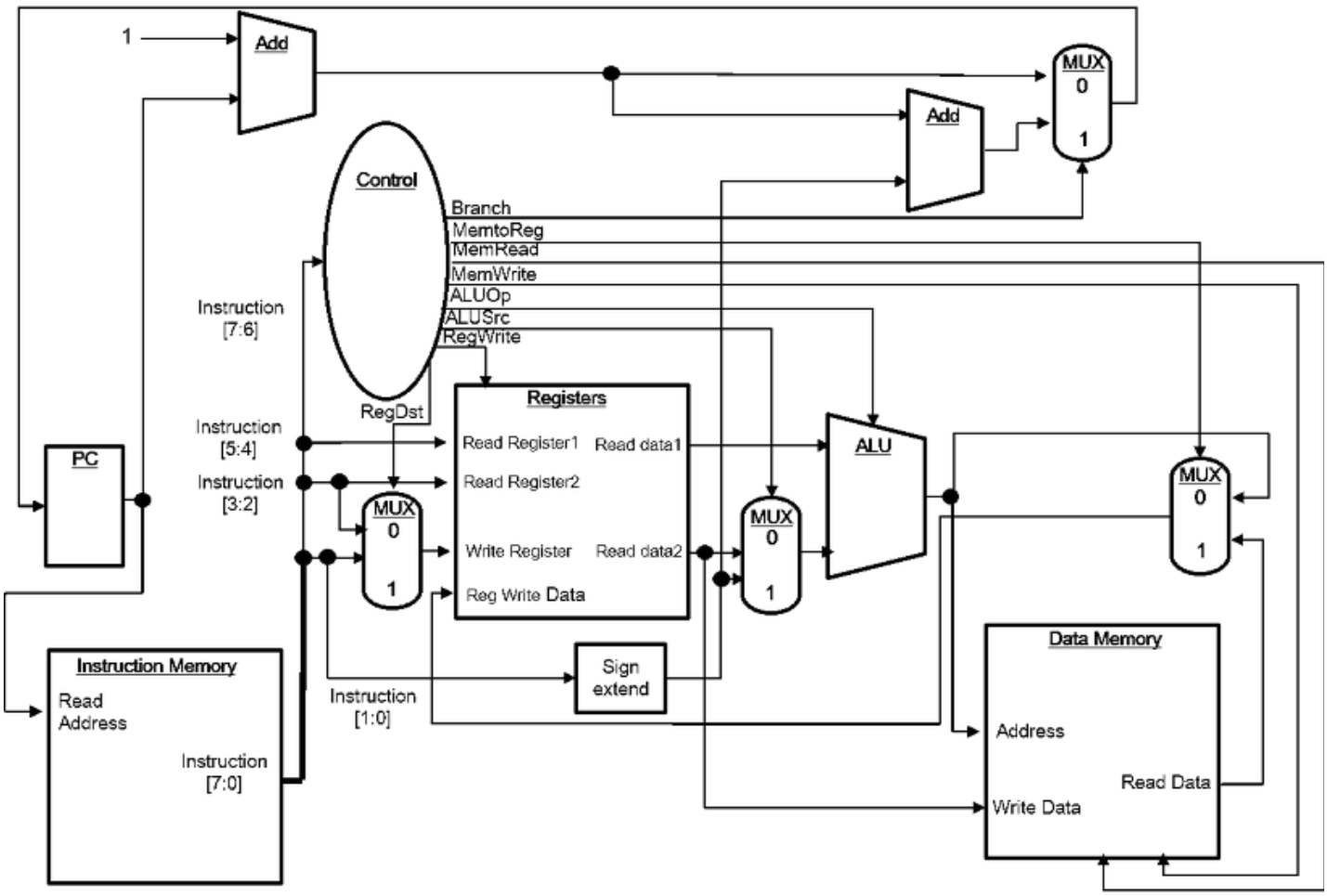
9강. CPU 간주

1. 8-bit Microprocessor 개요



(1) 입력

① 외부 메모리(TA의 FPGA)로부터 Instruction을 읽어 들인다.

② 〈 NET "reset" CLOCK_DEDICATED_ROUTE = FALSE; 〉를 .ucf 파일에 추가한다.

③ Instruction[7:0] 포트: (M) 130 129 127 126 125 124 121 120 (L)

④ Read_Address[7:0] 포트: (M) 105 104 103 102 101 99 98 96 (L)

(2) 출력

① 기본: RegWriteData에 해당하는 데이터, 7 segment 디스플레이용(J1, J2)

- sw에서 Data Memory의 Address를 가리킨다. (∵ MemtoReg = 0)
- j에서 얼마나 건너뛸지(imm)를 가리킨다.

