RISC-V (RV32I)

CPU 설계 발표

AI 시스템 반도체 설계 2기 – 이현수

목차

01 프로젝트 개요 PROJECT

- **주제**

RISC-V RV32I 기반 Multi-Cycle CPU 설계 및 검증

- **목표**

- RV32I 핵심 명령군(R/I/S/B/U/J) 기능 정확성 확보
- Load/Store 의 byte/half/word 및 sign/zero 확장 정확성
- C 코드(sort) 엔드-투-엔드 실행으로 서브루틴 호출/스택 동작 확인

02 개발 환경

Language

ASM

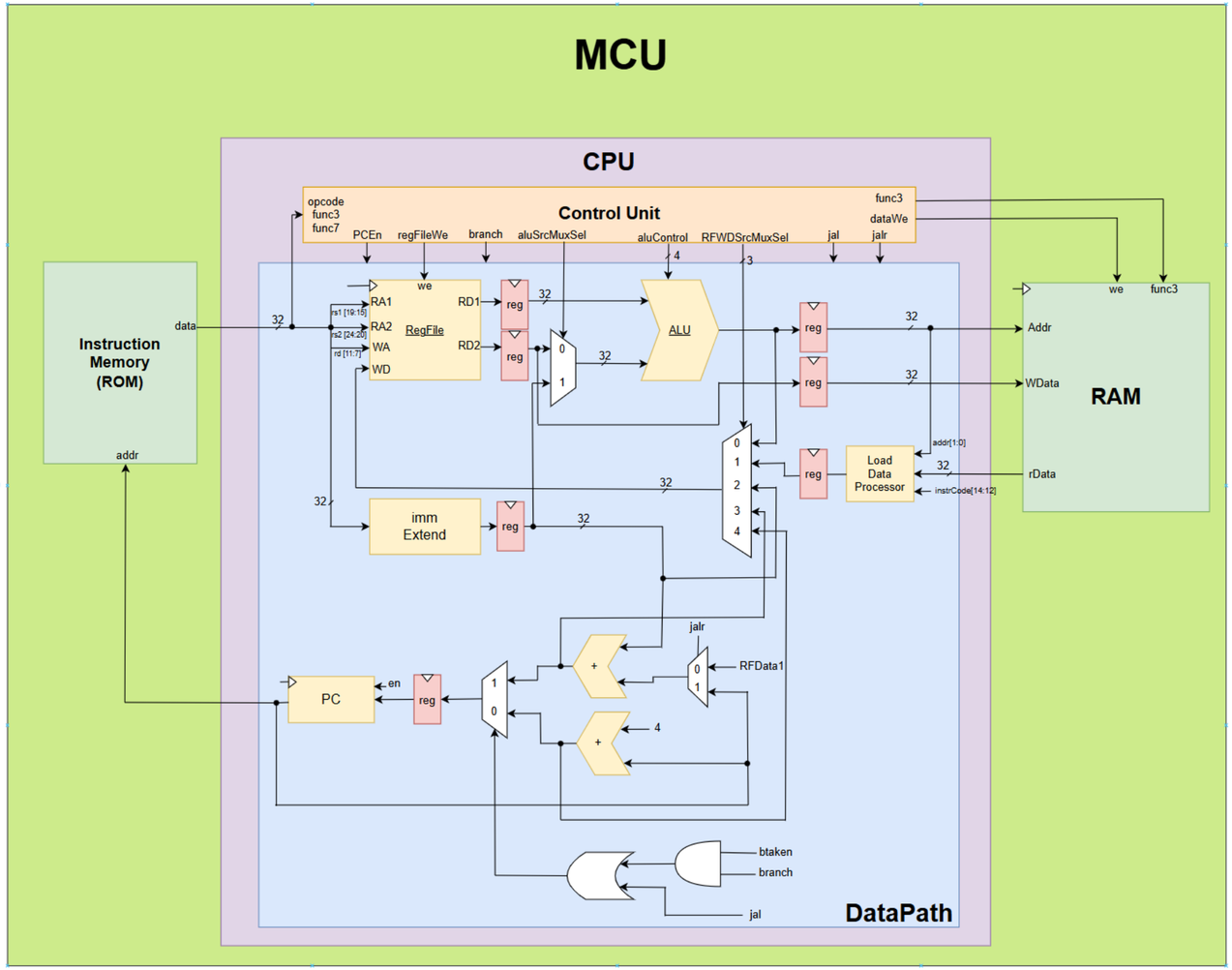
Visual Studio Code

Simulation

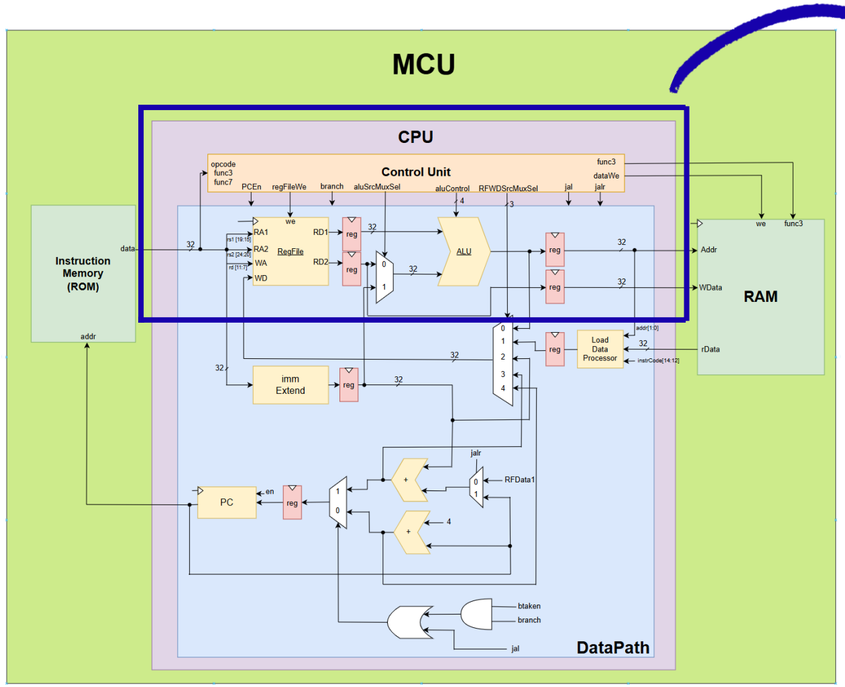
MobaXterm

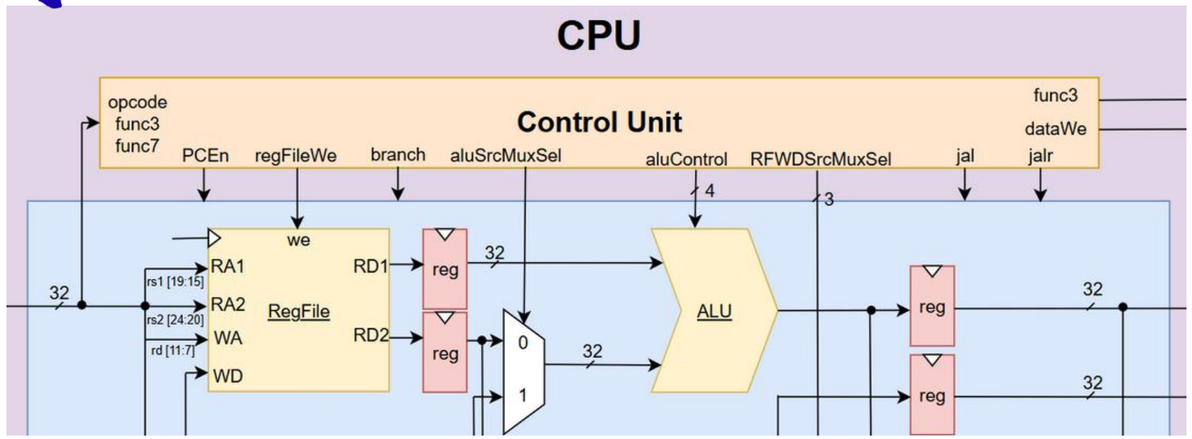
SYNOPSYS®

03 Block Diagram

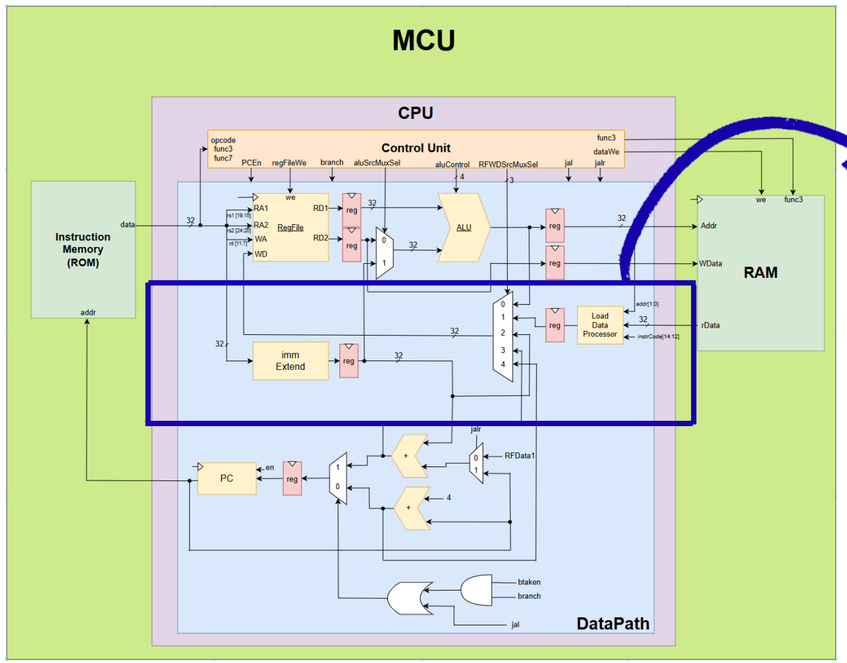


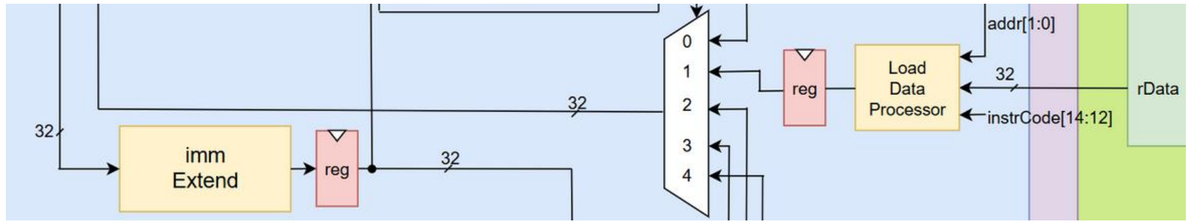
03 Block Diagram



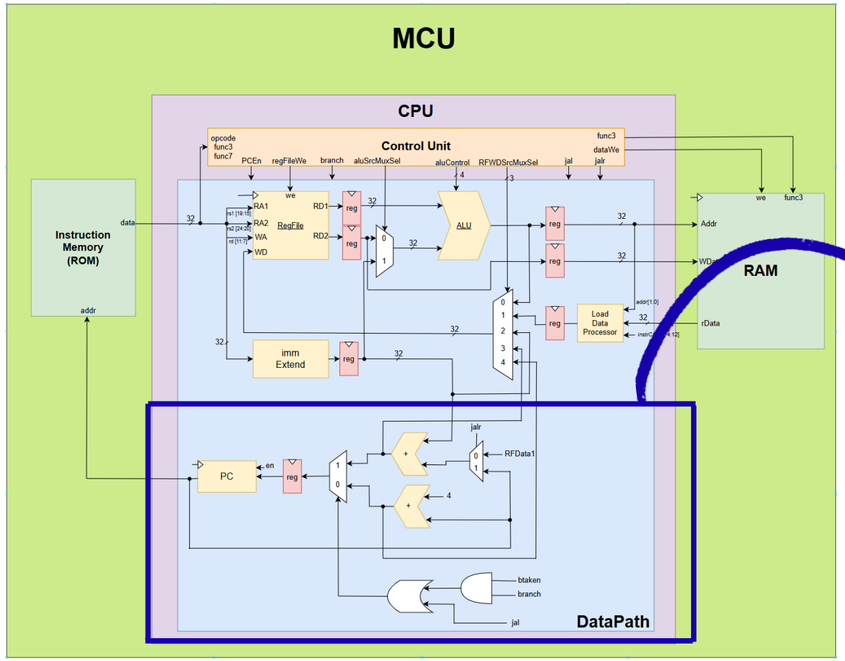


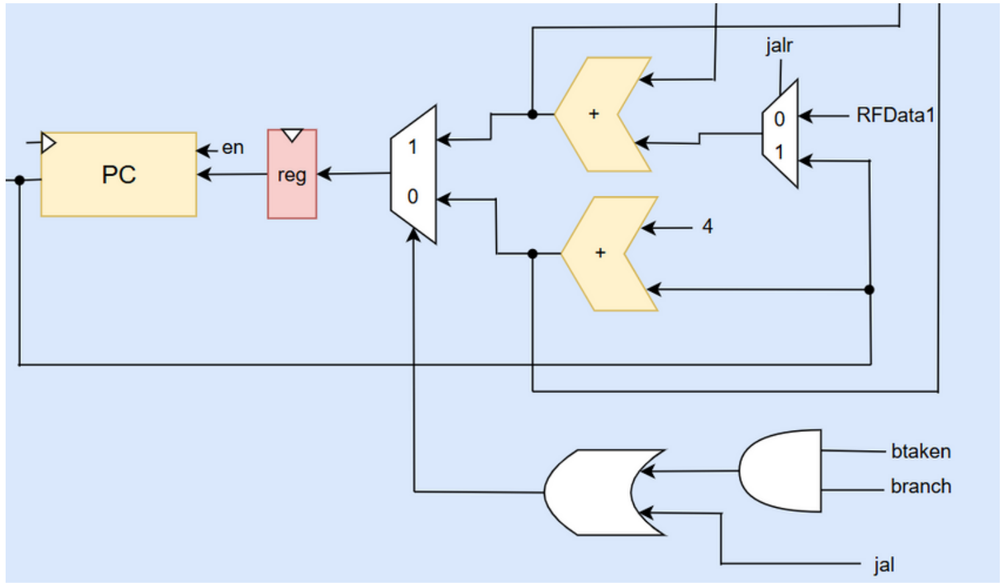
03 Block Diagram



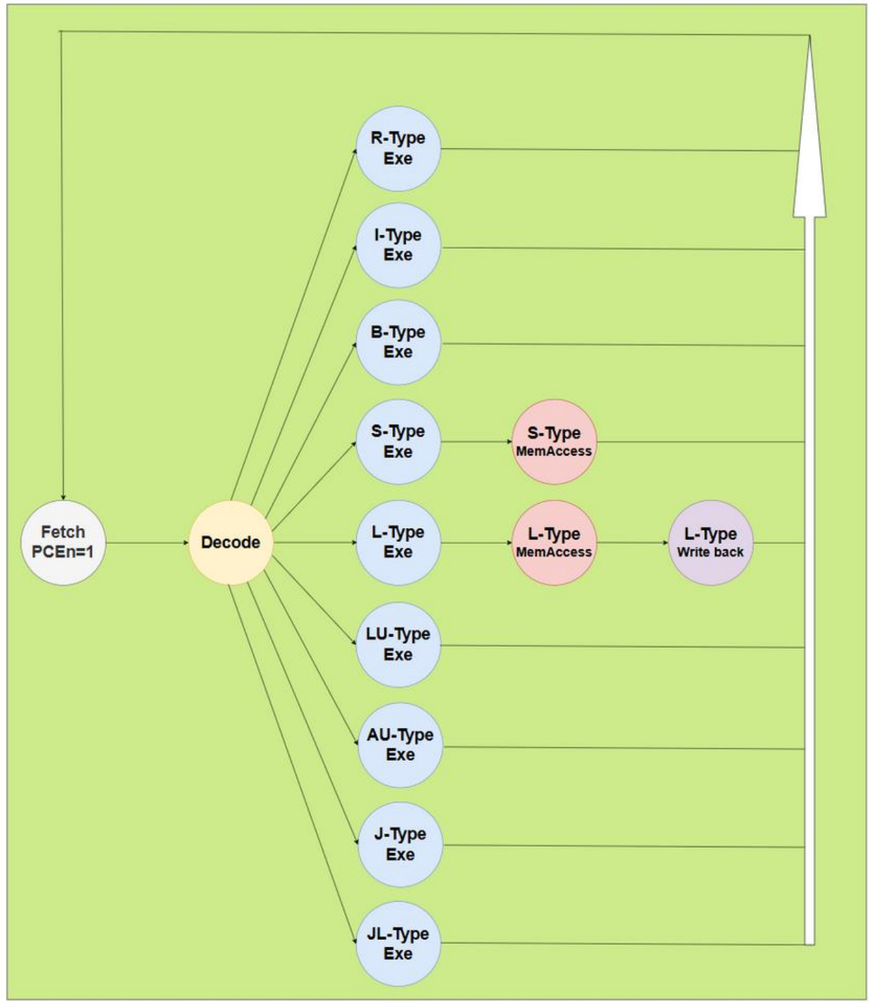


03 Block Diagram





03 Block Diagram FSM



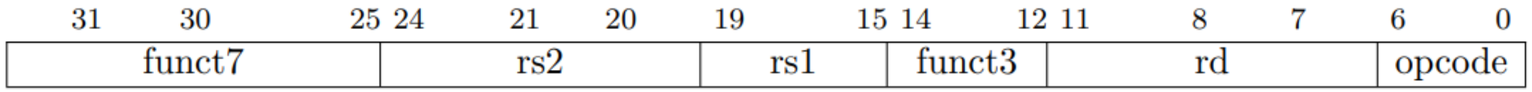
I-TYPE

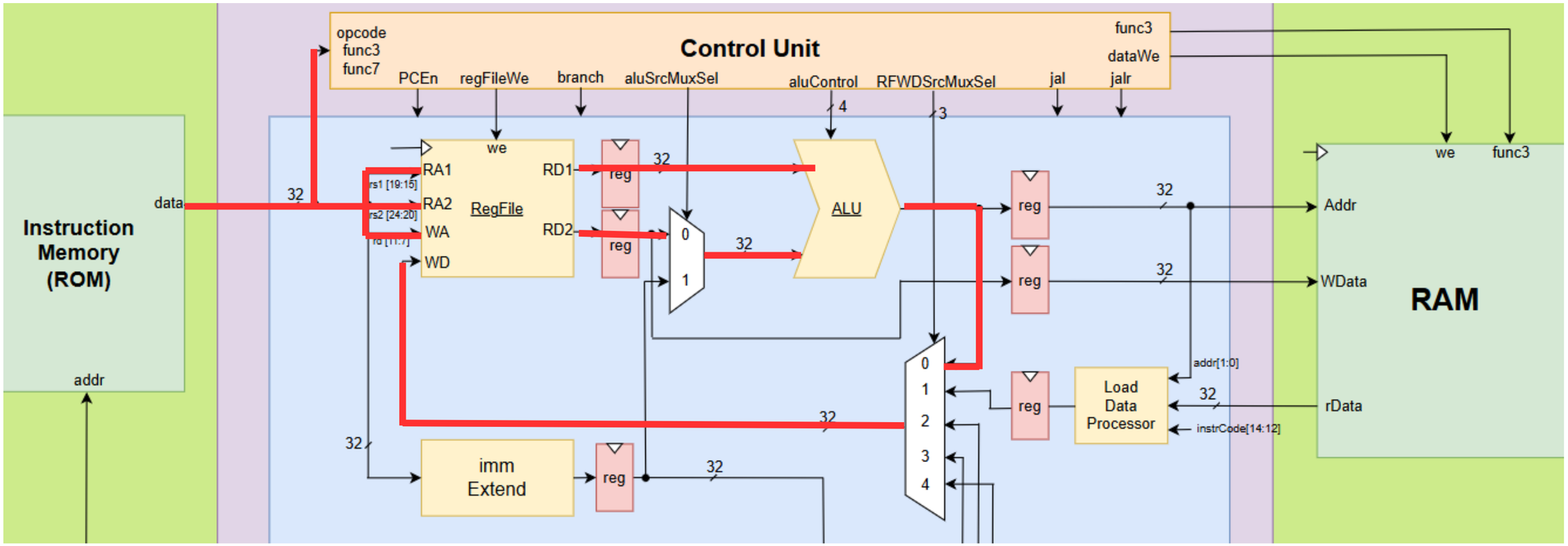
S-TYPE

L-TYPE

04 Instruction Set Type

