

SYSTEM VERILOG PROJECT
RISC-V CPU 설계

발표자 : 김태형, 김호준, 안유한

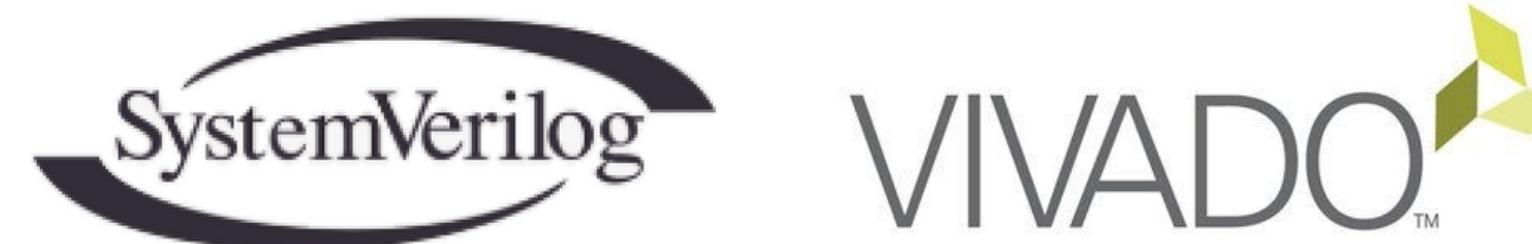
목차

1. RISC-V 개요
2. Block Diagram
3. 명령어 Type 시뮬레이션 및 검증

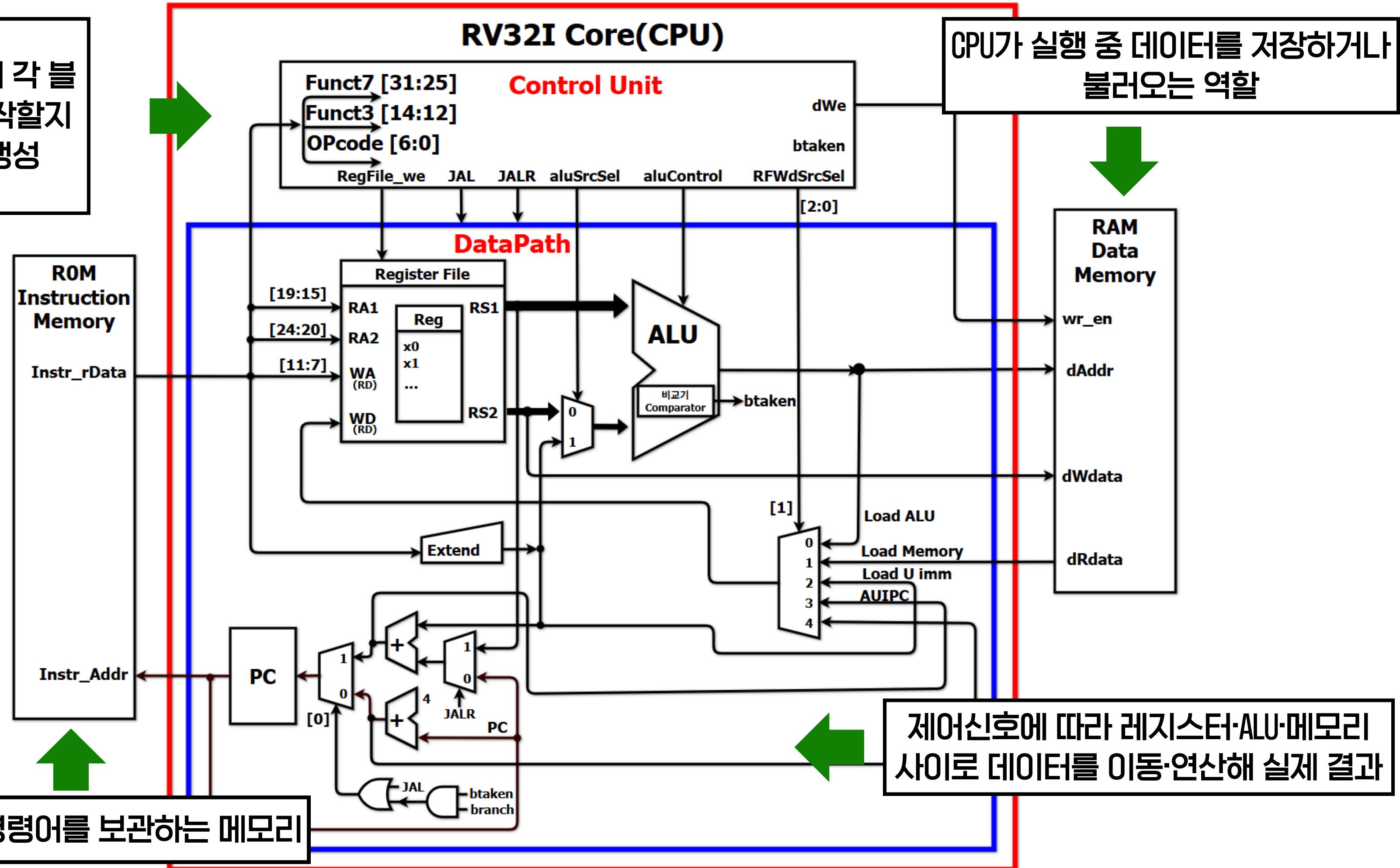
RISC-V 개요

📌 RISC-V

- ISA (명령어 집합 구조)
 - CPU가 이해하고 실행할 수 있는 명령어들의 집합
 - 하드웨어(프로세서 설계)와 소프트웨어(컴파일러, 운영체제)를 연결하는 인터페이스 역할
- RISC-V ISA 특징
 - 오픈소스 표준 ISA → 무료, 누구나 사용·확장 가능
 - 단순·규칙적 명령어 구조
 - 모듈식 확장성
- 설계 환경
 - System Verilog
 - Vivado



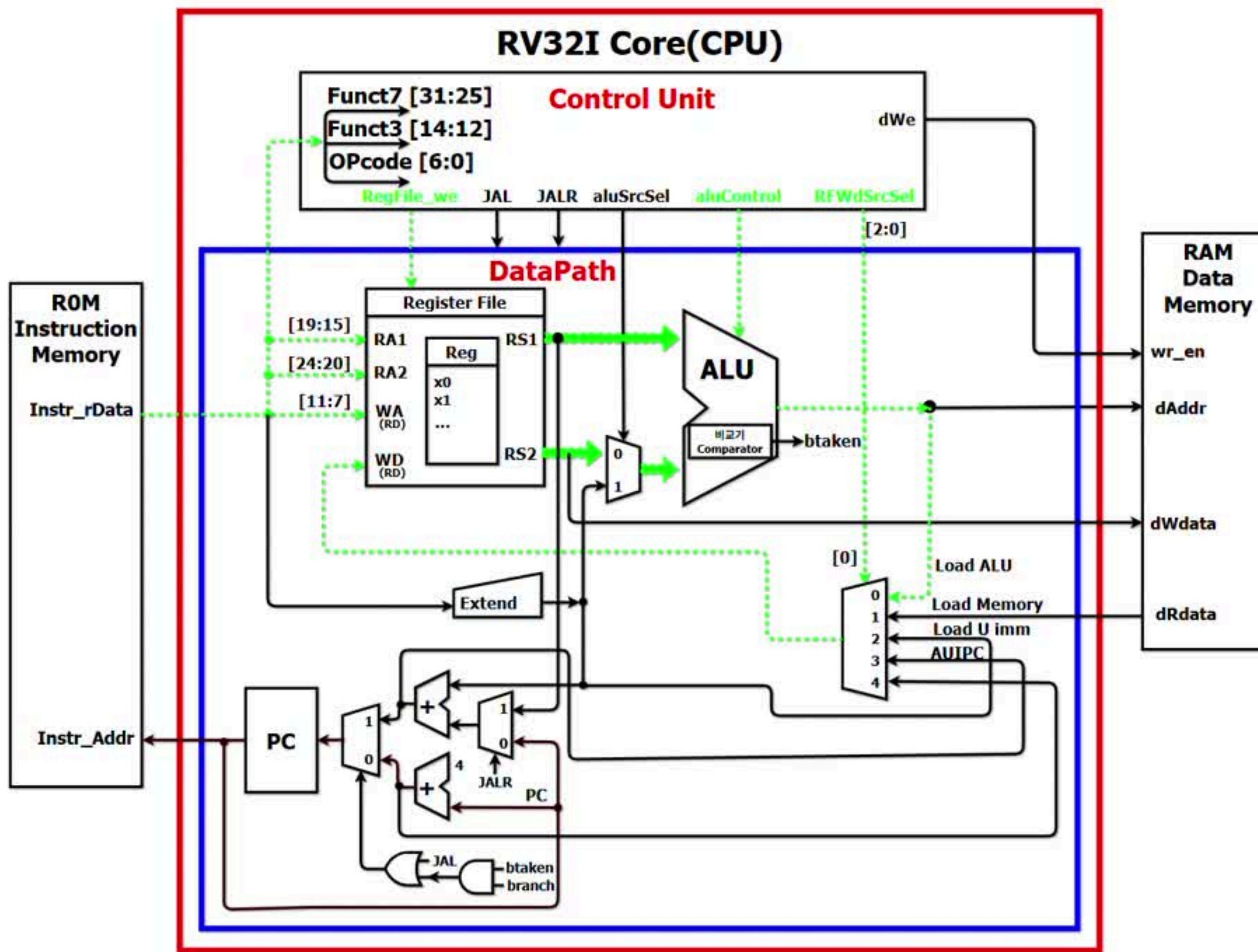
명령어를 해독해 각 블록이 어떻게 동작할지 제어신호를 생성



RISC-V Type

| 31 | 27 | 26 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 | |
|--------------|----|-----------------------|----|-----|----|--------|----|-------------|----|--------|--------|---|--------|--|
| funct7 | | rs2 | | rs1 | | funct3 | | | rd | | opcode | | R-type | |
| imm[11:0] | | | | rs1 | | funct3 | | | rd | | opcode | | I-type | |
| imm[11:5] | | rs2 | | rs1 | | funct3 | | imm[4:0] | | opcode | | | S-type | |
| imm[12 10:5] | | rs2 | | rs1 | | funct3 | | imm[4:1 11] | | opcode | | | B-type | |
| | | imm[31:12] | | | | | | | rd | | opcode | | U-type | |
| | | imm[20 10:1 11 19:12] | | | | | | | rd | | opcode | | J-type | |

R Type



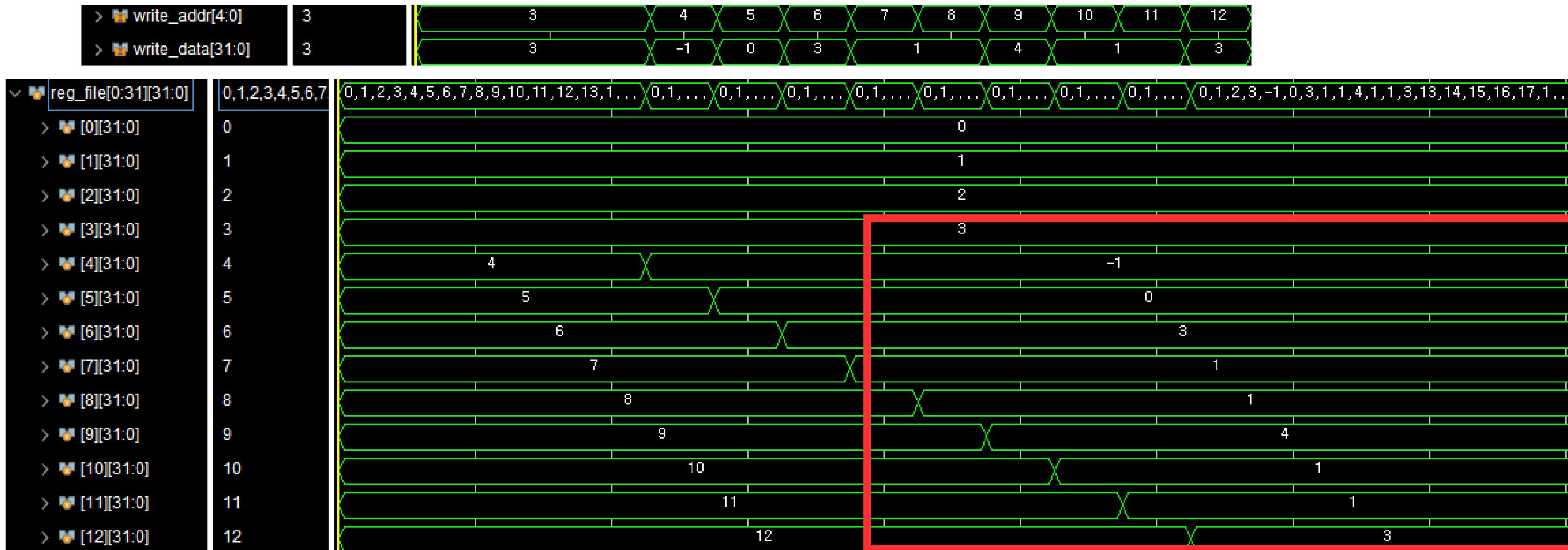
RISC-V Type

R Type

| instruction | funct7 | rs2 | rs1 | funct3 | rd | opcode |
|-------------|---------|-----|-----|--------|----|---------|
| ADD | 0000000 | rs2 | rs1 | 000 | rd | 0110011 |
| SUB | 0100000 | rs2 | rs1 | 000 | rd | 0110011 |
| SLL | 0000000 | rs2 | rs1 | 001 | rd | 0110011 |
| SLT | 0000000 | rs2 | rs1 | 010 | rd | 0110011 |
| SLTU | 0000000 | rs2 | rs1 | 011 | rd | 0110011 |
| XOR | 0000000 | rs2 | rs1 | 100 | rd | 0110011 |
| SRL | 0000000 | rs2 | rs1 | 101 | rd | 0110011 |
| SRA | 0100000 | rs2 | rs1 | 101 | rd | 0110011 |
| OR | 0000000 | rs2 | rs1 | 110 | rd | 0110011 |
| AND | 0000000 | rs2 | rs1 | 111 | rd | 0110011 |

