S Y S T E M V E R I L O G

RISC-V CPU Design

하만 세미콘 아카데미 2기 김은성

2025.10.02

Contents

Introduction

Introduction

Project Goal

- **RISC-V**의 핵심 명령어 세트(ISA) 구조 이해 및 처리 로직 설계
- **RISC-V CPU 코어(Core)** 설계 및 구현
- 시뮬레이션을 통한 **명령어별 기능 검증** 및 CPU 동작 확인

Introduction

RISC-V

- **R**educed **I**nstruction **S**et **C**omputer
- 오픈 소스 명령어 집합 구조 (Instruction Set Architecture)
- 하드웨어 및 소프트웨어 설계에 유연성을 제공

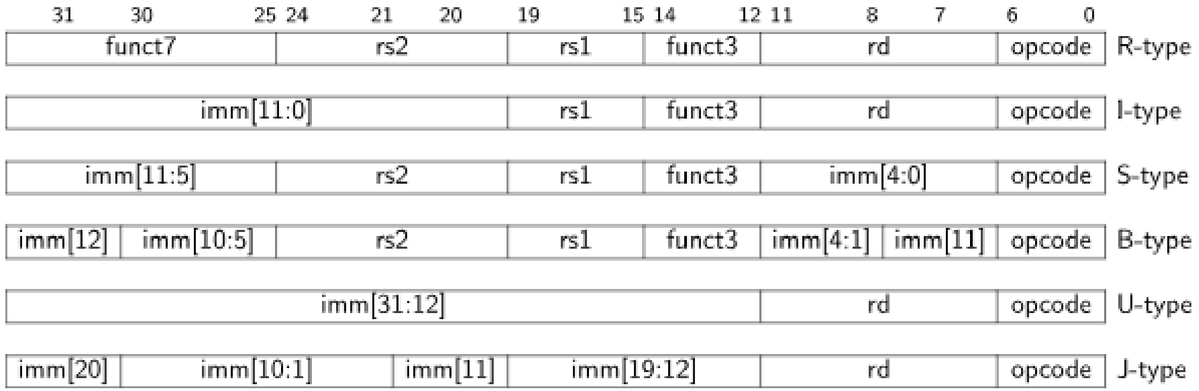
RV32I

- 'RV' **RISC-V**, '32' **32 bit** , 'I' **Integer** (정수)
- **RV32I Instruction Formats** :

- RISC-V의 기본 명령어 집합
- 필드들 (opcode, rd, funct3, rs1, rs2, funct7 등)이 특정 비트 범위를 차지

Introduction

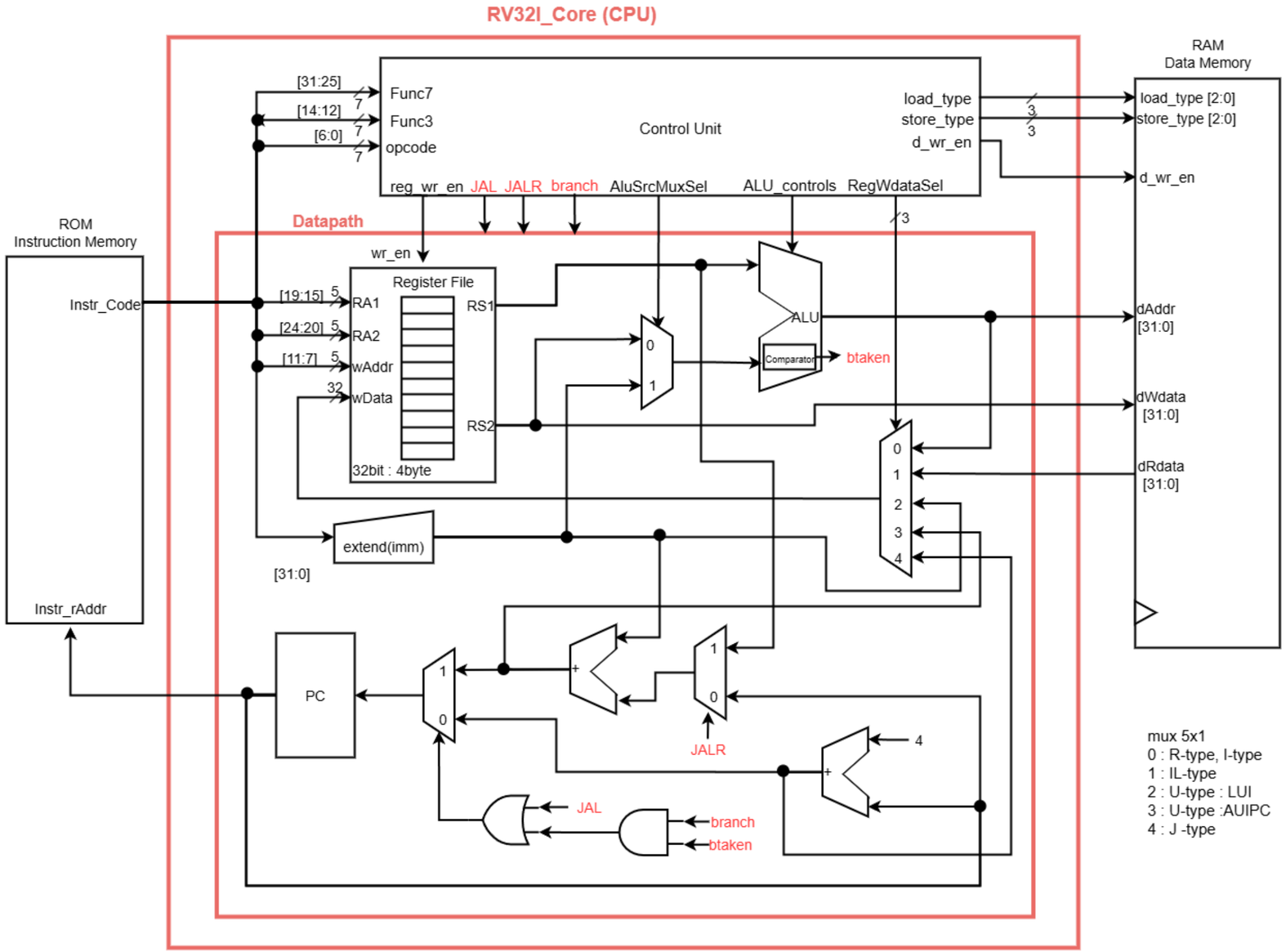
RV32I Instruction Formats



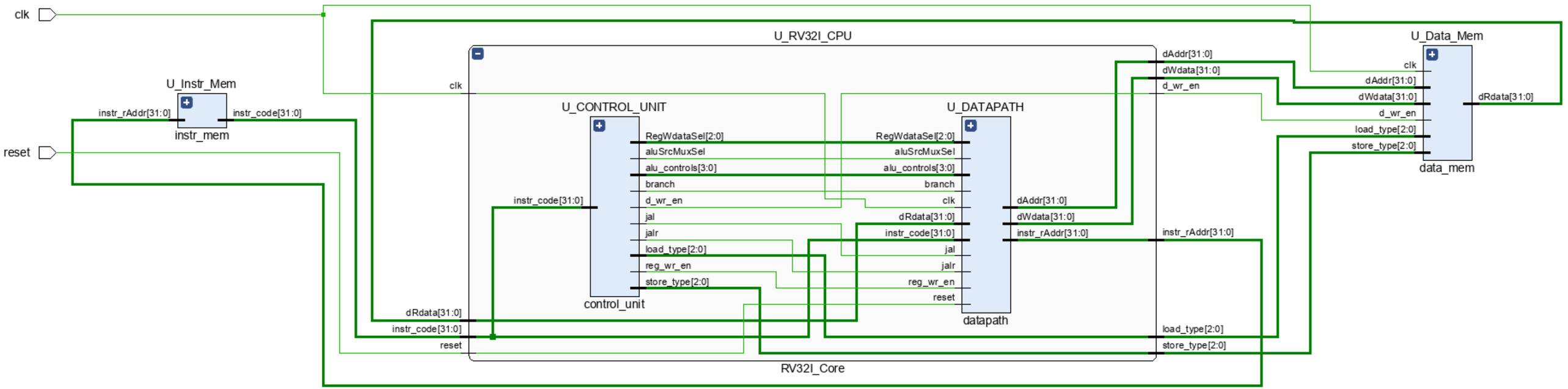
- **R-Type :** 연산자 중심 명령어 (ex. add, sub)
- **I-Type :** imm을 사용하는 명령어 (ex. addi, ori)
- **IL–Type** : 메모리에서 데이터를 불러오는 명령어 (ex. lw)
- **S-Type :** 메모리에 데이터를 저장하는 명령어 (ex. sw)
- **B-Type :** 조건 분기 명령어 (ex. beq, bne)
- **U-Type :** 상위 비트 로딩 명령어 (ex. lui)
- **J-Type :** 점프 명령어 (ex. jal)

Diagram-Based System Design

RV32I Core Block Diagram

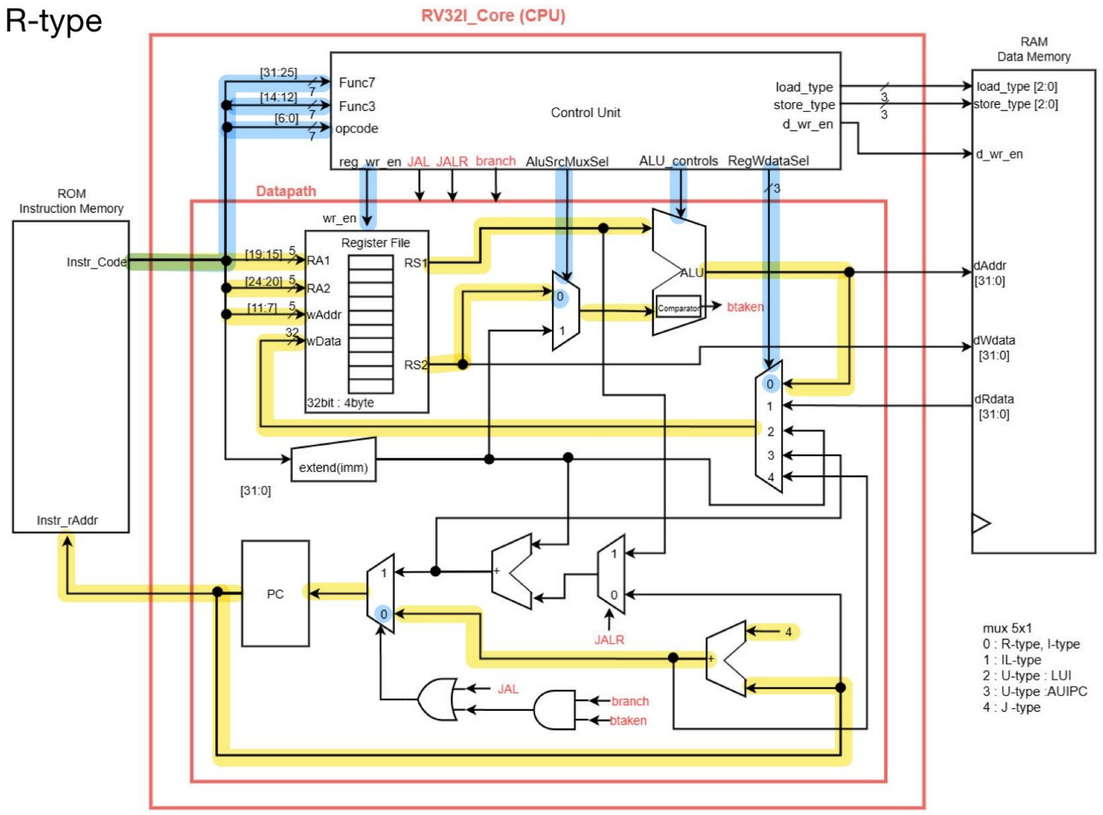


RV32I Core Schematic



Simulation

R – Type (ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND)



