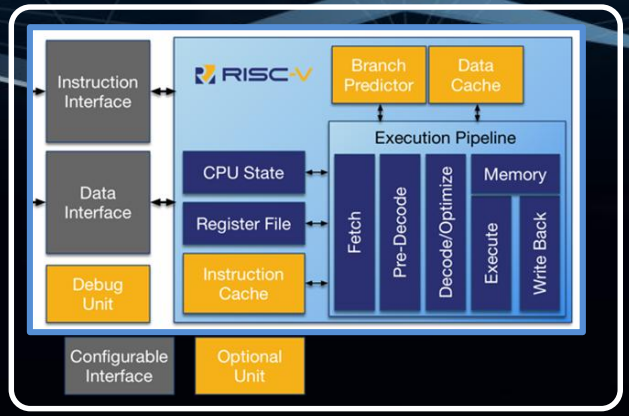
발표자 : 최지우

RISC-V RV32I Instruction Set Architecture

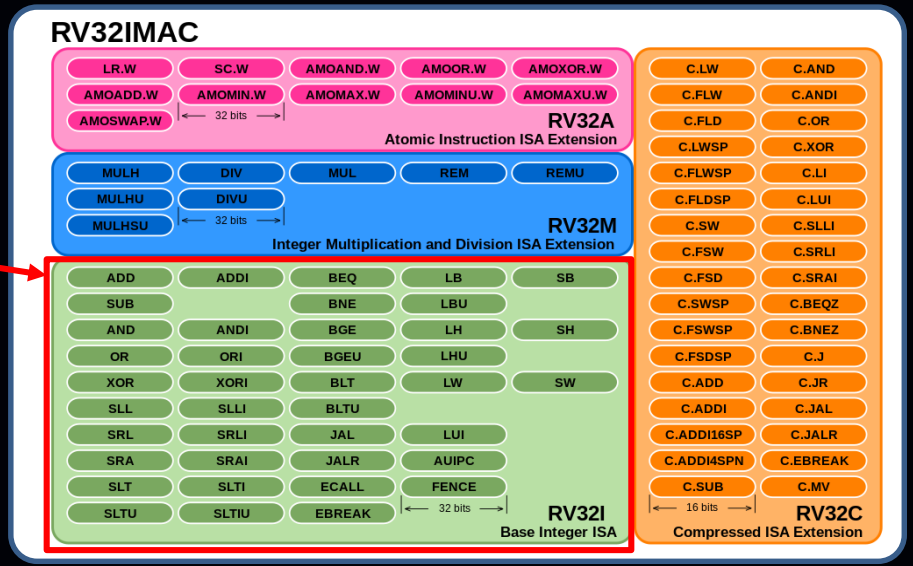
대한상공회의소
서울기술교육센터

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.



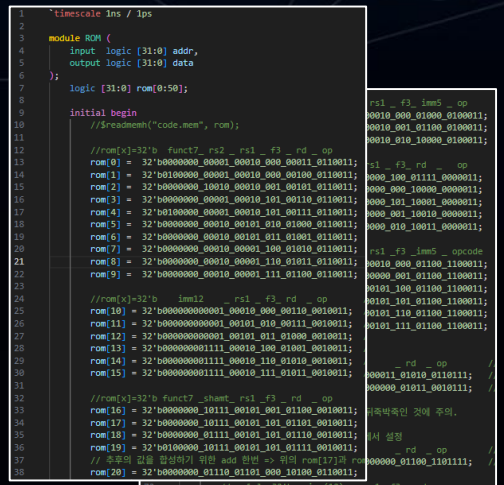
목차

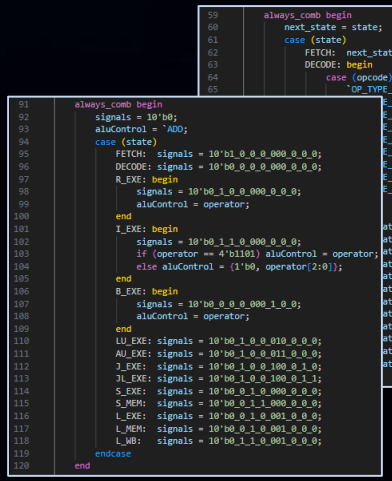
1. 개발 환경 및 언어 소개

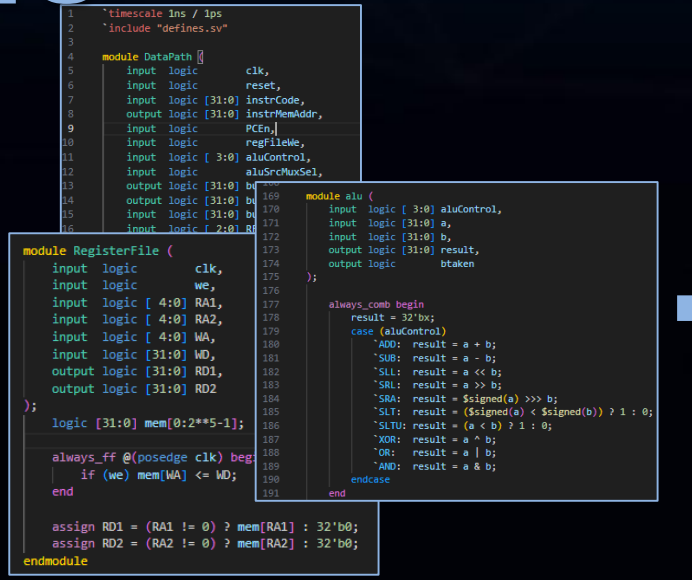
개발 환경

사용 언어

2. 코드 동작 설명







ROM

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

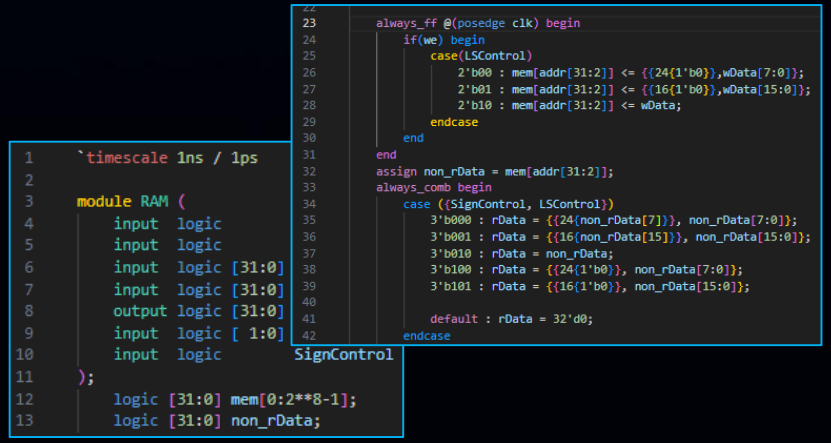
Control Unit

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

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

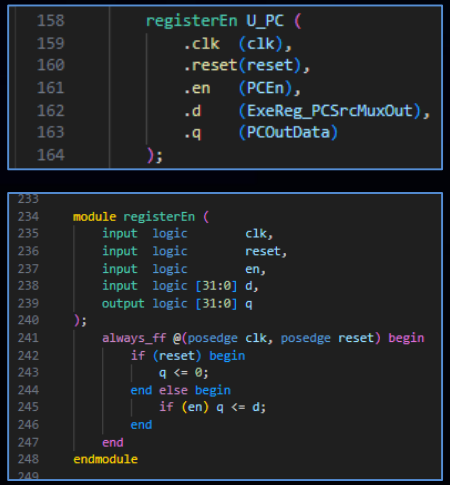


```
`timescale 1ns / 1ps

module RAM (
    input  logic
    input  logic
    input  logic [31:0]
    input  logic [31:0]
    output logic [31:0]
    input  logic [ 1:0]
    input  logic          SignControl
);
    logic [31:0] mem[0:2**8-1];
    logic [31:0] non_rData;

    always_ff @(posedge clk) begin
        if(we) begin
            case(LSControl)
                2'b00 : mem[addr[31:2]] <= {{24{1'b0}},wData[7:0]};
                2'b01 : mem[addr[31:2]] <= {{16{1'b0}},wData[15:0]};
                2'b10 : mem[addr[31:2]] <= wData;
            endcase
        end
    end
    assign non_rData = mem[addr[31:2]];
    always_comb begin
        case ({SignControl, LSControl})
            3'b000 : rData = {{24{non_rData[7]}}, non_rData[7:0]};
            3'b001 : rData = {{16{non_rData[15]}}, non_rData[15:0]};
            3'b010 : rData = non_rData;
            3'b100 : rData = {{24{1'b0}}, non_rData[7:0]};
            3'b101 : rData = {{16{1'b0}}, non_rData[15:0]};

            default : rData = 32'd0;
        endcase
```



```
    registerEn U_PC (
        .clk   (clk),
        .reset(reset),
        .en    (PCEn),
        .d     (ExeReg_PCSrcMuxOut),
        .q     (PCOutData)
    );

module registerEn (
    input  logic        clk,
    input  logic        reset,
    input  logic        en,
    input  logic [31:0] d,
    output logic [31:0] q
);
    always_ff @(posedge clk, posedge reset) begin
        if (reset) begin
            q <= 0;
        end else begin
            if (en) q <= d;
        end
    end
endmodule
```

