

발표자 : 최지우

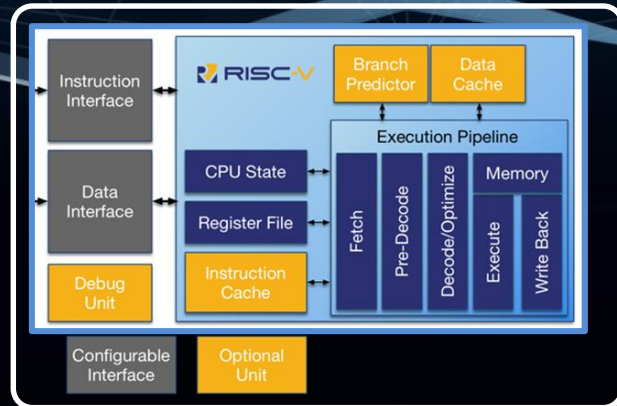
**RISC-V RV32I**

**Instruction Set Architecture**



대한상공회의소  
서울기술교육센터

# 개요

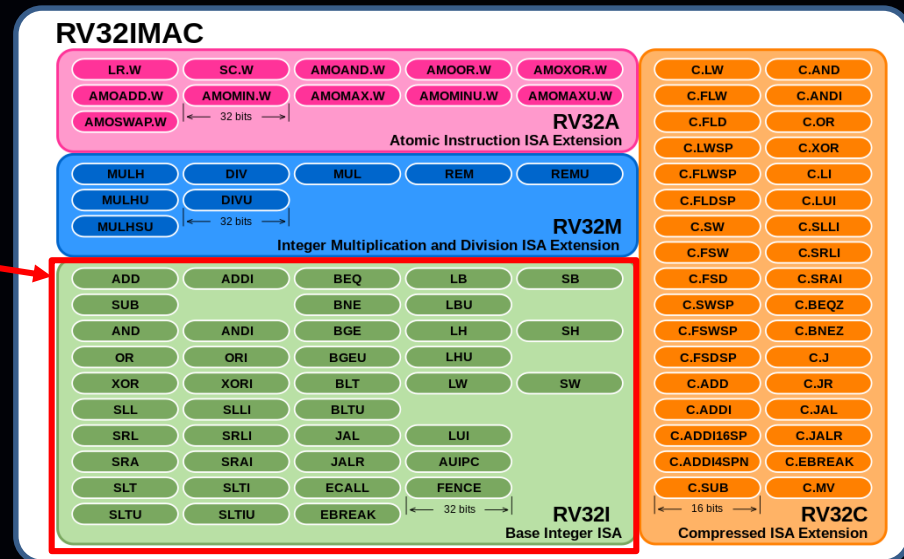


## • RISC-V 란?

- 2010년에 미국 UC 버클리에서 개발된 무료(오픈소스) RISC 명령어 집합 아키텍처(ISA).
- 특징 : 개방성 / 간결성 / 확장성 / 모듈성

## • RISC-V의 RV32I

- RISC-V 아키텍처의 가장 기본적인 명령어 집합.
- 32 : 32비트 기반으로 동작.
- I : 정수(Integer) 연산을 위한 명령어 집합.



# 목차

1. 개발 환경 및 언어 소개
2. 코드 동작 설명
3. Addr/Data 동작 분석
4. Timing Diagram 검증
5. Test Program 설명 및 분석
6. Trouble Shooting
7. 프로젝트 고찰

# 1. 개발 환경 및 언어 소개

RISC-V Online Assembler



MobaXterm

# 개발 환경



# 사용 언어

## 2. 코드 동작 설명

```

1 `timescale 1ns / 1ps
2
3 module ROM (
4     input logic [31:0] addr,
5     output logic [31:0] data
6 );
7     logic [31:0] rom[0:50];
8
9     initial begin
10         // $readmemh("code.mem", rom);
11     end
12
13     //rom[x]=32'b func7; rs2 = rs1 - f3 - rd - op
14     rom[0] = 32'b00000000_0000_0000_0000_0110011;
15     rom[1] = 32'b01000000_0000_0000_0000_00010100_0110011;
16     rom[2] = 32'b00000000_10010_0000_0001_00101_0110011;
17     rom[3] = 32'b00000000_0000_0000_0001_101_01010_0110011;
18     rom[4] = 32'b01000000_0000_0000_0001_101_01111_0110011;
19     rom[5] = 32'b00000000_0000_0010_010_01000_0110011;
20     rom[6] = 32'b00000000_0000_0010_011_01001_0110011;
21     rom[7] = 32'b00000000_0000_0000_100_01010_0110011;
22     rom[8] = 32'b00000000_0000_0000_110_01011_0110011;
23     rom[9] = 32'b00000000_0000_0000_111_01100_0110011;
24
25     //rom[x]=32'b imm12 - rs1 - f3 - rd - op
26     rom[10] = 32'b00000000_0000_0000_0001_00010_0010_0010011;
27     rom[11] = 32'b00000000_0000_0001_0010_010_00111_0010011;
28     rom[12] = 32'b00000000_0000_0001_011_01000_0010011;
29     rom[13] = 32'b00000000_0000_0001_100_01001_0010011;
30     rom[14] = 32'b00000000_0000_0001_110_01010_0010011;
31     rom[15] = 32'b00000000_0000_0001_111_01011_0010011;
32
33     //rom[x]=32'b func7; rs1 - f3 - rd - op
34     rom[16] = 32'b00000000_10111_00101_001_01100_0010011;
35     rom[17] = 32'b00000000_10111_00101_001_01101_0010011;
36     rom[18] = 32'b00000000_10111_00101_001_01110_0010011;
37     rom[19] = 32'b00000000_10111_00101_001_01111_0010011;
38     //로딩된 인스트럭션의 워드 값을 반환. 즉, 주어진 rom[32]의 값을 반환
39     rom[20] = 32'b00000000_01110_01101_000_10100_0110011;
40
41     //rom[x]=32'b imm12 - rs1 - f3 - rd - op
42     rom[48] = 32'b00000000_0000_0100_000_01101_0110011;
43
44     end
45
46     assign data = rom[addr[31:2]];
47
48 endmodule

```

### # ROM

- CPU의 명령어를 저장하는 ROM(Instruction Memory)에 Test Code를 가입.
- 이 Code에서 가리키는 값에 따라서 Control Unit과 Data Path가 역할을 수행.

```

99 always_comb begin
100     next_state = state;
101     case (state)
102         FETCH: next_state = DECODE;
103         DECODE: begin
104             //OP_TYPE_R: next_state = R_EXE;
105             E_R: next_state = E_EXE;
106             E_E: next_state = E_EXE;
107             E_L: next_state = L_EXE;
108             E_AU: next_state = AU_EXE;
109             E_J: next_state = J_EXE;
110             E_JL: next_state = JL_EXE;
111             E_S: next_state = S_EXE;
112             E_L: next_state = L_EXE;
113
114             signals = 10'b0_0_0_0_000_0_0_0;
115             R_EXE: begin
116                 signals = 10'b0_1_0_0_000_0_0_0;
117                 DECODE: signals = 10'b0_0_0_0_000_0_0_0;
118                 aluControl = operator;
119             end
120             I_EXE: begin
121                 signals = 10'b0_1_1_0_000_0_0_0;
122                 If (operator == 4'b1101) aluControl = operator;
123                 else aluControl = (1'b0, operator[2:0]);
124             end
125             R_EXE: begin
126                 signals = 10'b0_0_0_0_000_1_0_0;
127                 aluControl = operator;
128             end
129             LU_EXE: signals = 10'b0_1_0_0_010_0_0_0;
130             AU_EXE: signals = 10'b0_1_0_0_011_0_0_0;
131             J_EXE: signals = 10'b0_1_0_0_100_0_1_0;
132             JL_EXE: signals = 10'b0_1_0_0_100_0_1_0;
133             S_EXE: signals = 10'b0_0_1_0_000_0_0_0;
134             S_MEM: signals = 10'b0_0_1_1_000_0_0_0;
135             L_EXE: signals = 10'b0_0_1_0_001_0_0_0;
136             L_MEM: signals = 10'b0_0_1_0_001_0_0_0;
137             L_WB: signals = 10'b0_1_1_0_001_0_0_0;
138
139         endcase
140     end

```

### # Control Unit

- ROM에서 넘어온 명령어를 해독하여 그에 대한 정보를 DataPath와 RAM으로 전달.
- Multi Cycle로 동작하는 CPU 전체의 State를 통제 및 제어.

```

1 `timescale 1ns / 1ps
2 `include "defines.sv"
3
4 module DataPath (
5     input logic clk,
6     input logic reset,
7     input logic [31:0] instrCode,
8     output logic [31:0] instrMemAddr,
9     input logic PCEN,
10    input logic regFileMe,
11    input logic [ 3:0] aluControl,
12    input logic aluSrcMuxSel,
13    output logic [31:0] data,
14    output logic [31:0] bdata,
15    input logic [31:0] b,
16    input logic [ 2:0] R,
17    output logic [31:0] result,
18    output logic btaken
19 );
20
21 module alu (
22     input logic [ 3:0] aluControl,
23     input logic [31:0] a,
24     input logic [31:0] b,
25     output logic [31:0] result,
26     output logic btaken
27 );
28
29 always_comb begin
30     result = 32'b;
31     case (aluControl)
32         "ADD": result = a + b;
33         "SUB": result = a - b;
34         "SLT": result = a < b;
35         "SRL": result = a >> b;
36         "SRA": result = $signed(a) >>> b;
37         "SLR": result = ($signed(a) < $signed(b)) ? 1 : 0;
38         "SLTU": result = (a < b) ? 1 : 0;
39         "XOR": result = a ^ b;
40         "OR": result = a | b;
41         "AND": result = a & b;
42     endcase
43 end
44
45 module RegisterFile (
46     input logic clk,
47     input logic we,
48     input logic [ 4:0] RA1,
49     input logic [ 4:0] RA2,
50     input logic [31:0] WD,
51     output logic [31:0] RD1,
52     output logic [31:0] RD2
53 );
54     logic [31:0] mem[0:2**5-1];
55
56     always_ff @(posedge clk) begin
57         if (we) mem[RA1] <= WD;
58     end
59 endmodule
60
61 assign RD1 = (RA1 != 0) ? mem[RA1] : 32'b0;
62 assign RD2 = (RA2 != 0) ? mem[RA2] : 32'b0;
63 endmodule

```

### # Data Path

- Control Unit의 제어에 따라 Mux가 활성화 및 비활성화.
- 32bit의 신호가 rs1, rs2, rd 등의 address 정보를 전달하고, alu의 동작을 제어한다.

## 2. 코드 동작 설명

```
1 `timescale 1ns / 1ps
2
3 module RAM (
4     input logic
5     input logic
6     input logic [31:0]
7     input logic [31:0]
8     output logic [31:0]
9     input logic [1:0]
10    input logic
11);
12    logic [31:0] mem[0:2**8-1];
13    logic [31:0] non_rData;
```

```
22
23    always_ff @(posedge clk) begin
24        if(we) begin
25            case(LSControl)
26                2'b00 : mem[addr[31:2]] <= {{24(1'b0)},wData[7:0]};
27                2'b01 : mem[addr[31:2]] <= {{16(1'b0)},wData[15:0]};
28                2'b10 : mem[addr[31:2]] <= wData;
29            endcase
30        end
31    end
32    assign non_rData = mem[addr[31:2]];
33    always_comb begin
34        case ({SignControl, LSControl})
35            3'b000 : rData = {{24(non_rData[7])}, non_rData[7:0]};
36            3'b001 : rData = {{16(non_rData[15])}, non_rData[15:0]};
37            3'b010 : rData = non_rData;
38            3'b100 : rData = {{24(1'b0)}, non_rData[7:0]};
39            3'b101 : rData = {{16(1'b0)}, non_rData[15:0]};
40            default : rData = 32'd0;
41        endcase
42    end
```

```
158    registerEn U_PC (
159        .clk (clk),
160        .reset(reset),
161        .en (PCEn),
162        .d (ExeReg_PCsrcMuxOut),
163        .q (PCOutData)
164    );
```

```
233
234    module registerEn (
235        input logic clk,
236        input logic reset,
237        input logic en,
238        input logic [31:0] d,
239        output logic [31:0] q
240    );
241    always_ff @(posedge clk, posedge reset) begin
242        if (reset) begin
243            q <= 0;
244        end else begin
245            if (en) q <= d;
246        end
247    end
248    endmodule
249
```

### # RAM

- S/L-Type의 명령어를 수행할 때 RAM 모듈이 활성화되고, Data의 저장/적재를 수행.
- Data의 Signed/Unsigned 및 Byte/Half/Word의 통제를 수행.

