# RISC-V RV32I CPU 설계

황석현

# 목차

- 프로젝트 목표
- RISC-V
- RV32I
- Block diagram & 시뮬레이션

# 프로젝트 목표

- CPU의 구성 요소를 Verilog HDL을 사용하여 모듈 단위로 구현하고 통합
- RV32I의 타입 명령어 형식 분석
- 명령어 타입에 맞게 Control Unit의 제어 신호가 정확하게 동작하고,
  데이터가 DataPath를 따라 올바르게 처리되는지 파형을 분석하여 확인하고 디버깅


개발 언어
: System Verilog


Simulation Tool
: Xilinx VIVADO

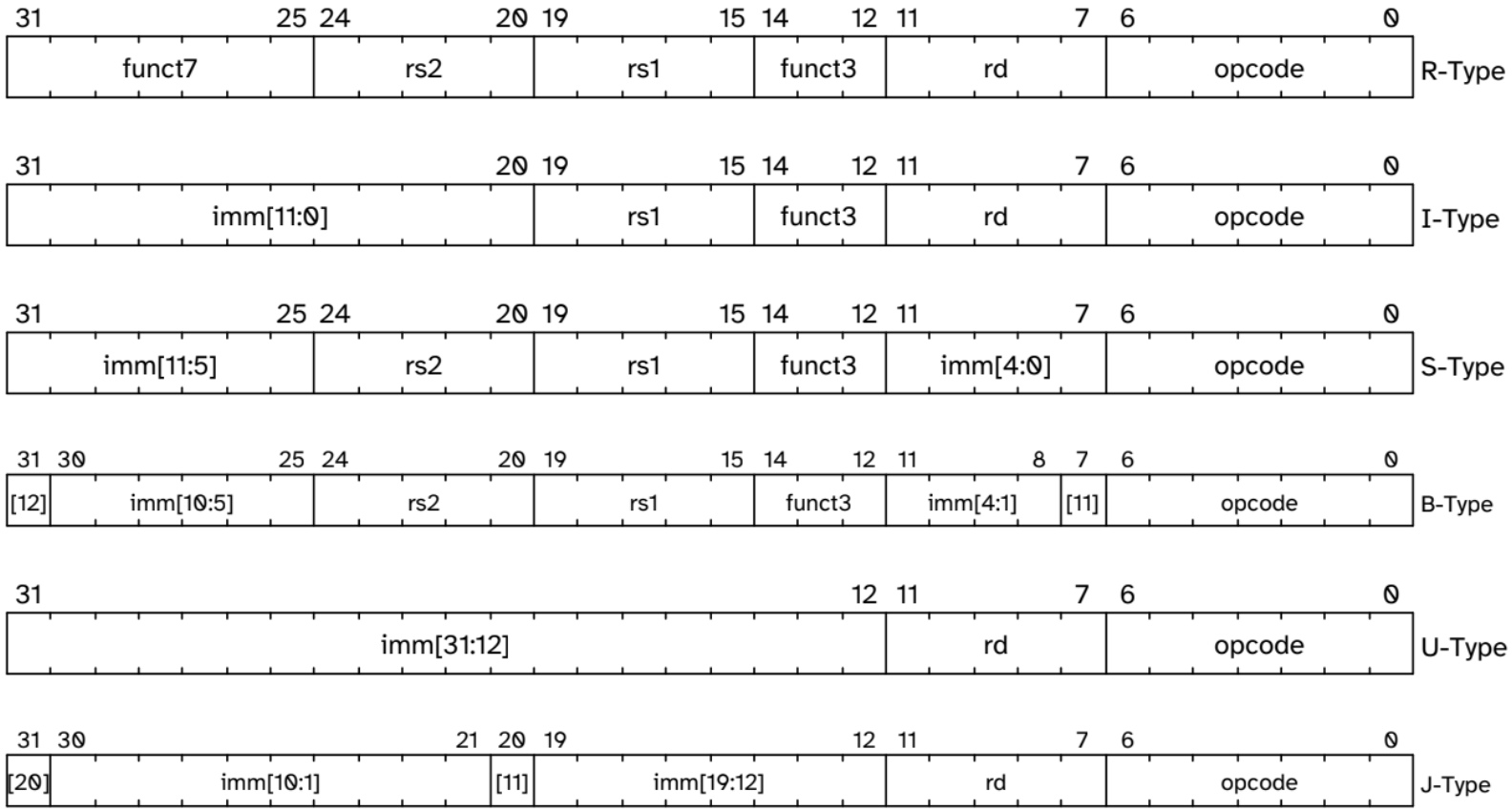
# RISC-V

오픈소스 명령어 집합 구조(ISA)
라이선스 비용이 없고, 특정 기업에 종속되지 않음.

- 특징
  1. 단순함 : 꼭 필요한 최소한의 명령어로 구성되어 있음.
  2. 모듈성 : 필요한 기능(모듈)만 선택하여 조합 가능함.
  3. 확장성 : 사용자가 직접 필요한 명령어를 추가할 수 있음.