RAM

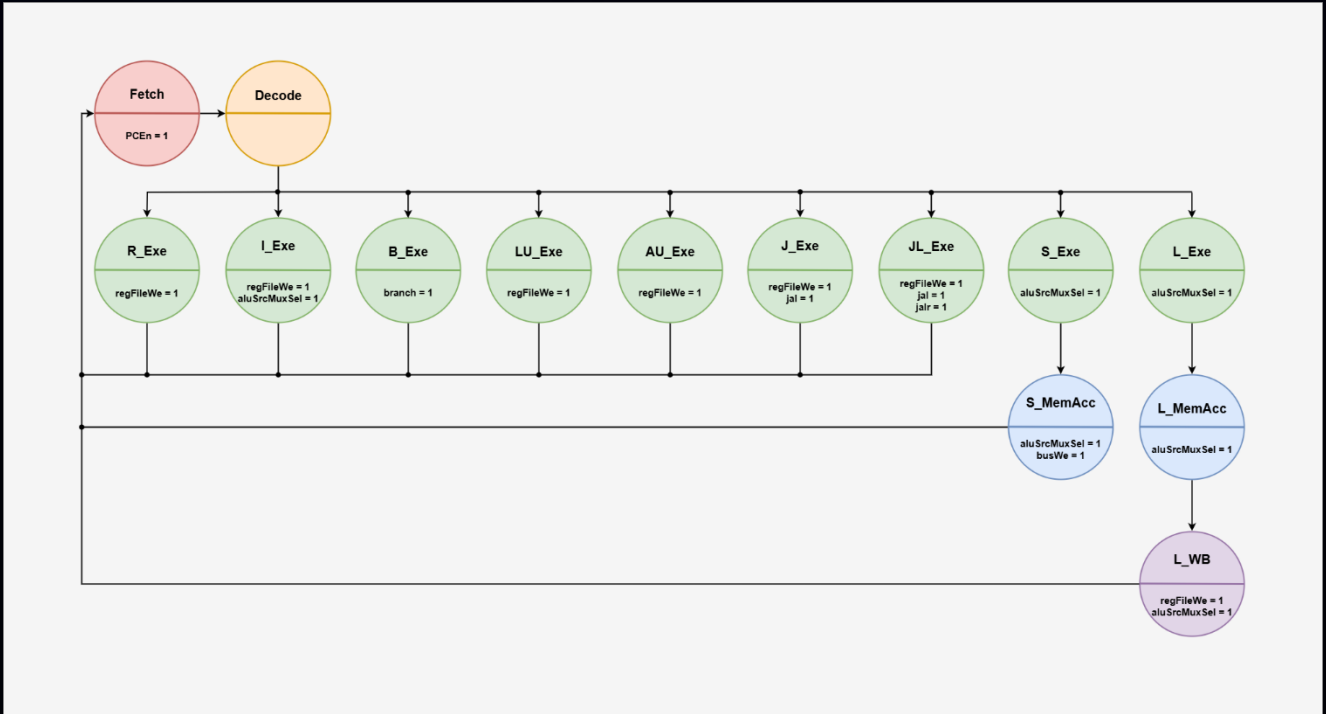
- S/L-Type의 명령어를 수행할 때 RAM 모듈이 활성화되고, Data의 저장/적재를 수행.
- Data의 Signed/Unsigned 및 Byte/Half/Word 의 통제를 수행.

Program Counter

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

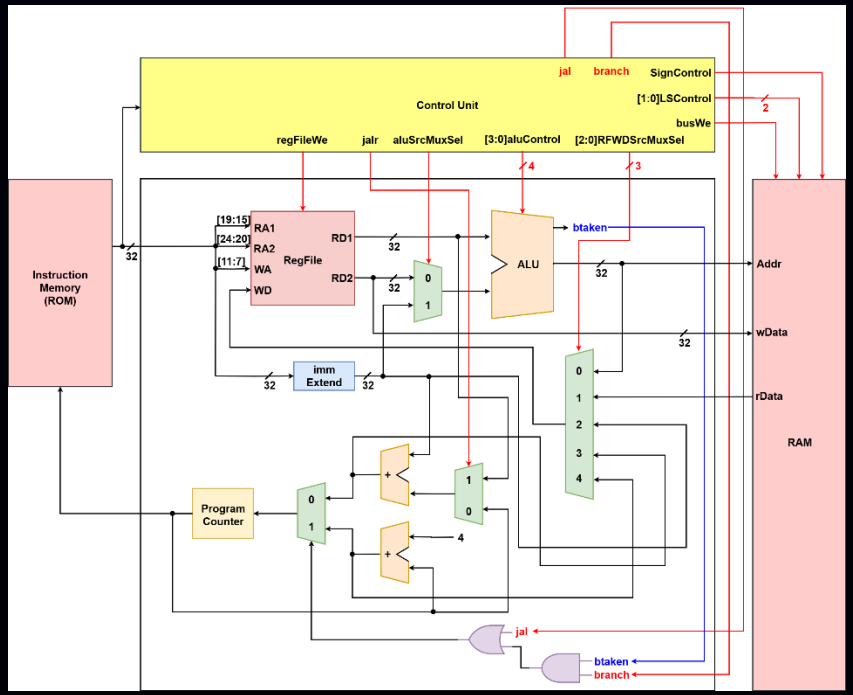
2. 코드 동작 설명

- FSM(Finite State Machine)



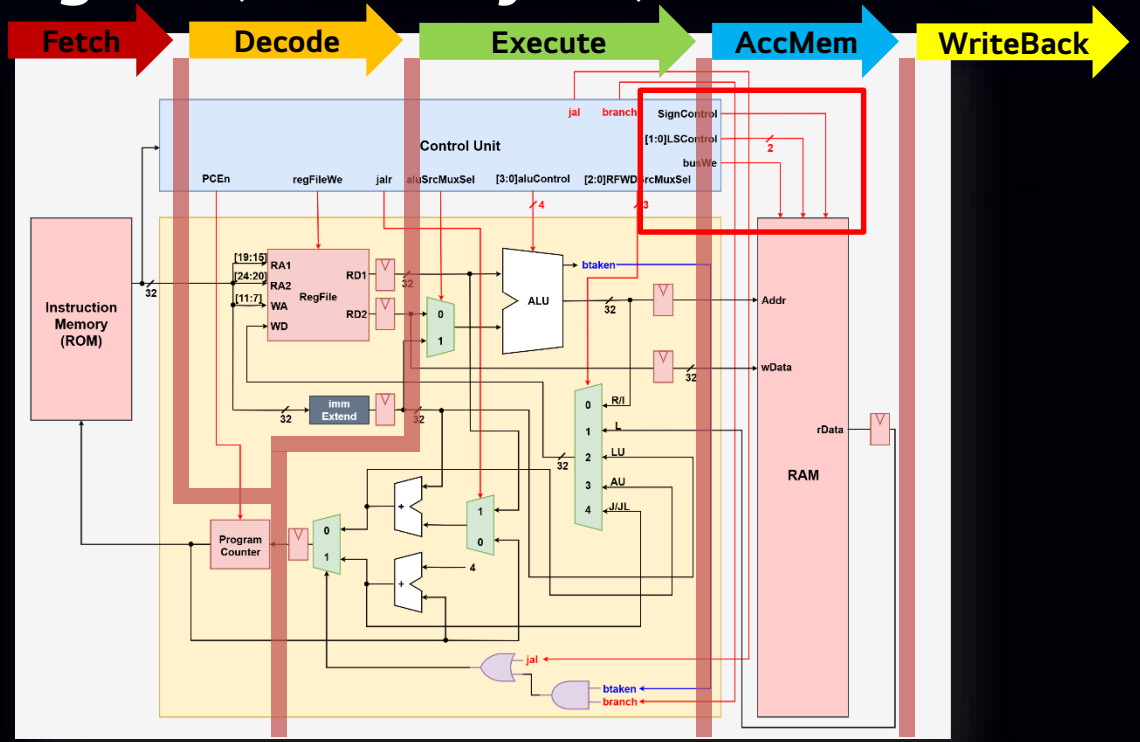
2. 코드 동작 설명

- 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 << 12)을 레지스터에 저장.
AU-Type	즉시값에 12 만큼의 shift left logical을 수행한 값(즉시값 << 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).