Simulation

ADD SUB AND OR SLT SLTU SLL SRL SRA XOR

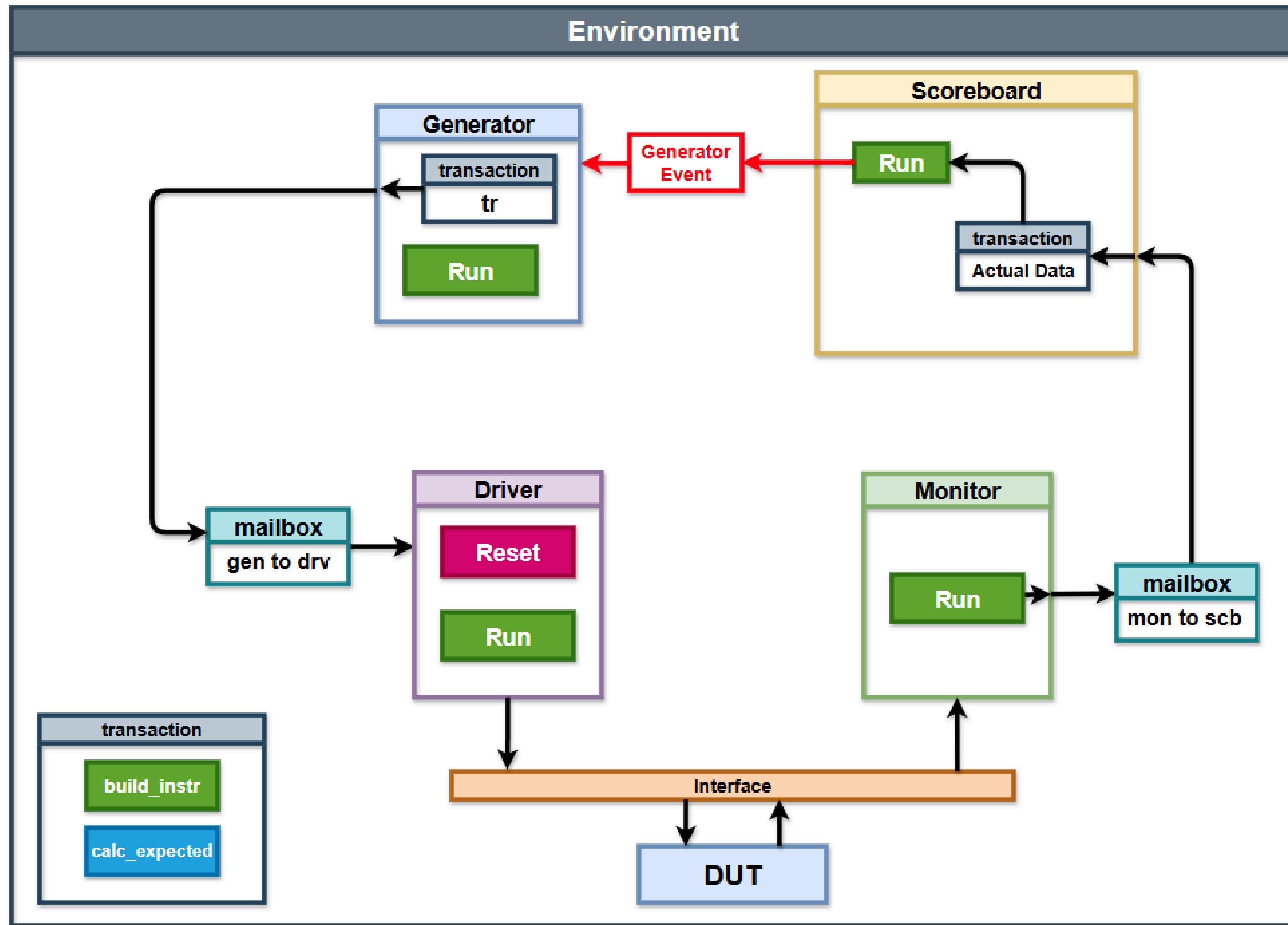


// 가정: rf[1]=1, rf[2]=2

```

rom[0] = 32'b0000000_00010_0001_00011_0110011; // ADD rf[3] = rf[1] + rf[2] // 예상값: rf[3] = 1 + 2 = 3
rom[1] = 32'b0100000_00010_0001_00100_0110011; // SUB rf[4] = rf[1] - rf[2] // 예상값: rf[4] = 1 - 2 = -1
rom[2] = 32'b0000000_00010_0001_111_00101_0110011; // AND rf[5] = rf[1] & rf[2] // 예상값: rf[5] = 1 & 2 = 0
rom[3] = 32'b0000000_00010_0001_110_00110_0110011; // OR rf[6] = rf[1] | rf[2] // 예상값: rf[6] = 1 | 2 = 3
rom[4] = 32'b0000000_00010_0001_010_00111_0110011; // SLT rf[7] = (rf[1] < rf[2]) ? 1:0 // 예상값: rf[7] = (1 < 2) -> 1
rom[5] = 32'b0000000_00010_0001_011_01000_0110011; // SLTU rf[8] = (rf[1] < rf[2]) ? 1:0 // 예상값: rf[8] = (1 < 2) -> 1
rom[6] = 32'b0000000_00001_00010_001_01001_0110011; // SLL rf[9] = rf[4] << rf[1] // 예상값: rf[9] = 2 << 1 = 4
rom[7] = 32'b0000000_00001_00010_101_01010_0110011; // SRL rf[10] = rf[4] >> rf[1] // 예상값: rf[10] = 2 >> 1 = 1
rom[8] = 32'b0100000_00001_00010_101_01011_0110011; // SRA rf[11] = rf[4] >>> rf[1]// 예상값: rf[11] = 2 >>> 1 = 1
rom[9] = 32'b0000000_00010_00001_100_01100_0110011; // XOR rf[12] = rf[1] ^ rf[2] // 예상값: rf[12] = 1 ^ 2 = 3

```



TR

```
class transaction;
    rand bit [ 2:0] funct3;
    rand bit      funct7b5;
    rand bit [ 4:0] rs1, rs2, rd;

    bit      [31:0] rs1_val;
    bit      [31:0] rs2_val;
    bit      [31:0] instr_code;
    bit      [31:0] exp_result;

    function void build_instr();
        bit [6:0] opcode = 7'b0110011;
        bit [6:0] funct7 = (funct7b5) ? 7'b0100000 : 7'b0000000;
        instr_code = {funct7[6:0], rs2[4:0], rs1[4:0], funct3[2:0], rd[4:0], opcode[6:0]};
    endfunction

    function void action();
        case ({funct7b5, funct3})
            4'b0000: exp_result = rs1_val + rs2_val;          // ADD
            4'b1000: exp_result = rs1_val - rs2_val;          // SUB
            4'b0111: exp_result = rs1_val & rs2_val;          // AND
            4'b0110: exp_result = rs1_val | rs2_val;          // OR
            4'b0100: exp_result = rs1_val ^ rs2_val;          // XOR
            4'b0010: exp_result = ($signed(rs1_val) < $signed(rs2_val)) ? 1 : 0; // SLT
            4'b0011: exp_result = (rs1_val < rs2_val) ? 1 : 0; // SLTU
            4'b0001: exp_result = rs1_val << rs2_val[4:0];    // SLL
            4'b0101: exp_result = rs1_val >> rs2_val[4:0];   // SRL
            4'b1101: exp_result = $signed(rs1_val) >>> rs2_val[4:0]; // SRA
            default: exp_result = 32'hDEAD_BEEF;
        endcase
    endfunction
endclass
```

인스트럭션 코드 생성 및 연산 로직 선언

GEN

```
task run(int n);
    repeat (n) begin
        transaction tr = new();
        assert (tr.randomize());
        tr.build_instr();
        gen2drv.put(tr);

        $display("[GEN] instr=0x%08h (%s) rs1=%0d rs2=%0d", tr.instr_code,
                 | tr.get_instr_name(), tr.rs1, tr.rs2);
        total_count++;
        @(gen_next_event);
    end
endtask
```

R 명령어 중 하나를 랜덤 생성

소스 레지스터(rs1, rs2), 목적지 레지스터(rd), funct3, funct7[5]을 제약 조건에 맞게 무작위로 생성

이 값을 조합하여 32비트 기계어 코드(inst_code)를 생성

DRV

```
task run();
    forever begin
        transaction tr;
        gen2drv.get(tr);

        @(posedge vif.clk);
        vif.instr_code = tr.instr_code;
        vif.exp_result = tr.exp_result;

        $display("%0t [DRV] instr=%08h (%s) rs1=%0d rs2=%0d",
            |   |   | $time, tr.instr_code, tr.get_instr_name(), tr.rs1, tr.rs2);

        drv2mon.put(tr);
    end
endtask
```

gen2drv.get(tr);로 제너레이터가 만든 트랜잭션을 받아오고, @(posedge vif.clk)에 맞춰 구동

vif.instr_code = tr.instr_code로 DUT에 명령어(32b)를 드라이브하고,
vif.exp_result = tr.exp_result는 파형 확인용 예상값 생성

MON

```
task run(int n);
    repeat (n) begin
        transaction tr;
        drv2mon.get(tr);

        @(posedge vif.clk);
        tr.rs1_val = tb_rv32i.dut.U_RV32I_CPU.U_data_path.U_REG_FILE.reg_file[tr.rs1];
        tr.rs2_val = tb_rv32i.dut.U_RV32I_CPU.U_data_path.U_REG_FILE.reg_file[tr.rs2];

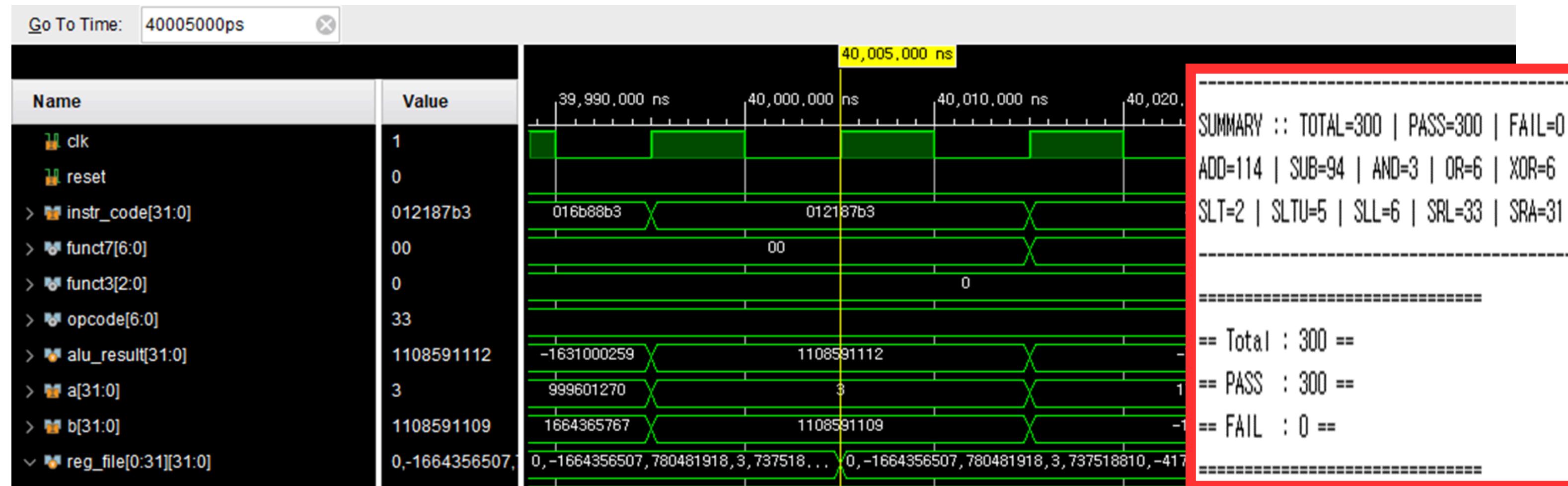
        tr.action();
        tr.display("MON", vif.aiu_result);

        mon2scb.put(tr);
    end
endtask
```

drv2mon.get(tr);로 드라이버가 넘긴 동일 트랜잭션을 받고, 다음
@(posedge vif.clk)에서 샘플링

tr.rs1_val/rs2_val = tb_reg_file[]로 DUT 레지스터 파일 값을 직접 읽어 트
랜잭션에 기록한 뒤, tr.action();에서 이 값들을 이용해 예상 동작/결과 산출
을 수행

Verification



[GEN] instr=0x012187b3 (ADD) rs1=3 rs2=18



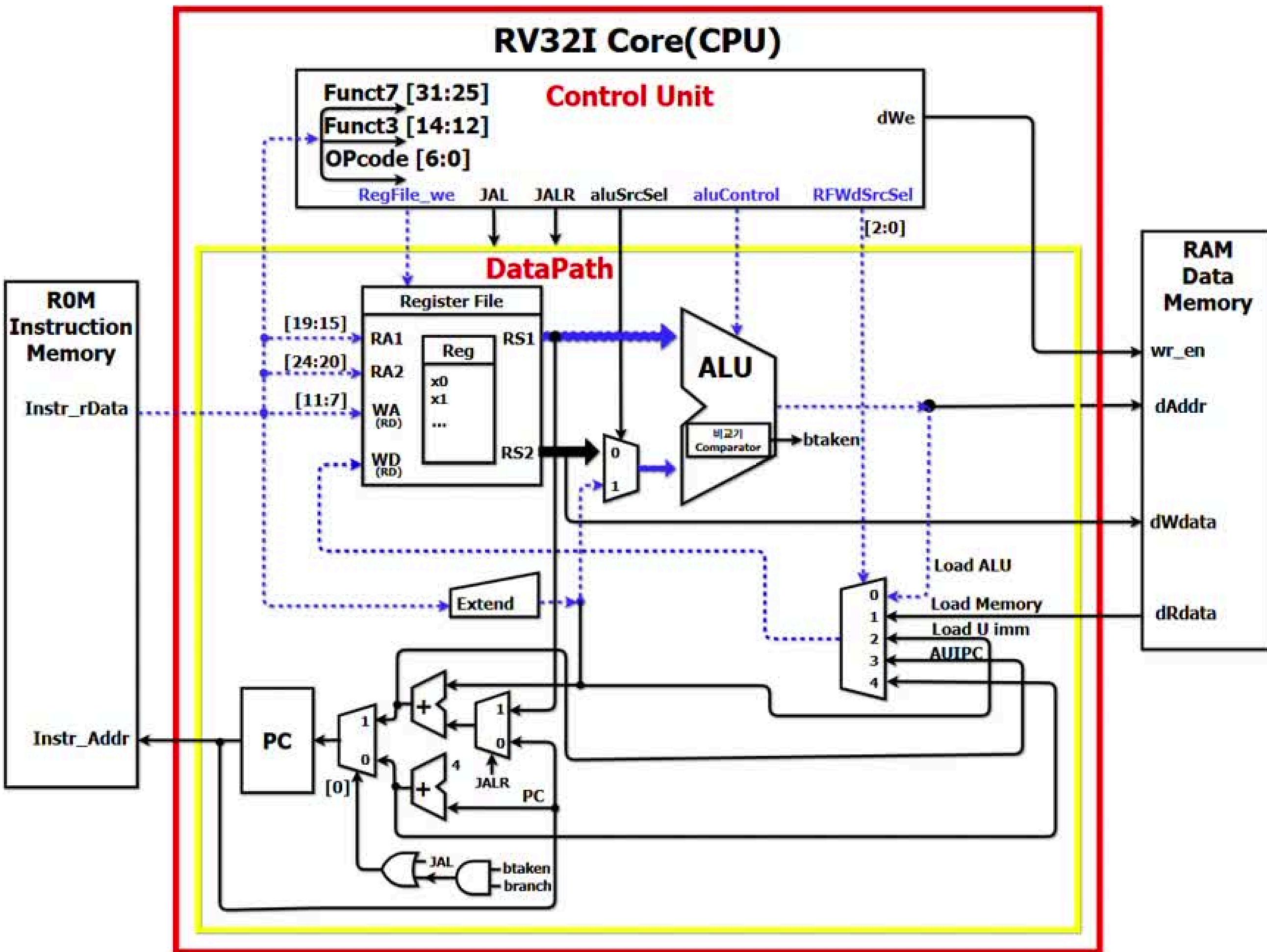
| func7 | rs2 | rs1 | func3 | rd | opcode |
|---------|-------|-------|-------|-------|---------|
| 0000000 | 10010 | 00011 | 000 | 01111 | 0110011 |

