# RV32I Multi-Cycle CPU

## (32bit RISC-V processor)

System Verilog

대한상공회의소

AI 시스템 반도체 설계 (2기)

엄찬하

# CONTENTS

About Us

# Chapter 01

## RISC-V란?

About Us

## 01
## 모듈화



기본명령어 세트
+ 추가 명령어 세트(M,F 등)

## 02
## 개방성

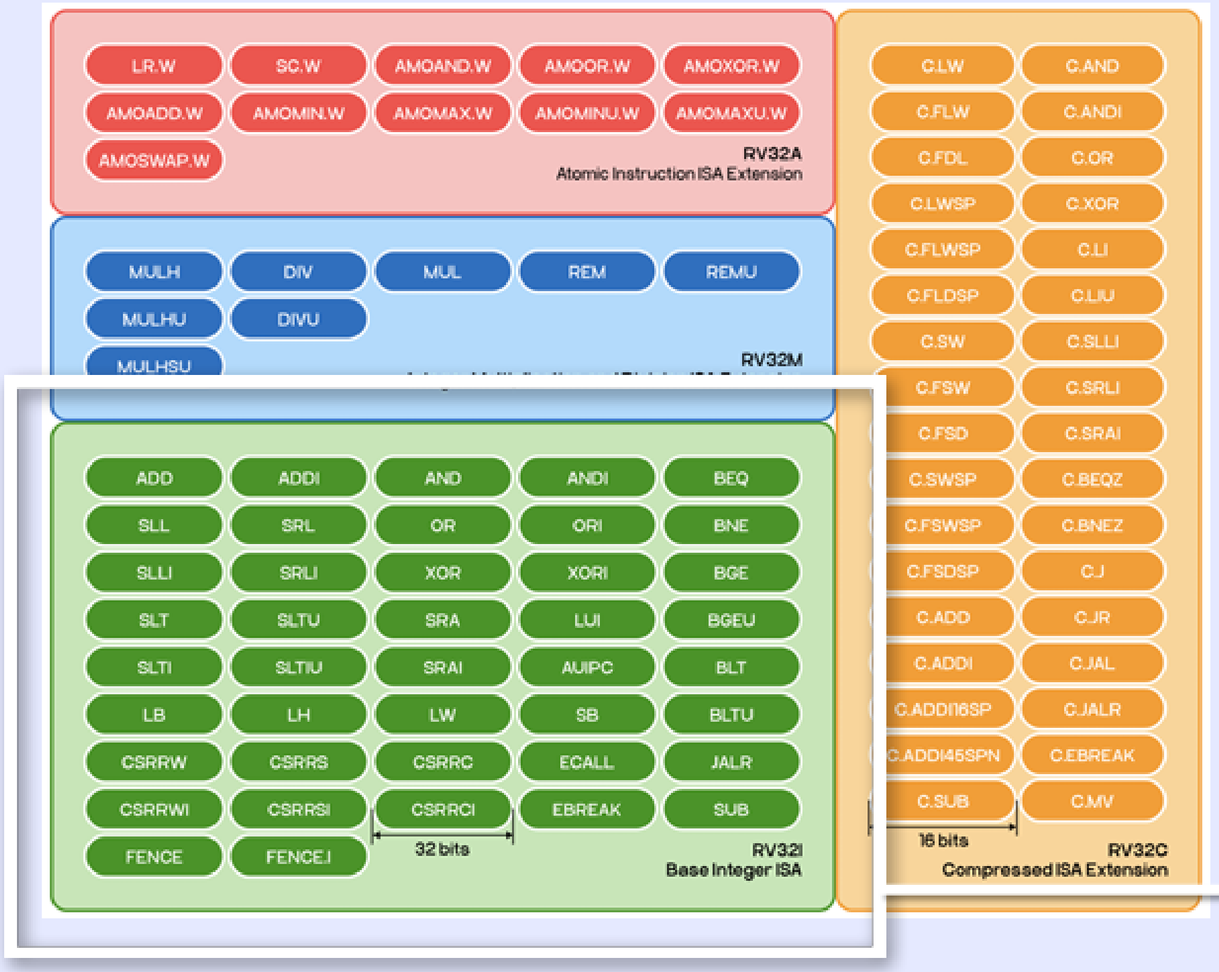
오픈소스 ISA
상업적 제약 無

## 03
## RV32I



RISC-V의 32비트 기본 명령어 세트
Single CYCLE

# RV32I Instruction Set




01

## Instruction(명령어)

컴퓨터에 명령을 내리기 위한 언어
여러 명령어들의 집합을 Instruction Set이라 함

02

## Instruction Set Architecture(ISA)

Instruction Set이 하드웨어에 적용될 수 있도록
구조와 인터페이스를 정의
하드웨어의 레지스터 구조, 명령어 실행 방식 등 포함

32bit 정수 연산을 지원하는 RISC-V 기본 ISA

| 확장 이름 | 버전 | 상태 | 설명 |
|---|---|---|---|
| "M" | 2.0 | **Ratified** | 정수 곱셈 및 나눗셈(integer multiplication and division) |
| "A" | 2.1 | **Ratified** | 원자적 연산(atomic memory instructions) |
| "F" | 2.2 | **Ratified** | 단정밀도(32-bit) 부동소수점 |
| "D" | 2.2 | **Ratified** | 배정밀도(64-bit) 부동소수점 |
| "Q" | 2.2 | **Ratified** | 사배정도(128-bit) 부동소수점 |
| "C" | 2.0 | **Ratified** | 16-bit 압축된 명령어 포맷 |
| "Counters" | 2.0 | *Draft* | |
| "L" | 0.0 | *Draft* | 10진수 부동소수점 |
| "B" | 0.0 | *Draft* | 비트 조작 |
| "J" | 0.0 | *Draft* | 동적 변환 언어 |
| "T" | 0.0 | *Draft* | |
| "P" | 0.2 | *Draft* | Packed SIMD 연산 |
| "V" | 1.0 | *Frozen* | 벡터 연산 |
| "Zicsr" | 2.0 | **Ratified** | CSR(Control and Status Register) 명령어 |
| "Zifencei" | 2.0 | **Ratified** | 명령어 인출 fence |
| "Zihintpause" | 2.0 | **Ratified** | |
| "Zihintntl" | 0.3 | *Frozen* | |
| "Zam" | 0.1 | *Draft* | |
| "Zfh" | 1.0 | **Ratified** | 반정밀도(16-bit) 부동소수점 |
| "Zfhmin" | 1.0 | **Ratified** | |
| "Zfinx" | 1.0 | **Ratified** | |
| "Zdinx" | 1.0 | **Ratified** | |
| "Zhinx" | 1.0 | **Ratified** | |
| "Zhinxmin" | 1.0 | **Ratified** | |
| "Zmmul" | 1.0 | **Ratified** | |
| "Ztso" | 1.0 | **Ratified** | |
| "Zfa" | 0.1 | *Draft* | |

# [GAM] ①딥시크 열풍이 끌어올린 'RISC-V', A주 투자 테마로 급부상

기사입력 : 2025년03월07일 07:36 | 최종수정 : 2025년03월07일 07:37

**X86, ARM과 함께 3대 주류 ISA로 꼽힘**
**AI 시대에 가장 최적화된 아키텍처 평가**
**RISC-V가 주목 받는 배경과 강점 진단**

지디넷코리아

## 삼성전자, '칩렛' 최적화 RISC-V 칩 기술 공개

Arm 대항마로 떠오른 RISC-V는 2010년 UC버클리에서 개발한 오픈소스(개방형) 아키텍처다. Arm의 IP를 이용하는 기업은 허락받고 사용료를 내야 한다.

2024. 10. 18.

AI타임스

## 메타, 'AI 훈련용' 자체 RISC-V AI 칩 테스트 시작

이 칩은 메타의 자체 개발 칩 시리즈인 '훈련 및 추론 가속기(MTIA)'의 최신 모델이다. MTIA 칩은 과거에도 진행됐으나, 초기 테스트에서 만족스러운 결과...

1개월 전

Chapter

# 02

# 구현 환경

# 구현환경

| 설계 언어 | 개발 툴 | 합성 | 테스트 환경 | 코딩 툴 |
| --- | --- | --- | --- | --- |
| ▪ SystemVerilog | ▪ Vivado 2020.2 | ▪ Vivado Synthesis | ▪ ROM기반 테스트 벤치 | ▪ VS Code |

Chapter

# 03

# Instruction

type

instruction datapath

**R type** Register

**I type** Immediate

**B type** Branch

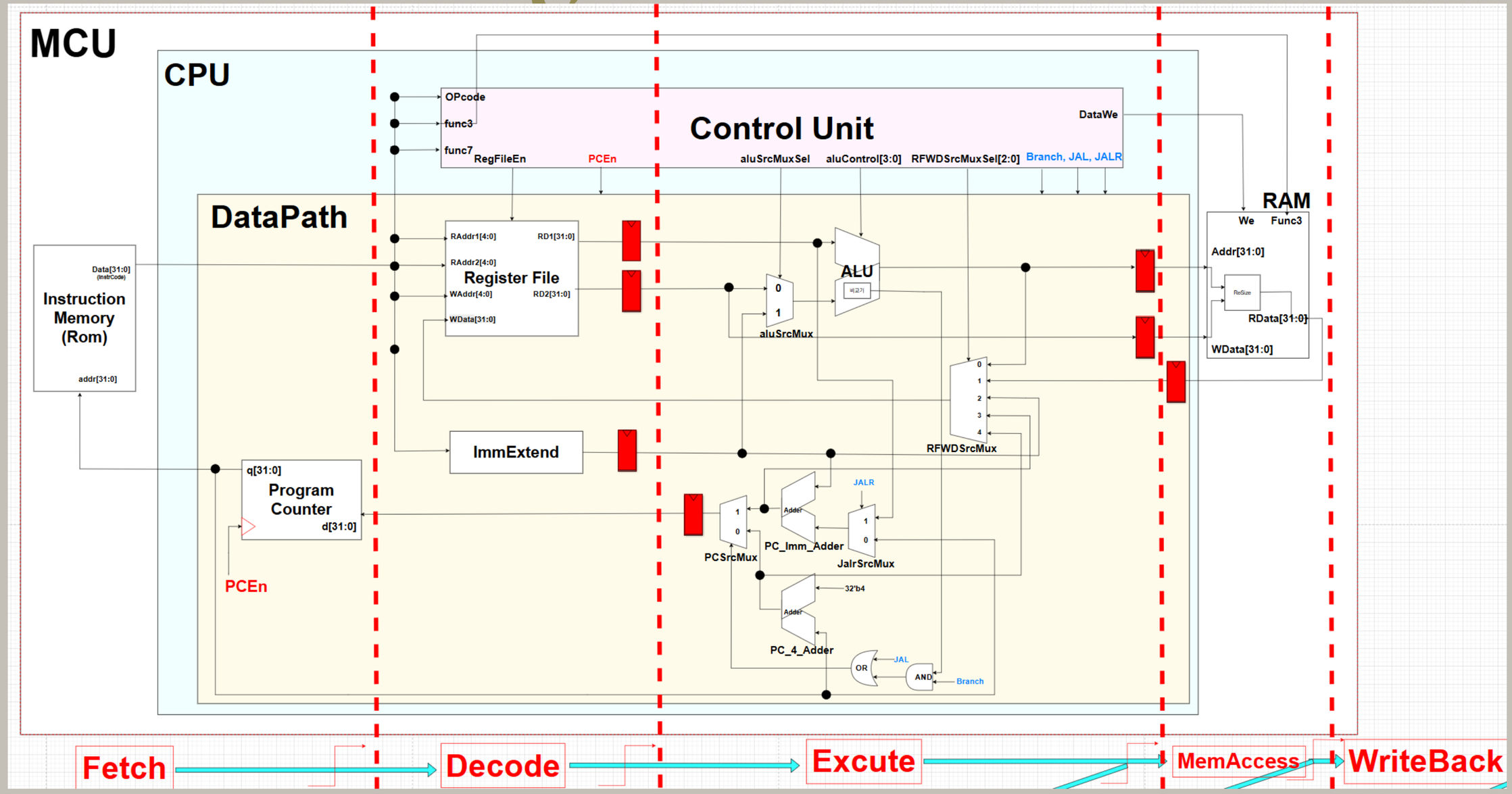
**L type** Load

**S type** Store

**The Otrher** J, AU, J, JL type

# Block Diagram





CHAPTER 3

# R-type

| Instruction | Funct7 | Register Source 2 | Register Source 1 | Funct3 | Register Destination | Opcode |
|---|---|---|---|---|---|---|
| **ADD** | 0000000 | rs2 | rs1 | 000 | rd | 0110011 |
| **SUB** | 0100000 | rs2 | rs1 | 000 | rd | 0110011 |
| **SLL** | 0000000 | rs2 | rs1 | 001 | rd | 0110011 |
| **SRL** | 0000000 | rs2 | rs1 | 101 | rd | 0110011 |
| **SRA** | 0100000 | rs2 | rs1 | 101 | rd | 0110011 |
| **SLT** | 0000000 | rs2 | rs1 | 010 | rd | 0110011 |
| **SLTU** | 0000000 | rs2 | rs1 | 011 | rd | 0110011 |
| **XOR** | 0000000 | rs2 | rs1 | 100 | rd | 0110011 |
| **OR** | 0000000 | rs2 | rs1 | 110 | rd | 0110011 |
| **AND** | 0000000 | rs2 | rs1 | 111 | rd | 0110011 |

# R-Type



## Role

- Register 간 연산 수행

## ROM

- Instruction Code Memory

## Register File

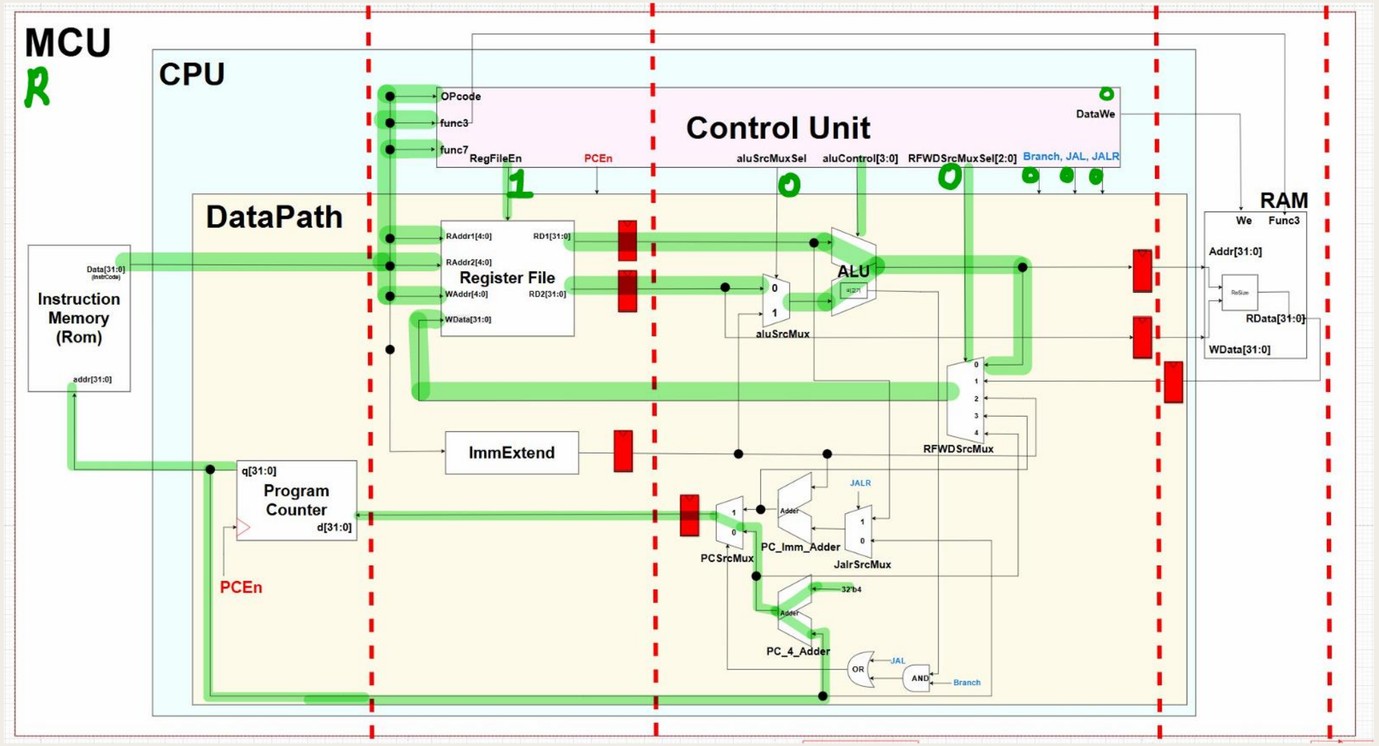
- Store & Load General Purpose Register
- R0 always out **Zero**