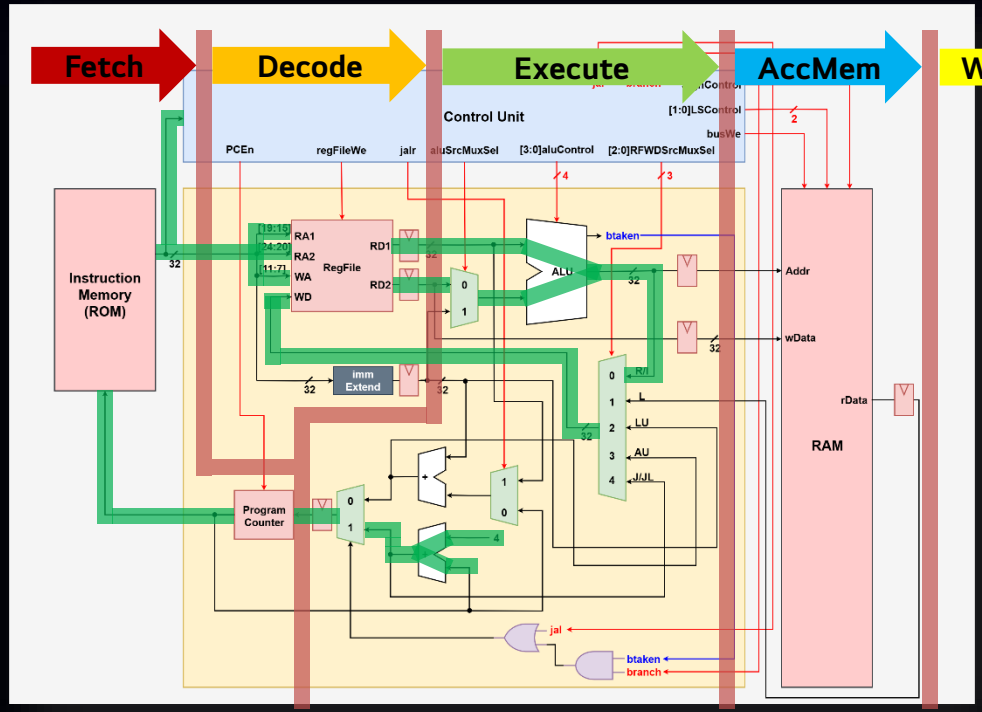
2. 코드 동작 설명

- **Block Diagram : R-Type**

- 두 레지스터의 값을 연산 후 결과를 다른 레지스터에 저장.

- 주로 산술 / 논리 연산.

- 즉시값 사용 X.

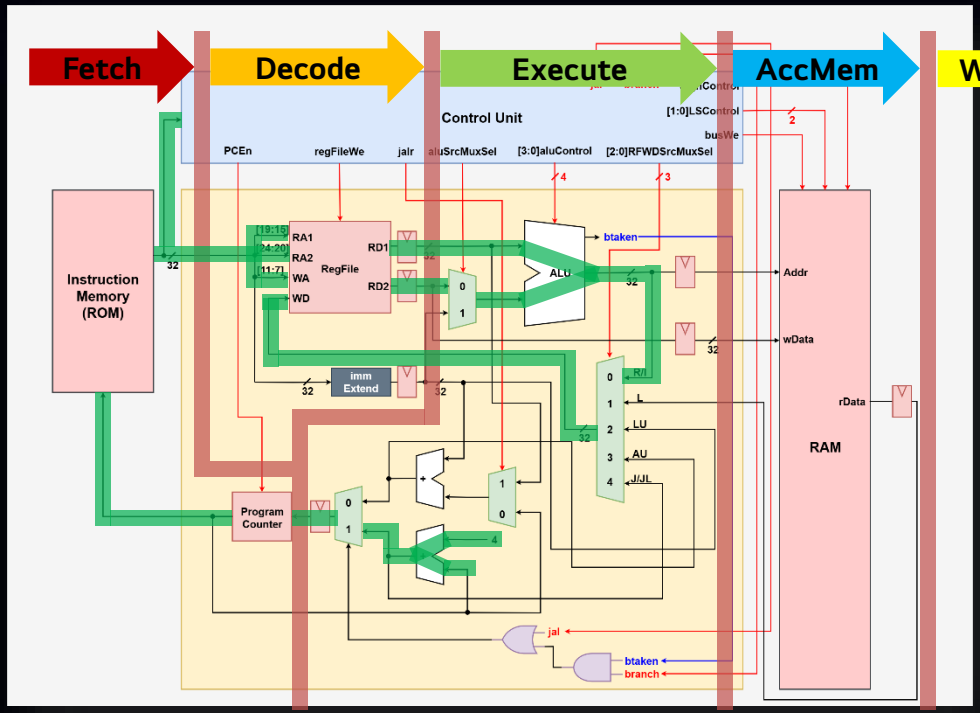


2. 코드 동작 설명

- **Block Diagram : I-Type**

- 즉시값(immediate, 12bit)과 레지스터 값을 연산.

- 산술/논리 연산의 즉시값 버전.

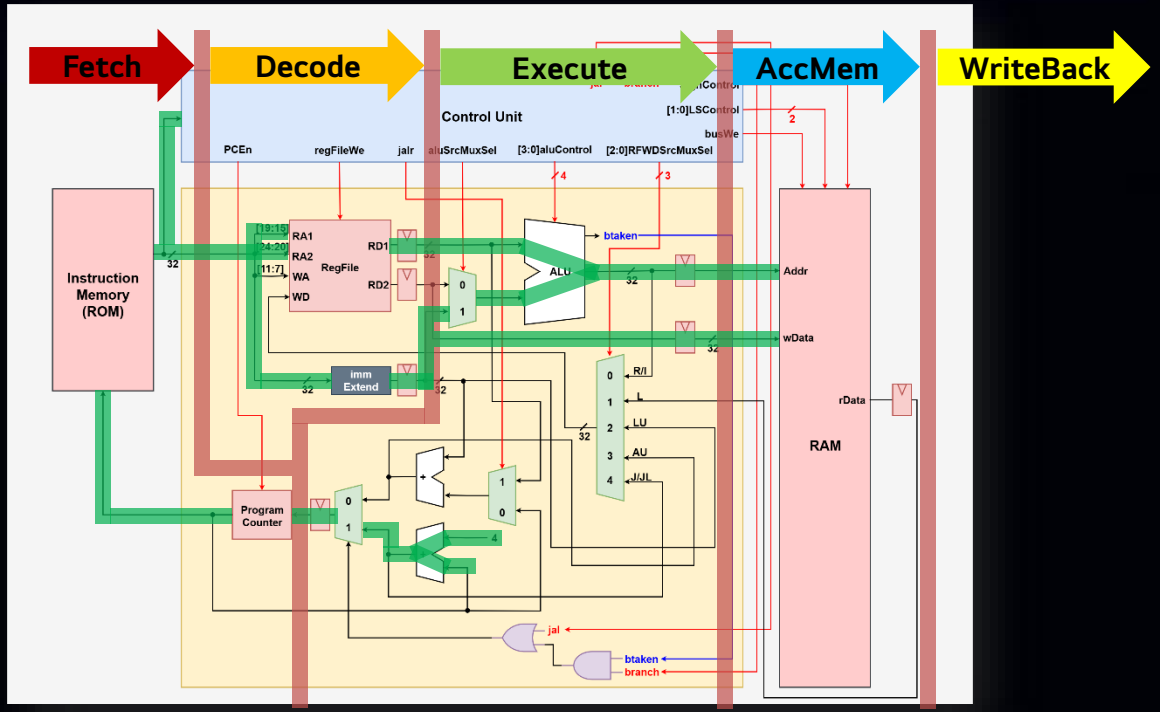


2. 코드 동작 설명

- **Block Diagram : S-Type**

- Store 명령어 전용.(레지스터 데이터를 메모리에 저장)

- 연산 결과를 rd가 아닌 메모리 주소에 저장하는 것에 주의.