39995000 [DRV] instr=0x012187b3 (ADD) rs1=3 rs2=18

40005000 [MON] instr=0x012187b3 (ADD) | rs1=3(val=3) | rs2=18(val=1108591109) | rd=15(val=114117588)



I Type



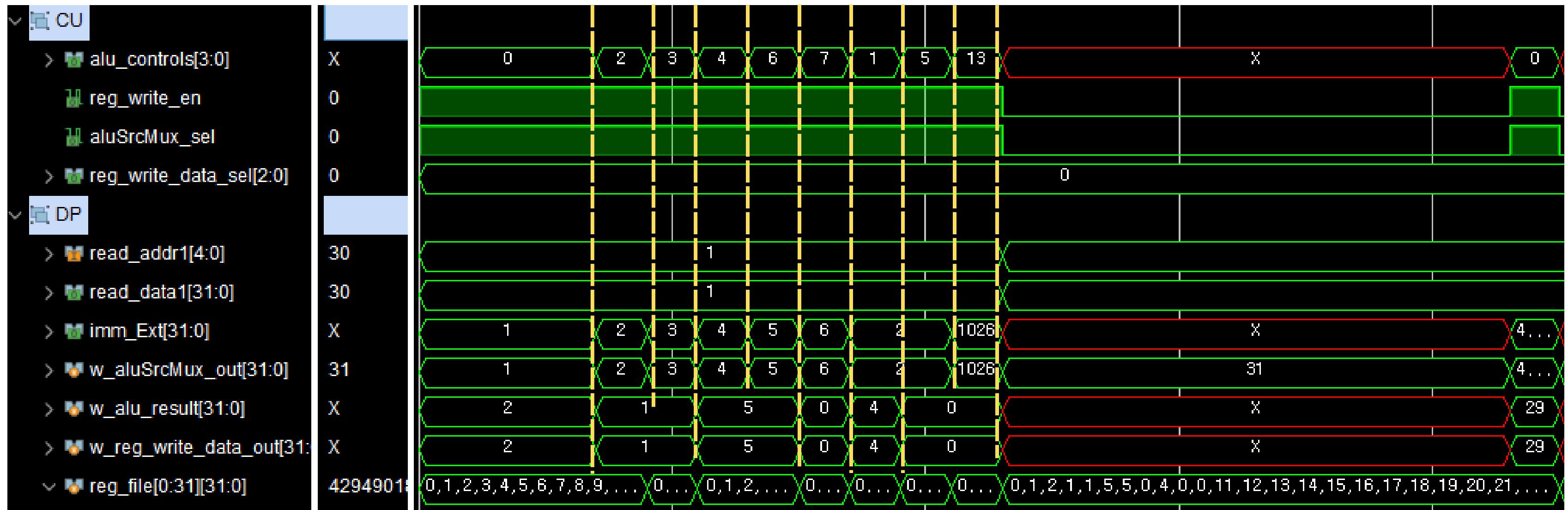
RISC-V Type

I Type

| instruction | IMM [11:0] | rs1 | funct3 | rd | opcode |
|-------------|------------|-----|--------|----|---------|
| ADDI | IMM | rs1 | 000 | rd | 0010011 |
| SLTI | IMM | rs1 | 010 | rd | 0010011 |
| SLTIU | IMM | rs1 | 011 | rd | 0010011 |
| XORI | IMM | rs1 | 100 | rd | 0010011 |
| ORI | IMM | rs1 | 110 | rd | 0010011 |
| ANDI | IMM | rs1 | 111 | rd | 0010011 |

| instruction | FUNCT7 | SHAMT | rs1 | funct3 | rd | opcode |
|-------------|---------|-------|-----|--------|----|---------|
| SLLI | 0000000 | SHAMT | rs1 | 001 | rd | 0010011 |
| SRLI | 0000000 | SHAMT | rs1 | 101 | rd | 0010011 |
| SRAI | 0100000 | SHAMT | rs1 | 101 | rd | 0010011 |

Simulation



// I-type 산술/논리 연산 예시

```

rom[0] = 32'b000000000001_0001_00010_0010011; // ADDI rd=2, rs1=1, imm=1 // 예상 rd[2]=2
rom[1] = 32'b000000000010_0001_010_00011_0010011; // SLTI rd=3, rs1=1, imm=2 // 1<2 → rd[3]=1
rom[2] = 32'b000000000011_0001_011_00100_0010011; // SLTIU rd=4, rs1=1, imm=3 // 1<3 → rd[4]=1
rom[3] = 32'b000000000100_0001_100_00101_0010011; // XORI rd=5, rs1=1, imm=4 // 1 ^ 4 = 5 → rd[5]=5
rom[4] = 32'b000000000101_0001_110_00110_0010011; // ORI rd=6, rs1=1, imm=5 // 1 | 5 = 5 → rd[6]=5
rom[5] = 32'b000000000110_0001_111_00111_0010011; // ANDI rd=7, rs1=1, imm=6 // 1 & 6 = 0 → rd[7]=0

```

// I-type 시프트 연산 예시

```

rom[6] = 32'b0000000_00010_0001_001_01000_0010011; // SLLI rd=8, rs1=1, shamt=2 // 1 << 2 = 4 → rd[8]=4
rom[7] = 32'b0000000_00010_00001_101_01001_0010011; // SRLI rd=9, rs1=1, shamt=2 // 1 >> 2 = 0 → rd[9]=0
rom[8] = 32'b0100000_00010_00001_101_01010_0010011; // SRAI rd=10, rs1=1, shamt=2 // 1 >>> 2 = 0 → rd[10]=0

```

TR

```
class transaction;
    rand bit [ 2:0] funct3;
    rand bit         funct7b5;
    rand bit [ 4:0] rs1,        rs2, rd;
    rand bit [11:0] imm_Ext;
    bit      [31:0] rs1_val;
    bit      [31:0] instr_code;
    bit      [31:0] exp_result;
    bit signed [31:0] signed_imm;

    function void build_instr();
        bit [6:0] opcode = 7'b0010011;

        if (funct3 == 3'b101) begin
            bit [6:0] funct7 = (funct7b5) ? 7'b0100000 : 7'b0000000;
            instr_code = {funct7, imm_Ext[4:0], rs1, funct3, rd, opcode};
        end else begin // 일반 I-Type 명령어
            instr_code = {imm_Ext, rs1, funct3, rd, opcode};
        end
    endfunction

    function void calc_expected();
        case ({funct7b5, funct3})
            4'b0000: exp_result = rs1_val + signed_imm;
            4'b0010: exp_result = ($signed(rs1_val) < signed_imm) ? 1 : 0; // SLTI (signed)
            4'b0011: exp_result = (rs1_val < signed_imm) ? 1 : 0; // SLTIU (unsigned)
            4'b0100: exp_result = rs1_val ^ signed_imm;
            4'b0110: exp_result = rs1_val | signed_imm;
            4'b0111: exp_result = rs1_val & signed_imm;
            4'b0001: exp_result = rs1_val << imm_Ext[4:0];
            4'b0101: exp_result = rs1_val >> imm_Ext[4:0];
            4'b1101: exp_result = $signed(rs1_val) >>> imm_Ext[4:0];
            default: exp_result = 32'hDEAD_BEEF;
        endcase
    endfunction

```

인스트럭션 코드 생성 및 연산 로직 선언

GEN

```
task run(int n);
    repeat (n) begin
        transaction tr = new();
        assert (tr.randomize());
        tr.build_instr();
        gen2drv.put(tr);

        $display("%0t [GEN] instr=%08h (%s) rs1=%0d imm_Ext=%0d",
                $time, tr.instr_code, tr.get_instr_name(), tr.rs1, tr.signed_imm);

        total_count++;
        @(gen_next_event);
    end

```

I 명령어 중 하나를 랜덤 생성

소스 레지스터(rs1, rs2), 목적지 레지스터(rd), 오프셋(imm), funct7[5]을 제약 조건에 맞게 무작위로 생성

이 값을 조합하여 32비트 기계어 코드(instr_code)를 생성

DRV

```
task run();
    forever begin
        transaction tr;
        gen2drv.get(tr);
        @(posedge vif.clk);
        vif.instr_code = tr.instr_code;
        vif.exp_result = tr.exp_result;
        $display("%0t [DRV] instr=%08h (%s) rs1=%0d imm_Ext=%0d ",
            |   | $time, tr.instr_code, tr.get_instr_name(), tr.rs1, tr.signed_imm);
        // tr.display("DRV", vif.alu_result);
        drv2mon.put(tr);
    end
endtask
```

GEN 메일박스에서 트랜잭션을 받아 @(posedge clk)에 맞춰 instr_code 등 입력 신호만 인터페이스로 드라이브합니다.

그 트랜잭션을 drv2mon으로 넘겨 모니터가 동일 컨텍스트로 관측·검증 할 수 있게 합니다

MON

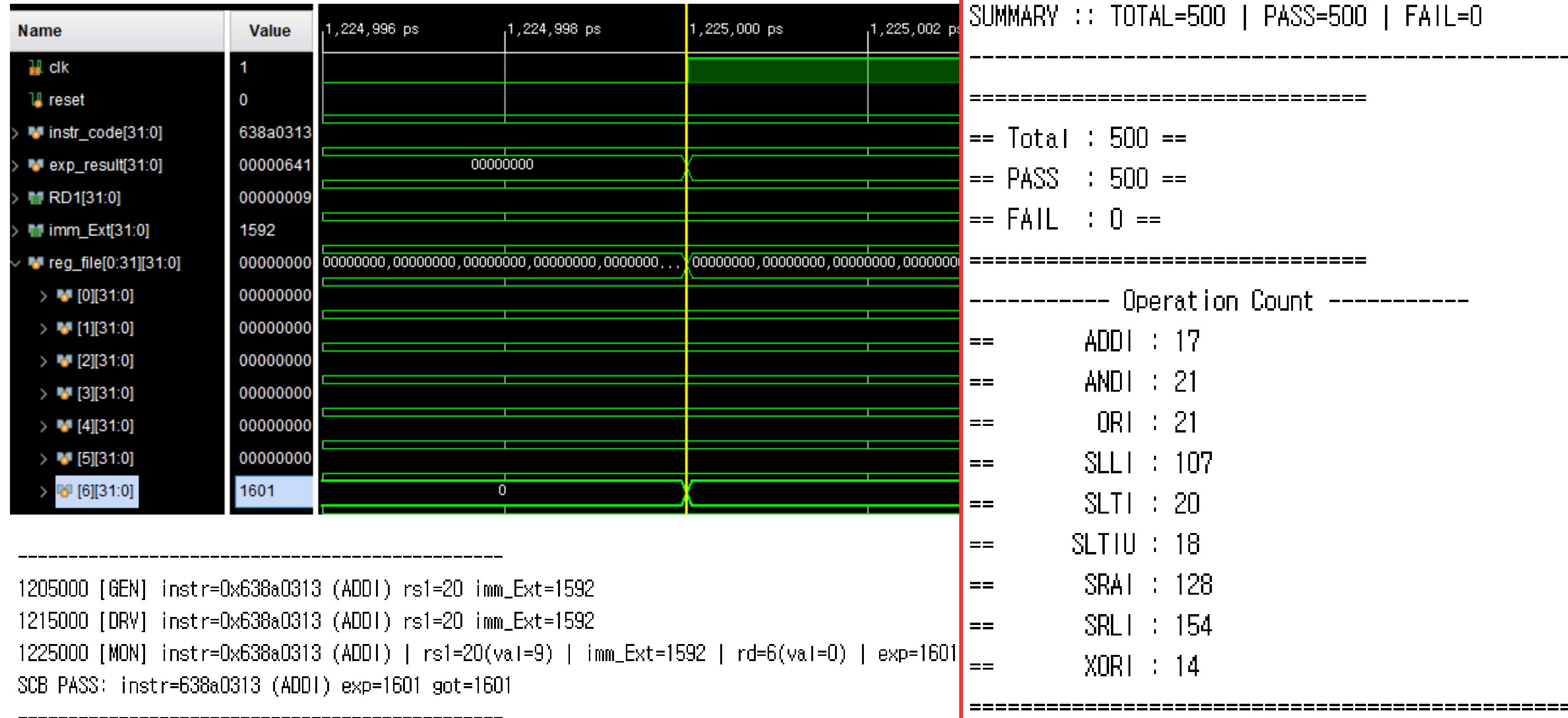
```
task run(int n);
    repeat (n) begin
        transaction tr;
        drv2mon.get(tr);
        @(posedge vif.clk);
        tr.rs1_val = tb_rv32i.dut.U_RV32I_CPU.U_data_path.U_REG_FILE.reg_file[tr.rs1];
        tr.calc_expected();
        vif.exp_result = tr.exp_result;
        tr.display("MON", vif.alu_result);

        mon2scb.put(tr);
    end
endtask
```

