RISC-V 기반 cpu 설계

박찬호

Contents

01 개요

02 Block Diagram

03 RV32I type

04 TrobleShooting

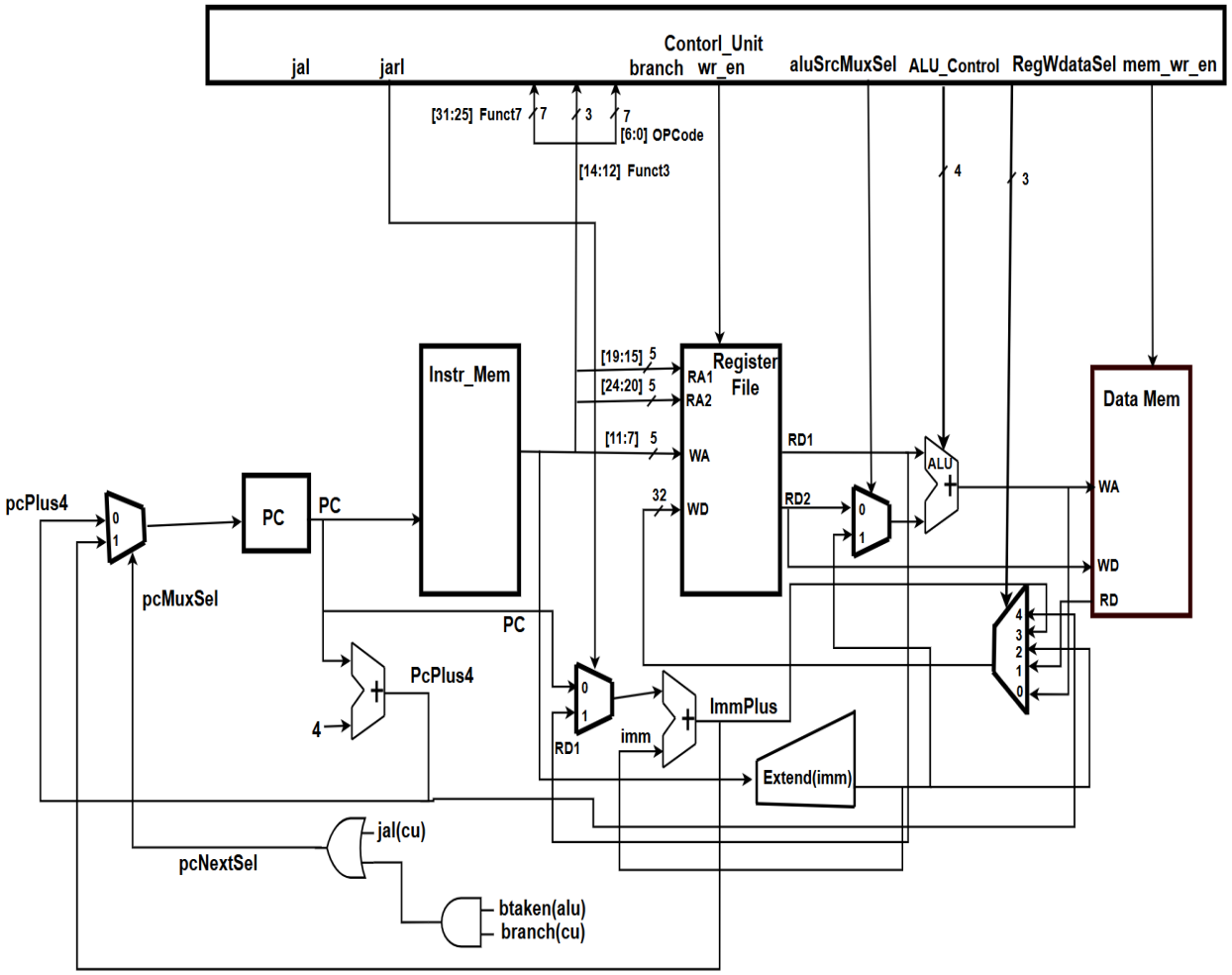
05 ASM code 동작 분석

개요

	RISC(Reduced Instruction Set Computer)	CISC(Complex Instruction Set Computer)
명령어 수	수십 개 ~ 100여 개	수백개
명령어 길이	고정	가변적
하드웨어 구조	단순함	복잡함
파이프라인	설계 용이	설계 복잡
ISA	ARM, RISC-V	x86

- RISC 아키텍처는 하드웨어가 단순하고 효율적이기 때문에 저전력, 저비용으로 설계 가능하여 임베디드 분야에서 많이 사용된다.
- RISC-V는 오픈소스 ISA로, 자유롭게 확장·설계할 수 있으며 업계에서도 활용되고 있다.

Block Diagram



- Control Unit :
  - Instrunction code를 해석하여 각 명령어에 맞는 제어 신호를 생성.
- Register files :
  - 32개의 레지스터의 묶음
  - 연산용 피연산자 저장, 임시 저장등 다양한 목적으로 사용
- Program Counter :
  - 다음에 실행할 명령어의 주소의 저장하는 레지스터
- ALU :
  - 산술 논리 연산 수행
- Instrunction memory, Data memory :
  - 명령어와 데이터를 저장하는 외부 메모리이며, 하버드 구조로 분리되어 있다.

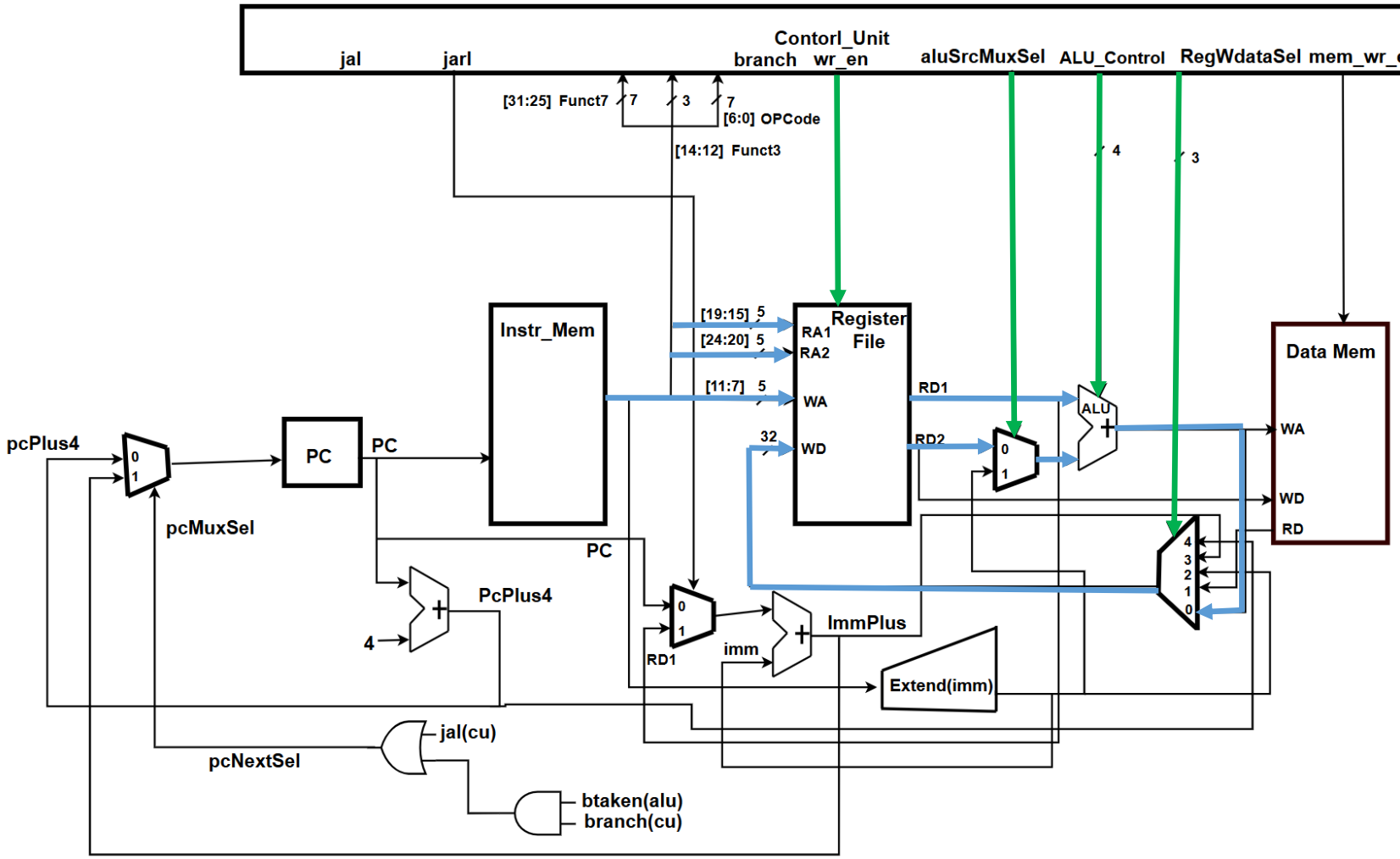
R-type Instructions

- R-type : 두 개의 register의 data를 연산 후 register에 저장

funct7	rs2	rs1	funct3	rd	opcode	R-type

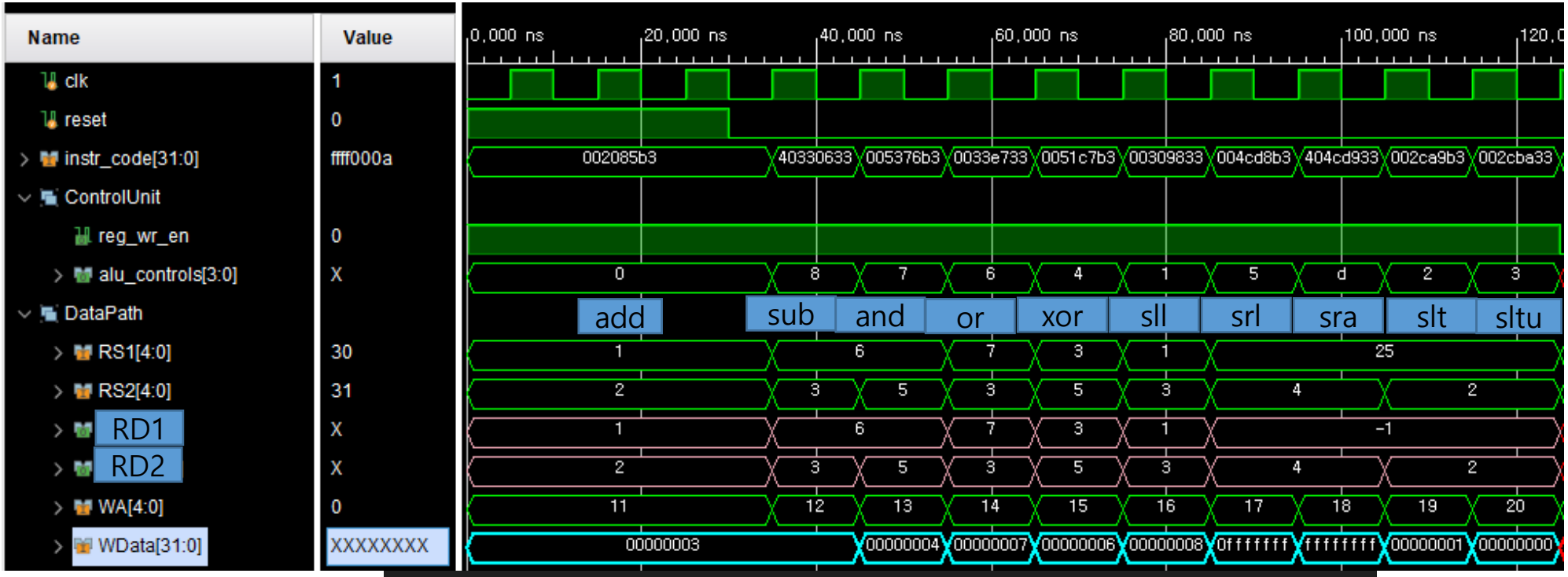
Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends