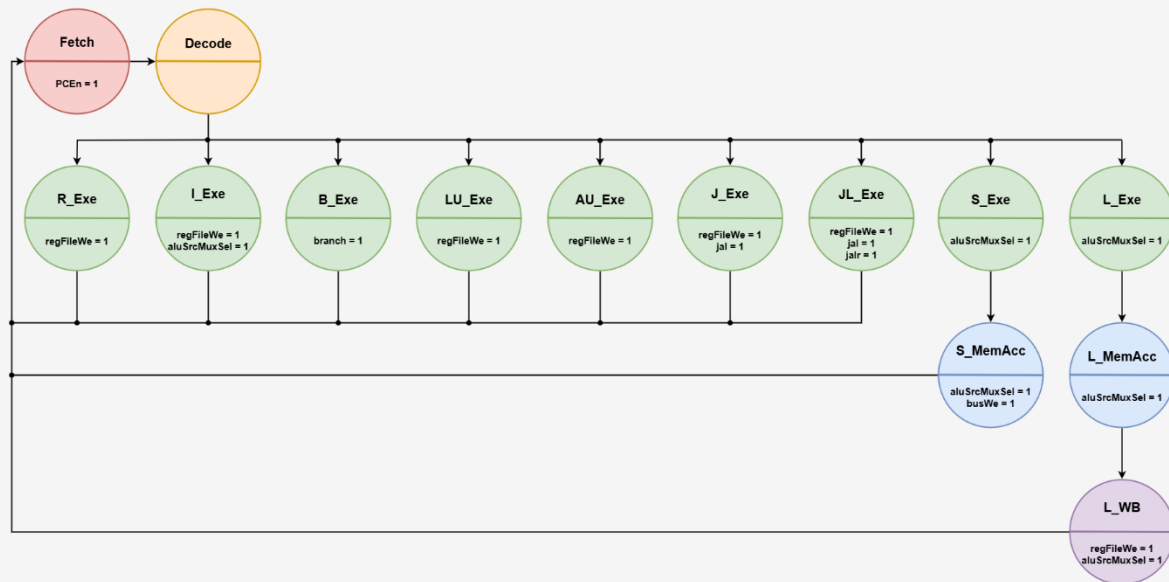
### # Program Counter

- 다음 번에 가리킬 메모리의 주소 값을 저장.
- Multi Cycle 동작을 위해서 Enable 신호가 들어왔을 때에만 값을 출력.



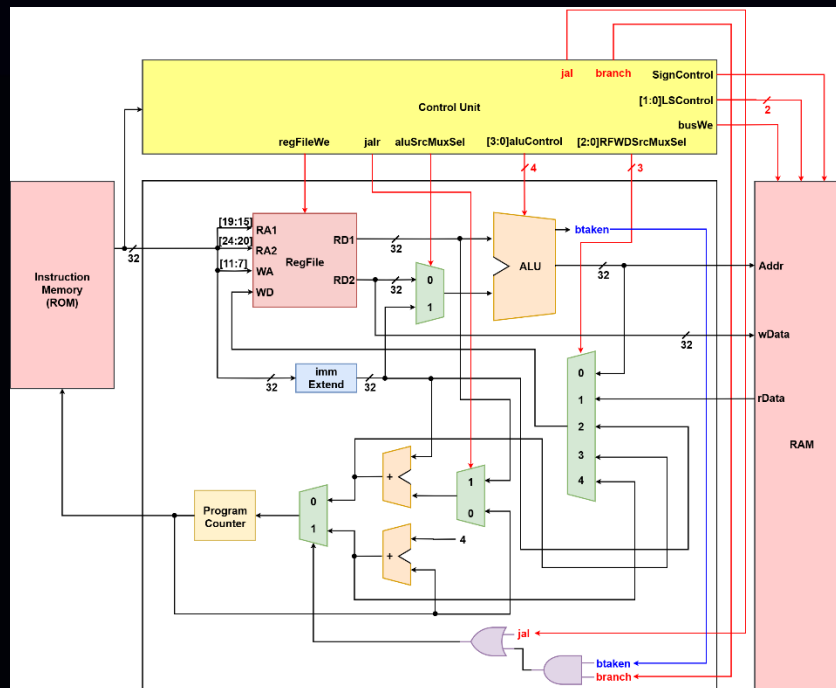
## 2. 코드 동작 설명

- FSM(Finite State Machine)



## 2. 코드 동작 설명

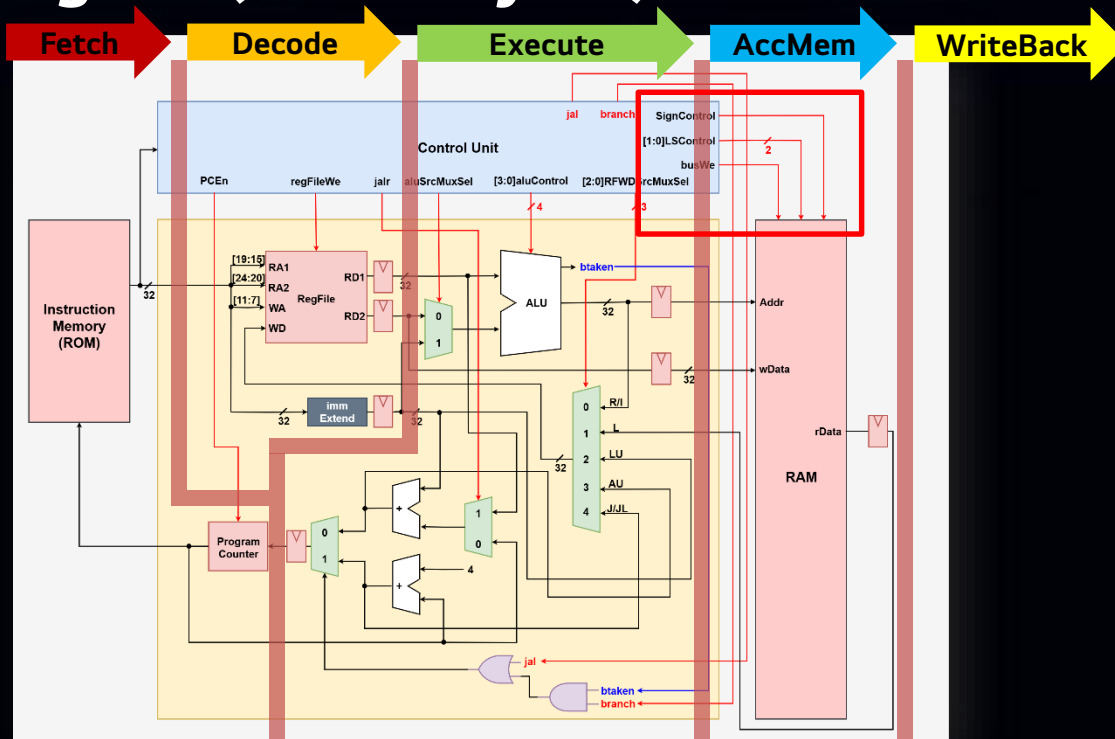
- Block Diagram(Single Cycle)





## 2. 코드 동작 설명

- Block Diagram(Multi Cycle)



## 2. 코드 동작 설명

### • Type 별 동작 설명

Type	Description
R-Type	레지스터 간의 연산 결과를 레지스터에 저장.
L-Type	메모리에 있는 데이터를 적재(load)하여 레지스터에 저장.
I-Type	레지스터의 특정 주소에 있는 값과 즉시값 사이의 연산.
S-Type	레지스터에 있는 값을 메모리에 저장.

Type	Description
B-Type	비교 연산 결과에 따라 PC의 값이 분기.
LU-Type	즉시값에 12 만큼의 shift left logical을 수행한 값(즉시값 $\ll 12$ )을 레지스터에 저장.
AU-Type	즉시값에 12 만큼의 shift left logical을 수행한 값(즉시값 $\ll 12$ )과 PC 값을 더한 결과를 레지스터에 저장.
JAL-Type	다음 명령어 주소를 레지스터에 저장. 다음 PC의 값이 분기( $PC = PC + imm$ ).
JALR-Type	다음 명령어 주소를 레지스터에 저장. 다음 PC의 값이 분기( $PC = rs1 + imm$ ).

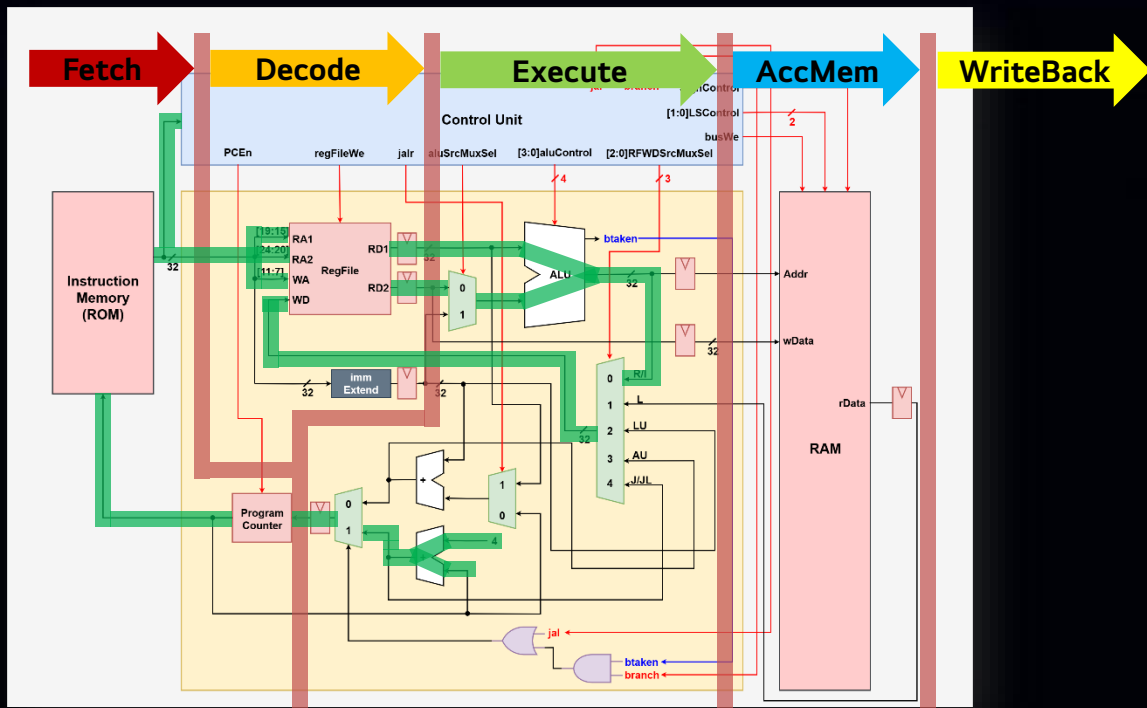
## 2. 코드 동작 설명

### • Block Diagram : R-Type

- 두 레지스터의 값을 연산 후 결과를 다른 레지스터에 저장.

- 주로 산술 / 논리 연산.

- 즉시값 사용 X.