R - TYPE

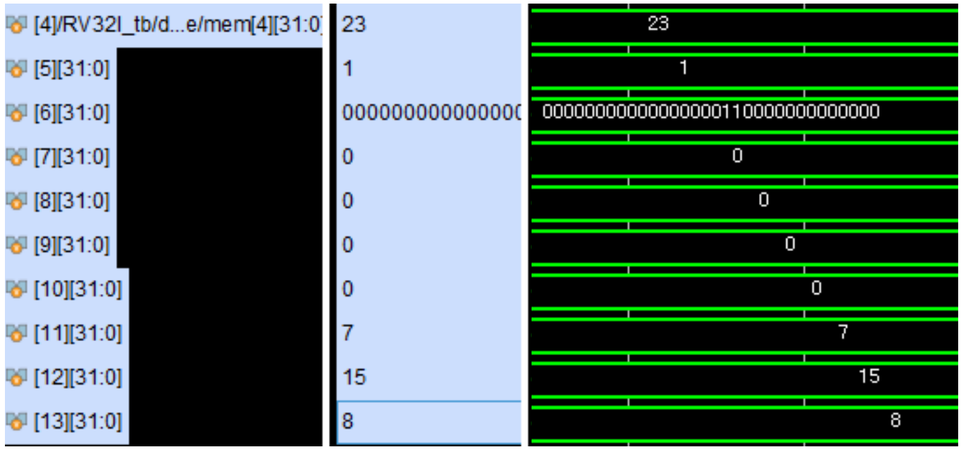




04 Instruction Set Type

R - TYPE

```
// R-Type  funct7(7) | rs2(5) | rs1(5) | funct3(3) | rd(5) | opcode(7)
rom[0] = 32'b0000000_00001_00010_000_00100_0110011;  // add  x4,  x2, x1
rom[1] = 32'b0100000_00001_00010_000_00101_0110011;  // sub  x5,  x2, x1
rom[2] = 32'b0000000_00001_00010_001_00110_0110011;  // sll  x6,  x2, x1
rom[3] = 32'b0000000_00001_00010_101_00111_0110011;  // srl  x7,  x2, x1
rom[4] = 32'b0100000_00001_00010_101_01000_0110011;  // sra  x8,  x2, x1
rom[5] = 32'b0000000_00001_00010_010_01001_0110011;  // slt  x9,  x2, x1
rom[6] = 32'b0000000_00001_00010_011_01010_0110011;  // sltu x10, x2, x1
rom[7] = 32'b0000000_00001_00010_100_01011_0110011;  // xor  x11, x2, x1
rom[8] = 32'b0000000_00001_00010_110_01100_0110011;  // or   x12, x2, x1
rom[9] = 32'b0000000_00001_00010_111_01101_0110011;  // and  x13, x2, x1
```



