# 컴퓨터 구조 Lab7

20190782 컴퓨터공학과 최서영
20190439 컴퓨터공학과 오승훈

## Introduction

이번 Lab은 기존의 구현했었던 Pipelined cpu과 cache에 external I/O device가 CPU를 거치지 않고 직접적으로 memory에 접근하는 DMA를 구현하는 것이다. external I/O device와 CPU 모두 memory를 access하고자 할 때 한 개의 bus를 이용해야 한다. 따라서 bus를 사용하는 장치의 결정이 필요한데 이를 DMA controller에서 하게 된다. 이번 lab에서는 특정 clock cycle에서 특정한 memory에 12 size의 data를 기록하는 DMA controller를 구현하였다.

## Design

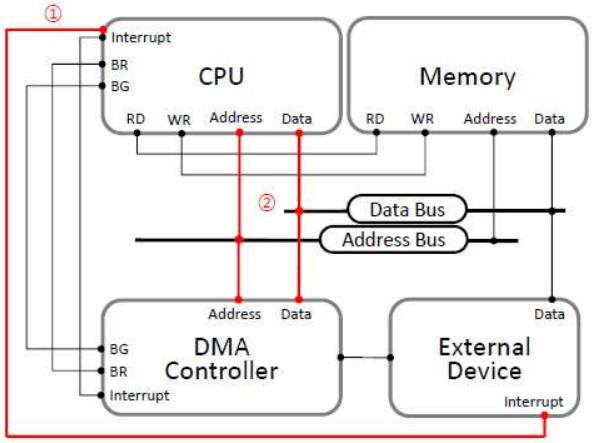



**그림 1. CPU, MEM, DMAC, External Device 간의 Diagram**

작동하는 방식은 다음과 같다.

1. External Device가 CPU에 DMA start를 요청하는 interrupt를 보낸다. 그럼 언제든 interrupt를 받을 수 있던 CPU가 interrupt를 받아들인다. 그후 CPU가 interrupt를 handling하여 DMA controller에 address와 length의 정보가 담긴 command를 보낸다.
2. DMA controller에서 command를 받으면 signal BR를 raise하여 CPU에 보내준다.
3. CPU에서 signal BR이 raise 한 것을 받아들이면 CPU 내부에서 memory에 접근하고 있는지를 체크한다. 만약 접근 중이라면, memory access가 끝 날때까지 기다린 후, 끝나고 signal

BG를 raise한다.

4. BG와 BR 둘 다 raise하다면 external device에서 memory access를 시작한다. 이때 4word씩 memory에 적으며 external device에서 접근하고자 하는 data는 DMA controller에서 offset을 사용하여 알려준다. 또한 4word의 data를 적기위해선 6cycle이 걸리게 된다.

   external device에서 memory access 중 cpu에서 memory access가 필요하다면 external device의 접근이 끝날 때까지 기다리게 된다. 이때, memory access가 되지 않는 것이지 cache에 저장된 값을 이용하여 cpu는 계속 작동을 하게 된다.

5. 그후 DMA controller가 지정된 length만큼의 data를 저장하면 access를 중단한다.

6. External device에서 memory access를 다하면 BR signal을 clear한다.

7. CPU가 BR signal이 clear되면 BG signal 또한 clear한다. 또한 CPU에서 memory access가 가능해진다.

8. DMA controller가 interrupt를 발생시킨다.

이번 lab에서는 pipelined cpu에서 사용하였던 adder, alu, alu_control_unit, control_unit, cpu, datapath, register_file, util, sign_extend, forwarding_unit, branch_predictor, I_cache, D_cache와 더불어 DMA_controlh와 external_device 모듈이 추가되었고, cpu 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 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.

DMA_control은 위에 설명한 DMA 시나리오에서 DMA controller의 역할을 하는 module이다. external_device는 가상의 external I/O device의 역할을 하는 module이다. 이 두 모듈의 작동방식은 위에 설명한 것과 같다.

## Implementation

A. DMA_control 부분

DMA_CONTROL 부분은 현재 DMA가 동작을 실행하는지를 알려주는 control unit이다. counter를 이용하여 구현을 하였고, 이 counter는 address의 offset을 알려주게 된다.

```
initial begin
        counter <= 4;
        BR <= 0;
        interrupt <= 0;
        interrupt_control <= 0;
end
```

먼저 위와 같이 interrupt와 BR을 0으로 초기화를 하고, counter는 4로 초기화를 시켜둔다.

```
if(address === 16'bz) begin
        BR = 0;
end
else if(counter == 3) begin
        BR = 0;
        interrupt = 1;
end
else if(counter == 5) begin
        interrupt = 0;
end
else begin
        BR = 1;
end
```

```
if(BG) begin
        if(counter == 4) begin
                counter = 0;
        end
end
```

