

Lab 7 - DMA

20160595 수학과 최규민
20160169 물리학과 최수민

1. Introduction

이번 과제에서는 외부 I/O Device와의 DMA(Direct Memory Access)를 구현한다. DMA가 없다면 외부 I/O Device와 메모리 사이의 데이터 전송을 하려면 CPU가 I/O Device에서 데이터를 읽고 이를 메모리에 작성하는 등, CPU가 둘 사이의 데이터 전송에 관여해야 한다. 따라서 CPU는 다른 일을 처리하지 못하고 stall되어야 해서 성능의 저하가 발생할 수 있다. 이 때 DMA Controller가 대신 데이터 전송을 처리한다면, I/O Device와 메모리 사이의 데이터 전송을 하는 동안 CPU는 메모리가 필요하지 않은 일들을 처리할 수 있기 때문에 성능 저하를 일부 방지할 수 있다. 따라서 이러한 DMA Controller를 구현하여 외부 I/O Device로 인한 데이터 전송을 CPU 대신 처리하여 성능 저하를 방지하도록 한다.

이번 과제를 통해 배워야 하는 것은 다음의 두 가지로 요약할 수 있다.

1. 외부 I/O Device와 메모리 사이의 데이터 처리를 위해 Bus transaction을 어떻게 관리할 수 있는지 이해한다.
2. DMA Controller 구현을 통해 외부 I/O Device로 인한 성능 저하를 어떻게 방지할 수 있는지 이해한다.

2. Design

먼저 주어진 시나리오에 따라 어떤 값을 어떻게 전달해야 하는지 파악한다.

1. 외부 I/O Device에서 CPU로 interrupt를 발생시킨다.

임의로 interrupt는 num_clock이 184일 때 발생시키고, num_clock이 185인 cycle 1개 동안 유지한다. 이 때 CPU는 언제든지 interrupt를 받아 처리해야 하므로 interrupt handling은 combinational logic으로 작성한다.

2. CPU에서 DMA Controller로 데이터 길이와 저장할 메모리 주소를 전달한다.

데이터의 길이는 12 bytes로, byte 단위의 값인 12를 보낸다. 저장할 메모리 주소 범위는 23~340이므로 그 시작 주소인 23을 보낸다. 이 때 전송한 값이 유효한 값임을 보장하기 위해 dma_valid를 1로 함께 설정한다.

3. DMA Controller에서 BR(Bus Request) 시그널을 가한다.

`dma_valid`값이 1이 되면 CPU에서 전송한 메모리 주소와 데이터 길이를 받아 이를 저장해둔다. 그와 동시에 BusRequest 시그널을 1로 만든다.

4. CPU에서 BR 시그널을 받으면 진행하던 메모리 작업을 마치고 메모리의 address bus, data bus, RD/WD line의 사용을 막는다.

진행하던 메모리 작업이 있는지에 대한 여부는 메모리의 port2의 read, write signal인 `read_m2`, `write_m2` 중 하나라도 값이 1인 게 있는지 확인한다. 만약 하나라도 값이 1이면 그 값이 0이 될 때까지 기다리고, 둘 다 0일 때부터 CPU의 메모리 port2에 대한 bus access를 끊고 busGrant 시그널을 1로 설정한다(5번의 내용).

5. CPU에서 BG(Bus Granted) 시그널을 가한다.

6. DMA Controller에서 BG 시그널을 받으면, 외부 I/O Device가 지정된 메모리 주소에 12 words를 쓰도록 한다.

BG 시그널을 받으면 DMA Controller의 메모리 port2에 대한 bus access를 활성화시킨다. 이후 `offset`을 변화시키며 메모리에 외부 I/O Device의 데이터를 쓴다. 메모리에는 한 번에 4 words만큼만 쓸 수 있으므로, CPU에서 전달받은 데이터의 길이를 4로 나눈 횟수만큼 메모리 요청을 해야 한다. 그에 따라 `offset`을 0부터 (데이터의 길이 / 4) - 1까지 변화시켜가며 메모리 요청을 한다. `offset`을 바꾸는 시점은 메모리에서 쓰기가 완료되었음을 의미하는 시그널인 `ackOutput20| assert`될 때로 한다.