# RV32I

: **R**ISC-**V 32**-bit base **I**nteger의 약자로, 가장 기본적인 32비트 정수 명령어 세트




출처 : https://cpu.kanataso.net/04-impl-rv32i.html

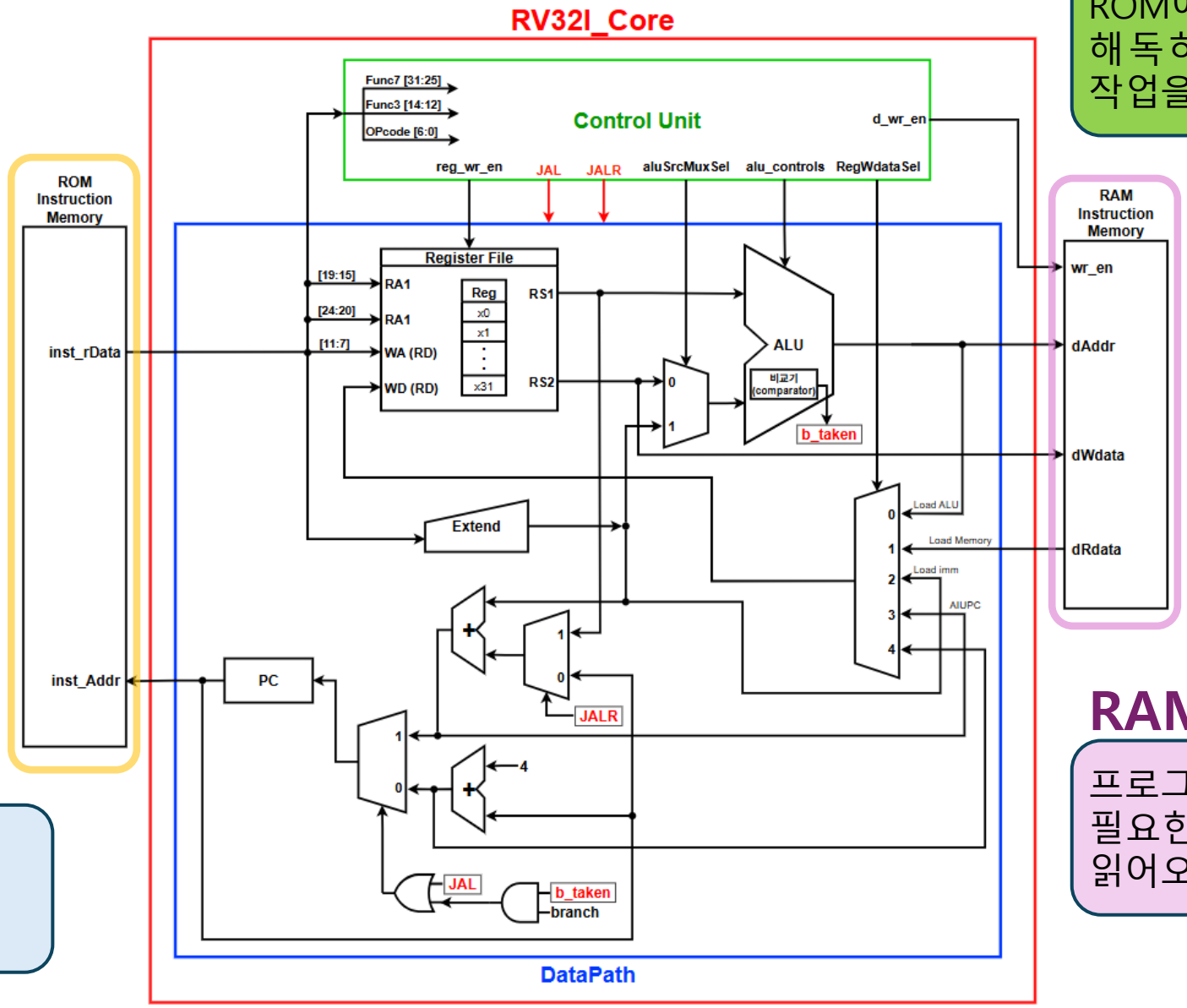
# RV32I

## ROM

실행할 명령어(Instruction) 저장되어 있음

## DataPath

Control Unit 신호를 받아 작업 처리




## Control Unit

ROM에서 전달받은 명령어를 해독하여, DataPath가 어떤 작업을 할지 결정하고 지시

## RAM

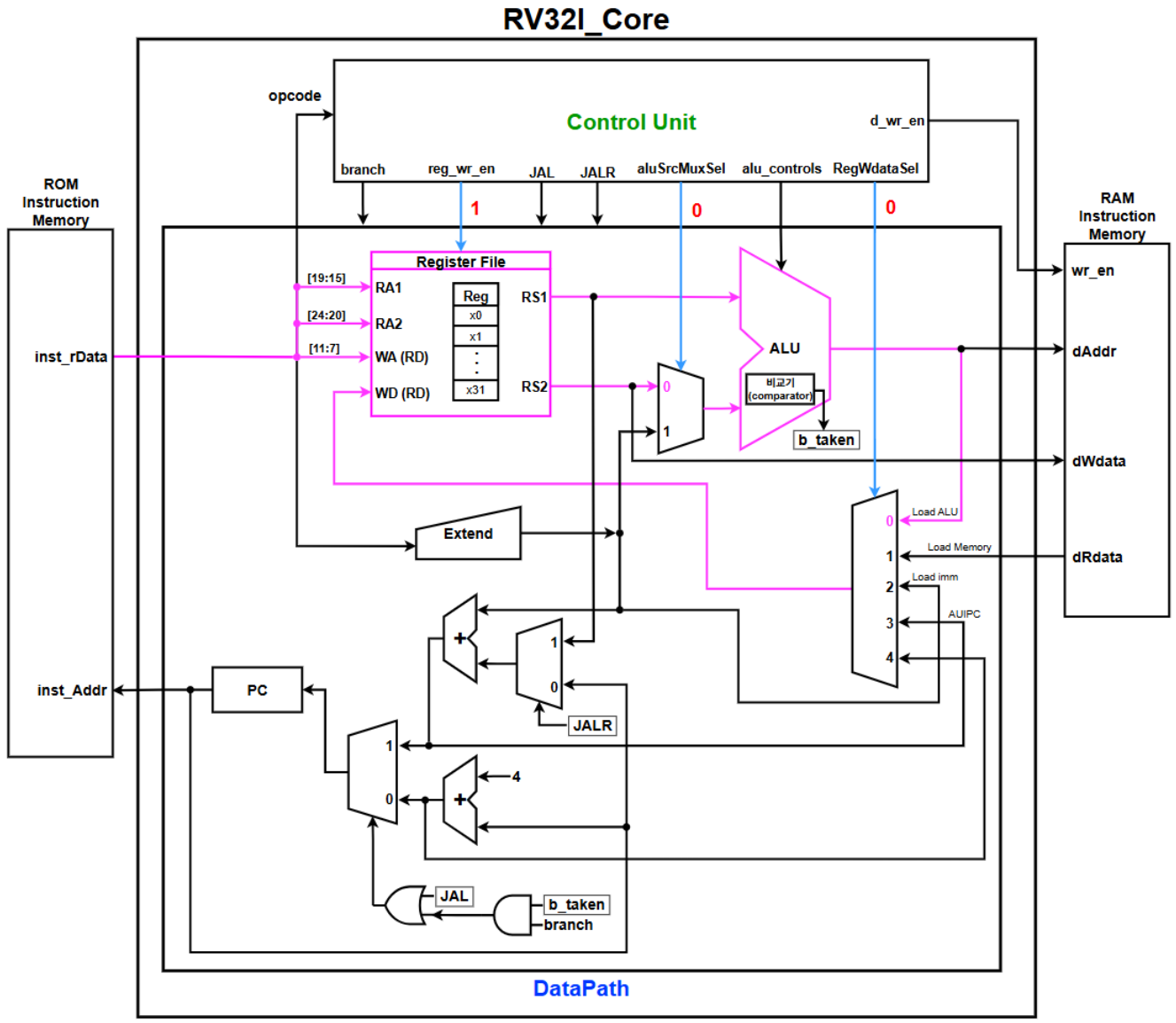
프로그램이 실행되는 동안 필요한 데이터를 저장하거나 읽어오는 공간

# 1. R-type

: 레지스터와 레지스터 간의 산술 및 논리 연산을 위해 사용되는 instruction format

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| funct7 | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | rd | | | | | opcode | | | | | | | name | Descript | Note |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | rs2 | | | | | rs1 | | | | | 0 | 0 | 0 | rd | | | | | 0 | 1 | 1 | 0 | 0 | 1 | 1 | ADD | rd = rs1 + rs2 | |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | rs2 | | | | | rs1 | | | | | 0 | 0 | 0 | rd | | | | | 0 | 1 | 1 | 0 | 0 | 1 | 1 | SUB | rd = rs1 - rs2 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | rs2 | | | | | rs1 | | | | | 0 | 0 | 1 | rd | | | | | 0 | 1 | 1 | 0 | 0 | 1 | 1 | SLL | rd = rs1 << rs2 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | rs2 | | | | | rs1 | | | | | 0 | 1 | 0 | rd | | | | | 0 | 1 | 1 | 0 | 0 | 1 | 1 | SLT | rd = (rs1 < rs2)?1:0 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | rs2 | | | | | rs1 | | | | | 0 | 1 | 1 | rd | | | | | 0 | 1 | 1 | 0 | 0 | 1 | 1 | SLTU | rd = (rs1 < rs2)?1:0 | zero-extends |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | rs2 | | | | | rs1 | | | | | 1 | 0 | 0 | rd | | | | | 0 | 1 | 1 | 0 | 0 | 1 | 1 | XOR | rd = rs1 ^ rs2 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | rs2 | | | | | rs1 | | | | | 1 | 0 | 1 | rd | | | | | 0 | 1 | 1 | 0 | 0 | 1 | 1 | SRL | rd = rs1 >> rs2 | |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | rs2 | | | | | rs1 | | | | | 1 | 0 | 1 | rd | | | | | 0 | 1 | 1 | 0 | 0 | 1 | 1 | SRA | rd = rs1 >> rs2 | msb-extends |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | rs2 | | | | | rs1 | | | | | 1 | 1 | 0 | rd | | | | | 0 | 1 | 1 | 0 | 0 | 1 | 1 | OR | rd = rs1 \| rs2 | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | rs2 | | | | | rs1 | | | | | 1 | 1 | 1 | rd | | | | | 0 | 1 | 1 | 0 | 0 | 1 | 1 | AND | rd = rs1 & rs2 | |

# 1. R-type




- **데이터 흐름**

1. ROM에서 명령어 인출.
2. 두 레지스터(RS1, RS2) 값 읽음.
3. ALU는 RS1과 RS2 값을 입력으로 받아 연산 수행.
4. ALU의 연산 결과는 mux 0번 입력을 통해 Register_File의 데이터로 저장.

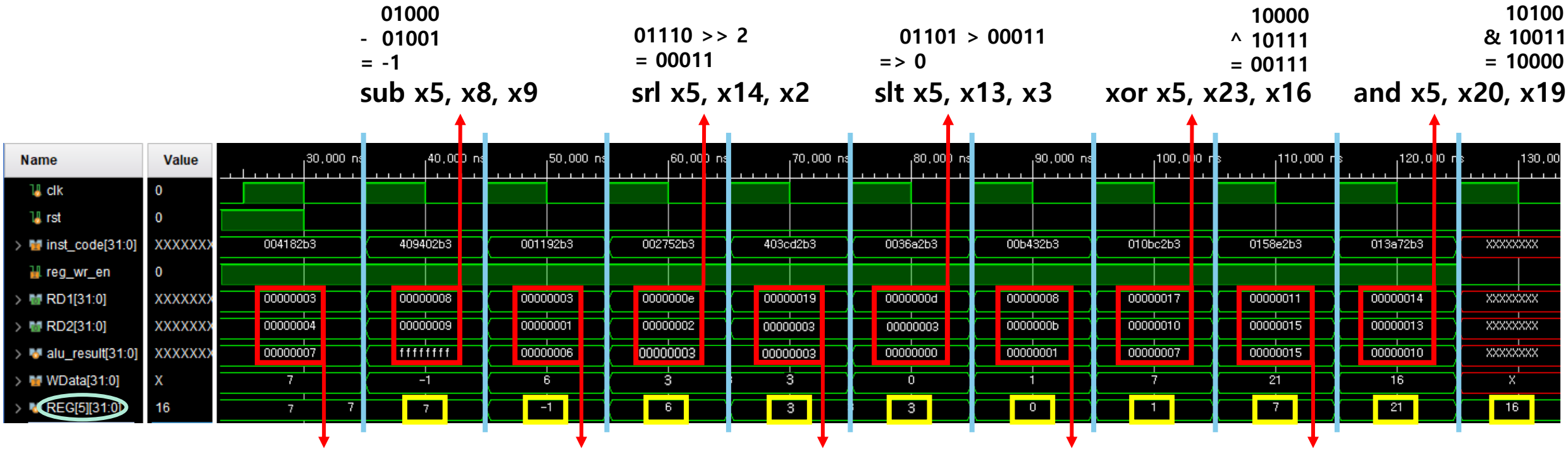
- **신호 제어**

reg_wr_en = 1
: 연산 결과를 Register_File에 써야 하므로 활성화.

aluSrcMuxSel = 0
: 레지스터 RS2 값 선택.

RegWdataSel = 0
: Reg_file에 쓸 데이터로 ALU의 연산 결과를 선택.

