Lab 3 - Single Cycle CPU

20160595 수학과 최규민
20160169 물리학과 최수민

1. Introduction

이번 과제에서는 Single Cycle TSC CPU를 Verilog로 구현한다. CPU 모듈의 input은 메모리의 쓰기 완료와 읽기 대기를 나타내는 두 신호 ackOutput과 inputReady, reset signal(reset_n), clock signal(clk)로 이루어져 있고, output으로는 메모리에 쓰기 및 읽기 신호를 주는 readM과 writeM, 그리고 읽고 싶은 메모리의 주소 address로 이루어져있고, input과 output으로 모두 사용할 수 있는 data가 있다. CPU 모듈은 하나의 instruction을 하나의 clock cycle마다 처리한다.

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

1. Single Cycle CPU 구현을 통해 Datapath와 Control Unit을 설계하는 방법을 이해한다.
2. CPU의 하위 component들의 역할을 정확히 이해하고, 이를 modularization을 통해 적절히 설계하는 법을 익힌다.

2. Design

먼저 주어진 TSC instruction을 분석해 CPU의 Datapath를 구성했다. 강의 중 다룬 RISC-V의 Single cycle CPU와 비슷한 구조로 구성했지만, Instruction set이 다르기 때문에 달라진 부분들이 있다.

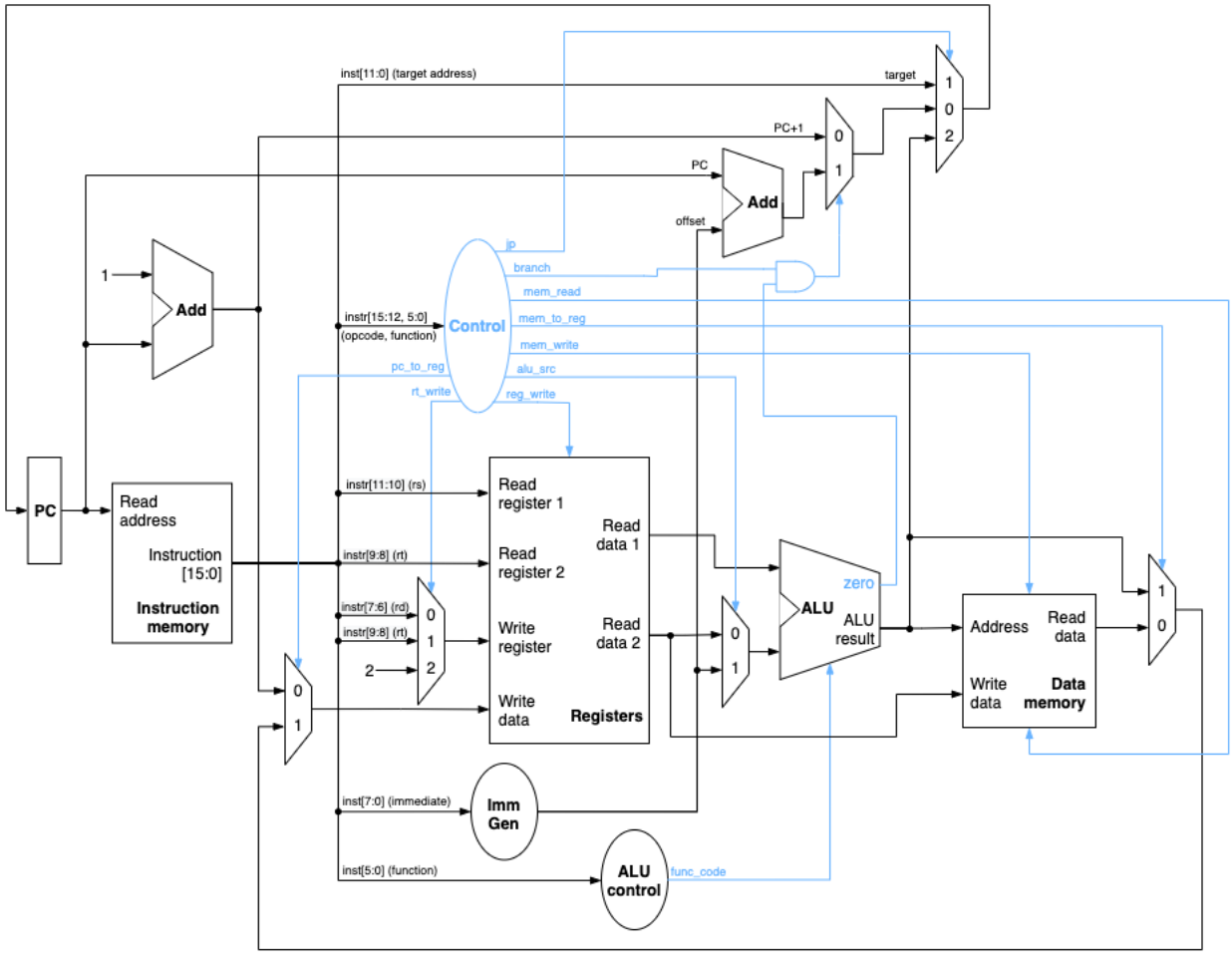
1. ADI, ORI, LHI, LWD instruction의 경우 레지스터 rd가 아니라 rt에 값을 쓴다. 또한 JAL, JRL은 레지스터 $2에 기존 PC값을 저장해야 한다. 이를 위해 Register file의 Write register 앞에 멀티플렉서를 추가하고, Control line에도 rt_write라는 값을 추가했다.
2. JPR, JRL의 경우 레지스터로부터 pc를 대체할 값을 가져오는 반면 JMP, JAL instruction의 경우 Instruction에 있는 Target address으로 값을 대체한다. 따라서 PC를 대체할 값을 결정할 때 사용하는 멀티플렉서를 확장했다.
3. ALU instruction과 JPR, JRL instruction의 Opcode가 모두 15로 동일하지만, Control signal은 다르게 나와야 한다. 따라서 Control 모듈에 Instruction의 Opcode 부분뿐만 아니라 Function에 해당하는 부분도 함께 Input으로 설정했다.