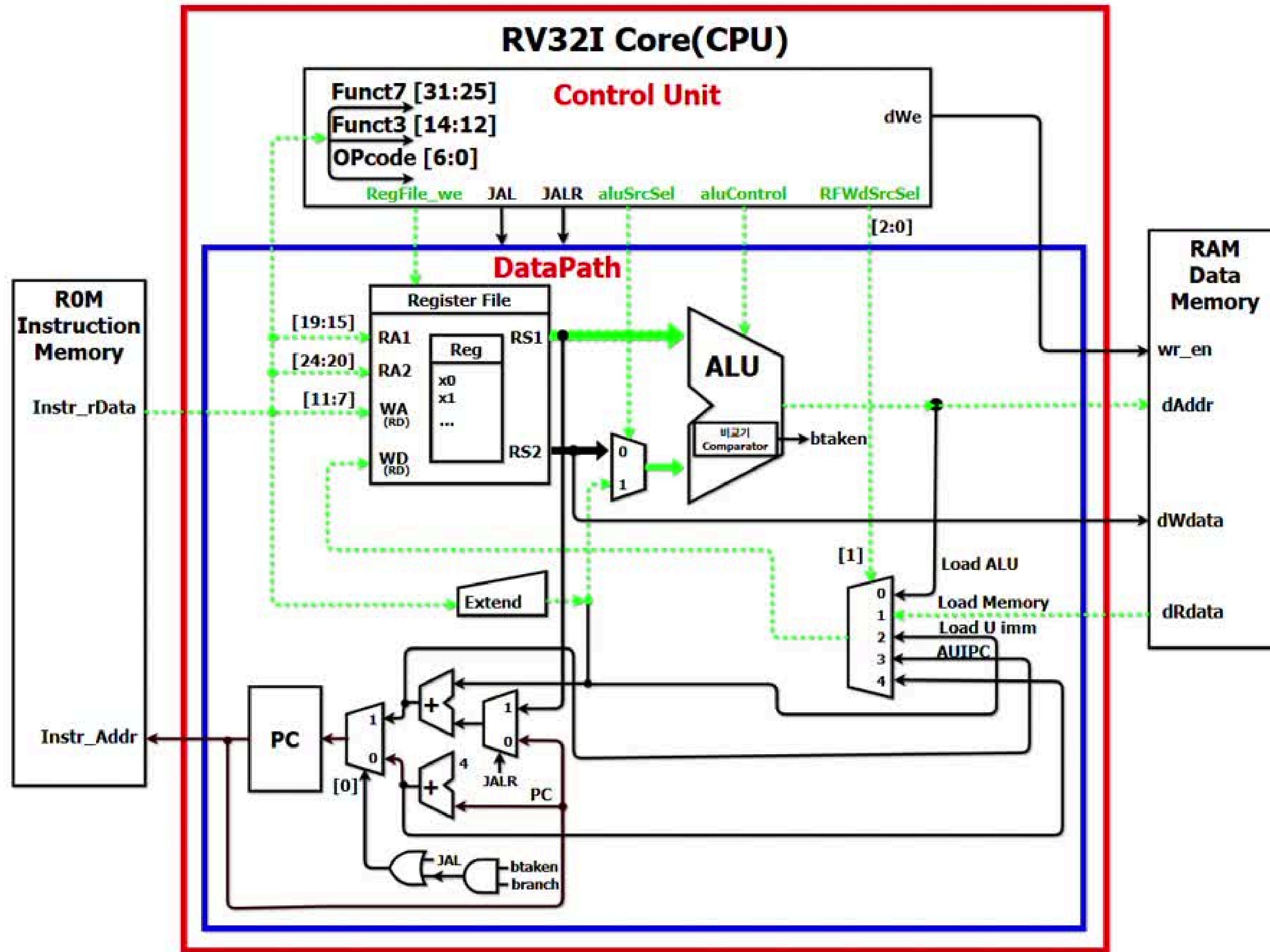
drv2mon에서 받은 트랜잭션을 다음 @(posedge clk)에 샘플링하고, 관측한 값을 바탕으로 calc_expected()로 예상값을 산출합니다.

산출된 exp와 DUT의 실제 결과를 비교하도록 트랜잭션을 mon2scb로 전달 해 스코어보드 비교를 트리거합니다

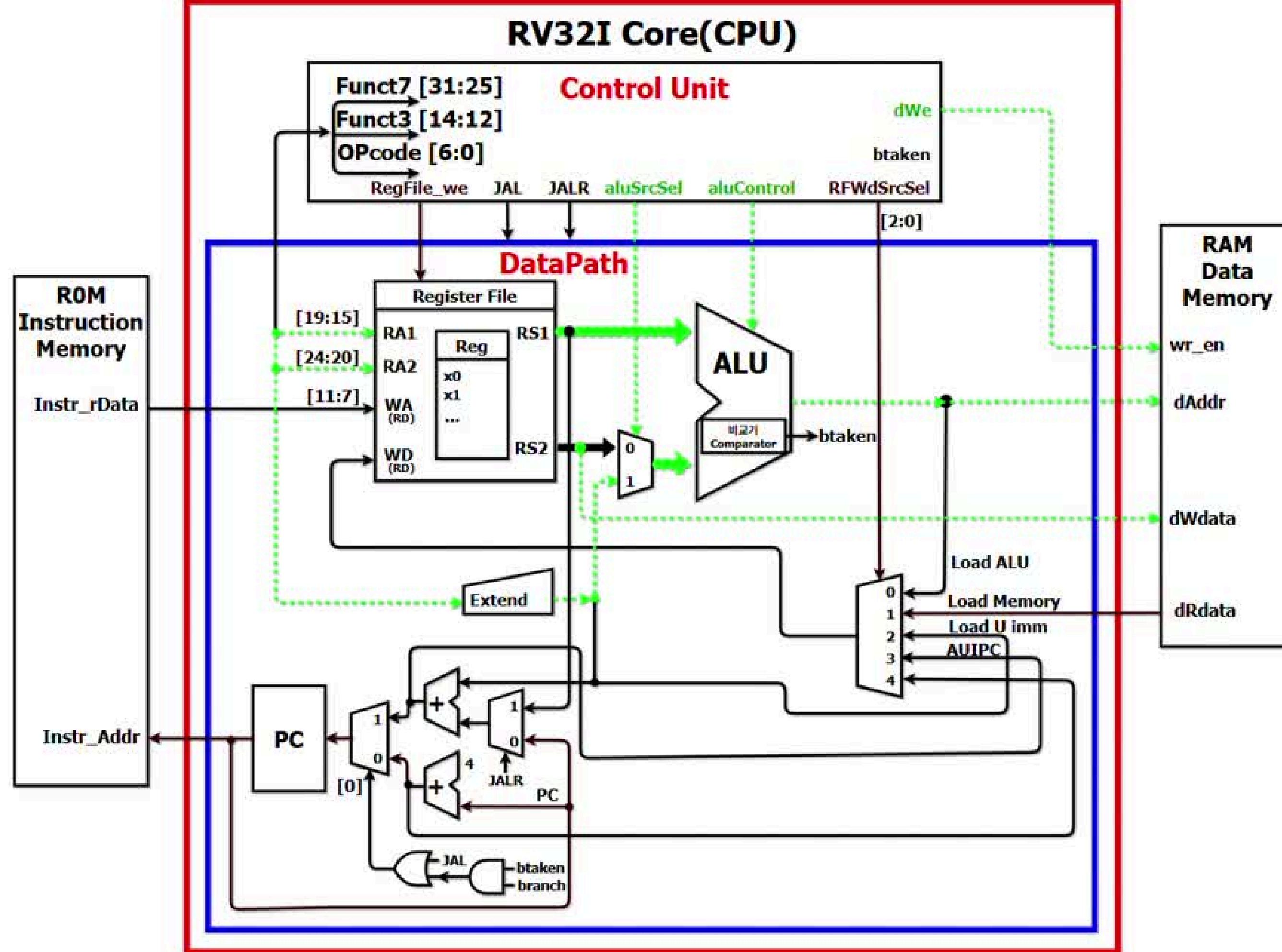
I-type Verification



L Type



S Type



RISC-V Type

L Type

| instruction | IMM [11:0] | rs1 | funct3 | rd | opcode |
|-------------|------------|-----|--------|----|---------|
| LB | IMM | rs1 | 000 | rd | 0000011 |
| LH | IMM | rs1 | 001 | rd | 0000011 |
| LW | IMM | rs1 | 010 | rd | 0000011 |
| LBU | IMM | rs1 | 100 | rd | 0000011 |
| LHU | IMM | rs1 | 101 | rd | 0000011 |

- $imm + rs1 = 메모리 주소 계산$
- $rd에 메모리에서 읽은 값을 저장$
- LBU/LHU는 무부호 로드
- B rd = $M[rs1+imm][0:7]$
- H rd = $M[rs1+imm][0:15]$
- W rd = $M[rs1+imm][0:31]$

S Type

| instruction | IMM [11:5] | rs2 | rs1 | funct3 | IMM[4:0] | opcode |
|-------------|------------|-----|-----|--------|----------|---------|
| SB | IMM | rs2 | rs1 | 000 | IMM | 0100011 |
| SH | IMM | rs2 | rs1 | 001 | IMM | 0100011 |
| SW | IMM | rs2 | rs1 | 010 | IMM | 0100011 |

- 메모리 주소 = $rs1 + imm$
- $rs2의 값을 메모리에 저장$
- imm은 상위/하위 12비트로 분리되어 instruction에 encoding
- B $[rs1+imm][0:7] = rs2[0:7]$
- H $[rs1+imm][0:15] = rs2[0:15]$
- W $[rs1+imm][0:31] = rs2[0:31]$

S Type

CONTROL_UNIT

```
case (funct3)
    `SB: s_type_controls = 3'b001;
    `SH: s_type_controls = 3'b010;
    `SW: s_type_controls = 3'b100;
    default: s_type_controls = 3'b000;
endcase
```

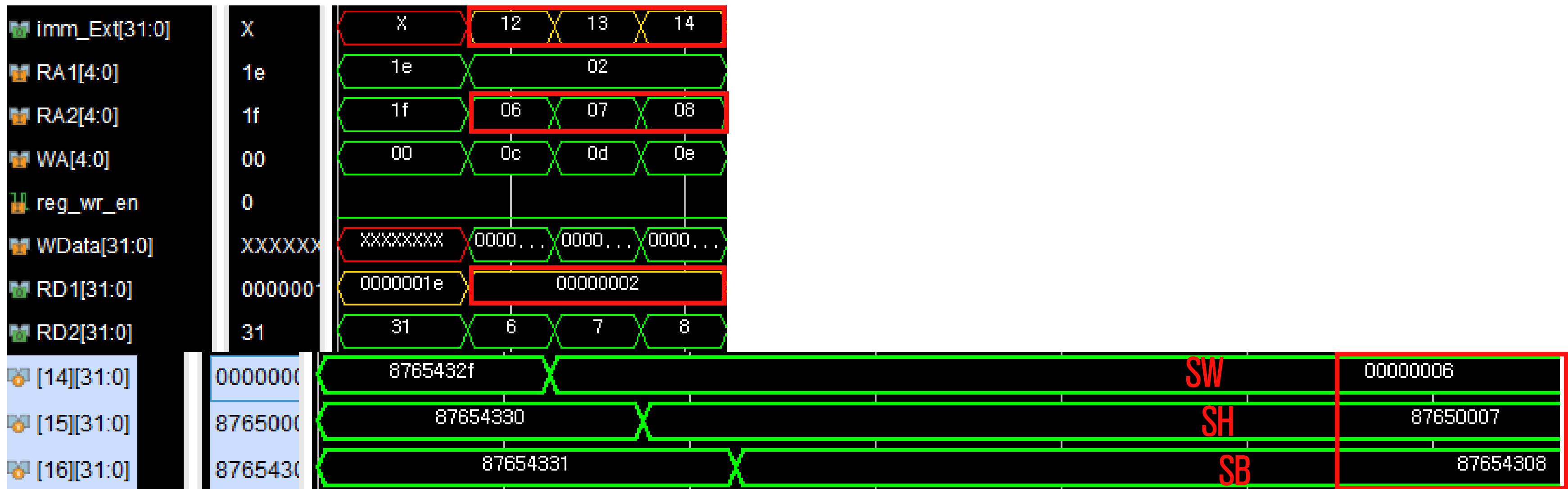
instr_mem

```
// rom[1]: sw x6, 12(x2)
rom[1] = 32'h00612623;
// rom[2]: sh x7, 13(x2)
rom[2] = 32'h007116A3;
// rom[3]: sb x8, 14(x2)
rom[3] = 32'h00810723;
```

DATA_MEMORY

```
// Store (SB, SH, SW)
always_ff @(posedge clk) begin
    if (d_write_en) begin
        //data_mem[d_addr] <= d_write_data;
        case (s_type_controls)
            3'b001: begin // sb
                data_mem[d_addr] <= d_write_data[7:0];
            end
            3'b010: begin // sh
                data_mem[d_addr] <= d_write_data[7:0];
                data_mem[d_addr+1] <= d_write_data[15:8];
            end
            3'b100: begin // sw
                data_mem[d_addr] <= d_write_data[7:0];
                data_mem[d_addr+1] <= d_write_data[15:8];
                data_mem[d_addr+2] <= d_write_data[23:16];
                data_mem[d_addr+3] <= d_write_data[31:24];
            end
        endcase
    end
end
```

Simulation



register file

| | | |
|-----------|----------|----------|
| [6][31:0] | 00000000 | 00000006 |
| [7][31:0] | 00000000 | 00000007 |
| [8][31:0] | 00000000 | 00000008 |

S-type Verification

```
task run(int n);
    repeat (n) begin
        transaction tr = new();
        assert (tr.randomize());

        tr.rs1_val =
tb_s_type.dut.U_RV32I_CPU.U_data_path.U_REG_FILE.reg_file[tr.rs1];
        tr.rs2_val =
tb_s_type.dut.U_RV32I_CPU.U_data_path.U_REG_FILE.reg_file[tr.rs2];
        tr.exp_addr = tr.rs1_val + $signed(tr.imm);
        tr.before_store = tb_s_type.dut.U_DATA_RAM.data_mem[tr.exp_addr];

        tr.build_instr();
        tr.calc_expected();

        gen2drv.put(tr);
        tr.display("GEN");
        total_count++;

        @(gen_next_event);
    end
endtask
```