7. 메모리 전송이 이루어지는 동안, CPU는 메모리 port2에 접근하지 못하고 cache만 사용 가능하다.

4번에서 메모리 port2에 대한 bus access를 끊기 때문에 CPU는 메모리 port2에 접근하지 못한다. 이 때 만약 Cache miss가 되어 메모리에 대한 접근이 필요한 상태가 되면 캐시에서 그 상태를 유지하고 datapath의 pipeline을 stall하여 메모리 접근을 하지 않도록 한다.

8. DMA Controller에서 작업을 마치면 BR 시그널을 지운다.

DMA Controller에서 `offset`이 (데이터의 길이 / 4) - 1인 경우까지 메모리 쓰기가 완료되면 BR 시그널을 0으로 만든다. 이와 동시에 DMA Controller에서의 메모리 port2에 대한 bus access를 비활성화시킨다.

9. CPU에서 BR 시그널이 지워진 것을 확인하면 BG 시그널을 끄고 memory port2에 대한 bus access를 다시 활성화시킨다.

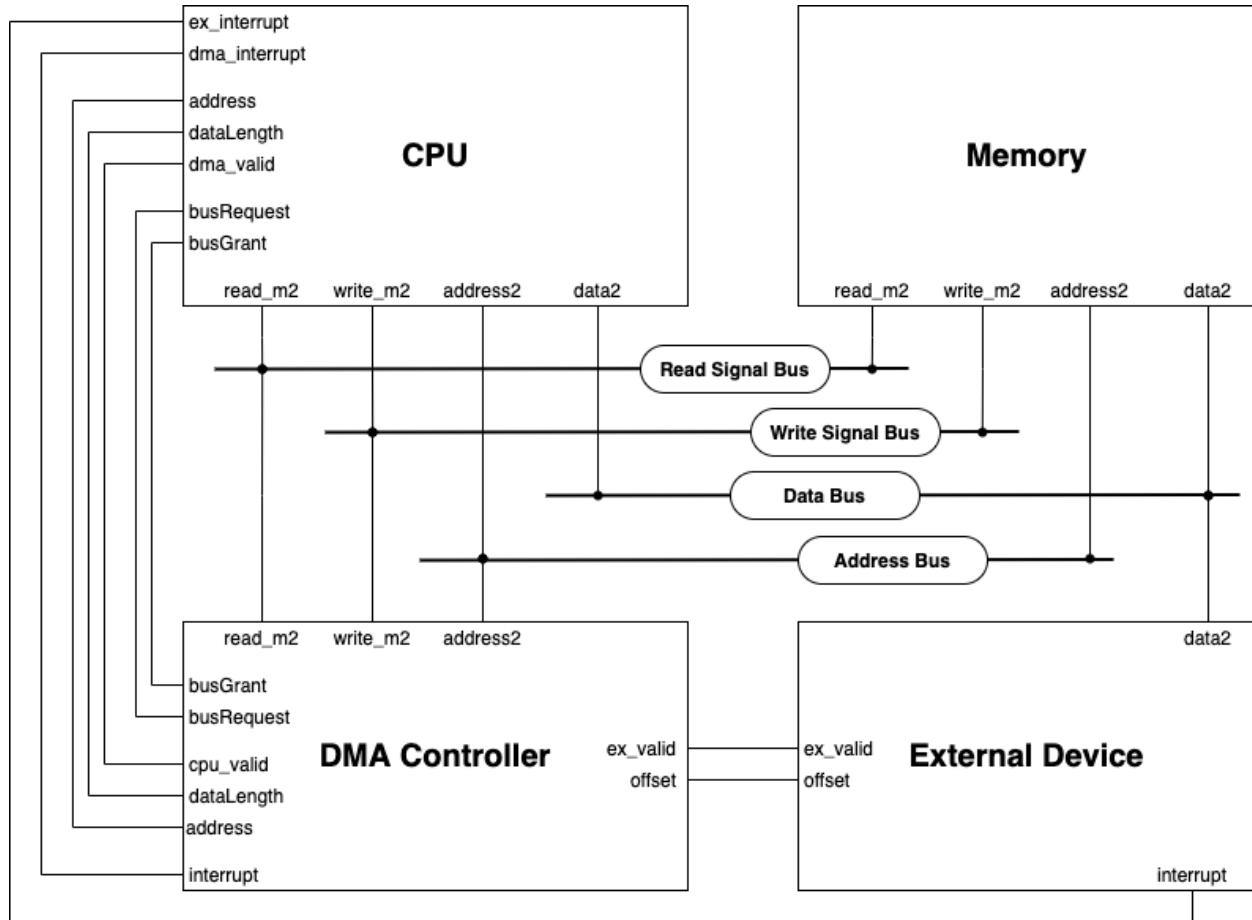
10. DMA Controller가 외부 I/O Device의 메모리 작성이 끝났음을 알리는 interrupt를 발생시킨다.