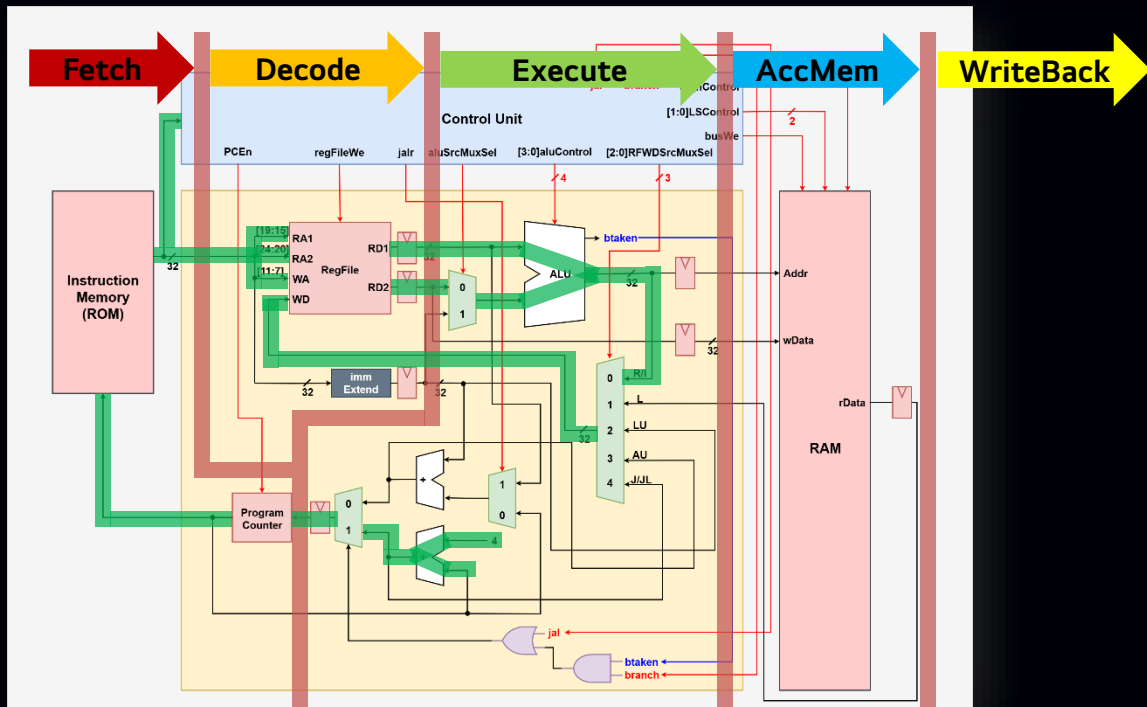


## 2. 코드 동작 설명

- Block Diagram : I-Type

- 즉시값(immediate, 12bit)과 레지스터 값을 연산.

- 산술/논리 연산의 즉시값 버전.

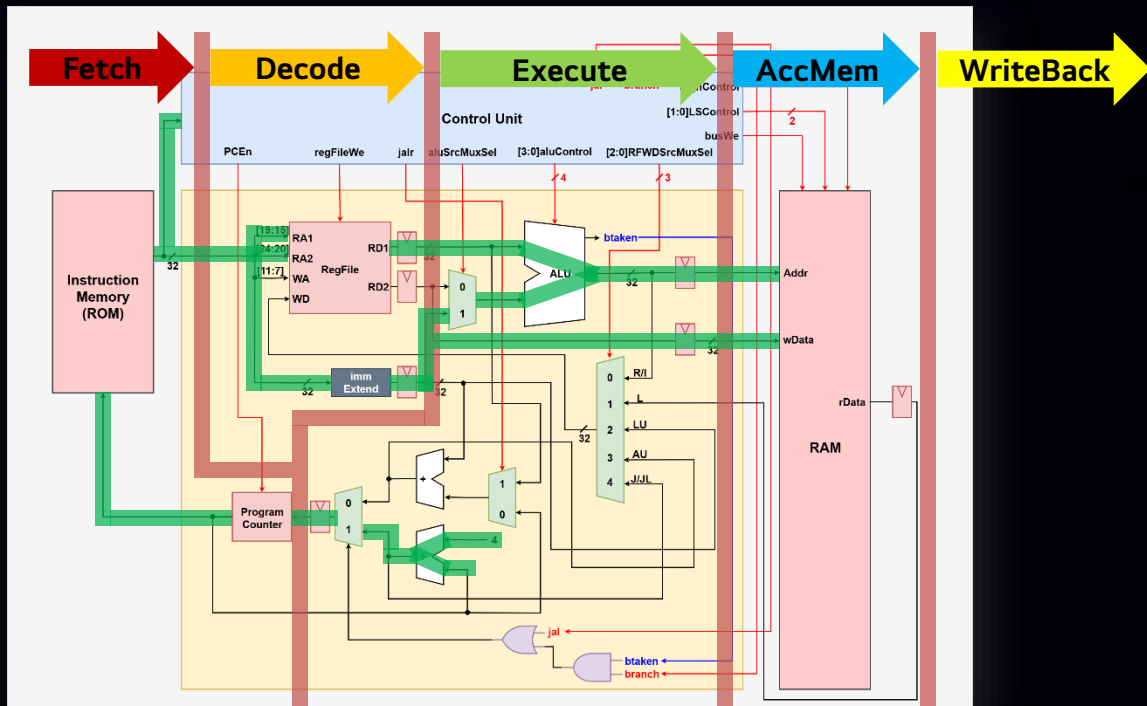


## 2. 코드 동작 설명

- Block Diagram : S-Type

- Store 명령어 전용.(레지스터 데이터를 메모리에 저장)

- 연산 결과를 rd가 아닌 메모리 주소에 저장하는 것에 주의.

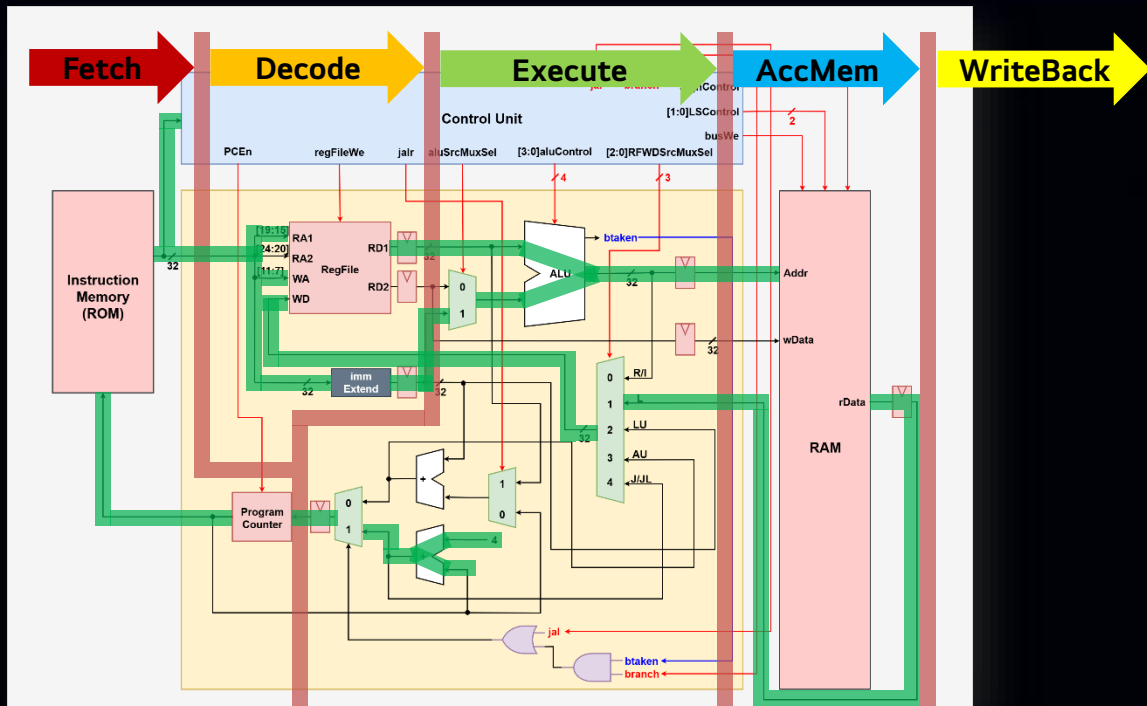


## 2. 코드 동작 설명

- Block Diagram : L-Type

- 공식적으로 L-Type은 존재하지 않고, load 명령어는 I-Type에 포함됨.

- 내용을 정리하기 위해서 따로 L-Type을 명시하여, RAM에서 RegFile로 데이터를 적재하는 용도로 명시.



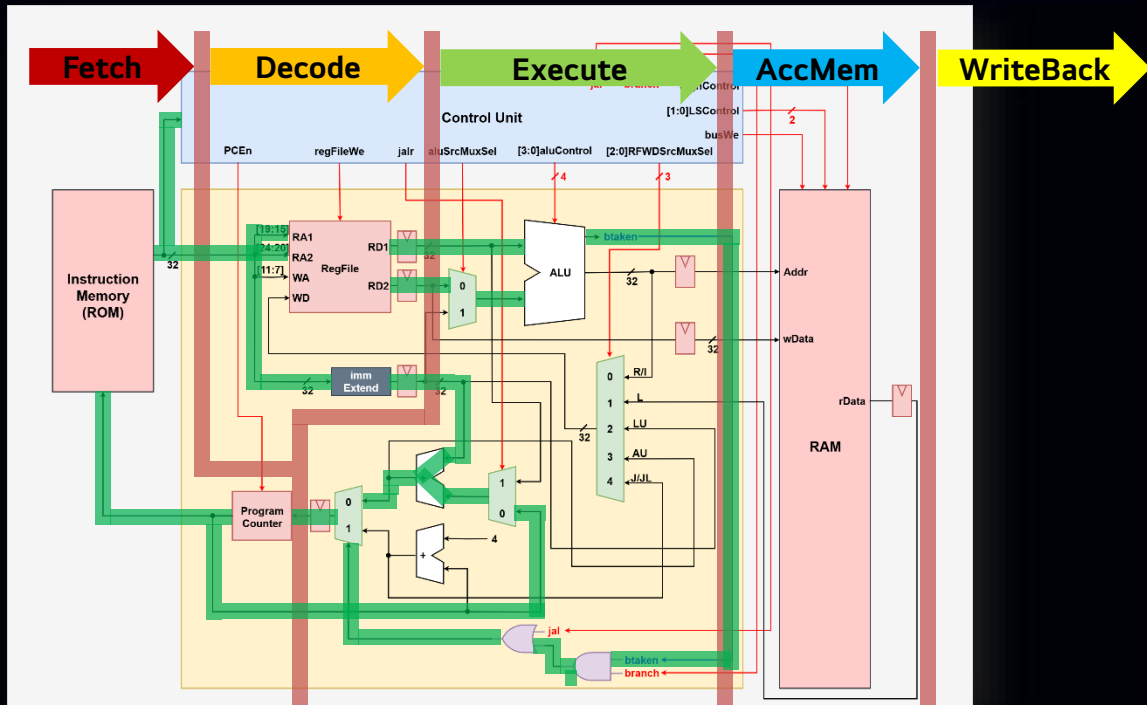


- **Block Diagram : B-Type**

- ### - 조건 분기(Branch) 명령어 전용 Type.

- offset(12bit) 즉시값을 사용.  
(단, PC는 상대적인 주소)

- 조건식 평가 후 분기 수행.