그 후 combinational logic을 이용하여 DMA_control에 address가 16'bz가 아닌 실제 값을 access 하고 싶은 값이 들어오게 되면 그때 BR을 1로 해주는 else 구문이 존재하는 것을 확인할 수 있다. 그후 BG가 1이면서 counter가 4이면 counter를 0으로 만들어주고, 동작을 시작한다.

```
always @ (posedge clk) begin
        if(BG) begin
                if(clk_counter === 5) begin
                        counter <= counter + 1;
                end
        end
```

Counter는 clk_counter라고 하는 external_device의 counter가 5이면 1씩 올라가게 되는데 이는 external device 역시 memory 접근을 하는 것에 6cycle이 걸리기 때문이다. 그래서 이것을 총 3번을 반복하면 counter 값이 3이 되고, combinational logic에 있는 것에 따라 BR은 0이 되고, interrupt는 1이 되면서 DMA_control unit에서 해야하는 모든 동작을 완료를 한다.

B. External_device 부분

```
        interrupt <= 0;
end
else begin
        wait_count <= wait_count + 1;
        if(wait_count  == 2044) begin
                interrupt <= 1;
        end
        else begin
                interrupt <= 0;
        end
end
```

```
else begin
        if(offset_counter == 4) begin
                output_data <= 64'bz;
                counter <= 0;
        end
        else if(counter == 0) 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
                counter <= 5;
                write_m2 <= 1;
                output_data [15:0]<= data[offset];
                output_data [31:16]<= data[offset + 1];
                output_data [47:32]<= data[offset + 2];
                output_data [63:48]<= data[offset + 3];
        end
        else if(counter == 5) begin
                write_m2 <= 0;
                counter <= 0;
        end
end
```

Reset_n이 1이 되면 그 때부터 wait_count를 1씩 추가를 해준다. 그래서 2044 cycle이 되면 그때 interrupt를 시작하게 된다. 그리고 DMA_control 에 존재하는 counter가 0이되면 그때부터 counter가 매 posedge clk 마다 1씩 증가하게 동작을 하는데 이는 위에서도 언급했듯이 memory 접근을 하는데 6cycle이 걸리기 때문이고, 4번째 clock일 때 memory에 값을 적기 위해서 data를 밖으로 내보내고, output_data를 보내주게 된다.

C. Cpu와 Cache, 그리고 datapath 수정 부분

```
assign BG = (BR == 1) && (i_counter_sig_out == 0) && (d_counter_sig_out == 0) ? 1 : 0;
assign BG_input = (BR == 1) && (i_counter_sig_out == 0) && (d_counter_sig_out == 0) ? 1 : 0;
```

다음과 같이 BG를 구현을 하였고, i_counter_sig_out 과 d_counter_sig_out은 I-cache와 D-cache에서 현재 memory를 접근을 하고 있는지 아닌지를 판별해주는 bit이다. 그래서 이게 아니라면 그때부터 BG bit를 1로 사용할 수 있도록 하였고, I cache와 D cache 모두에 밑에 보이는 사진과 같이 BG_input 즉 BG 값을 넣어서 miss가 나면 handling은 되지 않지만 miss bit이 1이 되면서, stall 또는 flush가 되도록 구현을 하였다.

```
else begin                                          //cache miss
        if(BG_input) begin
                BG_flush_signal = 1;
                miss = 1;
                mem_read_m = 0;
        end
        else begin
                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;
                BG_flush_signal = 0;
        end
end
```

그래서 밑의 사진에 나오는 is_flush bit는 설정이 되면 pc_value가 올라가지 않도록 한 것이기 때문에 mem_counter 와 BG_flush 값이 현재 I_cache와 D_cache의 miss bit이므로 pc_value가 유지가 됨을 확인할 수 있다.,

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

Stall은 instruction과 다른 register 값들이 현재 register에 유지가 되면서, pc_value가 유지가 될 수 있도록 구현을 한 것을 확인할 수 있다.

```
always @(posedge clk) begin
        if(miss == 1) begin
                id_ex_is_BJ_type_update <= id_ex_is_BJ_type_update;
                id_ex_jalr <= id_ex_jalr;
                id_ex_pc_to_reg <= id_ex_pc_to_reg;
                id_ex_pc_reg <= id_ex_pc_reg;
                id_ex_S <= id_ex_S;
                id_ex_reg1 <= id_ex_reg1;
                id_ex_reg2 <= id_ex_reg2;
                id_ex_sign_out <= id_ex_sign_out;
                id_ex_reg_index1 <= id_ex_reg_index1;
                id_ex_reg_index2 <= id_ex_reg_index2;
                id_ex_rd_index <= id_ex_rd_index;
                id_ex_write_m2 <= id_ex_write_m2;
                id_ex_read_m2 <= id_ex_read_m2;
                id_ex_reg_write <= id_ex_reg_write;
                id_ex_halt <= id_ex_halt;
                id_ex_wwd <= id_ex_wwd;
                id_ex_alu_src <= id_ex_alu_src;
                id_ex_instruction <= id_ex_instruction;
                id_ex_F <= id_ex_F;
        end
```

