way = 0;
                        cache_LRU[memory_address[2]][0] = 0;
                        cache_LRU[memory_address[2]][1] = 1;
                        mem_read_m = 0;
                end
                else if (cache_tag[memory_address[2]][1] == memory_address[15:3] ) begin
                        hit = 1;
                        miss = 0;
                        way = 1;
                        cache_LRU[memory_address[2]][0] = 1;
                        cache_LRU[memory_address[2]][1] = 0;
                        mem_read_m = 0;
                end
                else begin                                              //cache miss
                        hit = 0;
                        miss = 1;
                        mem_read_m = 1;
                        way = cache_LRU[memory_address[2]][0] ? 0 : 1;
                        out_address[15:2] = memory_address[15:2];
                        out_address[1:0] = 2'b00;
                        counter_signal = 1;
                        counter = 0;
                end

        end
        else begin      //cold miss
                hit = 0;
                miss = 1;
                mem_read_m = 1;
                way = 0;
                cache_LRU[memory_address[2]][0] = 0;
                out_address[15:2] = memory_address[15:2];
                out_address[1:0] = 2'b00;
                cache_tag[memory_address[2]][0] = memory_address[15:3];
                counter_signal = 1;
                counter = 0;
        end
end
```

위와 같이 valid bit이 있는 경우 없는 경우를 나눠서 구현을 하는데 valid bit이 1인 경우 tag 값이 같은 경우를 확인을 하고, tag 값이 같은 것이 존재한다면 hit으로 나타내고, 그 때의 값을 hit을 1을 함으로써 값을 output 시켜준다. Miss인 경우 counter를 작동을 시키기 위해서 counter_signal을 1로 해주고, mem_read_m을 1로 한다. 그래서 counter가 작동을 하면 다음과 같이 동작을 한다.

```
if(counter_signal) begin
        if(counter == 0 || counter == 6) begin
                counter <= 1;
        end
        else if(counter == 1) begin
                counter <= 2;
        end
        else if(counter == 2) begin
                counter <= 3;
        end
        else if(counter == 3) begin
                counter <= 4;
        end
        else if(counter == 4) begin
                miss_count = miss_count + 1;
                if(cache_valid[memory_address[2]][way] == 1) begin
                        evict_count <= evict_count + 1;
                end
                else begin
                        evict_count <= evict_count;
                end
                cache_line[memory_address[2]][way]<= instruction;
                cache_valid[memory_address[2]][way] <= 1;
                cache_tag[memory_address[2]][way] <= memory_address[15:3];
                cache_LRU[memory_address[2]][way] <= 0;
                cache_LRU[memory_address[2]][!way] <= 1;
                counter <= 5;
                if(memory_address[15:2] === out_address[15:2]) begin
                        hit <= 1;
                        mem_read_m <= 0;
                end
                else begin
                        hit <= 0;
                        mem_read_m <= 1;
                end
        end
        else if(counter == 5) begin
                if(memory_address[15:2] === out_address[15:2]) begin
                        counter <= 0;
                        counter_signal <= 0;
                end
                else begin
                        counter <= 6;
                        counter_signal <= 1;
                        out_address[15:2] <= memory_address[15:2];
                        out_address[1:0] <= 2'b00;
                end
```

Counter 값은 posedge 마다 1씩 증가하도록 구현을 하였고, 총 6cycle을 돌아야지 counter 값이 종료가 된다. 이때 4~5cycle을 돌고 있는 중 cache line에 값이 적히게 되고, 필요한 bit과 LRU 값을 조정을 해주게 된다. 그리고 counter가 4->5로 갈 때 memory_address[15:2]와 out_address[15:2]의 값이 다르다면 이것은 flush가 발생한 것이기 때문에 flush가 발생하면 한 번 더 6cycle을 돌아서 다른 address에 존재하는 cache line을 가져와야하기 때문에 mem_read_m은 1로, hit은 0으로 하여 값은 나가지 않도록 하며, cycle이 5->6으로 갈 때 counter 값을 6으로 해줌으로써 다시 한 번 더 cycle이 돌 수 있도록 해준다. 그리고 6cycle이 더 돌았을 때 최종적으로 cache_line을 읽어오고 instruction을 내보내 줄 수 있도록 동작을 구성하였다. 그리고 memory_address[15:2]와 out_address[15:2]가 같은 경우에는 flush가 동작을 하지 않았고, 현재 있는 cache_line이 맞게 동작을 수행한 것이기 때문에 hit을 1로 해주고, mem_read_m을 0으로 하여 memory read를 끝내고,