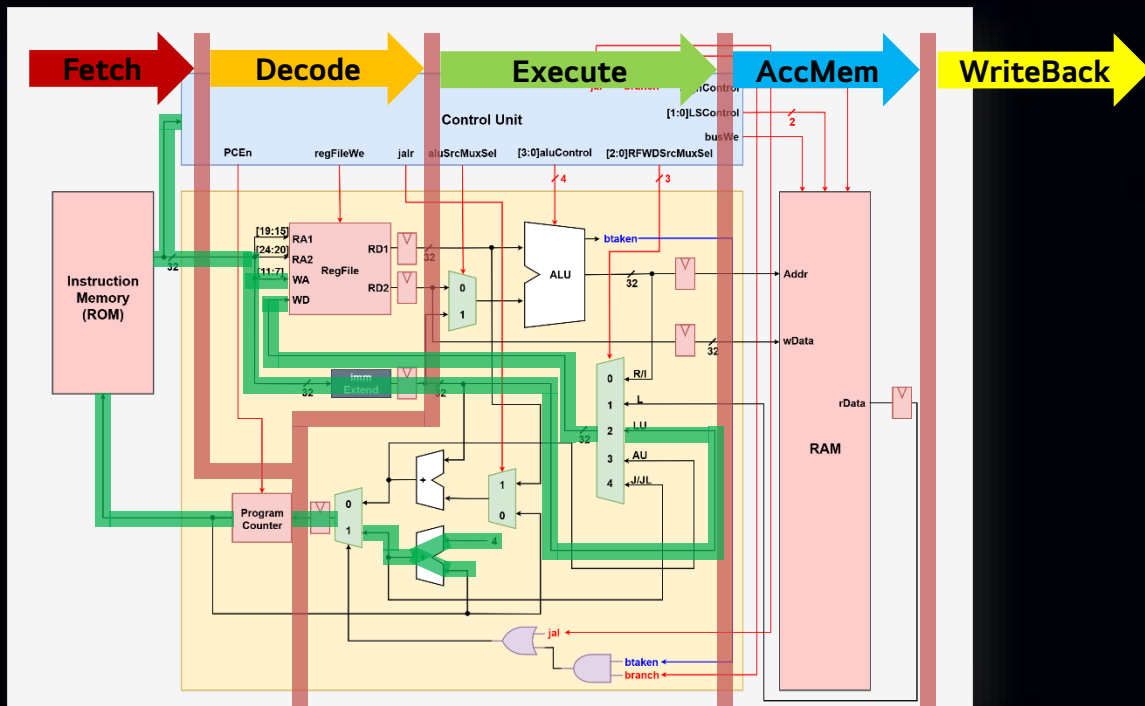


## 2. 코드 동작 설명

- Block Diagram : U-Type(LU)

- 상위 20bit의 즉시값을 rd에 저장.(하위 12bit는 0으로 채움)

- 큰 상수 로딩이나 주소 계산에 활용.

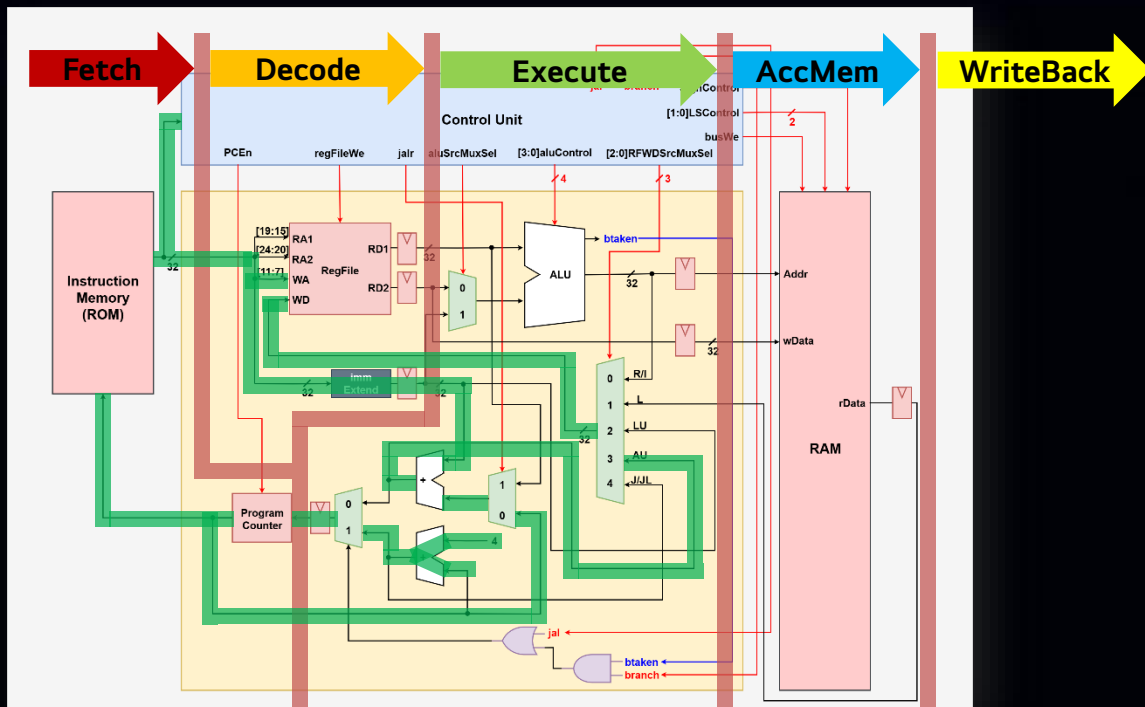


## 2. Block Diagram

- Block Diagram : U-Type(AU)

- 상위 20bit의 즉시값과 PC를 더한 값을 rd에 저장.(하위 12bit는 0으로 채움)

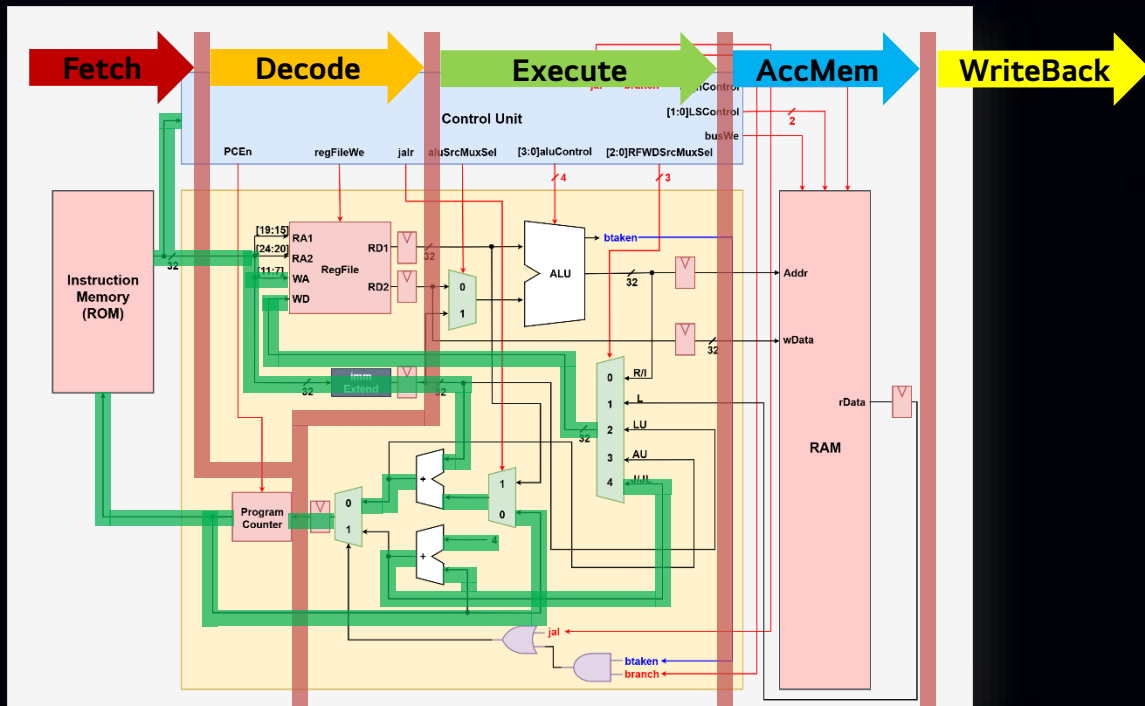
- ex) auipc rd, imm20



## 2. Block Diagram

### • Block Diagram : J-Type(JAL)

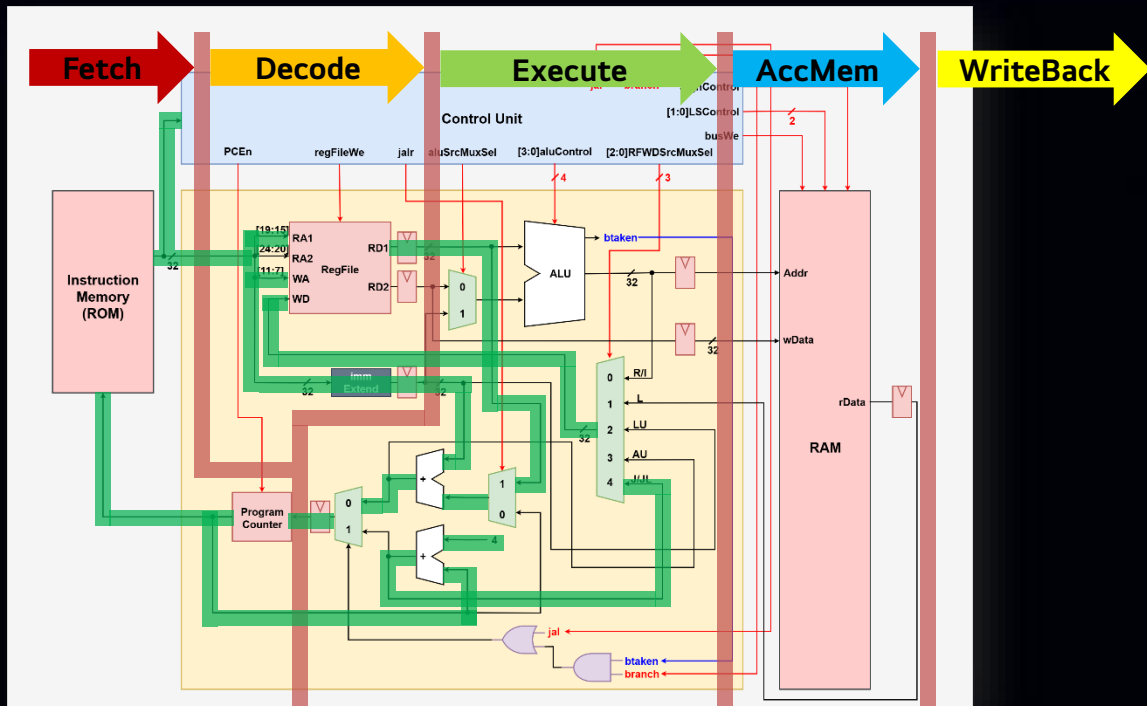
- Jump 명령어의 일종.
- 현재 PC에 대하여 상대적으로 Jump 처리.
- 현재 PC의 값은 잠시 다른 rd에 저장한 채로, 다른 PC의 값을 활용.
- JAL의 경우, 현재 PC의 값에 imm이 더해진 값이 다시 PC로 들어간다.



## 2. Block Diagram

### • Block Diagram : JALR-Type

- Jump 명령어의 일종.
- 현재 PC에 대하여 상대적으로 Jump 처리.
- 현재 PC의 값은 잠시 다른 rd에 저장한 채로, 다른 PC의 값을 활용.
- JALR의 경우, rs1의 값과 imm을 더해서 PC의 값으로 처리.



### 3. Addr/Data 동작 분석

#### • R-Type(1)

```
//rom[x]=32'b funct7_ rs2 _ rs1 _ f3 _ rd _ op  
rom[0] = 32'b00000000_00001_00010_000_00011_0110011;  
rom[1] = 32'b0100000_00001_00010_000_00100_0110011;  
rom[2] = 32'b00000000_10010_00010_001_00101_0110011;  
rom[3] = 32'b00000000_00001_00010_101_00110_0110011;  
rom[4] = 32'b0100000_00001_00010_101_00111_0110011;
```



```
add x3, x2, x1  
sub x4, x2, x1  
sll x5, x2, x18  
srl x6, x2, x1  
sra x7, x2, x1
```

- 주소 x2, x1에 위치한 데이터(12, 11)를 더한 값(23)을 주소 x3에 적재.
- 주소 x2, x1에 위치한 데이터(12, 11)를 뺀 값(1)을 주소 x4에 적재.
- 주소 x2에 위치한 데이터(32'b00...01100)에 대하여, 주소 x18에 위치한 데이터(28)만큼 sll(shift left logical) 연산을 실행한 값(32'b1100...00)을 주소 x5에 적재.
- 주소 x2에 위치한 데이터(32'b00...01100)에 대하여, 주소 x1에 위치한 데이터(11)만큼 srl(shift right logical) 연산을 실행한 값(0)을 주소 x6에 적재.
- 주소 x2에 위치한 데이터(32'b00...01100)에 대하여, 주소 x1에 위치한 데이터만큼 sra(shift right arithmetic) 연산을 실행한 값(0)을 주소 x7에 적재.