BG 시그널이 꺼진 것을 확인하면 interrupt를 발생시킨다. 이 때 외부 I/O Device의 interrupt와 마찬가지로 한 cycle동안 유지한다.

11. CPU에서 interrupt를 처리한다.

이 때 별도로 처리할 것이 없기 때문에 interrupt만 받고 실제로 수행하는 것은 없다.

그림 1은 위의 시나리오 각 단계의 설명에서 모듈별로 필요한 interface와 그 연결 방식을 나타낸 것이다.



[그림 1] CPU, Memory, DMA Controller, External Device의 interface. CPU의 포트들을 기준으로 ex_interrupt, dma_interrupt 포트는 각각 External Device와 DMA Controller에서 발생시키는 interrupt를 받기 위한 것이고, address, dataLength, dma_valid 포트는 ex_interrupt가 일어났을 때 DMA Controller에 데이터를 쓸 메모리 주소, 데이터의 길이(bytes), 그리고 요청이 유효함을 나타내는 시그널을 위한 것이다. busRequest와 busGrant는 시나리오에서 BR, BG 시그널에 해당하는 포트이다.

지난 Lab6에서의 Cache가 있는 CPU와 크게 달라진 점은 없기 때문에 변경할 모듈만 언급하려고 한다. DMA Controller와 External Device 연결을 위해 추가되거나 변경한 Design 사항은 다음과 같다.

2.1. CPU 모듈

기존 CPU 모듈에서 추가된 port들을 설명하면, input으로 ex_interrupt, dma_interrupt, busRequest를 받고, output으로는 dma_valid, address, dataLength, busGrant가 있다. ex_interrupt, dma_interrupt는 각각 external device와 DMA controller에서 보내주는 interrupt 시그널을 받는 port이고, busRequest또한 DMA controller에서 보내주는 시그널을 읽는다. dma_valid, address, dataLength는 CPU가 external device로부터 interrupt를 받은 후 보내주는 command이고, busGrant는 busRequest를 받은 이후 CPU의 bus 사용을 모두 마친 후 DMA에게 보내주는 시그널이다. Interrupt, BR, BG 등의 시그널을 통해 external device, DMA controller와 소통하며 메모리 접근을 관리한다. 내부적으로는 CPU의 Memory Access 여부를 판단하기 위해 내부적으로 bus_access reg를 활용하는데, 이 값에 따라서 CPU의 Memory port2 사용을 막는다.

2.2. datapath 모듈

기존 datapath 모듈에서 달라진 점은 거의 없다. 단, external device 혹은 DMA controller가 보내는 두 종류의 interrupt인 ex_interrupt(DMA begin), dma_interrupt(DMA end)가 들어오면 CPU가 멈춰야 하므로, 전체적인 파이프라인을 stall 해준다는 점이 이전 모듈과의 차이점이다.