GEN

명령어 랜덤 생성
레지스터(rs1, rs2)와 임시값(imm)을 랜덤하게 생성

베이스 주소 레지스터(rs1)의 현재 값
메모리에 쓸 데이터가 담긴 레지스터(rs2)의 현재 값
을 모두 읽어옴

데이터가 쓰여질 메모리 위치에 현재 저장되어 있는 값
(before_store)

DRV

Gen이 생성한 Store 명령어를 받아 CPU 실행

```
task run();
    forever begin
        transaction tr;
        gen2drv.get(tr);

        @(posedge vif.clk);
        vif.instr_code = tr.instr_code;

        $display("%0t [DRV] Driving instr=%0d (%s)", $time, tr.instr_code,
            tr.get_instr_name());
        drv2mon.put(tr);
    end
endtask
```

S-type Verification

```
task run(int n);
    repeat (n) begin
        transaction tr;
        logic [31:0] monitored_val;
        drv2mon.get(tr);
        @(posedge vif.clk);

        #1;
        monitored_val = tb_s_type.dut.U_DATA_RAM.data_mem[vif.dAddr];
        $display("%0t [MON] Store Detected -> mem[%0d]: %h", $time,
            vif.dAddr, monitored_val);
        tr.actual_result = monitored_val;
        mon2scb.put(tr);
    end
endtask
```

MON

실제로 데이터가 쓰여진 메모리 주소에 접근하여
그 최종 결과 값(actual_result)을 읽어옴

```
if (vif.d_wr_en) begin
    if (vif.dAddr === tr.exp_addr && vif.dWdata === tr.exp_dWdata
    && tr.actual_result === tr.exp_result) begin
        pass_count++;
        $display("SCB PASS: : Type match");
        $display("Store Data[%0d]:%8h", vif.dAddr,
            tr.actual_result);
        $display("Expected Store Data[%0d]:%8h", vif.dAddr,
            tr.exp_result);
    end
end
```

SCB

쓰기 신호 &주소&데이터&최종 결과가
모두 일치하는지 확인

S-type Verification

25000 [GEN] instr=0x1c520a3(SW) | rs1=x10(val=0x10) | rs2=x1c(val=0x1c) | imm=1 | exp_addr=0x11 | exp_wdata=0x28 | exp_result=0000001c

35000 [DRV] Driving instr=29696163 (SW)

46000 [MON] Store Detected -> mem[11]: 0000001c

SCB PASS: : Type match

Store Data[11]:0000001c

Expected Store Data[11]:0000001c



47000 [GEN] instr=0x2713a3(SH) | rs1=x14(val=0x14) | rs2=x2(val=0x2) | imm=7 | exp_addr=0

55000 [DRV] Driving instr=2560931 (SH)

66000 [MON] Store Detected -> mem[21]: 87650002

SCB PASS: : Type match

Store Data[21]:87650002

Expected Store Data[21]:87650002



87000 [GEN] instr=0xd204a3(SB) | rs1=x4(val=0x4) | rs2=xd(val=0xd) | imm=9 | exp_addr=0x13 | exp_wdata=0x13 | exp_result=8765430d

95000 [DRV] Driving instr=13763747 (SB)

106000 [MON] Store Detected -> mem[13]: 8765430d

SCB PASS: : Type match

Store Data[13]:8765430d

Expected Store Data[13]:8765430d



SUMMARY :: TOTAL=50 | PASS=50 | FAIL=0

INSTRUCTION COUNT :: SB=18 | SH=16 | SW=16

L Type

CONTROL_UNIT

```
case (funct3)
`LB:    i_type_controls = 3'b001;
`LH:    i_type_controls = 3'b010;
`LW:    i_type_controls = 3'b011;
`LBU:   i_type_controls = 3'b100;
`LHU:   i_type_controls = 3'b101;
default: i_type_controls = 3'b000;
endcase
```

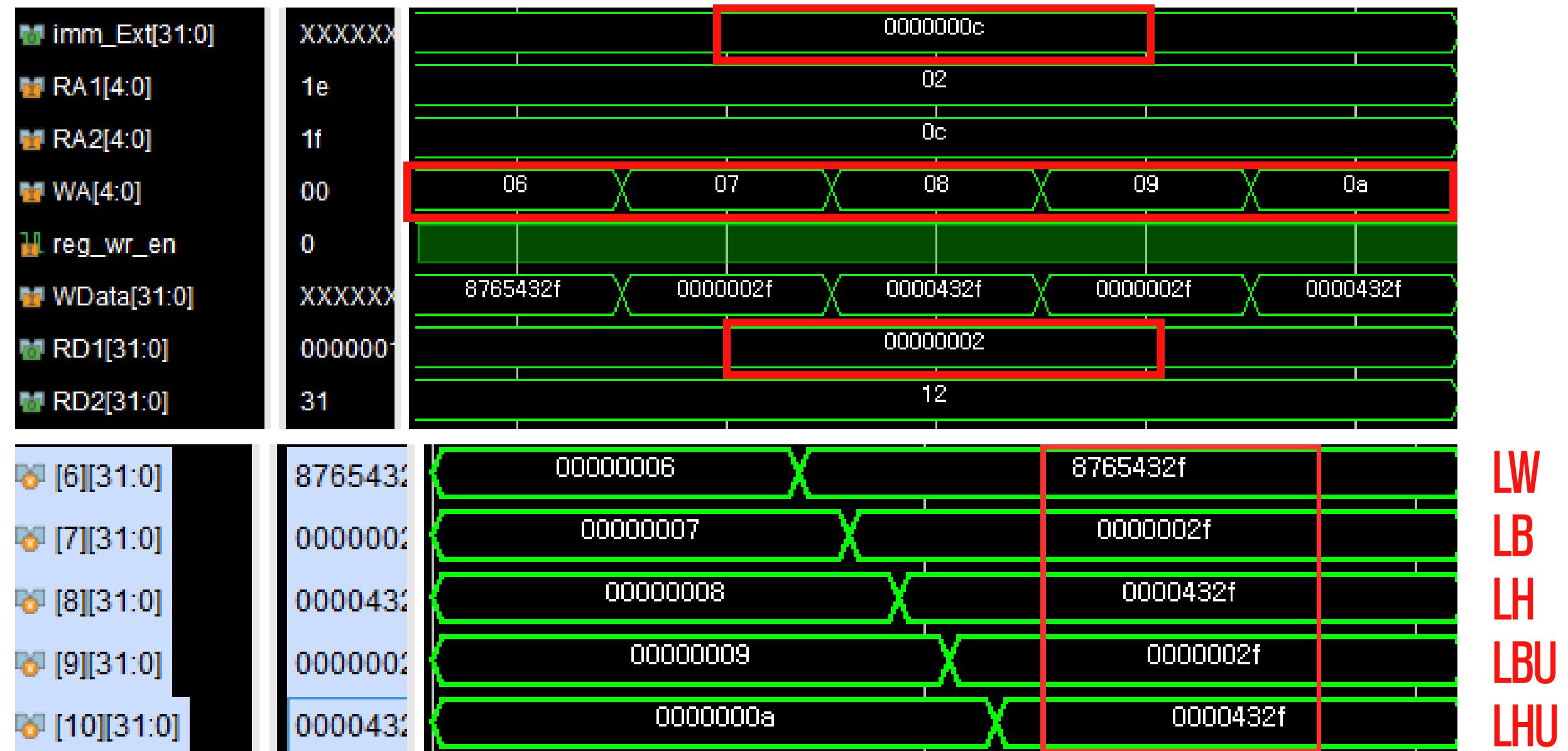
DATA_MEMORY

```
always_comb begin
  case (load_type)
    `LB  : i_data = {{24{dRdata[7]}},dRdata[7:0]};
    `LH  : i_data = {{16{dRdata[15]}},dRdata[15:0]};
    `LW  : i_data = dRdata;
    `LBU : i_data = {{24{1'b0}},dRdata[7:0]};
    `LHU: i_data = {{16{1'b0}},dRdata[15:0]};
    default: i_data = 32'bx;
  endcase
end
```

instr_mem

```
//IL-type
rom[6] = 32'h00C12303;
// rom[6]: lw x6, 12(x2)
// 32'b000000001100_00010_010_00110_0000011
// rom[7]: lb x7, 12(x2)
rom[7] = 32'h00C10383;
// 32'b000000001100_00010_000_00111_0000011
// rom[8]: lh x8, 12(x2)
rom[8] = 32'h00C11403;
// 32'b000000001100_00010_001_01000_0000011
// rom[9]: lbu x9, 12(x2)
rom[9] = 32'h00C14483;
// 32'b000000001100_00010_100_01001_0000011
// rom[10]: lhu x10, 12(x2)
rom[10] = 32'h00C15503;
// 32'b000000001100_00010_101_01010_0000011
```

Simulation



RAM



L-type Verification

GEN

```
task run(int n, logic [7:0] virtual_mem[0:255]);
    repeat (n) begin
        tr = new();
        if (prev_write_addr != 0) begin
            assert (tr.randomize() with {rs1 != prev_write_addr;});
        end else begin
            assert (tr.randomize());
        end
        tr.build_instr();
        gen2drv.put(tr);
        tr.display("GEN");
        total_count++;
    end
    @(gen_next_event);
endtask
```

DRV

```
task run();
    forever begin
        transaction tr;
        gen2drv.get(tr);
        @(posedge cpu_if.clk);

        tr.rs1_val =
            tb_verifi_cpu.dut.U_RV32I_CPU.U_DATAPATH.U_REG_FILE.reg_file[tr.rs1];
        tr.calc_expected(tb_verifi_cpu.env.virtual_mem);

        $display(
            "%0t [DRV] Driving inst=%0h (%s) | rs1_val=%0d | exp_addr=%0d
            | exp_data=%0d",
            $time, tr.inst_code, tr.get_inst_name(), tr.rs1_val,
            tr.exp_addr, tr.exp_read_data);

        cpu_if.inst_code = tr.inst_code;
        drv2mon.put(tr);
    end
endtask
```

Load 명령어 (LW, LH, LB) 중 하나를 랜덤 생성

소스 레지스터(rs1), 목적지 레지스터(rd), 오프셋(imm)을
제약 조건에 맞게 무작위로 생성

이 값들을 조합하여 32비트 기계어 코드(inst_code)를 생성

생성된 명령어를 CPU에 전달하기 전에, CPU의 현재 rs1 레지스터 값 read

읽어온 rs1 값과 명령어의 imm 값을 더해 데이터를 가져올
메모리 주소(exp_addr)를 계산

해당 주소에서 exp_read_data를 미리 계산

L-type Verification

MON

```
task run(int n);
    repeat (n) begin
        transaction tr;
        drv2mon.get(tr);

        repeat (3) @(posedge cpu_if.clk);

        tr.actual_addr = cpu_if.d_addr;
        tr.actual_read_data = cpu_if.d_read_data;

        $display(
            "%0t [MON] ACTUAL Addr=%0d | Data=%0d | Expected Addr=%0d |
            Data=%0d",
            $time, tr.actual_addr, tr.actual_read_data, tr.exp_addr,
            tr.exp_read_data);

        mon2scb.put(tr);
    end
endtask
```

CPU가 명령어를 실행하는 동안 메모리에 실제로 접근
하는 주소(actual_addr)와 읽어온 데이터
(actual_read_data)를 캡쳐

SCB

```
if(tr.actual_addr === tr.exp_addr && tr.actual_read_data ===
    tr.exp_read_data) begin
    pass_count++;
    $display("[SCB PASS] %s | Addr=%0d | Data=%0d",
        tr.get_inst_name(), tr.actual_addr,
        tr.actual_read_data);
end |
```

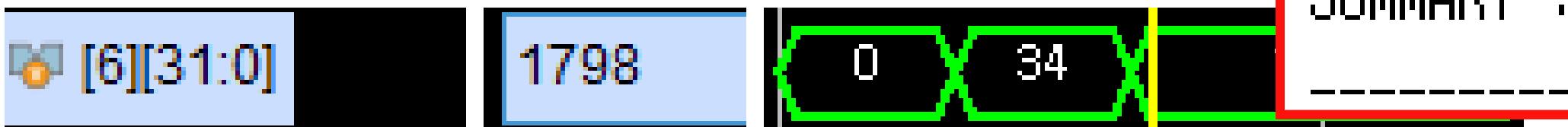
Driver가 계산한 예상 결과(exp_addr, exp_read_data)
와 Monitor가 관찰한 실제 결과(actual_addr,
actual_read_data)를 비교

L-type Verification

1505000 [GEN] instr=0x472603(LW) | rs1=x14 | rd=12 | imm=4 | exp_addr=x | exp_data=x
1515000 [DRV] Driving inst=0x472603 (LW) | rs1_val=9508 | exp_addr=9512 | exp_data=x
1545000 [MON] ACTUAL Addr=9512 | Data=x | Expected Addr=9512 | Data=x
[SCB PASS] LW | Addr=9512 | Data=x



305000 [GEN] instr=0x651303(LH) | rs1=x10 | rd=6 | imm=6 | exp_addr=x | exp_data=x
315000 [DRV] Driving inst=0x651303 (LH) | rs1_val=0 | exp_addr=6 | exp_data=1798
345000 [MON] ACTUAL Addr=6 | Data=1798 | Expected Addr=6 | Data=-----
[SCB PASS] LH | Addr=6 | Data=1798

