Lab 3. Single Cycle CPU

CSED311 Computer Architecture

20190065 강두경, 20190316 유병호

1. Introduction

이번 랩의 목표는 single cycle cpu를 구현하는 것이다. 테스트 벤치에서 이미 instruction memory와 memory는 구현되어 있기 때문에 이번 랩의 CPU는 memory의 instruction를 input으로 cpu에서 받아와 적절한 연산을 수행해주고, 다음 instruction memory를 output으로 보내주어 다음 instruction을 수행하는 방식으로 동작해야 한다.

추가적으로 load의 경우에는 instruction을 읽어온 후 주어진 메모리 address를 갖는 데이터를 가져와야 하며, store의 경우에는 저장해야 되는 데이터와 데이터를 저장할 메모리의 address를 output으로 줘야 한다.

single cycle cpu를 이번 랩에 구현하면서 반드시 필요한 모듈은 다음과 같다.

1) control_unit (single cycle cpu에 포함되어 있는 모듈들의 input과 output을 컨트롤하는 비트를 생성)

2) resiter_file (이번 랩에선 4개의 resister를 사용한다.)

3) ALU (resister 및 pc 등의 값을 이용해 instruction에 따라 각기 다른 연산 결과를 보내줌.)

4) cpu (cpu를 구성하는 모든 모듈을 포함하는 마더 모듈. output에 대한 control도 진행해준다.)

2. Design

일반적으로 수업 시간에 배웠던 RISC-V Single Cycle CPU의 구조를 참고했으나, 명령어 구조의 차이 등을 감안하여 많은 부분을 수정해야 했다.

2.1. Program Counter FSM

우선 현재 인스트럭션 주소를 가리키는 PC의 경우 주소를 저장하는 별도의 레지스터가 필요했다. 초기값은 0이며, 메모리부터 인스트럭션 주소에 해당하는 인스트럭션을 읽어온다. 그러면 이 인스트럭션에 따라 다음 인스트럭션 주소를 Combinational Logic으로 계산하는 FSM을 디자인했다. 끝으로 clk이 0에서

1이 될 때, 즉 posedge clk 조건일 때 현재 인스트럭션 주소를 다음 인스트럭션 주소로 바꿔주는 Sequential Logic을 구현한다. instruction은 data를 읽어왔을 때, 즉 inputReady가 1이 되었을 때 data로 바꿔준다.

다음 instruction 주소는 기본적으로 인스트럭션의 opcode와 funccode에 의존한다. JRL이나 JAR일 경우 rs 레지스터의 값을, JAL이나 JMP의 경우 immediate 값을, 나머지의 경우 현재 인스트럭션 주소 + 1의 값을 가지며, 여기에 branch 조건일 경우 immediate 값을 더한 만큼의 값을 갖는다.

2.2. Control Unit

인스트럭션은 기본적으로 opcode, funccode에 따라 동작이 결정된다. control bit를 통해 특정 연산, 레지스터의 값을 쓸 지 상수값을 사용할 지 등등의 분기점을 결정하며 이 역할을 해 주는 모듈이 Control Unit이다. Control Bit에는 alu_src, reg_write, mem_read, mem_write, mem_to_reg, jp, jpr, branch, pc_to_reg의 9종류가 있다. 각 control bit의 역할 및 해당하는 instruction은 다음과 같다.

종류	설명	해당하는 Instruction
alu_src	ALU에서 계산한 값 대신 Immediate 값을 output으로 사용	ADI, ORI, LHI, LWD, SWD, JMP, JAL
reg_write	Register에 output 값을 저장함	모든 R type, ORI, ADI, LHI
mem_read	메모리로부터 값을 읽어들임	LWD
mem_write	메모리에 값을 저장함	SWD
mem_to_reg	메모리에 있는 값을 레지스터로 저장함	LWD
jp	immediate 값으로 jump함	JMP, JAL
jpr	register 값으로 jump 함	JPR, JRL
branch	특정 조건을 만족하면 instruction address 이동	BNE, BEQ, BLZ, BGZ
pc_to_reg	현재 PC 값을 Register에 저장=	JAL, JRL

2.3. Register File

Register File은 레지스터의 읽고 쓰기를 담당한다. 내부에 레지스터 역할을 하는 register가 있으며, reg_write bit가 1일 때 register을 write해주는 sequential logic을 구현한다. register 쓰기는 인스트럭션이 로드된 이후에 해야 하기 때문에, clk 값을 CPU의 inputReady와 연결하였다. inputReady가 켜지면 posedge 조건에서 입력받은 write 레지스터 번호의 값을 write data 값으로 바꿔준다.