## ALU

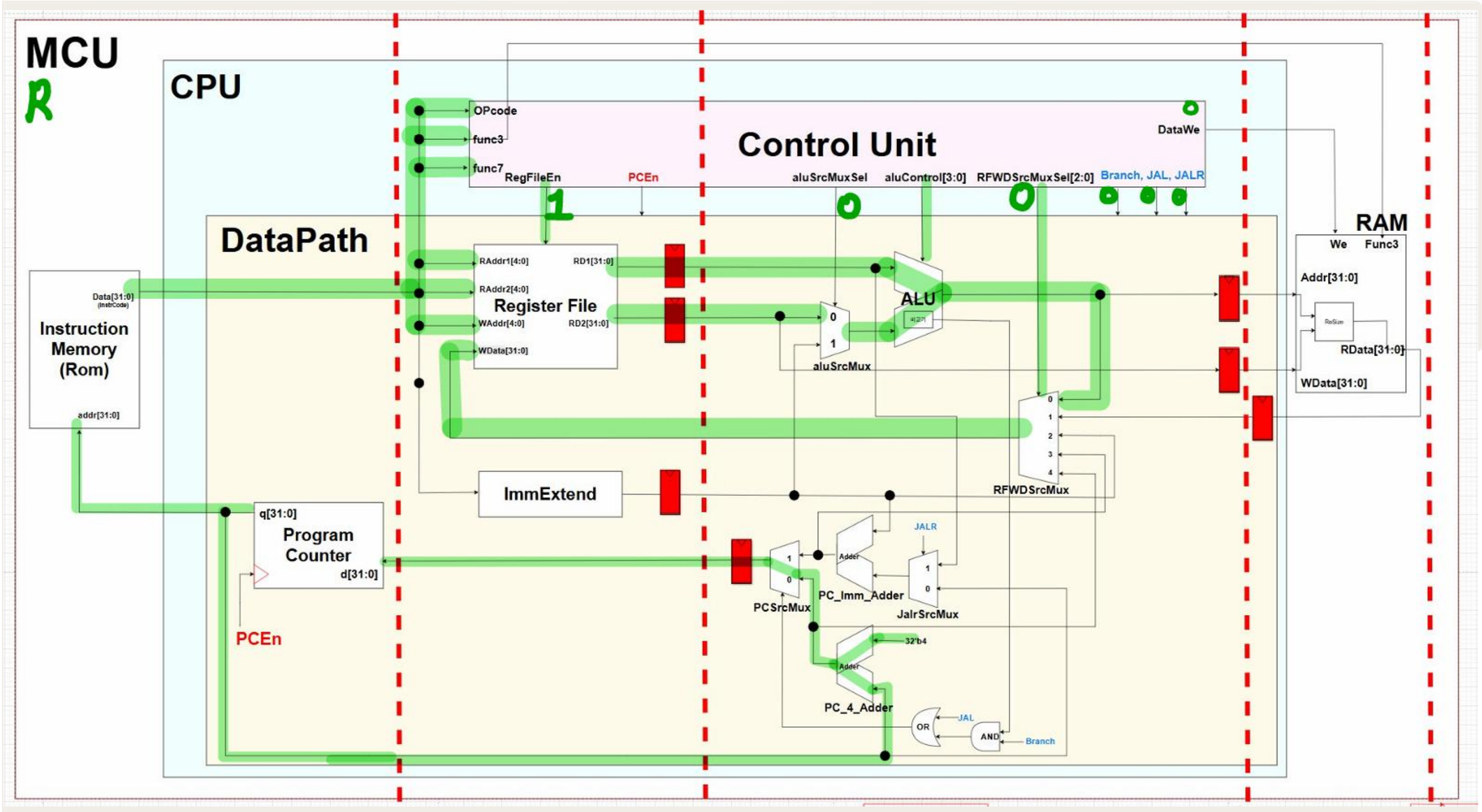
- **Arithmetic & Logical Operate**

## PC

- **Send next instruction address to ROM**



# R-Type


MCU
R
CPU
OPcode
func3
func7
Control Unit
DataWe
RegFileEn
PCEn
aluSrcMuxSel
aluControl[3:0]
RFWDSrcMuxSel[2:0]
Branch, JAL, JALR
1
0
0
0
0
0
DataPath
RAM
We
Func3
Addr[31:0]
RData[31:0]
WData[31:0]
RAddr1[4:0]
RD1[31:0]
RAddr2[4:0]
Register File
WAddr[4:0]
RD2[31:0]
WData[31:0]
ALU
aluSrcMux
Data[31:0]
(InstrCode)
Instruction Memory (Rom)
addr[31:0]
ImmExtend
RFWDSrcMux
q[31:0]
Program Counter
d[31:0]
PCEn
JALR
Adder
PC_Imm_Adder
PCSrcMux
JalrSrcMux
32'b4
Adder
PC_4_Adder
OR
JAL
AND
Branch

# L-type

| Instruction | immediate | Register Source 2 | Register Source 1 | Funct3 | Register Destination | Opcode |
|---|---|---|---|---|---|---|
| **LB** | imm[11:0] | | rs1 | 000 | rd | 0000011 |
| **LH** | imm[11:0] | | rs1 | 001 | rd | 0000011 |
| **LW** | imm[11:0] | | rs1 | 010 | rd | 0000011 |
| **LBU** | imm[11:0] | | rs1 | 100 | rd | 0000011 |
| **LHU** | imm[11:0] | | rs1 | 101 | rd | 0000011 |
| **SB** | imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 |
| **SH** | imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 |
| **SW** | imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 |

# L-Type



## Role

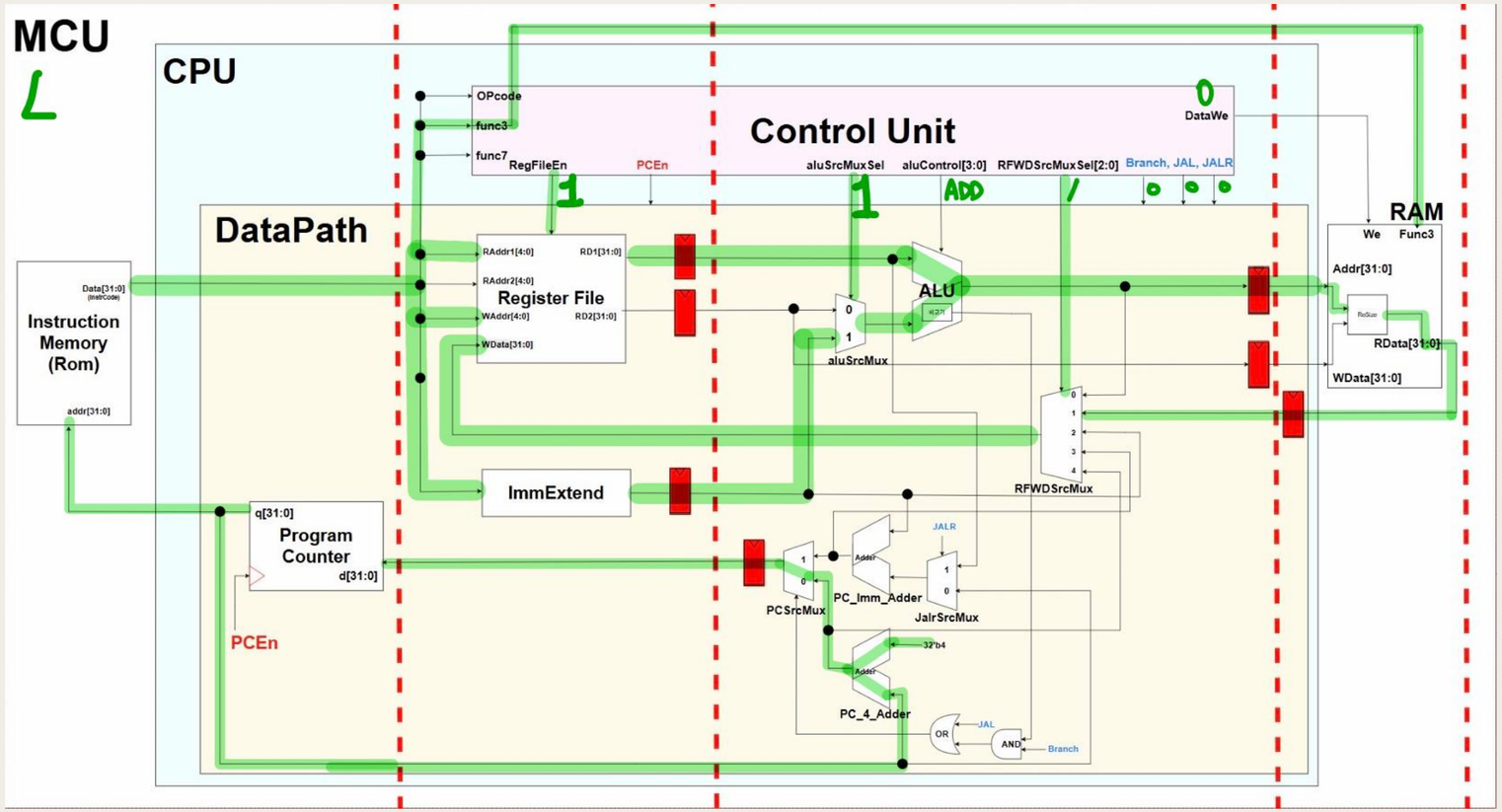
- Memory(RAM)의 Data를 Register로 가져옴

## RAM

- Data Memory

## Load_Val_Decision

- Load Data의 범위 지정
- Addr[1:0]으로 범위 선택

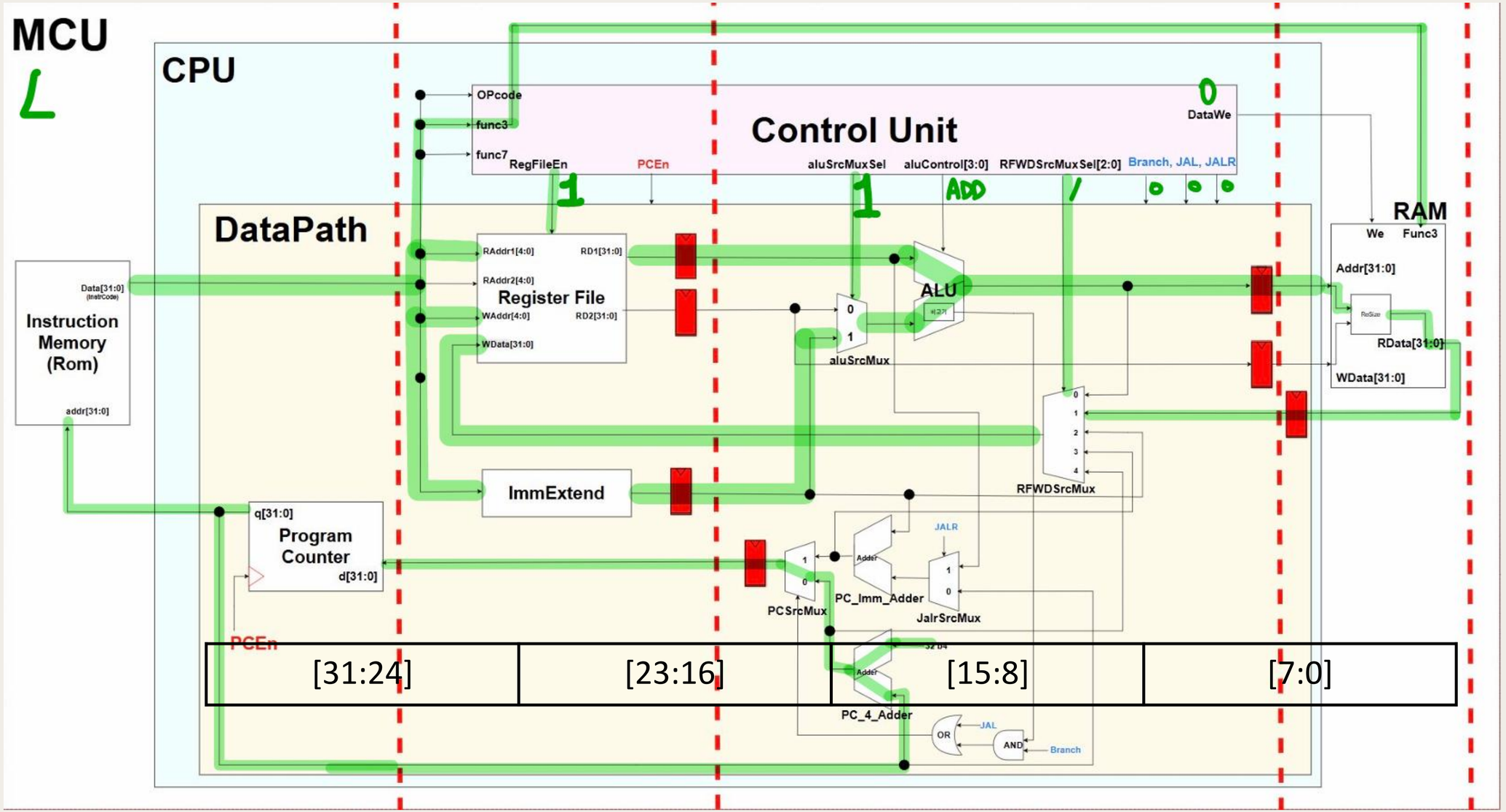



| [31:24] | [23:16] | [15:8] | [7:0] |
|---|---|---|---|

Addr[1:0] ⟶

# L-Type




MCU
L
CPU
OPcode
func3
func7
Control Unit
RegFileEn
PCEn
aluSrcMuxSel
aluControl[3:0]
RFWDSrcMuxSel[2:0]
Branch, JAL, JALR
DataWe
0
1
1
ADD
/
0 0 0
DataPath
RAM
We
Func3
Addr[31:0]
RData[31:0]
WData[31:0]
Data[31:0]
(InstrCode)
Instruction Memory (Rom)
addr[31:0]
RAddr1[4:0]
RD1[31:0]
RAddr2[4:0]
Register File
WAddr[4:0]
RD2[31:0]
WData[31:0]
ALU
aluSrcMux
ImmExtend
RFWDSrcMux
q[31:0]
Program Counter
d[31:0]
PCEn
JALR
Adder
PC_Imm_Adder
PCSrcMux
JalrSrcMux
[31:24]
[23:16]
[15:8]
[7:0]
PC_4_Adder
OR
AND
JAL
Branch

# I-type

| Instruction | Funct7 | shift amount | Register Source 1 | Funct3 | Register Destination | Opcode |
|---|---|---|---|---|---|---|
| **ADDI** | imm[11:0] | | rs1 | 000 | rd | 0010011 |
| **SLTI** | imm[11:0] | | rs1 | 010 | rd | 0010011 |
| **SLTIU** | imm[11:0] | | rs1 | 011 | rd | 0010011 |
| **XORI** | imm[11:0] | | rs1 | 100 | rd | 0010011 |
| **ORI** | imm[11:0] | | rs1 | 110 | rd | 0010011 |
| **ANDI** | imm[11:0] | | rs1 | 111 | rd | 0010011 |
| **SLLI** | 0000000 | shamt | rs1 | 001 | rd | 0010011 |
| **SRLI** | 0000000 | shamt | rs1 | 101 | rd | 0010011 |
| **SRAI** | 0100000 | shamt | rs1 | 101 | rd | 0010011 |