R-type Data Flow



- Reg_wr_en : 연산된 결과 레지스터에 저장
- ALU_Control : ALU가 수행할 연산 종류를 지정하는 제어 신호

R-type Simulation



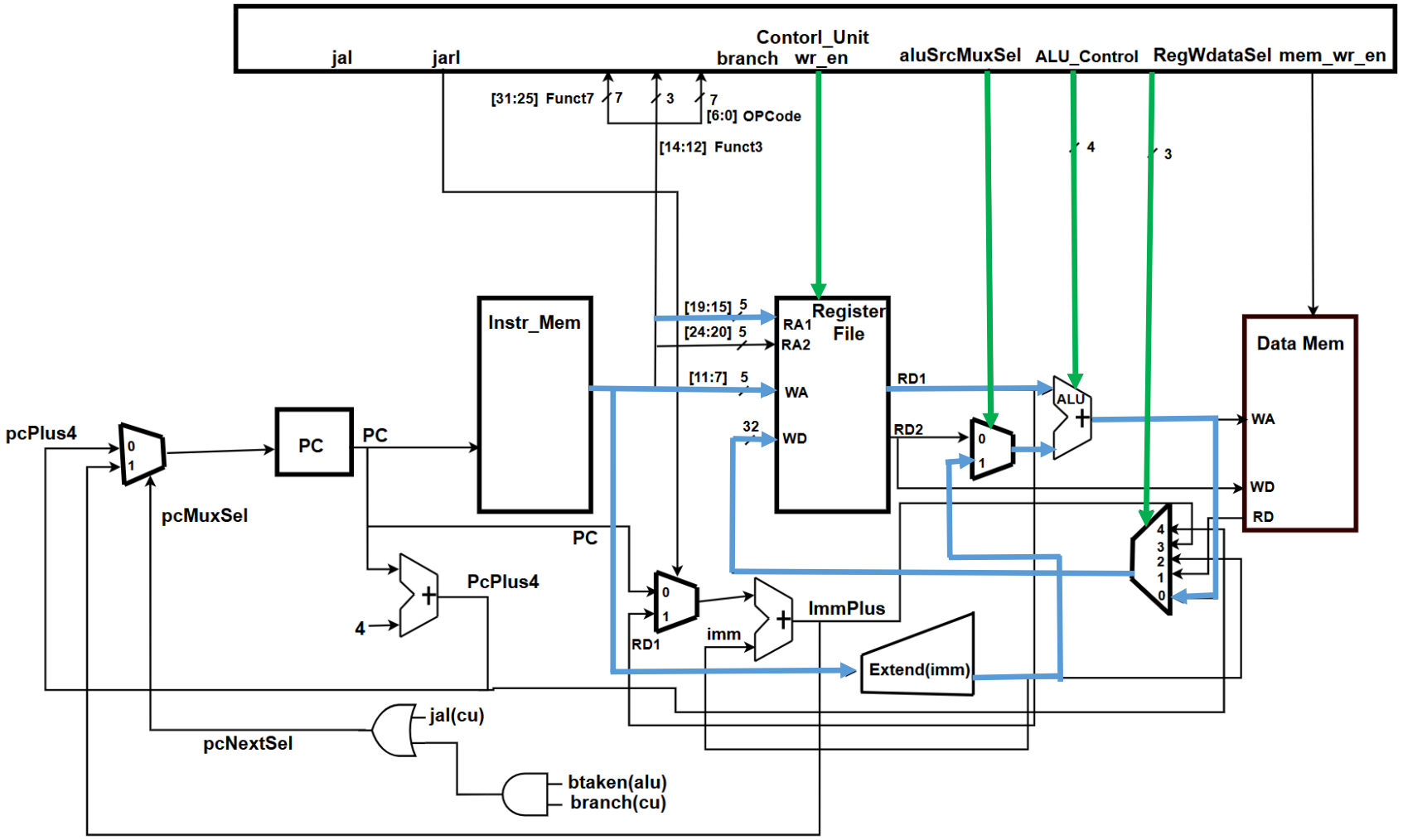

```
// R-type
rom[0] = 32'h002085B3;  // add x11, x1, x2   = 1 + 2 = 3
rom[1] = 32'h40330633;  // sub x12, x6, x3   = 6 - 3 = 3
rom[2] = 32'h005376B3;  // and x13, x6, x5   = 0110 & 0101 = 0100(4)
rom[3] = 32'h0033E733;  // or x14, x7, x3    = 0111 | 0011 = 0111(7)
rom[4] = 32'h0051C7B3;  // xor x15, x3, x5   = 0011 ^ 0101 = 0110(6)
rom[5] = 32'h00309833;  // sll x16, x1, x3   = 0001 << 3 = 1000(8)
rom[6] = 32'h004CD8B3;  // srl x17, x25, x4  = ffff_ffff >> 4 = 0fff_ffff
rom[7] = 32'h404CD933;  // sra x18, x25, x4  = ffff_ffff >>> 4 = ffff_ffff
rom[8] = 32'h002CA9B3;  // slt x19, x25, x2  = ffff_ffff(-1) < 2 = 1
rom[9] = 32'h002CBA33;  // sltu x20, x25, x2 = ffff_ffff > 2 = 0
```

I-type Instructions

- I-type : register의 data와 Immediate 값을 연산 후 register에 저장
- R-type과 달리 sub 연산이 없는데 Immediate 값을 음수로 세팅하고 add하면 sub 연산을 할 수 있기 때문에 필요 없다.

imm[11:0]		rs1		funct3	rd	opcode	I-type	
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm		
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm		
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm		
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm		
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]		
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]		
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]		msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0		
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0		zero-extends

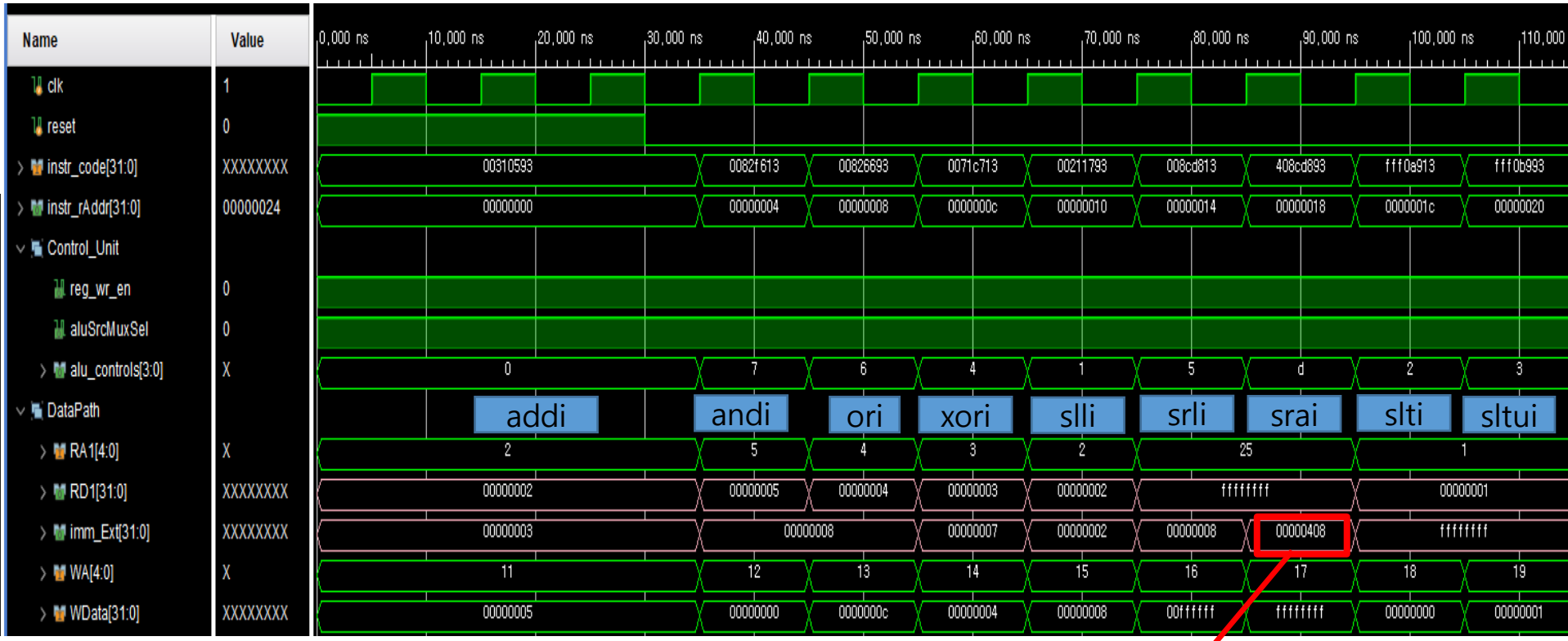
I-type Data Flow



- Reg_wr_en : 연산된 결과 레지스터에 저장
- aluSrcMuxSel : 레지스터 값이 아닌 Immediate 값 선택하기 위해 1로 출력

I-type Simulation

```
// I-type
rom[0] = 32'h00310593;  // addi  x11, x2, 3 = 5
rom[1] = 32'h0082F613;  // andi  x12, x5, 8 = 0
rom[2] = 32'h00826693;  // ori   x13, x4, 8 = 12
rom[3] = 32'h0071C713;  // xori  x14, x3, 7 = 4
rom[4] = 32'h00211793;  // slli  x15, x2, 2 = 8
rom[5] = 32'h008CD813;  // srli  x16, x25, 8 = 00ff_ffff
rom[6] = 32'h408CD893;  // srai  x17, x25, 8 = ffff_ffff
rom[7] = 32'hFFF0A913;  // slti  x18, x1, -1 = 0
rom[8] = 32'hFFF0B993;  // sltiu x19, x1, -1 = 1
```



srli와 srai는 imm[10]으로 구분하지만, ALU에서는 imm[4:0]만 사용하므로 실제 연산에는 영향이 없다.