2.3. Memory 모듈

Memory 모듈은 이전 모듈과 동일한 구현을 가지고 있다. 단, external device에서 입력받은 data를 memory에 쓰고, 입력이 끝난 후에는 실제 값을 Memory에서 읽을 수 있어야 하는데, 구현을 위해 사용되지 않는 주소에 external device data를 저장할 수 있게 한다. 이를 위해 설정한 메모리 주소는 Memory ['d23] ~ ['d34]이다.

2.4. d_cache 모듈

d_cache 모듈에서도 CPU에서 사용하는 bus_access의 값에 따라서 Memory와의 연결을 끊는다. 따라서 Cache Miss가 일어나더라도, bus_access가 assert되지 않고 있으면 더 이상 메모리에 접근할 수 없고, 작업을 중단한다.

2.5. dma_controller 모듈

dma_controller 모듈은 External Device와 Memory 간 Data transfer가 있을 때 CPU가 다른 작업을 할 수 있도록 data transfer를 관리해주는 모듈이다. Input으로는 CPU로부터 cpu_valid, address, dataLength, busGrant를 받고, Memory에서 ackOutput2를 받는다. output으로는 CPU로 interrupt, busRequest를 보내주고, external device로는 ex_valid, offset, Memory로 read_m2, write_m2, address2를 보내준다. cpu_valid, address, dataLength를 통해 CPU로부터 memory의 어느 주소에 data를 얼마나 저장할 지 전달받고, busRequest로 CPU에 memory 접근 요청 시그널을 보낸 후 busGrant를 통해 CPU의 memory 접근 block이 완료되어

dma_controller가 이제 memory에 접근할 수 있음을 전달받는다. 그 후 external device에 ex_valid, offset신호로 memory에 접근 요청을 보내고, memory와는 read_m2, write_m2, address2, ackOutput2 신호를 주고받아 소통한다.

2.6. external_device 모듈

external device 모듈은 input으로는 DMA Controller에서 받는 ex_valid, offset이 있고, output으로는 CPU로 보내는 interrupt, inout 포트로 data2를 가지고 있다. 184 cycle이 지난 후 interrupt를 raise한다. 이것은 현재 device가 메모리로 보낼 데이터를 최소 12 word 가지고 있다는 것을 의미하며, dma_controller에서 보내주는 offset을 이용하여 적절한 data를 data2 bus로 보내준다.

3. Implementation

3.1. CPU_TB 모듈

```
// for DMA
wire ex_interrupt, dma_interrupt, dma_valid;
wire [WORD_SIZE-1:0] address;
wire [WORD_SIZE-1:0] dataLength;
wire busGrant, busRequest;
wire ex_valid;
wire [WORD_SIZE-1:0] offset;

// instantiate the unit under test
cpu UUT (clk, reset_n, read_m1, address1, data1, inputReady1, read_m2, write_m2, address2, data2, inputReady2, ackOutput2, num_inst, output_port, is_halted,
         ex_interrupt, dma_interrupt, dma_valid, address, dataLength, busGrant, busRequest);
Memory NUUT (clk, reset_n, read_m1, address1, data1, inputReady1, read_m2, write_m2, address2, data2, inputReady2, ackOutput2);
external_device EX (clk, reset_n, ex_interrupt, ex_valid, offset, data2);
dma_controller DMA (clk, reset_n, dma_interrupt, dma_valid, address, dataLength, busGrant, busRequest, ex_valid, offset, read_m2, write_m2, address2, ackOutput2);
```

[그림 2] CPU_TB 모듈 중 일부. cpu, Memory, external_device, dma_controller 모듈을 인스턴스화하고, 각 포트를 알맞게 연결해주고 있다.