2.4. ALU

ALU는 기본적으로 opcode가 ALU일 때, funccode를 받아와서 해당하는 연산을 처리하는 모듈이다. 하지만 본 lab에서는 immediate instruction일 때의 연산과 branch 연산까지 모두 한 ALU 모듈에 구현하

였다.

우선 opcode가 ALU_OP일 때는 다음 behavior를 따른다. ALU의 두 input을 a, b이라 하고 output을 y라 할 때,

FuncCode	Behavior
ADD	`y = a + b`
SUB	`y = a - b`
AND	`y = a & b`
ORR	`y = a | b`
NOT	`y = ~a - 1`
TCP	`y = ~a + 1`
SHL	`y = a << 1`
SHR	`y = a >> 1`

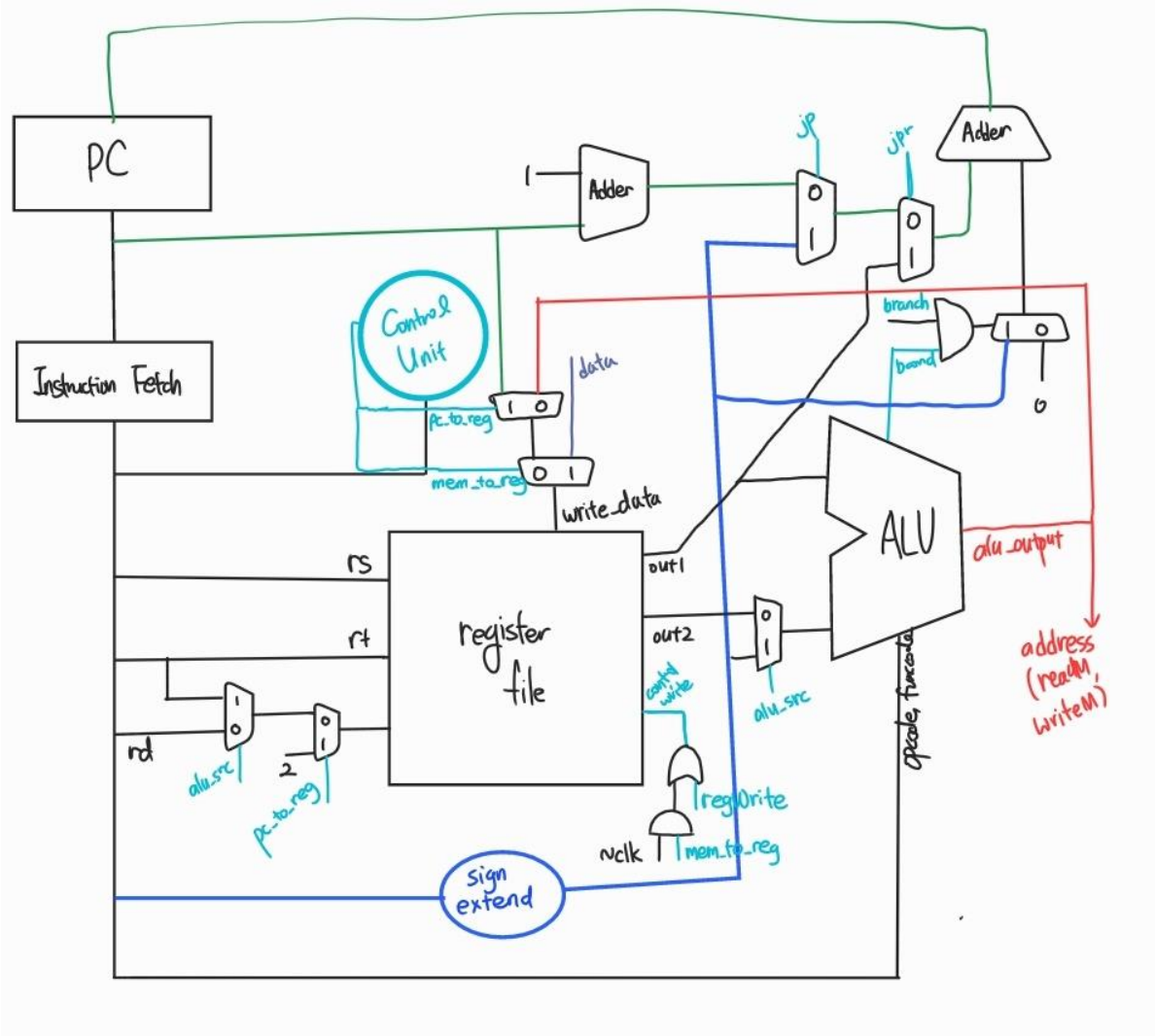
opcode가 ALU가 아닐 때는 다음 behavior을 따른다. 마찬가지로 ALU의 두 input을 a, b라 하고 output을 y라 하자. 또한 branch instruction일 때 condition code를 c라 하자.

