

# Logic Design Project Report

2013-11826 임주경

## 1. Introduction

Verilog를 이용해서 Simple 8-bit microprocessor를 구현한다. 구현한 Microprocessor는 FPGA Board에 프로그램하고, 정상적인 작동을 검사한다. Instruction, Register, Data의 size는 8-bit이다. 구현해야 할 명령어는 add, sub, load, store, jump, nop, addi 총 7개이며, Register File의 Register는 4개로 구현한다. CPU 설계를 위해 구현해야 할 모듈은 Program Counter, Register File, Control Logic Unit, Sign Extension, ALU가 있다.

## 2. Implementation

### 2.1. ALU

```
endmodule
module alu
(
    input [1:0] op,
    input [7:0] alu_in1,
    input [7:0] alu_in2,

    output reg [7:0] alu_out
);

    always @ (*) begin
        case(op)
            2'b00: alu_out = alu_in1;
            2'b01: alu_out = alu_in1 + alu_in2;
            2'b10: alu_out = alu_in1 - alu_in2;
            2'b11: begin
                if(alu_in2[7] == 0) alu_out = alu_in1 + alu_in2;
                else alu_out = alu_in1 - ~(alu_in2) + 1;
            end
        endcase
    end
end
endmodule
```

ALU는 두 개의 8-bit Input과 2-bit Control Input, 8-bit Output으로 구현하였다. 2-bit Control Input은 주어진 CPU의 SPEC에 맞는 명령어의 연산을 수행하도록 case를 나눠 연산한다. Control Input에 따라서 2'b00일 때는 무의미한 연산으로 입력1을 출력하도록 하였고, 2'b01은 입력1과 입력2의 덧셈, 2'b10은 뺄셈, 2'b11은 추가 구현인 addi를 구현하기 위해서 2's complement를 고려해 입력2의 MSB에 따라서 덧셈, 뺄셈 두가지로 나누어 구현하였다. Addi는 덧셈으로만 구현해도 문제 없었으나, 코드에 두 가지 상황을 명시하기 위해 이러한 구현을 선택하였다.

## 2.2. Sign Extension

```
module sign_extension
(
    input [5:0] sign_extension_in,
    input jump,
    output reg [7:0] sign_extension_out
);

    always @ (sign_extension_in or jump) begin
        if(jump == 1) begin
            sign_extension_out [5:0] <= sign_extension_in;
            sign_extension_out [7:6] <= {2{sign_extension_in[5]}};
        end
        else begin
            sign_extension_out [1:0] <= sign_extension_in [1:0];
            sign_extension_out [7:2] <= {6{sign_extension_in [1]}};
        end
    end
endmodule
```

Sign Extension은 해당 SPEC에 따라서 6-bit의 Input을 받으며, Control Input으로 jump를 설정하였다. Output은 8-bit로 입력의 sign extend된 값을 출력한다. 기능은 Input값이 변경될 때, 일어난다. 이때, jump가 asserted면, 6-bit 상수를 sign extend해서 PC에 전달해야 하므로 출력의 [5:0]은 입력과 동일하게, [7:6]은 입력의 [5]과 같은 값을 갖고, jump가 asserted가 아니면, 2-bit 상수를 sign extend하기 위해 출력의 [1:0]은 입력과 동일하게, [7:2]는 입력의 [1]과 같은 값을 갖도록 구현한다.

## 2.3. Register File

```
module register_file
(
    input clk,
    input areset,
    input reg_write_enable,
    input [1:0] reg_write_addr,
    input [7:0] reg_write_data,
    input [1:0] reg_read_addr1,
    output [7:0] reg_read_data1,
    input [1:0] reg_read_addr2,
    output [7:0] reg_read_data2
);

    reg [7:0] r [3:0];

    always @ (posedge clk or posedge areset) begin
        if(areset) begin
            r[0] <= 8'b00000000;
            r[1] <= 8'b00000000;
            r[2] <= 8'b00000000;
            r[3] <= 8'b00000000;
        end
        else begin
            if(reg_write_enable) begin
                r[reg_write_addr] <= reg_write_data;
            end
        end
    end

    assign reg_read_data1 = r[reg_read_addr1];
    assign reg_read_data2 = r[reg_read_addr2];
endmodule
```

Register File은 Clock과 synchronous하게 구현하였다. Areset 입력시에 모든 레지스터 값을 0으로

초기화한다. 그외에 Clock의 posedge마다 control input인 write enable이 asserted면 인풋으로 받은 write address에 해당하는 주소의 레지스터 값을 변경한다. 이때, 변경 값은 input write data 값으로 변경된다. 레지스터 파일의 주소에 대한 입력 값은 2-bit이며, 레지스터 4개에 대한 주소를 의미한다. 레지스터 파일의 출력 값인 register data 1과2는 Clock에 관계없이 항상 입력된 read address 1, 2에 해당하는 주소의 레지스터 값으로 지정한다.

