RISC-V 32bit Single Cycle RTL Design Project

1. 개요

- RV32I instruction Set을 활용하여 직접 Binary Instruction을 작성하고 RV32I 에 기반한 32bit Single Cycle Processor를 설계하고 명령어들 실행을 해본다.

- RV32I Instruction Set은 32bit의 크기인 명령어를 기준으로 하며, 이는 word가 32bit이며 32개의 레지스터, 레지스터의 크기 또한 32bit라는 의미이다.

- 해당 프로젝트는 MUX, Adder단 구현보단 실제 프로세서의 동작 과정에 초점을 맞추어 진행하였다.

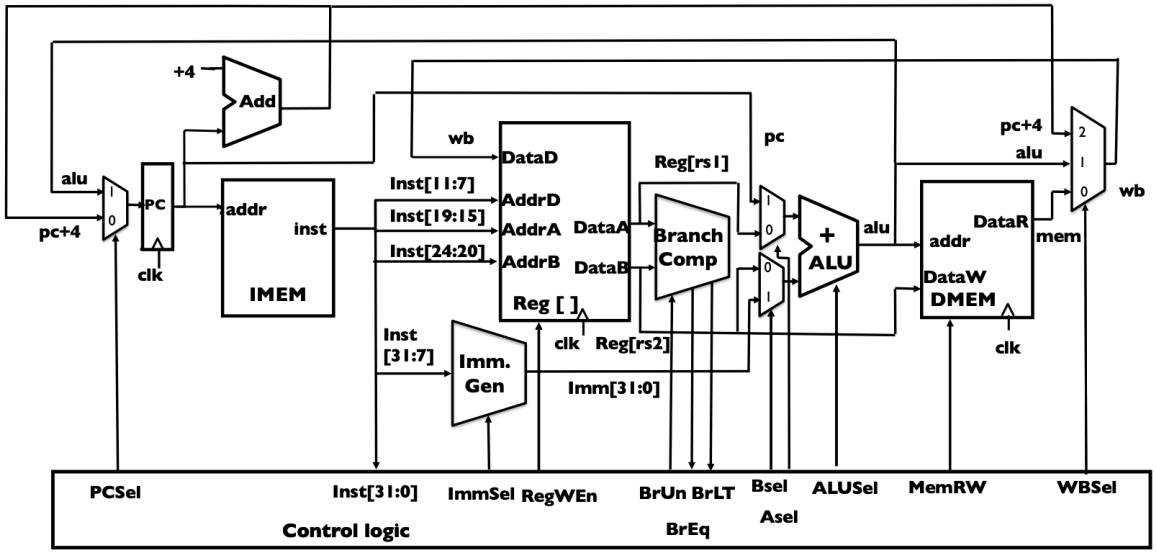


Fig 1. 전체 하드웨어 구성

- 해당 구성에서 나오는 Signal 및 Hardware들은 프로세서의 명령어 실행 과정에 따라 과정 모듈 내 존재하며 Signal 또한 Instruction에 따른 동작 과정에 맞춰 설계를 진행하였다.

2. 하드웨어 실행 순서 설명 및 하드웨어 설명

A. 실행 순서

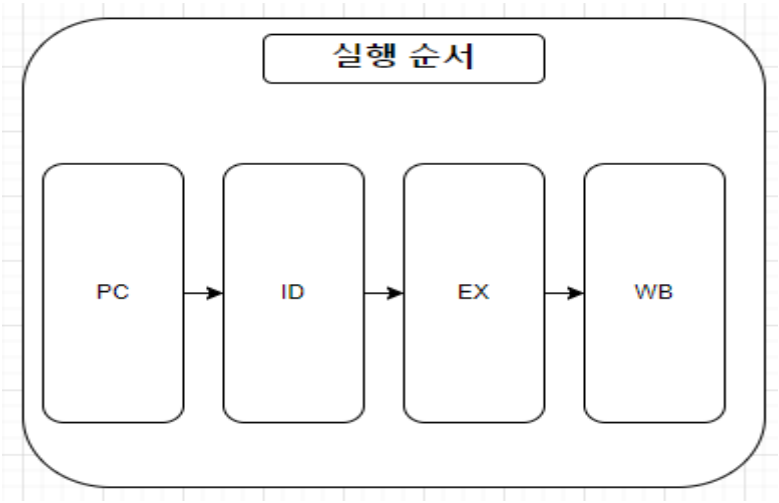


Fig 2. Single Cycle 실행 순서

Single Cycle Processor는 클럭과 동기화되어 한번의 클럭에 하나의 명령어를 실행하는 프로세서라 할 수 있다.

1) PC (Instruction Fetch) 단계

클럭의 posedge와 동기화되어 PC단계에서 Instruction Fetch 과정이 발생한다.

Instruction Fetch는 Program Counter 레지스터에 배치된 명령어 주소에 따라 명령어 메모리에서 명령어를 가져오는 단계이다. 이 단계가 PC module은 클럭과 동기화된 Sequential logic으로 작동하게 된다.

PC는 일반적으로 분기가 없을 시 4씩 증가하게 되는데 여기서 4는 4 Byte라는 의미이며, RV32I에서 32라는 수가 32bit인 word의 단위이면서 명령어의 길이를 의미한다.

명령어의 길이가 32비트라는 의미이고 이는 4바이트와 동일하다고 할 수 있다.

메모리는 일반적으로 바이트 단위의 주소체계를 가지기에 메모리에서 1은 1byte를 의미하고 그렇기에 메모리 주소체계를 맞추기 위해 +32가 아닌 +4씩 증가하게 되는 것이다.

Blt, Jal 같은 분기 명령어가 존재할 경우에는 PC값은 offset값에 맞춰 변화한다.

2) ID (Instruction Decode) 단계

ID는 Instruction Decode 단계이다. Instruction Fetch를 통해 Instruction에 대해 정보를 알게 되고 그에 따라 명령어 실행을 위해 하드웨어에 Control Signal을 보내는 단계라 할 수 있다. Fig 1을 참고하면 하드웨어에 대한 그림이 아닌 실행순서에 따라 나뉘어져 있는데 여기서 ID 단계에서 필요한 하드웨어들에 대한 Signal들을 조정해준다.

예를 들어 ADD같은 연산을 처리하기 위해 Register를 읽고 쓰기 위한 Signal들이 RegFile과 ALU 등에 보내지는 것이다.

3) EX (Execute) 단계

일반적으로 ALU 장치의 연산이 이뤄진다고 볼 수 있는 단계로 ID 단계에서 보낸 Signal에 따라 필요한 연산 및 처리가 이뤄진다고 볼 수 있다.

4) WB (Write Back) 단계

WriteBack 단계로 Register에 존재하는 데이터를 메모리 및 Register File로 다시 쓰는 작업을 할 때 사용되는 부분이다.

B. 하드웨어 구성

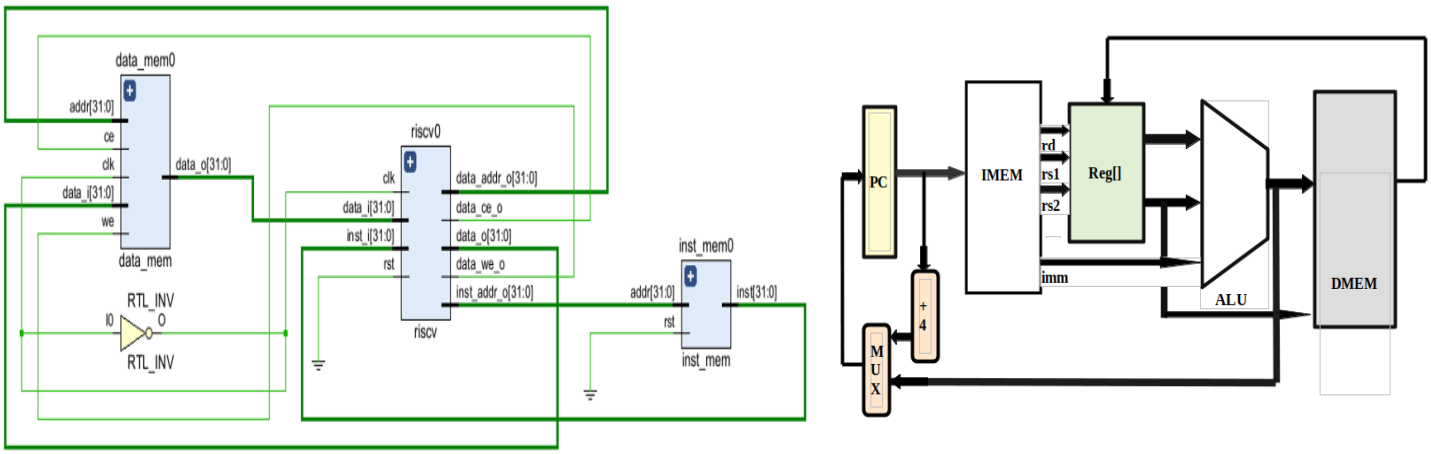


Fig 3. 하드웨어 구성 (RTL Schematic , block diagram)

일반적인 하드웨어 구성을 살펴보면 PC내 명령어 메모리 주소를 활용하여 명령어 메모리를 받는다. 일반적으로 RV32I 명령어의 구조는 6가지의 종류로 나눌 수 있지만 본 프로젝트에서는 R, I , B, J 이렇게 큰 4가지 분류만 설계하였다.

R 종류 명령어는 2개의 레지스터(rs1, rs2)를 활용하여 결과값을 저장 레지스터(rd)에 저장하는 명령어라 할 수 있으며 ADD, SUB 같은 명령에 해당한다.