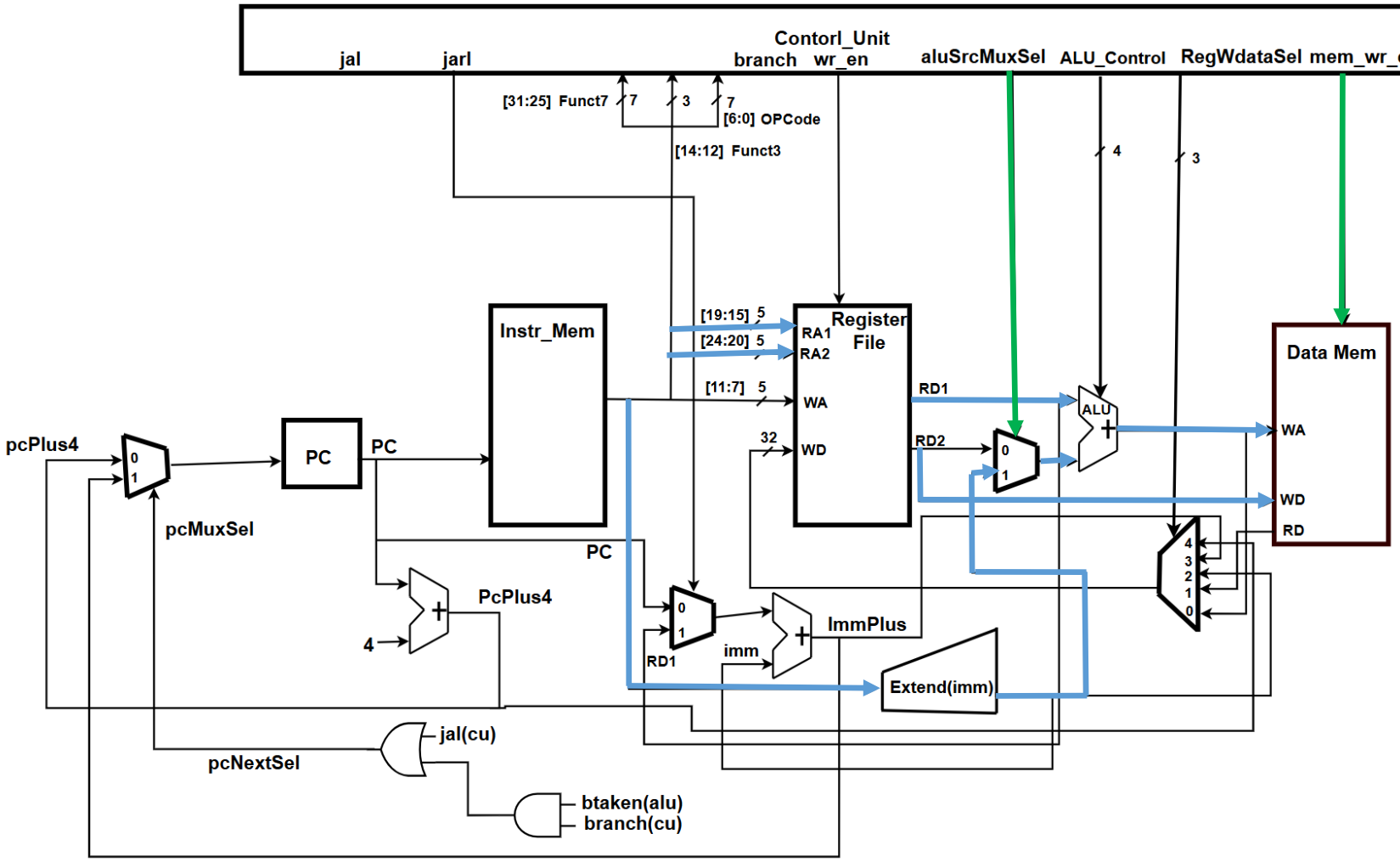
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4] msb-extends

S-type Instructions

- S-type : register의 data를 RAM에 store
- rs1과 Immediate 값을 더해 저장할 주소를 계산하고 rs2의 데이터를 저장

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]

S-type Data Flow

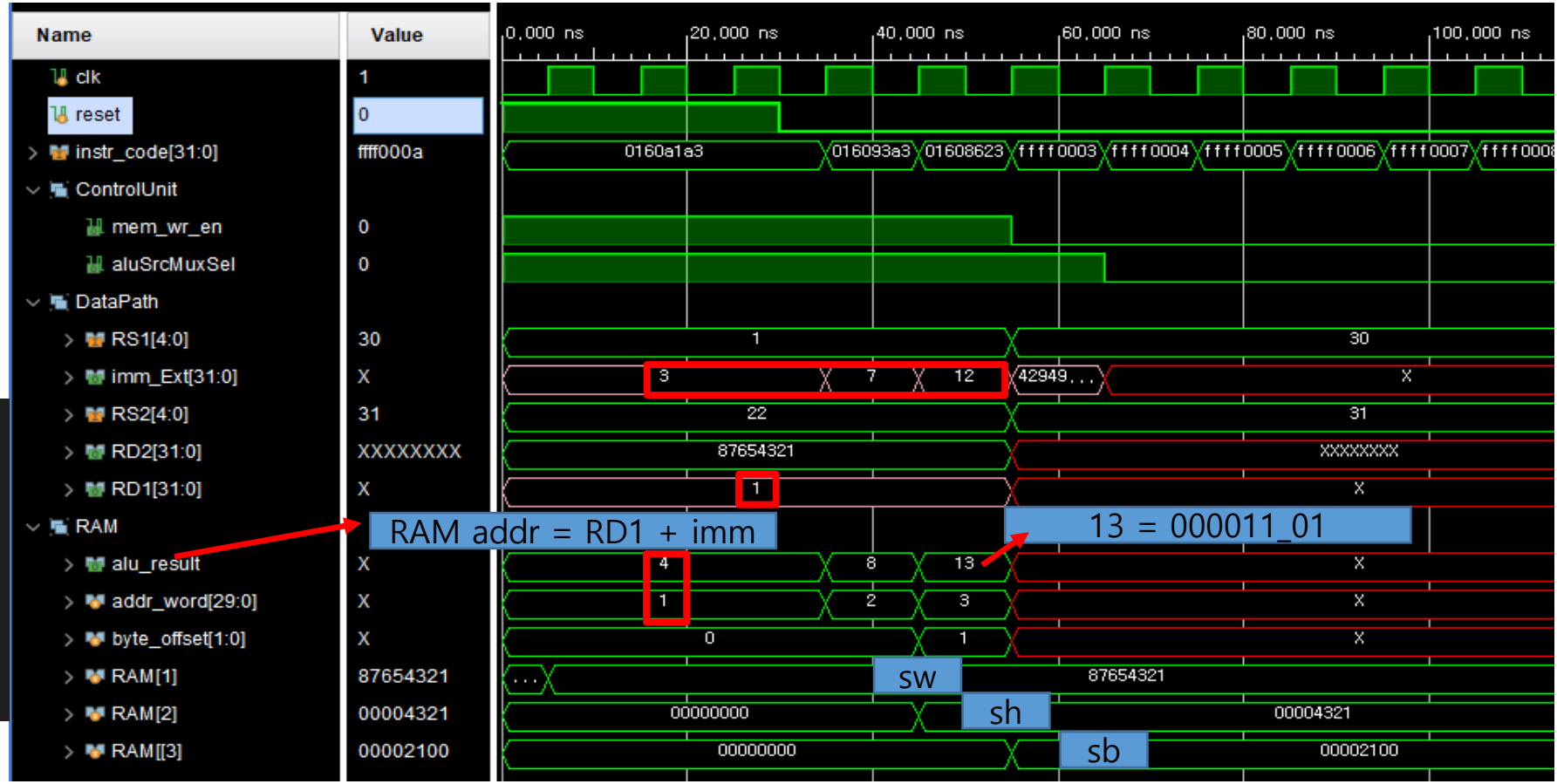


- mem_wr_en : 레지스터의 데이터를 RAM에 저장하기 위해 1로 출력
- aluSrcMuxSel : RAM의 주소 = RD1 + imm

S-type Simulation

바이트 단위로 메모리에 접근하기 위해 주소의 하위 2비트는 byte_offset으로 사용

```
logic [31:0] data_mem[0:63];
assign addr_word   = dAddr[31:2];
assign byte_offset = dAddr[1:0];  // for byte/halfword access

// S-type
rom[0] = 32'h0160A1A3;  //sw x22 3(x1)  mem[1] = 32'h8765_4321
rom[1] = 32'h016093A3;  //sh x22 7(x1)  mem[2] = 32'h0000_4321
rom[2] = 32'h01608623;  //sb x22 12(x1) mem[3] = 32'h0000_2100
```

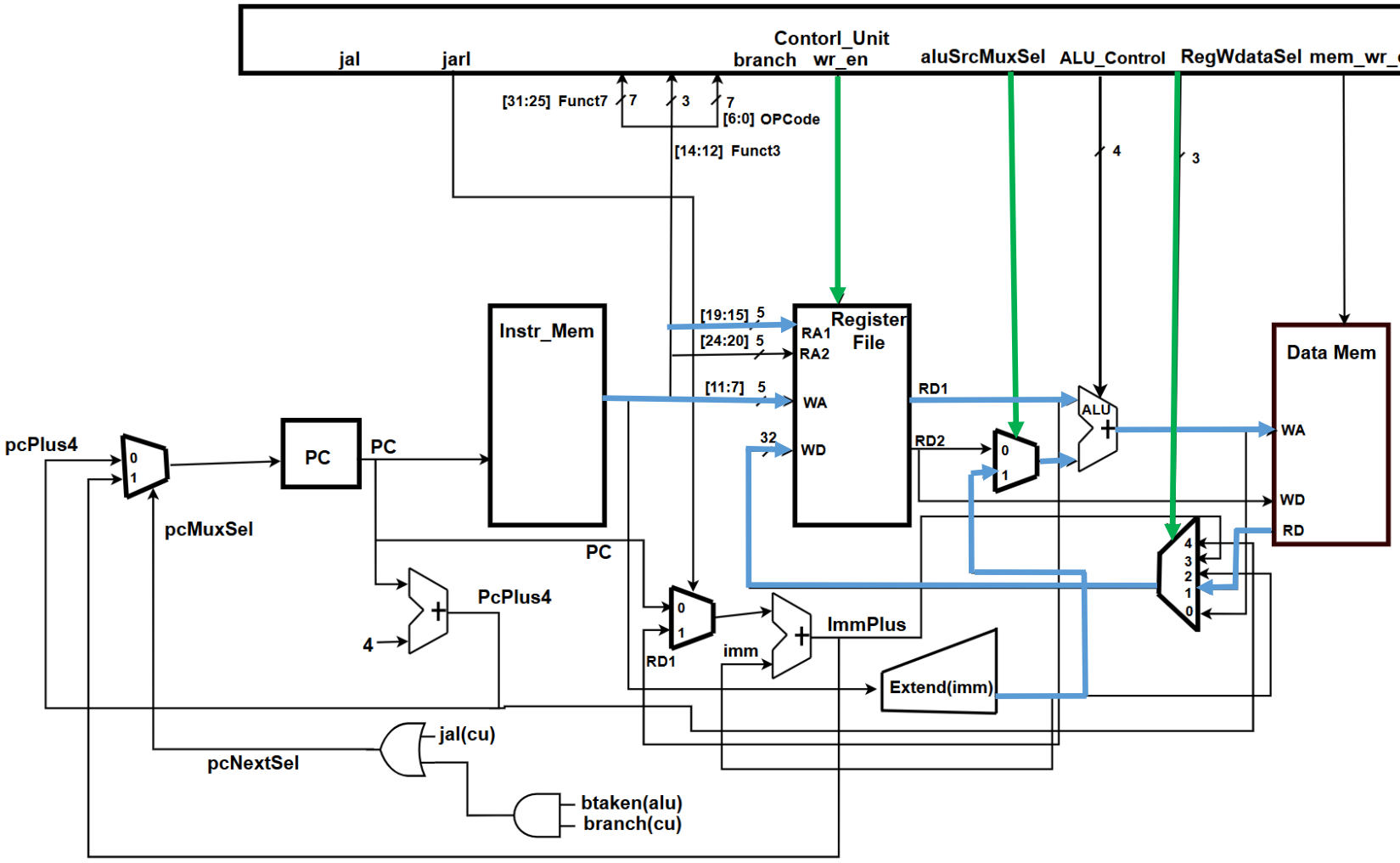


IL-type Instructions

- IL-type : RAM의 data를 register에 Load
- rs1과 Immediate 값을 더해 load할 주소를 계산하고 rd에 저장

		imm[11:0]	rs1	funct3	rd	opcode	I-type	
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]		
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]		
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]		
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]		zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]		zero-extends