# I-Type



## Role

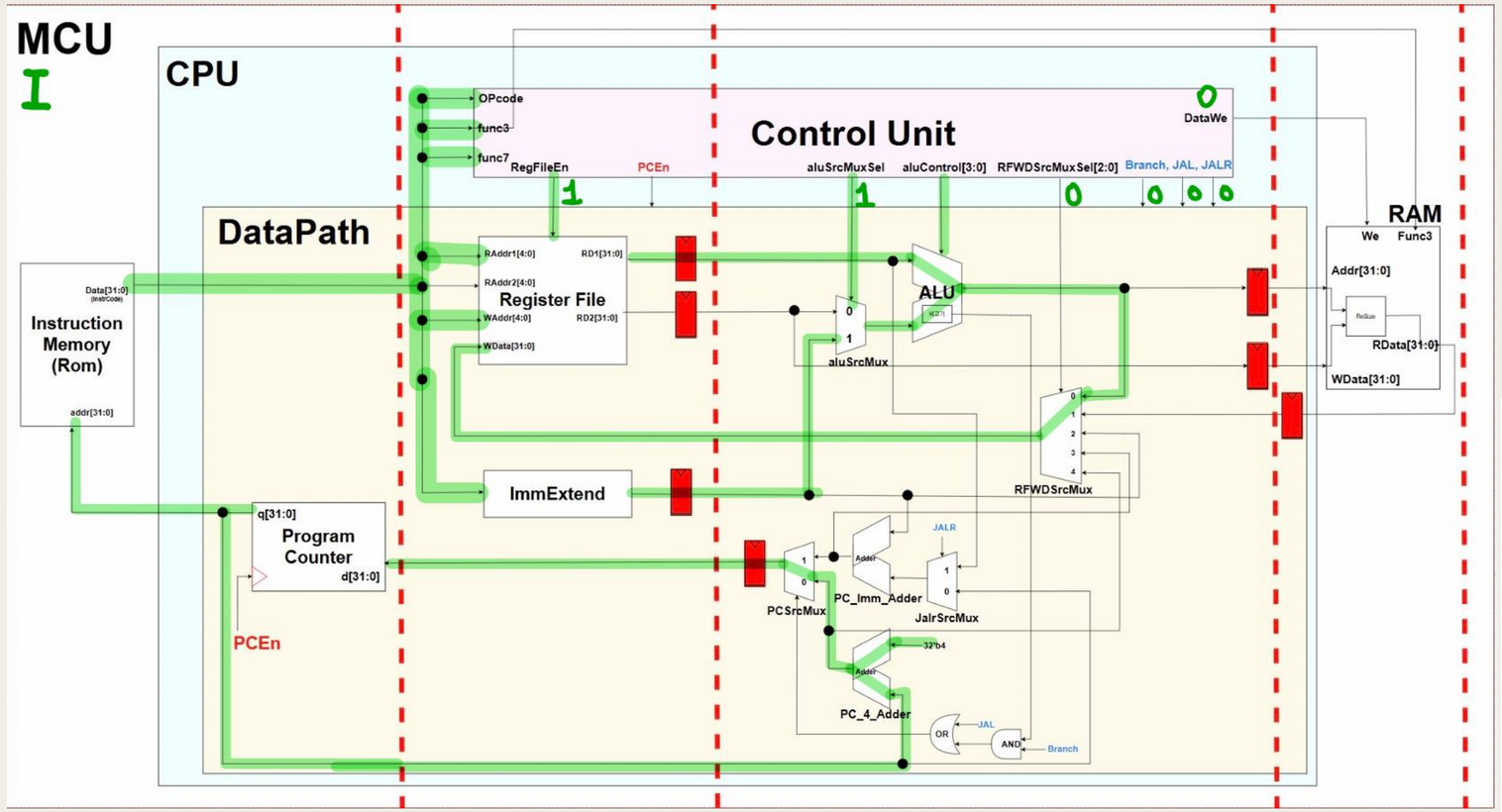
- Register(rs1)와 Immediate Value간 연산 수행

## IMM Extend

- 12bit Imm value extend to 32bit

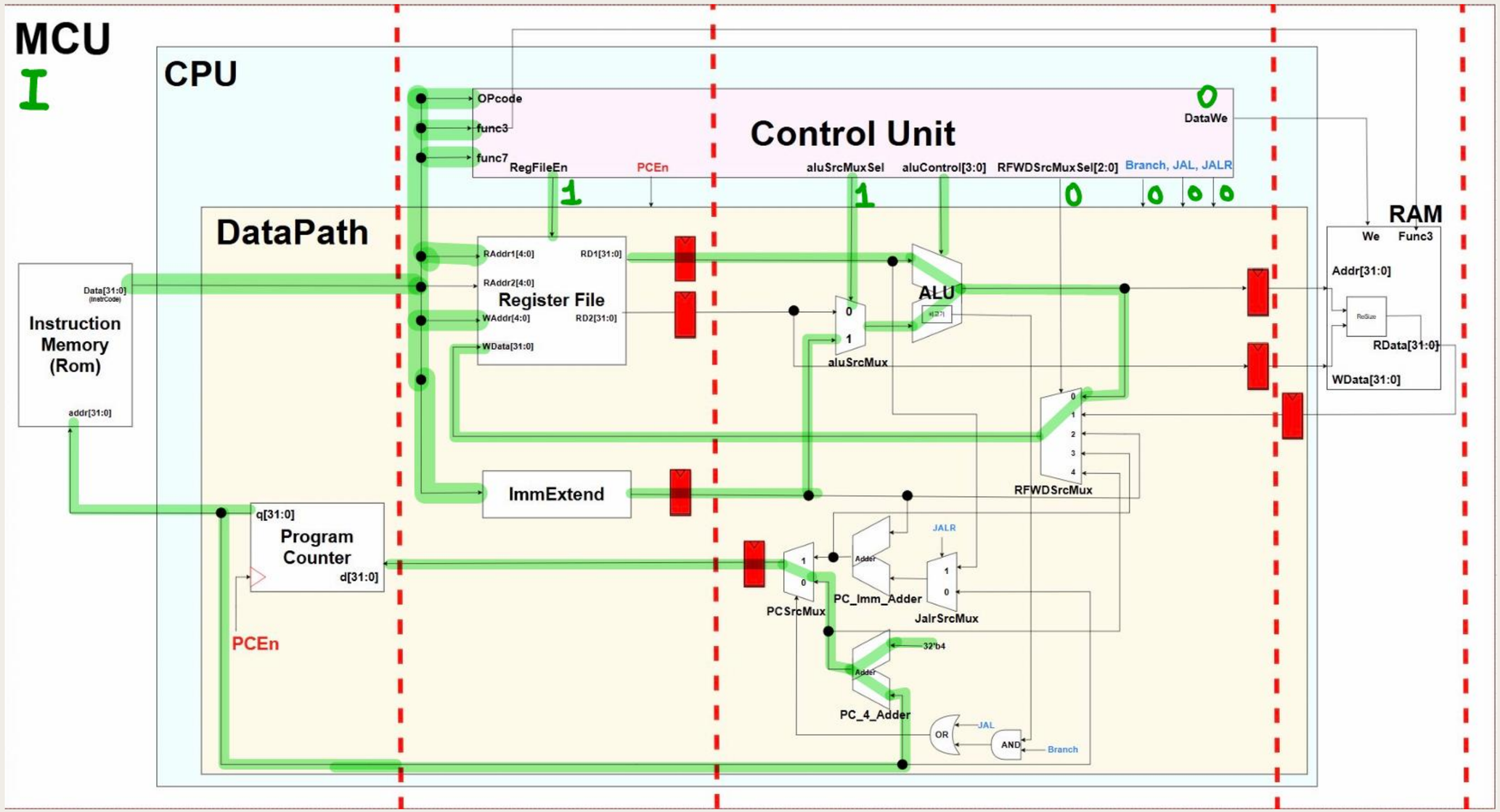
## ALUSrcMux

- Select ALU Input between RD2 & Immediate Value



# I-Type




MCU
I
CPU
DataPath
Control Unit
OPcode
func3
func7
RegFileEn
PCEn
aluSrcMuxSel
aluControl[3:0]
RFWDSrcMuxSel[2:0]
Branch, JAL, JALR
DataWe
1
1
0
0 0 0 0
RAM
We
Func3
Addr[31:0]
RData[31:0]
WData[31:0]
Instruction Memory (Rom)
Data[31:0]
(InstrCode)
addr[31:0]
RAddr1[4:0]
RAddr2[4:0]
WAddr[4:0]
WData[31:0]
RD1[31:0]
RD2[31:0]
Register File
ALU
aluSrcMux
RFWDSrcMux
ImmExtend
q[31:0]
Program Counter
d[31:0]
PCEn
JALR
Adder
PC_Imm_Adder
PCSrcMux
JalrSrcMux
32'b4
PC_4_Adder
OR
AND
JAL
Branch

# S-type

| Instruction | Immediate(7) | Register Source 2 | Register Source 1 | Funct3 | Immediate(5) | Opcode |
|---|---|---|---|---|---|---|
| SB | imm[12][10:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 |
| SH | imm[12][10:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 |
| SW | imm[12][10:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 |

# S-Type

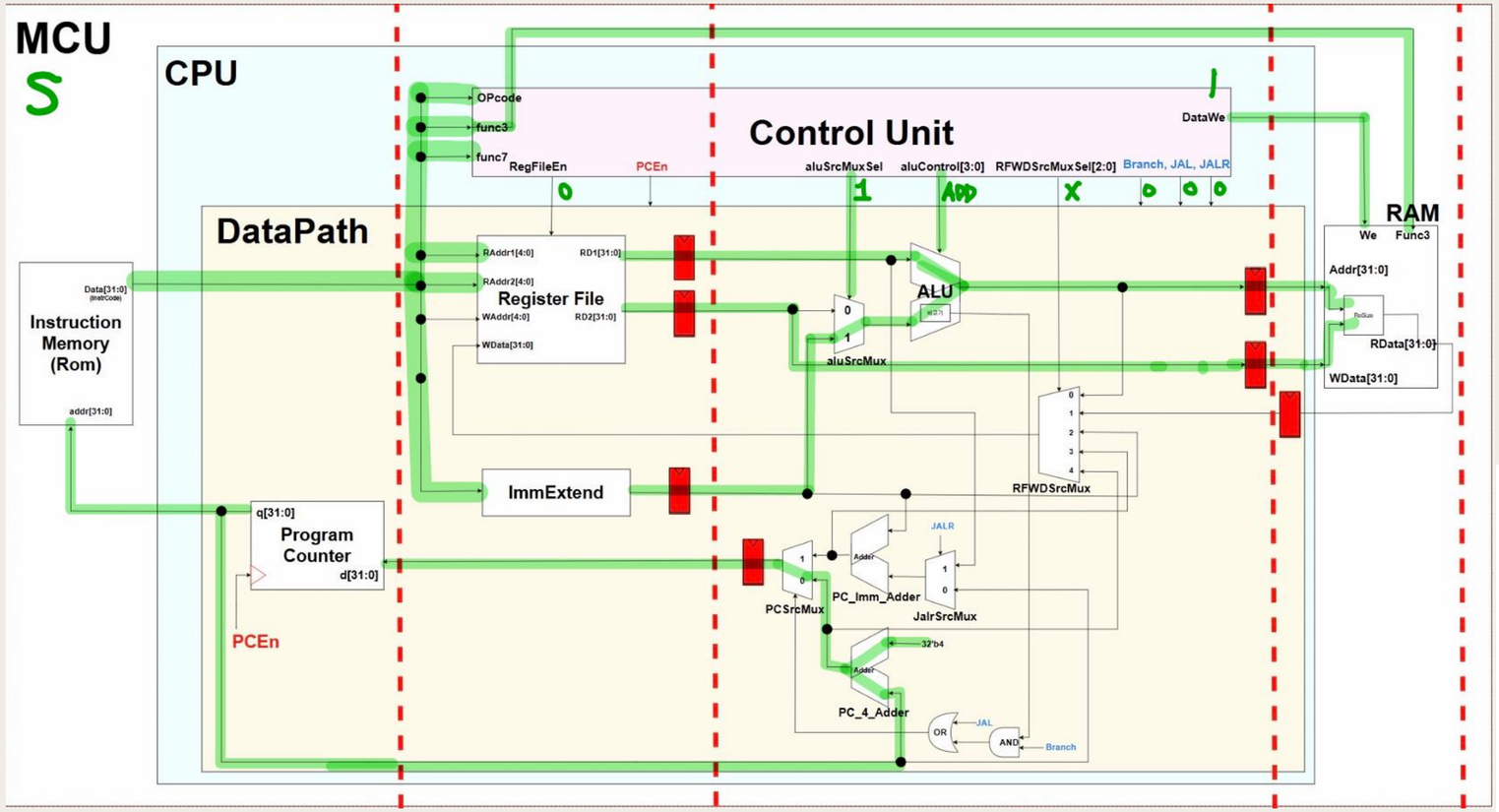


## Role

- Register의 Data를 RAM에 저장

## Store_Val_Decision

- 저장할 위치의 Data를 읽어옴
- S_mode에 따라 읽어온 Data의 특정 부분만 변경
- 변경된 Data를 RAM에 저장

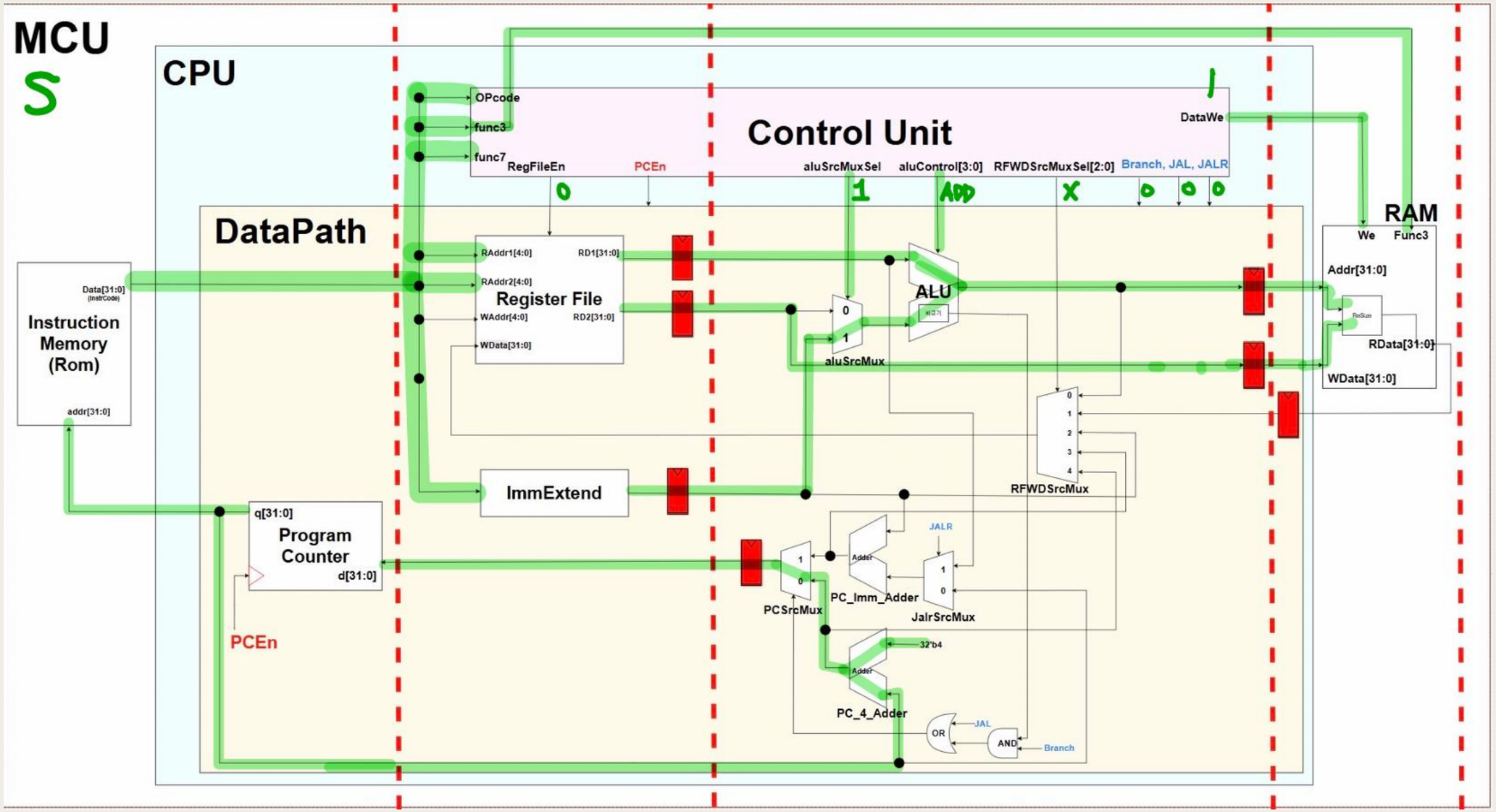



| [31:24] | [23:16] | [15:8] | [7:0] |
|---|---|---|---|

Addr[1:0] ⟶

# S-Type




MCU
S
CPU
Control Unit
OPcode
func3
func7
RegFileEn
PCEn
aluSrcMuxSel
aluControl[3:0]
RFWDSrcMuxSel[2:0]
Branch, JAL, JALR
DataWe
1
0
1
ADD
X
0
0
0
DataPath
RAM
We
Func3
Addr[31:0]
RData[31:0]
WData[31:0]
Instruction Memory (Rom)
Data[31:0]
addr[31:0]
RAddr1[4:0]
RD1[31:0]
RAddr2[4:0]
Register File
WAddr[4:0]
RD2[31:0]
WData[31:0]
ALU
aluSrcMux
ImmExtend
RFWDSrcMux
q[31:0]
Program Counter
d[31:0]
PCEn
PCSrcMux
PC_Imm_Adder
JALR
JalrSrcMux
32'b4
PC_4_Adder
OR
AND
JAL
Branch

CHAPTER 3

# B-type

| Instruction | immediate | Register Source 2 | Register Source 1 | Funct3 | immediate | Opcode |
|---|---|---|---|---|---|---|
| **BEQ** | imm[12][10:5] | rs2 | rs1 | 000 | imm[4:1][11] | 1100011 |
| **BNE** | imm[12][10:5] | rs2 | rs1 | 001 | imm[4:1][11] | 1100011 |
| **BLT** | imm[12][10:5] | rs2 | rs1 | 100 | imm[4:1][11] | 1100011 |
| **BGE** | imm[12][10:5] | rs2 | rs1 | 101 | imm[4:1][11] | 1100011 |
| **BLTU** | imm[12][10:5] | rs2 | rs1 | 110 | imm[4:1][11] | 1100011 |
| **BGEU** | imm[12][10:5] | rs2 | rs1 | 111 | imm[4:1][11] | 1100011 |