I 종류 명령어는 하나의 레지스터(rs1)과 immediate 값을 활용하여 결과값을 저장 레지스터 (rd)에 저장하는 명령어라 할 수 있으며 ADDI 같은 명령어가 존재한다.

B 종류 명령어는 두개의 레지스터를 비교하여 결과에 따라 명령어 주소를 바꾸는 명령어이며 명령어 주소는 immediate 값에 따라 바뀌게 된다.

(Branch 명령어에 해당한다.)

다음으로 J 명령어는 명령어 주소를 바꾸는 명령어에 해당한다. 위 프로젝트에서는 Jal 명령을 활용할 것이며 Jal 명령어는 jump and link로 현재 PC +4의 값을 레지스터에 저장하고 원하는 주소(immediate)로 이동하는 명령어이다.

JAL 명령은 주로 함수 호출 및 goto문 등에 사용되며 Stack 메모리 사용과 관련이 깊다.

3. 베릴로그 코드 구현 및 설명

```
module PC(
    input   wire        clk,
    input   wire        rst,
    input   wire        Branch,  // if branch or not
    input   wire[31:0]  Addr,    // target address
    output  reg         ce,
    output  reg [31:0]  PC

);

always @ (posedge clk) begin
    if (rst)
        ce <= 1'b0;
    else
        ce <= 1'b1;
end
always @ (posedge clk) begin
    if (rst)
        PC <= 32'b0;
    else if (Branch)
        PC <= Addr;
    else
        PC <= PC + 32'd4; // 32bit single cycle  ->  inst length -> 4byte
end

endmodule
```

Fig 4. PC module

PC 단계를 구현하기 위한 모듈로 명령어 메모리 주소를 가리키는 곳이라 할 수 있다.

먼저 clock에 동기화 시켜 프로세서를 작동시키기 위해 명령어 주소 변화를 clock과 동기화 시켜 주었다. Rst는 reset을 의미한다.

Signal ce는 PC값의 변화가 유효한지를 의미하는 것으로 프로세서가 동작하고 있는지 안하고 있는지 확인하게 해주는 모듈이라 할 수 있다.

ce값은 Instruction Memory Module로 가게 되며 이 값을 받아 해당 주소의 명령어를 보내준다.

Branch는 앞서 말한 B, J 명령어의 실행 결과를 반영하기 위해 존재하며 Branch가 나타나지 않을 경우에는 PC값은 클럭에 동기화되어 4씩 증가하게 된다. 이 4라는 값은 4byte를 의미하며 명령어 메모리 주소 체계에 영향 받는다.

```
module ID(

    input   wire        rst,
    input   wire[31:0]  pc_i,
    input   wire[31:0]  inst_i,

    input   wire[31:0]  RegData1,
    input   wire[31:0]  RegData2,
    output  reg         RegRead1,     // if read register or not
    output  reg         RegRead2,     // if read register or not
    output  reg [4:0]   RegAddr1,
    output  reg [4:0]   RegAddr2,

    output  reg [4:0]   ALUop,
    output  reg [31:0]  Reg1,
    output  reg [31:0]  Reg2,
    output  reg [4:0]   WriteData,
    output  reg         WriteReg,     // if write register or not

    output  reg         Branch,       // if branch or not
    output  reg [31:0]  BranchAddr,   // branch address
    output  reg [31:0]  LinkAddr,     // link address for jal
    output  wire[31:0]  inst_o

);

    reg inst_valid;
    reg [31:0] imm;

    wire[31:0] pc_add_4;
    wire[31:0] pc_add_imm_B;
    wire[31:0] pc_add_imm_J;
    wire[4:0]  rs1_addr = inst_i[19:15];
    wire[4:0]  rs2_addr = inst_i[24:20];
    wire[4:0]  rd_addr  = inst_i[11:7];


    wire[31:0] imm_I = {{21{inst_i[31:31]}}, inst_i[30:20]};
    wire[31:0] imm_B = {{20{inst_i[31:31]}}, inst_i[ 7: 7], inst_i[30:25], inst_i[11:8], 1'b0};
    wire[31:0] imm_J = {{12{inst_i[31:31]}}, inst_i[19:12], inst_i[20:20], inst_i[30:25], inst_i[24:21], 1'b0};
```

Fig 5. ID-1

해당 모듈은 ID 단계의 모듈의 일부만 나타내고 있다.

ID모듈은 클럭에 동기화되지 않고 주어진 명령어에 따라 signal배치를 즉각적으로 이뤄지게 된다.

pc_i 값은 현재 PC값을 의미하고 Inst_i는 PC값을 통해 Inst_MEM module에서 들어오는 명령어 값이라 할 수 있다.

RegData1 , RegData2는 rs1, rs2를 의미하며 RegRead1, RegRead2 는 해당 Register값을 읽어야 하는지에 대한 Signal을 의미한다.

RegAddr들은 RegFile 모듈에서 사용되는 레지스터 번호를 가리키게 된다.

ALUop는 ALU연산 종류를 나타내게 되며 Controller라 할 수 있는 ID 모듈에서 명령어에 따라 ALU 연산 종류를 정하게 된다. Reg1, Reg2는 ALU 모듈에서 연산으로 사용될 레지스터들의 값을 나타내고 있다.

WriteData는 저장 레지스터의 번호(주소)가 들어가 있으며 WriteReg Signal은 ALU결과 값 및 주소 분기 값을 Reg에 써야하는 지 를 나타낸다.

Branch는 JAL 혹은 blt 같은 B,J 명령어가 존재하여 분기가 필요한 지에 대한 값을 설정해 놓은 것이고 blt의 결과값이 False라도 일단 controller는 Branch Signal을 보내 미리 대비하게 만들어 준다.

```
reg inst_valid;
reg [31:0] imm;

wire[31:0] pc_add_4;
wire[31:0] pc_add_imm_B;
wire[31:0] pc_add_imm_J;
wire[4:0]  rs1_addr = inst_i[19:15];
wire[4:0]  rs2_addr = inst_i[24:20];
wire[4:0]  rd_addr  = inst_i[11:7];


wire[31:0] imm_I = {{21{inst_i[31:31]}}, inst_i[30:20]};
wire[31:0] imm_B = {{20{inst_i[31:31]}}, inst_i[ 7: 7], inst_i[30:25], inst_i[11:8], 1'b0};
wire[31:0] imm_J = {{12{inst_i[31:31]}}, inst_i[19:12], inst_i[20:20], inst_i[30:25], inst_i[24:21], 1'b0};

assign inst_o = inst_i;
assign pc_add_4 = pc_i + 4;
assign pc_add_imm_B = pc_i + imm_B;
assign pc_add_imm_J = pc_i + imm_J;
```

Fig 6. ID-2

위 내용은 주소 분기가 어떻게 이뤄지는 지 나타내고 있다. IMM_I는 I 명령어 형식의 immediate 값이 무엇인지를 담고 있고 각각 Imm_B, Imm_J 또한 명령어 형식에 맞춰 Imme값이 무엇인지 받

아오고 있다.

여기서 독특한 것이 B와 J의 값만 뒤에 비트를 추가하여 2씩 곱해주고 있는데 이는 B와 J의 명령어 형식에서 immediate값이 2가 곱해진 상태에서 설정되기 때문이다.

분기 주소 값은 현재 PC값에 imme 값을 더해주는 것으로 나타내며 그에 따라 해당 주소들을 미리 할당해준다.

```
always @ (*) begin
    if (rst)
        ALUop <= 5'b0;
    else begin
        casex (inst_i)
            32'bxxxxxxxxxxxxxxxxxxxxxxxxx1101111: ALUop <= 5'b10000;  // jal
            32'bxxxxxxxxxxxxxxxxx000xxxxx1100011: ALUop <= 5'b10001;  // beq
            32'bxxxxxxxxxxxxxxxxx100xxxxx1100011: ALUop <= 5'b10010;  // blt
            32'bxxxxxxxxxxxxxxxxx010xxxxx0000011: ALUop <= 5'b10100;  // lw
            32'bxxxxxxxxxxxxxxxxx010xxxxx0100011: ALUop <= 5'b10101;  // sw
            32'bxxxxxxxxxxxxxxxxx000xxxxx0010011: ALUop <= 5'b01100;  // addi
            32'b0000000xxxxxxxxxx000xxxxx0110011: ALUop <= 5'b01101;  // add
            32'b0100000xxxxxxxxxx000xxxxx0110011: ALUop <= 5'b01110;  // sub
            32'b0000000xxxxxxxxxx001xxxxx0110011: ALUop <= 5'b01000;  // sll
            32'b0000000xxxxxxxxxx100xxxxx0110011: ALUop <= 5'b00110;  // xor
            32'b0000000xxxxxxxxxx101xxxxx0110011: ALUop <= 5'b01001;  // srl
            32'b0000000xxxxxxxxxx110xxxxx0110011: ALUop <= 5'b00101;  // or
            32'b0000000xxxxxxxxxx111xxxxx0110011: ALUop <= 5'b00100;  // and
            default: ALUop <= 5'b0;
        endcase
    end
end
```

```
always @ (*) begin
    if (rst)
        WriteReg <= 1'b0;
    else begin
        casex (inst_i)
            32'bxxxxxxxxxxxxxxxxxxxxxxxxx1101111: WriteReg <= 1'b1;  // jal
            32'bxxxxxxxxxxxxxxxxx000xxxxx1100011: WriteReg <= 1'b0;  // beq
            32'bxxxxxxxxxxxxxxxxx100xxxxx1100011: WriteReg <= 1'b0;  // blt
            32'bxxxxxxxxxxxxxxxxx010xxxxx0000011: WriteReg <= 1'b1;  // lw
            32'bxxxxxxxxxxxxxxxxx010xxxxx0100011: WriteReg <= 1'b0;  // sw
            32'bxxxxxxxxxxxxxxxxx000xxxxx0010011: WriteReg <= 1'b1;  // addi
            32'b0000000xxxxxxxxxx000xxxxx0110011: WriteReg <= 1'b1;  // add
            32'b0100000xxxxxxxxxx000xxxxx0110011: WriteReg <= 1'b1;  // sub
            32'b0000000xxxxxxxxxx001xxxxx0110011: WriteReg <= 1'b1;  // sll
            32'b0000000xxxxxxxxxx100xxxxx0110011: WriteReg <= 1'b1;  // xor
            32'b0000000xxxxxxxxxx101xxxxx0110011: WriteReg <= 1'b1;  // srl
            32'b0000000xxxxxxxxxx110xxxxx0110011: WriteReg <= 1'b1;  // or
            32'b0000000xxxxxxxxxx111xxxxx0110011: WriteReg <= 1'b1;  // and
            default: WriteReg <= 1'b0;
        endcase
    end
```