FuncCode	Behavior
ADI	y = a + b
ORI	y = a | b
LHI	y = b << 8
LWD	y = a + b
SWD	y = a + b
BNE	c = a != b
BEQ	c = (a == b)
BGZ	c = a > 0
BLZ	c = a < 0

2.5. CPU

CPU에서는 위의 모듈을 이용해 어떤 instruction이 들어왔을 때 이를 처리하는 역할을 한다. register에 써 주는 값, 비트, ALU 처리 방법, ALU output 등은 모두 control bit에 의존한다.

Memory 접근 부분을 제외하고 combinational logic을 회로로 구현한 그림은 다음과 같다.



Register File의 경우, Instruction으로부터 2비트를 얻어 읽을 레지스터 번호와 쓸 레지스터 번호를 입력한다. 하나의 인스트럭션은 최대 3개의 instruction 번호를 포함한다. rs는 instruction의 11번째와 10번째 비트를 가리키고, rt는 instruction의 9번째와 8번째 비트를 가리킨다. rd는 instruction의 7번째와 6번째 비트를 가리키는데 이는 R-type에만 존재한다.

쓸 레지스터 번호는 instruction의 종류에 따라 조금 다른데, R-type ALU 연산이면 rd에, I-type 연산이면 rt에, PC to reg(JRL, JAL)일 경우 2번 레지스터에 저장한다.

regsiter file 모듈은 쓸 데이터 값을 입력받아야 한다. ALU의 값을 저장하는 경우, PC 값을 저장하는 경우, 메모리 값을 저장하는 경우가 다르며 각각을 control bit에 따라 mux로 처리하여 입력한다.

register file은 레지스터 입력 control 값이 1일 때만 레지스터 값을 바꾼다. reg_write control bit이 1일 경우 바로 바꾸는데, mem_to_reg가 1일 경우에는 clk이 0일 때만 바꾸도록 한다. 그 이유는 mem_to_reg의 경우 메모리 값을 읽어들여야 register 값을 바꿀 수 있는데, instruction이 아니라 lwd,

swd를 위해 따로 메모리를 접근할 때는 clk이 0일 때 메모리를 접근한다. 따라서 clk이 0일 때에만 inputReady가 켜질 때 입력값이 메모리에서 읽어들인 값이 되기 때문이다.

ALU의 경우 첫 번째 입력값은 첫 번째 레지스터 값이며, 두 번째 입력값은 R type ALU 연산일 때는 두 번째 레지스터 값을, immediate 연산일 경우에는 immediate 값을 넣는다.

instruction address의 다음 state 값을 jp, jpr, branch 컨트롤 비트에 의존한다. JP일 경우에는 imm 값으로, JPR instruction일 때는 첫 번째 register 값으로 점프한다. branch 이며 ALU에서 연산한 branch condition이 1일 경우 다음 레지스터 값 + immediate 값으로 이동한다. 모두 아닐 경우에는 다음 인스트럭션 주소, 즉 현재 인스트럭션 주소 + 1로 이동한다.

2.6. Sequential Logic

하지만 Combinational Logic만 가지고 컴퓨터를 만들 수는 없다. 특히 메모리를 읽고 쓸 때 시간이 걸리며, 한 명령어 당 한 싸이클을 소요하기 때문에 이 부분에 대해서 Sequential Logic으로 처리를 해 주어야 한다.