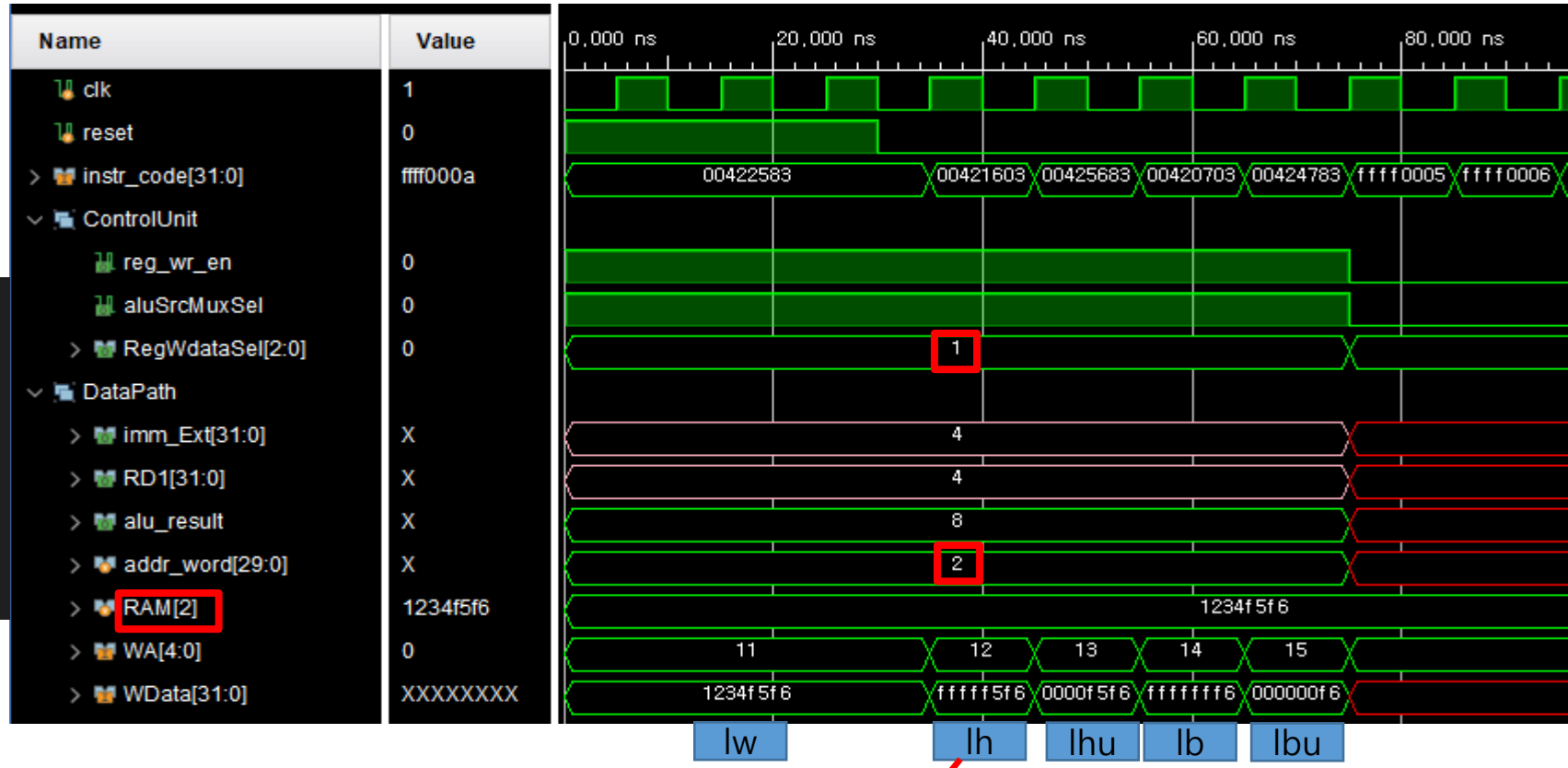
IL-type Data Flow



- reg_wr_en : RAM에서 읽어 온 데이터를 레지스터에 저장
- RegWdataSel : RAM에서 읽어 온 데이터를 레지스터에 입력

IL-type Simulation

```
// IL-type
rom[0] = 32'h00422583;  // lw x11 4(x4)
rom[1] = 32'h00421603;  // lh x12 4(x4)
rom[2] = 32'h00425683;  // lhu x13 4(x4)
rom[3] = 32'h00420703;  // lb x14 4(x4)
rom[4] = 32'h00424783;  // lbu x15 4(x4)
```



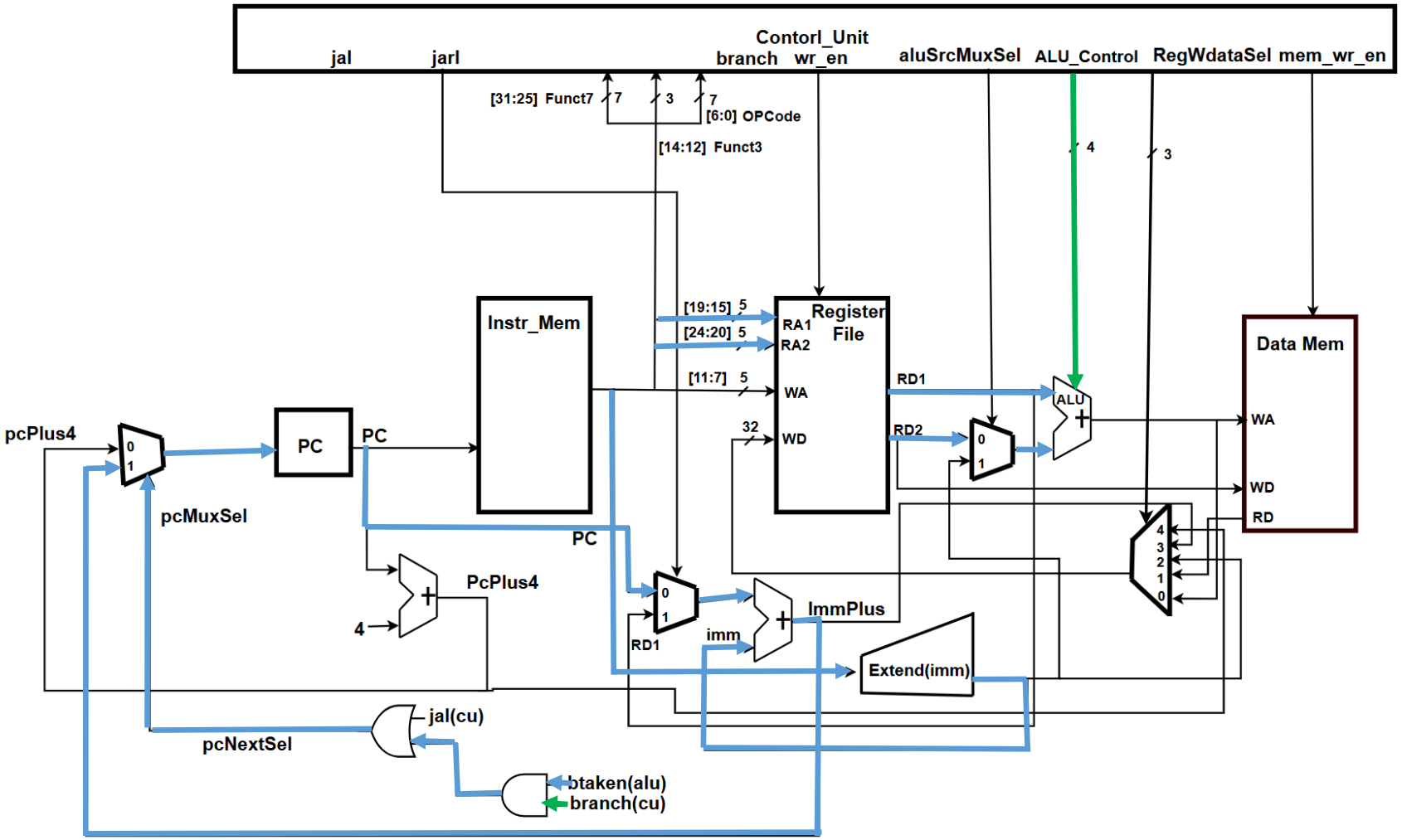
lh, lb : sign-extend
lhu, lbu : zero-extend

B-type Instructions

- B-type : Branch 조건이 맞다면 pc + Immediate로 분기
- 어셈블리어에서 if, for, while 같은 조건문 구현에 활용된다.

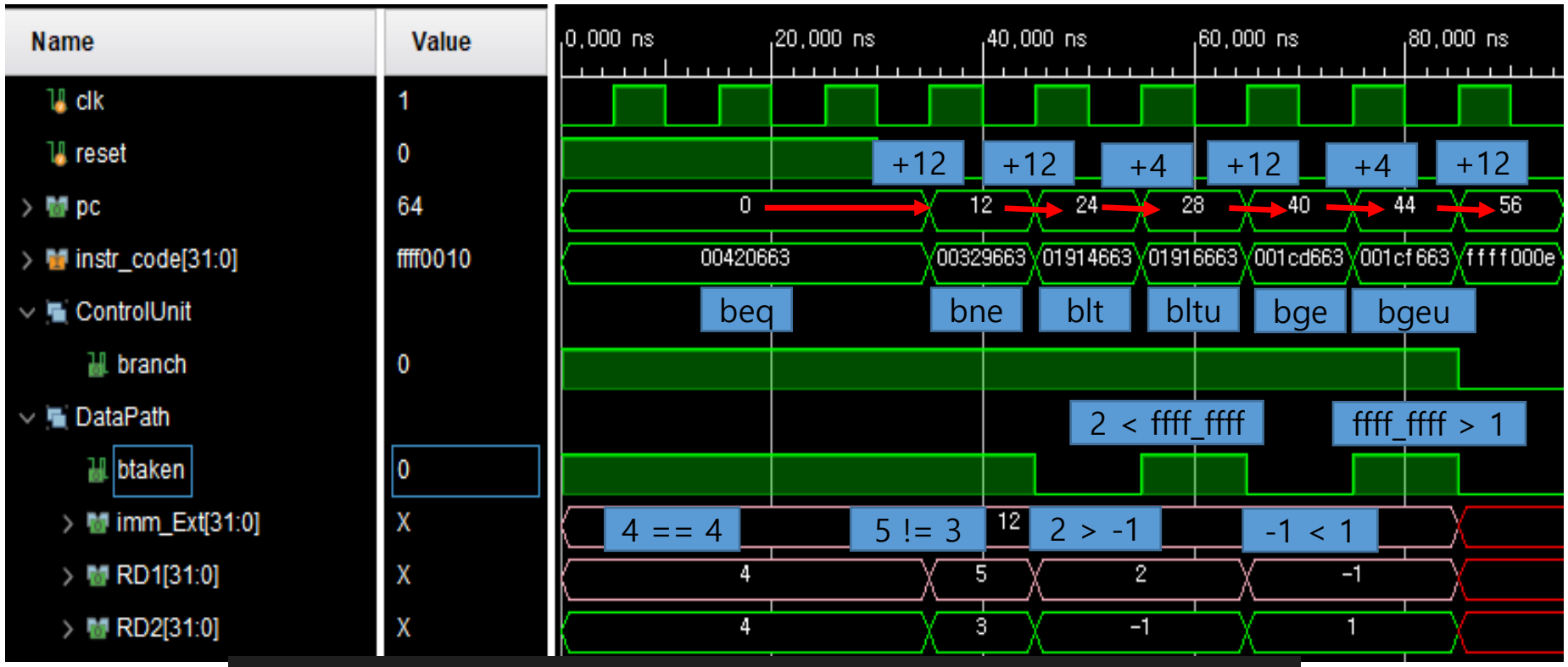
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends

B-type Data Flow



- branch : control_unit에서 1 출력
- Btaken : 조건이 맞다면 1 아니면 0 (alu 출력)