```
mux4_1 cache_word (memory_address[1:0], cache_line[memory_address[2]][way][15 : 0], cache_line[memory_address[2]][way][31 : 16],
        cache_line[memory_address[2]][way][47 : 32], cache_line[memory_address[2]][way][63 : 48], cache_word_mux);

assign data = hit ? cache_word_mux : 16'bz;
```

위와 같이 offset의 값을 거쳐서 나온 최종 cache_line의 값을 hit이 1이 됨으로써 cycle이 5~6일 때 내보내주고, 6cycle이 지나면 동작이 끝나도록 구현을 하였다.

D_cache의 경우 동작 방식은 동일하고, 구성은 동일하지만 flush가 나지 않는다는 점과 D_cache 에서는 evict 될 때 memory write를 해야하는 점이 생긴다. 그러므로 memory write를 하는 경우와 아닌 경우를 나눠서 진행을 해주면 된다. Read 즉 load instruction과

write (store instruction) 각각을 보게 되면 다음과 같이 진행이 된다. Read 인 경우에는 I cache와 hit에 대해서는 동일하게 동작을 수행하고,

```
end
else begin                                              //cache miss
        hit = 0;
        miss = 1;
        mem_read_m = 1;
        miss_count = miss_count + 1;
        way = cache_LRU[memory_address[2]][0] ? 0 : 1;
        if(cache_valid[memory_address[2]][way] == 0) begin
                mem_write_m = 0;
        end
        else begin
                if(dirty_bit[memory_address[2]][way] == 1) begin
                        mem_write_m = 1;
                end
                else begin
                        mem_write_m = 0;
                end
                evict_count = evict_count + 1;
        end
        cache_valid[memory_address[2]][way] = 1;
        cache_tag[memory_address[2]][way] = memory_address[15:3];
        cache_LRU[memory_address[2]][way] = 0;
        cache_LRU[memory_address[2]][!way] = 1;
        counter_signal = 1;
end
```

Miss인 경우에 한해서 다르게 동작을 수행하는데 위와 같이 cache의 dirty bit이 1인 경우에는 mem_write_m을 1로 하여 cache의 값이 memory에 write_back이 될 수 있도록 구현을 하고, 0인 경우에는 mem_write_m을 0으로 하여 cache의 값이 memory에 적히지 않도록 동작을 수행한다. 이 경우 위의 miss인 상태와 동일하게 counter가 동작을 수행하는데

```
if(counter_signal && (dirty_bit[memory_address[2]][way] == 0)) begin
        if(counter == 0) begin
                cache_line[memory_address[2]][way] <= inout_data;
                evict_pc[memory_address[2]][way] <= {memory_address[15:2], 2'b00};
                counter <= 1;
                mem_read_m <= 0;
        end
        else if(counter == 1) begin
                counter <= 2;
        end
        else if(counter == 2) begin
                counter <= 3;
        end
        else if(counter == 3) begin
                counter <= 4;
        end
        else if(counter == 4) begin
                counter <= 5;
                if(read_m == 1) begin
                        hit <= 1;
                end
                miss <= 0;
        end
        else if(counter == 5) begin
                if(write_m == 1) begin
                        if(memory_address[1:0] == 0) begin
                                cache_line[memory_address[2]][way][15 : 0] <= memory_store_data;
                        end
                        else if(memory_address[1:0] == 1) begin
                                cache_line[memory_address[2]][way][31 : 16] <= memory_store_data;
                        end
                        else if(memory_address[1:0] == 2) begin
                                cache_line[memory_address[2]][way][47 : 32] <= memory_store_data;
                        end
                        else if(memory_address[1:0] == 3) begin
                                cache_line[memory_address[2]][way][63 : 48] <= memory_store_data;
                        end
                        dirty_bit[memory_address[2]][way] <= 1;
                end
                else if (read_m == 1) begin
                        dirty_bit[memory_address[2]][way] <= 0;
                end
                counter <= 0;
                counter_signal <= 0;
        end
end
```

다음과 같이 dirty bit이 0인 경우 cache_line을 memory부터 읽어오고, 그 후 6cycle이 돌

고 난 후 hit을 1로 해줌으로써 원하는 data가 내보내질 수 있도록 한다. 그리고 write를 하는 경우 바로 dirty_bit이 1이 되기 때문에 write_m이 1인 경우와 read_m이 1인 경우 즉 store와 load에 대해서 나눠서 counter 값마다 진행이 되는 것을 확인할 수 있다.