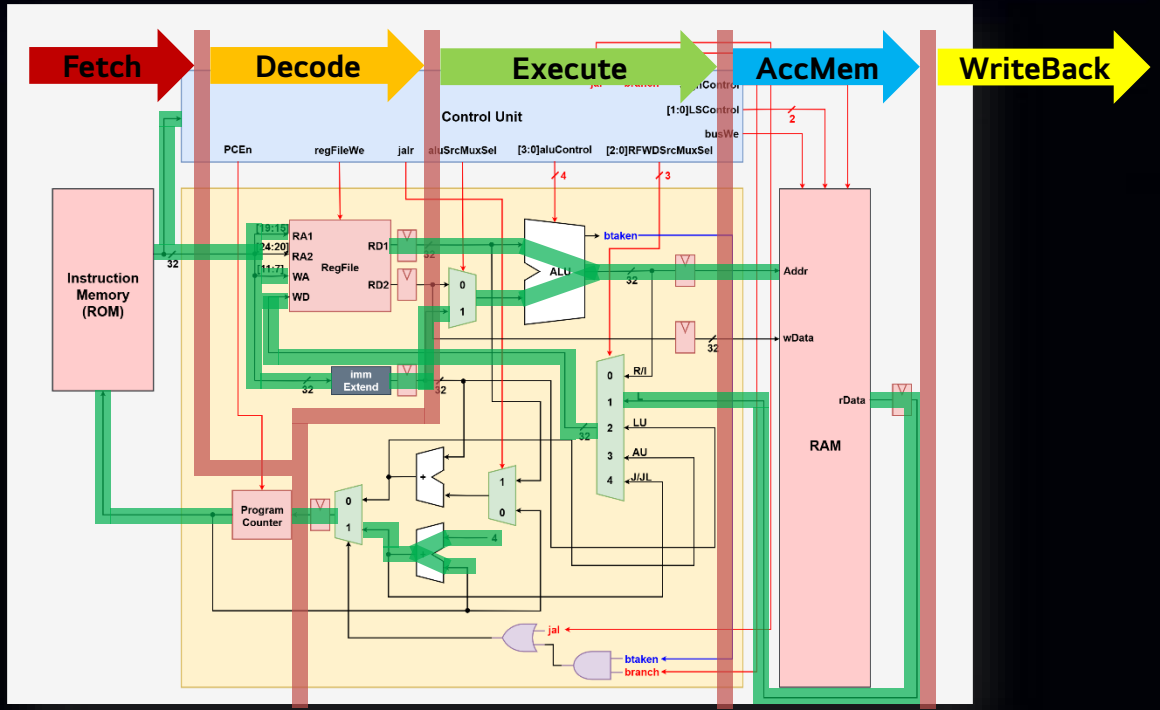


2. 코드 동작 설명

- **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.



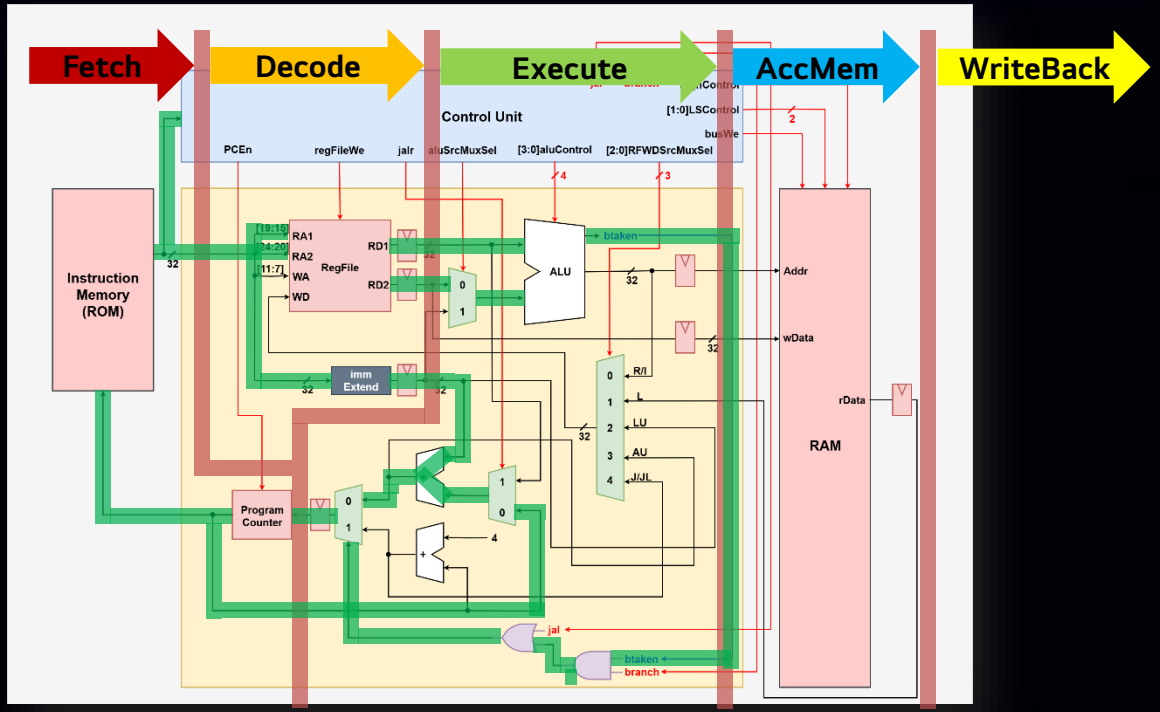
2. 코드 동작 설명

- **Block Diagram : B-Type**

- 조건 분기(Branch) 명령어 전용 Type.

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

- 조건식 평가 후 분기 수행.

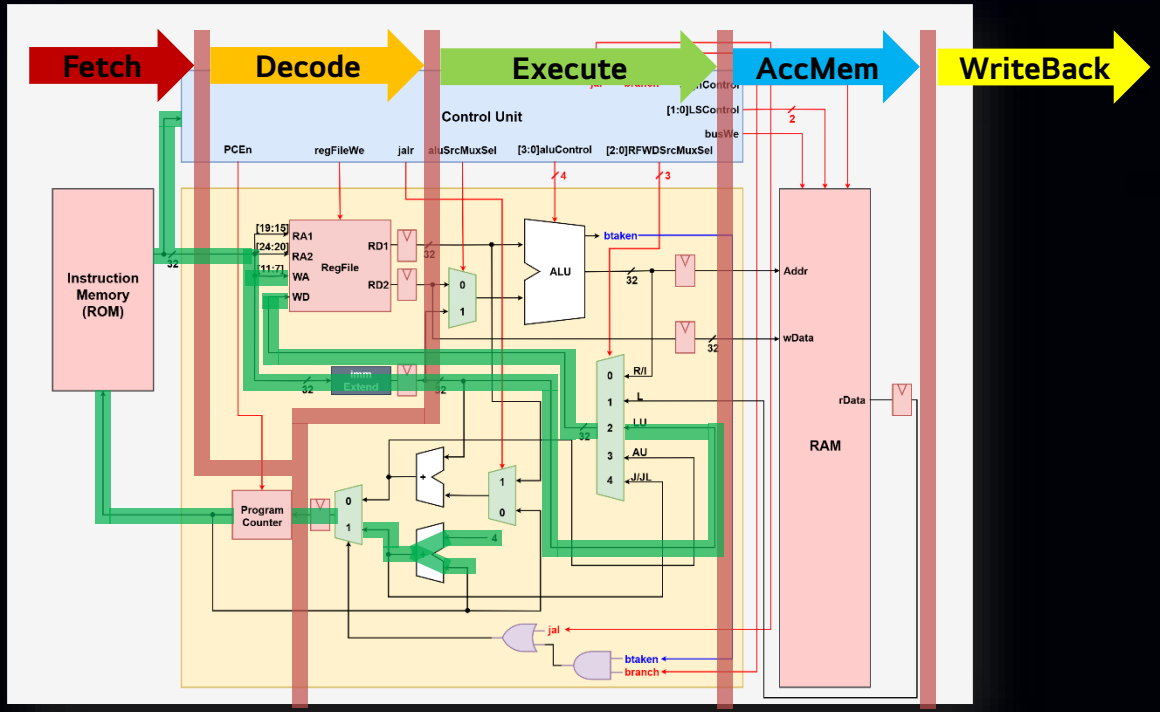


2. 코드 동작 설명

- **Block Diagram : U-Type(LU)**

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

- 큰 상수 로딩이나 주소 계산에 활용.