2개의 source register(rs1, rs2) 값을 사용하여 연산

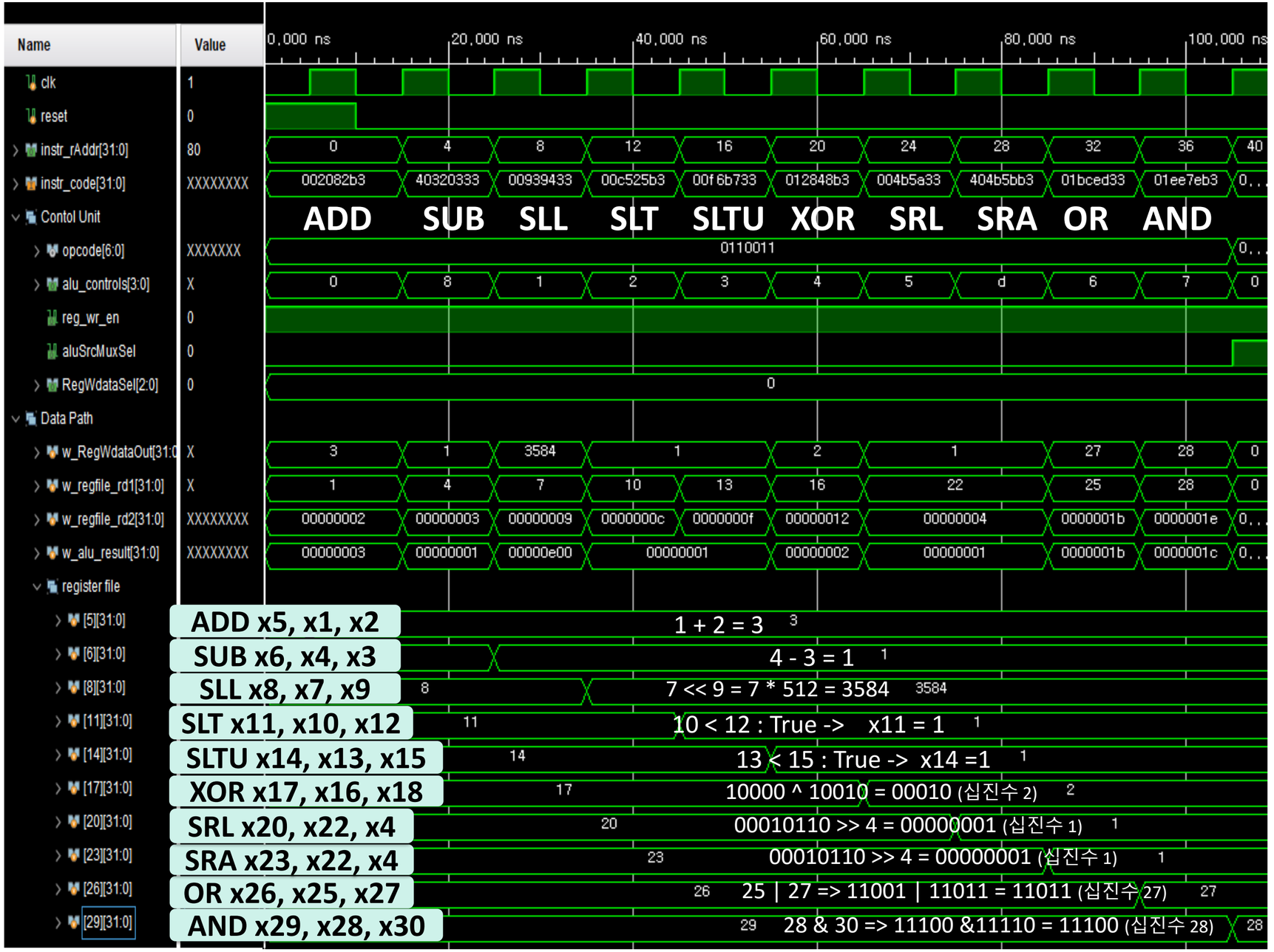
-> 연산 결과를 목적지 register (rd)에 저장

Type	Mnemonic	Descript	Extension
R	ADD	rd = rs1 + rs2	
	SUB	rd = rs1 - rs2	
	SLL	rd = rs1 << rs2	
	SLT	rd = (rs1 < rs2)?1:0	
	SLTU	rd = (rs1 < rs2)?1:0	zero-extends
	XOR	rd = rs1 ^ rs2	
	SRL	rd = rs1 >> rs2	
	SRA	rd = rs1 >> rs2	msb-extends
	OR	rd = rs1 | rs2	
	AND	rd = rs1 & rs2	

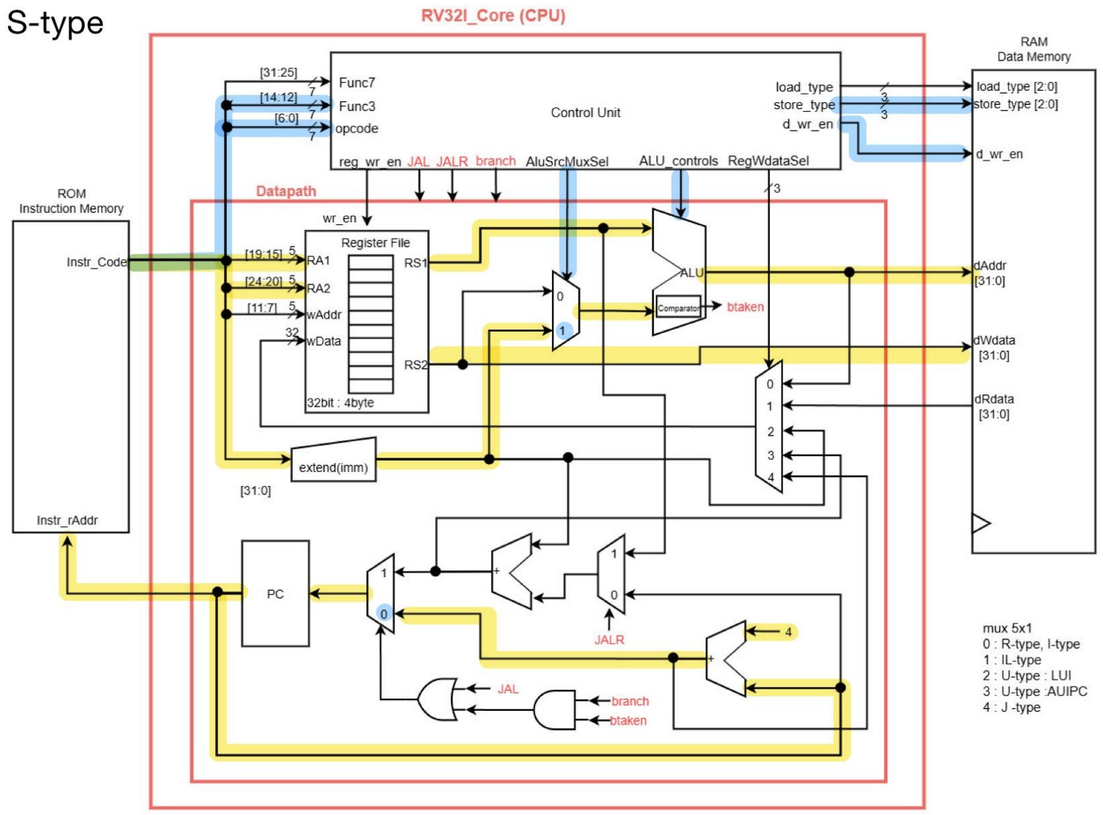
R – Type Simulation

Type	Mnemonic	Descript	Extension
R	ADD	rd = rs1 + rs2	
	SUB	rd = rs1 - rs2	
	SLL	rd = rs1 << rs2	
	SLT	rd = (rs1 < rs2)?1:0	
	SLTU	rd = (rs1 < rs2)?1:0	zero-extends
	XOR	rd = rs1 ^ rs2	
	SRL	rd = rs1 >> rs2	
	SRA	rd = rs1 >> rs2	msb-extends
	OR	rd = rs1 | rs2	
	AND	rd = rs1 & rs2	

```
rom[0]  = 32'h002082B3;   // add  x5, x1, x2
rom[1]  = 32'h40320333;   // sub  x6, x4, x3
rom[2]  = 32'h00939433;   // sll  x8, x7, x9
rom[3]  = 32'h00C525B3;   // slt  x11, x10, x12
rom[4]  = 32'h00F6B733;   // sltu x14, x13, x15
rom[5]  = 32'h012848B3;   // xor  x17, x16, x18
rom[6]  = 32'h004B5A33;   // srl  x20, x22, x4
rom[7]  = 32'h404B5BB3;   // sra  x23, x22, x4
rom[8]  = 32'h01BCED33;   // or   x26, x25, x27
rom[9]  = 32'h01EE7EB3;   // and  x29, x28, x30
```



S – Type (SB, SH, SW)



특정 레지스터(rs2)에 있는 데이터를
메모리에 저장(Store)

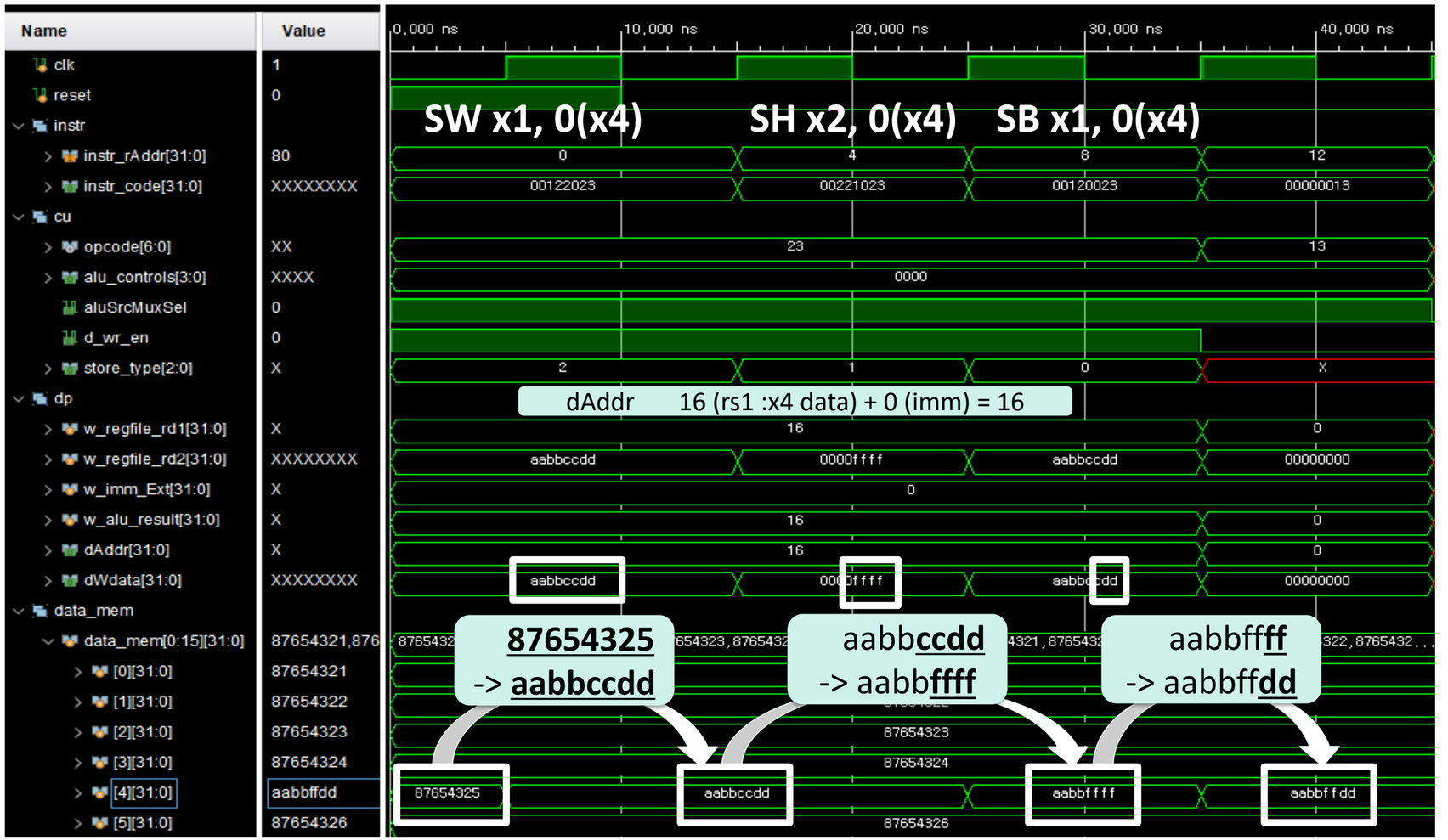
Type	Mnemonic	Descript
S	SB	M[rs1+imm][0:7] = rs2[0:7]
	SH	M[rs1+imm][0:15] = rs2[0:15]
	SW	M[rs1+imm][0:31] = rs2[0:31]