# 1. R-type Simulation




| name | Descript |
|---|---|
| ADD | rd = rs1 + rs2 |
| SUB | rd = rs1 - rs2 |
| SLL | rd = rs1 << rs2 |
| SLT | rd = (rs1 < rs2)?1:0 |
| SLTU | rd = (rs1 < rs2)?1:0 |
| XOR | rd = rs1 ^ rs2 |
| SRL | rd = rs1 >> rs2 |
| SRA | rd = rs1 >> rs2 |
| OR | rd = rs1 \| rs2 |
| AND | rd = rs1 & rs2 |

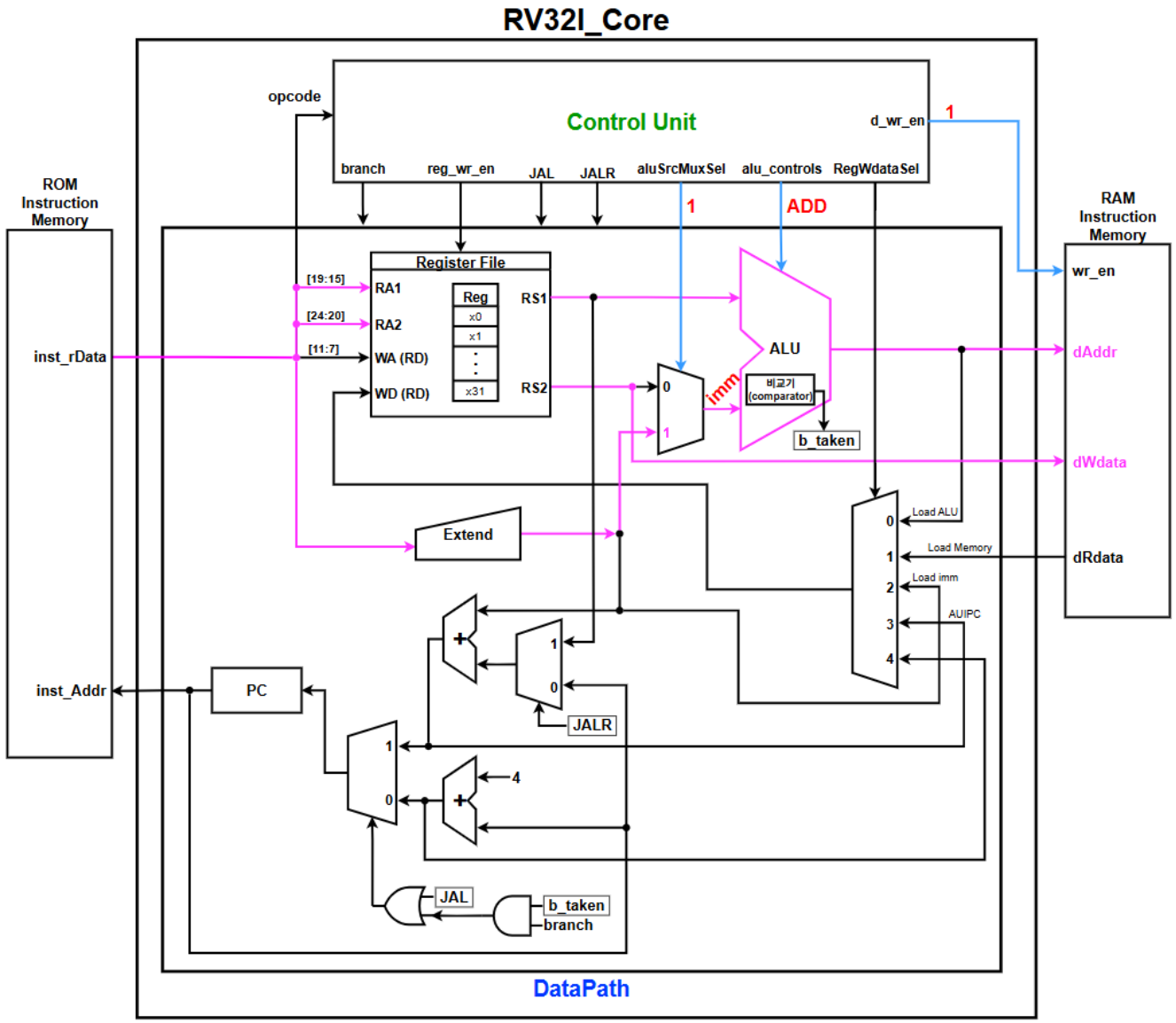
rs1, rs2가 ALU를 거쳐 계산된 값 alu_result가 register_file(REG)에 할당된다.

# 2. S-type

: 레지스터의 데이터를 메모리에 저장(Store)하기 위해 사용되는 instruction format

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| imm[12,10:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:1,11] | | | | | opcode | | | | | | | name | Descript |
| imm[11:5] | | | | | | | rs2 | | | | | rs1 | | | | | 0 | 0 | 0 | imm[4:0] | | | | | 0 | 1 | 0 | 0 | 0 | 1 | 1 | SB | M[rs1+imm][0:7] = rs2[0:7] |
| imm[11:5] | | | | | | | rs2 | | | | | rs1 | | | | | 0 | 0 | 1 | imm[4:0] | | | | | 0 | 1 | 0 | 0 | 0 | 1 | 1 | SH | M[rs1+imm][0:15] = rs2[0:15] |
| imm[11:5] | | | | | | | rs2 | | | | | rs1 | | | | | 0 | 1 | 0 | imm[4:0] | | | | | 0 | 1 | 0 | 0 | 0 | 1 | 1 | SW | M[rs1+imm][0:31] = rs2[0:31] |

# 2. S-type




- **데이터 흐름**

1. ROM에서 명령어 인출.
2. RS1과 imm이 ALU에서 더해져 데이터를 저장할 주소 계산.
3. RAM에 저장할 데이터인 RS2값을 RAM에 전달.

- **신호 제어**

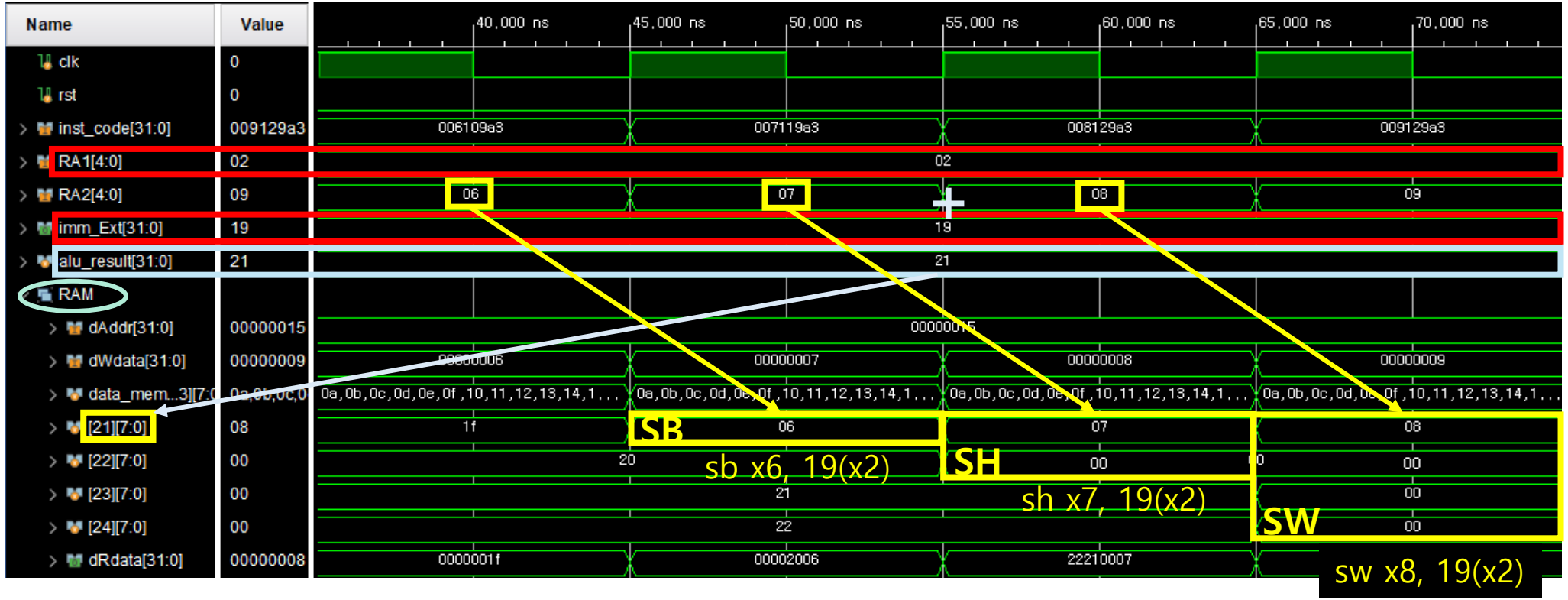
aluSrcMuxSel = 1
: imm 값 선택.

d_wr_en = 1
: RAM에 데이터를 써야 하므로 활성화.

# 2. S-type Simulation

```
initial begin
    for (int i = 0; i < 32; i++) begin
        reg_file[i] = i;
    end
end
```

레지스터와 주소의 값이 같음.