RegFile	
zero	
11	
12	
13 -> 23	
14 -> 1	
15 -> 32'b11000.....00	
16 -> 0	
17 -> 0	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	



### 3. Addr/Data 동작 분석

#### • R-Type(2)

```
//rom[x]=32'b funct7_ rs2 _ rs1 _ f3 _ rd _ op  
rom[5] = 32'b0000000_00010_00101_010_01000_0110011;  
rom[6] = 32'b0000000_00010_00101_011_01001_0110011;  
rom[7] = 32'b0000000_00010_00001_100_01010_0110011;  
rom[8] = 32'b0000000_00010_00001_110_01011_0110011;  
rom[9] = 32'b0000000_00010_00001_111_01100_0110011;
```



```
slt x8 x5, x2  
sltu x9 x5, x2  
xor x10 x1, x2  
or x11 x1, x2  
and x12 x1, x2
```

- 주소 x5, x2에 위치한 데이터에 slt(set less than) 연산을 실행하여, 참(1)이나 거짓(0)을 주소 x8에 적재.
- 주소 x5, x2에 위치한 데이터(??, 12)에 sltu(set less than unsign) 연산을 실행하여, 참(1)이나 거짓(0)을 주소 x9에 적재.
- 주소 x1, x2에 위치한 데이터(32'b0...01011, 32'b0...01100)에 xor 연산을 실행한 값을 주소 x10에 적재.
- 주소 x1, x2에 위치한 데이터에 or 연산을 실행한 값(32'b0...01111)을 주소 x11에 적재.
- 주소 x1, x2에 위치한 데이터에 and 연산을 실행한 값(32'b0...01010)을 주소 x12에 적재.

0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20

RegFile	
zero	
11	
12	
13 -> 23	
14 -> 1	
15 -> 32'b11000...00	
16 -> 0	
17 -> 0	
18 -> 1	
19 -> 0	
20 -> 32'b0..00111	
21 -> 32'b0..01111	
22 -> 32'b0..01000	
23	
24	
25	
26	
27	
28	
29	
30	

# 3. Addr/Data 동작 분석

## • I-Type(1)

```
//rom[x]=32'b    imm12    _rs1 _f3_ rd _op
```

```
rom[10] = 32'b000000000001_00010_000_00110_0010011;  
rom[11] = 32'b000000000001_00101_010_00111_0010011;  
rom[12] = 32'b000000000001_00101_011_01000_0010011;  
rom[13] = 32'b0000000001111_00010_100_01001_0010011;  
rom[14] = 32'b000000001111_00010_110_01010_0010011;  
rom[15] = 32'b000000001111_00010_111_01011_0010011;
```



```
addi x6, x2, 1  
slti x7, x5, 1  
sltiu x8, x5, 1  
xori x9, x2, 15  
ori x10, x2, 15  
andi x11, x2, 15
```

- 주소 x2에 위치한 데이터(12)와 immediate 값(1)에 addi 연산을 실행한 값(13)을 주소 x6에 적재.
- 주소 x5에 위치한 데이터와 immediate 값(1)에 slti 연산을 실행하여 주소 x7에 적재.
- 주소 x5에 위치한 데이터와 immediate 값(1)에 sltiu 연산을 실행하여 주소 x8에 적재.
- 주소 x2에 위치한 데이터(1100)와 immediate 값(1111)에 xor 연산을 실행하여 주소 x9에 적재.
- 주소 x2에 위치한 데이터(1100)와 immediate 값(1111)에 ori 연산을 실행하여 주소 x10에 적재.
- 주소 x2에 위치한 데이터(1100)와 immediate 값(1111)에 andi 연산을 실행하여 주소 x11에 적재.

RegFile	
zero	
11	
12	
13	
13 -> 23	
14 -> 1	
15 -> 32'b11000...00	
16 -> 0 -> 13	
17 -> 0 -> 1	
18 -> 1 -> 0	
19 -> 0 -> 32'b0...000011	
20 -> 32'b0...00111 -> 32'b0...01111	
21 -> 32'b0...01111 -> 32'b0...01100	
22 -> 32'b0...01000	
23	
24	
25	
26	
27	
28	
29	
30	

### 3. Addr/Data 동작 분석

#### • I-Type(2)

```
//rom[x]=32'b funct7 _shamt_ rs1_f3 _rd _op
```

```
rom[16] = 32'b0000000_10111_00101_001_01100_0010011;  
rom[17] = 32'b0000000_10111_00101_101_01101_0010011;  
rom[18] = 32'b0000000_01111_00101_101_01110_0010011;  
rom[19] = 32'b0100000_10111_00101_101_01111_0010011;
```

```
rom[20] = 32'b0000000_01110_01101_000_10100_0110011;
```

```
slli x12, x5, 23  
srli x13, x5, 23  
srli x14, x5, 15  
srai x15, x5, 23
```

```
add x20, x13, x14
```

- 주소 x5에 위치한 데이터에 immediate 값(23) 만큼의 slli(shift left logical immediate) 연산을 실행하여 주소 x12에 적재.

- 주소 x5에 위치한 데이터에 immediate 값 만큼의 srli(shift right logical immediate) 연산을 실행하여 주소 x13에 적재.

- 주소 x5에 위치한 데이터에 immediate 값 만큼의 srli(shift right logical immediate) 연산을 실행하여 주소 x14에 적재.

- 주소 x5에 위치한 데이터에 immediate 값 만큼의 srai(shift right arithmetic immediate) 연산을 실행하여 주소 x15에 적재.

RegFile	
zero	
11	
12	
13 -> 23	
14 -> 1	
15 -> 32'b11000...00	
16 -> 0 -> 13	
17 -> 0 -> 1	
18 -> 1 -> 0	
19 -> 0 -> 32'b0...000011	
20 -> 32'b0...00111 -> 32'b0...01111	
21 -> 32'b0...01111 -> 32'b0...01100	
22 -> 32'b0...01000 -> 32'b0...00000	
23 -> 32'b0...01_1000_0000	
24 -> 32'b0...01_1000_0000_0000_0000	
25 -> 32'b1..1_1000_0000	
26	
27	
28	
29	
30 -> 32'b0...01_1000_0001_1000_0000	

# 3. Addr/Data 동작 분석

## • S-Type

```
//rom[x]=32'b imm7 _ rs2 _ rs1 _ f3_ imm5 _ op
```

```
rom[21] = 32'b0000000_10100_00010_000_01000_0100011;  
rom[22] = 32'b0000000_10100_00010_001_01100_0100011;  
rom[23] = 32'b0000000_10100_00010_010_10000_0100011;
```



```
sb x20, 8(x2)  
sh x20, 12(x2)  
sw x20, 16(x2)
```

- 주소 x20에 있는 데이터를 byte(8bit) 크기만큼 주소 8(x2) 주소의 RAM에 저장(store).
- 주소 x20에 있는 데이터를 half(16bit) 크기만큼 주소 12(x2) 주소의 RAM에 저장(store).
- 주소 x20에 있는 데이터를 word(32bit) 크기만큼 주소 16(x2) 주소의 RAM에 저장(store).

	RAM
0	
1	
2	
3	
4	
5	
6	32'b0...00_0000_0000_1000_0000
7	32'b0...00_1000_0001_1000_0000
8	32'b0...01_1000_0001_1000_0000

	RegFile
0	zero
1	11
2	12
3	13 -> 23
4	14 -> 1
5	15 -> 32'b11000...00
6	16 -> 0 -> 13
7	17 -> 0 -> 1
8	18 -> 1 -> 0
9	19 -> 0 -> 32'b0...00011
10	20 -> 32'b0...00111 -> 32'b0...01111
11	21 -> 32'b0...01111 -> 32'b0...01100
12	22 -> 32'b0...01000 -> 32'b0...0000
13	23 -> 32'b0...01_1000_0000
14	24 -> 32'b0...01_1000_0000_0000_0000
15	25 -> 32'b1..1_1000_0000
16	26
17	27
18	28
19	29
20	30 -> 32'b0...01_1000_0001_1000_0000

# 3. Addr/Data 동작 분석

## L-Type

```
//rom[x]=32'b imm12 _ rs1 _ f3_ rd _ op
```

```
rom[24] = 32'b000000011100_00000_100_0111_0000011;  
rom[25] = 32'b000000011100_00000_000_10000_0000011;  
rom[26] = 32'b000000011100_00000_101_10001_0000011;  
rom[27] = 32'b000000011100_00000_001_10010_0000011;  
rom[28] = 32'b000000011100_00000_010_10011_0000011;
```



```
lbu x15, 28(x0)  
lb x16, 28(x0)  
lhu x17, 28(x0)  
lh x18, 28(x0)  
lw x19, 28(x0)
```

- RAM의 주소 28(x0)에 있는 데이터를 Byte Unsigned로 Reg의 x15에 적재(Load).

- RAM의 주소 28(x0)에 있는 데이터를 Byte Signed로 Reg의 x16에 적재(Load)

- RAM의 주소 28(x0)에 있는 데이터를 Half Unsigned로 Reg의 x17에 적재(Load)

- RAM의 주소 28(x0)에 있는 데이터를 Half Signed로 Reg의 x18에 적재(Load)

- RAM의 주소 28(x0)에 있는 데이터를 Word로 Reg의 x19에 적재(Load)

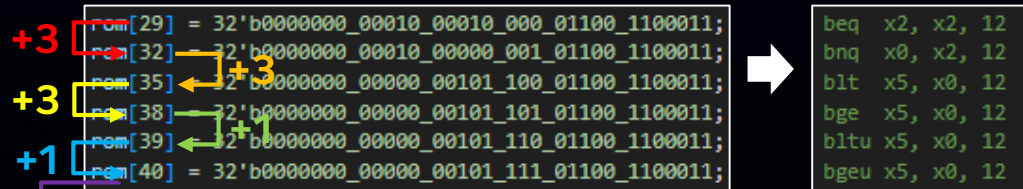
	RAM
0	
1	
2	
3	
4	
5	
6	32'b0...00_0000_0000_1000_0000
7	32'b0...00_1000_0001_1000_0000
8	32'b0...01_1000_0001_1000_0000

	RegFile
0	zero
1	11
2	12
3	13 -> 23
4	14 -> 1
5	15 -> 32'b11000...00
6	16 -> 0 -> 13
7	17 -> 0 -> 1
8	18 -> 1 -> 0
9	19 -> 0 -> 32'b0...00011
10	20 -> 32'b0...00111 -> 32'b0...01111
11	21 -> 32'b0..01111 -> 32'b0...01100
12	22 -> 32'b0..01000 -> 32'b0...0000
13	23 -> 32'b0..01_1000_0000
14	24 -> 32'b0..01_1000_0000_0000_0000
15	25 -> 32'b1..1_1000_0000 -> 32'b0..0_0000_0000_1000_0000
16	26 -> 32'b1..1_1000_0000
17	27 -> 32'b0..0_1000_0001_1000_0000
18	28 -> 32'b1..1_1000_0001_1000_0000
19	29 -> 32'b0...01_1000_0001_1000_0000
20	30 -> 32'b0...01_1000_0001_1000_0000

# 3. Addr/Data 동작 분석

## • B-Type

```
//rom[x]=32'b imm7 _ rs2 _ rs1 _f3 _imm5 _ opcode
```



- 주소 x2, x2에 있는 값이 동등할 경우, 다음 PC가 immediate(12) 만큼 분기(branch).  
=> 분기 O
- 주소 x0, x2에 있는 값이 다를 경우, 다음 PC가 immediate(12) 만큼 분기.  
=> 분기 O
- 주소 x5에 있는 signed 값이 주소 x0에 있는 값보다 작을 경우, 다음 PC가 immediate(12)만큼 분기.  
=> 분기 O
- 주소 x5에 있는 signed 값이 주소 x0에 있는 값보다 클 경우, 다음 PC가 immediate(12)만큼 분기.  
=> 분기 X
- 주소 x5에 있는 unsigned 값이 주소 x0에 있는 값보다 작을 경우, 다음 PC가 immediate(12)만큼 분기.  
=> 분기 X
- 주소 x5에 있는 unsigned 값이 주소 x0에 있는 값보다 클 경우, 다음 PC가 immediate(12)만큼 분기.  
=> 분기 O