# B-Type



## Role

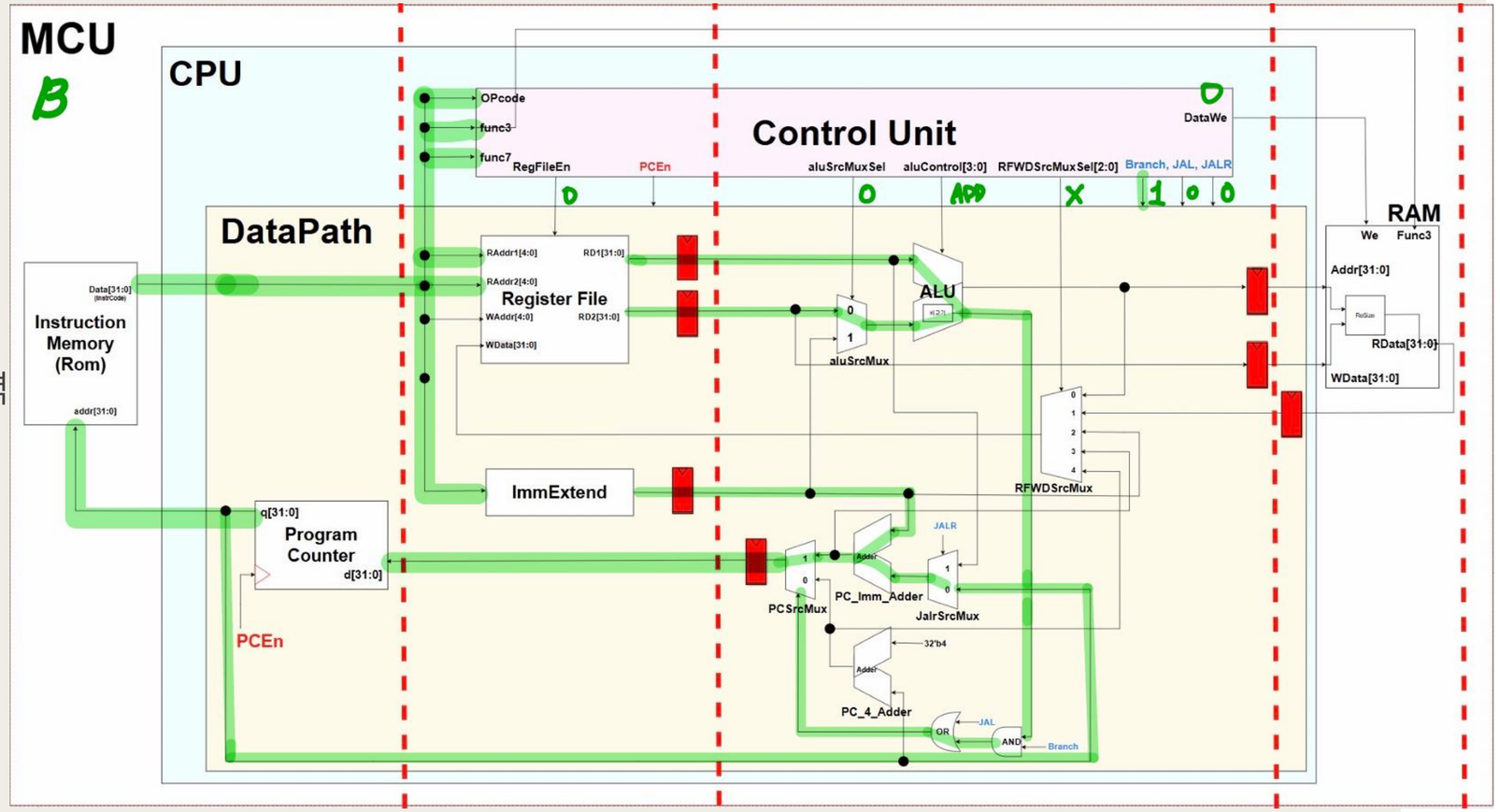
- 조건 비교 후 분기 결정

## Comparator

- 조건 비교를 위한 비교기
- ALU안에 내장 되어있음
- 조건 결과를 **btaken** 신호로 출력

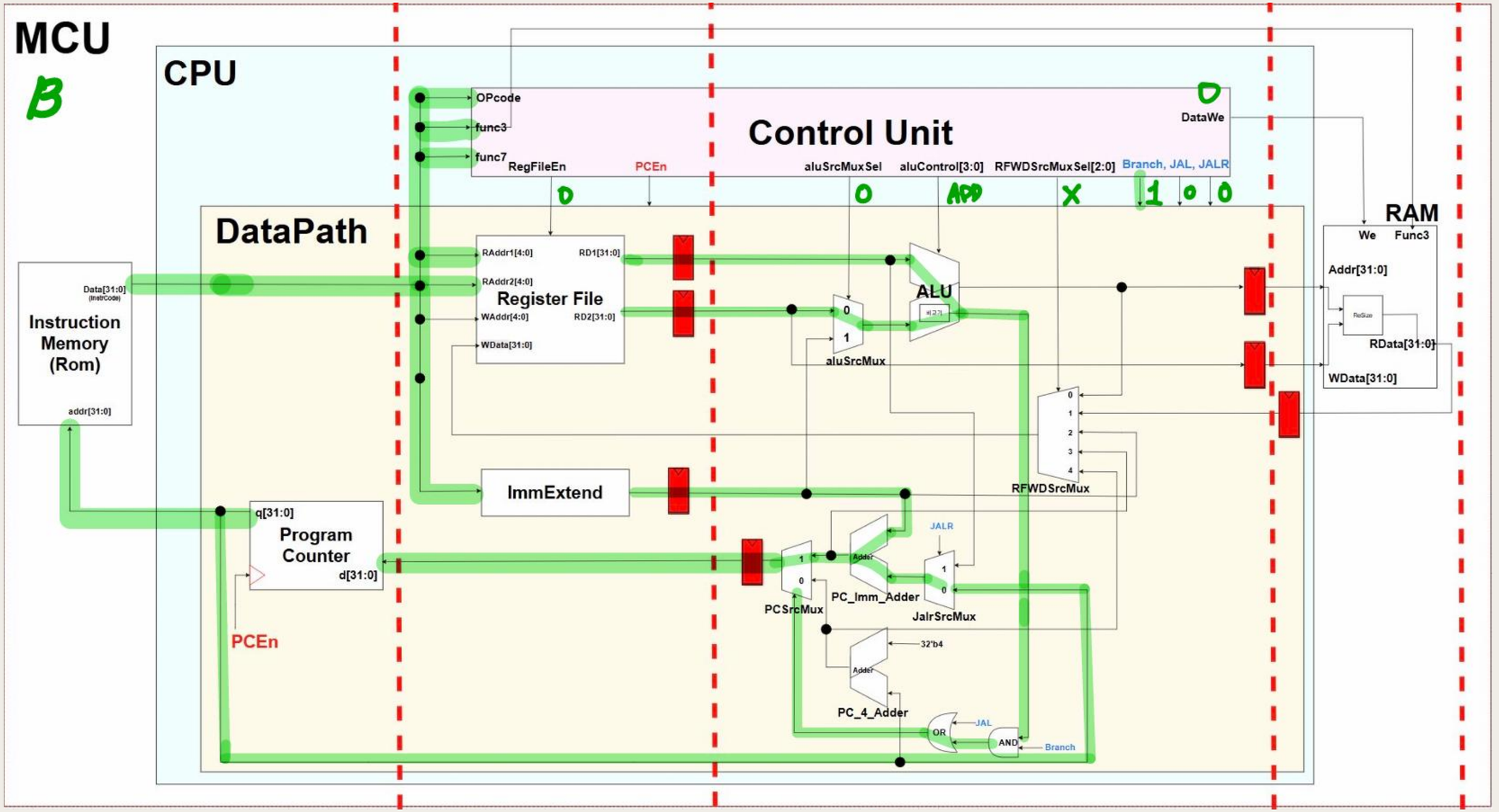
## PCSrcMux

- 조건 결과에 따라 PC와 더할 값 선택 필요
- If (btaken) PC + Imm
- If(! btaken) PC + 4



# B-Type




MCU
B
CPU
DataPath
Control Unit
OPcode
func3
func7
RegFileEn
PCEn
aluSrcMuxSel
aluControl[3:0]
RFWDSrcMuxSel[2:0]
Branch, JAL, JALR
DataWe
0
0
ADD
X
1
0
0
Instruction Memory (Rom)
Data[31:0]
(InstrCode)
addr[31:0]
RAddr1[4:0]
RAddr2[4:0]
WAddr[4:0]
WData[31:0]
RD1[31:0]
RD2[31:0]
Register File
ALU
aluSrcMux
ImmExtend
RFWDSrcMux
RAM
We
Func3
Addr[31:0]
RData[31:0]
WData[31:0]
q[31:0]
Program Counter
d[31:0]
PCEn
PCSrcMux
PC_Imm_Adder
JALR
JalrSrcMux
32'b4
PC_4_Adder
OR
AND
JAL
Branch

CHAPTER 3

# LU,AU,J,JL-type

| Instruction | immediate | Register Source 1 | | Register Destination | Opcode |
|---|---|---|---|---|---|
| **LUI** | imm[31:12] | | | rd | 0110111 |
| **AUIPC** | imm[31:12] | | | rd | 0010111 |
| **JAL** | imm[20][10:1][11][19:12] | | | rd | 1101111 |
| **JALR** | imm[11:0] | rs1 | 000 | rd | 1100111 |

| Instruction | Descript | Opcode |
|---|---|---|
| **AUIPC** | rd = Pc+(imm<<12) | 0010111 |
| **JAL** | rd = PC+4, PC += imm | 1101111 |
| **JALR** | rd= PC+4 ,PC = rs1 + imm | 1100111 |

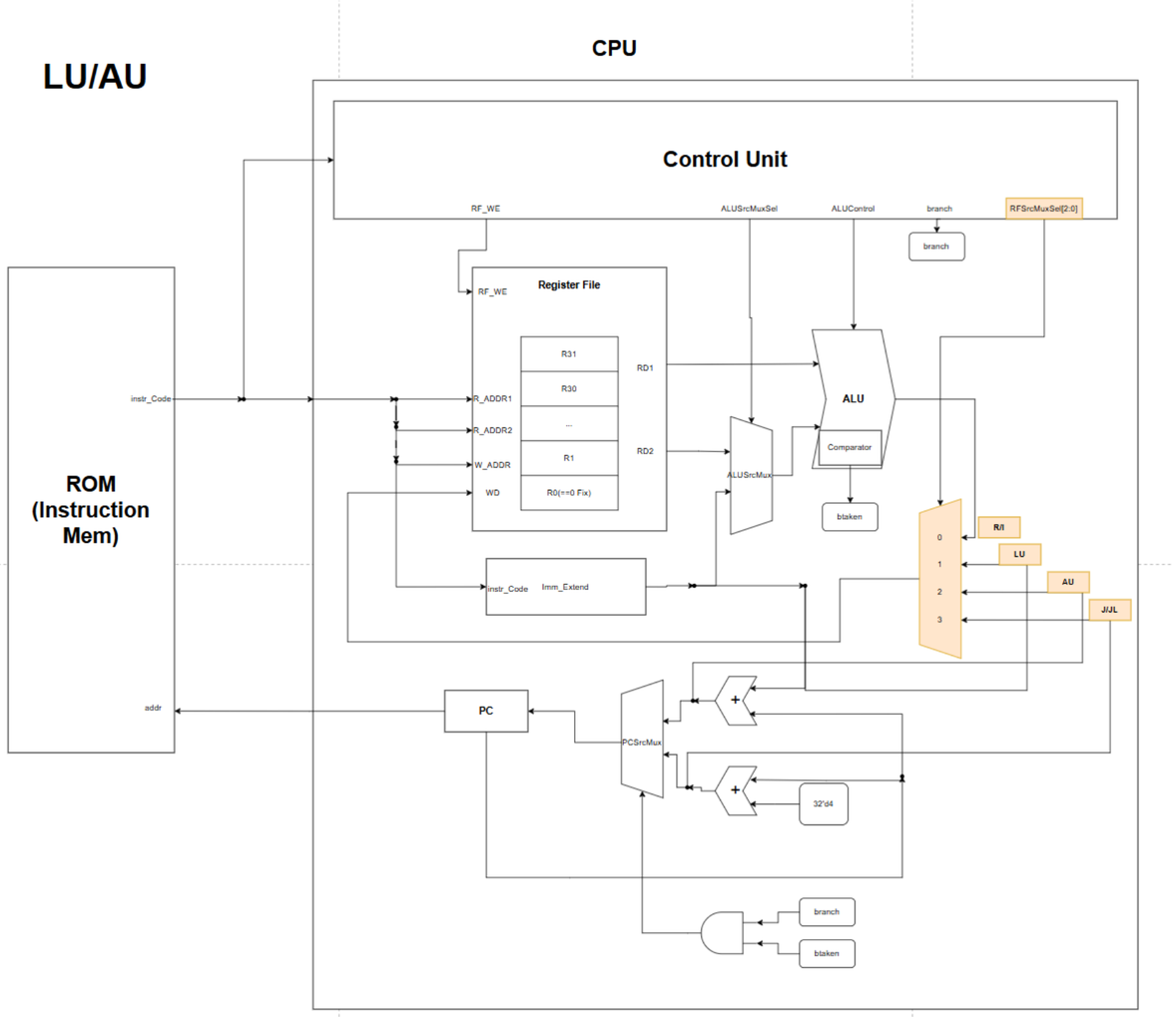
# LU/AU-Type: Description & HW Architecture

## Role

- **LU**: 매우 큰 수를 할당할 때 사용
  - ADDI의 경우, 12비트 데이터만 할당 가능
  - LU로 상위 20비트를 write하고 ADDI로 하위 12비트 write
- **AU**: PC 상대 주소 지정에 사용
  - 주로 LW나 JALR과 같이 사용

## RFSrcMux(5:1 Mux)

- Register File에 들어갈 Write Data 선택 필요
- 이후, Type들의 Data도 고려해 미리 5:1 Mux로 설계



# JAL/JALR-Type: Description & HW Architecture

## Role

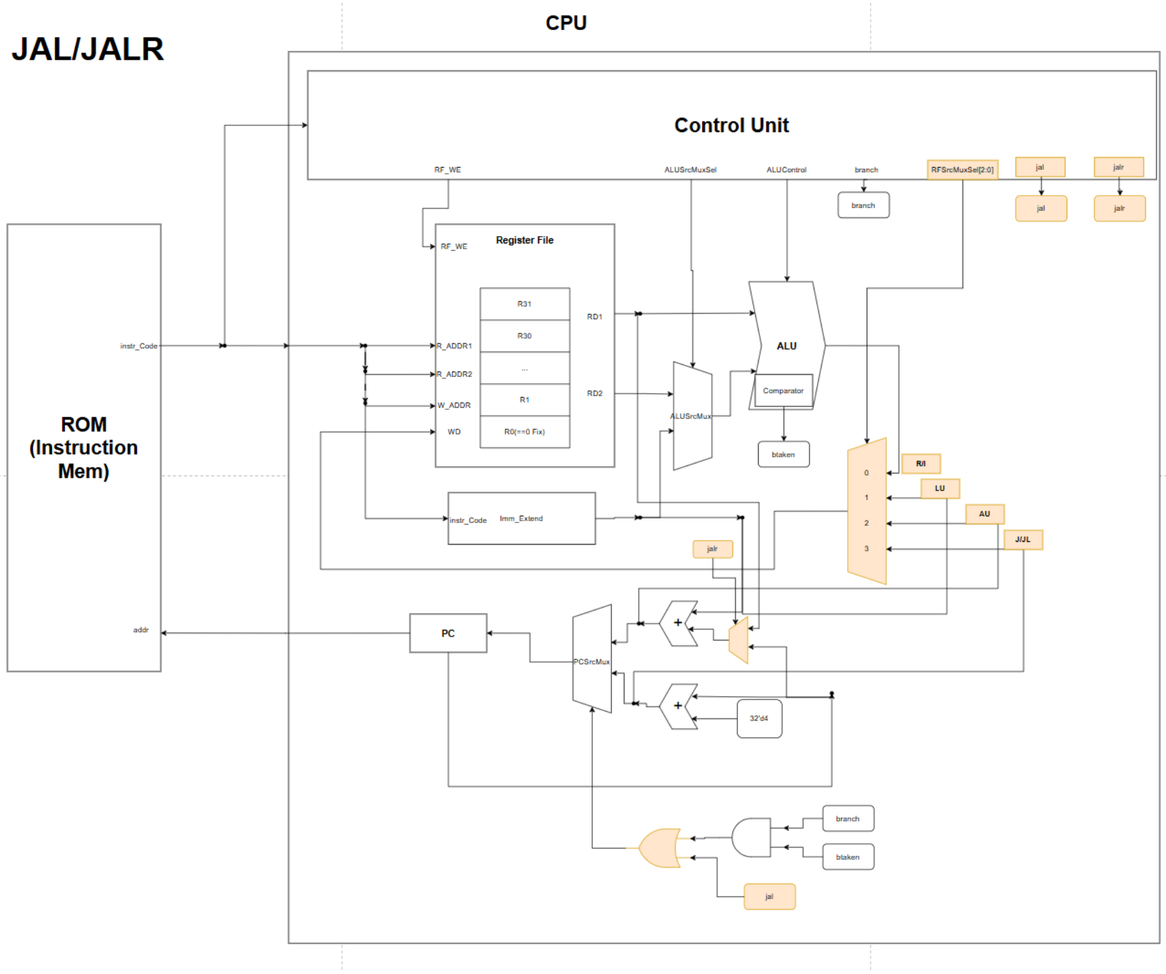
- **JAL**: PC 상대 주소 지정
  - 현재 PC기준 상대 위치 함수 이동에 사용
- **JALR**: 레지스터 주소 지정
  - 연산속도: Memory < Register
  - 포인터 혹은 배열 인덱싱

## PC_Imm_AdderSrcMux

- JALR-Type의 경우 Adder에 PC 대신 **rs1**값이 더해짐
- JALR Signal이 참일 때, 피연산자로 rs1 선택

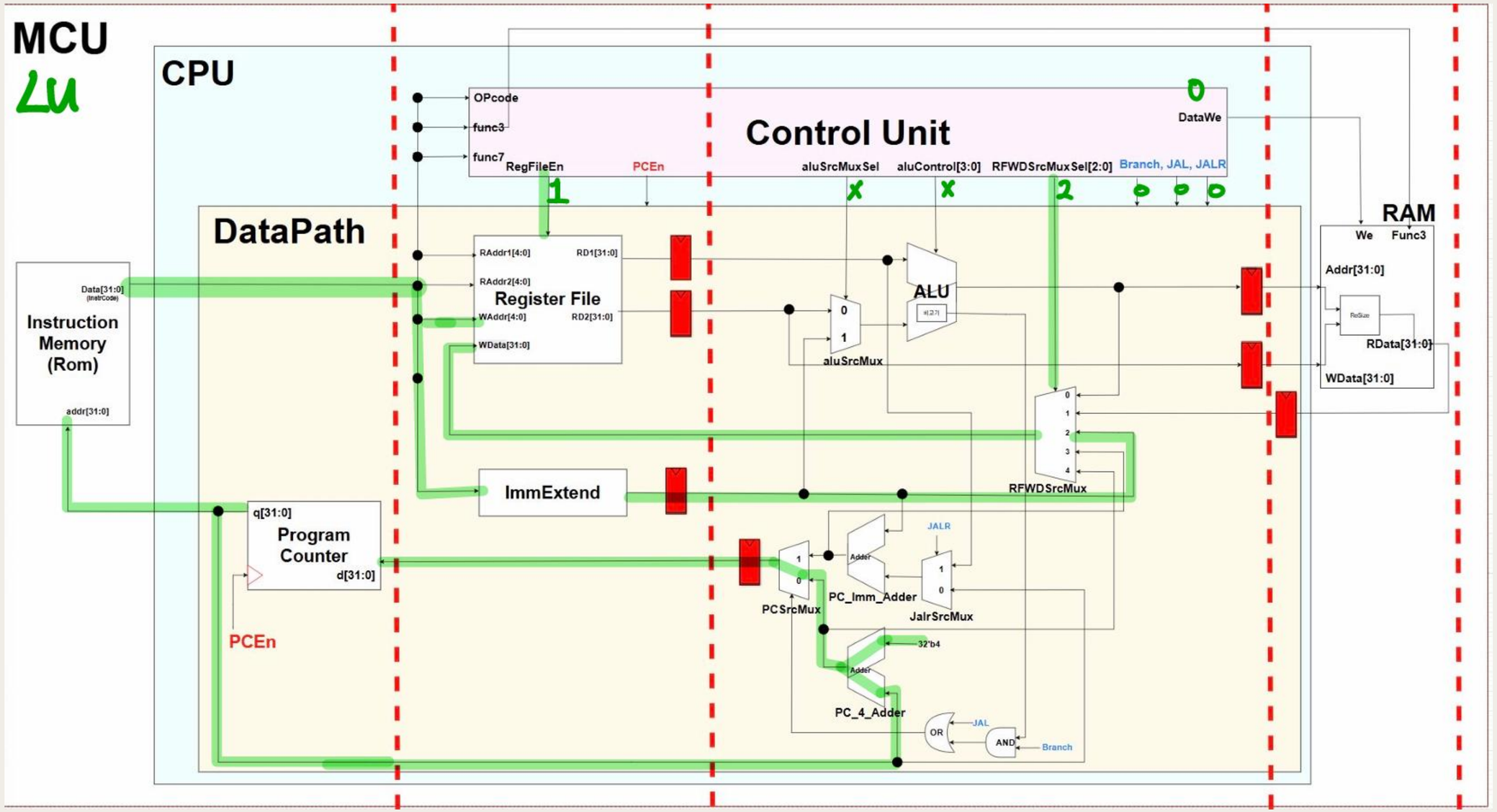
## PCSrcMux_Sel

- B-Type 뿐만 아니라 JAL-Type도 PC+Imm이 다음 PC값이 됨
- JAL Signal을 brach AND와 OR로 연결



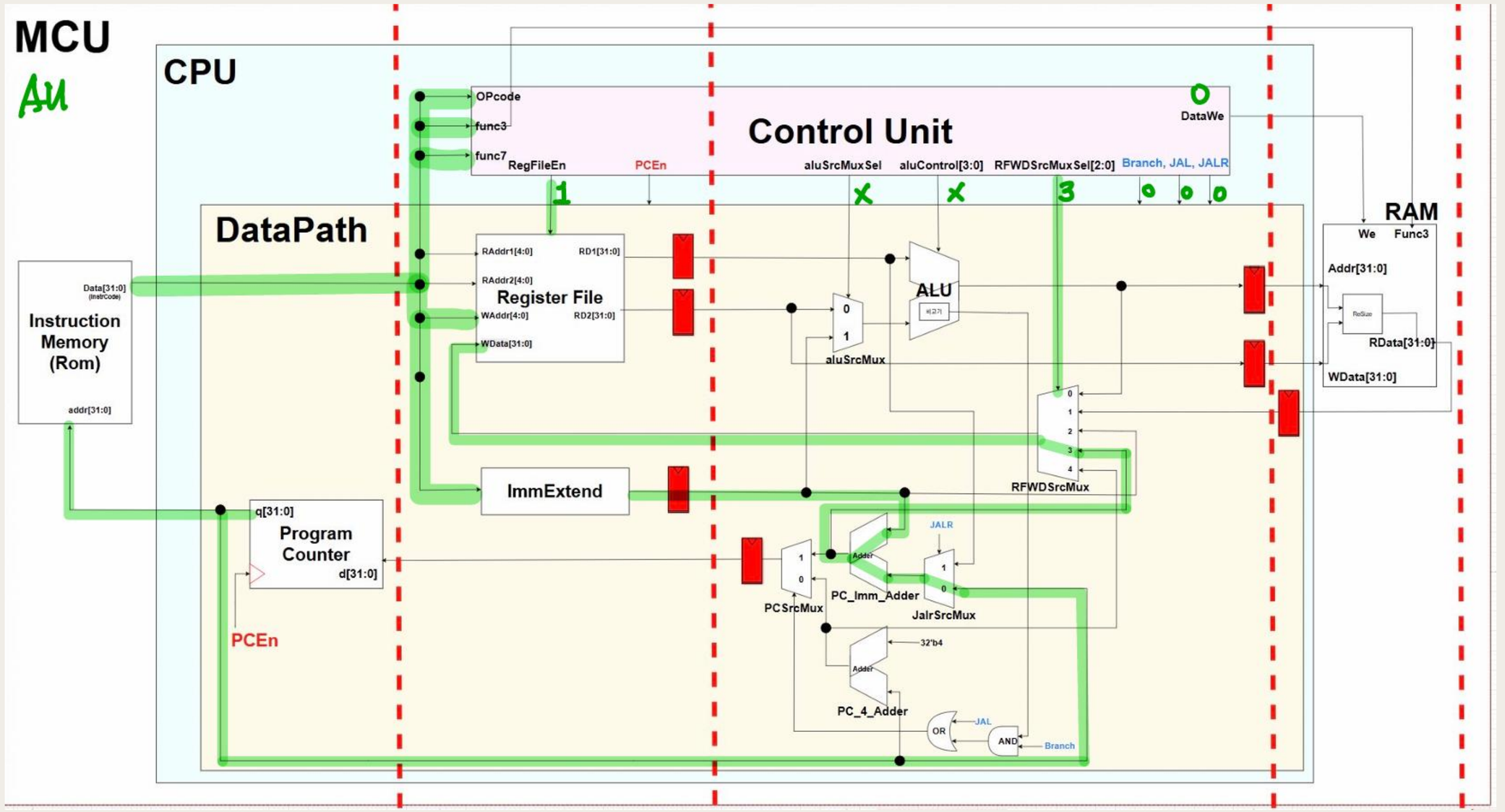
# LU-Type




MCU
Lu
CPU
Control Unit
OPcode
func3
func7
RegFileEn
PCEn
aluSrcMuxSel
aluControl[3:0]
RFWDSrcMuxSel[2:0]
Branch, JAL, JALR
DataWe
0
1
x
x
2
0 0 0
DataPath
RAM
We
Func3
Addr[31:0]
RData[31:0]
WData[31:0]
Data[31:0]
(InstrCode)
Instruction Memory (Rom)
addr[31:0]
RAddr1[4:0]
RD1[31:0]
RAddr2[4:0]
Register File
WAddr[4:0]
RD2[31:0]
WData[31:0]
ALU
aluSrcMux
ImmExtend
RFWDSrcMux
q[31:0]
Program Counter
d[31:0]
PCEn
JALR
Adder
PC_Imm_Adder
PCSrcMux
JalrSrcMux
32'b4
PC_4_Adder
OR
AND
JAL
Branch