ADD : 12 + 11 = 23

SUB : 12 - 11 = 1

SLL: 12 << 11 = ~110_000_000_000_000

SRL : 12 >> 11 = 0

SRA : 12 >>> 11 = 0

SLT : (12 < 11) ? 1 : 0

SLTU: (12 < 11) ? 1 : 0

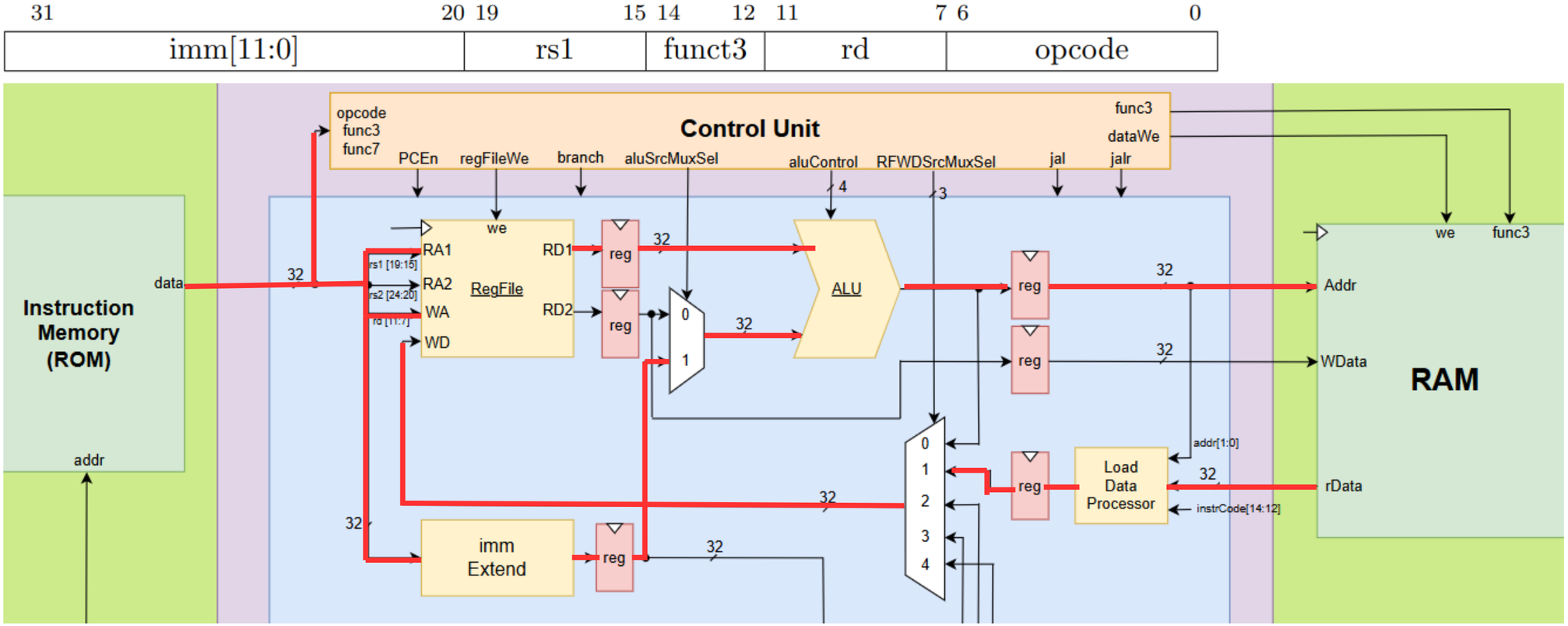
XOR : 1100 ^ 1011 = 0111 (7 in dec)

OR : 1100 | 1011 = 1111 (15 in dec)

AND : 1100 & 1011 = 1000 (8 in dec)