RAM Byte Align
rs2의 값이 RAM 주소 (rs1+imm)에 store됨.
SB : 1Byte / SH : 2Byte / SW : 4Byte 저장.

| name | Descript |
|---|---|
| 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] |

# 3. I-type

: 레지스터와 상수(Immediate) 값 연산이나 메모리 Load를 위해 사용되는 instruction format

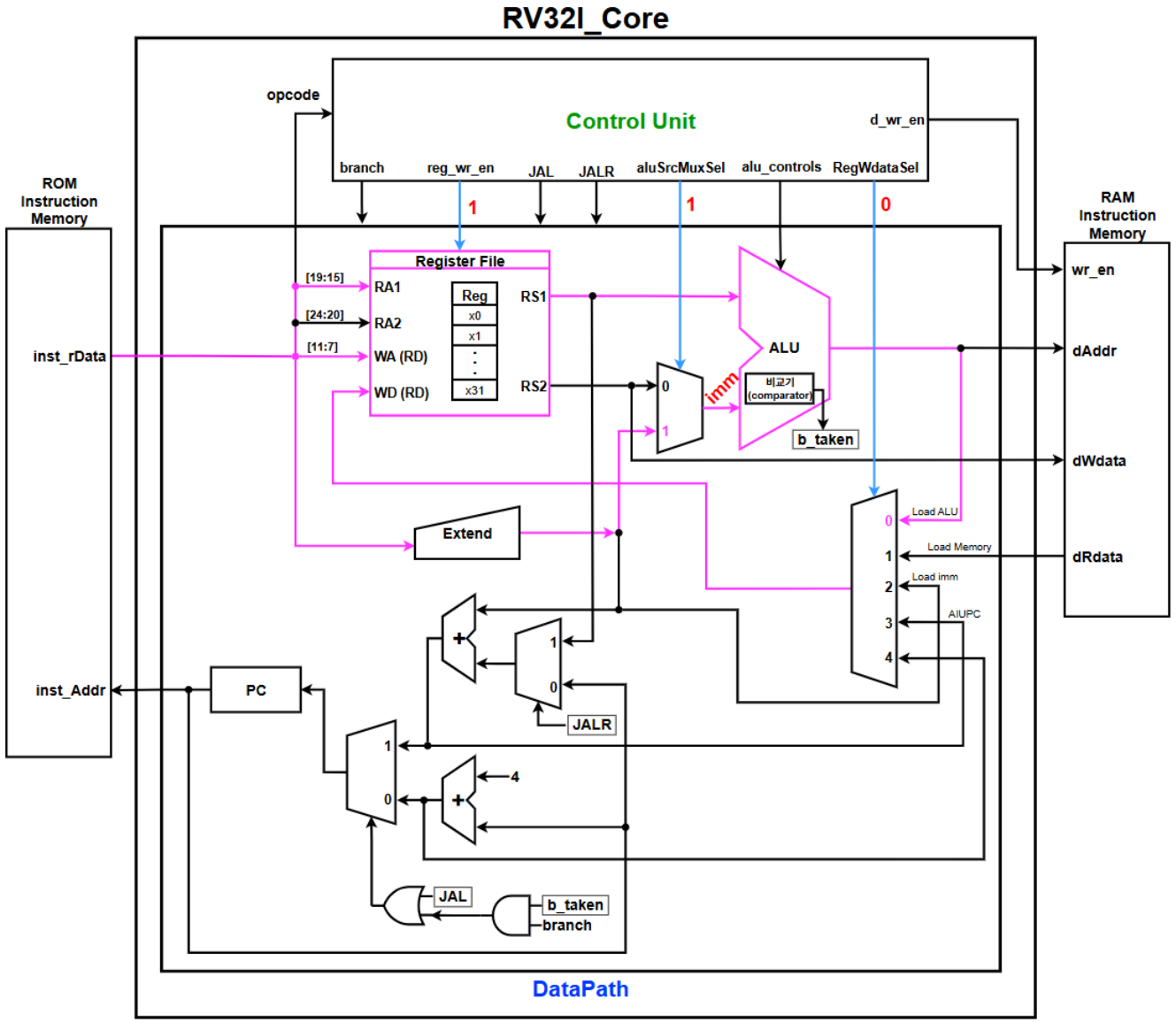
I

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | name | Descript | Note |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | funct3 | | | rd | | | | | opcode | | | | | | | | | | |
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | 0 | 0 | 0 | rd | | | | | 0 | 0 | 1 | 0 | 0 | 1 | 1 | ADDI | rd = rs1 + imm | |
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | 0 | 1 | 0 | rd | | | | | 0 | 0 | 1 | 0 | 0 | 1 | 1 | SLTI | rd = (rs1 < imm)?1:0 | |
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | 0 | 1 | 1 | rd | | | | | 0 | 0 | 1 | 0 | 0 | 1 | 1 | SLTIU | rd = (rs1 < imm)?1:0 | zero-extends |
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | 1 | 0 | 0 | rd | | | | | 0 | 0 | 1 | 0 | 0 | 1 | 1 | XORI | rd = rs1 ^ imm | |
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | 1 | 1 | 0 | rd | | | | | 0 | 0 | 1 | 0 | 0 | 1 | 1 | ORI | rd = rs1 \| imm | |
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | 1 | 1 | 1 | rd | | | | | 0 | 0 | 1 | 0 | 0 | 1 | 1 | ANDI | rd = rs1 & imm | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | shamt | | | | | rs1 | | | | | 0 | 0 | 1 | rd | | | | | 0 | 0 | 1 | 0 | 0 | 1 | 1 | SLLI | rd = rs1 << imm[0:4] | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | shamt | | | | | rs1 | | | | | 1 | 0 | 1 | rd | | | | | 0 | 0 | 1 | 0 | 0 | 1 | 1 | SRLI | rd = rs1 >> imm[0:4] | |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | shamt | | | | | rs1 | | | | | 1 | 0 | 1 | rd | | | | | 0 | 0 | 1 | 0 | 0 | 1 | 1 | SRAI | rd = rs1 >> imm[0:4] | msb-extends |

IL

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | name | Descript | Note |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | funct3 | | | rd | | | | | opcode | | | | | | | | | | |
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | 0 | 0 | 0 | rd | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | LB | rd = M[rs1+imm][0:7] | |
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | 0 | 0 | 1 | rd | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | LH | rd = M[rs1+imm][0:15] | |
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | 0 | 1 | 0 | rd | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | LW | rd = M[rs1+imm][0:31] | |
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | 1 | 0 | 0 | rd | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | LBU | rd = M[rs1+imm][0:7] | zero-extends |
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | 1 | 0 | 1 | rd | | | | | 0 | 0 | 0 | 0 | 0 | 1 | 1 | LHU | rd = M[rs1+imm][0:15] | zero-extends |

# 3. I-type ( I )




- **데이터 흐름**

1. ROM에서 명령어 인출.
2. RS1 값과 imm 값이 ALU를 거치며 연산.
3. 연산 결과를 mux 0번 입력을 통해 Register_File의 데이터로 저장.

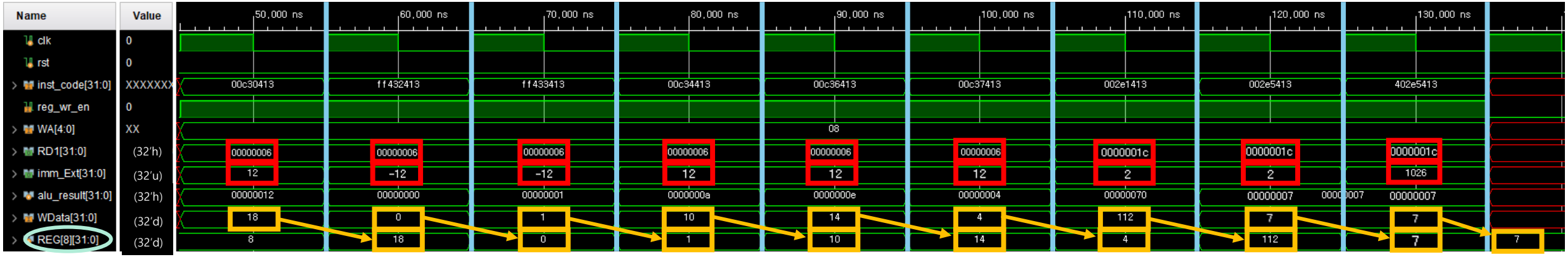
- **신호 제어**

reg_wr_en = 1
: 연산 결과를 Register_File에 써야 하므로 활성화.

aluSrcMuxSel = 1
: imm 값 선택.

RegWdataSel = 0
: Reg_file에 쓸 데이터로 ALU의 연산 결과를 선택.