S – Type Simulation

Type	Mnemonic	Descript
S	SB	M[rs1+imm][0:7] = rs2[0:7]
	SH	M[rs1+imm][0:15] = rs2[0:15]
	SW	M[rs1+imm][0:31] = rs2[0:31]

```
// x1 레지스터 : sw/sb 쓰기 데이터
reg_file[1] = 32'hAABBCCDD;
// x2 레지스터: sh 쓰기 데이터
reg_file[2] = 32'h0000FFFF;
// x4 레지스터 : 베이스 주소 (0(x4)에서 주소 16)
reg_file[4] = 32'h00000010;
```

```
// Store Word AABBCCDD
rom[0] = 32'h00122023;   // sw x1, 0(x4)
// Store Half-word FFFF, overwriting CCDD
rom[1] = 32'h00221023;   // sh x2, 0(x4)
// Store Byte DD, overwriting one FF
rom[2] = 32'h00120023;   // sb x1, 0(x4)
rom[3] = 32'h00000013;   // addi x0, x0, 0 (NOP)
```



Store Word

RAM에
Wdata 저장

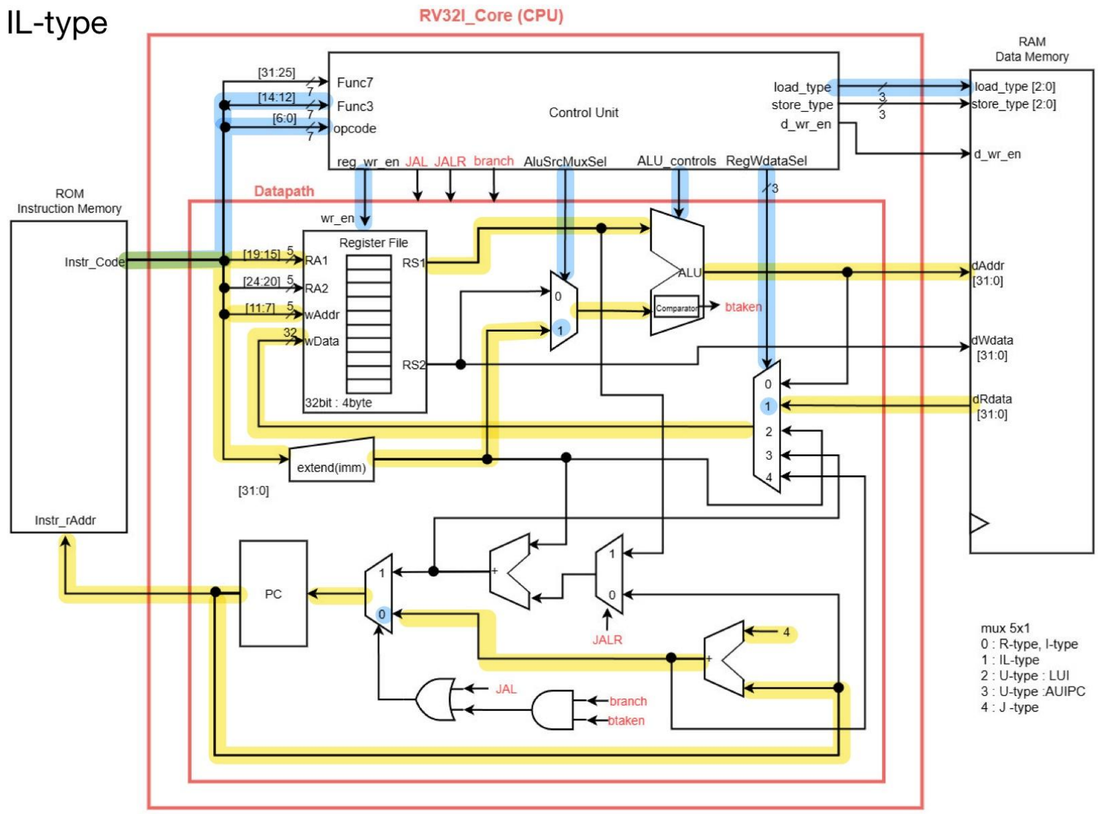
Store Half-Word

RAM에
Wdata 16bit저장

Store Byte

RAM에
Wdata 8bit저장

IL – Type (LB, LH, LW, LBU, LHU)



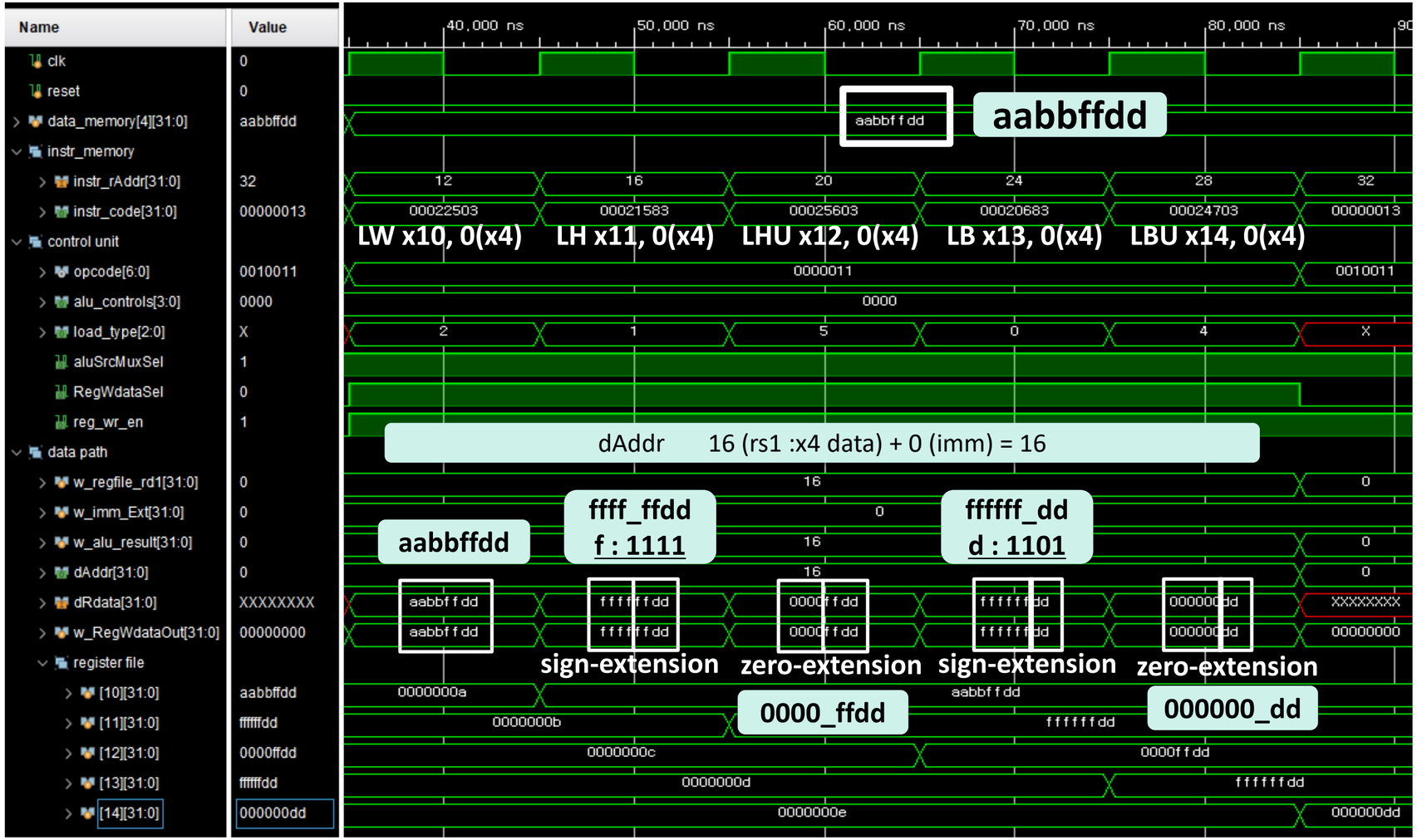
베이스 주소(rs1)에 12비트 offset을 더하여
최종 메모리 주소를 계산
-> 해당 주소에서 데이터를 읽어와서
목적지 레지스터(rd)에 로드(Load)

Type	Mnemonic	Descript	Extension
IL	LB	rd = M[rs1+imm][0:7]	
	LH	rd = M[rs1+imm][0:15]	
	LW	rd = M[rs1+imm][0:31]	
	LBU	rd = M[rs1+imm][0:7]	zero-extends
	LHU	rd = M[rs1+imm][0:15]	zero-extends

IL – Type Simulation

Type	Mnemonic	Descript	Extension
IL	LB	rd = M[rs1+imm][0:7]	
	LH	rd = M[rs1+imm][0:15]	
	LW	rd = M[rs1+imm][0:31]	
	LBU	rd = M[rs1+imm][0:7]	zero-extends
	LHU	rd = M[rs1+imm][0:15]	zero-extends

```
// Load Word                    Expected x10 = AABBFFDD
rom[3] = 32'h00022503;          // lw  x10, 0(x4)
// Load Half-word (signed)      Expected x11 = FFFFFFDD
rom[4] = 32'h00021583;          // lh  x11, 0(x4)
// Load Half-word (unsigned)    Expected x12 = 0000FFDD
rom[5] = 32'h00025603;          // lhu x12, 0(x4)
// Load Byte (signed)           Expected x13 = FFFFFFDD
rom[6] = 32'h00020683;          // lb  x13, 0(x4)
// Load Byte (unsigned)         Expected x14 = 000000DD
rom[7] = 32'h00024703;          // lbu x14, 0(x4)
```