D. Cpu_TB

```
assign MEM_READ_1 = BG ? 0 : read_m1;
assign MEM_READ_2 = BG ? 0 : read_m2;
assign MEM_WRITE_2 = BG ? EX_WRITE_M2 : write_m2;
assign data2 = BG ? output_data : (read_m2 ? 64'bz : data2_CPU);
assign data2_CPU = read_m2 ? data2 : 64'bz;

assign MEM_ADDRESS_2 = BG ? (DM_address + offset_counter * 4) : address2;
// instantiate the unit under test
cpu UUT (clk, reset_n, read_m1, address1, data1, read_m2, write_m2, address2, data2_CPU, num_inst, output_port, is_halted, BG, BR,
interrupt_EX, interrupt_CON, DM_address, data_length);
Memory NUUT(!clk, reset_n, MEM_READ_1, address1, data1, MEM_READ_2, MEM_WRITE_2, MEM_ADDRESS_2, data2);
DMA_control DMA(clk, DM_address, data_length, clk_counter, BG, BR, interrupt_CON, offset_counter);
external_device EX_DEVICE(clk, reset_n, offset_counter, output_data, interrupt_EX, clk_counter, EX_WRITE_M2);
```

TB에서는 다음과 같이 module을 추가해주고, Memory의 read_1, read_2, mem_write가 BG bit가 1이냐 아니냐에 따라서 값이 바뀌기 때문에 BG 값에 따라 삼항 연산으로 위와 같이 진행이 된 것을 확인할 수 있고, data2의 경우 inout port이므로 BG가 설정이 된 경우와 그 외에도 read를 하냐 write을 하냐에 따라 wire의 흐름이 달라지기 때문에 그 흐름에 맞게 read_m2가 1인 경우에는 CPU로 memory가 들어갈 수 있도록 구현을 해주고, read_m2가 0인 경우에는 write가 될 수 있으니 data2로 cpu top module에서 온 data가 memory module로 들어갈 수 있도록 구현을 해주면 된다.

E. 나머지 module

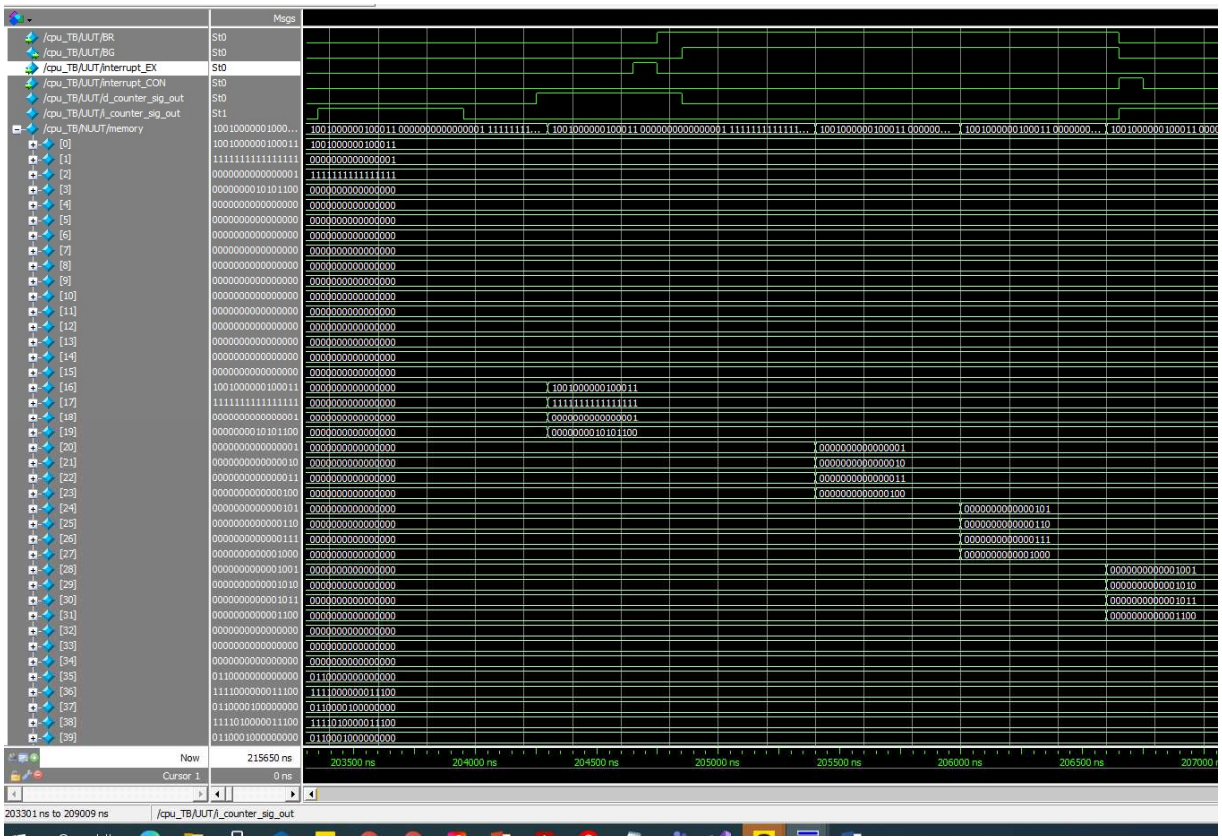
나머지 모든 module은 기존의 것과 동일하게 구현이 되었습니다.