또한 이러한 Datapath와 Control line의 설계를 바탕으로 구현할 때 고려해야 할 부분들을 미리 정리했다.

1. 메모리는 Magic Memory로 가정하여 Clock cycle보다 작은 Delay를 갖기 때문에 Single cycle CPU로 충분히 구현할 수 있다. 다만 Instruction memory와 Data memory가 2개의

메모리로 분리되어 있지 않기 때문에 Verilog로 구현할 때 Posedge와 Negedge를 모두 사용하여 한 Cycle에서 메모리를 두 번 접근하도록 해야 한다. 즉, Posedge엔 Instruction을 가져오고, Negedge엔 메모리에서 데이터를 읽어오거나 메모리에 데이터를 쓰는 동작을 하도록 구현해야 한다.

2. ORI instruction의 경우 Immediate 값을 Zero-extension을 해야 한다. 설계하는 단계에서는 Sign-extension을 할지 Zero-extension을 할지 결정하는 Control line을 포함하지 않았는데, 실제 구현할 때 이를 유의해서 ALU 모듈을 구현해야 한다.



[그림 2] TSC instruction set을 분석한 것을 바탕으로 그린 single cycle CPU의 datapath와 control line. Write register를 선택하기 위해 rt_write control signal을 받는 멀티플렉서를 추가했고, jp control signal을 받는 멀티플렉서의 경우 TSC에 있는 jump instruction이 다양해 멀티플렉서를 확장했다.

이후 설계를 바탕으로 실제 구현할 때의 Verilog 모듈을 나누었다. 메인 모듈인 cpu 모듈 하나와 서브 모듈들로 나누었다. 서브 모듈은 설계에서의 각 역할에 따라 Instruction memory와 Data memory에 해당하는 메모리에 접근하는 memory_access 모듈, Instruction을 받아 Control signal을 생성하는 control_unit 모듈과 alu_control_unit 모듈, Register file에 해당하는 register_file 모듈, ALU에 해당하는 alu 모듈, Immediate Generate에 해당하는

immediate_generator 모듈, 그리고 각종 멀티플렉서에 해당하는 mux2to1 모듈과 mux4to1 모듈로 나누었다.