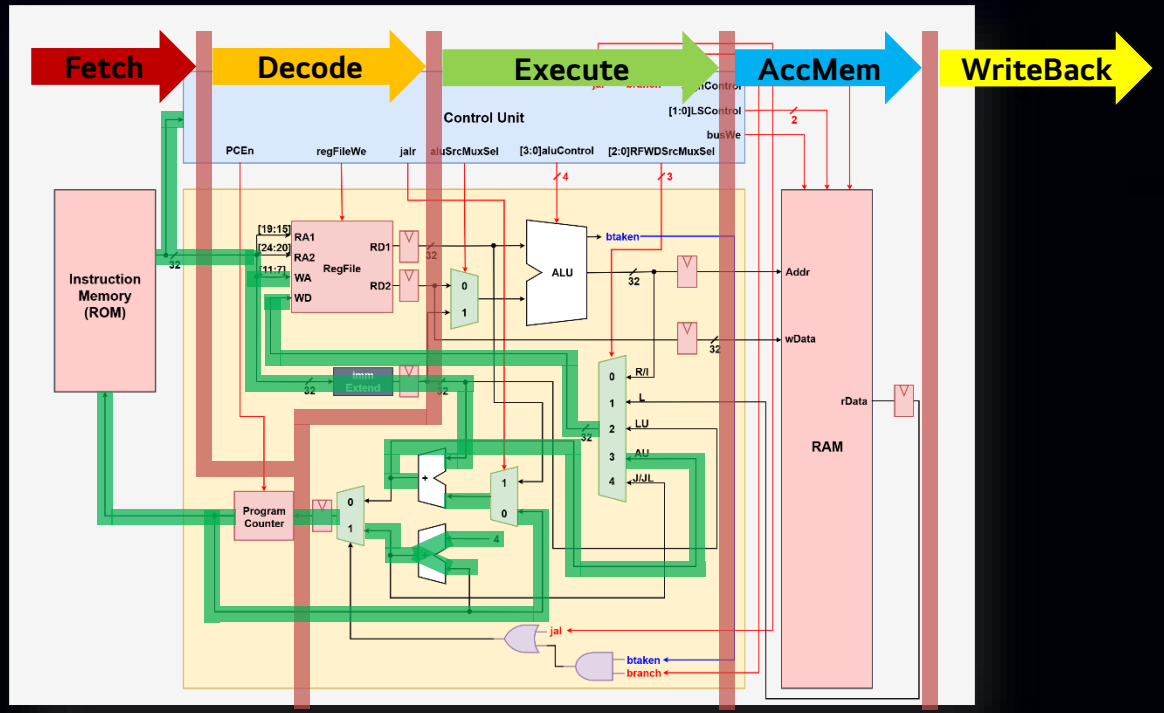


2. Block Diagram

- **Block Diagram : U-Type(AU)**

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

 - ex) auipc rd, imm20



2. Block Diagram

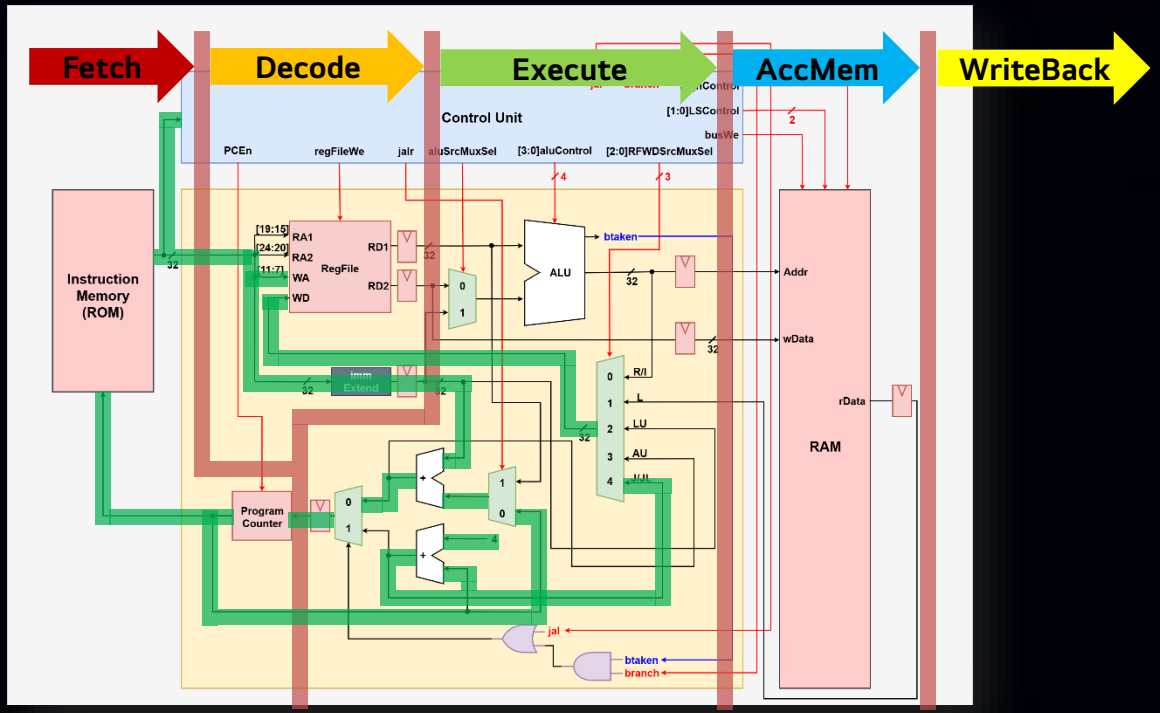
- **Block Diagram : J-Type(JAL)**

  - 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

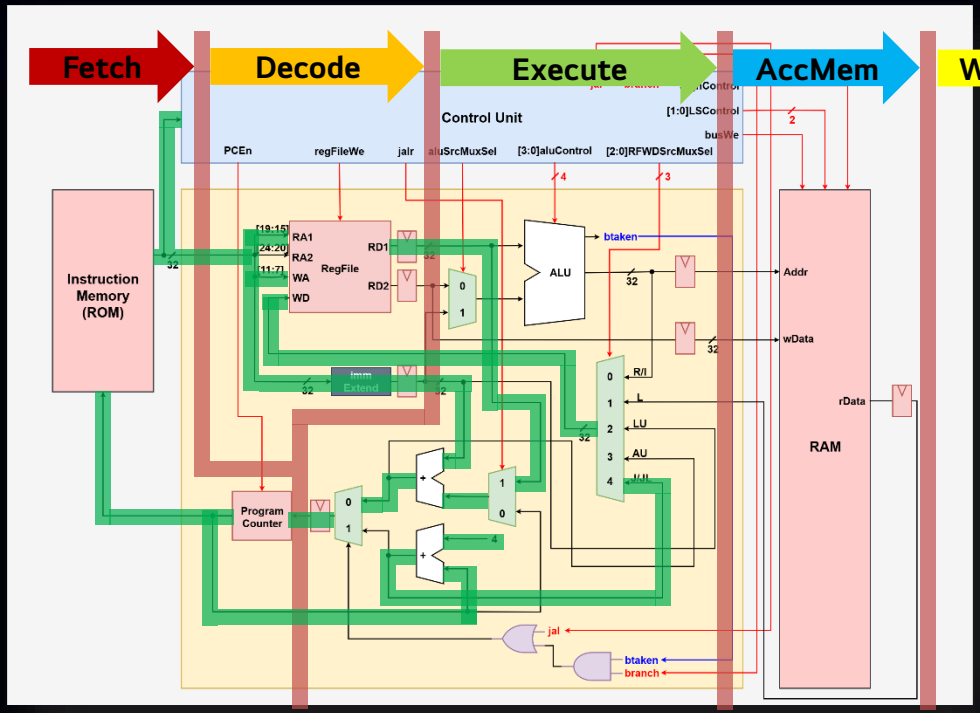
- **Block Diagram ：JALR-Type**

  - 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'b0000000_00001_00010_000_00011_0110011;
rom[1] =  32'b0100000_00001_00010_000_00100_0110011;
rom[2] =  32'b0000000_10010_00010_001_00101_0110011;
rom[3] =  32'b0000000_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
```

	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
9	19
10	20
11	21
12	22
13	23
14	24
15	25
16	26
17	27
18	28
19	29
20	30

- 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 x6 에 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.

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
```

	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

- 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.