04 Instruction Set Type

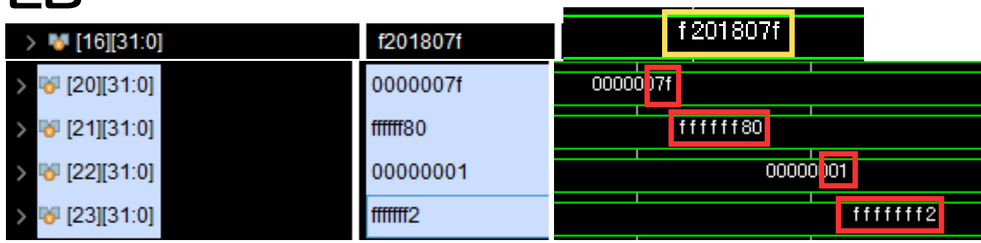
L - TYPE



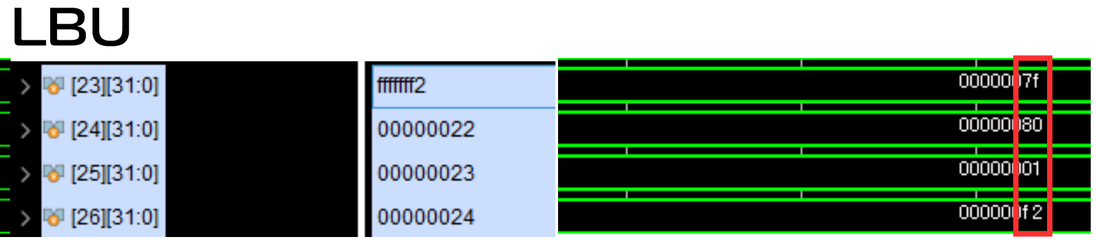
04 Instruction Set Type

L - TYPE

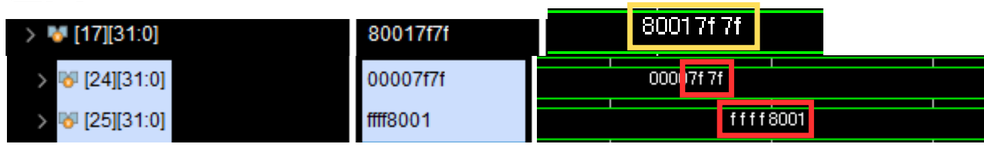
LB



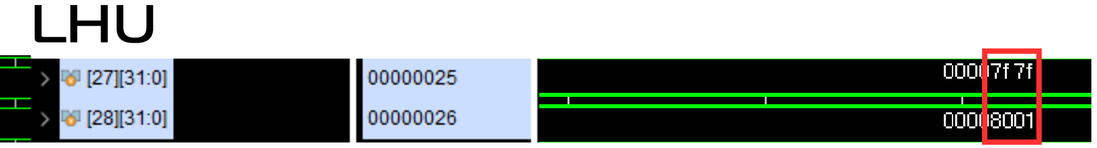
LBU



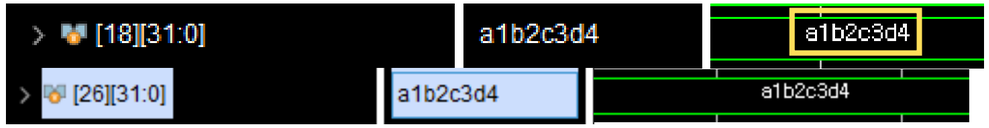
LH



LHU

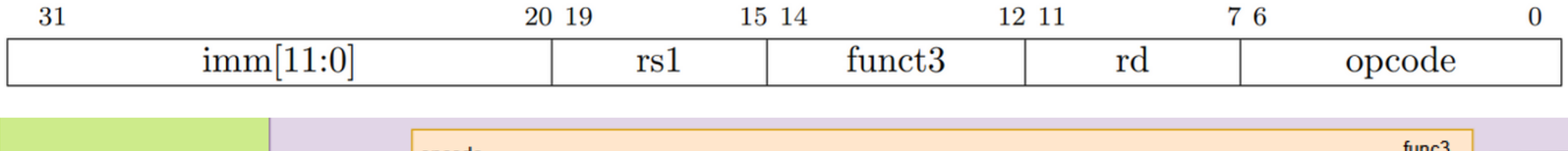


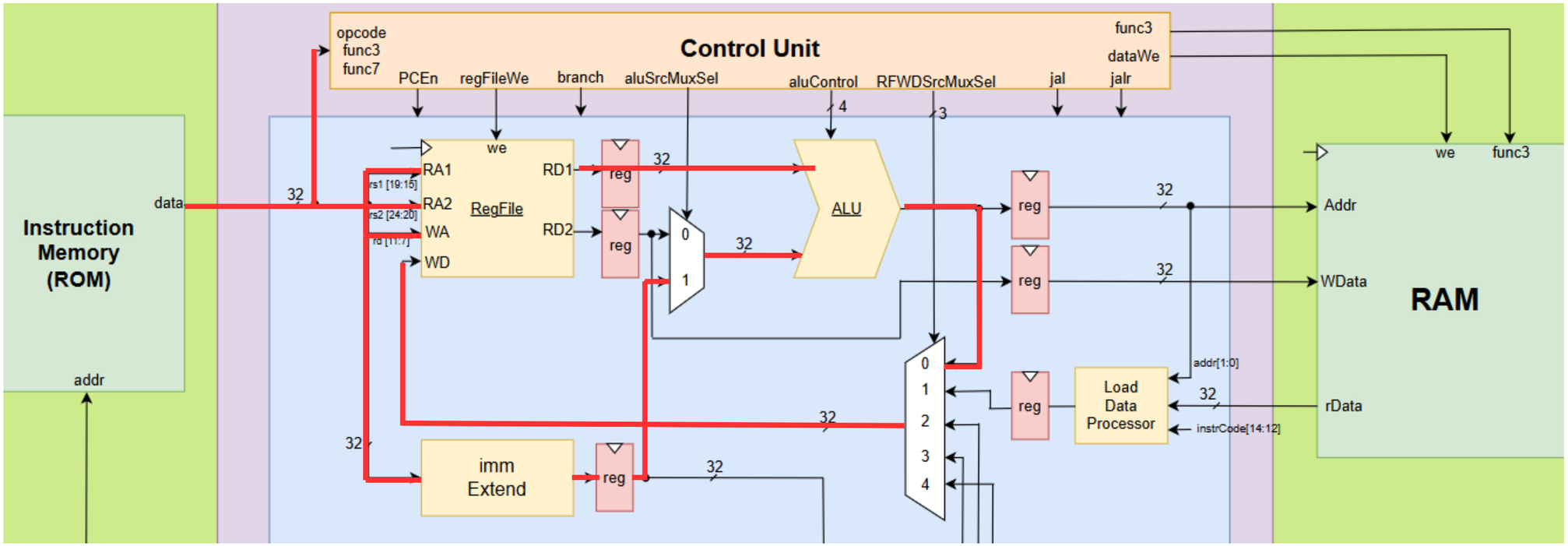
LW



04 Instruction Set Type

I - TYPE

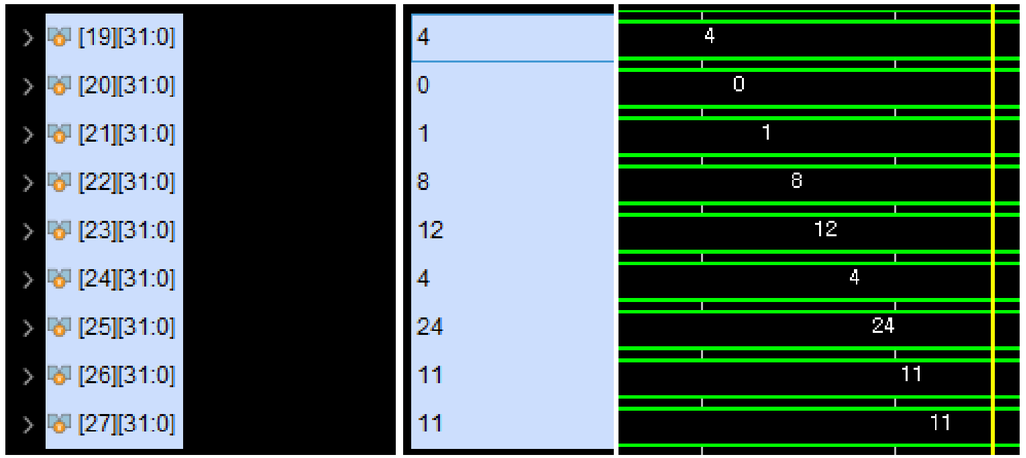




04 Instruction Set Type

I - TYPE

```
// I-Type  imm[11:0](12) | rs1(5) | funct3(3) | rd(5) | opcode(7)
rom[18] = 32'b111111111000_00010_000_10011_0010011;  // addi  x19, x2, -8
rom[19] = 32'b111111111100_00010_010_10100_0010011;  // slti  x20, x2, -4
rom[20] = 32'b111111111100_00010_011_10101_0010011;  // sltiu x21, x2, -4
rom[21] = 32'b000000000100_00010_100_10110_0010011;  // xori  x22, x2,  4
rom[22] = 32'b000000000100_00010_110_10111_0010011;  // ori   x23, x2,  4
rom[23] = 32'b000000000100_00010_111_11000_0010011;  // andi  x24, x2,  4
rom[24] = 32'b0000000_00001_00010_001_11001_0010011; // slli  x25, x2,  1
rom[25] = 32'b0000000_00001_00100_101_11010_0010011; // srli  x26, x4,  1
rom[26] = 32'b0100000_00001_00100_101_11011_0010011; // srai  x27, x4,  1
```



ADDi : 12 - 8 = 4

SLTi : (12 < -4) ? 1 : 0

SLTiu : (12 < 4,294,967,292) ? 1 : 0

XORi : 1100 ^ 0100 = 1000 (8 in dec)

ORi : 1100 | 0100 = 1100 (12 in dec)

ANDi : 1100 & 0100 = 0100 (4 in dec)

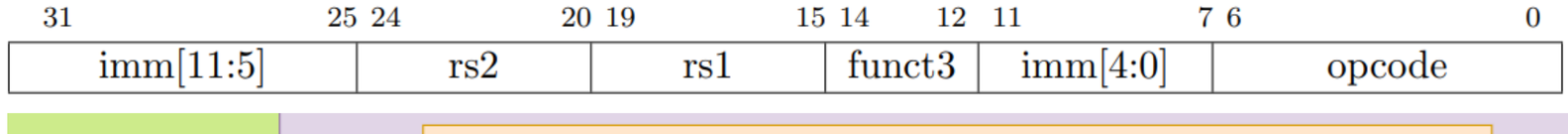
SLLi : 1100 << 1 = 11000 (24 in dec)

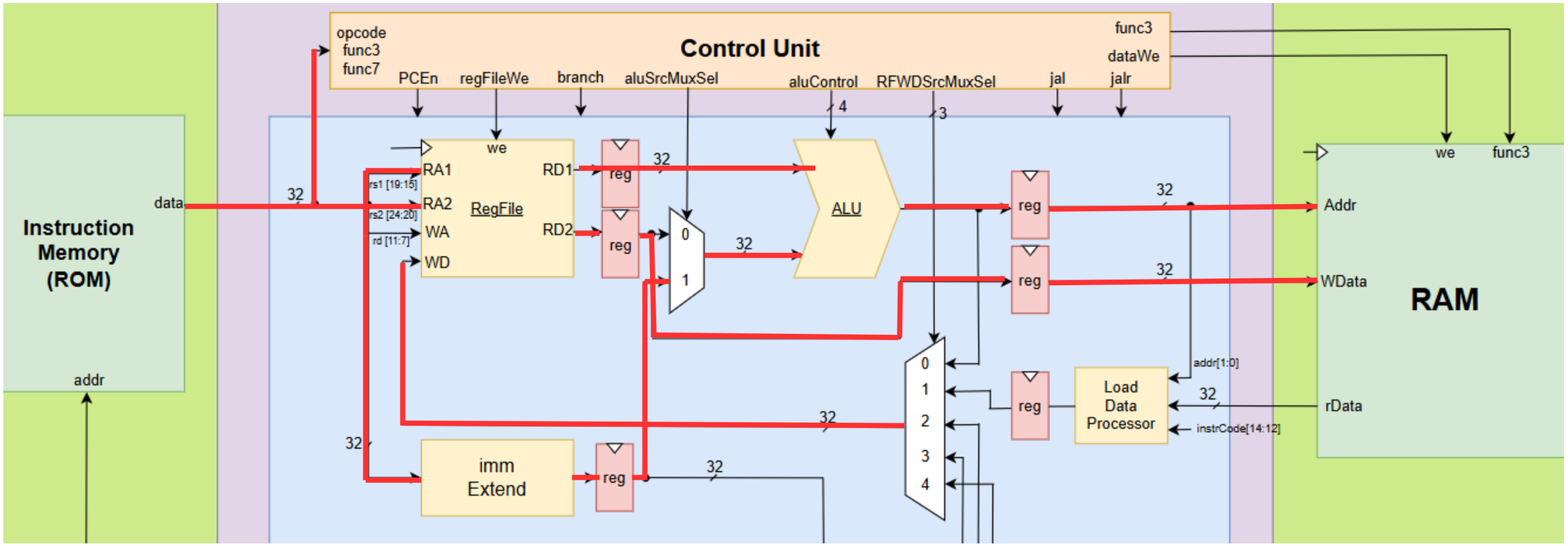
SRLi : 10111 >> 1 = 01011 (11 in dec)

SRAi : 10111 >>> 1 = 01011 (11 in dec)