CPU_TB 모듈은 테스트 벤치에 해당하는 모듈로, 기존에 cpu와 Memory의 인스턴스만 선언되어 있었는데 external_device와 dma_controller에 해당하는 모듈을 추가로 인스턴스화하였다. 이 때 새롭게 생긴 포트들을 연결하기 위해 기존에 없던 wire를 선언해주었고, 이를 이용하여 각 포트를 알맞게 연결하였다.

3.2. CPU 모듈

```
// when busGrant is asserted, block cpu's usage of memory port2
assign read_m2 = bus_access? d_read_m2 : 'bz;
assign write_m2 = bus_access? d_write_m2 : 'bz;
assign address2 = bus_access? d_address2 : 'bz;
assign data2 = bus_access? (read_m2? 'bz: d_data2) : 'bz;
assign d_data2 = bus_access ? (read_m2? data2: 'bz) : 'bz;

// use combinational logic so that CPU is always ready for the interrupt
always @(*) begin
    // 2. CPU send a address and dataLength to a DMA controller
    if (ex_interrupt) begin
        dma_valid = 1;
        address = 16'h17;
        dataLength = 12;
    end else begin
        // dma_valid is asserted for only one cycle
        dma_valid = 0;
    end

    // 4. CPU receive BusRequest signal and blocks its usage of the memory port2
    if (busRequest) begin
        // when current memory access is done
        // deassert the bus_access to block future memory access
        if (!read_m2 && !write_m2) begin
            bus_access = 0;
            busGrant = 1;
        end
    end

    // 9. CPU clears the BG signals and enables the usage of memory buses
    if (busGrant && !busRequest) begin
        busGrant = 0;
        bus_access = 1;
    end

    // 11. The CPU handles the interrupt from DMA controller
    if (dma_interrupt) begin
        // do nothing
    end
end
```

[그림 3] CPU 모듈의 일부. external_device와 dma_controller에서 발생한 interrupt를 처리하는 부분이다.
주석으로 표시한 숫자(2, 4, 9, 11)은 시나리오에서의 순서를 의미한다.

CPU 모듈에는 datapath 모듈과 cache 모듈이 선언되어 이들과 Memory간의 포트가 연결되어 있었다. 여기에 external_device와 dma_controller에서 발생시킨 interrupt를 처리하기 위해 combinational logic을 추가했다. Combinational logic에서는 interrupt를 처리하고, dma_controller와 BG, BR 시그널을 주고받으며 CPU에서의 메모리 port2 접근을 bus_access값을 통제한다. 먼저 external_device에서 DMA의 시작을 알리는 ex_interrupt를

발생시키면 dma_controller에 dataLength와 address값을 전달하는데, 이 두 값은 각각 12와 16'h17로 하드코딩되어 있다. 이후 dma_controller로부터 busRequest 시그널을 받으면 현재 진행 중인 memory access가 끝나자마자 CPU의 memory_access를 차단하고 busGrant 시그널을 1로 만든다. 이후 dma_controller의 busRequest 시그널이 꺼지면 busGrant 시그널도 0으로 만든다. DMA의 끝을 알리는 dma_interrupt가 발생하면 DMA는 끝난다. 이 때 dma_interrupt가 발생했을 때는 이번 시나리오에선 별도로 처리해야 할 내용이 없어 빈칸으로 두었다.

3.3. datapath 모듈

```

// update pc
if(!flush && (stall || instr_stall || mem_data_stall || interrupt)) 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 && !interrupt) begin
    pc_id <= pc;
    new_inst_id <= 1'bl;
end else if(!flush && (stall || instr_stall || mem_data_stall || interrupt)) 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 & !interrupt) 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 || interrupt) 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 4] datapath 모듈의 일부로, interrupt 발생 여부에 따라 pipeline을 stall하는 로직이 추가되었다.

Datapath 모듈은 외부 device로 인한 interrupt 발생 여부를 받아들이는 interrupt port가 추가되었으며, 이 interrupt값에 따라 기존의 stall 로직에 더하여 interrupt가 1이면 전체 파이프라인을 stall시키는 코드가 추가되었다. 이 외의 변경사항은 없다.

3.4. d_cache 모듈