# 3. I-type ( I ) Simulation




ADDI
x8, x6, 12

SLTI
x8, x6, -12

SLTIU
x8, x6, -12

XORI
x8, x6, 12

ORI
x8, x6, 12

ANDI
x8, x6, 12

SLLI
x8, x6, 12

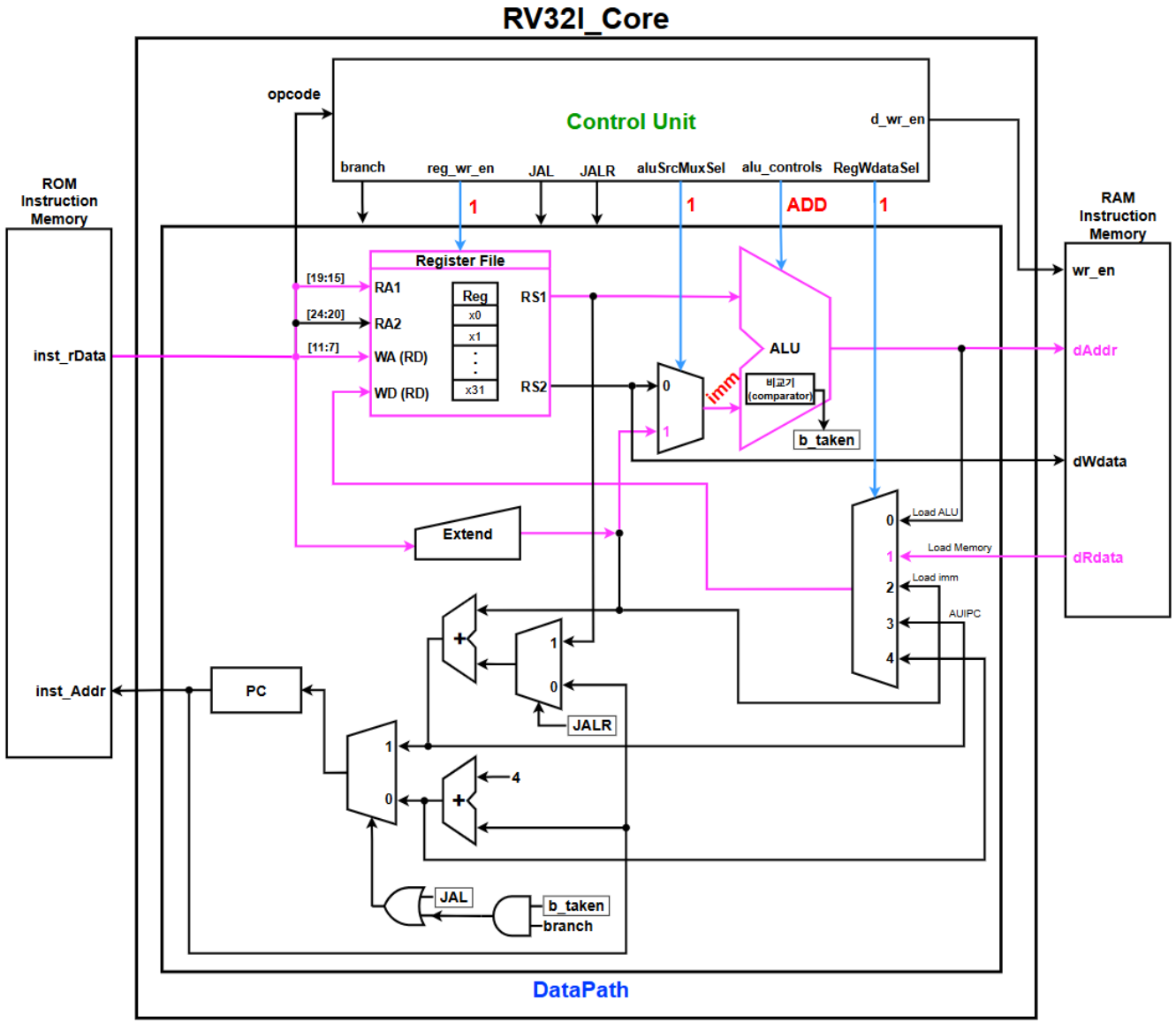
SRLI
x8, x6, 12

SRAI
x8, x6, 12

rs1, imm이 ALU를 거쳐 계산된 값이 register_file(REG)에 할당됨.

| name | Descript |
|---|---|
| ADDI | rd = rs1 + imm |
| SLTI | rd = (rs1 < imm)?1:0 |
| SLTIU | rd = (rs1 < imm)?1:0 |
| 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] |

# 3. I-type ( IL )




- **데이터 흐름**

1. ROM에서 명령어 인출.
2. ALU를 거치며 RS1 값과 imm을 더해 데이터를 가져올 주소로 사용.
3. RAM의 해당 주소에 저장된 데이터를 읽어 출력.
4. Mux 1번 신호로 데이터를 register_file에 저장.

- **신호 제어**

reg_wr_en = 1
: 메모리에서 가져온 데이터를 Register_File에 써야 하므로 활성화.

aluSrcMuxSel = 1
: 메모리 주소( rs1+imm )를 계산하기 위해 imm 값 선택.

RegWdataSel = 1
: 메모리에서 읽은 값 선택.

# 3. I-type ( IL ) Simulation

```
initial begin
    for (int j = 0; j < 64; j++) begin
        data_mem[j] = j + 8'h0A;
    end
end
```

레지스터와 주소의 값이 같음.


lb x5, 12(x2) lh x6, 12(x2) lw x7, 12(x2) lbu x8, 12(x2) lhu x9, 12(x2)

RAM data는 주소 + 8'h0A로 초기화되어 있음.
RAM[rs1+imm]의 값을 reg_file(REG)에 load.
LB, LBU : 1Byte / LH, LHU : 2Byte / LW : 4Byte 출력.

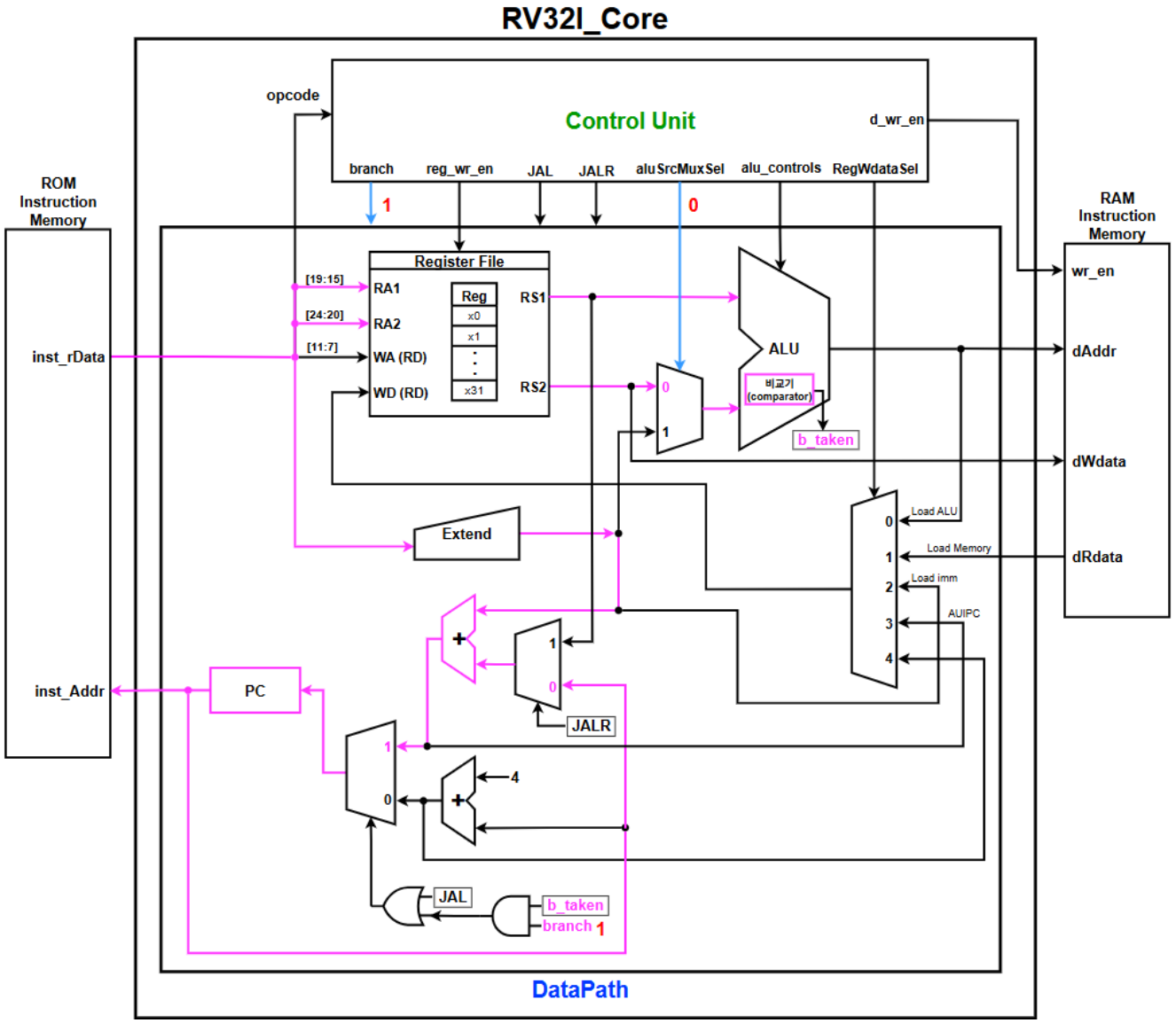
| name | Descript |
|---|---|
| LB | rd = M[rs1+imm][0:7] |
| LH | rd = M[rs1+imm][0:15] |
| LW | rd = M[rs1+imm][0:31] |
| LBU | rd = M[rs1+imm][0:7] |
| LHU | rd = M[rs1+imm][0:15] |