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'b000000001111_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
```

	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
13	23
14	24
15	25
16	26
17	27
18	28
19	29
20	30

- 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 immediate값(1111)에 andi 연산을 실행하여 주소 x11에 적재.

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
```

	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

- 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.

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	32'b0…00_0000_0000_1000_0000
6	32'b0…00_1000_0001_1000_0000
7	32'b0…01_1000_0001_1000_0000
8	

	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_01111_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	32'b0...00_0000_0000_1000_0000
6	32'b0...00_1000_0001_1000_0000
7	32'b0...01_1000_0001_1000_0000
8	

	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
rom[29] = 32'b0000000_00010_00010_000_01100_1100011;
rom[32] = 32'b0000000_00010_00000_001_01100_1100011;
rom[35] = 32'b0000000_00000_00101_100_01100_1100011;
rom[38] = 32'b0000000_00000_00101_101_01100_1100011;
rom[39] = 32'b0000000_00000_00101_110_01100_1100011;
rom[40] = 32'b0000000_00000_00101_111_01100_1100011;
```

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

+3 +3 +3 +1 +1 +3

- 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
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

3. Addr/Data 동작 분석

- **LU/AU/J/JL-Type**

```
//rom[x]= 32'b        imm20        _ rd  _ op
rom[43] = 32'b00000000000000000011_01010_0110111;
rom[44] = 32'b00000000000000000000_01011_0010111;

//rom[x]= 32'b      imm(20)        _ rd  _ op
rom[45] = 32'b00000000110000000000_01100_1101111;

//rom[x]= 32'b   imm(12)  _ rs1 _f3 _ rd  _  op
rom[48] = 32'b000000010000_01100_000_01101_1100111;
```

```
lui   x10, 3
auipc x11, 0
jal   x12, 12
jalr x13, x12, 16
```

	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 -> 32'b0...011_0000_0000_0000
11	21 -> 32'b0..01111 -> 32'b0...01100 -> 176
12	22 -> 32'b0..01000 -> 32'b0...000 -> 184
13	23 -> 32'b0..01_1000_0000 -> 196
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

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

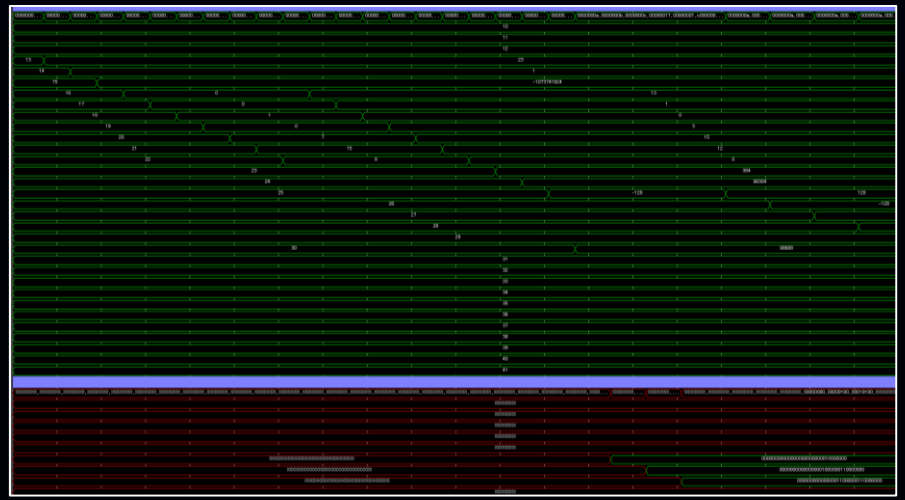
- immediate(0)을 '12만큼 upper(<< 12)'한 값과 PC를 더해서 주소 11에 적재.

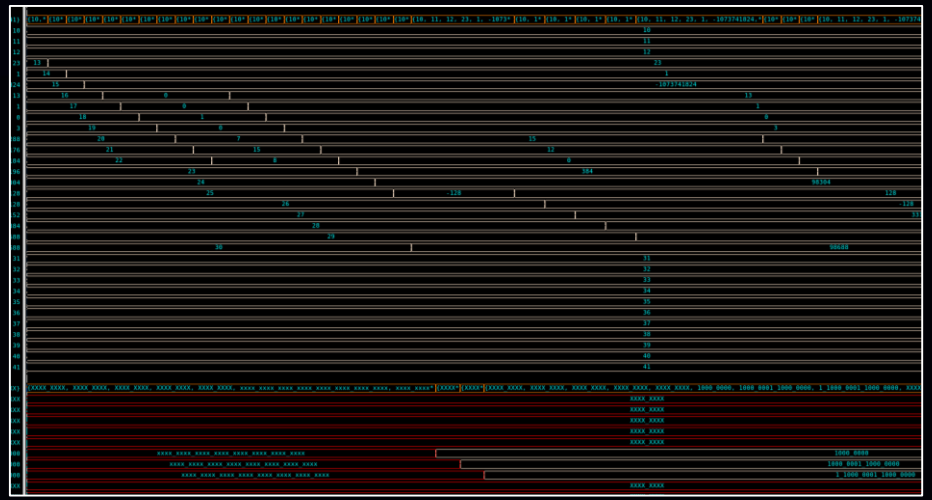
- 다음 PC가 immediate(12)만큼 Jump하고, 기존의 PC는 4를 더해서 x12에 적재.

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

4. Timing Diagram

- Simulation(Timing Chart) 교차 분석





Xilinx Vivado

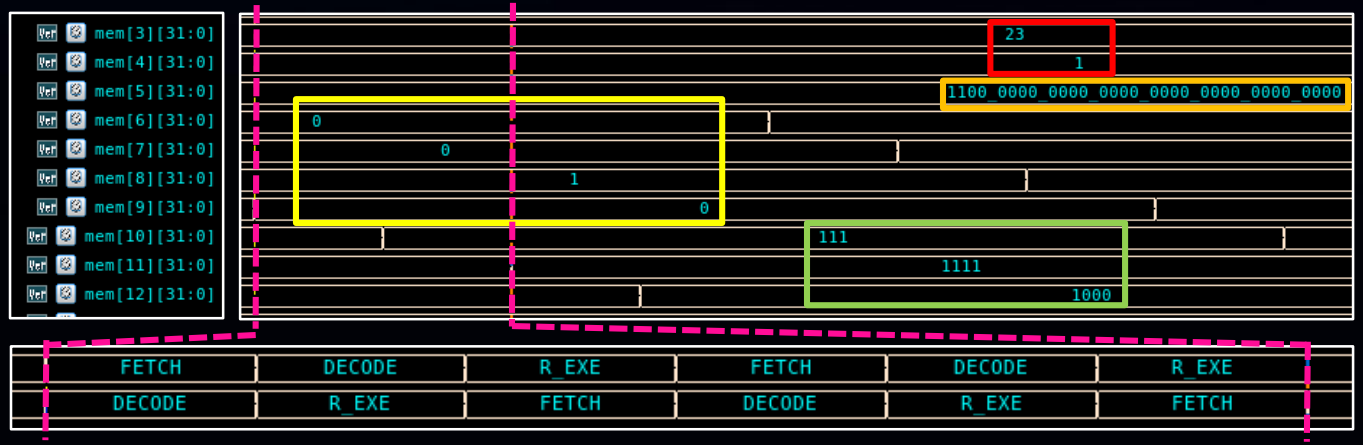
Synopsys Verdi

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

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

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

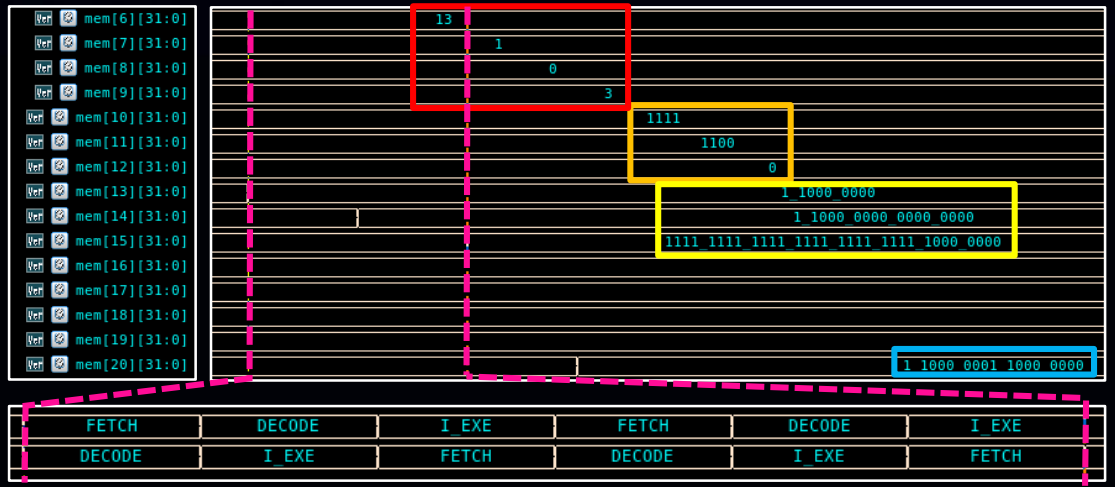
```
addi x6, x2, 1
slti x7, x5, 1
sltiu x8, x5, 1
xori x9, x2, 15
ori  x10, x2, 15
andi x11, x2, 15
```

```
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.