RegFile	
zero	
11	
12	
13 -> 23	
14 -> 1	
15 -> 32'b11000...00	
16 -> 0 -> 13	
17 -> 0 -> 1	
18 -> 1 -> 0	
19 -> 0 -> 32'b0...000011	
20 -> 32'b0...00111 -> 32'b0...01111	
21 -> 32'b0...01111 -> 32'b0...01100	
22 -> 32'b0...01000 -> 32'b0...00000	
23 -> 32'b0...01_1000_0000	
24 -> 32'b0...01_1000_0000_0000_0000	
25 -> 32'b1..1_1000_0000 -> 32'b0..0_0000_0000_1000_0000	
26 -> 32'b1..1_1000_0000	
27 -> 32'b0..0_1000_0001_1000_0000	
28 -> 32'b1..1_1000_0001_1000_0000	
29 -> 32'b0...01_1000_0001_1000_0000	
30 -> 32'b0...01_1000_0001_1000_0000	



# 3. Addr/Data 동작 분석

## • LU/AU/J/JL-Type

```
//rom[x]= 32'b      imm20      _ rd _ op
```

```
rom[43] = 32'b000000000000000000000011_01010_0110111;  
rom[44] = 32'b000000000000000000000000_01011_0010111;
```



```
lui  x10, 3  
auipc x11, 0
```

```
//rom[x]= 32'b      imm(20)      _ rd _ op
```

```
rom[45] = 32'b0000000001100000000000_01100_1101111;
```



```
jal  x12, 12
```

```
//rom[x]= 32'b      imm(12)      _ rs1 _f3 _ rd _ op
```

```
rom[48] = 32'b0000000010000_01100_000_01101_1100111;
```



```
jalr x13, x12, 16
```

- immediate(3)을 '12만큼 upper(<< 12)' 한 값을 주소 x10에 적재(load).

- immediate(0)을 '12만큼 upper(<< 12)'한 값과 PC를 더해서 주소 11에 적재.

- 다음 PC가 immediate(12)만큼 Jump하고, 기존의 PC는 4를 더해서 x12에 적재.

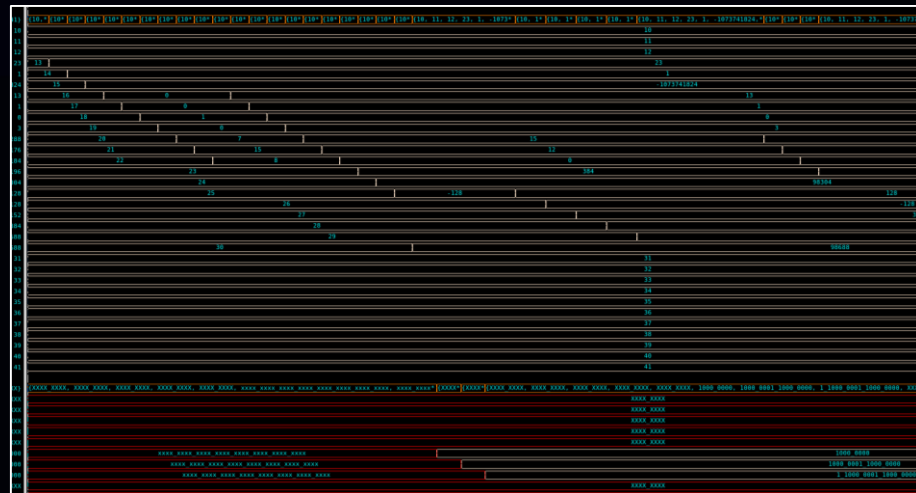
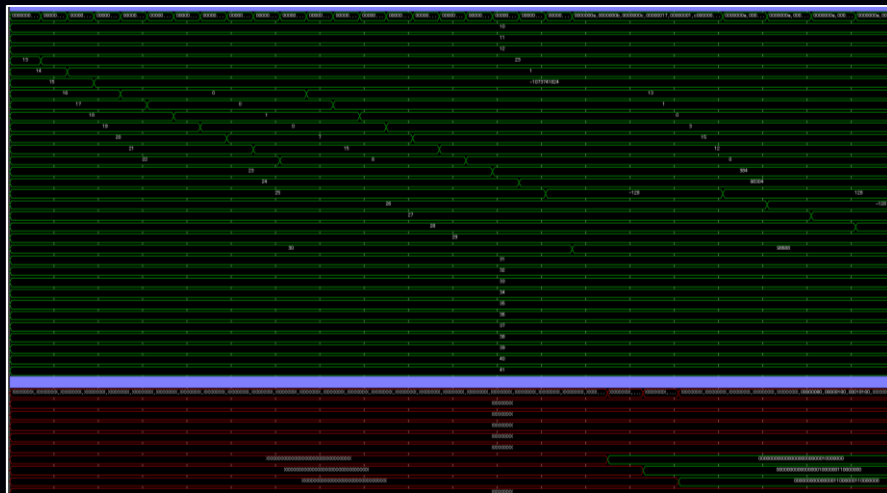
- 다음 PC는 주소 x12에 있는 값과 immediate(16)를 더한 값으로 Jump하고, 기존의 PC는 4를 더해서 x13에 적재.

0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20

RegFile	
zero	
11	
12	
13 -> 23	
14 -> 1	
15 -> 32'b11000...00	
16 -> 0 -> 13	
17 -> 0 -> 1	
18 -> 1 -> 0	
19 -> 0 -> 32'b0...00011	
20 -> 32'b0...00111 -> 32'b0...01111 -> 32'b0...011_0000_0000_0000	
21 -> 32'b0...01111 -> 32'b0...01100 -> 176	
22 -> 32'b0...01000 -> 32'b0...000 -> 184	
23 -> 32'b0...01_1000_0000 -> 196	
24 -> 32'b0...01_1000_0000_0000_0000	
25 -> 32'b1..1_1000_0000 -> 32'b0..0_0000_0000_1000_0000	
26 -> 32'b1..1_1000_0000	
27 -> 32'b0..0_1000_0001_1000_0000	
28 -> 32'b1..1_1000_0001_1000_0000	
29 -> 32'b0...01_1000_0001_1000_0000	
30 -> 32'b0...01_1000_0001_1000_0000	

# 4. Timing Diagram

- Simulation(Timing Chart) 교차 분석



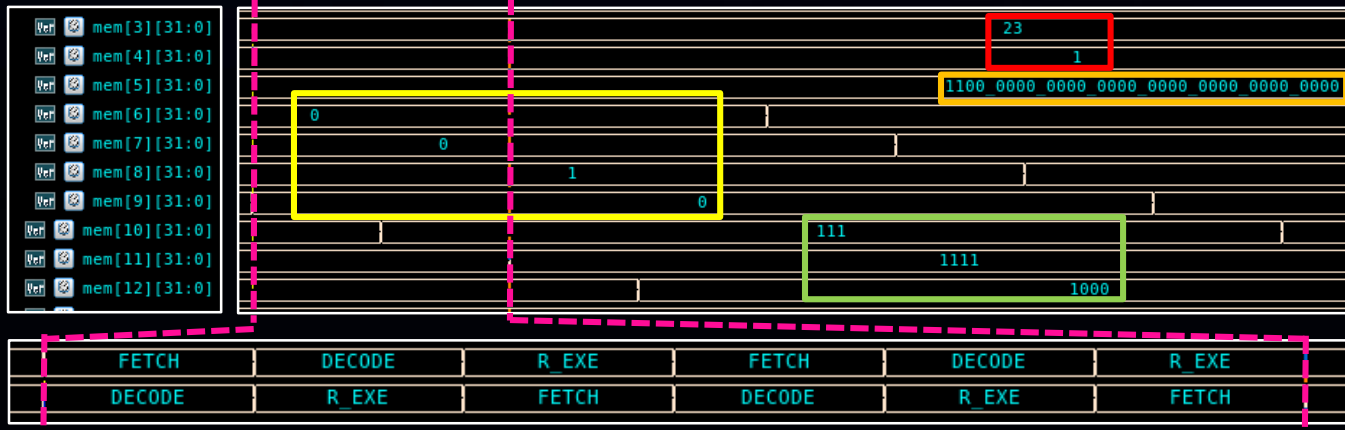
Xilinx Vivado



Synopsys Verdi

# 4. Timing Diagram

## • R-Type



- FETCH -> DECODE -> R\_EXE 의 순서로 state가 반복.

- mem[8], mem[9] : signed/unsigned 차이를 둔 결과, 다른 데이터(1/0)가 적재.

RegFile	
0	zero
1	11
2	12
3	13 -> 23
4	14 -> 1
5	15 -> 32'b11000...00
6	16 -> 0
7	17 -> 0
8	18 -> 1
9	19 -> 0
10	20 -> 32'b0..00111
11	21 -> 32'b0..01111
12	22 -> 32'b0..01000
13	23
14	24
15	25
16	26
17	27
18	28
19	29
20	30

```

addi x6, x2, 1
slli x7, x5, 1
sltiu x8, x5, 1
xori x9, x2, 15
ori x10, x2, 15
andi x11, x2, 15

```

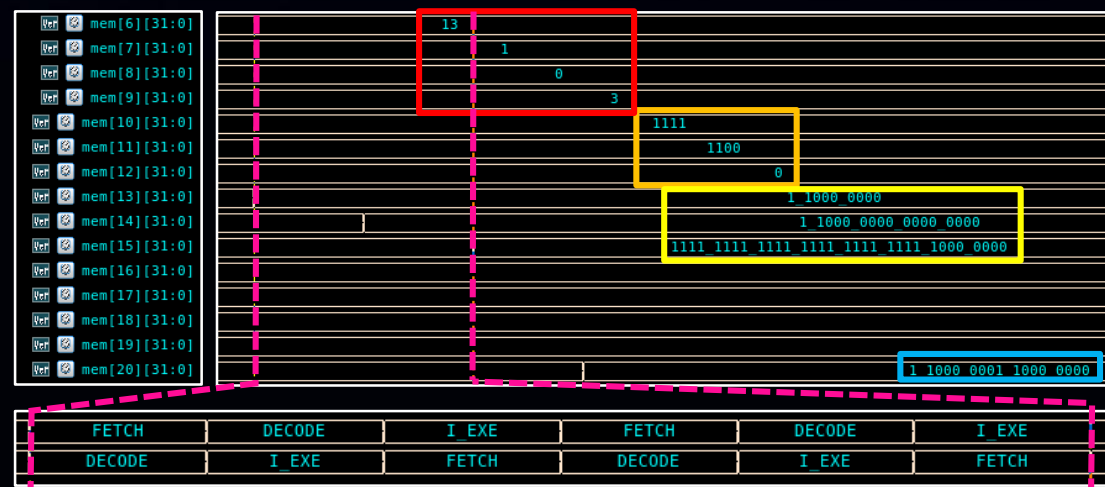
```

slli x12, x5, 23
srli x13, x5, 23
srli x14, x5, 15
srai x15, x5, 23
add x20, x13, x14

```

# 4. Timing Diagram

## I-Type



- mem[7], mem[8] : signed/unsigned 차이를 둔 결과, 다른 데이터(1/0)가 적재.
- mem[15] : sra(shift right arithmetic) 연산을 실행한 결과, msb인 1이 복제.