우선 clk이 0에서 1이 될 때 새로운 instruction을 읽어들이기 시작한다. instruction_address state를 다음 instruction_address로 바꿔주고, address에 instruction_address을 assign한 다음 readM을 1로 바꾼다. 그렇게 되면 일정 시간이 지난 후 inputReady가 켜지면서 data 값이 high-Z에서 메모리 값으로 바뀌게 된다. CPU에서는 inputReady를 감지하여 0에서 1이 될 때 instruction 값에 data를 저장한다. 이 때 control bit, alu 연산이 끝나고 register도 write할 수 있다.

memory에서 읽어들여서 inputReady가 1이 되었다가 0이 될 때까지 걸리는 시간이 반 사이클이 조금 안 된다. 여기까지는 LWD, SWD를 제외한 모든 인스트럭션을 다 처리할 수 있다. LWD, SWD의 경우는 남은 반 cycle 동안 메모리를 한 번 더 접근해야 한다.

SWD의 경우 clk이 0이 되는 negedge 조건에 data에 immediate 값을, address에 alu_output을 연결하고 writeM을 켜 줘야 한다. 그러면 일정 시간이 지난 후 반 사이클 안에 해당 메모리 안에 값이 저장된다. 이 때가 ackOutput이 1이 될 때이며 writeM을 다시 0으로 설정해주면 된다.

LWD의 경우에 clk이 0이 되는 negedge 조건에 address에 alu_output을 연결하고 readM을 켜 줘야 한다. 그러면 일정 시간이 지난 후 해당 메모리 값이 data로 들어오며 이 때 readM을 0으로 설정하고 받아들인 값을 레지스터에 저장한다.

Implementation

sign_extender.v

8비트의 immediate를 받아와서 16비트크기로 sign extend시켜주는 모듈이다. immediate 값을 처음 읽

을 때는 8비트인데, 대부분의 연산을 16비트로 처리하기 때문에 sign extend 시켜줘야 한다.

```
module sign_extender(in, out);
input wire [7:0] in;
output wire [15:0] out;
    begin
        assign out[7:0] = in[7:0];
        assign out[15:8] = in[7] ? 8'b11111111 : 8'b0;
    end
endmodule
```

sign extender의 코드는 위와 같다. 일반적으로 LSB부터 복사하며, 8번째 비트가 0이면 MSB부터 0으로 채우고, 1이면 1로 채운다.

---

control_unit

주어진 instruction에 따라서 cpu내의 모듈이 작동하는 방식, 모듈의 input과 output을 모두 컨트롤하는 control bit들을 만들어주는 모듈이다.

< input / output >

```
module control_unit
  input [`WORD_SIZE-1:0] instr;          test bench 의 memory 에서 받아온 instruction.
  output reg alu_src;
  output reg reg_write;
  output reg mem_read;
  output reg mem_to_reg;
  output reg mem_write;
  output reg jp;
  output reg jpr;
  output reg pc_to_reg;
  output reg branch;
  input reset_n;
```

종류	설명	해당하는 Instruction
alu_src	ALU에서 계산한 값 대신 Immediate 값을 output으로 사용	ADI, ORI, LHI, LWD, SWD, JMP, JAL
reg_write	Register에 output 값을 저장함	모든 R type, ORI, ADI, LHI
mem_read	메모리로부터 값을 읽어들임	LWD
mem_write	메모리에 값을 저장함	SWD

mem_to_reg	메모리에 있는 값을 레지스터로 저장함	LWD
jp	immediate 값으로 jump함	JMP, JAL
jpr	register 값으로 jump 함	JPR, JRL
branch	특정 조건을 만족하면 instruction address 이동	BNE, BEQ, BLZ, BGZ
pc_to_reg	현재 PC 값을 Register에 저장=	JAL, JRL

```
assign opcode = instr[15:12];
assign funccode = instr[5:0];
```