Load Word

32bit data load

Load Half-Word

하위 16bit load
+ sign-extension

Load Half-Word Unsigned

하위 16bit load
+ zero-extension

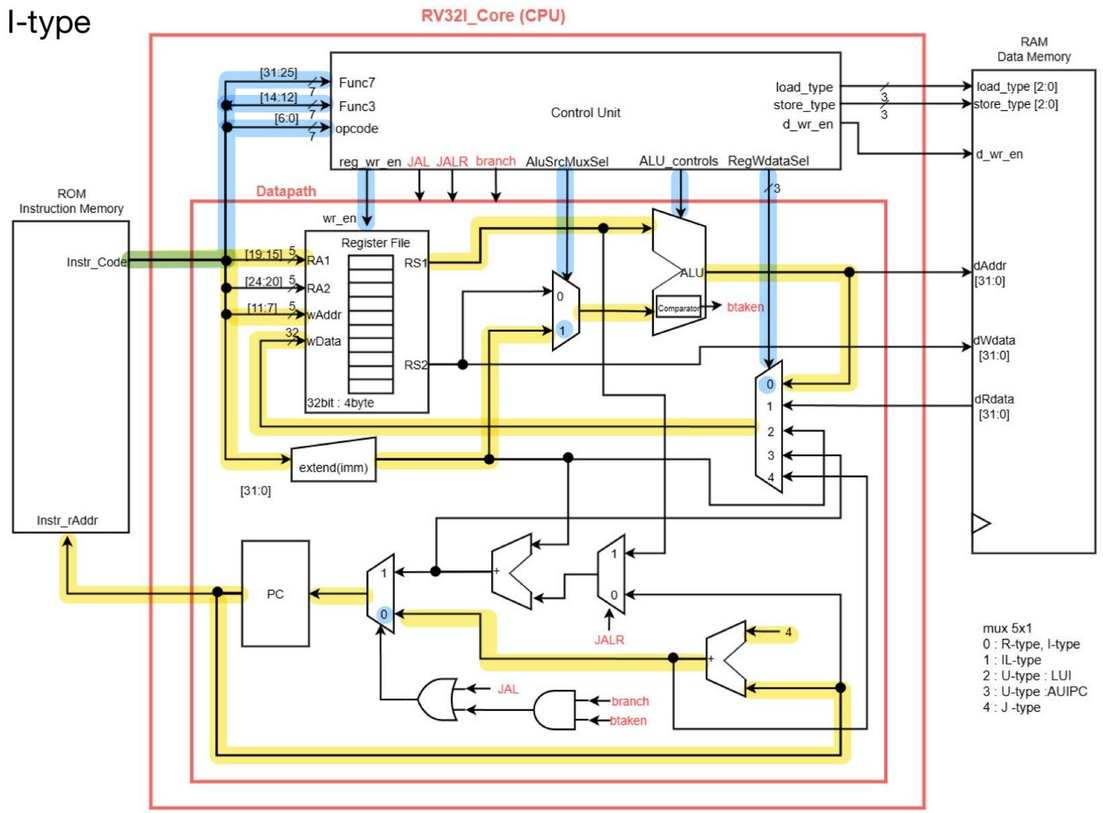
Load Byte

하위 8bit load
+ sign-extension

Load Byte Unsigned

하위 8bit load
+ zero-extension

I – Type (ADDI, SLTI, SLTIU, XORI, ORI, ANDI, SLLI, SRLI, SRAI)



source register(rs1)와 12비트 immediate를 사용하여 산술/논리 연산을 수행

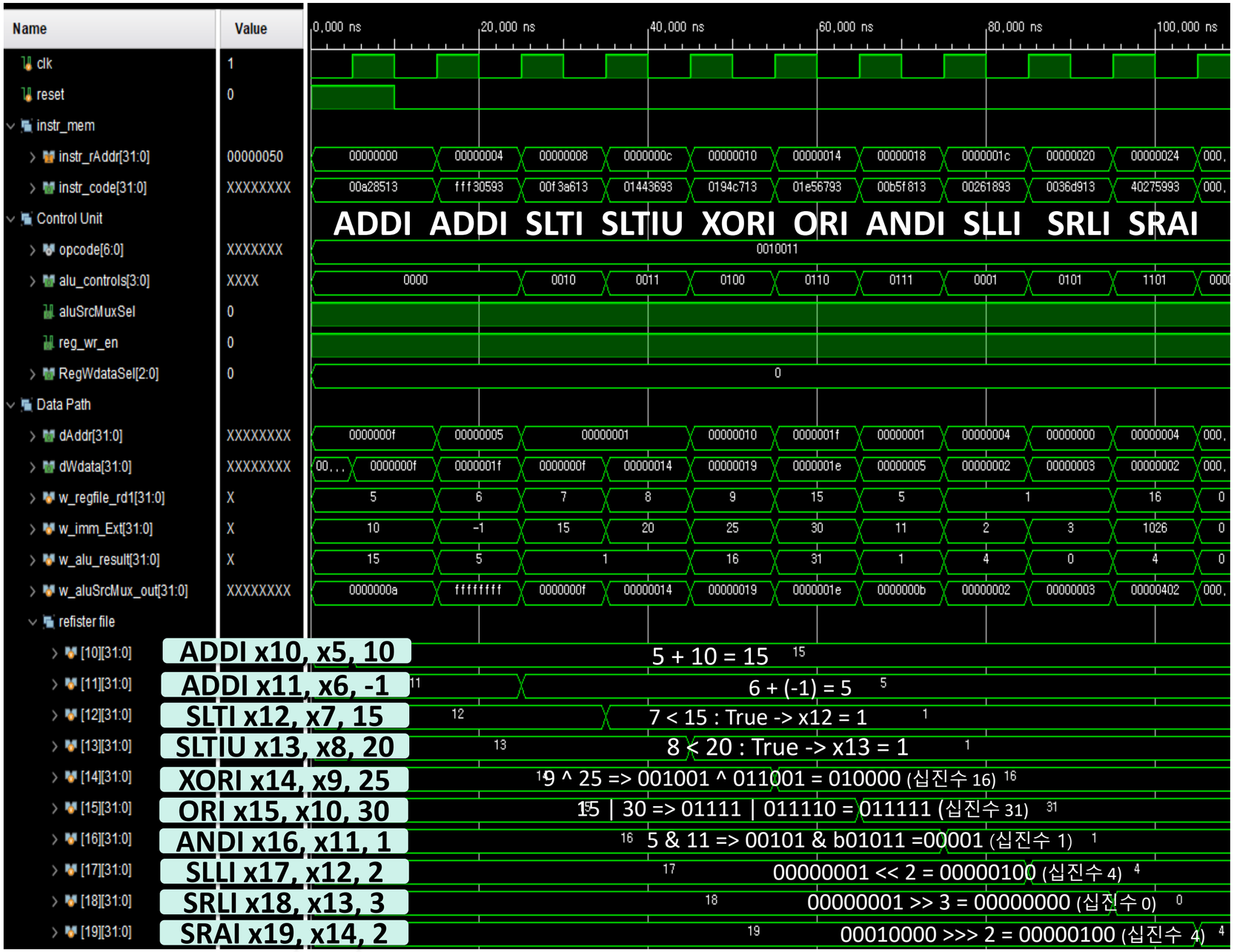
-> 결과를 목적지 register (rd)에 저장

Type	Mnemonic	Descript	Extension
I	ADDI	rd = rs1 + imm	
	SLTI	rd = (rs1 < imm)?1:0	
	SLTIU	rd = (rs1 < imm)?1:0	zero-extends
	XORI	rd = rs1 ^ imm	
	ORI	rd = rs1 | imm	
	ANDI	rd = rs1 & imm	
	SLLI	rd = rs1 << imm[0:4]	
	SRLI	rd = rs1 >> imm[0:4]	
	SRAI	rd = rs1 >> imm[0:4]	msb-extends

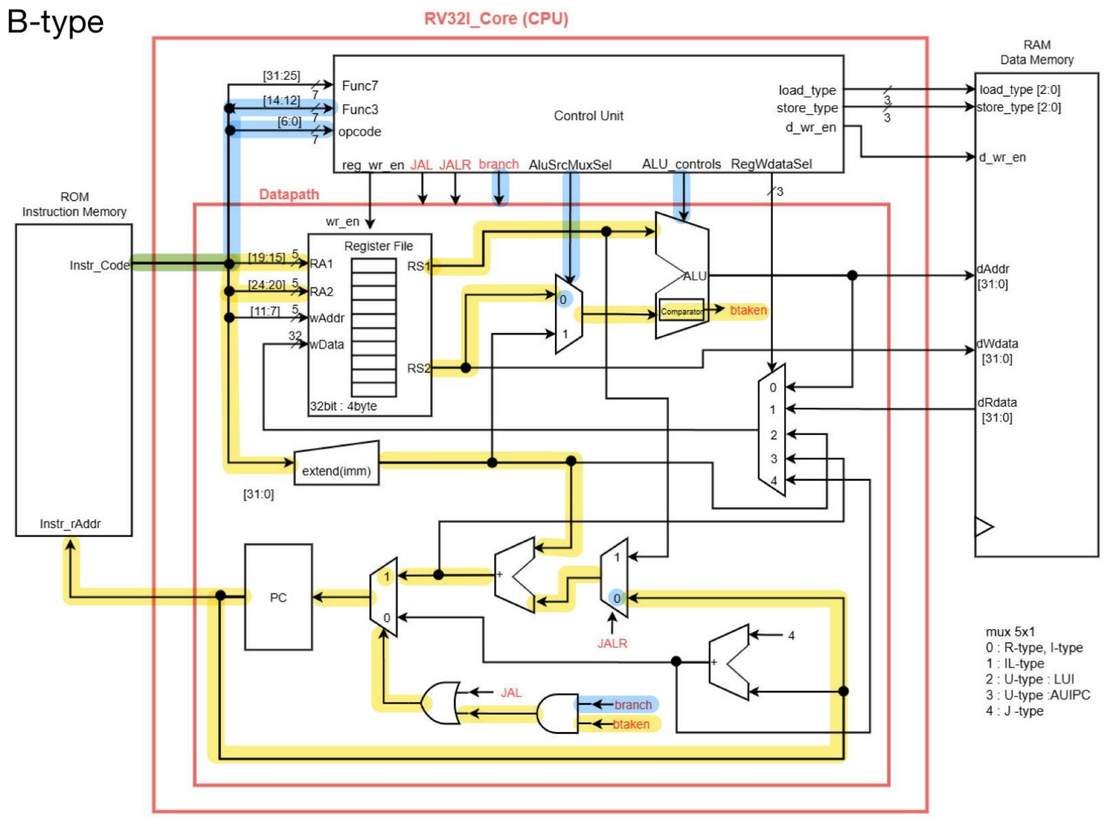
I – Type Simulation

Type	Mnemonic	Descript	Extension
I	ADDI	rd = rs1 + imm	
	SLTI	rd = (rs1 < imm)?1:0	
	SLTIU	rd = (rs1 < imm)?1:0	zero-extends
	XORI	rd = rs1 ^ imm	
	ORI	rd = rs1 | imm	
	ANDI	rd = rs1 & imm	
	SLLI	rd = rs1 << imm[0:4]	
	SRLI	rd = rs1 >> imm[0:4]	
	SRAI	rd = rs1 >> imm[0:4]	msb-extends

```
rom[0] = 32'h00A28513;   // addi  x10, x5, 10
rom[1] = 32'hFFF30593;   // addi  x11, x6, -1
rom[2] = 32'h00F3A613;   // slti  x12, x7, 15
rom[3] = 32'h01443693;   // sltiu x13, x8, 20
rom[4] = 32'h0194C713;   // xori  x14, x9, 25
rom[5] = 32'h01E56793;   // ori   x15, x10, 30
rom[6] = 32'h00B5F813;   // andi  x16, x11, 11
rom[7] = 32'h00261893;   // slli  x17, x12, 2
rom[8] = 32'h0036D913;   // srli  x18, x13, 3
rom[9] = 32'h40275993;   // srai  x19, x14, 2
```