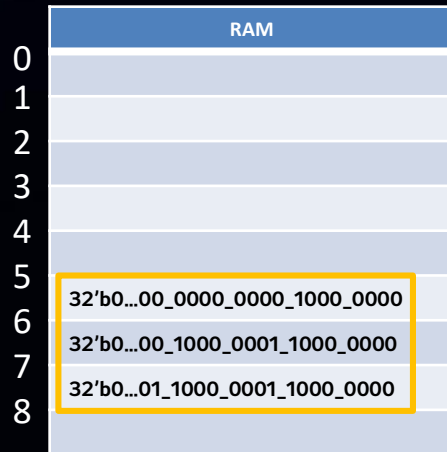
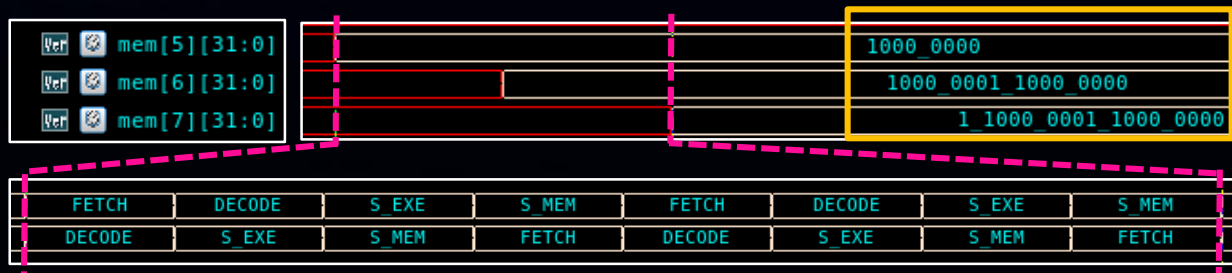
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20

RegFile	
zero	
11	
12	
13 -> 23	
14 -> 1	
15 -> 32'b11000...00	
16 -> 0 -> 13	
17 -> 0 -> 1	
18 -> 1 -> 0	
19 -> 0 -> 32'b0...000011	
20 -> 32'b0..00111 -> 32'b0...01111	
21 -> 32'b0..01111 -> 32'b0...01100	
22 -> 32'b0..01000 -> 32'b0...0000	
23 -> 32'b0..01_1000_0000	
24 -> 32'b0..01_1000_0000_0000_0000	
25 -> 32'b1..1_1000_0000	
26	
27	
28	
29	
30 -> 32'b0...01_1000_0001_1000_0000	

```
sb x20, 8(x2)
sh x20, 12(x2)
sw x20, 16(x2)
```

## 4. Timing Diagram

### • S-Type



- mem[5] : sb(store byte) 명령어를 실행하여 데이터의 8bit만 저장.
- mem[6] : sh(store half) 명령어를 실행하여 데이터의 16bit만 저장.
- mem[7] : sw(store word) 명령어를 실행하여 32bit가 온전히 저장.

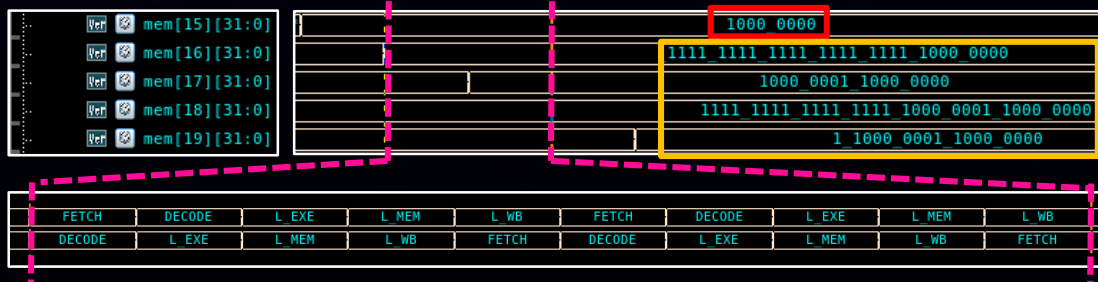
```

lbu  x15, 28(x0)
lb   x16, 28(x0)
lhu  x17, 28(x0)
lh   x18, 28(x0)
lw   x19, 28(x0)

```

## 4. Timing Diagram

### • L-Type



- mem[15] : lbu(load byte unsigned) 명령어의 실행으로 8bit가 적재.
- mem[16] : lb(load byte) 명령어의 실행으로 8bit가 sign extension 되어 적재.
- mem[17] : lhu(load half unsigned) 명령어의 실행으로 16bit가 적재.
- mem[18] : lh(load half) 명령어의 실행으로 16bit가 sign extension 되어 적재.
- mem[19] : lw(load word) 명령어의 실행으로 32bit가 적재.

0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20

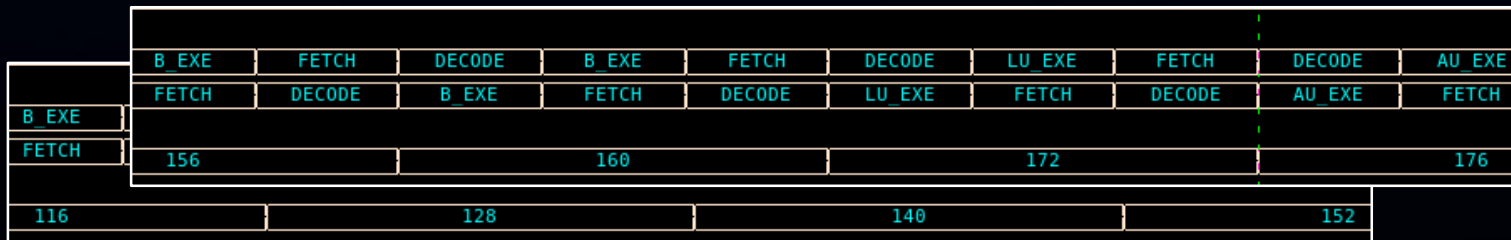
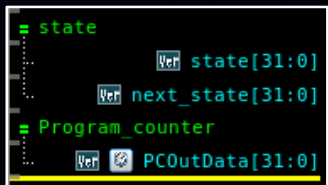
RegFile	
zero	
11	
12	
13 -> 23	
14 -> 1	
15 -> 32'b11000...00	
16 -> 0 -> 13	
17 -> 0 -> 1	
18 -> 1 -> 0	
19 -> 0 -> 32'b0...00011	
20 -> 32'b0...00111 -> 32'b0...01111	
21 -> 32'b0...01111 -> 32'b0...01100	
22 -> 32'b0...01000 -> 32'b0...0000	
23 -> 32'b0...01_1000_0000	
24 -> 32'b0...01_1000_0000_0000_0000	
25 -> 32'b1..1 1000 0000 -> 32'b0..0_0000_0000_1000_0000	
26 -> 32'b1..1_1000_0000	
27 -> 32'b0..0_1000_0001_1000_0000	
28 -> 32'b1..1_1000_0001_1000_0000	
29 -> 32'b0...01_1000_0001_1000_0000	
30 -> 32'b0...01_1000_0001_1000_0000	



# 4. Timing Diagram

```
beq x2, x2, 12
bnq x0, x2, 12
blt x5, x0, 12
bge x5, x0, 12
bltu x5, x0, 12
bgeu x5, x0, 12
```

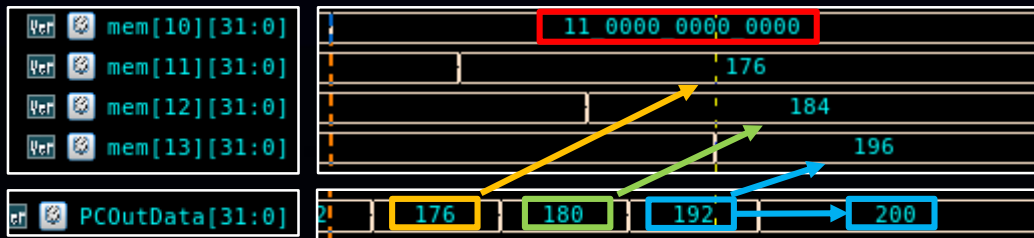
## • B-Type



- mem[29] : beq 명령어가 **True**로 실행되어, 다음 PC의 출력이 **분기**. → (다음 PC) =  $\{(29*4) + 12\} = 128$
- mem[32] : bne 명령어가 **True**로 실행되어, 다음 PC의 출력이 **분기**. → (다음 PC) =  $\{(32*4) + 12\} = 140$
- mem[35] : blt 명령어가 **True**로 실행되어, 다음 PC의 출력이 **분기**. → (다음 PC) =  $\{(35*4) + 12\} = 152$
- mem[38] : bge 명령어가 **False**로 실행되어, 다음 PC의 출력이 **유지**. → (다음 PC) =  $\{(38*4) + 4\} = 156$
- mem[39] : bltu 명령어가 **False**로 실행되어, 다음 PC의 출력이 **유지**. → (다음 PC) =  $\{(39*4) + 4\} = 160$
- mem[40] : bgeu 명령어가 **True**로 실행되어, 다음 PC의 출력이 **분기**. → (다음 PC) =  $\{(40*4) + 12\} = 172$

# 4. Timing Diagram

## • LU/AU/JAL/JALR-Type



- mem[43] : lui 명령어의 실행으로 (3 << 12)의 값이 RegFile에 저장.
- mem[44] : auipc 명령어의 실행으로 현재 PC의 값이 RegFile에 저장.
- mem[45] : 현재 PC의 주소에 4를 더하여 RegFile에 저장하고, 다음 PC는 Jump.
- mem[48] : 현재 PC의 주소에 4를 더하여 RegFile에 저장하고, 다음 PC는 Jump.

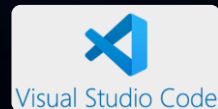
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20

RegFile	
zero	
11	
12	
13 -> 23	
14 -> 1	
15 -> 32'b11000...00	
16 -> 0 -> 13	
17 -> 0 -> 1	
18 -> 1 -> 0	
19 -> 0 -> 32'b0...00011	
20 -> 32'b0..00111 -> 32'b0...01111 -> 32'b0...011_0000_0000_0000	
21 -> 32'b0..01111 -> 32'b0...01100 -> 176	
22 -> 32'b0..01000 -> 32'b0...000 -> 184	
23 -> 32'b0..01_1000_0000 -> 196	
24 -> 32'b0..01_1000_0000_0000_0000	
25 -> 32'b1..1_1000_0000 -> 32'b0..0_0000_0000_1000_0000	
26 -> 32'b1..1_1000_0000	
27 -> 32'b0..0_1000_0001_1000_0000	
28 -> 32'b1..1_1000_0001_1000_0000	
29 -> 32'b0...01_1000_0001_1000_0000	
30 -> 32'b0...01_1000_0001_1000_0000	

# 5. Test Program 설명 및 분석



RISC-V Online Assembler



```
void sort(int *pdata, int size)
{
    void swap(int *pa, int *pb);

    int main()
    {
        int arData[] = {5, 4, 3, 2, 1};
        sort(arData, 5);
        return 0;
    }

    void sort(int *pdata, int size)
    {
        for(int i = 0; i < size; i++) {
            for(int j = i + 1; j < size; j++) {
                if(pdata[j] < pdata[i]) {
                    swap(&pdata[i], &pdata[j]);
                }
            }
        }
    }

    void swap(int *pa, int *pb)
    {
        int temp;
        temp = *pa;
        *pa = *pb;
        *pb = temp;
    }
}
```

sort.c

```
RISC-V (32-bits) gcc (tru v... -OO
1 main:
2 addi sp,sp,-48
3 sw ra,44(sp)
4 sw s0,40(sp)
5 addi s0,sp,48
6 sw zero,-40(s0)
7 sw zero,-36(s0)
8 sw zero,-32(s0)
9 sw zero,-28(s0)
10 sw zero,-24(s0)
11 sw zero,-20(s0)
12 li a5,5
13 sw a5,-40(s0)
14 li a5,4
15 sw a5,-36(s0)
16 li a5,3
17 sw a5,-32(s0)
18 li a5,2
19 sw a5,-28(s0)
20 li a5,1
21 sw a5,-24(s0)
22 addi sp,sp,-48
23 li a5,5
24 mv a0,a5
25 call sort
26 li a5,0
27 mv a0,a5
28 jr ra
29
30 sort:
31 addi sp,sp,-48
32 sw ra,44(sp)
33 sw s0,40(sp)
34 addi s0,sp,48
35 sw zero,-40(s0)
36 sw zero,-36(s0)
37 sw zero,-32(s0)
38 sw zero,-28(s0)
39 sw zero,-24(s0)
40 j .L8
41 .L8:
42 sw zero,-20(s0)
43 j .L5
44 .L5:
45 lw a5,-24(s0)
46 slll a5,a5,2
47 lw a4,-36(s0)
48 add a0,a5,a4
49 lw a4,40(s0)
50 lw a4,-24(s0)
51 addi a5,a5,1
52 slll a5,a5,2
53 lw a3,-36(s0)
54 add a5,a5,a3
55 lw a3,40(s0)
56 bne a4,a5,.L5
57 lw a2,-24(s0)
```