먼저 opcode와 funccode의 값에 따라서 control_bit들이 달라지기 때문에 instruction의 opcode와 funccode를 저장할 수 있는 레지스터를 모듈 내에 선언하고 적절히 instr을 파싱해서 저장해준다.

```
initial begin
    alu_src = 0;
    mem_read = 0;
    mem_write = 0;
    mem_to_reg = 0;
    reg_write = 0;
    jp = 0;
    jpr = 0;
    branch = 0;
    pc_to_reg = 0;
end
```

cpu가 처음으로 동작할 때는 initial begin을 통해 모든 output을 0으로 초기화 한다.

```
always @(instr) begin
    if (!reset_n) begin     // reset_n 이 0 일 경우 output 들의 값을 0 으로 초기화.
        alu_src = 0; ……
```

cpu 내의 control bit들은 cpu가 수행해야하는 instruction 이 바뀔 때마다 새롭게 업데이트 된다.

그렇기 때문에 combinational logic을 이용해 instr(instruction)이 바뀔 때마다 바뀐 opcode와 funccode에 알맞도록 control_bit들을 설정해준다.

example)

```
    else begin
```

```
    alu_src =                       // alu_src 를 1 로 만들어주어야 하는 op_code 와
(opcode == `ADI_OP) ||    // input 으로 들어온 opcode 가 같은 경우를 or
(opcode == `ORI_OP) ||    // 로 연결하여 alu_src 의 값을 정해준다.
(opcode == `LHI_OP) ||
(opcode == `LWD_OP) ||
(opcode == `SWD_OP) ||
(opcode == `JMP_OP) ||
(opcode == `JAL_OP);  ..............
```

register_file.v

cpu에서 사용하는 register를 저장하고 있는 모듈이다.

이번 랩에서는 TSC_CPU 구현이 목적이기 때문에 16비트짜리 4개의 레지스터를 구현해야한다.

<input / output>

```
output [15:0] read_out1;
output [15:0] read_out2;
input [1:0] read1;
input [1:0] read2;
input [1:0] write_reg;
input [15:0] write_data;
input reg_write;
input clk;
input reset_n;
```

read1, read2 : 데이터 값을 읽어야 하는 레지스터의 번호. index.

reg_write : 레지스터의 데이터를 읽지않고, 레지스터에 데이터를 저장해야 할 때 1이 되는 control bit.

write_reg : reg_write가 1일 때 write_data를 적어야 하는 레지스터의 번호. index.

write_data : reg_wrtie가 1일 때 register[write_reg]에 저장해야하는 데이터.

read_out1, read_out2 : 리제스터에서 읽은 데이터.

```
reg [15:0] register [3:0];
```

16비트짜리 4개의 레지스터를 구현해야 하므로 다음과 같이 모듈 내에 regstier를 선언해주었다.

```
assign read_out1 = register[read1];
```

```
assign read_out2 = register[read2];
```

이렇게 만들어진 register들 중 read1과 read2에 해당하는 register의 데이터 값을 assign문을 이용해 read_out1과 read_out2에 할당해준다. 위와 같은 방식으로 register를 read할 수 있다.

```
always @(posedge clk or negedge reset_n) begin
  if (reset_n == 0) begin
    register[0] <= 16'b0;
    register[1] <= 16'b0;
    register[2] <= 16'b0;
    register[3] <= 16'b0;
  end
  else begin
    if (reg_write == 1'b1)
      register[write_reg] <= write_data;
  end

end
```

register read외에, register에 새롭게 데이터를 write하는 경우는 총 2가지가 있다.

1) reset_n == 0 이 될때, 레지스터의 값을 모두 0 으로 초기화한다.

2) reg_write가 1일 때, register[write_reg]<=write_data 해준다. 이러한 register write를 sequential logic을 이용해서 posedge clk와 negedge reset_n에 수행해준다.

alu.v

cpu에서

```
input [`NumBits-1:0] alu_input_1;
input [`NumBits-1:0] alu_input_2;
input [3:0] alu_op;
input [2:0] func_code;

output reg [`NumBits-1:0] alu_output;
output reg condition_bit;
```

alu_input_1, alu_input_2 : alu에서 산술연산과 logical 연산을 진행하기 위한 데이터.

alu_op, func_code : instruction의 alu_op와 func_code

alu_output : alu_input_1과 alu_input_2를 주어진 alu_op와 func_code에 대해 알맞은 연산을 해준 후의 결과값.

condition_bit : branch instruction에 해당하는 alu_op가 입력되었을 때, alu_intput_1과 alu_input_2을에 대해 알맞은 비교를 한 후 참이면 1을 거짓이면 0을 출력한다.