04 Instruction Set Type

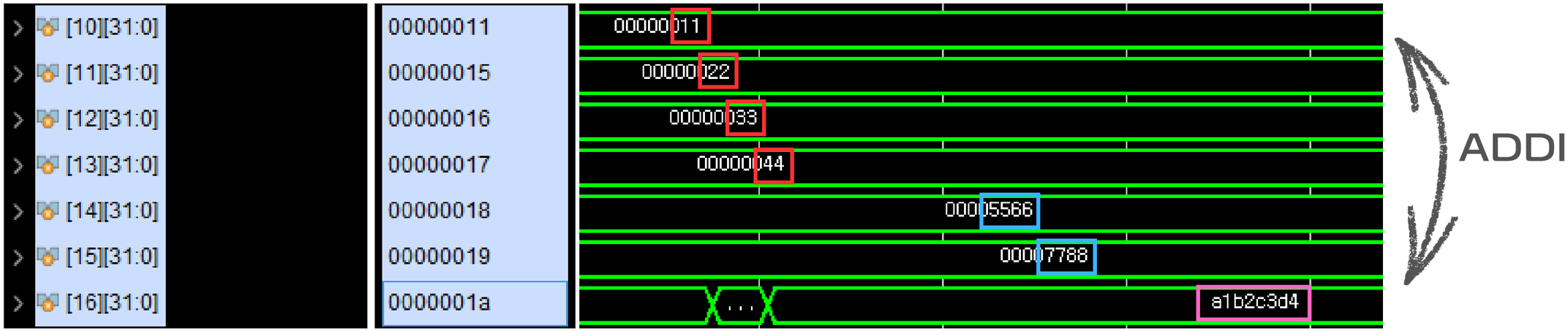
S - TYPE

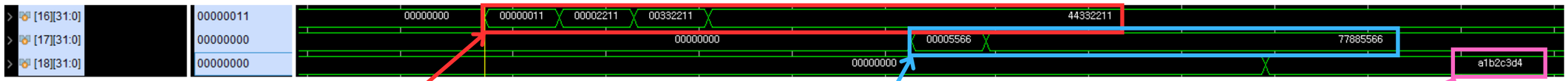




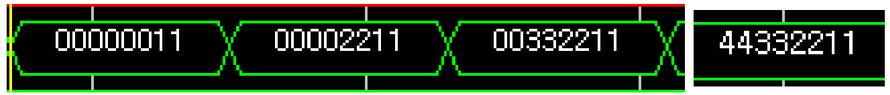
04 Instruction Set Type

S - TYPE





SB

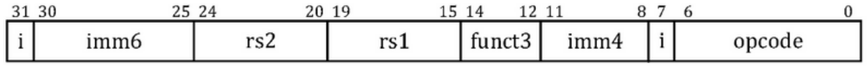


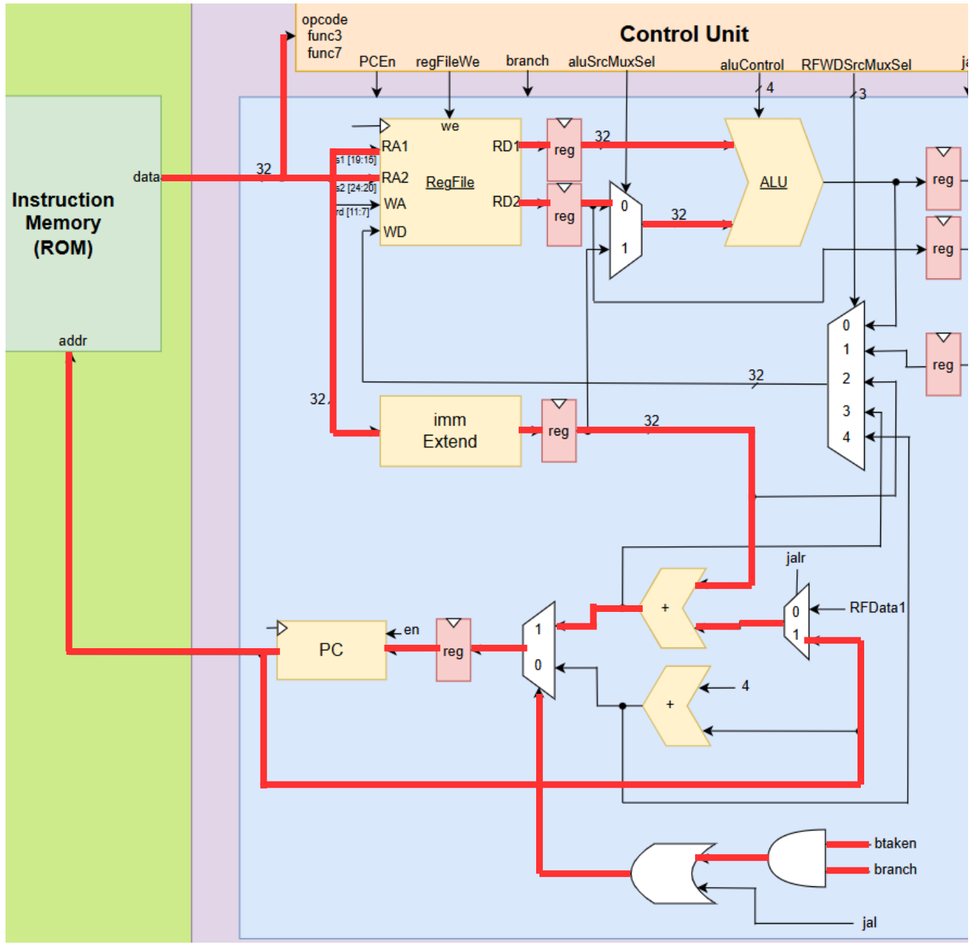
SH

SW

04 Instruction Set Type

B - TYPE

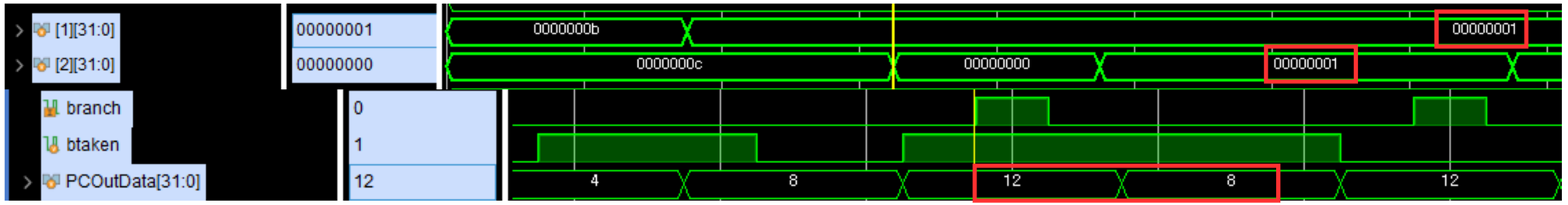




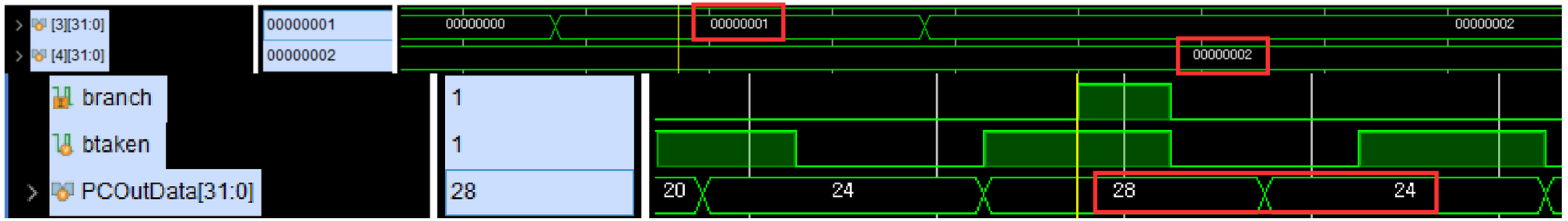
04 Instruction Set Type

B - TYPE

BEQ (1 == 1 이면 분기)



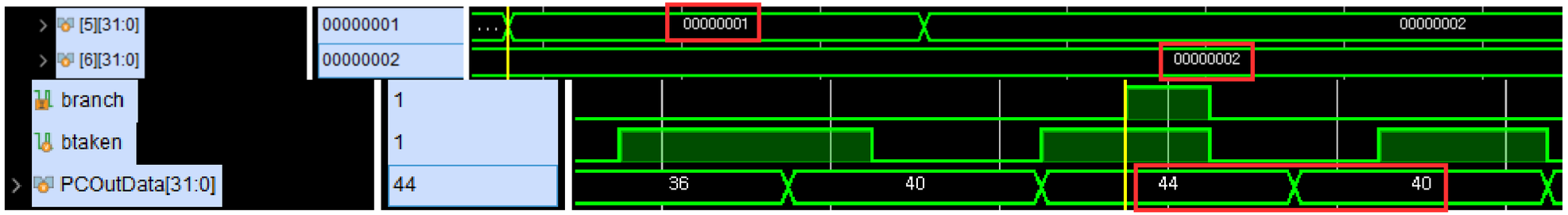
BNE (1 != 2 이면 분기)



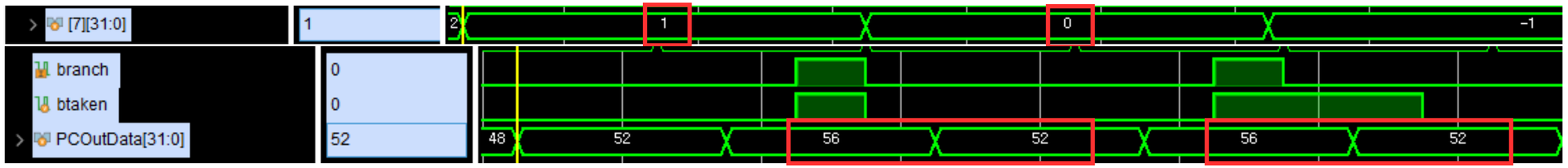
04 Instruction Set Type

B - TYPE

BLT (1 < 2 이면 분기)



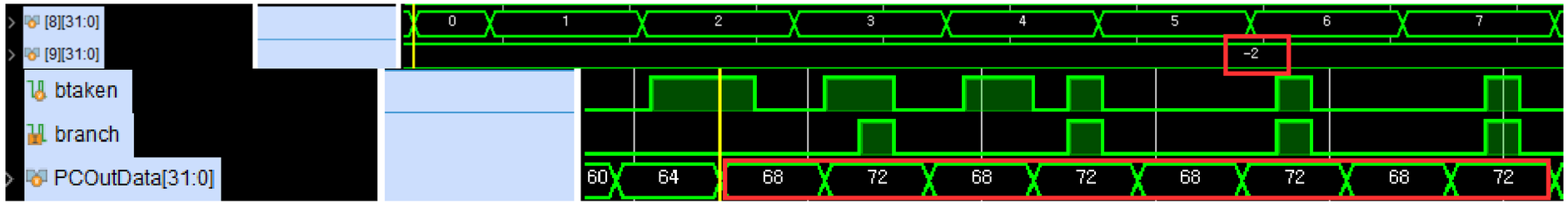
BGE (0보다 크거나 같으면 분기)



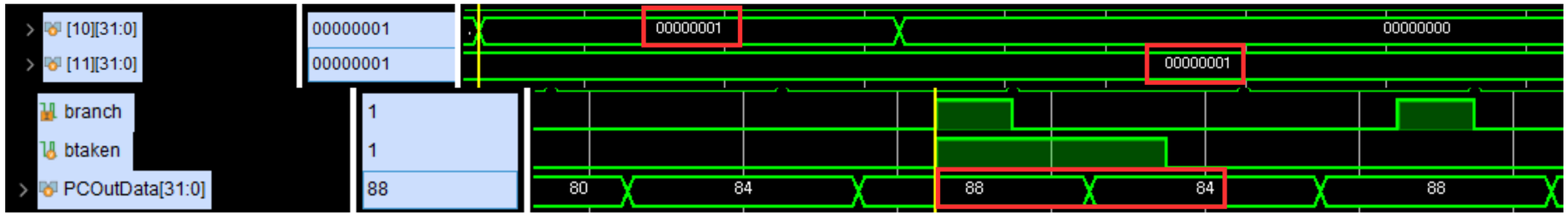
04 Instruction Set Type

B - TYPE

BLTU (0 < -2 이면 분기)

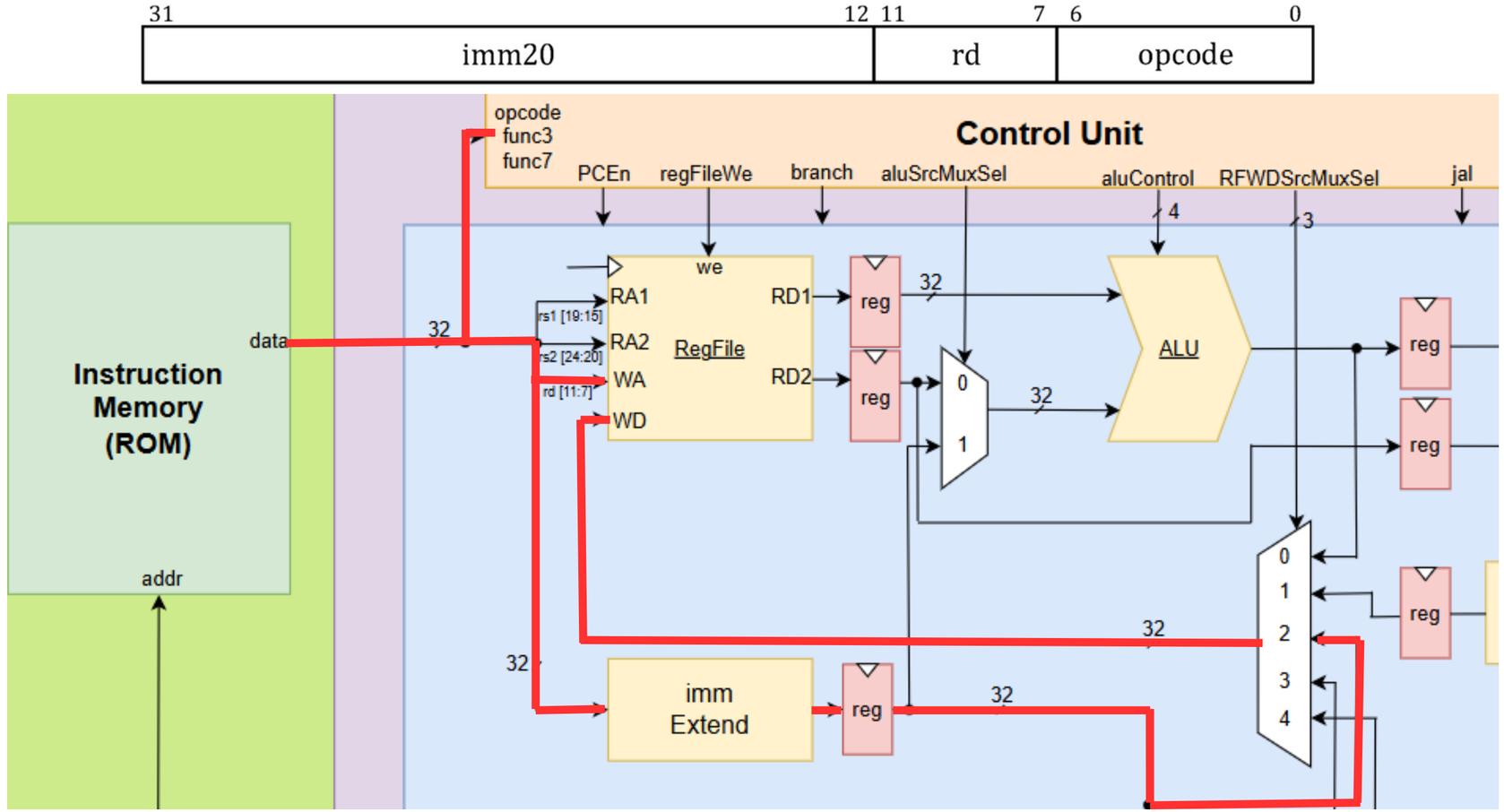


BGEU (1 >= 1 이면 분기)