② **추가**: 연산에 참여하는 레지스터(0 ~ 3), 7 segment 디스플레이용(J3)

③ **추가**: sw 시 해당 레지스터에 저장된 값, 7 segment 디스플레이용(J4, J5)

④ **추가**: PC의 출력값 중 일의 자리, 7 segment 디스플레이용(J6)

⑤ **추가**: op 코드(one-hot encoding), LED(D1, D2, D3, D4)

⑥ **추가**: Arduino 연동용 9-bit 출력

(3) 채점기준

① 보고서(40%)

- 전체적인 디자인과 구조를 자세히 설명한다.

- 구현 상의 각 모듈의 기능을 구체적으로 설명한다.
- 시뮬레이션 결과가 올바른 결과인지 확인한다.

② 완결성(60%): 테스트 Instruction 세트가 테스트될 것이다.

③ 가산점(~5%): 레지스터 display, op 코드 display 등

⑷ 마감일

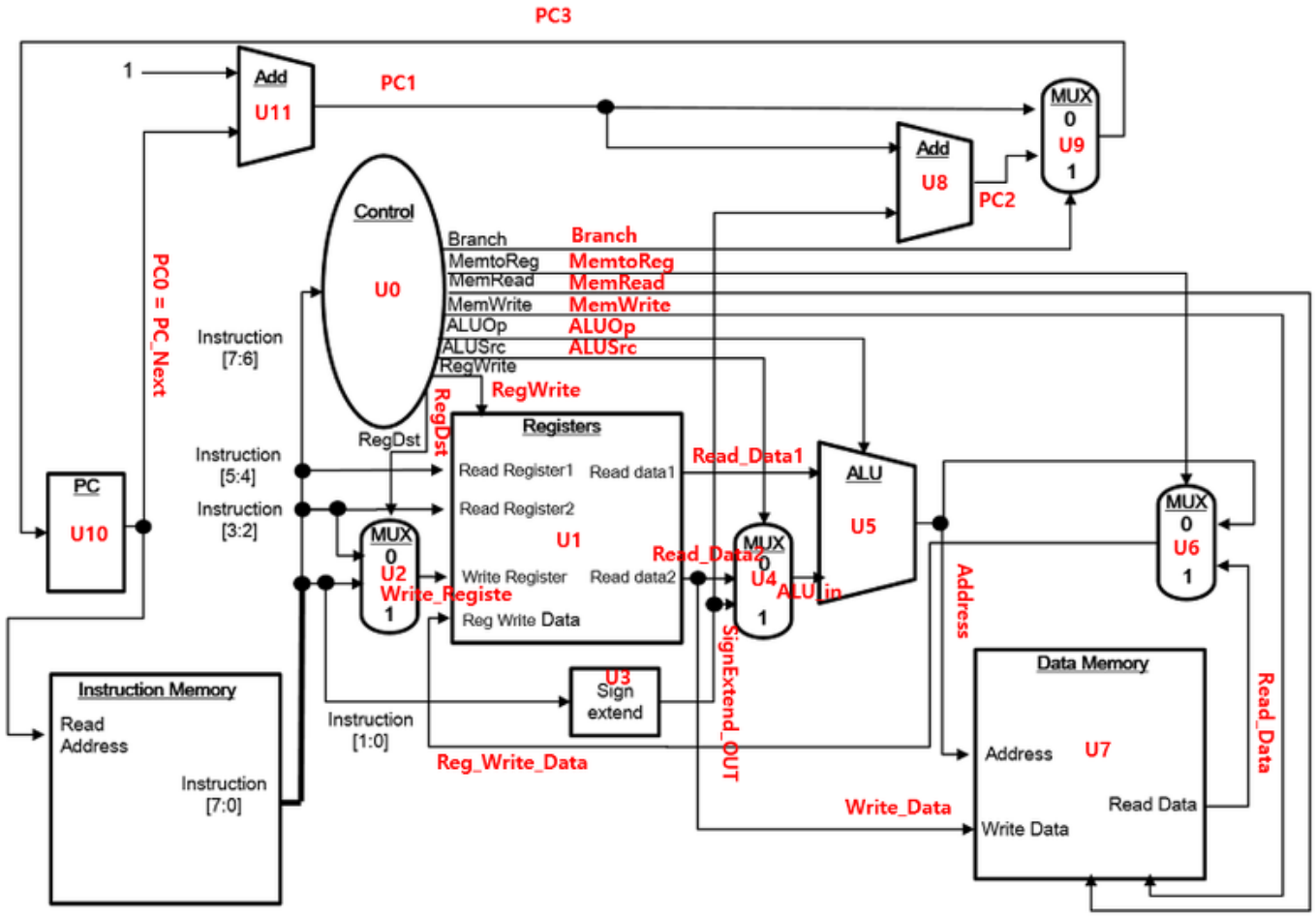
① 데모: 세 날짜 중 하나를 선택하고 재시험이 가능하다.

- 20th, Dec., Tue. 18:30 ~
- 22nd, Dec., Thu. 15:30 ~ 18:00
- 26th, Dec., Mon. 16:00 ~ 18:00
- 데모는 26th, Dec., Mon 18:00 이후로 불가능하다.