```
else if(counter_signal && (dirty_bit[memory_address[2]][way] == 1)) begin
        if(counter == 0) begin
                counter <= 1;
                mem_write_m <= 0;
                mem_read_m <= 1;
        end
        else if(counter == 1) begin
                cache_line[memory_address[2]][way] <= inout_data;
                evict_pc[memory_address[2]][way] <= {memory_address[15:2], 2'b00};
                mem_read_m <= 0;
                counter <= 2;
        end
        else if(counter == 2) begin
                counter <= 3;
        end
        else if(counter == 3) begin
                counter <= 4;
        end
        else if(counter == 4) begin
                counter <= 5;
                if(read_m == 1) begin
                        hit <= 1;
                end
                miss <= 0;
        end
        else if(counter == 5) begin
                if(write_m == 1) begin
                        if(memory_address[1:0] == 0) begin
                                cache_line[memory_address[2]][way][15 : 0] <= memory_store_data;
                        end
                        else if(memory_address[1:0] == 1) begin
                                cache_line[memory_address[2]][way][31 : 16] <= memory_store_data;
                        end
                        else if(memory_address[1:0] == 2) begin
                                cache_line[memory_address[2]][way][47 : 32] <= memory_store_data;
                        end
                        else if(memory_address[1:0] == 3) begin
                                cache_line[memory_address[2]][way][63 : 48] <= memory_store_data;
                        end
                        dirty_bit[memory_address[2]][way] <= 1;
                end
                else if (read_m == 1) begin
                        dirty_bit[memory_address[2]][way] <= 0;
                end
                counter <= 0;
                counter_signal <= 0;
        end
```

이것은 dirty_bit이 1인 경우 evict가 되기 위해서 mem_write_m을 1로 했다가 counter가 0->1로 갈 때 memory에 store가 끝나므로 mem_write_m을 종료를 하고, 그 후 cache_line을 읽어와서 cache_line을 각각에 맞게 넣어주는 것을 확인할 수 있다. 그 후 counter == 4일 때 counter 값을 5로 바꿔주면서 read 인 경우(load) hit을 1로 하여 load 된 값을 내보내줄 수 있도록 구성을 한다. 이 역시

```
mux4_1 cache_word (memory_address[1:0], cache_line[memory_address[2]][way][15 : 0], cache_line[memory_address[2]][way][31 : 16],
        cache_line[memory_address[2]][way][47 : 32], cache_line[memory_address[2]][way][63 : 48], cache_word_mux);

assign out_address = mem_read_m ? {memory_address[15:2], 2'b00} : evict_pc[memory_address[2]][way];
assign inout_data = mem_write_m ? cache_line[memory_address[2]][way] : 64'bz;
assign caching_data = (hit && read_m) ? cache_word_mux : 16'bz;
```

Hit과 read_m이 모두 1인 경우에 한해서 offset과 mux를 통해서 결정된 cache 값을 내보내줌을 확인할 수 있다.

그리고 pc 값을 제어를 하기 위해서 D cache의 miss가 난 경우 그때의 miss 값을 이용해 cpu를 stall을 해주고,

```
assign read_m2 = mem_read_m;
assign write_m2 = mem_write_m;
assign mem_stall = miss;
```

```
assign is_flush = (id_ex_jalr && !(jalr_check)) || (!(b_j_check) && is_BJ_type_update) || (is_stall === 1) || mem_read_m || mem_counter;
```

Cpu top module에서 mem_counter(==D_cache의 miss 값) 와 mem_read_m이라고 되어 있는 I_cache의 miss인 경우 각각의 경우에 한해서 branch_predictor가 값을 바꾸지 못하

도록 하여 pc_value가 유지가 될 수 있도록 한다. 그리고 datapath 내부에서는 baseline에서 진행이 된 것처럼 D_cache의 miss가 난 경우에 더 이상의 datapath가 흘러가지 못하도록 매 cycle 마다 자기자신의 값을 넣어서 각각의 state에 해당하는 register에 있는 값이 변경 불가하도록 제어를 해뒀다.

```
always @(posedge clk) begin
        if(is_stall || miss) begin
                if_id_pc_reg <= if_id_pc_reg;
                if_id_instruction_reg <= if_id_instruction_reg;

        end
        else if(id_ex_jalr && !(jalr_check)) begin
                if_id_pc_reg <= if_id_pc_reg;
                if_id_instruction_reg <= 16'bz;

        end
        else if(is_BJ_type_update && !(b_j_check)) begin
                if_id_pc_reg <= if_id_pc_reg;
                if_id_instruction_reg <= 16'bz;

        end

        else begin
                if_id_pc_reg <= pc_value;       //reg pc? ?? ?? ?
                if_id_instruction_reg<= data1;      //instruction ????? ?