Instruction	alu_src	reg_ write	rt_write	mem_ read	mem_ to_reg	mem _write	jp	branch	pc_to_ reg
ALU instructions	0	1	0	0	0	0	0	0	0
ADI, ORI, LHI	1	1	1	0	0	0	0	0	0
LWD	1	1	1	1	1	0	0	0	0
SWD	1	0	0	0	0	1	0	0	0
Branch instructions	0	0	0	0	0	0	0	1	0
JMP	0	0	0	0	0	0	1	0	0
JAL	0	1	0	0	0	0	1	0	1
JPR	0	0	0	0	0	0	2	0	0
JRL	0	1	0	0	0	0	2	0	1

[표1] TSC instruction 별 control signal의 값. ALU instructions에는 ADD, SUB, AND, ORR, NOT, TCP, SHL, SHR이 포함된다. Branch instructions에는 BNE, BEQ, BGZ, BLZ가 포함된다.

2.1. CPU 모듈

CPU 모듈의 input은 메모리의 쓰기 완료와 읽기 대기를 나타내는 두 신호 ackOutput과 inputReady, reset signal(reset_n), clock signal(clk)로 이루어져 있고, output으로는 메모리에 쓰기 및 읽기 신호를 주는 readM과 writeM, 그리고 읽고 싶은 메모리의 주소 address로 이루어져있고, input과 output으로 모두 사용할 수 있는 data가 있다. CPU 모듈은 하나의 instruction을 하나의 clock cycle마다 처리한다.

메모리로부터 읽어온 instruction을 register 변수에 저장하고, 다른 서브 모듈들에 적절한 input과 output을 연결한다.

2.2. memory_access 모듈

memory_access 모듈은 readM, writeM, address, data의 값을 실질적으로 수정하는 모듈로, clock의 posedge에서는 instruction을 읽어오고 clock의 negedge에서는 instruction에 따라 메모리의 데이터를 읽어오거나, 메모리에 데이터를 쓰는 역할을 한다. input은 control_unit에서 생성한 mem_read, mem_write control signal들과 접근할 주소와 쓰기를 위한 데이터, 그리고 clock signal(clk)가 필요하고, output으로 메모리에 접근할 때 필요한 값인 readM, writeM, address, data가 연결된다.

이 때 CPU 모듈에서 data가 inout으로 선언되어있기 때문에, 데이터를 읽고 싶을 땐 그 값을 Z(High impedance)로 설정해야 하는 것을 주의해야 한다.

clock		readM	writeM	address	data
posedge		1	0	PC	Z(High impedance)
negedge	LWD	1	0	읽고 싶은 주소	Z(High impedance)
	SWD	0	1	쓰고 싶은 주소	쓰고 싶은 데이터

[표2] clock에 따라 memory_access에서 지정해야 할 output의 값. posedge의 경우 instruction을 가져오기 위해 readM을 1로 설정하고, address는 PC로 설정한다. negedge의 경우 instruction에 따라 메모리를 읽거나 쓰기 위한 값으로 설정한다.

2.3. control_unit 모듈

control_unit 모듈은 memory_access에서 읽어온 instruction을 바탕으로 control signal들을 생성하는 combinational logic으로 구성된다. input으로는 instruction의 opcode와 function을 받아와 output으로 alu_src, reg_write, rt_write, mem_read, mem_to_reg, jp, branch, pc_to_reg의 control signal들을 가진다. instruction에 따른 control signal의 값은 표1에 있다.

2.4. alu_control_unit 모듈

alu_control_unit 모듈은 input으로 memory_access에서 읽어온 instruction의 function 부분을 받아와 output으로 적절한 alu_func_code를 생성하는 control logic으로 구성된다. 설계하는 단계에서는 ALU instruction들의 alu_func_code인 0~7까지의 값과 JPR, JRL의 alu_func_code인 25, 26을 그대로 사용하는 것으로 구상했고, 실제 구현을 하면서 필요한 alu_func_code를 추가하거나 변경하기로 했다.

2.5. register_file 모듈

register_file 모듈은 input으로 읽을 register의 번호 2개(read1, read2)와 쓸 register의 번호(write_reg)와 변경할 데이터(write_data), 레지스터 쓰기를 assert할 수 있는 control signal reg_write와 clock signal(clk)을 받아온다. 이후 읽어온 값을 output으로 보내주고(read_out1, read_out2), reg_write가 assert된 경우 해당 레지스터의 값을 변경한다.

TSC CPU는 16비트짜리 register 4개를 사용하므로, register_file 내부에서 reg array를 선언하여 각 레지스터의 데이터를 저장할 수 있어야 한다.