Fig 7. ID-3

Fig ID-3은 명령어에 따라 ALUop, WriteReg의 Signal을 결정해준다.

일반적으로 WriteReg는 Jal, lw(load word), R, I 형식 명령어에 필수적으로 사용된다고 할 수 있다.

(WriteReg 신호가 존재하면 Register File내 Register에 값을 쓴다.)

```
always @ (*) begin
    if (rst)
        inst_valid <= 1'b0;
    else begin
        casex (inst_i)
            32'bxxxxxxxxxxxxxxxxxxxxxxxxx1101111: inst_valid <= 1'b0;  // jal
            32'bxxxxxxxxxxxxxxxxx000xxxxx1100011: inst_valid <= 1'b0;  // beq
            32'bxxxxxxxxxxxxxxxxx100xxxxx1100011: inst_valid <= 1'b0;  // blt
            32'bxxxxxxxxxxxxxxxxx010xxxxx0000011: inst_valid <= 1'b0;  // lw
            32'bxxxxxxxxxxxxxxxxx010xxxxx0100011: inst_valid <= 1'b0;  // sw
            32'bxxxxxxxxxxxxxxxxx000xxxxx0010011: inst_valid <= 1'b0;  // addi
            32'b0000000xxxxxxxxxx000xxxxx0110011: inst_valid <= 1'b0;  // add
            32'b0100000xxxxxxxxxx000xxxxx0110011: inst_valid <= 1'b0;  // sub
            32'b0000000xxxxxxxxxx001xxxxx0110011: inst_valid <= 1'b0;  // sll
            32'b0000000xxxxxxxxxx100xxxxx0110011: inst_valid <= 1'b0;  // xor
            32'b0000000xxxxxxxxxx101xxxxx0110011: inst_valid <= 1'b0;  // srl
            32'b0000000xxxxxxxxxx110xxxxx0110011: inst_valid <= 1'b0;  // or
            32'b0000000xxxxxxxxxx111xxxxx0110011: inst_valid <= 1'b0;  // and
            default: inst_valid <= 1'b1;
        endcase
    end
end
```

```
always @ (*) begin
    if (rst)
        RegRead1 <= 1'b0;
    else begin
        casex (inst_i)
            32'bxxxxxxxxxxxxxxxxxxxxxxxxx1101111: RegRead1 <= 1'b0;  // jal
            32'bxxxxxxxxxxxxxxxxx000xxxxx1100011: RegRead1 <= 1'b1;  // beq
            32'bxxxxxxxxxxxxxxxxx100xxxxx1100011: RegRead1 <= 1'b1;  // blt
            32'bxxxxxxxxxxxxxxxxx010xxxxx0000011: RegRead1 <= 1'b1;  // lw
            32'bxxxxxxxxxxxxxxxxx010xxxxx0100011: RegRead1 <= 1'b1;  // sw
            32'bxxxxxxxxxxxxxxxxx000xxxxx0010011: RegRead1 <= 1'b1;  // addi
            32'b0000000xxxxxxxxxx000xxxxx0110011: RegRead1 <= 1'b1;  // add
            32'b0100000xxxxxxxxxx000xxxxx0110011: RegRead1 <= 1'b1;  // sub
            32'b0000000xxxxxxxxxx001xxxxx0110011: RegRead1 <= 1'b1;  // sll
            32'b0000000xxxxxxxxxx100xxxxx0110011: RegRead1 <= 1'b1;  // xor
            32'b0000000xxxxxxxxxx101xxxxx0110011: RegRead1 <= 1'b1;  // srl
            32'b0000000xxxxxxxxxx110xxxxx0110011: RegRead1 <= 1'b1;  // or
            32'b0000000xxxxxxxxxx111xxxxx0110011: RegRead1 <= 1'b1;  // and
            default: RegRead1 <= 1'b0;
        endcase
    end
end
```

Fig 8. ID-4

Inst_valid는 명령어의 유효성을 의미한다. 미리 설정해준 명령어 종류가 아닌 다른 종류 혹은 잘못된 명령어가 들어올 경우 해당 명령어 실행을 막아 오류를 방지한다.

RegRead1은 rs1에 대한 read의 필요성을 나타낸다. 미리 정해준 명령어를 분석하여 필요하면 RegRead1 값은 1로, 아닌 경우에는 RegRead2 값은 0 으로 설정된다.

```
always @ (*) begin
    if (rst)
        RegRead2 <= 1'b0;
    else begin
        casex (inst_i)
            32'bxxxxxxxxxxxxxxxxxxxxxxxx1101111: RegRead2 <= 1'b0;  // jal
            32'bxxxxxxxxxxxxxxxxx000xxxxx1100011: RegRead2 <= 1'b1;  // beq
            32'bxxxxxxxxxxxxxxxxx100xxxxx1100011: RegRead2 <= 1'b1;  // blt
            32'bxxxxxxxxxxxxxxxxx010xxxxx0000011: RegRead2 <= 1'b0;  // lw
            32'bxxxxxxxxxxxxxxxxx010xxxxx0100011: RegRead2 <= 1'b1;  // sw
            32'bxxxxxxxxxxxxxxxxx000xxxxx0010011: RegRead2 <= 1'b0;  // addi
            32'b0000000xxxxxxxxxx000xxxxx0110011: RegRead2 <= 1'b1;  // add
            32'b0100000xxxxxxxxxx000xxxxx0110011: RegRead2 <= 1'b1;  // sub
            32'b0000000xxxxxxxxxx001xxxxx0110011: RegRead2 <= 1'b1;  // sll
            32'b0000000xxxxxxxxxx100xxxxx0110011: RegRead2 <= 1'b1;  // xor
            32'b0000000xxxxxxxxxx101xxxxx0110011: RegRead2 <= 1'b1;  // srl
            32'b0000000xxxxxxxxxx110xxxxx0110011: RegRead2 <= 1'b1;  // or
            32'b0000000xxxxxxxxxx111xxxxx0110011: RegRead2 <= 1'b1;  // and
            default: RegRead2 <= 1'b0;
        endcase
    end
end
```

```
always @ (*) begin
    if (rst)
        imm <= 32'b0;
    else if (inst_i[14:12] == 3'b000 && inst_i[6:0] == 7'b0010011)  // addi
        imm <= imm_i;
    else
        imm <= 32'b0;
end
```

```
always @ (*) begin
    if (rst)
        LinkAddr <= 32'b0;
    else if (inst_i[6:0] == 7'b1101111)  // jal
        LinkAddr <= pc_add_4; // store next PC . for return baseCode after poping stack
    else
        LinkAddr <= 32'b0;
end
```

```
always @ (*) begin
    if (rst)
        BranchAddr <= 32'b0;

    else if (inst_i[6:0] == 7'b1101111)  // jal
        BranchAddr <= pc_add_imm_J; // Address where to go?

    else if (inst_i[6:0] == 7'b1100011 && inst_i[13:12] == 2'b0) begin // beq, blt
        if(RegData1 <RegData2) begin
            BranchAddr <= pc_add_imm_B;
        end
        else if (RegData1 ==RegData2)begin
            BranchAddr <= pc_add_imm_B;
        end
        else begin
            BranchAddr <= pc_add_4;
        end
    end
    else
        BranchAddr <= 32'b0;
end
```

Fig 9. ID-5

먼저 RegRead2는 위의 RegRead1과 동일한 역할을 수행한다.

다음으로 imme를 살펴보면 imm 변수는 I 형식 명령어 처리를 위해 존재한다.

I 형식 명령어에는 ADDI, SUBI등이 존재하지만 해당 프로젝트에선 ADDI 만을 설정하여 진행하였다.

다음으로 LinkAddr변수는 JAL 발생시 현재 명령어의 주소 + 4의 값을 저장 레지스터에 저장해야 하는데 이때의 값을 설정하는 변수라 할 수 있다.

즉, Jal명령어 분기가 존재할 경우 PC+4값을 저장하게 된다.

다음으로 BranchAddr는 명령어 형식 B와 J에서 발생하는 명령어주소 변경에 따른 주소 값을 저장하기 위한 변수라 할 수 있다.