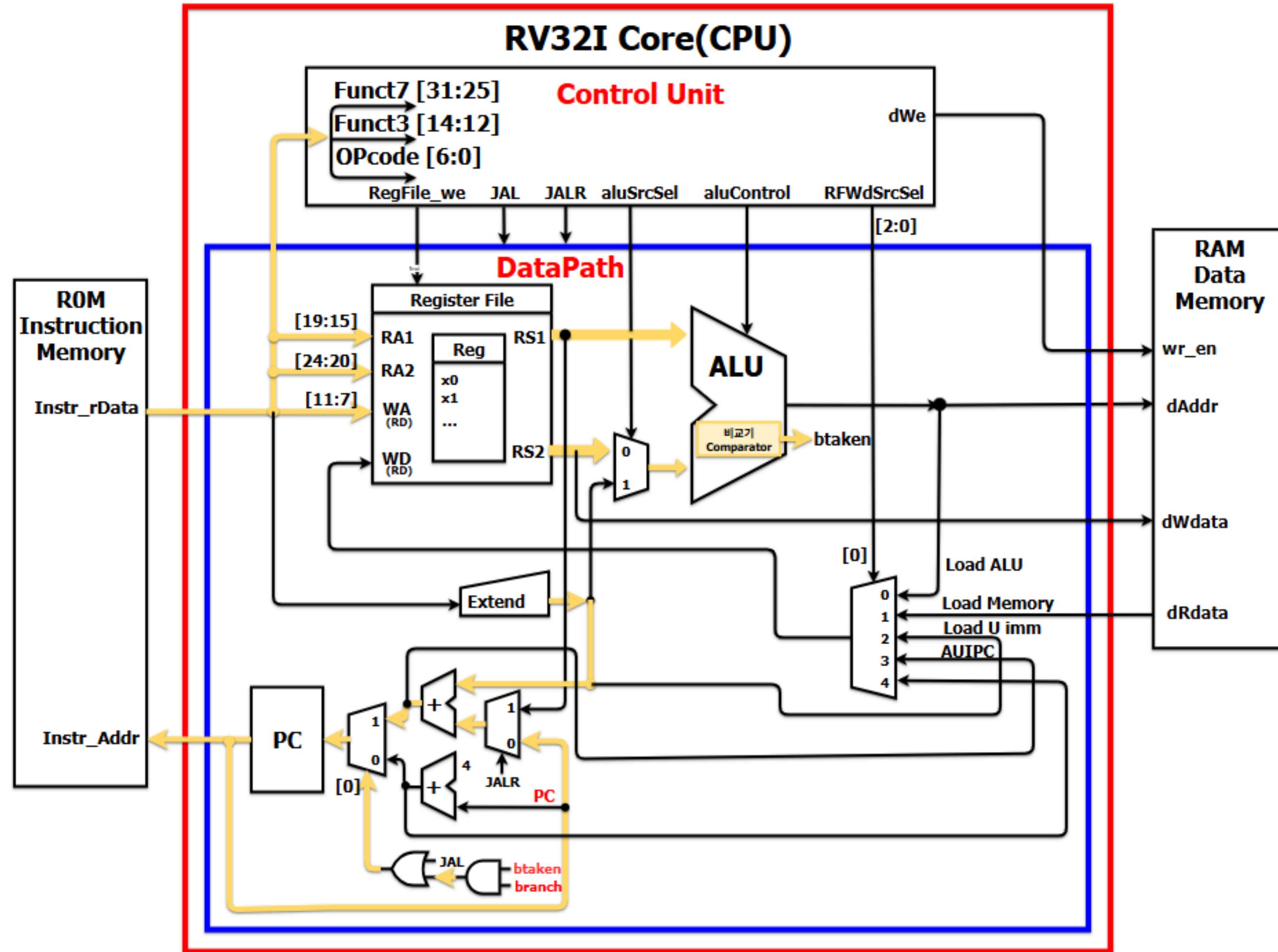


1225000 [GEN] instr=0x598d83(LB) | rs1=x19 | rd=27 | imm=5 | exp_addr=x | exp_data=x
1235000 [DRV] Driving inst=0x598d83 (LB) | rs1_val=24 | exp_addr=29 | exp_data=29
1265000 [MON] ACTUAL Addr=29 | Data=29 | Expected Addr=29 | Data=29
[SCB PASS] LB | Addr=29 | Data=29



SUMMARY :: TOTAL=100 | PASS=100 | FAIL=0

B Type



RISC-V Type

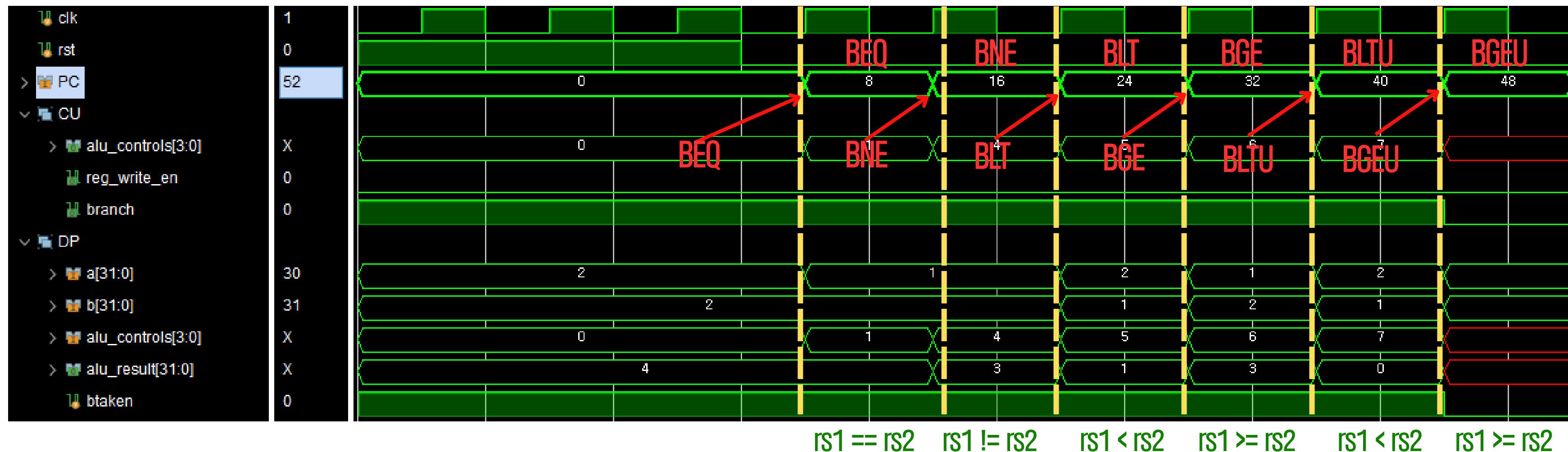
B Type

| instruction | IMM [12, 10:5] | rs2 | rs1 | funct3 | IMM[4:1, 11] | opcode |
|-------------|----------------|-----|-----|--------|--------------|---------|
| BEQ | IMM | rs2 | rs1 | 000 | IMM | 0110011 |
| BNE | IMM | rs2 | rs1 | 001 | IMM | 0110011 |
| BLT | IMM | rs2 | rs1 | 100 | IMM | 0110011 |
| BGE | IMM | rs2 | rs1 | 101 | IMM | 0110011 |
| BLTU | IMM | rs2 | rs1 | 110 | IMM | 0110011 |
| BGEU | IMM | rs2 | rs1 | 111 | IMM | 0110011 |

| | |
|------|--------------------------|
| 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 |
| BGEU | if(rs1 >= rs2) PC += imm |

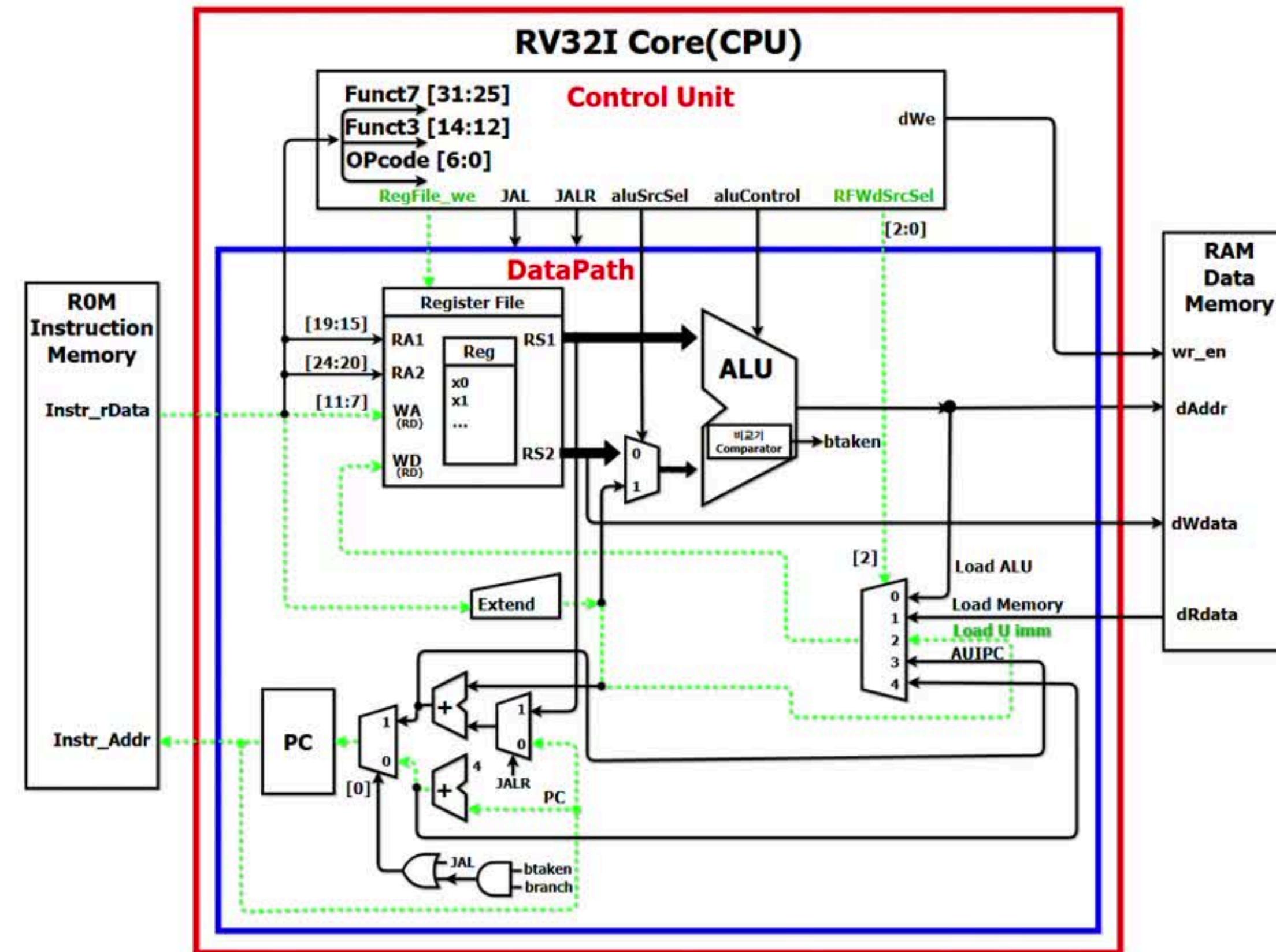
- 조건 분기(Conditional Branch) 명령어
- PC(Program Counter)를 변경하여 특정 조건에 따라 코드 흐름을 제어
- Branch 발생
 - 조건이 참이면 $PC = PC + imm$
 - 조건이 거짓이면 $PC = PC + 4$ (다음 명령어)

Simulation

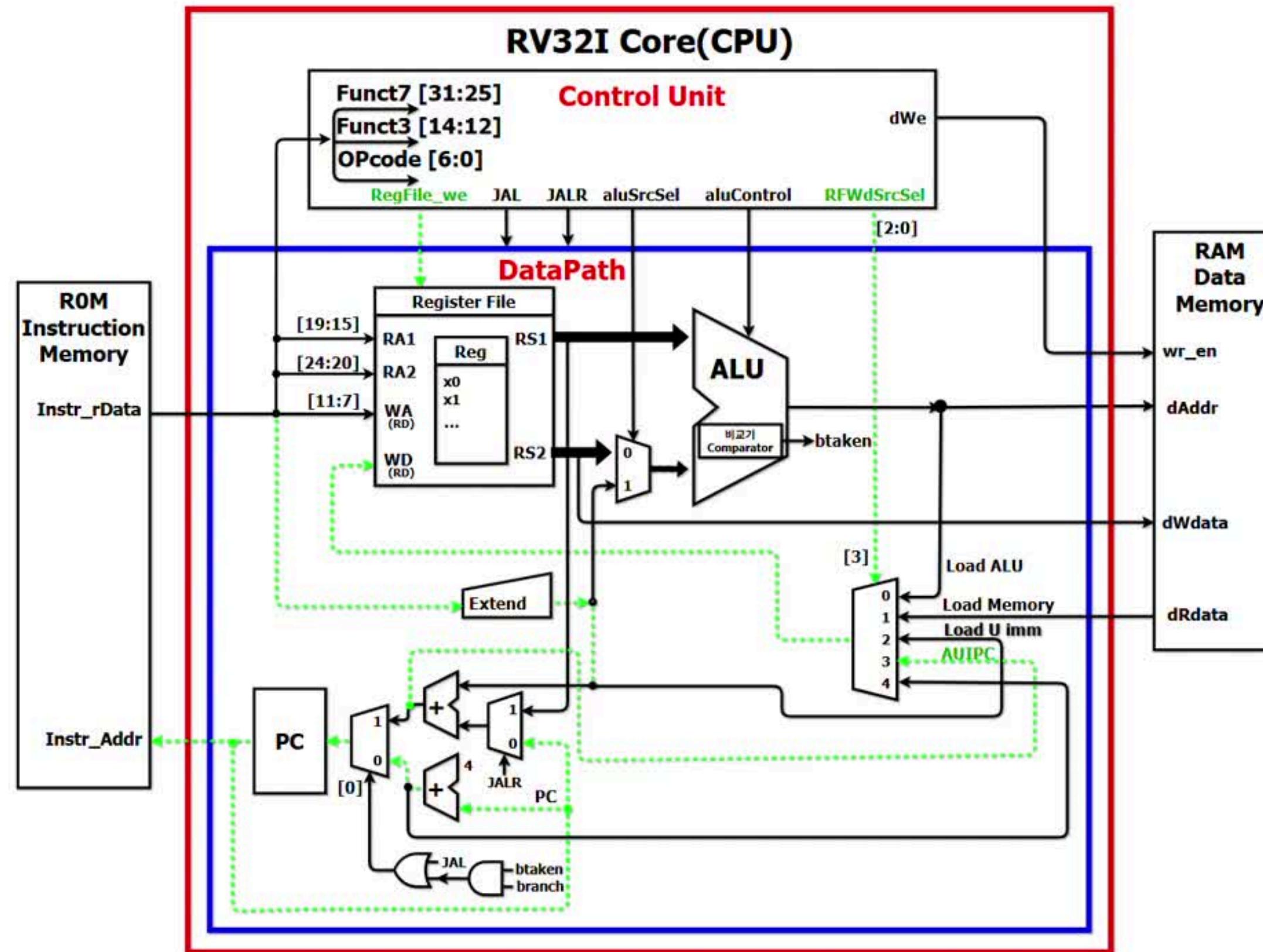


```
// B-type 명령어 구성 결과를 참으로 두어 8씩 증가함을 확인
// 32'b imm(7bit) _ rs2(5bit) _ rs1(5bit) _ funct3(3bit) _ imm (5bit) _ opcode(7'b1100011)
rom[0] = 32'b0000000_00010_00010_000_01000_1100011; // BEQ x2, x2, 8
rom[2] = 32'b0000000_00010_00001_001_01000_1100011; // BNE x1, x2, 8
rom[4] = 32'b0000000_00010_00001_100_01000_1100011; // BLT x1, x2, 8 (signed)
rom[6] = 32'b0000000_00001_00010_101_01000_1100011; // BGE x2, x1, 8 (signed)
rom[8] = 32'b0000000_00010_00001_110_01000_1100011; // BLTU x1, x2, 8 (unsigned)
rom[10] = 32'b0000000_00001_00010_111_01000_1100011; // BGEU x2, x1, 8 (unsigned)
```