2.6. alu 모듈

alu 모듈은 input으로 연산을 할 두 수(alu_input_1, alu_input_2)와 alu_control_unit 모듈의 output인 alu_func_code를 받아 output으로 연산의 결과(alu_output)와 그 결과가 0인지(zero)를 보내는 combinational logic으로 구성된다. alu_func_code에 따라 덧셈이나 뺄셈 등의 적절한 연산을 수행해야 한다.

2.7. immediate_generator 모듈

immediate_generator 모듈은 input으로 instruction의 immediate 부분을 받아 output으로 sign-extended 혹은 zero-extended된 immediate값을 보내는 combinational logic으로 구성된다. instruction에 따라 output immediate의 총 길이와 extension 방법이 달라지므로 instruction의 일부를 받거나 control signal을 받아야 한다.

2.8. mux2to1, mux4to1 모듈

각각 2 to 1 멀티플렉서와 4 to 1 멀티플렉서를 구현한다. 입력 2개 또는 4개와 select 값을 input으로 받아 select값에 따른 입력값을 output으로 한다. 중복하여 많이 사용될 것 같아 모듈화를 하기로 결정했다.

3. Implementation

3.1. CPU 모듈

```
// get data from memory
always @(*) begin
        // instruction fetch
        if (readM && inputReady && instrFetch)
                instruction = data;
        else
                instruction = instruction;

        // memory data fetch
        if (readM && inputReady && memAccess)
                memory_data = data;
        else
                memory_data = memory_data;
end
```

[그림 2] CPU 모듈의 코드 중 일부. 메모리의 값을 읽어들이고 있음을 나타내는 readM과 inputReady가 모두 1이면 모듈의 inout 포트인 data의 값을 읽어 instruction 또는 memory_data에 그 값을 저장한다.

CPU 모듈에는 각 서브 모듈의 input과 output을 전달할 수 있는 각종 wire 변수와 메모리에서 읽어온 데이터를 저장하는 reg 변수 instruction과 memory_data, 그리고 각 서브 모듈의 인스턴스들이 선언되어 있다. readM과 inputReady가 모두 1이면 메모리에서 값을 읽어들이고 있으며, 그 값이 안정화되었음을 의미한다. 이 때 instrFetch 값이 1이면 instruction 변수에 읽어온 데이터를 저장하고, memAccess 값이 1이면 memory_data 변수에 읽어온 데이터를 저장한다.

이렇게 구현된 이유는 instruction memory와 data memory가 하나의 메모리로 사용되고 있고 일정 시간의 딜레이 후에 값을 읽어올 수 있기 때문에, 딜레이가 지난 뒤 읽어온 데이터가 instruction인지 데이터인지 구분할 필요가 있기 때문이다. 읽어온 데이터의 종류를 구분하는 데 사용되는 instrFetch와 memAccess는 memory_access 모듈에서 그 값을 변경한다.

3.2. memory_access 모듈

```
        reg [`WORD_SIZE-1:0] temp_data;

        assign data = temp_data;


// update pc
always @(posedge clk) begin
        if (!reset_n) begin
                // reset all states.
                pc <= 0;
                pc_nxt <= 0;
        end
        else begin
                pc_nxt <= pc_update + 1;
                pc <= pc_update;
        end
end

// instruction fetch
always @(posedge clk) begin
        readM <= 1;
        writeM <= 0;
        address <= pc_update;
        temp_data <= `WORD_SIZE'bz;

        instrFetch <= 1;
        memAccess <= 0;
end
```

[그림 3] memory_access 모듈의 코드 중 일부. pc값을 업데이트하고 instruction을 가져온다.

memory_access 모듈은 posedge에는 pc를 업데이트하고 instruction을 fetch해오며, negedge에는 메모리를 읽거나 쓴다. 그림 3은 posedge일 때 작동하는 코드를 보여준다. data에 적절한 값을 넣어주기 위해 reg 변수인 temp_data를 선언하여 data에 temp_data값을 assign한다. 이 temp_data의 값은 always 구문에서 변경된다.