② 보고서와 코드 제출(e-mail) 마감: 26th, Dec., 23:59

③ FPGA 제출 마감: 데모 직후

2. System: 자체 Instruction Memory를 활용하고자 하는 경우, **input** instruction, **output** PC_Next를 지운 뒤 아래에 그 이름의 wire를 추가하고 Instruction Memory 모듈을 추가하면 된다.




```
`timescale 1ns / 1ps

module System(
    input[7:0] instruction,
    input reset,
    input clk,
    output [7:0] PC_Next,
    output [3:0] op,
```

```
output [6:0] OUT0,
output [6:0] OUT1,
output [6:0] OUT2,
output [6:0] OUT3,
output [6:0] OUT4,
output [6:0] OUT5,
output [8:0] send
);

clkdiv clkdiv_0(.clkin(clk), .clkout(CLK));

// declaration
wire Branch, MemtoReg, MemRead, MemWrite, ALUOp, ALUSrc, RegWrite, RegDst;
wire [7:0] register0;
wire [7:0] register1;
wire [7:0] register2;
wire [7:0] register3;
wire [1:0] Write_Register;
wire [7:0] Reg_Write_Data;
wire [7:0] Read_Data1, Read_Data2;
wire [7:0] SignExtend_OUT;
wire [7:0] ALU_in;
wire [7:0] Address;
wire [7:0] Read_Data;
wire [7:0] Write_Data;
wire [7:0] PC0, PC1, PC2, PC3;
wire start;

// wiring
assign Write_Data = Read_Data2;
assign PC_Next = PC0;
Counter cc(.IN(reset), .count(start));
Control U0(.op(instruction[7:6]), .branch(Branch), .MemtoReg(MemtoReg),
         .MemRead(MemRead), .MemWrite(MemWrite), .ALUSrc(ALUSrc),
         .ALUOp(ALUOp), .RegWrite(RegWrite), .RegDst(RegDst));
Registers U1(.Read_Register1(instruction[5:4]), .Read_Register2(instruction[3:2]),
                .Write_Register(Write_Register), .Reg_Write_Data(Reg_Write_Data),
                .Read_Data1(Read_Data1), .Read_Data2(Read_Data2),
                .RegWrite(RegWrite), .reset(reset), .CLK(CLK),
                .register0(register0), .register1(register1), .register2(register2), .register3(register3));
MUX2 U2(.IN1(instruction[3:2]), .IN2(instruction[1:0]), .S0(RegDst), .OUT(Write_Register));
SignExtend U3(.IN(instruction[1:0]), .OUT(SignExtend_OUT));
MUX8 U4(.IN1(Read_Data2), .IN2(SignExtend_OUT), .S0(ALUSrc), .OUT(ALU_in));
ALU U5(.ENA(ALUOp), .IN1(Read_Data1), .IN2(ALU_in), .OUT(Address));
Data_Memory U7(.Address(Address), .MemWrite(MemWrite), .MemRead(MemRead),
                   .reset(reset), .Write_Data(Write_Data), .Read_Data(Read_Data));
MUX8 U6(.IN1(Address), .IN2(Read_Data), .S0(MemtoReg), .OUT(Reg_Write_Data));
ALU U8(.ENA(1), .IN1(PC1), .IN2(SignExtend_OUT), .OUT(PC2));
MUX8 U9(.IN1(PC1), .IN2(PC2), .S0(Branch), .OUT(PC3));
PC U10(.IN(PC3), .CLK(CLK & start), .reset(reset), .OUT(PC0));
ALU U11(.ENA(1), .IN1(8'b00000001), .IN2(PC0), .OUT(PC1));

// output
// LED D1, D2, D3, D4 (one-hot encoding for op)
MUX4 op1(.IN(4'b1000), .S1(instruction[7]), .S0(instruction[6]), .OUT(op[3]));
MUX4 op2(.IN(4'b0100), .S1(instruction[7]), .S0(instruction[6]), .OUT(op[2]));
MUX4 op3(.IN(4'b0010), .S1(instruction[7]), .S0(instruction[6]), .OUT(op[1]));
MUX4 op4(.IN(4'b0001), .S1(instruction[7]), .S0(instruction[6]), .OUT(op[0]));
// 7 segment J1, J2
wire [7:0] mux_Reg_Write_Data;
MUX84 mux0(.IN3(Reg_Write_Data), .IN2(Reg_Write_Data), .IN1(Reg_Write_Data), .IN0(SignExtend_OUT),
             .S(instruction[7:6]), .OUT(mux_Reg_Write_Data));
segment out0(.IN(mux_Reg_Write_Data[3:0]), .OUT(OUT0));
segment out1(.IN(mux_Reg_Write_Data[7:4]), .OUT(OUT1));
// 7 segment J3
wire [7:0] target_Register;
MUX84 mux1(.IN3({6'b0, Write_Register}), .IN2({6'b0, Write_Register}), .IN1({6'b0, Read_Register2}),
          .IN0(8'b0), .S(instruction[7:6]), .OUT(target_Register));
segment out2(.IN(target_Register[3:0]), .OUT(OUT2));
// 7 segment J4, J5
wire [7:0] Register_value;
MUX84 mux2(.IN3(8'b0), .IN2(8'b0), .IN1(Read_Data2), .IN0(8'b0),
           .S(instruction[7:6]), .OUT(Register_value));
segment out3(.IN(Register_value[7:4]), .OUT(OUT3));
segment out4(.IN(Register_value[3:0]), .OUT(OUT4));
// 7 segment J6
segment out5(.IN(PC_Next[3:0]), .OUT(OUT5));
// Arduino
Output so(.PC(PC_Next), .instruction(instruction), .register0(register0), .register1(register1),
         .register2(register2), .register3(register3), .Reg_Write_Data(mux_Reg_Write_Data),
         .clk(clk), .start(start), .send(send));

// display setting for test bench
always@(instruction) begin
    case(instruction[7:6])
        2'b00 :
```

```
                $display("register[%d] = register[%d] + register[%d]",
                         instruction[1:0], instruction[5:4], instruction[3:2]);
            2'b01 : begin
                case(instruction[5:4])
                    2'b00 :
                        $display("register[%d] = memory[%0d+%d]",
                                 instruction[3:2], register0, $signed(instruction[1:0]));
                    2'b01 :
                        $display("register[%d] = memory[%0d+%d]",
                                 instruction[3:2], register1, $signed(instruction[1:0]));
                    2'b10 :
                        $display("register[%d] = memory[%0d+%d]",
                                 instruction[3:2], register2, $signed(instruction[1:0]));
                    2'b11 :
                        $display("register[%d] = memory[%0d+%d]",
                                 instruction[3:2], register3, $signed(instruction[1:0]));
                endcase
            end
            2'b10 : begin
                case(instruction[5:4])
                    2'b00 :
                        $display("memory[%0d+%d] = register[%d]",
                                 register0, $signed(instruction[1:0]), instruction[3:2]);
                    2'b01 :
                        $display("memory[%0d+%d] = register[%d]",
                                 register1, $signed(instruction[1:0]), instruction[3:2]);
                    2'b10 :
                        $display("memory[%0d+%d] = register[%d]",
                                 register2, $signed(instruction[1:0]), instruction[3:2]);
                    2'b11 :
                        $display("memory[%0d+%d] = register[%d]",
                                 register3, $signed(instruction[1:0]), instruction[3:2]);
                endcase
            end
            2'b11 :
                $display("jump NextPC + %0d", $signed(instruction[1:0]));
        endcase
    end
endmodule
```