        end
end
```

그 예시는 위에서 보는 바와 같이 miss 값을 이용하여 자기 자신을 그대로 넣어주도록 하였다.

C. 나머지 모든 module

나머지 모든 모듈은 전 lab에서 진행한 pipeline과 동일하게 구성이 되어 있다.

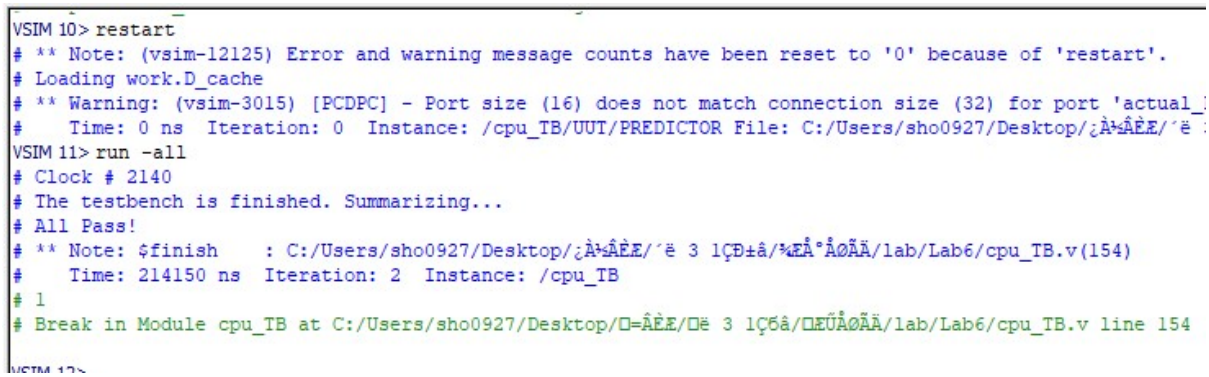
Discussion

이번 lab에서는 현재까지 memory 접근 역시 1cycle이 걸린다고 가정을 했던 것을 memory 접근을 할 때에는 특정 cycle이 걸린다는 것을 기반으로 하여, cache가 있는 pipeline과 없는 pipeline의 구현을 해보고 성능을 비교해보는 것이 목표가 되는 lab이다. 우리의 lab의 baseline cpu부터 성능을 보자면

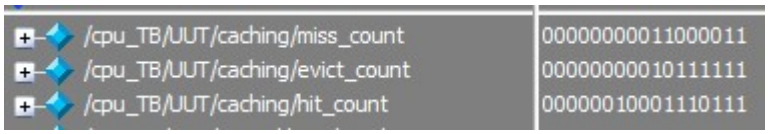
```
# WARNING: No extended dataflow license exists
VSIM 2> run -all
# Clock # 2444
# The testbench is finished. Summarizing...
# All Pass!
# ** Note: $finish    : C:/intelFPGA_pro/20.4/cpu_TB.v(154)
#    Time: 244550 ns  Iteration: 2  Instance: /cpu_TB
# 1
# Break in Module cpu_TB at C:/intelFPGA_pro/20.4/cpu_TB.v lin

VSIM 3>
```

다음과 같이 2444 cycle이 걸리는 것을 확인할 수 있다.

Cache가 있는 pipeline을 보면



2140 cycle로 감소를 한 것을 확인할 수 있다. 그리고 또한 I cache와 D cache 각각에 대해서 hit, miss, evict 횟수를 count를 해보면, I cache에 있어서는 hit 횟수가 948회, miss가 195, evict가 191 회인 것을 확인할 수 있었고,



우리의 hit_count는 miss가 난 경우 counter가 5가 되었을 때 최종적으로 cache line에 있는 data를 내보내기 위해서 hit_count를 1 증가시키기 때문에 위에 있는 wave form으로만 했을 때는 1143회 지만 miss 횟수만큼을 뺀 948회가 정확한 hit 횟수이다. 이 두개를 합쳐보면 948 cycle의 hit과 miss가 나는 경우 6 cycle의 penalty가 존재하는 것을 합치면 2118 cycle이고, data hazard stall이나 D_cache stall에 의해서 2140 cycle과는 조금의 차이가 존재하는 것을 확인 할 수 있다.

D_cache의 경우 hit이 234회, miss가 6회, evict가 2회가 일어나는 것을 확인할 수 있고,

위의 I_cache와 동일하게 hit 횟수는 240회로 count 되었지만 miss인 경우 hit이 1씩 더 올라가는 것을 고려하면 234회의 hit이 난 것을 확인할 수 있다.

Cache를 구현한 pipeline이 1.14배의 성능 향상이 있는 것을 확인할 수 있다.

Conclusion

결과적으로 이번 cache lab을 모두 구현을 완료 하였고 cache의 모든 기능을 구현을 완료하였다. 또한 cache의 hit과 miss, evict 횟수를 계산을 하는 것 역시 완료를 하여 제시를 하였고, Cache가 있는 cpu와 baseline cpu (cache가 없는 cpu)의 성능 차이와 동작의 차이를 확인할 수 있는 lab이었다고 생각된다. 또한 LRU, write back에서 cache line이 어떻게 update가 어떻게 되는지를 확인

할 수 있었다. 마지막으로 memory 접근을 지금까지는 1cycle이라고 가정을 하고 구현을 하였던 pipeline cpu에서 memory 접근이 몇 cycle이 필요하게 됨에 따라서 hit인 경우 빠르게 memory 값을 가져올 수 있는 cache의 필요성과 그 성능의 향상을 알게 되었다.