B – Type (BEQ, BNE, BLT, BGE, BLTU, BGEU)



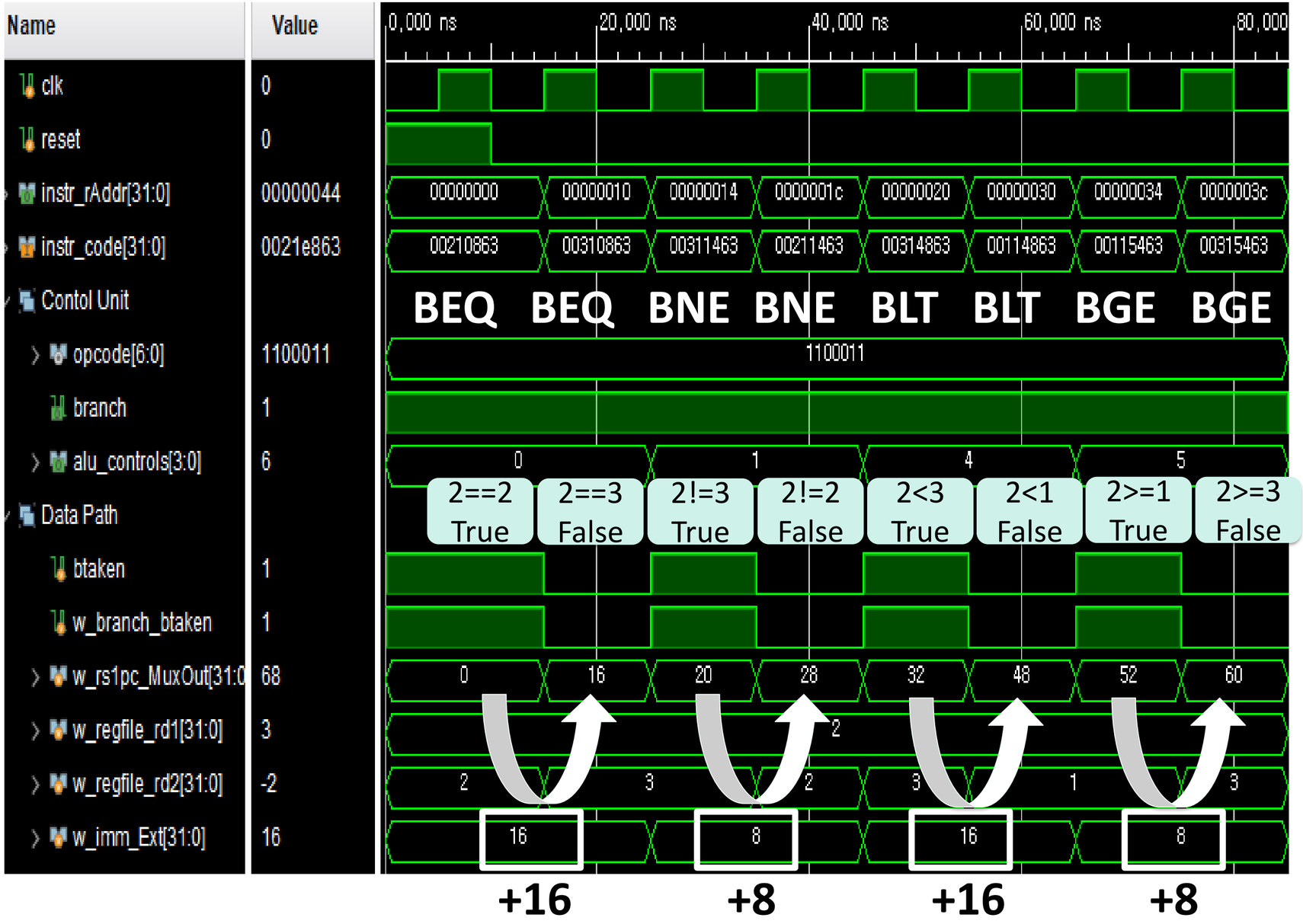
2개의 source register(rs1, rs2)를 비교하여
특정 조건이 만족되면
프로그램 카운터(PC)를 변경해
특정 주소로 분기

Type	Mnemonic	Descript	Extension
B	BEQ	if(rs1 == rs2) PC += imm	
	BNE	if(rs1 != rs2) PC += imm	
	BLT	if(rs1 < rs2) PC += imm	
	BGE	if(rs1 >= rs2) PC += imm	
	BLTU	if(rs1 < rs2) PC += imm	zero-extends
	BGEU	if(rs1 >= rs2) PC += imm	zero-extends

B – Type Simulation

Type	Mnemonic	Descript	Extension
B	BEQ	if(rs1 == rs2) PC += imm	
	BNE	if(rs1 != rs2) PC += imm	
	BLT	if(rs1 < rs2) PC += imm	
	BGE	if(rs1 >= rs2) PC += imm	
	BLTU	if(rs1 < rs2) PC += imm	zero-extends
	BGEU	if(rs1 >= rs2) PC += imm	zero-extends

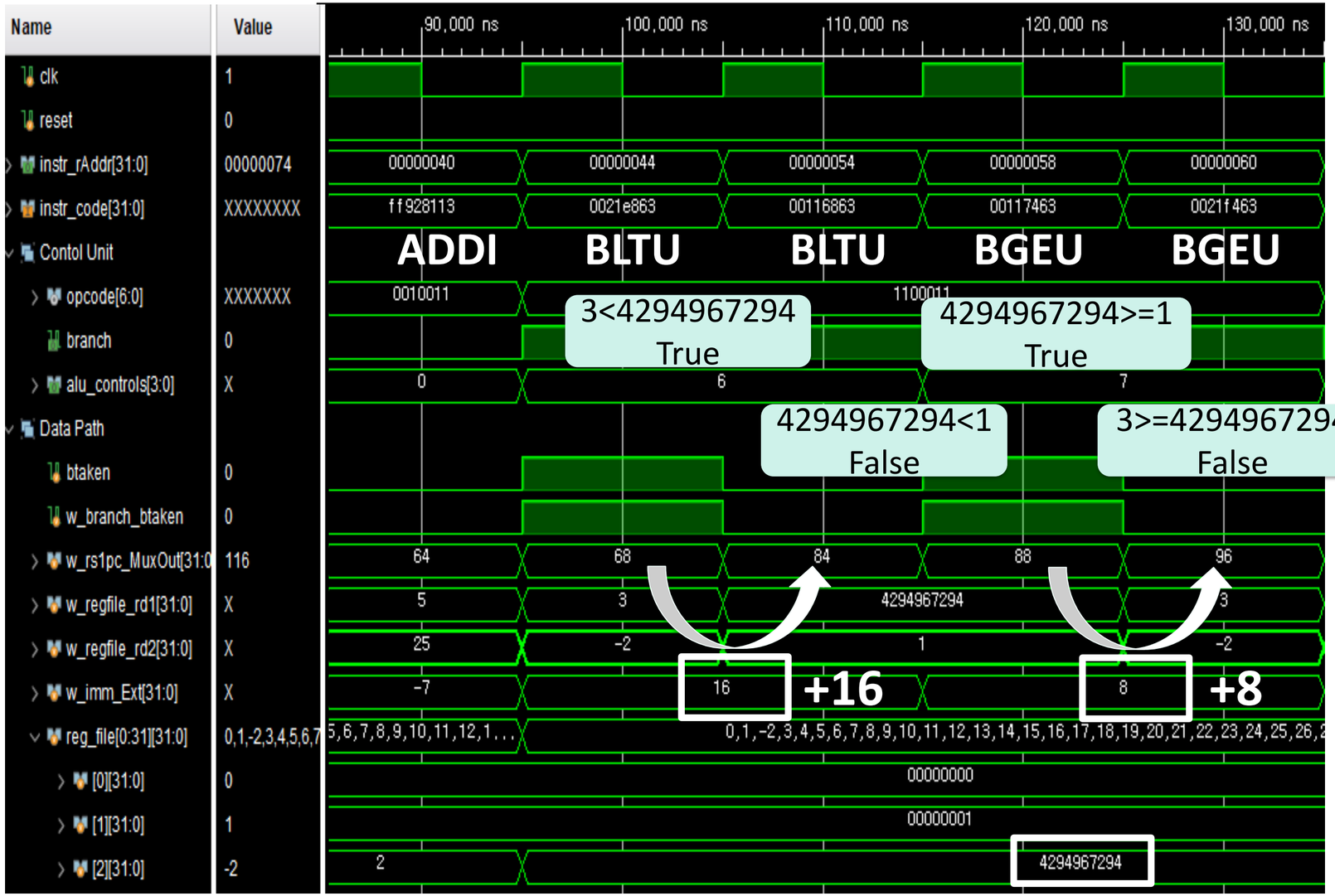
```
rom[0] = 32'h00_21_08_63;    // BEQ x2, x2, 16 t
rom[4] = 32'h00310863;       // BEQ x2, x3, 16 f
rom[5] = 32'h00311463;       // BNE x2, x3, 8 t
rom[7] = 32'h00211463;       // BNE x2, x2, 8 f
rom[8] = 32'h00314863;       // BLT x2, x3, 16 t
rom[12] = 32'h00114863;      // BLT x2, x1, 16 f
rom[13] = 32'h00115463;      // BGE x2, x1, 8 t
rom[15] = 32'h00315463;      // BGE x2, x3, 8 f
```



B – Type Simulation

Type	Mnemonic	Descript	Extension
B	BEQ	if(rs1 == rs2) PC += imm	
	BNE	if(rs1 != rs2) PC += imm	
	BLT	if(rs1 < rs2) PC += imm	
	BGE	if(rs1 >= rs2) PC += imm	
	BLTU	if(rs1 < rs2) PC += imm	zero-extends
	BGEU	if(rs1 >= rs2) PC += imm	zero-extends