3. Control Unit (Controller)

```
`timescale 1ns / 1ps

module Control(
    input [1:0] op,
    output branch,
    output MemtoReg,
    output MemRead,
    output MemWrite,
    output ALUSrc,
    output ALUOp,
    output RegWrite,
    output RegDst
    );


     MUX4 U0(.IN(4'b0001), .S1(op[1]), .S0(op[0]), .OUT(branch));
     MUX4 U1(.IN(4'b0100), .S1(op[1]), .S0(op[0]), .OUT(MemtoReg));
     MUX4 U2(.IN(4'b0100), .S1(op[1]), .S0(op[0]), .OUT(MemRead));
     MUX4 U3(.IN(4'b0010), .S1(op[1]), .S0(op[0]), .OUT(MemWrite));
     MUX4 U4(.IN(4'b0110), .S1(op[1]), .S0(op[0]), .OUT(ALUSrc));
     MUX4 U5(.IN(4'b1110), .S1(op[1]), .S0(op[0]), .OUT(ALUOp));
     MUX4 U6(.IN(4'b1100), .S1(op[1]), .S0(op[0]), .OUT(RegWrite));
     MUX4 U7(.IN(4'b1000), .S1(op[1]), .S0(op[0]), .OUT(RegDst));

endmodule
```

⑴ op 코드에 따라 필요한 레지스터와 메모리를 준비하고 그 경로를 뚫는다.

⑵ I/O: 2-bit op 코드를 입력으로 하여 1-bit control 신호들(8개)을 출력한다.

⑶ Control Signal Table

	출력: LOW	출력: HIGH
RegDst	Write_Register에 rt를 할당	Write_Register에 rd를 할당
RegWrite	·	rd에 Reg_Write_Data를 덮어 씀
ALUSrc	ALU의 둘째 연산자가 Read_Data2	ALU의 둘째 연산자가 SignExtend_OUT
Branch	PC ← PC + 1	PC ← PC + 1 + imm
MemRead	·	Address의 데이터가 Read_Data에 인가
MemWrite	·	Address에 Write_Data를 덮어 씀
MemtoReg	Address를 Reg_Write_Data에 인가	Read_Data를 Reg_Write_Data에 인가
ALUOp	·	ALU 기능(덧셈 기능 only)을 활성화