B-type Simulation



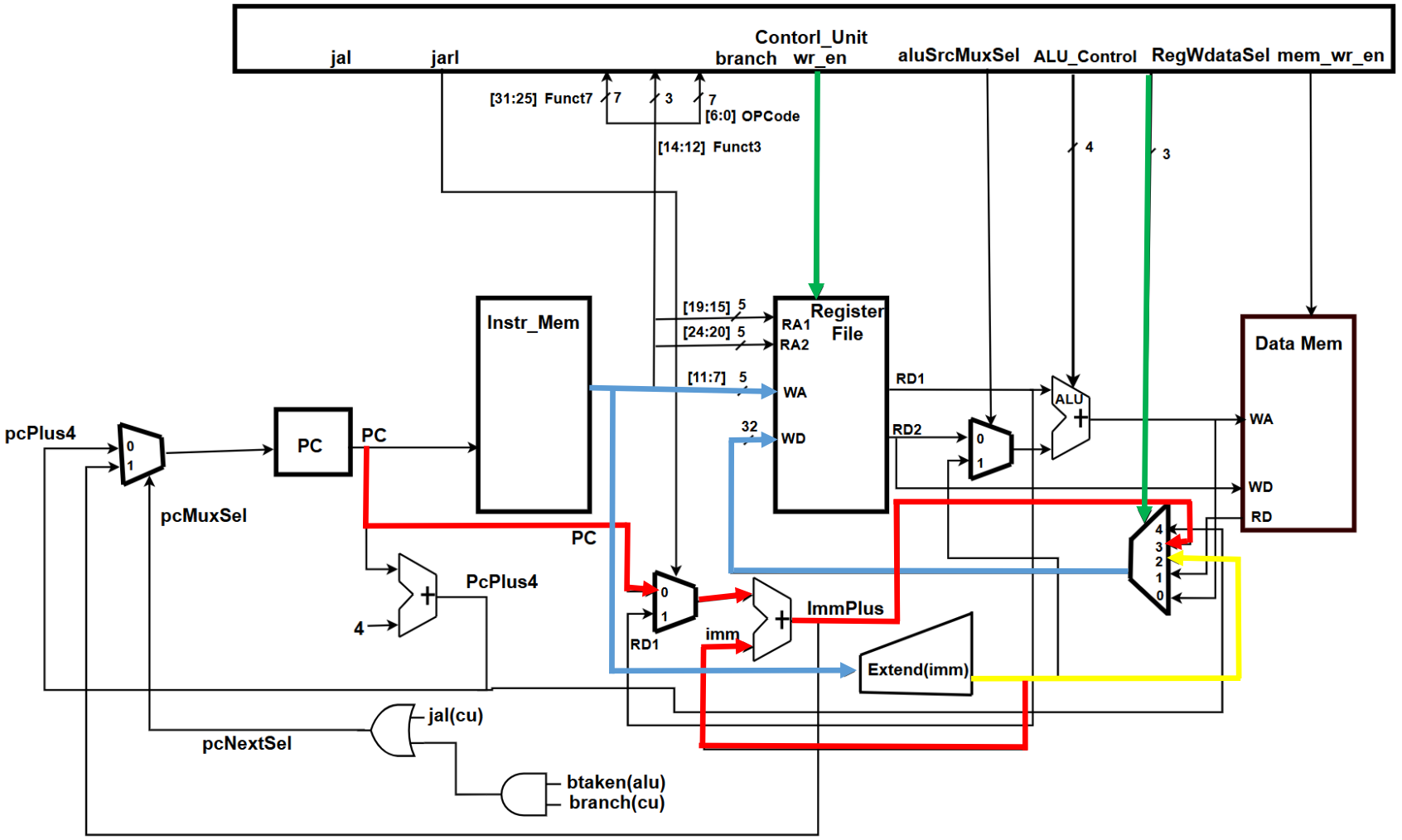

```
// B-type
rom[0] = 32'h00420663;  // beq x4, x4, 12 -> 4 == 4 branch pc -> 12
rom[3] = 32'h00329663;  // bne x5, x3, 12 -> 5 != 3 branch pc -> 24
rom[6] = 32'h01914663;  //blt x2 x25 12   -> 2 > -1 miss pc -> 28
rom[7] = 32'h01916663;  //bltu x2 x25 12  -> 2 < ffff_ffff branch pc -> 40
rom[10] = 32'h001CD663;  //bge x25 x1 12  -> -1 < 1 miss pc -> 44
rom[11] = 32'h001CF663;  //bgeu x25 x1 12 -> ffff_ffff > 1 branch pc -> 56
```

U-type Instructions

- U-type : Immediate 값을 레지스터에 저장
- 다른 type들은 sign extension을 하지만, U-type은 left shift하기 때문에 큰 값을 저장할 수 있다.

imm[31:12]					rd	opcode	U-type
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	

U-type Data Flow



lui

auipc

- reg_wr_en : 레지스터에 shift된 데이터 저장
- RegWdataSel :
  - lui -> Immediate << 12 저장 할 때 2로 설정
  - auipc -> pc + Immediate << 12 저장 할 때 3으로 설정

U-type Simulation


```
// U-type
rom[0] = 32'h000035B7;  // lui x11 32'h0000_0003 -> x11 = 32'h0000_3000
rom[1] = 32'h00004617;  // auipc x12 32'h0000_0004 -> x12 = 32'h0000_4004
```

lui x11, 32'h3
rd = 3 << 12

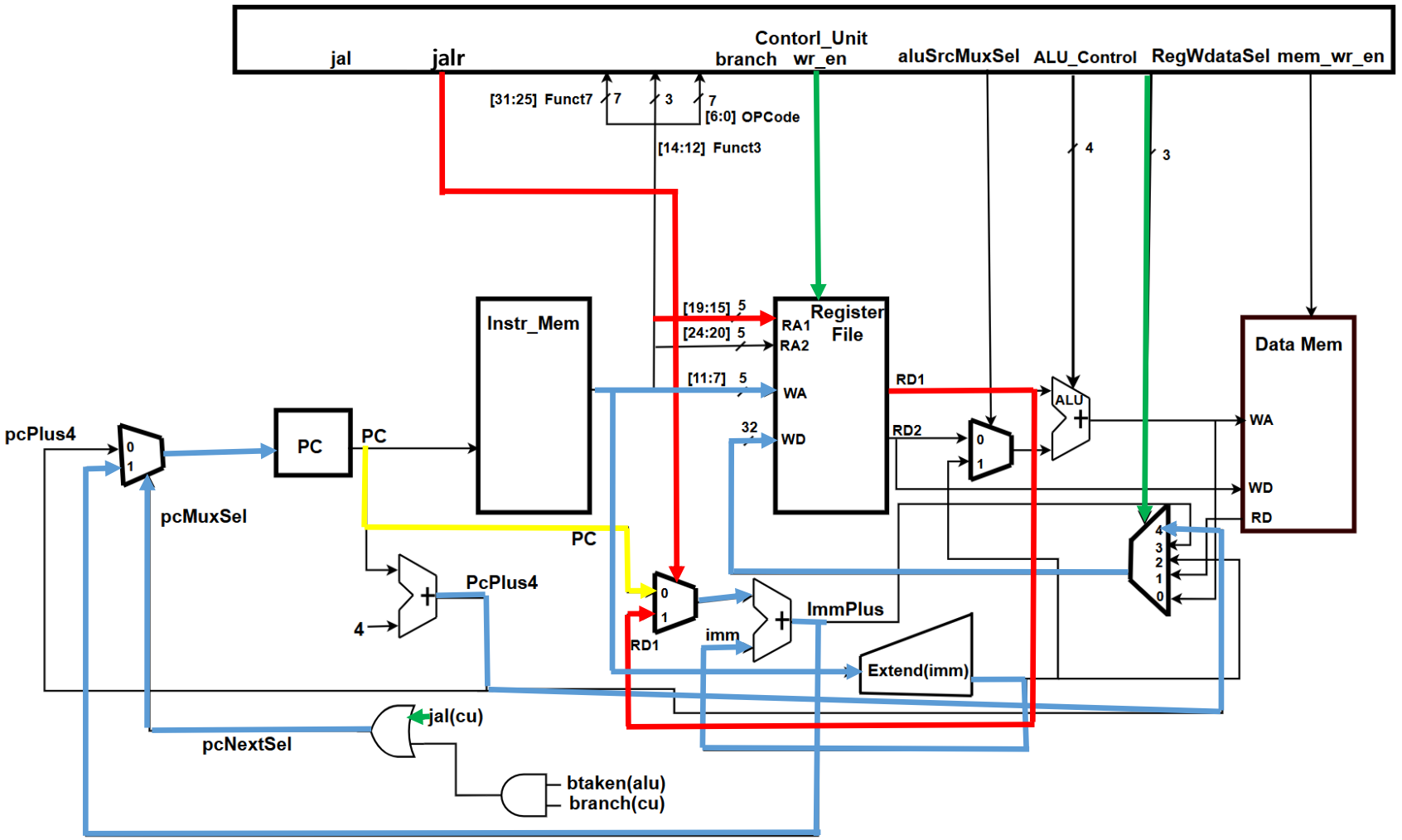
auipc x12, 32h'4
rd = 4 << 12 + 4

J-type Instructions

- J-type : 현재 pc 값 + 4를 register에 저장, pc 특정 값으로 점프
- 어셈블리어에서 함수 호출 구현에 사용된다.

imm[20|10:1|11|19:12]					rd	opcode	J-type
imm[11:0]		rs1		funct3	rd	opcode	I-type
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	

J-type Data Flow

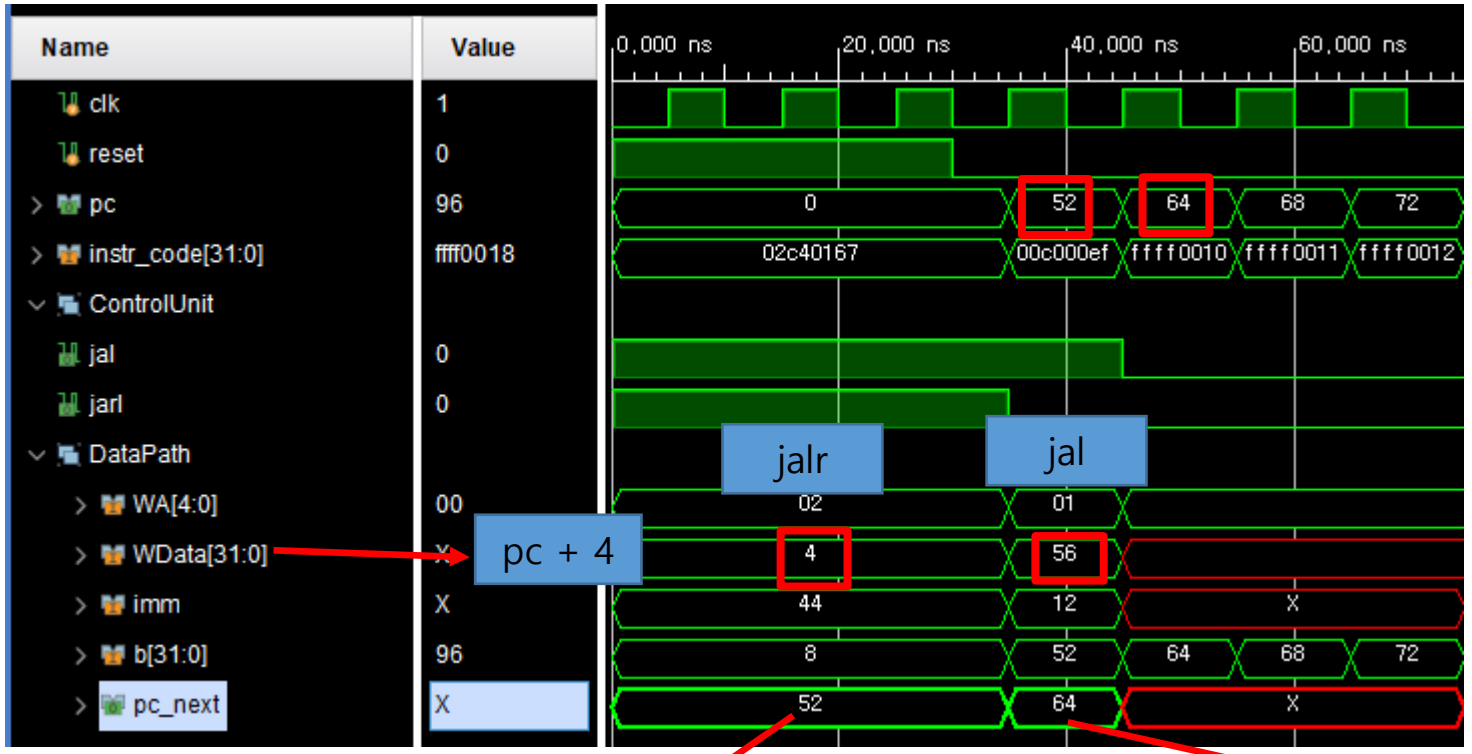


jal

jalr

- reg_wr_en : 레지스터에 pc + 4 저장
- RegWdataSel : pc + 4 선택하기 위해 4로 설정
- Jarl :
  - 0 : jal -> pc + Immediate
  - 1 : jalr -> RD1 + Immediate