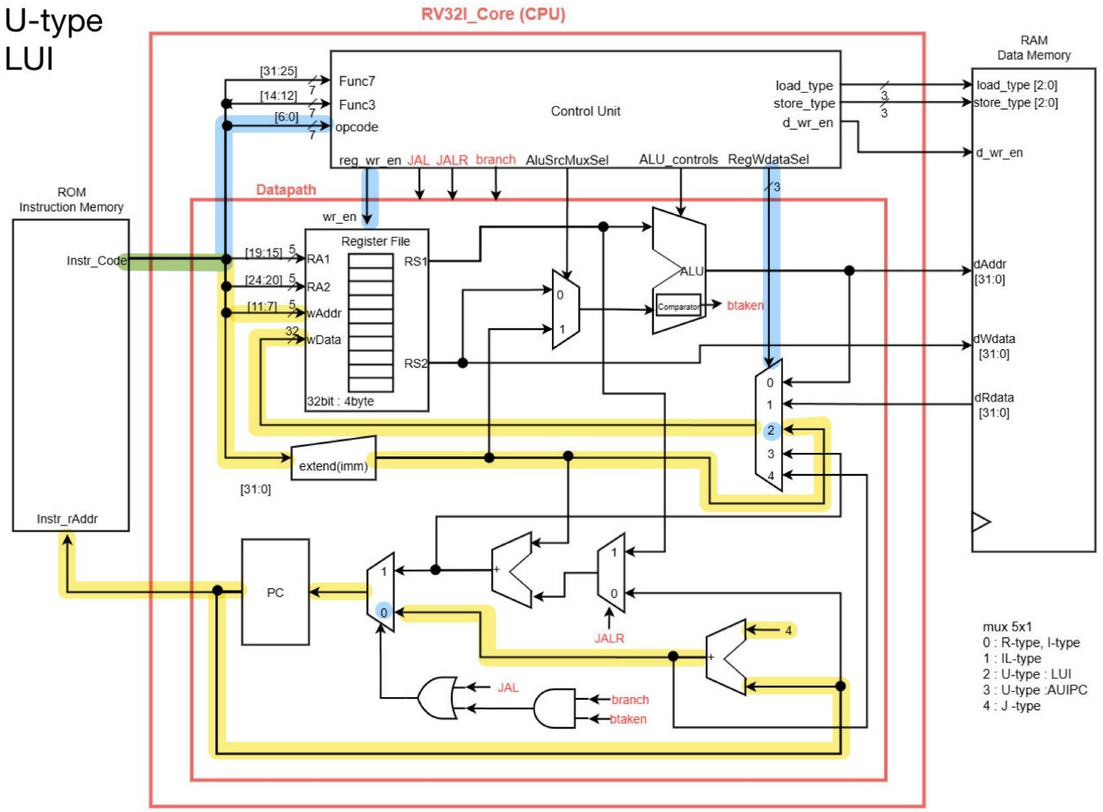
```
rom[16] = 32'hff928113;   // 음수 만들어주기 addi x2, x5, -7  => x2 : -2
rom[17] = 32'h0021e863;   // BLTU x3, x2, 16  => t
rom[21] = 32'h00116863;   // BLTU x2, x1, 16  => f
rom[22] = 32'h00117463;   // BGEU x2, x1, 8   => t
rom[24] = 32'h0021F463;   // BGEU x3, x2, 8   => f
```



-2(32bit signed) = 4294967294 (32bit unsigned)

U – Type (LUI, AUIPC)

U-type
LUI



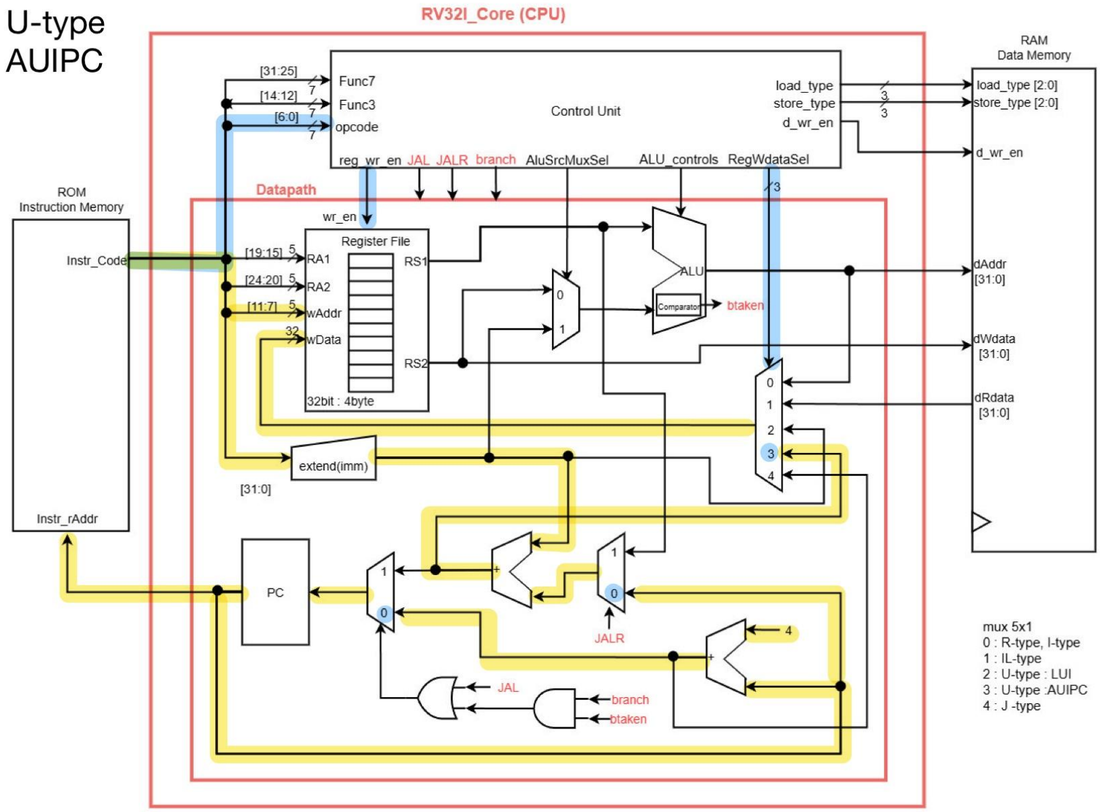
LUI(Load Upper Immediate)

20비트의 immediate를 register의 상위 비트(Upper bits)에 로드, 나머지 하위 12비트는 0으로 채우기

-> 그 결과를 목적지 register에 저장

Type	Mnemonic	Descript
U	LUI	rd = imm << 12
	AUIPC	rd = PC + (imm<<12)

U – Type (LUI, AUIPC)



LUI(Load Upper Immediate)

20비트의 immediate를 register의 상위 비트(Upper bits)에 로드, 나머지 하위 12비트는 0으로 채우기

AUIPC(Add Upper Immediate to PC)

immediate를 12비트 왼쪽 시프트 후
그 값을 PC 값에 더하기

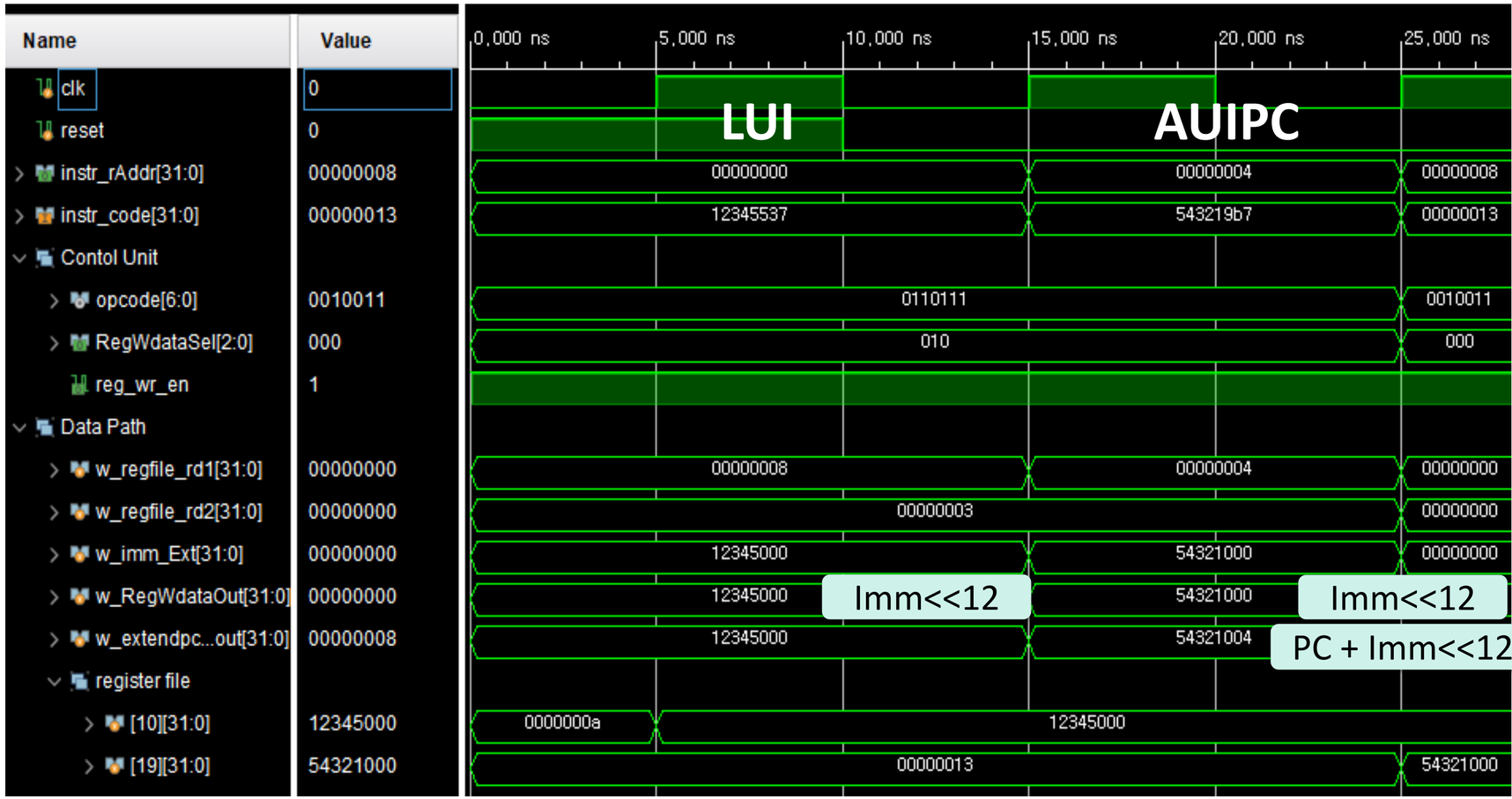
-> 그 결과를 목적지 register에 저장

Type	Mnemonic	Descript
U	LUI	rd = imm << 12
	AUIPC	rd = PC + (imm<<12)

U – Type Simulation

Type	Mnemonic	Descript
U	LUI	rd = imm << 12
	AUIPC	rd = PC + (imm<<12)

```
//U-type
rom[0] = 32'h12345537;  // lui x10, 0x12345
rom[1] = 32'h543219B7;  // auipc x19, 0x54321
```



lui x10, 0x12345

32’h12345537
-> 32’h12345000

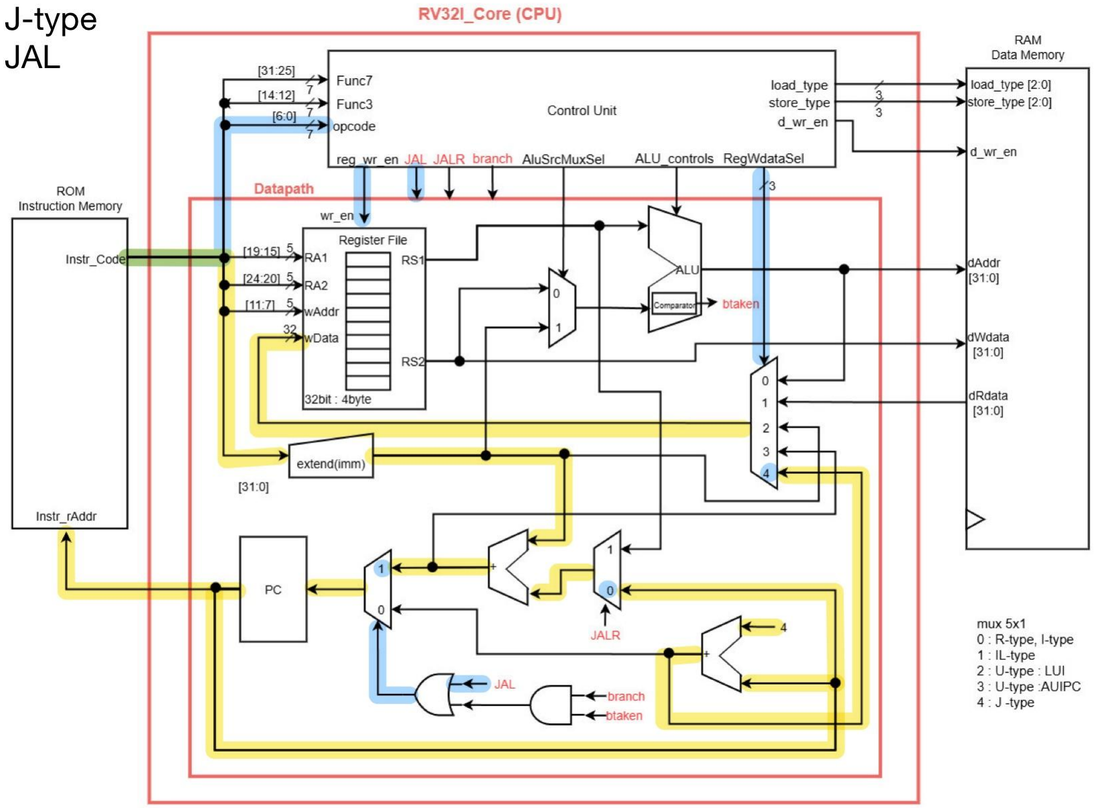
상위 20bit 그대로 넣고
하위 12bit 0으로 채우기

auipc x19, 0x54321

PC + (imm << 12)
= 0x4 + (0X54321 << 12)
= 0x4 + 0x54321000
= 0x54321004

J/JI – Type (JAL, JALR)