J명령어에서 JAL은 무조건 분기이기에 BranchAddr값은 PC+ imm_j 값을 저장하게 되지만

B형식은 조건부 분기이기에 조건에 따라 명령어 주소를 PC + 4 저장할지 아니면 PC + imm_b를 저장할 지 결정되는 것이다. 해당 프로젝트는 명령어로 blt를 구성하여 입력해주었다.

```
always @ (*) begin
    if (rst)
        Branch <= 1'b0;
    else if (inst_i[6:0] == 7'b1101111 || (inst_i[6:0] == 7'b1100011 && inst_i[13:12] == 2'b01))  // jal, beq, blt
        Branch <= 1'b1;
    else
        Branch <= 1'b0;
end

always @ (*) begin
    if (rst)
        WriteData <= 5'b0;
    else
        WriteData <= rd_addr;
end
// Register 1  address setting
always @ (*) begin
    if (rst)
        RegAddr1 <= 5'b0;
    else
        RegAddr1 <= rs1_addr;
end
// Register 2 address setting
always @ (*) begin
    if (rst)
        RegAddr2 <= 5'b0;
    else
        RegAddr2 <= rs2_addr;
end
```

```
// RegRead exists -> Read Reg Data using RegAddr
always @ (*) begin
    if (rst)
        Reg1 <= 32'b0;
    else if (RegRead1)
        Reg1 <= RegData1;
    else
        Reg1 <= imm;
end

always @ (*) begin
    if (rst)
        Reg2 <=  32'b0;
    else if (RegRead2)
        Reg2 <= RegData2;
    else
        Reg2 <= imm;
end

endmodule
```

Fig 10. ID-6

먼저 branch는 blt 혹은 jal등 J 혹은 B 형식 명령어가 존재할 때 Branch signal을 할당시켜주게 된다.

WriteData에서는 저장레지스터 (rd)에 대한 레지스터 번호(주소)가 저장되게 된다.

RegAddr1 과 RegAddr2는 각각 피연산자로 사용되는 레지스터의 번호(주소)를 가지게 된다. RegData1, RegData2는 각각 RegAddr1, RegAddr2의 값이 바뀜에 따라 바뀌게 되는 값들이고 RegFile이라는 모듈에서 들어오는 값이라 할 수 있다. 해당 값들을 Reg1, Reg2에 배치해주어 EX 모듈에서 사용되도록 만들어준다.

Reg1, Reg2는 ALU 연산이 이뤄지는 EX 모듈에서 피연산자로 사용되며 I 형식 명령어 또한 고려하기 위해 RegRead1, RegRead2 Signal에 따라 Reg1, Reg2에 유동적으로 imme를 배치하게 된다.

해당 프로젝트에서 Reg1에 imme가 배치될 경우는 존재하지 않을 것이다.

I형식 명령어 ADDI의 경우 rs2가 아닌 rs1을 사용하기에 Reg1에는 무조건 레지스터내 값이 배치가 되기 때문이다.

```
3  `timescale 1ns / 1ps
4  module Registers(
5      input   wire clk, rst, we,
6      input   wire[4:0]   WriteAddr,
7      input   wire[31:0]  WriteData,
8      input   wire ReadReg1, ReadReg2,
9      input   wire[4:0]   ReadAddr1,
10     input   wire[4:0]   ReadAddr2,
11     output  reg [31:0]  ReadData1,
12     output  reg [31:0]  ReadData2
13     );
14     integer i;
15     reg [31:0] regFile [0:32]; // register file setting
16 always @ (posedge clk) begin
17     regFile[32'b0] <= 32'b0;
18     if (rst)
19         for (i = 0; i < 32; i = i + 1)
20             regFile[i] <= 32'b0; // reset all reg to zero
21
22     if (!rst && we && WriteAddr != 5'h0) begin
23         regFile[WriteAddr] <= WriteData;
24     end
25 end
26 always @ (*) begin
27     if (rst || ReadAddr1 == 5'h0)
28         ReadData1 <= 32'b0;
29     else if (ReadReg1) begin
30         ReadData1 <= regFile[ReadAddr1];
31     end else
32         ReadData1 <= 32'b0;
33 end
34 always @ (*) begin
35     if (rst || ReadAddr2 == 5'h0)
36         ReadData2 <= 32'b0;
37     else if (ReadReg2) begin
38         ReadData2 <= regFile[ReadAddr2];
39     end else
40         ReadData2 <= 32'b0;
41 end
42 endmodule
```

Fig 11. Register File

32 bit Processor의 경우 크기가 32bit인 32개의 레지스터를 일반적으로 가지게 된다. 그에 따라 Register 집합으로 Register File이라는 모듈을 작성하여 전체 레지스터의 쓰고 읽기를 담당하도록 설정해주었다.

Register File(이하 RF)에서는 Clock에 동기화되어 데이터를 쓰는 작업이 진행되게 된다. 이는 Register에서 데이터를 읽는 과정에서 쓰는 과정이 클럭 동기화 없이 즉각적으로 일어나게 된다면 읽는 동작과 쓰는 동작이 같은 Register에서 발생할 경우 Data Hazard가 발생하기에 이를 방지하기 위해 레지스터에 쓰는 작업은 Clock에 동기화하여 진행하였다.

이로써, ADD x1, x2, x1같은 작업을 진행할 때, x1의 데이터에 동시적 읽기 쓰기가 진행되지 않음으로써 데이터의 모호함을 없앨 수 있었다.

나머지 RegRead 동작은 clock에 동기화하지 않고 즉각적으로 이루어지게 진행하였다.

프로세서인 CPU는 직접적으로 메모리(DRAM)공간의 데이터를 사용할 수 없고 레지스터를 통해 연산을 진행한다.

```
`timescale 1ns / 1ps
module EX(
    input  wire        rst,
    input  wire[4:0]   ALUop_i,
    input  wire[31:0]  Oprend1,
    input  wire[31:0]  Oprend2,
    input  wire[4:0]   WriteDataNum_i,
    input  wire        WriteReg_i,
    input  wire[31:0]  LinkAddr,
    input  wire[31:0]  inst_i,
    output reg         WriteReg_o,
    output wire[4:0]   ALUop_o,
    output reg [4:0]   WriteDataNum_o,
    output reg [31:0]  WriteData_o,
    output wire[31:0]  MemAddr_o,
    output wire[31:0]  Result
);

always @ (*) begin
    if (rst)
        WriteDataNum_o <= 5'b0;
    else
        WriteDataNum_o <= WriteDataNum_i;
end

always @ (*) begin
  if (rst)
    WriteReg_o <= 1'b0;
  else
    WriteReg_o <= WriteReg_i;
end
```

```
always @ (*) begin
  if (rst)
    WriteReg_o <= 1'b0;
  else
    WriteReg_o <= WriteReg_i;
end


always @ (*) begin
  if (rst)
    WriteData_o <= 32'b0;
  else begin
    case (ALUop_i)
      5'b10000: WriteData_o <= LinkAddr;                        // jal
      5'b10001: WriteData_o <= LinkAddr;                       // beq
      5'b10010: WriteData_o <= LinkAddr;                        // blt
      5'b10100: WriteData_o <= 32'b0;                  // lw
      5'b10101: WriteData_o <= 32'b0;                  // sw
      5'b01100: WriteData_o <= Oprend1 +  Oprend2;          // addi
      5'b01101: WriteData_o <= Oprend1 +  Oprend2;          // add
      5'b01110: WriteData_o <= Oprend1 -  Oprend2;          // sub
      5'b01000: WriteData_o <= Oprend1 << Oprend2[4:0]; // sll
      5'b00110: WriteData_o <= Oprend1 ^  Oprend2;        // xor
      5'b01001: WriteData_o <= Oprend1 >> Oprend2[4:0]; // srl
      5'b00101: WriteData_o <= Oprend1 |  Oprend2;          // or
      5'b00100: WriteData_o <= Oprend1 &  Oprend2;          // and
      default:  WriteData_o <= 32'b0;
    endcase
  end
end

assign ALUop_o   = ALUop_i;
assign Result    = Oprend2;

// Whether LW or SW  , LW ->  [6:0] == 0000011, SW -> [6:0] == 0100011
// For write or load
assign MemAddr_o = Oprend1 + ((inst_i[6:0] == 7'b0000011) ? {{20{inst_i[31:31]}},
       inst_i[31:20]} :  {{20{inst_i[31:31]}}, inst_i[31:25], inst_i[11:7]});

endmodule
```

Fig 12. EX Module

EX 모듈은 실제 ALU 역할이 이뤄지는 모듈이라 할 수 있다.

EX 단계는 execute 단계로 명령어를 실제 수행하는 부분이라 할 수 있다.

여기서 나오는 Operand는 ID 단계에서의 IMME 값 혹은 Reg1, Reg2의 값에 해당하게 된다. 해당 ALU 연산들은 즉각적으로 이뤄지게 되어 있으며 MemAddr 변수가 나타내는 값은 SW 혹은 LW 명령어에 따라 달라지는 Addr 값을 나타내기 위함이다.