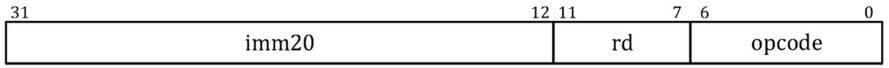
04 Instruction Set Type

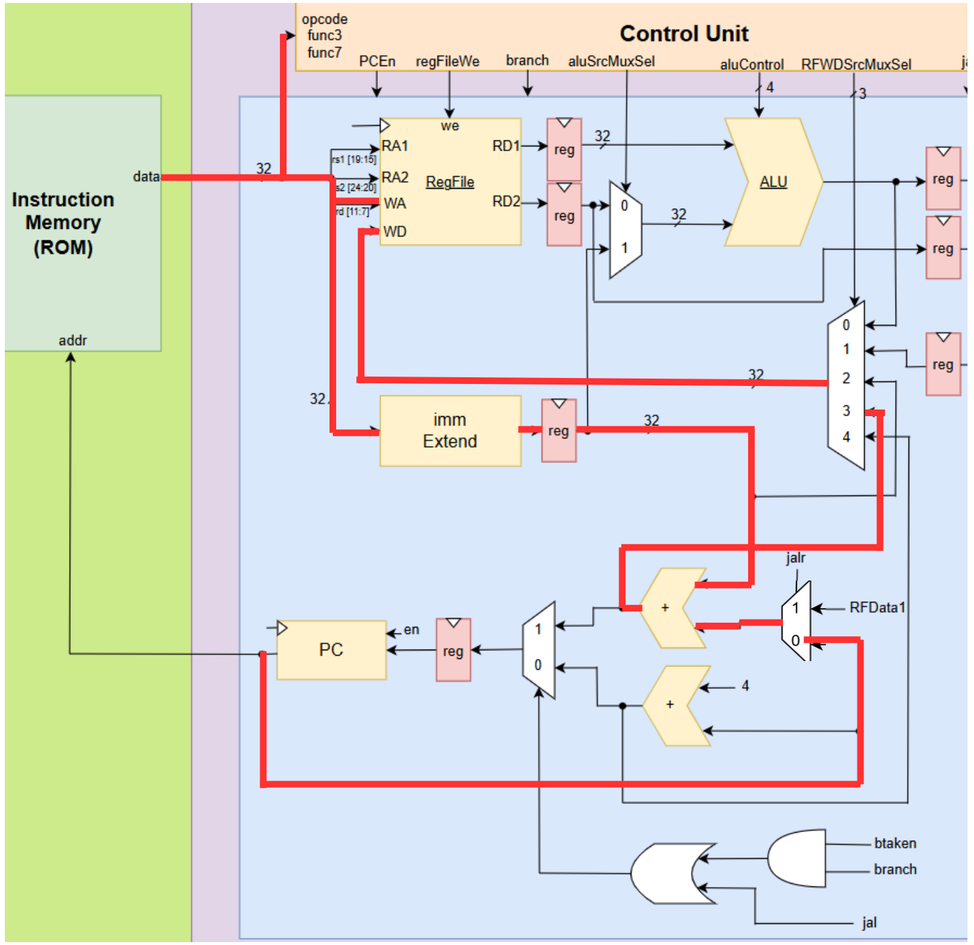
LU - TYPE



04 Instruction Set Type

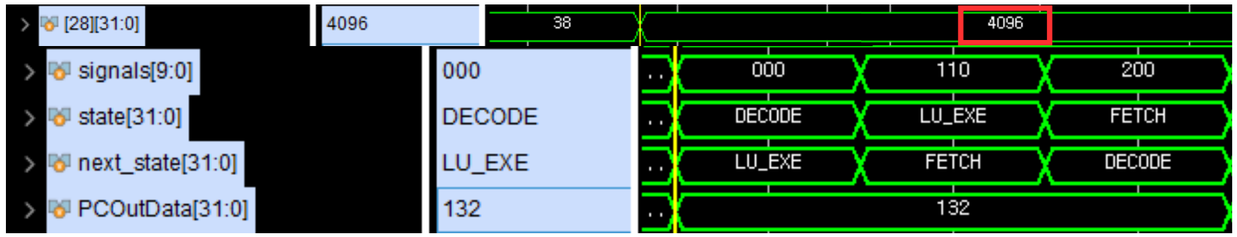
AU - TYPE





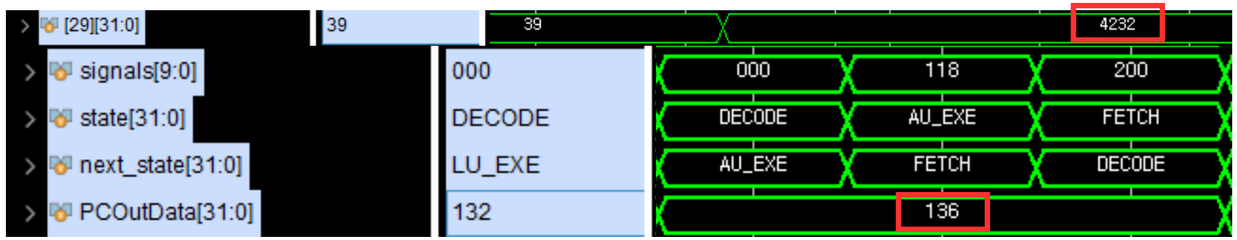
04 Instruction Set Type

LU - TYPE



→ LUI x28 , 1 → 2 ^ 12 = 4096

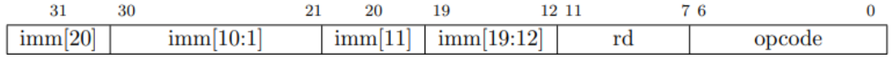
AU - TYPE

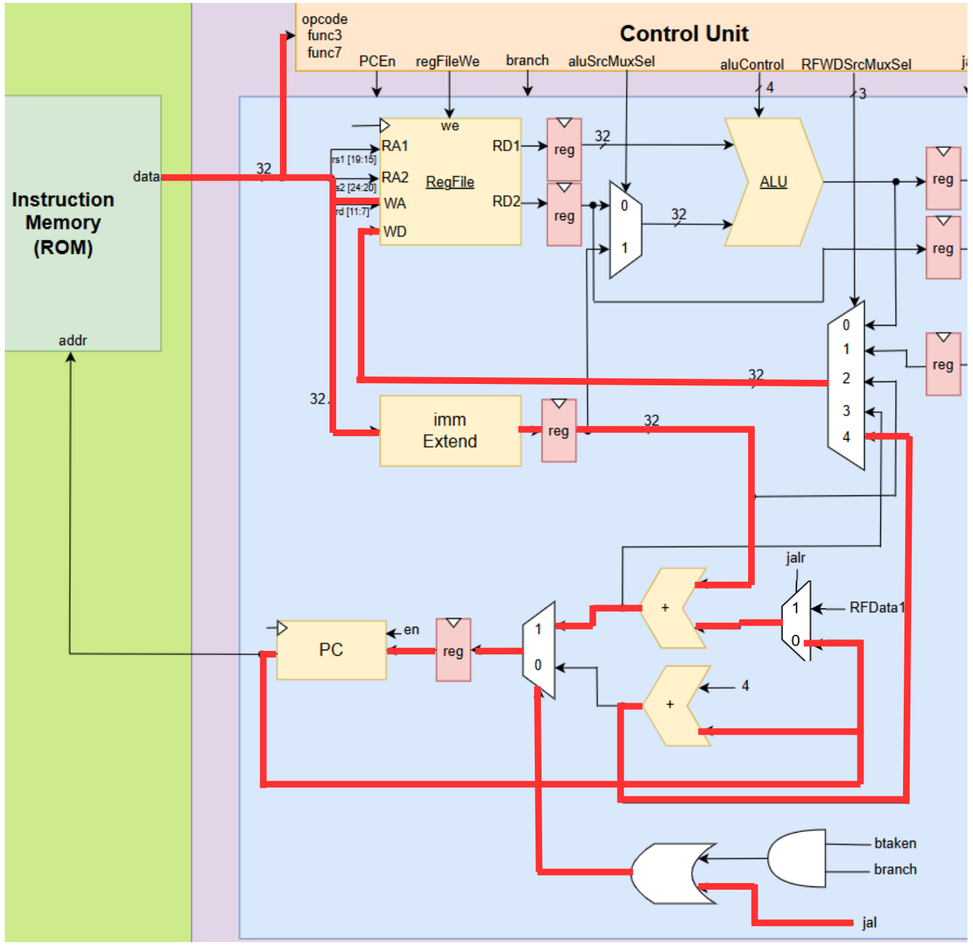


→ AUIPC x29 , 1 → (2 ^ 12) + 136 = 4232

04 Instruction Set Type

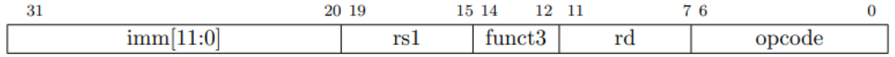
J - TYPE

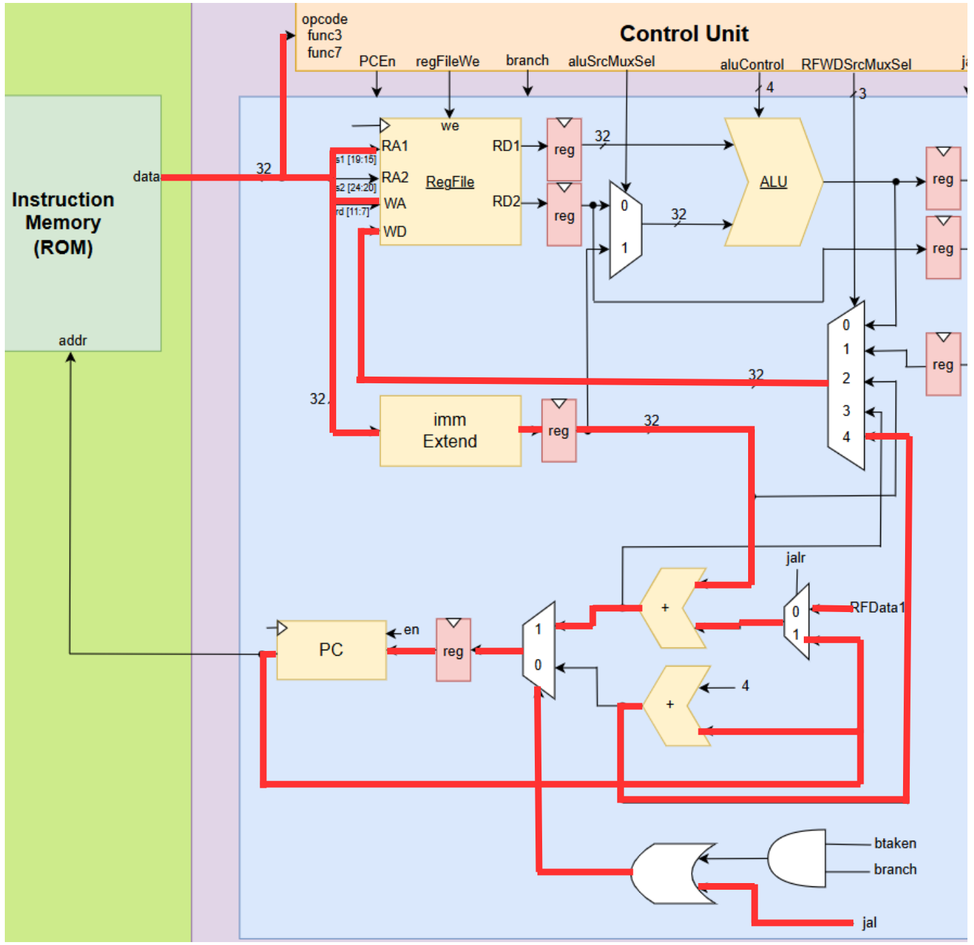




04 Instruction Set Type

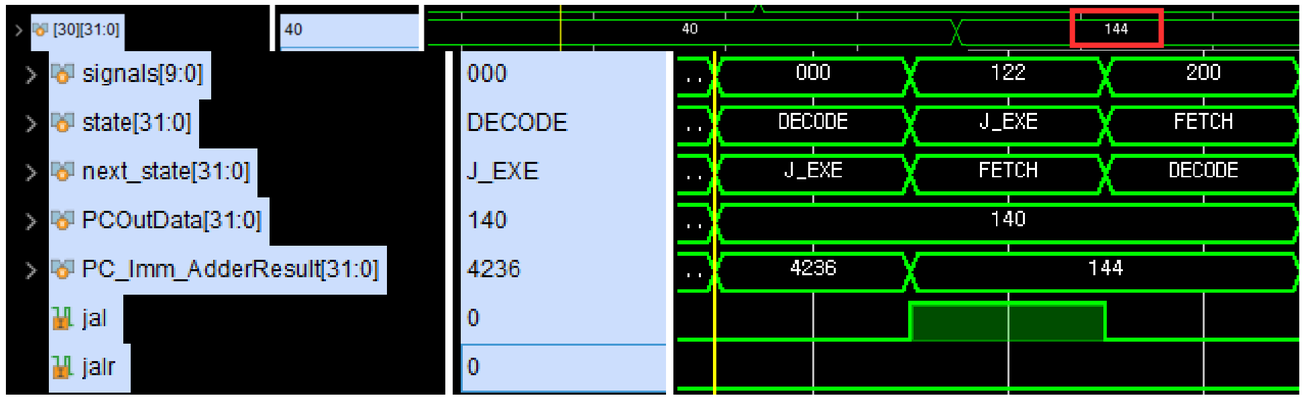
JL - TYPE





04 Instruction Set Type

J - TYPE

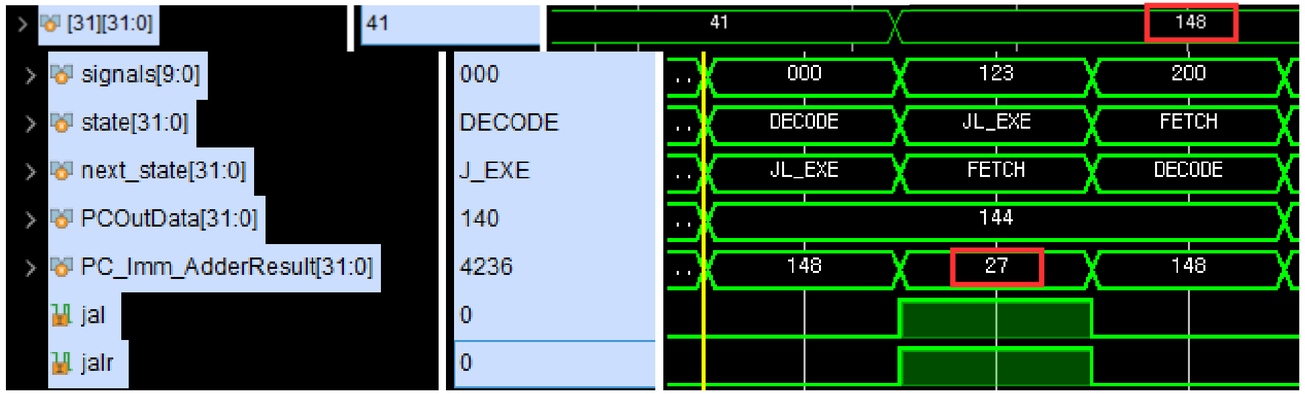


JAL x30, 4 :

Rd → 140 + 4 = 144

PC → 140 + 4 = 144

JL - TYPE



JALR x31, x4, 4 :

Rd → 144 + 4 = 148

PC → 23 + 4 = 27

05 Test Program

SORTING

C code

```c
int adder (int a, int b);
void sort(int *pData, int size);
void swap(int *pA, int *pB);

int main() {
    int arData[6] = { 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=0;j<size-i-1;j++){
