

Lab #10 RegFile

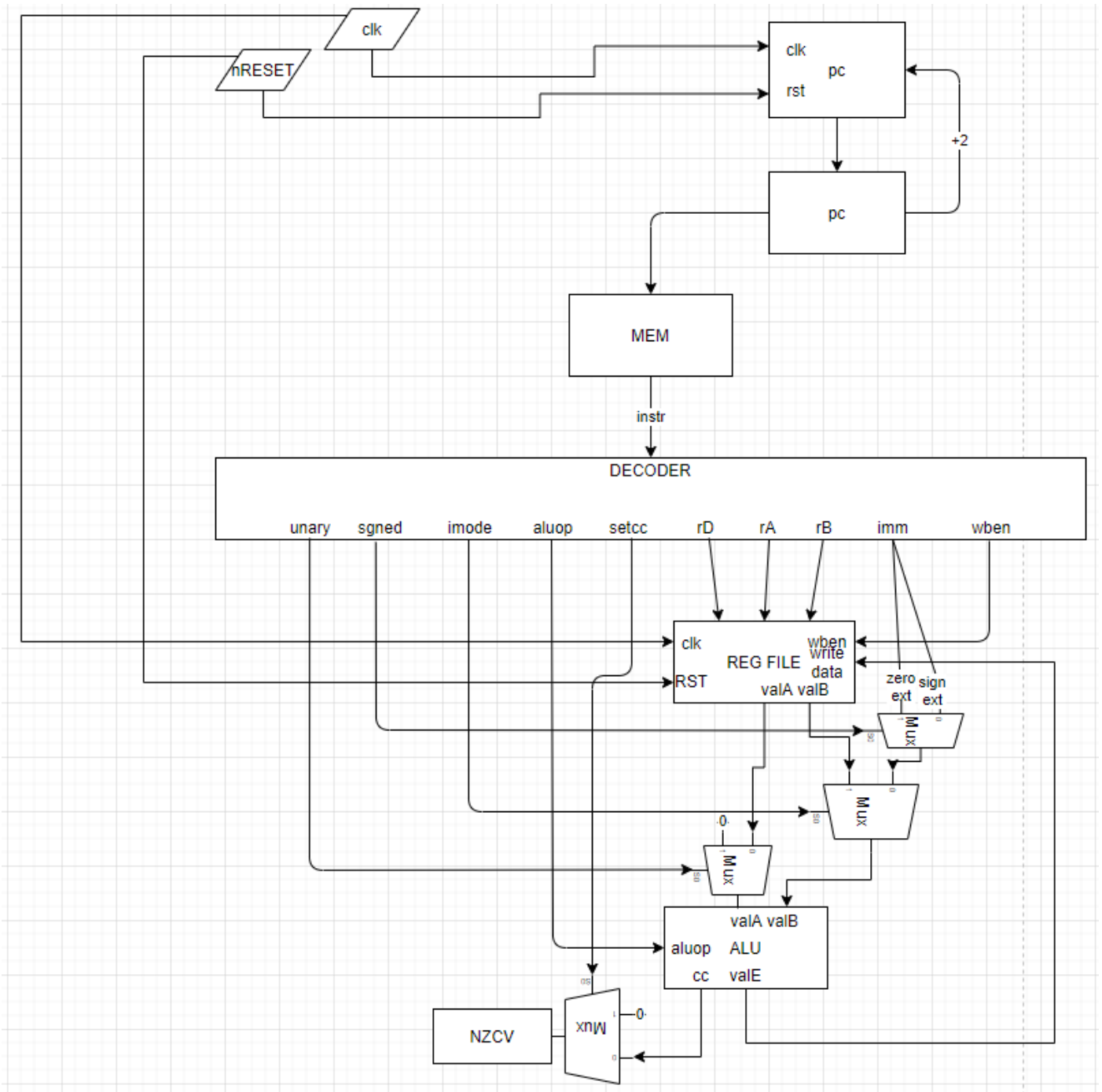
Class : 00

201602004 박태현

## I. 실습 목적

간단한 명령어를 인식하는 CPU의 구현

## II. Design Procedure



### III. Simulation

```
`define ADD      5'b00000
`define ADDI     5'b00001
`define SUB      5'b00010
`define SUBI     5'b00011
`define MOV      5'b00100
`define MOVI     5'b00101
`define SHL      5'b00110
`define SHLI     5'b00111
`define SHR      5'b01000
`define SHRI     5'b01001
`define ASR      5'b01010
`define ASRI     5'b01011
`define ROL      5'b01100
`define ROLI     5'b01101
`define ROR      5'b01110
`define RORI     5'b01111
`define AND      5'b10000
`define ANDI     5'b10001
`define OR       5'b10010
`define ORI      5'b10011
`define NOT |    5'b10100
`define MUL      5'b10101
`define MULI     5'b10110

`define READ 1'b1
`define WRITE 1'b0

`define add 4'b0001
`define sub 4'b0010
`define mov 4'b0011

`define shl 4'b0101
`define asr 4'b0110
`define shr 4'b0111

`define rol 4'b1000
`define ror 4'b1001

`define _and 4'b1011
`define _or 4'b1100
`define _not 4'b1110

`define mul 4'b1111
```