U Type_LUI



U Type AUIPC



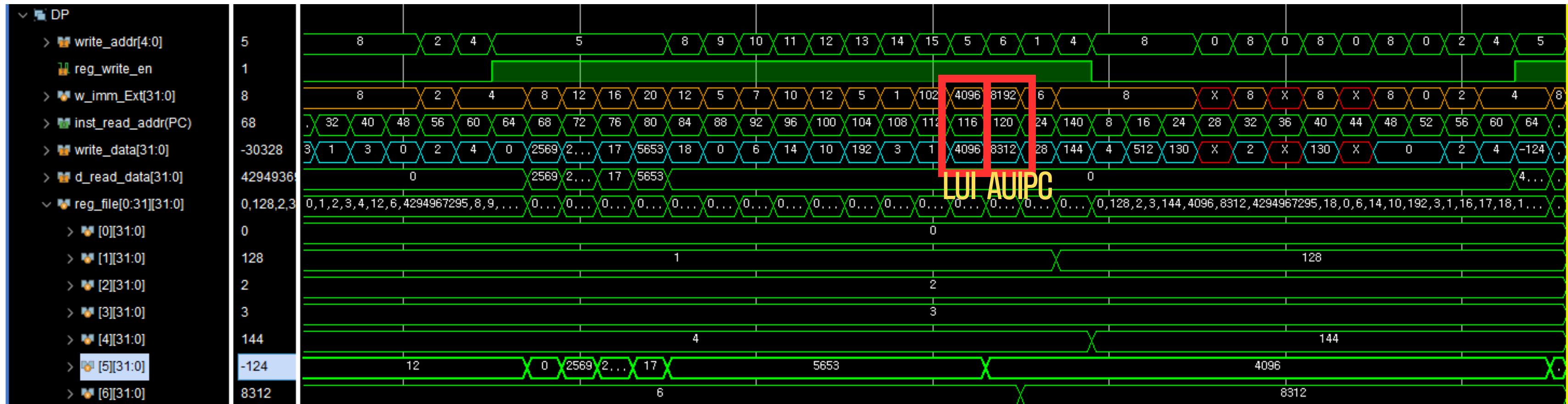
RISC-V Type

U Type

| instruction | IMM [31:12] | rd | opcode |
|-------------|-------------|----|---------|
| LUI | IMM | rd | 0110111 |
| AUIPC | IMM | rd | 0010111 |

- **LUI**
 - Immediate의 상위 20비트를 레지스터에 로드
 - 하위 12비트는 0으로 채움
 - 주로 큰 상수값 로드 또는 주소 계산 초기값으로 사용
 - $rd = imm[31:12] \ll 12$
 - 즉, 하위 12비트 0
- **AUIPC**
 - $rd = PC + (imm[31:12] \ll 12)$
 - PC를 기준으로 20비트 상수를 상위에 얹어서 큰 주소/상대주소를 rd에 만든다

Simulation



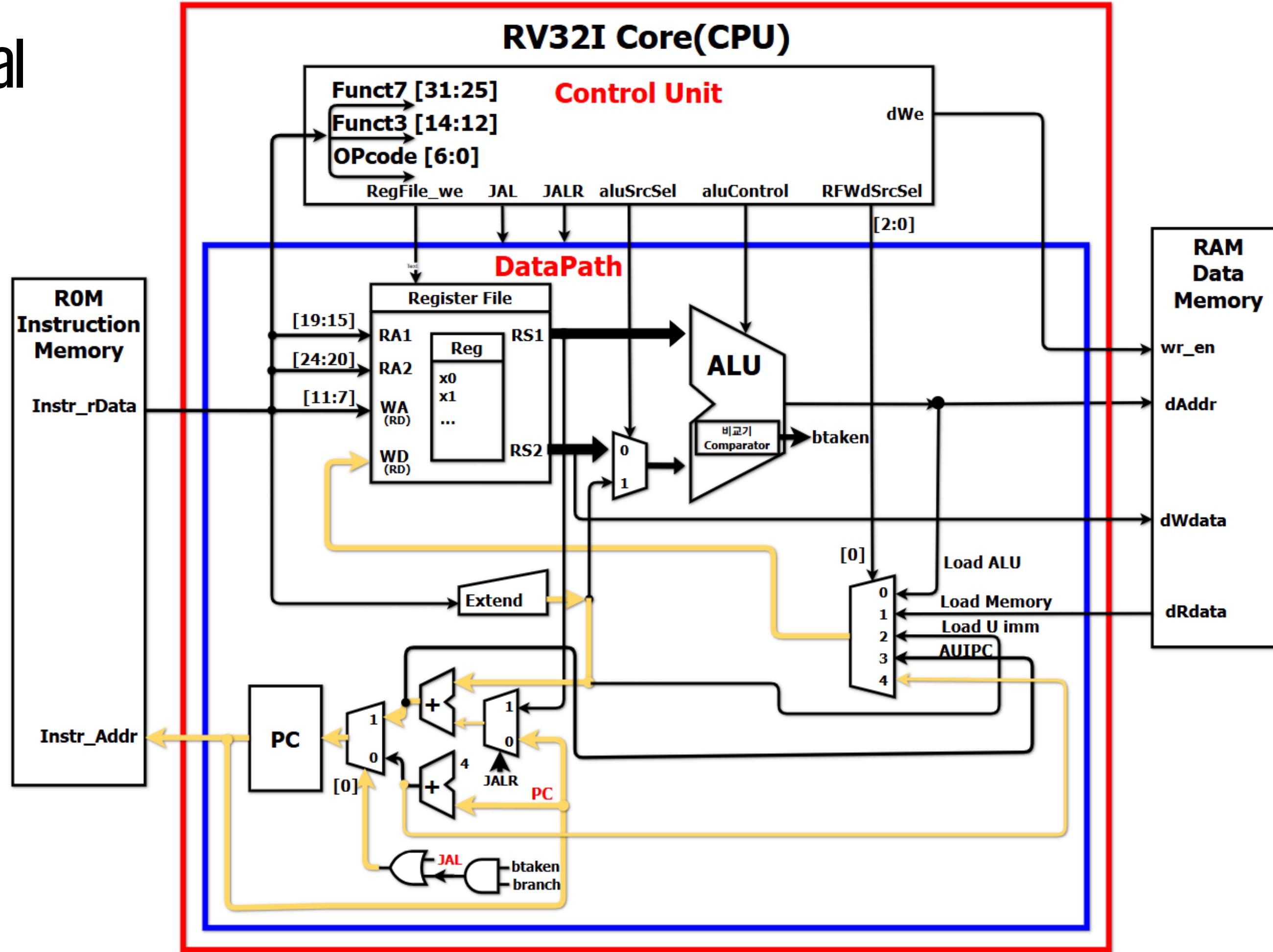
```
// U-type
rom[29] = 32'b00000000000000000001_00101_0110111; // LUI    x5, 0x1
rom[30] = 32'b00000000000000000010_00110_0010111; // AUIPC x6, 0x2
```

| | |
|-------|---------------------------------|
| LUI | $rd = \text{imm} \ll 12$ |
| AUIPC | $rd = PC + (\text{imm} \ll 12)$ |

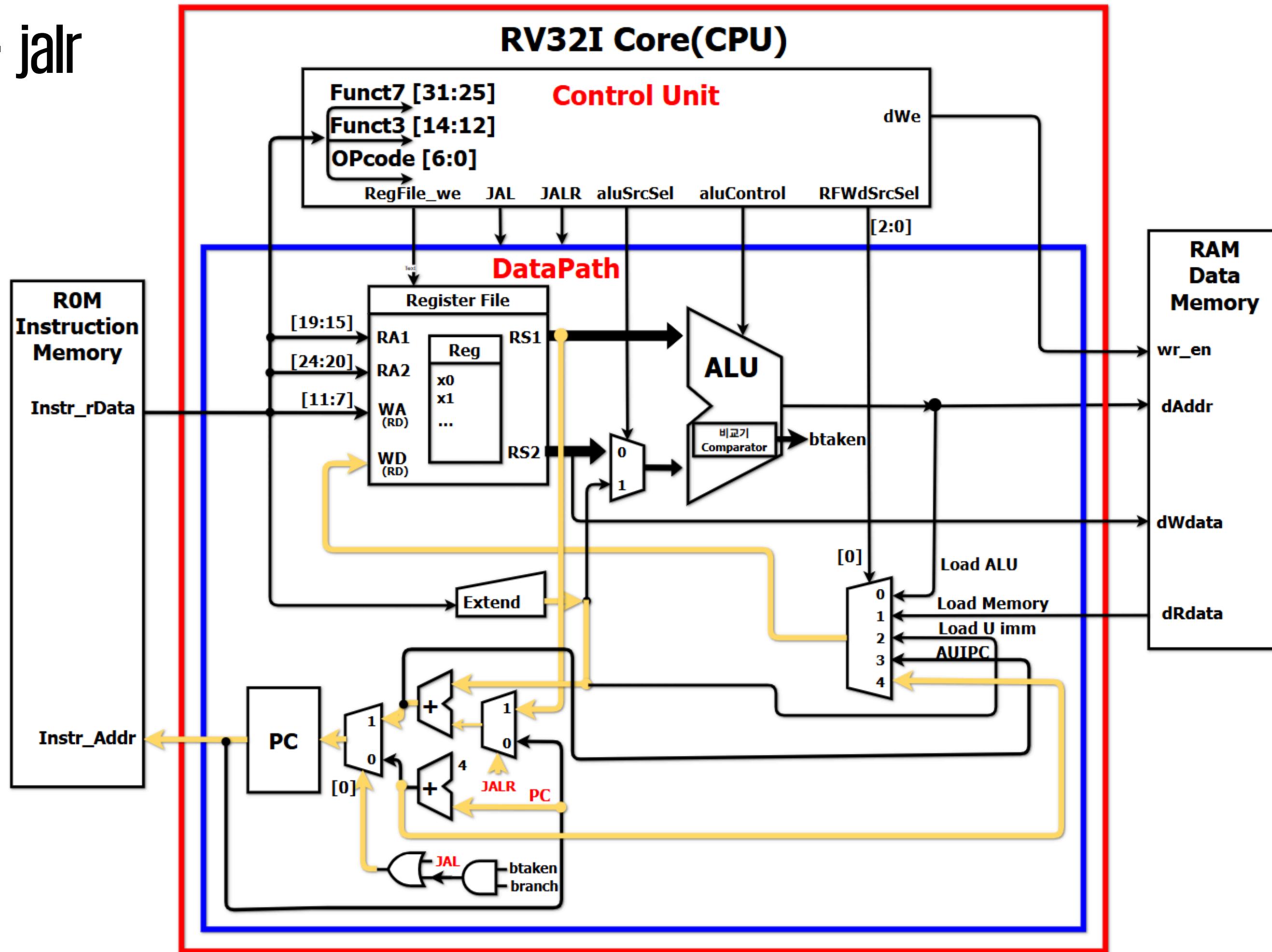
$$\text{LUI} \rightarrow rd(\text{write_addr}) = 4096$$

$$\text{AUIPC} \rightarrow 120 + 8192 = 8312$$

J Type - jal



J Type - jalr



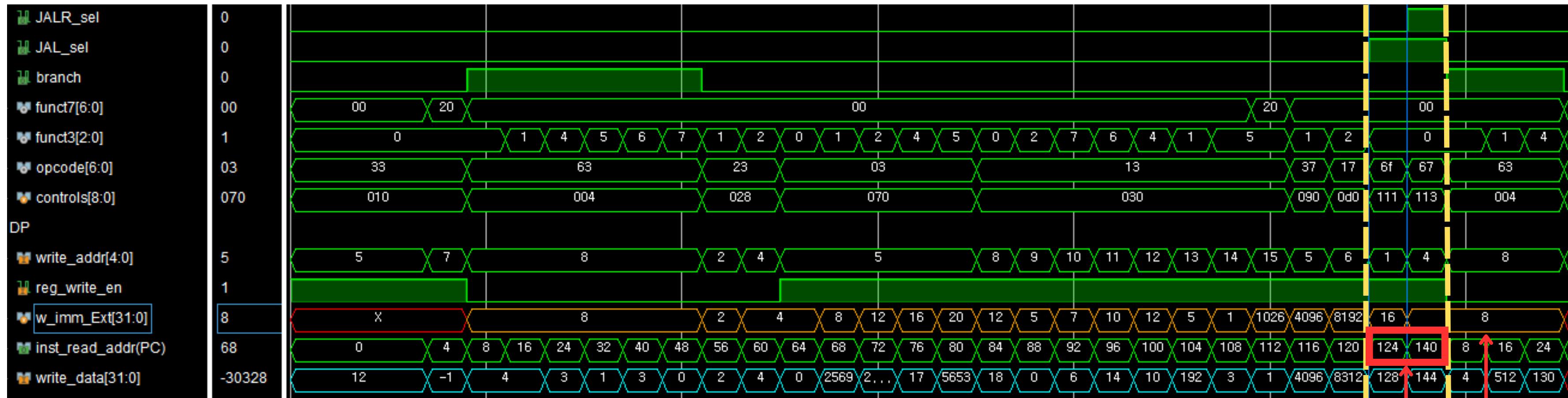
RISC-V Type

J Type

| | | | | | |
|------|--------------------------------|-----|----------------|----|----------------|
| JAL | imm [20:0] [10:1] [11] [19:12] | rd | opcode 1101111 | | |
| JALR | imm [11:0] | rs1 | funct3_000 | rd | opcode 1100111 |