```
// when busAccess is deasserted, cut the connection to data bus
assign data2 = busAccess? (read_m2? 'bz: mem_req_data2): 'bz;

// cache miss
else begin
    if (!busAccess) begin
        // if busAccess is not asserted, cache cannot use memory
        vstate = CHECK;
```

[그림 5] d_cache 모듈에서 수정된 부분. CPU의 bus_access값을 입력받는 busAccess 포트가 추가되었고, 이 값에 따라 data2 포트에 대한 사용을 끊고, cache miss가 발생했을 때 memory에 접근할지 여부를 결정한다.

두 개의 캐시 모듈 i_cache와 d_cache에서 i_cache는 메모리 port2에 대한 접근을 하지 않기 때문에 d_cache에만 코드 수정 사항이 발생하였다. d_cache 모듈에서 메모리 port2에 대한 접근을 조절하기 위해 CPU 모듈에서의 bus_access값을 input busAccess 포트로 받으며, busAccess값이 1인 동안에만 data2 포트의 값을 바꿀 수 있도록 한다. 또한 busAccess값이 0일 때 cache miss가 발생하면 Memory에 접근하지 못하므로 cache miss일 때 계속해서 CHECK 상태에 머무르도록 구현하였다.

3.5. dma_controller 모듈

```
// assert proper memory signal and send address through address bus
assign read_m2 = ex_valid? 0: 'bz;
assign write_m2 = ex_valid? 1: 'bz;
assign address2 = ex_valid? req_address + offset * 4: 'bz;

always @(posedge clk) begin
    // 10. The DMA controller raises an interrupt
    if (dma_on && !busRequest && !busGrant) begin
        dma_on <= 0;
        interrupt <= 1;
    end

    // turn off interrupt signal after one cycle
    if (interrupt) begin
        interrupt <= 0;
    end
end

always @(*) begin
    // 3. The DMA controller saves the address and dataLength sent from CPU
    //      and raises a BusRequest signal
    if (cpu_valid) begin
        dma_on = 1;
        req_address = address;
        req_data_length = dataLength;
        busRequest = 1;
    end

    // 6. The DMA controller get BG signal.
    //      Make external device writes 12 words of data at designated memory address
    if (busGrant) begin
        if (!access_memory) begin
            // first cycle after BG signal is asserted
            // send valid signal and offset to external device and remember we started memory access
            ex_valid = 1;
            offset = 0;
            access_memory = 1;
        end else if (ackOutput2) begin
            // get write-done signal from memory
            if (offset < req_data_length / 4 - 1) begin
                // if the data is left, increase offset
                offset = offset + 1;
            end else begin
                // if all data is written, make external device stop
                // 8. When the DMA controller finishes its work, it clears the BR signal
                ex_valid = 0;
                busRequest = 0;
            end
        end
    end

    // 9. CPU clears the BG signals and enables the usage of memory buses
    if (access_memory && !busGrant) begin
        access_memory = 0;
    end
end
```

[그림 6] dma_controller 모듈의 일부. 시나리오에 따라 필요한 값을 변경한다. 주석으로 표시한 숫자(3, 6, 9, 10)은 시나리오에서의 순서를 의미한다.

dma_controller 모듈은 external_device에서 memory로의 데이터 쓰기를 처리하는 모듈이다. 시나리오에서 CPU로부터 address와 dataLength값을 받으면 이를 저장하고, busRequest 시그널을 assert한다. 이후 CPU로부터 busGrant 시그널을 받으면, ackOutput2값에 따라

offset을 바꿔가며 external_device로 하여금 Memory에 offset에 따라 알맞은 4 words의 데이터를 쓰도록 한다. 이후 모든 데이터를 작성한 경우 busRequest 시그널을 끄고, 마지막으로 busGrant 시그널도 꺼진 것을 확인하면 interrupt를 발생한다.

3.6. External Device 모듈