            if(pData[j] > pData[j+1])
                swap(&pData[j], &pData[j+1]);
        }
    }
}

void swap(int *pA, int *pB){
    int temp;
    temp = *pA;
    *pA = *pB;
    *pB = temp;
}
```

ASM

```
sort:
        addi    sp,sp,-48
        sw      ra,44(sp)
        sw      s0,40(sp)
        addi    s0,sp,48
        sw      a0,-36(s0)
        sw      a1,-40(s0)
        sw      zero,-20(s0)
        j       .L4
.L8:
        sw      zero,-24(s0)
        j       .L5
.L7:
        lw      a5,-24(s0)
        slli    a5,a5,2
        lw      a4,-36(s0)
        add     a5,a4,a5
        lw      a4,0(a5)
        lw      a5,-24(s0)
        addi    a5,a5,1
        slli    a5,a5,2
        lw      a3,-36(s0)
        add     a5,a3,a5
        lw      a5,0(a5)
        ble     a4,a5,.L6
        lw      a5,-24(s0)
        slli    a5,a5,2
```

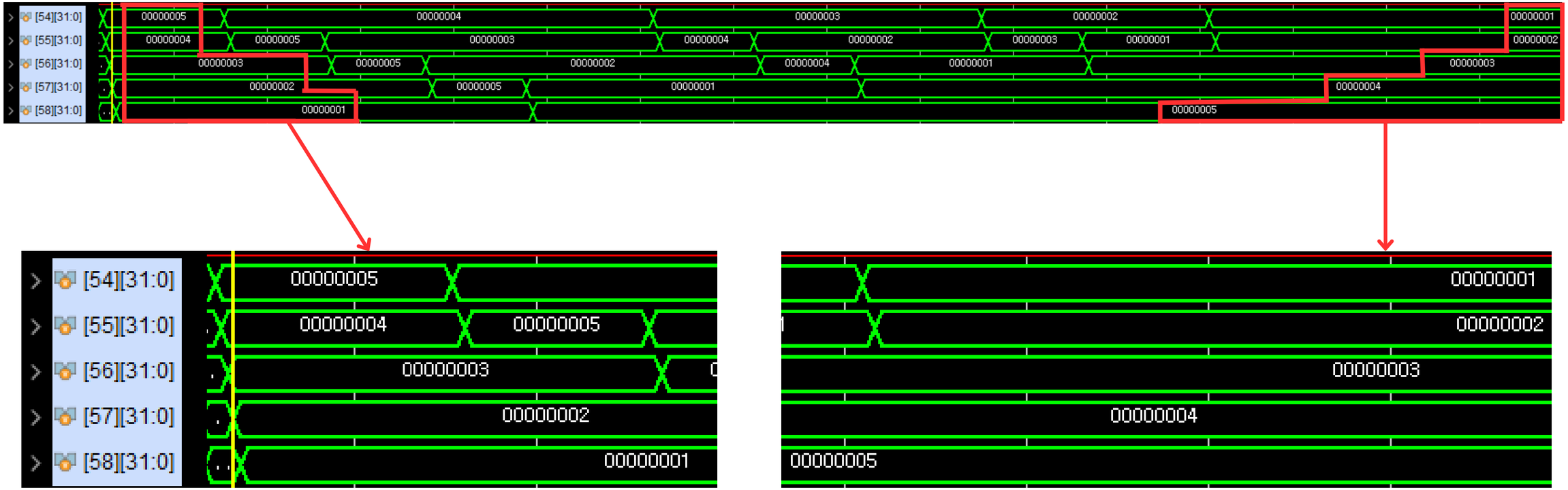
Machine Code

```
10000113
fd010113
02112623
02812423
03010413
fc042c23
fc042e23
fe042023
fe042223
fe042423
fe042623
00500793
fcf42c23
```