⑷ Instruction과 Control Signal의 관계

Instruction	RegDst	RegWrite	ALUSrc	Branch	MemRead	MemWrite	MemtoReg	ALUOP
add	1	1	0	0	0	0	0	1
lw	0	1	1	0	1	0	1	0
sw	x	0	1	0	0	1	x	0
j	x	0	0	1	0	0	x	0

4. Instruction Memory

```
`timescale 1ns / 1ps

module Instruction_Memory(
        input [7:0] Read_Address,
        output [7:0] Instruction
    );

    wire [7:0] MemByte[17:0];

    assign MemByte[0] = 8'b01000101;
    assign MemByte[1] = 8'b01011001;
    assign MemByte[2] = 8'b00011011;
    assign MemByte[3] = 8'b00011111;
    assign MemByte[4] = 8'b10011101;
    assign MemByte[5] = 8'b00110101;
    assign MemByte[6] = 8'b00010101;
    assign MemByte[7] = 8'b00011001;
    assign MemByte[8] = 8'b00011011;
    assign MemByte[9] = 8'b01100101;
    assign MemByte[10] = 8'b00110111;
    assign MemByte[11] = 8'b01110100;
    assign MemByte[12] = 8'b10101100;
    assign MemByte[13] = 8'b01100000;
    assign MemByte[14] = 8'b11000001;
    assign MemByte[15] = 8'b00000000; // undifined
    assign MemByte[16] = 8'b00110111;
    assign MemByte[17] = 8'b11000010;
    assign Instruction = MemByte[Read_Address];
endmodule
```

⑴ Instruction Memory I/O

① Instruction(≒ 명령어)들을 저장해 두는 장소이다.

② PC로부터의 입력(8-bit)이 어떤 명령어를 읽어 들일지를 결정한다.

③ 해당 Instruction(8-bit)을 출력한다.

⑵ Instruction 기본 스펙

① Instruction 크기: 8-bit

② 입력 포트 개수: 8-bit (Instruction 주소); 256개의 Read_Address가 존재

③ 출력 포트 개수: 8-bit (레지스터에 저장된 Instruction)

⑶ Instruction

CMD	타입	예시								Assembly code	C 언어 (의사코드)
		7	6	5	4	3	2	1	0		
	R	op		rs		rt		rd			
	I	op		rs		rt		imm			
	J	op						imm			
add	R	0		1		2		0		add $s0, $s1, $s2	$s0 = $s1 + $s2
lw	I	1		3		0		0		lw $s0, 0[$s3]	$s0 = Mem[$s3+0]
sw	I	2		3		0		1		sw $s0, 1[$s3]	Mem[$s3+1] = $s0
j	J	3						-2		j -2	goto (next PC-2)

① Instruction 형식

- [7:6] op 코드 (operation code): 00(0), 01(1), 10(2), 11(3)
- [5:4] rs(source register): 연산에 참여하는 레지스터
- [3:2] rt(source register two): 연산에 참여하는 레지스터
- [1:0] rd(destination register): 연산 결과를 저장하는 레지스터(cf. add)
- [1:0] imm(immediate): 바로 활용하는 값; 00(0), 01(1), 10(-2), 11(-1)

② 레지스터 (8-bit): PC, Registers($s0, $s1, $s2, $s3 등 4개의 8-bit 레지스터)

③ 메모리 (8-bit)

- Instruction Memory: 0x00 - 0xFF
- Data Memory: 0x00 - 0xFF, 위 표에서 Mem으로 표시

④ Instruction Memory는 TA로부터 주어진다.

5. Registers(register file)

```
`timescale 1ns / 1ps

module Registers(
    input [1:0] Read_Register1,
    input [1:0] Read_Register2,
    input [1:0] Write_Register,
    input [7:0] Reg_Write_Data,
    input RegWrite,
    input reset,
    input CLK,
    output [7:0] Read_Data1,
    output [7:0] Read_Data2,
    output [7:0] register0,
    output [7:0] register1,
    output [7:0] register2,
    output [7:0] register3
    );

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

     initial begin
         register[0] = 8'd0;
         register[1] = 8'd0;
         register[2] = 8'd0;
         register[3] = 8'd0;
     end

     assign Read_Data1 = register[Read_Register1];
```

```
    assign Read_Data2 = register[Read_Register2];

    always@(posedge CLK or posedge reset) begin
        if(reset) begin
            register[0] = 8'd0;
            register[1] = 8'd0;
            register[2] = 8'd0;
            register[3] = 8'd0;
        end
        else begin
            if(RegWrite) begin
                register[Write_Register] = Reg_Write_Data;
            end
        end
    end

    assign register0 = register[0];
    assign register1 = register[1];
    assign register2 = register[2];
    assign register3 = register[3];

endmodule
```

⑴ 기능: 변수들을 저장하는 장소이다.

① 4개의 8-bit 범용 레지스터로 구성된다.

② 변수를 저장하는 장소로 $s0, $s1, $s2, $s3로 표시한다.

⑵ I/O

① Control Unit으로부터 RegWrite라는 입력 신호를 받는다.