첫 번째 always 구문에서는 pc값을 업데이트한다. reset signal(reset_n)이 들어오면 pc와 pc_nxt 모두 0으로 초기화시킨다. 그게 아니면 pc는 pc_update의 값으로, pc_nxt는 pc_update + 1의 값으로 업데이트한다. 이 때 처음 설계에서는 adder를 사용하도록 했는데, 실제 구현할 땐 Verilog의 add operation을 바로 사용했다.
두 번째 always 구문에서는 instruction 값을 가져오기 위해 readM을 1로, address는 pc값으로, temp_data는 Z로 설정한다. 또한 instrFetch값을 1로 변경하여 cpu 모듈에서 memory의 데이터를 읽을 때 instruction에 저장하도록 한다.

```
// read or write memory
always @(negedge clk) begin
        readM <= mem_read;
        writeM <= mem_write;
        address <= mem_address;
        temp_data <= mem_write? mem_data: `WORD_SIZE'bz;

        instrFetch <= 0;
        memAccess <= 1;
end
```

[그림 4] memory_access 모듈의 코드 중 일부. 메모리에서 데이터를 읽어오거나 값을 변경한다.

그림 4는 negedge일 때 작동하는 코드를 보여준다. mem_read, mem_write는 control_unit에서 생성한 control signal로, 이 값들을 readM과 writeM에 넣어준다. mem_address와 mem_data는 접근하고 싶은 메모리의 주소와 만약 메모리에 쓰는 instruction일 경우 쓰고 싶은 값을 나타낸다. mem_address는 address에 넣고, mem_data는 mem_write가 1일 때만 temp_data에 넣어주고, 그게 아니면 Z를 넣어준다. 또한 memAccess값을 1로 변경하여 cpu 모듈에서 memory의 데이터를 읽을 때 memory_data에 저장하도록 한다.

```
// update writeM to 0 after writing is done
always @(ackOutput) begin
        if (ackOutput == 1) begin
                writeM = 0;
        end
        else begin
                writeM = writeM;
        end
end

// update readM to 0 after reading is done
always @(inputReady) begin
        if (inputReady == 1) begin
                readM = 0;
        end
        else begin
                readM = readM;
        end
end
```

[그림 5] memory_access 모듈의 코드 중 일부. 메모리에서 값을 읽어들이고 있음을 나타내는 inputReady와 값을 쓰고 있음을 나타내는 ackOutput의 값이 1로 바뀌면 readM과 writeM의 값을 0으로 바꾼다.

메모리에서 값을 읽어오는 중에는 inputReady가 1로, 값을 쓰고 있는 중에는 ackOutput이 1이 되고, 각 수행이 완료되면 값이 모두 0으로 바뀐다. 그에 따라 readM과 writeM도 한 cycle에 두 번 읽거나 쓰지 않도록 0으로 바꿔준다.

3.3. control_unit 모듈

```
always @(*) begin
    // reset each control value;s
    alu_src = 0;
    reg_write = 0;
    mem_read = 0;
    mem_to_reg = 0;
    mem_write = 0;
    jp = 0;
    branch = 0;
    pc_to_reg = 0;
    rt_write = 0;
    case (opcode)
        `ALU_OP, `JPR_OP, `JRL_OP: begin      // R-type Instructions
            case (func_code)
                `INST_FUNC_JPR: begin         // JPR
                    jp = 2;
                end
                `INST_FUNC_JRL: begin         // JRL
                    reg_write = 2;
                    jp = 2;
                    pc_to_reg = 1;
                end
                default: begin                // ALU operations
                    reg_write = 1;
                end
            endcase
        end
        `ADI_OP, `ORI_OP, `LHI_OP: begin
            alu_src = 1;
            reg_write = 1;
            rt_write = 1;
        end
```

[그림 6] control_unit 모듈의 코드 중 일부. instruction의 opcode와 func_code에 따라 적절한 control signal을 생성한다.

그림 6은 control_unit 모듈의 memory_access에서 읽어온 instruction을 바탕으로 control signal들을 생성하는 combinational logic을 나타낸다. 이 때 ALU instruction들과 JPR, JRL instruction은 opcode가 15로 동일하기 때문에 instruction의 function(func_code) 값도 확인해서 control signal을 생성한다. 그림 6에서는 두 가지 케이스만 나타냈지만 실제 코드에는 각 instruction의 opcode와 func_code에 따라 각각 다른 control signal을 생성한다. 각 instruction에 따른 control signal의 값은 표1을 따른다.