05 Test Program

결과

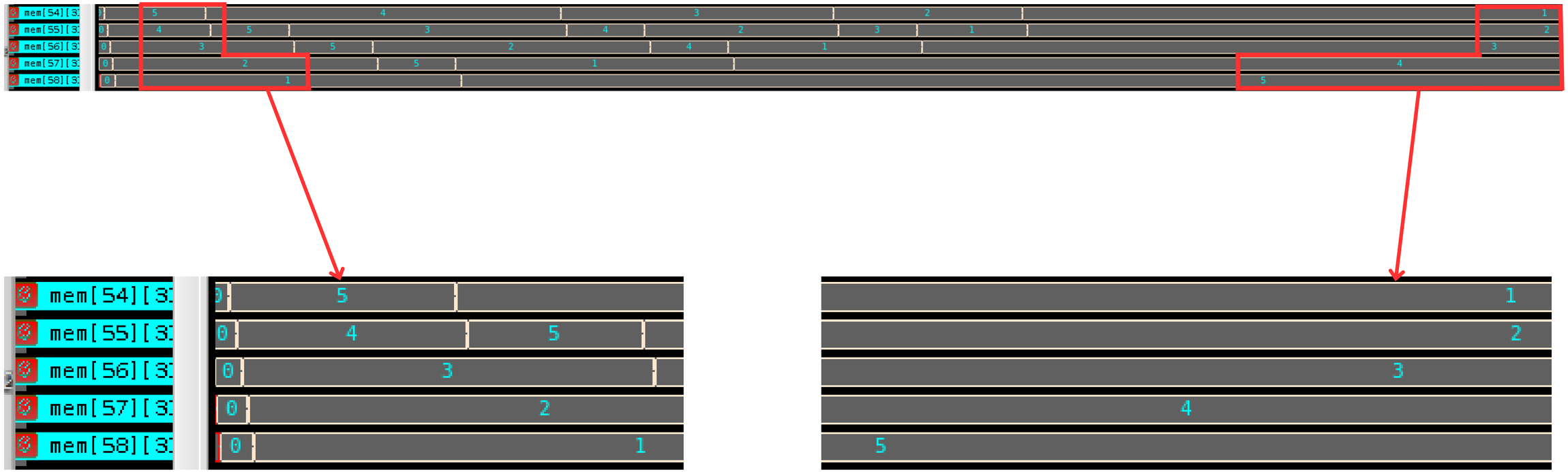
Vivado



05 Test Program

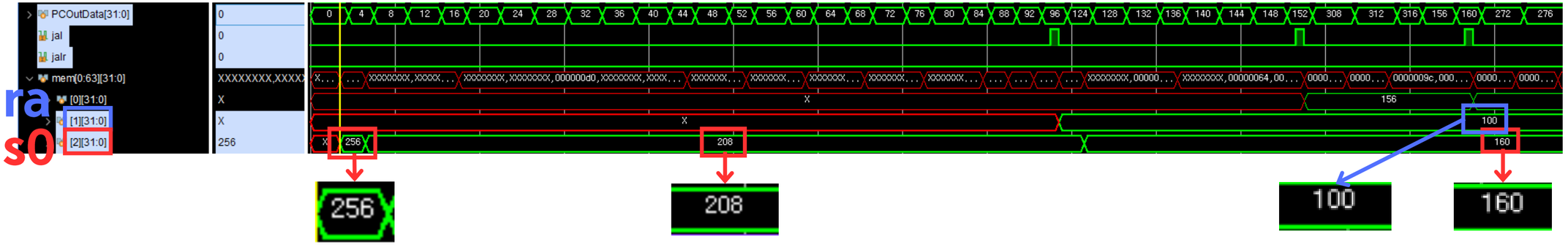
결과

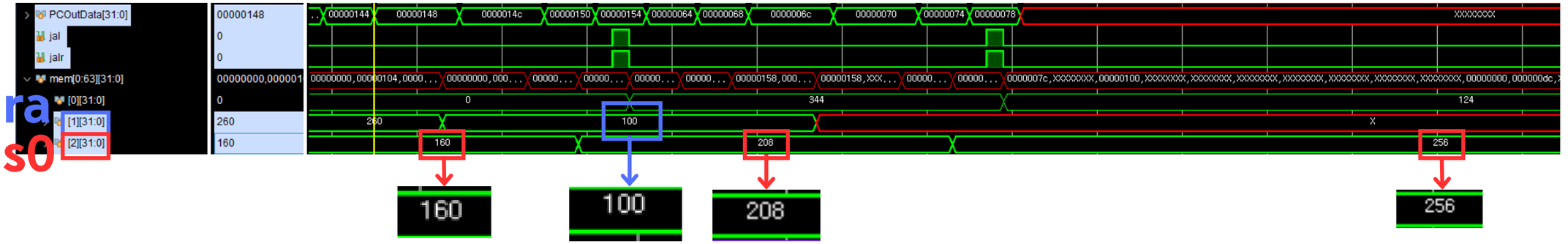
Verdi



05 Test Program

SP해제 및 RA 복귀

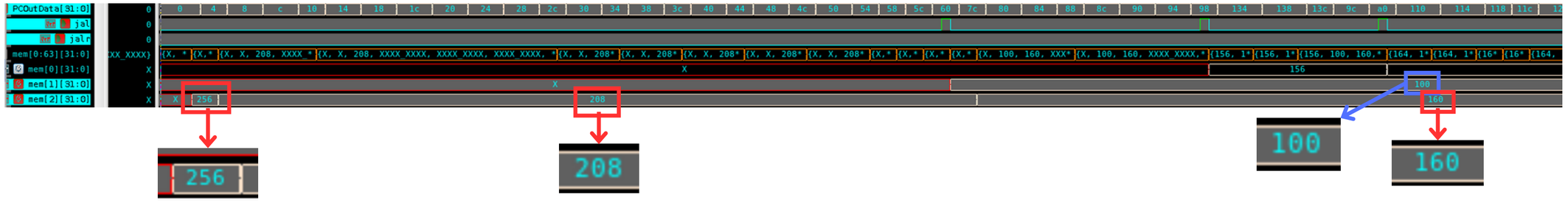




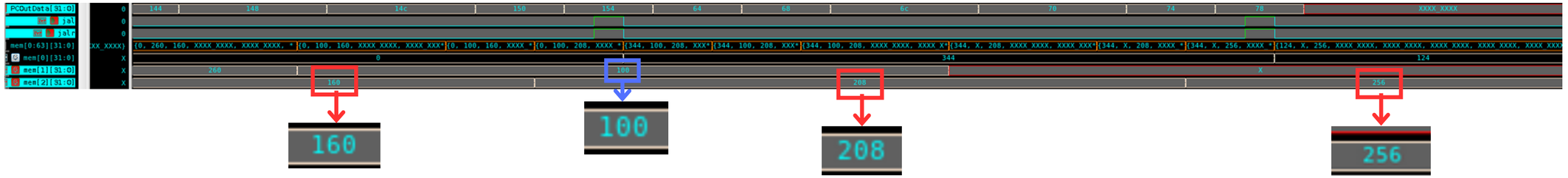
05 Test Program

SP해제 및 RA 복귀

ra
s0



ra
s0



06 Trouble Shooting

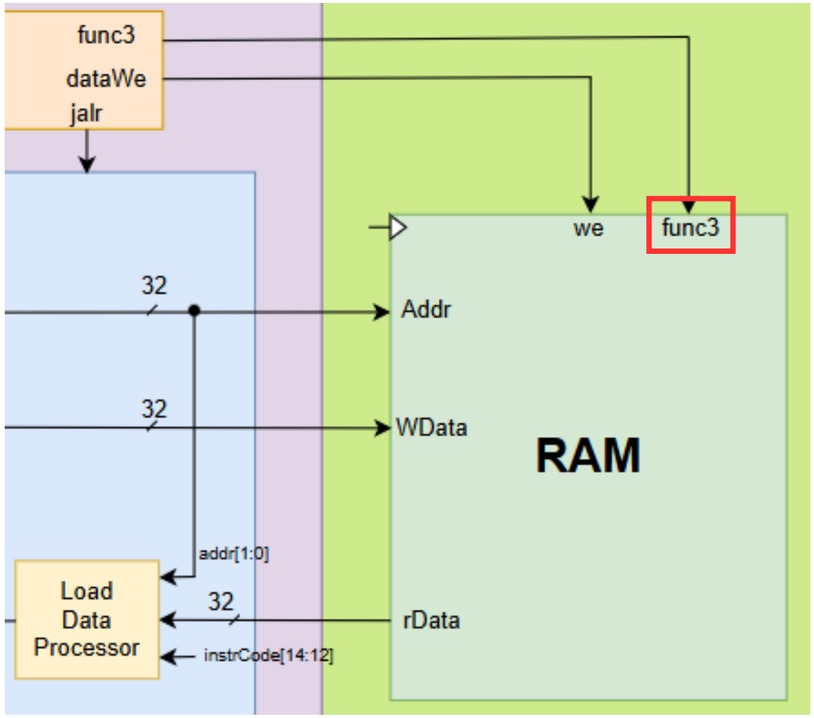
1. Store

Byte, half 설계

- 데이터 접근 방법
- Store / load 동시 처리 ?

x[base +: width] ⇒ x[base + width – 1 : base]

```
3'b000: begin // SB
    mem[width][8*addr[1:0] +: 8] <= wData[7:0];
end
3'b001: begin // SH
    mem[width][16*addr[1] +: 16] <= wData[15:0];
end
```

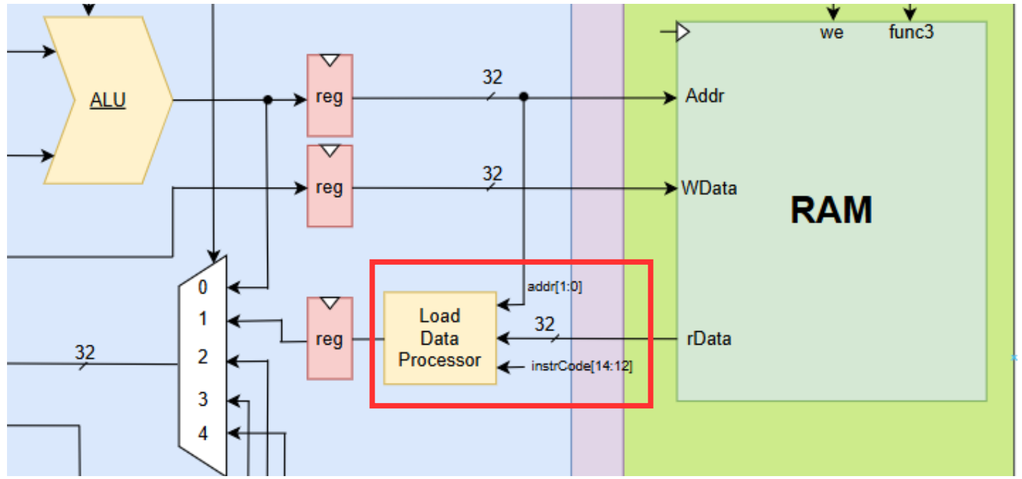


06 Trouble Shooting

2. Load

Byte, half 설계

- 데이터 접근 방법
- Store / load 동시 처리 ?




```
always_comb begin
    b = rdata >> (8 * addr);
    h = rdata >> (16 * addr[1]);
    case (func3)
        3'b000:  y = {{24{b[7]}}, b};  // LB
        3'b001:  y = {{16{h[15]}}, h};  // LH
        3'b010:  y = rdata;  // LW
        3'b100:  y = {24'b0, b};  // LBU
        3'b101:  y = {16'b0, h};  // LHU
        default: y = rdata;
    endcase
end
```

07 느낀점

imm 비트 배치

하드웨어 최적화를 위한 설계로,
디코딩을 단순화하고 처리를
더 효율적으로 만듦

스택 공간 확보

Assembly 코드에서
main, 각 함수 진입 시
스택 공간을 미리 확보하고
종료 시 되돌리는 구조를 확인

설계 우선순위

Logic 보다 Interface를
먼저 명확히 잡으면 이후의
설계·검증이 더 수월해짐

Q & A

THANK YOU

감사합니다