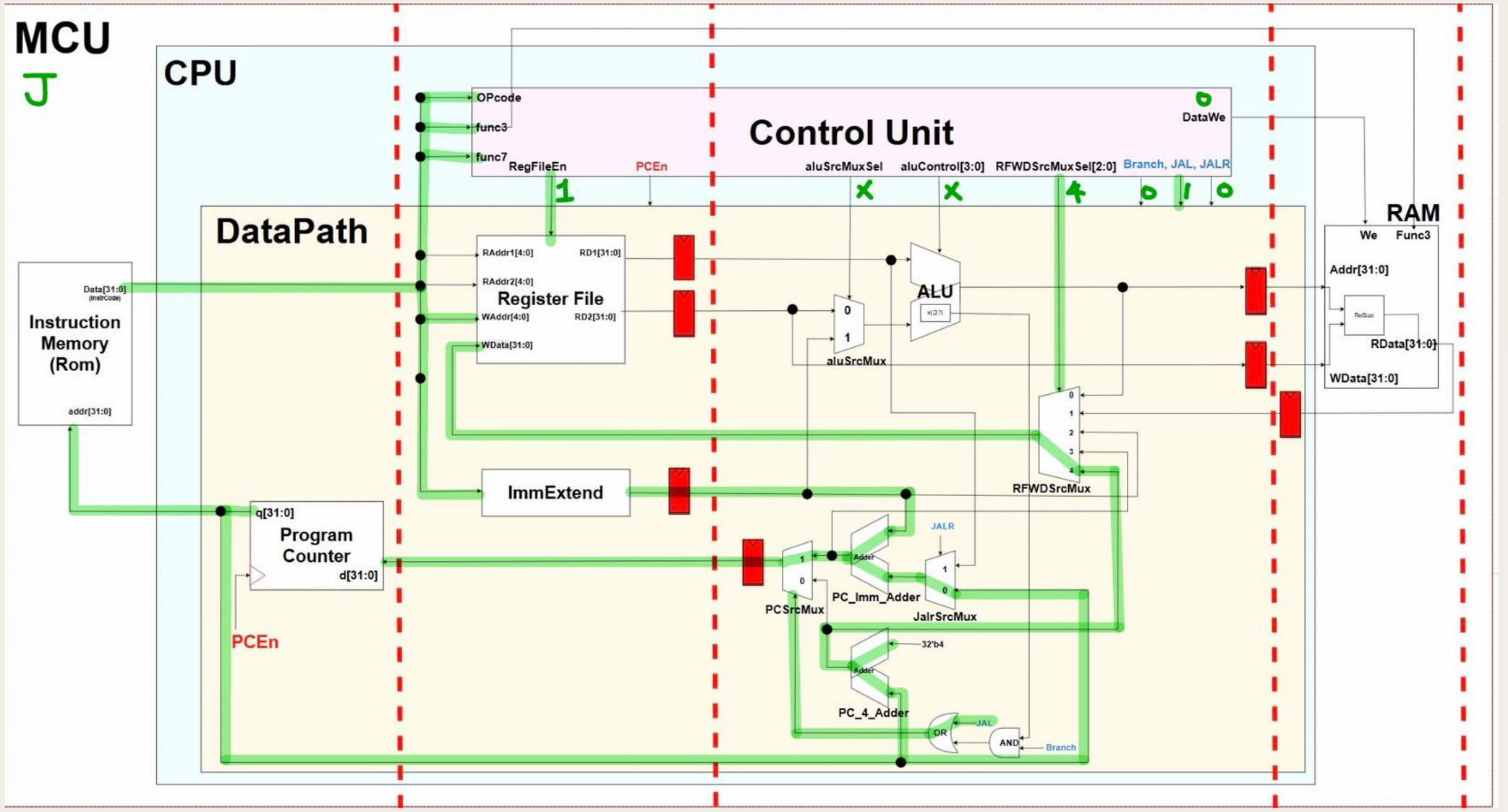
# AU-Type



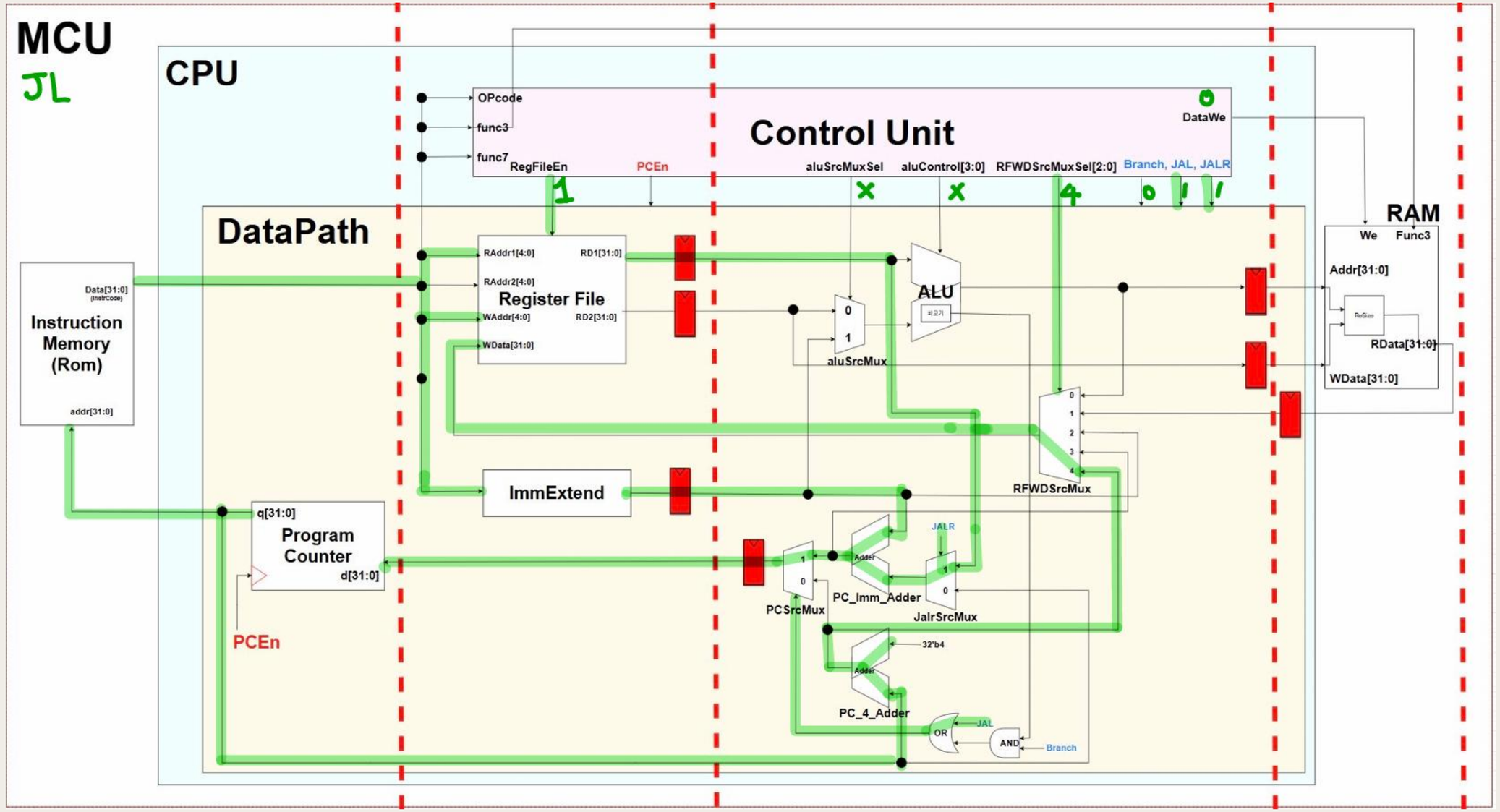

MCU
AU
CPU
Control Unit
OPcode
func3
func7
RegFileEn
PCEn
aluSrcMuxSel
aluControl[3:0]
RFWDSrcMuxSel[2:0]
Branch, JAL, JALR
DataWe
0
1
x
x
3
0
0
0
DataPath
RAM
We
Func3
Addr[31:0]
RData[31:0]
WData[31:0]
RAddr1[4:0]
RD1[31:0]
RAddr2[4:0]
Register File
WAddr[4:0]
RD2[31:0]
WData[31:0]
ALU
aluSrcMux
Data[31:0]
(InstrCode)
Instruction Memory (Rom)
addr[31:0]
ImmExtend
RFWDSrcMux
q[31:0]
Program Counter
d[31:0]
PCEn
JALR
Adder
PC_Imm_Adder
PCSrcMux
JalrSrcMux
32'b4
PC_4_Adder
OR
AND
JAL
Branch

# J-Type



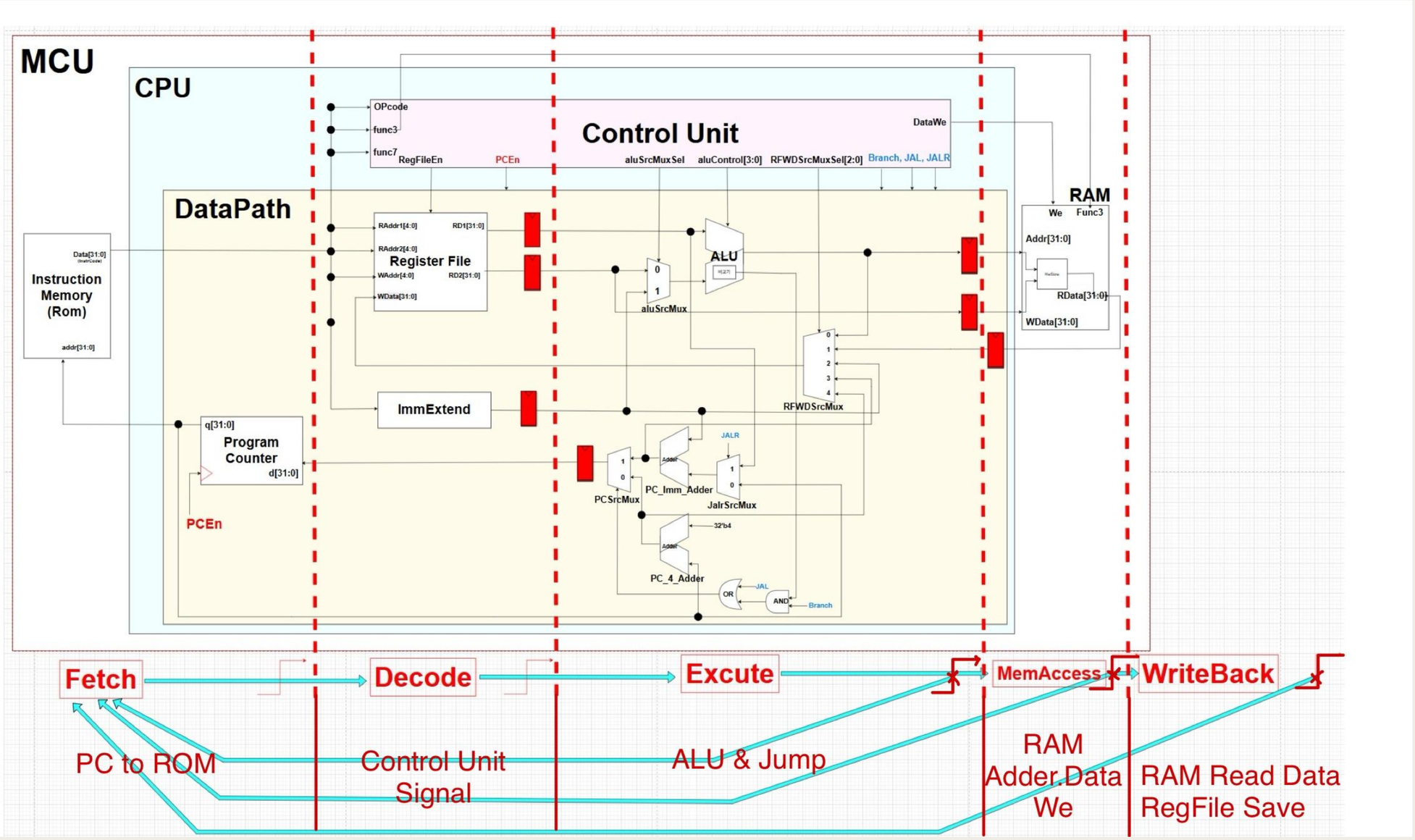

MCU
J
CPU
Control Unit
OPcode
func3
func7
RegFileEn
PCEn
aluSrcMuxSel
aluControl[3:0]
RFWDSrcMuxSel[2:0]
Branch, JAL, JALR
DataWe
1
X
X
4
0
1
0
DataPath
RAM
We
Func3
Addr[31:0]
RData[31:0]
WData[31:0]
Instruction Memory (Rom)
Data[31:0]
(InstrCode)
addr[31:0]
RAddr1[4:0]
RD1[31:0]
RAddr2[4:0]
Register File
WAddr[4:0]
RD2[31:0]
WData[31:0]
ALU
aluSrcMux
ImmExtend
RFWDSrcMux
q[31:0]
Program Counter
d[31:0]
PCEn
JALR
PC_Imm_Adder
PCSrcMux
JalrSrcMux
32'b4
PC_4_Adder
JAL
OR
AND
Branch

# JL-Type

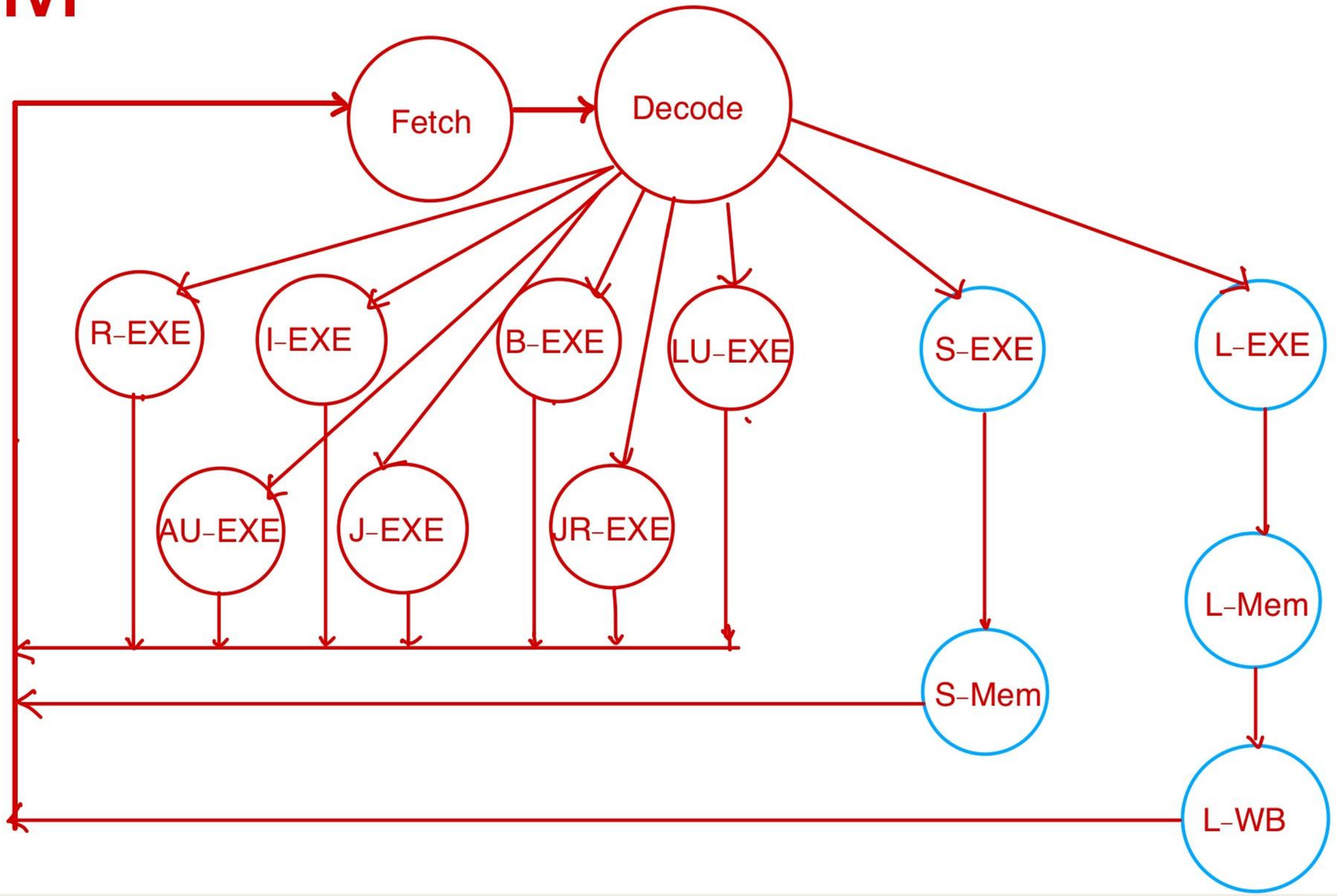

MCU
JL
CPU
Control Unit
OPcode
func3
func7
RegFileEn
PCEn
aluSrcMuxSel
aluControl[3:0]
RFWDSrcMuxSel[2:0]
Branch, JAL, JALR
DataWe
0
1
x
x
4
0
1
1
DataPath
RAM
We
Func3
Addr[31:0]
RData[31:0]
WData[31:0]
Data[31:0]
(InstrCode)
Instruction Memory (Rom)
addr[31:0]
RAddr1[4:0]
RD1[31:0]
RAddr2[4:0]
Register File
WAddr[4:0]
RD2[31:0]
WData[31:0]
ALU
aluSrcMux
ImmExtend
RFWDSrcMux
q[31:0]
Program Counter
d[31:0]
PCEn
JALR
Adder
PC_Imm_Adder
PCSrcMux
JalrSrcMux
32'b4
PC_4_Adder
OR
AND
JAL
Branch