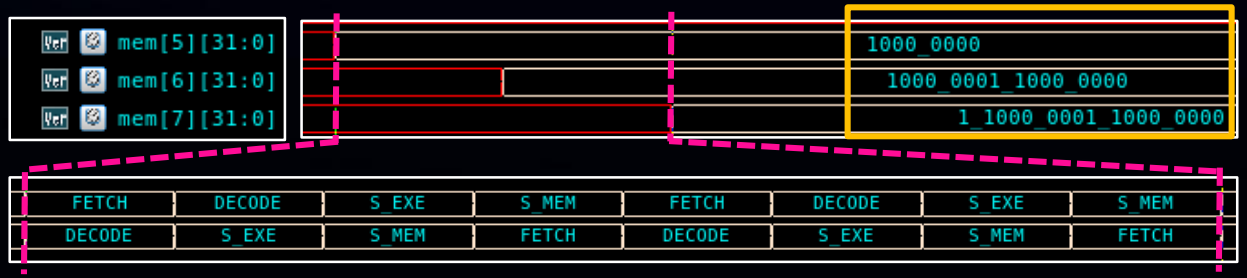
- mem[15] : sra(shift right arithmetic) 연산을 실행한 결과, msb인 1이 복제.

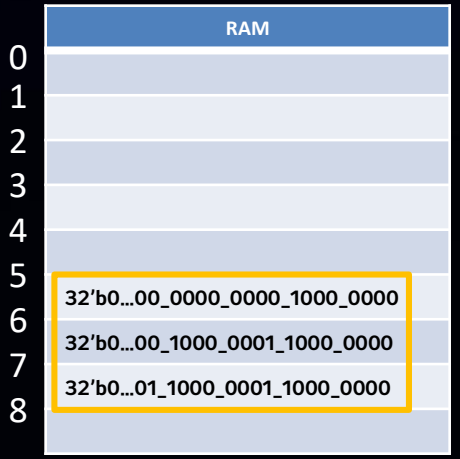
	RegFile
0	zero
1	11
2	12
3	13 -> 23
4	14 -> 1
5	15 -> 32'b11000...00
6	16 -> 0 -> 13
7	17 -> 0 -> 1
8	18 -> 1 -> 0
9	19 -> 0 -> 32'b0...00011
10	20 -> 32'b0..00111 -> 32'b0...01111
11	21 -> 32'b0..01111 -> 32'b0...01100
12	22 -> 32'b0..01000 -> 32'b0...0000
13	23 -> 32'b0..01_1000_0000
14	24 -> 32'b0..01_1000_0000_0000_0000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

```
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.

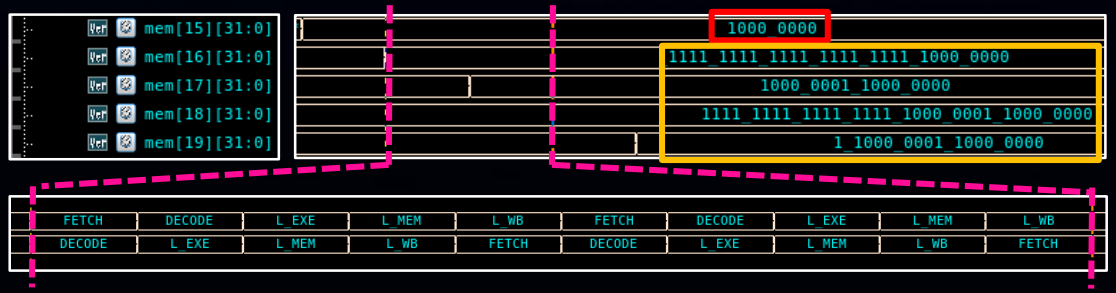
- mem[6] : sh(store half) 명령어를 실행하여 데이터의 16bit만 저장.

- 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.

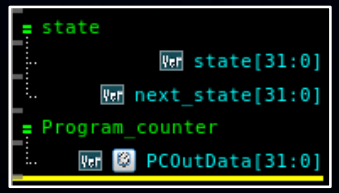
- mem[19] : lw(load woard) 명령어의 실행으로 32bit가 적재.

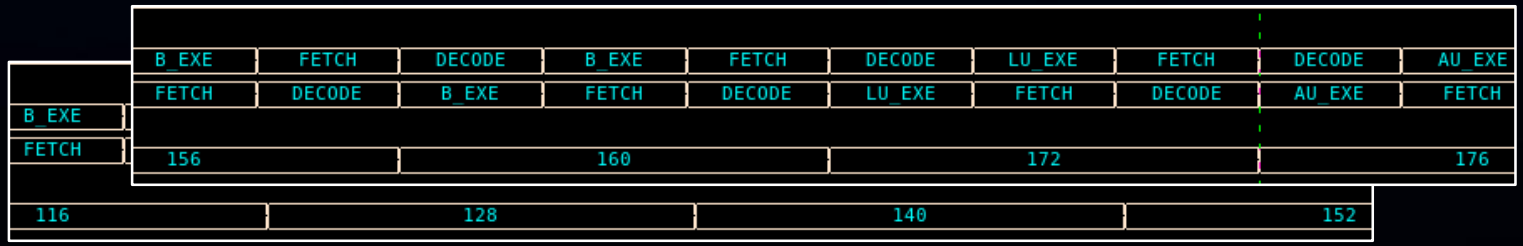
	RegFile
0	zero
1	11
2	12
3	13 -> 23
4	14 -> 1
5	15 -> 32'b11000...00
6	16 -> 0 -> 13
7	17 -> 0 -> 1
8	18 -> 1 -> 0
9	19 -> 0 -> 32'b0...00011
10	20 -> 32'b0..00111 -> 32'b0...01111
11	21 -> 32'b0..01111 - > 32'b0...01100
12	22 -> 32'b0..01000 -> 32'b0...0000
13	23 -> 32'b0..01_1000_0000
14	24 -> 32'b0..01_1000_0000_0000_0000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

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

4. Timing Diagram

- **B-Type**





- mem[29] : beq 명령어가 True로 실행되어, 다음 PC의 출력이 분기. → (다음 PC) = {(29*4) + 12} = 128

- mem[32] : bne 명령어가 True로 실행되어, 다음 PC의 출력이 분기. → (다음 PC) = {(32*4) + 12} = 140

- mem[35] : blt 명령어가 True로 실행되어, 다음 PC의 출력이 분기. → (다음 PC) = {(35*4) + 12} = 152

- mem[38] : bge 명령어가 False로 실행되어, 다음 PC의 출력이 유지. → (다음 PC) = {(38*4) + 4} = 156

- mem[39] : bltu 명령어가 False로 실행되어, 다음 PC의 출력이 유지. → (다음 PC) = {(39*4) + 4} = 160

-mem[40] : bgeu 명령어가 True로 실행되어, 다음 PC의 출력이 분기. → (다음 PC) = {(40*4) + 12} = 172

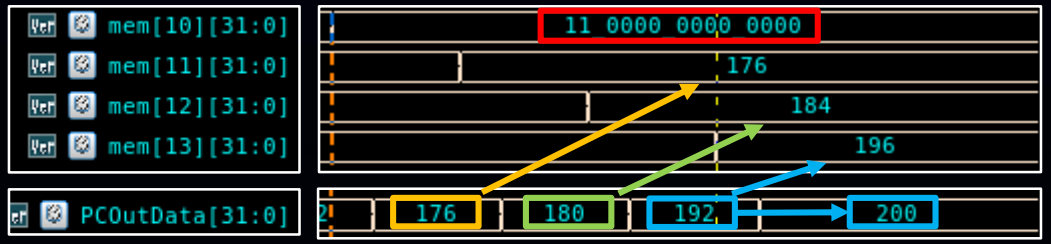
```
lui   x10, 3
auipc x11, 0
```