# 4. B-type

: 두 레지스터 값을 비교한 결과에 따라 조건부로 특정 주소로 분기(Branch)하기 위해 사용되는 instruction format

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| imm[12,10:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:1,11] | | | | | opcode | | | | | | | name | Descript | Note |
| imm[12,10:5] | | | | | | | rs2 | | | | | rs1 | | | | | 0 | 0 | 0 | imm[4:1,11] | | | | | 1 | 1 | 0 | 0 | 0 | 1 | 1 | BEQ | if(rs1 == rs2) PC += imm | |
| imm[12,10:5] | | | | | | | rs2 | | | | | rs1 | | | | | 0 | 0 | 1 | imm[4:1,11] | | | | | 1 | 1 | 0 | 0 | 0 | 1 | 1 | BNE | if(rs1 != rs2) PC += imm | |
| imm[12,10:5] | | | | | | | rs2 | | | | | rs1 | | | | | 1 | 0 | 0 | imm[4:1,11] | | | | | 1 | 1 | 0 | 0 | 0 | 1 | 1 | BLT | if(rs1 < rs2) PC += imm | |
| imm[12,10:5] | | | | | | | rs2 | | | | | rs1 | | | | | 1 | 0 | 1 | imm[4:1,11] | | | | | 1 | 1 | 0 | 0 | 0 | 1 | 1 | BGE | if(rs1 >= rs2) PC += imm | |
| imm[12,10:5] | | | | | | | rs2 | | | | | rs1 | | | | | 1 | 1 | 0 | imm[4:1,11] | | | | | 1 | 1 | 0 | 0 | 0 | 1 | 1 | BLTU | if(rs1 < rs2) PC += imm | zero-extends |
| imm[12,10:5] | | | | | | | rs2 | | | | | rs1 | | | | | 1 | 1 | 1 | imm[4:1,11] | | | | | 1 | 1 | 0 | 0 | 0 | 1 | 1 | BGEU | if(rs1 >= rs2) PC += imm | zero-extends |

# 4. B-type




- **데이터 흐름**

1. ROM에서 명령어 인출.
2. 비교기는 RS1과 RS2 값을 이용하여 조건 참/거짓을 판단하고, 결과에 따라 b_taken 신호 생성.
4. b_taken 신호가 1이면 현재 주소(PC)를 PC와 imm을 더한 값으로, 0이면 PC=PC+4 로 업데이트

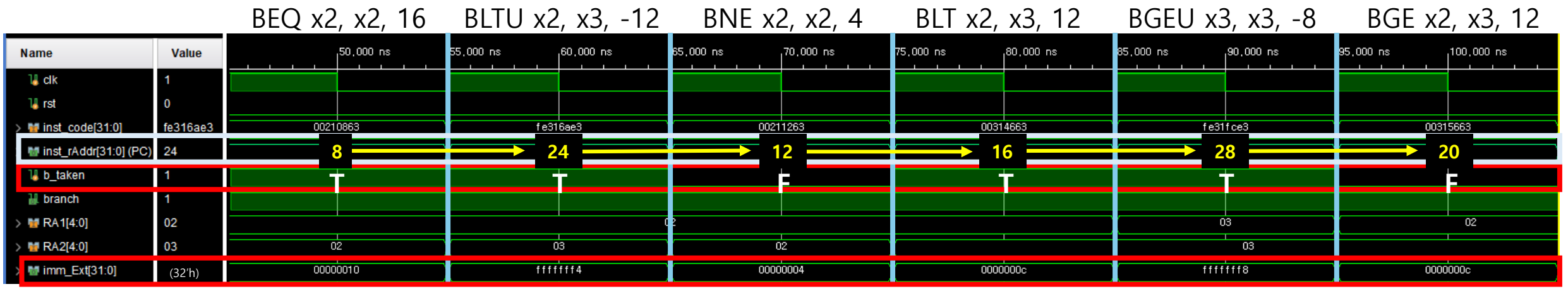
- **신호 제어**

branch = 1
: 분기 명령어임을 알림.

aluSrcMuxSel = 0
: RS2 값 선택.

# 4. B-type Simulation

레지스터와 주소의 값이 같음.




비교기 결과가 참이면 PC가 imm만큼 증가하고,
거짓이면 PC가 4 증가한다.

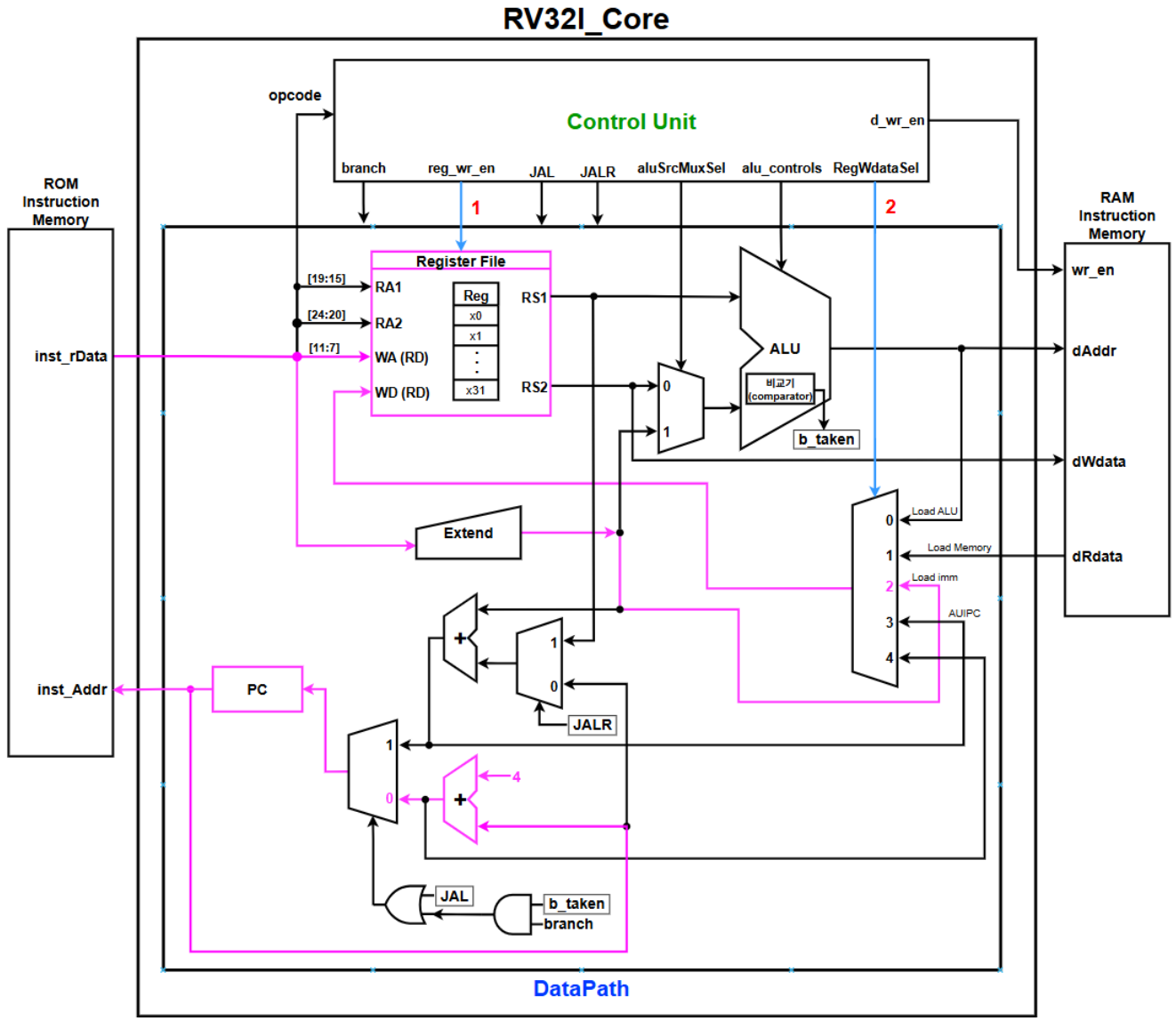
| name | Descript |
|---|---|
| BEQ | if(rs1 == rs2) PC += imm |
| BNE | if(rs1 != rs2) PC += imm |
| BLT | if(rs1 < rs2) PC += imm |
| BGE | if(rs1 >= rs2) PC += imm |
| BLTU | if(rs1 < rs2) PC += imm |
| BGEU | if(rs1 >= rs2) PC += imm |

# 5. U-type

: 긴 상수 값을 레지스터의 상위 20비트에 Load하기 위해 사용되는 instruction format

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| imm[31:12] | | | | | | | | | | | | | | | | | | | | rd | | | | | opcode | | | | | | | name | Descript |
| imm[31:12] | | | | | | | | | | | | | | | | | | | | rd | | | | | 0 | 1 | 1 | 0 | 1 | 1 | 1 | LUI | rd = imm |
| imm[31:12] | | | | | | | | | | | | | | | | | | | | rd | | | | | 0 | 0 | 1 | 0 | 1 | 1 | 1 | AUIPC | rd = PC + imm |