② RegWrite = 1(add 또는 lw): Write_Register의 레지스터에 Write_Register에 인가된 데이터(8-bit)를 쓴다. 값은 바로 쓰는 게 아니라 다음 clock edge에서 쓰게 된다. (∴ CLK을 입력으로 함) 만약 값을 바로 쓰게 되면 add 연산 시 rs 또는 rt와 rd가 같은 경우 루프를 형성하며, 그 결과 그 레지스터는 알 수 없는 값을 가지게 된다.

③ Read_Register과 같은 별도의 신호 없이 Read_Register1이나 Register2에 해당(각각 rs, rt)하는 주소의 레지스터의 데이터가 각각 Read_Data1, Read_Data2에 인가된다.

⑶ reset의 rising edge에서 Registers는 다음과 같이 초기화된다.

$s0 = 0, $s1 = 0, $s2 = 0, $s3 = 0

6. Data Memory

```
`timescale 1ns / 1ps

module Data_Memory(
    input [7:0] Address,
    input MemWrite,
    input MemRead,
    input reset,
    input clk,
    input [7:0] Write_Data,
    output reg [7:0] Read_Data
    );

     reg [7:0] memory[31:0];
     reg [4:0] i;

     initial begin
         for(i=0; i<16; i=i+1) memory[i] = i;
         for(i=0; i<16; i=i+1) memory[i+16] = -i;
     end

     always@(reset or MemWrite or MemRead or Write_Data or Address) begin
         if(reset) begin
             for(i=0; i<16; i=i+1) memory[i] = i;
             for(i=0; i<16; i=i+1) memory[i+16] = -i;
         end
         else begin
             if(MemWrite && !MemRead) begin
```

```
                memory[Address] = Write_Data;
            end
            else if(!MemWrite && MemRead) begin
                Read_Data = memory[Address];
            end
        end
    end

endmodule
```

⑴ 기능

① 8-bit Address(즉, 데이터는 256개), 각 Address에 8-bit 레지스터

○ FPGA의 용량 제한으로 인해 오직 32개의 8-bit registers만 고려한다.

② 하드디스크와 유사한 개념

⑵ I/O: Control Unit으로부터 나온 MemRead, MemWrite를 입력으로 한다.

① MemRead = 1: 해당 Address의 레지스터의 값을 Read_Data에 인가한다.

② MemWrite = 1: 해당 Address의 레지스터에 Write_Data를 덮어 쓴다. Data Memory에서는 Registers와 같은 이슈가 없으므로 해당 clock 구간에서 바로 값을 덮어 쓴다.

⑶ reset의 rising edge에서 Data Memory는 다음과 같이 초기화된다.

① [0] = 0, [1] = 1, ..., [15] = 15

② [16] = 0, [17] = -1, ..., [31] = -15

7. PC(program counter)

```
`timescale 1ns / 1ps

module PC(
    input [7:0] IN,
    input CLK,
    input reset,
    output reg [7:0] OUT
    );

     always@(posedge CLK or posedge reset) begin
         if(reset) OUT <= -1;
         else OUT <= IN;
     end

endmodule
```

⑴ Instruction Memory에서 어느 줄(Read_Address)의 Instruction을 읽을지를 나타낸다.

⑵ 분기 명령이 실행되는 경우 그 목적지 주소로 갱신됨

⑶ 기능: j가 아니면 기존 PC의 값에 1을 더하고, j이면 기존 PC의 값에 1 + imm을 더한다.

⑷ reset의 rising edge에서 PC는 -1로 초기화된다. PC가 초기화된 바로 다음 clock posedge부터 Instruction을 실행시키기 위해 -1로 초기화한 것이다.

8. MUX

⑴ MUX2: 2-bit 데이터 2개 중 하나를 선택하는 모듈 (ex. U2)

```
`timescale 1ns / 1ps

module MUX2(
    input [1:0] IN1,
    input [1:0] IN2,
    input S0,
    output reg [1:0] OUT
    );
```

```
    always@(IN2 or IN1 or S0) begin
        if(S0) OUT = IN2;
        else OUT = IN1;
    end

endmodule
```

⑵ MUX4: 1-bit 데이터 4개 중 하나를 선택하는 모듈 (ex. op1, op2, op3, op4)

```
`timescale 1ns / 1ps

module MUX4(
    input [3:0] IN,
    output reg OUT,
    input S1,
    input S0
    );

    always@(IN or S1 or S0) begin
        case({S1,S0})
            2'b00 : OUT = IN[3];
            2'b01 : OUT = IN[2];
            2'b10 : OUT = IN[1];
            2'b11 : OUT = IN[0];
            default: OUT = 0;
        endcase
    end

endmodule
```

⑶ MUX8: 8-bit 데이터 2개 중 하나를 선택하는 모듈 (ex. mux_Reg_Write_Data를 위한 모듈, target_Register를 위한 모듈, Register_value를 위한 모듈)

```
`timescale 1ns / 1ps

module MUX8(
    input [7:0] IN1,
    input [7:0] IN2,
    input S0,
    output reg [7:0] OUT
    );

    always@(IN1 or IN2 or S0) begin
        if(S0) OUT = IN2;
        else OUT = IN1;
    end

endmodule
```