```

localparam
    INTERRUPT_CLK = 'd184;

// when ex_valid is asserted, send data through data bus
assign data2 = ex_valid? output_data: 'bz;

always @(*) begin
    output_data = {data[4 * offset], data[4 * offset + 1], data[4 * offset + 2], data[4 * offset + 3]};
end

always @(posedge clk) begin
    if(!reset_n) begin
        data[16'd0] <= 16'h0001;
        data[16'd1] <= 16'h0002;
        data[16'd2] <= 16'h0003;
        data[16'd3] <= 16'h0004;
        data[16'd4] <= 16'h0005;
        data[16'd5] <= 16'h0006;
        data[16'd6] <= 16'h0007;
        data[16'd7] <= 16'h0008;
        data[16'd8] <= 16'h0009;
        data[16'd9] <= 16'h000a;
        data[16'd10] <= 16'h000b;
        data[16'd11] <= 16'h000c;
        num_clk <= 0;
    end
    else begin
        num_clk <= num_clk+1;
    end
    // 1. An external device sends an interrupt to a CPU
    if (num_clk == INTERRUPT_CLK) begin
        interrupt <= 1;
    end else begin
        // the interrupt is kept for one cycle
        interrupt <= 0;
    end
end
end

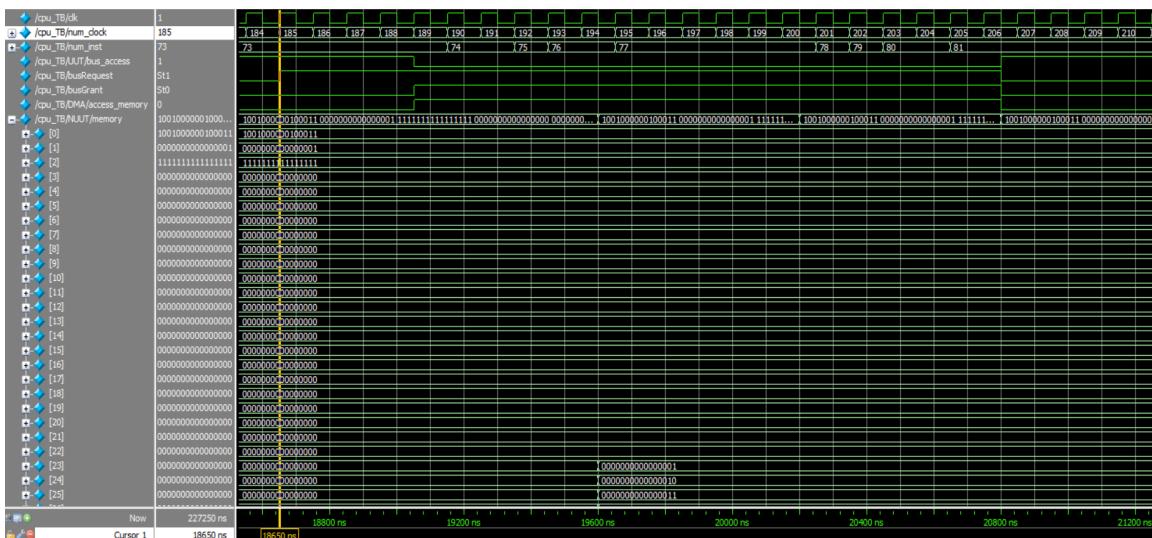
```

[그림 7] external_device 모듈 중 일부. 특정 클럭에서 interrupt를 발생시키고, dma_controller로부터 전달받은 ex_valid, offset값에 따라 적절한 4 words의 데이터를 메모리로 전송한다.

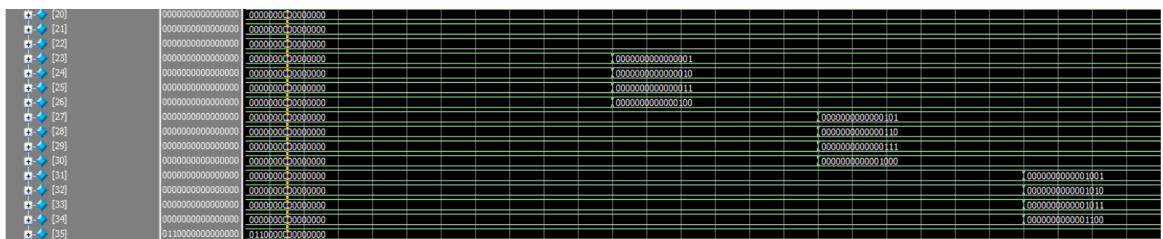
external_device 모듈은 메모리에 쓸 12 words가 선언되어 있으며, 클럭의 개수를 저장하고 있다가 INTERRUPT_CLK에 해당하는 클럭이 되면 interrupt를 1 cycle동안 발생시킨다. 이후 dma_controller에서 전달받은 ex_valid 값과 offset 값에 따라 Memory에 쓸 4 words씩 끊어 data2 버스로 전달한다.

4. Discussion

- 동작 확인



위 그림은 이번 구현의 waveform이다. bus_access(CPU), busRequest, busGrant, access_memory(DMA)의 waveform을 살펴보면 busRequest가 제일 먼저 인가되고 CPU 준비가 완료되면 CPU의 bus_access는 해제되면서 busGrant를 인가시키고, DMA의 access_memory가 인가되는 것을 확인할 수 있다. DMA의 메모리 접근이 끝나면 busRequest, busGrant, access_memory를 모두 해제하여 CPU의 bus_access도 다시 1로 돌아가게 된다.



Memory는 4 word 단위로 접근되므로, 총 3번의 memory transaction을 통해 총 12 word의 data를 memory에 쓰고, 이후 memory에서 읽어올 수 있음을 확인할 수 있다. 각 memory transaction은 6 cycle씩 소요되어 memory access에는 18 cycles이 걸렸다. README.txt에 이 waveform을 어떻게 확인할 수 있는지 자세히 적혀있다.

DMA Controller로 인한 성능 저하 방지

Without DMA Controller and External Device

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

With DMA Controller and External Device