# 5. U-type (LUI)




- **데이터 흐름**

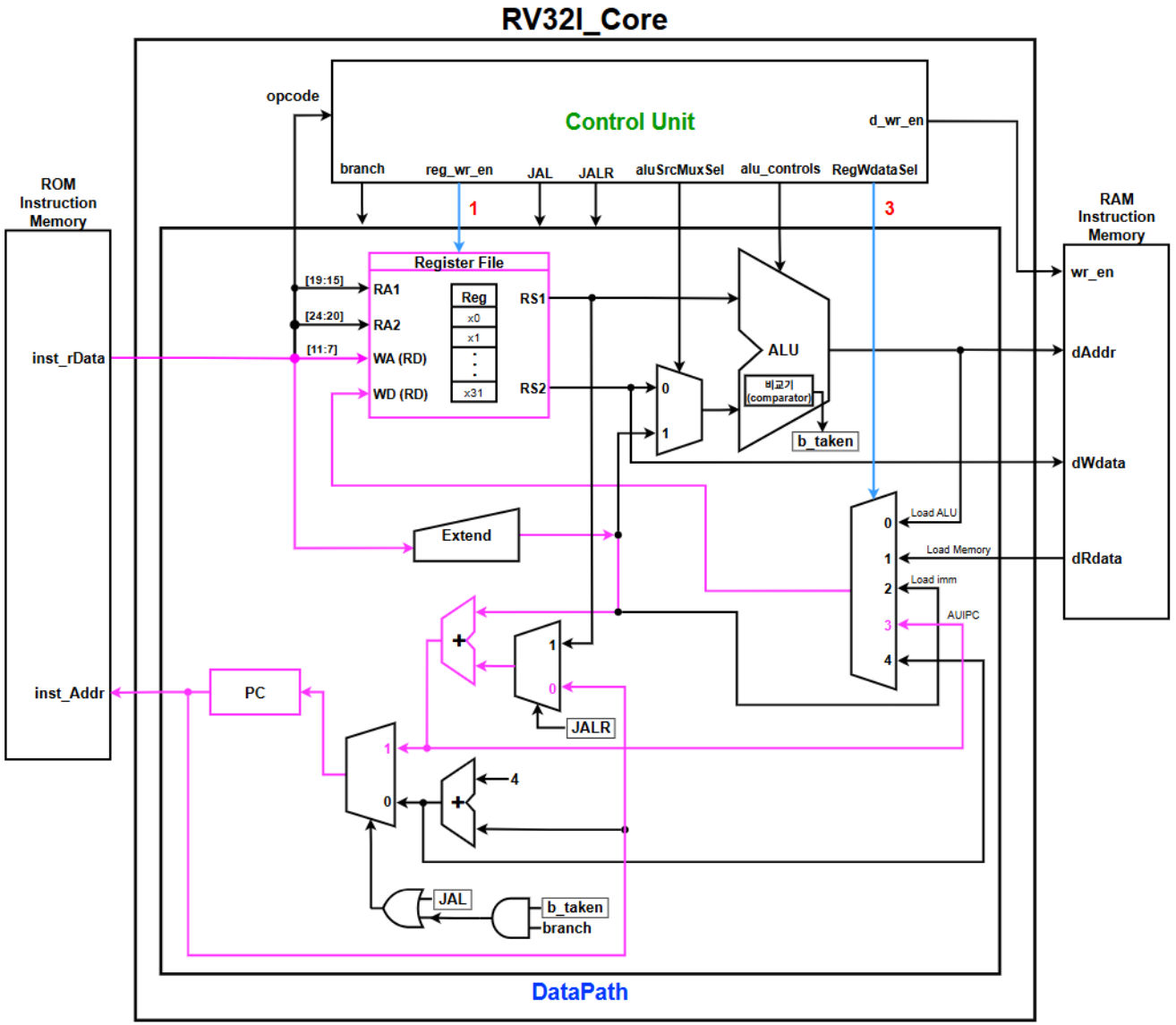
1. ROM에서 명령어 인출.
2. 20bit imm이 Extend 유닛을 거쳐 32bit로 확장.
   (하위 12bit는 0으로 채움)
3. 이 값은 ALU를 거치지 않고 mux 2번 입력으로
   바로 Register_file로 전달됨.

- **신호 제어**

reg_wr_en = 1
: imm값을  Register_File에 저장해야 하므로 활성화.

RegWdataSel = 2
: imm 값 선택.

# 5. U-type (AUIPC)




- **데이터 흐름**

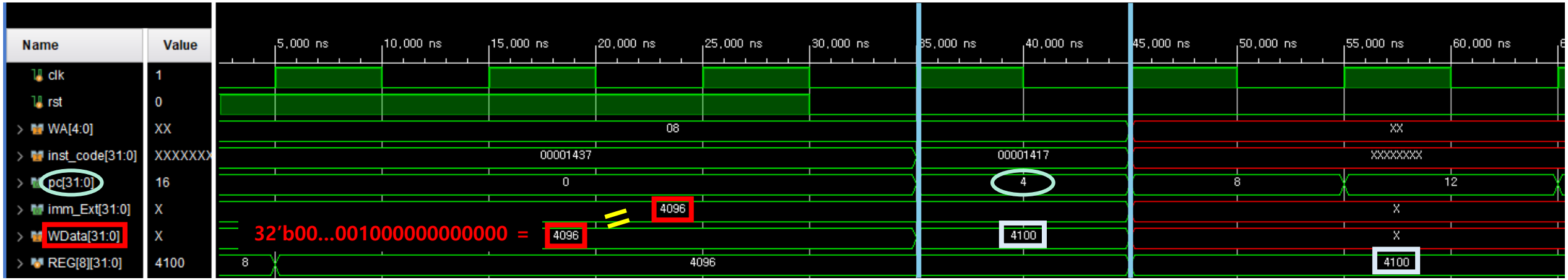
1. ROM에서 명령어 인출.
2. 20bit imm이 Extend 유닛을 거쳐 32bit로 확장.
   (하위 12bit는 0으로 채움)
3. 이 값은 mux 3번 입력으로 PC값과 더해져
   Register_file로 전달됨.

- **신호 제어**

reg_wr_en = 1
: imm값을  Register_File에 저장해야 하므로 활성화.

RegWdataSel = 3
: PC + imm 연산 결과 선택.

# 5. U-type Simulation




LUI x8, 1

AUIPC x8, 1

LUI : imm 12bit shift left하고 그 값을 reg_file(REG)에 할당.
AUIPC : imm 12bit shift left하고 PC와 더한 값을 reg_file(REG)에 할당.

| name | Descript |
|---|---|
| LUI | rd = imm |
| AUIPC | rd = PC + imm |

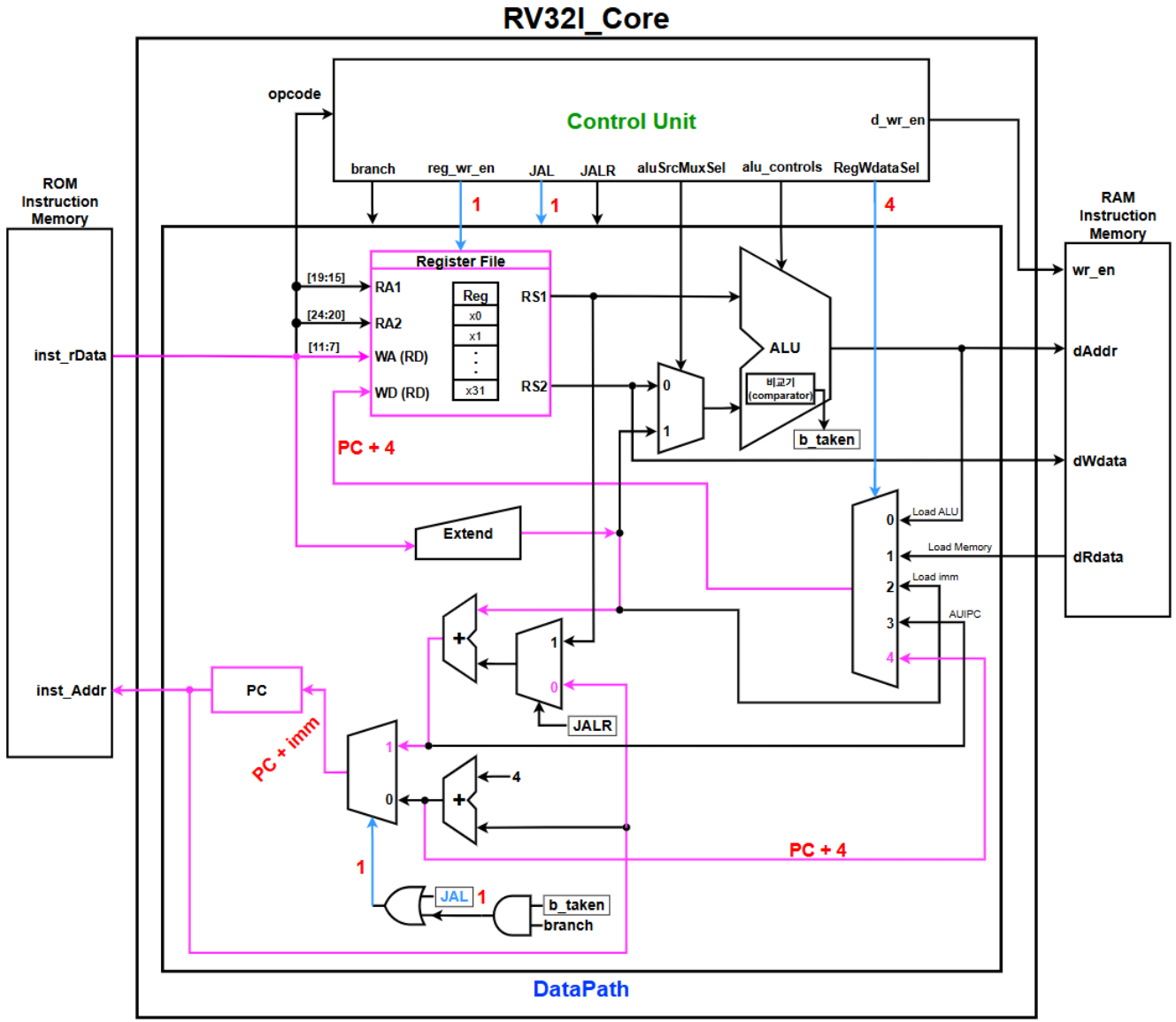
# 6. J-type

: 현재 위치에서 먼 거리의 주소로 점프(Jump)하기 위해 사용되는 instruction format

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| imm[20,10:1,11,19:12] | | | | | | | | | | | | | | | | | | | | rd | | | | | opcode | | | | | | | name | Descript |
| imm[20,10:1,11,19:12] | | | | | | | | | | | | | | | | | | | | rd | | | | | 1 | 1 | 0 | 1 | 1 | 1 | 1 | JAL | rd = PC+4; PC += imm |