# Chapter 04

# Control Signal & FSM


MCU
CPU
Control Unit
OPcode
func3
func7
RegFileEn
PCEn
aluSrcMuxSel
aluControl[3:0]
RFWDSrcMuxSel[2:0]
Branch, JAL, JALR
DataWe
DataPath
RAM
We
Func3
Addr[31:0]
RData[31:0]
WData[31:0]
Data[31:0]
Instruction Memory (Rom)
addr[31:0]
RAddr1[4:0]
RD1[31:0]
RAddr2[4:0]
Register File
WAddr[4:0]
RD2[31:0]
WData[31:0]
ALU
aluSrcMux
RFWDSrcMux
ImmExtend
q[31:0]
Program Counter
d[31:0]
PCEn
JALR
PC_Imm_Adder
PCSrcMux
JalrSrcMux
32'b4
PC_4_Adder
OR
JAL
AND
Branch
Fetch
Decode
Excute
MemAccess
WriteBack
PC to ROM
Control Unit Signal
ALU & Jump
RAM Adder.Data We
RAM Read Data RegFile Save

# FSM


Fetch
Decode
R-EXE
I-EXE
B-EXE
LU-EXE
S-EXE
L-EXE
AU-EXE
J-EXE
JR-EXE
L-Mem
S-Mem
L-WB

# Control Signals

| Control Singals | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **PC_EN** | Don’t En PC | PC En | | | |
| **regFileWe** | Don’t Write | Register File Write | | | |
| **AluSrcMuxSel** | RS2 | immExt | | | |
| **BusWe** | Don’t Write | RAM Write | | | |
| **RFWDSrcMuxSel** | aluResult | RAM RData | immExt | PC+imm | PC+4 |
| **Branch** | PC+4 | PC+imm | | | |
| **Jal** | Don’t Jump | PC+imm(Jump) | | | |
| **Jalr** | Don’t Jump | Rs1+imm(Jump) | | | |

# Control Signals

| State | PCEn | regFileWe | aluSrcMuxSel | busWe | REWDSrcMuxSel | branch | Jal | jalr |
|---|---|---|---|---|---|---|---|---|
| FETCH | 1 | 0 | 0 | 0 | 000 | 0 | 0 | 0 |
| DECODE | 0 | 0 | 0 | 0 | 000 | 0 | 0 | 0 |
| R-EXE | 0 | 1 | 0 | 0 | 000 | 0 | 0 | 0 |
| I-EXE | 0 | 1 | 1 | 0 | 000 | 0 | 0 | 0 |
| B-EXE | 0 | 0 | 0 | 0 | 000 | 1 | 0 | 0 |
| LU-EXE | 0 | 1 | 0 | 0 | 010 | 0 | 0 | 0 |
| AU-EXE | 0 | 1 | 0 | 0 | 011 | 0 | 0 | 0 |
| J-EXE | 0 | 1 | 0 | 0 | 100 | 0 | 1 | 0 |
| JL-EXE | 0 | 1 | 0 | 0 | 100 | 0 | 1 | 1 |
| S-EXE | 0 | 0 | 1 | 0 | 000 | 0 | 0 | 0 |
| S-MEM | 0 | 0 | 1 | 1 | 000 | 0 | 0 | 0 |
| L-EXE | 0 | 0 | 1 | 0 | 001 | 0 | 0 | 0 |
| L-MEM | 0 | 0 | 1 | 0 | 001 | 0 | 0 | 0 |
| L-WB | 0 | 1 | 1 | 0 | 001 | 0 | 0 | 0 |

# Chapter 05

# Simulation 검증

# 검증 시나리오

```
// R Type
// rom[x] =  func7    rs2   rs1  fc3  rd   opcode
rom[0] = 32'b0000000_00001_00010_000_00100_0110011;
rom[1] = 32'b0100000_00001_00010_000_00101_0110011;
rom[2] = 32'b0000000_00011_00010_001_00110_0110011;
rom[3] = 32'b0000000_00001_00010_101_00111_0110011;
rom[4] = 32'b0100000_00001_00010_101_01000_0110011;
rom[5] = 32'b0000000_00001_00010_010_01001_0110011;
rom[6] = 32'b0000000_00001_00010_011_01010_0110011;
rom[7] = 32'b0000000_00001_00010_100_01011_0110011;
rom[8] = 32'b0000000_00001_00010_110_01100_0110011;
rom[9] = 32'b0000000_00001_00010_111_01101_0110011;
```

```
// S Type
// rom[x] =  imm(7)   rs2   rs1   f3  imm(5) opcode
rom[10] = 32'b0000000_00110_00000_010_00100_0100011;
rom[11] = 32'b0000000_00110_00000_000_01000_0100011;
rom[12] = 32'b0000000_00110_00000_001_01100_0100011;
```

```
//L Type
// rom[x] =     imm(12)     rs1  f3   rd   opcode
rom[13] = 32'b000000000100_00000_010_11100_0000011;
rom[14] = 32'b000000001000_00000_000_11101_0000011;
rom[15] = 32'b000000001100_00000_001_11110_0000011;
rom[16] = 32'b000000001000_00000_100_11111_0000011;
rom[17] = 32'b000000001100_00000_101_00000_0000011;
```

```
//I Type
// rom[x] =   imm(12)       rs1  f3   rd   opcode
rom[18] = 32'b000000000111_00100_000_01110_0010011;
rom[19] = 32'b000000000010_00100_010_01111_0010011;
rom[20] = 32'b000000000011_00101_011_10000_0010011;
rom[21] = 32'b000000000010_00001_100_10001_0010011;
rom[22] = 32'b000000000101_00100_110_10010_0010011;
rom[23] = 32'b000000000101_00100_111_10011_0010011;
//rom[x] =    imm(7)  shamt  rs1  f3   rd   opcode
rom[24] = 32'b0000000_00010_00010_001_10101_0010011;
rom[25] = 32'b0000000_00010_00010_101_10110_0010011;
rom[26] = 32'b0100000_00010_00010_101_10111_0010011;
```

```
//B Type
// 조건 만족한 경우
// rom[x] =   imm(7)_ rs2 _ rs1 _f3 _imm5 _ opcode;
rom[27] = 32'b0000000_00001_00001_000_01000_1100011;
rom[29] = 32'b0000000_00010_00001_001_01000_1100011;
rom[31] = 32'b0000000_00010_00001_100_01000_1100011;
rom[33] = 32'b0000000_00011_00100_101_01000_1100011;
rom[35] = 32'b0000000_00010_00001_110_01000_1100011;
rom[37] = 32'b0000000_00011_00100_111_01000_1100011;
// 조건 만족하지 않는 경우
// rom[x] =   imm(7)_ rs2 _ rs1 _f3 _imm5 _ opcode;
rom[39] = 32'b0000000_00010_00001_000_01000_1100011;
rom[40] = 32'b0000000_00001_00001_001_01000_1100011;
rom[41] = 32'b0000000_00010_00010_100_01000_1100011;
rom[42] = 32'b0000000_00100_00011_101_01000_1100011;
rom[43] = 32'b0000000_00010_00010_110_01000_1100011;
rom[44] = 32'b0000000_00100_00011_111_01000_1100011;
```

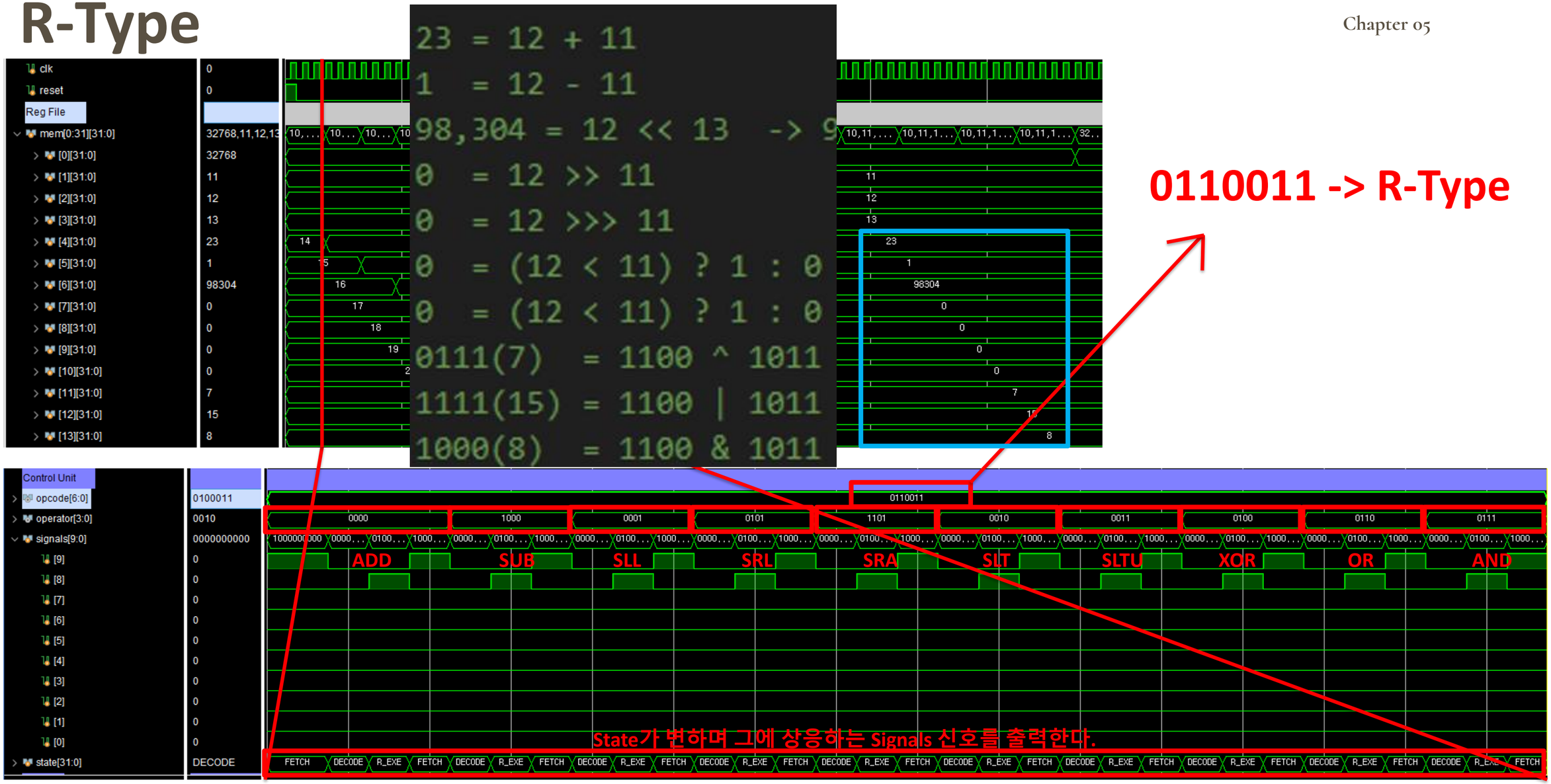
```
//LU Type
//rom[x]  = 32'b  imm(20)            rd    opcode
rom[45] = 32'b00000000000000000101_11000_0110111;

//AU Type
//rom[x]  = 32'b  imm(20)            rd    opcode
rom[46] = 32'b00000000000000000101_11001_0010111;

//J Type
//rom[x]  = 32'b  imm(20)            rd    opcode
rom[47] = 32'b00000000100000000000_11010_1101111;

//JL Type
//rom[x]  = 32'b  imm(12)    rs1  f3  rd   opcode
rom[49] = 32'b000000001000_00010_000_11011_1100111;
```

# R-Type



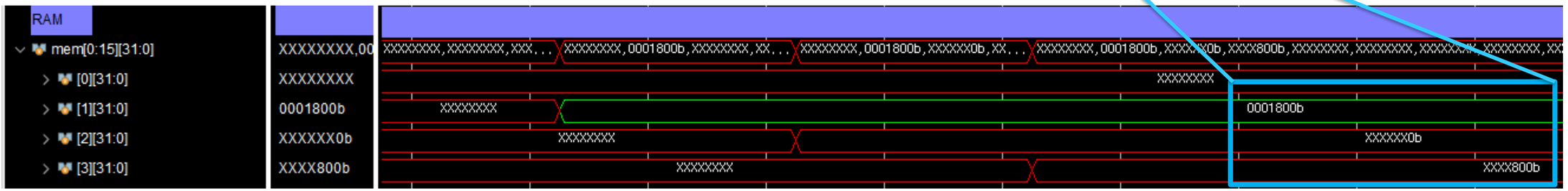
```
// R Type
// rom[x] =  func7    rs2   rs1  fc3  rd   opcode             rd   rs1 rs2
rom[0] = 32'b0000000_00001_00010_000_00100_0110011;  // add  x4,  x2, x1    23 = 12 + 11
rom[1] = 32'b0100000_00001_00010_000_00101_0110011;  // sub  x5,  x2, x1    1  = 12 - 11
rom[2] = 32'b0000000_00011_00010_001_00110_0110011;  // sll  x6,  x2, x3    98,304 = 12 << 13  -> 98,304(12를 bit로) << 13
rom[3] = 32'b0000000_00001_00010_101_00111_0110011;  // srl  x7,  x2, x1    0  = 12 >> 11
rom[4] = 32'b0100000_00001_00010_101_01000_0110011;  // sra  x8,  x2, x1    0  = 12 >>> 11
rom[5] = 32'b0000000_00001_00010_010_01001_0110011;  // slt  x9,  x2, x1    0  = (12 < 11) ? 1 : 0
rom[6] = 32'b0000000_00001_00010_011_01010_0110011;  // sltu x10, x2, x1    0  = (12 < 11) ? 1 : 0
rom[7] = 32'b0000000_00001_00010_100_01011_0110011;  // xor  x11, x2, x1    0111(7)  = 1100 ^ 1011
rom[8] = 32'b0000000_00001_00010_110_01100_0110011;  // or   x12, x2, x1    1111(15) = 1100 | 1011
rom[9] = 32'b0000000_00001_00010_111_01101_0110011;  // and  x13, x2, x1    1000(8)  = 1100 & 1011
```

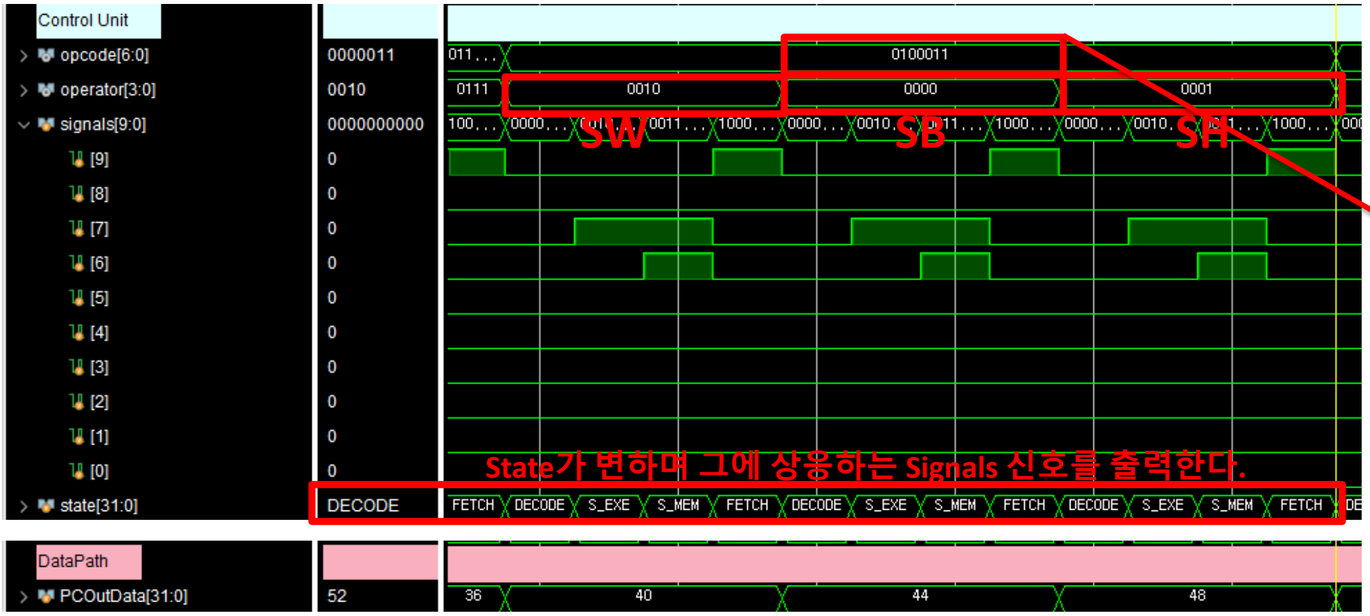
# R-Type


23 = 12 + 11
1 = 12 - 11
98,304 = 12 << 13 -> 9
0 = 12 >> 11
0 = 12 >>> 11
0 = (12 < 11) ? 1 : 0
0 = (12 < 11) ? 1 : 0
0111(7) = 1100 ^ 1011
1111(15) = 1100 | 1011
1000(8) = 1100 & 1011
clk
reset
Reg File
mem[0:31][31:0]
[0][31:0]
[1][31:0]
[2][31:0]
[3][31:0]
[4][31:0]
[5][31:0]
[6][31:0]
[7][31:0]
[8][31:0]
[9][31:0]
[10][31:0]
[11][31:0]
[12][31:0]
[13][31:0]
32768,11,12,13
32768
11
12
13
23
1
98304
0
7
15
8
0110011 -> R-Type
Control Unit
opcode[6:0]
operator[3:0]
signals[9:0]
[9]
[8]
[7]
[6]
[5]
[4]
[3]
[2]
[1]
[0]
state[31:0]
0100011
0010
0000000000
DECODE
0110011
0000
1000
0001
0101
1101
0010
0011
0100
0110
0111
ADD
SUB
SLL
SRL
SRA
SLT
SLTU
XOR
OR
AND
State가 변하며 그에 상응하는 Signals 신호를 출력한다.
FETCH
DECODE
R_EXE