- JAL
 - $rd = PC + 4; PC += imm$
 - 현재 PC 값 + 4 (다음 명령어 주소)를 rd 레지스터에 저장
 - sign-extend 된 즉시값(imm)을 PC에 더해 점프 대상 주소로 이동
- 레지스터 사용
 - $x1 (ra) \rightarrow$ return address 저장용 (함수 호출 시 복귀할 주소)
 - $jal x1, func \ # ra(x1)에 PC+4$ 저장, func로 점프
- JALR
 - $rd = PC + 4; PC = rs1 + imm$
 - $rd = PC + 4 \rightarrow$ 다음 명령어 주소를 rd에 저장
 - $PC = (rs1 + sign-extended imm) & \sim 1 \rightarrow$ 점프할 주소 계산 후 이동
 - JAL과 달리 즉시값은 rs1 레지스터 값으로 기준으로 더해짐
 - $jalr x1, x2, 0 \ # x1에 PC+4$ 저장, $x2+0$ 으로 점프

Simulation



```
// J-type
// rd = PC+4; PC += imm
rom[31] = 32'b000000000001_00000000_0000_0001_1101111; // JAL x16, 16
rom[35] = 32'b000000010000000000001001100111; // JALR x4, 8(x0) 8+x0 = 8 -> 8/4 = 2번지로
```

PC값 16증가

```
// B-type
// 32'b imm(7bit) _ rs2(5bit) _ rs1(5bit) _ funct3(3bit) _ imm (5bit) _ opcode(7'b1100011)
rom[2] = 32'b000000_00010_00010_000_01000_1100011; // BEQ x2, x2, 8
rom[4] = 32'b0000000_00010_00001_001_01000_1100011; // BNE x1, x2, 8
```

2번지이동 >

index:2 의 BEQ 명령어 실행으로 pc+8

J-Type Verification

```
task run(int n, logic [7:0] virtual_mem[0:255]);
    repeat (n) begin
        tr = new();

        // 이전 명령어의 목적지 레지스터(prev_write_addr)를
        // 현재 명령어의 수수 레지스터(rs1)로 사용하지 않도록 제약
        if (prev_write_addr != 0) begin
            assert (tr.randomize() with {rs1 != prev_write_addr;});
        end else begin
            assert (tr.randomize());
        end

        tr.build_instr();
    end
endtask
```

```
task run();
    forever begin
        transaction tr;
        gen2drv.get(tr);

        @(posedge cpu_if.clk);
        #1ps;

        // rs1_val 읽고 expected 계산 (JALR일 경우에만 필요)
        if (!tr.is_jal) begin
            tr.rs1_val = tb_verifi_cpu.dut.U_RV32I_CPU.U_DATAPATH.U_REG_FILE.reg_file[tr.rs1];
        end

        tr.calc_expected(cpu_if.pc);

        $display(
            "%0t [DRV] Driving inst=%0d (%s) | rs1_val=%0d | exp_next_pc=%0d | exp_rd_val=%0d",
            $time, tr.inst_code, tr.get_inst_name(), tr.rs1_val,
            tr.exp_next_pc, tr.exp_rd_val);

        cpu_if.inst_code = tr.inst_code;
        drv2mon.put(tr);
    end
endtask
```

Generator

랜덤으로 JAL / JALR 선택, rd, rs1, imm 값을 결정.

생성한 트랜잭션은 driver에게 전달(gen2drv.put(tr)).

이전 명령어에서 rd로 쓴 값이 아직 레지스터에 반영되기 전에,
다음 명령어에서 rs1로 읽는 상황을 방지

Driver

generator가 만든 트랜잭션(transaction)을 받아서 CPU 인터페이스
(cpu_if)에 반영

inst_code를 CPU에 넣고, JALR일 경우 rs1 값 읽어서 예상값 계산

DUT에 적용 후, 트랜잭션을 모니터에게 전달 (drv2mon.put(tr))

J-Type Verification

```
task run(int n);
  repeat (n) begin
    transaction tr;
    drv2mon.get(tr);

    @(posedge cpu_if.clk);
    tr.actual_next_pc = cpu_if.next_pc;

    if (cpu_if.reg_write_en && cpu_if.write_addr == tr.write_addr) begin
      tr.actual_rd_val = cpu_if.write_data;
    end else begin
  task run(int n);
  repeat (n) begin
    mon2scb.get(tr);

    if (tr.actual_next_pc === tr.exp_next_pc && tr.actual_rd_val === tr.exp_rd_val) begin
      pass_count++;
      $display("[SCB PASS] %s | NextPC=%0d | Rd_Val=%0d",
               tr.get_inst_name(), tr.actual_next_pc,
               tr.actual_rd_val);
    end else begin
      fail_count++;
      $display(
        "[SCB FAIL] %s | ACTUAL NextPC=%0d Rd_Val=%0d | EXP NextPC=%0d Rd_Val=%0d",
        tr.get_inst_name(), tr.actual_next_pc, tr.actual_rd_val,
        tr.exp_next_pc, tr.exp_rd_val);
    end
    ->gen_next_event;
  end
  $display("-----");
  $display("SUMMARY :: TOTAL=%0d | PASS=%0d | FAIL=%0d",
          pass_count + fail_count, pass_count, fail_count);
  $display("-----");
endtask
```

Monitor

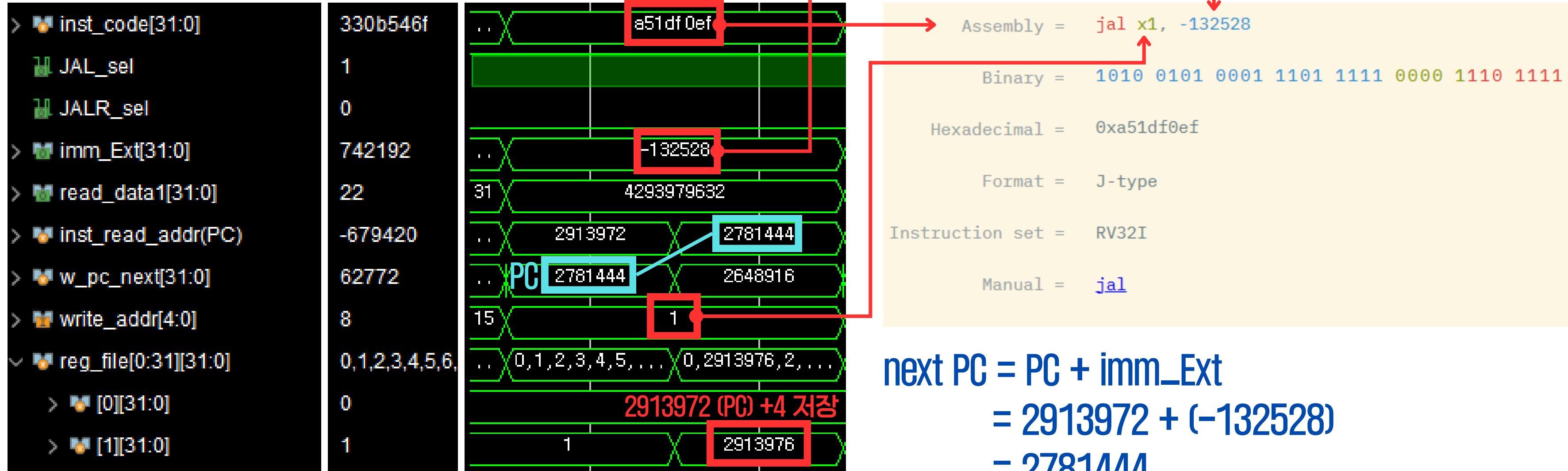
DUT에서 나온 신호를 관찰하고 트랜잭션에 실제값(actual)을 기록

1. 트랜잭션 수신: `drv2mon.get(tr)` → driver에서 보낸 트랜잭션을 받음
2. 클럭 대기
3. `next_pc`, `rd` 값 샘플링
4. Scoreboard로 값 전달

Scoreboard

scoreboard는 모니터로부터 실제 DUT 동작 트랜잭션을 받아 예상값과 비교하고, 테스트 결과를 통계로 관리하여 다음 트랜잭션 생성을 제어

J-Type Verification



| | |
|------|-----------------------------|
| JAL | $rd = PC+4; PC += imm$ |
| JALR | $rd = PC+4; PC = rs1 + imm$ |

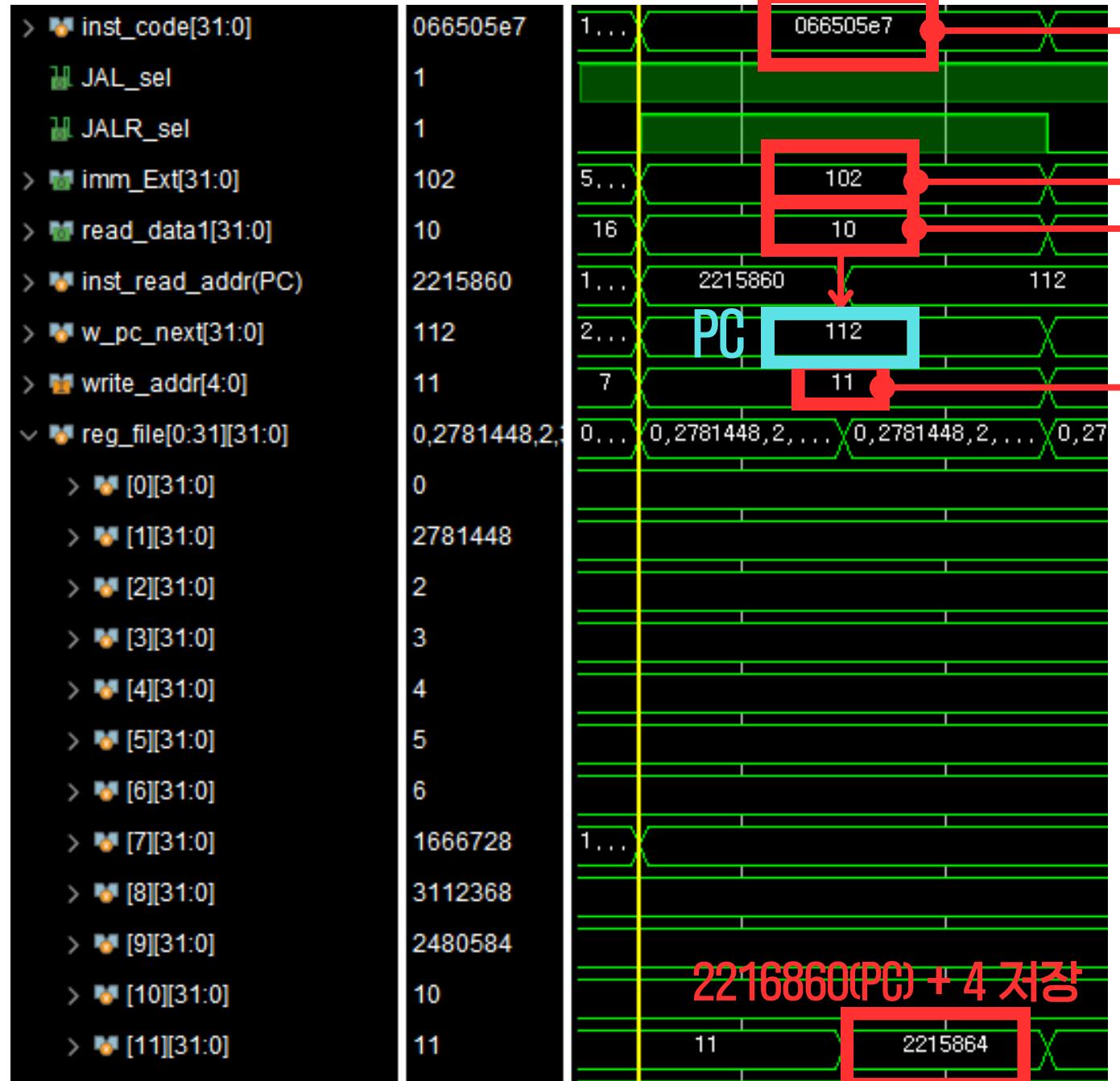
```

305000 [GEN] instr=0xa51df0ef (JAL) | rd=x1 | rs1=xx
315000 [DRV] Driving inst=2770202863 (JAL) | rs1_val=x | exp_next_pc=2781444 | exp_rd_val=2913976
325000 [MON] ACTUAL NextPC=2781444 | Rd_Val=2913976 -- EXPECTED NextPC=2781444 | Rd_Val=2913976
[SCB PASS] JAL | NextPC=2781444 | Rd_Val=2913976

```

SUMMARY :: TOTAL=50 | PASS=50 | FAIL=0

J-Type Verification



$$\begin{aligned} \text{next PC} &= \text{rs1} + \text{imm_Ext} \\ &= 10 + 102 \\ &= 112 \end{aligned}$$

```
465000 [GEN] instr=0x66505e7(JALR) | rd=x11 | rs1=x10
475000 [DRV] Driving inst=0x66505e7 (JALR) | rs1_val=10 | exp_next_pc=112 | exp_rd_val=2215864
485000 [MON] ACTUAL NextPC=112 | Rd_Val=2215864 -- EXPECTED NextPC=112 | Rd_Val=2215864
```

[SCB PASS] JALR | NextPC=112 | Rd_Val=2215864

| | |
|------|-----------------------------|
| JAL | $rd = PC+4; PC += imm$ |
| JALR | $rd = PC+4; PC = rs1 + imm$ |

SUMMARY :: TOTAL=50 | PASS=50 | FAIL=0

Trouble Shooting

L-type 검증 중 Byte 단위 동작으로 설계 했는데 값이 넘어가는 경우가 많아서 검증이 확실치 않음

```
task run(int n);
    repeat (n) begin
        transaction tr;
        drv2mon.get(tr);
        @(posedge vif.clk);
        #1;
```

검증 테스트 벤치 실행 중 1ns를 기다리는 동작이 있는데 이건 정확히 상승엣지에 값을 읽어올 경우 이전의 값을 캡쳐해 1ns만큼 지연