```
jal   x12, 12
```

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

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.

	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 -> 32'b0...011_0000_0000_0000
11	21 -> 32'b0..01111 -> 32'b0...01100 -> 176
12	22 -> 32'b0..01000 -> 32'b0...000 -> 184
13	23 -> 32'b0..01_1000_0000 -> 196
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

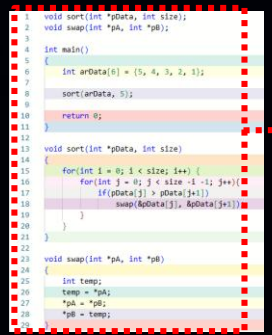
5. Test Program 설명 및 분석

COMPILER EXPLORER

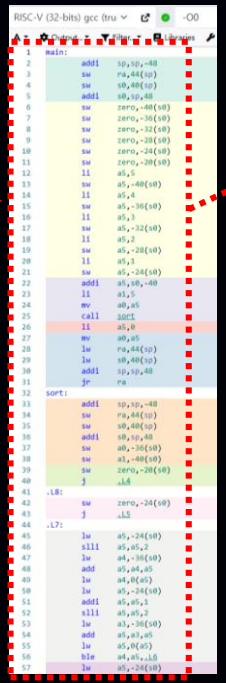
RISC-V Online Assembler

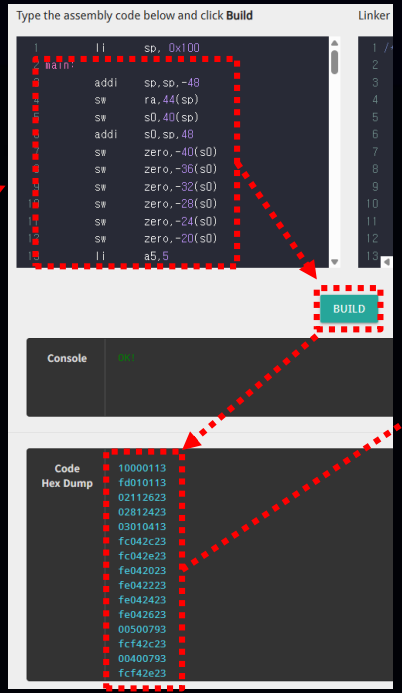
Visual Studio Code

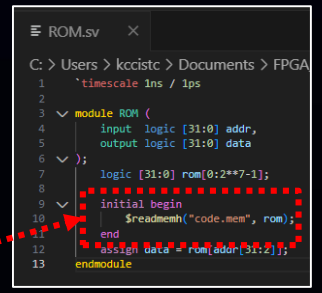
Linux

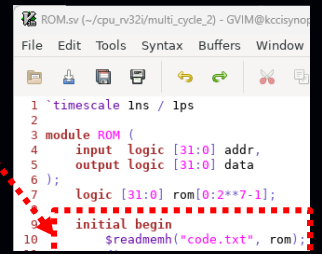


sort.c









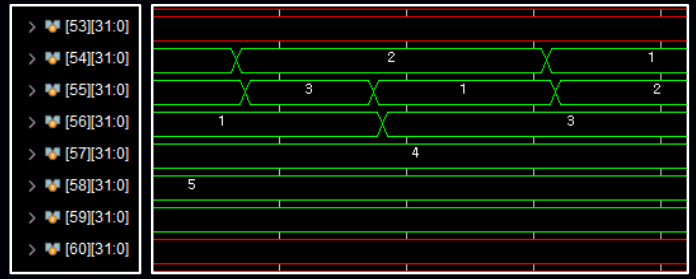
5. Test Program 설명 및 분석

tb_rv32i.sv

```
47      always #5 clk = ~clk;
48
49      localparam RAM_max_addr = 5;
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'd0;
60          end
61
62          clk = 0; reset = 1;
63          #10;
64          reset = 0;
65
66          #1000000;
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
89                  result_addr_array[0], result_addr_array[4], du
90          end
91
92          $finish;
93      end
```

```
Tcl Console | Messages | Log

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.glbl
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
Sort Success! : RAM[      54:      58] = [       1,       2,       3,       4,       5]
```

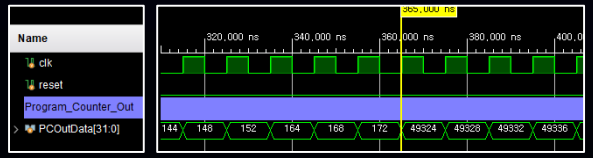


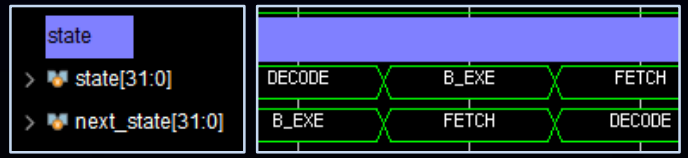
- 배열 [5, 4, 3, 2, 1, 0]에 대하여 0을 제외하고, 데이터를 오름차순으로 분류(sorting)된 데이터가 RAM에 저장(store).

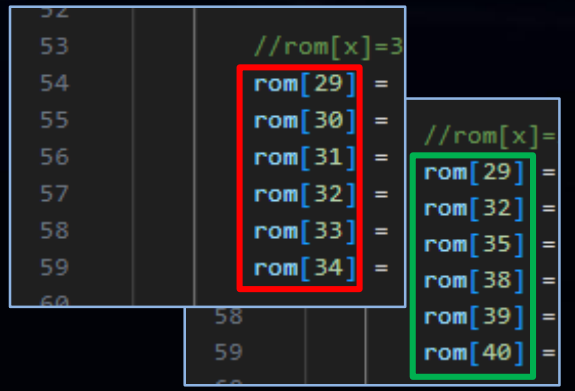
- Testbench 코드에 검증을 위한 코드를 추가하여, 어떤 주소에 데이터가 정상적으로 배열된 것을 확인.

6. Trouble Shooting

1. Program Counter Issue







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.

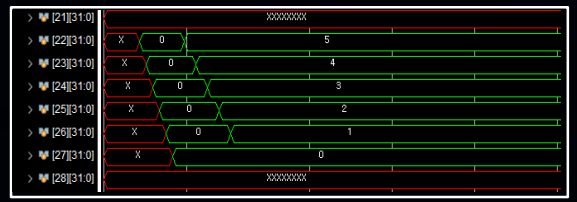
문제 해결

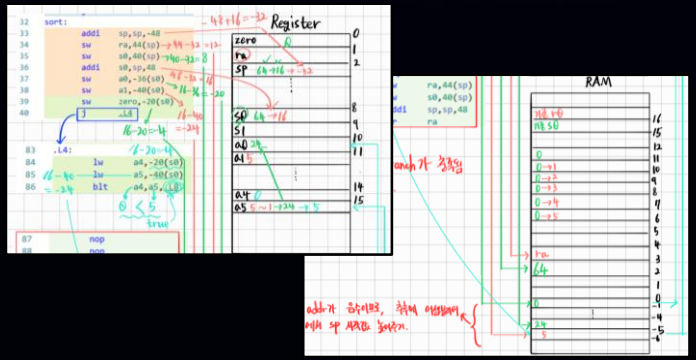
- 분기(Branch)가 발생할 경우, Program Counter의 값이 바뀌고, ROM의 Address도 이에 맞게 변경시켜서 문제를 해결.

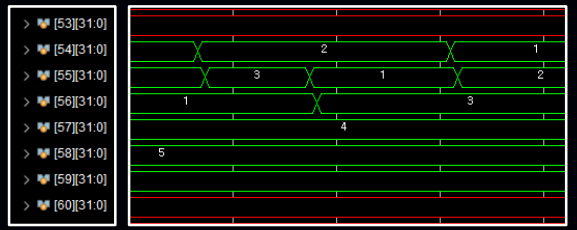
- CPU의 구조와 RV32I의 명령어를 완벽히 이해하지 못해서 발생한 Trouble Shooting.

6. 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