# S-Type

```
// S Type
// rom[x] =  imm(7)   rs2   rs1   f3  imm(5) opcode          rs1  imm rs2
rom[10] = 32'b0000000_00110_00000_010_00100_0100011; // SW    x0   4   x6  =>  RAM[0+1] = 0x0001_800b = 98,315
rom[11] = 32'b0000000_00110_00000_000_01000_0100011; // SB    x0   8   x6  =>  RAM[0+2] = 0x0000_000b = 11
rom[12] = 32'b0000000_00110_00000_001_01100_0100011; // SH    x0   12  x6  =>  RAM[0+3] = 0x0000_800b = 32,779
```



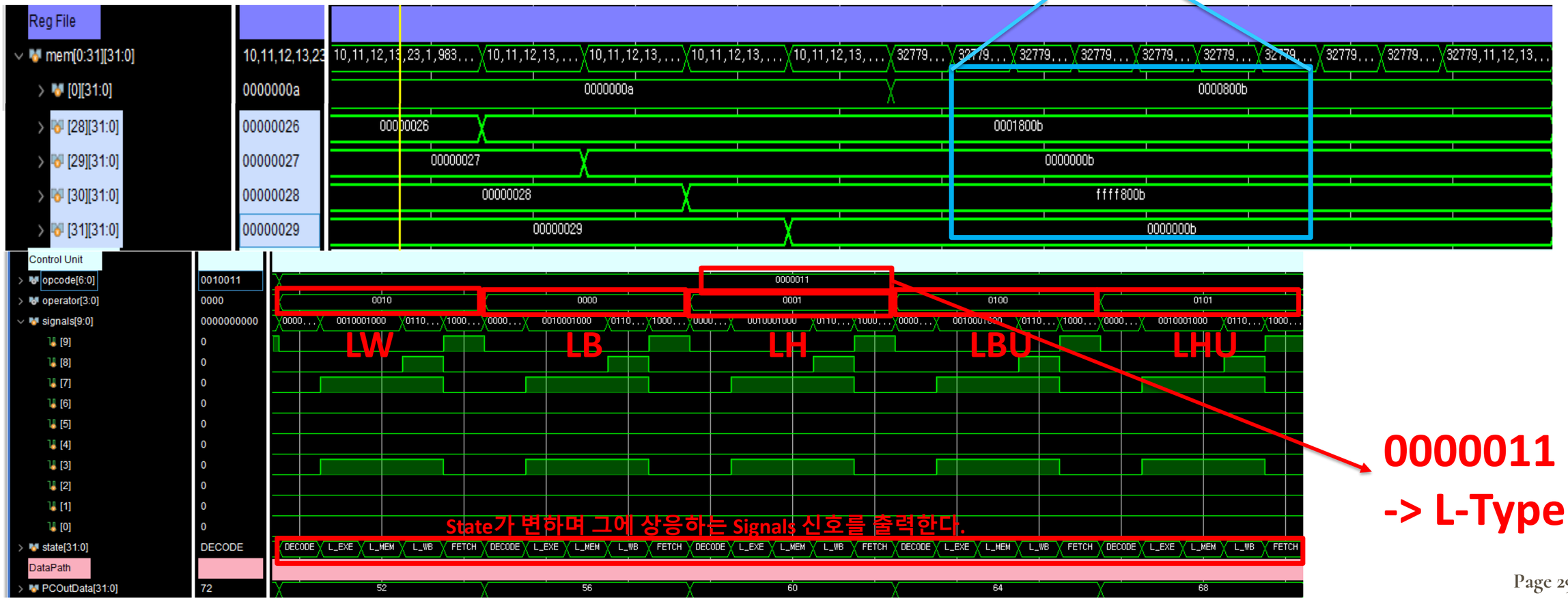




**0100011 -> S-Type**

# L-Type

```
//L Type
// rom[x] =     imm(12)     rs1  f3   rd   opcode           rd   rs1 imm
rom[13] = 32'b000000000100_00000_010_11100_0000011;  // LW   x28  x0  4    => regFile[0+28] = 98,315    =>  0x0001_800b
rom[14] = 32'b000000001000_00000_000_11101_0000011;  // LB   x29  x0  8    => regFile[0+29] = 11        =>  0x0000_000b
rom[15] = 32'b000000001100_00000_001_11110_0000011;  // LH   x30  x0  12   => regFile[0+30] = -32,779   =>  0xffff_800b
rom[16] = 32'b000000001000_00000_100_11111_0000011;  // LBU  x31  x0  8    => regFile[0+29] = 11        =>  0x0000_000b
rom[17] = 32'b000000001100_00000_101_00000_0000011;  // LHU  x0   x0  12   => regFile[0+30] = 32,779    =>  0x0000_800b
```




**0000011**
**-> L-Type**

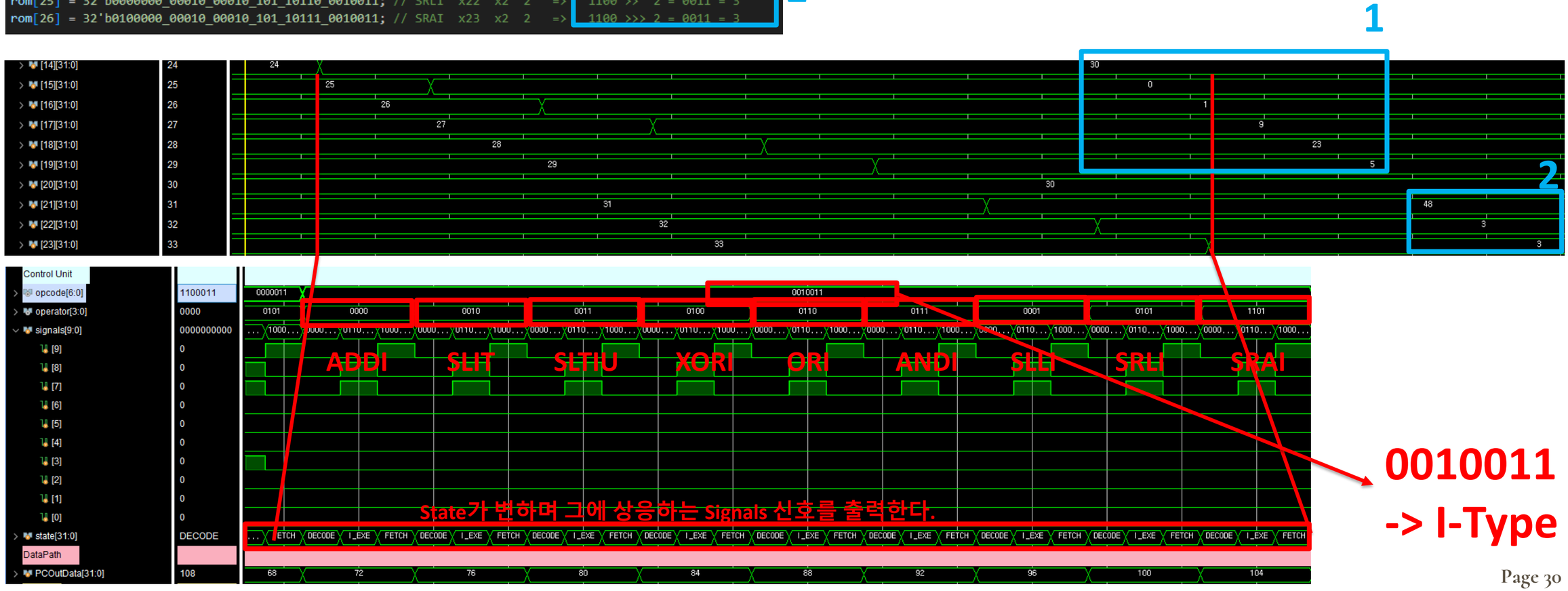
# I-Type



```
//I Type
// rom[x] =    imm(12)       rs1  f3   rd   opcode            rd   rs1 imm
rom[18] = 32'b000000000111_00100_000_01110_0010011; // ADDI   x14  x4  7   =>  23 + 7 = 30
rom[19] = 32'b000000000010_00100_010_01111_0010011; // SLIT   x15  x4  2   =>  (23 < 2) ? 1: 0 = 0
rom[20] = 32'b000000000011_00101_011_10000_0010011; // SLTIU  x16  x5  3   =>  (1 < 3) ? 1 :0 = 1
rom[21] = 32'b000000000010_00001_100_10001_0010011; // XORI   x17  x1  2   =>  1011 ^ 0010 = 1001 = 9
rom[22] = 32'b000000000101_00100_110_10010_0010011; // ORI    x18  x4  5   =>  10111 | 00101 = 10111 = 23
rom[23] = 32'b000000000101_00100_111_10011_0010011; // ANDI   x19  x4  5   =>  10111 & 00101 = 00101 = 5
//rom[x] =     imm(7)  shamt  rs1  f3   rd   opcode            rd   rs1 imm
rom[24] = 32'b0000000_00010_00010_001_10101_0010011; // SLLI   x21  x2  2   =>  1100 <<  2 = 110000 = 48
rom[25] = 32'b0000000_00010_00010_101_10110_0010011; // SRLI   x22  x2  2   =>  1100 >>  2 = 0011 = 3
rom[26] = 32'b0100000_00010_00010_101_10111_0010011; // SRAI   x23  x2  2   =>  1100 >>> 2 = 0011 = 3
```

1

2



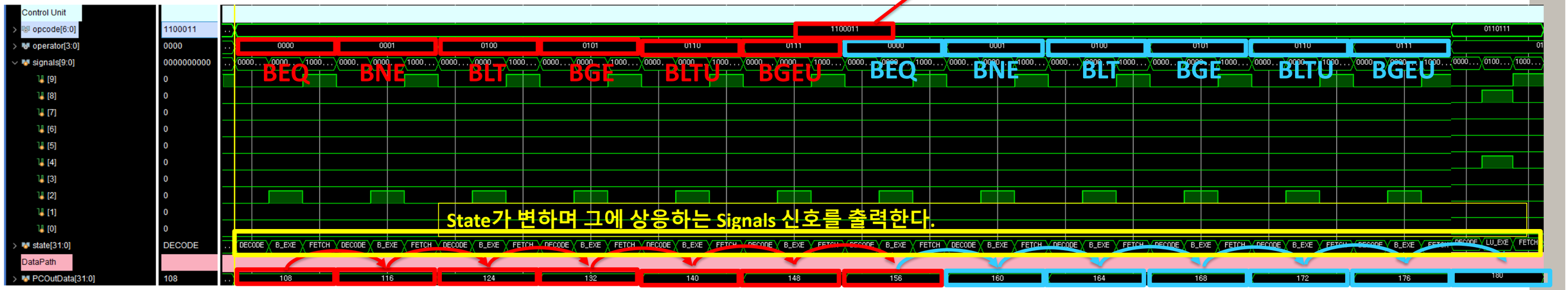

**0010011**
**-> I-Type**

# B-Type



```
//B Type
// 조건 만족한 경우
// rom[x] =   imm(7)_ rs2 _ rs1 _f3 _imm5 _ opcode;          rs1  rs2 imm     PC  -> PC + 8  |   2씩 분기
rom[27] = 32'b0000000_00001_00001_000_01000_1100011; // BEQ   x1   x1  8   =>   108 -> 116    |   rom[27] -> rom[29]
rom[29] = 32'b0000000_00010_00001_001_01000_1100011; // BNE   x1   x2  8   =>   116 -> 124    |   rom[29] -> rom[31]
rom[31] = 32'b0000000_00010_00001_100_01000_1100011; // BLT   x1   x2  8   =>   124 -> 132    |   rom[31] -> rom[33]
rom[33] = 32'b0000000_00011_00100_101_01000_1100011; // BGE   x4   x3  8   =>   132 -> 140    |   rom[33] -> rom[35]
rom[35] = 32'b0000000_00010_00001_110_01000_1100011; // BLTU  x1   x2  8   =>   140 -> 148    |   rom[35] -> rom[37]
rom[37] = 32'b0000000_00011_00100_111_01000_1100011; // BGEU  x4   x3  8   =>   148 -> 156    |   rom[37] -> rom[39]
// 조건 만족하지 않는 경우
// rom[x] =   imm(7)_ rs2 _ rs1 _f3 _imm5 _ opcode;          rs1  rs2 imm     PC  -> PC + 4  |   정상 진행
rom[39] = 32'b0000000_00010_00001_000_01000_1100011; // BEQ   x2   x1  8   =>   156 -> 160    |   rom[39] -> rom[40]
rom[40] = 32'b0000000_00001_00001_001_01000_1100011; // BNE   x1   x1  8   =>   160 -> 164    |   rom[40] -> rom[41]
rom[41] = 32'b0000000_00010_00010_100_01000_1100011; // BLT   x2   x2  8   =>   164 -> 168    |   rom[41] -> rom[42]
rom[42] = 32'b0000000_00100_00011_101_01000_1100011; // BGE   x3   x4  8   =>   168 -> 172    |   rom[42] -> rom[43]
rom[43] = 32'b0000000_00010_00010_110_01000_1100011; // BLTU  x2   x2  8   =>   172 -> 176    |   rom[43] -> rom[44]
rom[44] = 32'b0000000_00100_00011_111_01000_1100011; // BGEU  x3   x4  8   =>   176 -> 180    |   rom[44] -> rom[45]
```

**0010011 -> B-Type**

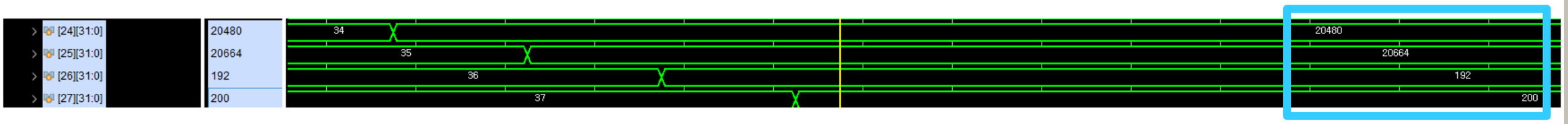


# LU, AU, J, JL-Type



```
//LU Type
//rom[x]  = 32'b  imm(20)            rd    opcode            rd   imm
rom[45] = 32'b00000000000000000101_11000_0110111;    // LUI   x24  5       =>  5 << 12 = 20480

//AU Type
//rom[x]  = 32'b  imm(20)            rd    opcode            rd   imm
rom[46] = 32'b00000000000000000101_11001_0010111;    // AUIPC x25  5       =>  PC(184) + (5 << 12) = 20664
```

```
//J Type
//rom[x]  = 32'b  imm(20)            rd    opcode            rd   imm
rom[47] = 32'b00000000100000000000_11010_1101111;    // JAL   x26  8       =>  x26 = 188 + 4 = 192   PC + 8  = 196

//JL Type
//rom[x]  = 32'b  imm(12)    rs1  f3  rd    opcode            rd   rs1  imm
rom[49] = 32'b000000001000_00010_000_11011_1100111; // JALR  x27  x2    8  =>  x27 = 196 + 4 = 200   PC = x2(12) + 8 = 20
```





**0110111 -> LU-Type**
**0010111 -> AU-Type**
**1101111 -> J-Type**
**1100111 -> JL-Type**

**J=PC+imm(8)**  **JL=RS1+PC**

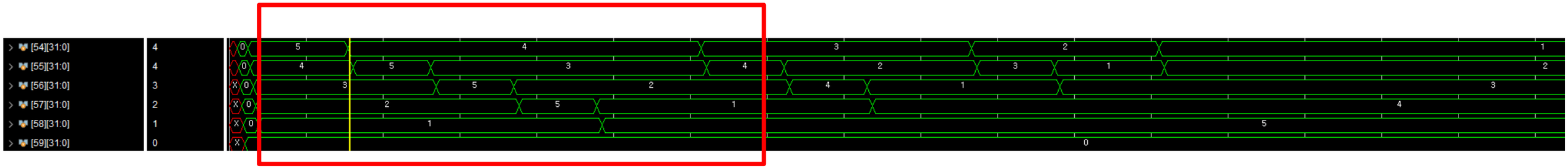
# Buble Sort

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

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

    sort(arData, 5);

    return 0;
}

void sort(int *pData, int size){
    for (int i = 0; i < size; i++){
        for (int j = 0; j < size - i - 1; j++){
            if (pData[j] > pData[j + 1])
                swap(&pData[j], &pData[j+1]);
        }
    }
}

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

C to Assembly →

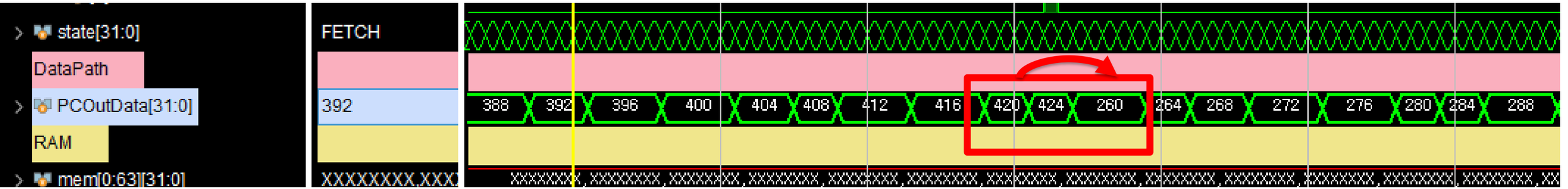
```asm
main:
        addi    sp,sp,-48
        sw      ra,44(sp)
        sw      s0,40(sp)
        addi    s0,sp,48
        sw      zero,-40(s0)
        sw      zero,-36(s0)
        sw      zero,-32(s0)
        sw      zero,-28(s0)
        sw      zero,-24(s0)
        sw      zero,-20(s0)
        li      a5,5
        sw      a5,-40(s0)
        li      a5,4
        sw      a5,-36(s0)
        li      a5,3
        sw      a5,-32(s0)
        li      a5,2
        sw      a5,-28(s0)
        li      a5,1
        sw      a5,-24(s0)
        addi    a5,s0,-40
        li      a1,5
        mv      a0,a5
        call    sort
        li      a5,0
        mv      a0,a5
        lw      ra,44(sp)
        lw      s0,40(sp)
        addi    sp,sp,48
        jr      ra
sort:
        addi    sp,sp,-48
        sw      ra,44(sp)
        sw      s0,40(sp)
        addi    s0,sp,48
        sw      a0,-36(s0)
        sw      a1,-40(s0)
        sw      zero,-20(s0)
```

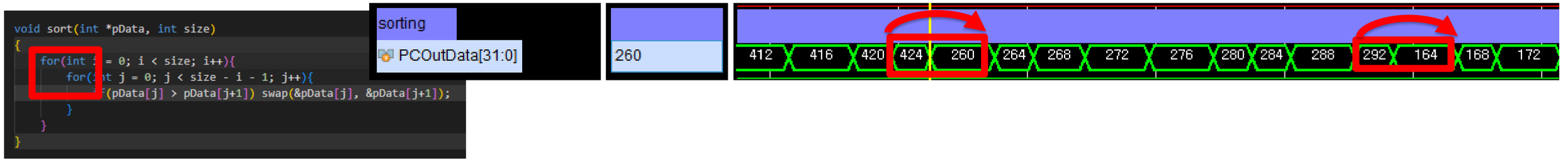
| | | | | | |
|---|---|---|---|---|---|
| 5 | 4 | 4 | 4 | 4 | 3 |
| 4 | 5 | 3 | 3 | 3 | 4 |
| 3 | 3 | 4 | 2 | 2 | 2 … |
| 2 | 2 | 2 | 5 | 1 | 1 |
| 1 | 1 | 1 | 1 | 5 | 5 |
| 0 | 0 | 0 | 0 | 0 | 0 |