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | funct3 | | | rd | | | | | opcode | | | | | | | name | Descript |
| imm[11:0] | | | | | | | | | | | | rs1 | | | | | 0 | 0 | 0 | rd | | | | | 1 | 1 | 0 | 0 | 1 | 1 | 1 | JALR | rd = PC+4; PC = rs1 + imm |

# 6. J-type (JAL)




- **데이터 흐름**

1. ROM에서 명령어 인출.
2. PC와 imm을 더해 점프할 목표 주소로 사용.
3. 복귀 주소인 PC+4가 계산되어 mux 4번 신호로 Register_File에 전달

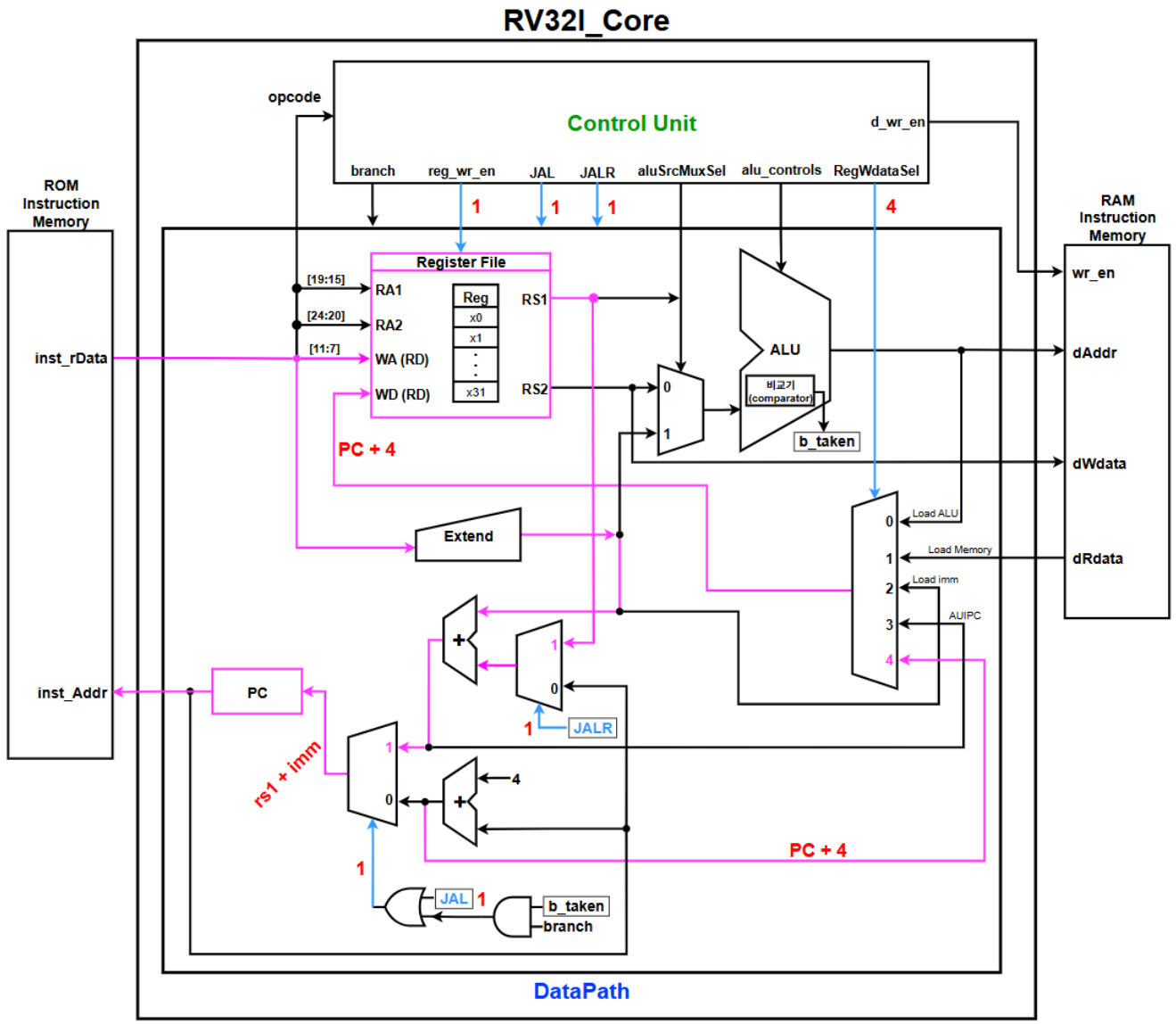
- **신호 제어**

jal = 1
: PC가 점프 주소로 업데이트 되도록 함.

reg_wr_en = 1
: 돌아올 주소(PC+4)를 Register_File에 저장해야 하므로 활성화.

RegWdataSel = 4
: Register_File에 쓸 데이터로 PC+4 선택.

# 6. J-type (JALR)




- **데이터 흐름**

1. ROM에서 명령어 인출.
2. rs1과 imm을 더해 점프할 목표 주소로 사용.
3. 복귀 주소인 PC+4가 계산되어 mux 4번 신호로 Register_File에 전달

- **신호 제어**

jal = 1
: PC가 점프 주소로 업데이트 되도록 함.

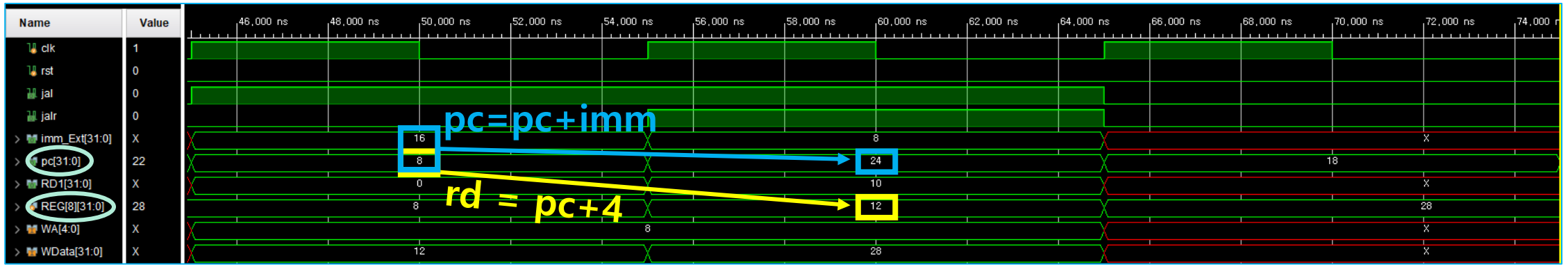
jalr = 1
: PC가 rs1+imm 값으로 업데이트되도록 mux 제어.

reg_wr_en = 1
: 돌아올 주소(PC+4)를 Register_File에 저장해야 하므로 활성화.

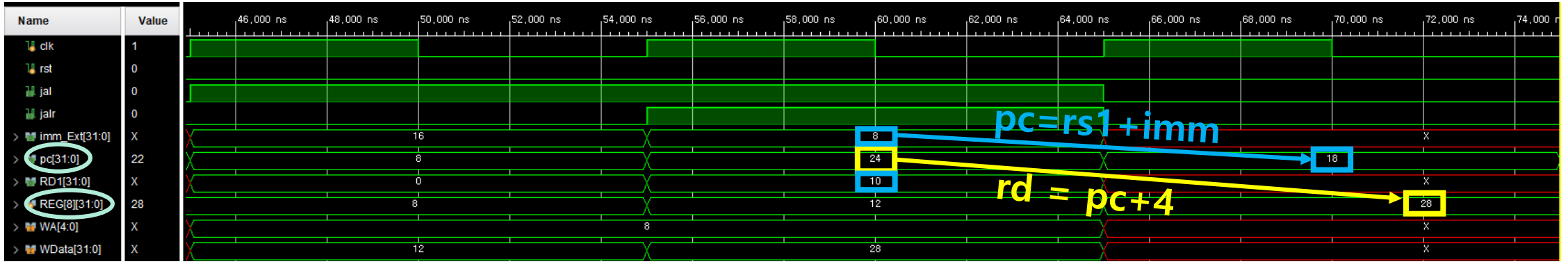
RegWdataSel = 4
: Register_File에 쓸 데이터로 PC+4 선택.

# 6. J-type Simulation

JAL x8, 16




JALR x8, 8(x10)




JAL : PC와 imm을 더하여 PC값으로 할당. rd는 PC + 4.
JALR : rs1값과 imm을 더하여 PC값으로 할당. rd는 PC + 4.

| name | Descript |
|---|---|
| JAL | rd = PC+4; PC += imm |

| name | Descript |
|---|---|
| JALR | rd = PC+4; PC = rs1 + imm |

감사합니다.