3.4. alu_control_unit 모듈

```
always @(*) begin
    case (opcode)
        `ALU_OP: begin
            case(func_code)
                `INST_FUNC_JPR: alu_func_code = `FUNC_IP1; // pc <- rs
                `INST_FUNC_JRL: alu_func_code = `FUNC_IP1; // pc <- rs
                default: alu_func_code = func_code[3:0];
            endcase
        end
        `ADI_OP: alu_func_code = `FUNC_ADD;
        `ORI_OP: alu_func_code = `FUNC_ORR;
        `LHI_OP: alu_func_code = `FUNC_IP2; // immediate : alu_input_2
        `LWD_OP: alu_func_code = `FUNC_ADD;
        `SWD_OP: alu_func_code = `FUNC_ADD;
        `BNE_OP: alu_func_code = `FUNC_BNE; // Zero if (input1 != input2)
        `BEQ_OP: alu_func_code = `FUNC_SUB; // Zero if (input1 == input2)
        `BGZ_OP: alu_func_code = `FUNC_BGZ; // Zero if (input1 != 0 and positive)
        `BLZ_OP: alu_func_code = `FUNC_BLZ; // Zero if (input1 is negative)
        default: alu_func_code = 4'd15;     // Don't use ALU
    endcase
end
```

[그림 7] alu_control_unit 모듈의 일부. Instruction의 opcode와 func_code에 따라 적절한 alu_func_code를 생성해 출력한다.

그림 7은 alu_control_unit 모듈의 instruction의 opcode와 function(func_code)에 따라 적절한 alu_func_code를 생성하는 combinational logic을 나타낸다. 설계에서 func_code 부분만 받아왔던 것과는 다르게, ALU, JPR, JRL instruction 뿐만 아니라 다른 instruction들도 ALU를 필요로 하는 경우가 많아서 opcode까지 받아오도록 구현했다. 또한 instruction의 func_code는 3비트였던 것에 비해, instruction의 동작에 따라 branch에 필요한 조건 연산과 input중 하나를 그대로 output해주는 연산을 할 수 있도록 4비트로 확장된 alu_func_code를 output으로 가진다.

3.5. register_file 모듈

```
reg [15:0] RF [3:0]; // 4 registers each 16 bits long

initial begin
    RF[0] = 16'b0;
    RF[1] = 16'b0;
    RF[2] = 16'b0;
    RF[3] = 16'b0;
end

assign read_out1 = RF[read1];
assign read_out2 = RF[read2];

always @(posedge clk) begin
    // write back if reg_write is high
    if (reg_write) RF[write_reg] <= write_data;
    else RF[write_reg] <= RF[write_reg];
end
```

[그림 8] register_file 모듈의 일부. 16비트짜리 reg 변수 4개가 있는 array를 선언하고 있으며, input에 따라 적절한 register의 값을 출력하고 변경한다.

register_file 모듈은 16비트짜리 register 4개를 사용하여 데이터를 저장하고 있으며, input의 read1, read2에 해당하는 register값을 읽어 output해주는 동시에, reg_write가 assert된 경우 posedge clk에 synchronous하게 해당 레지스터의 값을 변경한다.

3.6. alu 모듈

```
assign zero = (alu_output == 0);

always @(*) begin
        case(alu_func_code)
                `FUNC_ADD: alu_output = alu_input_1 + alu_input_2;
                `FUNC_SUB: alu_output = alu_input_1 - alu_input_2;
                `FUNC_AND: alu_output = alu_input_1 & alu_input_2;
                `FUNC_ORR: alu_output = alu_input_1 | alu_input_2;
                `FUNC_NOT: alu_output = ~alu_input_1;
                `FUNC_TCP: alu_output = ~alu_input_1 + 1;
                `FUNC_SHL: alu_output = {alu_input_1[14:0], 1'b0};
                `FUNC_SHR: alu_output = {alu_input_1[15], alu_input_1[15:1]};
                `FUNC_IP1: alu_output = alu_input_1;
                `FUNC_IP2: alu_output = alu_input_2;
                `FUNC_BNE: alu_output = (alu_input_1 != alu_input_2)? 0 : 1; // zero if not equal
                `FUNC_BGZ: alu_output = (alu_input_1 > 0)? 0 : 1; // zero if alu_input_1 > 0
                `FUNC_BLZ: alu_output = (alu_input_1 < 0)? 0 : 1; // zero if alu_input_1 < 0
               default: alu_output = 0; // not happen
        endcase
end
```