J-type Simulation



pc_next = 44(imm) + 8(x8)

pc_next = 12(imm) + 52(pc)

```
// J-type
rom[0]  = 32'h02C40167;  // jalr x2, x8, 44 x2 = 0 + 4 = 4, pc = 44 + 8 = 52
rom[13] = 32'h00C000EF;  // jal x1, 12, x1 = 52 + 4 = 56, pc = 12 + 52 = 64
```

스택포인터

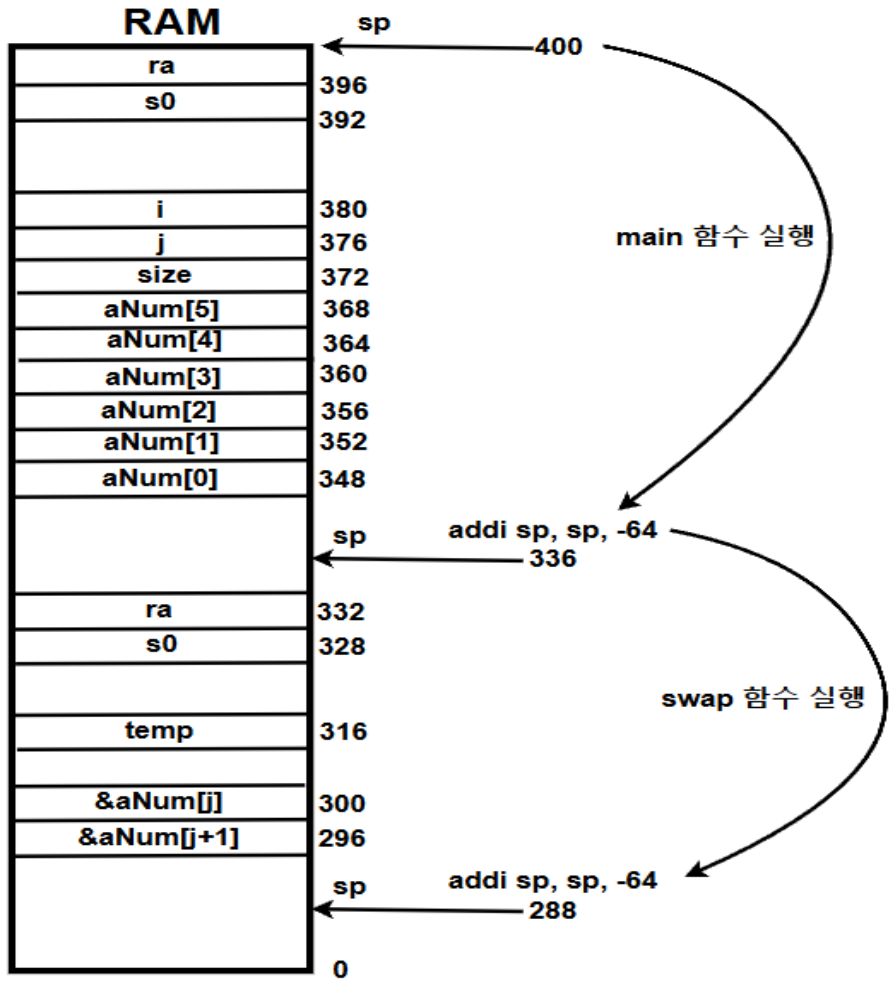
sp : 현재 스택 프레임의 가장 낮은 메모리 주소를 가리킨다.

sp의 가장 중요한 역할 중 하나는 함수가 시작될 때 해당 함수에서 사용할 지역 변수 및 임시 데이터가 저장될 메모리 공간을 확보한다.

```c
void swap(int *a, int *b);

int main(){
    int aNum[6] ={4, 1, 0, 2, 3};
    int size = 5;

    for(int i = 0; i < size-1; i++){
        for(int j = 0; j < size-i-1; j++){
            if(aNum[j] > aNum[j+1])
                swap(&aNum[j], &aNum[j+1]);
        }
    }
    return 0;
}

void swap(int *a, int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
    return ;
}
```

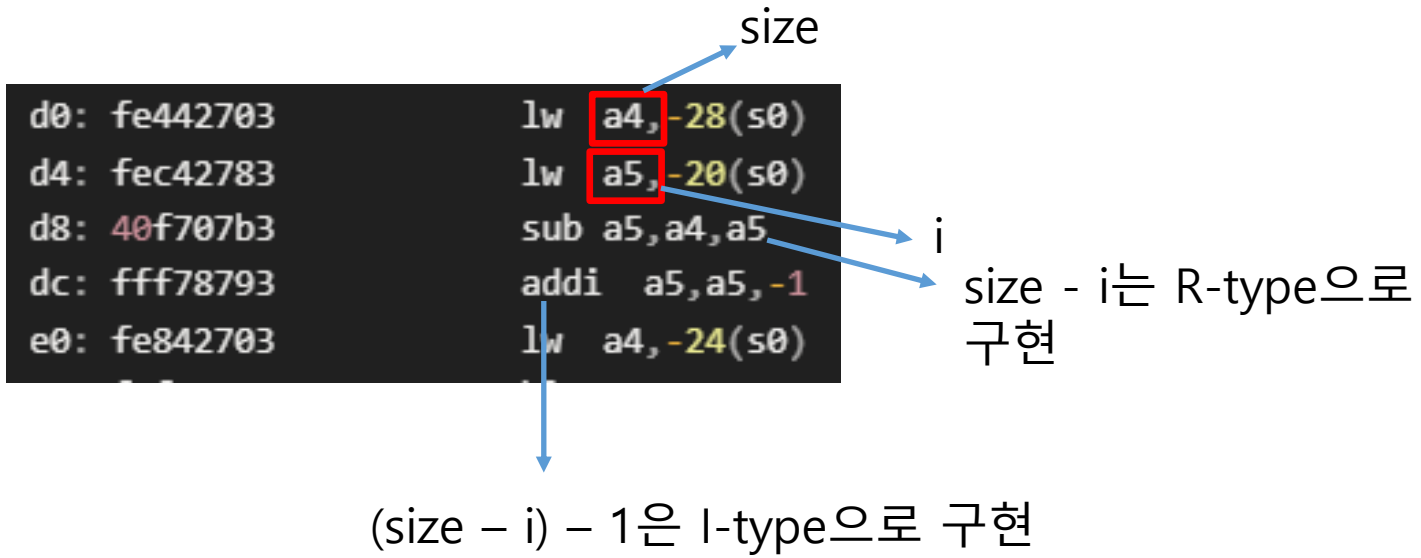
```
 4: fc010113            addi  sp,sp,-64
 8: 02112e23            sw  ra,60(sp)
 c: 02812c23            sw  s0,56(sp)
10: 04010413            addi  s0,sp,64
14: fc042623            sw  zero,-52(s0)
18: fc042823            sw  zero,-48(s0)
1c: fc042a23            sw  zero,-44(s0)
20: fc042c23            sw  zero,-40(s0)
24: fc042e23            sw  zero,-36(s0)
28: fe042023            sw  zero,-32(s0)
```

```
000000000000011c :
11c: fd010113           addi  sp,sp,-48
120: 02112623           sw  ra,44(sp)
124: 02812423           sw  s0,40(sp)
128: 03010413           addi  s0,sp,48
12c: fca42e23           sw  a0,-36(s0)
130: fcb42c23           sw  a1,-40(s0)
```



R-type vs I-type

```c
void swap(int *a, int *b);

int main(){
    int aNum[6] ={4, 1, 0, 2, 3};
    int size = 5;

    for(int i = 0; i < size-1; i++){
        for(int j = 0; j < size-i-1; j++){
            if(aNum[j] > aNum[j+1])
                swap(&aNum[j], &aNum[j+1]);
        }
    }
    return 0;
}

void swap(int *a, int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
    return ;
}
```



branch

```c
void swap(int *a, int *b);

int main(){
    int aNum[6] ={4, 1, 0, 2, 3};
    int size = 5;

    for(int i = 0; i < size-1; i++){
        for(int j = 0; j < size-i-1; j++){
            if(aNum[j] > aNum[j+1])
                swap(&aNum[j], &aNum[j+1]);
        }
    }
    return 0;
}

void swap(int *a, int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
    return ;
}
```

어셈블리어에서 if, for, while 같은 조건문 구현에 활용된다.

```
dc: fff78793          addi  a5,a5,-1
e0: fe842703          lw  a4,-24(s0)
e4: f8f740e3          blt a4,a5,64
```

if a4(i) < a5(size - 1) -> 분기

```
88: 00f687b3          add a5,a3,a5
8c: 0007a783          lw  a5,0(a5)
90: 02e7da63          ble a4,a5,c4
94: fcc40713          addi  a4,s0,-52
```

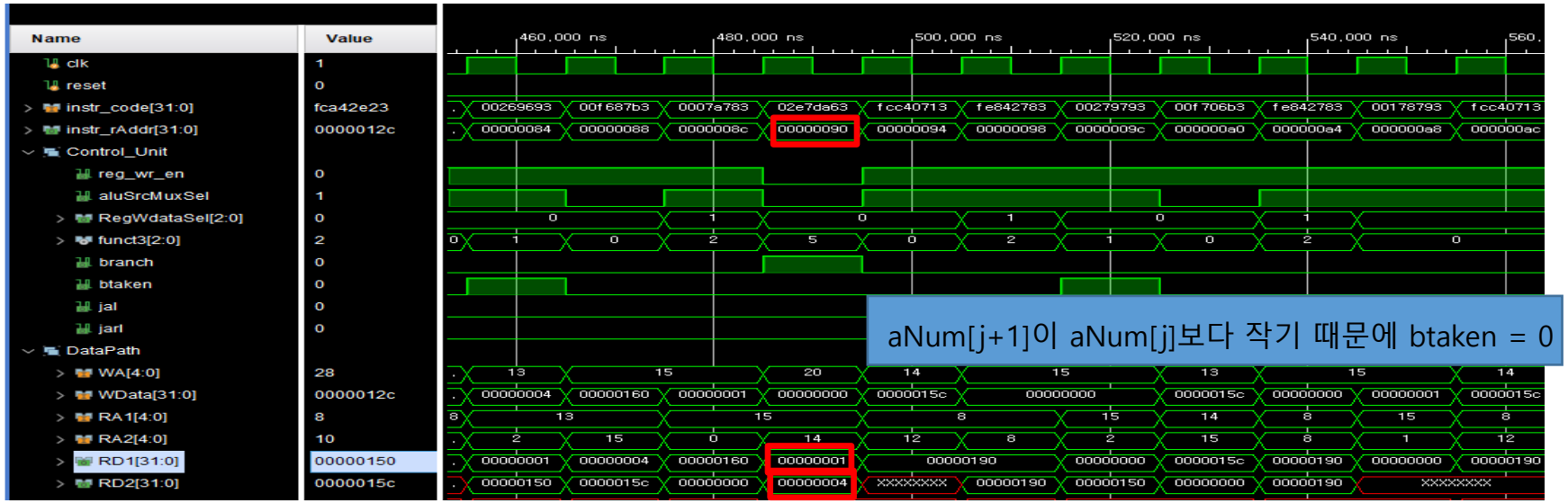
if aNum[j+1] >= aNum[j]-> 분기

ble rs, rt, offset	bge rt, rs, offset	Branch if ≤

branch Simulation


```
for(int i = 0; i < size-1; i++){
```

```
dc: fff78793        addi  a5,a5,-1
e0: fe842703        lw  a4,-24(s0)
e4: f8f740e3        blt a4,a5,64
```




```
if(aNum[j] > aNum[j+1])
```

```
88: 00f687b3        add a5,a3,a5
8c: 0007a783        lw  a5,0(a5)
90: 02e7da63        ble a4,a5,c4
94: fcc40713        addi  a4,s0,-52
```