MemAddr 변수는 Memory Controller 모듈로 전달되게 되고 이 값을 참고하여 Memory Controller 는 데이터 메모리에서 원하는 데이터를 참조한다.

lw

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:0]			rs1	010	rd	00000	11

Format	lw rd,offset(rs1)
Description	Loads a 32-bit value from memory and sign-extends this to XLEN bits before storing it in register rd.
Implementation	x[rd] = sext(M[x[rs1] + sext(offset)][31:0])

Fig 13. LW instruction

sw

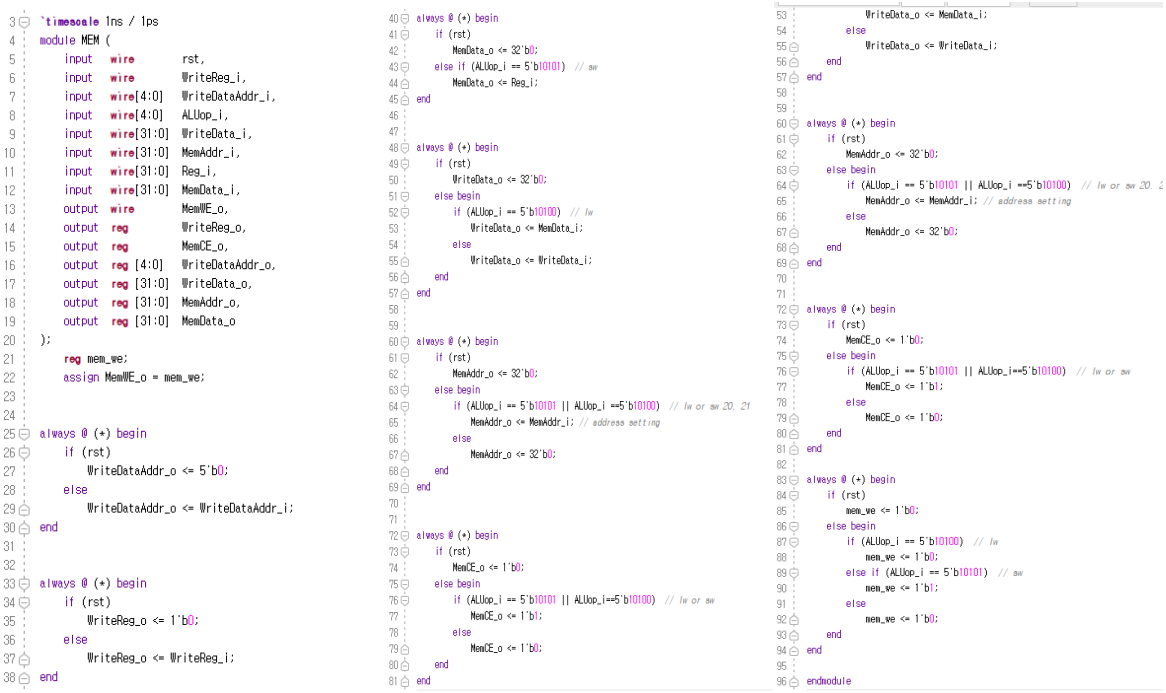
31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:5]		rs2	rs1	010	offset[4:0]	01000	11

Format	sw rs2,offset(rs1)
Description	Store 32-bit, values from the low bits of register rs2 to memory.
Implementation	M[x[rs1] + sext(offset)] = x[rs2][31:0]

Fig 14. SW instruction

LW 명령어는 offset값을 [31:20] 을 이용하게 된다. 반면 sw는 offset값을 [31:25], [11:7]까지 이용하게 된다.

이러한 차이점으로 인해 MemAddr 값을 서로 다르게 설정해주었다. 해당 프로젝트는 word 단위의 데이터 쓰기, 읽기를 지원하기에 byte 단위 및 half word 단위는 코드내 수정이 필요하다.



```
`timescale 1ns / 1ps
module MEM (
    input   wire        rst,
    input   wire        WriteReg_i,
    input   wire[4:0]   WriteDataAddr_i,
    input   wire[4:0]   ALUop_i,
    input   wire[31:0]  WriteData_i,
    input   wire[31:0]  MemAddr_i,
    input   wire[31:0]  Reg_i,
    input   wire[31:0]  MemData_i,
    output  wire        MemWE_o,
    output  reg         WriteReg_o,
    output  reg         MemCE_o,
    output  reg [4:0]   WriteDataAddr_o,
    output  reg [31:0]  WriteData_o,
    output  reg [31:0]  MemAddr_o,
    output  reg [31:0]  MemData_o
);
    reg mem_we;
    assign MemWE_o = mem_we;


always @ (*) begin
    if (rst)
        WriteDataAddr_o <= 5'b0;
    else
        WriteDataAddr_o <= WriteDataAddr_i;
end


always @ (*) begin
    if (rst)
        WriteReg_o <= 1'b0;
    else
        WriteReg_o <= WriteReg_i;
end

always @ (*) begin
    if (rst)
        MemData_o <= 32'b0;
    else if (ALUop_i == 5'b10101)  // sw
        MemData_o <= Reg_i;
end


always @ (*) begin
    if (rst)
        WriteData_o <= 32'b0;
    else begin
        if (ALUop_i == 5'b10100)  // lw
            WriteData_o <= MemData_i;
        else
            WriteData_o <= WriteData_i;
    end
end


always @ (*) begin
    if (rst)
        MemAddr_o <= 32'b0;
    else begin
        if (ALUop_i == 5'b10101 || ALUop_i ==5'b10100)  // lw or sw 20, 21
            MemAddr_o <= MemAddr_i; // address setting
        else
            MemAddr_o <= 32'b0;
    end
end


always @ (*) begin
    if (rst)
        MemCE_o <= 1'b0;
    else begin
        if (ALUop_i == 5'b10101 || ALUop_i==5'b10100)  // lw or sw
            MemCE_o <= 1'b1;
        else
            MemCE_o <= 1'b0;
    end
end
            WriteData_o <= MemData_i;
        else
            WriteData_o <= WriteData_i;
    end
end


always @ (*) begin
    if (rst)
        MemAddr_o <= 32'b0;
    else begin
        if (ALUop_i == 5'b10101 || ALUop_i ==5'b10100)  // lw or sw 20, 2
            MemAddr_o <= MemAddr_i; // address setting
        else
            MemAddr_o <= 32'b0;
    end
end


always @ (*) begin
    if (rst)
        MemCE_o <= 1'b0;
    else begin
        if (ALUop_i == 5'b10101 || ALUop_i==5'b10100)  // lw or sw
            MemCE_o <= 1'b1;
        else
            MemCE_o <= 1'b0;
    end
end

always @ (*) begin
    if (rst)
        mem_we <= 1'b0;
    else begin
        if (ALUop_i == 5'b10100)  // lw
            mem_we <= 1'b0;
        else if (ALUop_i == 5'b10101)  // sw
            mem_we <= 1'b1;
        else
            mem_we <= 1'b0;
    end
end


endmodule
```

Fig 15. MEM Module

MEM module은 메모리 컨트롤러의 역할을 하는 모듈이라 할 수 있다. Data memory와 연결되어 있으며 해당 메모리에서 필요한 데이터를 받아오거나 Register의 데이터를 메모리에 쓰는 과정의 신호선 설정을 담당하게 된다.

WriteDataAddr는 Register의 주소를 담고 있으며 해당 주소에 메모리 데이터가 저장되게 된다. (LW의 경우)

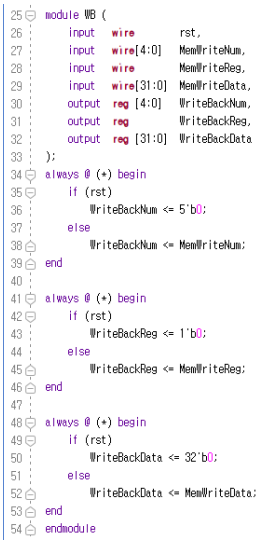
WriteReg는 LW 명령어에서 사용된다. 이는 RegisterFile의 writeReg 신호로 들어가게 된다.

MemData는 SW 명령어에서 메모리에 저장할 Register 데이터 값을 가지게 된다.

MemAddr는 메모리에 접근할 주소를 나타내는 변수이며 해당 값에 해당하는 주소에 접근하게 된다.

MemCE는 메모리 접근을 의미하는 신호이다. 해당 신호가 Data Memory에 전달되며 해당 신호에 따라 데이터 메모리내 데이터를 읽어오거나 쓴다.

MemWe는 메모리에 Write 신호를 의미한다. 해당신호가 MemCE와 동시에 Data Memory에 전달되면 메모리에 데이터 쓰는 작업이 진행된다.



```
module WB (
    input   wire        rst,
    input   wire[4:0]   MemWriteNum,
    input   wire        MemWriteReg,
    input   wire[31:0]  MemWriteData,
    output  reg [4:0]   WriteBackNum,
    output  reg         WriteBackReg,
    output  reg [31:0]  WriteBackData
);
always @ (*) begin
    if (rst)
        WriteBackNum <= 5'b0;
    else
        WriteBackNum <= MemWriteNum;
end

always @ (*) begin
    if (rst)
        WriteBackReg <= 1'b0;
    else
        WriteBackReg <= MemWriteReg;
end

always @ (*) begin
    if (rst)
        WriteBackData <= 32'b0;
    else
        WriteBackData <= MemWriteData;
end
endmodule
```

Fig 16. WriteBack Module

해당 모듈은 Register File에 쓰기를 진행할 때 사용되는 모듈이라 할 수 있다.

MemWriteNum은 Register File내 Register의 번호를 의미한다.

즉 레지스터 주소를 의미하는 값이라 할 수 있다. MemWriteReg는 RegisterFile내 쓰기를 하라는 신호이다. 즉, 해당 신호가 들어올 때 Write 행위가 진행된다고 할 수 있다.

MemWriteData는 Register에 쓰이는 값을 의미한다. 해당 값을 Register File에 전달하여 Register File 내 레지스터에 저장하게 된다.

WriteBackNum, WriteBackReg, WriteBackData는 모두 위 신호들을 받아 전달하는 역할을 하게 된다.