## Discussion

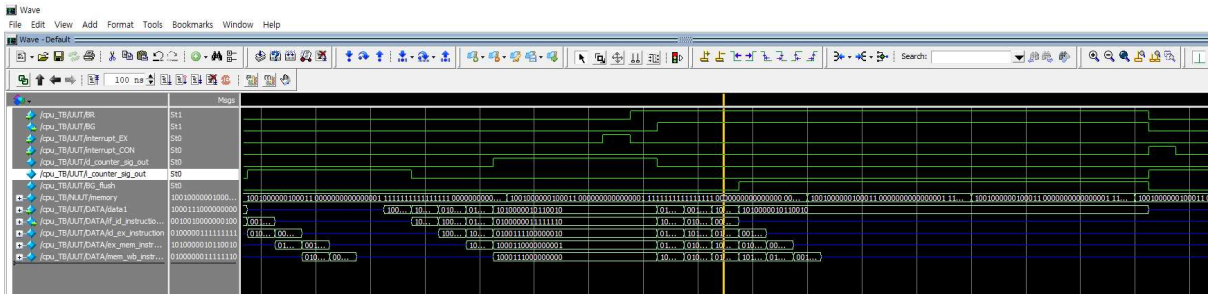
이번 lab에서는 현재까지는 이미 memory에 있는 data에 한해서 접근을 하고, instruction을 읽어와 동작을 수행했던 것을 그것과 별개로 있는 다른 module에서 값을 읽어오고, bus(wire)를 공유하여 memory 접근을 하는 방법을 구현해보는 lab이라고 할 수 있다.

다음과 같이 BR, BG, interrupt, 그리고 6cycle 마다 Memory에 data가 잘 적히는 것을 확인할 수

있다.



그리고 또한 BR이 request가 되더라도, 지금 현재 d_counter_sig_out이 1이기 때문에 d_counter_sig_out 즉 d_cache의 cache 접근이 모두 끝마친 후에 BG가 1이 됨 역시 확인할 수 있었고, 모든 external device의 동작이 완료된 후 BG, BR이 정상적으로 0으로 내려오며, DMA_control에서 interrupt 역시 발생하는 것을 확인할 수 있다. 그리고 중간에 i_cache의 stall이 일어나면서 pipeline 역시 stall 되는 것은 instruction을 열어봄으로써 확인이 가능한데



다음과 같이 BG_flush가 1이 되면서 I cache의 miss가 발생한 상황에서 더 이상 instruction이 진행되지 않고, flush로 채워지는 것 역시 확인이 가능하다.

## Conclusion

결과적으로 이번 cache lab을 모두 구현을 완료 하였고 DMA의 모든 기능을 구현을 완료하였다. 또한 TB를 통하여 Memory에 정상적으로 동작하는지와 I_cache 또는 D_cache에서 miss가 발생할

시 동작이 stall이 되는지 역시 확인을 하였고, 정상적으로 stall이 된다는 것을 확인을 할 수 있었다. 위와 같이 현재 여기서는 wire이지만 BUS를 공유하여 I/O 작업을 진행할 수 있으며, BUS를 공유하고 있다는 특성상 external device에서 동작을 수행할 때 I_cache나 D_cache는 cache line handling이 불가능하게 만들어야하며, external device가 모든 동작을 수행한 후 I_cache나 D_cache의 cache line을 수정을 해야한다는 것을 확인할 수 있었다.