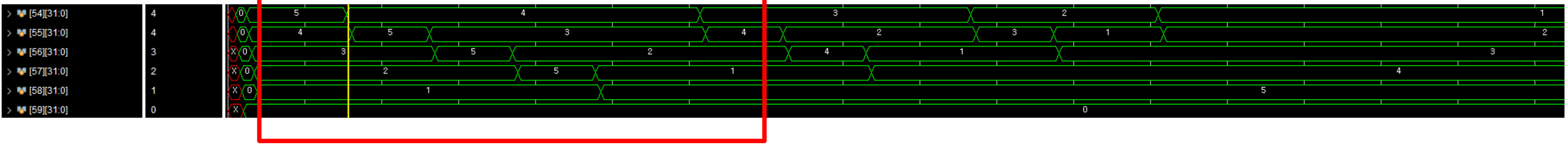


# Buble Sort

**Jump하며 계속 함수를 호출,
정렬되는 것을 반복함을
확인할 수 있음**



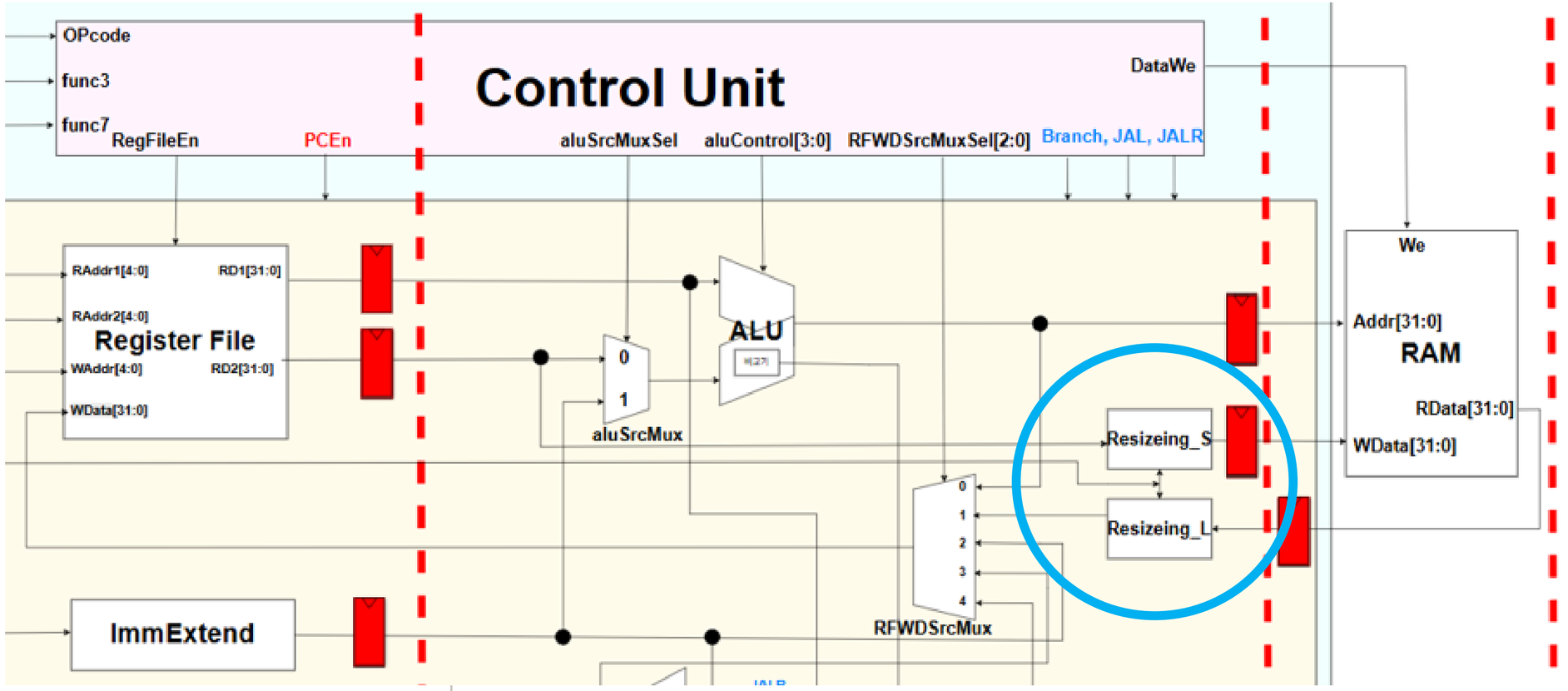



Chapter

# 06

# Trouble Shooting 및 고찰

# Trouble Shooting – S,L Type (half, Byte)




**Data Path 내부에서 구현**

# Trouble Shooting – S,L Type (half, Byte)

S - Type

```
module data_resize_s (
    input  logic [31:0] i_RegData,  // 레지스터에서 들어온 데이터
    input  logic [31:0] instrCode,  // instrCode[14:12] (S-type funct3)
    output logic [31:0] ram_w_data  // 메모리에 저장할 데이터
);
    wire [2:0] func3 = instrCode[14:12];

    always_comb begin
        case (func3)
            3'b000: ram_w_data = {{24{1'b0}}, i_RegData[7:0]};  // SB:
            3'b001: ram_w_data = {{16{1'b0}}, i_RegData[15:0]};  // SH:
            3'b010: ram_w_data = i_RegData;  // SW: store word
            default: ram_w_data = i_RegData;
        endcase
    end
endmodule
```

## S-Type

사이즈에 맞는 크기만 저장하는 것이 아닌 0으로 상위 비트를 사용

따라서  RAM 에 0xXXXX1234 형태가 아닌,

0x00001234 형태로 값이 손상됨

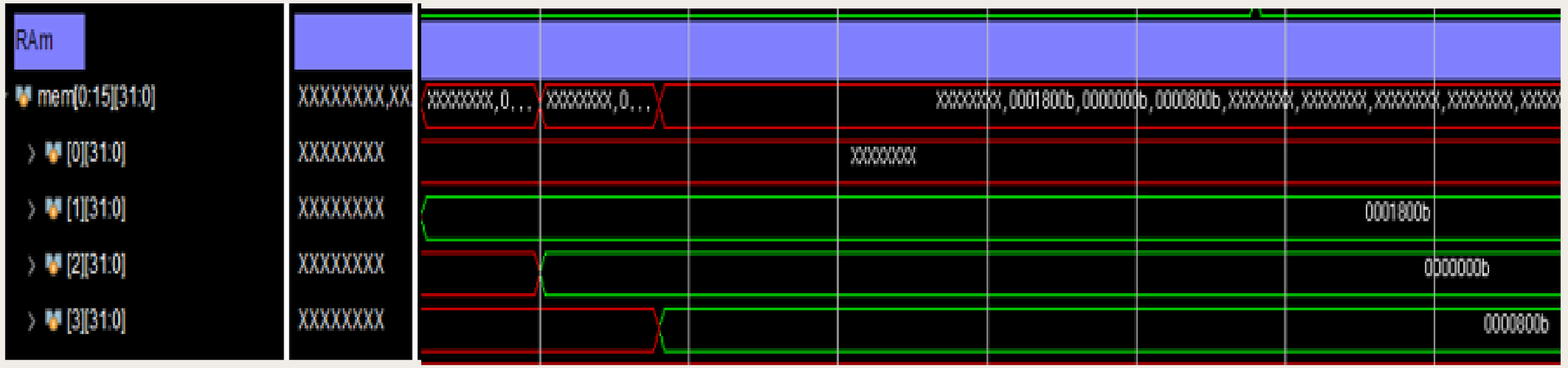
L - Type

```
module data_resize_l (
    input  logic [31:0] ram_r_data,  // 메모리에서 읽은 데이터
    input  logic [31:0] instrCode,       // instrCode[14:12] (I-type funct3)
    output logic [31:0] o_RegData    // 레지스터에 쓸 데이터
);
    wire [2:0] func3 = instrCode[14:12];

    always_comb begin
        case (func3)
            3'b000: o_RegData = $signed({{24{ram_r_data[7]}}, ram_r_data[7:0]});  // LB
            3'b001: o_RegData = $signed({{16{ram_r_data[15]}}, ram_r_data[15:0]});  // LH
            3'b010: o_RegData = ram_r_data;  // LW
            3'b100: o_RegData = {{24{1'b0}}, ram_r_data[7:0]};  // LBU
            3'b101: o_RegData = {{16{1'b0}}, ram_r_data[15:0]};  // LHU
            default: o_RegData = ram_r_data;
        endcase
    end
endmodule
```

## L-Type

Load의 경우는 정상 작동함

# Trouble Shooting – S,L Type (half, Byte)




**S-Type**

사이즈에 맞는 크기만 저장하는 것이 아닌 0으로 상위 비트를 사용
따라서  RAM 에 0xXXXX1234 형태가 아닌,
0x00001234 형태로 값이 손상됨

# Trouble Shooting – S,L Type (half, Byte) 해결 방법

S - Type

```
// ----------------------
// Store (S-type)
// ----------------------
always_comb begin
    case (func3)
        3'b000: begin  // SB
            case (addr[1:0])
                2'b00: w_Data_com = {mem[addr[31:2]][31:8], wData[7:0]};
                2'b01: w_Data_com = {mem[addr[31:2]][31:16], wData[7:0], mem[addr[31:2]][7:0]};
                2'b10: w_Data_com = {mem[addr[31:2]][31:24], wData[7:0], mem[addr[31:2]][15:0]};
                2'b11: w_Data_com = {wData[7:0], mem[addr[31:2]][23:0]};
                default: w_Data_com = mem[addr[31:2]];
            endcase
        end
        3'b001: begin  // SH
            case (addr[1])
                1'b0: w_Data_com = {mem[addr[31:2]][31:16], wData[15:0]};
                1'b1: w_Data_com = {wData[15:0], mem[addr[31:2]][15:0]};
                default: w_Data_com = mem[addr[31:2]];
            endcase
        end
        3'b010:  w_Data_com = wData;  // SW
        default: w_Data_com = mem[addr[31:2]];
    endcase
end
```

## S-Type

Func3를 받아 분류 후, addr의 하위 2비트를 기준으로

RAM 내부에서 addr[31:2]에 wData를 넣어줌

L - Type

```
// ----------------------
// Load L(-Type)
// ----------------------
always_comb begin
    case (func3)
        3'b000: begin  // LB
            case (addr[1:0])
                2'b00: rData = {{24{mem[addr[31:2]][7]}}, mem[addr[31:2]][7:0]};   // byte0
                2'b01: rData = {{24{mem[addr[31:2]][15]}}, mem[addr[31:2]][15:8]};  // byte1
                2'b10: rData = {{24{mem[addr[31:2]][23]}}, mem[addr[31:2]][23:16]};  // byte2
                2'b11: rData = {{24{mem[addr[31:2]][31]}}, mem[addr[31:2]][31:24]};  // byte3
                default: rData = 32'bx;
            endcase
        end
        3'b001: begin  // LH
            case (addr[1])
                1'b0: rData = {{16{mem[addr[31:2]][15]}}, mem[addr[31:2]][15:0]};
                1'b1: rData = {{16{mem[addr[31:2]][31]}}, mem[addr[31:2]][31:16]};
                default: rData = 32'bx;
            endcase
        end
```

## L-Type

같은 방식으로 해결함

# Trouble Shooting – Sort(Register File )

```
initial begin  // for simulation test
    for (int i = 0; i < 32; i++) begin
        mem[i] = 10 + i;
    end
end
```

Strategy

# 아쉬운 점 및 구현 아이디어

**RAM** RAM 내부에 기능을 구현했지만, Data Path에서 구현했더라면?

**Func3** Top 모듈에서 신호를 연결해 wire Delay증가

**FSM** 2중 case문을 사용해 Combination Delay 증가

Control Unit에서 FSM 상태로 제어 했더라면?

```
initial begin
    for (int i = 0; i < 16; i++) begin
        mem[i] = 32'bx;  // 명시적으로 unknown 상태로 설정
    end
end
always_ff @(posedge clk) begin
    if (we) begin
        case (size)
            2'b00: begin  // byte write
                case (addr[1:0])
                    2'b00: mem[addr[31:2]][7:0] <= wData[7:0];
                    2'b01: mem[addr[31:2]][15:8] <= wData[7:0];
                    2'b10: mem[addr[31:2]][23:16] <= wData[7:0];
                    2'b11: mem[addr[31:2]][31:24] <= wData[7:0];
                endcase
            end
            2'b01: begin  // halfword write
                case (addr[1])
                    1'b0: mem[addr[31:2]][15:0] <= wData[15:0];
                    1'b1: mem[addr[31:2]][31:16] <= wData[15:0];
                endcase
            end
            2'b10: mem[addr[31:2]] <= wData;  // word write (기존
            default: mem[addr[31:2]] <= wData;  // 안전장치
        endcase
    end
end

assign rData = mem[addr[31:2]];  // 항상 전체 word 반환
```

# FSM 에서 signal 제어

```
end
S_EXE: begin
    signals = 10'b0_0_1_0_000_0_0_0;
    // Store 명령어의 메모리 크기 설정
    case (func3)
        3'b000: memSize = 2'b00; // sb
        3'b001: memSize = 2'b01; // sh
        3'b010: memSize = 2'b10; // sw (기존)
        default: memSize = 2'b10; // 안전장치
    endcase
end
S_MEM: begin
    signals = 10'b0_0_1_1_000_0_0_0;
    // Store 명령어의 메모리 크기 설정 (S_EXE와 동일하게 유지)
    case (func3)
        3'b000: memSize = 2'b00; // sb
        3'b001: memSize = 2'b01; // sh
        3'b010: memSize = 2'b10; // sw (기존)
        default: memSize = 2'b10; // 안전장치
    endcase
end
```

```
L_EXE: begin
    signals = 10'b0_0_1_0_001_0_0_0;
    // Load 명령어의 메모리 크기 및 부호 확장 설정
    case (func3)
        3'b000: begin memSize = 2'b00; memUnsigned = 1'b0; end // lb
        3'b001: begin memSize = 2'b01; memUnsigned = 1'b0; end // lh
        3'b010: begin memSize = 2'b10; memUnsigned = 1'b0; end // lw (기존)
        3'b100: begin memSize = 2'b00; memUnsigned = 1'b1; end // lbu
        3'b101: begin memSize = 2'b01; memUnsigned = 1'b1; end // lhu
        default: begin memSize = 2'b10; memUnsigned = 1'b0; end // 안전장치
    endcase
end
L_MEM: begin
    signals = 10'b0_0_1_0_001_0_0_0;
    // Load 명령어의 메모리 크기 및 부호 확장 설정 (L_EXE와 동일하게 유지)
    case (func3)
        3'b000: begin memSize = 2'b00; memUnsigned = 1'b0; end // lb
        3'b001: begin memSize = 2'b01; memUnsigned = 1'b0; end // lh
        3'b010: begin memSize = 2'b10; memUnsigned = 1'b0; end // lw (기존)
        3'b100: begin memSize = 2'b00; memUnsigned = 1'b1; end // lbu
        3'b101: begin memSize = 2'b01; memUnsigned = 1'b1; end // lhu
        default: begin memSize = 2'b10; memUnsigned = 1'b0; end // 안전장치
    endcase
end
L_WB: begin
    signals = 10'b0_1_1_0_001_0_0_0;
    // Load 명령어의 메모리 크기 및 부호 확장 설정 (L_EXE와 동일하게 유지)
    case (func3)
        3'b000: begin memSize = 2'b00; memUnsigned = 1'b0; end // lb
        3'b001: begin memSize = 2'b01; memUnsigned = 1'b0; end // lh
        3'b010: begin memSize = 2'b10; memUnsigned = 1'b0; end // lw (기존)
        3'b100: begin memSize = 2'b00; memUnsigned = 1'b1; end // lbu
        3'b101: begin memSize = 2'b01; memUnsigned = 1'b1; end // lhu
        default: begin memSize = 2'b10; memUnsigned = 1'b0; end // 안전장치
    endcase
end
```

Overview

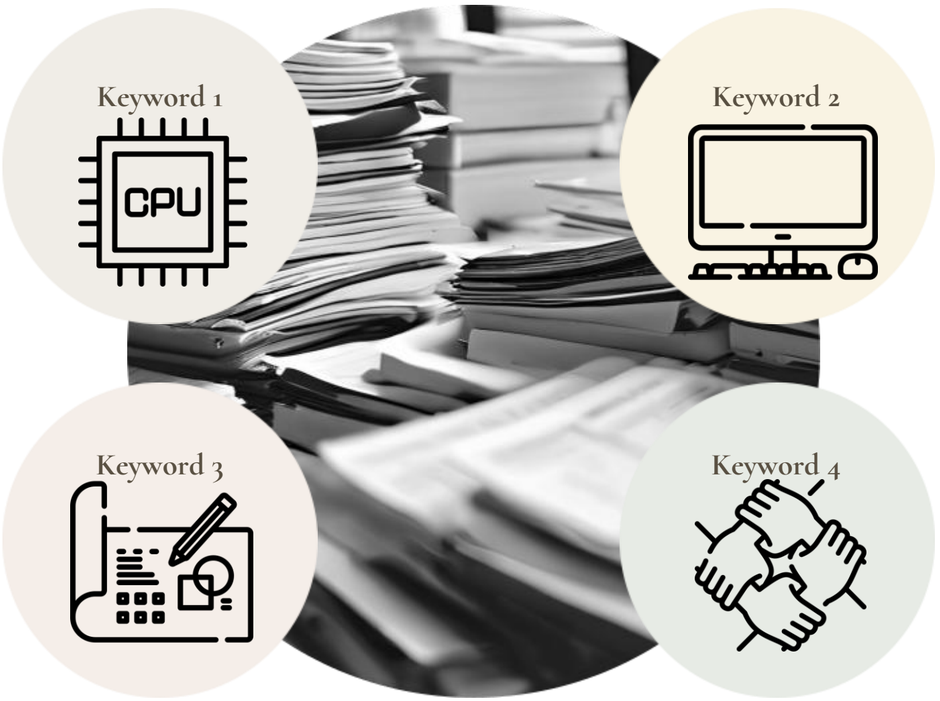
# 설계 과정에서 배운 점과 느낀 점

### 명령어 흐름 파악

명령어 Type별 동작 원리를
직접 설계함으로써 직관적으로 이해함

### 효율적 설계 고민

Data Sizing을 위한 모듈을 구상하며
다양한 방식을 고민함




### 구조적 사고 향상

컴퓨터 구조의 이론을 실제 동작에 적용
모듈화를 이용해 CYPU를 설계

### 다양한 구현 방법

같은 결과를 내기 위한 설계 방식의 다양함을
팀원들을 통해 배울 수 있었음

# Thank You!