⑷ MUX84: 8-bit 데이터 4개 중 하나를 선택하는 모듈

```
`timescale 1ns / 1ps

module MUX84(
    input [7:0] IN3,
    input [7:0] IN2,
    input [7:0] IN1,
    input [7:0] IN0,
    input [1:0] S,
    output reg [7:0] OUT
    );

     always@(S or IN0 or IN1 or IN2 or IN3) begin
         case(S)
             2'b00 : OUT = IN3;
             2'b01 : OUT = IN2;
             2'b10 : OUT = IN1;
             2'b11 : OUT = IN0;
             default: OUT = 0;
         endcase
```

```
    end

endmodule
```

9. ALU

```
`timescale 1ns / 1ps

module ALU(
    input ENA,
    input [7:0] IN1,
    input [7:0] IN2,
    output reg [7:0] OUT
    );

     always@(ENA or IN1 or IN2) begin
         if(ENA) OUT <= IN1 +IN2;
         else OUT <= 0;
     end

endmodule
```

⑴ 기능: 이번 프로젝트에서는 덧셈 기능만 수행한다. (overflow 무시)

⑵ I/O

① 첫 번째 입력은 Read_data1(rs)이고, 두 번째 입력은 Control Unit에 따라 Read_data2(rt)이거나 SignExtend를 거친 Instruction[1:0](imm)이다.

② enable 신호로서 Control Unit으로부터 ALUOp가 입력으로 주어진다.

10. SignExtend: 2의 보수로 표현된 2-bit 수를 2의 보수로 표현된 8-bit 수로 변환하는 모듈

```
`timescale 1ns / 1ps

module SignExtend(
    input [1:0] IN,
    output [7:0] OUT
    );

     assign OUT [1:0] = IN;
     assign OUT [2] = IN[1];
     assign OUT [3] = IN[1];
     assign OUT [4] = IN[1];
     assign OUT [5] = IN[1];
     assign OUT [6] = IN[1];
     assign OUT [7] = IN[1];

endmodule
```

11. clkdiv: 내재된 50 MHz clock을 주기가 1초인 clock으로 바꿔주는 모듈

```
`timescale 1ns / 1ps

module clkdiv(
    input clkin,
    input clr,
    output reg clkout
    );

    reg [31:0] cnt;

    initial // every negedge appears at 1s, 2s, ...
        clkout = 0;
```

```
    always@(posedge clkin) begin
        if(clr) begin
            cnt <= 32'd0;
            clkout <= 1'b0;
        end
        else if(cnt == 32'd25000000) begin  // 50000000 times per second
            cnt <= 32'd0;
            clkout <= ~clkout;
        end
        else begin
            cnt <= cnt+1;
        end
    end

endmodule
```

12. segment: 7 segment 디스플레이를 위해 4-bit 16진수 숫자를 변환하는 모듈

```
`timescale 1ns / 1ps

module segment(
    input [3:0] IN,
    output reg [6:0] OUT
    );

    always @(IN) begin
        case(IN[3:0])
            4'b0000:    OUT = 7'b1111110;
            4'b0001:    OUT = 7'b0110000;
            4'b0010:    OUT = 7'b1101101;
            4'b0011:    OUT = 7'b1111001;
            4'b0100:    OUT = 7'b0110011;
            4'b0101:    OUT = 7'b1011011;
            4'b0110:    OUT = 7'b1011111;
            4'b0111:    OUT = 7'b1110000;
            4'b1000:    OUT = 7'b1111111;
            4'b1001:    OUT = 7'b1111011;
            4'b1010:    OUT = 7'b1110111;
            4'b1011:    OUT = 7'b0011111;
            4'b1100:    OUT = 7'b1001110;
            4'b1101:    OUT = 7'b0111101;
            4'b1110:    OUT = 7'b1001111;
            4'b1111:    OUT = 7'b1000111;
            default:OUT = 7'b0000000;
        endcase
    end

endmodule
```

13. Counter: reset이 한 번 눌러진 이후부터 start가 1로 유지되도록 하는 모듈 (단, start는 PC에 대한 enable 신호이다.)

```
`timescale 1ns / 1ps

module Counter(
    input IN,
    output reg count
    );

    reg _count;
    initial begin
        _count = 0;
        count = 0;
    end

    always@(posedge IN) begin
        if(_count==count)
            count = ~count;
    end
endmodule
```