Type the assembly code below and click Build

```
1 main:
2 li sp, 0x100
3 addi sp, sp, -48
4 sw ra, 44(sp)
5 sw s0, 40(sp)
6 addi s0, s0, 48
7 sw zero, -40(s0)
8 sw zero, -36(s0)
9 sw zero, -32(s0)
10 sw zero, -28(s0)
11 sw zero, -24(s0)
12 li a5, 5
13 sw a5, -40(s0)
14 li a5, 4
15 sw a5, -36(s0)
16 li a5, 3
17 sw a5, -32(s0)
18 li a5, 2
19 sw a5, -28(s0)
20 li a5, 1
21 sw a5, -24(s0)
22 addi sp, sp, -48
23 li a5, 5
24 mv a0, a5
25 call sort
26 li a5, 0
27 mv a0, a5
28 jr ra
29
30 sort:
31 addi sp, sp, -48
32 sw ra, 44(sp)
33 sw s0, 40(sp)
34 addi s0, s0, 48
35 sw zero, -40(s0)
36 sw zero, -36(s0)
37 sw zero, -32(s0)
38 sw zero, -28(s0)
39 sw zero, -24(s0)
40 j .L8
41 .L8:
42 sw zero, -20(s0)
43 j .L5
44 .L5:
45 lw a5, -24(s0)
46 slll a5, a5, 2
47 lw a4, -36(s0)
48 add a0, a5, a4
49 lw a4, 40(s0)
50 lw a4, -24(s0)
51 addi a5, a5, 1
52 slll a5, a5, 2
53 lw a3, -36(s0)
54 add a5, a5, a3
55 lw a3, 40(s0)
56 bne a4, a5, .L5
57 lw a2, -24(s0)
```

Console

Code Hex Dump

```
10000113
fd010113
02112623
02812423
03010413
fc042c23
fc042e23
fe042023
fe042223
fe042423
fe042623
00500793
fcfc42c23
fcfc42e23
00400793
00400793
fcfc42e23
```

code.mem

```
C:\Users> kccicst -cpu_v32
1 10000113
2 fd010113
3 02112623
4 02812423
5 03010413
6 fc042c23
7 fc042e23
8 fe042023
9 fe042223
10 fe042423
11 fe042623
12 00500793
13 fcfc42c23
14 fcfc42e23
15 fe042023
16 fe042223
17 fe042423
18 fe042623
19 fe042823
20 fe042a23
21 fe042c23
22 fe042e23
23 00500793
24 00078513
25 01c000ef
26 00000793
27 00078513
28 02c12083
29 02012403
30 02812403
31 03010113
32 fd010113
33 02112623
34 02812423
35 03010413
36 fc042e23
37 fc042e23
38 fc042e23
39 09c000ef
40 fe042e23
41 070000ef
42 fe042783
43 00279793
44 fdc42703
45 00f707b3
```

ROM.sv

```
C:\Users> kccicst -cpu_v32
1 timescale 1ns / 1ps
2
3 module ROM (
4     input logic [31:0] addr,
5     output logic [31:0] data
6 );
7     logic [31:0] rom[0:2**7-1];
8     initial begin
9         $readmemh("code.mem", rom);
10    end
11    assign data = rom[addr[31:2]];
12
13 endmodule
```

ROM.sv (-cpu\_v32/multi\_cycle\_2) - GVIM@kccicst

```
1 timescale 1ns / 1ps
2
3 module ROM (
4     input logic [31:0] addr,
5     output logic [31:0] data
6 );
7     logic [31:0] rom[0:2**7-1];
8     initial begin
9         $readmemh("code.txt", rom);
10    end
```

## 5. Test Program 설명 및 분석

```

47: always #5 clk = !clk;
48:
49: localparam RAM_max_addr = 32'h1000000;
50:
51: int i, j;
52: logic [32:0] result_addr_array [0:4];
53:
54: initial begin
55:     #0;
56:     j = 0;
57:
58:     for(int k = 0; k < 5; k++) begin
59:         result_addr_array[k] = 32'h0;
60:     end
61:
62:     clk = 0; reset = 1;
63:     #10;
64:     reset = 0;
65:
66:     #10000000;
67:
68:     for(i = 0; i < RAM_max_addr; i++) begin
69:         if(dut_U_RAM.mem[i] == j+1) begin
70:             if(j == 5) begin
71:                 break; // 반복문 탈출
72:             end
73:             result_addr_array[j] = i;
74:             j = j + 1;
75:         end else begin
76:             j = 0;
77:         end
78:     end
79:
80:     if(dut_U_RAM.mem[result_addr_array[0]] != 32'd1 ||
81:        dut_U_RAM.mem[result_addr_array[1]] != 32'd2 ||
82:        dut_U_RAM.mem[result_addr_array[2]] != 32'd3 ||
83:        dut_U_RAM.mem[result_addr_array[3]] != 32'd4 ||
84:        dut_U_RAM.mem[result_addr_array[4]] != 32'd5)
85:         begin
86:             $display("Sort Failed!");
87:         end else begin
88:             $display("Sort Success! : RAM[%d:%d] = [%d, %d, %d, %d, %d]
89:                result_addr_array[0], result_addr_array[4], dut
90:
91:
92:             $finish;
93:         end

```

The screenshot shows the Tcl Console interface with a toolbar at the top containing icons for search, zoom, run, pause, copy, paste, and delete. The console output displays the following sequence of events:

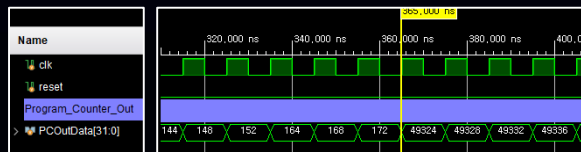
- Compiling module xil\_defaultlib.registerEn
- Compiling module xil\_defaultlib.DataPath
- Compiling module xil\_defaultlib.CPU\_RV32I
- Compiling module xil\_defaultlib.RAM
- Compiling module xil\_defaultlib.MCU
- Compiling module xil\_defaultlib.tb\_rv32i\_s
- Compiling module xil\_defaultlib.gtbl
- Built simulation snapshot tb\_rv32i\_s\_behav
- INFO: [USF-XSim-69] "elaborate" step finished in '2' seconds
- Vivado Simulator 2020.2
- Time resolution is 1 ps
- relaunch\_sim: Time (s): cpu = 00:00:01 ; elapsed = 00:00:05 . Memory (MB): peak = 1108.547 ; gain = 0.000
- A red box highlights the output: Sort Success! : RAM[ 54: 58] = [ 1, 2, 3, 4, 5]
- At the bottom, a file path is visible: C:\Users\...\.Xilinx\...\Project\_1\src\sim\sim\_behav.o

The diagram illustrates the execution of a parallel merge sort algorithm on 8 processors (P0 through P7). The processors are arranged in a horizontal row at the top. Below them, the array is shown as a series of horizontal bars representing data segments. The process begins with the array being split into 8 individual elements. The merge steps are numbered 1 through 5, showing how the data is combined back into sorted segments. The final result is a fully sorted array of 8 elements.

- 배열 [5, 4, 3, 2, 1, 0]에 대하여 0을 제외하고, 데이터를 오름차순으로 분류(sorting)된 데이터가 RAM에 저장(store).
- Testbench 코드에 검증을 위한 코드를 추가하여, 어떤 주소에 데이터가 정상적으로 배열된 것을 확인.

# 6. Trouble Shooting

## 1. Program Counter Issue



52			
53			//rom[x]=3
54			rom[29] =
55			rom[30] =
56			rom[31] =
57			rom[32] =
58			rom[33] =
59			rom[34] =
60			
	58		//rom[x]=
	59		rom[29] =
			rom[32] =
			rom[35] =
			rom[38] =
			rom[39] =
			rom[40] =

### # 문제 발생

- B-Type에 대하여 ISA Test를 진행할 때 Program Counter가 급격히 커지는 문제를 발견.

- 이전까지의 PC가 정상적이고, 출력이 32'bX 로 나오지 않는 것으로 보아, Data Path 내의 연결은 문제가 없을 것으로 추측.

### # 원인 추적

- State의 출력이 B\_EXE 임을 확인하여, Control Unit의 문제가 아님을 확인.

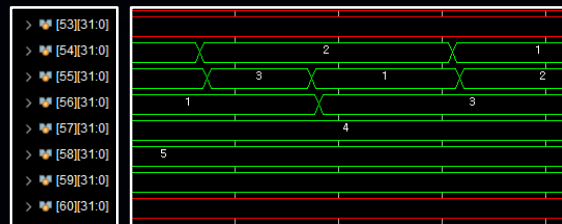
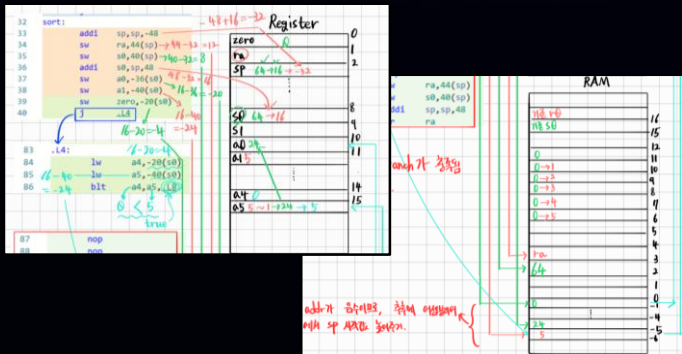
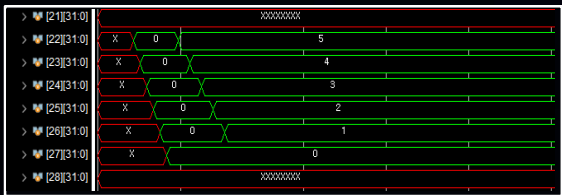
- 따라서, ROM Code에서 잘못된 정보를 기입하였을 것으로 추측.

### # 문제 해결

- 분기(Branch)가 발생할 경우, Program Counter의 값이 바뀌고, ROM의 Address도 이에 맞게 변경시켜서 문제를 해결.

- CPU의 구조와 RV32I의 명령어를 완벽히 이해하지 못해서 발생한 Trouble Shooting.

## 2. Stack Point 초기화



## # 문제 발생

- 초기값인 [5, 4, 3, 2, 1]은 RAM에  
저장이 되지만, 그 이후에는 데이터 변동이  
생기지 않는 문제를 발견.

- 초기값이 제대로 저장된 것을 통해서,  
Data Path에서 문제 원인이 발생한 것이  
아니라고 추측.

## # 원인 추적

- 직접 Register와 RAM의 data 및 address를 추적한 결과, Stack Pointer가 RAM의 음수를 가리키는 것을 발견.

- 현재 프로젝트는 System Verilog를 활용해서 RV32I를 구현한 것이므로, Stack Pointer가 음수를 가리켜서는 안된다.

## # 문제 해결

- 초기에 Stack Pointer의 데이터를  
설정할 때, 더 큰 값을 할당하여 문제 해결.

- ex) `li sp, 0x40` -> `li sp, 0x100`

## 7. 프로젝트 고찰

- CPU 및 메모리의 구조/동작

- RISC-V의 ISA를 배우면서 기존에 알고 있던 Computer Architecture에 대한 취약점을 보완.
- '어째서 CPU는 이렇게 복잡하고 까다로운 절차로 동작하는가?'에 대해서 숙고.

- 어셈블리 언어에 대한 지식

- 평소에 이해하지 못했던 어셈블리 언어에 대한 강의 및 과제를 진행하면서 이해도 증가.
- 해당 지식은 이후에 인베디드 분야로 나아가거나, 다른 CPU에 대해 학습할 때 큰 도움이 될 것.



# 감사합니다