결론적으로 보자면, WB와 MEM을 통해 WB작업에서는 Register에 값을 쓰는 작업이 진행되며 MEM에서는 Data Memory와 관련된 작업이 진행되게 된다.

```
module inst_mem(

    input   wire        rst,    // chip select signal
    input   wire[31:0]  addr,   // instruction address
    output  reg [31:0]  inst    // instruction

);

    reg[31:0]  inst_memory[0:1000];

    initial $readmemb ("machinecode.txt", inst_memory); // read test assembly code file
always @ (*) begin
    if (rst)
        inst <= 32'b0;
    else
        inst <= inst_memory[addr[31:2]]; // for expressing 4 byte pre inst, just use [31:2] bits
end

endmodule
```

Fig 17. Instruction Memory

Instruction Memory는 PC 모듈과 연관되어 있다고 할 수 있다.

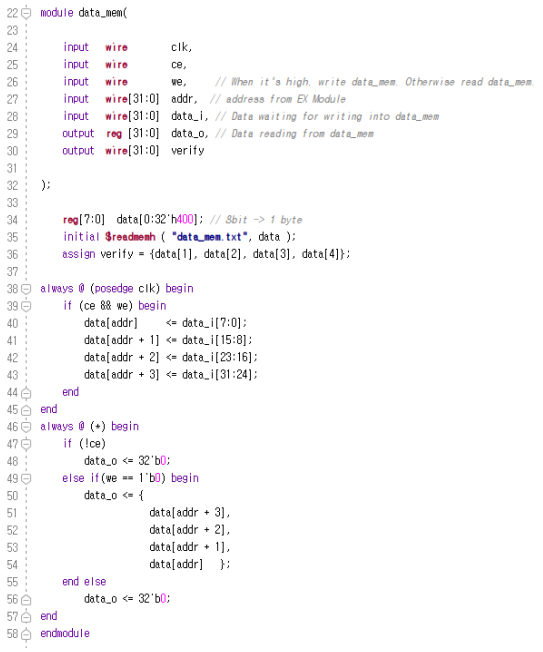
Readmemb를 통해 미리 Binary Code로 짜놓은 명령어를 읽어와 실행하게 된다.

Instruction Memory는 바이트 기준으로 나뉘어져 있으며 32bit가 명령어 길이기에 4byte단위로 명령어가 나뉘어져 있다고 할 수 있다.

해당 모듈에서는 4byte기준으로 메모리를 나뉘어져 있다고 할 수 있다.

그에 따라 받아온 addr를 4로 나눠주어 진행하게 된다.

이는 addr에서는 1byte기준으로 주소값을 설정하였지만 실제 Instruction memory 모듈에서는 4byte기준으로 주소값을 설정하였기 때문이다.



```
module data_mem(

    input   wire        clk,
    input   wire        ce,
    input   wire        we,     // When it's high, write data_mem. Otherwise read data_mem.
    input   wire[31:0]  addr,  // address from EX Module
    input   wire[31:0]  data_i, // Data waiting for writing into data_mem
    output  reg [31:0]  data_o, // Data reading from data_mem
    output  wire[31:0]  verify

);

    reg[7:0]  data[0:32'h400]; // 8bit -> 1 byte
    initial $readmemh ( "data_mem.txt", data );
    assign verify = {data[1], data[2], data[3], data[4]};

always @ (posedge clk) begin
    if (ce && we) begin
        data[addr]     <= data_i[7:0];
        data[addr + 1] <= data_i[15:8];
        data[addr + 2] <= data_i[23:16];
        data[addr + 3] <= data_i[31:24];
    end
end
always @ (*) begin
    if (!ce)
        data_o <= 32'b0;
    else if(we == 1'b0) begin
        data_o <= {
                    data[addr + 3],
                    data[addr + 2],
                    data[addr + 1],
                    data[addr]   };
    end else
        data_o <= 32'b0;
end
endmodule
```

Fig 18. Data Memory

해당 모듈은 Data Memory를 구현한 것이다. 데이터 메모리는 미리 설정해준 데이터 텍스트 파일에서 hex 데이터를 읽어와 값을 설정해준다.

Verify는 미리 설정해준 Instruction내 SW 명령어에서 데이터가 저장될 주소들의 값을 가져오는 역할을 한다. 이는 시뮬레이션이 제대로 진행되었는지 판단하게 도와준다.

Data Memory는 Instruction Memory와 달리 1byte 기준으로 메모리를 설정해주었다. 하지만 lw, sw 명령어에서 사용되는 word는 32bit으로 4byte에 해당한다.

이에 따라 data를 읽거나 쓸 때 위와 같이 4개의 주소값에서 쓰거나 읽어오는 것이다.

4. 시뮬레이션 결과

```
00000000000010000000000010010011  // x1  = 00000001 ADDI x1 x1 1
00000000001100000000000100010011  // x2  = 00000003 ADDI x2 x2 3
00000000011000000000000110010011  // x3  = 00000006 ADDI x3 x3 6
00000000000100010001001000110011  // x4  = 00000006  SLL x4 x2 x1 x2 << x1
00000000000100011101001010110011  // x5  = 00000003  SRL x5 x3 x1  x3 >> x1 3
00000000000000001001000110000011  // x6  = 00000000   LW x6 0(x2)  addr 3  x6 ==8
00000000010000010111001110110011  // x7  = 00000002   AND x7  x2 x4   x7 =2
00000000010000010110010000110011  // x8  = 00000007  OR x8 x2 x4  x8 = 7
00000000010000010100010010110011  // x9  = 00000005 XOR x9  x2 x4 x9 = 5
00000000000010000100001010110011  // x10 = 00000002 ADD x10 x1 x1 x10 = 2
01000000000101010000010110110011  // x11 = 00000001 SUB x11 x10 x1 1 x11
00000000010000101100010001100011 // blt x5, x4 imme (X5<x4 -> pc = pc +imme)  imm=4
00000000001000000000101001101111 // jal x20, imme imme ==1
00000000001001100010000010100011  // sw x2 , 1(x12)  //x12 =0
```

Fig 19. Instruction

위 Instruction 순으로 진행되며 해당 내용들이 어떻게 RTL Design에서 작동하는 지 시뮬레이션 결과를 통해 분석해본다.

1) 00000000000010000000000010010011 // x1 = 00000001 ADDI x1 x1 1

⇒ RegAddr 1에 해당하는 레지스터 x1의 값에 1을 더해준다.

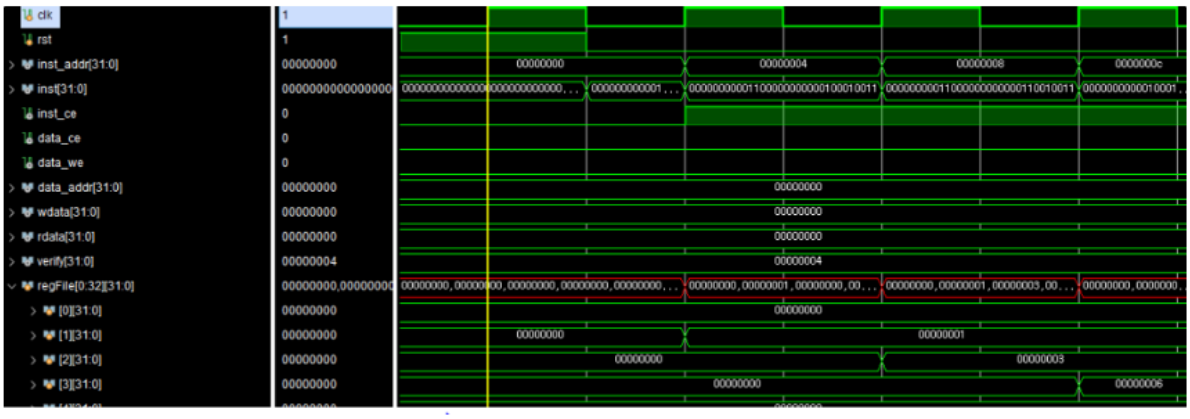


Fig 20. 시뮬레이션 결과(RegFile 1번)

이와 같이 명령어는 posedge clk 에 동기화되어 하나의 클럭 당 하나의 명령어가 실행되게 되어 있다.

Register 에 쓰는 동작 또한 Posedge clk 에 동기화되어 있기에 명령어가 바뀌는 다음 클럭에 해당 레지스터에 쓰는 동작이 반영되는 것을 확인가능하다.

2) 00000000001100000000000100010011 // x2 = 00000003 ADDI x2 x2 3

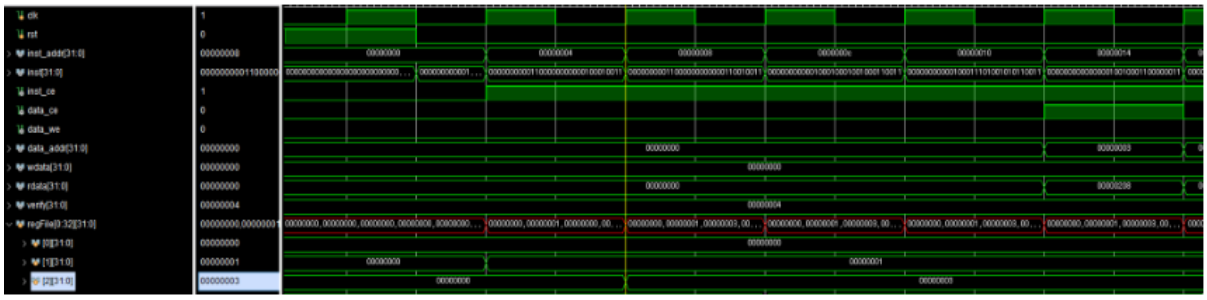


Fig 21. 시뮬레이션 결과(RegFile 2번)

⇒ 레지스터 x2 에 3 을 더해준다. 기존 Reg 들이 0 으로 다 초기화되어 있기에 자연스럽게 Imme add 가 실행가능하다.