## 2.4. Control Logic Unit

```

module control
(
    input [1:0] mode,
    input [1:0] opcode,
    input areset,

    output reg [1:0] alu_op,
    output reg alu_src, jump, reg_dst, mem_to_reg, mem_write, reg_write
);

    always @ (*) begin
        if(areset == 1) begin
            reg_dst = 0;
            mem_to_reg = 0;
            alu_src = 0;
            alu_op = 2'b00;
            jump = 0;
            mem_write = 0;
            reg_write = 0;
        end

        else begin
            case(mode)
                2'b00: begin // jump
                    reg_dst = 0;
                    mem_to_reg = 0;
                    alu_src = 0;
                    alu_op = 2'b01;
                    jump = 1;
                    mem_write = 0;
                    reg_write = 0;
                end
                2'b01: begin // load
                    reg_dst = 0;
                    mem_to_reg = 1;
                    alu_src = 0;
                    alu_op = 2'b01;
                    jump = 0;
                    mem_write = 0;
                    reg_write = 1;
                end
                2'b10: begin // store
                    reg_dst = 0;
                    mem_to_reg = 0;
                    alu_src = 0;
                    alu_op = 2'b01;
                    jump = 0;
                    mem_write = 1;
                    reg_write = 0;
                end
                2'b11: begin // arith
                    case(opcode)
                        2'b00: begin // nop
                            reg_dst = 0;
                            mem_to_reg = 0;
                            alu_src = 0;
                            alu_op = 2'b00;
                            jump = 0;
                            mem_write = 0;
                            reg_write = 0;
                        end
                        2'b01: begin // add
                            reg_dst = 1;
                            mem_to_reg = 0;
                            alu_src = 1;
                            alu_op = 2'b01;
                            jump = 0;
                            mem_write = 0;
                            reg_write = 1;
                        end
                        2'b10: begin // sub
                            reg_dst = 1;
                            mem_to_reg = 0;
                            alu_src = 1;
                            alu_op = 2'b10;
                            jump = 0;
                            mem_write = 0;
                            reg_write = 1;
                        end
                        2'b11: begin // addi
                            reg_dst = 1;
                            mem_to_reg = 0;
                            alu_src = 0;
                            alu_op = 2'b11;
                            jump = 0;
                            mem_write = 0;
                            reg_write = 1;
                        end
                    end
                end
            end
        end
    end
end

```

Control Logic Unit은 명령어에 맞는 Control 변수 값들을 Output으로 다른 모듈에 전달해준다.

Areset이 asserted되면, 모든 Output은 0으로 지정한다. 2-bit Input Mode가 2'b00일 때는, jump를 수행해야 하므로, ALU OP는 덧셈 연산을 위한 2'b01, jump는 1, 나머지는 0을 출력한다. Mode가 2'b01일 때는, load를 수행해야 한다. Memory to Register는 1, ALU OP는 메모리 주소의 덧셈 연산을 위한 2'b01, 레지스터에 값을 저장해야 하므로, Register write는 1, 나머지는 0을 출력한다. Mode가 2'b10일 때, store를 수행해야 한다. Load와 마찬가지로 ALU OP는 2'b01, 메모리에 값을 저장해야 하므로 Memory Write는 1, 나머지는 0을 출력한다. Mode가 2'b11일 때는 또 다른 인풋 OP code에 따라 경우를 나누어야 한다. OP code가 2'b00은 nop 구현을 위해 모든 Output을 0으로 출력하며, 2'b01, 2'b10은 add와 sub 구현으로 REG DST를 1로 설정하여, write register를 Instruction data [3:2]의 값을 주소로 갖도록 설정하며, ALU SRC를 1로 설정하여, ALU의 Input2를 Register File의 data2와 연결시킨다. ALU OP는 각 연산에 해당하는 OP code를 그대로 전달하며, 연산 결과를 레지스터에 저장하기 위해서 Reg Write를 1로 출력한다. 추가 구현인 addi의 경우 ALU OP는 마찬가지로 OP code를 전달하고, ALU SRC는 0으로 설정해서 Sign Extension의 상수를 ALU의 Operand가 될 수 있도록 연결한다. 그 외에 나머지는 add, sub의 경우와 같다.

## 2.5. Program Counter

```
// PC
always @ (posedge clk or posedge areset) begin
    if(areset == 1) pc_current <= 8'b00000000;
    else pc_current <= pc_next;
end

assign imem_addr = pc_current;

// Next PC Input
assign pc_next = (jump == 0)? (pc_current + 8'b00000001) : (pc_current + sign_extension_output);
```

Program Counter는 모듈을 분리하지 않고 CPU 내부에서 구현하였다. 현재 PC의 Output을, pc current로 하여, CPU의 instruction address와 연결하였다. 다음 사이클에서 입력될 PC Input은 pc next로 구현하였다. Jump wire의 값에 따라서 pc next의 값은 달라진다. PC의 상태는 Clock의 posedge마다 바뀌거나, areset이 asserted일 때, 8'b00000000의 값으로 설정된다.

## 2.6. Overall Design and Structure

전체적인 Data Path는 프로젝트 매뉴얼에 있는 회로도를 참고하였으며, PC를 제외하고 wiring과 structure는 매뉴얼과 동일하다.