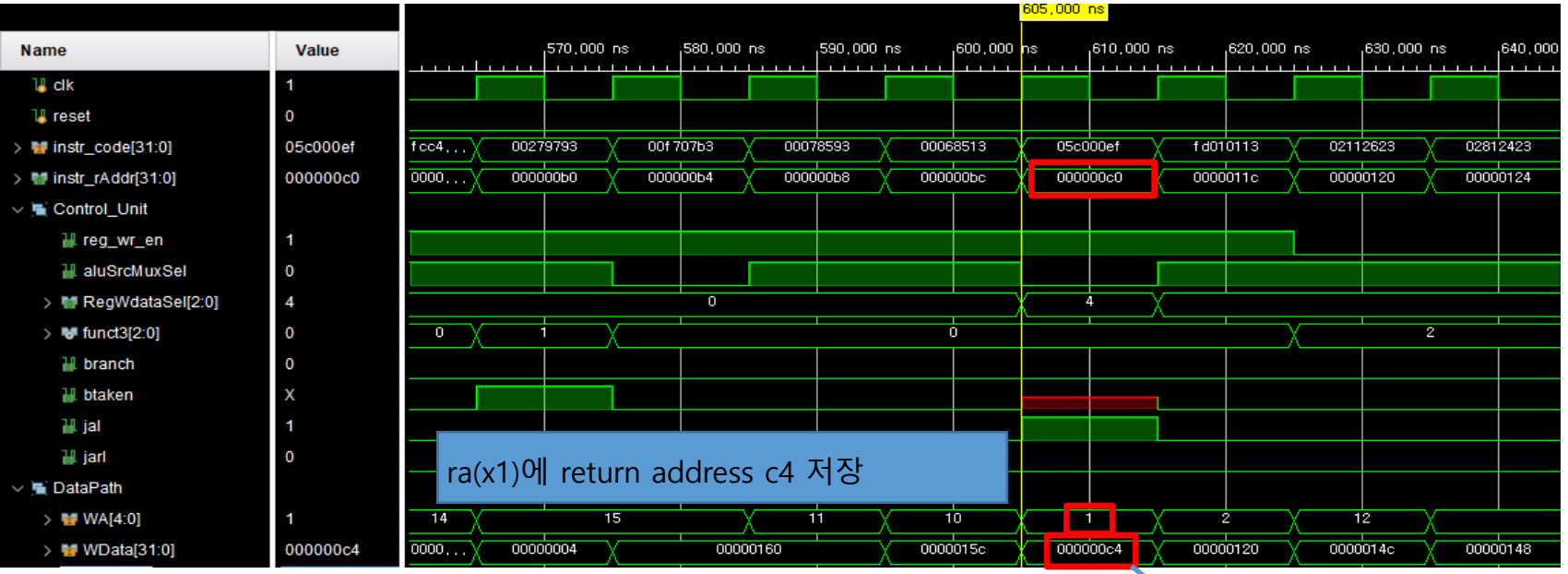
call, return

```c
void swap(int *a, int *b);

int main(){
    int aNum[6] ={4, 1, 0, 2, 3};
    int size = 5;

    for(int i = 0; i < size-1; i++){
        for(int j = 0; j < size-i-1; j++){
            if(aNum[j] > aNum[j+1])
                swap(&aNum[j], &aNum[j+1]);
        }
    }
    return 0;
}

void swap(int *a, int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
    return ;
}
```

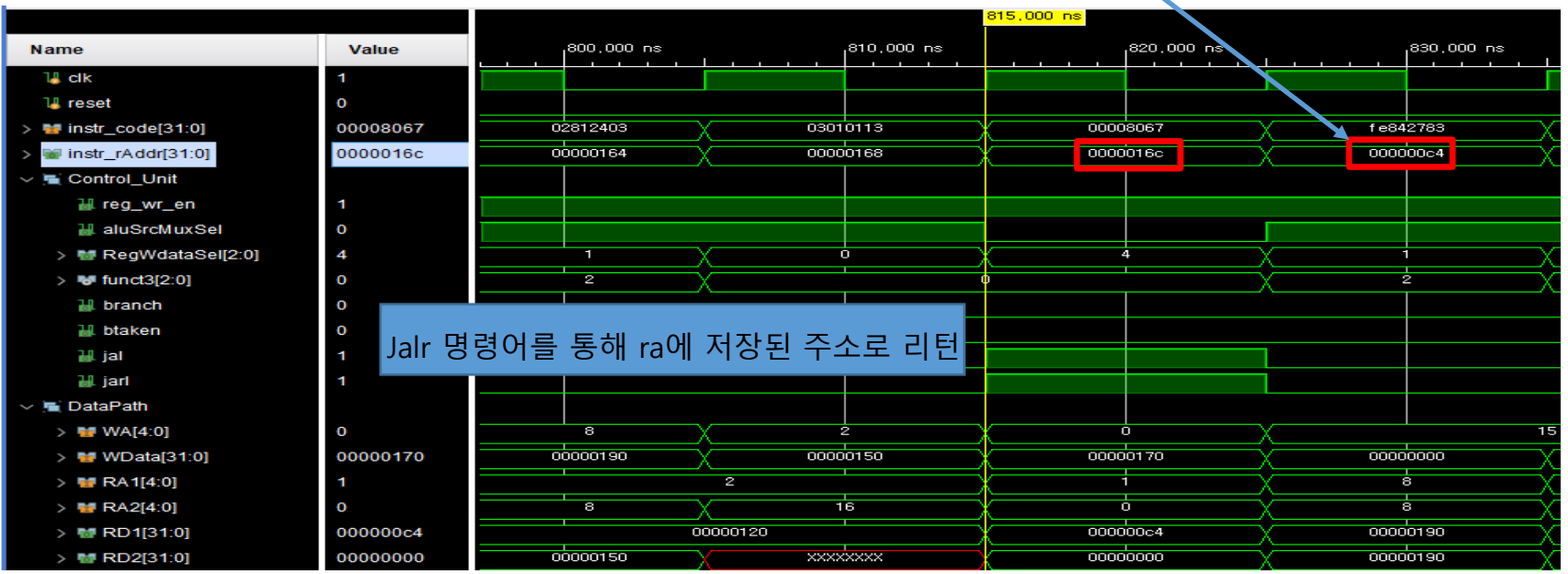
```
bc: 00068513            mv  a0,a3
c0: 05c000ef            jal ra,11c
c4: fe842783            lw  a5,-24(s0)

000000000000011c :
 11c: fd010113          addi  sp,sp,-48
 120: 02112623          sw  ra,44(sp)
 124: 02812423          sw  s0,40(sp)
 128: 03010413          addi  s0,sp,48
 12c: fca42e23          sw  a0,-36(s0)
 130: fcb42c23          sw  a1,-40(s0)
 134: fdc42783          lw  a5,-36(s0)
 138: 0007a783          lw  a5,0(a5)
 13c: fef42623          sw  a5,-20(s0)
 140: fd842783          lw  a5,-40(s0)
 144: 0007a703          lw  a4,0(a5)
 148: fdc42783          lw  a5,-36(s0)
 14c: 00e7a023          sw  a4,0(a5)
 150: fd842783          lw  a5,-40(s0)
 154: fec42703          lw  a4,-20(s0)
 158: 00e7a023          sw  a4,0(a5)
 15c: 00000013          nop
 160: 02c12083          lw  ra,44(sp)
 164: 02812403          lw  s0,40(sp)
 168: 03010113          addi  sp,sp,48
 16c: 00008067          ret
```

`ret`	`jalr x0, x1, 0`	Return from subroutine

call, return Simulation




```
swap(&aNum[j], &aNum[j+1]);
bc: 00068513            mv  a0,a3
c0: 05c000ef            jal ra,11c
c4: fe842783            lw  a5,-24(s0)
```

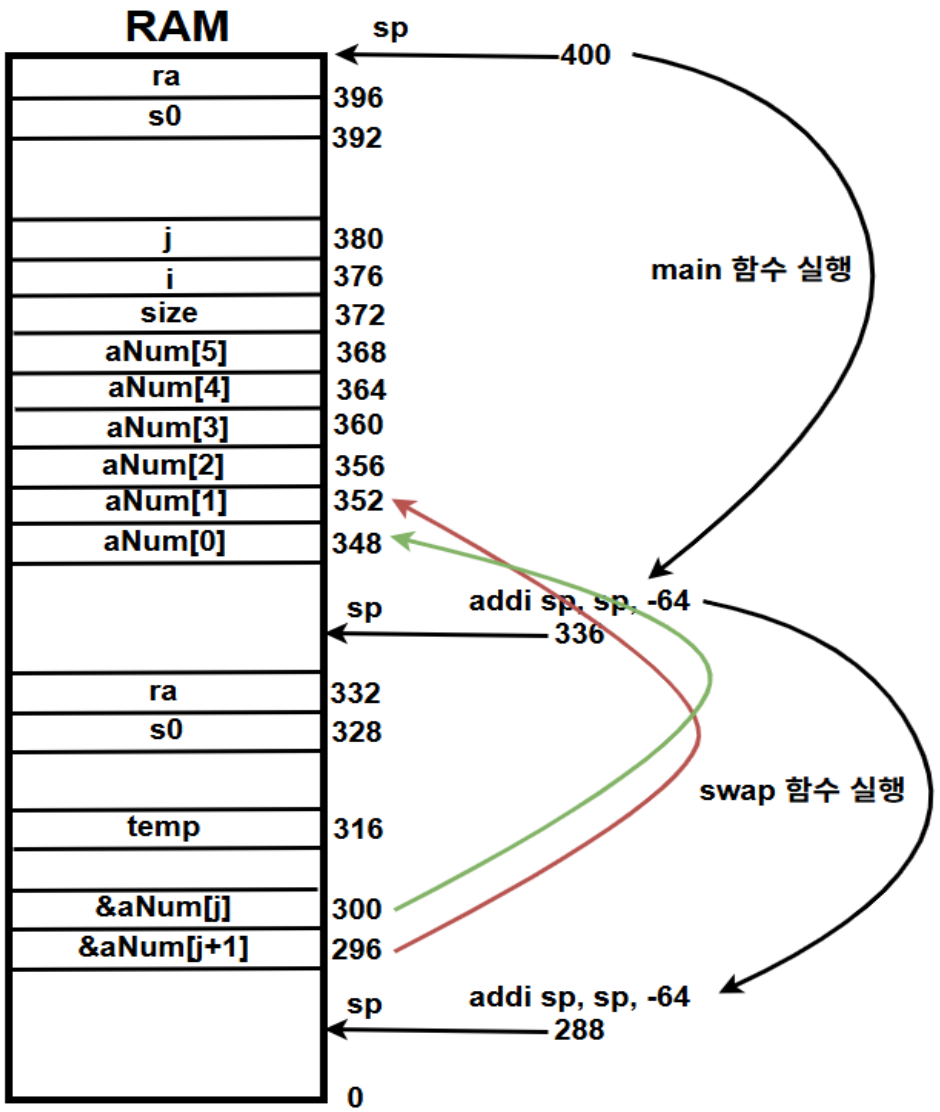



```
void swap(int *a, int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
    return ;
}
```

```
160: 02c12083           lw  ra,44(sp)
164: 02812403           lw  s0,40(sp)
168: 03010113           addi  sp,sp,48
16c: 00008067           ret
```