INSTRUCTION OPREATION과 ALU OPERATION을 정의해줍니다

```

module cpu (clk, nRESET);

    input clk;
    input nRESET;
    reg [15:0] pc;
    reg [ 3:0] NZCV;

    wire [15:0] instr;

    wire unary;
    wire sgned;
    wire imode;
    wire [3:0] aluop;
    wire setcc;
    wire [2:0] rD, rA, rB;
    wire [3:0] imm;
    wire wben;

    wire [15:0] valA, valB, svalA, svalB, valE;
    wire [ 3:0] cc;

```

Cpu의 입력과 내부 연결을 정의해줍니다

```

/* PC register */
always @(posedge clk or negedge nRESET)
    if (!nRESET) pc <= 16'b0;
    else pc <= pc+2;

/* instruction memory */
memory imem (clk, `READ, pc, instr);
/* instruction decoder */
decoder idec (instr, unary, sgned, imode, aluop, setcc, rD, rA, rB, imm, wben);
/* register file */
register_file ireg_file (clk, nRESET, wben, rD, valE, rA, rB, valA, valB);

/* ALU */
assign svalA = unary ? 16'b0 : valA;
assign svalB = imode ? sgned ? {{12{imm[3]}}, imm}/*sign ext*/ : {12'b0, imm}/*zero ext*/ : valB;
alu ialu (svalA, svalB, aluop, cc, valE);

/* condition code register */
always @(posedge clk or negedge nRESET) begin
    if (!nRESET) NZCV <= 4'b0;
    else if(setcc) NZCV <= cc;
    else NZCV <= 4'bx;
end

```

PC를 클럭에 따라 증가시켜줍니다

명령어를 읽어온 메모리와 읽어온 메모리를 디코딩할 디코더, 명령어의 결과를 잠시 저장해두는 레지스터를 정의하고 연결해줍니다

디코딩된 결과를 통해서 레지스터에서 값을 읽어오고, 그것을 alu에 넣어서 연산을 해줍니다.