이 또한 레지스터에 명령어가 바뀐 다음 클럭에 반영되기에 위와 동일한 시뮬레이션흐름을 확인할 수 있다.

3) 00000000011000000000000110010011 // x3 = 00000006 ADDI x3 x3 6

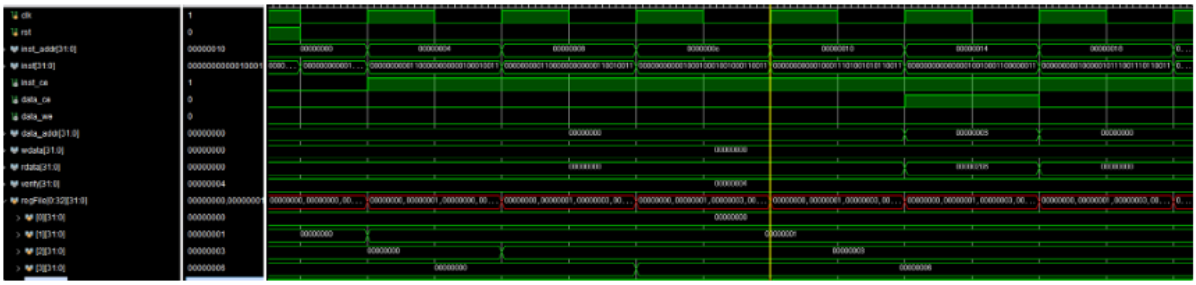


Fig 22. 시뮬레이션 결과(RegFile 3번)

⇒ 레지스터 3 에 6 을 더해주는 명령어이다. 기존의 Reg 들이 0 으로 초기화되어 있기에 자연스럽게 addi 가 실행가능하다.

이 또한 레지스터에 명령어가 바뀐 다음 클럭에 반영되기에 위와 동일한 시뮬레이션 흐름을 확인가능하다.

4) **00000000000100010001001000110011 // x4 = 00000006 SLL x4 x2 x1 x2 << x1**

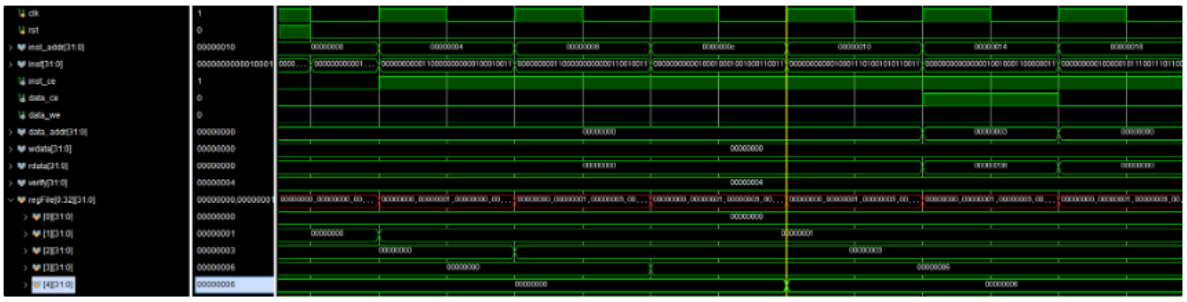


Fig 23. 시뮬레이션 결과(RegFile 4번)

⇒ SLL 은 비트 연산자에 해당하며 Left side 로 이동하게 된다. 즉, 2 의 거듭제곱만큼 값이 증가한다.

x2 의 값은 3 이고 x1 의 값은 1 이기에 3<< 1 의 연산이 이뤄지고 비트가 왼쪽으로 1 번 밀리기에 2 가 곱해진 것과 동일하다. X4 에는 6 의 값이 저장된다.

5) **00000000000100011101001010110011 // x5 = 00000003 SRL x5 x3 x1 x3 >> x1 3**

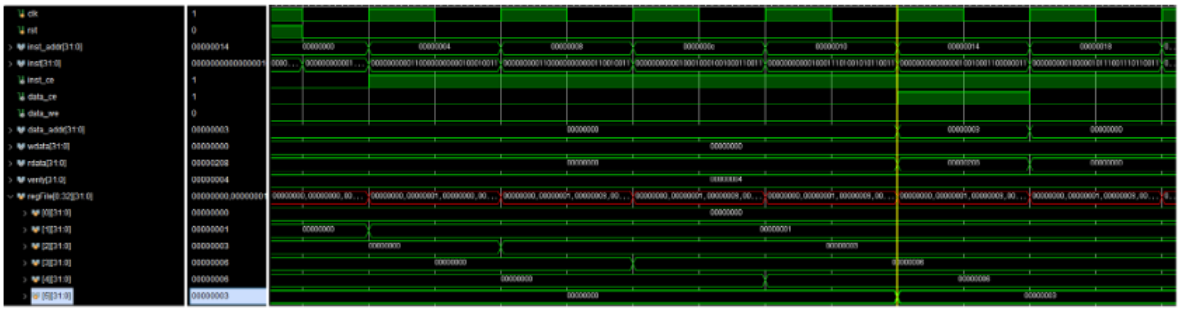


Fig 24. 시뮬레이션 결과(RegFile 5번)

⇒ SRL 은 비트 연산자에 해당하며 Right side 로 이동하게 된다. 즉 2 의 거듭제곱만큼 값이 나눠진다는 것이다. 레지스터 x3 에는 6 이 들어있고 x1 에는 1 이 저장되어 있기에 6 >>1 을 진행하면 3 이된다. 레지스터 x5 에는 3 의 값이 저장된다.

6) **00000000000000010010001100000011 // x6 = 00000000 LW x6 0(x2) addr 3 x6 ==8**

Fig 25. 시뮬레이션 결과(RegFile 6번)

⇒ lw 는 메모리내 데이터를 가져와 레지스터에 저장하는 명령어이다. 데이터는 메모리에 32 비트로 저장되고 하나의 레지스터 또한 32bit 를 가지게 된다. 메모리는 1 개의 Byte 로 주소가 구성되고 4 바이트를 읽어야 lw , Load word 로 word 단위를 읽을 수 있기에 해당하는 주소 (0(x2))에서 4 개의 바이트 정보를 읽는다.

또한 data_ce 를 보면 메모리에 접근하는 신호임을 알 수 있는데 이 신호를 통해 메모리에 접근이 이뤄지고 data_we 가 동시에 켜지면 SW, data_we 가 켜지지 않으면 LW 신호에 해당하게 된다.

7) 00000000010000010111001110110011 // x7 = 00000002 AND x7 x2 x4 x7 =2

Fig 26. 시뮬레이션 결과(RegFile 7번)

⇒ AND 연산의 결과를 레지스터 x7 에 저장하게 된다. 레지스터 x2 의값이 3 이고,

레지스터 x4 의 값이 6 이기에 해당 두개의 값의 AND 연산은 2 가 되게 된다.

레지스터 한 개당 32bit 를 가지게 되고 이는 word 와 동일한 크기라 할 수 있다.

8) 00000000010000010110010000110011 // x8 = 00000007 OR x8 x2 x4 x8 = 7

Fig 27. 시뮬레이션 결과(RegFile 8번)

⇒ OR 연산의 결과를 레지스터 x8 에 저장하게 된다. 레지스터 x2 의 값이 3 이고 레지스터 x4 의 값이 6 이기에 해당 두개의 값의 OR 연산은 7 이 되게 된다.

9) 00000000010000010100010010110011 // x9 = 00000005 XOR x9 x2 x4 x9 = 5

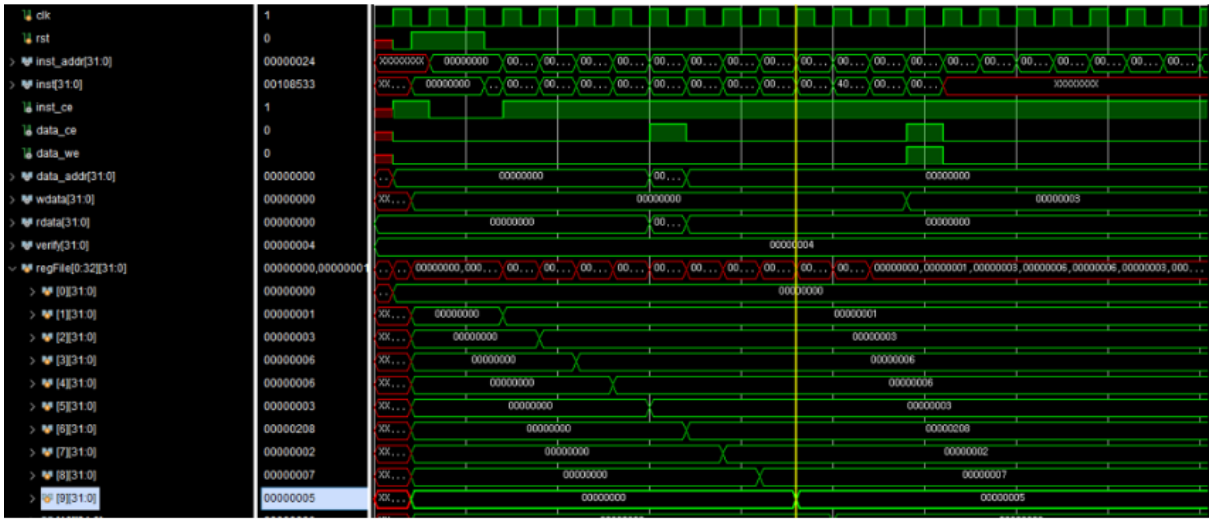


Fig 28. 시뮬레이션 결과(RegFile 9번)

⇒ xor 연산의 결과를 레지스터 x9 에 저장하게 된다. 레지스터 x2 의 값이 3 이고 레지스터 x4 의 값이 6 이기에 xor 연산의 결과는 5 가 되게 된다.