[그림 9] alu 모듈의 일부. alu_func_code에 따라 두 input에 대해 적절한 연산을 수행하여 그 결과를 반환한다.

alu 모듈은 alu_control_unit에서 생성한 alu_func_code에 따라 적절한 연산을 수행하여 output해준다. 연산결과인 alu_output이 0이 될 경우에는 zero를 assert하여 branch 연산에서 사용할 수 있도록 해준다.

alu 모듈은 cpu 모듈에서 인스턴스화될 때 adder로도 사용되었다. adder로 사용될 경우엔 alu_func_code에 `FUNC_ADD를 하드코딩해 사용했다.

3.7. immediate_generator 모듈

```
always @(*) begin
    case (opcode)
        `ADI_OP: immediate <= {{8{imm[7]}}, imm[7:0]};
        `ORI_OP: immediate <= {8'b0, imm[7:0]};
        `LHI_OP: immediate <= {imm[7:0], 8'b0};
        `LWD_OP: immediate <= {{8{imm[7]}}, imm[7:0]};
        `SWD_OP: immediate <= {{8{imm[7]}}, imm[7:0]};
        `BNE_OP: immediate <= {{8{imm[7]}}, imm[7:0]};
        `BEQ_OP: immediate <= {{8{imm[7]}}, imm[7:0]};
        `BGZ_OP: immediate <= {{8{imm[7]}}, imm[7:0]};
        `BLZ_OP: immediate <= {{8{imm[7]}}, imm[7:0]};
       default: immediate <= 16'b0; // not happen
    endcase
end
```

[그림 10] immediate_generator 모듈의 일부. Instruction의 opcode에 따라 적절한 immediate값을 생성한다.

immediate_generator 모듈은 input으로 instruction의 opcode와 immediate 부분을 받아 opcode에 맞는 형식으로 immediate를 적절하게 manipulate시킨 후 output해주는 combinational logic이다. 처음 설계에서 immediate_generator의 control signal을 모호하게 설계했는데, 구현할 때 instruction에서 immediate 값을 받아와야하기 때문에 아예 instruction 전체를 input으로 받아와 opcode와 immediate를 가져오고, opcode로부터 immediate의 extension 방식을 정하도록 했다. ORI의 경우 zero-extend, LHI의 경우에는 most significant halfword가 되도록, 나머지 I format instruction에서는 sign-extend를 시켜준다. 그 외 immediate가 사용되지 않을 때는 0을 output해준다.

3.8. mux2to1, mux4to1 모듈

```
module mux2to1 #(parameter DATA_WIDTH = 16) (in1, in2, sel, out);
    input [DATA_WIDTH-1:0] in1, in2;
    input sel;
    output reg [DATA_WIDTH-1:0] out;

    always @(*) begin
        case(sel)
            1: out = in2;
            default: out = in1;
        endcase
    end
endmodule

module mux4to1 #(parameter DATA_WIDTH = 16) (in1, in2, in3, in4, sel, out);
    input [DATA_WIDTH-1:0] in1, in2, in3, in4;
    input [1:0] sel;
    output reg [DATA_WIDTH-1:0] out;

    always @(*) begin
        case(sel)
            1: out = in2;
            2: out = in3;
            3: out = in4;
            default: out = in1;
        endcase
    end
endmodule
```

[그림 11] mux2to1, mux4to1 모듈. input인 sel 값에 따라 여러 입력 중 하나의 값을 output인 out으로 보낸다.

멀티플렉서는 중복하여 많이 사용될 수 있도록 DATA_WIDTH parameter를 사용하여 여러종류의 bit수를 사용할 수 있도록 구현하였다. 신호가 없을 때(0) 첫번째 input을 output하도록 구현하여 안정성을 높였다.

```
mux4to1 #(.DATA_WIDTH(2)) MUX_rt_write(
        .in1(instruction[7:6]),
        .in2(instruction[9:8]),
        .in3(2'b10),
        .in4(2'b10),
        .sel(rt_write),
        .out(MUX_rt_write_out)
);
```

[그림 12] cpu 모듈에서 mux4to1의 인스턴스 중 하나. rt_write control signal을 받아 input 3개 중 하나를 고르는 로직으로, in3와 in4에 같은 값을 넣어줬다.