그리고 그 결과를 valE로 내보내고, 그것이 다시 레지스터와 연결되어 있어서 레지스터에 값이 써집니다

```
module decoder (instr, unary, signed, imode, aluop, setcc, rD, rA, rB, imm, wben);
    input [15:0] instr;
    output reg unary;
    output reg signed;
    output reg imode;
    output reg setcc;
    output reg [3:0] aluop; // ADD,SUB,SHL,SHR,AND,OR,NOT (3 bits for 7 ops) output setcc;
    output reg [2:0] rD;
    output reg [2:0] rA;
    output reg [2:0] rB;
    output reg [3:0] imm;
    output reg wben;
    wire [4:0] opcode;

    assign opcode = instr[4:0];

    always @(instr)
    case (opcode)
        // OP : {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {unary,sgnd,imod,alop,setcc,rD,rA,rB,imm,wben}
        `ADD: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b1,1'b0,`add,instr[5],instr[8:6],instr[11:9],instr[14:12],4'bx,1'b1};
        `ADDI: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b1,1'b1,`add,instr[5],instr[8:6],instr[11:9],3'bx,instr[15:12],1'b1};
        `SUB: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b1,1'b0,`sub,instr[5],instr[8:6],instr[11:9],instr[14:12],4'bx,1'b1};
        `SUBI: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b1,1'b1,`sub,instr[5],instr[8:6],instr[11:9],3'bx,instr[15:12],1'b1};
        `MOV: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b1,1'b1,1'b0,`add,1'b0,instr[8:6],3'bx,instr[14:12],4'bx,1'b1};
        `MOVI: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b1,1'b1,1'b1,`add,1'b0,instr[8:6],3'bx,3'bx,instr[15:12],1'b1};
        `SHL: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b0,1'b0,`shl,instr[5],instr[8:6],instr[11:9],instr[14:12],4'bx,1'b1};
        `SHLI: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b0,1'b1,`shl,1'b0,instr[8:6],instr[11:9],3'bx,instr[15:12],1'b1};
        `SHR: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b0,1'b0,`shr,instr[5],instr[8:6],instr[11:9],instr[14:12],4'bx,1'b1};
        `SHRI: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b0,1'b1,`shr,1'b0,instr[8:6],instr[11:9],3'bx,instr[15:12],1'b1};
        `ASR: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b1,1'b0,`asr,instr[5],instr[8:6],instr[11:9],instr[14:12],4'bx,1'b1};
        `ASRI: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b1,1'b1,`asr,1'b0,instr[8:6],instr[11:9],3'bx,instr[15:12],1'b1};
        `ROL: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b0,1'b0,`rol,instr[5],instr[8:6],instr[11:9],instr[14:12],4'bx,1'b1};
        `ROLI: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b0,1'b1,`rol,1'b0,instr[8:6],instr[11:9],3'bx,instr[15:12],1'b1};
        `ROR: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b0,1'b0,`ror,instr[5],instr[8:6],instr[11:9],instr[14:12],4'bx,1'b1};
        `RORI: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b0,1'b1,`ror,1'b0,instr[8:6],instr[11:9],3'bx,instr[15:12],1'b1};
        `AND: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b0,1'b0,`and,instr[5],instr[8:6],instr[11:9],instr[14:12],4'bx,1'b1};
        `ANDI: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b0,1'b1,`and,1'b0,instr[8:6],instr[11:9],3'bx,instr[15:12],1'b1};
        `OR: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b0,1'b0,`or,instr[5],instr[8:6],instr[11:9],instr[14:12],4'bx,1'b1};
        `ORI: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b0,1'b1,`or,1'b0,instr[8:6],instr[11:9],3'bx,instr[15:12],1'b1};
        `NOT: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b1,1'b0,1'b0,`not,1'b0,instr[8:6],3'bx,instr[14:12],4'bx,1'b1};
        `MUL: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b1,1'b0,`mul,instr[5],instr[8:6],instr[11:9],instr[14:12],4'bx,1'b1};
        `MULI: {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = {1'b0,1'b1,1'b1,`mul,1'b0,instr[8:6],instr[11:9],3'bx,instr[15:12],1'b1};
        default : {unary,signed,imode,aluop,setcc,rD,rA,rB,imm,wben} = 22'bx;
    endcase
endmodule
```

명령어에 맞게 인스트럭션을 디코딩 해줍니다

여기서 나온 시그널을 토대로 alu를 연산하고, reg에서 읽거나 새로 쓰는 작업을 합니다

```
module memory (clk, op, addr, data);
    input clk;
    input op;
    input [15:0] addr;
    output [15:0] data;

    reg [7:0] mem [0:65535];

    assign data = {mem[addr], mem[addr+1]};

endmodule
```

메모리는 제공해준 대로 구현을 하였고, ALU, 레지스터 파일은 이전주차에 구현한 것을 사용했습니다

```

module cpu_tb;

    reg clk, nRESET;

    always #2.5 clk = ~clk;

    cpu icpu(clk, nRESET);

    integer i;

    initial begin
        $readmemb("my_program.txt",icpu.imem.mem);

        clk = 1'b1;
        nRESET = 1'b1;

        #5 nRESET = 1'b0;
        #5 nRESET = 1'b1;

        #500;
        $finish;
    end
endmodule

```

문서에 저장된 프로그램을 실행시키도록 했습니다

```

// prob 1
0010_xxx_0 00_0_00101 // movi $0 5
0101_xxx_0 01_0_00101 // movi $1 2
0001_xxx_0 10_0_00101 // movi $2 1
0011_xxx_0 11_0_00101 // movi $3 3
0_000_001_0 00_0_10101 // mul $0 $0 $1
0_010_011_0 01_0_10101 // mul $1 $2 $3
0_001_000_0 00_0_00010 // sub $0 $0 $1

```

$5*2-1*3=7$  이 \$0에 저장됩니다

```
// prob 2
0011_000_0 00_0_00101 // movi $0 3
0010_000_0 01_0_00101 // movi $1 2
0101_000_0 10_0_00101 // movi $2 5

0_000_000_0 00_0_10101 // mul $0 $0 $0
0_001_001_0 01_0_10101 // mul $1 $1 $1
0_010_010_0 10_0_10101 // mul $2 $2 $2

0_001_000_0 00_0_00000 // add $0 $0 $1
0_010_000_0 00_0_00000 // add $0 $0 $2

0001_000_0 00_0_01011 // asri $0 $0 1
```