```
# Clock # 2271  
# The testbench is finished. Summarizing...  
# All Pass!
```

외부 I/O Device로부터의 Memory 쓰기가 없는 기존 CPU 구현의 경우 2270 Clock이 걸리고, 이번에 구현한 외부 I/O Device로부터의 Memory 쓰기를 DMA Controller가 처리하는 경우 2271 Clock이 걸린다. I/O Device가 Memory에 접근하는 데는 최소 18 Clock이 필요한데도 불구하고, DMA Controller에서 이를 담당하여 처리하는 동안 CPU에서 메모리를 사용하지 않는 다른 일을 할 수 있으므로 성능의 저하가 거의 없다. 늘어난 1 cycle은 interrupt로 인한 pipeline stall로부터 온 것으로 보인다. 이를 통해 DMA Controller가 외부 I/O device로 인한 성능 저하를 잘 방지한다는 것을 확인할 수 있었다.

- **Bus Control**

이번 랩에서 CPU, Memory, I/O Device, DMA Controller가 address 및 data bus를 공유하도록 구현해야 했다. 이를 위해 BusRequest 시그널과 BusGrant 시그널을 사용하여 CPU가 버스를 사용할 수 있는지, DMA Controller 및 I/O Device가 버스를 사용할 수 있는지를 결정하였다.

5. Conclusion

외부 I/O Device와 DMA 구현을 통해 I/O Device와 메모리 사이의 Data Bus Transaction의 관리에 대해 이해할 수 있었고, I/O Device와 메모리 사이의 데이터 전송 도중 CPU는 메모리가 필요없는 일들을 처리하므로 I/O Device로 인한 성능저하가 없음을 확인하였다.