ret	jalr x0, x1, 0	Return from subroutine

포인터




```c
void swap(int *a, int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
    return ;
}
```

```
swap:
        addi    sp,sp,-48
        sw      ra,44(sp)
        sw      s0,40(sp)
        addi    s0,sp,48
        sw      a0,-36(s0)
        sw      a1,-40(s0)
        lw      a5,-36(s0)
        lw      a5,0(a5)
        sw      a5,-20(s0)
        lw      a5,-40(s0)
        lw      a4,0(a5)
        lw      a5,-36(s0)
        sw      a4,0(a5)
        lw      a5,-40(s0)
        lw      a4,-20(s0)
        sw      a4,0(a5)
        nop
        lw      ra,44(sp)
        lw      s0,40(sp)
        addi    sp,sp,48
        jr      ra
```

300번지에 aNum[j]의 주소 저장

296번지에 aNum[j+1]의 주소 저장

a5에 300번지에 있는 값(aNum[j]의 주소) load

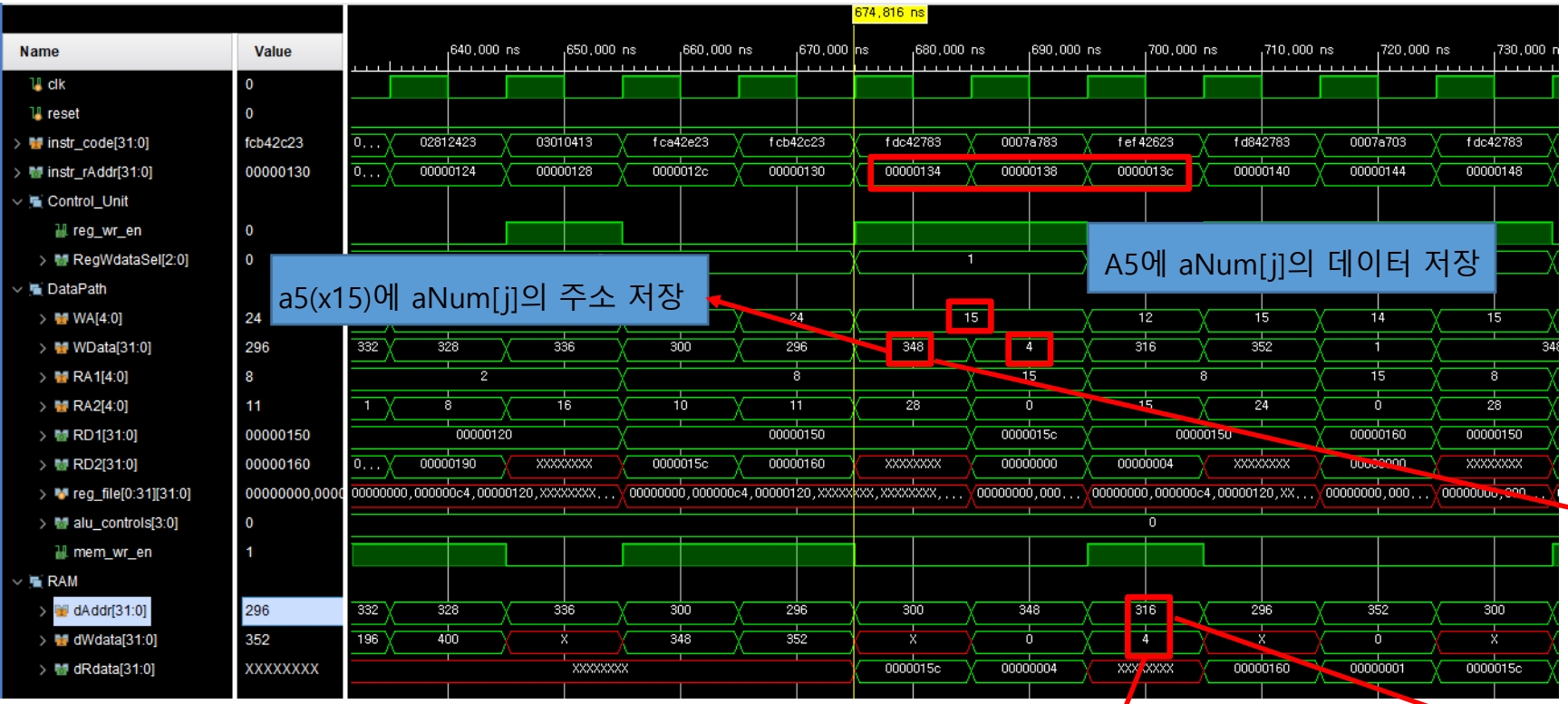
a5에 저장된 aNum[j]의 주소를 통해 a5에 aNum[j]의 값 load

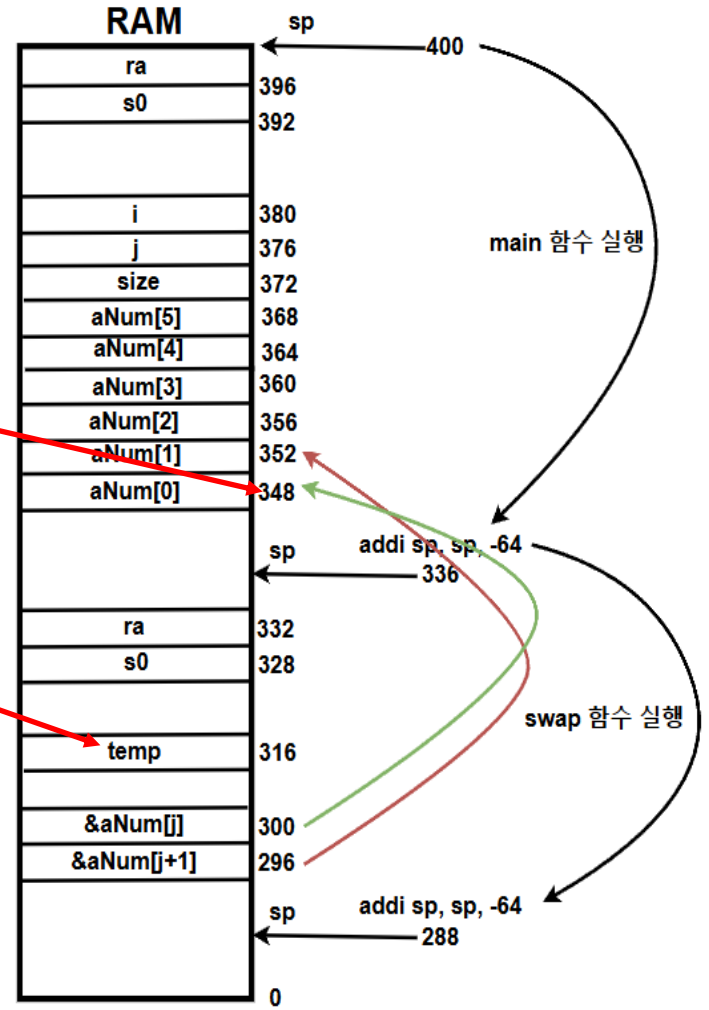
temp에 a5(aNum[j]의 데이터) 값 저장

포인터 Simulation

```
void swap(int *a, int *b){
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
    return ;
}
```

```
134: fdc42783    lw  a5,-36(s0)
138: 0007a783    lw  a5,0(a5)
13c: fef42623    sw  a5,-20(s0)
```



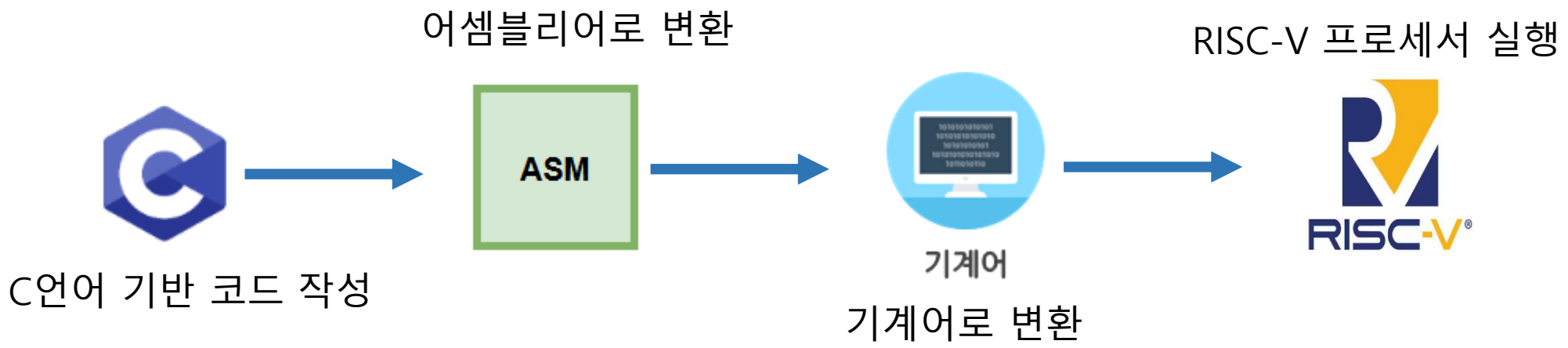


TroubleShooting

- 초기 RAM 설계에서는 32비트 전체를 주소 비트로 사용했습니다. 이로 인해 **sb, sh, lb, lh** 명령어 실행 시, 하위 바이트 영역에서만 접근이 가능해지는 비효율적인 문제가 발생했습니다.
- 이를 해결하기 위해 주소의 하위 2비트를 **byte_offset**으로 활용하고, 나머지 30비트를 실제 주소 비트로 사용하도록 구조를 수정했습니다. 이 방식으로 하위 비트뿐 아니라 상위 영역에서도 명령어가 정상적으로 동작하도록 개선할 수 있었습니다.
- 다만 주소 비트 수가 줄어들면서 접근 가능한 메모리 용량이 감소하는 부작용이 있었습니다. 따라서 실제 설계 환경에서 byte나 half 단위 접근이 빈번하지 않다면, **byte_offset** 비트를 생략하고 주소 폭을 유지하는 방식이 더 효율적일 것으로 판단됩니다.

Summary

- 이번 프로젝트에서는 C 코드가 하드웨어에서 실행되는 과정을 직접 구현하며 시스템 구조를 깊이 있게 이해할 수 있었습니다. 연산뿐만 아니라 메모리 접근과 분기, 점프 등 제어 흐름의 동작 원리를 설계 관점에서 체험했고, 명령어 실행에 따른 데이터 경로 변화를 SystemVerilog로 구현하면서 하드웨어 설계 능력을 향상시켰습니다. 이러한 과정은 향후 파이프라인 구조 설계로 확장하는 데 중요한 발판이 될 것이라고 생각합니다.



감사합니다.