$3*3+2*2+5*5=38$ ,  $38 > 1$ (나누기 2) 19가 \$0에 저장됩니다

```
// prob 3
1010_xxx_0 00_0_00101 // movi $0 0xA

1001_xxx_0 01_0_00101 // movi $1 0x9
0100_001_0 01_0_00111 // shli $1 $1 4
1010_001_0 01_0_10011 // ori $1 0xA
0100_001_0 01_0_00111 // shli $1 $1 4
1011_001_0 01_0_10011 // ori $1 0xB
0100_001_0 01_0_00111 // shli $1 $1 4
1100_001_0 01_0_10011 // ori $1 0xC

0_001_xxx_0 11_0_10100 // not $3 $1
0_000_xxx_0 10_0_10100 // not $2 $0

0_011_000_1 00_0_10000 // and $4 $0 $3
0_001_010_1 01_0_10000 // and $5 $2 $1
0_101_100_0 00_0_10010 // or $0 $4 $5
```

0xA를 레지스터에 넣으면서 부호확장을 통해 0xFFFF를 저장했습니다

0x9ABC를 9,A,B,C 순서대로 넣으면서 중간중간에 시프트를 하고 or을 하여 저장을 했습니다

Not 값을 임시로 저장하고 and or을 순서대로 해주었습니다

1111 1111 1111 1010

1001 1010 1011 1100

1111 1111 1111 1010

0110 0101 0100 0011 ) AND

0110 0101 0100 0010

0000 0000 0000 0101

1001 1010 1011 1100 ) AND

0000 0000 0000 0100

0110 0101 0100 0010

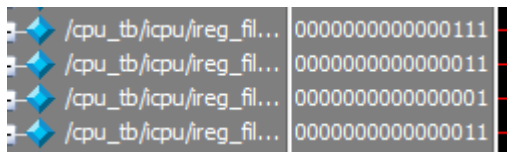
0000 0000 0000 0100 ) OR

0110 0101 0100 0110 = RES

요구사항의 세가지 프로그램을 작성하여 실행시켜보았습니다

#### IV. Evaluation

1.



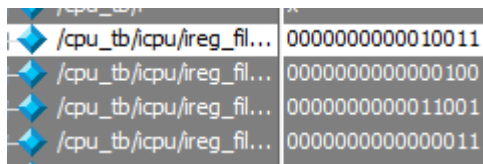
```
/cpu_tb/icpu/ireg_fil... 0000000000000111  
/cpu_tb/icpu/ireg_fil... 0000000000000011  
/cpu_tb/icpu/ireg_fil... 0000000000000001  
/cpu_tb/icpu/ireg_fil... 0000000000000011
```

$5*2-1*3=7$

위 명령을 실행 후 \$0에 7이 저장되는 것을 확인했습니다



2.

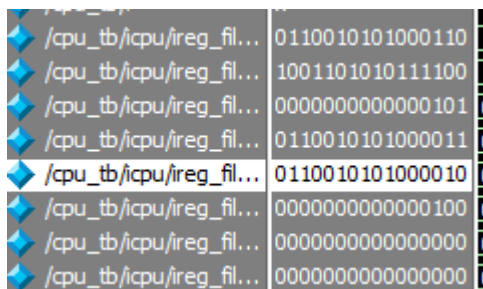


/cpu_tb/icpu/ireg_fil...	00000000000010011
/cpu_tb/icpu/ireg_fil...	0000000000000100
/cpu_tb/icpu/ireg_fil...	00000000000011001
/cpu_tb/icpu/ireg_fil...	0000000000000011

$(3*3+2*2+5*5)/2$ 를 실행 후 19가 나온 것을 확인했습니다

시프트로 나누기2를 수행했습니다

3.



/cpu_tb/icpu/ireg_fil...	0110010101000110
/cpu_tb/icpu/ireg_fil...	1001101010111100
/cpu_tb/icpu/ireg_fil...	0000000000000101
/cpu_tb/icpu/ireg_fil...	0110010101000011
/cpu_tb/icpu/ireg_fil...	0110010101000010
/cpu_tb/icpu/ireg_fil...	0000000000000100
/cpu_tb/icpu/ireg_fil...	0000000000000000
/cpu_tb/icpu/ireg_fil...	0000000000000000

비트연산이 잘 되는지 확인을 했습니다

위에서 계산한 결과가 \$0에 들어가 있는 것을 확인했습니다;다

## V. Discussion

이론으로만 배웠던 CPU를 직접 베릴로그로 설계를 해볼 수 있어서 좋은 경험이었습니다.