10) 00000000000100001000010100110011 // x10 = 00000002 ADD x10 x1 x1

x10 = 2

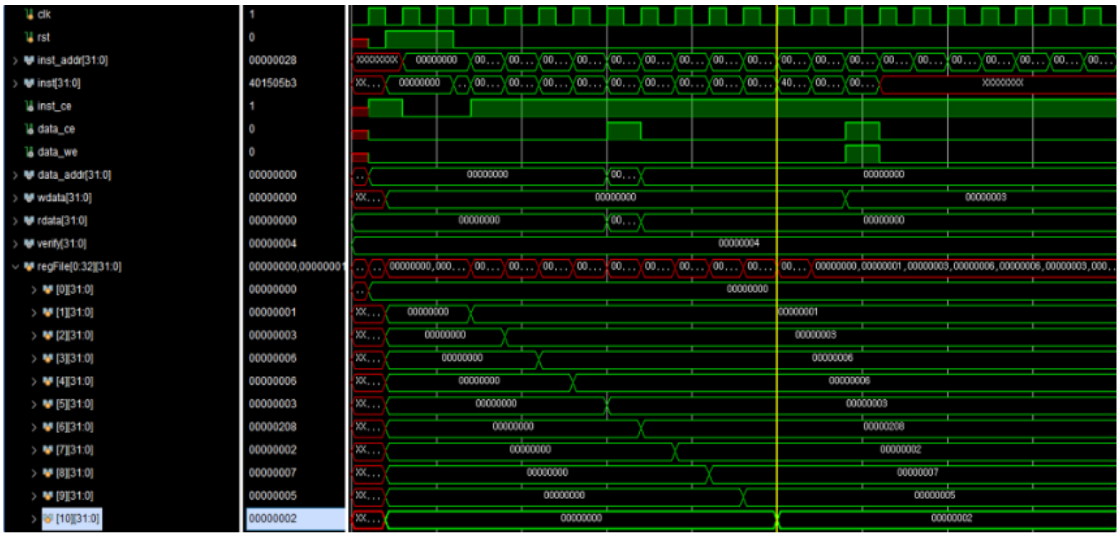


Fig 29. 시뮬레이션 결과(RegFile 10번)

⇒ ADD 연산 결과를 레지스터 x10 에 저장하게 된다. 레지스터 x1 의 값 두개를 더하기에 레지스터 x1 의 값은 1 인 상태에서 x10 은 2 의 값을 저장하게 된다.

11) 01000000000101010000010110110011 // x11 = 00000001 SUB x11 x10 x1 1 x11

Fig 30. 시뮬레이션 결과(RegFile 11번)

⇒sub 명령을 처리하게 되는데 sub 명령의 결과가 레지스터 x11 에 들어가게 된다. 레지스터 x10 의 값은 2 이고 x1 의 값은 1 이기에 레지스터 x11 의 값은 2-1 인 1 이 되게 된다.

12) 00000000010000101100010001100011 // blt x5, x4 imme (X5<x4 -> pc = pc +imme) imm=4

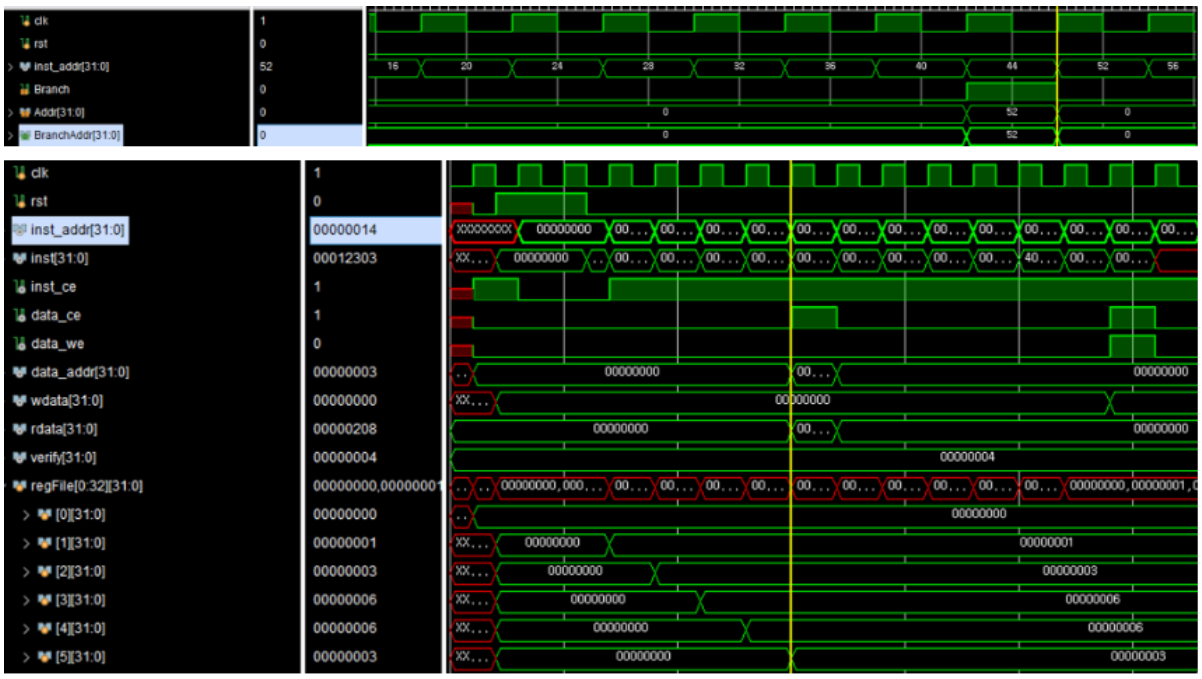


Fig 31. 시뮬레이션 결과

⇒ 레지스터 x5 값이 3 이고 레지스터 x4 값이 6 인 상태에서 x5 <x4 이 True 일 때 PC = PC+ imme 가 된다. X5 의 값이 X4 보다 작기에 분기가 일어나게 된다. 분기의 대상이 PC+4 인 주소에 해당한다.

blt 에는 분기가 일어난다. PC = PC+8 , immediate 의 비트가 0 이 아닌 1 부터 시작하기에 전체 imme 값은 레지스터 x2 된 값에 해당하며 PC 값은 PC+8 에 해당한다.

13) 00000000010000000000101001101111 // jal x20, imme imme ==2

⇒ jal 명령은 jump and link 명령에 해당한다. 이는 레지스터에 현재 PC 값의 다음 명령어 주소를 저장하여 돌아올 장소를 지정해주고 PC 값을 PC+ immex2 로 바꿔준다.

함수 호출 과 같은 명령어 실행시 사용되는 명령어라고 할 수 있다.

14) 00000000001001100010000010100011 // sw x2 , 1(x12) //x12 =0

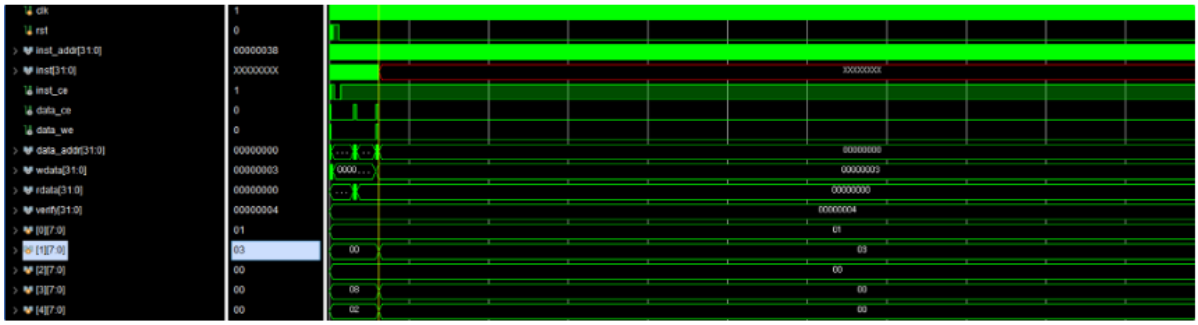


Fig 32. 시뮬레이션 결과(Data Memory 1,2,3,4번)

⇒ 이는 SW 로 레지스터의 값을 메모리에 저장하는 명령어라 할 수 있다.

Store word 에 해당하며 워드는 32bit 에 해당한다. 메모리 주소는 1byte 로 하나의 주소를 형성하고 있기에 32bit 의 크기인 레지스터 x2 의 데이터를 한 워드를 저장하는 것은 4byte 정보를 데이터 메모리에 쓰는 것과 동일하다.

레지스터 x12 의 주소는 0 에 해당하고 imme 값은 1 이기에 데이터 메모리 0 번째 주소부터 데이터가 쓰이기 시작한다. 1, 2, 3, 4 번째 주소 순으로 이렇게 하나의 워드 데이터가 쓰인다.