J-type
JAL



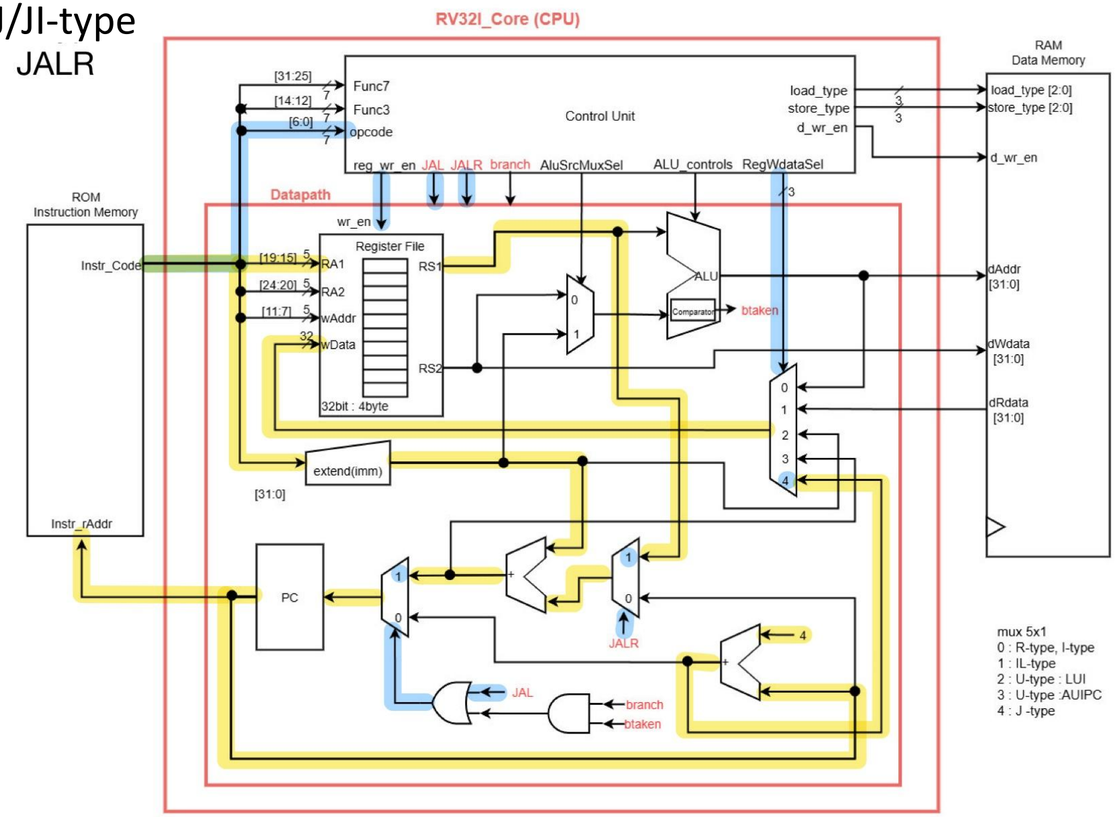
JAL (Jump and Link)

현재 PC에 immediate 오프셋을 더한 주소로 점프하며, 복귀 주소(PC+4)를 rd에 저장

Type	Mnemonic	Descript
J/JI	JAL	rd = PC+4; PC += imm
	JALR	rd = PC+4; PC = rs1 + imm

J/JI – Type (JAL, JALR)

J/JI-type
JALR



JAL (Jump and Link)

현재 PC에 immediate 오프셋을 더한 주소로 점프하며, 복귀 주소(PC+4)를 rd에 저장

JALR (Jump and Link Register)

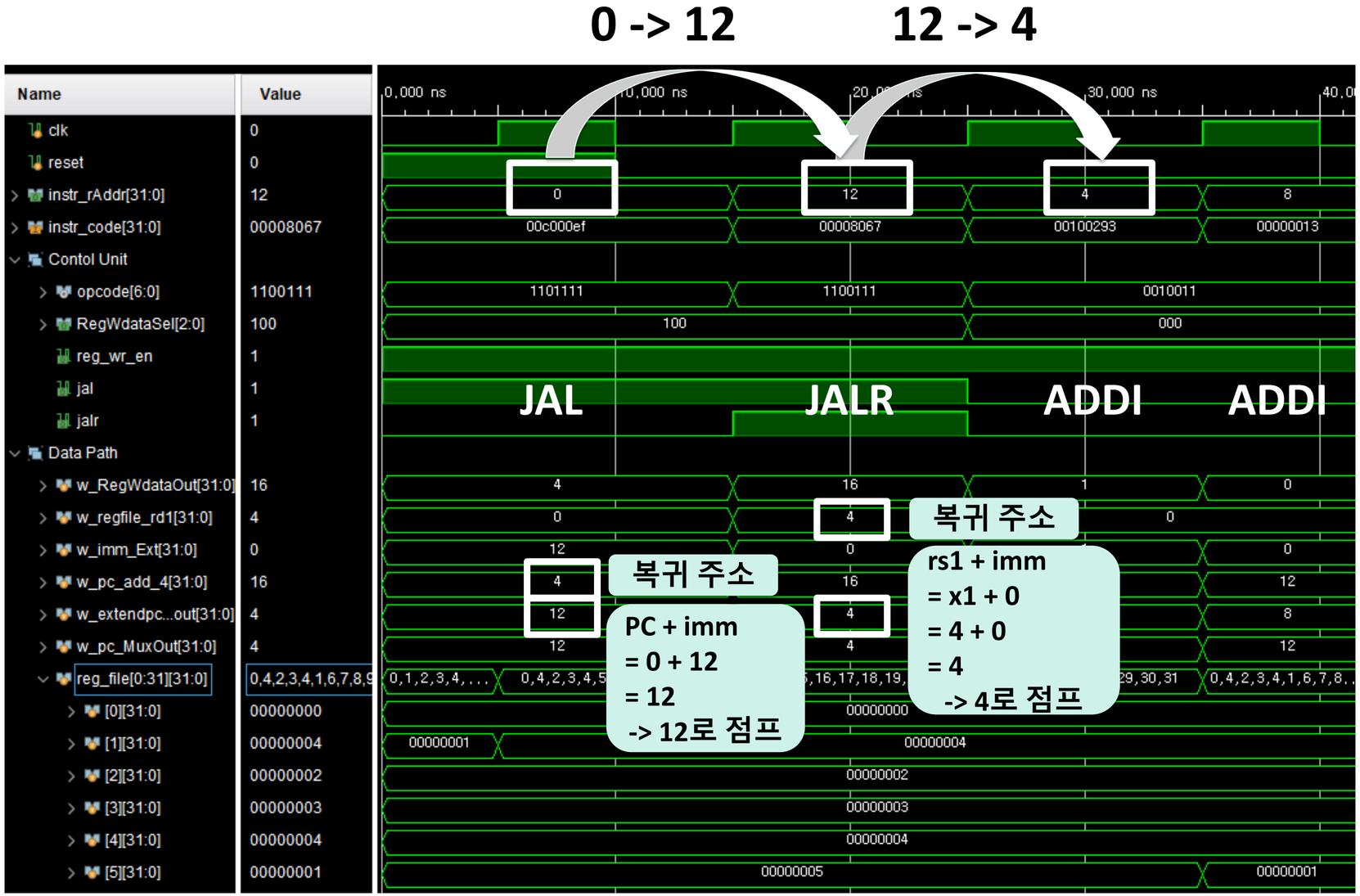
rs1 레지스터 값에 immediate 오프셋을 더한 주소로 점프하며, 복귀 주소(PC+4)를 rd에 저장

Type	Mnemonic	Descript
J/JI	JAL	rd = PC+4; PC += imm
	JALR	rd = PC+4; PC = rs1 + imm

J/JI – Type Simulation

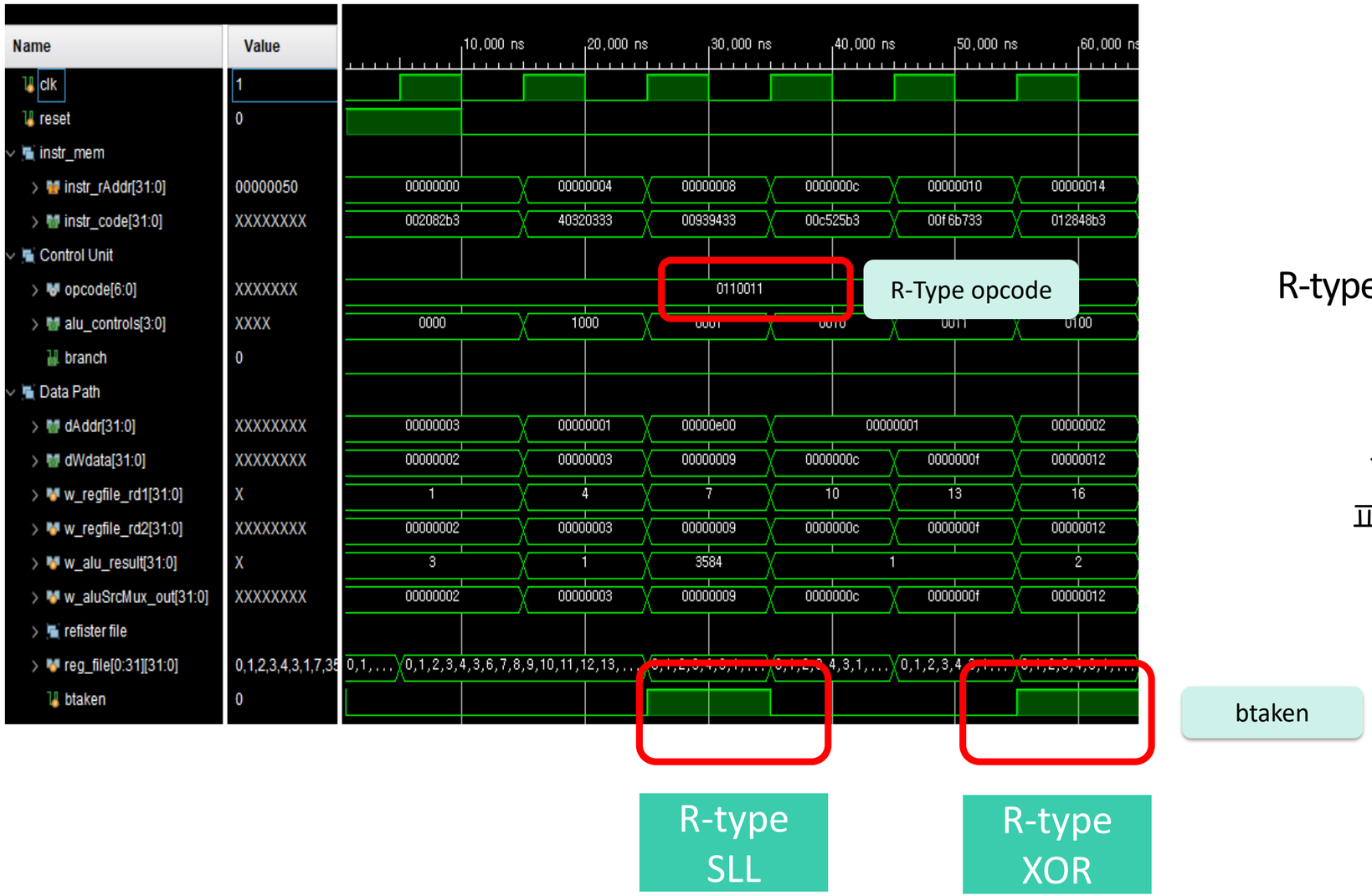
Type	Mnemonic	Descript
J/JI	JAL	rd = PC+4; PC += imm
	JALR	rd = PC+4; PC = rs1 + imm

```
rom[0] = 32'h00C000EF;   // jal x1, 12
rom[1] = 32'h00100293;   // addi x5, x0, 1
rom[2] = 32'h00000013;   // addi x0, x0, 0
rom[3] = 32'h00008067;   // jalr x0, x1, 0
```