---

initial 에는 모든 output 값을 0으로 초기화 시켜준다.

이후에는 combinational logic을 이용해서 input의 값이 바뀔 때마다 output을 바꿔준다.

input으로 받아온 op_code가 `ALU_OP와 같을 경우에는 func_code를 조건으로 사용하는

case 문으로 주어진 func_code에 알맞은 alu_inptut1과 alu_input2를 계산해주어 alu_output에 할당해준다.

```
always @(*) begin
  if (alu_op == `ALU_OP) begin
    case(func_code)
    `FUNC_ADD: begin
    alu_output = alu_input_1 + alu_input_2;
    end
    `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 << 1;
    `FUNC_SHR: alu_output = alu_input_1 >> 1;
    default: alu_output = alu_output;
    endcase
  end
  else begin
    case(alu_op)
    `ADI_OP: alu_output = alu_input_1 + alu_input_2;
    `ORI_OP: alu_output = alu_input_1 | alu_input_2;
    `LHI_OP: alu_output = alu_input_2 << 8;
    `LWD_OP: alu_output = alu_input_1 + alu_input_2;
    `SWD_OP: alu_output = alu_input_1 + alu_input_2;
    `BNE_OP: condition_bit = (alu_input_1 != alu_input_2);
    `BEQ_OP: condition_bit = (alu_input_1 == alu_input_2);
    `BGZ_OP: condition_bit = (alu_input_1 > 0);
```

```
        `BLZ_OP: condition_bit = (alu_input_1 < 0);
        default: alu_output = alu_output;
        endcase
    end
end
```

input으로 받아온 op_code가 `ALU_OP와 다를 경우에는 op_code를 조건으로 사용하는

case 문으로 op_code에 알맞은 alu_inptut1과 alu_input2를 계산해주어 alu_output에 할당해준다.

이때 branch instruction의 조건 확인을 위한 condition_bit에 대한 계산도 이 case문에서 진행된다.

cpu.v

cpu를 구성하는 모든 모듈을 포함하고 있는 마더 모듈이다.

clk에 따라 readM과 writeM을 주기적으로 1로 만들어 주고, ackoutput과 inputReady에 따라 입력받은 data를 instruction으로 활용할 지, regsiter에 저장할 데이터로 사용할지 결정해준다.

그리고 instruction을 fatch하고, control_unit에서 발생한 control_bit와 mux를 이용해 CPU 모듈의 데이터와

< input / output. >

```
output reg readM;
output reg writeM;
output reg [`WORD_SIZE-1:0] address;
inout [`WORD_SIZE-1:0] data;
input ackOutput;
input inputReady;
input reset_n;
```

readM : readM이 1이 될 때, testbench의 memory[address]에서 instruction 혹은 data를 읽어온다.

writeM : writeM이 1이 될 때 testbench의 memory[address] 에 data를 저장한다.

address : 읽어야 하는 혹은 저장해야 하는 memory의 address이다.

평소에는 다음 instruction memory의 address를 의미하지만, load일 경우 데이터 값을 읽어와야 하는 memory의 address를 의미하고, store일 경우에는 데이터 값을 저장해야 하는 target memory의 address이다.

data : inout port로 한동안 high impedance를 유지하고 있다

ackOutput : cpu module에서 output으로 나온 데이터가 testbench에 위치한 메모리에서 제대로 저장될 때까지 1의 값을 유지하고, 이후 0의 값을 갖게 되는 control_bit.

inputReady : testbench의 memory에서 instruction 혹은 메로리에 저장된 데이터를 읽어올 때 1이 되는 control_bit, 일정시간 1의 값을 유지하다 0의 값이 된다.

reset_n : 0이 되면 cpu내의 레지스터, instruction, 모든 control_bit를 0으로 만든다.

cpu 모듈의 output들을 모두 equential logic을 이용해 control 된다.

먼저 readM은 posedge clk에 1의 값을 갖는다. 이후 posedge inputReady에서 0이 된다.

그리고 예외적으로 load instruction의 경우 메모리에서 instruction을 읽어온 후, 다시 메모리에 저장되어 있는 어떤 데이터 값을 읽어와야 한다.