4 to 1 멀티플렉서의 경우 설계상 3개의 input만 쓰는데, 이런 경우 세 번째 input과 네 번째 input을 동일하게 설정해줬다.

3.9. Next PC를 결정하는 로직

```
alu Adder(
        .alu_input_1(pc_nxt),
        .alu_input_2(immediate),
        .alu_func_code(`FUNC_ADD),
        .alu_output(adder_result),
        .zero()
);

mux2to1 MUX_branch_high(
        .in1(pc_nxt),
        .in2(adder_result),
        .sel(branch & zero),
        .out(MUX_branch_high_out)
);

mux4to1 MUX_jp(
        .in1(MUX_branch_high_out),
        .in2({pc[15:12], instruction[11:0]}),
        .in3(alu_result),
        .in4(alu_result),
        .sel(jp),
        .out(MUX_jp_out)
);
```

[그림 13] cpu 모듈에서 branch, jump instruction의 동작을 구현하는 데 사용된 Adder와 mux2to1, mux4to1 모듈.

PC의 다음 값은 branch instruction과 jump instruction에 의해 달라질 수 있다. 설계와 달라진 부분은 Adder의 한 input으로 pc가 아닌 pc + 1(pc_nxt)에 해당하는 값을 넣어주는 부분이다. 이는 TSC instruction set에서 branch instruction를 수행할 때 pc에 pc + offset + 1 값을 넣어주기 때문이다.

mux2to1의 인스턴스인 branch_high에서 control signal인 branch와 ALU의 output인 zero로부터 pc_nxt, adder_result의 값 중 하나를 output으로 결정하고, mux4to1의 인스턴스인 MUX_jp에서 control signal인 jp를 받아 branch_high 멀티플렉서의 output과 target값({pc[15:12], instruction[11:0]}), 그리고 ALU의 결과값($rs + offset) 중 하나로 output을 결정한다. 이 때 target값의 경우 설계할 때는 instruction의 immediate값을 바로 가져오는 것으로 설계했는데, 구현 과정에서 매뉴얼을 확인하니 pc[15:12]의 비트와 합치도록 되어있어서 이 부분을 수정했다. 최종적으로 MUX_jp의 output이 next pc의 값이 된다.

4. Discussion

- 메모리의 데이터를 받아오고 쓸 때 사용했던 inout port의 사용이 어려웠다. 기존까지 input이거나 output이었던 단방향 데이터 전송만 가능했던 포트들을 사용하다가 양방향으로 값을 읽어올 수 있다는 것을 이해하는 데 시간이 걸렸다. 구현하는 과정에서 memory write를 하는 instruction에서 data에 제대로된 값을 넣어주고 있었는데 갑자기 값이 X로 바뀌는 버그가 있었는데, inout 포트에 대한 이해가 얕은 상태에서는 버그의 원인을 찾기 어려웠었다. 이는 inout 포트로 data를 보내주고 있을 때 메모리 읽기가 다시 한 번 일어나서 발생한 것으로 확인했고, readM을 inputReady가 1이 되자마자 0으로 바꿔줌으로써 메모리 읽기가 다시 발생하지 않도록 조정하여 해결할 수 있었다.

- instruction memory와 data memory가 하나의 메모리로 사용되고 있고 일정 시간의 딜레이 후에 값을 읽어올 수 있기 때문에, 딜레이가 지난 뒤 읽어온 데이터가 instruction인지 데이터인지 구분할 필요가 있었다. instrFetch, memAccess state를 만들고 그에 따라 reg 변수 instruction과 memory_data에 구분하여 저장함으로써 문제를 해결했다.

- 처음에 설계 단계에서 RISC-V와 다르게 rd가 아니라 rt 레지스터의 값을 변경하는 instruction들이 있어서 어떻게 설계할지 고민했었다. 이는 멀티플렉서와 control line(rt_write)의 추가로 해결했다.

5. Conclusion

TSC Single Cycle CPU 구현을 통해 instruction set으로부터 datapath와 control line을 설계하는 과정을 통해 각 instruction마다 사용되는 하위 module들의 역할과 적절한 control의 사용, synchronous하게 동작하는 Single Cycle CPU에 대해 이해할 수 있었다.