Trouble Shooting

Trouble Shooting – R-type btaken 발생



R-type SLL, XOR 에서 btaken 신호 : 1

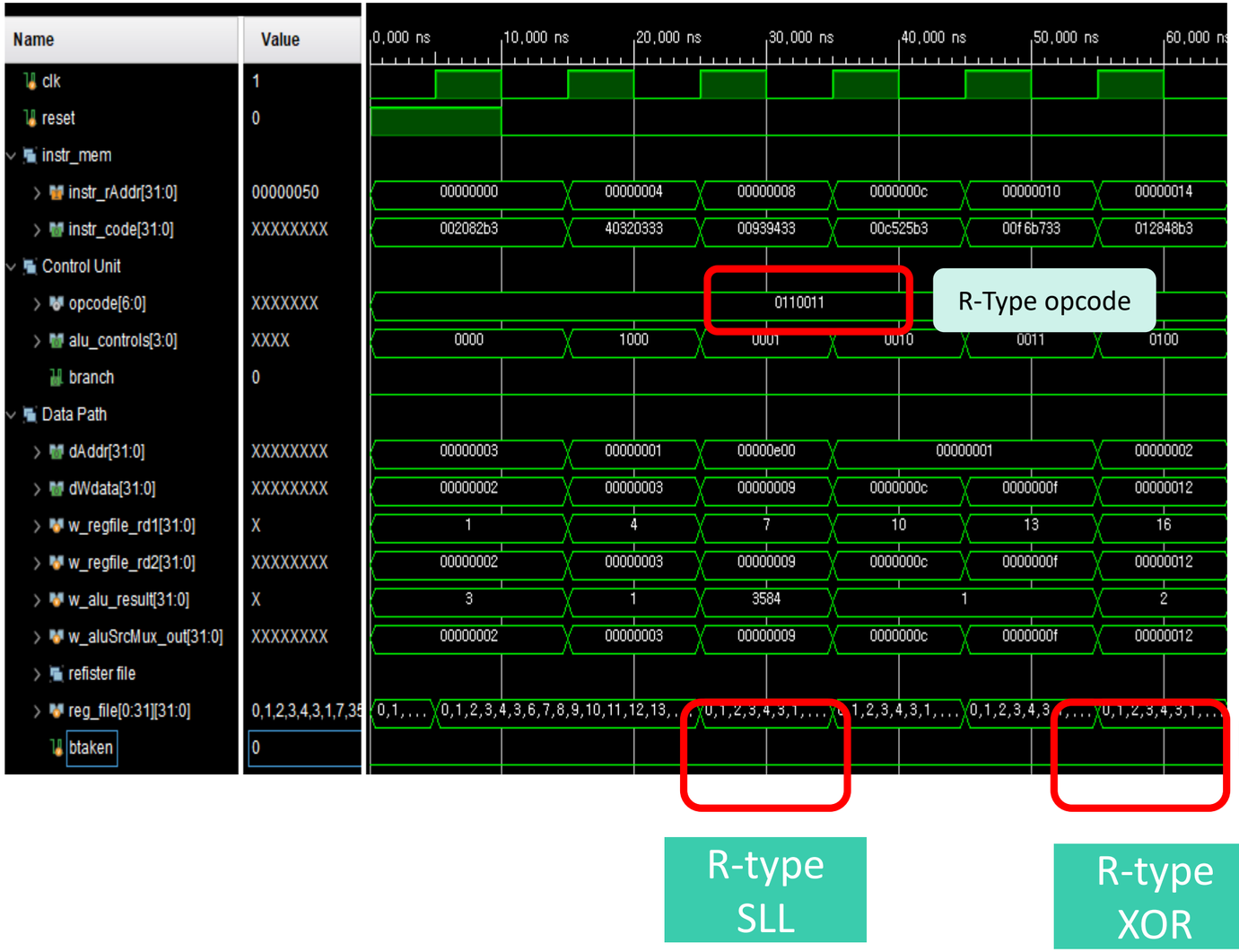
-> 불필요한 신호 변화

-> 파이프라인으로 확장 시,

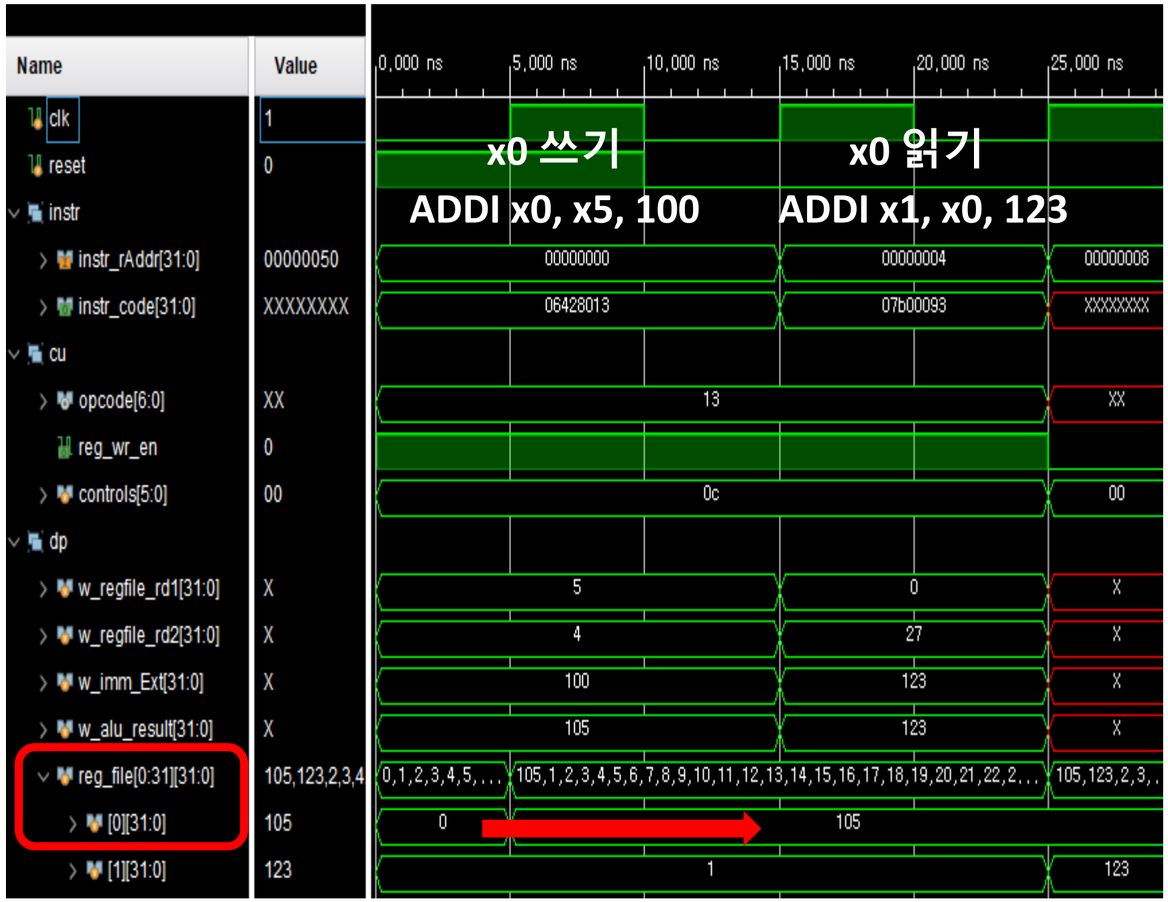
파이프라인 플러시 유발 가능

Trouble Shooting – R-type btaken 발생

```
// branch
always_comb begin
    if (branch) begin
        case (alu_controls[2:0])
            `BEQ: btaken = ($signed(a) == $signed(b));
            `BNE: btaken = ($signed(a) != $signed(b));
            `BLT: btaken = ($signed(a) < $signed(b));
            `BGE: btaken = ($signed(a) >= $signed(b));
            `BLTU: btaken = ($unsigned(a) < $unsigned(b));
            `BGEU: btaken = ($unsigned(a) >= $unsigned(b));
            default: btaken = 1'b0;
        endcase
    end else begin
        btaken = 1'b0;
    end
end
```



Trouble Shooting – register file[0] 쓰기 오류



register x0
0 -> 105

Corner Case

```
// rom[0]: x0 쓰기
rom[0] = 32'h06428013;   // addi x0, x5, 100

// rom[1]: x0 읽기
rom[1] = 32'h07B00093;   // addi x1, x0, 123
```

RISC-V 목적지 레지스터 x0

-> 쓰기 동작 무시되어야 함

Registers

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—

Trouble Shooting – register file[0] 쓰기 오류

```
always_ff @(posedge clk) begin
if (reg_wr_en && (WA != 5'b0)) begin
    reg_file[WA] <= WData;
end
```

```
// rom[0]: x0 쓰기
rom[0] = 32'h06428013;   // addi x0, x5, 100

// rom[1]: x0 읽기
rom[1] = 32'h07B00093;   // addi x1, x0, 123
```

Discussion & Future Plans

Discussion

RISC-V CPU 설계의 전반적인 과정을 경험

-> RISC-V ISA와 각 명령어의 동작 명확히 이해

시뮬레이션을 통하여 sign-extension이나 zero-extension과 같은
세부적인 동작까지 파형을 통해 직접 확인

-> RISC-V CPU Core 설계의 정확성을 높일 수 있음

Future Plans

- Type 별 기능 검증
- 파이프라인 아키텍처 도입 -> CPU의 성능을 개선
  (여러 명령어를 동시에 처리하여 전체 시스템의 명령어 처리량 높이기)

Thank you