따라서 negedge clk 에서 control_mem_read(mem_read)가 1일 때, 다시 readM을 1로 만들어주어 한 clock cycle내에서 readM이 두번 1, 메모리를 두번 읽어올 수 있도록 만들었다.

writeM은 store instruction을 메모리에서 읽은 후 cpu의 data output을 메로리에 저장할 때 1의 값을 가져야 하는 control_bit이다.

writeM은 negdege clk에서 control_mem_write(mem_write)의 값이 1일 경우 1의 값을 갖게 된다. negedege clk에서 writeM과 readM을 1로 동시에 만들어 줄 수 있는 sequential logic을 사용할 수 있는 이유는, 이런 case는 존재하지 않기 때문이다.

negedge clk에서 readM이 1이 되는 경우는 load instruction일 때, writeM이 1이 되는 경우는 store instruction일 때 이므로, readM과 writeM이 동시에 negedge에서 1이 되는 경우는 발생하지 않는다.

address 는 instruction의 주소를 가리키는 instruction_address 와 메모리에 데이터를 저장하거나 데이터를 읽기 위해 사용되는 data_address 두 가지로 사용된다.

1) instruction_address

instruction_address는 처음에는 0으로부터 시작하며 posedge clk 마다 cpu에서 계산된 instruction_address_next(다음 instruction의 주소)가 assign된다.

```
assign instruction_target_plus1 = instruction_address + 16'd1;
  //assign instruction_target_branch = instruction_target_plus1;
  assign instruction_address_next = (control_jpr ? read_out1
                        : control_jp ? extended_immediate
                        : instruction_target_plus1) +
                        ((condition_bit & control_branch) ?
                        extended_immediate : 0);
  assign address = clk ? instruction_address : alu_output;
```

```
assign data = writeM ? read_out2 : 16'bz;
```

```
  wire reg_write_control_bit;
  assign reg_write_control_bit = control_reg_write | control_mem_to_reg & ~clk;

  assign reg_write_data = control_mem_to_reg ? data :
                  control_pc_to_reg ? instruction_address :
                  alu_output;
  assign reg_input2 = control_pc_to_reg ? 2'b10 :
                control_alu_src ? if_read_reg2 :
                if_write_reg;
```

```
  assign alu_input2 = control_alu_src ? extended_immediate : read_out2;
```

Discussion

이번 single cycle cpu에서 구현이 가장 힘들었던 부분은 readM과 writeM, 그리고 address의 값을 바꿔주고 업데이트 해주는 sequential logic을 구현하는 것이었다. single cycle안에 instruction을 수행하기 위해선 posedge clk와 negedge clk를 적절히 사용해야 한다는 것을 알 수 있었다. 그리고 clk의 posedge와 negedge에서 값을 읽을 때, 좀더 쉽게 컨트롤 할 수 있도록 도와줄 수 있는 inputReady와 ackOutput을 이용해 시퀄셜 로직을 구현하는 것이 이번 랩의 중요 포인트였음을 알 수 있었다.

또, 같은 메모리에서 instruction과 data를 모두 읽어오기 때문에 data와 address를 각각의 상황마아 어떻게 해석해야하는지에 대해서도 어려움이 있었다. 하지만 이러한 부분은 위 readM과 writeM을 control

하는 sequential logic에 대한 정확한 이해를 하게 되니 쉽게 구현할 수 있었다.

결과적으로 우리 조는 이번 single cycle cpu를 구현해보며 하나의 메모리에서 데이터를 읽어오고, 데이터를 작성할 때 필요한 inout port해석과 address 해석을 달리 하는 sequential logic의 구현 방법을 완전히 이해하게 되었고, 더 나아가 베릴로그에서 다중의 state change을 한 clock cycle 내에 구현할 수 있는 sequential logic 구현 방법에 대해서 알게 되었다.

Conclusion

FSM, single sycle cpu를 거치며 clk를 활용한 sequential logic 구현 방법에 대해 많이 배울 수 있었다. 특히, 이번 랩을 통해 하나의 edge에서 state를 change하는 것이 아닌, posedge, negedge를 넘어서 다중의 sequential logic을 이용한 다중의 state를 change하는 방법을 익힐 수 있었다.