14. Output: 타이밍에 따라 적당한 8-bit 데이터를 아두이노에 송신하는 모듈

```
`timescale 1ns / 1ps

module Output(
    input [7:0] PC,
    input [7:0] instruction,
    input [7:0] register0,
    input [7:0] register1,
    input [7:0] register2,
    input [7:0] register3,
    input [7:0] Reg_Write_Data,
    input clk,
    input start,
    output reg [7:0] send
    );

    wire clk_rnm0;

    clkdiv2 clkdiv2(.clkin(clk), .clkout(clk_rnm0));

    reg [3:0] count;

    initial begin
        count = 3'b000;
        send = 8'b00000000;
    end

    wire CLK = clk_rnm0 & start;

    always@(posedge CLK) begin
        if(!start) send = 0;
        else begin
            // step1
            if(count == 0 && start)
                send = 8'b11110000;
            // endstep1
            else if(count == 1 && start)
                send = PC;
            // step2
            else if(count == 2 && start)
                send = 8'b11110001;
            // endstep2
            else if(count == 3 && start)
                send = instruction;
            // step3
            else if(count == 4 && start)
                send = 8'b11110010;
            // endstep3
            else if(count == 5 && start)
                send = register0;
            // step4
            else if(count == 6 && start)
                send = 8'b11110011;
            // endstep4
            else if(count == 7 && start)
                send = register1;
            // step5
            else if(count == 8 && start)
                send = 8'b11100001;
            // endstep5
            else if(count == 9 && start)
                send = register2;
            // step6
            else if(count == 10 && start)
                send = 8'b11110010;
            // endstep6
            else if(count == 11 && start)
                send = register3;
            // step7
            else if(count == 12 && start)
                send = 8'b11110001;
            // endstep7
            else if(count == 13 && start)
                send = Reg_Write_Data;
            count = count + 1;
        end
    end

endmodule
```

15. System.ucf

```
# PlanAhead Generated physical constraints
NET "OUT0[6]" LOC = P11;
NET "OUT0[5]" LOC = P12;
NET "OUT0[4]" LOC = P13;
NET "OUT0[3]" LOC = P15;
NET "OUT0[2]" LOC = P16;
NET "OUT0[1]" LOC = P18;
NET "OUT0[0]" LOC = P19;
NET "OUT1[6]" LOC = P3;
NET "OUT1[5]" LOC = P4;
NET "OUT1[4]" LOC = P5;
NET "OUT1[3]" LOC = P6;
NET "OUT1[2]" LOC = P7;
NET "OUT1[1]" LOC = P8;
NET "OUT1[0]" LOC = P10;
NET "reset" LOC = P47;
NET "reset" CLOCK_DEDICATED_ROUTE = FALSE;
NET "clk" LOC = P57;
NET "instruction[7]" LOC = P130;
NET "instruction[6]" LOC = P129;
NET "instruction[5]" LOC = P127;
NET "instruction[4]" LOC = P126;
NET "instruction[3]" LOC = P125;
NET "instruction[2]" LOC = P124;
NET "instruction[1]" LOC = P121;
NET "instruction[0]" LOC = P120;
NET "PC_Next[7]" LOC = P105;
NET "PC_Next[6]" LOC = P104;
NET "PC_Next[5]" LOC = P103;
NET "PC_Next[4]" LOC = P102;
NET "PC_Next[3]" LOC = P101;
NET "PC_Next[2]" LOC = P99;
NET "PC_Next[1]" LOC = P98;
NET "PC_Next[0]" LOC = P96;
NET "op[3]" LOC = P87;
NET "op[2]" LOC = P88;
NET "op[1]" LOC = P90;
NET "op[0]" LOC = P91;
NET "OUT2[6]" LOC = P20;
NET "OUT2[5]" LOC = P21;
NET "OUT2[4]" LOC = P24;
NET "OUT2[3]" LOC = P25;
NET "OUT2[2]" LOC = P27;
NET "OUT2[1]" LOC = P28;
NET "OUT2[0]" LOC = P29;
NET "OUT3[6]" LOC = P55;
NET "OUT3[5]" LOC = P58;
NET "OUT3[4]" LOC = P59;
NET "OUT3[3]" LOC = P60;
NET "OUT3[2]" LOC = P62;
NET "OUT3[1]" LOC = P63;
NET "OUT3[0]" LOC = P64;
NET "OUT4[6]" LOC = P68;
NET "OUT4[5]" LOC = P69;
NET "OUT4[4]" LOC = P70;
NET "OUT4[3]" LOC = P71;
NET "OUT4[2]" LOC = P72;
NET "OUT4[1]" LOC = P75;
NET "OUT4[0]" LOC = P76;
NET "OUT5[6]" LOC = P77;
NET "OUT5[5]" LOC = P78;
NET "OUT5[4]" LOC = P79;
NET "OUT5[3]" LOC = P82;
NET "OUT5[2]" LOC = P83;
NET "OUT5[1]" LOC = P84;
NET "OUT5[0]" LOC = P85;
NET "send[7]" LOC = P117;
NET "send[6]" LOC = P116;
NET "send[5]" LOC = P115;
NET "send[4]" LOC = P114;
NET "send[3]" LOC = P113;
NET "send[2]" LOC = P112;
NET "send[1]" LOC = P111;
NET "send[0]" LOC = P110;
```

입력 : 2016.11.29 02:03